Core concepts of GNU Radio

Author: GNU Radio contributors

Translator: Yoon Kyong Lok(steveyoon@telechips.com)

Contents

Core Concepts of GNU Radio

이 문서는 GNU Radio의 매우 기초적인 입문 튜토리얼 입니다. 여러분은 다른 것을 보기 전에 이 문서를 확실히 읽으셔야 합니다. 여기에서 다루는 모든 걸 전에 들었을지라도 적어도 놓치는 것이 있는지 한번 훑어 보셔야만 합니다.

Flow graphs – and what they're made of

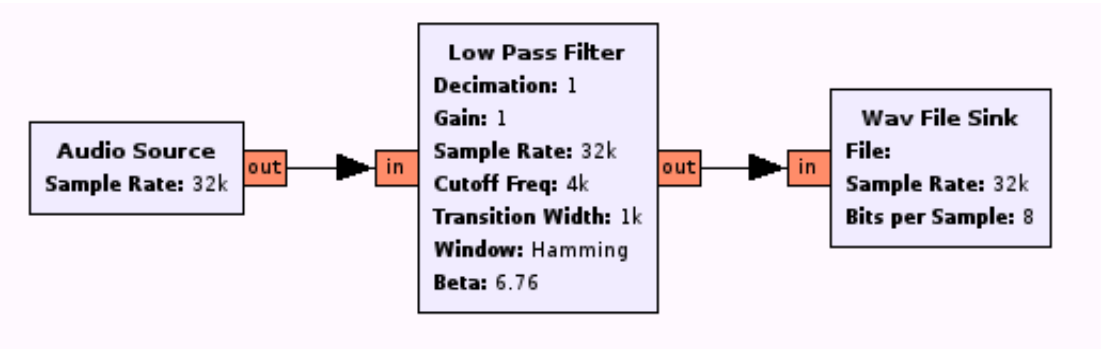
어딘가로 가기 전에, 먼저 우리는 GNU Radio에 대해서 가장 기초적인 개념(흐름 그래프와 블록들)을 이해해야 합니다.

흐름 그래프는 데이터 흐름들을 통과하는 그래프(graph theory에 따르는)입니다. 많은 GNU Radio 응용 프로그램들은 흐름 그래프 말고 다른 것은 포함하고 있지 않습니다. 그래프와 같은 노드들을 *블록들(blocks)*이라고 부르고 데이터는 그 끄트머리$_{\text{edges}}$를 따라 흐릅니다.

어떤 실제 신호 처리 기능도 블록으로 완성됩니다. 이상적으로, 모든 블록은 정확히 하나의 일을 합니다 - 이 방식으로 GNU Radio의 모듈성과 융통성이 유지됩니다. 블록들은 대게 C++(아마 파이썬으로도)로 작성됩니다. 그리고 새로운 블록을 만드는 것이 아주 어렵지는 않습니다.

이 조금 산만한 주제를 명확히 하기 위해, 예제 하나를 시작하겠습니다. (이 모든 예제는 GNU Radio의 GUI인 GNU Radio Companion (GRC)로 만들어졌습니다.)

여기, 세 개의 블록들(사각형들)이 있습니다. 이 예제에서 데이터는 왼쪽에서 오른쪽으로 흘러 갑니다. 그것은 오디오 쏘스에서 시작하여 로우 패쓰 필터를 통과하고 하드 디스크 파일로 쓰여지면서 끝나는 것을 의미하고 있습니다.



블록들은 *포트*$_{ports}$로 연결됩니다. 첫번째 블록은 입력 포트가 없고 샘플들을 만들어 냅니다. 오직 출력 포트만 가지고 있는 블록을 *쏘스*$_{source}$라고 합니다. 아날로그 관습으로 출력이 없는 마지막 블록은 *싱크*$_{sink}$라고 합니다.

때때로 혼란스러운게, 사용자 관점에서 오디오 블록은 사운드 카드에서 오디오 샘플을 얻어오므로 바로 처리의 한 부분이라는 겁니다. 하지만 우리가 싱크와 쏘스를 말할 때, 항상 흐름 그래프의 관점에서 이야기 합니다. [1]

그래서 여기에서 무슨 일이 벌어질까요. 오디오 쏘스 블록은 사운드 카드 드라이버와 연결되고 오디오 샘플들을 출력합니다. 이 샘플들은 로우 패쓰 필터에 연결되고 로우 패쓰 필터는 샘플들을 더 처리합니다. 마지막으로 샘플들은 WAV 파일로 쓰여지는 블록으로 전달됩니다.

