

PySDR українською: практичне SDR/DSP ядро

Українська RST-доріжка PySDR; TeX-конверсія

Сесія 09, 2026-06-01

Статус конверсії

Цей модуль є механічною XeLaTeX-конверсією вже наявних українських RST-розділів PySDR у машинно-читабельний TeX. Мета: отримати компільовану, редаговану TeX-основу для подальшого QA, термінологічного вирівнювання та локальних агентів. Формули, кодові ідентифікатори і технічні скорочення збережено настільки буквально, наскільки дозволяє автоматична RST-to-TeX конверсія.

Рисунки в цьому прохоті замінено контрольованими плейсхолдерами, щоб компіляція не залежала від повного дерева Sphinx/HTML-ресурсів. Для фінального видання треба або вбудувати оригінальні зображення PySDR, або замінити їх власними векторними схемами.

Зміст

1	Вступ	1
1.1	Мета та цільова аудиторія	1
1.2	Долучитися	2
1.3	Подяки	2
2	Вибірка IQ	4
2.1	Основи вибірки	4
2.2	Вибірка Найквіста	5
2.3	Квадратурна дискретизація	6
2.4	Комплексні числа	7
2.5	Комплексні числа у ШПФ	8
2.6	Сторона приймача	8
2.7	Перетворення носія і понижуючий коефіцієнт	9
2.8	Архітектура приймачів	11
2.9	Сигнали основної та смугових частот	11
2.10	Налаштування стрибкоподібного та зміщеного постійного струму	12
2.11	Семплірування за допомогою нашого SDR	13
2.12	Обчислення середньої потужності	13
2.13	Обчислення спектральної щільності потужності	14
2.14	Подальше читання	15
3	Частотна область	16
3.1	Ряди Фур'є	16
3.2	Часово-частотні пари	17
3.3	Перетворення Фур'є	18
3.4	Часо-частотні властивості	19
3.5	Швидке перетворення Фур'є (ШПФ)	22
3.6	Від'ємні частоти	23
3.7	Порядок в часі не має значення	23
3.8	ШПФ у Python	24
3.9	Віконна функція	26
3.10	Визначення розміру ШПФ	27
3.11	Спектрограма/Водоспад	27
3.12	Реалізація ШПФ	28
4	Фільтри	31
4.1	Основи фільтрів	31
4.2	Представлення фільтрів	32
4.3	Реалізація фільтрів	37
4.4	Інструменти для проектування фільтрів	38
4.5	Згортка	38
4.6	Проектування фільтрів у Python	40
4.7	Вступ до формування імпульсів	41

5 Шум і дБ	44
5.1 Гауссівський шум	44
5.2 Децибели (дБ)	45
5.3 Шум в частотній області	47
5.4 Комплексний шум	48
5.5 AWGN	49
5.6 SNR і SINR	49
5.7 Зовнішні ресурси	50
6 Цифрова модуляція	51
6.1 Символи	51
6.2 Бездротові символи	52
6.3 Амплітудна маніпуляція (ASK)	52
6.4 Фазова маніпуляція (PSK)	53
6.5 IQ-графіки/сузір'я	53
6.6 Квадратурна амплітудна модуляція (QAM)	55
6.7 Частотна маніпуляція (FSK)	55
6.8 Диференціальне кодування	56
6.9 Приклад на Python	57
6.10 Додаткова інформація	58
7 Pulse Shaping	59
7.1 Міжсимвольна інтерференція (ISI)	59
7.2 Узгоджений фільтр	59
7.3 Розділення фільтра навіпіл	60
7.4 Специфічні фільтри формування імпульсів	61
7.5 Коефіцієнт згортання	62
7.6 Вправа на Python	62
8 Синхронізація	66
8.1 Вступ	66
8.2 Моделювання бездротового каналу	66
8.3 Синхронізація часу	68
8.4 Синхронізація часу за допомогою інтерполяції	70
8.5 Груба частотна синхронізація	72
8.6 Точна частотна синхронізація	75
8.7 Синхронізація кадрів	78
9 Канальне кодування	80
9.1 Для чого потрібне канальне кодування	80
9.2 Типи Кодів	81
9.3 Коефіцієнт Кодування	81
9.4 Модуляція та Кодування	82
9.5 Код Хеммінга	82
9.6 (Порівняння м'якого та жорсткого декодування) Soft vs Hard Decoding	83
9.7 Границя Шеннона	84
9.8 Найсучасніші коди	85
10 IQ файли та SigMF	86
10.1 Двійкові файли	86
10.2 Приклади на Python	87
10.3 Перехід з MATLAB	88
10.4 Візуальний аналіз радіочастотного файлу	89
10.5 Максимальні значення та насиченість	89

10.6 SigMF та анотування IQ файлів	90
10.7 Колекція SigMF для масивних записів	92
A QA-позначки після конверсії	95

Розділ 1

Вступ

1.1. Мета та цільова аудиторія

Перш за все, кілька важливих термінів:

Програмно-визначене радіо (SDR): Як *концепція*, це використання програмного забезпечення для виконання завдань обробки сигналів, які традиційно виконувалися апаратними засобами та пов'язані з радіо/RF застосунками. Це програмне забезпечення може працювати на комп'ютері загального призначення (CPU), FPGA або навіть GPU і може застосовуватися як у режимі реального часу, так і для офлайн-обробки записаних сигналів. Аналогічні терміни: «програмне радіо» та «радіочастотна цифрова обробка сигналів».

Як *пристрій* (наприклад, «SDR») це зазвичай пристрій, до якого можна під'єднати антену й приймати радіочастотні сигнали, а оцифровані RF-вибірки надсилаються на комп'ютер для обробки чи запису (наприклад, через USB, Ethernet, PCI). Багато SDR також мають можливості передавання, що дозволяють комп'ютеру надсилати вибірки до SDR, який потім випромінює сигнал на заданій радіочастоті. Деякі вбудовані SDR мають інтегрований комп'ютер.

Цифрова обробка сигналів (DSP): Цифрова обробка сигналів; у нашому випадку — радіосигналів.

Цей підручник є практичним вступом до галузей DSP, SDR та бездротового зв'язку. Він призначений для тих, хто:

1. Зацікавлений у *використанні* SDR для створення крутих речей
2. Добре знається на Python
3. Відносно новачок в ЦОС (DSP), бездротовому зв'язку та SDR
4. Візуальний учень, що віддає перевагу анімації, а не рівнянням
5. Краще розуміє рівняння *після* вивчення концепцій
6. Шукає стислі пояснення, а не 1000-сторінковий підручник

Прикладом може бути студент факультету комп'ютерних наук, зацікавлений у роботі, пов'язаній з бездротовим зв'язком, після закінчення навчання, хоча ним може користуватися будь-хто, хто прагне вивчити SDR і має досвід програмування. Таким чином, він охоплює необхідну теорію для розуміння методів DSP без складної математики, яка зазвичай присутня в курсах DSP. Замість того, щоб занурюватися в рівняння, автор використовує велику кількість зображень і анімацій, які допомагають передати концепції, наприклад анімацію комплексної площини ряду Фур'є, наведену нижче. Я вважаю, що рівняння найкраще розуміються

після опанування концепцій за допомогою візуалізацій і практичних вправ. Інтенсивне використання анімацій — причина, чому PySDR ніколи не матиме друкованої версії, що продається на Amazon.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/fft_logo_wide.gif

Цей підручник покликаний швидко і плавно ознайомити з поняттями, що дозволить читачеві виконувати DSP і розумно використовувати SDR. Він не задуманий як довідник з усіх тем DSP/SDR; уже існує безліч чудових підручників, таких як [Analog Device's SDR textbook](#) та [dspguide.com](#). Ви завжди можете скористатися Google, щоб пригадати тригонометричні тотожності або межу Шеннона. Сприймайте цей підручник як «ворота» у світ DSP та SDR: він легший і потребує менше часу та коштів у порівнянні з більш традиційними курсами і підручниками.

Щоб охопити фундаментальну теорію DSP, цілий семестр курсу «Сигнали і системи», типового для електротехнічних спеціальностей, стиснуто до кількох розділів. Після вивчення основ DSP ми переходимо до SDR, хоча поняття DSP і бездротового зв'язку продовжують з'являтися протягом усього підручника.

Приклади коду наведено мовою Python. Використовується NumPy — стандартна бібліотека Python для масивів і високорівневої математики. Приклади також покладаються на Matplotlib — бібліотеку побудови графіків Python, що надає простий спосіб візуалізації сигналів, масивів і комплексних чисел. Зауважте, що хоча Python загалом «повільніша», ніж C++, більшість математичних функцій у Python/NumPy реалізовано на C/C++ і добре оптимізовано. Так само API SDR, який ми використовуємо, — це набір Python-обгортки для функцій/класів C/C++. Ті, хто має небагато досвіду з Python, але міцну базу в MATLAB, Ruby або Perl, ймовірно, почуватимуться впевнено після ознайомлення із синтаксисом Python.

1.2. Долучитися

Якщо ви отримали користь від PySDR, поділіться ним із колегами, студентами та іншими допитливими людьми, яких може зацікавити цей матеріал. Ви також можете зробити пожертву через [PySDR Patreon](#) як спосіб подяки та отримати згадку зліва на кожній сторінці під списком розділів.

Якщо ви прочитаєте будь-яку частину цього підручника і напишете мені на marc@pysdr.org з питаннями/коментарями/пропозиціями, то вітаємо, ви зробили свій внесок у створення цього підручника! Ви також можете редагувати вихідний матеріал безпосередньо на сторінці [підручника на GitHub](#) (ваша зміна покладе початок новому запиту на заміну). Не соромтеся надсилати проблему або навіть запит на вилучення (PR) з виправленнями або покращеннями. Ті, хто надсилає цінні відгуки/виправлення, будуть постійно додаватися до розділу подяк нижче. Не дуже добре володієте Git'ом, але хочете запропонувати зміни? Не соромтеся писати мені на marc@pysdr.org.

1.3. Подяки

Дякуємо всім, хто прочитав будь-яку частину цього підручника і надіслав відгук, а особливо:

- [Barry Duggan](#)
- Matthew Hannon
- James Hayek
- Deidre Stuffer

- Tarik Benaddi за [переклад PySDR французькою](#)
- [Daniel Versluis](#) за [переклад PySDR нідерландською](#)
- [mrbloom](#) за [переклад PySDR українською](#)
- [Yimin Zhao](#) за [переклад PySDR спрощеною китайською](#)
- [Eduardo Chancay](#) за [переклад PySDR іспанською](#)
- John Marcovici

А також усім прихильникам [PySDR Patreon!](#)

Розділ 2

Вибірка IQ

У цій главі ми представляємо концепцію під назвою IQ вибірка, також відома як комплексна вибірка або квадратурна вибірка. Ми також розглядаємо дискретизацію Найквіста, комплексні числа, радіочастотні носії, понижувальне перетворення та спектральну щільність потужності. Вибірка IQ — це форма вибірки, яку виконує SDR, а також багато цифрових приймачів (і передавачів). Це дещо складніша версія звичайної цифрової вибірки (каламбур), тому ми будемо робити це повільно, і з деякою практикою ця концепція обов’язково спрацює!

2.1. Основи вибірки

Перш ніж перейти до вибірки IQ, давайте обговоримо, що насправді означає вибірка. Можливо, ви стикалися із семплюванням, не усвідомлюючи цього, записуючи звук за допомогою мікрофона. Мікрофон — це перетворювач, який перетворює звукові хвилі в електричний сигнал (рівень напруги). Цей електричний сигнал перетворюється аналого-цифровим перетворювачем (АЦП), створюючи цифрове представлення звукової хвилі. Щоб спростити, мікрофон вловлює звукові хвилі, які перетворюються на електрику, а ця електрика, у свою чергу, перетворюється на числа. АЦП діє як міст між аналоговою та цифровою областями. СПЗ напрочуд схожі. Однак замість мікрофона вони використовують антену, хоча вони також використовують АЦП. В обох випадках рівень напруги вимірюється за допомогою АЦП. Для СПЗ подумайте, що радіохвилі входять, а потім зменшуються.

Незалежно від того, чи ми маємо справу з аудіо чи радіочастотами, ми повинні взяти вибірку, якщо ми хочемо захопити, обробити або зберегти сигнал у цифровому вигляді. Вибірка може здатися простою, але в ній багато чого. Більш технічний спосіб дискретизації сигналу полягає в тому, щоб отримати значення в певний момент часу та зберегти їх у цифровому вигляді. Скажімо, у нас є якась випадкова функція $S(t)$, яка може представляти будь-що, і це неперервна функція, яку ми хочемо взяти на вибірку:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив’язка: ../_images/sampling.svg

Ми записуємо значення $S(t)$ через рівні проміжки в T секунди, відомі як **період вибірки**. Частота, з якою ми беремо вибірку, тобто кількість вибірок, взятих за секунду, є просто $\frac{1}{T}$. Ми називаємо це **частотою дискретизації**, і вона зворотна до періоду дискретизації. Наприклад, якщо ми маємо частоту дискретизації 10 Гц, то період дискретизації становить 0,1 секунди; між кожним зразком буде 0,1 секунди. На практиці наші частоти дискретизації будуть порядку сотень кГц до десятків МГц або навіть вище. Коли ми знімаємо сигнали, ми повинні пам’ятати про частоту дискретизації, це дуже важливий параметр.

Для тих, хто вважає за краще бачити математику; нехай S_n представляє вибірку n , зазвичай ціле число, починаючи з 0. Використовуючи цю угоду, процес вибірки може бути представлений математично як $S_n = S(nT)$ для цілих значень n . Тобто ми оцінюємо аналоговий сигнал $S(t)$ на цих інтервалах nT .

2.2. Вибірка Найквіста

Для даного сигналу головним питанням часто є те, як швидко ми повинні проводити вибірку? Давайте розглянемо сигнал, який є просто синусоїдальною хвилею частоти f , показаної зеленим кольором нижче. Скажімо, ми беремо вибірку зі швидкістю F_s (зразки показано синім кольором). Якщо ми відберемо цей сигнал зі швидкістю, що дорівнює f (тобто $F_s = f$), ми отримаємо щось таке:

:: `../_images/sampling_Fs_0.3.svg align center`

Червона пунктирна лінія на наведеному вище зображенні реконструює іншу (неправильну) функцію, яка могла призвести до запису тих самих семплів. Вона вказує на те, що наша частота дискретизації була занадто низькою, тому що ті самі відліки могли бути отримані від двох різних функцій, що призвело до неоднозначності. Якщо ми хочемо точно реконструювати оригінальний сигнал, ми не можемо допустити такої неоднозначності.

Спробуймо дискретизувати трохи швидше, з $F_s = 1.2f$:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: `../_images/sampling_Fs_0.36.svg`

Знову ж таки, існує інший сигнал, який може відповідати цим вибіркам. Ця неоднозначність означає, що якби хтось надав нам цей список зразків, ми не змогли б розрізнити, який сигнал є оригінальним, на основі нашої вибірки.

А як щодо дискретизації при $F_s = 1.5f$:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: `../_images/sampling_Fs_0.45.svg`

Все ще недостатньо швидко! Згідно з теорією DSP, в яку ми не будемо занурюватися, вам потрібно робити вибірки з частотою, вдвічі більшою за частоту сигналу, щоб усунути неоднозначність, з якою ми стикаємося:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: `../_images/sampling_Fs_0.6.svg`

Цього разу немає неправильного сигналу, тому що ми робили вибірки досить швидко, щоб не існувало жодного сигналу, який би відповідав цим вибіркам, окрім того, який ви бачите (якщо тільки ви не підніметеся *вище* за частотою, але про це ми поговоримо пізніше).

У наведеному вище прикладі наш сигнал був простою синусоїдою, більшість реальних сигналів матимуть багато частотних складових. Для точної дискретизації будь-якого сигналу частота дискретизації повинна бути "щонайменше вдвічі більшою за частоту максимальної частотної складової". Ось візуалізація на прикладі графіку частотної області, зверніть увагу, що завжди буде існувати рівень шуму, тому найвища частота зазвичай є наближеним значенням:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: `../_images/max_freq.svg`

Ми повинні визначити найвищу частотну складову, потім подвоїти її, і переконатися, що ми робимо вибірку з такою частотою або швидше. Мінімальна частота,

з якою ми можемо робити вибірку, називається частотою Найквіста. Іншими словами, частота Найквіста - це мінімальна частота, з якою сигнал (зі скінченною смугою пропускання) повинен бути дискретизований, щоб зберегти всю його інформацію. Це надзвичайно важлива частина теорії в DSP і SDR, яка служить мостом між безперервними і дискретними сигналами.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/nyquist_rate.png

Якщо ми робимо вибірку недостатньо швидко, ми отримуємо те, що називається аліасинг, про який ми дізнаємось пізніше, але ми намагаємось уникати цього за будь-яку ціну. Наші SDR (і більшість приймачів взагалі) відфільтровують все, що перевищує $F_s/2$, безпосередньо перед тим, як виконується дискретизація. Якщо ми спробуємо прийняти сигнал із занадто низькою частотою дискретизації, цей фільтр відсіче частину сигналу. Наші SDR роблять все можливе, щоб забезпечити нас зразками, вільними від аліасингу та інших недосконалостей.

2.3. Квадратурна дискретизація

Термін "квадратура" має багато значень, але в контексті DSP і SDR він відноситься до двох хвиль, які знаходяться на 90 градусів у протифазі. Чому 90 градусів у протифазі? Подумайте про те, що дві хвилі, які знаходяться в протифазі на 180 градусів, по суті, є однією і тією ж хвилею, але помноженою на -1. При зсуві на 90 градусів вони стають ортогональними, а з ортогональними функціями можна робити багато цікавих речей. Для простоти ми використовуємо синус і косинус як дві синусоїди, що зсунуті на 90 градусів у фазі.

Далі призначимо змінні для представлення **амплітуди** синуса і косинуса. Ми будемо використовувати I для $\cos()$ і Q для $\sin()$:

$$I \cos(2\pi ft)$$

$$Q \sin(2\pi ft)$$

Ми можемо побачити це візуально, побудувавши графік I і Q , що дорівнює 1:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/IQ_wave.png

Ми називаємо $\cos()$ "синфазним" компонентом, звідси і назва I , а $\sin()$ - зсув на 90 градусів або "квадратурний" компонент, звідси і назва Q . Хоча якщо ви випадково переплутаєте їх і віднесете Q до $\cos()$, а I до $\sin()$, це не матиме значення для більшості ситуацій.

Дискретизацію IQ легше зрозуміти, використовуючи точку зору передавача, тобто розглядаючи задачу передачі радіосигналу через повітря. Ми хочемо передати одну синусоїду з певною фазою, що можна зробити, передавши суму $\sin()$ і $\cos()$ з фазою 0, завдяки тригонометричній тотожності: $a \cos(x) + b \sin(x) = A \cos(x - \phi)$. Нехай $x(t)$ - це наш сигнал для передачі:

$$x(t) = I \cos(2\pi ft) + Q \sin(2\pi ft)$$

Що відбувається, коли ми додаємо синус і косинус? Або, точніше, що станеться, коли ми додамо дві синусоїди, які зсунуті на 90 градусів у фазі? У відео нижче є повзунок для регулювання I і повзунок для регулювання Q . На графіку зображені косинус, синус, а потім їхня сума.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/IQ3.gif

(Код, використаний для цієї програми на основі pyqtgraph у Python, можна знайти [тут](#))

Важливим висновком є те, що коли ми додаємо $\cos()$ і $\sin()$, ми отримуємо ще одну чисту синусоїду з іншою фазою і амплітудою. Крім того, фаза зміщується, коли ми повільно видаляємо або додаємо одну з двох частин. Амплітуда також змінюється. Це все є результатом тригонометричної тотожності: $a \cos(x) + b \sin(x) = A \cos(x - \phi)$, до якої ми повернемося трохи згодом. "Корисність" такої поведінки полягає в тому, що ми можемо керувати фазою і амплітудою результуючої синусоїди, змінюючи амплітуди I і Q (нам не потрібно змінювати фазу косинуса або синуса). Наприклад, ми можемо регулювати I і Q таким чином, щоб амплітуда залишалася постійною, а фаза була такою, якою ми хочемо. Для передавача ця здатність надзвичайно корисна, оскільки ми знаємо, що нам потрібно передавати синусоїдальний сигнал, щоб він пролетів у повітрі у вигляді електромагнітної хвилі. І набагато простіше відрегулювати дві амплітуди і виконати операцію додавання, ніж відрегулювати амплітуду і фазу. В результаті наш передавач буде виглядати приблизно так:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/IQ_diagram.png

Нам потрібно згенерувати лише одну синусоїду і зсунути її на 90 градусів, щоб отримати Q-частину.

2.4. Комплексні числа

Зрештою, IQ - це альтернативний спосіб представлення амплітуди і фази, який приводить нас до комплексних чисел і можливості представляти їх на комплексній площині. Можливо, ви вже зустрічалися з комплексними числами в інших класах. Візьмемо для прикладу комплексне число $0.7-0.4j$:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/complex_plane_1.png

Комплексне число - це просто два числа разом, дійсна і уявна частина. Комплексне число також має амплітуду і фазу, що має більше сенсу, якщо думати про нього як про вектор, а не точку. Величина - це довжина лінії між початком координат і точкою (тобто довжина вектора), а фаза - це кут між вектором і 0 градусів, який ми визначаємо як додатну дійсну вісь:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/complex_plane_2.png

Таке представлення синусоїди відоме як "фазорна діаграма". Це просто побудова комплексних чисел і поводження з ними як з векторами. Яка ж величина і фаза нашого прикладу комплексного числа $0.7-0.4j$? Для даного комплексного числа, де a - дійсна частина, а b - уявна частина:

$$\text{magnitude} = \sqrt{a^2 + b^2} = 0.806$$

$$\text{phase} = \tan^{-1} \left(\frac{b}{a} \right) = -29.7^\circ = -0.519 \text{ radians}$$

У Python ви можете використовувати функції `np.abs(x)` і `np.angle(x)` для амплітуди і фази. На вхід може подаватися комплексне число або масив комплексних чисел, а на виході буде **дійсне** число (з типом даних `float`).

Можливо, ви вже з'ясували, як ця векторна або фазова діаграма пов'язана з умовною позначкою IQ: I - це дійсне число, а Q - уявне. З цього моменту, коли

ми будемо малювати комплексну площину, ми будемо позначати її не дійсними і уявними числами, а I і Q. Це все одно комплексні числа!

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/complex_plane_3.png

Тепер припустимо, що ми хочемо передати наш приклад точки 0.7-0.4j. Ми будемо передавати:

$$\begin{aligned} x(t) &= I \cos(2\pi ft) + Q \sin(2\pi ft) \\ &= 0.7 \cos(2\pi ft) - 0.4 \sin(2\pi ft) \end{aligned}$$

Ми можемо використати тригонометричну тотожність $a \cos(x) + b \sin(x) = A \cos(x - \phi)$, де A - наша величина, знайдена за допомогою $\sqrt{I^2 + Q^2}$ і ϕ - наша фаза, яка дорівнює $\tan^{-1}(Q/I)$. Вищенаведене рівняння набуває вигляду:

$$x(t) = 0.806 \cos(2\pi ft + 0.519)$$

Незважаючи на те, що ми почали з комплексного числа, те, що ми передаємо, є реальним сигналом з певною амплітудою і фазою; насправді ви не можете передати щось уявне за допомогою електромагнітних хвиль. Ми просто використовуємо уявні/комплексні числа для представлення того, "що" ми передаємо. Незабаром ми поговоримо про f .

2.5. Комплексні числа у ШПФ

Наведені вище комплексні числа розглядалися як приклади часової області, але ви також зіткнетеся з комплексними числами, коли будете застосовувати ШПФ. Коли ми розглядали ряди Фур'є і ШПФ в минулому розділі, ми ще не занурювалися в комплексні числа. Коли ви застосовуєте ШПФ до серії відліків, він знаходить представлення в частотній області. Ми говорили про те, як ШПФ з'ясовує, які частоти існують в цьому наборі відліків (величина ШПФ вказує на силу кожної частоти). Але ШПФ також обчислює затримку (часовий зсув), необхідну для застосування до кожної з цих частот, щоб набір синусоїд можна було скласти для відновлення сигналу в часовій області. Ця затримка є просто фазою ШПФ. Результатом ШПФ є масив комплексних чисел, і кожне комплексне число дає вам амплітуду і фазу, а індекс цього числа дає вам частоту. Якщо ви згенеруєте синусоїди на цих частотах/амплітудах/фазах і підсумуйте їх разом, ви отримаєте вихідний сигнал в часовій області (або щось дуже близьке до нього, і саме тут вступає в дію теорема дискретизації Найквіста).

2.6. Сторона приймача

Тепер давайте подивимось на радіоприймач, який намагається прийняти сигнал (наприклад, FM-радіосигнал). Використовуючи IQ-семплінг, діаграма тепер має такий вигляд:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/IQ_diagram_rx.png

На вхід надходить реальний сигнал, отриманий на нашу антену, і він перетворюється на значення IQ. Ми робимо вибірку гілок I і Q окремо, використовуючи два АЦП, а потім об'єднуємо пари і зберігаємо їх у вигляді комплексних чисел. Іншими словами, на кожному кроці ви будете відбирати одне значення I і одне

значення Q і об'єднувати їх у вигляді $I + jQ$ (тобто, одне комплексне число на вибірку IQ). Завжди буде існувати "частота дискретизації", тобто швидкість, з якою виконується вибірка. Хтось може сказати: "У мене є SDR з частотою дискретизації 2 МГц". Вони мають на увазі, що SDR отримує два мільйони відліків IQ в секунду.

Якщо хтось дасть вам купу відліків IQ, це буде виглядати як одномірний масив/вектор комплексних чисел. Саме до цього моменту, комплексного чи ні, ми будували всю цю главу, і ми нарешті досягли його.

Протягом усього підручника ви будете дуже добре знайомі з тим, як працюють IQ-тести, як їх отримувати та передавати за допомогою SDR, як обробляти їх у Python і як зберігати у файл для подальшого аналізу.

Останнє важливе зауваження: на рисунку вище показано, що відбувається **всередині** SDR. Насправді нам не потрібно генерувати синусоїду, зсувати на 90, множити або додавати - SDR робить це за нас. Ми повідомляємо SDR, на якій частоті ми хочемо зробити вибірку, або на якій частоті ми хочемо передати наші вибірки. На стороні приймача SDR надасть нам зразки IQ. На стороні передавача ми повинні надати SDR зразки IQ. З точки зору типу даних, це будуть або складні числа, або числа з плаваючою точкою.

2.7. Перетворення носія і понижуючий коефіцієнт

До цього моменту ми не обговорювали частоту, але ми бачили, що в рівняннях, що включають $\cos()$ і $\sin()$ є f . Ця частота є центральною частотою сигналу, який ми фактично посилаємо через повітря (частота електромагнітної хвилі). Ми називаємо її "несучою", тому що вона переносить наш сигнал на певній радіочастоті. Коли ми налаштовуємося на частоту за допомогою SDR і отримуємо відліки, наша інформація зберігається в I і Q ; ця несуча не відображається в I і Q , якщо припустити, що ми налаштувалися на несучу.

`[font=Largebfseriesfffamily] draw (0,0) node[align=center]{$\cdot \cos(2\pi ft + \phi)$} (0,-2) node[align=center]{$\left(\sqrt{I^2+Q^2}\right)\cos\left(2\pi ft + \tan^{-1}\left(\frac{Q}{I}\right)\right)$}; draw[>,red,thick] (-2,-0.5) -- (-2.5,-1.2); draw[>,red,thick] (1.9,-0.5) -- (2.4,-1.5); draw[>,red,thick] (0,-4) node[red, below, align=center]{Це те, що ми називаємо носієм} -- (-0.6,-2.7);`

Для довідки: радіосигнали, такі як FM-радіо, WiFi, Bluetooth, LTE, GPS тощо, зазвичай використовують частоту (тобто несучу) між 100 МГц і 6 ГГц. Ці частоти дуже добре поширюються в повітрі, але вони не потребують наддовгих антен або великої потужності для передачі чи прийому. Ваша мікрохвильова піч готує їжу за допомогою електромагнітних хвиль на частоті 2,4 ГГц. Якщо дверцята мікрохвильової печі протікають, вона глушитиме сигнали WiFi і, можливо, також обпече вашу шкіру. Іншою формою електромагнітних хвиль є світло. Видиме світло має частоту близько 500 ТГц. Це настільки висока частота, що ми не використовуємо традиційні антени для передачі світла. Ми використовуємо такі методи, як світлодіоди, які є напівпровідниковими пристроями. Вони створюють світло, коли електрони перескакують між атомними орбітами напівпровідникового матеріалу, і колір залежить від того, як далеко вони перескакують. Технічно радіочастота (РЧ) визначається як діапазон від приблизно 20 кГц до 300 ГГц. Це частоти, на яких енергія електричного струму, що коливається, може випромінюватися з провідника (антени) і поширюватися в просторі. Діапазон від 100 МГц до 6 ГГц є найбільш корисними частотами, принаймні для більшості сучасних застосувань. Частоти вище 6 ГГц десятиліттями використовувалися для радарів і супутникового зв'язку, а зараз застосовуються в 5G "mmWave" (24 - 29 ГГц) для доповнення нижніх діапазонів і збільшення швидкості.

Коли ми швидко змінюємо значення IQ і передаємо нашу несучу, це називається "модуляцією" несучої (даними або чим завгодно). Коли ми змінюємо I і Q, ми змінюємо фазу і амплітуду несучої. Інший варіант - змінити частоту несучої, тобто зсунути її трохи вгору або вниз, як це робить FM-радіо.

Як простий приклад, скажімо, ми передаємо зразок $IQ\ 1+0j$, а потім переходимо на передачу $0+1j$. Ми переходимо від передачі $\cos(2\pi ft)$ до $\sin(2\pi ft)$, тобто наша несуча зсувається по фазі на 90 градусів, коли ми переходимо від однієї вибірки до іншої.

Легко заплутатися між сигналом, який ми хочемо передати (який зазвичай містить багато частотних компонентів), і частотою, на якій ми його передаємо (наша несуча частота). Сподіваємось, це стане зрозумілим, коли ми розглянемо базові та смугові сигнали.

А тепер повернемося на секунду до дискретизації. Замість того, щоб отримувати відліки шляхом множення сигналу з антени на $\cos()$ і $\sin()$, а потім записувати I і Q, що, якби ми подавали сигнал з антени на один АЦП, як в архітектурі з прямою дискретизацією, яку ми щойно обговорювали? Скажімо, несуча частота 2,4 ГГц, як у WiFi або Bluetooth. Це означає, що нам доведеться робити вибірки на частоті 4,8 ГГц, як ми вже дізналися. Це надзвичайно швидко! АЦП, який робить такі швидкі вибірки, коштує тисячі доларів. Замість цього ми "понижуємо" сигнал так, щоб сигнал, який ми хочемо отримати, був зосереджений навколо постійного струму або 0 Гц. Це перетворення відбувається до того, як ми зробимо вибірку. Ми переходимо від

$$I \cos(2\pi ft)$$

$$Q \sin(2\pi ft)$$

до просто I та Q.

Візуалізуємо даунконверсію у частотній області:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/downconversion.png

Коли ми відцентрували сигнал на 0 Гц, максимальна частота вже не становить 2,4 ГГц, а базується на характеристиках сигналу, оскільки ми видалили несучу. Більшість сигналів мають ширину смуги пропускання від 100 кГц до 40 МГц, тому за допомогою низхідного перетворення ми можемо робити вибірки на *набагато* нижчій частоті. I B2X0 USRP, і PlutoSDR містять радіочастотну інтегральну схему (RFIC), яка може робити вибірки на частоті до 56 МГц, що досить високо для більшості сигналів, з якими ми зустрінемося.

Нагадаємо, що процес пониження частоти виконується нашим SDR; як користувачеві SDR нам не потрібно нічого робити, окрім як вказати йому, на яку частоту налаштуватися. Низькочастотне (і високочастотне) перетворення здійснюється за допомогою компонента, який називається мікшером, що зазвичай зображується на схемах у вигляді символу множення всередині кола. Мікшер приймає сигнал, виводить перетворений вниз/вгору сигнал і має третій порт, який використовується для підключення генератора. Частота генератора визначає зсув частоти, що застосовується до сигналу, а мікшер, по суті, є просто функцією множення (нагадаємо, що множення на синусоїду викликає зсув частоти).

