

Think DSP

Digital Signal Processing in Python

Versione 1.1.4

(Trad. Ita: 29 dicembre 2023)

Think DSP

Digital Signal Processing in Python

Versione 1.1.4

(Trad. Ita: 29 dicembre 2023)

Allen B. Downey

Green Tea Press

Needham, Massachusetts

Traduzione italiana: Baldassarre Cesarano

Copyright © 2014 Allen B. Downey.

Green Tea Press
9 Washburn Ave
Needham, Massachusetts

Traduzione italiana: Baldassarre Cesarano

Needham MA 02492

È concesso il permesso di copiare, distribuire e/o modificare questo documento in base ai termini della Licenza Internazionale Creative Commons Attribuzione-Non Commerciale-Condividi allo stesso modo 4.0, disponibile al link <http://creativecommons.org/licenses/by-nc-sa/4.0/>.

Il sorgente L^AT_EX per questo libro è disponibile su <http://think-dsp.com>.

Prefazione

L’elaborazione dei segnali, il Signal Processing, è uno dei miei argomenti favoriti. È utile in molte aree della scienza e dell’ingegneria e, se si comprendono le idee fondamentali, fornisce informazioni su molte cose che vediamo nel mondo, e soprattutto sulle cose che ascoltiamo.

Ma a meno che non si abbiano alle spalle studi di ingegneria elettronica o meccanica, probabilmente non si c’è stata la possibilità di apprendere l’elaborazione dei segnali. Il problema è che la maggior parte dei libri (e i docenti che li usano) presentano il materiale dal basso verso l’alto, a partire da astrazioni matematiche come i fasori. Tendono, quindi, ad essere teorici, con poche applicazioni e poca rilevanza apparente.

La premessa di questo libro è che se si sa programmare, si può usare tale abilità per imparare altre cose, divertendosi nel farlo.

Con un approccio basato sulla programmazione, si possono presentare subito le idee più importanti. Entro la fine del primo capitolo, è possibile analizzare le registrazioni audio e altri segnali e generare nuovi suoni. Ogni capitolo introduce una nuova tecnica e un’applicazione a segnali reali. Ad ogni passaggio si impara prima come usare una tecnica e poi come funziona.

Tale approccio è più pratico e, spero che sarete d’accordo, più divertente.

0.1 Per chi è questo libro?

Gli esempi e il codice a supporto per questo libro sono in Python. Si dovrebbero conoscere le basi di Python e avere familiarità con le funzionalità dell’object-oriented, almeno saper usare gli oggetti se non crearne dei propri.

Se non si ha già una certa familiarità con Python, si potrebbe iniziare con l’altro mio altro libro, *Think Python*, che è un’introduzione a Python per chi non ha mai programmato, oppure con *Learning Python* di Mark Lutz,

che potrebbe risultare migliore per chi ha già una certa esperienza con la programmazione.

Io uso tantissimo NumPy e SciPy. Se si conoscono già, è fantastico, ma spiegherò anche le funzioni e le strutture dati utilizzate.

Presumo che il lettore conosca la matematica di base, compresi i numeri complessi. Non c'è bisogno di molta matematica; se si comprendono i concetti di integrazione e differenziazione, andrà bene. Uso un po' di algebra lineare, ma la spiegherò man mano che si procede.

0.2 Utilizzo del codice

Il codice e i suoni di esempio utilizzati in questo libro sono disponibili su <https://github.com/AllenDowney/ThinkDSP>. Git è un sistema di controllo delle versioni che permette di tener traccia dei file che compongono un progetto. Una raccolta di file sotto il controllo di Git è detta “repository”. GitHub è un servizio di hosting che fornisce spazio di archiviazione per i repository Git e una comoda interfaccia web.

La home page di GitHub per il mio repository offre diversi modi per lavorare con il codice:

- Si può creare una copia del repository su GitHub premendo il pulsante **Fork**. Se non si ha già un account GitHub, se ne dovrà creare uno. Dopo il fork, si avrà il proprio repository su GitHub, utilizzabile per tenere traccia del codice che si scrive mentre si lavora con questo libro. Poi si può clonare il repository, il che significa che si copiano i file sul proprio computer.
- Oppure si potrebbe clonare il mio repository. Non è necessario un account GitHub per farlo, ma non si sarà in grado di aggiornare le modifiche su GitHub.
- Se non si desidera utilizzare Git, si può scaricare il tutto in un file Zip utilizzando il pulsante nell'angolo in basso a destra della pagina di GitHub.

Tutto il codice è scritto per funzionare sia in Python 2 che in Python 3 senza modifiche.

Questo libro è stato sviluppato utilizzando Anaconda della Continuum Analytics, una distribuzione Python gratuita che include tutti i pacchetti necessari per eseguire il codice (e molto altro). Ho trovato Anaconda facile

da installare. Per default, esegue un'installazione a livello di utente, non a livello di sistema, quindi non sono necessari i privilegi di amministratore. Supporta sia Python 2 che Python 3. Anaconda si può scaricare da <https://www.anaconda.com/distribution/>.

Se non si vuol usare Anaconda, saranno necessari i seguenti pacchetti:

- NumPy per il calcolo numerico di base, <http://www.numpy.org/>;
- SciPy per il calcolo scientifico, <http://www.scipy.org/>;
- matplotlib per la visualizzazione, <http://matplotlib.org/>.

Sebbene questi siano pacchetti comunemente usati, non sono inclusi in tutte le installazioni di Python e possono essere difficili da installare in certi ambienti. Se ci sono problemi nell'installazione, consiglio di utilizzare Anaconda o una delle altre distribuzioni Python che includono questi pacchetti.

La maggior parte degli esercizi utilizza script Python, ma alcuni usano anche i notebook Jupyter. Se non si conosce Jupyter, ci si può informare su <http://jupyter.org>.

Esistono tre modi per lavorare con i notebook Jupyter:

Eseguire Jupyter sul proprio computer Se si è installato Anaconda, probabilmente si ha Jupyter per default. Per verificarlo, avviare il server dalla riga di comando, in questo modo:

```
$ jupyter notebook
```

Se non lo è già, lo si può installare in Anaconda in questo modo:

```
$ conda install jupyter
```

Quando si avvia il server, questo dovrebbe avviare il browser Web predefinito o, se il browser è già aperto, creare una nuova scheda.

Eseguire Jupyter su Binder Binder è un servizio che esegue Jupyter in una macchina virtuale. Se si segue questo link, <http://mybinder.org/repo/AllenDowney/ThinkDSP>, si dovrebbe arrivare a una home page di Jupyter con i notebook per questo libro con i dati di supporto e gli script.

Gli script si possono eseguire e modificare per eseguire il proprio codice, ma la macchina virtuale in cui lo si fa è temporanea. Qualsiasi modifica apportata scomparirà, insieme alla macchina virtuale, se la si lasca inattiva per più di un'ora circa.

Visualizzare i notebook su nbviewer Quando ci riferiremo ai notebook, più avanti nel libro, forniremo dei link a nbviewer, che propone una vista statica del codice e dei risultati. È possibile utilizzare tali link per leggere i notebook e ascoltare gli esempi, ma non sarà possibile modificare o eseguire il codice né utilizzare i widget interattivi.

Buona fortuna e buon divertimento!

Elenco dei contributori

Per qualsiasi suggerimento o correzione, inviare un'email a downey@allendowney.com. [Per la traduzione italiana: baldassarre.cesarano AT libero.it]. Se apporto una modifica in base al tuo feedback, ti aggiungerò nell'elenco dei contributori (a meno che tu non chieda di essere omesso).

Se si include almeno una parte della frase in cui compare l'errore, è facile per me cercare. Anche i numeri di pagina e di sezione vanno bene, ma non ci si lavora altrettanto facilmente. Grazie!

- Le mie idee su questo libro, prima di iniziare a scrivere, hanno tratto beneficio dalle conversazioni con Boulos Harb di Google e Aurelio Ramos, ex Harmonix Music Systems.
- Durante il semestre dell'autunno 2013, Nathan Lintz e Ian Daniher hanno lavorato con me a un progetto di studio indipendente e aiutandomi con la prima bozza di questo libro.
- Sul forum DSP di Reddit, l'utente anonimo RamjetSoundwave mi ha aiutato a risolvere un problema con la mia implementazione del Rumore Browniano. E andodli ha trovato un errore di battitura.
- Nella primavera del 2015 ho avuto il piacere di insegnare questo materiale insieme al Prof. Oscar Mur-Miranda e al Prof. Siddhartan Govindasamy. Entrambi hanno fornito molti suggerimenti e correzioni.
- Silas Gyger ha corretto un errore aritmetico.
- Giuseppe Masetti ha inviato una serie di suggerimenti molto utili.
- Eric Peters ha inviato molti suggerimenti utili.

Un ringraziamento speciale a Freesound, che è la fonte di molti dei campioni sonori che uso in questo libro e agli utenti di Freesound che hanno caricato quei suoni. Includo alcuni dei loro file wave nel repository GitHub per questo libro, usando i nomi dei file originali, quindi dovrebbe essere facile trovare le loro fonti.

Sfortunatamente, la maggior parte degli utenti di Freesound non rendono disponibili i loro veri nomi, quindi posso solo ringraziarli usando i loro nomi utente. I campioni usati in questo libro sono stati forniti dagli utenti di Freesound: iluppai, wcfl10, thir-sk, docquesting, kleeb, landup, zippi1, themusicalnomad, bcjordan, rockwehrmann, marcgascon7, jcveliz. Grazie a tutti!

Indice

Prefazione	v
0.1 Per chi è questo libro?	v
0.2 Utilizzo del codice	vi
1 Suoni e Segnali	1
1.1 Segnali periodici	2
1.2 Decomposizione spettrale	3
1.3 Segnali	5
1.4 Leggere e Scrivere Wave	7
1.5 Spettri	7
1.6 Oggetti Wave	8
1.7 Oggetti Signal	9
1.8 Esercizi	11
2 Armoniche	13
2.1 Onde triangolari	13
2.2 Onde quadre	16
2.3 Aliasing	17
2.4 Calcolo dello spettro	19
2.5 Esercizi	21

3 Segnali non-periodici	23
3.1 Chirp lineare	23
3.2 Chirp esponenziale	26
3.3 Spettro di un chirp	27
3.4 Spettrogramma	28
3.5 Il limite di Gabor	29
3.6 Leakage [Perdita o dispersione]	30
3.7 Windowing	31
3.8 Implementazione di spettrogrammi	32
3.9 Esercizi	34
4 Rumore	37
4.1 Rumore non correlato	37
4.2 Spettro integrato	40
4.3 Rumore Browniano	41
4.4 Rumore Rosa	44
4.5 Rumore Gaussiano	46
4.6 Esercizi	48
5 Autocorrelazione	51
5.1 Correlazione	51
5.2 Correlazione seriale	54
5.3 Autocorrelazione	55
5.4 Autocorrelazione dei segnali periodici	56
5.5 Correlazione come prodotto scalare	60
5.6 Utilizzo di NumPy	61
5.7 Esercizi	62

6 Trasformata discreta del coseno	65
6.1 Sintesi	66
6.2 Sintesi con gli array	67
6.3 Analisi	69
6.4 Matrici ortogonali	70
6.5 DCT-IV	72
6.6 DCT inversa	73
6.7 La classe Dct	74
6.8 Esercizi	76
7 Trasformata Discreta di Fourier	77
7.1 Esponenziali complessi	78
7.2 Segnali complessi	80
7.3 Il problema della sintesi	81
7.4 Sintesi con le matrici	83
7.5 Il problema dell'analisi	85
7.6 Analisi efficiente	85
7.7 DFT	87
7.8 La DFT è periodica	88
7.9 DFT di segnali reali	89
7.10 Esercizi	91
8 Filtraggio e Convoluzione	93
8.1 Smoothing	93
8.2 Convolution	96
8.3 Il dominio della frequenza	97
8.4 Il teorema della convoluzione	99
8.5 Filtro Gaussiano	100

8.6	Convoluzione efficiente	101
8.7	Autocorrelazione efficiente	103
8.8	Esercizi	105
9	Differenziazione e Integrazione	107
9.1	Differenze finite	107
9.2	Il dominio della frequenza	109
9.3	Differenziazione	109
9.4	Integrazione	113
9.5	Somma cumulativa	114
9.6	Integrazione del rumore	118
9.7	Esercizi	119
10	Sistemi Dinamici Lineari Stazionari	121
10.1	Segnali e sistemi	121
10.2	Finestre e filtri	123
10.3	Risposta acustica	125
10.4	Sistemi e convoluzione	127
10.5	Dimostrazione del Teorema della Convoluzione	131
10.6	Esercizi	133
11	Modulazione e campionamento	135
11.1	Convoluzione con gli impulsi	135
11.2	Modulazione d'ampiezza	137
11.3	Campionamento	139
11.4	Aliasing	143
11.5	Interpolazione	146
11.6	Esercizi	148

Capitolo 1

Suoni e Segnali

Un **segnale** rappresenta una quantità che varia nel tempo. Questa definizione è piuttosto astratta, quindi iniziamo con un esempio concreto: il suono. Il suono è una variazione della pressione dell'aria. Un segnale acustico rappresenta le variazioni della pressione dell'aria nel tempo.

Un microfono è un dispositivo che misura queste variazioni e genera un segnale elettrico che rappresenta il suono. Un altoparlante è un dispositivo che riceve un segnale elettrico e produce un suono. Microfoni e altoparlanti sono detti **trasduttori** perché trasducono, o convertono, i segnali da una forma all'altra.

Questo libro tratta dell'elaborazione dei segnali, che comprende i processi per sintetizzare, trasformare e analizzare i segnali. Ci si concentrerà sui segnali sonori, ma gli stessi metodi si applicano ai segnali elettronici, alle vibrazioni meccaniche e ai segnali in molti altri domini.

Si applicano anche ai segnali che variano nello spazio piuttosto che nel tempo, come la salita lungo un sentiero escursionistico. E si applicano a segnali in più di una dimensione, come un'immagine, che si può pensare come a un segnale che varia nello spazio bidimensionale. O a un film, che è un segnale che varia nello spazio bidimensionale *e* nel tempo.

Ma iniziamo con un semplice suono unidimensionale.

Il codice per questo capitolo si trova in `chap01.ipynb`, nel repository di questo libro (vedere la Sezione 0.2). Lo si può anche visualizzare su <http://tinyurl.com/thinkdsp01>.

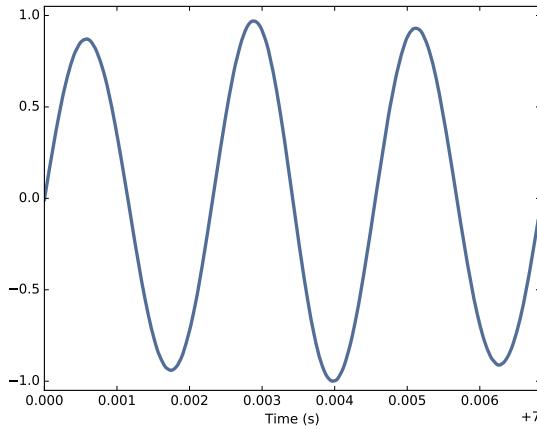


Figura 1.1: Segmento della registrazione di una campana.

1.1 Segnali periodici

Inizieremo con i **segnali periodici**, questi sono segnali che si ripetono dopo un certo periodo di tempo. Ad esempio, se si colpisce una campana, questa vibra e genera un suono. Se si registra quel suono e si disegna il segnale trasdotto, apparirà come nella Figura 1.1.

Questo segnale assomiglia a una **sinusoide**, il che significa che ha la stessa forma della funzione trigonometrica seno.

Si può vedere che questo segnale è periodico. La durata è stata scelta in modo da mostrare tre ripetizioni complete, note anche come **cicli**. La durata di ogni ciclo, detto **periodo**, è di circa 2.3 ms.

La **frequenza** di un segnale è il numero di cicli al secondo, che è l'inverso del periodo. Le unità di frequenza sono i cicli al secondo, o **Hertz**, abbreviato con "Hz". (A rigor di termini, il numero di cicli è un numero adimensionale, quindi un Hertz è davvero un "al secondo").

La frequenza di questo segnale è di circa 439 Hz, leggermente inferiore a 440 Hz, che è il tono di accordatura standard per la musica orchestrale. Il nome musicale di questa nota è A (it:LA), o più esattamente, A4 (it: LA della quarta ottava del pianoforte). Se non si ha familiarità con la "scientific pitch notation", il suffisso numerico indica in quale ottava si trova la nota. A4 è la A sopra la C (it:DO) centrale. A5 è un'ottava più alta. Vedere http://en.wikipedia.org/wiki/Scientific_pitch_notation.

Un diapason genera una sinusoide perché la vibrazione dei rebbi è una forma di moto armonico semplice. La maggior parte degli strumenti musicali produce

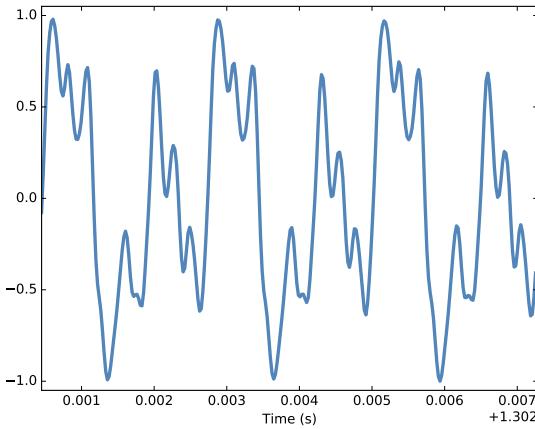


Figura 1.2: Segmento della registrazione di un violino.

segnali periodici, ma la forma di questi segnali non è sinusoidale. Per esempio, La Figura 1.2 mostra un segmento di una registrazione di un violino che suona il Quintetto per Archi n. 5 di Boccherini in E (Mi maggiore), terzo movimento.

Ancora una volta possiamo vedere che il segnale è periodico, ma la forma del segnale è più complessa. La forma di un segnale periodico è chiamata **forma-d'onda**. La maggior parte degli strumenti musicali produce forme d'onda più complesse di una sinusoide. La forma-d'onda determina il **timbro** musicale, che è la nostra percezione della qualità del suono. Le persone di solito percepiscono le forme-d'onda complesse come ricche, calde e più interessanti delle sinusoidi.

1.2 Decomposizione spettrale

L'argomento più importante in questo libro è la **decomposizione spettrale**, un'idea secondo cui qualsiasi segnale possa essere espresso come la somma di più sinusoidi con frequenze diverse.

Lo strumento matematico più importante in questo libro è la **trasformata discreta di Fourier**, o **DFT**, che prende un segnale e ne produce lo **spettro**. Lo spettro è l'insieme delle sinusoidi che si sommano per produrre il segnale.

L'algoritmo più importante in questo libro è la **Fast Fourier Transform**, o **FFT**, che è un modo efficiente per calcolare la DFT.

Per esempio, la Figura 1.3 mostra lo spettro della registrazione del violino della Figura 1.2. L'asse x è la gamma di frequenze che compongono il segnale. L'asse y mostra la forza o l'**ampiezza** di ciascuna componente di frequenza.

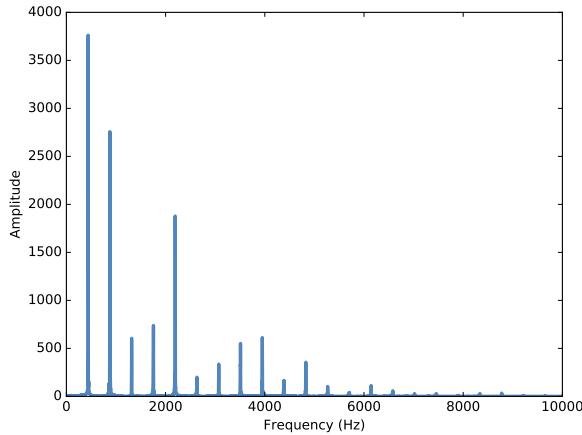


Figura 1.3: Spettro di un segmento dalla registrazione del violino.

La componente di frequenza più bassa è detta **frequenza fondamentale**. La frequenza fondamentale di questo segnale è prossima a 440 Hz (in realtà un po' più bassa, o “calante”).

In questo segnale la frequenza fondamentale ha la massima ampiezza, quindi è anche la **frequenza dominante**. Normalmente l'altezza percepita di un suono è determinata dalla frequenza fondamentale, anche se non è dominante.

Gli altri picchi nello spettro sono alle frequenze 880, 1320, 1760 e 2200, che sono multipli interi della fondamentale. Queste componenti sono dette **armoniche** perché sono musicalmente armoniose con la fondamentale:

- 880 è la frequenza di A5 (LA5), un’ottava più alta della fondamentale. Un’**ottava** è un raddoppio della frequenza.
- 1320 è approssimativamente E6 (MI6), che è una quinta perfetta sopra A5. Se non si ha familiarità con gli intervalli musicali come la "quinta perfetta", vedere [https://en.wikipedia.org/wiki/Interval_\(music\)](https://en.wikipedia.org/wiki/Interval_(music)).
- 1760 è A6, due ottave sopra la fondamentale.
- 2200 è all’incirca C \sharp 7, che è una terza maggiore sopra A6.

Queste armoniche costituiscono le note di un accordo in A maggiore, sebbene non tutte nella stessa ottava. Alcuni di essi sono solo approssimativi perché le note che compongono la musica occidentale sono state regolate per il **temperamento equabile** (vedere https://it.wikipedia.org/wiki/Temperamento_equabile).

Date le armoniche e le loro ampiezze, è possibile ricostruire il segnale sommando le sinusoidi. Successivamente vedremo come.

1.3 Segnali

Ho scritto un modulo Python chiamato `thinkdsp.py` che contiene classi e funzioni per lavorare con segnali e spettri¹. Lo si troverà nel repository di questo libro (vedere la Sezione 0.2).

Per rappresentare i segnali, `thinkdsp` fornisce una classe chiamata `Signal`, che è la classe genitore per diversi tipi di segnali, tra cui `Sinusoid`, che rappresenta sia i segnali seno che coseno.

`thinkdsp` fornisce funzioni per creare segnali seno e coseno:

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

`freq` è la frequenza in Hz. `amp` è l'ampiezza in unità non specificate dove 1.0 è definita come l'ampiezza massima che possiamo registrare o riprodurre.

`offset` è un'**offset di fase** in radianti. L'offset della fase determina dove inizia il segnale nel periodo. Ad esempio, un segnale seno con `offset=0` inizia a sin 0, che è 0. Con `offset=pi/2` inizia a sin $\pi/2$, che è 1.

I segnali hanno un metodo `__add__`, quindi si può usare l'operatore `+` per sommarli:

```
mix = sin_sig + cos_sig
```

Il risultato è un `SumSignal`, che rappresenta la somma di due o più segnali.

Un `Signal` è fondamentalmente una rappresentazione Python di una funzione matematica. La maggior parte dei segnali sono definiti per tutti i valori di `t`, da infinito negativo a infinito.

Non si può fare molto con un segnale finché non lo si valuta. In questo contesto, “valutare” significa prendere una sequenza di punti nel tempo, `ts`, e calcolare i valori corrispondenti del segnale, `ys`. Si rappresentano `ts` e `ys` usando gli array NumPy e si incapsulano in un oggetto chiamato Wave.

¹[Nota per la versione inglese] Il plurale inglese di “spectrum” è spesso scritto “spectra”, ma si preferisce usare il plurale standard inglese. Se si è abituati a “spectra”, si spera che questa scelta non risulti troppo strana (ndt: Nella versione inglese).

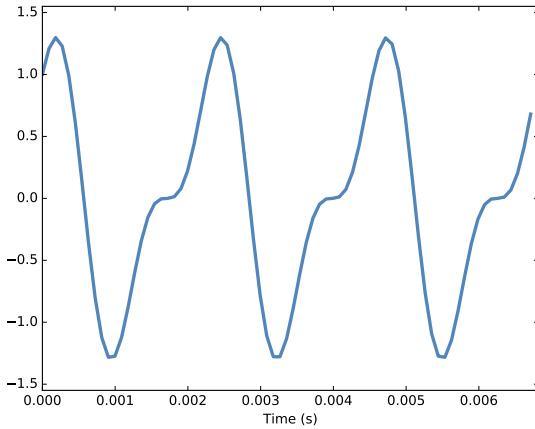


Figura 1.4: Segmento da un mix di due segnali sinusoidali.

Una Wave rappresenta un segnale valutato in una sequenza di punti nel tempo. Ogni punto nel tempo è detto **frame** (un termine preso in prestito dal mondo fotografico). La misura stessa è chiamata **campione**, sebbene a volte “frame” e “campione” siano usati in modo intercambiabile.

Signal fornisce `make_wave`, che restituisce un nuovo oggetto Wave:

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

`duration` è la lunghezza di Wave in secondi. `start` è il tempo di inizio, anch’esso in secondi. `framerate` è il numero (intero) di frame al secondo, che è anche il numero di campioni al secondo.

11,025 frame al secondo è uno dei numerosi framerate comunemente utilizzati nei formati dei file audio, tra cui Waveform Audio File (WAV) e mp3.

Questo esempio valuta il segnale da $t=0$ a $t=0.5$ con 5,513 frame equidistanti (perché 5,513 è la metà di 11,025). Il tempo tra i frame, o **timestep**, è di $1/11025$ secondi, circa $91 \mu\text{s}$.

Wave fornisce un metodo `plot` che usa `pyplot`. L’onda si può disegnare in questo modo:

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

`pyplot` fa parte di `matplotlib`; è incluso in molte distribuzioni Python o potrebbe essere necessario installarlo.

Con `freq=440` ci sono 220 periodi in 0.5 secondi, quindi questo grafico apparirebbe come una macchia colorata. Per ingrandire un piccolo numero di periodi, possiamo usare `segment`, che copia un segmento di un'onda e restituisce una nuova onda:

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

`period` è una proprietà di `Signal`; restituisce il periodo in secondi.

`start` e `duration` sono in secondi. Questo esempio copia i primi tre periodi da `mix`. Il risultato è un oggetto `Wave`.

Se si disegna `segment`, questo apparirà come nella Figura 1.4. Tale segnale contiene due componenti di frequenza, quindi è più complicato di quello del diapason, ma meno complicato del segnale del violino.

1.4 Leggere e Scrivere Wave

`thinkdsp` fornisce `read_wave`, che legge un file WAV e restituisce un `Wave`:

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

E `Wave` fornisce `write`, che scrive un file WAV:

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

Si può ascoltare un `Wave` con qualsiasi lettore multimediale che riproduce file WAV. Sui sistemi UNIX, utilizziamo `aplay`, che è semplice, robusto e incluso in molte distribuzioni Linux.

`thinkdsp` fornisce anche `play_wave`, che esegue il lettore multimediale come sottoprocesso:

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

Utilizza `aplay` per default, ma si può fornire il nome di un altro player.

1.5 Spettri

`Wave` fornisce `make_spectrum`, che restituisce uno `Spectrum`:



Figura 1.5: Relazioni tra le classi in `thinkdsp`.

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

E `Spectrum` fornisce `plot`:

```
spectrum.plot()
```

`Spectrum` fornisce tre metodi che modificano lo spettro:

- `low_pass` applica un filtro passa-basso, il che significa che le componenti al di sopra di una data frequenza (di taglio) vengono attenuate (cioè ridotte in ampiezza) di un dato fattore.
- `high_pass` applica un filtro passa-alto, il che significa che attenua le componenti al di sotto della frequenza di taglio.
- `band_stop` attenua le componenti nella banda di frequenze tra due limiti.

Questo esempio attenua tutte le frequenze superiori a 600 del 99%:

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

Un filtro passa-basso rimuove gli acuti, i suoni ad alta frequenza, quindi il risultato è ovattato e più cupo. Per sentire come suona, si può riconvertire lo spettro in un `Wave` e poi riprodurlo.

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

Il metodo `play` scrive l'onda [`wave`] in un file e poi la riproduce. Se si usano i notebook Jupyter, si può utilizzare `make_audio`, che crea un widget Audio che riproduce il suono.

1.6 Oggetti Wave

Non c'è niente di complicatissimo in `thinkdsp.py`. La maggior parte sono dei leggeri wrapper attorno alle funzioni di NumPy e SciPy.

Le classi principali in `thinkdsp` sono `Signal`, `Wave` e `Spectrum`. Dato un `Signal`, si può creare un `Wave`. Dato un `Wave`, si può creare uno `Spectrum`, e viceversa. Queste relazioni sono mostrate nella Figura 1.5.

Un oggetto `Wave` contiene tre attributi: `ys` è un array NumPy che contiene i valori nel segnale; `ts` è un array dei tempi in cui il segnale è stato valutato o campionato, e `framerate` è il numero di campioni per unità di tempo. L'unità di tempo è solitamente in secondi, ma potrebbe non esserlo. In uno degli esempi, sono giorni.

`Wave` fornisce anche tre proprietà di sola lettura: `start`, `end` e `duration`. Se si modifica `ts`, queste proprietà cambiano di conseguenza.

Per modificare un'onda, si può accedere direttamente a `ts` e a `ys`. Per esempio:

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

La prima riga scala l'onda di un fattore 2, rendendola più forte. La seconda riga sposta l'onda nel tempo, facendola iniziare 1 secondo dopo.

Ma `Wave` fornisce metodi che eseguono molte operazioni comuni. Ad esempio, le stesse due trasformazioni si potrebbero scrivere:

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

La documentazione di questi due metodi, e di altri, è disponibile su <http://greenteapress.com/thinkdsp.html>.

1.7 Oggetti Signal

`Signal` è una classe genitore che fornisce funzioni comuni a tutti i tipi di segnali, come `make_wave`. Le classi figlie ereditano questi metodi e forniscono `evaluate`, che valuta il segnale in una data sequenza di tempi.

Per esempio, `Sinusoid` è una classe figlia di `Signal`, con questa definizione:

```
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
```

I parametri di `__init__` sono:

- `freq`: frequenza in cicli al secondo, o Hz.
- `amp`: ampiezza. Le unità dell'ampiezza sono arbitrarie, solitamente scelte in modo che 1.0 corrisponda all'ingresso massimo da un microfono o all'uscita massima verso un altoparlante.
- `offset`: indica dove, nel suo periodo, inizia il segnale; l'unità `offset` è in radianti, per i motivi che verranno spiegati in seguito.
- `func`: una funzione Python usata per valutare il segnale in un particolare momento. Di solito è `np.sin` oppure `np.cos`, producendo un segnale seno o coseno.

Come molti metodi di inizializzazione, anche questo inserisce dei parametri per un uso futuro.

Signal fornisce `make_wave`, che assomiglia a questo:

```
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` e `duration` sono il tempo di inizio e la durata in secondi. `framerate` è il numero di frame (campioni) al secondo.

`n` è il numero di campioni, e `ts` è un array NumPy di tempi di campionamento.

Per calcolare la `ys`, `make_wave` invoca `evaluate`, che è fornita da `Sinusoid`:

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

Scorriamo questa funzione un passo per volta:

1. `self.freq` è la frequenza in cicli al secondo e ogni elemento di `ts` è un tempo in secondi, quindi il loro prodotto è il numero di cicli dal tempo di inizio.

2. PI2 è una costante che memorizza 2π . Moltiplicando per PI2 si converte da cicli a **fase**. Si può pensare alla fase come ai “cicli dal tempo di inizio” espressi in radianti. Ogni ciclo è 2π radianti.
3. **self.offset** è la fase quando t è **ts[0]**. Ha l’effetto di spostare il segnale a sinistra (prima) o a destra (dopo) nel tempo.
4. Se **self.func** è **np.sin** o **np.cos**, il risultato è un valore compreso tra -1 e $+1$.
5. Moltiplicando per **self.amp** si ottiene un segnale che va da **-self.amp** a **+self.amp**.

In notazione matematica, **evaluate** è scritto in questo modo:

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

dove A è l’ampiezza, f è la frequenza, t è il tempo e ϕ_0 è l’offset della fase. Può sembrare che sia stato scritto molto codice per valutare una semplice espressione, ma come vedremo, questo codice fornisce una struttura per gestire tutti i tipi di segnali, non solo le sinusoidi.

1.8 Esercizi

Prima di iniziare questi esercizi, è necessario scaricare il codice per questo libro, seguendo le istruzioni nella Sezione 0.2.

Le soluzioni a questi esercizi sono in **chap01soln.ipynb**.

Esercizio 1.1 Se si ha Jupyter, caricare **chap01.ipynb**, gli esempi vanno letti ed eseguiti. Questo notebook è visualizzabile su <http://tinyurl.com/thinkdsp01>.

Esercizio 1.2 Andare su <http://freesound.org> e scaricare un esempio audio che includa musica, parlato o altri suoni con un tono ben definito. Selezionare un segmento di circa mezzo secondo in cui il tono sia costante. Calcolare e disegnare lo spettro del segmento selezionato. Che collegamento si riesce a stabilire tra il timbro del suono e la struttura armonica che si vede nello spettro?

Usare **high_pass**, **low_pass** e **band_stop** per filtrare (eliminare) alcune delle armoniche. Poi riconvertire lo spettro in un’onda [wave] e ascoltarlo. Come si relaziona il suono alle modifiche effettuate nello spettro?

Esercizio 1.3 Sintetizza un segnale composto creando oggetti SinSignal e CosSignal e sommandoli. Valutare il segnale per ottenerne un Wave, ed ascoltarlo. Calcolarne lo spettro e disegnarlo. Cosa succede se si aggiungono componenti di frequenza che non siano multipli della fondamentale?

Esercizio 1.4 Scrivere una funzione chiamata `stretch` che prenda un Wave e un fattore di stretch e acceleri o rallenti l'onda modificando `ts` e `framerate`. Suggerimento: dovrebbero essere necessarie solo due righe di codice.

Capitolo 2

Armoniche

In questo capitolo vengono presentate diverse nuove forme d'onda; guarderemo i loro spettri per capire la loro **struttura armonica**, che è l'insieme di sinusoidi di cui sono costituiti.

Verrà introdotto anche uno dei fenomeni più importanti nell'elaborazione del segnale digitale: l'aliasing. E si spiegherà un po' di più su come funziona la classe Spectrum.

Il codice per questo capitolo si trova in `chap02.ipynb`, nel repository di questo libro (vedere la Sezione 0.2). Lo si può anche visualizzare su <http://tinyurl.com/thinkdsp02>.

2.1 Onde triangolari

Una sinusoide contiene una sola componente di frequenza, quindi il suo spettro ha un solo picco. Forme d'onda più complicate, come la registrazione del violino, producono DFT con molti picchi. In questa sezione esaminiamo la relazione tra le forme d'onda e i loro spettri.

Si inizierà con una forma d'onda triangolare, che è come una versione con segmenti rettilinei di una sinusoide. La Figura 2.1 mostra una forma d'onda triangolare con una frequenza di 200 Hz.

Per generare un'onda triangolare, si può usare `thinkdsp.TriangleSignal`:

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

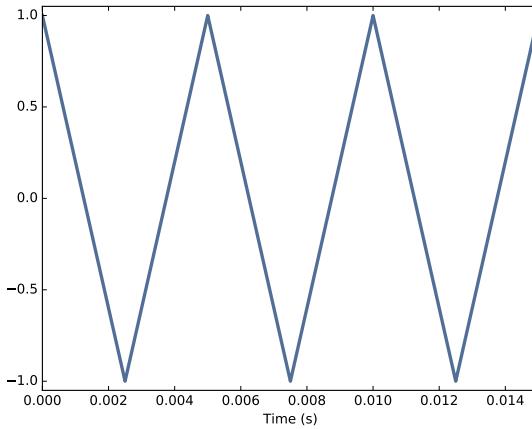


Figura 2.1: Segmento di un segnale triangolare a 200 Hz.