Items

일반적으로 우리는 블록의 출력으로 어떤 아이템이든 호출할 수 있습니다. 이전 예제에서 하나의 아이템은 오디오 드라이버에서 만들어진 하나의 샘플을 표현하는 부동소수점 값이었습니다. 그러나 아이템은 디지털 방식으로 표현될 수 있는 어떤 것이라도 될 수 있습니다. 샘플들의 가장 일반적인 데이터형은 실수 샘플들 (앞의 예처럼), 복소수 샘플들 (Software defined radio에서 가장 일반적인 데이터형), 정수 데이터형, 그리고 이들 스칼라 데이터형들의 벡터들이 있습니다.

이 이후의 개념을 이해하기 위해, FFT 분석을 생각해 보겠습니다. 우리가 신호를 파일로 저장하기 전에 FFT를 수행하려고 한다고 해보겠습니다. 물론 우리는 FFT를 계산하는 시점에 확실한 수의 샘플들이 필요합니다 그리고 FFT는 필터와는 다르게 샘플 단위로 동작하지 않습니다.

여기에 '스트림에서 벡터로(Stream to Vector)'라는 새로운 블록이 있습니다. 이것의 특별한 점은 입력 데이터형과 출력 데이터형이 다르다는 것 입니다. 이 블록은 1024 샘플들(즉, 1024개의 아이템)을 가져와서 그것들을 1024 샘플의 벡터 하나(하나의 아이템)로 출력합니다. 복소수 FFT 출력들은 복소수의 2승으로 변환되고 그것은 실수 데이터형입니다. (우리가 포트마다 서로 다른 데이터형을 지시하기 위해 어떻게 서로 다른 색을 사용했는지 보십시오.)

그러니까 아이템은 샘플, 비트 묶음, 필터 계수 세트, 또는 무엇이든 될 수 있다는 것을 기억하십시오.

Summary

지금까지 배운 것 중 여러분이 반드시 알아야 하는 것은 다음과 같습니다.

[1] 흐름 그래프는 싱크, 쏘스, 처리 블록으로 이루어지는 데 첫번째 예제에서 쏘스가 사운드 카드에서 오디오를 입력 받아서 샘플을 출력하므로 처리 블록일 것 같지만, 흐름 그래프 관점에서는 사운드 카드가 없으므로 쏘스 블록으로 항상 얘기한다는 말입니다.

- GNU Radio의 모든 신호 처리는 흐름 그래프를 통해 이루어집니다.
- 흐름 그래프는 블록들로 구성됩니다. 블록은 필터링, 신호 추가, 변환, 디코딩, 하드웨어 접근 등과 같이 많은 것들 중 하나의 신호 처리 운용을 수행합니다.
- 데이터는 복소수나 실수 정수, 부동소수점 수나, 기본적으로 여러분이 정의할 수 있는 어떤 종류의 데이터 형이든 다양한 형식으로 블록들 사이에 통과됩니다.
- 모든 흐름 그래프는 적어도 하나의 싱크와 쏘스를 필요로 합니다.

So what does GNU Radio do?

첫번째, 여러분이 할 것은 흐름 그래프를 설계하고, 블록을 선택하고, 연결을 정의하고, 이 모든 것을 GNU Radio에 이야기 하는 것 입니다. GNU Radio는 여기에서 두 번 사용됩니다. 첫째로 많은 블록들을 여러분에게 제공해줍니다. 둘째로, 일단 흐름 그래프가 정의된다면, 하나씩 잇따라 블록들을 호출함으로써 그 흐름 그래프를 실행시키고 아이템이 한 블록에서 다른 블록으로 통과되는 것을 확인합니다.

Sampling rates

물론, 여러분이 이것을 읽고 있다는 것은 여러분은 이미 샘플레이트가 무엇인지 알고 있다는 것일 겁니다. (아니라면, 얼른 공부하고 오십시오.) 샘플레이트가 흐름그래프와 어떻게 관련되는지 한 번 보겠습니다. 첫번째 예제에서, 오디오 쏘스는 32ksps[2]의 고정된 샘플레이트를 가지고 있습니다. 필터가 샘플레이트를 바꾸지 않기 때문에 샘플레이트는 전체 흐름그래프에서 사용됩니다.