Нарешті, вам може бути цікаво, як швидко сигнали поширюються в повітрі. Згадайте з шкільного курсу фізики, що радіохвилі - це просто електромагнітні хвилі на низьких частотах (приблизно від 3 кГц до 80 ГГц). Видиме світло - це також електромагнітні хвилі, але на значно вищих частотах (від 400 ТГц до 700 ТГц). Всі електромагнітні хвилі поширюються зі швидкістю світла, яка становить близько 3×10^8 м/с, принаймні, коли вони проходять через повітря або вакуум. Оскільки

вони завжди рухаються з однаковою швидкістю, відстань, яку хвиля проходить за одне повне коливання (один повний цикл синусоїди), залежить від її частоти. Ми називаємо цю відстань довжиною хвилі і позначаємо її λ . Ви, мабуть, бачили цю залежність раніше:

$$f = \frac{c}{\lambda}$$

де c - швидкість світла, зазвичай дорівнює 3×10^8 , коли f - у Гц, а λ - у метрах. У бездротовому зв'язку це співвідношення стає важливим, коли ми переходимо до антен, тому що для прийому сигналу на певній несучій частоті, f , потрібна антена, яка відповідає його довжині хвилі, λ , зазвичай антена має $\lambda/2$ або $\lambda/4$ довжину. Однак, незалежно від частоти/довжини хвилі, інформація, що міститься в цьому сигналі, завжди буде рухатися зі швидкістю світла від передавача до приймача. При обчисленні цієї затримки в повітрі можна скористатися емпіричним правилом, що світло проходить приблизно один фут за одну наносекунду. Ще одне емпіричне правило: сигнал, що проходить шлях до супутника на геостационарній орбіті і назад, займає приблизно 0,25 секунди на весь шлях.

2.8. Архітектура приймачів

На малюнку в розділі "Приймач" показано, як вхідний сигнал перетворюється і розділяється на I і Q. Така схема називається "пряме перетворення", або "нульова ПЧ", тому що радіочастоти безпосередньо перетворюються до базової смуги частот. Інший варіант - взагалі не перетворювати частоти вниз і робити вибірку так швидко, щоб захопити все від 0 Гц до $1/2$ частоти дискретизації. Ця стратегія називається "пряма вибірка" або "пряма ВЧ", і вона вимагає надзвичайно дорогого чіпа АЦП. Третя архітектура, популярна тому, що саме так працювали старі радіоприймачі, відома як "супергетеродин". Вона передбачає перетворення вниз, але не до 0 Гц. Він поміщає сигнал, що нас цікавить, на проміжну частоту, відому як "ПЧ". Підсилювач з низьким рівнем шуму (LNA) - це просто підсилювач, призначений для сигналів надзвичайно низької потужності на вході. Ось блок-схеми цих трьох архітектур, зверніть увагу, що існують також варіації та гібриди цих архітектур:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/receiver_arch_diagram.svg

2.9. Сигнали основної та смугових частот

Ми називаємо сигнал з центром близько 0 Гц "основною смугою". І навпаки, "смуговий" означає, що сигнал існує на певній радіочастоті, не близькій до 0 Гц, яка була зміщена вгору з метою бездротової передачі. Поняття "передача в основній смузі частот" не існує, тому що ви не можете передати щось уявне. Сигнал в основній смузі може бути ідеально відцентрований на 0 Гц, як на правій частині малюнка в попередньому розділі. Він може бути *близько* 0 Гц, як два сигнали, показані нижче. Ці два сигнали все ще вважаються основною смугою. Також показано приклад смугового сигналу з центром на дуже високій частоті, позначений f_c .

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/baseband_bandpass.png

Ви також можете почути термін "проміжна частота" (скорочено ПЧ); поки що уявіть собі ПЧ як проміжний крок перетворення у радіоприймачі між базовою смугою та смугою пропускання/ВЧ.

Ми, як правило, створюємо, записуємо або аналізуємо сигнали в базовій смузі, оскільки можемо працювати з меншою частотою дискретизації (з причин, описаних у попередньому підрозділі). Важливо зазначити, що сигнали базової смуги часто є складними сигналами, в той час як сигнали в смузі пропускання (наприклад, сигнали, які ми фактично передаємо через радіочастоти) є реальними. Подумайте про це: оскільки сигнал, що подається через антену, повинен бути реальним, ви не можете безпосередньо передавати складний/уявний сигнал. Ви знатимете, що сигнал точно є комплексним, якщо від'ємна та додатна частоти сигналу не збігаються в точності. Зрештою, комплексні числа - це те, як ми представляємо від'ємні частоти. Насправді не існує від'ємних частот; це просто частина сигналу нижче несучої частоти.

У попередньому розділі, де ми гралися з комплексною точкою $0,7 - 0,4j$, це був, по суті, один відлік у сигналі основної смуги. Більшість часу, коли ви бачите комплексні відліки (IQ-відліки), ви перебуваєте в основній смузі частот. Сигнали рідко представляються або зберігаються в цифровому вигляді в радіочастотному діапазоні через велику кількість даних, а також через те, що нас зазвичай цікавить лише невелика частина радіочастотного спектру.

2.10. Налаштування стрибкоподібного та зміщеного постійного струму

Як тільки ви починаєте працювати з SDR, ви часто бачите великий пік в центрі БПФ. Це називається "зміщенням постійного струму" або "стрибком постійного струму", або іноді "витоком LO", де LO означає локальний осцилятор.

Ось приклад стрибка постійного струму:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/dc_spike.png

Оскільки SDR налаштовується на центральну частоту, ділянка 0 Гц у БПФ відповідає центральній частоті. При цьому стрибок постійного струму не обов'язково означає, що на центральній частоті є енергія. Якщо є лише стрибок постійного струму, а решта БПФ виглядає як шум, то, швидше за все, там, де він вам показує, насправді немає сигналу.

Зміщення постійного струму є поширеним артефактом у приймачах прямого перетворення, які використовують архітектуру SDR, таких як PlutoSDR, RTL-SDR, LimeSDR і багатьох Ettus USRP. У приймачах прямого перетворення генератор, LO, перетворює сигнал з його фактичної частоти в базову смугу. В результаті, витоки від цього генератора з'являються в центрі смуги пропускання, що спостерігається. Витоки LO - це додаткова енергія, створена комбінацією частот. Видалити цей додатковий шум важко, оскільки він близький до бажаного вихідного сигналу. Багато радіочастотних інтегральних схем (RFIC) мають вбудовану функцію автоматичного видалення постійного зсуву, але для її роботи зазвичай потрібен сигнал. Ось чому стрибок постійного струму буде дуже помітним за відсутності сигналу.

Швидкий спосіб впоратися зі зміщенням постійного струму - передискретизувати сигнал і розстроїти його. Для прикладу, скажімо, ми хочемо переглянути 5 МГц спектра на частоті 100 МГц. Замість цього ми можемо зробити вибірку на 20 МГц з центральною частотою 95 МГц.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/offtuning.png

Синя рамка вище показує, що насправді виділяється SDR, а зелена рамка відображає частину спектру, яку ми хочемо отримати. Наш LO буде встановлено на 95 МГц, тому що це частота, на яку ми просимо SDR налаштуватися. Оскільки 95 МГц знаходиться за межами зеленого квадрата, ми не отримаємо жодного сплеску постійного струму.

Є одна проблема: якщо ми хочемо, щоб наш сигнал був центрований на 100 МГц і містив лише 5 МГц, нам доведеться самотійно виконати зсув частоти, фільтрацію і пониження дискретизації сигналу (про це ми дізнаємося пізніше). На щастя, цей процес розстроювання, так зване застосування LO-зсуву, часто вбудовано в SDR, де вони автоматично виконують розстроювання, а потім зміщують частоту до бажаної центральної частоти. Ми виграємо, коли SDR може робити це самотійно: нам не потрібно передавати вищу частоту дискретизації через USB або Ethernet-з'єднання, які обмежують можливості використання високої частоти дискретизації.

Цей підрозділ, присвячений зміщенням постійного струму, є гарним прикладом того, чим цей підручник відрізняється від інших. У середньостатистичному підручнику з ЦОС обговорюється дискретизація, але в ньому, як правило, не розглядаються перешкоди при реалізації, такі як зміщення постійного струму, незважаючи на їхню поширеність на практиці.

2.11. Семплірування за допомогою нашого SDR

Для отримання специфічної для SDR інформації про виконання дискретизації див. одну з наступних глав:

- pluto-chapter Розділ
- usrp-chapter Глава

2.12. Обчислення середньої потужності

У радіочастотному DSP ми часто хочемо обчислити потужність сигналу, наприклад, щоб виявити наявність сигналу перед тим, як намагатися виконати подальшу обробку. Для дискретного складного сигналу, тобто сигналу, який ми дискретизували, ми можемо знайти середню потужність, взявши величину кожного відліку, піднісши її до квадрата і знайшовши середнє значення:

$$P = \frac{1}{N} \sum_{n=1}^N |x[n]|^2$$

Пам'ятайте, що абсолютне значення комплексного числа - це просто величина, тобто $\sqrt{I^2 + Q^2}$

У мові Python обчислення середнього степеня буде мати вигляд:

```
avg_pwr = np.mean(np.abs(x)**2)
```

Це дуже корисний трюк для обчислення середньої потужності дискретизованого сигналу. Якщо ваш сигнал має приблизно нульове середнє значення - що зазвичай буває в SDR (пізніше ми побачимо чому) - то потужність сигналу можна знайти, взявши дисперсію відліків. За цих обставин ви можете обчислити потужність таким чином у Python:

```
avg_pwr = np.var(x) # (сигнал повинен мати приблизно нульове середнє)
```

Причина, чому дисперсія вибірок обчислює середню потужність, досить проста: рівняння для дисперсії має вигляд $\frac{1}{N} \sum_{n=1}^N |x[n] - \mu|^2$ де μ - середнє значення сигналу. Це рівняння виглядає знайомим! Якщо μ дорівнює нулю, то рівняння для визначення дисперсії відліків стає еквівалентним рівнянню для потужності. Ви також можете відняти середнє значення від вибірок у вашому вікні спостережень, а потім взяти дисперсію. Просто знайте, що якщо середнє значення не дорівнює нулю, то дисперсія і потужність не рівні.

2.13. Обчислення спектральної щільності потужності

У попередньому розділі ми дізналися, що можна перетворити сигнал у частотну область за допомогою ШПФ, а результат називається спектральною щільністю потужності (PSD). PSD є надзвичайно корисним інструментом для візуалізації сигналів у частотній області, і багато алгоритмів ЦОС виконуються в частотній області. Але для того, щоб дійсно знайти PSD пачки відліків і побудувати її графік, ми робимо більше, ніж просто беремо БПФ. Для обчислення PSD потрібно виконати наступні шість операцій:

1. Беремо БПФ наших відліків. Якщо у нас є x відліків, то за замовчуванням розмір БПФ буде дорівнювати довжині x . Давайте використаємо перші 1024 відліки як приклад для створення ШПФ розміром 1024. На виході ми отримаємо 1024 комплексних числа з плаваючою комою.
2. Візьмемо величину виходу ШПФ, яка дає нам 1024 дійсних числа.
3. Піднесіть отриману величину до квадрату, щоб отримати потужність.
4. Нормалізуємо: ділимо на розмір ШПФ (N) і частоту дискретизації (F_s).
5. Конвертуємо в дБ за допомогою $10 \log_{10}()$; ми завжди переглядаємо PSD в логарифмічній шкалі.
6. Виконайте зсув ШПФ, описаний у попередньому розділі, щоб перемістити "0 Гц" у центр, а від'ємні частоти - ліворуч від центру.

Ці шість кроків у Python виглядають так:

```
Fs = 1e6 # скажімо, ми зробили вибірку на частоті 1 МГц
# припустимо, що x містить ваш масив відліків IQ
N = 1024
x = x[0:N] # ми візьмемо ШПФ тільки перших 1024 відліків, див. текст нижче
PSD = np.abs(np.fft.fft(x))**2 / (N*Fs)
PSD_log = 10.0*np.log10(PSD)
PSD_shifted = np.fft.fftshift(PSD_log)
```

За бажанням ми можемо застосувати вікно, про яке ми дізналися у розділі `freq-domain-chapter`. Вікно з'явиться безпосередньо перед рядком коду з `fft()`.

```
# додати наступний рядок після виконання x = x[0:1024]
x = x * np.hamming(len(x)) # застосовуємо вікно Хеммінга
```

Для побудови цієї PSD нам потрібно знати значення осі x . Як ми дізналися з попереднього розділу, коли ми робимо вибірку сигналу, ми "бачимо" лише спектр між $-F_s/2$ і $F_s/2$, де F_s - це частота дискретизації. Роздільна здатність, якої ми досягаємо в частотній області, залежить від розміру нашого ШПФ, який за замовчуванням дорівнює кількості відліків, над якими ми виконуємо операцію ШПФ. У цьому випадку наша вісь x - це 1024 рівномірно розташовані точки між $-0,5$ МГц

і 0,5 МГц. Якби ми налаштували наш SDR на 2,4 ГГц, наше вікно спостереження було б між 2,3995 ГГц і 2,4005 ГГц. У Python зміщення вікна спостереження буде виглядати так:

```
center_freq = 2.4e9 # частота, на яку ми налаштували наш SDR
f = np.arange(Fs/-2.0, Fs/2.0, Fs/N) # старт, стоп, крок. з центром навколо 0 Гц
f += center_freq # тепер додаємо центральну частоту
plt.plot(f, PSD_shifted)
plt.show()
```

У нас повинен залишитися гарний PSD!

Якщо ви хочете знайти PSD для мільйонів відліків, не робіть ШПФ з мільйонами точок, тому що це займе вічність. Зрештою, це дасть вам на виході мільйон "частотних бінів", що занадто багато, щоб показати на графіку. Замість цього я пропоную зробити кілька менших PSD і усереднити їх разом або відобразити за допомогою графіка спектрограми. Крім того, якщо ви знаєте, що ваш сигнал не змінюється швидко, достатньо використати кілька тисяч відліків і знайти PSD з них; за цей часовий проміжок у кілька тисяч відліків ви, ймовірно, захопите достатньо сигналу, щоб отримати гарне представлення.

Ось повний приклад коду, який включає генерування сигналу (комплексна експонента з частотою 50 Гц) і шуму. Зверніть увагу, що N, кількість відліків для імітації, стає довжиною ШПФ, оскільки ми беремо ШПФ всього імітованого сигналу.

```
import numpy as np
import matplotlib.pyplot as plt

Fs = 300 # частота дискретизації
Ts = 1/Fs # період дискретизації
N = 2048 # кількість відліків для моделювання

t = Ts*np.arange(N)
x = np.exp(1j*2*np.pi*50*t) # імітує синусоїду 50 Гц

n = (np.random.randn(N) + 1j*np.random.randn(N))/np.sqrt(2) # комплексний шум з одиничною
↳ потужністю
noise_power = 2
r = x + n * np.sqrt(noise_power)

PSD = np.abs(np.fft.fft(r))**2 / (N*Fs)
PSD_log = 10.0*np.log10(PSD)
PSD_shifted = np.fft.fftshift(PSD_log)

f = np.arange(Fs/-2.0, Fs/2.0, Fs/N) # старт, стоп, крок

plt.plot(f, PSD_shifted)
plt.xlabel("Частота [Гц]")
plt.ylabel("Амплітуда [дБ]")
plt.grid(True)
plt.show()
```

Виведення на екран:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/fft_example1.svg

2.14. Подальше читання

1. http://rfic.eecs.berkeley.edu/~niknejad/ee242/pdf/eecs242_lect3_rxarch.pdf

Розділ 3

Частотна область

Ця глава знайомить з частотною областю і охоплює ряди Фур'є, перетворення Фур'є, властивості Фур'є, ШПФ, вікна і спектрограми, використовуючи приклади на Python.

Одним з найцікавіших побічних ефектів вивчення DSP і бездротового зв'язку є те, що ви також навчитеся мислити в частотній області. Досвід "роботи" в частотній області для більшості людей обмежується регулюванням низьких/середніх/високих частот в автомобільній аудіосистемі. А "перегляд" чогось в частотній області для більшості людей обмежується переглядом динамічного звукового спектра еквайзера, як на анімації приведений нижче:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/audio_equalizer.webp

Наприкінці цього розділу ви зрозумієте, що насправді означає частотна область, як перетворювати часову в частотну область і навпаки (а також що при цьому відбувається), і деякі цікаві принципи, які ми будемо використовувати під час вивчення ЦОС (DSP) і ПКР (SDR). До кінця цього підручника ви гарантовано станете майстром у роботі з частотною областю!

По-перше, розберемось чому нам подобається розглядати сигнали в частотній області? Ось два приклади сигналів, показаних як в часовій, так і в частотній області.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/time_and_freq_domain_example_signals.png

Як ви можете бачити, в часовій області обидва сигнали виглядають як шум, але в частотній області ми бачимо різні особливості. В природі все знаходиться в часовій області; коли ми робимо оцифровку сигналів, ми робимо їх в часовій області, тому що ми не можемо *безпосередньо* зробити оцифровку сигналу в частотній області. Але найцікавіші речі зазвичай відбуваються саме в частотній області.

3.1. Ряди Фур'є

Основи для розуміння частотної області починається з того, що будь-який сигнал може бути представлений, як сума синусоїдальних хвиль. Коли ми розкладаємо сигнал на складові синусоїди, ми називаємо це рядом Фур'є. Ось приклад сигналу, який складається лише з двох синусоїд:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/summing_sinusoids.svg

Ось ще один приклад: червона крива нижче апроксимує пилкоподібну хвилю сумою 10 синусоїд. Ми бачимо, що це дає не ідеальне наближення - для відтворення цієї пилкоподібної хвилі знадобилася б нескінченна кількість синусоїд через те, що пилкоподібний сигнал має різкі стрибки:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/fourier_series_triangle.gif

Деякі сигнали вимагають більше синусоїд, ніж інші, а деякі вимагають нескінченної кількості, хоча і їх завжди можна апроксимувати скінченною кількістю синусоїд. Ось ще один приклад розбиття сигналу на серію синусоїд:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/fourier_series_arbitrary_function.gif

Щоб зрозуміти, як можна розкласти сигнал на синусоїдальні хвилі, або синусоїди, нам потрібно спочатку розглянути три властивості синусоїди:

1. Амплітуда
2. Частота
3. Фаза

Амплітуда вказує на "силу" хвилі, тоді як **частота** - це кількість хвиль в секунду. **Фаза** використовується для представлення того, як синусоїда зсунута у часі, в межах від 0 до 360 градусів (або від 0 до 2π), але фаза відносно значення, тобто щоб мати якесь значення фаза повинна бути виміряна відносно чогось. Наприклад, два сигнали з однаковою частотою можуть бути зсунуті на 30 градусів відносно один одного.

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/amplitude_phase_period.svg

Тепер ви зрозуміли, що "сигнал" - це, по суті, просто функція, зазвичай представлена "в часі" (тобто на осі x). Іншою її властивістю, яку легко запам'ятати, є **період**, який є оберненою величиною до **частоти**. Період синусоїди - це кількість часу в секундах, за який хвиля завершує один цикл. Таким чином, одиницею частоти є 1/секунда, або Гц.

Коли ми розкладаємо сигнал на суму синусоїд, кожна з них матиме певну **амплітуду**, **фазу** і **частоту**. Амплітуда кожної синусоїди покаже нам, наскільки є сильною складова **частоти** у сигналі. Не хвилюйтеся надто про **фазу** поки що, окрім усвідомлення того, що єдина різниця між $\sin()$ і $\cos()$ - це фазовий зсув (часовий зсув).

Важливіше зрозуміти концепцію, що лежить в основі, ніж самі рівняння, які потрібно розв'язати для отримання ряду Фур'є, але для тих, хто цікавиться рівняннями, я відсилаю вас до стислого пояснення Вольфрама: <https://mathworld.wolfram.com/FourierSeries.html>.

3.2. Часово-частотні пари

Ми з'ясували, що сигнали можуть бути представлені у вигляді синусоїд, які мають декілька властивостей. Тепер давайте навчимося будувати графіки сигналів у частотній області. У той час як часова область демонструє, як сигнал змінюється з часом, частотна область показує, яка частина сигналу на яких частотах знаходиться. Замість того, щоб по осі абсцис відкладати час, ми будемо відкладати частоту. Ми можемо побудувати графік заданого сигналу як по часу, так і по частоті. Для початку розглянемо кілька простих прикладів.

Ось як виглядає синусоїда з частотою f в часовій і частотній області:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/sine-wave.png

Часова область має виглядати дуже знайомо. Це коливальна функція. Не турбуйтеся про те, в якій точці періоду вона починається або як довго триває. Суть в тому, що сигнал має **єдину частоту**, тому в частотній області ми бачимо один пік. На якій би частоті не коливалася ця синусоїда, ми побачимо пік у частотній області. Математична назва такого піку називається "імпульс".

А що, якби ми мали імпульс у часовій області? Уявіть собі звукозапис від хлопавки або удару молотка по цвяху. Ця пара перетворення з час в частоту трохи менш інтуїтивно зрозуміла.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/impulse.png

Як ми бачимо, пік/імпульс у часовій області є пласким у частотній області, і теоретично він містить кожен частоту. Теоретично ідеального імпульсу не існує, оскільки він мав би бути нескінченно коротким у часовій області. Як і у випадку з синусоїдою, не має значення, де в часовій області відбувається імпульс. Важливим висновком тут є те, що швидкі зміни в часовій області призводять до виникнення багатьох частот.

Далі давайте подивимося на часові та частотні діаграми прямокутної хвилі:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/square-wave.svg

Цей графік також менш інтуїтивно зрозумілий, але ми бачимо, що в частотній області є сильний пік, який знаходиться на частоті прямокутної хвилі, але з підвищенням частоти піків стає більше. Це пов'язано зі швидкою зміною часової області, як і в попередньому прикладі. Але частота не рівномірна. Вона має піки через певні проміжки часу, і рівень повільно спадає (хоча ніколи теоретично не спадає до 0). Прямокутна хвиля в часовій області в частотній області буде мати вигляд $\sin(x)/x$ (так звана *sinc*-функція).

А що, якщо у нас є постійний сигнал у часовій області? Тоді постійний сигнал не має ніякої "частоти". Тобто графік буде наступний:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/dc-signal.png

Оскільки частота відсутня, у частотній області ми маємо імпульс на частоті 0 Гц. Це має фізичний сенс, якщо добре подумати про це. Частотна область не може бути "порожньою", тому що це відбувається лише тоді, коли немає зовсім сигналу (тобто, у часовій області 0). Назавемо сигнал з частотою 0 Гц у частотній області "постійним струмом", тому що він викликаний сигналом постійного струму в часі (тобто постійним сигналом, який не змінюється). Зауважте, що якщо ми збільшимо амплітуду нашого постійного сигналу в часовій області, імпульс на 0 Гц в частотній області також збільшиться.

Пізніше ми дізнаємося, які велечини відкладаються по вісі ординат графіку в частотній області, але поки що ви можете думати про ці велечини як про своєрідну амплітуду, частотних складових присутніх в сигналі в часовій області.

3.3. Перетворення Фур'є

Математично "перетворення", яке ми використовуємо для переходу з часової області в частотну і навпаки, називається перетворенням Фур'є. Воно визначається наступним чином:

$$X(f) = \int x(t)e^{-j2\pi ft} dt$$

Для сигналу $x(t)$ ми можемо отримати частотну версію $X(f)$, використовуючи цю формулу. Ми будемо позначати часову версію функції через $x(t)$ або $y(t)$, а відповідну частотну версію через $X(f)$ та $Y(f)$. Зверніть увагу, що "t" означає час, а "f" - частоту. "j" - це просто уявна одиниця. Ви могли бачити її як "i" на уроках математики в середній школі. Ми використовуємо "j" в інженерії та комп'ютерних науках, тому що "i" часто позначає струм, а в програмуванні часто використовується як ітератор.

Зворотнє перетворення з частотної в часову область відбувається майже так само, за винятком масштабного коефіцієнта та зміни знаку степені:

$$x(t) = \frac{1}{2\pi} \int X(f)e^{j2\pi ft} df$$

Зверніть увагу, що у багатьох підручниках та інших джерелах замість ω використовується $2\pi f$. ω - кутова частота у радіанах за секунду, тоді як f - у Гц. Все, що вам потрібно знати, це те, що

$$\omega = 2\pi f$$

Хоча це додає член 2π до багатьох рівнянь, простіше дотримуватися частоти у Гц. Зрештою, ви будете працювати з Гц у ваших SDR та RF програмних застосунках.

Наведене вище рівняння для перетворення Фур'є є у неперервній формі, яку ви побачите лише у математичних задачах. Дискретна форма цієї формули набагато ближча до того, що реалізується у кодї:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-j\frac{2\pi}{N}kn}$$

Зауважте, що основна відмінність полягає у тому, що ми замінили інтеграл на суму. Індекс k змінюється від 0 до $N-1$.

Нічого страшного, якщо жодне з цих рівнянь не має для вас особливого значення. Насправді нам не потрібно використовувати їх безпосередньо, щоб робити круті речі з DSP і SDR!

3.4. Часо-частотні властивості

Вище ми розглянули приклади того, як сигнали представляються в часовій і частотній областях. Зараз ми розглянемо п'ять важливих "властивостей Фур'є". Це властивості вказують нам, що якщо ми зробимо ____ з нашим сигналом у часовій області, то ____ станеться з цим сигналом у частотній області. Це дасть нам важливе розуміння типів цифрової обробки сигналів (ЦОС), які ми будемо виконувати з сигналами у часовій області на практиці.

1. Властивість лінійності:

$$ax(t) + by(t) \leftrightarrow aX(f) + bY(f)$$

Ця властивість, мабуть, найпростіша для розуміння. Якщо ми додаємо два сигнали в часі, то частотна версія також буде двома частотними сигналами, доданими разом. Вона також говорить нам, що якщо ми помножимо будь-який з них

на коефіцієнт масштабування, частотна область також масштабуватиметься на ту саму величину. Корисність цієї властивості стане більш очевидною, коли ми додамо разом кілька сигналів.

2. Властивість зсуву частоти:

$$e^{2\pi j f_0 t} x(t) \leftrightarrow X(f - f_0)$$

Член зліва від $x(t)$ - це те, що ми називаємо "комплексною синусоїдою" або "комплексною експонентою". Наразі, все, що нам потрібно знати, це те, що по суті це просто синусоїда з частотою f_0 . Ця властивість говорить нам, що якщо ми візьмемо сигнал $x(t)$ і помножимо його на синусоїду, то у частотній області ми отримаємо $X(f)$, тільки зсунутий на певну частоту, f_0 . Цей зсув частоти може бути легше візуалізувати:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/freq-shift.svg

Зсув частоти є невід'ємною властивістю ЦОС, оскільки з багатьох причин нам може знадобитися зсув сигналів вгору або вниз по частоті. Ця властивість показує нам, як це зробити (помножити на синусоїду). Ось ще один спосіб візуалізувати цю властивість:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/freq-shift-diagram.svg

3. Масштабування у властивості Time:

$$x(at) \leftrightarrow X\left(\frac{f}{a}\right)$$

У лівій частині рівняння ми бачимо, що ми масштабуємо наш сигнал $x(t)$ у часовій області. Ось приклад масштабування сигналу в часі, а потім те, що відбувається з частотними версіями кожного з них.

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/time-scaling.svg

Масштабування в часі, по суті, стискає або розширює сигнал по осі x . Ця властивість говорить нам про те, що масштабування в часовій області спричиняє зворотне масштабування в частотній області. Наприклад, коли ми передаємо біти швидше, ми повинні використовувати більшу пропускну здатність. Ця властивість допомагає пояснити, чому сигнали з вищою швидкістю передачі даних займають більшу смугу пропускання/спектр. Якби масштабування час-частота було пропорційним, а не обернено пропорційним, оператори стільникового зв'язку могли б передавати стільки біт в секунду, скільки вони хочуть, не платячи мільярди за спектр! На жаль, це не так.

Ті, хто вже знайомий з цією властивістю, можуть помітити відсутність масштабового коефіцієнта; він не враховується заради простоти. Для практичних цілей це не має значення.

4. Згортка по часу:

$$\int x(\tau)y(t-\tau)d\tau \leftrightarrow X(f)Y(f)$$

Вона називається згорткою по часу, тому що у часовій області ми згортаємо $x(t)$ та $y(t)$. Можливо, ви ще не знаєте про операцію згортки, тому поки уявіть її як крос-кореляцію, адже ми зануримося у згортки глибше у цьому розділі <convolution-section>. Коли ми згортаємо часові сигнали, це еквівалентно перемноженню частотних версій цих двох сигналів. Це дуже відрізняється від додавання двох сигналів. Коли ви додаєте два сигнали, як ми бачили, нічого насправді не відбувається, ви просто додаєте частотні версії. Але коли ви згортаєте два сигнали, ви ніби створюєте з них новий третій сигнал. Згортка - це найважливіша техніка в ЦОС, але для того, щоб повністю її зрозуміти, ми повинні спочатку зрозуміти, як працюють фільтри.

Перш ніж ми продовжимо, щоб коротко пояснити, чому ця властивість настільки важлива, розглянемо таку ситуацію: у вас є один сигнал, який ви хочете отримати, і поруч з ним є сигнал, що заважає.

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/two-signals.svg

Концепція маскуванню широко використовується у програмуванні, тому давайте використемо її тут. Що, якби ми могли створити маску наведену на рисунку нижче і помножити її на сигнал наведений вище, щоб замаскувати той, який нам не потрібен?

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/masking.svg

Зазвичай ми виконуємо операції ЦОС у часовій області, тому давайте скористаємося властивістю згортки, щоб побачити, як ми можемо зробити це маскуванню у часовій області. Скажімо, що $x(t)$ - це отриманий сигнал. Нехай $Y(f)$ - це маска, яку ми хочемо застосувати у частотній області. Це означає, що $y(t)$ є часовим представленням нашої маски, і якщо ми згорнемо її з $x(t)$, ми зможемо "відфільтрувати" небажаний сигнал.

[font=Largebfseriesfffamily] definecolor{babyblueeyes}{rgb}{0.36, 0.61, 0.83}
draw (0,0) node[align=center,babyblueeyes] {Наприклад, наш отриманий сигнал};
draw (0,-4) node[below, align=center,babyblueeyes] {Наприклад, маска}; draw
(0,-2) node[align=center,scale=2]{\$\int x(\tau)y(t-\tau)d\tau \leftrightarrow X(f)Y(f)\$};
draw[>,babyblueeyes,thick] (-4,0) -- (-5.5,-1.2); draw[>,babyblueeyes,thick]
(2.5,-0.5) -- (3,-1.3); draw[>,babyblueeyes,thick] (-2.5,-4) -- (-3.8,-2.8); draw[>,babyblueeyes,thick] (3,-4) -- (5.2,-2.8);

Коли ми будемо обговорювати фільтрацію, згортки матимуть більше сенсу.

5. Згортка по частоті:

Насамкінець, я хочу зазначити, що властивість згортки працює і у зворотному напрямку, хоча ми не будемо використовувати її так часто, як властивість згортки у часовій області:

$$x(t)y(t) \leftrightarrow \int X(g)Y(f-g)dg$$

Існують і інші властивості, але наведені вище п'ять, на мою думку, є найбільш важливими для розуміння. Навіть якщо ми не довели кожную з них, суть в тому, що ми використовуємо математичні властивості, щоб розуміти, що відбувається з реальними сигналами при аналізі та обробці. Не зациклюйтеся на рівняннях. Переконайтеся, що ви розумієте опис кожної властивості.

3.5. Швидке перетворення Фур'є (ШПФ)

Тепер повернемося до перетворення Фур'є. Я показав вам рівняння для дискретного перетворення Фур'є, але 99.9% часу ви будете використовувати під час кодування функцію ШПФ, `fft()`. Швидке перетворення Фур'є (ШПФ) - це просто алгоритм для обчислення дискретного перетворення Фур'є. Його було розроблено десятки років тому, і хоча існують різні варіанти реалізації, він все ще залишається лідером з обчислення дискретного перетворення Фур'є. Нам пощастило, що вони назвали його "швидким".