```

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` eredita `__init__` da `Sinusoid`, quindi accetta gli stessi argomenti: `freq`, `amp` e `offset`.

L'unica differenza è `evaluate`. Come abbiamo visto prima, `ts` è la sequenza di tempi di campionamento in cui vogliamo valutare il segnale.

Esistono molti modi per generare un'onda triangolare. I dettagli non sono importanti, ma ecco come funziona `evaluate`:

1. `cycles` è il numero di cicli dal tempo di inizio. `np.modf` divide il numero di cicli nella parte frazionaria, memorizzata in `frac`, e nella parte intera, che viene ignorata ¹.
2. `frac` è una sequenza che va da 0 a 1 con la frequenza data. Sottraendo 0.5 si ottengono valori compresi tra -0.5 e 0.5. Prendendo il valore assoluto si ottiene una forma d'onda che va a zig-zag tra 0.5 e 0.
3. `unbias` sposta la forma d'onda verso il basso in modo che sia centrata su 0; quindi `normalizza`, scalandolo, all'ampiezza data, `amp`.

¹L'uso di un trattino basso come nome di una variabile è una convenzione che significa, "Non intendo usare questo valore".

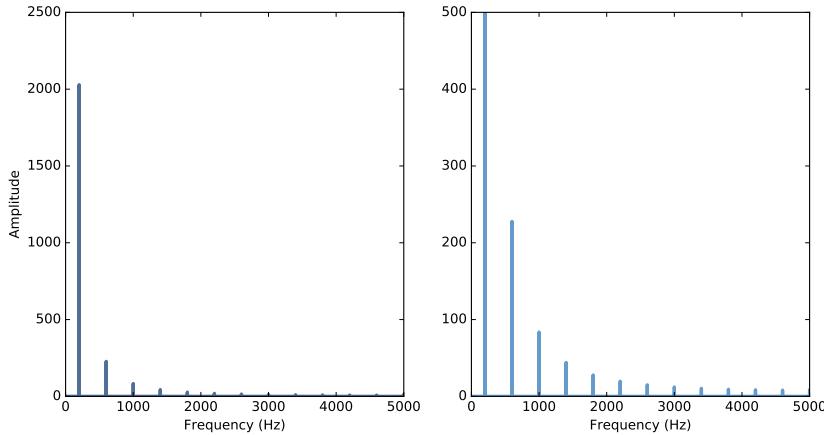


Figura 2.2: Lo spettro di un segnale triangolare a 200 Hz, mostrato su due scale verticali. La versione a destra taglia via la fondamentale per mostrare più chiaramente le armoniche.

Ecco il codice che genera la Figura 2.1:

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

Poi possiamo usare il segnale per creare una Wave per poi creare uno Spectrum:

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

La Figura 2.2 mostra due videate del risultato; quella a destra viene ridimensionata per mostrare più chiaramente le armoniche. Come previsto, il picco più alto è alla frequenza fondamentale, 200 Hz e ci sono picchi aggiuntivi alle frequenze delle armoniche, che sono multipli interi di 200.

Ma una sorpresa è che non ci sono picchi ai multipli pari: 400, 800, ecc. Le armoniche di un'onda triangolare sono tutte multiple dispari della frequenza fondamentale, in questo esempio 600, 1000, 1400, ecc.

Un'altra caratteristica di questo spettro è la relazione tra l'ampiezza e la frequenza delle armoniche. La loro ampiezza diminuisce in proporzione al quadrato della frequenza. Ad esempio, il rapporto di frequenza delle prime due armoniche (200 e 600 Hz) è 3 e il rapporto di ampiezza è di circa 9. Il rapporto di frequenza delle due armoniche successive (600 e 1000 Hz) è 1.7 e il rapporto di ampiezza è di circa $1.7^2 = 2.9$. Questa relazione è detta **struttura armonica**.

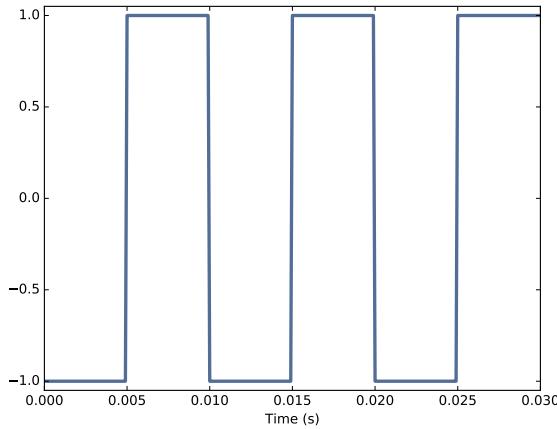


Figura 2.3: Segmento di un segnale quadrato a 100 Hz.

2.2 Onde quadre

thinkdsp fornisce anche `SquareSignal`, che rappresenta un segnale quadrato. Ecco la definizione della classe:

```
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
```

Come `TriangleSignal`, `SquareSignal` eredita `__init__` da `Sinusoid`, quindi accetta gli stessi parametri.

Il metodo `evaluate` è simile. Di nuovo, `cycles` è il numero di cicli dal tempo di inizio e `frac` è la parte frazionaria, che aumenta da 0 a 1 ogni periodo.

`unbias` sposta `frac` in modo da aumentare da -0.5 a 0.5, poi `np.sign` mappa i valori negativi a -1 e i valori positivi a 1. Moltiplicando per `amp` si ottiene un'onda quadra che salta tra `-amp` e `amp`.

La Figura 2.3 mostra tre periodi di un'onda quadra con frequenza di 100 Hz e la Figura 2.4 ne mostra lo spettro.

Come l'onda triangolare, quella quadra contiene solo armoniche dispari, motivo per cui ci sono picchi a 300, 500 e 700 Hz, ecc. Ma l'ampiezza delle armoniche

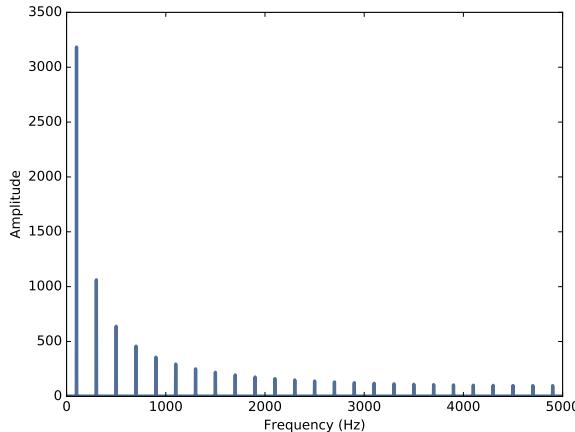


Figura 2.4: Spettro di un segnale quadrato a 100 Hz.

diminuisce più lentamente. Nello specifico, l’ampiezza scende in proporzione alla frequenza (non al quadrato della frequenza).

Gli esercizi alla fine di questo capitolo danno la possibilità di esplorare altre forme d’onda e altre strutture armoniche.

2.3 Aliasing

C’è una confessione. Gli esempi, nella sezione precedente, sono stati scelti appositamente per evitare di mostrare qualcosa di confuso. Ma ora è il momento di confondersi.

La Figura 2.5 mostra lo spettro di un’onda triangolare a 1100 Hz, campionata a 10,000 frame al secondo. Di nuovo, la vista a destra viene ridimensionata per mostrare le armoniche.

Le armoniche di quest’onda dovrebbero essere a 3300, 5500, 7700 e 9900 Hz. Nella figura, ci sono picchi a 1100 e 3300 Hz, come previsto, ma il terzo picco è a 4500, non a 5500 Hz. Il quarto picco è a 2300, non a 7700 Hz. E se si guarda da vicino, il picco che dovrebbe essere a 9900 è in realtà a 100 Hz. Cosa succede?

Il problema è che quando si valuta il segnale in punti discreti nel tempo, si perdono informazioni su ciò che è accaduto tra i campioni. Per le componenti a bassa frequenza, questo non è un problema, perché si hanno molti campioni per ciascun periodo.

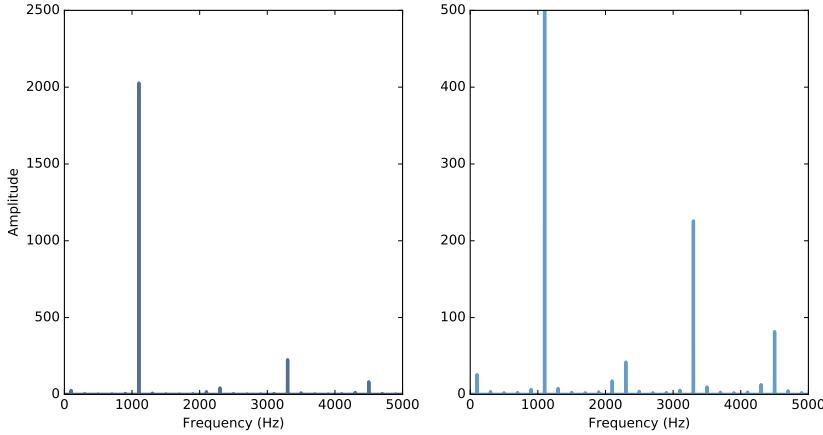


Figura 2.5: Spettro di un segnale triangolare a 1100 Hz campionato a 10.000 frame al secondo. La vista a destra è ridimensionata per mostrare le armoniche.

Ma se si campiona un segnale a 5000 Hz con 10,000 frame al secondo, si hanno solo due campioni per periodo. Questo risulta essere appena sufficiente, ma se la frequenza è più alta, non lo è più.

Per capire perché, generiamo segnali coseno a 4500 e 5500 Hz e campioniamoli a 10,000 frame al secondo:

```
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()
```

La Figura 2.6 mostra il risultato. I segnali sono stati disegnati con sottili linee grigie e i campioni con delle linee verticali, per facilitare il confronto delle due onde. Il problema dovrebbe essere chiaro: anche se i segnali sono diversi, le onde sono identiche!

Quando si campiona un segnale di 5500 Hz a 10,000 frame al secondo, il risultato è indistinguibile da un segnale di 4500 Hz. Per lo stesso motivo, un segnale di 7700 Hz è indistinguibile da quello a 2300 Hz e un segnale di 9900 Hz è indistinguibile da quello di 100 Hz.

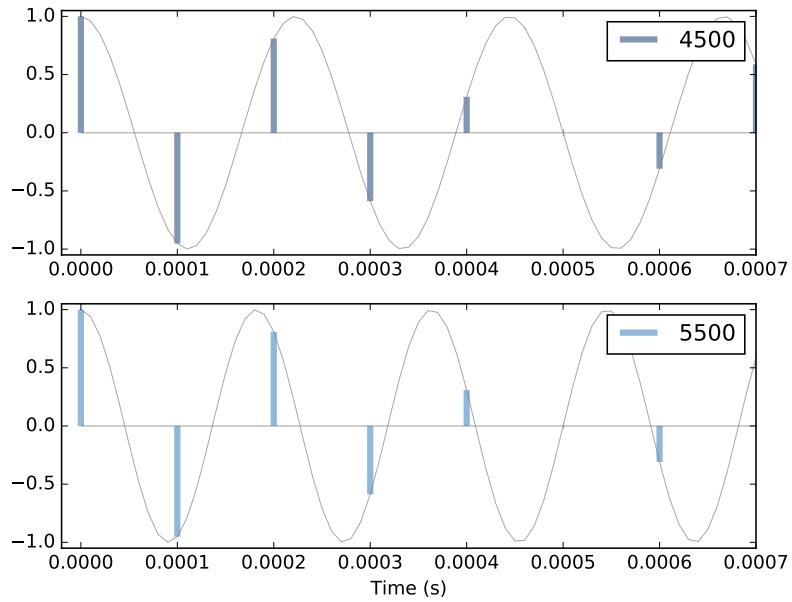


Figura 2.6: Segnali coseno a 4500 e 5500 Hz, campionati a 10,000 frame al secondo. I segnali sono diversi, ma i campioni sono identici.

Questo effetto è chiamato **aliasing** [copia] perché quando viene campionato il segnale ad alta frequenza, sembra essere un segnale a bassa frequenza.

In questo esempio, la frequenza più alta che possiamo misurare è 5000 Hz, che è la metà della frequenza di campionamento. Le frequenze superiori a 5000 Hz vengono ripiegate sotto i 5000 Hz, motivo per cui questa soglia viene talvolta chiamata “folding frequency” [frequenza di ripiegamento]. A volte è anche detta **frequenza di Nyquist**. Vedere http://en.wikipedia.org/wiki/Nyquist_frequency.

Lo schema del ripiegamento continua se la frequenza con aliasing scende sotto lo zero. Ad esempio, la 5a armonica dell'onda triangolare di 1100 Hz è a 12,100 Hz. Ripiegata a 5000 Hz, apparirebbe a -2100 Hz, ma viene ripiegata di nuovo a 0 Hz, e ancora a 2100 Hz. In effetti, si può vedere un piccolo picco a 2100 Hz nella Figura 2.4, e il successivo a 4300 Hz.

2.4 Calcolo dello spettro

Abbiamo visto più volte il metodo `make_spectrum` di Wave. Ecco l'implementazione (tralasciando alcuni dettagli di cui parleremo in seguito):

```

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)

```

Il parametro `self` è un oggetto `Wave`. `n` è il numero di campioni nell'onda, e `d` è l'inverso del frame rate, che è il tempo tra i campioni.

`np.fft` è il modulo NumPy che fornisce funzioni relative alla **Fast Fourier Transform** (FFT), un algoritmo efficiente che calcola la Trasformata Discreta di Fourier (Discrete Fourier Transform) (DFT).

`make_spectrum` usa `rfft`, che sta per “real FFT”, perché `Wave` contiene valori reali, non complessi. Più avanti vedremo la FFT completa, in grado di gestire segnali complessi. Il risultato di `rfft`, chiamato `hs`, è un array NumPy di numeri complessi che rappresenta l'ampiezza e l'offset della fase di ogni componente di frequenza nell'onda.

Il risultato di `rfftfreq`, chiamato `fs`, è un array che contiene frequenze corrispondenti a `hs`.

Per comprendere i valori in `hs`, si considerino questi due modi di pensare ai numeri complessi:

- Un numero complesso è la somma di una parte reale e di una parte immaginaria, spesso scritta $x + iy$, dove i è l'unità immaginaria, $\sqrt{-1}$. Si può pensare a x e y come alle coordinate Cartesiane.
- Un numero complesso è anche il prodotto di una grandezza e di un esponenziale complesso, $Ae^{i\phi}$, dove A è l'**ampiezza** e ϕ è l'**angolo** in radianti, detto anche “argomento”. Si può pensare ad A e a ϕ come coordinate polari.

Ogni valore in `hs` corrisponde a una componente di frequenza: la sua grandezza è proporzionale all'ampiezza della componente corrispondente; il suo angolo è l'offset di fase.

La classe `Spectrum` fornisce due proprietà a sola lettura, `amps` e `angles`, che restituiscono array NumPy che rappresentano le ampiezze e gli angoli di `hs`. Quando si disegna un oggetto `Spectrum`, di solito si disegna `amps` rispetto a `fs`. A volte è anche utile disegnare `angles` rispetto a `fs`.

Anche se si potrebbe essere tentati di guardare le parti reali e immaginarie di `hs`, non se ne avrà quasi mai bisogno. Si incoraggia a pensare al DFT come a un vettore di ampiezze e sfasamenti che sono codificati sotto forma di numeri complessi.

Per modificare uno spettro, è possibile accedere direttamente ad `hs`. Per esempio:

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

La prima riga moltiplica gli elementi di `hs` per 2, il che raddoppia le ampiezze di tutte le componenti. La seconda riga imposta a 0 solo gli elementi di `hs` dove la frequenza corrispondente supera una certa frequenza di taglio.

Ma `Spectrum` fornisce anche dei metodi per eseguire queste operazioni:

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

La documentazione di questi due metodi, e di altri, è disponibile su <http://greenteapress.com/thinkdsp.html>.

A questo punto si dovrebbe avere un'idea migliore di come funzionano le classi `Signal`, `Wave` e `Spectrum`, ma non è stato spiegato come funziona la Fast Fourier Transform. Questo richiederà alcuni capitoli in più.

2.5 Esercizi

Le soluzioni a questi esercizi sono in `chap02soln.ipynb`.

Esercizio 2.1 Se si usa Jupyter, caricare `chap02.ipynb` e provare gli esempi. Si può anche visualizzare il notebook su <http://tinyurl.com/thinkdsp02>.

Esercizio 2.2 Un segnale a dente di sega ha una forma d'onda che aumenta linearmente da -1 a 1, quindi scende a -1 e si ripete. Vedere http://en.wikipedia.org/wiki/Sawtooth_wave

Scrivere una classe chiamata `SawtoothSignal` che estenda `Signal` e fornisca `evaluate` per valutare un segnale a dente di sega.

Calcolare lo spettro di un'onda a dente di sega. Come si confronta la struttura armonica con le onde triangolari e quadrate?

Esercizio 2.3 Creare un segnale quadrato a 1100 Hz e creare un'onda che lo campiona a 10000 frame al secondo. Se si disegna lo spettro, si può vedere che la maggior parte delle armoniche ha un alias. Quando si ascolta l'onda, si riescono a sentire le armoniche con alias?

Esercizio 2.4 Se si ha un oggetto spettro, `spectrum`, e se ne stampano i primi valori di `spectrum.fs`, si vedrà che iniziano da zero. Quindi `spectrum.hs[0]` è l'ampiezza della componente con frequenza 0. Ma cosa significa?

Provare questo esperimento:

1. Creare un segnale triangolare con frequenza 440 e creare una Wave con durata di 0.01 secondi. Disegnare la forma d'onda.
2. Creare un oggetto Spectrum e stampare `spectrum.hs[0]`. Qual è l'ampiezza e la fase di questa componente?
3. Impostare `spectrum.hs[0] = 100`. Creare una Wave dallo Spectrum modificato e disegnarla. Che effetto ha questa operazione sulla forma d'onda?

Esercizio 2.5 Scrivere una funzione che prenda uno spettro come parametro e lo modifichi dividendo ogni elemento di `hs` per la frequenza corrispondente da `fs`. Suggerimento: poiché la divisione per zero non è definita, si potrebbe voler impostare `spectrum.hs[0] = 0`.

Verificare la propria funzione utilizzando un'onda quadrata, triangolare o a dente di sega.

1. Calcolarne uno Spectrum e disegnarlo.
2. Modificare Spectrum usando la propria funzione e disegnarlo di nuovo.
3. Creare una Wave dallo Spectrum modificato e ascoltarlo. Che effetto ha questa operazione sul segnale?

Esercizio 2.6 Le onde triangolari e quadre hanno solo armoniche dispari; l'onda a dente di sega ha armoniche sia pari che dispari. Le armoniche delle onde quadrate e a dente di sega diminuiscono in proporzione a $1/f$; le armoniche dell'onda triangolare si riducono come $1/f^2$. Si riesce a trovare una forma d'onda che abbia armoniche pari e dispari che cadono come $1/f^2$?

Suggerimento: Ci sono due approcci: si potrebbe costruire il segnale sommando le sinusoidi, oppure si potrebbe iniziare con un segnale che è simile a quello desiderato e modificarlo.

Capitolo 3

Segnali non-periodici

I segnali con cui abbiamo lavorato finora sono periodici, il che significa che si ripetono per sempre. Significa anche che le componenti in frequenza contenute non cambiano nel tempo. In questo capitolo, consideriamo i segnali non periodici, le cui componenti in frequenza *cambiano* nel tempo. In altre parole, praticamente tutti i segnali sonori.

Questo capitolo presenta anche gli spettrogrammi, un modo comune per visualizzare i segnali non periodici.

Il codice per questo capitolo si trova in `chap03.ipynb`, nel repository di questo libro (vedere la Sezione 0.2). Lo si può anche visualizzare su <http://tinyurl.com/thinkdsp03>.

3.1 Chirp lineare

Inizieremo con un **chirp** [cinguettìo], che è un segnale con una frequenza variabile. `thinkdsp` fornisce un segnale chiamato `Chirp` che crea una sinusode che scorre linearmente attraverso una gamma di frequenze.

Ecco un esempio che spazia da 220 a 880 Hz, ovvero due ottave da A3 ad A5:

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

La Figura 3.1 mostra i segmenti di quest'onda in prossimità dell'inizio, del centro e della fine. È chiaro che la frequenza va aumentando.

Prima di andare avanti, vediamo come viene implementata `Chirp`. Ecco la definizione della classe:

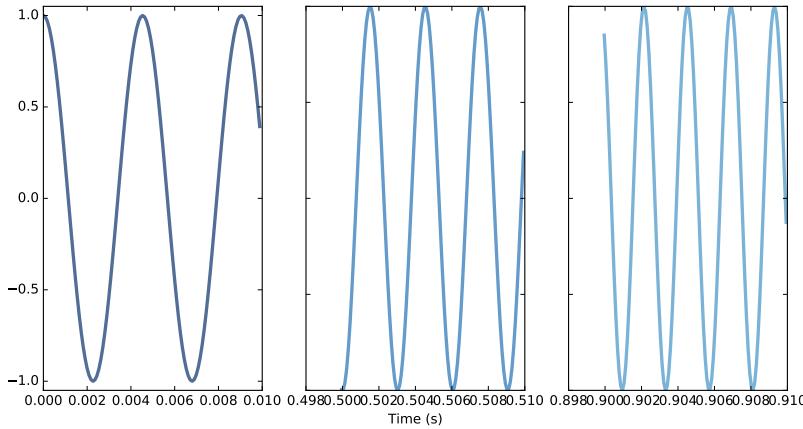


Figura 3.1: La forma d'onda di un 'chirp' all'inizio, nel mezzo e alla fine.

```
class Chirp(Signal):
```

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

`start` ed `end` sono le frequenze, in Hz, all'inizio e alla fine del chirp. `amp` è l'ampiezza.

Ecco la funzione che valuta il segnale:

```
def evaluate(self, ts):
    freqs = np.linspace(self.start, self.end, len(ts))
    dts = np.diff(ts, prepend=0)
    dphis = PI2 * freqs * dts
    phases = np.cumsum(dphis)
    ys = self.amp * np.cos(phases)
    return ys
```

`ts` è la sequenza di punti nel tempo in cui il segnale dovrebbe essere valutato; per semplificare questa funzione, si suppone che siano equidistanti.

Per calcolare la frequenza in ogni momento, si utilizza `np.linspace`, che restituisce un array NumPy di n valori compresi tra `start` e `end`.

`np.diff` calcola la differenza tra elementi adiacenti di `ts`, restituendo la lunghezza di ogni intervallo in secondi. Se gli elementi di `ts` sono equidistanti, i `dts` sono tutti uguali.

Il passo successivo è capire di quanto cambia la fase durante ogni intervallo. Nella Sezione 1.7 abbiamo visto che quando la frequenza è costante, la fase, ϕ , aumenta linearmente nel tempo:

$$\phi = 2\pi f t$$

Quando la frequenza è una funzione del tempo, il *cambio* in fase durante un breve intervallo di tempo, Δt è:

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

In Python, poiché `freqs` contiene $f(t)$ e `dts` contiene gli intervalli di tempo, possiamo scrivere:

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

Ora, poiché `dphis` contiene le modifiche in fase, possiamo ottenere la fase totale in ogni passo temporale sommando le modifiche:

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

`np.cumsum` calcola la somma cumulativa, che è quasi quella che vogliamo, ma non inizia da 0. Poi si usa `np.insert` per aggiungere uno 0 all'inizio.

Il risultato è un array NumPy dove l'elemento *i*-esimo contiene la somma dei primi *i* termini di `dphis`; ovvero la fase totale alla fine dell'intervallo *i*-esimo. Infine, `np.cos` calcola l'ampiezza dell'onda in funzione della fase (ci si ricordi che la fase è espressa in radianti).

Se si conosce il calcolo, si noterà che il limite, quando Δt diventa piccolo, è:

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

Dividendo per dt si ottiene

$$\frac{d\phi}{dt} = 2\pi f(t)$$

In altre parole, la frequenza è la derivata della fase. Al contrario, la fase è l'integrale della frequenza. Quando abbiamo usato `cumsum` per passare dalla frequenza alla fase, stavamo approssimando l'integrazione.

3.2 Chirp esponenziale

Quando si ascolta questo chirp [cinguettìo], si noterà che il tono all'inizio sale rapidamente e poi rallenta. Il chirp si estende su due ottave, ma occorrono solo 2/3 s per estendere la prima ottava e il doppio per estendere la seconda.

Il motivo è che la nostra percezione dell'altezza dipende dal logaritmo della frequenza. Di conseguenza, l'**intervallo** che sentiamo tra due note dipende dal *rappporto* delle loro frequenze, non dalla differenza. “Intervallo” è il termine musicale per la differenza percepita tra due altezze.

Ad esempio, un'ottava è un intervallo in cui il rapporto tra due altezze è 2. Quindi l'intervallo da 220 a 440 è un'ottava e anche l'intervallo da 440 a 880 è un'ottava. La differenza di frequenza è maggiore, ma il rapporto è lo stesso.

Di conseguenza, se la frequenza aumenta linearmente, come in un chirp lineare, l'altezza percepita aumenta logaritmicamente.

Se si desidera che il tono percepito aumenti in modo lineare, la frequenza deve aumentare in modo esponenziale. Un segnale con quella forma è detto **chirp esponenziale**.

Ecco il codice che ne crea uno:

```
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)
```

Invece di `np.linspace`, questa versione di 'evaluate' usa `np.logspace`, che crea una serie di frequenze i cui logaritmi sono equidistanti, il che significa che aumentano in modo esponenziale.

Questo è tutto; il resto è lo stesso di Chirp. Ecco il codice che ne crea uno:

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

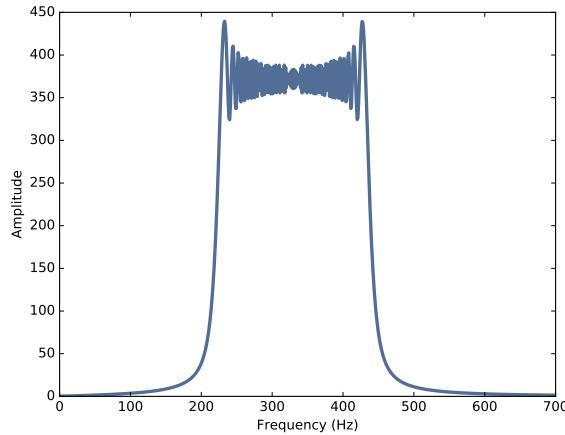


Figura 3.2: Spettro di un secondo un'ottava di chirp.

Questi esempi si possono ascoltare in `chap03.ipynb` e confrontare i chirp lineari e quelli esponenziali.

3.3 Spettro di un chirp

Cosa succederà se si calcola lo spettro di un chirp? Ecco un esempio che costruisce un chirp di un secondo, per un'ottava, e il suo spettro:

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

La Figura 3.2 mostra il risultato. Lo spettro ha componenti ad ogni frequenza da 220 a 440 Hz, con variazioni che assomigliano un po' all'Occhio di Sauron (vedere <http://en.wikipedia.org/wiki/Sauron>).

Lo spettro è approssimativamente piatto tra 220 e 440 Hz, indicando che il segnale trascorre lo stesso tempo a ciascuna frequenza in questo intervallo. Sulla base di questa osservazione, si dovrebbe essere in grado di indovinare l'aspetto dello spettro di un chirp esponenziale.

Lo spettro fornisce suggerimenti sulla struttura del segnale, ma nasconde la relazione tra frequenza e tempo. Ad esempio, non possiamo dire, guardando questo spettro, se la frequenza è aumentata o diminuita, o entrambe.

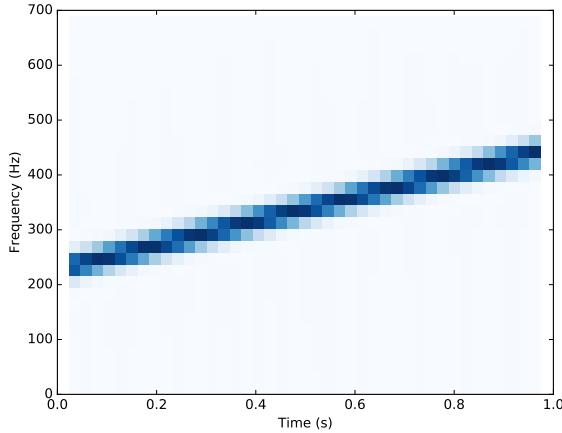


Figura 3.3: Spettrogramma di un secondo, un'ottava di un chirp.

3.4 Spettrogramma

Per recuperare la relazione tra frequenza e tempo, possiamo suddividere il chirp in segmenti e disegnare lo spettro di ogni segmento. Il risultato è detto **Trasformata di Fourier di breve durata** (STFT).

Esistono diversi modi per visualizzare uno STFT, ma il più comune è uno **spettrogramma**, che mostra il tempo sull'asse x e la frequenza sull'asse y. Ciascuna colonna dello spettrogramma mostra lo spettro di un breve segmento, utilizzando il colore o la scala di grigi per rappresentare l'ampiezza.

Per esempio, si calcolerà lo spettrogramma di questo chirp:

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

Wave fornisce `make_spectrogram`, che restituisce un oggetto `Spectrogram`:

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

`seg_length` è il numero di campioni in ogni segmento. Si è scelto 512 perché la FFT è più efficiente quando il numero di campioni è una potenza di 2.

La Figura 3.3 mostra il risultato. L'asse x mostra il tempo da 0 a 1 secondo. L'asse y mostra la frequenza da 0 a 700 Hz. È stata tagliata la parte superiore dello spettrogramma; l'intera gamma va a 5512.5 Hz, che è la metà del framerate.

Lo spettrogramma mostra chiaramente che la frequenza aumenta linearmente nel tempo. Allo stesso modo, nello spettrogramma di un chirp esponenziale, possiamo vedere la forma della curva esponenziale.

Tuttavia, si noti che il picco in ciascuna colonna è sfocato su 2-3 celle. Questa sfocatura riflette la limitata risoluzione dello spettrogramma.

3.5 Il limite di Gabor

La **risoluzione temporale** dello spettrogramma è la durata dei segmenti, che corrisponde alla larghezza delle celle nello spettrogramma. Poiché ogni segmento è 512 frame e ci sono 11,025 frame al secondo, la durata di ogni segmento è di circa 0,046 secondi.

La **risoluzione in frequenza** è l'intervallo di frequenza tra gli elementi nello spettro, che corrisponde all'altezza delle celle. Con 512 frame, otteniamo 256 componenti di frequenza in un intervallo da 0 a 5512.5 Hz, quindi l'intervallo tra le componenti è 21.6 Hz.

Più in generale, se n è la lunghezza del segmento, lo spettro contiene $n/2$ componenti. Se il framerate è r , la frequenza massima nello spettro è $r/2$. Quindi la risoluzione temporale è n/r e la risoluzione in frequenza è:

$$\frac{r/2}{n/2}$$

che è r/n .

Idealmente vorremmo che la risoluzione temporale fosse piccola, in modo da poter vedere i cambiamenti rapidi della frequenza. E vorremmo che la risoluzione in frequenza fosse piccola in modo da poter vedere piccoli cambiamenti nella frequenza. Ma non si possono avere entrambi. Si noti che la risoluzione temporale, n/r , è l'inverso della risoluzione in frequenza, r/n . Quindi se uno diventa più piccolo, l'altro diventa più grande.

Ad esempio, se si raddoppia la lunghezza del segmento, si dimezza la risoluzione della frequenza (il che è positivo), ma si raddoppia la risoluzione temporale (che è male). Anche aumentare il framerate non aiuta. Si ottengono più campioni, ma la gamma di frequenze aumenta allo stesso tempo.

Questo compromesso è chiamato il **limite di Gabor** ed è un limite fondamentale di questo tipo di analisi tempo-frequenza.

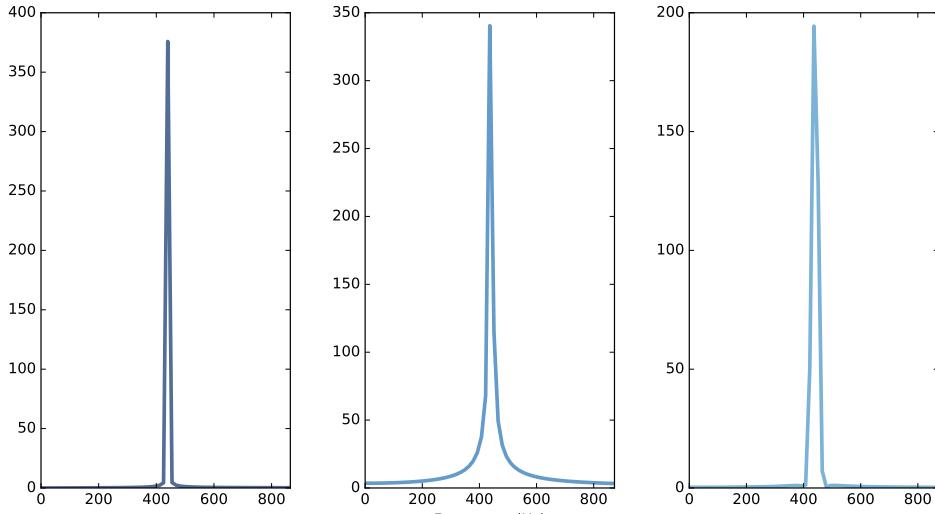


Figura 3.4: Spettro di un segmento periodico di una sinusode (a sinistra), un segmento non periodico (al centro), un segmento non periodico con finestra (a destra).