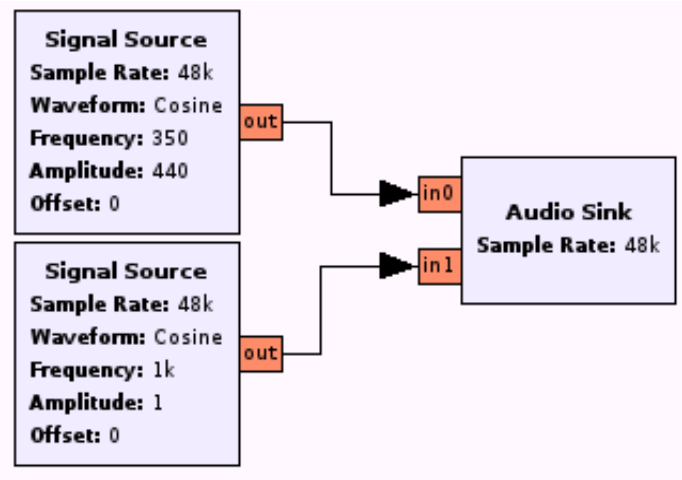
두번째 예제에서, 두번째 블록(stream to vector)은 매 1024 입력 아이템마다 하나의 아이템을 생산합니다. 그래서, 아이템을 생산하는 곳의 샘플레이트가 아이템을 소비하는 곳의 샘플레이트보다 1024배 작습니다. (실제로 생산하는 바이트가 소비되는 것과 같은 샘플레이트를 갖는다는 사실은 여기에서 무의미합니다.) 그와 같이 블록을 *학살자*$_{decimator}$라고 부릅니다. 음, 왜냐하면 그것이 아이템 샘플레이트를 학살하니까요.[3] 아이템을 받을 때보다 더 많이 출력하는 블록은 *말참견쟁이*$_{interpolator}$라고 부릅니다.[4] 만약에 데이터의 생산과 소비를 같은 샘플레이트로 한다면 그것을 *싱크*$_{sync}$ 블록이라고 부릅니다.

이제, 두번째 예제로 돌아가겠습니다. 언급했던것처럼, 이 예제는 흐름 그래프를 통틀어 다른 샘플레이트를 쓰고 있습니다. 하지만 기반 샘플레이트는 무엇일까요?

좋습니다. 시작합니다: 그와 같은 것은 없습니다. 샘플레이트를 고정하는 하드웨어 클럭이 있지 않는 한 샘플 레이트는 의미 없습니다. 그냥 상대적인 비율일 뿐입니다. 즉, 입력 대 출력 비율이 중요합니다. 여러분의 컴퓨터는 그것이 원하는 만큼 빠르게 샘플들을 다룰 수 있을 것 입니다. (이것이 여러분의 신호처리를 위해 CPU를 100% 할당해서 컴퓨터를 먹통으로 만들 수 있다는 것도 아셔야 합니다.)

여기에 또다른 예제가 있습니다.

[2] 오타가 아닙니다. sample per second로 보입니다.
[3] 번역하는 저도 소름이 끼쳤습니다.
[4] 번역하는 저도 소름이 끼쳤습니다.



무엇보다도, 여기에서 새로운 것은 싱크가 두 개의 입력을 갖는다는 것 입니다. 각 포트는 고정된 샘플레이트로 실행되는 사운드 카드의 채널 하나씩(left, right)을 다룹니다.

More on blocks (and atomicity)

이제 블록으로 돌아갑니다. GNU Radio의 가장 큰 부분은 GNU Radio에서 제공하는 엄청 많은 수의 블록들 입니다. 여러분이 GNU Radio를 이용하기 시작할 때, 블록과 블록을 연결할 것 입니다. 지금이든 나중이든, GNU Radio에서 제공하지 않는 블록이 필요할 수도 있습니다. 그러면 직접 만들 수 있어요. 그건 그렇게 어렵지 않습니다.

얼마나 많은 것들을 하나의 블록에 넣을 수 있을까요? 이상적으로, 블록은 원자성이 있다고 볼 수 있습니다. 모든 블록은 정확히 하나의 일을 수행합니다. 이 방식으로 GNU Radio의 모듈성과 융통성이 지켜지는 겁니다. 그러나 때때로 이게 지켜지지 않을 때도 있습니다. 어떤 블록들은 한 번에 많은 일을 수행합니다. 여러분은 아마도 퍼포먼스 대 모듈성의 트레이드 오프를 알아볼 수 있을 겁니다.

A note for Simulink users

시뮬링크 사용자 여러분, 프레임 기반 처리 대 샘플 기반 처리에 대해 알고 있는 모든 것을 무시하십시오.

이 장은 시뮬링크 사용자들을 위한 특별한 장입니다. 시뮬링크에서, 흐름 그래프는 프레임 기반이나 샘플 기반으로 설정할 수 있습니다. 그 둘의 차이점은 샘플 기반 모델에서 샘플은 블록과 블록 사이에 개별적으로 통과된다는 것 입니다. 따라서 스트림의 신호처리가 최대로 관리됩니다. 하지만 이것은 퍼포먼스를 잃게 합니다. 그래서 시뮬링크는 프레임 기반 처리를 도입했습니다.