ШПФ - це функція з одним входом і одним виходом. Вона перетворює сигнал з часу в частоту:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: [../_images/fft-block-diagram.svg](#)

У цьому підручнику ми розглядатимемо лише одновимірні ШПФ (двовимірні використовуються для обробки зображень та інших застосувань). Для наших цілей вважатимемо, що функція ШПФ має один вхід: вектор відліків, і один вихід: частотну версію цього вектора відліків. Розмір виходу завжди дорівнює розміру входу. Якщо я подам на вхід ШПФ 1,024 відліки, я отримаю 1,024 на виході. Складність полягає в тому, що результат завжди буде в частотній області, а отже, "розмах" осі x , якщо ми побудуємо графік, не зміниться залежно від кількості відліків на вході в часовій області. Давайте візуалізуємо це, подивившись на вхідні та вихідні масиви разом з одиницями виміру їхніх індексів:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: [../_images/fft-io.svg](#)

Оскільки вихідні дані знаходяться в частотній області, значення діапазону по осі x залежить від частоти дискретизації, яку ми розглянемо в наступній главі. Коли ми використовуємо більше відліків для вхідного вектора, ми отримуємо кращу роздільну здатність у частотній області (як додаток до обробки більшої кількості відліків за один раз). Насправді ми не "бачимо" більше частот, маючи довший вхідний сигнал. Єдиний спосіб збільшити кількість частот - збільшити частоту дискретизації (зменшити період дискретизації Δt).

Як нам насправді відобразити цей вихідний результат ШПФ? Для прикладу припустимо, що наша частота дискретизації становить 1 мільйон відліків за секунду (1 МГц). Як ми дізнаємося з наступного розділу, це означає, що ми можемо бачити тільки сигнали з частотою до 0,5 МГц, незалежно від того, скільки відліків ми подаємо на ШПФ. Вихідні дані ШПФ можна представити наступним чином:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: [../_images/negative-frequencies.svg](#)

Це завжди так; на виході ШПФ завжди будуть значення від $-f_s/2$ до $f_s/2$, де f_s - частота дискретизації. Тобто на виході завжди буде від'ємна частина і додатна частина. Якщо вхідний сигнал комплексний, то від'ємна і додатна частини будуть відрізнятися, але якщо він дійсний, то вони будуть ідентичні.

Щодо частотного інтервалу, то кожен відлік відповідає f_s/N Гц, тобто подача більшої кількості відліків на кожне ШПФ призведе до більш деталізованої роздільної здатності на виході. Дуже незначна деталь, яку можна проігнорувати, якщо ви новачок: математично останній індекс не відповідає *точно* $f_s/2$, скоріше це $f_s/2 - f_s/N$, що для великого N буде приблизно дорівнювати $f_s/2$.

3.6. Від'ємні частоти

Що таке від'ємна частота? Наразі просто вважайте, що це пов'язано з використанням комплексних чисел (уявних чисел) - насправді не існує такого поняття, як "від'ємна частота", коли мова йде про передачу/прийом радіосигналів, це просто представлення, яке ми використовуємо. Можна думати про це наступним чином. Уявімо, що ми говоримо нашому SDR налаштуватися на частоту 100 МГц (FM-діапазон) і робити оцифровку з частотою дискретизації 10 МГц. Іншими словами, ми будемо бачити спектр від 95 МГц до 105 МГц. Незхай в цьому діапазоні присутні три сигнали:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/negative-frequencies2.svg

І, коли SDR зробить ШПФ ми отримаємо на виході:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/negative-frequencies3.svg

Пам'ятайте, що ми налаштували SDR на 100 МГц. Отже, сигнал, який був на частоті близько 97,5 МГц, у цифровому вираженні виглядає як -2,5 МГц, що технічно є від'ємною частотою. Насправді це просто частота, нижча за центральну частоту. Це матиме більше сенсу, коли ми дізнаємося більше про дискретизацію і отримаємо досвід використання наших SDR.

З математичної точки зору, негативні частоти можна уявити наступним чином, розглянемо комплексну експоненціальну функцію $e^{2j\pi ft}$. Якщо у нас від'ємна частота, то це еквівалентно тому, що в полярних координатах ця комплексна синусоїда обертається у протилежному напрямку.

$$e^{2j\pi ft} = \cos(2\pi ft) + j \sin(2\pi ft) \quad \text{blue}$$

$$e^{2j\pi(-f)t} = \cos(2\pi ft) - j \sin(2\pi ft) \quad \text{red}$$

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/negative_freq_animation.gif

Ми використали комплексну експоненту вище тому, що $\cos()$ або $\sin()$ містить як додатні, так і від'ємні частоти, як видно з формули Ейлера, застосованої до синусоїди на частоті f з часом t :

$$\begin{aligned} \cos(2\pi ft) &= \underbrace{\frac{1}{2}e^{2j\pi ft}}_{\text{positive}} + \underbrace{\frac{1}{2}e^{-2j\pi ft}}_{\text{negative}} \\ \sin(2\pi ft) &= \underbrace{\frac{1}{2j}e^{2j\pi ft}}_{\text{positive}} - \underbrace{\frac{1}{2j}e^{-2j\pi ft}}_{\text{negative}} \end{aligned}$$

Отже, в обробці радіочастотних сигналів ми зазвичай використовуємо комплексні експоненти замість косинусів і синусів.

3.7. Порядок в часі не має значення

Остання властивість перед тим, як ми перейдемо до ШПФ. Функція ШПФ ніби "перемішує" вхідний сигнал так, щоб сформувати вихідний сигнал, який має інший

масштаб і одиниці виміру. Після чого, ми більше не перебуваємо в часовій області. Гарний спосіб усвідомити цю різницю між областями - це усвідомити, що зміна порядку подій у часовій області не змінює частотні компоненти сигналу. Тобто, виконання одного ШПФ для приведених нижче на рисунку двох сигналів матиме однакові два піки, оскільки сигнал - це просто дві синусоїди на різних частотах. Зміна порядку виникнення синусоїд не змінює того факту, що це дві синусоїди на різних частотах. Це передбачає, що обидві синусоїди виникають в один і той самий проміжок часу, що подається на ШПФ; якщо скоротити розмір ШПФ та виконати кілька ШПФ (як ми зробимо в розділі "Спектрограма"), то можна розрізнити ці дві синусоїди.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/fft_signal_order.png

Технічно, фаза значень ШПФ зміниться через часовий зсув синусоїд. Однак у перших кількох розділах цього підручника нас цікавитиме здебільшого амплітуди ШПФ.

3.8. ШПФ у Python

Тепер, коли ми дізналися про те, що таке ШПФ і як представляється результат, давайте розглянемо код на Python і скористаємося функцією ШПФ Numpy, `np.fft.fft()`. Рекомендується використовувати повноцінну консоль/IDE Python на вашому комп'ютері, але в крайньому випадку ви можете скористатися веб-консоллю Python, посилання на яку знаходиться внизу навігаційної панелі зліва.

Спочатку нам потрібно створити сигнал у часовій області. Ви можете скористатися власною консоллю Python для відтворення прикладів. Для спрощення ми створимо просту синусоїду з частотою 0,15 Гц. Ми також будемо використовувати частоту дискретизації 1 Гц, тобто в часі ми будемо робити відліки через 0, 1, 2, 3 секунди і т.д.

```
import numpy as np
t = np.arange(100)
s = np.sin(0.15*2*np.pi*t)
```

Якщо ми побудуємо графік `s`, то він буде виглядати так:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/fft-python1.png

Далі скористаємося функцією ШПФ Numpy:

```
S = np.fft.fft(s)
```

Якщо ми подивимось на `S`, то побачимо, що це масив комплексних чисел:

```
S = array([-0.01865008 +0.00000000e+00j, -0.01171553 -2.79073782e-01j, 0.02526446
→ -8.82681208e-01j, 3.50536075 -4.71354150e+01j, -0.15045671 +1.31884375e+00j,
→ -0.10769903 +7.10452463e-01j, -0.09435855 +5.01303240e-01j, -0.08808671
→ +3.92187956e-01j, -0.08454414 +3.23828386e-01j, -0.08231753 +2.76337148e-01j,
→ -0.08081535 +2.41078885e-01j, -0.07974909 +2.13663710e-01j, .
```

Порада: незалежно від того, що ви робите, якщо ви зіткнулись з комплексними числами, спробуйте обчислити амплітуду і фазу і подивіться, може це додасть більше розуміння. Давайте так і зробимо, і побудуємо графік амплітуди і фази. У більшості мов для знаходження амплітуди комплексного числа є функція `abs()`. Функція для фази може бути різною, але у Python це `np.angle()`.

```
import matplotlib.pyplot as plt
S_mag = np.abs(S)
S_phase = np.angle(S)
plt.plot(t, S_mag, '-.')
plt.plot(t, S_phase, '-.')
```

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/fft-python2.png

Наразі ми не додаємо розмірності до вісі x для графіків, це лише індекс масиву (що рахується від 0). З математичних міркувань, вихідні дані після ШПФ мають наступний формат:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/fft-python3.svg

Але ми хотіли б мати 0 Гц (постійний струм) в центрі і від'ємні частоти зліва (це те як зазвичай ми представляємо графіки). Отже, кожного разу, коли ми робимо ШПФ, нам потрібно виконати "зсув ШПФ", який є простою операцією перегрупування масиву, на кшталт кільцевого зсуву, але більше схожого на "покладіть це сюди, а це туди". На наведеній нижче схемі повністю описано, що робить операція зсуву ШПФ:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/fft-python4.svg

Для зручності у Numpy є функція зсуву ШПФ, `np.fft.fftshift()`. Замініть рядок `np.fft.fft()` на:

```
S = np.fft.fftshift(np.fft.fft(s))
```

Нам також потрібно розібратися зі значеннями/мітками по осі x . Пам'ятайте, що ми використовували частоту дискретизації 1 Гц для спрощення. Це означає, що лівий край графіка частотної області буде -0,5 Гц, а правий - 0,5 Гц. Якщо що це поки незрозуміло, то стане зрозуміло після того, як ви прочитаєте розділ `sampling-chapter`. Давайте дотримуватися цього припущення, що наша частота дискретизації становить 1 Гц, і побудуємо графік амплітуди і фази вихідного сигналу ШПФ з відповідною міткою на осі абсцис. Ось остаточна версія цього прикладу на Python і результат:

```
import numpy as np
import matplotlib.pyplot as plt

Fs = 1 # Гц
N = 100 # кількість точок для моделювання та розмір нашого ШПФ

t = np.arange(N) # оскільки наша частота дискретизації 1 Гц
s = np.sin(0.15*2*np.pi*t)
S = np.fft.fftshift(np.fft.fft(s))
S_mag = np.abs(S)
S_phase = np.angle(S)
f = np.arange(Fs/-2, Fs/2, Fs/N)
plt.figure(0)
plt.plot(f, S_mag, '-.')
plt.figure(1)
plt.plot(f, S_phase, '-.')
plt.show()
```

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/fft-python5.png

Зверніть увагу, що ми бачимо наш пік на частоті 0.15 Гц, тобто на частоті, яку ми використовували при створенні синусоїди. Це означає, що наше ШПФ спрацювало! Якби ми не знали коду, який використовувався для генерації синусоїди, а нам просто дали список зразків, ми могли б використати ШПФ для визначення частоти. Причина, чому ми бачимо пік на частоті -0,15 Гц, пов'язана з тим, що це був дійсний, а не комплексний сигнал, і цей випадок ми глибше розглянемо пізніше.

3.9. Віконна функція

Коли ми використовуємо ШПФ для вимірювання частотних складових нашого сигналу, ми припускаємо, що ШПФ обробляє тільки фрагмент *періодичного* сигналу. Тобто результат ШПФ на фрагменті такий, ніби поданий сигнал продовжується далі повторюватися до нескінченності. Тобто останній відлік фрагменту має з'єднуватися з першим відліком наступного фрагменту, але так як фрагмент циклічно повторюється - то це ж і є першим відліком нашого фрагменту. Це впливає з теорії, що лежить в основі перетворення Фур'є. Це для нас означає, що ми не хочемо різких перепадів амплітуди між цим останім і першим відліком, бо різкі перепади в часовій області приведуть в частотній області до виникнення багатьох додаткових частот, але насправді для довільного сигналу наш останній і перший відлік можуть не так плавно з'єднуватися. Простіше кажучи, якщо ми робимо ШПФ зі 100 відліків, використовуючи `np.fft.fft(x)`, ми хочемо, щоб `x[0]` і `x[99]` були рівними або близькими за значенням.

Для того щоб зкомпенсувати цей перепад при циклічному повторенні фрагменту ми використовуємо "віконну функцію". Безпосередньо перед операцією ШПФ ми множимо фрагмент сигналу на віконну функцію, яка дорівнює довжині фрагменту і може бути будь-якою функцією, що спадає до нуля на обох кінцях. Це гарантує, що фрагмент сигналу буде починатися і закінчуватися в нулі і таким чином плавно з'єднуватися. До поширених віконних функцій належать функції Геммінга, Ганнінга, Блекмана та Кайзера. Якщо ви не застосовуєте жодної віконної функції, це називається використанням "прямокутного" вікна, тому що це еквівалентно множенню фрагменту на масив одиниць. Ось як виглядають деякі віконні функції:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: [../_images/windows.svg](#)

Простим підходом для початківців є використання вікна Гамінга, яке можна створити у Python за допомогою `np.hamming(N)`, де `N` - це кількість елементів у масиві, що є розміром вашого фрагменту для ШПФ. У наведеній вище вправі ми застосуємо вікно безпосередньо перед ШПФ. Після 2-го рядка в коді ми б вставили

```
s = s * np.hamming(100)
```

Якщо ви боїтеся обрати неправильне вікно, не бійтеся. При використанні різниці між Hamming, Hanning, Blackman і Kaiser вікнами дуже мінімальна у порівнянні з тем ефектом, що буде якщо взагалі їх не використати, оскільки всі вони знижують значення амплітуд фрагментів до нуля з обох боків і вирішують основну проблему плавного з'єднання.

3.10. Визначення розміру ШПФ

Останнє, на що слід звернути увагу - це розмір ШПФ. Через спосіб реалізації алгоритму ШПФ - найкращий розмір фрагменту сигналу ШПФ завжди має дорівнювати числу, що є порядком 2. Ви можете використовувати розмір, який не є порядком 2, але при цьому алгоритм буде працювати повільніше. Найпоширеніші розміри виборки сигналу - від 128 до 4096, хоча, звичайно, можна використовувати і вибірки більшого розміру. На практиці нам, можливо, доведеться обробляти сигнали довжиною в мільйони або мільярди відліків, тому нам потрібно розбити сигнал на фрагменти і виконати над ними багато операцій ШПФ. Це означає, що ми отримаємо багато результатів ШПФ. Далі ми можемо або усереднити їх, або будувати графік частот що змінюється з часом (це особливо зручно якщо наш сигнал змінюється з часом). Вам не обов'язково пропускати через ШПФ *кожний* відлік сигналу, щоб отримати гарне представлення цього сигналу в частотній області. Наприклад, ви можете застосувати ШПФ лише до 1,024 з кожних 100 тис. відліків сигналу, і отримані частоти, ймовірно, будуть добре відповідати спектру, якщо ви маєте неперервний сигнал.

3.11. Спектрограма/Водоспад

Спектрограма - це графік, який показує зміну амплітуд частот з часом. Це просто набір результатів ШПФ, складений разом (по вертикалі, якщо вам потрібна частота на горизонтальній осі). Ми також можемо показати його в реальному часі, часто його називають водоспадом. Аналізатор спектру - це прилад, який показує цю спектрограму/водоспад. На схемі нижче показано, як розділяють масив відліків IQ на зрізи так, щоб сформувати спектрограму:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/spectrogram_diagram.svg

Оскільки спектрограма передбачає побудову двовимірних даних від часу то, вона фактично є тривимірним графіком, і щоб представити ці значення частот від часу на двовимірному графіку нам треба додатково використати кольорову карту. Нижче наведений приклад спектрограми, з частотою по горизонтальній осі X і часом по вертикальній осі Y. Синій колір відповідає найнижчому рівню енергії, а червоний - найвищому. Ми бачимо, що в центрі графіку є сильний пік амплітуди постійного струму (0 Гц) навколо якого є змінні частоти. Синій колір представляє наш рівень шуму.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/waterfall.png

Пам'ятайте, що це просто ШПФ від зрізів з нашого сигналу, накладені один на одного, кожен ряд - це одна операція ШПФ над фрагментом (технічно, амплітуди частот отримані після ШПФ). Тому не забудьте розбити вхідний сигнал на зрізи, розмір яких дорівнює розміру вашого ШПФ (наприклад, 1024 відліків на зріз). Перш ніж перейти до коду для створення спектрограми, наведемо приклад сигналу, який ми будемо використовувати, це просто тон з накладеним на нього білим шумом:

```
import numpy as np
import matplotlib.pyplot as plt

sample_rate = 1e6
```



```
# Згенерувати тон плюс шум
t = np.arange(1024*1000)/sample_rate # вектор часу
f = 50e3 # частота тону
x = np.sin(2*np.pi*f*t) + 0.2*np.random.randn(len(t))
```

Ось як це виглядає в часовій області (перші 200 відліків):

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/spectrogram_time.svg

Ось як з Python ми можемо згенерувати його спектрограму:

```
# імітуємо сигнал вище, або використовуємо свій власний сигнал

fft_size = 1024
num_rows = len(x) // fft_size # // цілочисельне ділення, яке округлюється вниз
spectrogram = np.zeros((num_rows, fft_size))
for i in range(num_rows):
    spectrogram[i,:] =
        ↪ 10*np.log10(np.abs(np.fft.fftshift(np.fft.fft(x[i*fft_size:(i+1)*fft_size])))**2)

plt.imshow(spectrogram, aspect='auto', extent = [sample_rate/-2/1e6, sample_rate/2/1e6,
        ↪ 0, len(x)/sample_rate])
plt.xlabel("Частота [МГц]")
plt.ylabel("Час [с]")
plt.show()
```

У результаті ми отримаємо наступну спектрограму, яка не дуже цікава, оскільки немає жодних змін частот у часі. Тут є два тон-сигнали, так як ми моделюємо дійсний сигнал, а дійсні сигнали завжди мають рівні але дзеркально відображені позитивну і негативну частину у своєму спектрі щільності потужності (PSD). Більше цікавих прикладів спектрограм можна знайти на сайті <https://www.IQEngine.org>!

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/spectrogram.svg

3.12. Реалізація ШПФ

Незважаючи на те, що NumPy вже реалізує ШПФ, корисно знати основи того, як він працює під капотом. Найпопулярнішим алгоритмом ШПФ є алгоритм ШПФ Кулі-Тьюкі, вперше винайдений близько 1805 року Карлом Фрідріхом Гаусом, а потім перевідкритий і популяризований Джеймсом Кулі і Джоном Тьюкі в 1965 році.

Базова версія цього алгоритму працює на виборках розміру степені двійки і призначена для комплексних вхідних даних, але також може працювати і з дійсними вхідними даними. Будівельний блок цього алгоритму відомий як "метелик", який по суті є ШПФ розміру $N = 2$, що складається з двох множень і двох підсумовувань:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/butterfly.svg

або

$$y_0 = x_0 + x_1 w_N^k$$

$$y_1 = x_0 - x_1 w_N^k$$

де $w_N^k = e^{j2\pi k/N}$ це комплексний коефіцієнт обертання (N - розмір під-шПФ, а k - індекс). Зауважте, що вхідні та вихідні дані мають бути комплексними, наприклад, x_0 може бути $0.6123 - 0.5213j$, і суми/множники є також комплексними.

Алгоритм є рекурсивним і розбивається навпіл, поки не залишиться лише серія метеликів, дивись нижче БПФ розміру 8:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: [../_images/butterfly2.svg](#)

Кожен стовпчик у цьому шаблоні є набором операцій, які можна виконувати паралельно, за $\log_2(N)$ кроків, тому обчислювальна складність ШПФ становить $O(N \log N)$, тоді як ДПФ - $O(N^2)$.

Для тих, хто краще розуміє код, ніж рівняння, нижче наведено просту реалізацію ШПФ на Python, а також приклад сигналу, що складається з тону і шуму, на якому можна спробувати цю реалізацію ШПФ.

```
import numpy as np
import matplotlib.pyplot as plt

def fft(x):
    N = len(x)
    if N == 1:
        return x
    twiddle_factors = np.exp(-2j * np.pi * np.arange(N/2) / N)
    x_even = fft(x[::2]) # ура рекурсії!
    x_odd = fft(x[1::2])
    return np.concatenate([x_even + twiddle_factors * x_odd,
                           x_even - twiddle_factors * x_odd])

# Імітуємо тон + шум
sample_rate = 1e6
f_offset = 0.2e6 # Зсув від несучої на 200 кГц
N = 1024
t = np.arange(N)/sample_rate
s = np.exp(2j*np.pi*f_offset*t)
n = (np.random.randn(N) + 1j*np.random.randn(N))/np.sqrt(2) # одиничний комплексний шум
r = s + n # 0 dB SNR

# Виконати fft, fftshift, перевести в дБ
X = fft(r)
X_shifted = np.roll(X, N/2) # еквівалентно np.fft.fftshift
X_mag = 10*np.log10(np.abs(X_shifted)**2)

# Виведення результатів на екран
f = np.linspace(sample_rate/-2, sample_rate/2, N)/1e6 # plt у МГц
plt.plot(f, X_mag)
plt.plot(f[np.argmax(X_mag)], np.max(X_mag), 'rx') # показати max
plt.grid()
plt.xlabel('Частота [МГц]')
plt.ylabel('Амплітуда [дБ]')
plt.show()
```

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: [../_images/fft_in_python.svg](#)

Для тих, хто цікавиться реалізаціями на JavaScript та/або WebAssembly, зверніть увагу на бібліотеку [WebFFT](#) для виконання ШПФ у веб- або NodeJS-додатках,

вона містить кілька реалізацій, а також інструмент [benchmarking tool](#) для порівняння продуктивності кожної реалізації.

Розділ 4

Фільтри

У цій главі ми дізнаємося про цифрові фільтри за допомогою Python. Ми розглянемо типи фільтрів (FIR/IIR та низькочастотні/високочастотні/смугові/смугоподібні), способи представлення фільтрів у цифровому вигляді, а також їхню конструкцію. Ми закінчуємо вступом до формування імпульсів, який ми розглянемо в розділі `pulse-shaping-chapter`.

4.1. Основи фільтрів

Фільтри використовуються у багатьох дисциплінах. Наприклад, обробка зображень широко використовує двовимірні фільтри, де входними і вихідними даними є зображення. Ви можете щоранку використовувати фільтр для приготування кави, який відфільтровує тверді частинки від рідини. В DSP фільтри в першу чергу використовуються для

1. Розділення об'єднаних сигналів (наприклад, для виділення потрібного вам сигналу)
2. Видалення надлишкового шуму після отримання сигналу
3. Відновлення сигналів, які були певним чином спотворені (наприклад, звуковий еквайзер є фільтром)

Безумовно, є й інші способи використання фільтрів, але ця глава призначена для того, щоб представити концепцію, а не пояснити всі способи, якими може відбуватися фільтрація.

Ви можете подумати, що нас цікавлять лише цифрові фільтри, адже цей підручник вивчає DSP. Однак важливо знати, що багато фільтрів будуть аналоговими, як ті, що стоять у наших SDR, розміщених перед аналого-цифровим перетворювачем (АЦП) на стороні приймача. На наступному зображенні порівнюється схема аналогового фільтра з блок-схемою, що представляє алгоритм цифрової фільтрації.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: `../_images/analog_digital_filter.png`

У DSP, де вхід і вихід є сигналами, фільтр має один вхідний сигнал і один вихідний сигнал:

```
[font=sffamilyLarge, scale=2] definecolor{babyblueeyes}{rgb}{0.36, 0.61, 0.83} node [draw, color=white, fill=babyblueeyes, minimum width=4cm, minimum height=2.4cm ] (filter) {Filter}; draw[<, very thick] (filter.west) -- ++(-2,0) node[left,align=center]{Input\(\text{time domain}\)} ; draw[->, very thick] (filter.east) -- ++(2,0) node[right,align=center]{Output\(\text{time domain}\)};
```

Ви не можете подати два різних сигнали на один фільтр, не склавши їх попередньо або не виконавши якусь іншу операцію. Аналогічно, на виході завжди буде один сигнал, тобто одномірний масив чисел.

Існує чотири основні типи фільтрів: низькочастотні, високочастотні, смугові та режекторні. Кожен тип модифікує сигнали, фокусуючись на різних діапазонах частот всередині них. Наведені нижче графіки демонструють, як частоти в сигналах фільтруються для кожного типу, спочатку представлені лише додатні частоти (простіше для розуміння), а потім також і від'ємні.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/filter_types.png

```
[font=sffamilylarge] draw[->, thick] (-5,0) -- (5,0) node[below]{Frequency}; draw[->, thick] (0,-0.5) node[below]{0 Hz} -- (0,5) node[left=1cm]{textbf{Low-Pass}}; draw[red, thick, smooth] plot[tension=0.5] coordinates{(-5,0) (-2.5,0.5) (-1.5,3) (1.5,3) (2.5,0.5) (5,0)};
```

```
[font=sffamilylarge] draw[->, thick] (-5,0) -- (5,0) node[below]{Frequency}; draw[->, thick] (0,-0.5) node[below]{0 Hz} -- (0,5) node[left=1cm]{textbf{High-Pass}}; draw[red, thick, smooth] plot[tension=0.5] coordinates{(-5,3) (-2.5,2.5) (-1.5,0.3) (1.5,0.3) (2.5,2.5) (5,3)};
```

```
[font=sffamilylarge] draw[->, thick] (-5,0) -- (5,0) node[below]{Frequency}; draw[->, thick] (0,-0.5) node[below]{0 Hz} -- (0,5) node[left=1cm]{textbf{Band-Pass}}; draw[red, thick, smooth] plot[tension=0.5] coordinates{(-5,0) (-4.5,0.3) (-3.5,3) (-2.5,3) (-1.5,0.3) (1.5, 0.3) (2.5,3) (3.5, 3) (4.5,0.3) (5,0)};
```

```
[font=sffamilylarge] draw[->, thick] (-5,0) -- (5,0) node[below]{Frequency}; draw[->, thick] (0,-0.5) node[below]{0 Hz} -- (0,5) node[left=1cm]{textbf{Band-Stop}}; draw[red, thick, smooth] plot[tension=0.5] coordinates{(-5,3) (-4.5,2.7) (-3.5,0.3) (-2.5,0.3) (-1.5,2.7) (1.5, 2.7) (2.5,0.3) (3.5, 0.3) (4.5,2.7) (5,3)};
```

Кожен фільтр дозволяє певним частотам залишатися в сигналі, блокуючи інші частоти. Діапазон частот, які пропускає фільтр, називається "смугою пропускання", а "смуга затримки" - це те, що блокується. У випадку низькочастотного фільтра, він пропускає низькі частоти і затримує високі, тому 0 Гц завжди буде в смузі пропускання. Для фільтрів високих частот і смугових фільтрів 0 Гц завжди буде в смузі затримки.

Не плутайте ці типи фільтрації з алгоритмічною реалізацією фільтра (наприклад, IIR vs FIR). Найпоширенішим типом на сьогоднішній день є фільтр нижніх частот (ФНЧ), оскільки ми часто представляємо сигнали в базовій смузі. ФНЧ дозволяє нам відфільтрувати все "навколо" нашого сигналу, видаляючи надлишковий шум та інші сигнали.

4.2. Представлення фільтрів

Для більшості фільтрів, які ми побачимо (відомих як фільтри типу FIR, або фільтри зі скінченною імпульсною характеристикою), ми можемо представити сам фільтр за допомогою одного масиву плаваючих елементів. Для симетричних у частотній області фільтрів ці числа будуть дійсними (а не комплексними), і їхня кількість, як правило, буде непарною. Ми називаємо цей масив "відгалуженнями фільтра". Ми часто використовуємо h як символ для позначення відводів фільтра. Ось приклад набору відгалужень фільтра, які визначають один фільтр:

```
h = [ 9.92977939e-04  1.08410297e-03  8.51595307e-04  1.64604862e-04
      -1.01714338e-03 -2.46268845e-03 -3.58236429e-03 -3.55412543e-03
      -1.68583512e-03  2.10562324e-03  6.93100252e-03  1.09302641e-02
      1.17766532e-02  7.60955496e-03 -1.90555639e-03 -1.48306750e-02
```

```
-2.69313236e-02 -3.25659606e-02 -2.63400086e-02 -5.04184562e-03
3.08099470e-02 7.64264738e-02 1.23536693e-01 1.62377258e-01
1.84320776e-01 1.84320776e-01 1.62377258e-01 1.23536693e-01
7.64264738e-02 3.08099470e-02 -5.04184562e-03 -2.63400086e-02
-3.25659606e-02 -2.69313236e-02 -1.48306750e-02 -1.90555639e-03
7.60955496e-03 1.17766532e-02 1.09302641e-02 6.93100252e-03
2.10562324e-03 -1.68583512e-03 -3.55412543e-03 -3.58236429e-03
-2.46268845e-03 -1.01714338e-03 1.64604862e-04 8.51595307e-04
1.08410297e-03 9.92977939e-04]
```

4.2.1. Example Use-Case

Щоб дізнатися, як використовуються фільтри, давайте розглянемо приклад, де ми налаштуємо наш SDR на частоту існуючого сигналу і хочемо ізолювати його від інших сигналів. Пам'ятайте, що ми вказуємо нашому SDR, на яку частоту налаштуватися, але зразки, які він захоплює, знаходяться в базовій смузі, тобто сигнал буде відображатися з центром близько 0 Гц. Нам доведеться відстежувати, на яку частоту ми сказали SDR налаштуватися. Ось що ми можемо отримати:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/filter_use_case.png

Оскільки наш сигнал вже відцентровано на постійному струмі (0 Гц), ми знаємо, що нам потрібен фільтр нижніх частот. Ми повинні вибрати "частоту зрізу" (так звану кутову частоту), яка визначатиме, коли смуга пропускання переходить в смугу зупинки. Частота зрізу завжди буде в одиницях Гц. У цьому прикладі 3 кГц здається хорошим значенням:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/filter_use_case2.png

Однак, за принципом роботи більшості фільтрів нижніх частот, межа від'ємних частот також буде -3 кГц. Тобто, вона симетрична відносно постійного струму (пізніше ви зрозумієте чому). Наші частоти зрізу виглядатимуть приблизно так (смуга пропускання - це область між ними):

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/filter_use_case3.png

Після створення та застосування фільтра з частотою зрізу 3 кГц, маємо:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/filter_use_case4.png

Цей відфільтрований сигнал виглядатиме заплутано, доки ви не згадаєте, що наш рівень шуму був на зеленій лінії біля -65 дБ. Незважаючи на те, що ми все ще бачимо сигнал перешкод з центром на частоті 10 кГц, ми *значно* зменшили потужність цього сигналу. Тепер вона нижча за рівень шуму! Ми також видалили більшу частину шуму, що існував у смузі зупинки.

На додаток до частоти зрізу, інший основний параметр нашого фільтра низьких частот називається "ширина перехідного процесу". Ширина переходу, яка також вимірюється в Гц, вказує фільтру, як швидко він повинен перейти від смуги пропускання до смуги зупинки, оскільки миттєвий перехід неможливий.

Давайте візуалізуємо ширину перехідного процесу. На діаграмі нижче зелена лінія представляє ідеальну реакцію для переходу між смугою пропускання і смугою зупинки, яка, по суті, має ширину переходу, рівну нулю. Червона лінія демонструє результат реалістичного фільтра, який має деяку пульсацію і певну ширину перехідного процесу.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/realistic_filter.png

Вам може бути цікаво, чому ми просто не встановили ширину переходу якомога меншою. Причина в тому, що менша ширина перехідного процесу призводить до більшої кількості перемикачів, а більша кількість перемикачів означає більше обчислень - незабаром ми побачимо, чому. Фільтр на 50 відгалужень може працювати цілий день, використовуючи 1% процесора Raspberry Pi. Тим часом, фільтр на 50 000 відводів призведе до того, що ваш процесор вибухне!