3.6 Leakage [Perdita o dispersione]

Per spiegare come funziona `make_spectrogram`, si deve spiegare il windowing; e per spiegare il windowing, si deve mostrare il problema che si intende affrontare, che è il leakage [perdita].

La Trasformata Discreta di Fourier (DFT), utilizzata per calcolare gli spettri, tratta le onde come se fossero periodiche; cioè, assume che il segmento finito su cui opera sia un periodo completo di un segnale infinito che si ripete nel tempo. Nella pratica, questa ipotesi è spesso falsa, il che crea problemi.

Un problema comune è la discontinuità all'inizio e alla fine del segmento. Poiché la DFT presume che il segnale sia periodico, collega implicitamente la fine del segmento al suo inizio per creare un loop [ciclo continuo]. Se la fine non si connette agevolmente all'inizio, la discontinuità crea componenti di frequenza aggiuntive nel segmento che non esistono nel segnale.

Ad esempio, iniziamo con un segnale sinusoidale che contiene solo una componente di frequenza a 440 Hz.

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

Se si seleziona un segmento che sembra essere un multiplo intero del periodo, la fine del segmento si collega senza problemi con l'inizio e la DFT si comporta bene.

```

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

```

La Figura 3.4 (a sinistra) mostra il risultato. Come previsto, c'è un singolo picco a 440 Hz.

Ma se la durata non è un multiplo del periodo, accadono cose strane. Con `duration = signal.period * 30.25`, il segnale inizia da 0 e termina a 1.

La figura 3.4 (al centro) mostra lo spettro di questo segmento. Anche in questo caso, il picco è a 440 Hz, ma ora ci sono componenti aggiuntive distribuite da 240 a 640 Hz. Questa diffusione è chiamata **leakage spettrale** [perdita], perché parte dell'energia che è effettivamente alla frequenza fondamentale si disperde in altre frequenze.

In questo esempio, la perdita si verifica perché stiamo utilizzando la DFT su un segmento che diventa discontinuo quando lo trattiamo come periodico.

3.7 Windowing

Possiamo ridurre la dispersione [leakage] attenuando la discontinuità tra l'inizio e la fine del segmento, e un modo per farlo è il **windowing** [suddivisione con finestre].

Una “window” [finestra] è una funzione progettata per trasformare un segmento non periodico in qualcosa che possa passare per periodico. La Figura 3.5 (in alto) mostra un segmento in cui la fine non si connette agevolmente all'inizio.

La Figura 3.5 (al centro) mostra una “finestra di Hamming”, una delle funzioni di windowing più comuni. Nessuna funzione di windowing è perfetta, ma si può dimostrare che alcune sono ottimali per diverse applicazioni e la Hamming è una buona finestra per tutti gli usi.

La Figura 3.5 (in basso) mostra il risultato della moltiplicazione della finestra per il segnale originale. Quando la finestra è prossima a 1, il segnale è invariato. Dove la finestra è prossima a 0, il segnale è attenuato. Poiché la finestra si assottiglia ad entrambe le estremità, la fine del segmento si collega dolcemente all'inizio.

La Figura 3.4 (a destra) mostra lo spettro del segnale col windowing. Il windowing ha ridotto il leakage [le perdite] in modo sostanziale, ma non completamente.

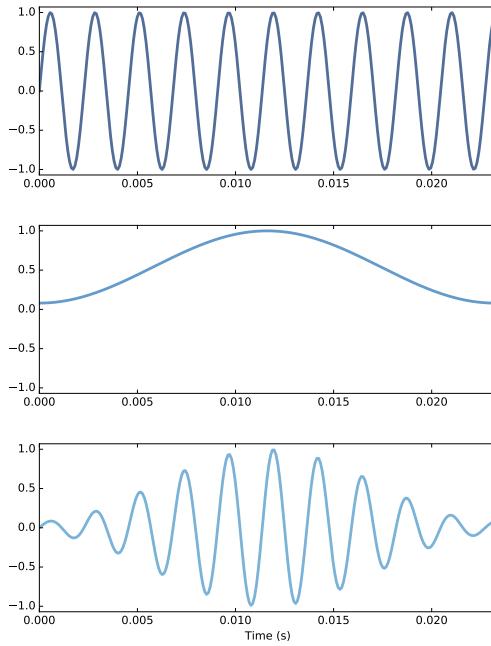


Figura 3.5: Segmento di una sinusode (in alto), finestra di Hamming (al centro), prodotto del segmento e della finestra (in basso).

Ecco come appare il codice. Wave fornisce `window`, che applica una finestra di Hamming:

```
#class Wave:
    def window(self, window):
        self.y *= window
```

E NumPy fornisce `hamming`, che calcola una finestra di Hamming con una data lunghezza:

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

NumPy fornisce funzioni per calcolare altre funzioni di windowing, tra cui `bartlett`, `blackman`, `hanning`, e `kaiser`. Uno degli esercizi alla fine di questo capitolo chiede di sperimentare con queste altre finestre.

3.8 Implementazione di spettrogrammi

Ora che comprendiamo il windowing, possiamo capire l'implementazione dello spettrogramma. Ecco il metodo `Wave` che calcola gli spettrogrammi:

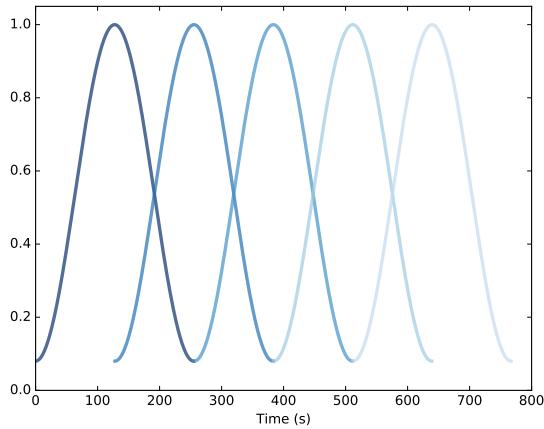


Figura 3.6: Finestre di Hamming sovrapposte.

```
#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)
```

Questa è la funzione più lunga del libro, quindi se si riesce a gestirla, si può gestire qualsiasi cosa.

Il parametro, `self`, è un oggetto Wave. `seg_length` è il numero di campioni in ogni segmento.

`window` è una finestra di Hamming con la stessa lunghezza dei segmenti.

`i` e `j` sono gli indici delle fette che selezionano i segmenti dall'onda. `step` è l'offset tra i segmenti. Poiché `step` è la metà di `seg_length`, i segmenti si sovrappongono per metà. La Figura 3.6 mostra l'aspetto di queste finestre sovrapposte.

`spec_map` è un 'dictionary' che mappa da un timestamp a uno Spectrum.

All'interno del ciclo while, si seleziona una porzione dall'onda e si applica la finestra; quindi si costruisce un oggetto Spectrum e lo si aggiunge a `spec_map`. Il tempo nominale di ogni segmento, `t`, è il punto medio.

Poi si incrementano `i` e `j`, e si continua finché `j` non va oltre la fine di Wave.

Infine, il metodo costruisce e restituisce uno spettrogramma. Ecco la definizione di Spectrogram:

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

Come molti metodi init, anche questo memorizza i parametri come attributi.

`Spectrogram` fornisce `plot`, che genera un grafico con pseudo-colori con il tempo sull'asse x e la frequenza sull'asse y.

Ed è così che vengono implementati gli spettrogrammi.

3.9 Esercizi

Le soluzioni a questi esercizi sono in `chap03soln.ipynb`.

Esercizio 3.1 Eseguire e ascoltare gli esempi in `chap03.ipynb`, nel repository di questo libro disponibile anche su <http://tinyurl.com/thinkdsp03>.

Nell'esempio del leakage, provare a sostituire la finestra di Hamming con una delle altre finestre fornite da NumPy e vedere quale effetto hanno sul leakage. Vedere <http://docs.scipy.org/doc/numpy/reference/routines.window.html>

Esercizio 3.2 Scrivere una classe chiamata `SawtoothChirp` che estende `Chirp` e sovrappone [override] `evaluate` per generare una forma d'onda a dente di sega con la frequenza che aumenta (o diminuisce) linearmente.

Suggerimento: combinare le funzioni di valutazione di `Chirp` e `SawtoothSignal`.

Disegnare uno schizzo di come si pensa che sia lo spettrogramma di questo segnale, e disegnarlo. L'effetto dell'aliasing dovrebbe essere visivamente evidente e, se si ascolta attentamente, lo si può sentire.

Esercizio 3.3 Creare un chirp a dente di sega che va da 2500 a 3000 Hz, e usarlo per creare un'onda con durata 1 s e framerate 20 kHz. Disegnare uno schizzo di come si pensa che sarà lo spettro. Poi plottare lo spettro e vedere se si è capito bene.

Esercizio 3.4 Nella terminologia musicale, un “glissando” è una nota che scorre da un tono all’altro, quindi è simile a un chirp.

Trovare o registrare un glissando e disegnare uno spettrogramma dei primi secondi. Un suggerimento: la *Rhapsody in Blue* di George Gershwin inizia con un famoso glissando per clarinetto, che si può scaricare da <http://archive.org/details/rhapblue11924>.

Esercizio 3.5 Un suonatore di trombone può suonare un glissando estendendo il ‘tiro’ del trombone soffiando continuamente. Man mano che il ‘tiro’ si estende, la lunghezza totale del tubo si allunga e il passo risultante è inversamente proporzionale alla lunghezza.

Supponendo che il musicista sposti il ‘tiro’ a una velocità costante, come varia la frequenza nel tempo?

Scrivere una classe chiamata `TromboneGliss` che estende `Chirp` e fornisce `evaluate`. Creare un’onda che simuli un glissando di trombone da C3 a F3 e di nuovo in C3. C3 è 262 Hz; F3 è 349 Hz.

Disegnare uno spettrogramma dell’onda risultante. Un glissando di trombone è più simile a un chirp lineare o a uno esponenziale?

Esercizio 3.6 Creare o trovare una registrazione di una serie di suoni vocalici e osservarne lo spettrogramma. Si riescono ad identificare le diverse vocali?

Capitolo 4

Rumore

In inglese, “noise” [rumore] indica un suono indesiderato o sgradevole. Nel contesto dell’elaborazione del segnale, ha due diversi significati:

1. Come in inglese, può significare un segnale indesiderato di qualsiasi tipo. Se due segnali interferiscono tra loro, ciascun segnale considererebbe l’altro come rumore.
2. “Rumore” si riferisce anche a un segnale che contiene componenti a molte frequenze, quindi manca la struttura armonica dei segnali periodici vista nei capitoli precedenti.

Questo capitolo tratta del secondo tipo.

Il codice per questo capitolo si trova in `chap04.ipynb`, nel repository di questo libro (vedere la Sezione 0.2). Lo si può anche visualizzare su <http://tinyurl.com/thinkdsp04>.

4.1 Rumore non correlato

Il modo più semplice per comprendere il rumore è generarlo e il tipo più semplice da generare è il rumore uniforme non correlato (rumore UU). “Uniforme” significa che il segnale contiene valori casuali da una distribuzione uniforme; vale a dire, ogni valore nell’intervallo è ugualmente probabile. “Non correlato” significa che i valori sono indipendenti; cioè, conoscere un valore non fornisce informazioni sugli altri.

Ecco una classe che rappresenta il rumore UU:

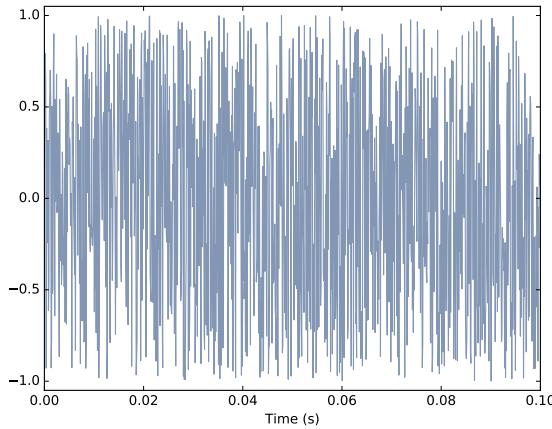


Figura 4.1: Forma d'onda di rumore uniforme non correlato.

```
class UncorrelatedUniformNoise(_Noise):

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

`UncorrelatedUniformNoise` eredita da `_Noise`, che eredita da `Signal`.

Come al solito, la funzione di valutazione prende `ts`, i tempi in cui il segnale dovrebbe essere valutato. Usa `np.random.uniform`, che genera valori da una distribuzione uniforme. In questo esempio, i valori sono compresi nell'intervallo tra `-amp` e `amp`.

L'esempio seguente genera rumore UU con durata 0.5 secondi a 11,025 campioni al secondo.

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

Se si riproduce quest'onda, suona come il fruscio che si sente se si sintonizza la radio tra due canali. La Figura 4.1 mostra come appare la forma d'onda. Come previsto, sembra piuttosto casuale.

Ora diamo un'occhiata allo spettro:

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

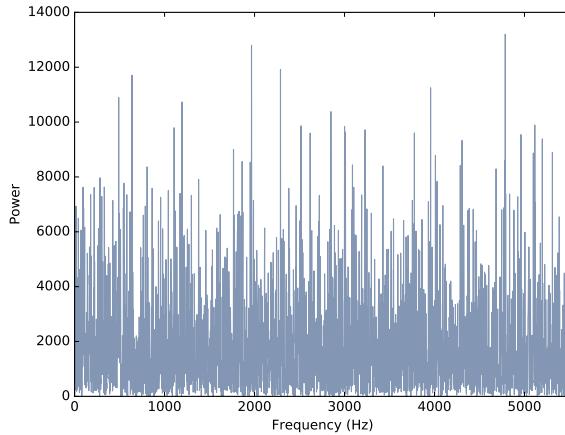


Figura 4.2: Spettro di potenza del rumore uniforme non correlato.

`Spectrum.plot_power` è simile a `Spectrum.plot`, tranne per il fatto che disegna la potenza anziché l'ampiezza. La potenza è il quadrato dell'ampiezza. In questo capitolo si passa dall'ampiezza alla potenza perché è più convenzionale nel contesto del rumore.

La Figura 4.2 mostra il risultato. Come il segnale, lo spettro sembra piuttosto casuale. In effetti, è casuale, ma dobbiamo essere più precisi sulla parola “casuale”. Ci sono almeno tre cose che vorremmo sapere sul segnale di un rumore e sul suo spettro:

- Distribuzione: la distribuzione di un segnale casuale è l'insieme dei possibili valori e delle loro probabilità. Ad esempio, nel segnale di rumore uniforme, l'insieme dei valori è compreso tra -1 e 1 e tutti i valori hanno la stessa probabilità. Un'alternativa è il **rumore Gaussiano**, dove l'insieme di valori è l'intervallo da infinito negativo a quello positivo, ma i valori vicini a 0 sono i più probabili, con una probabilità che diminuisce secondo la curva Gaussiana o curva a “campana”.
- Correlazione: Ogni valore nel segnale è indipendente dagli altri o ci sono dipendenze tra loro? Nel rumore UU, i valori sono indipendenti. Un'alternativa è il **rumore Browniano**, dove ogni valore è la somma del valore precedente e un “passo” casuale. Quindi, se il valore del segnale è alto in un particolare momento, ci si aspetta che rimanga alto, e se è basso, ci aspetta che rimanga basso.
- Relazione tra potenza e frequenza: Nello spettro del rumore UU, la potenza a tutte le frequenze è ricavata dalla stessa distribuzione; ovvero, la potenza media è la stessa per tutte le frequenze. Un'alternativa è il

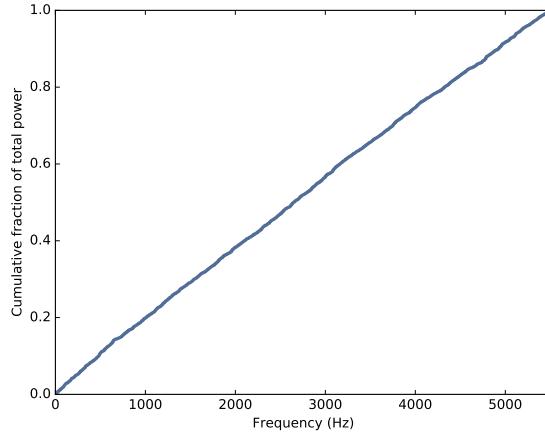


Figura 4.3: Spettro integrato di rumore uniforme non correlato.

rumore rosa, dove la potenza è inversamente proporzionale alla frequenza; cioè, la potenza alla frequenza f è ricavata da una distribuzione la cui media è proporzionale a $1/f$.

4.2 Spettro integrato

Per il rumore UU possiamo vedere la relazione tra potenza e frequenza più chiaramente osservando lo **spettro integrato**, che è una funzione della frequenza, f , e che mostra la potenza cumulativa nello spettro fino a f .

`Spectrum` fornisce un metodo che calcola `IntegratedSpectrum`:

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

`self.power` un array NumPy contenente la potenza per ciascuna frequenza. `np.cumsum` calcola la somma cumulativa delle potenze. La divisione per l'ultimo elemento normalizza lo spettro integrato, quindi va da 0 a 1.

Il risultato è un `IntegratedSpectrum`. Ecco la definizione della classe:

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

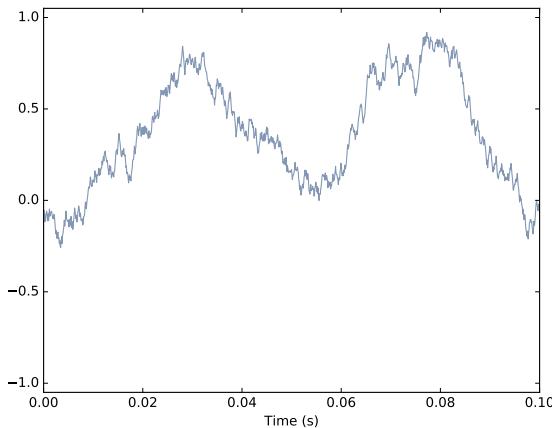


Figura 4.4: Forma d'onda del rumore Browniano.

Come `Spectrum`, `IntegratedSpectrum` fornisce `plot_power`, quindi possiamo calcolare e disegnare lo spettro integrato in questo modo:

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

Il risultato, mostrato nella Figura 4.3, è una linea retta, indicando che la potenza a tutte le frequenze è mediamente costante. Il rumore di uguale potenza a tutte le frequenze è detto **rumore bianco** per analogia con la luce, perché un mix uguale di luce a tutte le frequenze visibili è bianca.

4.3 Rumore Browniano

Il rumore UU non è correlato, il che significa che ogni valore non dipende dagli altri. Un'alternativa è il **rumore Browniano**, in cui ogni valore è la somma del valore precedente e un “passo” casuale.

Si chiama “Browniano” per analogia con il moto browniano, in cui una particella sospesa in un fluido si muove apparentemente in modo casuale, a causa di interazioni invisibili con il fluido. Il moto browniano viene spesso descritto utilizzando una **random walk** [passeggiata aleatoria], che è un modello matematico di un percorso in cui la distanza tra i passaggi è caratterizzata da una distribuzione casuale.

In una passeggiata aleatoria unidimensionale, la particella si muove verso l'alto o verso il basso di una quantità casuale ad ogni passo temporale. La posizione della particella in qualsiasi momento è la somma di tutti i passaggi precedenti.

Questa osservazione suggerisce un modo per generare del rumore Browniano: generare passaggi casuali non correlati e poi sommarli. Ecco una definizione di classe che implementa tale algoritmo:

```
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` usa `np.random.uniform` per generare un segnale non correlato e `np.cumsum` per calcolare la loro somma cumulativa.

Poiché è probabile che la somma sfugga all'intervallo da -1 a 1, dobbiamo usare `unbias` per spostare la media a 0 e `normalize` per ottenere l'ampiezza massima desiderata.

Ecco il codice che genera un oggetto `BrownianNoise` e disegna la forma d'onda.

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

La Figura 4.4 mostra il risultato. La forma d'onda vaga su e giù, ma c'è una chiara correlazione tra i valori successivi. Quando l'ampiezza è alta, tende a rimanere alta e viceversa.

Se si disegna lo spettro del rumore Browniano su una scala lineare, come nella Figura 4.5 (a sinistra), non si vede molto. Quasi tutta la potenza è alle frequenze più basse; le componenti a frequenza più alta non sono visibili.

Per vedere più chiaramente la forma dello spettro, possiamo disegnare la potenza e la frequenza su una scala log-log. Ecco il codice:

```
import matplotlib.pyplot as plt

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

Il risultato è nella Figura 4.5 (a destra). La relazione tra potenza e frequenza è rumorosa, ma approssimativamente lineare.

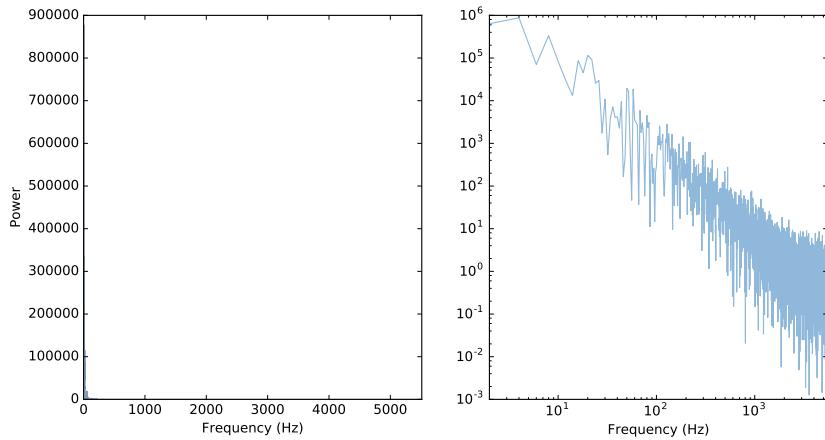


Figura 4.5: Spettro del rumore Browniano su scala lineare (a sinistra) e scala log-log (a destra).

`Spectrum` fornisce `estimate_slope`, che utilizza SciPy per calcolare l'approssimazione dei minimi quadrati allo spettro di potenza:

```
#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
```

Viene scartato il primo componente dello spettro perché corrisponde a $f = 0$, e $\log 0$ non è definito.

`estimate_slope` restituisce il risultato di `scipy.stats.linregress` che è un oggetto che contiene la stima della pendenza e dell'intercetta, il coefficiente di determinazione (R^2), il valore p e l'errore standard. Per i nostri scopi, abbiamo solo bisogno della pendenza.

Per il rumore Browniano, la pendenza dello spettro di potenza è -2 (vedremo perché nel Capitolo 9), quindi possiamo scrivere questa relazione:

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

dove P è la potenza, f è la frequenza, e k è l'intercetta della linea, che non è importante per i nostri scopi. Esponenziando entrambi i lati si ottiene:

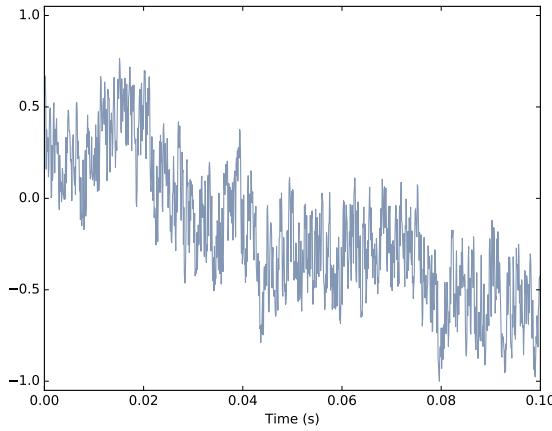


Figura 4.6: Forma d'onda del rumore rosa con $\beta = 1$.

$$P = K/f^2$$

dove K è e^k , ma non è ancora importante. È più rilevante il fatto che la potenza è proporzionale a $1/f^2$, che è la caratteristica del rumore Browniano.

Tale rumore è anche chiamato **rumore rosso**, per lo stesso motivo per cui il rumore bianco si chiama “bianco”. Se si combina la luce visibile con una potenza proporzionale a $1/f^2$, la maggior parte della potenza starebbe all'estremità della bassa frequenza dello spettro, che è rossa. Il rumore Browniano è talvolta detto anche “rumore marrone”, ma pensiamo che sia fonte di confusione, quindi non lo si userà.

4.4 Rumore Rosa

Per il rumore rosso, la relazione tra frequenza e potenza è

$$P = K/f^2$$

Non c’è niente di speciale nell’esponente 2. Più in generale, possiamo sintetizzare il rumore con qualsiasi esponente, β .

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

Quando $\beta = 0$, la potenza è costante a tutte le frequenze, quindi il risultato è rumore bianco. Quando $\beta = 2$ il risultato è il rumore rosso.

Quando β è compreso tra 0 e 2, il risultato è tra il rumore bianco e il rumore rosso, quindi è detto **rumore rosa**.

Esistono diversi modi per generare rumore rosa. Il più semplice è generare rumore bianco e quindi applicare un filtro passa-basso con l'esponente desiderato. `thinkdsp` fornisce una classe che rappresenta un segnale di rumore rosa:

```
class PinkNoise(_Noise):

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

`amp` è l'ampiezza desiderata del segnale. `beta` è l'esponente desiderato. `PinkNoise` fornisce `make_wave`, che genera un'onda.

```
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` è la lunghezza dell'onda in secondi. `start` è il tempo di inizio dell'onda; è incluso in modo che `make_wave` abbia la stessa interfaccia per tutti i tipi di segnale, ma per il rumore casuale, il tempo di inizio è irrilevante. `framerate` è il numero di campioni al secondo.

`make_wave` crea un'onda di rumore bianco, ne calcola lo spettro, applica un filtro con l'esponente desiderato e poi converte lo spettro filtrato in un'onda. Quindi elimina il bias [lo spostamento] e normalizza l'onda.

`Spectrum` fornisce `pink_filter`:

```
def pink_filter(self, beta=1.0):
    denom = self.fs ** (beta/2.0)
```

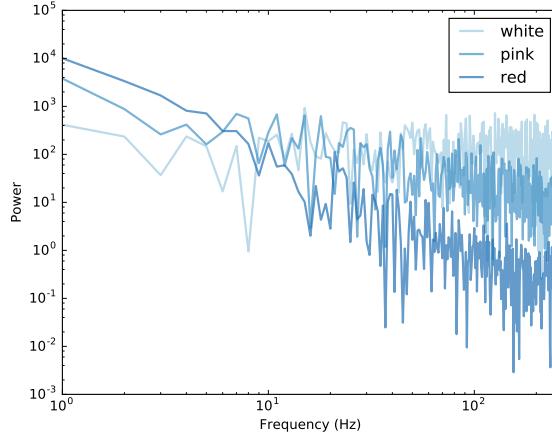


Figura 4.7: Spettro del rumore bianco, rosa e rosso su scala logaritmica (log-log).

```
denom[0] = 1
self.hs /= denom
```

`pink_filter` divide ciascun elemento dello spettro per $f^{\beta/2}$. Poiché la potenza è il quadrato dell’ampiezza, questa operazione divide la potenza su ciascuna componente per f^β . Tratta la componente $f = 0$ come un caso speciale, in parte per evitare di dividere per 0, e in parte perché questo elemento rappresenta la polarizzazione del segnale, che comunque imposteremo a 0.

La Figura 4.6 mostra la forma d’onda risultante. Come il rumore Browniano, vaga su e giù suggerendo l’esistenza di una correlazione tra i valori successivi, ma almeno visivamente sembra più casuale. Nel prossimo capitolo torneremo su questa osservazione e saremo più precisi su cosa si intende per “correlazione” e “più casuale”.

Infine, la Figura 4.7 mostra uno spettro per il rumore bianco, rosa e rosso sulla stessa scala log-log. La relazione tra l’esponeente, β , e la pendenza dello spettro è evidente in questa figura.

4.5 Rumore Gaussiano

Abbiamo iniziato con il rumore uniforme non correlato (UU) e abbiamo dimostrato che, poiché il suo spettro ha la stessa potenza a tutte le frequenze, in media il rumore UU è bianco.

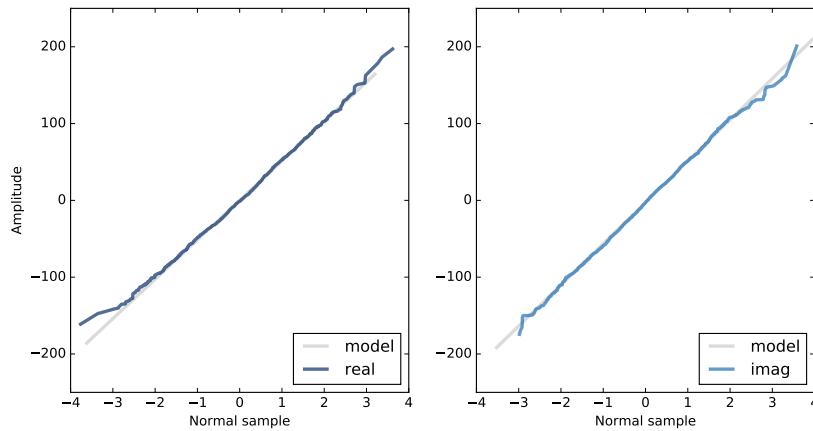


Figura 4.8: Grafico di probabilità normale per le parti reale e immaginaria dello spettro del rumore Gaussiano.

Ma quando le persone parlano di “rumore bianco”, non intendono sempre rumore UU. Infatti, più spesso intendono rumore Gaussiano (UG) non correlato.

`thinkdsp` fornisce un’implementazione del rumore UG:

```
class UncorrelatedGaussianNoise(_Noise):

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

`np.random.normal` restituisce un array NumPy di valori da una distribuzione Gaussiana, in questo caso con media 0 e deviazione standard `self.amp`. In teoria l’intervallo di valori va da infinito negativo a quello positivo, ma ci si aspetta che circa il 99% dei valori sia compreso tra -3 e 3.

Il rumore UG è simile in molti modi al rumore UU. Lo spettro ha la stessa potenza a tutte le frequenze, in media, quindi anche UG è bianco. E ha un’altra proprietà interessante: anche lo spettro del rumore UG è rumore UG. Più precisamente, le parti reale e immaginaria dello spettro sono valori Gaussiani non correlati.