이제 여러분이 시뮬링크에서 시작할 때 **이 모든 것을 잊어버리십시오**[5]. GNU Radio에서 그것은 안 통합니다. GNU Radio에는 오직 *아이템 기반*$_{item\text{-}based}$만 있습니다. 그리고 꽤 자주 아이템은 샘플을 말합니다. (그러나 벡터일 수도 있습니다.) 아이템의 크기는 입력 포트에서 여러분이 무엇을 가져오느냐의 논리적 설명입니다.

GNU Radio가 (비록 아이탬을 개별적으로 다룬다고 할지라도) 퍼포먼스 상실을 가질 수 없는 이유는 한 번에 가능한 많은 아이템을 처리하려고 하기 때문입니다. 의식적으로, 그것은 샘플 기반이면서 프레임 기반입니다.

GNU Radio의 불리한 접근법은 순환적인 흐름 그래프의 도입이 사소한 것이 아니라는 점입니다.[6]

Metadata

샘플의 스트림은 수신 시간, 중심 주파수, 샘플레이트 또는 노드 표시와 같은 프로토콜 특정의 정보 처럼 그 스트림에 연결된 파싱 가능한 메타 데이터가 있을 때 더 많이 재미있습니다.

[5] 샘플 기반 처리와 프레임 기반 처리에 대해서 잊어버리라는 것 입니다.

[6] 아마도 시뮬링크에서 GNU Radio를 사용하는 경우 GNU Radio의 흐름 그래프를 순환적인 콜로 정의하면 퍼포먼스에 문제가 생긴다는 말을 하려는 것이 아닌가 합니다.

GNU Radio에서 샘플 스트림에 메타 데이터를 추가하는 것은 **스트림 태그~stream tags~**라 불리는 메카니즘을 통해서 할 수 있습니다. 스트림 태그는 특정한 아이템 (예를 들어 샘플)에 연결되는 객체입니다. 이것은 어떤 데이터 형(벡터, 리스트, 딕셔너리$_{\text{dictionary}}$[7] 또는 사용자가 정의하는 무엇이든)이든 스칼라 값이 될 수 있습니다.

하드 디스크에 스트림을 저장할 때, 메타 데이터도 함께 저장될 수 있습니다. (메타 데이터 문서화 페이지를 보십시오)

Streams vs. Messages: Passing PDUs

발표된 모든 블록들은 지금까지 "무한의 스트림" 블록으로 운용되었습니다. 즉, 블록들은 단순히 블록의 입력이 공급되는 한 계속 동작합니다. 로우 패쓰 필터가 좋은 예 입니다. 모든 새 아이템이 새로운 샘플로 해석되고 출력은 항상 입력의 로우 패쓰 필터 된 버전입니다. 그것은 신호의 컨텐츠에 대해서 관심을 갖고 있지 않습니다. 노이즈이던, 데이터이던, 무엇이던 말이죠.

패킷(또는 PDUs~protocol data units~)을 다룰 때, 그와 같은 행동은 충분치 않습니다. PDU 경계를 식별할 방법이 반드시 있어야만 합니다. 즉, 어느 바이트가 패킷의 처음이고 패킷이 얼마나 긴지를 말합니다.

GNU Radio는 이것을 위해 두 가지 방법을 제공합니다: 메세지 전달과 태그된 스트림 블록입니다.

첫번째 것은 직접 PDU들을 비동기적으로 하나의 블록에서 다른 블록으로 직접 전달하는 방법입니다. MAC 계층 위에서, 이것은 아마도 바람직한 동작일 것 입니다. 블록은 한 PDU를 받는다면, 패킷 헤더를 추가하고, 전체 패킷을 (새 헤더를 포함하여) 다른 블록으로 새 PDU로서 전달합니다. (메시지 전달 매뉴얼 페이지를 보십시오.)

태그된 스트림 블록은 PDU 경계를 식별하기 위해 스트림 태그를 이용하는 보통의 스트리밍 블록입니다. (태그된 스트림 블록 매뉴얼 페이지를 보십시오.) 이것은 PDU에 대해서 아는 블록들과 그것을 무시하는 블록들을 섞는 것을 허용합니다. 또 메시지 전달 방식과 태그된 스트림 블록 방식 사이의 전환을 위한 블록들도 있습니다.[8]

[7] 파이썬에서 제공하는 자료형 중 하나입니다.
[8] 역자도 무슨 말인지 모릅니다. 직역을 했을 뿐 입니다. 메시지 전달 매뉴얼과 태그된 스트림 블록 매뉴얼을 보아야만 할 것 같습니다.