Вам може бути цікаво, чому ми просто не встановили якомога меншу ширину переходу. Причина в тому, що менша ширина переходу призводить до більшої кількості переходів, а більша кількість переходів означає більше обчислень - незабаром ми побачимо, чому. Фільтр на 50 відгалужень може працювати цілий день, використовуючи 1% процесора Raspberry Pi. Тим часом, фільтр на 50 000 відводів призведе до того, що ваш процесор вибухне! Зазвичай ми використовуємо інструмент для створення фільтрів, потім дивимося, скільки відгалужень він видає, і якщо їх занадто багато (наприклад, більше 100), ми збільшуємо ширину переходу. Звичайно, все залежить від програми та обладнання, на якому працює фільтр.

У наведеному вище прикладі фільтрації ми використовували відсічення 3 кГц і ширину переходу 1 кГц (на цих скріншотах важко визначити ширину переходу). Отриманий фільтр мав 77 відводів.

Повернемося до представлення фільтрів. Незважаючи на те, що ми можемо показати список відгалужень для фільтра, ми зазвичай представляємо фільтри візуально в частотній області. Ми називаємо це "частотною характеристикою" фільтра, і вона показує нам поведінку фільтра в частотній області. Ось частотна характеристика фільтра, який ми щойно використовували:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/filter_use_case5.png

Зауважте, що те, що я показую тут, не є сигналом - це лише представлення фільтра у частотній області. Спочатку це може бути трохи важко зрозуміти, але коли ми подивимося на приклади і код, все стане зрозумілим.

Даний фільтр також має представлення в часовій області; його називають "імпульсною характеристикою" фільтра, тому що це те, що ви побачите в часовій області, якщо візьмете імпульс і пропустите його через фільтр. (Щоб дізнатися більше про те, що таке імпульс, погугліть "дельта-функція Дірака"). Для фільтра типу KIX імпульсна характеристика - це просто самі відгалуження. Для фільтра з 77 відгалуженнями, який ми використовували раніше, відгалуженнями є..:

```
h = [-0.00025604525581002235, 0.00013669139298144728, 0.0005385575350373983,
0.0008378280326724052, 0.000906112720258534, 0.0006353431381285191,
-9.884083502996931e-19, -0.0008822851814329624, -0.0017323142383247614,
-0.0021665366366505623, -0.0018335371278226376, -0.0005912294145673513,
0.001349081052467227, 0.0033936649560928345, 0.004703888203948736,
0.004488115198910236, 0.0023609865456819534, -0.0013707970501855016,
-0.00564080523326993, -0.008859002031385899, -0.009428252466022968,
-0.006394983734935522, 4.76480351940553e-18, 0.008114570751786232,
0.015200719237327576, 0.018197273835539818, 0.01482443418353796,
0.004636279307305813, -0.010356673039495945, -0.025791890919208527,
-0.03587324544787407, -0.034922562539577484, -0.019146423786878586,
0.011919975280761719, 0.05478153005242348, 0.10243935883045197,
0.1458890736103058, 0.1762896478176117, 0.18720689415931702,
0.1762896478176117, 0.1458890736103058, 0.10243935883045197,
0.05478153005242348, 0.011919975280761719, -0.019146423786878586,
-0.034922562539577484, -0.03587324544787407, -0.025791890919208527,
-0.010356673039495945, 0.004636279307305813, 0.01482443418353796,
0.018197273835539818, 0.015200719237327576, 0.008114570751786232,
4.76480351940553e-18, -0.006394983734935522, -0.009428252466022968,
```

```
-0.008859002031385899, -0.00564080523326993, -0.0013707970501855016,
0.0023609865456819534, 0.004488115198910236, 0.004703888203948736,
0.0033936649560928345, 0.001349081052467227, -0.0005912294145673513,
-0.0018335371278226376, -0.0021665366366505623, -0.0017323142383247614,
-0.0008822851814329624, -9.884083502996931e-19, 0.0006353431381285191,
0.000906112720258534, 0.0008378280326724052, 0.0005385575350373983,
0.00013669139298144728, -0.00025604525581002235]
```

І хоча ми ще не перейшли до дизайну фільтрів, ось код на Python, який згенерував цей фільтр:

```
import numpy as np
from scipy import signal
import matplotlib.pyplot as plt

num_taps = 51 # it helps to use an odd number of taps
cut_off = 3000 # Hz
sample_rate = 32000 # Hz

# create our low pass filter
h = signal.firwin(num_taps, cut_off, fs=sample_rate)

# plot the impulse response
plt.plot(h, '-.')
plt.show()
```

Простий графік цього масиву з плаваючими числами дає нам імпульсну характеристику фільтра:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/impulse_response.png

А ось код, який було використано для створення частотної характеристики, показаної раніше. Він трохи складніший, оскільки нам потрібно створити масив частот по осі x .

```
# будуємо частотну характеристику
H = np.abs(np.fft.fft(h, 1024)) # взяти 1024-точкове ШПФ та амплітуду
H = np.fft.fftshift(H) # робимо 0 Гц в центрі
w = np.linspace(-sample_rate/2, sample_rate/2, len(H)) # вісь x
plt.plot(w, H, '-.')
plt.show()
```

4.2.2. Реальні та комплексні фільтри

Фільтр, який я вам показав, має реальні відводи, але відводи можуть бути і складними. Реальні чи комплексні відводи не обов'язково повинні відповідати сигналу, який ви пропускаєте через них, тобто ви можете пропустити комплексний сигнал через фільтр з реальними відводами і навпаки. Коли відводи реальні, частотна характеристика фільтра буде симетричною навколо постійного струму (0 Гц). Зазвичай ми використовуємо комплексні відводи, коли нам потрібна асиметрія, що трапляється не дуже часто.

```
[font=sffamilyLarge,scale=2] definecolor{babyblueeyes}{rgb}{0.36, 0.61, 0.83}
draw[>,thick] (-5,0) node[below]{$-\frac{f_s}{2}$} -- (5,0) node[below]{$\frac{f_s}{2}$};
draw[>,thick] (0,-0.5) node[below]{0 Hz} -- (0,1); draw[babyblueeyes, smooth,
line width=3pt] plot[tension=0.1] coordinates{(-5,0) (-1,0) (-0.5,2) (0.5,2) (1,0) (5,0)};
draw[>,thick] (6,0) node[below]{$-\frac{f_s}{2}$} -- (16,0) node[below]{$\frac{f_s}{2}$};
draw[>,thick] (11,-0.5) node[below]{0 Hz} -- (11,1); draw[babyblueeyes, smooth, line
width=3pt] plot[tension=0] coordinates{(6,0) (11,0) (11,2) (11.5,2) (12,0) (16,0)};
```


`draw[font=hugebfseries] (0,2.5) node[above,align=center]{Example Low-Pass Filter\with Real Taps}; draw[font=hugebfseries] (11,2.5) node[above,align=center]{Example Low-Pass Filter\with Complex Taps};`

Як приклад складних відгалужень, повернімося до прикладу фільтрації, за винятком того, що цього разу ми хочемо отримати інший сигнал, що заважає (без необхідності переналаштування радіоприймача). Це означає, що нам потрібен смуговий фільтр, але не симетричний. Ми хочемо залишити (так званий "пропуск") лише частоти від 7 до 13 кГц (ми не хочемо також пропускати від -13 кГц до -7 кГц):

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/filter_use_case6.png

Один із способів створити такий фільтр - це створити фільтр нижніх частот з частотою зрізу 3 кГц, а потім зсунути його за частотою. Пам'ятайте, що ми можемо зсунути частоту $x(t)$ (у часовій області), помноживши її на $e^{j2\pi f_0 t}$. У цьому випадку f_0 має дорівнювати 10 кГц, що зсуває наш фільтр на 10 кГц вгору. Нагадаємо, що в нашому Python-коді зверху h було відгалуженням фільтра нижніх частот. Для того, щоб створити наш смуговий фільтр, нам просто потрібно помножити ці відводи на $e^{j2\pi f_0 t}$, хоча це передбачає створення вектора для представлення часу на основі нашого періоду дискретизації (оберненого до частоти дискретизації):

```
# (h було знайдено за допомогою першого фрагмента коду)

# Зсуваємо фільтр за частотою множенням на exp(j*2*pi*f0*t)
f0 = 10e3 # величина, на яку будемо зсувати
Ts = 1.0/sample_rate # період дискретизації
t = np.arange(0.0, Ts*len(h), Ts) # вектор часу. args are (start, stop, step)
exponential = np.exp(2j*np.pi*f0*t) # це по суті комплексна синусоїда

h_band_pass = h * exponential # робимо зсув

# будуємо графік імпульсної характеристики
plt.figure('impulse')
plt.plot(np.real(h_band_pass), '-.')
plt.plot(np.imag(h_band_pass), '-.')
plt.legend(['real', 'imag'], loc=1)

# будуємо частотну характеристику
N = np.abs(np.fft.fft(h_band_pass, 1024)) # взяти 1024-точкове ШПФ та амплітуду
N = np.fft.fftshift(N) # робимо 0 Гц по центру
w = np.linspace(-sample_rate/2, sample_rate/2, len(N)) # вісь x
plt.figure('freq')
plt.plot(w, N, '-.')
plt.xlabel('Frequency [Hz]')
plt.show()
```

Нижче наведено графіки імпульсної та частотної характеристик:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/shifted_filter.png

Оскільки наш фільтр не симетричний відносно 0 Гц, він повинен використовувати складні відгалуження. Тому нам потрібні дві лінії для відображення цих складних відгалужень. Те, що ми бачимо на лівому графіку вище, все ще є імпульсною характеристикою. Наш графік частотної характеристики - це те, що дійсно підтверджує, що ми створили саме такий фільтр, на який сподівалися, і що він відфільтрує все, окрім сигналу з центром близько 10 кГц. Знову ж таки, пам'ятайте, що графік вище - це *не* реальний сигнал: це лише представлення фільтра. Це може бути дуже складно зрозуміти, тому що коли ви застосовуєте фільтр до сигналу

і будувати графік вихідного сигналу в частотній області, в багатьох випадках він буде виглядати приблизно так само, як і частотна характеристика самого фільтра.

Якщо цей підрозділ додав плутанини, не хвилюйтеся, 99% часу ви все одно матимете справу з простими фільтрами нижніх частот з реальними відгалуженнями.

4.3. Реалізація фільтрів

Ми не будемо заглиблюватися в реалізацію фільтрів. Я зосереджуся на проектуванні фільтрів (ви можете знайти готову до використання реалізацію на будь-якій мові програмування). Наразі, ось один висновок: щоб відфільтрувати сигнал за допомогою КІХ-фільтра, ви просто згорнете імпульсну характеристику (масив відводів) з вхідним сигналом. (Не хвилюйтеся, згортання буде описано в наступному розділі.) У дискретному світі ми використовуємо дискретне згортання (приклад нижче). Трикутники, позначені як b - це відводи. На блок-схемі квадрати, позначені z^{-1} над трикутниками, означають затримку на один часовий крок.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/discrete_convolution.png

Ви можете зрозуміти, чому ми називаємо ці фільтри "відгалуженнями", виходячи з того, як реалізовано сам фільтр.

4.3.1. FIR vs IIR

Існує два основних класи цифрових фільтрів: КІХ та ІКХ

1. Скінченна імпульсна характеристика (FIR)
2. Нескінченна імпульсна характеристика (IIR)

Ми не будемо заглиблюватися в теорію, але поки що просто запам'ятайте: FIR-фільтри легше спроектувати і вони можуть робити все, що завгодно, якщо ви використовуєте достатню кількість відгалужувачів. IIR-фільтри складніші і можуть бути нестабільними, але вони більш ефективні (використовують менше процесора і пам'яті для даного фільтра). Якщо хтось просто дає вам список відгалужень, вважається, що це відгалуження для КІХ-фільтра. Якщо вони починають згадувати "полюси", вони говорять про IIR-фільтри. У цьому підручнику ми розглядатимемо саме БІХ-фільтри.

Нижче наведено приклад частотної характеристики для порівняння БІХ- та ІІХ-фільтрів, які виконують майже однакову фільтрацію; вони мають схожу ширину перехідної смуги, яка, як ми вже дізналися, визначає, скільки відгалужень потрібно для фільтрації. FIR-фільтр має 50 відгалужень, а IIR-фільтр має 12 полюсів, що з точки зору необхідних обчислень дорівнює 12 відгалуженням.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/FIR_IIR.png

Урок полягає в тому, що FIR-фільтр вимагає набагато більше обчислювальних ресурсів, ніж IIR-фільтр, щоб виконати приблизно ту саму операцію фільтрації.

Ось кілька реальних прикладів фільтрів FIR і IIR, які ви, можливо, використовували раніше.

Якщо ви обчислюєте "ковзне середнє" для списку чисел, то це буде просто КІХ-фільтр з відсіканням одиниць: $-h = [1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1]$ для фільтра ковзного середнього з розміром вікна 10. Він також є фільтром низьких частот; чому

це так? Яка різниця між використанням одиниць і використанням відводів, що спадають до нуля?

Фільтр ковзного середнього - це фільтр низьких частот, оскільки він згладжує "високочастотні" зміни, і саме тому його зазвичай використовують. Причина використання відгалужувачів, які затухають до нуля на обох кінцях, полягає в тому, щоб уникнути раптових змін на виході, наприклад, якщо сигнал, який фільтрується, деякий час був нульовим, а потім раптово підскочив вгору.

Тепер приклад IIR. Хто-небудь з вас коли-небудь робив це:

$$x = x*0.99 + \text{new_value}*0.01$$

де 0.99 і 0.01 представляють швидкість оновлення значення (або швидкість розпаду, що одне і те ж). Це зручний спосіб повільно оновлювати деяку змінну без необхідності запам'ятовувати останні кілька значень. Це фактично різновид низькочастотного IIR-фільтра. Сподіваюся, ви зрозуміли, чому IIR-фільтри мають меншу стабільність, ніж FIR-фільтри. Значення ніколи не зникають повністю!

4.4. Інструменти для проектування фільтрів

На практиці більшість людей використовують інструмент для створення фільтрів або функцію в коді, яка створює фільтр. Існує багато різних інструментів, але для студентів я рекомендую цей простий у використанні веб-додаток Пітера Ізі, який покаже вам імпульсну та частотну характеристику: <http://t-filter.engineerjs.com>. Використовуючи значення за замовчуванням, принаймні на момент написання цієї статті, він налаштований на створення фільтра нижніх частот зі смугою пропускання від 0 до 400 Гц і смугою зупинки від 500 Гц і вище. Частота дискретизації становить 2 кГц, тому максимальна частота, яку ми можемо "побачити" - 1 кГц.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/filter_designer1.png

Натисніть кнопку "Дизайнерський фільтр", щоб створити відводи і побудувати частотну характеристику.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/filter_designer2.png

Клацніть текст "Імпульсна характеристика" над графіком, щоб побачити імпульсну характеристику, яка є графіком відгалужень, оскільки це KIX-фільтр.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/filter_designer3.png

Ця програма навіть містить вихідний код C++ для реалізації та використання цього фільтра. Веб-додаток не містить жодного способу проектування IIR-фільтрів, які загалом набагато складніше проектувати.

4.5. Згортка

Ми зробимо невеликий обхідний маневр, щоб представити оператор згортки. Ви можете пропустити цей розділ, якщо ви вже знайомі з ним.

Додавання двох сигналів є одним із способів об'єднання двох сигналів в один. У розділі freq-domain-chapter ми розглянули, як застосовується властивість лінійності при додаванні двох сигналів. Згортка - це ще один спосіб об'єднання двох сигналів в один, але він дуже відрізняється від простого додавання. Згортка двох

сигналів схожа на ковзання одного по іншому та інтегрування. Це дуже схоже на крос-кореляцію, якщо ви знайомі з цією операцією. Насправді це еквівалентно крос-кореляції у багатьох випадках. Ми зазвичай використовуємо символ `::code::*` для позначення згортки, особливо у математичних рівняннях.

Я вважаю, що операцію згортки найкраще вивчати на прикладах. У цьому першому прикладі ми згорнемо два прямокутних імпульси разом:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/rect_rect_conv.gif

Ми маємо два вхідних сигнали (один червоний, один синій), а потім вихід згортки відображається чорним кольором. Ви можете бачити, що результатом є інтеграція двох сигналів, коли один з них ковзає по іншому. Оскільки це просто ковзне інтегрування, результатом є трикутник з максимумом у точці, де обидва квадратні імпульси ідеально вирівнялися.

Давайте розглянемо ще кілька згорток:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/rect_fat_rect_conv.gif

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/rect_exp_conv.gif

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/gaussian_gaussian_conv.gif

Зверніть увагу, що гаусс, згорнутий з гауссом, є ще одним гауссом, але з ширшим імпульсом і меншою амплітудою.

Через цю "ковзаючу" природу, довжина вихідного сигналу фактично довша за вхідний. Якщо один сигнал має M відліків, а інший сигнал має N відліків, згортка цих двох сигналів може дати $N+M-1$ відліків. Однак у таких функціях, як `numpy.convolve()` є можливість вказати, чи хочете ви отримати весь результат ($\max(M, N)$ відліків), чи лише ті відліки, де сигнали повністю перекриваються ($\max(M, N) - \min(M, N) + 1$, якщо вам цікаво). Не потрібно заціклюватися на цих деталях. Просто знайте, що довжина результату згортки - це не просто довжина вхідних даних.

Так чому ж згортка важлива в DSP? Для початку, щоб відфільтрувати сигнал, ми можемо просто взяти імпульсну характеристику цього фільтра і згорнути її з сигналом. FIR-фільтрація - це просто операція згортки.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/filter_convolve.png

Це може бути незрозуміло, оскільки раніше ми згадували, що згортка приймає два *сигнали*, а видає один. Ми можемо розглядати імпульсну характеристику як сигнал, а згортка - це математичний оператор, який оперує двома одновимірними масивами. Якщо один з цих одновимірних масивів є імпульсною характеристикою фільтра, інший одновимірний масив може бути фрагментом вхідного сигналу, і на виході ми отримаємо відфільтровану версію вхідного сигналу.

Давайте розглянемо ще один приклад, який допоможе зробити цей клік. У наведеному нижче прикладі трикутник представлятиме імпульсну характеристику нашого фільтра, а сигнал зелений - це наш сигнал, що фільтрується.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/convolution.gif

Сигнал `red` на виході є відфільтрованим сигналом.

Питання: Яким типом фільтра був трикутник?

Він згладжував високочастотні складові зеленого сигналу (тобто різкі переходи квадрата), тому він діє як фільтр низьких частот.

Тепер, коли ми починаємо розуміти, що таке згортка, я представлю математичне рівняння для неї. Зірочка (*) зазвичай використовується як символ згортки:

$$(f * g)(t) = \int f(\tau)g(t - \tau)d\tau$$

In this above expression, $g(t)$ is the signal or input that is flipped and slides across $f(t)$, but $g(t)$ and $f(t)$ can be swapped and it's still the same expression. Typically, the shorter array will be used as $g(t)$. Convolution is equal to a cross-correlation, defined as $\int f(\tau)g(t + \tau)$, when $g(t)$ is symmetrical, i.e., it doesn't change when flipped about the origin.

4.6. Проектування фільтрів у Python

Зараз ми розглянемо один із способів самостійного проектування КІХ-фільтра на мові Python. Хоча існує багато підходів до проектування фільтрів, ми будемо використовувати метод, який полягає в тому, що ми починаємо в частотній області і працюємо в зворотному напрямку, щоб знайти імпульсну характеристику. Зрештою, саме так виглядає наш фільтр (за допомогою його відводів).

Ви починаєте зі створення вектора бажаної частотної характеристики. Давай-те створимо фільтр низьких частот довільної форми, як показано нижче:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/filter_design1.png

Код, який використовується для створення цього фільтра, досить простий:

```
import numpy as np
import matplotlib.pyplot as plt
H = np.hstack((np.zeros(20), np.arange(10)/10, np.zeros(20)))
w = np.linspace(-0.5, 0.5, 50)
plt.plot(w, H, '-.')
plt.show()
```

`hstack()` - це один зі способів конкатенації масивів у `numpy`. Ми знаємо, що це призведе до створення фільтра зі складними відгалуженнями. Чому?

Він не симетричний відносно 0 Гц.

Наша кінцева мета - знайти відводи цього фільтра, щоб ми могли його використовувати. Як нам отримати відгалуження, враховуючи частотну характеристику? Як нам перетворити частотну область назад у часову? Інверсне ШПФ (IFFT)! Нагадаємо, що функція IFFT майже точно така ж, як і функція FFT. Нам також потрібно зсунути нашу бажану частотну характеристику перед IFFT, а потім ще раз зсунути її після IFFT (ні, вони не скасовують один одного, але ви можете спробувати). Цей процес може здатися заплутаним. Просто пам'ятайте, що ви завжди повинні робити FFT-зсув після FFT і IFF-зсув після IFFT.

```
h = np.fft.ifftshift(np.fft.ifft(np.fft.ifftshift(H)))
plt.plot(np.real(h))
plt.plot(np.imag(h))
plt.legend(['real', 'imag'], loc=1)
plt.show()
```

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/filter_design2.png

Ми будемо використовувати ці крани, показані вище, як наш фільтр. Ми знаємо, що імпульсна характеристика будується на основі відводів, тому те, що ми

бачимо вище - це і є наша імпульсна характеристика. Давайте візьмемо ШПФ наших відводів, щоб побачити, як насправді виглядає частотна область. Ми зробимо ШПФ для 1024 точок, щоб отримати високу роздільну здатність:

```
H_fft = np.fft.fftshift(np.abs(np.fft.fft(h, 1024)))
plt.plot(H_fft)
plt.show()
```

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/filter_design3.png

Подивіться, що частотна характеристика не дуже пряма... вона не дуже добре відповідає нашому оригіналу, якщо ви пам'ятаєте форму, для якої ми спочатку хотіли зробити фільтр. Основна причина полягає в тому, що наша імпульсна характеристика не закінчила затухати, тобто ліва і права частини не досягають нуля. У нас є два варіанти, які дозволять йому спадати до нуля:

Варіант 1: Ми "вікні" нашу поточну імпульсну характеристику, щоб вона спадала до 0 з обох сторін. Це передбачає множення нашої імпульсної характеристики на "віконну функцію", яка починається і закінчується на нулі.

```
# Після створення h за допомогою попереднього коду, створюємо і застосовуємо вікно
window = np.hamming(len(h))
h = h * window
```

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/filter_design4.png

Варіант 2: Ми повторно генеруємо нашу імпульсну характеристику, використовуючи більше точок, щоб вона встигла затухати. Нам потрібно додати роздільну здатність до нашого оригінального масиву частотної області (це називається інтерполяція).

```
H = np.hstack((np.zeros(200), np.arange(100)/100, np.zeros(200)))
w = np.linspace(-0.5, 0.5, 500)
plt.plot(w, H, '-.')
plt.show()
# (решта коду не змінюється)
```

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/filter_design5.png

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/filter_design6.png

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/filter_design7.png

Обидва варіанти спрацювали. Який би ви вибрали? Другий метод призвів до більшої кількості відгалужень, але перший метод призвів до не дуже гострої АЧХ і не дуже крутого спаду. Існує безліч способів спроектувати фільтр, кожен з яких має свої власні компроміси. Багато хто вважає проектування фільтрів мистецтвом.

4.7. Вступ до формування імпульсів

Ми коротко представимо дуже цікаву тему в рамках DSP - формування імпульсів. Пізніше ми розглянемо цю тему в окремій главі, див. pulse-shaping-chapter. Варто згадати її поряд з фільтрацією, оскільки формування імпульсів - це, зрештою,

різновид фільтра, який використовується з певною метою і має особливі властивості.

Як ми вже дізналися, цифрові сигнали використовують символи для представлення одного або декількох бітів інформації. Ми використовуємо схему цифрової модуляції, таку як ASK, PSK, QAM, FSK тощо, для модуляції несучої, щоб інформацію можна було передавати бездротовим способом. Коли ми моделювали QPSK у розділі modulation-chapter, ми моделювали лише одну вибірку на символ, тобто кожне комплексне число, яке ми створили, було однією з точок на сузір'ї - це був один символ. На практиці ми зазвичай генеруємо декілька відліків на символ, і це пов'язано з фільтрацією.

Ми використовуємо фільтри для створення "форми" наших символів, оскільки форма в часовій області змінює форму в частотній області. Частотна область інформує нас про те, скільки спектру/пропускної здатності буде використовувати наш сигнал, і ми зазвичай хочемо мінімізувати її. Важливо розуміти, що спектральні характеристики (частотна область) символів базової смуги не змінюються, коли ми модулюємо несучу; це просто зміщує базову смугу вгору по частоті, в той час як форма залишається незмінною, що означає, що кількість смуги пропускання, яку вона використовує, залишається незмінною. Коли ми використовуємо 1 вибірку на символ, це схоже на передачу прямокутних імпульсів. Насправді BPSK з використанням 1 вибірки на символ - це просто квадратна хвиля випадкових 1 і -1:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/bpsk.svg

І як ми вже з'ясували, прямокутні імпульси не є ефективними, оскільки вони використовують надмірну кількість спектру:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/square-wave.svg

Отже, ми "формуємо імпульс" цих символів, що виглядають як блоки, таким чином, щоб вони займали меншу смугу пропускання у частотній області. Ми "формуємо імпульс" за допомогою фільтра низьких частот, оскільки він відкидає високочастотні компоненти наших символів. Нижче показано приклад символів у часовій (вгорі) і частотній (внизу) областях до і після застосування фільтра, що формує імпульс:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/pulse_shaping.png

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/pulse_shaping_freq.png

Зверніть увагу, наскільки швидше падає частота сигналу. Бічні пелюстки стали на ~ 30 дБ нижчими після формування імпульсу; це в 1000 разів менше! І що більш важливо, головна пелюстка вужча, тому використовується менше спектру для тієї ж кількості біт на секунду.

Наразі, майте на увазі, що поширені фільтри, що формують імпульс, включають в себе:

1. Фільтр підвищеної косинусоїди
2. Кореневий косинусоїдальний фільтр
3. Синусоїдальний фільтр
4. Гаусів фільтр

Ці фільтри зазвичай мають параметр, який ви можете регулювати для зменшення смуги пропускання. Нижче показано часову та частотну області фільтра

з підвищеною косинусністю з різними значеннями β , параметра, який визначає, наскільки крутим є спадання.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/pulse_shaping_rolloff.png

Ви можете бачити, що менше значення β зменшує спектр, що використовується (для тієї ж кількості даних). Однак, якщо значення занадто мале, то часовим символам знадобиться більше часу, щоб розпастися до нуля. Насправді, коли $\beta = 0$ символи ніколи не розпадаються повністю до нуля, що означає, що ми не можемо передавати такі символи на практиці. Значення β близько 0.35 є поширеним.

Ви дізнаєтеся набагато більше про формування імпульсів, зокрема про деякі особливі властивості, яким мають задовольняти фільтри, що формують імпульси, у розділі pulse-shaping-chapter.

Розділ 5

Шум і дБ

У цій главі ми обговоримо шум, включаючи те, як він моделюється і обробляється в системі бездротового зв'язку. Поняття включають AWGN, комплексний шум і SNR/SINR. Ми також познайомимось з децибелами (дБ), оскільки вони широко використовуються в бездротовому зв'язку та SDR.

5.1. Гауссівський шум

Більшість людей знайомі з поняттям шуму: небажані флуктуації, які можуть за-
тмарювати бажаний сигнал (сигнали). Шум виглядає приблизно так:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/noise.png

Зверніть увагу, що середнє значення дорівнює нулю на часовому графіку. Якби середнє значення не дорівнювало нулю, то ми могли б відняти середнє значення, назвати його зсувом, і у нас залишилося б середнє значення, що дорівнює нулю. Також зверніть увагу, що окремі точки на графіку *не* "рівномірно випадкові", тобто більші значення зустрічаються рідше, більшість точок ближче до нуля.

Ми називаємо цей тип шуму "гауссівським шумом". Це хороша модель для типу шуму, який походить від багатьох природних джерел, таких як теплові коливання атомів у кремнії радіочастотних компонентів нашого приймача. Теорема про центральну межу говорить нам, що сума багатьох випадкових процесів буде мати гауссівський розподіл, навіть якщо окремі процеси мають інші розподіли. Іншими словами, коли відбувається і накопичується багато випадкових подій, результат виглядає приблизно гауссівським, навіть якщо окремі події не розподілені за гауссівським законом.

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/central_limit_theorem.svg

Розподіл Гауса також називають "нормальним" розподілом (згадайте криву дзвону).

Розподіл Гауса має два параметри: середнє значення і дисперсію. Ми вже обговорювали, як середнє значення можна вважати нульовим, тому що ви завжди можете вилучити середнє значення, або зміщення, якщо воно не дорівнює нулю. Дисперсія показує, наскільки "сильним" є шум. Вища дисперсія призводить до більших чисел. Саме з цієї причини дисперсія визначає потужність шуму.

Дисперсія дорівнює стандартному відхиленню в квадраті (σ^2).

5.2. Децибели (дБ)

Ми зробимо невеликий екскурс, щоб формально ввести дБ. Можливо, ви вже чули про дБ, і якщо ви вже знайомі з ним, можете пропустити цей розділ.

Робота в дБ надзвичайно корисна, коли нам потрібно мати справу з малими і великими числами одночасно, або просто з купою дуже великих чисел. Розглянемо, наскільки громіздкою була б робота з числами шкали в Прикладі 1 і Прикладі 2.

Приклад 1: Сигнал 1 приймається потужністю 2 Вт, а рівень шуму становить 0,0000002 Вт.

Приклад 2: Сміттепровід працює в 100 000 разів голосніше, ніж у тихій сільській місцевості, а ланцюгова пила в 10 000 разів голосніше, ніж сміттепровід (з точки зору потужності звукових хвиль).

Без децибел, тобто працюючи в звичайних "лінійних" термінах, ми повинні використовувати багато 0 для представлення значень у прикладах 1 і 2. Чесно кажучи, якби ми побудували графік чогось подібного до сигналу 1 у часі, ми б навіть не побачили рівень шуму. Наприклад, якби шкала осі Y змінювалася від 0 до 3 Вт, шум був би надто малим, щоб його можна було побачити на графіку. Щоб представити ці шкали одночасно, ми працюємо в логарифмічній шкалі.

Щоб ще більше проілюструвати проблеми масштабу, з якими ми стикаємося при обробці сигналів, розглянемо наведені нижче водоспади трьох однакових сигналів. Ліворуч - вихідний сигнал у лінійному масштабі, а праворуч - сигнали, перетворені в логарифмічну шкалу (дБ). Обидва представлення використовують однакову кольорову карту, де синій колір означає найнижче значення, а жовтий - найвище. Ви ледве можете побачити сигнал зліва в лінійній шкалі.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/linear_vs_log.png

Для заданого значення x ми можемо представити x у дБ за допомогою наступної формули:

$$x_{dB} = 10 \log_{10} x$$

У мові Python

```
x_db = 10.0 * np.log10(x)
```

Ви могли бачити, що $10 \cdot$ може бути $20 \cdot$ в інших доменах. Щоразу, коли ви маєте справу з якоюсь потужністю, ви використовуєте 10, і ви використовуєте 20, якщо ви маєте справу з неенергетичними величинами, такими як напруга або струм. В DSP ми, як правило, маємо справу з потужністю. Насправді, в цьому підручнику немає жодного разу, коли нам потрібно було б використовувати 20 замість 10.

Ми перетворюємо з дБ назад в лінійні (звичайні числа) за допомогою:

$$x = 10^{x_{dB}/10}$$

У Python

```
x = 10.0 ** (x_db / 10.0)
```

Не зациклюйтеся на формулі, оскільки тут є ключова концепція, яку потрібно винести за дужки. В DSP ми маємо справу з дуже великими і дуже малими числами (наприклад, рівень сигналу в порівнянні з рівнем шуму). Логарифмічна шкала в дБ дозволяє нам мати більший динамічний діапазон, коли ми виражаємо числа або будуємо графіки. Вона також надає деякі зручності, наприклад,

можливість додавання, коли ми зазвичай множимо (як ми побачимо у розділі [link-budgets-chapter](#)).