Per testare tale affermazione, possiamo generare lo spettro del rumore UG e poi generare un “diagramma di probabilità normale”, che è un modo grafico per verificare se una distribuzione è Gaussiana.

```
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` è fornito da `thinkstats2`, incluso nel repository di questo libro. Se non si ha familiarità con i grafici delle probabilità normali, ci si può informare leggendo il Capitolo 5 di *Think Stats* su <http://thinkstats2.com>.

La Figura 4.8 mostra i risultati. Le linee grigie mostrano un modello lineare approssimato dei dati; le linee scure mostrano i dati.

Una linea retta su un grafico di probabilità normale indica che i dati provengono da una distribuzione Gaussiana. Ad eccezione di alcune variazioni casuali agli estremi, queste linee sono rette, indicando che lo spettro del rumore UG è rumore UG.

Lo spettro del rumore UU è anche rumore UG, almeno approssimativamente. In effetti, secondo il Teorema del Limite Centrale, lo spettro di quasi tutti i rumori non correlati è approssimativamente Gaussiano, purché la distribuzione abbia una media e una deviazione standard finite e il numero di campioni sia ampio.

4.6 Esercizi

Le soluzioni a questi esercizi sono in `chap04soln.ipynb`.

Esercizio 4.1 “A Soft Murmur” è un sito web che riproduce un mix di sorgenti di rumore naturali, tra cui pioggia, onde, vento, ecc. Su <http://asoftmurmur.com/about/> si trova l’elenco delle loro registrazioni, la maggior parte delle quali si trova su <http://freesound.org>.

Scaricare alcuni di questi file e calcolare lo spettro di ogni segnale. Lo spettro di potenza sembra rumore bianco, rumore rosa o rumore Browniano? Come varia lo spettro nel tempo?

Esercizio 4.2 Nel segnale di un rumore, il mix di frequenze cambia nel tempo. A lungo termine, ci si aspetta che la potenza a tutte le frequenze sia uguale, ma in ogni campione, la potenza a ciascuna frequenza è casuale.

Per stimare la potenza media a lungo termine a ciascuna frequenza, possiamo suddividere un lungo segnale in segmenti, calcolare lo spettro di potenza per

ogni segmento e quindi calcolare la media tra i segmenti. Si possono leggere ulteriori informazioni su questo algoritmo su http://en.wikipedia.org/wiki/Bartlett's_method.

Implementare il metodo di Bartlett e usarlo per stimare lo spettro di potenza dell'onda di un rumore. Suggerimento: guardare l'implementazione di `make_spectrogram`.

Esercizio 4.3 Su <http://www.coindesk.com> si può scaricare il prezzo giornaliero di un BitCoin come file CSV. Leggere questo file e calcolare lo spettro dei prezzi di BitCoin in funzione del tempo. Assomiglia al rumore bianco, rosa o Browniano?

Esercizio 4.4 Un contatore Geiger è un dispositivo che rileva le radiazioni. Quando una particella ionizzante colpisce il rilevatore, emette una scarica di corrente. L'output totale in un punto nel tempo può essere modellato come rumore di Poisson non correlato (UP), in cui ogni campione è una quantità casuale di una distribuzione di Poisson, che corrisponde al numero di particelle rilevate durante un intervallo.

Scrivere una classe chiamata `UncorrelatedPoissonNoise` che eredita da `thinkdsp._Noise` e fornisce `evaluate`. Dovrebbe usare `np.random.poisson` per generare valori casuali da una distribuzione di Poisson. Il parametro di questa funzione, `lam`, è il numero medio di particelle durante ogni intervallo. È possibile utilizzare l'attributo `amp` per specificare `lam`. Ad esempio, se il framerate è 10 kHz e `amp` è 0.001, ci si aspettano circa 10 “click” al secondo.

Generare circa un secondo di rumore UP e ascoltalo. Per valori bassi di `amp`, come 0.001, dovrebbe suonare come un contatore Geiger. Per valori più alti dovrebbe suonare come un rumore bianco. Calcolare e disegnare lo spettro di potenza per vedere se sembra rumore bianco.

Esercizio 4.5 L'algoritmo in questo capitolo per la generazione del rumore rosa è concettualmente semplice ma computazionalmente costoso. Esistono alternative più efficienti, come l'algoritmo di Voss-McCartney. Ricercare questo metodo, implementarlo, calcolarne lo spettro e confermare che ha la relazione desiderata tra potenza e frequenza.

Capitolo 5

Autocorrelazione

Nel capitolo precedente è stato caratterizzato il rumore bianco come “non correlato”, il che significa che ogni valore è indipendente dagli altri, e il rumore Browniano come “correlato”, perché ogni valore dipende dal valore precedente. In questo capitolo si definiscono questi termini in modo più preciso e viene presentata la **funzione di autocorrelazione**, che è uno strumento utile per l’analisi del segnale.

Il codice per questo capitolo si trova in `chap05.ipynb`, nel repository di questo libro (vedere la Sezione 0.2). Lo si può anche visualizzare su <http://tinyurl.com/thinkdsp05>.

5.1 Correlazione

In generale, la correlazione tra le variabili significa che se si conosce il valore di una, si hanno alcune informazioni sull’altra. Esistono diversi modi per quantificare la correlazione, ma il più comune è il coefficiente di correlazione momento-prodotto di Pearson, solitamente indicato con ρ . Per due variabili, x e y , ciascuna contenente N valori:

$$\rho = \frac{\sum_i (x_i - \mu_x)(y_i - \mu_y)}{N\sigma_x\sigma_y}$$

Dove μ_x e μ_y sono le medie di x e y , e σ_x e σ_y sono le loro deviazioni standard.

La correlazione di Pearson è sempre compresa tra -1 e +1 (estremi inclusi). Se ρ è positivo, diciamo che la correlazione è positiva, il che significa che quando

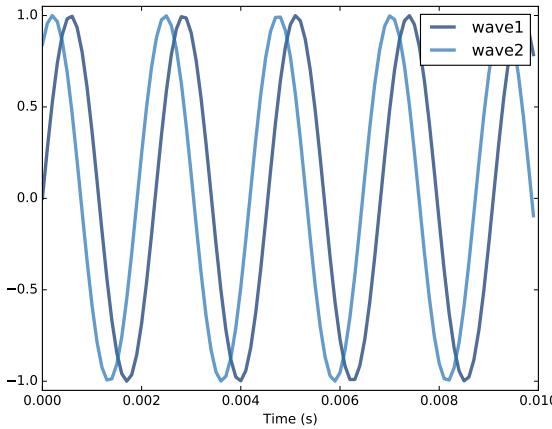


Figura 5.1: Due onde sinusoidali che differiscono per un offset di fase di 1 radiante; il loro coefficiente di correlazione è 0.54.

una variabile è alta, l'altra tende ad essere alta. Se ρ è negativo, la correlazione è negativa, quindi quando una variabile è alta, l'altra tende ad essere bassa.

La grandezza di ρ indica la forza della correlazione. Se ρ è 1 o -1, le variabili sono perfettamente correlate, il che significa che se si conosce una, si può fare una previsione perfetta sull'altra. Se ρ è prossimo allo zero, la correlazione è probabilmente debole, quindi il conoscere uno, non dice molto sugli altri.

Diciamo “probabilmente debole” perché è anche possibile che ci sia una relazione non lineare che non viene catturata dal coefficiente di correlazione. Le relazioni non lineari sono spesso importanti nelle statistiche, ma raramente rilevanti per l'elaborazione del segnale, quindi non se ne parlerà più qui.

Python fornisce diversi modi per calcolare le correlazioni. `np.corrcoef` accetta un numero qualsiasi di variabili e calcola una **matrice di correlazione** che comprende le correlazioni tra ciascuna coppia di variabili.

Verrà presentato un esempio con solo due variabili. Innanzitutto, si definisce una funzione che costruisce onde sinusoidali con diversi offset di fase:

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

Poi si instanziano due onde con offset diversi:

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

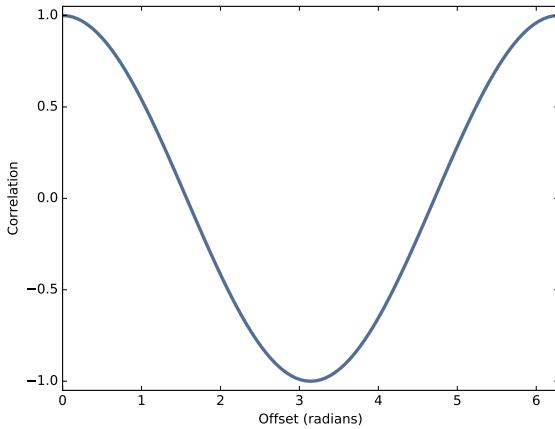


Figura 5.2: La correlazione di due onde sinusoidali in funzione dell'offset di fase tra di loro. Il risultato è un coseno.

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

La Figura 5.1 mostra l'aspetto dei primi periodi di queste onde. Quando un'onda è alta, l'altra è solitamente alta, quindi ci si aspetta che siano correlate.

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

L'opzione `ddof=0` indica che `corrcoef` deve dividere per N , come nell'equazione precedente, anziché utilizzare il valore di default, $N - 1$.

Il risultato è una matrice di correlazione: il primo elemento è la correlazione di `wave1` con se stesso, che è sempre 1. Allo stesso modo, l'ultimo elemento è la correlazione di `wave2` con se stesso.

Gli elementi esterni alla diagonale contengono il valore che ci interessa, la correlazione di `wave1` e `wave2`. Il valore 0.54 indica che la forza della correlazione è moderata.

All'aumentare dell'offset della fase, questa correlazione diminuisce fino a quando le onde non sono sfasate di 180 gradi, il che produce una correlazione -1. Poi aumenta fino a quando l'offset differisce di 360 gradi. A quel punto abbiamo chiuso il cerchio e la correlazione è 1.

La Figura 5.2 mostra la relazione tra correlazione e offset di fase per un'onda sinusoidale. La forma di quella curva dovrebbe apparire familiare; è un coseno.

thinkdsp fornisce una semplice interfaccia per calcolare la correlazione tra le onde:

```
>>> wave1.corr(wave2)
0.54
```

5.2 Correlazione seriale

I segnali rappresentano spesso misure di quantità che variano nel tempo. Ad esempio, i segnali sonori con cui abbiamo lavorato rappresentano misure di tensione (o corrente), che corrispondono alle variazioni della pressione dell'aria che percepiamo come suono.

Misure come questa hanno quasi sempre una correlazione seriale, che è la correlazione tra ogni elemento e il successivo (o il precedente). Per calcolare la correlazione seriale, possiamo spostare un segnale e quindi calcolare la correlazione della versione spostata con l'originale.

```
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` accetta un oggetto Wave e `lag`, che è il numero intero di posizioni in cui spostare l'onda. Calcola la correlazione dell'onda con una versione spostata di se stessa.

Possiamo testare questa funzione con i segnali del rumore del capitolo precedente. Ci aspettiamo che il rumore UU non sia correlato, in base al modo in cui viene generato (per non parlare del nome):

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

Quando è stato eseguito questo esempio, si è ottenuto 0,006, che indica una correlazione seriale molto piccola. Si potrebbe ottenere un valore diverso quando si rieseguirà, ma dovrebbe restare relativamente piccolo.

Nel segnale di un rumore Browniano, ogni valore è la somma del valore precedente e un “passo” casuale, quindi ci si aspetta una forte correlazione seriale:

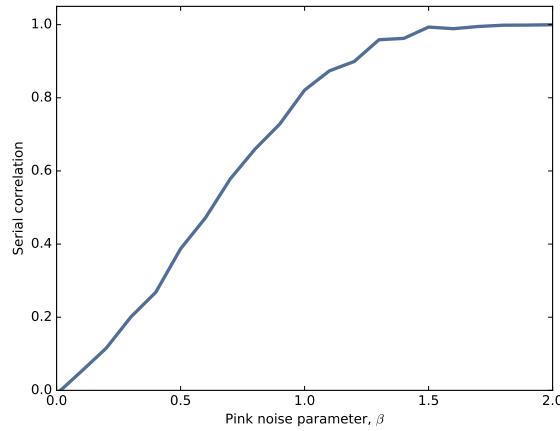


Figura 5.3: Correlazione seriale per il rumore rosa con una gamma di parametri.

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

Abbastanza sicuramente, il risultato ottenuto è maggiore di 0.999.

Poiché il rumore rosa sta in un certo senso tra il rumore Browniano e il rumore UU, potremmo aspettarci una correlazione intermedia:

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

Con il parametro $\beta = 1$, si è ottenuta una correlazione seriale di 0.851. Poiché si varia il parametro da $\beta = 0$, che è rumore non correlato, a $\beta = 2$, che è Browniano, la correlazione seriale varia da 0 a quasi 1, come mostrato nella Figura 5.3.

5.3 Autocorrelazione

Nella sezione precedente abbiamo calcolato la correlazione tra ogni valore e il successivo, quindi abbiamo spostato di 1 gli elementi dell'array. Ma possiamo facilmente calcolare correlazioni seriali con ritardi diversi.

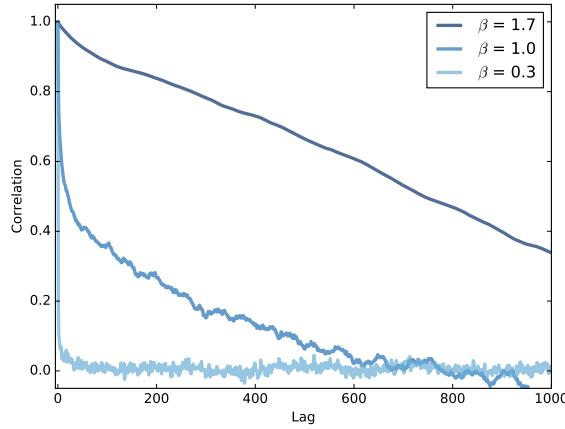


Figura 5.4: Funzioni di autocorrelazione per il rumore rosa con una gamma di parametri.

Si può pensare a `serial_corr` come una funzione che mappa a ciascun valore del ritardo `[lag]` la correlazione corrispondente e possiamo valutare quella funzione scorrendo i valori di `lag`:

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

`autocorr` prende un oggetto `Wave` e restituisce la funzione di autocorrelazione come una coppia di sequenze: `lags` è una sequenza di numeri interi da 0 a metà della lunghezza dell'onda; `corrs` è la sequenza di correlazioni seriali per ogni ritardo.

La Figura 5.4 mostra le funzioni di autocorrelazione per il rumore rosa con tre valori di β . Per valori bassi di β , il segnale è meno correlato e la funzione di autocorrelazione scende rapidamente a zero. Per valori maggiori, la correlazione seriale è più forte e diminuisce più lentamente. Con $\beta = 1.7$ la correlazione seriale è forte anche per lunghi ritardi; questo fenomeno è detto **dipendenza a lungo raggio**, perché indica che ogni valore nel segnale dipende da molti valori precedenti.

5.4 Autocorrelazione dei segnali periodici

L'autocorrelazione del rumore rosa ha proprietà matematiche interessanti, ma applicazioni limitate. L'autocorrelazione dei segnali periodici è più utile.

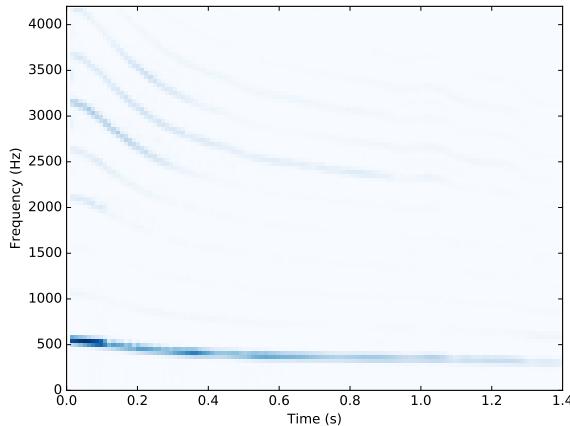


Figura 5.5: Spettrogramma di un chirp vocale.

Ad esempio, è stato scaricato da freesound.org una registrazione di qualcuno che canta un chirp; il repository di questo libro include il file: `28042_bcjordan__voicedownbew.wav`. Si può usare il notebook di Jupyter per questo capitolo, `chap05.ipynb`, per riprodurlo.

La Figura 5.5 mostra lo spettrogramma di quest'onda. La frequenza fondamentale e alcune delle armoniche vengono visualizzate chiaramente. Il chirp inizia vicino a 500 Hz e scende a circa 300 Hz, all'incirca da C5 a E4.

Per stimare il tono in un determinato momento, potremmo usare lo spettro, ma non funziona molto bene. Per vedere perché no, prenderemo un breve segmento dall'onda e se ne disegnerà lo spettro:

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

Questo segmento inizia a 0.2 secondi e dura 0.01 secondi. La Figura 5.6 ne mostra lo spettro. C'è un picco netto vicino a 400 Hz, ma è difficile identificare con precisione l'altezza. La lunghezza del segmento è di 441 campioni con un framerate di 44100 Hz, quindi la risoluzione in frequenza è di 100 Hz (vedere la Sezione 3.5). Ciò significa che il tono stimato potrebbe essere sfasato di 50 Hz; in termini musicali, la gamma da 350 Hz a 450 Hz è di circa 5 semitonni, che è una grande differenza!

Potremmo ottenere una risoluzione migliore in frequenza prendendo un segmento più lungo, ma poiché il tono cambia nel tempo, otterremmo anche

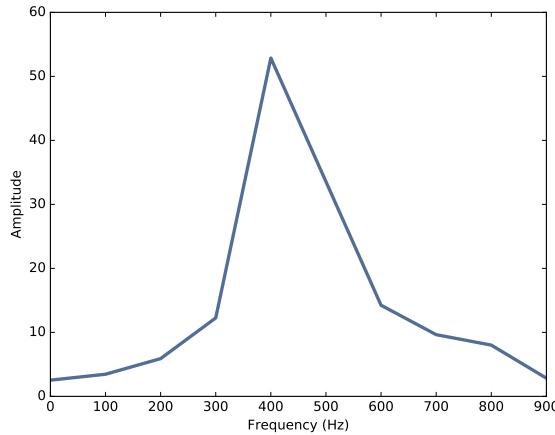


Figura 5.6: Spettro di un segmento di un chirp vocale.

del “motion blur” (effetto mosso in fotografia); cioè, il picco si diffonderebbe tra laltezza iniziale e quella finale del segmento, come abbiamo visto nella Sezione 3.3.

Possiamo stimare laltezza in modo più preciso usando l'autocorrelazione. Se un segnale è periodico, ci aspettiamo che l'autocorrelazione aumenti quando il ritardo è uguale al periodo.

Per mostrare perché funziona, disegneremo due segmenti della stessa registrazione.

```
import matplotlib.pyplot as plt

def plot_shifted(wave, offset=0.001, start=0.2):
    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           //Errorissimo: rimuovere i backslash
    plt.text(segment1.start+0.0005, -0.8, text)
    plt.xlabel('Time (s)')
```

Un segmento inizia a 0.2 secondi; laltro, 0.0023 secondi dopo. La Figura 5.7 mostra il risultato. I segmenti sono simili e la loro correlazione è 0.99. Questo

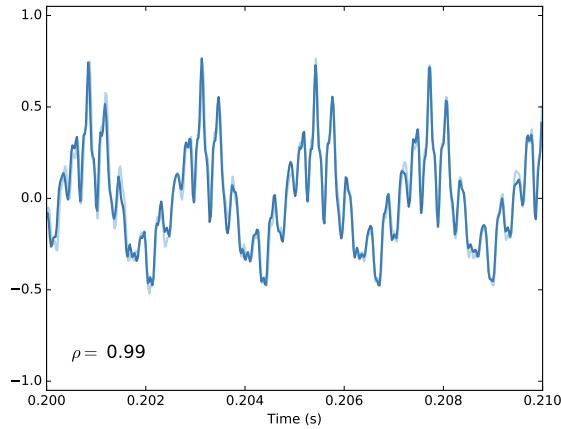


Figura 5.7: Due segmenti di un chirp, uno che inizia 0.0023 secondi dopo l’altro.

risultato suggerisce che il periodo è vicino a 0.0023 secondi, che corrisponde a una frequenza di 435 Hz.

Per questo esempio, il periodo è stato stimato per tentativi. Per automatizzare il processo, possiamo utilizzare la funzione di autocorrelazione.

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

La Figura 5.8 mostra la funzione di autocorrelazione per il segmento che inizia da $t = 0.2$ secondi. Il primo picco si verifica a `lag=101`. Possiamo calcolare la frequenza che corrisponde a quel periodo in questo modo:

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

La frequenza fondamentale stimata è 437 Hz. Per valutare la precisione della stima, possiamo eseguire lo stesso calcolo con i ritardi 100 e 102, che corrispondono alle frequenze 432 e 441 Hz. La precisione della frequenza utilizzando l’autocorrelazione è inferiore a 10 Hz, rispetto ai 100 Hz utilizzando lo spettro. In termini musicali, l’errore atteso è di circa 30 centesimi (un terzo di semitono).

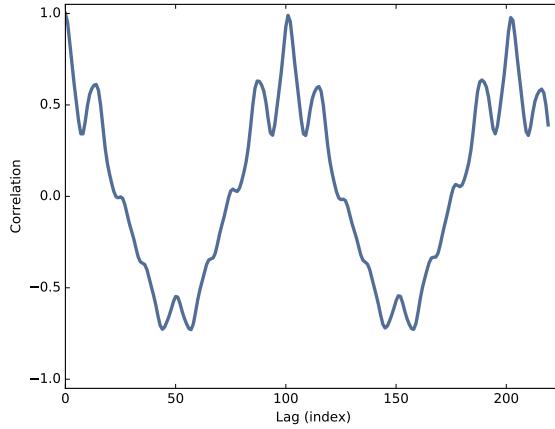


Figura 5.8: Funzione di autocorrelazione per un segmento di un chirp.

5.5 Correlazione come prodotto scalare

Il capitolo è iniziato con questa definizione del coefficiente di correlazione di Pearson:

$$\rho = \frac{\sum_i (x_i - \mu_x)(y_i - \mu_y)}{N\sigma_x\sigma_y}$$

Poi si è usato ρ per definire la correlazione seriale e l'autocorrelazione. Ciò è coerente con il modo in cui questi termini vengono utilizzati nelle statistiche, ma nel contesto dell'elaborazione del segnale, le definizioni sono leggermente diverse.

Nell'elaborazione dei segnali, si lavora spesso con segnali non spostati [unbiased], dove la media è 0, e con segnali normalizzati, dove la deviazione standard è 1. In tal caso, la definizione di ρ si semplifica in:

$$\rho = \frac{1}{N} \sum_i x_i y_i$$

Ed è normale semplificare ulteriormente:

$$r = \sum_i x_i y_i$$

Questa definizione di correlazione non è “standardizzata”, quindi generalmente non è compresa tra -1 e 1. Ma ha altre proprietà utili.

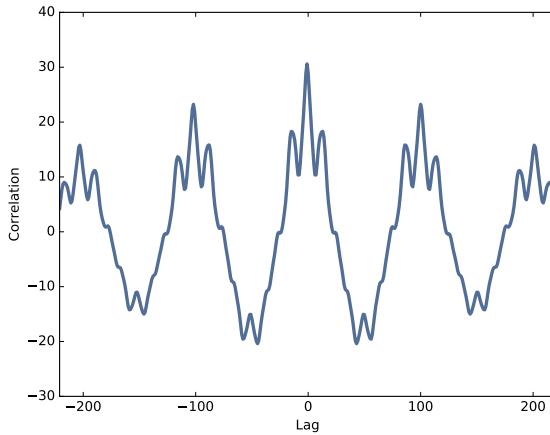


Figura 5.9: Funzione di autocorrelazione calcolata con `np.correlate`.

Se si pensa a x e y come vettori, si riconoscerà questa formula come il **prodotto scalare**, $x \cdot y$. Vedere https://it.wikipedia.org/wiki/Prodotto_scalare.

Il prodotto scalare indica il grado di somiglianza dei segnali. Se sono normalizzati in modo che le loro deviazioni standard siano 1,

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

dove θ è l'angolo tra i vettori. E questo spiega perché la Figura 5.2 è un coseno.

5.6 Utilizzo di NumPy

NumPy fornisce una funzione, `correlate`, che calcola la correlazione di due funzioni o l'autocorrelazione di una funzione. Possiamo usarla per calcolare l'autocorrelazione del segmento dalla sezione precedente:

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

L'opzione `mode` indica a `correlate` l'intervallo di `lag` da usare. Con il valore '`same`', l'intervallo va da $-N/2$ a $N/2$, dove N è la lunghezza della matrice dell'onda.

La Figura 5.9 mostra il risultato. È simmetrico perché i due segnali sono identici, quindi un ritardo negativo su uno ha lo stesso effetto di un ritardo positivo sull'altro. Per confrontarlo con i risultati di `autocorr`, possiamo selezionare la seconda metà:

```
N = len(corsrs2)
half = corsrs2[N//2:]
```

Se si confronta la Figura 5.9 con la Figura 5.8, si noterà che le correlazioni calcolate da `np.correlate` si riducono all'aumentare dei ritardi. Questo perché `np.correlate` utilizza la definizione di correlazione non standardizzata; all'aumentare del ritardo, la sovrapposizione tra i due segnali si riduce, quindi l'entità delle correlazioni diminuisce.

Possiamo correggerlo dividendo per le lunghezze:

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

Infine, possiamo standardizzare i risultati in modo che la correlazione con `lag=0` sia 1.

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

Con queste correzioni, i risultati calcolati da `autocorr` e `np.correlate` sono quasi gli stessi. Differiscono ancora dell'1-2%. Il motivo non è importante, ma per curiosità: `autocorr` standardizza le correlazioni indipendentemente per ogni ritardo `[lag]`; per `np.correlate`, li abbiamo standardizzati tutti alla fine.

Fatto più importante, ora si sa cos'è l'autocorrelazione, come usarla per stimare il periodo fondamentale di un segnale e due modi per calcolarla.

5.7 Esercizi

Le soluzioni a questi esercizi sono in `chap05soln.ipynb`.

Esercizio 5.1 Il notebook Jupyter per questo capitolo, `chap05.ipynb`, include un'interazione che consente di calcolare le autocorrelazioni per ritardi diversi. Usare questa interazione per stimare l'altezza del chirp vocale per alcuni diversi tempi di inizio.

Esercizio 5.2 Il codice di esempio in `chap05.ipynb` mostra come utilizzare l'autocorrelazione per stimare la frequenza fondamentale di un segnale periodico. Incapsulare questo codice in una funzione chiamata `estimate_fundamental`, e usarlo per disegnare l'altezza di un suono registrato.

Per vedere come funziona, provare a sovrapporre le proprie stime dell'altezza ad uno spettrogramma della registrazione.

Esercizio 5.3 Se sono stati eseguiti gli esercizi nel capitolo precedente, è stato scaricato il prezzo storico dei BitCoin e stimato lo spettro di potenza delle variazioni di prezzo. Utilizzando gli stessi dati, calcolare l'autocorrelazione dei prezzi di BitCoin. La funzione di autocorrelazione si interrompe rapidamente? Ci sono prove di comportamenti periodici?

Esercizio 5.4 Nel repository di questo libro si trova un notebook Jupyter chiamato `saxophone.ipynb` che esplora l'autocorrelazione, la percezione dell'altezza e un fenomeno chiamato **fondamentale mancante**. Leggere questo notebook ed eseguire gli esempi. Provare a selezionare un segmento diverso della registrazione e rieseguire gli esempi.

Vi Hart ha un ottimo video intitolato “What is up with Noises? (The Science and Mathematics of Sound, Frequency, and Pitch)”; mostra il fenomeno della fondamentale mancante e spiega come funziona la percezione dell'altezza (almeno, nella misura in cui sappiamo). Lo si può vedere su https://www.youtube.com/watch?v=i_0DXxNeaQ0.

Capitolo 6

Trasformata discreta del coseno

L'argomento di questo capitolo è la **Trasformata Discreta del Coseno** (DCT), utilizzata in MP3 e formati simili per la compressione della musica, in JPEG e formati simili per le immagini e la famiglia dei formati MPEG per i video.

La DCT è simile in molti modi alla trasformata discreta di Fourier (DFT), utilizzata per l'analisi spettrale. Una volta appreso come funziona la DCT, sarà più facile spiegare la DFT.

Ecco i passaggi per arrivarci:

1. Inizieremo con il problema della sintesi: dato un insieme di componenti di frequenza e le loro ampiezze, come possiamo costruire un'onda?
2. Poi riscriveremo il problema della sintesi usando gli array NumPy. Questo passaggio è utile per le prestazioni e fornisce anche informazioni per il passo successivo.
3. Vedremo il problema dell'analisi: dato un segnale e un insieme di frequenze, come possiamo trovare l'ampiezza di ogni componente di frequenza? Inizieremo con una soluzione concettualmente semplice ma lenta.
4. Infine, utilizzeremo alcuni principi dell'algebra lineare per trovare un algoritmo più efficiente. Se si conosce già l'algebra lineare, è fantastico, ma spiegheremo di cosa c'è bisogno man mano che procediamo.

Il codice per questo capitolo si trova in `chap06.ipynb` nel repository di questo libro (vedere la Sezione 0.2). Lo si può anche visualizzare su <http://tinyurl.com/thinkdsp06>.

6.1 Sintesi

Supponiamo che venga dato un elenco di ampiezze e uno di frequenze e si chieda di costruire un segnale che sia la somma di queste componenti di frequenza. Utilizzando gli oggetti nel modulo `thinkdsp`, c'è un modo semplice per eseguire questa operazione, che si chiama **sintesi**:

```
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
```

`amps` è un elenco di ampiezze, `fs` è l'elenco delle frequenze e `ts` è la sequenza dei tempi in cui il segnale deve essere valutato.

`components` è un elenco di oggetti `CosSignal`, uno per ciascuna coppia ampiezza-frequenza. `SumSignal` rappresenta la somma di queste componenti di frequenza.

Infine, `evaluate` calcola il valore del segnale per ciascun tempo in `ts`.

Possiamo testare questa funzione in questo modo:

```
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, ts, framerate)
```

Questo esempio crea un segnale che contiene una frequenza fondamentale a 100 Hz e tre armoniche (100 Hz è un G2 [SOL2] acuto). Rende il segnale per un secondo a 11,025 frame al secondo e inserisce i risultati in un oggetto `Wave`.

Concettualmente, la sintesi è piuttosto semplice. Ma in questa forma non aiuta molto con l'**analisi**, che è il problema inverso: data l'onda, come si identificano le componenti di frequenza e le loro ampiezze?

$$\begin{array}{c}
 M \\
 \cdot \\
 \cdot \\
 t_n \\
 \cdot \\
 \cdot
 \end{array}
 \left[\begin{array}{cccc} 0.6 \\ 0.25 \\ 0.1 \\ 0.05 \end{array} \right] = \text{amps}$$

$$\left[\begin{array}{cccc} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ a & b & c & d \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{array} \right] \left[\begin{array}{c} \cdot \\ \cdot \\ e \\ \cdot \\ \cdot \end{array} \right] = \text{ys}$$

$$\cdot \quad f_k \quad \cdot \quad \cdot$$

Figura 6.1: Sintesi con gli array.

6.2 Sintesi con gli array

Ecco un altro modo per scrivere `synthesize`:

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

Questa funzione sembra molto diversa, ma fa la stessa cosa. Vediamo come funziona:

1. `np.outer` calcola il prodotto esterno di `ts` e `fs`. Il risultato è un array con una riga per ogni elemento di `ts` e una colonna per ogni elemento di `fs`. Ogni elemento dell'array è il prodotto di una frequenza e di un tempo, ft .
2. Moltiplichiamo `args` per 2π e applichiamo `cos`, quindi ogni elemento del risultato è $\cos(2\pi ft)$. Poiché `ts` scorre lungo le colonne, ogni colonna contiene un segnale coseno a una particolare frequenza, valutato in una sequenza di tempi.
3. `np.dot` moltiplica ogni riga di `M` per `amps`, in termini di elementi, poi somma i prodotti. In termini di algebra lineare, stiamo moltiplicando una matrice, `M`, per un vettore, `amps`. In termini di segnali, stiamo calcolando la somma ponderata delle componenti di frequenza.

La Figura 6.1 mostra la struttura di questo calcolo. Ogni riga della matrice, M , corrisponde a un tempo compreso tra 0.0 a 1.0 secondi; t_n è il tempo della n -esima riga. Ogni colonna corrisponde a una frequenza da 100 a 400 Hz, f_k è la frequenza della k -esima colonna.

L' n -esima riga è etichettata con le lettere dalla a alla d ; ad esempio, il valore di a è $\cos[2\pi(100)t_n]$.

Il risultato del prodotto scalare, `ys`, è un vettore con un elemento per ogni riga di M . L'elemento n -esimo, etichettato e , è la somma dei prodotti:

$$e = 0.6a + 0.25b + 0.1c + 0.05d$$

E allo stesso modo con gli altri elementi di `ys`. Quindi ogni elemento di `ys` è la somma di quattro componenti di frequenza, valutate in un determinato momento e moltiplicate per le ampiezze corrispondenti. Ed è esattamente quello che volevamo.

Possiamo usare il codice della sezione precedente per verificare che le due versioni di `synthesize` producono gli stessi risultati.

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

La più grande differenza tra `ys1` e `ys2` è circa `1e-13`, che è ciò che ci si aspetta a causa degli errori in virgola mobile.

Scrivere questo calcolo in termini di algebra lineare rende il codice più piccolo e più veloce. L'algebra lineare fornisce una notazione concisa per le operazioni sulle matrici e sui vettori. Ad esempio, potremmo scrivere `synthesize` in questo modo:

$$\begin{aligned} M &= \cos(2\pi t \otimes f) \\ y &= Ma \end{aligned}$$

dove a è un vettore di ampiezze, t è un vettore di tempi, f è un vettore di frequenze e \otimes è il simbolo del prodotto esterno di due vettori.

6.3 Analisi

Ora siamo pronti per risolvere il problema dell'analisi. Supponiamo che venga data un'onda e che ci si dica che è la somma dei coseni con un dato insieme di frequenze. Come si cercherebbe l'ampiezza per ogni componente di frequenza? In altre parole, dati `ys`, `ts` e `fs`, si può recuperare `amps`?

In termini di algebra lineare, il primo passo è lo stesso della sintesi: calcoliamo $M = \cos(2\pi t \otimes f)$. Poi vogliamo trovare a in modo che $y = Ma$; in altre parole, vogliamo risolvere un sistema lineare. NumPy fornisce `linalg.solve`, che fa esattamente questo.

Ecco come appare il codice:

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

Le prime due righe utilizzano `ts` e `fs` per costruire la matrice, `M`. Poi `np.linalg.solve` calcola `amps`.

Ma c'è un intoppo. In generale possiamo risolvere un sistema di equazioni lineari solo se la matrice è quadrata; ovvero, il numero di equazioni (righe) è uguale al numero di incognite (colonne).

In questo esempio, abbiamo solo 4 frequenze, ma abbiamo valutato il segnale 11,025 volte. Quindi abbiamo molte più equazioni che incognite.

In generale se `ys` contiene più di 4 elementi, è improbabile che possiamo analizzarlo utilizzando solo 4 frequenze.

Ma in questo caso, sappiamo che gli `ys` sono stati effettivamente generati aggiungendo solo 4 componenti di frequenza, quindi possiamo usare 4 valori qualsiasi dalla matrice d'onda per recuperare `amps`.

Per semplicità, useremo i primi 4 campioni dal segnale. Utilizzando i valori di `ys`, `fs` e `ts` della sezione precedente, possiamo eseguire `analyze1` in questo modo:

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

Ed è abbastanza sicuro, `amps2` è

[0.6 0.25 0.1 0.05]

Questo algoritmo funziona, ma è lento. La risoluzione di un sistema lineare di equazioni richiede un tempo proporzionale a n^3 , dove n è il numero di colonne in M . Possiamo fare di meglio.

6.4 Matrici ortogonali

Un modo per risolvere i sistemi lineari è invertire le matrici. L'inverso di una matrice M si scrive M^{-1} , e ha la proprietà che $M^{-1}M = I$. I è la matrice identità, che ha il valore 1 su tutti gli elementi diagonali e 0 altrove.

Quindi, per risolvere l'equazione $y = Ma$, possiamo moltiplicare entrambi i lati per M^{-1} , che restituisce:

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

Sul lato destro, possiamo sostituire $M^{-1}M$ con I :

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

Se moltiplichiamo I per qualsiasi vettore a , il risultato è a , quindi

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

Ciò implica che se possiamo calcolare M^{-1} in modo efficiente, possiamo trovare a con una semplice moltiplicazione di matrici (usando `np.dot`). Ciò richiede un tempo proporzionale a n^2 , che è migliore di n^3 .

L'inversione di una matrice è lenta, in generale, ma alcuni casi speciali sono più veloci. In particolare, se M è **ortogonale**, l'inverso di M è semplicemente la trasposizione di M , scritto M^T . In NumPy la trasposizione di un array è un'operazione a tempo costante. In realtà non sposta gli elementi dell'array; ma, crea una “vista” che cambia il modo in cui si accede agli elementi.

Inoltre, una matrice è ortogonale se la sua trasposizione è anche la sua inversa; cioè, $M^T = M^{-1}$. Ciò implica che $M^TM = I$, il che significa che possiamo verificare se una matrice è ortogonale calcolando M^TM .