Деякі типові помилки, з якими можуть зіткнутися новачки у дБ, такі:

1. Використання натурального логарифма замість логарифма з основою 10, оскільки функція $\log()$ у більшості мов програмування насправді є натуральним логом.
2. Забути включити дБ при вираженні числа або позначенні осі. Якщо ми маємо справу з дБ, нам потрібно десь його позначити.
3. Коли ви використовуєте дБ, ви додаєте/віднімаєте значення замість того, щоб множити/ділити, наприклад:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/db.png

Також важливо розуміти, що дБ технічно не є "одиницею". Значення в дБ саме по собі не має одиниць виміру, наприклад, якщо щось в 2 рази більше, то одиниць виміру немає, поки я не скажу вам одиниці виміру. дБ - це відносна величина. В аудіо, коли говорять дБ, насправді мають на увазі дБА, що є одиницею вимірювання рівня звуку (А - це одиниці). У бездротовому зв'язку ми зазвичай використовуємо вати для позначення фактичного рівня потужності. Тому ви можете побачити dBW як одиницю, яка відноситься до 1 Вт. Ви також можете побачити dBmW (часто пишуть dBm для скорочення), яка відноситься до 1 мВт. Наприклад, хтось може сказати "наш передавач налаштований на 3 дБВт" (тобто 2 Вт). Іноді ми використовуємо дБ сам по собі, маючи на увазі, що він відносний і не має одиниць виміру. Можна сказати: "наш сигнал був прийнятий на 20 дБ вище рівня шуму". Ось невелика підказка: 0 дБм = -30 дБВт.

Ось кілька поширених перетворень, які я рекомендую запам'ятати:

Лінійні	дБ
1x	0 дБ
2x	3 дБ
10x	10 дБ
0.5x	-3 дБ
0.1x	-10 дБ
100x	20 дБ
1000x	30 дБ
10000x	40 дБ

Нарешті, щоб розглянути ці цифри в перспективі, нижче наведені деякі приклади рівнів потужності в дБм:

80 дБм	Передавальна потужність сільської FM-радіостанції
--------	---

62 дБм	Максимальна потужність радіоаматорського передавача
60 дБм	Потужність типової домашньої мікрохвильової печі
37 дБм	Максимальна потужність типової портативної радіостанції СВ або радіоаматорської радіостанції
27 дБм	Типова потужність передавача мобільного телефону
15 дБм	Типова потужність передачі WiFi
10 дБм	Максимальна потужність передачі Bluetooth (версія 4)
-10 дБм	Максимальна потужність прийому для WiFi
-70 дБм	Приклад прийнятої потужності для радіосигналу
-100 дБм	Мінімальна потужність прийому для WiFi
-127 дБм	Типова потужність прийому від супутників GPS

5.3. Шум в частотній області

У розділі `freq-domain-chapter` ми розглянули "пари Фур'є", тобто те, як певний сигнал часової області виглядає у частотній області. Як же виглядає гаусівський шум у частотній області? На наступних графіках показано деякий змодельований шум у часовій області (вгорі) і графік спектральної щільності потужності (PSD) цього шуму (внизу). Ці графіки взято з GNU Radio.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: `../_images/noise_freq.png`

Ми бачимо, що він виглядає приблизно однаково на всіх частотах і є досить плоским. Виходить, що гаусівський шум у часовій області є також гаусівським шумом у частотній області. Так чому ж два графіки вище не виглядають однаково? Це тому, що графік в частотній області показує величину ШПФ, тому там будуть тільки позитивні числа. Важливо, що він використовує логарифмічну шкалу, або показує величину в дБ. Інакше ці графіки виглядали б однаково. Ми можемо довести це собі, згенерувавши деякий шум (у часовій області) у Python, а потім отримавши ШПФ.

```
import numpy as np
import matplotlib.pyplot as plt

N = 1024 # кількість відліків для моделювання, виберіть будь-яке число
x = np.random.randn(N)
plt.plot(x, '-.')
plt.show()

X = np.fft.fftshift(np.fft.fft(x))
X = X[N/2:] # дивимось тільки додатні частоти. пам'ятайте, що // це просто цілочисельне
            ↪ ділення
plt.plot(np.real(X), '-.')
plt.show()
```

Зверніть увагу, що функція `randn()` за замовчуванням використовує `mean = 0` і `variance = 1`. Обидва графіки будуть виглядати приблизно так:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: `../_images/noise_python.png`

Потім ви можете створити пласку PSD, яку ми мали у GNU Radio, взявши запис і усереднивши його разом. Сигнал, який ми згенерували і взяли ШПФ, був реальним сигналом (а не комплексним), і ШПФ будь-якого реального сигналу матиме відповідні від'ємні та додатні частини, тому ми зберегли лише додатну частину

результату ШПФ (2-гу половину). Але чому ми згенерували лише "реальний" шум, і як комплексні сигнали пов'язані з цим?

5.4. Комплексний шум

"Комплексний гаусівський" шум - це те, що ми відчуваємо, коли маємо сигнал в основній смузі частот; потужність шуму ділиться між реальною та уявною частинами порівну. І найголовніше, що реальна та уявна частини не залежать одна від одної; знаючи значення однієї з них, ви не отримаєте значення іншої.

Ми можемо згенерувати складний гаусівський шум у Python за допомогою

```
n = np.random.randn() + 1j * np.random.randn()
```

Але зачекайте! Наведене вище рівняння не генерує таку ж "кількість" шуму, як `np.random.randn()`, з точки зору потужності (відомої як потужність шуму). Ми можемо знайти середню потужність сигналу (або шуму) з нульовим середнім значенням за допомогою:

```
power = np.var(x)
```

де `np.var()` - функція для дисперсії. Тут потужність нашого сигналу `n` дорівнює 2. Для того, щоб згенерувати складний шум з "одиночною потужністю", тобто потужністю 1 (що робить речі зручними), ми повинні використовувати

```
n = (np.random.randn(N) + 1j*np.random.randn(N))/np.sqrt(2) # AWGN з одиночною потужністю
```

Для побудови графіка складного шуму в часовій області, як і будь-якого складного сигналу, нам знадобляться два рядки:

```
n = (np.random.randn(N) + 1j*np.random.randn(N))/np.sqrt(2)
plt.plot(np.real(n), '-.')
plt.plot(np.imag(n), '-.')
plt.legend(['real', 'imag'])
plt.show()
```

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/noise3.png

Ви можете бачити, що дійсна та уявна частини повністю незалежні.

Як виглядає складний гаусівський шум на IQ-діаграмі? Пам'ятайте, що графік IQ показує дійсну частину (горизонтальна вісь) і уявну частину (вертикальна вісь), обидві з яких є незалежними випадковими гаусівськими величинами.

```
plt.plot(np.real(n), np.imag(n), '-.')
plt.grid(True, which='both')
plt.axis([-2, 2, -2, 2])
plt.show()
```

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/noise_iq.png

Це виглядає так, як ми і очікували: випадкова пляма з центром в $0 + 0j$, або в початку координат. Заради інтересу спробуємо додати шум до QPSK-сигналу, щоб побачити, як виглядає IQ-діаграма:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/noisey_qpsk.png

А що станеться, коли шум буде сильнішим?

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/noisey_qpsk2.png

Ми починаємо розуміти, чому передача даних бездротовим способом не така проста. Ми хочемо відправити якомога більше бітів на символ, але якщо шум занадто високий, ми отримуємо помилкові біти на приймальному боці.

5.5. AWGN

Additive White Gaussian Noise (AWGN) - це аббревіатура, яку ви часто чуєте в світі DSP і SDR. Про GN, гауссівський шум, ми вже говорили. Адитивний просто означає, що шум додається до прийнятого сигналу. Білий колір в частотній області означає, що спектр є плоским у всій смузі спостереження. На практиці він майже завжди буде білим, або приблизно білим. У цьому підручнику ми будемо використовувати AWGN як єдину форму шуму, коли маємо справу з лініями зв'язку, бюджетами ліній зв'язку тощо. Шум, який не є AWGN, як правило, є вузькоспеціалізованою темою.

5.6. SNR і SINR

Відношення сигнал/шум (SNR) - це те, як ми будемо вимірювати різницю в силі між сигналом і шумом. Це відношення не має одиниць виміру. На практиці SNR майже завжди вимірюється в дБ. Часто при моделюванні ми кодуємо сигнали таким чином, щоб вони мали одиничну потужність (потужність = 1). Таким чином, ми можемо створити SNR 10 дБ, генеруючи шум потужністю -10 дБ, регулюючи дисперсію під час генерації шуму.

$$\text{SNR} = \frac{P_{\text{signal}}}{P_{\text{noise}}}$$

$$\text{SNR}_{\text{dB}} = P_{\text{signal_dB}} - P_{\text{noise_dB}}$$

Якщо хтось каже "SNR = 0 дБ", це означає, що потужність сигналу і шуму однакова. Позитивне значення SNR означає, що наш сигнал має більшу потужність, ніж шум, тоді як негативне значення SNR означає, що шум має більшу потужність. Виявлення сигналів з від'ємним SNR зазвичай досить складне.

Як ми вже згадували раніше, потужність сигналу дорівнює дисперсії сигналу. Отже, ми можемо представити SNR як відношення дисперсії сигналу до дисперсії шуму:

$$\text{SNR} = \frac{P_{\text{signal}}}{P_{\text{noise}}} = \frac{\sigma_{\text{signal}}^2}{\sigma_{\text{noise}}^2}$$

Відношення сигнал/завада плюс шум (SINR) по суті те саме, що й SNR, за винятком того, що до знаменника ви додаєте заваду разом із шумом.

$$\text{SINR} = \frac{P_{\text{signal}}}{P_{\text{interference}} + P_{\text{noise}}}$$

Що є завадою, залежить від застосування/ситуації, але зазвичай це інший сигнал, який заважає сигналу, що становить інтерес (SOI), і який або перекриває SOI по частоті, або не може бути відфільтрований з якихось причин.

5.7. Зовнішні ресурси

1. https://en.wikipedia.org/wiki/Additive_white_Gaussian_noise
2. https://en.wikipedia.org/wiki/Signal-to-noise_ratio
3. <https://en.wikipedia.org/wiki/Variance>

Розділ 6

Цифрова модуляція

У цьому розділі ми обговоримо *фактичну передачу даних* за допомогою цифрової модуляції та бездротових символів! Ми розробимо сигнали, які передають "інформацію", наприклад, одиниці та нулі, використовуючи схеми модуляції, такі як ASK, PSK, QAM та FSK. Ми також обговоримо IQ-графіки та сузір'я, а наприкінці глави наведемо кілька прикладів на Python.

Основне завдання модуляції - втиснути якомога більше даних у якомога меншу частку спектра. З точки зору техніки, це значить, що ми хочемо максимізувати "спектральну ефективність" в одиницях вимірювання біт/сек/Гц. Передача одиниць і нулів з більшою швидкістю, збільшує і смугу пропускання що потрібна для нашого сигналу (згадайте тему про властивості перетворення Фур'є), а це означає, що буде використано більше смуга з спектру. Окрім прискорення швидкості передачі, ми також розглянемо інші методи. При виборі способу модуляції є багато компромісів, але також є і простір для творчості.

6.1. Символи

Увага, новий термін! Наш сигнал передачі буде складатися з "символів". Кожен символ буде нести певну кількість бітів інформації, і ми будемо передавати символи один за одним, тисячі або навіть мільйони разів поспіль.

Як спрощений приклад, уявімо, що у нас є дріт і ми передаємо одиниці та нулі, використовуючи високі та низькі рівні напруги. Символ - це одне з двох цих значень (одиниці або нуля):

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/symbols.png

У наведеному вище прикладі кожен символ представляє один біт. Як можна передати більше одного біта одним символом? Розглянемо сигнали, які передаються по кабелю Ethernet, що визначений в стандарті IEEE під назвою IEEE 802.3 1000BASE-T. При звичайному режимі роботи Ethernet використовує 4-рівневу амплітудну модуляцію (2 біти на символ) з тривалістю символів 8 нс.

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/ethernet.svg

Спробуйте відповісти на ці питання:

1. Скільки біт в секунду передається в наведеному вище прикладі?
2. Скільки пар цих дрітів потрібно для передачі даних зі швидкістю 1 гігабіт/с?
3. Якщо схема модуляції має 16 різних рівнів, скільки біт припадає на один символ?

4. Якщо схема модуляції має 16 різних рівнів і тривалість символу 8 нс, то розрахуйте швидкість передачі даних в біт на секунду?
1. 250 Мбіт/с - $(1/8e-9)*2$
 2. Чотири (саме стільки мають ethernet-кабелі)
 3. 4 біти на символ - $\log_2(16)$
 4. 0.5 Гбіт/с - $(1/8e-9)*4$

6.2. Бездротові символи

Питання: Чому ми не можемо безпосередньо передавати сигнал ethernet, преведені на рисунку вище? Існує багато причин, дві з яких є основними:

1. Низькі частоти вимагають *величезних* антен, а вищенаведений сигнал містить частоти аж до постійного струму (0 Гц). Передавати постійний струм ми не можемо.
2. Прямокутні хвилі займають надмірну кількість спектру для певної швидкості в бітах на секунду - згадайте розділ `freq-domain-chapter` де було показано, що різкі зміни у часовій області призводять до розширення смуги пропускання/спектру сигналу:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: `../_images/square-wave.svg`

Що ми робимо для бездротових сигналів, так це починаємо з несучої, яка є просто синусоїдою. Наприклад, FM-радіо використовує несучу частоту 101,1 МГц або 100,3 МГц. Ми модулюємо цю несучу певним чином (є багато способів). Для FM-радіо це аналогова модуляція, а не цифрова, але це те саме поняття, що й цифрова модуляція.

Яким чином ми можемо модулювати несучу? Інший спосіб поставити те саме питання: які різні властивості синусоїди?

1. Амплітуда
2. Фаза
3. Частота

Ми можемо модулювати наші дані на носій, змінюючи один (або більше) з цих трьох параметрів.

6.3. Амплітудна маніпуляція (ASK)

Амплітудна маніпуляція (ASK) - це перша схема цифрової модуляції, яку ми обговоримо, оскільки амплітудна модуляція є найпростішою для візуалізації з трьох властивостей синусоїди. Ми буквально модулюємо **амплітуду** несучої. Ось приклад дворівневої ASK, яка називається 2-ASK:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: `../_images/ASK.svg`

Зверніть увагу, що середнє значення дорівнює нулю; ми завжди надаємо перевагу цьому, коли це можливо.

Ми можемо використовувати більше двох рівнів, що дозволяє отримати більше бітів на символ. Нижче наведено приклад 4-ASK. У цьому випадку кожен символ містить 2 біти інформації.

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/ask2.svg

Питання: Скільки символів показано у фрагменті сигналу вище? Скільки бітів представлено загалом?

20 символів, тобто 40 біт інформації

Як ми можемо створити цей сигнал в цифровому вигляді, за допомогою коду? Все, що нам потрібно зробити, це створити вектор з N відліків на символ, а потім помножити цей вектор на синусоїду. Це модулює сигнал на несучу (синусоїда виступає в якості такої несучої). У прикладі нижче показано 2-ASK з 10 відліками на символ.

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/ask3.svg

Верхній графік показує дискретні відліки, представлені червоними крапками, тобто наш цифровий сигнал. Нижній графік показує, як виглядає результуючий модульований сигнал, який можна було б передавати в ефір. У реальних системах частота несучої зазвичай набагато вища за швидкість зміни символів. У цьому прикладі в кожному символі лише три цикли синусоїди, але на практиці їх можуть бути тисячі, залежно від того, наскільки високо в спектрі передається сигнал.

6.4. Фазова маніпуляція (PSK)

Тепер давайте розглянемо модуляцію фази так само, як ми це робили з амплітудою. Найпростішою формою є двійкова PSK, також відома як BPSK, де є два рівні фази:

1. Без зміни фази
2. Зміна фази на 180 градусів

Приклад BPSK (зверніть увагу на зміну фази):

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/bpsk.svg

Дивитися на такі графіки не дуже цікаво:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/bpsk2.svg

Замість цього ми зазвичай представляємо фазу на комплексній площині.

6.5. IQ-графіки/сузір'я

Ви вже бачили IQ-графіки раніше в підрозділі комплексних чисел розділу `sampling-chapter`, але зараз ми використаємо їх у новий і цікавий спосіб. Для заданого символу ми можемо показати амплітуду і фазу на IQ-діаграмі. Для прикладу BPSK ми сказали, що маємо фази 0 і 180 градусів. Нанесімо ці дві точки

на IQ-діаграму. Ми візьмемо значення 1. На практиці не має значення, яке значення ви використовуєте; більше значення означає більшу потужність сигналу, але ви також можете просто збільшити коефіцієнт підсилення підсилювача.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/psk_iq.png

Наведена вище IQ-діаграма показує, що ми будемо передавати, а точніше набір символів, з яких ми будемо передавати. Вона не показує несучу, тому ви можете думати про неї, як про представлення символів у базовій смузі. Коли ми показуємо набір можливих символів для даної схеми модуляції, ми називаємо його "сузір'ям". Багато схем модуляції можна визначити за допомогою сузір'я.

Для прийому і декодування BPSK ми можемо використовувати дискретизацію IQ, про яку ми дізналися в минулому розділі, і дослідити, куди потрапляють точки на графіку IQ. Однак, через бездротовий канал буде випадковий поворот фази, оскільки сигнал буде мати випадкову затримку, коли він проходить через повітря між антенами. Випадковий поворот фази можна виправити різними методами, про які ми дізнаємося пізніше. Ось приклад кількох різних способів, якими BPSK-сигнал може з'явитися на приймачі (це без урахування шуму):

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/psk3.png

Повернемося до PSK. Що, якщо ми хочемо отримати чотири різних рівні фази? Тобто 0, 90, 180 і 270 градусів. У цьому випадку на IQ-діаграмі це буде виглядати так, як на діаграмі, і утворює схему модуляції, яку ми називаємо квадратурною фазовою маніпуляцією (QPSK):

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/qpsk.png

Для PSK ми завжди маємо N різних фаз, рівномірно розподілених навколо 360 градусів для досягнення найкращих результатів. Ми часто показуємо одиничне коло, щоб підкреслити, що всі точки мають однакову величину:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/psk_set.png

Питання: Що поганого у використанні схеми PSK, як на наведеному нижче зображенні? Чи є це дійсною схемою модуляції PSK?

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/weird_psk.png

У цій схемі PSK немає нічого неправильного. Ви, звичайно, можете використовувати її, але, оскільки символи розташовані нерівномірно, ця схема не настільки ефективна, як могла б бути. Ефективність схеми стане зрозумілою, коли ми обговоримо, як шум впливає на наші символи. Коротка відповідь полягає в тому, що ми хочемо залишити якомога більше місця між символами, якщо є шум, щоб символ не був інтерпретований приймачем як один з інших (неправильних) символів. Ми не хочемо, щоб 0 був прийнятий як 1.

Давайте на мить повернемося до ASK. Зауважте, що ми можемо показати ASK на графіку IQ так само, як і PSK. Ось графік IQ для 2-ASK, 4-ASK і 8-ASK у біполярній конфігурації, а також 2-ASK і 4-ASK в уніполярній конфігурації.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/ask_set.png

Як ви могли помітити, біполярна 2-ASK і BPSK однакові. Зсув фази на 180 градусів - це те саме, що помножити синусоїду на -1. Ми називаємо його BPSK, ймовірно, тому, що PSK використовується набагато частіше, ніж ASK.

6.6. Квадратурна амплітудна модуляція (QAM)

Що, якщо ми об'єднаємо ASK і PSK? Ми називаємо таку схему модуляції квадратурною амплітудною модуляцією (QAM). Зазвичай QAM виглядає приблизно так:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/64qam.png

Ось деякі інші приклади QAM:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/qam.png

Для схеми модуляції QAM ми можемо технічно поставити точки де завгодно на діаграмі IQ, оскільки фаза і амплітуда модулюються. "Параметри" даної схеми QAM найкраще визначити, показавши сузір'я QAM. Крім того, ви можете перерахувати значення I і Q для кожної точки, як показано нижче для QPSK:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/qpsk_list.png

Зауважте, що більшість схем модуляції, за винятком різних ASK і BPSK, досить важко "побачити" в часовій області. Щоб довести мою думку, наведемо приклад QAM у часовій області. Чи можете ви розрізнити фазу кожного символу на зображенні нижче? Це важко.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/qam_time_domain.png

Враховуючи складність розрізнення схем модуляції в часовій області, ми надаємо перевагу використанню IQ-діаграм замість відображення сигналу в часовій області. Тим не менш, ми можемо показати сигнал у часовій області, якщо є певна структура пакетів або послідовність символів має значення.

6.7. Частотна маніпуляція (FSK)

Останньою у списку є частотна маніпуляція (Frequency Shift Keying, FSK). FSK досить простий для розуміння - ми просто перемикаємося між N частотами, де кожна частота - це один можливий символ. Однак, оскільки ми модулюємо несучу, це насправді наша несуча частота \pm ці N частот. Наприклад, ми можемо перебувати на несучій частоті 1,2 ГГц і зміщуватися між цими чотирма частотами:

1. 1.2005 ГГц
2. 1.2010 ГГц
3. 1.1995 ГГц
4. 1.1990 ГГц

У наведеному вище прикладі буде 4-FSK, і на кожен символ припадатиме по два біти. Сигнал 4-FSK у частотній області може виглядати приблизно так:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/fsk.svg

Якщо ви використовуєте FSK, ви повинні поставити важливе питання: Якою має бути відстань між частотами? Ми часто позначаємо цю відстань як Δf у Гц. Ми хочемо уникнути перекриття в частотній області, щоб приймач знав, на якій частоті використовується даний символ, тому Δf має бути достатньо великою. Ширина кожної несучої у частотній області є функцією нашої швидкості передачі

символів. Чим більше символів на секунду, тим коротші символи, а отже, ширша смуга пропускання (згадайте обернену залежність між масштабуванням часу і частоти). Чим швидше ми передаємо символи, тим ширшою буде кожна несуча, і, відповідно, тим більшим буде Δf , щоб уникнути перекриття несучих. У цьому підручнику ми не будемо вдаватися у подробиці побудови FSK.

IQ-діаграми не можна використовувати для відображення різних частот. Вони показують амплітуду і фазу. Хоча можна показати FSK в часовій області, але більше ніж 2 частоти ускладнюють розрізнення символів:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/fsk2.svg

Зверніть увагу, що FM-радіо використовує частотну модуляцію (FM), яка є аналоговою версією FSK. Замість дискретних частот, між якими ми перестрибуємо, FM-радіо використовує безперервний аудіосигнал для модуляції частоти несучої. Нижче наведено приклад FM і AM модуляції, де "сигнал" вгорі - це аудіосигнал, який модулюється на несучу частоту.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/Carrier_Mod_AM_FM.webp

У цьому підручнику ми розглядаємо переважно цифрові форми модуляції.

6.8. Диференціальне кодування

У багатьох протоколах бездротового (і дротового) зв'язку ви, швидше за все, зіткнетеся з так званим диференціальним кодуванням. Щоб продемонструвати його корисність, розглянемо прийом сигналу BPSK. Коли сигнал пролітає повітрям, він зазнає випадкової затримки між передавачем і приймачем, що спричиняє випадкове обертання сузір'я, як ми вже згадували раніше. Коли приймач синхронізується з ним і вирівнює BPSK по осі "I", він не має можливості дізнатися, чи є зсув у фазі на 180 градусів чи ні, оскільки сузір'я симетричне. Одним з варіантів є передача символів, значення яких одержувач знає заздалегідь, змішаних з інформацією, відомих як пілотні символи. Одержувач може використовувати ці відомі символи, щоб визначити, який кластер є 1 або 0, у випадку BPSK. Пілотні символи повинні надсилатися через певний проміжок часу, пов'язаний з тим, наскільки швидко змінюється бездротовий канал, що в кінцевому підсумку зменшує швидкість передачі даних.

Замість того, щоб підмішувати пілотні символи до форми сигналу, що передається, ми можемо використовувати диференціальне кодування. У своїй найпростішій формі, яка використовується для BPSK, диференціальне кодування передбачає передачу 0, коли вхідний біт збігається з попереднім вихідним бітом, і передачу 1, коли вони відрізняються. Таким чином, ми все ще передаємо ту саму кількість бітів (за винятком одного додаткового біта, необхідного на початку для початку вихідної послідовності), але тепер нам не потрібно турбуватися про 180-градусну неоднозначність фази. Щоб продемонструвати, як це працює, розглянемо передачу послідовності бітів [1, 1, 0, 0, 0, 0, 1, 0] з використанням BPSK. Припустимо, що ми почнемо вихідну послідовність з 1; насправді не має значення, чи ви використовуєте 1 або 0. Після застосування диференціального кодування ми в кінцевому підсумку передамо [1, 0, 1, 1, 1, 1, 0, 0]. Одиниці та нулі все ще зіставляються з позитивними та негативними символами, про які ми говорили раніше. Можливо, буде простіше візуалізувати вхідні та вихідні послідовності, складені у стек таким чином:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/differential_coding.svg

Великим недоліком використання диференціального кодування є те, що якщо у вас є бітова помилка, це призведе до двох бітових помилок. Альтернативою використанню диференціального кодування для BPSK є періодичне додавання пілотних символів, які є символами, вже відомими приймачу, і він може використовувати відомі значення, щоб не тільки визначити, який кластер є 1, а який 0, але й обернути багатопроменевість, спричинену каналом. Одна з проблем з пілот-символами полягає в тому, що бездротовий канал може змінюватися дуже швидко, порядку десятків або сотень символів, якщо це рухомий приймач і/або передавач, тому вам знадобляться пілотні символи досить часто, щоб відображати зміну каналу. Отже, якщо бездротовий протокол приділяє велику увагу зменшенню складності приймача, як, наприклад, RDS, який ми розглядаємо у розділі rds-chapter, він може використовувати диференційне кодування.

6.9. Приклад на Python

Як короткий приклад на Python, давайте згенеруємо QPSK на базовій смузі і побудуємо графік сузір'я.

Хоча ми могли б згенерувати складні символи безпосередньо, почнемо з того, що QPSK має чотири символи з інтервалом 90 градусів навколо одиничного кола. Ми будемо використовувати 45, 135, 225 і 315 градусів для наших точок. Спочатку ми згенеруємо випадкові числа від 0 до 3 і виконаємо математичні дії, щоб отримати потрібні нам градуси, а потім перетворимо їх у радіани.

```
import numpy as np
import matplotlib.pyplot as plt

num_symbols = 1000

x_int = np.random.randint(0, 4, num_symbols) # від 0 до 3
x_degrees = x_int*360/4.0 + 45 # 45, 135, 225, 315 градусів
x_radians = x_degrees*np.pi/180.0 # sin() і cos() беруть в радіанах
x_symbols = np.cos(x_radians) + 1j*np.sin(x_radians) # отримуємо наші комплексні символи
↳ QPSK
plt.plot(np.real(x_symbols), np.imag(x_symbols), '.')
plt.grid(True)
plt.show()
```

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/qpsk_python.svg

Подивіться, як всі символи, які ми згенерували, перекриваються. Шум відсутній, тому всі символи мають однакове значення. Давайте додамо трохи шуму:

```
phase_noise = np.random.randn(len(x_symbols)) * 0.1 # adjust multiplier for "strength" of
↳ phase noise
r = x_symbols * np.exp(1j*phase_noise)
```

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/phase_jitter.svg

Розглянемо, як адитивний білий гаусівський шум (AWGN) створює рівномірний розподіл навколо кожної точки сузір'я. Якщо шуму занадто багато, то символи починають перетинати межу (чотири квадранти) і будуть інтерпретуватися приймачем як неправильний символ. Спробуйте збільшити `noise_power`, поки цього не станеться.

Для тих, хто зацікавлений в імітації фазового шуму, який може виникнути внаслідок фазового джиттера у локальному генераторі (LO), замініть `r` на:

```
phase_noise = np.random.randn(len(x_symbols)) * 0.1 # підлаштовуємо множник під "силу"  
↳ фазового шуму  
r = x_symbols * np.exp(1j*phase_noise)
```

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/phase_jitter.svg

Ви навіть можете комбінувати фазовий шум з AWGN, щоб отримати повний ефект:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/phase_jitter_awgn.svg

На цьому ми зупинимося. Якби ми хотіли побачити, як виглядає QPSK-сигнал у часовій області, нам потрібно було б згенерувати кілька відліків на символ (у цій вправі ми зробили лише 1 відлік на символ). Ви дізнаєтеся, чому потрібно генерувати кілька відліків на символ, коли ми обговоримо формування імпульсів. Вправа з Python у розділі `pulse-shaping-chapter` буде продовжена з того місця, на якому ми зупинилися.

6.10. Додаткова інформація

1. https://en.wikipedia.org/wiki/Differential_coding

Розділ 7

Pulse Shaping

У цій главі розглядається формування імпульсів, міжсимвольна інтерференція, узгоджена фільтрація та фільтри з підвищеною косинусністю. Наприкінці ми використаємо Python для додавання формування імпульсів до символів BPSK. Ви можете розглянути цей розділ у частині II розділу "Фільтри", де ми глибше зануримося у формування імпульсів.

7.1. Міжсимвольна інтерференція (ISI)

У розділі `filters-chapter` ми дізналися, що символи/імпульси блокової форми використовують надмірну кількість спектра, і ми можемо значно зменшити кількість спектра, що використовується, "формуючи" наші імпульси. Однак, ви не можете використовувати будь-який низькочастотний фільтр, інакше ви можете отримати міжсимвольну інтерференцію (ISI), коли символи впливаються один в одного і заважають один одному.

Коли ми передаємо цифрові символи, ми передаємо їх один за одним (а не чекаємо деякий час між ними). Коли ви застосовуєте фільтр, що формує імпульс, він продовжує імпульс у часовій області (щоб ущільнити його за частотою), що призводить до того, що сусідні символи накладаються один на одного. Таке перекриття є нормальним, якщо ваш фільтр відповідає одному критерію: всі імпульси повинні дорівнювати нулю при кожному кратному періоді нашого символу T , за винятком одного з імпульсів. Найкраще цю ідею можна зрозуміти за допомогою наступної візуалізації:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/pulse_train.svg

Як ви можете бачити, на кожному інтервалі T є один пік імпульсу, тоді як решта імпульсів дорівнює 0 (вони перетинають вісь x). Коли приймач робить вибірку сигналу, він робить це в ідеальний момент часу (на піку імпульсів), тобто це єдиний момент часу, який має значення. Зазвичай у приймачі є блок синхронізації символів, який забезпечує вибірку символів на піках.

7.2. Узгоджений фільтр

Один з трюків, який ми використовуємо в бездротовому зв'язку, називається узгодженою фільтрацією. Щоб зрозуміти суть узгодженої фільтрації, ви повинні спочатку зрозуміти ці два моменти:

1. Імпульси, які ми обговорювали вище, повинні бути ідеально вирівняні лише на приймачі перед вибіркою. До цього моменту не має значення, чи є ISI, тобто сигнали можуть летіти по повітрю з ISI і це нормально.
2. Нам потрібен фільтр низьких частот у передавачі, щоб зменшити кількість спектру, який використовує наш сигнал. Але приймач також потребує низькочастотного фільтра, щоб усунути якомога більше шуму/перешкод поруч із сигналом. В результаті ми маємо фільтр низьких частот на передавачі (Tx) і ще один на приймачі (Rx), а дискретизація відбувається після обох фільтрів (і впливу бездротового каналу).

Що ми робимо в сучасному зв'язку, так це ділимо фільтр формування імпульсів порівну між Tx і Rx. Це не обов'язково повинні бути ідентичні фільтри, але теоретично оптимальним лінійним фільтром для максимізації SNR за наявності AWGN є використання *однакового* фільтра на Tx і Rx. Ця стратегія називається концепцією "узгодженого фільтра".

Інший спосіб мислення про узгоджені фільтри полягає в тому, що приймач співвідносить отриманий сигнал з відомим шаблонним сигналом. Шаблонний сигнал - це, по суті, імпульси, які надсилає передавач, незалежно від фазових/амплітудних зсувів, що застосовуються до них. Нагадаємо, що фільтрація здійснюється шляхом згортки, яка по суті є кореляцією (насправді вони математично однакові, коли шаблон симетричний). Цей процес кореляції отриманого сигналу з шаблоном дає нам найкращий шанс відновити те, що було надіслано, і саме тому він є теоретично оптимальним. Як аналогію, уявіть собі систему розпізнавання зображень, яка шукає обличчя, використовуючи шаблон обличчя і двовимірну кореляцію:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/face_template.png

7.3. Розділення фільтра навпіл

Як насправді розділити фільтр навпіл? Згортання є асоціативним, що означає:

$$(f * g) * h = f * (g * h)$$

Уявімо собі f як наш вхідний сигнал, а g і h - це фільтри. Фільтрація f за допомогою g , а потім h - це те ж саме, що і фільтрація одним фільтром, рівним $g * h$.