Quindi vediamo come appare la matrice in `synthesize2`. Nell'esempio precedente, M ha 11,025 righe, quindi potrebbe essere una buona idea lavorare con un esempio più piccolo:

```
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` è lo stesso vettore di ampiezze che abbiamo visto prima. Poiché abbiamo 4 componenti di frequenza, campioneremo il segnale in 4 punti nel tempo. In questo modo, M è quadrata.

`ts` è un vettore di tempi di campionamento equidistanti nell'intervallo da 0 a 1 unità di tempo. Si è scelta l'unità di tempo come 1 millisecondo, ma è una scelta arbitraria e vedremo tra un minuto che esce comunque dal calcolo.

Poiché il frame rate è N campioni per unità di tempo, la frequenza di Nyquist è $N / \text{time_unit} / 2$, che è 2000 Hz in questo esempio. Quindi `fs` è un vettore di frequenze equidistanti tra 0 e 2000 Hz.

Con questi valori di `ts` e `fs`, la matrice, M , è:

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

Si riconoscerà 0.707 come un'approssimazione di $\sqrt{2}/2$, che è $\cos \pi/4$. Si noterà anche che questa matrice è **simmetrica**, il che significa che l'elemento in (j, k) è sempre uguale all'elemento in (k, j) . Ciò implica che M è la propria trasposizione; cioè, $M^T = M$.

Ma purtroppo, M non è ortogonale. Se calcoliamo $M^T M$, otteniamo:

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

E questa non è la matrice identità.

6.5 DCT-IV

Ma se sceglieremo con attenzione `ts` e `fs`, possiamo rendere M ortogonale. Esistono diversi modi per farlo, motivo per cui esistono diverse versioni della trasformata discreta del coseno (DCT).

Una semplice opzione consiste nello spostare `ts` e `fs` di mezza unità. Questa versione è chiamata DCT-IV, dove “IV” è un numero romano che indica che questa è la quarta di otto versioni della DCT.

Ecco una versione aggiornata di `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)
```

Se la si confronta con la versione precedente, si noteranno due modifiche. Innanzitutto, si è aggiunto 0.5 a `ts` e a `fs`. Poi, è stato cancellato `time_units`, che semplifica l'espressione per `fs`.

Con questi valori, M è

```
[[ 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]]
```

E $M^T M$ è

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

Alcuni degli elementi esterni alla diagonale vengono visualizzati come -0, il che significa che la rappresentazione in virgola mobile è un piccolo numero negativo. Questa matrice è molto vicina a $2I$, il che significa che M è quasi ortogonale; è appena fuori di un fattore 2. E per i nostri scopi, va abbastanza bene.

Poiché M è simmetrica e (quasi) ortogonale, l'inverso di M è solo $M/2$. Ora possiamo scrivere una versione più efficiente di `analyze`:

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

Invece di usare `np.linalg.solve`, moltiplichiamo semplicemente per $M/2$.

Combinando `test2` e `analyze2`, possiamo scrivere un'implementazione della DCT-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
```

Di nuovo, `ys` è l'array di onde. Non dobbiamo passare `ts` e `fs` come parametri; `dct_iv` può calcolarli in base a `N`, la lunghezza di `ys`.

Se abbiamo capito bene, questa funzione dovrebbe risolvere il problema dell'analisi; cioè, dato `ys` dovrebbe essere in grado di recuperare `amps`. Possiamo testarlo in questo modo:

```
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))
```

A partire da `amps`, sintetizziamo un array di onde, quindi utilizziamo `dct_iv` per calcolare `amps2`. La più grande differenza tra `amps` e `amps2` è circa `1e-16`, che è ciò che ci si aspetta a causa di errori in virgola mobile.

6.6 DCT inversa

Infine, notare che `analyze2` e `synthesize2` sono quasi identiche. L'unica differenza è che `analyze2` divide il risultato per 2. Possiamo usare questa

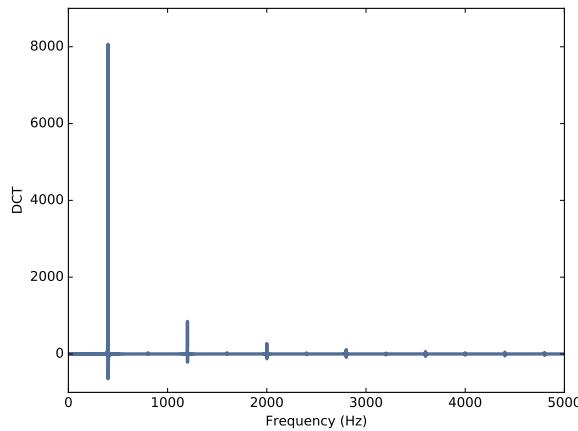


Figura 6.2: DCT di un segnale triangolare a 400 Hz, campionato a 10 kHz.

intuizione per calcolare la DCT inversa:

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

`inverse_dct_iv` risolve il problema della sintesi: prende il vettore delle ampiezze e restituisce la matrice dell'onda, `ys`. Possiamo testarlo iniziando con `amps`, applicando `inverse_dct_iv` e `dct_iv`, provando a recuperare quello con cui abbiamo iniziato.

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

Ancora una volta, la differenza più grande è all'incirca `1e-16`.

6.7 La classe Dct

`thinkdsp` fornisce una classe `Dct` che incapsula la DCT nello stesso modo in cui la classe `Spectrum` incapsula la FFT. Per creare un oggetto `Dct`, si può invocare `make_dct` su una `Wave`.

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

Il risultato è la DCT di un'onda triangolare a 400 Hz, mostrato nella Figura 6.2. I valori della DCT possono essere positivi o negativi; un valore negativo nella DCT corrisponde a un coseno negato o, equivalentemente, a un coseno spostato di 180 gradi.

`make_dct` utilizza la DCT-II, che è il tipo più comune di DCT, fornita da `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)
```

I risultati di `dct` vengono memorizzati in `hs`. Le frequenze corrispondenti, calcolate come nella Sezione 6.5, sono memorizzate in `fs`. Quindi vengono utilizzati entrambi per inizializzare l'oggetto `Dct`.

`Dct` fornisce `make_wave`, che esegue la DCT inversa. Possiamo testarlo in questo modo:

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

La più grande differenza tra `ys1` e `ys2` è circa `1e-16`, che è ciò che ci aspettiamo a causa degli errori in virgola mobile.

`make_wave` usa `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)
```

Per default, l'inversa della DCT non normalizza il risultato, quindi dobbiamo dividerlo per $2N$.

6.8 Esercizi

Per i seguenti esercizi, si fornisce del codice iniziale in `chap06starter.ipynb`. Le soluzioni sono in `chap06soln.ipynb`.

Esercizio 6.1 In questo capitolo si sostiene che `analyze1` impiega un tempo proporzionale a n^3 e `analyze2` impiega un tempo proporzionale a n^2 . Per vedere se è vero, eseguirli su una gamma di dimensioni di input e cronometrarli. In Jupyter, si può usare il “comando magico” `\%timeit`.

Se si disegna il tempo di esecuzione rispetto alle dimensioni di input su una scala log-log, si dovrebbe ottenere una linea retta con pendenza 3 per `analyze1` e una pendenza 2 per `analyze2`.

Si potrebbe testare anche `dct_iv` e `scipy.fftpack.dct`.

Esercizio 6.2 Una delle principali applicazioni della DCT è la compressione sia per il suono che per le immagini. Nella sua forma più semplice, la compressione basata sulla DCT funziona in questo modo:

1. Spezzare un lungo segnale in segmenti.
2. Calcolare la DCT di ogni segmento.
3. Identificare le componenti di frequenza con ampiezze così basse da non essere udibili e rimuoverle. Memorizzare solo le frequenze e le ampiezze rimanenti.
4. Per riprodurre il segnale, caricare le frequenze e le ampiezze per ogni segmento e applicare la DCT inversa.

Implementare una versione di questo algoritmo e applicarla a una registrazione di musica o di parlato. Quante componenti si possono eliminare prima che la differenza sia percettibile?

Per rendere pratico questo metodo, è necessario un modo per memorizzare un array sparso; ovvero, un array in cui la maggior parte degli elementi è zero. NumPy fornisce diverse implementazioni di array sparsi, di cui si può leggere su <http://docs.scipy.org/doc/scipy/reference/sparse.html>.

Esercizio 6.3 Nel repository di questo libro si troverà un notebook Jupyter chiamato `phase.ipynb` che esplora l’effetto della fase sulla percezione del suono. Leggere questo notebook ed eseguire gli esempi. Scegliere un altro segmento di suono ed eseguire gli stessi esperimenti. Si riesce a trovare una relazione generale tra la struttura della fase di un suono e il modo in cui lo percepiamo?

Capitolo 7

Trasformata Discreta di Fourier

Abbiamo utilizzato la trasformata discreta di Fourier (DFT) dal Capitolo 1, ma non abbiamo spiegato come funziona. Ora è il momento.

Se si comprende la trasformata discreta del coseno (DCT), si capirà la DFT. L'unica differenza è che invece di usare la funzione coseno, useremo la funzione esponenziale complessa. Inizieremo spiegando gli esponenziali complessi, poi seguiremo la stessa progressione del Capitolo 6:

1. Inizieremo con il problema della sintesi: dato un insieme di componenti di frequenza e le loro ampiezze, come possiamo costruire un segnale? Il problema della sintesi è equivalente alla DFT inversa.
2. Poi riscriveremo il problema della sintesi sotto forma di moltiplicazione di matrici usando gli array NumPy.
3. Successivamente risolveremo il problema dell'analisi, che equivale alla DFT: dato un segnale, come trovare l'ampiezza e l'offset della fase delle sue componenti in frequenza?
4. Infine, utilizzeremo l'algebra lineare per trovare un modo più efficiente per calcolare la DFT.

Il codice per questo capitolo è in `chap07.ipynb`, nel repository di questo libro (vedere la Sezione 0.2). Lo si può anche visualizzare su <http://tinyurl.com/thinkdsp07>.

7.1 Esponenziali complessi

Una delle mosse più interessanti in matematica è la generalizzazione di un'operazione da un tipo a un altro. Ad esempio, il fattoriale è una funzione che opera su numeri interi; la definizione naturale del fattoriale di n è il prodotto di tutti i numeri interi da 1 a n .

Se si possiede una certa inclinazione, ci si chiederà come calcolare il fattoriale di un non intero come 3.5. Poiché la definizione naturale non si applica, si potrebbero cercare altri modi per calcolare la funzione fattoriale, modi che funzionino con numeri non interi.

Nel 1730, Leonhard Euler ne trovò una, una generalizzazione della funzione fattoriale che conosciamo come funzione gamma (vedere https://it.wikipedia.org/wiki/Funzione_Gamma).

Eulero trovò anche una delle generalizzazioni più utili nella matematica applicata, la funzione esponenziale complessa.

La definizione naturale di esponenziale è una moltiplicazione ripetuta. Per esempio, $\phi^3 = \phi \cdot \phi \cdot \phi$. Ma questa definizione non si applica agli esponenti non interi.

Tuttavia, l'elevamento a potenza può anche essere espresso come una serie di potenze:

$$e^\phi = 1 + \phi + \phi^2/2! + \phi^3/3! + \dots$$

Questa definizione funziona con i numeri reali, quelli immaginari e, per semplice estensione, con i numeri complessi. Applicando questa definizione a un numero immaginario puro, $i\phi$, otteniamo:

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

Riorganizzando i termini, possiamo dimostrare che questo è equivalente a:

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

Si può vedere la derivazione su http://en.wikipedia.org/wiki/Euler's_formula.

Questa formula implica che $e^{i\phi}$ è un numero complesso di magnitudine 1; se lo si pensa come un punto nel piano complesso, è sempre sul cerchio unitario.

E se lo si pensa come un vettore, l'angolo in radianti tra il vettore e l'asse x positivo è l'argomento, ϕ .

Nel caso in cui l'esponente sia un numero complesso, abbiamo:

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

dove A è un numero reale che indica l'ampiezza e $e^{i\phi}$ è un numero complesso unitario che indica l'angolo.

NumPy fornisce una versione di `exp` che funziona con i numeri complessi:

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

Python usa `j` per rappresentare l'unità immaginaria, anziché `i`. Un numero che termina con `j` è considerato immaginario, quindi `1j` è solo i .

Quando l'argomento di `np.exp` è immaginario o complesso, il risultato è un numero complesso; in particolare, un `np.complex128`, è rappresentato da due numeri a virgola mobile a 64 bit. In questo esempio, il risultato è `0.0707+0.997j`.

I numeri complessi hanno gli attributi `real` e `imag`:

```
>>> z.real
0.0707
>>> z.imag
0.997
```

Per ottenere la grandezza [magnitudine], si può usare la funzione nativa `abs` o `np.absolute`:

```
>>> abs(z)
1.0
>>> np.absolute(z)
1.0
```

Per ottenere l'angolo, si può utilizzare `np.angle`:

```
>>> np.angle(z)
1.5
```

Questo esempio conferma che $e^{i\phi}$ è un numero complesso con magnitudine 1 e angolo ϕ radianti.

7.2 Segnali complessi

Se $\phi(t)$ è una funzione del tempo, anche $e^{i\phi(t)}$ è una funzione del tempo. In particolare,

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

Questa funzione descrive una quantità che varia nel tempo, quindi è un segnale. In particolare, è un **segnale esponenziale complesso**.

Nel caso speciale in cui la frequenza del segnale sia costante, $\phi(t)$ è $2\pi ft$ e il risultato è una **sinusoide complessa**:

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

O, più in generale, il segnale potrebbe iniziare con un offset di fase ϕ_0 , ottenendo:

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

`thinkdsp` fornisce un'implementazione di questo segnale, `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` eredita `__init__` da `Sinusoid`. Fornisce una versione di `evaluate` che è quasi identica a `Sinusoid.evaluate`; l'unica differenza è che utilizza `np.exp` invece di `np.sin`.

Il risultato è un array NumPy di numeri complessi:

```
>>> 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]
```

La frequenza di questo segnale è di 1 ciclo al secondo, l'ampiezza è 0.6 (in unità non specificate) e l'offset di fase è 1 radiante.

Questo esempio valuta il segnale in 4 punti equidistanti tra 0 e 1 secondo. I campioni risultanti sono numeri complessi.

7.3 Il problema della sintesi

Proprio come abbiamo fatto con le sinusoidi reali, possiamo creare segnali composti sommando sinusoidi complesse con frequenze diverse. E questo ci porta alla versione complessa del problema della sintesi: data la frequenza e l'ampiezza di ogni componente complesso, come valutiamo il segnale?

La soluzione più semplice è creare oggetti `ComplexSinusoid` e sommarli.

```
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
```

Questa funzione è quasi identica a `synthesize1` nella Sezione 6.1; l'unica differenza è che è stato sostituito `CosSignal` con `ComplexSinusoid`.

Ecco un esempio:

```
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)
```

Il risultato è:

```
[ 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]
```

Al livello più basso, un segnale complesso è una sequenza di numeri complessi. Ma come interpretarlo? Abbiamo qualche intuizione per segnali reali:

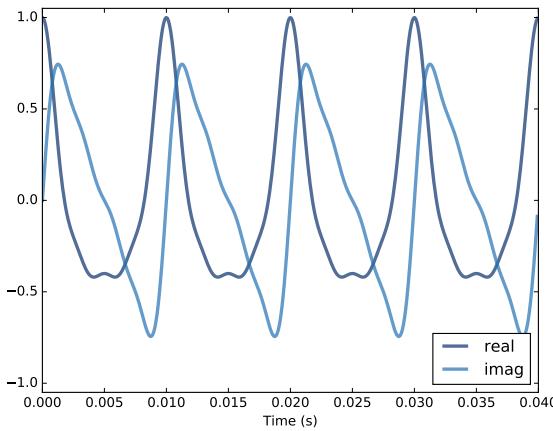


Figura 7.1: Parti reali e immaginarie di un mix di sinusoidi complesse.

rappresentano quantità che variano nel tempo; ad esempio, un segnale acustico rappresenta i cambiamenti nella pressione dell'aria. Ma nulla di ciò che misuriamo nel mondo reale produce numeri complessi.

Allora cos'è un segnale complesso? Non abbiamo una risposta soddisfacente a questa domanda. Il meglio che si possa offrire sono due risposte insoddisfacenti:

1. Un segnale complesso è un'astrazione matematica utile per il calcolo e l'analisi, ma non corrisponde direttamente a nulla nel mondo reale.
2. Se lo si desidera, si può pensare a un segnale complesso come una sequenza di numeri complessi contenente due segnali corrispondenti alle sue parti reali e immaginarie.

Prendendo il secondo punto di vista, possiamo suddividere il segnale precedente nelle sue parti reali e immaginarie:

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

La Figura 7.1 mostra un segmento del risultato. La parte reale è una somma di coseni; la parte immaginaria è una somma di seni. Sebbene le forme d'onda abbiano un aspetto diverso, contengono le stesse componenti di frequenza nelle stesse proporzioni. Alle nostre orecchie, suonano allo stesso modo (in generale, non sentiamo l'offset della fase).

7.4 Sintesi con le matrici

Come abbiamo visto nella Sezione 6.2, possiamo anche esprimere il problema della sintesi in termini di moltiplicazione di matrici:

```
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
```

Di nuovo, `amps` è un array NumPy che contiene una sequenza di ampiezze.

`fs` è una sequenza contenente le frequenze delle componenti. `ts` contiene i tempi in cui valuteremo il segnale.

`args` contiene il prodotto esterno di `ts` e `fs`, con la `ts` che scorre lungo le righe e `fs` che scorre tra le colonne (Si potrebbe far riferimento alla Figura 6.1).

Ogni colonna della matrice `M` contiene una sinusode complessa con una frequenza particolare, valutata in una sequenza di `ts`.

Quando moltiplichiamo `M` per le ampiezze, il risultato è un vettore i cui elementi corrispondono a `ts`; ogni elemento è la somma di più sinusoidi complesse, valutate in un determinato momento.

Ecco di nuovo l'esempio della sezione precedente:

```
>>> 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]
```

Il risultato è lo stesso.

In questo esempio le ampiezze sono reali, ma potrebbero anche essere complesse. Che effetto ha un'ampiezza complessa sul risultato? Ricordare che possiamo pensare a un numero complesso in due modi: o la somma di una parte reale e una immaginaria, $x + iy$, o come il prodotto di un'ampiezza reale e un esponenziale complesso, $Ae^{i\phi_0}$. Usando la seconda interpretazione, possiamo vedere cosa succede quando moltiplichiamo un'ampiezza complessa per una sinusode complessa. Per ogni frequenza, f , abbiamo:

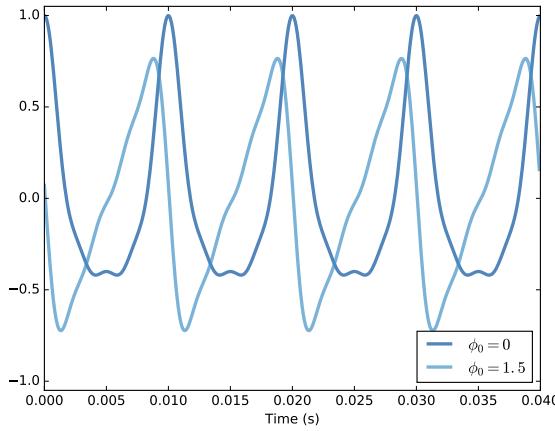


Figura 7.2: Parte reale di due segnali complessi che differiscono per un offset di fase.

$$A \exp i\phi_0 \cdot \exp i2\pi ft = A \exp i(2\pi ft + \phi_0)$$

Moltiplicando per $Ae^{i\phi_0}$ si moltiplica l'ampiezza per A e si somma l'offset di fase ϕ_0 .

Possiamo testare tale affermazione eseguendo l'esempio precedente con $\phi_0 = 1.5$ per tutte le componenti di frequenza:

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

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

Poiché `amps` è un array di reali, moltiplicando per `np.exp(1j * phi)` si ottiene un array di numeri complessi con sfasamento di fase di `phi` radianti e le stesse magnitudini di `amps`.

La Figura 7.2 mostra il risultato. L'offset di fase $\phi_0 = 1.5$ sposta l'onda a sinistra di circa un quarto di ciclo; cambia anche la forma d'onda, poiché lo stesso offset di fase applicato a frequenze diverse cambia il modo in cui le componenti della frequenza si allineano tra loro.

Ora che abbiamo la soluzione più generale al problema della sintesi - quella che gestisce ampiezze complesse - siamo pronti per il problema dell'analisi.

7.5 Il problema dell'analisi

Il problema dell'analisi è l'inverso del problema della sintesi: data una sequenza di campioni, y , e conoscendo le frequenze che compongono il segnale, possiamo calcolare le ampiezze complesse delle componenti, a ?

Come abbiamo visto nella Sezione 6.3, possiamo risolvere questo problema formando la matrice di sintesi, M , e risolvendo il sistema di equazioni lineari, $Ma = y$ per 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` accetta un array di onde (forse complesso), `ys`, una sequenza di frequenze reali, `fs` e una sequenza di tempi reali, `ts`. Restituisce una sequenza di ampiezze complesse, `amps`.

Continuando l'esempio precedente, possiamo confermare che `analyze1` recupera le ampiezze con cui siamo partiti. Affinché il risolutore di sistemi lineari funzioni, M deve essere quadrata, quindi abbiamo bisogno che `ys`, `fs` e `ts` abbiano la stessa lunghezza. Lo assicureremo tagliando `ys` e `ts` fino alla lunghezza di `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]
```

Queste sono approssimativamente le ampiezze con cui abbiamo iniziato, sebbene ogni componente abbia una piccola parte immaginaria a causa di errori della virgola mobile.

7.6 Analisi efficiente

Sfortunatamente, risolvere un sistema lineare di equazioni è lento. Per la DCT, siamo stati in grado di accelerare le cose scegliendo `fs` e `ts` in modo che M sia ortogonale. In questo modo, l'inverso di M è la trasposizione di M , e possiamo calcolare sia la DCT che la DCT inversa mediante moltiplicazione di matrici.

Faremo la stessa cosa per la DFT, con una piccola modifica. Poiché M è complesso, è necessario che sia **unitaria**, piuttosto che ortogonale, il che significa che l'inverso di M è la trasposta coniugata di M , che possiamo calcolare trasponendo la matrice e negando la parte immaginaria di ogni elemento. Vedere https://it.wikipedia.org/wiki/Matrice_unitaria.

I metodi NumPy `conj` e `transpose` fanno quello che vogliamo. Ecco il codice che calcola M per $N = 4$ componenti:

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

Se M è unitaria, $M^*M = I$, dove M^* è la trasposizione coniugata di M , e I è la matrice identità. Possiamo verificare se M è unitario in questo modo:

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

Il risultato, entro la tolleranza dell'errore in virgola mobile, è $4I$, quindi M è unitario ad eccezione di un fattore aggiuntivo di N , simile al fattore aggiuntivo di 2 che abbiamo trovato con la DCT.

Possiamo usare questo risultato per scrivere una versione più veloce di `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
```

E provarlo con i valori appropriati di `fs` e `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)
```

Anche in questo caso, il risultato è corretto entro la tolleranza dell'aritmetica in virgola mobile.

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

7.7 DFT

Come funzione, `analyze2` sarebbe difficile da usare perché funziona solo se `fs` e `ts` vengono scelti correttamente. Invece, la riscriveremo per prendendo solo `ys` e calcolando `fs` e `ts`.

Per prima cosa, si crea una funzione per calcolare la matrice di sintesi, 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
```

Poi si scrive la funzione che prende `ys` e restituisce `amps`:

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

Abbiamo quasi finito; `analyze3` calcola qualcosa di molto vicino alla DFT, con una differenza. La definizione convenzionale della DFT non divide per N :

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

Ora possiamo confermare che questa versione produce lo stesso risultato della FFT:

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

Il risultato è vicino ad `amps * N`. Ed ecco la versione in `np.fft`:

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

Sono uguali, entro l'errore della virgola mobile.

La DFT inversa è quasi la stessa, tranne che non dobbiamo trasporre e coniugare M , e ora dobbiamo dividere per N :

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

Infine, possiamo confermare che `dft(idft(amps))` produce `amps`.

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

Se si potesse tornare indietro nel tempo, si potrebbe cambiare la definizione della DFT in modo che divida per N e la DFT inversa no. Ciò sarebbe più coerente con la questa presentazione dei problemi di sintesi e analisi.

Oppure si potrebbe cambiare la definizione in modo che entrambe le operazioni si dividano per \sqrt{N} . Quindi la DFT e la DFT inversa sarebbero più simmetriche.

Ma non possiamo tornare indietro nel tempo (ancora!), Quindi siamo bloccati con una convenzione un po' strana. Per gli scopi pratici non ha molta importanza.

7.8 La DFT è periodica

In questo capitolo è stata presentata la DFT sotto forma di moltiplicazione di matrici. Calcoliamo la matrice di sintesi, M , e la matrice di analisi, M^* . Quando moltiplichiamo M^* per l'array dell'onda, y , ogni elemento del risultato è il prodotto di una riga da M^* e y , che possiamo scrivere sotto forma di somma:

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

dove k è un indice di frequenza da 0 a $N - 1$ e n è un indice di tempo da 0 a $N - 1$. Quindi $DFT(y)[k]$ è l'elemento k -esimo della DFT di y .

Normalmente valutiamo questa somma per N valori di k , compresi tra 0 e $N - 1$. *Potremmo* valutarlo per altri valori di k , ma non ha senso, perché iniziano a ripetersi. Cioè, il valore a k è uguale al valore a $k + N$ o a $k + 2N$ o a $k - N$, ecc.

Possiamo vederlo matematicamente inserendo $k + N$ nella somma:

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

Poiché c'è una somma nell'esponente, possiamo suddividerla in due parti:

$$DFT(y)[k + N] = \sum_n y[n] \exp(-2\pi i nk/N) \exp(-2\pi i nN/N)$$

Nel secondo termine, l'esponente è sempre un multiplo intero di 2π , quindi il risultato è sempre 1 e possiamo eliminarlo:

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

E possiamo vedere che questa somma è equivalente a $DFT(y)[k]$. Quindi la DFT è periodica, con periodo N . Ci sarà bisogno di questo risultato per uno degli esercizi seguenti, che chiede di implementare la Fast Fourier Transform (FFT).

Per inciso, scrivere la DFT sotto forma di sommatoria fornisce una panoramica di come funziona. Se si rivede il diagramma nella Sezione 6.2, si vedrà che ogni colonna della matrice di sintesi è un segnale valutato in una sequenza di tempi. La matrice di analisi è la trasposizione (coniugata) della matrice di sintesi, quindi ogni *riga* è un segnale valutato in una sequenza di tempi.

Pertanto, ciascuna somma è la correlazione di y con uno dei segnali nell'array (vedere la Sezione 5.5). Cioè, ogni elemento della DFT è una correlazione che quantifica la somiglianza della matrice dell'onda, y , e un esponenziale complesso a una particolare frequenza.

7.9 DFT di segnali reali

La classe Spectrum in `thinkdsp` è basata su `np.fft.rfft`, che calcola la “DFT reale”; cioè funziona con segnali reali. Ma la DFT presentata in questo capitolo è più generale; funziona con segnali complessi.

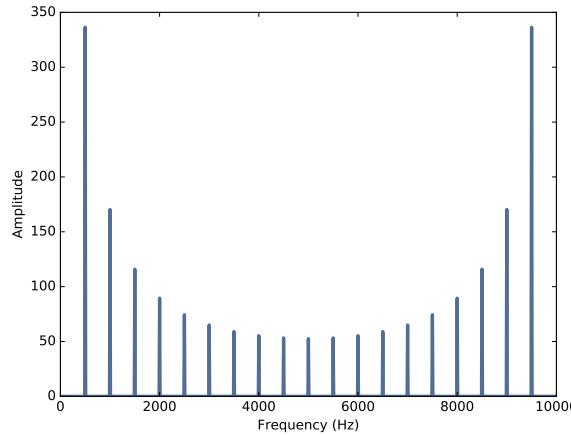


Figura 7.3: DFT di un segnale a dente di sega a 500 Hz campionato a 10 kHz.

Quindi cosa succede quando applichiamo la “DFT completa” a un segnale reale? Diamo un’occhiata a un esempio:

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

Questo codice crea un’onda a dente di sega con frequenza di 500 Hz, campionata a un framerate di 10 kHz. `hs` contiene la DFT complessa dell’onda; `amps` contiene l’ampiezza a ciascuna frequenza. Ma a quale frequenza corrispondono queste ampiezze? Se guardiamo il corpo di `dft`, vediamo:

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

Si è tentati di pensare che questi valori siano le frequenze giuste. Il problema è che `dft` non conosce la frequenza di campionamento. La DFT presume che la durata dell’onda sia 1 unità di tempo, quindi pensa che la frequenza di campionamento sia N per unità di tempo. Per interpretare le frequenze, dobbiamo convertire da queste unità di tempo arbitrarie a secondi, in questo modo:

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

Con questa modifica, la gamma di frequenze va da 0 al framerate effettivo, 10 kHz. Ora possiamo disegnare lo spettro:

```
plt.plot(fs, amps)
```

La Figura 7.3 mostra l’ampiezza del segnale per ogni componente di frequenza da 0 a 10 kHz. La metà sinistra della figura è ciò che dovremmo aspettarci: la frequenza dominante è a 500 Hz, con armoniche che si attenuano come $1/f$.

Ma la metà destra della figura è una sorpresa. Oltre i 5000 Hz, l’ampiezza delle armoniche ricomincia a crescere, raggiungendo il picco a 9500 Hz. Cosa succede?

La risposta: aliasing. Ci si deve ricordare che con un framerate di 10000 Hz, la frequenza di piegatura [folding] è di 5000 Hz. Come abbiamo visto nella Sezione 2.3, una componente a 5500 Hz è indistinguibile da una componente a 4500 Hz. Quando valutiamo la DFT a 5500 Hz, otteniamo lo stesso valore di 4500 Hz. Allo stesso modo, il valore a 6000 Hz è lo stesso di quello a 4000 Hz e così via.

La DFT di un segnale reale è simmetrica rispetto alla frequenza di ripiegamento [folding]. Dato che non ci sono informazioni aggiuntive oltre questo punto, possiamo risparmiare tempo valutando solo la prima metà della DFT, ed è esattamente ciò che fa `np.fft.rfft`.

7.10 Esercizi

Le soluzioni a questi esercizi sono in `chap07soln.ipynb`.

Esercizio 7.1 Il notebook per questo capitolo, `chap07.ipynb`, contiene ulteriori esempi e spiegazioni. Leggerlo ed eseguire il codice.

Esercizio 7.2 In questo capitolo è stato mostrato come possiamo esprimere la DFT e la DFT inversa con moltiplicazioni di matrici. Queste operazioni richiedono un tempo proporzionale a N^2 , dove N è la lunghezza della matrice dell’onda. Questo è abbastanza veloce per molte applicazioni, ma esiste un algoritmo più veloce, la Trasformata Veloce di Fourier (FFT), che richiede un tempo proporzionale a $N \log N$.

La chiave per la FFT è il lemma Danielson-Lanczos:

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

Dove $DFT(y)[n]$ è l’ n -esimo elemento della DFT di y ; e è un array di onde contenente gli elementi pari di y , e o contiene gli elementi dispari di y .

Questo lemma suggerisce un algoritmo ricorsivo per la DFT:

1. Dato un array di onde, y , lo si suddivide nei suoi elementi pari, e , e dispari, o .
2. Si calcola la DFT di e e di o effettuando chiamate ricorsive.
3. Si calcola $DFT(y)$ per ciascun valore di n utilizzando il lemma di Danielson-Lanczos.

Per il caso base di questa ricorsione, si potrebbe aspettare fino a quando la lunghezza di y è 1. In tal caso, $DFT(y) = y$. Oppure, se la lunghezza di y è sufficientemente piccola, è possibile calcolare la sua DFT mediante moltiplicazione di matrici, possibilmente utilizzando una matrice precalcolata.

Suggerimento: Implementare questo algoritmo in modo incrementale iniziando con una versione che non sia veramente ricorsiva. Nel passaggio 2, invece di effettuare una chiamata ricorsiva, utilizzare `dft`, come definito dalla Sezione 7.7, oppure `np.fft.fft`. Far funzionare il passaggio 3 e verificare che i risultati siano coerenti con le altre implementazioni. Poi aggiungere un caso di base e confermarne il funzionamento. Infine, sostituire il passaggio 2 con chiamate ricorsive.

Un altro suggerimento: ricordarsi che la DFT è periodica; potrebbe essere utile `np.tile`.

Ulteriori informazioni sulla FFT si trovano su https://en.wikipedia.org/wiki/Fast_Fourier_transform.

Capitolo 8

Filtraggio e Convoluzione

In questo capitolo si presenta una delle idee più importanti e utili relative all’elaborazione del segnale: il Teorema della Convoluzione. Ma prima di poter capire il Teorema della Convoluzione, dobbiamo capire la convoluzione. Inizieremo con un semplice esempio, lo smoothing, e andremo da lì.

Il codice per questo capitolo è in `chap08.ipynb`, nel repository di questo libro (vedere la Sezione 0.2). Lo si può anche visualizzare su <http://tinyurl.com/thinkdsp08>.

8.1 Smoothing

Lo smoothing è un’operazione che cerca di rimuovere le variazioni a breve termine da un segnale per rivelare le tendenze a lungo termine. Ad esempio, se si disegnano le variazioni giornaliere del prezzo di un’azione, queste sembrano rumorose; un operatore di smoothing potrebbe rendere più facile vedere se il prezzo sta generalmente aumentando o diminuendo nel tempo.

Un algoritmo di smoothing comune è una media mobile, che calcola la media degli n valori precedenti, per un dato valore di n .

Ad esempio, la Figura 8.1 mostra il prezzo di chiusura giornaliero di Facebook dal 17 maggio 2012 all’8 dicembre 2015. La linea grigia sono i dati originali, la linea più scura mostra la media mobile di 30 giorni. Lo smoothing rimuove i cambiamenti più estremi e rende più facile vedere le tendenze a lungo termine.

Le operazioni di smoothing si applicano anche ai segnali sonori. Per un esempio, inizieremo con un’onda quadra a 440 Hz. Come abbiamo visto nella Sezione 2.2, le armoniche di un’onda quadra diminuiscono lentamente, quindi contiene molte componenti ad alta frequenza.

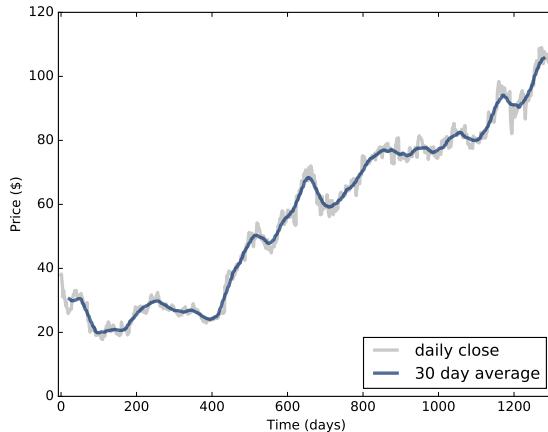


Figura 8.1: Prezzo di chiusura giornaliero delle azioni di Facebook e media mobile di 30 giorni.

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

`wave` è un segmento di 1 secondo del segnale; `segment` è un segmento più corto che useremo per la stampa.

Per calcolare la media mobile di questo segnale, creeremo una finestra con 11 elementi e la normalizzeremo in modo che gli elementi arrivino a 1.

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

Ora possiamo calcolare la media dei primi 11 elementi moltiplicando la finestra per l'array dell'onda:

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

`padded` è una versione della finestra con zeri aggiunti alla fine, quindi ha la stessa lunghezza di `segment.ys`.

`prod` è il prodotto della finestra e dell'array dell'onda.

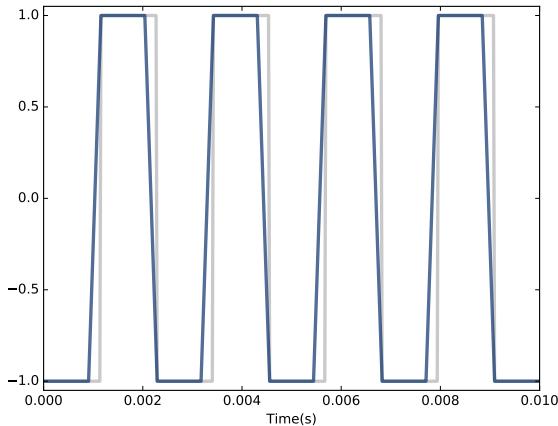


Figura 8.2: Un segnale quadrato a 400 Hz (grigio) e una media mobile di 11 elementi.

La somma dei prodotti degli elementi è la media dei primi 11 elementi dell'array. Poiché questi elementi sono tutti -1, la loro media è -1

Per calcolare l'elemento successivo del segnale con smoothing, spostiamo la finestra a destra e calcoliamo la media dei successivi 11 elementi della matrice dell'onda, a partire dal secondo.

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

Il risultato è di nuovo -1. Per calcolare la media mobile, ripetiamo questo processo e memorizziamo il risultato in un array denominato **smoothed**:

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

rolled è una copia di **padded** che viene spostata a destra di un elemento ogni volta nel ciclo. All'interno del ciclo, moltiplichiamo il segmento per **rolled** per selezionare 11 elementi, quindi li sommiamo.

La Figura 8.2 mostra il risultato. La linea grigia è il segnale originale; la linea più scura è il segnale con smoothing. Il segnale con smoothing inizia a salire quando il bordo anteriore della finestra raggiunge la prima transizione

e si livella quando la finestra attraversa la transizione. Di conseguenza, le transizioni sono meno brusche e gli angoli meno nitidi. Se si ascolta il segnale con smoothing, suona meno rumoroso e leggermente ovattato.

8.2 Convolution

L'operazione che abbiamo appena calcolato si chiama **convoluzione**, ed è un'operazione così comune che NumPy fornisce un'implementazione più semplice e veloce di questa versione:

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

`np.convolve` calcola la convoluzione dell'array dell'onda e della finestra. Il flag della modalità, `valid` indica che deve calcolare i valori solo quando la finestra e l'array dell'onda si sovrappongono completamente, quindi si ferma quando il bordo destro della finestra raggiunge la fine dell'array. A parte questo, il risultato è lo stesso della Figura 8.2.

In realtà, c'è un'altra differenza. Il ciclo nella sezione precedente calcola effettivamente la **cross-correlazione**:

$$(f \star g)[n] = \sum_{m=0}^{N-1} f[m]g[n+m]$$

dove f è un array di onde con lunghezza N , g è la finestra e \star è il simbolo della cross-correlazione. Per calcolare l'elemento n -esimo del risultato, spostiamo g a destra, motivo per cui l'indice è $n + m$.

La definizione di convoluzione è leggermente diversa:

$$(f * g)[n] = \sum_{m=0}^{N-1} f[m]g[n-m]$$

Il simbolo $*$ rappresenta la convoluzione. La differenza sta nell'indice di g : m è stato negato, quindi la somma itera gli elementi di g all'indietro (supponendo che gli indici negativi si estendano alla fine dell'array).

Poiché la finestra usata in questo esempio è simmetrica, la cross-correlazione e la convoluzione producono lo stesso risultato. Con altre finestre, dovremo stare più attenti.

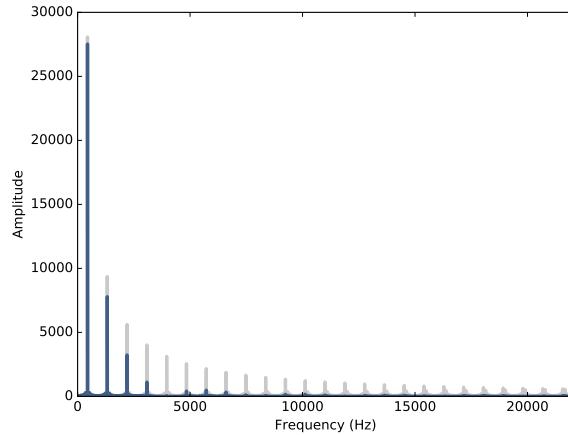


Figura 8.3: Spettro dell’onda quadra prima e dopo lo smoothing.

Ci si potrebbe chiedere perché la convoluzione è definita in questo modo. Ci sono due ragioni:

- Questa definizione si presenta naturalmente per diverse applicazioni, in particolare nell’analisi dei sistemi di elaborazione del segnale, che è l’argomento del Capitolo 10.
- Inoltre, questa definizione è la base del Teorema della Convoluzione, che verrà fuori molto presto.

8.3 Il dominio della frequenza

Lo smoothing rende le transizioni in un segnale quadrato meno brusche e rende il suono leggermente attutito. Vediamo che effetto ha questa operazione sullo spettro. Per prima cosa disegniamo lo spettro dell’onda originale:

```
spectrum = wave.make_spectrum()
spectrum.plot(color=GRAY)
```

Poi l’onda con smoothing:

```
convolved = np.convolve(wave.ys, window, mode='same')
smooth = thinkdsp.Wave(convolved, framerate=wave.framerate)
spectrum2 = smooth.make_spectrum()
spectrum2.plot()
```

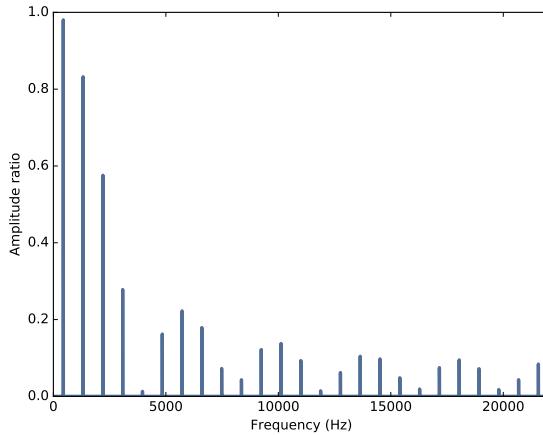


Figura 8.4: Rapporto degli spettri per l'onda quadra, prima e dopo lo smoothing.

Il flag della modalità `same` indica che il risultato dovrebbe avere la stessa lunghezza dell'input. In questo esempio, includerà alcuni valori che di “arrotolamento [wrap around]”, ma per ora va bene.

La Figura 8.3 mostra il risultato. La frequenza fondamentale è pressoché invariata; le prime poche armoniche vengono attenuate e le armoniche superiori vengono quasi eliminate. Quindi lo smoothing ha l'effetto di un filtro passabasso, che abbiamo visto nella Sezione 1.5 e nella Sezione 4.4.

Per vedere di quanto ogni componente sia stata attenuata, possiamo calcolare il rapporto tra i due spettri:

```
amps = spectrum.amps
amps2 = spectrum2.amps
ratio = amps2 / amps
ratio[amps<560] = 0
plt.plot(ratio)
```

`ratio` è il rapporto dell'ampiezza prima e dopo lo smoothing. Quando `amps` è piccolo, questo rapporto può essere grande e rumoroso, quindi per semplicità abbiamo impostato il rapporto su 0 tranne dove ci sono le armoniche.

La Figura 8.4 mostra il risultato. Come previsto, il rapporto è alto per le basse frequenze e scende a una frequenza di taglio vicino a 4000 Hz. Ma c'è un'altra caratteristica che non ci aspettavamo: al di sopra del taglio [cutoff], il rapporto rimbalza tra 0 e 0.2. Cosa succede da quella parte?

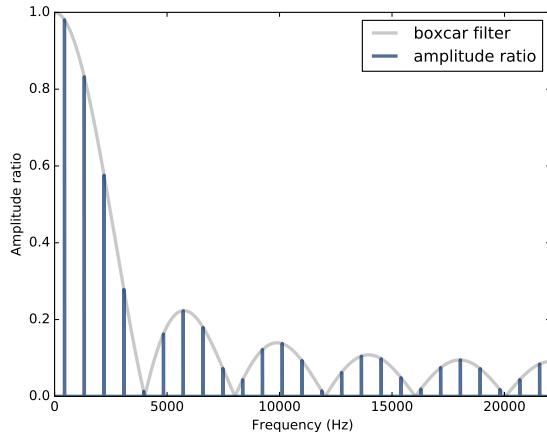


Figura 8.5: Rapporto degli spettri per l'onda quadra, prima e dopo lo smoothing, insieme alla DFT della finestra di smoothing.

8.4 Il teorema della convoluzione

La risposta è il teorema della convoluzione. espresso matematicamente:

$$\text{DFT}(f * g) = \text{DFT}(f) \cdot \text{DFT}(g)$$

dove f è l'array di un'onda e g è una finestra. In parole, il teorema di convoluzione dice che se convolgiamo f e g e poi calcoliamo la DFT, otteniamo la stessa risposta calcolando la DFT di f e g e moltiplicando poi il risultato per gli elementi. Più concisamente, la convoluzione nel dominio del tempo corrisponde alla moltiplicazione nel dominio della frequenza.

E questo spiega la Figura 8.4, perché quando si effettua la convoluzione di un'onda e una finestra, moltiplichiamo lo spettro dell'onda per lo spettro della finestra. Per vedere come funziona, possiamo calcolare la DFT della finestra:

```
padded = zero_pad(window, N)
dft_window = np.fft.rfft(padded)
plt.plot(abs(dft_window))
```

`padded` contiene la finestra, completata con zeri per avere la stessa lunghezza di `wave`. `dft_window` contiene la DFT della finestra di smoothing.

La Figura 8.5 mostra il risultato, insieme ai rapporti che abbiamo calcolato nella sezione precedente. I rapporti sono esattamente le ampiezze in `dft_window`. Matematicamente:

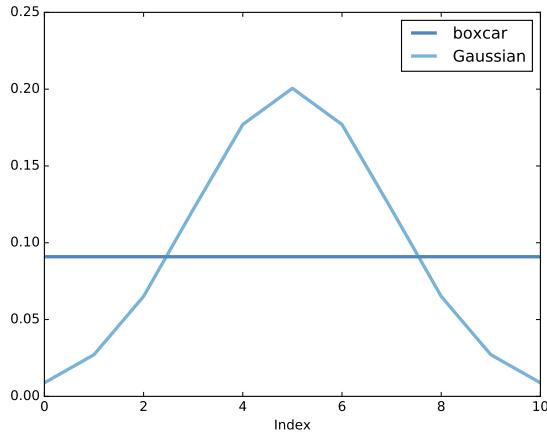


Figura 8.6: Boxcar e finestre Gaussiane.

$$\text{abs}(\text{DFT}(f * g)) / \text{abs}(\text{DFT}(f)) = \text{abs}(\text{DFT}(g))$$

In questo contesto, la DFT di una finestra è chiamata **filtro**. Per ogni finestra di convoluzione nel dominio del tempo, esiste un filtro corrispondente nel dominio della frequenza. E per ogni filtro che può essere espresso dalla moltiplicazione per ogni elemento nel dominio della frequenza, c'è una finestra corrispondente.

8.5 Filtro Gaussiano

La finestra della media mobile che abbiamo usato nella sezione precedente è un filtro passa-basso, ma non è molto buono. All'inizio la DFT si abbassa rapidamente, ma poi rimbalza. Questi rimbalzi sono chiamati **lobi laterali** e sono presenti perché la finestra della media mobile è come un'onda quadrata, quindi il suo spettro contiene armoniche ad alta frequenza che scendono proporzionalmente a $1/f$, che è relativamente lento.

Possiamo fare di meglio con una finestra Gaussiana. SciPy fornisce delle funzioni che calcolano molte finestre di convoluzione comuni, inclusa `gaussian`:

```
gaussian = scipy.signal.gaussian(M=11, std=2)
gaussian /= sum(gaussian)
```

`M` è il numero di elementi nella finestra; `std` è la deviazione standard della distribuzione Gaussiana usata per calcolarla. La Figura 8.6 mostra la forma della

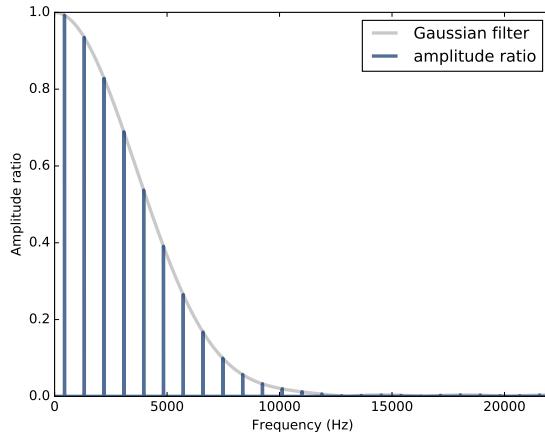


Figura 8.7: Rapporto degli spettri prima e dopo lo smoothing Gaussiano e la DFT della finestra.

finestra. È un'approssimazione discreta della “curva a campana” Gaussiana. La figura mostra anche la finestra della media mobile dell'esempio precedente, che a volte è chiamata **finestra boxcar** perché ha l'aspetto di un vagone ferroviario rettangolare.

Sono stati eseguiti di nuovo i calcoli delle sezioni precedenti con questa curva e si è generata la Figura 8.7, che mostra il rapporto degli spettri prima e dopo lo smoothing, insieme alla DFT della finestra Gaussiana.

Come filtro passa-basso, lo smoothing Gaussiano è migliore di una semplice media mobile. Dopo che il rapporto si è abbassato, rimane basso, con quasi nessuno dei lobi laterali visti con la finestra del boxcar. Quindi fa un lavoro migliore nel tagliare le frequenze più alte.

Il motivo per cui funziona così bene è che anche la DFT di una curva Gaussiana è una curva Gaussiana. Quindi il rapporto scende in proporzione a $\exp(-f^2)$, che è molto più veloce di $1/f$.

8.6 Convoluzione efficiente

Uno dei motivi per cui la FFT è un algoritmo così importante è che, combinato con il Teorema della Convoluzione, fornisce un modo efficiente per calcolare la convoluzione, la cross-correlazione e l'autocorrelazione.

Ancora una volta, il Teorema della Convoluzione afferma

$$\text{DFT}(f * g) = \text{DFT}(f) \cdot \text{DFT}(g)$$

Quindi un modo per calcolare una convoluzione è:

$$f * g = \text{IDFT}(\text{DFT}(f) \cdot \text{DFT}(g))$$

dove $IDFT$ è la DFT inversa. Una semplice implementazione della convoluzione richiede un tempo proporzionale a N^2 ; questo algoritmo, utilizzando la FFT, richiede un tempo proporzionale a $N \log N$.

Possiamo confermare che funziona calcolando la stessa convoluzione in entrambi i modi. Ad esempio, lo applicheremo ai dati del BitCoin mostrati nella Figura 8.1.

```
df = pandas.read_csv('coindesk-bpi-USD-close.csv',
                     nrows=1625,
                     parse_dates=[0])
ys = df.Close.values
```

Questo esempio utilizza Pandas per leggere i dati dal file CSV (incluso nel repository di questo libro). Se non si ha familiarità con i Panda, non c'è da preoccuparsi: non se ne farà molto in questo libro. Ma per saperne di più si può leggere *Think Stats* da <http://thinkstats2.com>.

Il risultato, `df`, è un DataFrame, una delle strutture dati fornite da Pandas. `ys` è un array NumPy che contiene i prezzi di chiusura giornalieri.

Successivamente creeremo una finestra Gaussiana per una convoluzione con `ys`:

```
window = scipy.signal.gaussian(M=30, std=6)
window /= window.sum()
smoothed = np.convolve(ys, window, mode='valid')
```

`fft_convolve` calcola la stessa cosa usando la FFT:

```
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)
```

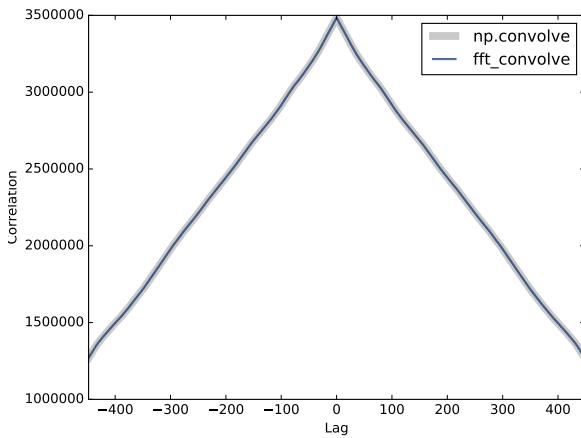


Figura 8.8: Funzioni di autocorrelazione calcolate da NumPy e `fft_convolve`.

Possiamo testarlo riempiendo la finestra alla stessa lunghezza di `ys` e poi calcolando la convoluzione:

```
padded = zero_pad(window, N)
smoothed2 = fft_convolve(ys, padded)
```

Il risultato ha $M - 1$ valori fasulli all'inizio, dove M è la lunghezza della finestra. Se escludiamo i valori fasulli, il risultato concorda con `fft_convolve` con circa 12 cifre di precisione.

```
M = len(window)
smoothed2 = smoothed2[M-1:]
```

8.7 Autocorrelazione efficiente

Nella Sezione 8.2 abbiamo presentato le definizioni di cross-correlazione e di convoluzione, e abbiamo visto che sono quasi le stesse, tranne che nella convoluzione la finestra è invertita.

Ora che abbiamo un algoritmo efficiente per la convoluzione, possiamo anche usarlo per calcolare cross-correlazioni e autocorrelazioni. Utilizzando i dati della sezione precedente, possiamo calcolare l'autocorrelazione dei prezzi delle azioni di Facebook:

```
corrs = np.correlate(close, close, mode='same')
```

Con `mode='same'`, il risultato ha la stessa lunghezza di `close`, corrispondente ai ritardi da $-N/2$ to $N/2 - 1$. La linea grigia nella Figura 8.8 mostra il risultato. Tranne che a `lag=0`, non ci sono picchi, quindi non vi è alcun comportamento periodico apparente in questo segnale. Tuttavia, la funzione di autocorrelazione diminuisce lentamente, suggerendo che questo segnale assomiglia al rumore rosa, come abbiamo visto nella Sezione 5.3.

Per calcolare l'autocorrelazione usando la convoluzione, dobbiamo aggiungere zeri al segnale per raddoppiarne la lunghezza. Questo trucco è necessario perché la FFT si basa sul presupposto che il segnale sia periodico; cioè, che si riavvolge dalla fine all'inizio. Con dati di serie temporali come questo, tale ipotesi non è valida. Con l'aggiunta degli zeri e col taglio dei risultati si rimuovono i valori fasulli.

Inoltre, ricordarsi che la convoluzione inverte la direzione della finestra. Per annullare questo effetto, invertiamo la direzione della finestra prima di chiamare `fft_convolve`, utilizzando `np.flipud`, che capovolge un array NumPy. Il risultato è una vista dell'array, non una copia, quindi questa operazione è veloce.

```
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
```

Il risultato di `fft_convolve` ha lunghezza $2N$. Di questi, i primi e ultimi $N/2$ sono validi; il resto è il risultato del riempimento con zeri. Per selezionare l'elemento valido, eseguiamo l'arrotolamento dei risultati e selezioniamo i primi N , corrispondente ai ritardi da $-N/2$ a $N/2 - 1$.

Come mostrato in Figura 8.8 i risultati di `fft_autocorr` e `np.correlate` sono identici (con circa 9 cifre di precisione).

Si noti che le correlazioni nella Figura 8.8 sono numeri grandi; potremmo normalizzarli (tra -1 e 1) come mostrato nella Sezione 5.6.

La strategia che abbiamo usato qui per l'auto-correlazione funziona anche per la cross-correlazione. Di nuovo, si devono preparare i segnali girandone uno e riempiendoli entrambi, poi si devono tagliare le parti non valide del risultato. Questo riempimento e taglio è fastidioso, ma è per questo che le librerie come NumPy forniscono delle funzioni per farlo.

8.8 Esercizi

Le soluzioni a questi esercizi sono in `chap08soln.ipynb`.

Esercizio 8.1 Il notebook per questo capitolo è `chap08.ipynb`. Leggerlo ed eseguire il codice.

Contiene un widget interattivo che consente di provare i parametri della finestra Gaussiana per vedere quale effetto hanno sulla frequenza di taglio.

Cosa va storto quando si aumenta la larghezza della Gaussiana, `std`, senza aumentare il numero di elementi nella finestra, `M`?

Esercizio 8.2 In questo capitolo si afferma che la trasformata di Fourier di una curva Gaussiana è anch'essa una curva Gaussiana. Per le trasformate di Fourier discrete, questa relazione è approssimativamente vera.

Provarlo per alcuni esempi. Cosa succede alla trasformata di Fourier quando si varia `std`?

Esercizio 8.3 Se sono stati eseguiti gli esercizi nel Capitolo 3, si è visto l'effetto della finestra di Hamming, e di alcune delle altre finestre fornite da NumPy, sulla dispersione spettrale. Possiamo avere un'idea dell'effetto di queste finestre osservando le loro DFT.

Oltre a quella Gaussiana che abbiamo usato in questo capitolo, creare una finestra di Hamming con le stesse dimensioni. Completare con degli zeri le finestre e disegnarne le DFT. Quale finestra funge da filtro passa basso migliore? Potrebbe essere utile disegnare la DFT su una scala log- y .

Sperimentare con alcune finestre diverse e alcune dimensioni diverse.

Capitolo 9

Differenziazione e Integrazione

Questo capitolo riprende da dove si era interrotto il capitolo precedente, esaminando la relazione tra le finestre nel dominio del tempo e i filtri nel dominio della frequenza.

In particolare, esamineremo l'effetto di una finestra alle differenze finite, che approssima la differenziazione, e l'operazione di somma cumulativa, che approssima l'integrazione.

Il codice per questo capitolo si trova in `chap09.ipynb`, nel repository di questo libro (vedere la Sezione 0.2). Lo si può anche visualizzare su <http://tinyurl.com/thinkdsp09>.

9.1 Differenze finite

Nella Sezione 8.1, abbiamo applicato una finestra di smoothing al prezzo delle azioni di Facebook e abbiamo scoperto che una finestra di smoothing nel dominio del tempo corrisponde a un filtro passa-basso nel dominio della frequenza.

In questa sezione, esamineremo le variazioni di prezzo giornaliere e vedremo che il calcolo della differenza tra elementi successivi, nel dominio del tempo, corrisponde a un filtro passa alto.

Ecco il codice per leggere i dati, memorizzarli come un'onda e calcolarne lo spettro.

```
import pandas as pd
```

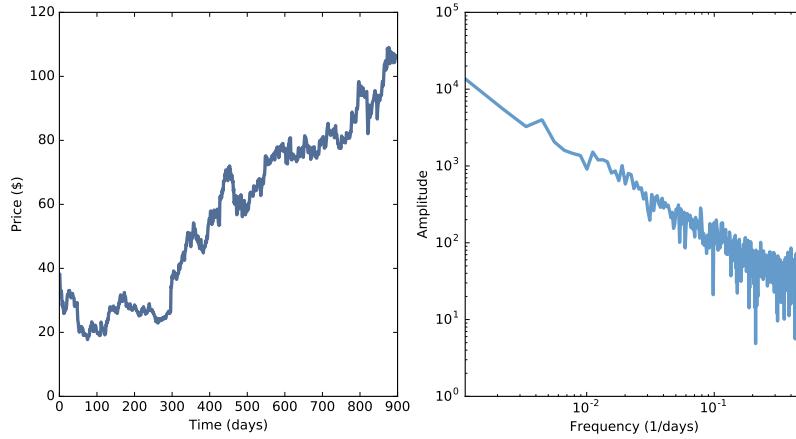


Figura 9.1: Prezzo di chiusura giornaliero di Facebook e spettro di questa serie temporale.

```
names = ['date', 'open', 'high', 'low', 'close', 'volume']
df = pd.read_csv('fb_1.csv', header=0, names=names)
ys = df.close.values[::-1]
close = thinkdsp.Wave(ys, framerate=1)
spectrum = close.make_spectrum()
```

Questo esempio utilizza Pandas per leggere il file CSV; il risultato è un DataFrame, `df`, con colonne per il prezzo di apertura, quello di chiusura, il prezzo massimo e quello minimo. Si selezionano i prezzi di chiusura e si salvano in un oggetto Wave. Il framerate è di 1 campione al giorno.

La Figura 9.1 mostra questa serie temporale e il suo spettro. Visivamente, la serie temporale assomiglia al rumore Browniano (vedere la Sezione 4.3). E lo spettro sembra una linea retta, anche se rumorosa. La pendenza stimata è -1,9, che è coerente con il rumore Browniano.

Ora calcoliamo la variazione di prezzo giornaliera utilizzando `np.diff`:

```
diff = np.diff(ys)
change = thinkdsp.Wave(diff, framerate=1)
change_spectrum = change.make_spectrum()
```

La Figura 9.2 mostra l'onda risultante e il suo spettro. I cambiamenti giornalieri assomigliano al rumore bianco e la pendenza stimata dello spettro, -0.06, è prossima allo zero, che è ciò che ci aspettiamo per il rumore bianco.

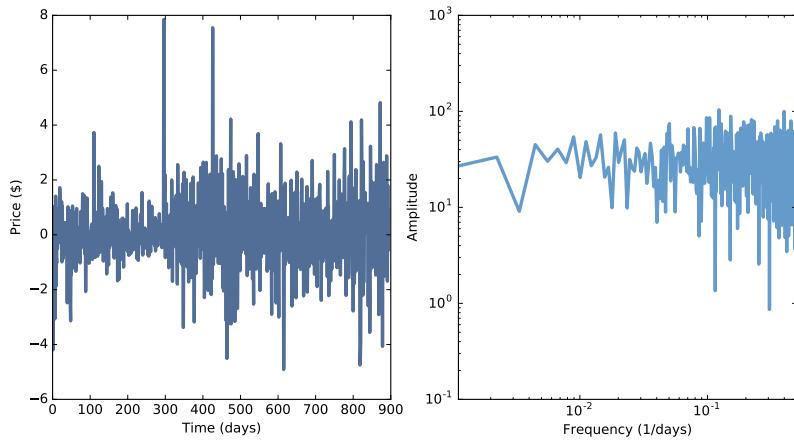


Figura 9.2: Variazione giornaliera del prezzo di Facebook e spettro di questa serie temporale.

9.2 Il dominio della frequenza

Il calcolo della differenza tra elementi successivi è lo stesso della convoluzione con la finestra `[1, -1]`. Se l'ordine di quegli elementi sembra invertito, si ricorda che la convoluzione inverte la finestra prima di applicarla al segnale.

Possiamo vedere l'effetto di questa operazione nel dominio della frequenza calcolando la DFT della finestra.

```
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()
```

La Figura 9.3 mostra il risultato. La finestra delle differenze finite corrisponde a un filtro passa-alto: la sua ampiezza aumenta con la frequenza, linearmente per le basse frequenze e dopo sub-linealmente. Nella prossima sezione vedremo perché.

9.3 Differenziazione

La finestra che abbiamo usato nella sezione precedente è un'approssimazione numerica della derivata prima, quindi il filtro approssima l'effetto della differenziazione.

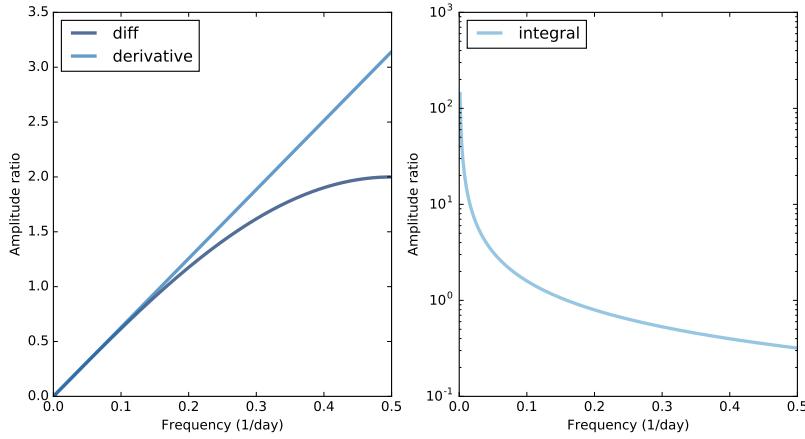


Figura 9.3: Filtri corrispondenti agli operatori diff e differenziazione (a sinistra) e all'operatore di integrazione (a destra, scala log- y).

La differenziazione nel dominio del tempo corrisponde a un semplice filtro nel dominio della frequenza; possiamo capire di cosa si tratta con un po' di matematica.

Supponiamo di avere una sinusoida complessa con frequenza f :

$$E_f(t) = e^{2\pi i f t}$$

La derivata prima di E_f è

$$\frac{d}{dt} E_f(t) = 2\pi i f e^{2\pi i f t}$$

che possiamo riscrivere come

$$\frac{d}{dt} E_f(t) = 2\pi i f E_f(t)$$

In altre parole, prendere la derivata di E_f equivale a moltiplicare per $2\pi i f$, che è un numero complesso con magnitudine $2\pi f$ ed un angolo di $\pi/2$.

Possiamo calcolare il filtro che corrisponde alla differenziazione, in questo modo:

```
deriv_filter = close.make_spectrum()
deriv_filter.hs = PI2 * 1j * deriv_filter.fs
```

Si è iniziato con lo spettro di `close`, che ha le dimensioni e il framerate corretti, poi è stato sostituito `hs` con $2\pi if$. La Figura 9.3 (a sinistra) mostra questo filtro; è una linea retta.

Come abbiamo visto nella Sezione 7.4, la moltiplicazione di una sinusode complessa per un numero complesso ha due effetti: moltiplica l'ampiezza, in questo caso per $2\pi f$, e sposta l'offset della fase, in questo caso per $\pi/2$.

Se si ha familiarità con gli operatori e le autofunzioni, ogni E_f è un'autofunzione dell'operatore di differenziazione, con il corrispondente autovalore $2\pi if$. Vedere <http://en.wikipedia.org/wiki/Eigenfunction>.

Se non si ha familiarità con quel linguaggio, ecco cosa significa:

- Un operatore è una funzione che accetta una funzione e restituisce un'altra funzione. Ad esempio, la differenziazione è un operatore.
- Una funzione, g , è un'autofunzione di un operatore, \mathcal{A} , se l'applicazione di \mathcal{A} a g ha l'effetto di moltiplicare la funzione per uno scalare. Cioè, $\mathcal{A}g = \lambda g$.
- In tal caso, lo scalare λ è l'autovalore che corrisponde all'autofunzione g .
- Un dato operatore potrebbe avere molte autofunzioni, ciascuna con un corrispondente autovalore.

Poiché le sinusodi complesse sono autofunzioni dell'operatore di differenziazione, sono facili da differenziare. Tutto quello che dobbiamo fare è moltiplicare per uno scalare complesso.

Per i segnali con più di una componente, il processo è solo leggermente più complicato:

1. Esprimere il segnale come somma di sinusodi complesse.
2. Calcolare la derivata di ogni componente per moltiplicazione.
3. Sommare le componenti differenziate.

Se questo processo suona familiare, è perché è identico all'algoritmo per la convoluzione nella Sezione 8.6: calcolare la DFT, moltiplicarla per un filtro e calcolare la DFT inversa.

`Spectrum` fornisce un metodo che applica il filtro di differenziazione:

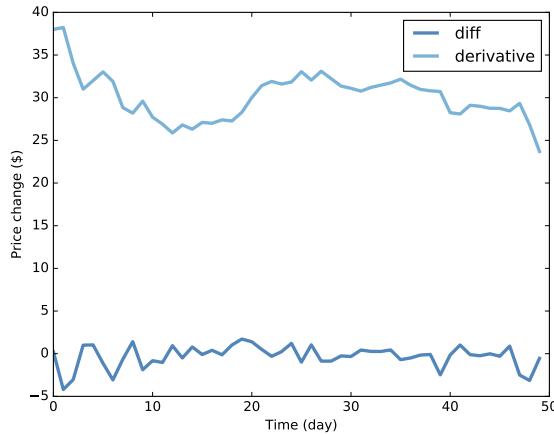


Figura 9.4: Confronto delle variazioni di prezzo giornaliere calcolato da `np.diff` e applicando il filtro di differenziazione.

```
# class Spectrum:

    def differentiate(self):
        self.hs *= PI2 * 1j * self.fs
```

Possiamo usarlo per calcolare la derivata della serie temporale di Facebook:

```
deriv_spectrum = close.make_spectrum()
deriv_spectrum.differentiate()
deriv = deriv_spectrum.make_wave()
```

La Figura 9.4 confronta le variazioni di prezzo giornaliere calcolate da `np.diff` con la derivata appena calcolata. Sono stati selezionati i primi 50 valori nelle serie temporali in modo da poter vedere le differenze più chiaramente.

La derivata è più rumorosa, perché amplifica maggiormente le componenti ad alta frequenza, come mostrato in Figura 9.3 (a sinistra). Inoltre, i primi pochi elementi della derivata sono molto rumorosi. Il problema è che la derivata basata sulla DFT si basa sul presupposto che il segnale sia periodico. In effetti, collega di nuovo l'ultimo elemento della serie temporale al primo elemento, creando artefatti ai bordi.

Per riassumere, abbiamo mostrato:

- Il calcolo della differenza tra valori successivi in un segnale può essere espresso come una convoluzione con una semplice finestra. Il risultato è un'approssimazione della derivata prima.



Figura 9.5: Confronto tra la serie storica originale e la derivata integrata.

- La differenziazione nel dominio del tempo corrisponde a un semplice filtro nel dominio della frequenza. Per i segnali periodici, il risultato è esattamente la derivata prima. Per alcuni segnali non periodici, può approssimare la derivata.

L’uso della DFT per calcolare le derivate è la base dei **metodi spettrali** per la risoluzione di equazioni differenziali (vedere http://en.wikipedia.org/wiki/Spectral_method).

In particolare, è utile per l’analisi di sistemi lineari, tempo-invarianti, che verrà trattata nel Capitolo 10.

9.4 Integrazione

Nella sezione precedente, abbiamo mostrato che la differenziazione nel dominio del tempo corrisponde a un semplice filtro nel dominio della frequenza: moltiplica ogni componente per $2\pi if$. Poiché l’integrazione è l’inverso della differenziazione, corrisponde anche a un semplice filtro: divide ogni componente per $2\pi if$.

Possiamo calcolare tale filtro in questo modo:

```
integ_filter = close.make_spectrum()
integ_filter.hs = 1 / (PI2 * 1j * integ_filter.fs)
```

La Figura 9.3 (a destra) mostra questo filtro su una scala log- y , che lo rende più facile da vedere.

`Spectrum` fornisce un metodo che applica il filtro di integrazione:

```
# class Spectrum:

    def integrate(self):
        self.hs /= PI2 * 1j * self.fs
```

Possiamo confermare che il filtro di integrazione è corretto applicandolo allo spettro della derivata che abbiamo appena calcolato:

```
integ_spectrum = deriv_spectrum.copy()
integ_spectrum.integrate()
```

Ma si noti che a $f = 0$, si sta dividendo per 0. Il risultato in NumPy è NaN, che è uno speciale valore a virgola mobile che rappresenta “not a number” [un non numero]. Possiamo affrontare parzialmente questo problema impostando questo valore su 0 prima di riconvertire lo spettro in un’onda:

```
integ_spectrum.hs[0] = 0
integ_wave = integ_spectrum.make_wave()
```

La Figura 9.5 mostra questa derivata integrata insieme alle serie temporali originali. Sono quasi identiche, ma la derivata integrata è spostata verso il basso. Il problema è che quando abbiamo eliminato la componente $f = 0$, abbiamo impostato lo spostamento [bias] del segnale a 0. Ma questo non dovrebbe sorprendere; in generale, la differenziazione perde informazioni sul bias e l’integrazione non può recuperarle. In un certo senso, il NaN a $f = 0$ ci dice che questo elemento è ignoto.

Se aggiungiamo questa “costante di integrazione”, i risultati sono identici, il che conferma che questo filtro di integrazione è l’inverso corretto del filtro di differenziazione.

9.5 Somma cumulativa

Allo stesso modo in cui l’operatore `diff` approssima la differenziazione, la somma cumulativa approssima l’integrazione. Lo dimostreremo con un segnale a dente di sega.

```
signal = thinkdsp.SawtoothSignal(freq=50)
in_wave = signal.make_wave(duration=0.1, framerate=44100)
```

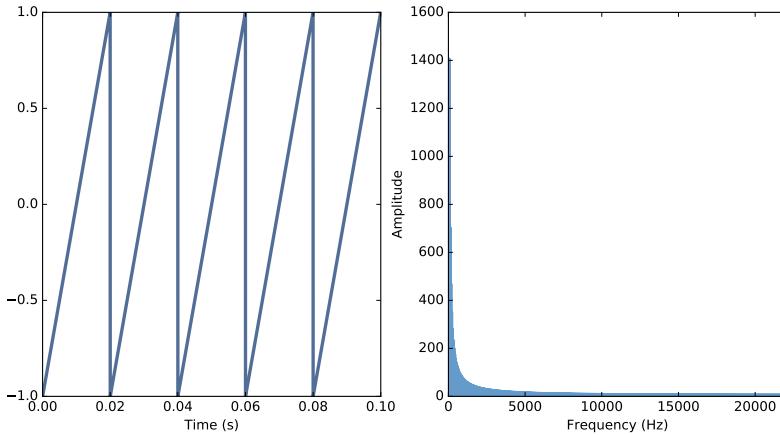


Figura 9.6: Un'onda a dente di sega e il suo spettro.

La Figura 9.6 mostra quest'onda e il suo spettro.

Wave fornisce un metodo che calcola la somma cumulativa di un array di onde e restituisce un nuovo oggetto Wave:

```
# class Wave:

    def cumsum(self):
        ys = np.cumsum(self.ys)
        ts = self.ts.copy()
        return Wave(ys, ts, self.framerate)
```

Possiamo usarlo per calcolare la somma cumulativa di `in_wave`:

```
out_wave = in_wave.cumsum()
out_wave.unbias()
```

La Figura 9.7 mostra l'onda risultante e il suo spettro. Se sono stati fatti gli esercizi nel Capitolo 2, questa forma d'onda dovrebbe apparire familiare: è un segnale parabolico.

Confrontando lo spettro del segnale parabolico con lo spettro del dente di sega, le ampiezze delle componenti diminuiscono più rapidamente. Nel Capitolo 2, abbiamo visto che le componenti del dente di sega diminuiscono proporzionalmente a $1/f$. Poiché la somma cumulativa approssima l'integrazione e l'integrazione filtra le componenti in proporzione a $1/f$, le componenti dell'onda parabolica diminuiscono proporzionalmente a $1/f^2$.

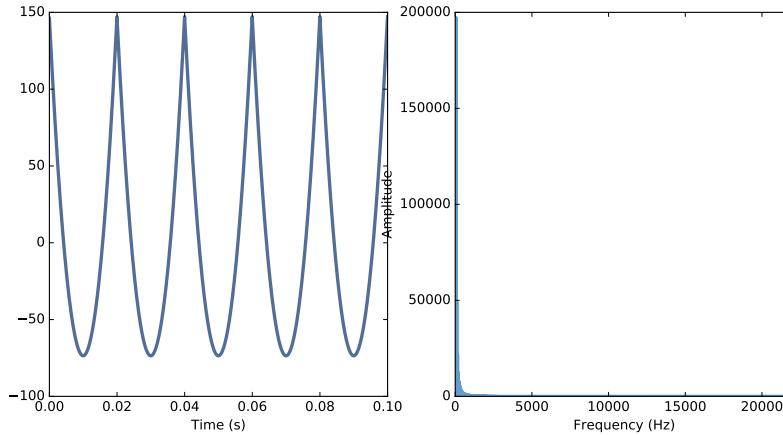


Figura 9.7: Un'onda parabolica e il suo spettro.

Possiamo vederlo graficamente calcolando il filtro che corrisponde alla somma cumulativa:

```
cumsum_filter = diff_filter.copy()
cumsum_filter.hs = 1 / cumsum_filter.hs
```

Poiché `cumsum` è l'operazione inversa di `diff`, iniziamo con una copia di `diff_filter`, che è il filtro che corrisponde all'operazione `diff`, e poi si invertono le `hs`.

La Figura 9.8 mostra i filtri corrispondenti alla somma cumulativa e all'integrazione. La somma cumulativa è una buona approssimazione dell'integrazione tranne che alle frequenze più alte, dove diminuisce un po' più velocemente.

Per confermare che questo è il filtro corretto per la somma cumulativa, possiamo confrontarlo con il rapporto tra lo spettro `out_wave` e lo spettro di `in_wave`:

```
in_spectrum = in_wave.make_spectrum()
out_spectrum = out_wave.make_spectrum()
ratio_spectrum = out_spectrum.ratio(in_spectrum, thresh=1)
```

Ed ecco il metodo che calcola i rapporti:

```
def ratio(self, denom, thresh=1):
    ratio_spectrum = self.copy()
    ratio_spectrum.hs /= denom.hs
    ratio_spectrum.hs[denomamps < thresh] = np.nan
    return ratio_spectrum
```

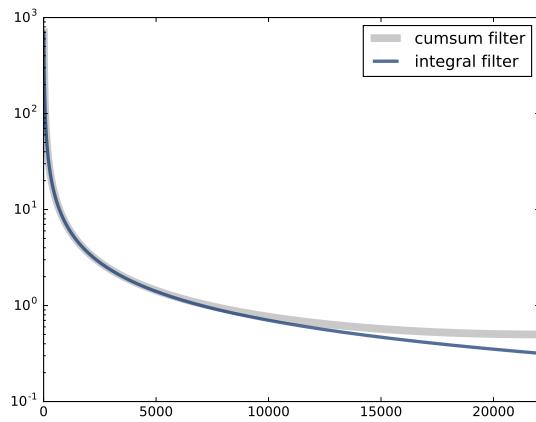


Figura 9.8: Filtri corrispondenti alla somma cumulativa e all'integrazione.

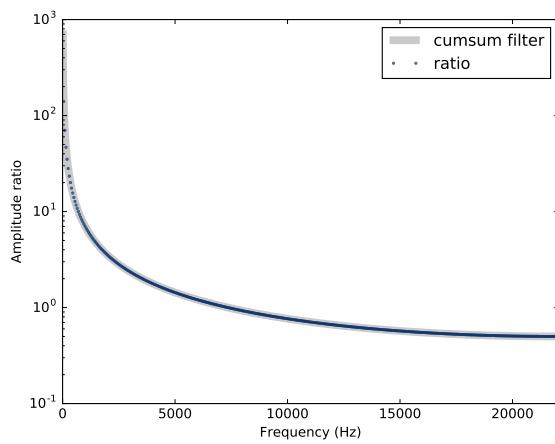


Figura 9.9: Filtro corrispondente alla somma cumulativa e ai rapporti effettivi degli spettri prima e dopo.

Quando `denom.amps` è piccolo, il rapporto risultante è rumoroso, quindi sono stati impostati questi valori a NaN.

La Figura 9.9 mostra i rapporti e il filtro corrispondente alla somma cumulativa. Corrispondono, il che conferma che invertendo il filtro per `diff` si ottiene il filtro per `cumsum`.

Infine, possiamo confermare che il Teorema della Convoluzione si realizza applicando il filtro `cumsum` nel dominio della frequenza:

```
out_wave2 = (in_spectrum * cumsum_filter).make_wave()
```

Entro i limiti dell'errore in virgola mobile, `out_wave2` è identico a `out_wave`, che abbiamo calcolato utilizzando `cumsum`, quindi il Teorema della Convoluzione funziona! Ma notare che questa dimostrazione funziona solo con segnali periodici.

9.6 Integrazione del rumore

Nella Sezione 4.3, abbiamo generato il rumore Browniano calcolando la somma cumulativa del rumore bianco. Ora che comprendiamo l'effetto di `cumsum` nel dominio della frequenza, abbiamo una panoramica dello spettro del rumore Browniano.

Il rumore bianco ha la stessa potenza a tutte le frequenze, in media. Quando calcoliamo la somma cumulativa, l'ampiezza di ogni componente viene divisa per f . Poiché la potenza è il quadrato dell'ampiezza, la potenza di ogni componente viene divisa per f^2 . Quindi, in media, la potenza alla frequenza f è proporzionale a $1/f^2$:

$$P_f = K/f^2$$

dove K è una costante non importante. Prendendo il logaritmo di entrambi i lati si ottiene:

$$\log P_f = \log K - 2 \log f$$

Ed è per questo che, quando si disegna lo spettro del rumore Browniano su una scala log-log, ci si aspetta di vedere una linea retta con pendenza -2, almeno approssimativamente.

Nella Sezione 9.1 abbiamo disegnato lo spettro dei prezzi di chiusura per Facebook e abbiamo stimato che la pendenza è -1.9, che è coerente con il rumore Browniano. Molti prezzi di azioni hanno spettri simili.

Quando usiamo l'operatore `diff` per calcolare le variazioni giornaliere, abbiamo moltiplicato l'*ampiezza* di ogni componente per un filtro proporzionale a f , il che significa che abbiamo moltiplicato la *potenza* di ciascuna componente per f^2 . Su una scala log-log, questa operazione aggiunge 2 alla pendenza dello spettro di potenza, motivo per cui la pendenza stimata del risultato è vicino a 0.1 (ma leggermente inferiore, perché `diff` approssima solo la differenziazione).

9.7 Esercizi

Il notebook per questo capitolo è `chap09.ipynb`. Leggerlo ed eseguire il codice.

Le soluzioni a questi esercizi sono in `chap09soln.ipynb`.

Esercizio 9.1 L'obiettivo di questo esercizio è esplorare l'effetto di `diff` e `differentiate` su un segnale. Creare un'onda triangolare e disegnarla. Applicare `diff` e disegnare il risultato. Calcolare lo spettro dell'onda triangolare, applicare `differentiate`, e disegnare il risultato. Convertire lo spettro di nuovo in un'onda e disegnarla. Ci sono differenze tra l'effetto di `diff` e `differentiate` per quest'onda?

Esercizio 9.2 L'obiettivo di questo esercizio è esplorare l'effetto di `cumsum` e `integrate` su un segnale. Creare un'onda quadra e disegnarla. Applicare `cumsum` e disegnare il risultato. Calcolare lo spettro dell'onda quadra, applicare `integrate`, e disegnare il risultato. Convertire lo spettro di nuovo in un'onda e disegnarla. Ci sono differenze tra l'effetto di `cumsum` e di `integrate` per quest'onda?

Esercizio 9.3 L'obiettivo di questo esercizio è esplorare l'effetto dell'integrazione doppia. Creare un'onda a dente di sega, calcolarne lo spettro, poi applicare `integrate` due volte. Disegnare l'onda risultante e il suo spettro. Qual è la forma matematica dell'onda? Perché assomiglia a una sinusoida?

Esercizio 9.4 L'obiettivo di questo esercizio è esplorare l'effetto della differenza seconda e della derivata seconda. Creare un `CubicSignal`, definito in `thinkdsp`. Calcolare la differenza seconda applicando due volte `diff`. Che aspetto ha il risultato? Calcola la derivata seconda applicando due volte `differentiate` allo spettro. Il risultato è lo stesso?

Disegnare i filtri che corrispondono alla differenza seconda e alla derivata seconda e confrontali. Suggerimento: per ottenere i filtri sulla stessa scala, utilizzare un'onda con framerate 1.

Capitolo 10

Sistemi Dinamici Lineari Stazionari

Questo capitolo presenta la teoria dei segnali e dei sistemi, utilizzando l'acustica musicale come esempio. Spiega un'importante applicazione del Teorema della Convoluzione, caratterizzazione dei sistemi LTI [Linear, Time-Invariant = Lineari Stazionari] (che definiremo a breve).

Il codice per questo capitolo è in `chap10.ipynb`, nel repository di questo libro (vedere la Sezione 0.2). Lo si può anche visualizzare su <http://tinyurl.com/thinkdsp10>.

10.1 Segnali e sistemi

Nel contesto dell'elaborazione del segnale, un **sistema** è una rappresentazione astratta di tutto ciò che accetta un segnale come input e produce un segnale come output.

Ad esempio, un amplificatore elettronico è un circuito che prende un segnale elettrico come ingresso e produce un segnale (più forte) come uscita.

Come altro esempio, quando si ascolta una performance musicale, si può pensare alla stanza come a un sistema che prende il suono della performance nel punto in cui viene generato e produce un suono leggermente diverso nel luogo in cui lo si sente.

Un **sistema dinamico lineare stazionario**¹ è un sistema con queste due proprietà:

¹La presentazione qui segue https://it.wikipedia.org/wiki/Sistema_dinamico_lineare_stazionario.

1. Linearità: se si inseriscono due input nel sistema contemporaneamente, il risultato è la somma dei loro output. Matematicamente, se un input x_1 produce un output y_1 e un altro input x_2 produce y_2 , allora $ax_1 + bx_2$ produce $ay_1 + by_2$, dove a e b sono scalari.
2. Invarianza temporale: l'effetto del sistema non varia nel tempo o dipende dallo stato del sistema. Quindi, se gli input x_1 e x_2 differiscono di uno spostamento nel tempo, i loro output y_1 e y_2 differiscono per lo stesso spostamento, ma per il resto sono identici.

Molti sistemi fisici hanno queste proprietà, almeno approssimativamente.

- I circuiti che contengono solo resistori, condensatori e induttori sono LTI, nella misura in cui le componenti si comportano come i loro modelli idealizzati.
- Anche i sistemi meccanici che contengono molle, masse e ammortizzatori sono LTI, assumendo molle lineari (forza proporzionale allo spostamento) e ammortizzatori (forza proporzionale alla velocità).
- Inoltre, e soprattutto per le applicazioni di questo libro, i media che trasmettono il suono (inclusi aria, acqua e solidi) sono ben modellati dai sistemi LTI.

I sistemi LTI sono descritti da equazioni differenziali lineari e le soluzioni di queste equazioni sono sinusoidi complesse (vedere https://it.wikipedia.org/wiki/Equazione_differenziale_lineare).

Questo risultato fornisce un algoritmo per calcolare l'effetto di un sistema LTI su un segnale di ingresso:

1. Esprimere il segnale come somma di componenti sinusoidali complesse.
2. Per ogni componente di input, calcolare la componente di output corrispondente.
3. Sommare le componenti di output.

A questo punto, si spera che tale algoritmo suoni familiare. È lo stesso algoritmo usato per la convoluzione nella Sezione 8.6, e per la differenziazione nella Sezione 9.3. Questo processo è chiamato **decomposizione spettrale** perché “decomponiamo” il segnale in ingresso nelle sue componenti spettrali.

Per applicare questo processo a un sistema LTI, dobbiamo **caratterizzare** il sistema trovando il suo effetto su ciascuna componente del segnale in ingresso.

Per i sistemi meccanici, si scopre che esiste un modo semplice ed efficiente per farlo: dargli un calcio e registrare l'output.

Tecnicamente, il “calcio” è chiamato **impulso** e l'output è detto **risposta all'impulso**. Ci si potrebbe chiedere come un singolo impulso possa caratterizzare completamente un sistema. Si può vedere la risposta calcolando la DFT di un impulso. Ecco un array di onde con un impulso a $t = 0$:

```
impulse = np.zeros(8)
impulse[0] = 1
impulse_spectrum = np.fft.fft(impulse)
```

Ecco l'array di onde:

```
[ 1.  0.   0.   0.   0.   0.   0.]
```

Ed ecco il suo spettro:

```
[ 1.+0.j  1.+0.j  1.+0.j  1.+0.j  1.+0.j  1.+0.j  1.+0.j  1.+0.j]
```

Lo spettro sono tutti uno; ovvero, un impulso è la somma di componenti di uguale grandezza a tutte le frequenze. Questo spettro non deve essere confuso con il rumore bianco, che ha la stessa potenza *media* a tutte le frequenze, ma varia intorno a quella media.

Quando si testa un sistema immettendo un impulso, si sta verificando la risposta del sistema a tutte le frequenze. E si possono testare tutti allo stesso tempo perché il sistema è lineare, quindi i test simultanei non interferiscono tra loro.

10.2 Finestre e filtri

Per mostrare perché questo tipo di caratterizzazione del sistema funziona, inizieremo con un semplice esempio: una media mobile a 2 elementi. Possiamo pensare a questa operazione come a un sistema che prende un segnale come ingresso e produce un segnale leggermente più fluido come uscita.

In questo esempio sappiamo qual è la finestra, quindi possiamo calcolare il filtro corrispondente. Ma di solito non è così; nella prossima sezione vedremo un esempio in cui non conosciamo in anticipo la finestra o il filtro.

Ecco una finestra che calcola una media mobile a 2 elementi (vedi Sezione 8.1):

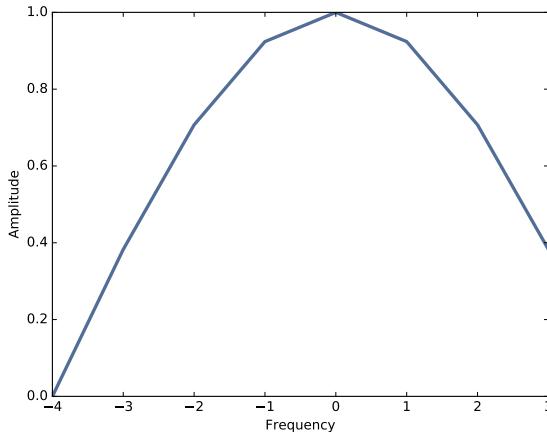


Figura 10.1: DFT di una finestra di media mobile a 2 elementi.

```
window_array = np.array([0.5, 0.5, 0, 0, 0, 0, 0, 0,])
window = thinkdsp.Wave(window_array, framerate=8)
```

Possiamo trovare il filtro corrispondente calcolando la DFT della finestra:

```
filtr = window.make_spectrum(full=True)
```

La Figura 10.1 mostra il risultato. Il filtro che corrisponde a una finestra della media mobile è un filtro passa-basso con la forma approssimativa di una curva Gaussiana.

Ora immaginiamo di non conoscere la finestra o il filtro corrispondente e di voler caratterizzare questo sistema. Lo faremmo inserendo un impulso e misurando la risposta all'impulso.

In questo esempio, possiamo calcolare la risposta all'impulso moltiplicando lo spettro dell'impulso e del filtro e quindi convertendo il risultato da uno spettro a un'onda:

```
product = impulse_spectrum * filtr
filtered = product.make_wave()
```

Poiché `impulse_spectrum` sono tutti uno, il prodotto è identico al filtro e l'onda filtrata è identica alla finestra.

Questo esempio dimostra due cose:

- Poiché lo spettro di un impulso sono tutti uno, la DFT della risposta all'impulso è identica al filtro che caratterizza il sistema.

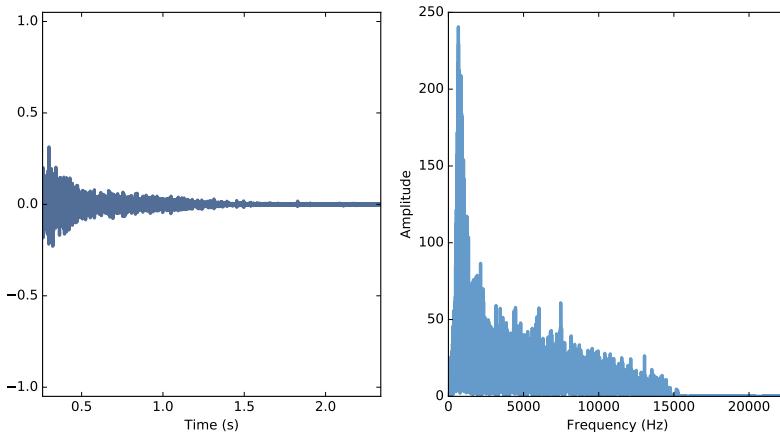


Figura 10.2: Forma d’onda di uno sparo.

- Pertanto, la risposta all’impulso è identica alla finestra di convoluzione che caratterizza il sistema.

10.3 Risposta acustica

Per caratterizzare la risposta acustica di una stanza o di uno spazio aperto, un modo semplice per generare un impulso è far scoppiare un pallone o sparare con una pistola. Un colpo di pistola mette un impulso nel sistema; il suono che si sente è la risposta all’impulso.

Come esempio, useremo una registrazione di uno sparo per caratterizzare la stanza in cui è stata sparata la pistola, quindi useremo la risposta all’impulso per simulare l’effetto di quella stanza su una registrazione di violino.

Questo esempio è in `chap10.ipynb`, nel repository di questo libro; lo si può anche visualizzare e ascoltare gli esempi su <http://tinyurl.com/thinkdsp10>.

Ecco lo sparo:

```
response = thinkdsp.read_wave('180960_kleeb_gunshots.wav')
response = response.segment(start=0.26, duration=5.0)
response.normalize()
response.plot()
```

Si seleziona un segmento a partire da 0.26 secondi per rimuovere il silenzio prima dello sparo. La Figura 10.2 (a sinistra) mostra la forma d’onda dello sparo. Successivamente calcoliamo la DFT di `response`:

```
transfer = response.make_spectrum()
transfer.plot()
```

La Figura 10.2 (a destra) mostra il risultato. Questo spettro codifica la risposta della stanza; per ogni frequenza, lo spettro contiene un numero complesso che rappresenta un moltiplicatore di ampiezza e uno sfasamento. Questo spettro è chiamato **funzione di trasferimento** perché contiene informazioni su come il sistema trasferisce l'input all'output.

Ora possiamo simulare l'effetto che questa stanza avrebbe sul suono di un violino. Ecco la registrazione del violino che abbiamo usato nella Sezione 1.1:

```
violin = thinkdsp.read_wave('92002__jcveliz__violin-original.wav')
violin.truncate(len(response))
violin.normalize()
```

Le onde del violino e degli spari sono state campionate allo stesso framerate, 44,100 Hz. E guarda caso, la durata di entrambi è più o meno la stessa. Abbiamo tagliato l'onda del violino alla stessa lunghezza del colpo di pistola.

Successivamente si calcola la DFT dell'onda del violino:

```
spectrum = violin.make_spectrum()
```

Ora conosciamo l'ampiezza e la fase di ogni componente in ingresso e conosciamo la funzione di trasferimento del sistema. Il loro prodotto è la DFT dell'output, che possiamo usare per calcolare l'onda di output:

```
output = (spectrum * transfer).make_wave()
output.normalize()
output.plot()
```

La Figura 10.3 mostra l'ingresso (in alto) e l'output (in basso) del sistema. Sono sostanzialmente diversi e le differenze sono chiaramente udibili. Caricare `chap10.ipynb` e ascoltarli. Una cosa sorprendente di questo esempio è che si può avere un'idea di come fosse la stanza; a me suona come una stanza lunga e stretta con pavimenti e soffitti duri. Cioè, come un poligono di tiro.

C'è una cosa che abbiamo sorvolato in questo esempio che verrà menzionato nel caso in cui dia fastidio a qualcuno. La registrazione del violino con cui si è iniziato è già stata trasformata da un sistema: la stanza in cui è stata registrata. Quindi quello che si è davvero calcolato nell'esempio è il suono del violino dopo due trasformazioni. Per simulare correttamente il suono di un violino in una stanza diversa, avremmo dovuto caratterizzare la stanza in cui

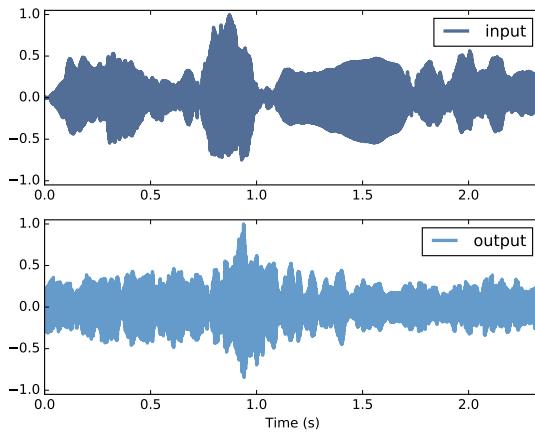


Figura 10.3: La forma d’onda della registrazione del violino prima e dopo la convoluzione.

il violino è stato registrato e applicare prima l’inverso di quella funzione di trasferimento.

10.4 Sistemi e convoluzione

Se pensate che l’esempio precedente sia magia nera, non siete i soli. Ci ho riflettuto per un po’ e mi fa ancora male la testa.

Nella sezione precedente, abbiamo suggerito un modo di pensarci:

- Un impulso è costituito da componenti con ampiezza 1 a tutte le frequenze.
- La risposta all’impulso contiene la somma delle risposte del sistema a tutte queste componenti.
- La funzione di trasferimento, che è la DFT della risposta all’impulso, codifica l’effetto del sistema su ciascuna componente di frequenza sotto forma di un moltiplicatore di ampiezza e uno sfasamento.
- Per qualsiasi input, possiamo calcolare la risposta del sistema suddividendo l’input in componenti, calcolando la risposta a ciascuna componente e sommandole.

Ma se non piace, c’è un altro modo di pensare al tutto: la convoluzione! Secondo il Teorema della Convoluzione, la moltiplicazione nel dominio della

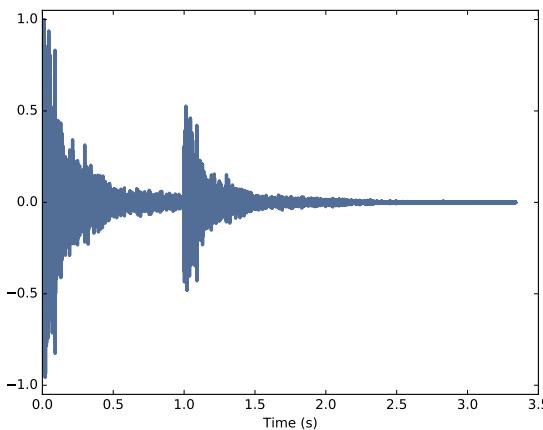


Figura 10.4: Somma di un'onda e una copia spostata e scalata.

frequenza corrisponde alla convoluzione nel dominio del tempo. In questo esempio, l'output del sistema è la convoluzione dell'input e la risposta del sistema.

Ecco le chiavi per capire perché funziona:

- Si può pensare ai campioni nell'onda di ingresso come a una sequenza di impulsi con ampiezza variabile.
- Ogni impulso nell'ingresso produce una copia della risposta all'impulso, spostata nel tempo (perché il sistema è invariante nel tempo) e scalata dall'ampiezza dell'ingresso.
- L'output è la somma delle copie spostate e ridimensionate della risposta all'impulso. Le copie si sommano perché il sistema è lineare.

Procediamo gradualmente: supponiamo che invece di sparare un colpo di pistola, ne spariamo due: uno grande con ampiezza 1 a $t = 0$ e uno più piccolo con ampiezza 0.5 a $t = 1$.

Possiamo calcolare la risposta del sistema sommando la risposta all'impulso originale con una copia in scala e spostata di se stesso. Ecco una funzione che crea una copia spostata e scalata di un'onda:

```
def shifted_scaled(wave, shift, factor):
    res = wave.copy()
    res.shift(shift)
    res.scale(factor)
    return res
```

Il parametro `shift` è uno spostamento temporale in secondi; `factor` è un fattore moltiplicativo.

Ecco come lo usiamo per calcolare la risposta a due spari:

```
shift = 1
factor = 0.5
gun2 = response + shifted_scaled(response, shift, factor)
```

La Figura 10.4 mostra il risultato. Lo si può sentire in `chap10.ipynb`. Non sorprende che suona come due colpi di pistola, il primo più forte del secondo.

Supponiamo ora che invece di due pistole, si sommino 100 pistole che sparano a una velocità di 441 colpi al secondo. Questo ciclo calcola il risultato:

```
dt = 1 / 441
total = 0
for k in range(100):
    total += shifted_scaled(response, k*dt, 1.0)
```

Con 441 colpi al secondo, quindi non si sentono i singoli colpi. Invece, suona come un segnale periodico a 441 Hz. Questo esempio, suona come un clacson in un garage.

E questo ci porta a una visione chiave: si può pensare a qualsiasi onda come una serie di campioni, in cui ogni campione è un impulso con un'ampiezza diversa.

Come esempio, genereremo un segnale a dente di sega a 441 Hz:

```
signal = thinkdsp.SawtoothSignal(freq=441)
wave = signal.make_wave(duration=0.1,
                         framerate=response.framerate)
```

Ora cicleremo attraverso la serie di impulsi che compongono il dente di sega e sommeremo le risposte all'impulso:

```
total = 0
for t, y in zip(wave.ts, wave.ys):
    total += shifted_scaled(response, t, y)
```

Il risultato è quello che suonerebbe come un'onda a dente di sega in un poligono di tiro. Anch'esso lo si può ascoltare in `chap10.ipynb`.

La Figura 10.5 mostra un diagramma di questo calcolo, dove f è il dente di sega, g è la risposta all'impulso e h è la somma delle copie spostate e scalate di g .

$$\begin{array}{r}
 \begin{array}{c} f[0] \quad [\quad g[0] \quad g[1] \quad g[2] \quad \dots \quad] \\ f[1] \quad [\quad \quad \quad g[0] \quad g[1] \quad g[2] \quad \dots \quad] \\ f[2] \quad [\quad \quad \quad \quad g[0] \quad g[1] \quad g[2] \quad \dots \quad] \\ \hline \end{array} \\
 [\quad \quad \quad h[2] \quad \quad \quad]
 \end{array}$$

Figura 10.5: Diagramma della somma delle copie ridimensionate e spostate di g .

Per l'esempio mostrato:

$$h[2] = f[0]g[2] + f[1]g[1] + f[2]g[0]$$

E più in generale,

$$h[n] = \sum_{m=0}^{N-1} f[m]g[n-m]$$

Si potrebbe riconoscere questa equazione dalla Sezione 8.2. È la convoluzione di f e g . Ciò mostra che se l'input è f e la risposta all'impulso del sistema è g , l'output è la convoluzione di f e g .

In sintesi, ci sono due modi per pensare all'effetto di un sistema su un segnale:

1. L'input è una sequenza di impulsi, quindi l'output è la somma delle copie scalate e spostate della risposta all'impulso; quella somma è la convoluzione dell'input e della risposta all'impulso.
2. La DFT della risposta all'impulso è una funzione di trasferimento che codifica l'effetto del sistema su ciascuna componente di frequenza come ampiezza e sfasamento di fase. La DFT dell'ingresso codifica l'ampiezza e l'offset di fase delle componenti di frequenza che contiene. Moltiplicando la DFT dell'input per la funzione di trasferimento si ottiene la DFT dell'output.

L'equivalenza di queste descrizioni non dovrebbe essere una sorpresa; è fondamentalmente un'affermazione del Teorema della Convoluzione: la convoluzione di f e g nel dominio del tempo corrisponde alla moltiplicazione nel dominio della frequenza. E se ci si chiedeva perché la convoluzione sia definita così, che sembrava al contrario quando si parlava di smoothing e finestre di differenza, ora se ne conosce il motivo: la definizione di convoluzione appare naturalmente nella risposta di un sistema LTI a un segnale.

10.5 Dimostrazione del Teorema della Convoluzione

Beh, è stata rimandata abbastanza a lungo. È tempo di dimostrare il Teorema della Convoluzione (CT), che afferma:

$$\text{DFT}(f * g) = \text{DFT}(f)\text{DFT}(g)$$

dove f e g sono vettori con la stessa lunghezza, N .

Si procede in due passaggi:

1. Mostreremo che nel caso speciale in cui f è un esponenziale complesso, la convoluzione con g ha l'effetto di moltiplicare f per uno scalare.
2. Nel caso più generale in cui f non è un esponenziale complesso, possiamo usare la DFT per esprimere come somma di componenti esponenziali, calcolare la convoluzione di ogni componente (per moltiplicazione) e quindi sommare i risultati.

Insieme, questi passaggi dimostrano il Teorema della Convoluzione. Ma prima, assembliamo i pezzi di cui avremo bisogno. La DFT di g , che chiameremo G è:

$$\text{DFT}(g)[k] = G[k] = \sum_n g[n] \exp(-2\pi i nk/N)$$

dove k è un indice di frequenza da 0 a $N - 1$ e n è un indice di tempo da 0 a $N - 1$. Il risultato è un vettore di N numeri complessi.

La DFT inversa di F , che chiameremo f , è:

$$\text{IDFT}(F)[n] = f[n] = \sum_k F[k] \exp(2\pi i nk/N)$$

Ecco la definizione di convoluzione:

$$(f * g)[n] = \sum_m f[m]g[n - m]$$

dove m è un altro indice di tempo compreso tra 0 e $N - 1$. La convoluzione è commutativa, quindi potremmo scrivere in modo equivalente:

$$(f * g)[n] = \sum_m f[n - m]g[m]$$

Consideriamo ora il caso speciale in cui f è un esponenziale complesso con frequenza k , che chiameremo e_k :

$$f[n] = e_k[n] = \exp(2\pi i n k / N)$$

dove k è un indice di frequenza e n è un indice di tempo.

Inserendo e_k nella seconda definizione di convoluzione si ottiene

$$(e_k * g)[n] = \sum_m \exp(2\pi i(n - m)k / N)g[m]$$

Possiamo suddividere il primo termine in un prodotto:

$$(e_k * g)[n] = \sum_m \exp(2\pi i n k / N) \exp(-2\pi i m k / N)g[m]$$

La prima metà non dipende da m , quindi possiamo estrarla dalla somma:

$$(e_k * g)[n] = \exp(2\pi i n k / N) \sum_m \exp(-2\pi i m k / N)g[m]$$

Ora riconosciamo che il primo termine è e_k , e la somma è $G[k]$ (usando m come indice del tempo). Quindi possiamo scrivere:

$$(e_k * g)[n] = e_k[n]G[k]$$

il che mostra che per ogni esponenziale complesso, e_k , la convoluzione con g ha l'effetto di moltiplicare e_k per $G[k]$. In termini matematici, ogni e_k è un autovettore di questa operazione e $G[k]$ è l'autovalore corrispondente (vedere la Sezione 9.3).

Ora per la seconda parte della dimostrazione. Se il segnale in ingresso, f , non è un esponenziale complesso, possiamo esprimere come somma di esponenziali complessi calcolando la sua DFT, F . Per ogni valore di k da 0 a $N - 1$, $F[k]$ è la grandezza complessa della componente con frequenza k .

Ogni componente di input è un esponenziale complesso con magnitudine $F[k]$, quindi ogni componente di output è un esponenziale complesso con magnitudine $F[k]G[k]$, basato sulla prima parte della dimostrazione.

Poiché il sistema è lineare, l'output è solo la somma delle componenti di output:

$$(f * g)[n] = \sum_k F[k]G[k]e_k[n]$$

Inserendo la definizione di e_k si ottiene

$$(f * g)[n] = \sum_k F[k]G[k] \exp(2\pi i nk/N)$$

Il lato destro è la DFT inversa del prodotto FG . Quindi:

$$(f * g) = \text{IDFT}(FG)$$

Sostituendo $F = \text{DFT}(f)$ e $G = \text{DFT}(g)$:

$$(f * g) = \text{IDFT}(\text{DFT}(f)\text{DFT}(g))$$

Infine, prendendo la DFT di entrambi i lati si ottiene il Teorema della Convoluzione:

$$\text{DFT}(f * g) = \text{DFT}(f)\text{DFT}(g)$$

CVD [Come volevasi dimostrare]

10.6 Esercizi

Le soluzioni a questi esercizi sono in `chap10soln.ipynb`.

Esercizio 10.1 Nella Sezione 10.4 si descrive la convoluzione come la somma di copie spostate e ridimensionate di un segnale. A rigor di termini, questa operazione è una convoluzione *lineare*, che non presuppone che il segnale sia periodico.

Ma nella Sezione 10.3, quando moltiplichiamo la DFT del segnale per la funzione di trasferimento, tale operazione corrisponde alla convoluzione *circolare*,

che assume che il segnale sia periodico. Di conseguenza, si potrebbe notare che l'output contiene una nota in più all'inizio, che si riavvolge dalla fine.

Fortunatamente, esiste una soluzione standard a questo problema. Se si aggiungono abbastanza zeri alla fine del segnale prima di calcolare la DFT, si può evitare l'arrotolamento [wrap-around] e calcolare una convoluzione lineare.

Modificare l'esempio in `chap10.ipynb` e confermare che il completamento con gli zeri elimina la nota in più all'inizio dell'output.

Esercizio 10.2 La libreria Open AIR fornisce una “risorsa on-line centralizzata per chiunque sia interessato all'auralizzazione e ai dati sulla risposta all'impulso acustico” (<http://www.openairlib.net>). Sfogliare la loro raccolta di dati sulla risposta all'impulso e scaricare quello che sembra interessante. Trovare una breve registrazione che abbia la stessa frequenza di campionamento della risposta all'impulso scaricata.

Simulare il suono della propria registrazione nello spazio in cui è stata misurata la risposta all'impulso, calcolato in due modi: con la convoluzione della registrazione con la risposta all'impulso e calcolando il filtro corrispondente alla risposta all'impulso e moltiplicando per la DFT della registrazione.

Capitolo 11

Modulazione e campionamento

Nella Sezione 2.3 abbiamo visto che quando un segnale viene campionato a 10,000 Hz, una componente a 5500 Hz è indistinguibile da una componente a 4500 Hz. In questo esempio, la frequenza di ripiegamento, 5000 Hz, è la metà della frequenza di campionamento. Ma non abbiamo spiegato perché.

Questo capitolo esplora l'effetto del campionamento e presenta il Teorema del Campionamento, che spiega l'aliasing e la frequenza di ripiegamento.

Inizieremo esplorando l'effetto della convoluzione con gli impulsi; lo useremo poi per spiegare la modulazione di ampiezza (AM), che risulta essere utile per comprendere il Teorema del Campionamento.

Il codice per questo capitolo si trova in `chap11.ipynb`, nel repository di questo libro (vedere la Sezione 0.2). Lo si può anche visualizzare su <http://tinyurl.com/thinkdsp11>.

11.1 Convoluzione con gli impulsi

Come abbiamo visto nella Sezione 10.4, la convoluzione di un segnale con una serie di impulsi ha l'effetto di creare copie spostate e ridimensionate del segnale.

Come esempio, leggeremo il segnale che suona come un beep:

```
filename = '253887__themusicalnomad__positive-beeps.wav'  
wave = thinkdsp.read_wave(filename)  
wave.normalize()
```

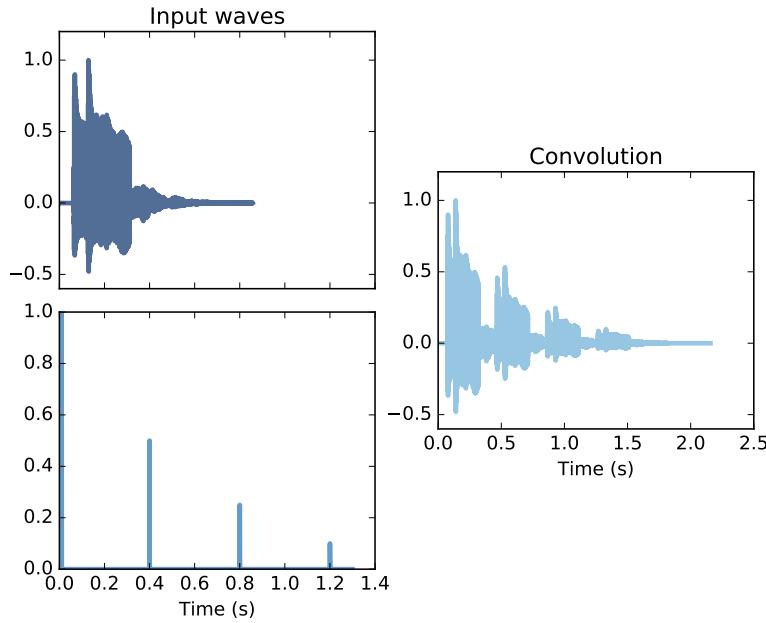


Figura 11.1: L'effetto della convoluzione di un segnale (in alto a sinistra) con una serie di impulsi (in basso a sinistra). Il risultato (a destra) è la somma delle copie spostate e ridimensionate del segnale.

E costruiremo un'onda con quattro impulsi:

```
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)
```

E poi effettueremo la convoluzione:

```
convolved = wave.convolve(impulses)
```

La Figura 11.1 mostra i risultati, con il segnale in alto a sinistra, gli impulsi in basso a sinistra e il risultato a destra.

Si può ascoltare il risultato in `chap11.ipynb`; suona come una serie di quattro segnali acustici con volume decrescente.

Lo scopo di questo esempio è solo dimostrare che la convoluzione con impulsi produce copie spostate e ridimensionate. Questo risultato sarà utile nella prossima sezione.

11.2 Modulazione d'ampiezza

La modulazione di ampiezza (AM) viene utilizzata per le trasmissioni radio AM, tra le altre applicazioni. Al trasmettitore, il segnale (che potrebbe contenere parlato, musica, ecc.) viene “modulato” moltiplicandolo per un segnale coseno che funge da “onda portante”. Il risultato è un’onda ad alta frequenza adatta per la trasmissione radio. Le frequenze tipiche per la radio AM negli Stati Uniti sono 500–1600 kHz (vedere https://en.wikipedia.org/wiki/AM_broadcasting).

Dal lato del ricevitore, il segnale trasmesso viene “demodulato” per recuperare il segnale originale. Sorprendentemente, la demodulazione funziona moltiplicando il segnale trasmesso, ancora una volta, per la stessa onda portante.

Per vedere come funziona, moduleremo un segnale con un’onda portante a 10 kHz. Ecco il segnale:

```
filename = '105977__wcfl10_favorite-station.wav'  
wave = thinkdsp.read_wave(filename)  
wave.unbias()  
wave.normalize()
```

Ed ecco la portante:

```
carrier_sig = thinkdsp.CosSignal(freq=10000)  
carrier_wave = carrier_sig.make_wave(duration=wave.duration,  
framerate=wave.framerate)
```

Possiamo moltiplicarli usando l’operatore `*`, che moltiplica gli array di onde per ciascun elemento:

```
modulated = wave * carrier_wave
```

Il risultato suona piuttosto male. Lo si può ascoltare in `chap11.ipynb`.

La Figura 11.2 mostra cosa sta succedendo nel dominio della frequenza. La riga in alto è lo spettro del segnale originale. La riga successiva è lo spettro del segnale modulato, dopo averlo moltiplicato per la portante. Contiene due copie dello spettro originale, spostato di più e meno 10 kHz.

Per capire perché, ricordarsi che la convoluzione nel dominio del tempo corrisponde alla moltiplicazione nel dominio della frequenza. Al contrario, la moltiplicazione nel dominio del tempo corrisponde alla convoluzione nel dominio della frequenza. Quando moltiplichiamo il segnale per la portante, stiamo effettuando la convoluzione del suo spettro con la DFT della portante.

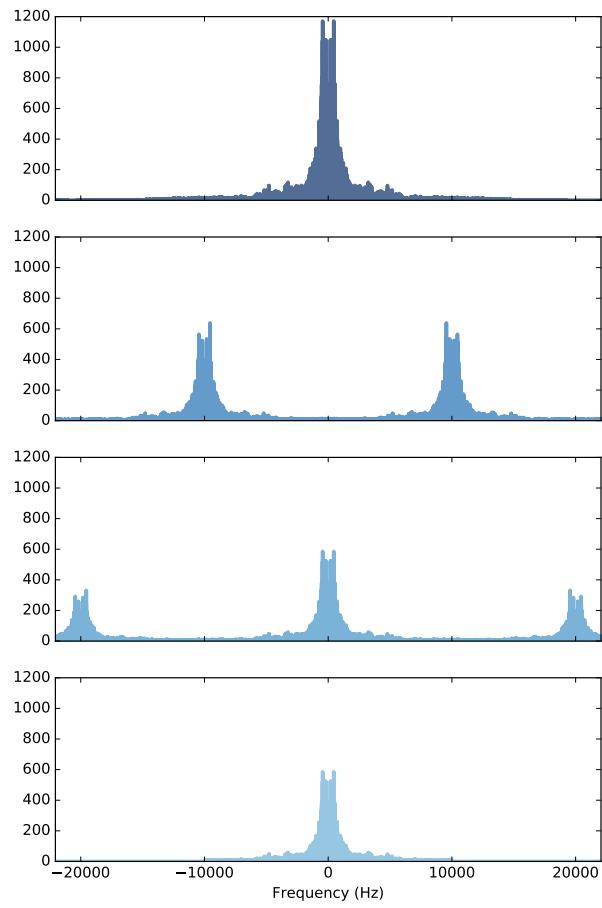


Figura 11.2: Dimostrazione della modulazione di ampiezza. La riga in alto è lo spettro del segnale; la riga successiva è lo spettro dopo la modulazione; la riga dopo è lo spettro dopo la demodulazione; l'ultima riga è il segnale demodulato dopo il filtraggio passa-basso.

Poiché la portante è una semplice onda coseno, la sua DFT è di due impulsi, a più e meno 10 kHz. La convoluzione con questi impulsi crea copie spostate e ridimensionate dello spettro. Notare che l'ampiezza dello spettro è minore dopo la modulazione. L'energia del segnale originale viene suddivisa tra le copie.

Demoduliamo il segnale, moltiplicandolo nuovamente per l'onda portante:

```
demodulated = modulated * carrier_wave
```

La terza riga della Figura 11.2 mostra il risultato. Ancora una volta, la moltiplicazione nel dominio del tempo corrisponde alla convoluzione nel dominio della frequenza, che produce copie spostate e ridimensionate dello spettro.

Poiché lo spettro modulato contiene due picchi, ogni picco viene diviso a metà e spostato di più e meno 20 kHz. Due delle copie si incontrano a 0 kHz e vengono sommate; le altre due copie finiscono centrate a più e meno 20 kHz.

Se si ascolta il segnale demodulato, suona abbastanza bene. Le copie extra dello spettro aggiungono componenti ad alta frequenza che non stavano nel segnale originale, ma sono così alte che gli altoparlanti probabilmente non possono riprodurlle e la maggior parte delle persone non le sente. Ma con buoni altoparlanti e buone orecchie, si potrebbe.

In tal caso, le componenti aggiuntive si possono eliminare applicando un filtro passa-basso:

```
demodulated_spectrum = demodulated.make_spectrum(full=True)
demodulated_spectrum.low_pass(10000)
filtered = demodulated_spectrum.make_wave()
```

Il risultato è abbastanza vicino all'onda originale, sebbene circa la metà della potenza venga persa dopo la demodulazione e il filtraggio. Ma questo non è un problema in pratica, perché molta più potenza viene persa durante la trasmissione e la ricezione del segnale di trasmissione. Dobbiamo comunque amplificare il risultato, un altro fattore 2 non è un problema.

11.3 Campionamento

Abbiamo spiegato la modulazione di ampiezza in parte perché è interessante, ma soprattutto perché ci aiuterà a capire il campionamento. Il “campionamento” è il processo di misura di un segnale analogico in una serie di punti nel tempo, solitamente con spaziatura uguale.

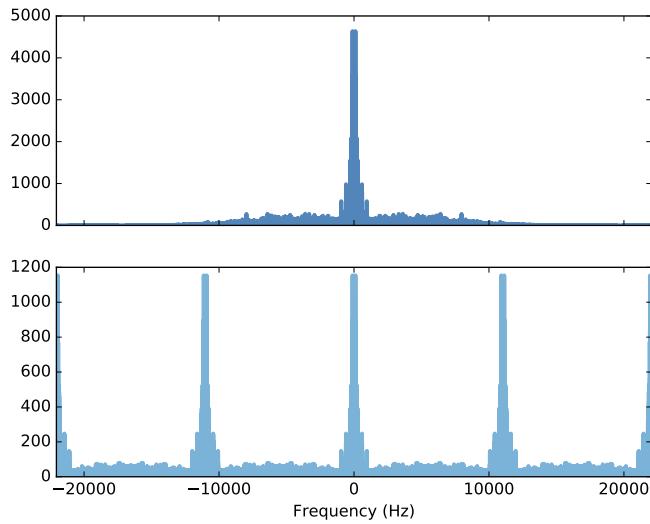


Figura 11.3: Spettro di un segnale prima (in alto) e dopo (in basso) il campionamento.

Ad esempio, i file WAV utilizzati come esempi sono stati registrati campionando l'uscita di un microfono utilizzando un convertitore analogico-digitale (ADC). La frequenza di campionamento per la maggior parte di essi è 44.1 kHz, che è la frequenza standard per il suono di “qualità CD”, o a 48 kHz, che è lo standard per il suono DVD.

A 48 kHz, la frequenza di ripiegamento è di 24 kHz, che è più alta di quanto la maggior parte delle persone possa sentire (vedere https://en.wikipedia.org/wiki/Hearing_range).

Nella maggior parte di queste onde, ogni campione ha 16 bit, quindi ci sono 2^{16} livelli distinti. Questa “profondità di bit” risulta essere sufficiente e l'aggiunta di più bit non migliora notevolmente la qualità del suono (vedere https://en.wikipedia.org/wiki/Digital_audio).

Ovviamente, applicazioni diverse dai segnali audio potrebbero richiedere velocità di campionamento più elevate, al fine di catturare frequenze più alte, o una maggiore profondità di bit, al fine di riprodurre le forme d'onda con maggiore fedeltà.

Per dimostrare l'effetto del processo di campionamento, inizieremo con un'onda campionata a 44.1 kHz e selezioneremo i campioni da essa a circa 11 kHz. Non è esattamente la stessa cosa del campionamento da un segnale analogico, ma l'effetto è lo stesso.

Per prima cosa caricheremo una registrazione di un assolo di batteria:

```
filename = '263868_kevicio_amen-break-a-160-bpm.wav'
wave = thinkdsp.read_wave(filename)
wave.normalize()
```

La Figura 11.3 (in alto) mostra lo spettro di quest'onda. Ora ecco la funzione che campiona dall'onda:

```
def sample(wave, factor=4):
    ys = np.zeros(len(wave))
    ys[::factor] = wave.ys[::factor]
    return thinkdsp.Wave(ys, framerate=wave.framerate)
```

La useremo per selezionare ogni quarto elemento:

```
sampled = sample(wave, 4)
```

Il risultato ha lo stesso framerate dell'originale, ma la maggior parte degli elementi è zero. Se si suona l'onda campionata, il suono non è molto buono. Il processo di campionamento introduce componenti ad alta frequenza che non c'erano nell'originale.

La Figura 11.3 (in basso) mostra lo spettro dell'onda campionata. Contiene quattro copie dello spettro originale (sembrano cinque copie perché una è divisa tra le frequenze più alte e più basse).

Per capire da dove provengono queste copie, possiamo pensare al processo di campionamento come a una moltiplicazione con una serie di impulsi. Invece di usare `sample` per selezionare ogni quarto elemento, potremmo usare questa funzione per creare una serie di impulsi, a volte chiamati **treno di impulsi**:

```
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)
```

E poi moltiplicare l'onda originale per il treno di impulsi:

```
impulses = make_impulses(wave, 4)
sampled = wave * impulses
```

Il risultato è lo stesso; non suona ancora molto bene, ma ora capiamo perché. La moltiplicazione nel dominio del tempo corrisponde alla convoluzione nel

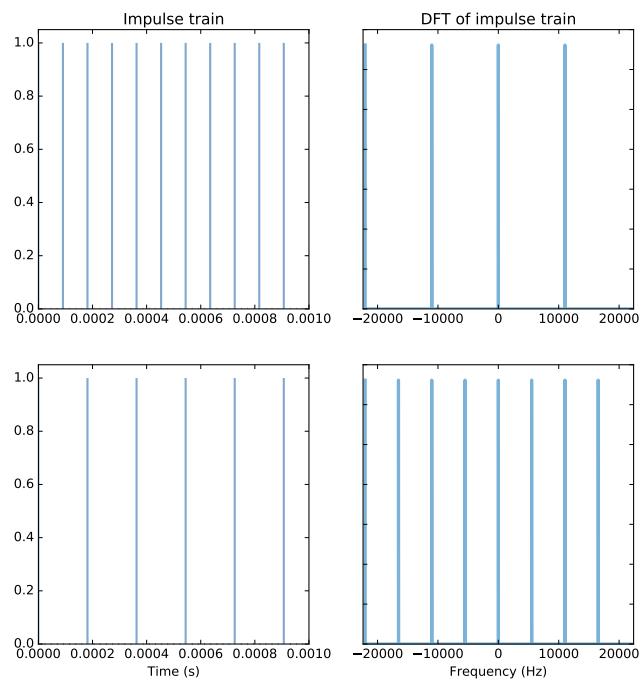


Figura 11.4: La DFT di un treno di impulsi è anche un treno di impulsi.

dominio della frequenza. Quando moltiplichiamo per un treno di impulsi, stiamo convolvendo con la DFT di un treno di impulsi. A quanto pare, anche la DFT di un treno di impulsi è un treno di impulsi.

La Figura 11.4 mostra due esempi. La riga superiore è il treno di impulsi nell'esempio, con frequenza 11,025 Hz. La DFT è un treno di 4 impulsi, motivo per cui otteniamo 4 copie dello spettro. La riga inferiore mostra un treno di impulsi con una frequenza inferiore, circa 5512 Hz. La sua DFT è un treno di 8 impulsi. In generale, più impulsi nel dominio del tempo corrispondono a meno impulsi nel dominio della frequenza.

In sintesi:

- Possiamo pensare al campionamento come a una moltiplicazione per un treno di impulsi.
- La moltiplicazione per un treno di impulsi corrisponde alla convoluzione con un treno di impulsi nel dominio della frequenza.
- La convoluzione con un treno di impulsi crea più copie dello spettro del segnale.

11.4 Aliasing

Nella Sezione 11.2, dopo aver demodulato un segnale AM abbiamo eliminato le copie extra dello spettro applicando un filtro passa-basso. Possiamo fare la stessa cosa dopo il campionamento, ma risulta non essere una soluzione perfetta.

La Figura 11.5 mostra perché no. La riga in alto è lo spettro dell'assolo di batteria. Contiene componenti ad alta frequenza che si estendono oltre i 10 kHz. Quando campioniamo quest'onda, convolgiamo lo spettro con il treno di impulsi (seconda riga), che crea copie dello spettro (terza riga). La riga inferiore mostra il risultato dopo l'applicazione di un filtro passa-basso con un taglio alla frequenza di ripiegatura, 5512 Hz.

Se riconvertiamo il risultato in un'onda, è simile all'onda originale, ma ci sono due problemi:

- A causa del filtro passa-basso, le componenti sopra i 5500 Hz sono state perse, quindi il risultato suona silenziato.

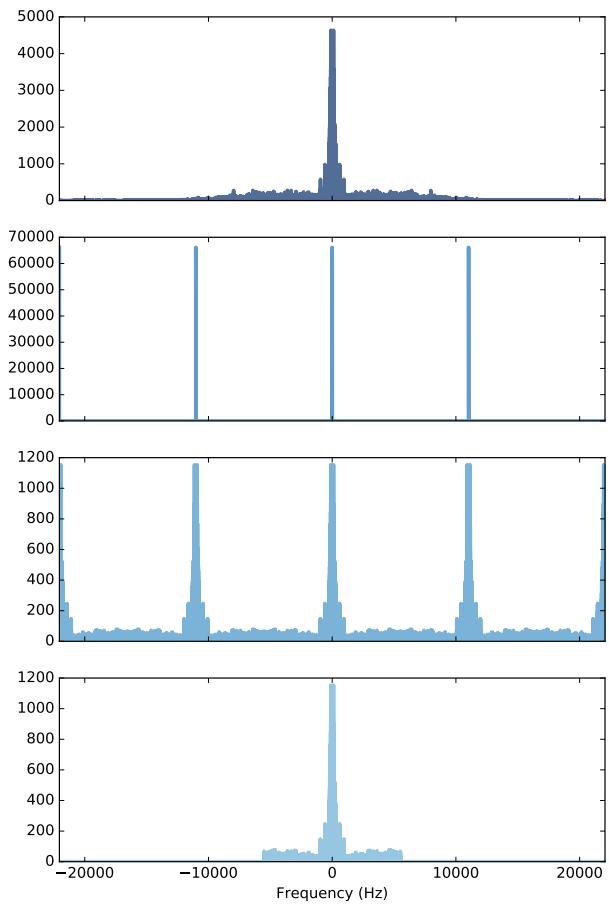


Figura 11.5: Spettro dell'assolo di batteria (in alto), spettro del treno di impulsi (seconda riga), spettro dell'onda campionata (terza riga), dopo il filtraggio passa-basso (in basso).

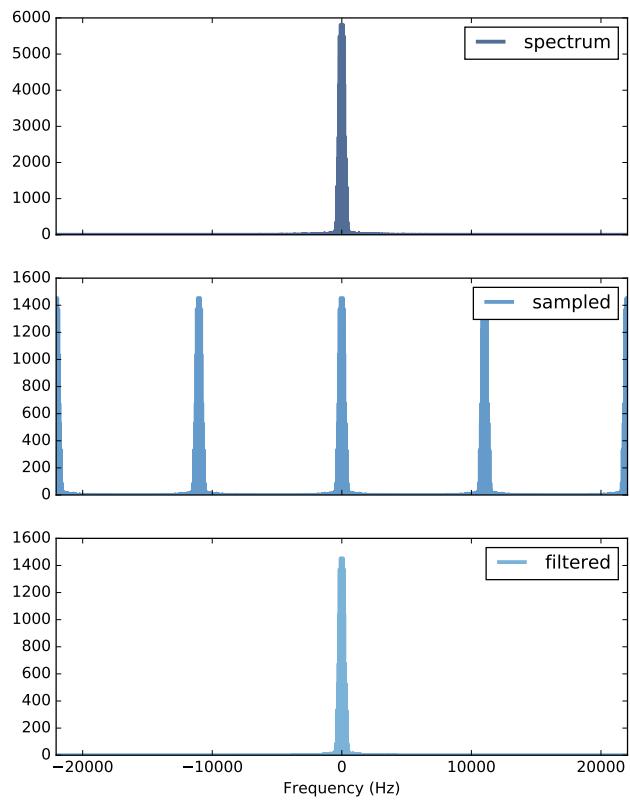


Figura 11.6: Spettro di un assolo di basso (in alto), il suo spettro dopo il campionamento (al centro) e dopo il filtraggio (in basso).

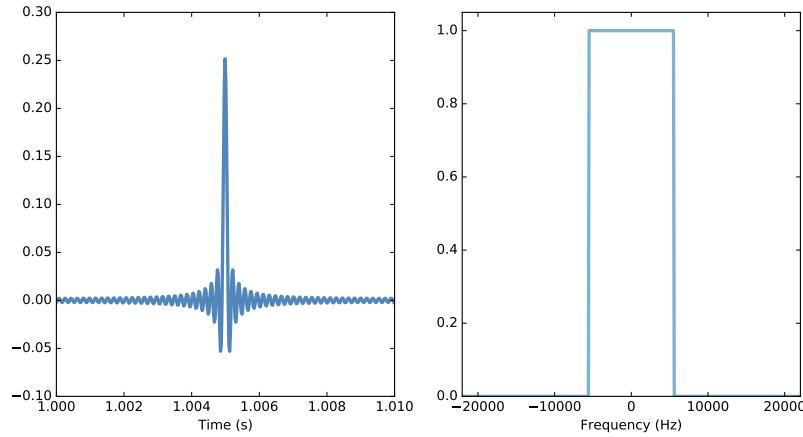


Figura 11.7: Un filtro passa-basso "muro di mattoni" (a destra) e la corrispondente finestra di convoluzione (a sinistra).

- Anche le componenti sotto i 5500 Hz non sono del tutto corrette, perché i contributi di inclusione provenienti dagli avanzi delle copie spettrali che abbiamo cercato di filtrare.

Se le copie spettrali si sovrappongono dopo il campionamento, perdiamo le informazioni sullo spettro e non saremo in grado di recuperarle.

Ma se le copie non si sovrappongono, le cose funzionano abbastanza bene. Come secondo esempio, carichiamo una registrazione di un assolo di basso.

La Figura 11.6 mostra il suo spettro (riga superiore), che non contiene energia visibile al di sopra di 5512 Hz. La seconda riga mostra lo spettro dell'onda campionata e la terza riga mostra lo spettro dopo il filtro passa basso. L'ampiezza è inferiore perché abbiamo filtrato parte dell'energia, ma la forma dello spettro è quasi esattamente quella con cui abbiamo iniziato. E se convertiamo di nuovo in un'onda, suona uguale.

11.5 Interpolazione

Il filtro passa basso usato nell'ultimo passaggio è un cosiddetto **brick wall filter** [filtro a muro di mattoni]; le frequenze al di sopra del limite vengono rimosse completamente, come se colpissero un muro di mattoni.

La Figura 11.7 (a destra) mostra l'aspetto di questo filtro. Ovviamente, la moltiplicazione per questo filtro, nel dominio della frequenza, corrisponde alla convoluzione con una finestra nel dominio del tempo. Possiamo scoprire cos'è

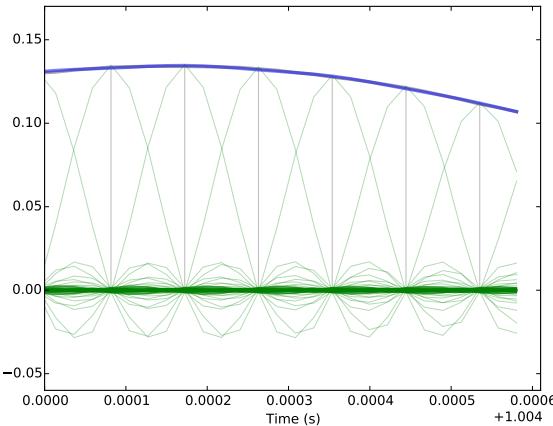


Figura 11.8: Esempio di interpolazione mediante l'aggiunta di copie spostate e ridimensionate di una funzione sinc.

quella finestra calcolando la DFT inversa del filtro, mostrata nella Figura 11.7 (a sinistra).

Quella funzione ha un nome; è la funzione sinc normalizzata, o almeno un'approssimazione discreta di essa (vedere https://en.wikipedia.org/wiki/Sinc_function):

$$\text{sinc}(x) = \frac{\sin \pi x}{\pi x}$$

Quando applichiamo il filtro passa-basso, stiamo convolvendo con una funzione sinc. Possiamo pensare a questa convoluzione come alla somma di copie spostate e ridimensionate della funzione sinc.

Il valore di sinc è 1 a 0 e 0 su ogni altro valore intero di x . Quando spostiamo la funzione sinc, spostiamo il punto zero. Quando lo scaliamo, cambiamo l'altezza al punto zero. Quindi, quando sommiamo le copie spostate e ridimensionate, queste interpolano tra i punti campionati.

La Figura 11.8 mostra come funziona utilizzando un breve segmento dell'assolo di basso. La linea nella parte superiore è l'onda originale. Le linee grigie verticali mostrano i valori campionati. Le curve sottili sono le copie spostate e in scala della funzione sinc. La somma di queste funzioni sinc è identica all'onda originale.

Leggere di nuovo l'ultima frase, perché è più sorprendente di quanto possa sembrare. Poiché abbiamo iniziato con un segnale che non conteneva energia

al di sopra di 5512 Hz, e abbiamo campionato a 11,025 Hz, siamo stati in grado di recuperare esattamente lo spettro originale.

In questo esempio, abbiamo iniziato con un'onda che era già stata campionata a 44,100 Hz, ed è stata ri-campionata a 11,025 Hz. Dopo il ricampionamento, il divario tra le copie spettrali è la frequenza di campionamento, 11.025 kHz. Se il segnale originale contiene componenti che superano la metà della frequenza di campionamento, 5512 Hz, le copie si sovrappongono e perdiamo informazioni.

Ma se il segnale è “a larghezza di banda limitata”; cioè non contiene energia al di sopra di 5512 Hz, le copie spettrali non si sovrappongono, non perdiamo informazioni e possiamo recuperare esattamente il segnale originale.

Questo risultato è noto come teorema del campionamento di Nyquist-Shannon (vedere https://en.wikipedia.org/wiki/Nyquist-Shannon_sampling_theorem).

Questo esempio non dimostra il Teorema del Campionamento, ma si spera che aiuti a capire cosa dice e perché funziona.

Si noti che l'argomento creato non dipende dalla frequenza di campionamento originale, 44.1 kHz. Il risultato sarebbe lo stesso se l'originale fosse stato campionato a una frequenza più alta, o anche se l'originale fosse stato un segnale analogico continuo: se campioniamo al framerate f , possiamo recuperare esattamente il segnale originale, purché non contenga energia a frequenze superiori a $f/2$.

11.6 Esercizi

Le soluzioni a questi esercizi sono in `chap11soln.ipynb`.

Esercizio 11.1 Il codice in questo capitolo è in `chap11.ipynb`. Leggere ed eseguire gli esempi.

Esercizio 11.2 Chris “Monty” Montgomery ha un ottimo video intitolato “D/A and A/D | Digital Show and Tell”; dimostra il teorema del campionamento in azione e presenta molte altre eccellenti informazioni sul campionamento. È visionabile su <https://www.youtube.com/watch?v=cIQ9IXSUzuM>.

Esercizio 11.3 Come abbiamo visto, se si campiona un segnale a un framerate troppo basso, le frequenze al di sopra della frequenza di piegatura [folding] producono un alias. Una volta che ciò accade, non è più possibile filtrare queste componenti, perché sono indistinguibili dalle frequenze più basse.

È una buona idea filtrare queste frequenze *prima* del campionamento; un filtro passa-basso utilizzato per questo scopo è detto **filtro anti-aliasing**.

Tornando all'esempio dell'assolo di batteria, applicare un filtro passa-basso prima del campionamento, poi applicare nuovamente il filtro passa-basso per rimuovere le copie spettrali introdotte dal campionamento. Il risultato dovrebbe essere identico al segnale filtrato.