Також нагадаємо, що згортка у часовій області - це множення у частотній області:

$$g(t) * h(t) \leftrightarrow G(f)H(f)$$

Щоб розділити фільтр навпіл, можна взяти квадратний корінь з частотної характеристики.

$$X(f) = X_H(f)X_H(f) \quad \text{where} \quad X_H(f) = \sqrt{X(f)}$$

Нижче показано спрощену схему ланцюга передачі та прийому з фільтром піднесеного косинуса (RC), розділеним на два фільтри кореневого піднесеного косинуса (RRC); той, що на стороні передачі, є фільтром формування імпульсів, а той, що на стороні прийому, є узгодженим фільтром. Разом вони призводять до того, що імпульси на демодуляторі виглядають так, ніби вони були сформовані одним RRC-фільтром.

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/splitting_rc_filter.svg

7.4. Специфічні фільтри формування імпульсів

Ми знаємо, що ми хочемо:

1. Спроекувати фільтр, який зменшує смугу пропускання нашого сигналу (щоб використовувати менше спектру) і всі імпульси, крім одного, повинні дорівнювати нулю через кожний символний інтервал.
2. Розділити фільтр навпіл, помістивши одну половину в T_x , а іншу в R_x .

Давайте розглянемо деякі конкретні фільтри, які зазвичай використовуються для формування імпульсів.

7.4.1. Фільтр піднесеного косинуса

Найпопулярнішим фільтром, що формує імпульс, здається, є фільтр "піднесеного косинуса". Це хороший фільтр низьких частот для обмеження смуги пропускання, яку займатиме наш сигнал, а також він має властивість обнулятися на інтервалах T :

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/raised_cosine.svg

Зверніть увагу, що наведений вище графік наведено у часовій області. На ньому зображено імпульсну характеристику фільтра. Параметр β є єдиним параметром для фільтра піднесеного косинуса, і він визначає швидкість спадання фільтра в часовій області, яка буде обернено пропорційна швидкості спадання в частотній області:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/raised_cosine_freq.svg

Причина, чому він називається фільтром з піднятим косинусом, полягає в тому, що частотна область при $\beta = 1$ являє собою півперіод косинусоїдальної хвилі, піднятої вгору, щоб розташуватися на осі x .

Рівняння, яке визначає імпульсну характеристику фільтра з піднятими косинусами, має вигляд:

$$h(t) = \frac{1}{T} \operatorname{sinc}\left(\frac{t}{T}\right) \frac{\cos\left(\frac{\pi\beta t}{T}\right)}{1 - \left(\frac{2\beta t}{T}\right)^2}$$

Більш детальну інформацію про функцію $\operatorname{sinc}()$ можна знайти [тут](#).

Пам'ятайте: ми ділимо цей фільтр між T_x і R_x порівну. Введіть фільтр кореневого піднесеного косинуса (RRC)!

7.4.2. Фільтр кореневого піднесеного косинуса

Фільтр кореневого піднесеного косинуса (RRC) - це те, що ми фактично застосовуємо в наших T_x і R_x . Разом вони утворюють звичайний фільтр піднесеного косинуса, як ми вже обговорювали. Оскільки поділ фільтра навпіл включає в себе квадратний корінь з частотної області, імпульсна характеристика стає трохи заплутаною:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/rrc_filter.png

На щастя, цей фільтр широко використовується, і для нього існує багато реалізацій, зокрема [у Python](#).

7.4.3. Інші фільтри формування імпульсів

Інші фільтри включають фільтр Гауса, який має імпульсну характеристику, що нагадує функцію Гауса. Існує також синусоїдальний фільтр, який еквівалентний фільтру піднесеного косинуса, коли $\beta = 0$. Синусоїдальний фільтр є більш ідеальним фільтром, тобто він усуває необхідні частоти без значної перехідної області.

7.5. Коефіцієнт згортання

Давайте розглянемо параметр β . Це число від 0 до 1, яке називають фактором "згортання" або іноді "надлишковою смугою пропускання". Він визначає, як швидко у часовій області фільтр згортається до нуля. Нагадаємо, що для використання в якості фільтра імпульсна характеристика повинна спадати до нуля з обох боків:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/rrc_rolloff.svg

Чим нижче значення β , тим більше потрібно відводів фільтра. При $\beta = 0$ імпульсна характеристика ніколи повністю не досягає нуля, тому ми намагаємося зробити β якомога нижчою, не викликаючи інших проблем. Чим меншим є спадання, тим більш компактним за частотою ми можемо створити наш сигнал для заданої швидкості передачі, що завжди важливо.

Загальне рівняння, яке використовується для наближеного обчислення смуги пропускання у Гц для заданої швидкості передачі символів і коефіцієнта рол-офф, має вигляд:

$$BW = R_S(\beta + 1)$$

R_S - це символна швидкість у Гц. Для бездротового зв'язку ми зазвичай використовуємо значення від 0,2 до 0,5. Як правило, цифровий сигнал, який використовує частоту символів R_S , займатиме трохи більше, ніж R_S спектра, включаючи як позитивну, так і негативну частини спектра. Після перетворення і передачі нашого сигналу, обидві сторони, безумовно, мають значення. Якщо ми передаємо QPSK зі швидкістю 1 мільйон символів на секунду (MSps), це займе близько 1,3 МГц. Швидкість передачі даних становитиме 2 Мбіт/с (нагадаємо, що QPSK використовує 2 біти на символ), включаючи всі накладні витрати, такі як кодування каналу і заголовки кадрів.

7.6. Вправа на Python

В якості вправи на Python давайте відфільтруємо і сформуємо деякі імпульси. Ми будемо використовувати символи BPSK, щоб було легше візуалізувати - до етапу формування імпульсів, BPSK передбачає передачу 1 або -1 з частиною "Q", що дорівнює нулю. З Q, що дорівнює нулю, ми можемо побудувати графік лише частини I, і на нього легше дивитися.

У цій симуляції ми використаємо 8 відліків на символ, і замість прямокутного сигналу, що складається з 1 та -1, ми використаємо імпульсну послідовність імпульсів. Коли ви пропускаєте імпульс через фільтр, на виході виходить імпульсна характеристика (звідси і назва). Тому, якщо ви хочете отримати серію імпульсів, вам потрібно використовувати імпульси з нулями між ними, щоб уникнути прямокутних імпульсів.

```
import numpy as np
import matplotlib.pyplot as plt
з scipy імпортуємо сигнал

num_symbols = 10
sps = 8

bits = np.random.randint(0, 2, num_symbols) # Наші дані для передачі, 1 та 0

x = np.array([])
для біта в бітах:
    pulse = np.zeros(sps)
    pulse[0] = bit*2-1 # встановлюємо перше значення в 1 або -1
    x = np.concatenate((x, pulse)) # додаємо 8 відліків до сигналу
plt.figure(0)
plt.plot(x, '-.')
plt.grid(True)
plt.show()
```

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/pulse_shaping_python1.png

На цьому етапі наші символи все ще складаються з 1 та -1. Не зациклюйтеся на тому, що ми використали імпульси. Насправді, може бути простіше *не* візуалізувати реакцію на імпульси, а думати про неї як про масив:

```
бітів: [0, 1, 1, 1, 1, 0, 0, 0, 1, 1]
BPSK символи: [-1, 1, 1, 1, 1, -1, -1, -1, 1, 1]
Застосування 8 відліків на символ: [-1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1,
↪ 0, 0, 0, 0, 0, 0, 0, ...]
```

Ми створимо фільтр підвищеного косинуса, використовуючи β 0.35, і зробимо його довжиною 101 відведення, щоб дати сигналу достатньо часу для затухання до нуля. Хоча рівняння піднесеного косинуса запитує наш період символу і вектор часу t , ми можемо припустити, що період **вибірки** дорівнює 1 секунді, щоб "нормалізувати" нашу симуляцію. Це означає, що наш період символу T_s дорівнює 8, оскільки ми маємо 8 відліків на символ. Тоді наш вектор часу буде списком цілих чисел. Зважаючи на те, як працює рівняння піднесеного косинуса, ми хочемо, щоб точка $t = 0$ була в центрі. Ми згенеруємо вектор часу довжиною 101, починаючи з -51 і закінчуючи +51.

```
# Створюємо наш фільтр підвищеного косинуса
num_taps = 101
beta = 0.35
Ts = sps # Припустимо, що частота дискретизації дорівнює 1 Гц, період дискретизації
↪ дорівнює 1, період *символу* дорівнює 8
t = np.arange(num_taps) - (num_taps-1)//2
h = 1/Ts*np.sinc(t/Ts) * np.cos(np.pi*beta*t/Ts) / (1 - (2*beta*t/Ts)**2)
plt.figure(1)
plt.plot(t, h, '-.')
plt.grid(True)
plt.show()
```

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/pulse_shaping_python2.png

Зверніть увагу, що вихідні дані однозначно спадають до нуля. Той факт, що ми використовуємо 8 відліків на символ, визначає, наскільки вузьким виглядає цей фільтр і як швидко він спадає до нуля. Наведена вище імпульсна характеристика виглядає як типовий низькочастотний фільтр, і ми не можемо визначити, що це саме фільтр, який формує імпульс, а не будь-який інший низькочастотний фільтр.

Нарешті, ми можемо відфільтрувати наш сигнал x і дослідити результат. Не зосереджуйтесь на введенні циклу `for` у наведеному коді. Ми обговоримо, навіщо він тут, після блоку коду.

```
# Фільтруємо наш сигнал, щоб застосувати формування імпульсу
x_shaped = np.convolve(x, h)
plt.figure(2)
plt.plot(x_shaped, '-.')
for i in range(num_symbols):
    plt.plot([i*sps+num_taps//2, i*sps+num_taps//2], [0, x_shaped[i*sps+num_taps//2]])
plt.grid(True)
plt.show()
```

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/pulse_shaping_python3.svg

Цей результуючий сигнал складається з багатьох наших імпульсних відгуків, приблизно половина з яких спочатку множиться на -1. Це може виглядати складно, але ми пройдемо через це разом.

По-перше, через фільтр і спосіб роботи згортки є перехідні відліки до і після даних. Ці додаткові відліки включаються в нашу передачу, але насправді вони не містять "піків" імпульсів.

Secondly, the vertical lines were created in the `for` loop for visualization's sake. They are meant to demonstrate where intervals of T_s occur. These intervals represent where this signal will be sampled by the receiver. Observe that for intervals of T_s the curve has the value of exactly 1.0 or -1.0, making them the ideal points in time to sample.

If we were to upconvert and transmit this signal, the receiver would have to determine when the boundaries of T_s are e.g., using a symbol synchronization algorithm. That way the receiver knows *exactly* when to sample to get the right data. If the receiver samples a little too early or late, it will see values that are slightly skewed due to ISI, and if it's way off then it will get a bunch of weird numbers.

Here is an example, created using GNU Radio, that illustrates what the IQ plot (a.k.a. constellation) looks like when we sample at the right and wrong times. The original pulses have their bit values annotated.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/symbol_sync1.png

The below graph represents the ideal position in time to sample, along with the IQ plot:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/symbol_sync2.png

Compare that to the worst time to sample. Notice the three clusters in the constellation. We are sampling directly in between each symbol; our samples are going to be way off.

. image:: ../_images/symbol_sync3.png scale 40 %

align center

alt Симуляція GNU Radio, що демонструє недосконалу вибірку за часом

Ось ще один приклад поганого часу дискретизації, де між нашим ідеальним і найгіршим випадками. Зверніть увагу на чотири кластери. З високим SNR ми могли б уникнути такого інтервалу часу вибірки, хоча це і не рекомендується.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/symbol_sync4.png

Пам'ятайте, що наші значення Q не показані на часовому графіку, оскільки вони приблизно дорівнюють нулю, що дозволяє графіку IQ поширюватися лише по горизонталі.

Якби ми хотіли перетворити і передати цей сигнал, приймач мав би визначити, коли знаходяться межі T_s , наприклад, за допомогою алгоритму символної синхронізації. Таким чином, приймач знає, коли саме робити вибірку, щоб отримати правильні дані. Якщо приймач зробить вибірку занадто рано або занадто пізно, він побачить значення, які будуть дещо викривлені через ISI, а якщо занадто пізно, то отримає купу дивних чисел.

Ось приклад, створений за допомогою GNU Radio, який ілюструє, як виглядає графік IQ (так зване сузір'я), коли ми робимо вибірки у правильній і неправильний час. Оригінальні імпульси мають бітові значення з анотаціями.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/symbol_sync1.png

Наведений нижче графік показує ідеальну позицію в часі для дискретизації, а також графік IQ:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/symbol_sync2.png

Порівняйте це з найгіршим часом для вибірки. Зверніть увагу на три кластери у сузір'ї. Ми робимо вибірку безпосередньо між кожним символом; наші вибірки будуть далекими від ідеальних.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/symbol_sync3.png

Ось ще один приклад поганого часу дискретизації, де між нашим ідеальним і найгіршим випадками. Зверніть увагу на чотири кластери. З високим SNR ми могли б уникнути такого інтервалу часу вибірки, хоча це і не рекомендується.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/symbol_sync4.png

Пам'ятайте, що наші значення Q не показані на часовому графіку, оскільки вони приблизно дорівнюють нулю, що дозволяє графіку IQ поширюватися лише по горизонталі.

Розділ 8

Синхронізація

У цьому розділі розглядається синхронізація бездротового сигналу в часі і частоті, щоб виправити зміщення несучої частоти і виконати вирівнювання часу на рівні символів і кадрів. Ми будемо використовувати техніку відновлення тактової частоти Мюллера і Мюллера, а також цикл Костаса в Python.

8.1. Вступ

Ми обговорили, як здійснювати цифрову передачу в ефірі, використовуючи схему цифрової модуляції, таку як QPSK, і застосовуючи формування імпульсів для обмеження смуги пропускання сигналу. Канальне кодування можна використовувати для роботи із зашумленими каналами, наприклад, коли у вас низький SNR на приймачі. Завжди корисно відфільтрувати якомога більше перед цифровою обробкою сигналу. У цьому розділі ми розглянемо, як виконується синхронізація на приймальному боці. Синхронізація - це набір операцій, які відбуваються перед демодуляцією і декодуванням каналу. Нижче показано загальний ланцюжок tx-канал-гх, де жовтим кольором виділено блоки, що розглядаються в цій главі. (Ця схема не є всеохоплюючою - більшість систем також включають еквалізацію і мультиплексування).

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/sync-diagram.svg

8.2. Моделювання бездротового каналу

Перш ніж ми навчимося реалізовувати часову та частотну синхронізацію, нам потрібно зробити наші імітовані сигнали більш реалістичними. Без додавання випадкової часової затримки, акт синхронізації в часі є тривіальним. Насправді, вам потрібно лише врахувати затримку дискретизації будь-яких фільтрів, які ви використовуєте. Ми також хочемо змодельовати зміщення частоти, тому що, як ми будемо обговорювати, генератори не є ідеальними; завжди буде деякий зсув між центральною частотою передавача і приймача.

Розглянемо код на Python для моделювання нецілочисельної затримки та зсуву частоти. Код на Python у цій главі почнеться з коду, який ми написали під час вправи з формування імпульсів на Python (натисніть нижче, якщо він вам потрібен); ви можете вважати його відправною точкою коду в цій главі, а весь новий код буде додано після неї.

```
import numpy as np
```

```

import matplotlib.pyplot as plt
з scipy import signal
import math

# ця частина прийшла з вправи на формування пульсу
num_symbols = 100
sps = 8
bits = np.random.randint(0, 2, num_symbols) # Наші дані для передачі, 1 та 0
pulse_train = np.array([])
для біта в бітах:
    pulse = np.zeros(sps)
    pulse[0] = bit*2-1 # встановлюємо перше значення в 1 або -1
    pulse_train = np.concatenate((pulse_train, pulse)) # додаємо 8 відліків до сигналу

# створюємо наш фільтр підвищеної косинусоїди
num_taps = 101
beta = 0.35
Ts = sps # Припустимо, що частота дискретизації 1 Гц, період дискретизації 1, період
↳ *символу* 8
t = np.arange(-51, 52) # пам'ятайте, що це не включно з кінцевим числом
h = np.sinc(t/Ts) * np.cos(np.pi*beta*t/Ts) / (1 - (2*beta*t/Ts)**2)

# Фільтруємо наш сигнал, щоб застосувати формування імпульсів
samples = np.convolve(pulse_train, h)

```

Ми пропустимо код, пов'язаний з побудовою графіків, оскільки ви вже навчилися будувати графіки будь-яких сигналів. Надання графікам красивого вигляду, як це часто робиться у цьому підручнику, вимагає багато додаткового коду, який не обов'язково розуміти.

8.2.1. Додавання затримки

Ми можемо легко імітувати затримку, зсуваючи відліки, але це імітує лише затримку, яка є цілим числом, кратним періоду нашого відліку. У реальному світі затримка буде становити деяку частку від періоду зразка. Ми можемо імітувати затримку на частку відрізка, створивши фільтр "дробової затримки", який пропускає всі частоти, але затримує відрізки на деяку величину, яка не обмежується інтервалом відрізка. Ви можете думати про це як про багатосмуговий фільтр, який застосовує однаковий фазовий зсув до всіх частот. (Нагадаємо, що часова затримка і фазовий зсув еквівалентні.) Код на Python для створення цього фільтра наведено нижче:

```

# Створити і застосувати фільтр дробової затримки
delay = 0.4 # дробова затримка, у відліках
N = 21 # кількість відведень
n = np.arange(-N/2, N/2) # ...-3,-2,-1,0,1,2,3...
h = np.sinc(n - delay) # обчислюємо відгалуження фільтру
h *= np.hamming(N) # вікно фільтра, щоб переконатися, що він розпадається до 0 з обох
↳ боків
h /= np.sum(h) # нормалізуємо, щоб отримати одиничний коефіцієнт підсилення, ми не хочемо
↳ змінювати амплітуду/потужність
samples = np.convolve(samples, h) # застосовуємо фільтр

```

Як бачите, ми обчислюємо відводи фільтра за допомогою функції `sinc()`. `Sinc` у часовій області - це прямокутник у частотній області, і наш прямокутник для цього фільтра охоплює весь частотний діапазон нашого сигналу. Цей фільтр не змінює форму сигналу, він лише затримує його в часі. У нашому прикладі ми затримуємо на 0,4 відрізка. Майте на увазі, що застосування *будь-якого* фільтра

затримує сигнал на половину відліків фільтра мінус один, через акт згортки сигналу через фільтр.

Якщо ми побудуємо графік "до" і "після" фільтрації сигналу, то зможемо побачити дробову затримку. На нашому графіку ми збільшили масштаб лише на кілька символів. Інакше дробову затримку не видно.

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: [../_images/fractional-delay-filter.svg](#)

8.2.2. Додавання частотного зсуву

Щоб зробити наш імітований сигнал більш реалістичним, ми застосуємо частотний зсув. Скажімо, наша частота дискретизації в цій симуляції становить 1 МГц (насправді не має значення, якою вона буде, але ви побачите, чому це полегшує вибір числа). Якщо ми хочемо змодельовати зсув частоти на 13 кГц (якесь довільне число), ми можемо зробити це за допомогою наступного коду:

```
# застосовуємо зсув частоти
fs = 1e6 # вважаємо, що наша частота дискретизації дорівнює 1 МГц
fo = 13000 # імітуємо зсув частоти
Ts = 1/fs # обчислюємо період дискретизації
t = np.arange(0, Ts*len(samples), Ts) # створюємо вектор часу
samples = samples * np.exp(1j*2*np.pi*fo*t) # виконуємо зсув частоти
```

Нижче демонструється сигнал до і після застосування зсуву частоти.

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: [../_images/sync-freq-offset.svg](#)

Ми не будували графік Q-частини, оскільки передавали BPSK, і тому Q-частина завжди дорівнювала нулю. Тепер, коли ми додаємо частотний зсув для імітації бездротових каналів, енергія розподіляється між I і Q. З цього моменту ми повинні будувати графіки як I, так і Q. Не соромтеся підставляти інший частотний зсув для вашого коду. Якщо ви зменшите зсув приблизно до 1 кГц, ви зможете побачити синусоїду в огинаючій сигналу, оскільки вона коливається досить повільно, щоб охопити кілька символів.

Що стосується вибору довільної частоти дискретизації, то якщо ви уважно подивитесь на код, то помітите, що важливим є співвідношення f_o до f_s .

Можна уявити, що два блоки коду, представлені раніше, імітують бездротовий канал. Код повинен стояти після коду на стороні передачі (що ми робили у розділі про формування імпульсів) і перед кодом на стороні прийому, який ми розглянемо у решті частини цього розділу.

8.3. Синхронізація часу

Коли ми передаємо сигнал бездротовим способом, він надходить до приймача з випадковим фазовим зсувом через пройдений час. Ми не можемо просто почати вибірку символів з нашою швидкістю, тому що ми навряд чи зможемо зробити вибірку у потрібній точці імпульсу, як це обговорюється у кінці розділу `pulse-shaping-chapter`. Якщо ви не зрозуміли, перегляньте три рисунки в кінці цього розділу.

Більшість методів синхронізації мають форму петлі фазової автопідстроювання (ФАПЧ); ми не розглядатимемо ФАПЧ тут, але важливо знати цей термін, і ви можете почитати про них самостійно, якщо вам цікаво. ФАПЧ - це замкнені

системи, які використовують зворотний зв'язок для постійного коригування чогось; у нашому випадку зсув у часі дозволяє нам робити вибірки на піку цифрових символів.

Ви можете уявити відновлення синхронізації як блок в приймачі, який приймає потік відліків і видає інший потік відліків (подібно до фільтра). Ми програмуємо цей блок відновлення синхронізації інформацією про наш сигнал, найважливішою з яких є кількість відліків на символ (або наше найкраще припущення, якщо ми не впевнені на 100%, що було передано). Цей блок діє як "дециматор", тобто наша вихідна вибірка буде часткою від кількості вхідних відліків. Нам потрібен один відлік на цифровий символ, тому частота децимації - це просто кількість відліків на символ. Якщо передавач передає зі швидкістю 1 млн. символів на секунду, а ми робимо дискретизацію зі швидкістю 16 Мс, то отримаємо 16 відліків на символ. Це буде частота дискретизації, що надходить у блок синхронізації. Частота дискретизації на виході з блоку буде 1 Msps, тому що нам потрібна одна вибірка на цифровий символ.

Більшість методів відновлення синхронізації ґрунтуються на тому, що наші цифрові символи піднімаються, а потім опускаються, і вершина - це точка, в якій ми хочемо зробити вибірку символу. Іншими словами, ми робимо вибірку в максимальній точці після зняття абсолютного значення:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/symbol_sync2.png

Існує багато методів відновлення синхронізації, які здебільшого нагадують ШІМ. Різниця між ними полягає у рівнянні, яке використовується для виконання "корекції" зсуву синхронізації, яке ми позначаємо як μ або m_i у коді. Значення m_i оновлюється на кожній ітерації циклу. Це значення в одиницях відліків, і ви можете думати про нього як про те, на скільки ми повинні зміститися, щоб мати змогу зробити вибірку в "ідеальний" момент часу. Отже, якщо $m_i = 3.61$, це означає, що нам потрібно зсунути вхідні дані на 3.61 відліки, щоб зробити вибірку в потрібному місці. Оскільки ми маємо 8 відліків на символ, якщо m_i перевищить 8, він просто повернеться до нуля.

Наступний код на Python реалізує техніку Мюллера і відновлення тактового генератора Мюллера.

```

mu = 0 # початкова оцінка фази зразка
out = np.zeros(len(samples) + 10, dtype=np.complex)
out_rail = np.zeros(len(samples) + 10, dtype=np.complex) # зберігає значення, на кожній
→ ітерації нам потрібні 2 попередні значення плюс поточне значення
i_in = 0 # індекс вхідних відліків
i_out = 2 # індекс виходу (нехай перші два виходи дорівнюють 0)
while i_out < len(samples) and i_in+16 < len(samples):
    out[i_out] = samples[i_in + int(mu)] # беремо те, що вважаємо "найкращим" зразком
    out_rail[i_out] = int(np.real(out[i_out]) > 0) + 1j*int(np.imag(out[i_out]) > 0)
    x = (out_rail[i_out] - out_rail[i_out-2]) * np.conj(out[i_out-1])
    y = (out[i_out] - out[i_out-2]) * np.conj(out_rail[i_out-1])
    mm_val = np.real(y - x)
    mu += sps + 0.3*mm_val
    i_in += int(np.floor(mu)) # округляємо до найближчого int, оскільки використовуємо
    → його як індекс
    mu = mu - np.floor(mu) # видаляємо цілу частину mu
    i_out += 1 # збільшити індекс виводу
out = out[2:i_out] # видаляємо перші два рядки і все, що після i_out (що ніколи не
→ заповнювалось)
samples = out # включайте цей рядок лише у тому випадку, якщо ви хочете пізніше з'єднати
→ цей фрагмент коду з циклом Костаса

```

На блок відновлення синхронізації подаються "отримані" відліки, і він видає

вихідний відлік по одному за раз (зверніть увагу на те, що `i_out` збільшується на 1 на кожній ітерації циклу). Блок відновлення не просто використовує "отримані" зразки один за одним, тому що цикл коригує `i_in`. Він пропускає деякі відліки, намагаючись витягнути "правильний" відлік, тобто той, що знаходиться на піку імпульсу. Коли цикл обробляє відліки, він повільно синхронізується з символом, або, принаймні, намагається це зробити, змінюючи `mi`. Враховуючи структуру коду, ціла частина `mi` додається до `i_in`, а потім вилучається з `mi` (майте на увазі, що `mm_val` може бути від'ємною або додатною кожного циклу). Після повної синхронізації цикл має витягувати лише центральний відлік з кожного символу/імпульсу. Ви можете налаштувати константу 0.3, яка змінює швидкість реакції циклу зворотного зв'язку; більше значення робить його реакцію швидшою, але з більшим ризиком проблем зі стабільністю.

На наступному графіку показано приклад вихідного сигналу, де ми *відключили* дробову затримку, а також зміщення частоти. Ми показуємо лише I, оскільки Q - це нулі, а частотний зсув вимкнено. Три графіки накладено один на одного, щоб показати, як біти вирівнюються по вертикалі.

Верхній графік Оригінальні символи BPSK, тобто 1 і -1. Пам'ятайте, що між ними є нулі, тому що нам потрібно 8 вибірок на символ.

Середній графік Відліки після формування імпульсів, але до синхронізатора.

Нижній графік Вихід символьного синхронізатора, який забезпечує лише 1 вибірку на символ. Тобто ці відліки можна подавати безпосередньо на демодулятор, який для BPSK перевіряє, чи значення більше або менше 0.

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: `../_images/time-sync-output.svg`

Зосередимося на нижньому графіку, який є виходом синхронізатора. Знадобилося майже 30 символів, щоб синхронізація зафіксувалася з потрібною затримкою. Через неминучий час, необхідний для синхронізації, багато протоколів зв'язку використовують преамбулу, яка містить послідовність синхронізації: вона діє як спосіб повідомити про те, що прибув новий пакет, і дає приймачу час для синхронізації з ним. Але після цих ~30 семплів синхронізатор працює ідеально. У нас залишаються ідеальні 1 і -1, які відповідають вхідним даним. Допомогає те, що в цьому прикладі не було додано жодного шуму. Не соромтеся додавати шум або часові зсуви і подивіться, як поводитиметься синхронізатор. Якби ми використовували QPSK, то мали б справу з комплексними числами, але підхід був би таким самим.

8.4. Синхронізація часу за допомогою інтерполяції

Синхронізатори символів, як правило, інтерполюють вхідні відліки на деяке число, наприклад, 16, так що вони можуть зміщуватися на частку відліку. Випадкова затримка, спричинена бездротовим каналом, навряд чи буде точно кратною відліку, тому пік символу може не збігатися з відліком. Це особливо актуально у випадку, коли на один символ може припадати лише 2 або 4 відліки. Інтерполюючи відліки, він дає нам можливість робити відліки "між" реальними відліками, щоб потрапити на самий пік кожного символу. На виході синхронізатора залишається лише 1 відлік на символ. Самі вхідні відліки інтерполюються.

Наш код синхронізації часу на Python, який ми реалізували вище, не включав ніякої інтерполяції. Щоб розширити наш код, увімкніть дробову часову затримку, яку ми реалізували на початку цього розділу, щоб наш отриманий сигнал мав

більш реалістичну затримку. Частотний зсув поки що залиште вимкненим. Якщо ви повторно запустите симуляцію, то побачите, що синхронізатор не може повністю синхронізуватися з сигналом. Це тому, що ми не інтерполюємо, тому код не має можливості "робити вибірку між вибірками", щоб компенсувати дробову затримку. Давайте додамо інтерполяцію.

Швидкий спосіб інтерполювати сигнал у Python - скористатися функціями `signal.resample` або `signal.resample_poly` з пакета `scipy`. Ці функції роблять одне й те саме, але працюють по-різному. Ми будемо використовувати останню функцію, оскільки вона, як правило, швидша. Давайте інтерполюватимемо на 16 (це довільний вибір, ви можете спробувати різні значення), тобто ми будемо вставляти 15 додаткових відліків між кожним відліком. Це можна зробити в одному рядку коду, і це повинно відбутися до того, як ми підемо виконувати синхронізацію часу (до великого фрагмента коду вище). Давайте також побудуємо графік до і після, щоб побачити різницю:

```
samples_interpolated = signal.resample_poly(samples, 16, 1)

# Побудувати графік старого та нового
plt.figure('before interp')
plt.plot(samples, '-.')
plt.figure('after interp')
plt.plot(samples_interpolated, '-.')
plt.show()
```

Якщо ми збільшимо масштаб, то побачимо, що це той самий сигнал, тільки з 16х більшою кількістю точок:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: [../_images/time-sync-interpolated-samples.svg](#)

Сподіваємось, причина, чому нам потрібно інтерполювати всередині блоку синхронізації часу, стає зрозумілою. Ці додаткові вибірки дозволять нам врахувати частку затримки вибірки. На додаток до обчислення `samples_interpolated`, нам також потрібно змінити один рядок коду в нашому синхронізаторі часу. Ми змінимо перший рядок всередині циклу `while` на `become`:

```
out[i_out] = samples_interpolated[i_in*16 + int(mu*16)]
```

Тут ми зробили кілька речей. По-перше, ми більше не можемо просто використовувати `i_in` як індекс вхідної вибірки. Ми повинні помножити його на 16, тому що ми інтерполювали наші вхідні відліки на 16. Пам'ятайте, що цикл зворотного зв'язку коригує змінну `mu`. Вона являє собою затримку, яка призводить до того, що ми робимо вибірку в потрібний момент. Також нагадаємо, що після обчислення нового значення `mu` ми додали цілу частину до `i_in`. Тепер ми будемо використовувати залишок, який є плаваючою частиною від 0 до 1, і представляє собою частку відрізка, на яку нам потрібно затримати дискретизацію. Раніше ми не могли затримати на частку відліку, але тепер ми можемо, принаймні з кроком у 16 частин відліку. Ми множимо `mu` на 16, щоб дізнатися, на скільки відліків нашого інтерпольованого сигналу нам потрібно затримати. А потім ми повинні округлити це число, оскільки значення в дужках є індексом і має бути цілим числом. Якщо цей абзац не має сенсу, спробуйте повернутися до початкового коду Мюллера і відновлення годинника Мюллера, а також прочитайте коментарі біля кожного рядка коду.

Фактичний вивід графіка цього нового коду має виглядати приблизно так само, як і раніше. Ми лише зробили нашу симуляцію більш реалістичною, додавши

затримку дробової вибірки, а потім додали інтерполятор до синхронізатора, щоб компенсувати цю затримку дробової вибірки.

Не соромтеся експериментувати з різними коефіцієнтами інтерполяції, тобто змінюйте всі 16 с на інші значення. Ви також можете спробувати увімкнути частотний зсув або додати білий гаусівський шум до сигналу до того, як він буде отриманий, щоб побачити, як це впливає на якість синхронізації (підказка: можливо, вам доведеться відкоригувати множник 0.3).

Якщо ми увімкнемо лише зміщення частоти, використовуючи частоту 1 кГц, ми отримаємо такі показники часової синхронізації. Тепер, коли ми додали зсув частоти, нам потрібно показати I і Q :

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: [../_images/time-sync-output2.svg](#)

Можливо, це важко помітити, але синхронізація часу все ще працює чудово. Потрібно приблизно 20-30 символів, щоб вона зафіксувалася. Однак, ми бачимо синусоїду, тому що у нас все ще є зсув частоти, і ми дізнаємося, як з ним впоратися в наступному розділі.

Нижче показано IQ-графік (так званий графік сузір'я) сигналу до і після синхронізації. Пам'ятайте, що ви можете нанести відліки на IQ-діаграму за допомогою діаграми розсіювання: `plt.plot(np.real(samples), np.imag(samples), '.')`. На анімації нижче ми спеціально пропустили перші 30 символів. Вони з'явилися до того, як закінчилася синхронізація часу. Всі символи, що залишилися, знаходяться приблизно на одиничному колі через зсув частоти.

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: [../_images/time-sync-constellation.svg](#)

Щоб отримати ще більше розуміння, ми можемо подивитися на сузір'я в часі, щоб побачити, що насправді відбувається з символами. На самому початку, протягом короткого проміжку часу, символи не дорівнюють 0 і не знаходяться на одиничному колі. Це період, коли синхронізація часу знаходить правильну затримку. Це відбувається дуже швидко, слідкуйте уважно! Обертання - це просто зсув частоти. Частота - це постійна зміна фази, тому зміщення частоти спричиняє обертання BPSK (створення кола на статичному/постійному графіку вище).

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: [../_images/time-sync-constellation-animated.gif](#)

Сподіваюся, побачивши приклад реальної синхронізації часу, ви зрозуміли, що вона робить, і отримали загальне уявлення про те, як вона працює. На практиці, створений нами цикл `while` працюватиме лише з невеликою кількістю семплів за раз (наприклад, 1000). Ви повинні пам'ятати значення між викликами функції синхронізації, а також останні пару значень `out` і `out_rail`.

Далі ми розглянемо частотну синхронізацію, яку ми розділили на грубу і точну. Груба зазвичай відбувається перед синхронізацією, а точна - після.

8.5. Груба частотна синхронізація

Навіть якщо ми скажемо передавачу і приймачу працювати на одній центральній частоті, між ними буде невеликий зсув частоти через недосконалість обладнання (наприклад, генератора) або доплерівський зсув від руху. Цей зсув частоти буде крихітним відносно несучої частоти, але навіть невеликий зсув може призвести до спотворення цифрового сигналу. Зсув, ймовірно, змінюватиметься з часом,

що вимагає постійного зворотного зв'язку для корекції зсуву. Наприклад, генератор всередині Плутона має максимальний зсув 25 PPM. Це 25 частин на мільйон відносно центральної частоти. Якщо ви налаштовані на 2,4 ГГц, максимальне зміщення становитиме +/- 60 кГц. Зразки, які нам надає SDR, знаходяться в базовій смузі частот, тому будь-яке зміщення частоти проявляється в цьому сигналі базової смуги. Сигнал BPSK з невеликим зсувом несучої буде виглядати приблизно так, як показано на часовій діаграмі нижче, що, очевидно, не дуже добре для демодуляції бітів. Перед демодуляцією ми повинні видалити будь-які частотні зсуви.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/carrier-offset.png

Частотну синхронізацію зазвичай поділяють на грубу та точну, де груба синхронізація виправляє великі зсуви порядку кГц або більше, а точна - все, що залишилося. Груба синхронізація відбувається до часової синхронізації, а точна - після.

Математично, якщо у нас є сигнал базової смуги $s(t)$ і він зазнає частотного (так званого несучого) зсуву на f_o Гц, ми можемо представити те, що отримуємо, як:

$$r(t) = s(t)e^{j2\pi f_o t} + n(t)$$

де $n(t)$ - шум.

Перший трюк, якому ми навчимося, щоб виконати грубу оцінку частотного зсуву (якщо ми можемо оцінити частоту зсуву, то ми можемо його виправити), - це взяти квадрат нашого сигналу. Для спрощення обчислень проігноруємо шум:

$$r^2(t) = s^2(t)e^{j4\pi f_o t}$$

Давайте подивимося, що станеться, коли ми піднесемо до квадрату наш сигнал $s(t)$, розглянувши, що зробить QPSK. Піднесення комплексних чисел до квадрата призводить до цікавої поведінки, особливо коли ми говоримо про такі сузір'я, як BPSK і QPSK. Наступна анімація показує, що відбувається, коли ви підносите QPSK до квадрата, а потім знову підносите до квадрата. Я спеціально використовував QPSK замість BPSK, тому що ви можете бачити, що коли ви підносите QPSK до квадрата один раз, ви по суті отримуєте BPSK. А потім після ще одного квадратування це стає одним кластером. (Дякуємо <http://ventrella.com/ComplexSquaring/>, який створив цей чудовий веб-додаток).

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/squaring-qpsk.gif

Давайте подивимося, що станеться, коли до нашого QPSK-сигналу застосувати невеликий поворот фази і масштабування амплітуди, що є більш реалістичним:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/squaring-qpsk2.gif

Це все одно стає одним кластером, просто зі зсувом по фазі. Основний висновок полягає в тому, що якщо ви піднесете QPSK до квадрата двічі (а BPSK - один раз), це об'єднає всі чотири кластери точок в один кластер. Чому це корисно? Ну, об'єднуючи кластери, ми по суті видаляємо модуляцію! Якщо всі точки тепер знаходяться в одному кластері, це все одно, що мати купу констант підряд. Це як якщо б модуляції більше не було, і єдине, що залишилося - це синусоїда, викликана зсувом частоти (у нас також є шум, але давайте поки що ігнорувати його). Виходить, що потрібно піднести сигнал до квадрату N разів, де N - це порядок використовуваної схеми модуляції, а це означає, що цей трюк працює лише тоді, коли ви знаєте схему модуляції заздалегідь. Рівняння дійсно має вигляд:

$$r^N(t) = s^N(t)e^{j2N\pi f_o t}$$

Для нашого випадку BPSK ми маємо схему модуляції 2-го порядку, тому для грубої частотної синхронізації будемо використовувати наступне рівняння:

$$r^2(t) = s^2(t)e^{j4\pi f_o t}$$

Ми з'ясували, що відбувається з частиною рівняння $s(t)$, але як щодо синусоїдальної частини (так званої комплексної експоненти)? Як бачимо, до неї додається член N , що робить її еквівалентною синусоїді на частоті Nf_o , а не просто f_o . Простий метод обчислення f_o - це взяти ШПФ сигналу після того, як ми піднесемо його до квадрату N разів і подивимось, де відбувається сплеск. Даваймо змодельуємо це на Python. Ми повернемося до генерації нашого BPSK-сигналу, і замість дробової затримки застосуємо до нього частотний зсув, помноживши сигнал на $e^{j2\pi f_o t}$ так само, як ми це робили у розділі `filters-chapter` для перетворення фільтра нижніх частот на фільтр верхніх частот.

Використовуючи код з початку цього розділу, додайте до вашого цифрового сигналу частотний зсув +13 кГц. Це може статися безпосередньо перед або відразу після додавання дробової затримки; це не має значення. Незалежно від цього, це повинно відбутися "після" формування імпульсів, але до того, як ми виконаємо будь-які функції на стороні прийому, такі як синхронізація часу.

Тепер, коли у нас є сигнал зі зсувом частоти на 13 кГц, даваймо побудуємо графік ШПФ до і після зведення в квадрат, щоб побачити, що відбувається. На цей момент ви вже повинні знати, як робити ШПФ, включаючи операції `abs()` і `fftshift()`. Для цієї вправи не має значення, чи берете ви лог, чи ні, чи підносите його до квадрату після застосування функції `abs()`.

Спочатку подивіться на сигнал до піднесення до квадрату (звичайне ШПФ):

```
psd = np.fft.fftshift(np.abs(np.fft.fft(samples)))
f = np.linspace(-fs/2.0, fs/2.0, len(psd))
plt.plot(f, psd)
plt.show()
```

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: [../_images/coarse-freq-sync-before.svg](#)

Насправді ми не бачимо жодного піку, пов'язаного зі зміщенням несучої. Він перекритий нашим сигналом.

Тепер додамо квадратуру (просто степінь 2, тому що це BPSK):

```
# Додаємо це перед рядком ШПФ
samples = samples**2
```

Ми повинні збільшити зображення, щоб побачити, на якій частоті знаходиться пік:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: [../_images/coarse-freq-sync.svg](#)

Ви можете спробувати збільшити кількість імітованих символів (наприклад, до 1000 символів), щоб мати достатньо зразків для роботи. Чим більше вибірок буде використано у нашому ШПФ, тим точніше буде наша оцінка частотного зсуву. Нагадую, що наведений вище код повинен стояти "до" синхронізатора.

Стрибок частоти зсуву з'являється за адресою Nf_o . Нам потрібно розділити цей бін (26,6 кГц) на 2, щоб отримати остаточну відповідь, яка дуже близька до

зсуву частоти на 13 кГц, який ми застосували на початку розділу! Якщо ви погралися з цим числом і воно вже не дорівнює 13 кГц, нічого страшного. Просто переконайтеся, що ви усвідомлюєте, на якому значенні ви його встановили.

Оскільки наша частота дискретизації становить 1 МГц, максимальні частоти, які ми можемо побачити, знаходяться в діапазоні від -500 кГц до 500 кГц. Якщо ми піднесемо наш сигнал до степеня N , це означає, що ми зможемо "побачити" лише частотні зсуви до $500e3/N$, або у випадку BPSK +/- 250 кГц. Якби ми приймали сигнал QPSK, то його частота була б лише +/- 125 кГц, а зсув несучої вище або нижче цього значення був би поза межами нашого діапазону за допомогою цього методу. Щоб дати вам уявлення про доплерівський зсув, якби ви передавали в діапазоні 2,4 ГГц, а передавач або приймач рухалися зі швидкістю 60 миль/год (важлива відносна швидкість), це призвело б до зсуву частоти на 214 Гц. Зсув через низьку якість генератора, ймовірно, буде головним винуватцем у цій ситуації.

Насправді виправлення цього зсуву частоти відбувається саме так, як ми імітували зсув спочатку: множенням на комплексну експоненту, тільки з від'ємним знаком, оскільки ми хочемо видалити зсув.

```
max_freq = f[np.argmax(psd)]
Ts = 1/fs # розраховуємо період дискретизації
t = np.arange(0, Ts*len(samples), Ts) # створюємо вектор часу
samples = samples * np.exp(-1j*2*np.pi*max_freq*t/2.0)
```

Вам вирішувати, чи хочете ви це виправити або змінити початкове зміщення частоти, яке ми застосували на початку, на менше число (наприклад, 500 Гц), щоб протестувати точну частотну синхронізацію, яку ми зараз навчимося робити.

8.6. Точна частотна синхронізація

Далі ми перемкнемо передачу на точну частотну синхронізацію. Попередній трюк більше підходить для грубої синхронізації, і він не є операцією із замкнутим контуром (типу зворотного зв'язку). Але для точної частотної синхронізації нам потрібен контур зворотного зв'язку, через який ми пропускаємо семпли, що знову ж таки буде формою PLL. Наша мета - звести частотний зсув до нуля і утримувати його на цьому рівні, навіть якщо зсув змінюється з часом. Ми повинні постійно відстежувати зміщення. Методи точної частотної синхронізації найкраще працюють з сигналом, який вже було синхронізовано в часі на рівні символів, тому код, який ми обговорюватимемо в цьому розділі, з'явиться *після* синхронізації в часі.

Ми будемо використовувати техніку, яка називається петлею Костаса. Це форма ШПФ, яка спеціально розроблена для корекції зсуву несучої частоти для цифрових сигналів, таких як BPSK і QPSK. Вона була винайдена Джоном П. Костасом в General Electric в 1950-х роках і мала великий вплив на сучасні цифрові комунікації. Петля Костаса усуває зсув частоти, а також фіксує будь-який зсув фази. Енергія вирівнюється з віссю I. Частота - це лише зміна фази, тому їх можна відстежувати як одне ціле. Петлю Костаса узагальнено за допомогою наступної діаграми (зауважте, що $1/2s$ не враховано в рівняннях, оскільки вони не мають функціонального значення).

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/costas-loop.svg

Генератор, керований напругою (VCO) - це просто генератор хвиль \sin/\cos , який використовує частоту на основі вхідного сигналу. У нашому випадку, оскільки ми моделюємо бездротовий канал, це не напруга, а скоріше рівень, представ-

лений змінною. Він визначає частоту і фазу генерованих синусоїдальних і косинусоїдальних хвиль. Він множить отриманий сигнал на внутрішньо згенеровану синусоїду, намагаючись вирівняти зсув частоти і фази. Ця поведінка схожа на те, як SDR перетворює сигнал вниз і створює гілки I і Q.

Нижче наведено код на Python, який є нашим циклом Костаса:

```
N = len(samples)
phase = 0
freq = 0
# Наступні два параметри - це те, що потрібно налаштувати, щоб зробити цикл зворотного
↳ зв'язку швидшим або повільнішим (що впливає на стабільність)
alpha = 0.132
beta = 0.00932
out = np.zeros(N, dtype=np.complex)
freq_log = []
for i in range(N):
    out[i] = samples[i] * np.exp(-1j*phase) # коригуємо вхідну вибірку на величину,
    ↳ обернену до оціненого фазового зсуву
    error = np.real(out[i]) * np.imag(out[i]) # Це формула похибки для петлі Костаса 2-го
    ↳ порядку (наприклад, для BPSK)

    # Просуваємо цикл (перераховуємо фазу і зсув частоти)
    freq += (beta * error)
    freq_log.append(freq * fs / (2*np.pi)) # перетворення кутової швидкості у Гц для
    ↳ логування
    phase += freq + (alpha * error)

    # Необов'язково: Відрегулюйте фазу так, щоб вона завжди була між 0 і 2pi, пам'ятайте,
    ↳ що фаза обертається навколо кожних 2pi
    while phase >= 2*np.pi:
        phase -= 2*np.pi
    while phase < 0:
        phase += 2*np.pi

# Побудувати графік залежності freq від часу, щоб побачити, скільки часу потрібно для
↳ досягнення потрібного зсуву
plt.plot(freq_log, '-.')
plt.show()
```

Тут багато рядків, тому давайте пройдемося по ним. Деякі рядки прості, а деякі дуже складні. `samples` - це наші вхідні дані, а `out` - вихідні. `phase` і `frequency` схожі на те з коду часової синхронізації. Вони містять поточні оцінки зсуву, і на кожній ітерації циклу ми створюємо вихідні відліки шляхом множення вхідних відліків на $\text{np.exp}(-1j \cdot \text{phase})$. Змінна `error` містить метрику "помилки", і для циклу Костаса 2-го порядку це дуже просте рівняння. Ми множимо дійсну частину відліку (I) на уявну частину (Q), і оскільки Q має дорівнювати нулю для BPSK, функція помилки мінімізується, коли немає фазового або частотного зсуву, який спричиняє зміщення енергії від I до Q. Для петлі Костаса 4-го порядку це все ще відносно просто, але не зовсім в один рядок, оскільки і I, і Q матимуть енергію навіть за відсутності фазового або частотного зсуву, для QPSK. Якщо вам цікаво, як вона виглядає, натисніть нижче, але ми поки що не будемо використовувати її в нашому коді. Причина, чому це працює для QPSK, полягає в тому, що коли ви берете абсолютне значення I і Q, ви отримаєте $\sqrt{2}$, і якщо немає фазового або частотного зсуву, то різниця між абсолютними значеннями I і Q повинна бути близькою до нуля.

```
# Для QPSK
def phase_detector_4(sample):
```

```

if sample.real > 0:
    a = 1.0
else
    a = -1.0
if sample.imag > 0
    b = 1.0
else
    b = -1.0
return a * sample.imag - b * sample.real

```

Змінні α і β визначають швидкість оновлення фази і частоти відповідно. Існує певна теорія, чому я вибрав саме ці два значення, але ми не будемо розглядати її тут. Якщо вам цікаво, ви можете спробувати змінити значення α та/або β і подивитися, що станеться.

Ми записуємо значення freq на кожній ітерації, щоб в кінці побудувати графік і побачити, як петля Костаса сходиться до правильного частотного зсуву. Нам потрібно помножити freq на частоту дискретизації і перевести з кутової частоти в Гц, поділивши на 2π . Зауважте, що якщо ви виконували синхронізацію часу перед циклом Костаса, вам доведеться також поділити на ваше значення sps (наприклад, 8), тому що семпли, які виходять з синхронізації часу, мають частоту, що дорівнює вашій початковій частоті, поділеній на sps .

Нарешті, після перерахунку фази, ми додаємо або забираємо достатню кількість 2π , щоб утримати фазу між 0 і 2π , що обертає фазу навколо.

Наш сигнал до і після петлі Костаса виглядає так:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: [../_images/costas-loop-output.svg](#)

І оцінка зсуву частоти з часом, зупиняючись на правильному зсуві (в цьому прикладі сигналу було використано зсув -300 Гц):

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: [../_images/costas-loop-freq-tracking.svg](#)

Алгоритму потрібно майже 70 відліків, щоб повністю зафіксуватися на частотному зсуві. Ви можете бачити, що в моєму симульованому прикладі після грубої частотної синхронізації залишилося близько -300 Гц. У вас може бути інакше. Як я вже згадував раніше, ви можете вимкнути грубу частотну синхронізацію і встановити початкове зміщення частоти на будь-яке значення, яке ви хочете, і подивитися, чи зрозуміє це петля Костаса.

Петля Костаса, окрім усунення зсуву частоти, вирівняла наш BPSK-сигнал по I-частині, зробивши добротність знову нульовою. Це зручний побічний ефект петлі Костаса, і він дозволяє петлі Костаса по суті діяти як наш демодулятор. Тепер все, що нам потрібно зробити, це взяти I і подивитися, чи є він більшим або меншим за нуль. Насправді ми не знатимемо, як перетворити від'ємне і додатне значення на 0 і 1, тому що інверсія може бути, а може і не бути; петля Костаса (або наша синхронізація часу) ніяк не може про це дізнатися. Саме тут в гру вступає диференціальне кодування. Воно усуває двозначність, тому що 1 і 0 базуються на тому, чи змінився символ, а не на тому, чи був він +1 чи -1. Якби ми додали диференціальне кодування, ми б все одно використовували BPSK. Ми б додали блок диференціального кодування безпосередньо перед модуляцією на стороні tx і відразу після демодуляції на стороні rx .

Нижче наведено анімацію роботи часової синхронізації плюс частотної синхронізації, часова синхронізація насправді відбувається майже миттєво, але частотна синхронізація займає майже весь час анімації, і це тому, що α і β

були встановлені занадто низько, до 0.005 і 0.001 відповідно. Код, використаний для створення цієї анімації, можна знайти [тут](#).

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/costas_animation.gif

8.7. Синхронізація кадрів

Ми обговорили, як виправити будь-які часові, частотні та фазові зсуви в отриманому сигналі. Але більшість сучасних протоколів зв'язку не є просто потоковою передачею бітів зі 100% робочим циклом. Замість цього вони використовують пакети/кадри. На приймачі нам потрібно мати можливість визначити, коли починається новий кадр. Зазвичай заголовок кадру (на рівні MAC) містить інформацію про кількість байт у кадрі. Ми можемо використовувати цю інформацію, щоб дізнатися довжину кадру, наприклад, в одиницях виміру або символах. Тим не менш, визначення початку кадру є окремим завданням. Нижче показано приклад структури кадру WiFi. Зверніть увагу, що найпершим передається заголовок фізичного рівня, а перша половина цього заголовка є "преамбулою". Ця преамбула містить послідовність синхронізації, яку приймач використовує для виявлення початку кадрів, і ця послідовність відома приймачу заздалегідь.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/wifi-frame.png

Поширеним і простим методом виявлення цих послідовностей на приймачі є перехресна кореляція отриманих зразків з відомою послідовністю. Коли послідовність зустрічається, ця крос-кореляція нагадує автокореляцію (з додаванням шуму). Зазвичай послідовності, вибрані для преамбул, мають гарні автокореляційні властивості, наприклад, автокореляція послідовності створює єдиний сильний пік в точці 0 і не має інших піків. Одним з прикладів є коди Баркера, у 802.11/WiFi послідовність Баркера довжиною 11 використовується для швидкостей 1 і 2 Мбіт/с:

+1 +1 +1 -1 -1 -1 +1 -1 -1 +1 -1

Ви можете думати про це як про 11 символів BPSK. Ми можемо дуже легко подивитися на автокореляцію цієї послідовності в Python:

```
import numpy as np
import matplotlib.pyplot as plt
x = [1,1,1,-1,-1,-1,1,-1,-1,1,-1]
plt.plot(np.correlate(x,x,'same'),'-.')
plt.grid()
plt.show()
```

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/barker-code.svg

Ви бачите, що по центру стоїть 11 (довжина послідовності), а для всіх інших затримок - -1 або 0. Це добре працює для пошуку початку кадру, тому що по суті інтегрує енергію 11 символів, намагаючись створити 1 бітовий сплеск на виході крос-кореляції. Насправді, найскладніша частина виявлення початку кадру - це визначення правильного порогу. Ви не хочете, щоб кадри, які насправді не є частиною вашого протоколу, викликали його спрацьовування. Це означає, що на додаток до перехресної кореляції вам також потрібно виконати певну нормалізацію потужності, яку ми тут не розглядатимемо. Вибираючи поріг, ви повинні знайти

компроміс між ймовірністю виявлення та ймовірністю хибних тривог. Пам'ятайте, що заголовок кадру містить інформацію, тому деякі хибні тривоги є нормальними; ви швидко виявите, що це насправді не кадр, коли спробуєте декодувати заголовок, і CRC неминуче зазнає невдачі (тому що це насправді не кадр). Проте, хоча деякі хибні спрацьовування є нормальними, відсутність виявлення кадру взагалі є поганим явищем.

Ще одна послідовність з чудовими автокореляційними властивостями - це послідовності ЗадOFFа-Чу, які використовуються в LTE. Їх перевага полягає в тому, що вони є наборами; ви можете мати кілька різних послідовностей, які мають хороші автокореляційні властивості, але вони не спрацьовуватимуть одна з одною (тобто також мають хороші властивості перехресної кореляції, коли ви перехресно корелюєте різні послідовності в наборі). Завдяки цій властивості різним вежам мобільного зв'язку будуть присвоєні різні послідовності, щоб телефон міг не тільки знайти початок кадру, але й знати, з якої вежі він отримує сигнал.

Розділ 9

Канальне кодування

У цьому розділі ми ознайомимось з основами кодування каналу, таким як пряма корекція помилок під час передачі (FEC), межі Шеннона, кодами Хеммінга, Турбо-кодами та LDPC-кодами. Кодування каналу — це величезна галузь бездротових комунікацій та галузь "теорії інформації", яка вивчає кількісну оцінку, зберігання та передачу інформації.

9.1. Для чого потрібне канальне кодування

Як ми вже дізналися noise-chapter з розділу про шуми, у бездротових каналів є шуми, і наші цифрові символи досягають приймача в спотвореному стані. Якщо ви проходили курс з мережевих технологій, ви, можливо, вже знаєте про перевірку циклічним надлишковим кодом (CRC), яка **виявляє** помилки при прийомі. Мета кодування каналу полягає у виявленні **і виправленні** помилок на боці приймача. Якщо дозволено певний рівень помилок, то наприклад можна передавати дані за допомогою модуляції вищого порядку, не маючи перерв у зв'язку. Як приклад розгляньте наступні зображення констеляції, QPSK (ліворуч) та 16QAM (праворуч) з однаковим рівнем шуму. QPSK забезпечує 2 біти на символ, тоді як 16QAM має вдвічі більшу швидкість передачі даних при 4 бітах на символ. Але зверніть увагу на те, як що QPSK символи у констеляції, як правило, не перетинають межу розрізнення символів, або вісі x, y, що означає, що символи будуть прийняті приймачем правильно. Тим часом на графіку 16QAM є перекриття кластерів, і, як наслідок, буде багато неправильно прийнятих символів.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/qpsk_vs_16qam.png

Невдала перевірка CRC зазвичай призводить до повторної передачі, принаймні при використанні протоколу, як-от TCP. Якщо Аліса надсилає повідомлення Бобу, ми б воліли, щоб Бобу не доводилося надсилати повідомлення назад Алісі з проханням повторно надіслати інформацію. Метою кодування каналу є передача **надлишкової** інформації. Надлишковість є запобіжником, який зменшує кількість помилкових пакетів, повторних передач або втрачених даних.

Ми обговорили, чому нам потрібне кодування каналу, тому давайте подивимося, де воно застосовується в ланцюжку передачі-прийому:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/tx_rx_chain.svg

Зверніть увагу, що в ланцюжку передачі-прийому є декілька етапів кодування. Кодування джерела, наш перший крок, не те саме, що і кодування каналу; кодування джерела призначене для максимального стиснення даних, які потрібно

передати, так само, як коли ви архівуєте файли, щоб зменшити простір, що вони займають. Тобто вихідний сигнал блоку кодування джерела повинен бути **меншим**, ніж вхідні дані, але вихідний сигнал каналного кодування буде більшим за його вхід, оскільки додається надлишковість.

9.2. Типи Кодів

Для каналного кодування ми використовуємо "код корекції помилок". Цей код показує нам, які саме біти ми маємо передавати з урахуванням тих бітів, які нам потрібно передати. Найпростіший код називається "кодуванням повторення", і він полягає в тому, що біт просто повторюється N разів поспіль. Для коду повторення-3 кожен біт передається тричі:

- 0 000
- 1 111

Повідомлення 10010110 передається як 111000000111000111111000 після каналного кодування.

Деякі коди працюють з "блоками" вхідних бітів, а інші з "поток" бітів. Коди, що працюють з блоками, даних певної довжини, називаються "Блоковими кодами", тоді як коди, що працюють з потоком бітів, де довжина даних є довільною, називаються "згортокковими кодами". Це два основні типи кодів. Наш код повторення-3 є блоковим кодом, де кожен блок складається з трьох бітів.

До речі, ці коди корекції помилок не використовуються виключно при кодуванні бездротових каналів зв'язку. Коли ви зберігаєте інформацію на жорсткий диск або SSD, вам не доводилося замислюватися, чому при зчитуванні інформації ніколи не виникає бітових помилок? Запис та зчитування з пам'яті подібні до системи зв'язку. Контролери жорстких/SSD дисків мають вбудовану корекцію помилок. Така корекція прозора для операційної системи і її алгоритм може бути пропрієтарним, оскільки вбудован безпосередньо в жорсткий диск/SSD. Для портативних носіїв, таких як компакт-диски, корекція помилок повинна бути стандартизованою. Зазвичай коди Ріда-Соломона використовуються в CD-ROM.

9.3. Коефіцієнт Кодування

Усі методи корекції помилок містять деяку форму надлишковості. Це означає, що якщо ми хочемо передати 100 бітів інформації, нам насправді доведеться відправити **більше ніж** 100 бітів. "Коефіцієнт кодування" – це співвідношення кількості інформаційних бітів до загальної кількості відправлених бітів (тобто інформаційних плюс надлишкових бітів). Повертаючись до прикладу кодування з трьома повтореннями, якщо у мене є 100 бітів інформації, то ми можемо визначити наступне:

- 300 бітів відправляються
- Тільки 100 бітів представляють інформацію
- Коефіцієнт кодування = $100/300 = 1/3$

Коефіцієнт кодування завжди буде меншим за 1, оскільки це компроміс між надлишковістю та пропускну здатністю. Нижча швидкість коду означає більшу надлишковість і меншу пропускну здатність.

9.4. Модуляція та Кодування

У розділі modulation-chapter ми розглядали вплив шуму при різних схемах модуляції. При низькому співвідношенні сигнал/шум (SNR) вам потрібно використовувати модуляцію нижчого порядку (наприклад, QPSK) для того, щоб впоратися з шумом, а при високому рівні сигнал/шум (SNR) можна використовувати модуляцію, таку як 256QAM, для отримання більшої кількості бітів за секунду. Канальне кодування працює за аналогічним принципом; ми хочемо мати нижчі коефіцієнти кодування при низьких значеннях c/p (SNR) та коефіцієнт кодування майже рівний 1 при високих значеннях c/p (SNR). Сучасні комунікаційні системи мають набір комбінації схем модуляції та кодування, які називаються модуляційно-кодувальна схема MKC (MCS Modulation and Coding Scheme). Кожен MKC (MCS) визначає схему модуляції та кодування, які повинні використовуватися при певних рівнях c/p (SNR).

Сучасні комунікації адаптивно змінюють MKC (MCS) в реальному часі відповідно умов бездротового каналу. Приймач надає зворотний зв'язок про якість каналу передавачу. Зворотний зв'язок повинен бути отриман до того як відбудуться зміни якості каналу, що може відповідати інтервалу часу порядку мілісекунд. Цей адаптивний процес призводить до максимально можливої пропускної здатності швидкості передачі даних та використовується в сучасних технологіях, таких як LTE, 5G та WiFi. Нижче зображено як змінює MKC (MCS) базова станція під час передачі залежно від відстані до користувача.

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/adaptive_mcs.svg

При використанні адаптивної MKC (MCS), графіка залежності пропускної здатності від c/p (SNR) буде у формі сходинки, як показано нижче. Протоколи, такі як LTE, часто мають таблицю, яка вказує, який MKC (MCS) слід використовувати при якому SNR.

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/adaptive_mcs2.svg

9.5. Код Хеммінга

Давайте розглянемо прості коди корекції помилок. Код Хеммінга був першим розробленим нетривіальним кодом. В кінці 1940-х років Річард Хеммінг працював в лабораторії Bell, використовуючи електромеханічний комп'ютер на перфокартках. Коли в машині виявлялися помилки, вона зупинялася, і операторам доводилося їх виправляти. Хеммінг був розчарований необхідністю перезавантажити свої програми заново через виявлені помилки. Він казав: "Чорт візьми, якщо машина може виявити помилку, чому вона не може визначити положення помилки і виправити її?" Наступні кілька років він розробляв код Хеммінга, щоб комп'ютер міг робити саме це.

У кодах Хеммінга додаткові біти, які називаються бітами парності або контрольними бітами, додаються до інформації для забезпечення надлишковості. Всі позиції бітів, які є ступенями двійки, є бітами парності: 1, 2, 4, 8 тощо. Інші позиції бітів призначені для інформації. В таблиці під цим абзацом біти парності виділені зеленим кольором. Кожен біт парності розраховується з всіх бітів, у яких операція побітового І (AND) позиції біта парності і позиції біта даних не є нульовим, ці позиції для різних бітів парності позначені червоним X, дивись нижче. Якщо ми

хочемо використовувати біт даних, нам потрібні біти парності, які його покривають. Тобто щоб мати можливість використовувати біт даних d9, нам буде потрібен біт парності p8 та всі попередні біти парності. Отже, ця таблиця підказує нам, скільки бітів парності нам потрібно для певної кількості бітів. Цей шаблон при необхідності більшої кількості бітів даних можна продовжувати нескінченно.

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/hamming.svg

Коди Хеммінга є блоковими кодами, тому вони працюють по N бітів даних за раз. Таким чином в одному блоці, три біти парності дають можливість мати у блоках чотири біта даних. Цю схему кодування від помилок представляємо як Hamming(7,4), де перший аргумент — це загальна кількість переданих бітів (біти парності + біти даних), а другий аргумент — це кількість бітів даних.

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/hamming2.svg

Наступні три важливі властивості кодів Хеммінга (код, кодове слово - це блок для певних даних та вирохованих бітів парності, що стоять на певних місцях):

- Мінімальна кількість змін бітів, необхідних для переходу від одного кодового слова до іншого, становить три
- Код Хеммінга може виправляти однобітові помилки. (При помилці, розрахунок бітов парності вкаже на позицію де відбулась помилка. При цьому неважливо був це біт парності, чи даних)
- Код Хеммінга може виявляти, але не виправляти двобітові помилки

Алгоритмічно процес кодування можна здійснити за допомогою простого множення матриць, використовуючи так звану "матрицю генератора". У наведеному нижче прикладі вектор 1011 є даними (інформацією), які потрібно закодувати і яку ми хочемо надіслати отримувачу. 2D-матриця називається матрицею генератора, і визначає схему кодування. Результат множення дає кодове слово для передачі.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/hamming3.png

Мета "занурення" в коди Хеммінга полягає в тому, щоб дати уявлення про те, як працює кодування для корекції помилок. Блокові коди, як правило, відповідають цьому типу схеми кодування. Згорткові коди працюють інакше, але зараз заглиблюватися в це ми не будемо; вони часто використовують декодування по схемі Трелліса, яке зображено на рисунку нижче:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/trellis.svg

9.6. (Порівняння м'якого та жорсткого декодування) Soft vs Hard Decoding

Ви пам'ятаєте, що в приймачі демодуляція відбувається перед декодуванням. Демодулятор може надати нам найкраще припущення щодо того, який символ був переданий, або може надати "м'яке" значення. Наприклад, для BPSK, замість того, щоб надати 1 або 0, демодулятор може надати 0.3423 або -1.1234, що є "м'яким" значенням символу. Зазвичай декодування розроблене для використання м'яких або жорстких значень.

- **Декодування з м'яким рішенням (Soft decision decoding)** – використовує м'які значення
- **Декодування з жорстким рішенням (Hard decision decoding)** – використовує лише 1 та 0

М'яке декодування є більш надійним, оскільки ви використовуєте всю доступну інформацію, але м'яке декодування також значно складніше впровадити. Коди Хеммінга, про які ми говорили, використовують декодування з жорстким рішенням, тоді як згорткові коди в більшості використовують декодування з м'яким рішенням.

9.7. Границя Шеннона

Границя Шеннона або пропускна здатність по Шеннону - це неймовірний висновок з теорії, який дає нам розрахувати скільки бітів за секунду інформації без помилок ми можемо передати:

$$C = B \cdot \log_2 \left(1 + \frac{S}{N} \right)$$

- C – Ємність каналу [біти/сек]
- B – Ширина смуги каналу [Гц]
- S – Середня потужність отриманого сигналу [Вати]
- N – Середня потужність шуму [Вати]

Ця рівняння представляє максимальне теоритичне значення, яке може представити певна система модуляції сигналу при достатньо високому значенні с/ш (SNR) для безпомилкової роботи. Логічніше представити границю в бітах/сек/Гц, тобто в бітах на секунду на певну частину спектра:

$$\frac{C}{B} = \log_2 (1 + \text{SNR})$$

де сш (SNR) виражений в разях (не в децибелах). Проте, на графіку нижче, ми представимо сш (SNR) у дБ для зручності:

SVG-рисунок не вбудовано у цю TeX-конверсію.
Оригінальна прив'язка: ../_images/shannon_limit.svg

Якщо ви побачитедесь графік границі Шеннона, що виглядає трохи інакше, то, ймовірно, по осі x відкладається "енергія на біт" або E_b/N_0 , що є просто альтернативною формою значенню сш (SNR).

Для розуміння, коли сш (SNR) досить високий (наприклад, 10 дБ або має більше значення), границю Шеннона може бути наближено приведено як $\log_2 (\text{SNR})$, що приблизно дорівнює $\text{SNR}_{\text{dB}}/3$ ([пояснено тут](#)). Наприклад, при значенні сш (SNR) 24 дБ ви отримуєте близько 8 біт/сек/Гц, тож якщо у вас є смуга в 1 МГц для використання, це відповідає швидкості передачі 8 Мбіт/с. Можна подумати, що це лише теоретична границя, але у сучасному зв'язку реальні значення досить близькі до цієї границі, тому як мінімум це дає вам приблизну оцінку пропускної здатності. Завжди можете поділити це число навпіл, щоб врахувати накладні витрати на пакети/кадри і або неідеальність МКС (MCS).

Максимальна пропускна здатність WiFi 802.11n в діапазоні 2,4 ГГц (яка використовує канали шириною 20 МГц) за специфікацію становить 300 Мбіт/с. Звичайно, ви можете сидіти поруч з маршрутизатором і отримати дуже високе значення

сш (SNR), і може дорівнювати 60 дБ, але для отримання надійних/реальних значень максимальної пропускної здатності для певної МКС (MCS) (згадайте криву залежності пропускної здатності від значення с/ш у вигляді сходинки, наведену вище) мало ймовірно, що вимагатиметься таке велике значення с/ш (SNR), що дорівнює 60 дБ. Можете навіть переглянути список МКС (MCS) для 802.11n <https://en.wikipedia.org/wiki/IEEE_802.11n-2009#Data_rates>. 802.11n підтримує до 64-QAM, і в поєднанні з кодуванням каналу вимагає с/ш (SNR) близько 25 дБ за цією таблицею <<https://d2cpnw0u24fjm4.cloudfront.net/wp-content/uploads/802.11n-and-802.11ac-MCS-SNR-and-RSSI.pdf>>. Це означає, що навіть при с/ш (SNR) 60 дБ ваш WiFi все ще буде використовувати 64-QAM. Таким чином, при 25 дБ границя Шеннона приблизно становить 8,3 біт/с/Гц, що при 20 МГц спектру дорівнює 166 Мбіт/с. Однак, якщо врахувати МІМО, яке ми розглянемо в майбутньому розділі, ви можете отримати чотири таких паралельних потоки, що дасть 664 Мбіт/с. Якщо це число поділити навпіл, ви отримаєте значення, дуже близьке до заявленої максимальної швидкості 300 Мбіт/с для WiFi 802.11n в діапазоні 2,4 ГГц.

Доведення границі Шеннона, досить складне; воно включає наступну математику:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/shannon_limit_proof.png

Додаткову інформацію дивись [тут](#).

9.8. Найсучасніші коди

Наразі найкращі схеми кодування каналів:

1. Турбо-коди, що використовуються в 3G, 4G, космічних апаратах NASA.
2. LDPC-коди, що використовуються в DVB-S2, WiMAX, IEEE 802.11n.

Обидва ці коди наближаються до границі Шеннона (тобто майже досягають її при певних значеннях с/ш (SNR)). Коди Хеммінга та інші простіші коди не наближаються до границі Шеннона. З точки зору нових досліджень, залишилося небагато можливостей для покращення самих кодів. Поточні дослідження більше зосереджені на підвищенні обчислювальної ефективності декодування та адаптації до зворотного зв'язку по якості каналу.

Коди з низькою щільністю перевірки парності (LDPC) є класом високоефективних лінійних блокових кодів. Вони були вперше представлені Робертом Г. Галлагером у його докторській дисертації в 1960 році в MIT. Через обчислювальну складність їх реалізації, ці коди ігнорувались до 1990-х років! На момент написання цього тексту (2020) винахідник цих кодів ще живий, йому 89 років, і він отримав багато нагород за свою роботу (через десятиліття після того, як він її винайшов). LDPC не патентовані і безкоштовні для використання (на відміну від турбо-кодів), що пояснює, чому вони використовувалися в багатьох відкритих протоколах.

Турбо-коди базуються на згорткових кодах. Це клас кодів, який поєднує два або більше простіших згорткових кодів та перемежувач. Основна патентна заявка на турбо-коди була подана 23 квітня 1991 року. Винахідники - французи, тому коли Qualcomm хотіла використовувати турбо-коди в CDMA для 3G, їм довелося створити платіжну патентну ліцензійну угоду з France Telecom. Основний патент закінчився 29 серпня 2013 року.

Розділ 10

IQ файли та SigMF

У всіх наших попередніх прикладах на Python ми зберігали сигнали у вигляді 1D NumPy масивів типу "complex float". У цій главі ми дізнаємося, як зберігати сигнали у файлі, а потім зчитувати їх назад у Python, а також познайомимося зі стандартом SigMF. Зберігання даних сигналів у файлі є надзвичайно корисним: ви можете записати сигнал у файл, щоб вручну проаналізувати його в автономному режимі, поділитися ним з колегою або створити цілий набір даних.

10.1. Двійкові файли

Нагадаємо, що цифровий сигнал у базовій смузі частот - це послідовність комплексних чисел.

Приклад: $[0.123 + j0.512, 0.0312 + j0.4123, 0.1423 + j0.06512, \dots]$.

Ці числа відповідають $[I+jQ, I+jQ, I+jQ, I+jQ, I+jQ, I+jQ, I+jQ, I+jQ, I+jQ, \dots]$.

Коли ми хочемо зберегти комплексні числа у файл, ми зберігаємо їх у форматі IQIQIQIQIQIQIQIQIQIQIQ. Тобто, ми зберігаємо купу цілих чисел або чисел з плаваючою комою підряд, а коли ми їх зчитуємо, ми повинні розділити їх назад на $[I+jQ, I+jQ, \dots]$.

Хоча комплексні числа можна зберігати в текстовому файлі або csv-файлі, ми вважаємо за краще зберігати їх у так званому "двійковому файлі", щоб заощадити місце. При високій частоті дискретизації ваші записи сигналів можуть легко займати кілька гігабайт, і ми хочемо бути максимально ефективними в плані використання пам'яті. Якщо ви коли-небудь відкривали файл у текстовому редакторі і він виглядав незрозуміло, як на скріншоті нижче, ймовірно, він був бінарним. Бінарні файли містять серію байтів, і вам доведеться самотійно відстежувати формат. Двійкові файли є найефективнішим способом зберігання даних, якщо припустити, що було виконано все можливе стиснення. Оскільки наші сигнали зазвичай виглядають як випадкова послідовність цілих чисел або чисел з плаваючою комою, ми зазвичай не намагаємося стискати дані. Двійкові файли використовуються для багатьох інших речей, наприклад, для компіляції програм (так званих "бінарників"). Коли вони використовуються для збереження сигналів, ми називаємо їх двійковими "IQ-файлами", використовуючи розширення .iq.

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/binary_file.png

У Python за замовчуванням комплексним типом є `np.complex128`, який використовує два 64-бітних числа з плаваючою комою на семпл. Але в DSP/SDR ми, як правило, використовуємо 16-бітні цілі числа або 32-бітні числа з плаваючою комою, тому що АЦП на наших SDR не мають **такої** точності, щоб гарантувати 64-бітні числа з плаваючою комою. Насправді більшість SDR мають 12-бітні АЦП,

тому ми можемо мінімізувати використання сховища, зберігаючи як 16-бітні цілі числа (`np.int16` у Python), що означає, що кожен IQ-семпл займатиме 4 байти, і наш RF-запис створюватиме файл розміром у байтах, що дорівнює частоті дискретизації, помноженій на 4, відомий як "правило 4x від Сіна". У наведених нижче прикладах Python ми будемо використовувати **np.complex64**, який використовує два 32-бітних числа з плаваючою комою, оскільки Python не має власного комплексного цілочисельного типу (це не заважає нам зберігати IQ як цілі числа у файлі, як ви побачите). Коли ви просто обробляєте сигнал у Python, це не має значення, але коли ви збираєтеся зберегти 1d-масив у файл, ви хочете спочатку переконатися, що це масив `np.complex64` (або `np.int16` з інтерлівом IQ).

10.2. Приклади на Python

У Python, зокрема у `numpy`, ми використовуємо функцію `tofile()` для збереження масиву `numpy` у файл. Ось короткий приклад створення простого QPSK-сигналу з шумом і збереження його у файлі в тому ж каталозі, звідки ми запускали наш скрипт:

```
import numpy as np
import matplotlib.pyplot as plt

num_symbols = 10000

# Масив x_symbols міститиме комплексні числа, що представляють символи QPSK. Кожен символ
#   ↳ буде комплексним числом
# з модулем 1 і фазовим кутом, що відповідає одній з чотирьох точок сузір'я QPSK (45,
#   ↳ 135, 225 або 315 градусів)
x_int = np.random.randint(0, 4, num_symbols) # від 0 до 3
x_degrees = x_int*360/4.0 + 45 # 45, 135, 225, 315 градусів
x_radians = x_degrees*np.pi/180.0 # sin() та cos() приймають радіани
x_symbols = np.cos(x_radians) + 1j*np.sin(x_radians) # це створює наші комплексні символи
#   ↳ QPSK
n = (np.random.randn(num_symbols) + 1j*np.random.randn(num_symbols))/np.sqrt(2) # AWGN з
#   ↳ одиничною потужністю
r = x_symbols + n * np.sqrt(0.01) # потужність шуму 0.01
print(r)
plt.plot(np.real(r), np.imag(r), '.')
plt.grid(True)
plt.show()

# Тепер зберігаємо у IQ-файл
print(type(r[0])) # Перевіряємо тип даних. Упс, 128, а не 64!
r = r.astype(np.complex64) # Переводимо в 64
print(type(r[0])) # Переконатись, що це 64
r.tofile('qpsk_in_noise.iq') # Зберегти у файл
```

Тепер подивіться на деталі створеного файлу і перевірте, скільки у ньому байт. Він має бути `num_symbols * 8`, тому що ми використовували `np.complex64`, який має 8 байт на семпл, 4 байти на число з плаваючою комою (2 числа з плаваючою комою на семпл).

Використовуючи новий скрипт Python, ми можемо прочитати цей файл за допомогою `np.fromfile()`, наприклад, так:

```
import numpy as np
import matplotlib.pyplot as plt
```

```

samples = np.fromfile('qpsk_in_noise.iq', np.complex64) # Читаємо у файл. Треба вказати,
↳ у якому він форматі
print(samples)

# Побудуємо сузір'я, щоб переконатися, що воно виглядає правильно
plt.plot(np.real(samples), np.imag(samples), '.')
plt.grid(True)
plt.show()

```

Велика помилка - забути вказати `np.fromfile()` формат файлу. Двійкові файли не містять жодної інформації про свій формат. За замовчуванням `np.fromfile()` припускає, що він читає у форматі масиву `float64`.

Більшість інших мов мають методи для читання у двійкових файлах, наприклад, у MATLAB ви можете використовувати `fread()`. Для візуального аналізу RF-файлу дивіться розділ нижче.

Якщо ви коли-небудь матимете справу з `int16` (так званими короткими `int`) або будь-яким іншим типом даних, для якого `numpy` не має комплексного еквівалента, ви будете змушені читати приклади як справжні, навіть якщо вони насправді є комплексними. Хитрість полягає у тому, щоб прочитати їх як дійсні, але потім перетворити їх назад у формат IQIQIQ... самостійно, кілька різних способів зробити це показано нижче:

```

samples = np.fromfile('iq_samples_as_int16.iq',
↳ np.int16).astype(np.float32).view(np.complex64)

або

samples = np.fromfile('iq_samples_as_int16.iq', np.int16)
samples /= 32768 # конвертуємо в -1 до +1 (необов'язково)
samples = samples[:,2] + 1j*samples[1:,2] # конвертувати в IQIQIQ...

```

10.3. Перехід з MATLAB

Якщо ви намагаєтеся перейти з MATLAB на Python, ви можете поцікавитися, як зберегти змінні MATLAB і файли `.mat` як двійкові IQ-файли. Спочатку нам потрібно обрати тип формату. Наприклад, якщо наші семпли є цілими числами між -127 і +127, ми можемо використати 8-бітні цілі числа. У такому випадку ми можемо скористатися наступним кодом MATLAB, щоб зберегти семпли у двійковий IQ-файл:

```

% припустимо, що наші IQ-семпли містяться у змінній samples
disp(samples(1:20))
filename = 'samples.iq'
fwrite(fopen(filename,'w'), reshape([real(samples);imag(samples)],[],1), 'int8')

```

Ви можете переглянути всі допустимі типи форматів для `fwrite()` в [документації MATLAB](#). Проте найкраще дотримуватися форматів `'int8'`, `'int16'` або `'float32'`.

З боку Python ви можете завантажити цей файл за допомогою:

```

samples = np.fromfile('samples.iq', np.int8)
samples = samples[:,2] + 1j*samples[1:,2]
print(samples[0:20]) # переконайтеся, що перші 20 семплів збігаються з MATLAB

```

Для `'float32'`, збереженого в MATLAB, ви можете використати `np.complex64` у Python, що відповідає інтерлівним `float32`, і тоді можна пропустити частину `samples[:,2] + 1j*samples[1:,2]`, тому що `numpy` автоматично інтерпретує інтерлівні числа з плаваючою комою як комплексні.

10.4. Візуальний аналіз радіочастотного файлу

Хоча ми навчилися створювати власні графіки спектрограм у розділі `freq-domain-chapter`, ніщо не зрівняється з використанням вже створеного програмного забезпечення. Коли справа доходить до аналізу радіочастотних записів без необхідності нічого встановлювати, найкращим сайтом є [IQEngine](#), який є цілим інструментарієм для аналізу, обробки та обміну радіочастотними записами.

Для тих, кому потрібен десктопний додаток, є також [inspectrum](#). Inspectrum - це досить простий, але потужний графічний інструмент для візуального сканування радіочастотного файлу з тонким контролем діапазону кольорової карти і розміру БПФ (масштабу). Ви можете утримувати клавішу `Alt` і використовувати колесо прокрутки для переміщення в часі. Програма має додаткові курсори для вимірювання дельта-часу між двома сплесками енергії, а також можливість експортувати фрагмент радіочастотного файлу до нового файлу. Для встановлення на платформах на основі Debian, таких як Ubuntu, скористайтеся наступними командами:

```
sudo apt-get install qt5-default libfftw3-dev cmake pkg-config libliquid-dev
git clone https://github.com/miek/inspectrum.git
cd inspectrum
mkdir build
cd build
cmake ..
зробити
sudo make install
inspectrum
```

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: `../_images/inspectrum.jpg`

10.5. Максимальні значення та насиченість

При отриманні семплів з SDR важливо знати максимальне значення семплу. Багато SDR виводять семпли як числа з плаваючою комою з максимальним значенням 1.0 і мінімальним -1.0. Інші SDR надають вам вибірки як цілі числа, зазвичай 16-розрядні, в цьому випадку максимальне і мінімальне значення буде +32767 і -32768 (якщо не вказано інше), і ви можете розділити на 32,768, щоб перетворити їх у значення з плаваючою комою від -1.0 до 1.0. Причина, по якій необхідно знати максимальне значення для вашого SDR, полягає в насиченні: при отриманні дуже гучного сигналу (або якщо коефіцієнт підсилення встановлено занадто високим), приймач "насититься" і обріже високі значення до того, яким би не було максимальне значення дискретизації. АЦП на наших SDR мають обмежену кількість бітів. При створенні SDR-додатків доцільно завжди перевіряти насичення, і коли це відбувається, ви повинні якось позначити це.

Сигнал, який є насиченим, буде виглядати нестабільним у часовій області, як це показано нижче:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: `../_images/saturated_time.png`

Через різкі зміни в часовій області, спричинені усіченням, частотна область може виглядати розмазаною. Іншими словами, частотна область буде включати помилкові особливості; особливості, які є результатом насичення і насправді не є частиною сигналу, що може збити людей з пантелику при аналізі сигналу.

10.6. SigMF та анотування IQ файлів

Оскільки сам IQ-файл не має жодних метаданих, пов'язаних з ним, зазвичай створюють 2-й файл, що містить інформацію про сигнал, з тим самим іменем, але з розширенням .txt або іншим. Він повинен містити, як мінімум, частоту дискретизації, яка використовувалася для збору сигналу, і частоту, на яку було налаштовано SDR. Після аналізу сигналу файл метаданих може містити інформацію про діапазони дискретизації цікавих особливостей, таких як сплески енергії. Індекс вибірки - це просто ціле число, яке починається з 0 і збільшується з кожною складною вибіркою. Якби ви знали, що є енергія від зразка 492342 до 528492, то ви могли б прочитати файл і витягнути цю частину масиву: `samples[492342:528493]`.

На щастя, зараз існує відкритий стандарт, який визначає формат метаданих для опису записів сигналів, відомий як [SigMF](#). Використовуючи відкритий стандарт, такий як SigMF, різні сторони можуть легше обмінюватися записами радіосигналів і використовувати різні інструменти для роботи з тими самими наборами даних, такі як [IQEngine](#). Це також запобігає "бітротству" наборів радіочастотних даних, коли деталі захоплення втрачаються з часом через те, що деталі запису не співпадають із самим записом.

Найпростіший (і мінімальний) спосіб використання стандарту SigMF для опису створеного вами бінарного IQ-файлу - перейменувати файл .iq на .sigmf-data і створити новий файл з тим самим ім'ям, але з розширенням .sigmf-meta, і переконатися, що поле типу даних у метафайлі відповідає бінарному формату вашого файлу даних. Цей метафайл є звичайним текстовим файлом, заповненим json, тому ви можете просто відкрити його за допомогою текстового редактора і заповнити вручну (пізніше ми обговоримо, як зробити це програмно). Ось приклад .sigmf-meta файлу, який ви можете використовувати як шаблон:

```
{
  "global": {
    "core:datatype": "cf32_le",
    "core:sample_rate": 1000000,
    "core:hw": "PlutoSDR з 915 МГц штирьовою антеною",
    "core:author": "Art Vandelay",
    "core:version": "1.0.0"
  },
  "captures": [
    {
      "core:sample_start": 0,
      "core:frequency": 915000000
    }
  ],
  "annotations": []
}
```

Зверніть увагу, що `core:cf32_le` вказує на те, що ваші .sigmf-дані мають тип IQIQIQ... з 32-бітними числами з плаваючою комою, тобто `pr.complex64`, як ми використовували раніше. Зверніться до специфікацій інших доступних типів даних, наприклад, якщо ви використовуєте дійсні дані замість комплексних, або використовуєте 16-розрядні цілі числа замість плаваючих для економії місця.

Окрім типу даних, найважливішими рядками для заповнення є `core:sample_rate` та `core:frequency`. Належною практикою є також введення інформації про апаратне забезпечення (`core:hw`), яке було використано для захоплення запису, наприклад, тип SDR та антени, а також опис того, що відомо про сигнал(и) у записі у `core:description`. Поле `core:version` - це просто версія стандарту SigMF, яка використовувалася на момент створення файлу метаданих.

Якщо ви записуєте радіосигнал з Python, наприклад, використовуючи API Python для SDR, ви можете уникнути необхідності створювати ці файли метаданих вручну, скориставшись пакетом SigMF Python. Його можна встановити на ОС на базі Ubuntu/Debian наступним чином:

```
pip install sigmf
```

Нижче наведено код Python для написання файлу .sigmf-meta для прикладу на початку цієї глави, куди ми зберегли qpsk_in_noise.iq:

```
import datetime as dt

import numpy as np
import sigmf
from sigmf import SigMFFile

# <код з прикладу

# r.tofile('qpsk_in_noise.iq')
r.tofile('qpsk_in_noise.sigmf-data') # замінити рядок вище на цей

# створюємо метадані
meta = SigMFFile(
    data_file='qpsk_in_noise.sigmf-data', # extension is optional
    global_info = {
        SigMFFile.DATATYPE_KEY: 'cf32_le',
        SigMFFile.SAMPLE_RATE_KEY: 8000000,
        SigMFFile.AUTHOR_KEY: 'Your name and/or email',
        SigMFFile.DESCRPTION_KEY: 'Simulation of qpsk with noise',
        SigMFFile.VERSION_KEY: sigmf.__version__,
    }
)

# створити ключ захоплення з часовим індексом 0
meta.add_capture(0, metadata={
    SigMFFile.FREQUENCY_KEY: 915000000,
    SigMFFile.DATETIME_KEY: dt.datetime.now(dt.timezone.utc).isoformat(),
})

# перевірка на помилки та запис на диск
meta.validate()
meta.tofile('qpsk_in_noise.sigmf-meta') # розширення не обов'язкове
```

Просто замінить 8000000 та 915000000 на змінні, які ви використовували для зберігання частоти дискретизації та центральної частоти відповідно.

Щоб прочитати запис у форматі SigMF у Python, скористайтеся наступним кодом. У цьому прикладі два SigMF-файли слід назвати qpsk_in_noise.sigmf-meta і qpsk_in_noise.sigmf-data.

```
from sigmf import SigMFFile, sigmffile

# Завантажити набір даних
filename = 'qpsk_in_noise'
signal = sigmffile.fromfile(filename)
samples = signal.read_samples().view(np.complex64).flatten()
print(samples[0:10]) # виводимо перші 10 зразків

# отримуємо метадані та всі анотації
sample_rate = signal.get_global_field(SigMFFile.SAMPLE_RATE_KEY)
sample_count = signal.sample_count
signal_duration = sample_count / sample_rate
```


За більш детальною інформацією зверніться до [документації SigMF Python](#).

Невеликий бонус для тих, хто дочитав до цього місця: логотип SigMF фактично зберігається як сам запис SigMF, і коли сигнал будується у вигляді сузір'я (IQ-діаграма) у часі, він створює наступну анімацію:

Рисунок не вбудовано у цю машинно-читабельну TeX-конверсію.
Оригінальна прив'язка: ../_images/sigmf_logo.gif

Код на Python, який використовується для зчитування файлу логотипу (розташованого [тут](#)) і створення анімованого gif-файлу, показано нижче, для тих, кому цікаво:

```
from pathlib import Path
from tempfile import TemporaryDirectory

import numpy as np
import matplotlib.pyplot as plt
import imageio.v3 as iio
from sigmf import SigMFFile, sigmf

# Завантажуємо набір даних
filename = 'sigmf_logo' # вважаємо, що він знаходиться у тому ж каталозі, що і цей скрипт
signal = sigmf.fromfile(filename)
samples = signal.read_samples().view(np.complex64).flatten()

# Додаємо нулі в кінці, щоб було зрозуміло, коли анімація повторюється
samples = np.concatenate((samples, np.zeros(50000)))

sample_count = len(samples)
samples_per_frame = 5000
num_frames = int(sample_count/samples_per_frame)

with TemporaryDirectory() as temp_dir:
    filenames = []
    output_dir = Path(temp_dir)
    for i in range(num_frames):
        print(f"frame {i} out of {num_frames}")
        # Побудувати графік кадру
        fig, ax = plt.subplots(figsize=(5, 5))
        samples_frame = samples[i*samples_per_frame:(i+1)*samples_per_frame]
        ax.plot(np.real(samples_frame), np.imag(samples_frame), color="cyan", marker=".",
                linestyle="None", markersize=1)
        ax.axis([-0.35,0.35,-0.35,0.35]) # зберігаємо вісь постійною
        ax.set_facecolor('black') # колір фону

        # Зберегти графік у файл
        filename = output_dir.joinpath(f"simgf_logo_{i}.png")
        fig.savefig(filename, bbox_inches='tight')
        plt.close()
        filenames.append(filename)

    # Створюємо анімований gif
    images = [iio.imread(f) for f in filenames]
    iio.imwrite('sigmf_logo.gif', images, fps=20)
```

10.7. Колекція SigMF для масивних записів

Якщо у вас є фазована антена, цифрова решітка MIMO, датчики TDOA або будь-яка інша ситуація, коли ви записуєте кілька каналів синхронізованих радіоданих,

ви, мабуть, замислюєтеся, як зберігати сирі IQ кількох потоків у файлі за допомогою SigMF. Система **Колекцій** SigMF була розроблена саме для таких випадків; колекція - це просто група записів SigMF (кожен складається з одного метафайлу та одного файлу даних), об'єднаних разом за допомогою верхнього рівня JSON-файлу з розширенням `.sigmf-collection`. Цей JSON-файл досить простий; він повинен містити версію SigMF, необов'язковий опис, а також список "потоків", що насправді є базовими назвами кожного запису SigMF у колекції. Ось приклад файлу `.sigmf-collection`:

```
{
  "collection": {
    "core:version": "1.2.0",
    "core:description": "a 4-element phased array recording",
    "core:streams": [
      {
        "name": "channel-0"
      },
      {
        "name": "channel-1"
      },
      {
        "name": "channel-2"
      },
      {
        "name": "channel-3"
      }
    ]
  }
}
```

Назви записів необов'язково мають бути `channel-0`, `channel-1`, ..., вони можуть бути будь-якими, лише б були унікальними і щоб кожна назва відповідала одному файлу даних і одному метафайлу. У наведеному вище прикладі цей файл `.sigmf-collection`, який ми могли б назвати, наприклад, `4_element_recording.sigmf-collection`, повинен бути в тому самому каталозі, що й файли метаданих і даних, тобто в тому ж каталозі ми матимемо:

- `4_element_recording.sigmf-collection`
- `channel-0.sigmf-meta`
- `channel-0.sigmf-data`
- `channel-1.sigmf-meta`
- `channel-1.sigmf-data`
- `channel-2.sigmf-meta`
- `channel-2.sigmf-data`
- `channel-3.sigmf-meta`
- `channel-3.sigmf-data`

Можливо, ви подумаєте, що це призведе до величезної кількості файлів, наприклад, масив із 16 елементів створить 33 файли! Саме з цієї причини SigMF запровадив систему **Архівів**, яка насправді є терміном SigMF для упакування набору файлів у tar-архів. Файл архіву SigMF використовує розширення `.sigmf`, а не `.tar`! Багато людей вважають, що файли `.tar` стиснені, але це не так; це просто спосіб об'єднати файли разом (це фактично конкатенація файлів без стиснення). Можливо, ви бачили файл `.tar.gz`; це tar-архів, який було стиснено за допомогою `gzip`. Для наших архівів SigMF ми не будемо їх стискати, оскільки файли даних уже є двійковими і не сильно стискаються, особливо якщо використовувалося автоматичне керування підсиленням. Якщо ви хочете створити архів SigMF у Python, ви можете запакувати всі файли в каталозі разом таким чином:

```
import tarfile
import os

target_dir = '/mnt/c/Users/marclightman/Downloads/exampletar/' # SigMF файли тут
with tarfile.open(os.path.join(target_dir, '4_element_recording.sigmf'), 'x') as tar: # x
    ↪ означає створити, але помилитись, якщо вже існує
    for file in os.listdir(target_dir):
        tar.add(os.path.join(target_dir, file), arcname=file) # arcname не дозволяє
        ↪ включати повний шлях у tar
```

І все! Спробуйте (тимчасово) перейменувати .sigmf на .tar і перегляньте файли у файловому менеджері. Щоб відкривати будь-які файли безпосередньо (без ручного розпакування tar) у Python, ви можете використати:

```
import tarfile
import json

collection_file =
    ↪ '/mnt/c/Users/marclightman/Downloads/exampletar/4_element_recording.sigmf'
tar_obj = tarfile.open(collection_file)
print(tar_obj.getnames()) # список рядків із назвами всіх файлів у tar
channel_0_meta = tar_obj.extractfile('channel-0.sigmf-meta').read() # читаємо один з
    ↪ метафайлів як приклад
channel_0_dict = json.loads(channel_0_meta) # перетворюємо на словник Python
print(channel_0_dict)
```

Для зчитування IQ-семплів безпосередньо з tar замість `np.fromfile()` ми використаємо `np.frombuffer()`:

```
import tarfile
import numpy as np

collection_file =
    ↪ '/mnt/c/Users/marclightman/Downloads/exampletar/4_element_recording.sigmf'
tar_obj = tarfile.open(collection_file)
channel_0_data_f = tar_obj.extractfile('channel-0.sigmf-data').read() # тип bytes
samples = np.frombuffer(channel_0_data_f, dtype=np.int16)
samples = samples[:,2] + 1j*samples[1:,2] # конвертувати в IQIQIQ...
samples /= 32768 # конвертувати в -1 до +1
print(samples[0:10])
```

Якщо ви хочете перейти до іншої частини файлу, використовуйте `tar_obj.extractfile('channel-0.sigmf-data').seek(offset)`. Потім, щоб прочитати конкретну кількість байтів, скористайтеся `.read(num_bytes)`. Переконайтеся, що кількість байтів є кратною вашому типу даних!

Підсумуємо: при створенні нового архіву колекції SigMF слід виконати такі кроки:

1. Створити файл .sigmf-meta та .sigmf-data для кожного каналу
2. Створити файл .sigmf-collection
3. Упакувати всі файли разом у файл .sigmf
4. (За бажанням) Поділитися файлом .sigmf з іншими!

Додаток А

QA-позначки після конверсії

- Перевірити всі посилання виду `chapter-label`: у RST вони часто були внутрішніми Sphinx-посиланнями, а у TeX потребують ручного `\ref` або простого текстового посилання.
- Перевірити терміни: *семплювання/дискретизація, сузір'я/констеляція, формування імпульсу, синхронізація несучої, узгоджена фільтрація*.
- Рисунки не вбудовано; це свідомо для зменшення залежностей і уникнення `missing-file failure`.
- Кодові блоки зроблено переносимими через `fvextra`.