Tutorials for writing Python applications

Author: GNU Radio contributors

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

Contents

Introduction

GNU Radio 초보자 여러분 환영합니다. 이 튜토리얼을 읽으시는 여러분은 아마 GNU Radio가 어떻게 동작하는지, 그게 무엇이고 무엇을 할 수 있는지에 대해서 다소 기초적인 지식을 이미 알고 있을 것 입니다. 그리고 이제 여러분 스스로 환상적인 오픈 소스 디지털 신호처리의 세계에 들어가길 원할 겁니다. 이 튜토리얼은 GNU Radio 3.7 버전을 이용하여 어떻게 Python으로 GNU Radio application을 작성하는지에 대한 것 입니다. 프로그래밍이나, 소프트웨어 라디오, 신호처리에 대한 것이 아니라, 새로운 블록을 추가하거나 소스 트리에 코드를 추가하여 GNU Radio를 어떻게 확장할지에 대한 것입니다. 여러분이 저런 주제(프로그래밍, 소프트웨어 라디오나 신호처리)에 대해 백그라운드를 가지고 계시고 GNU Radio로 작업을 시작했다면 이것은 아마 여러분에게 올바른 튜토리얼일 것 입니다. 여러분이 소프트웨어 라디오에 대해서 잘 모르신다거나 FIR filter가 무엇을 하는지 잘 모르신다면 여러분은 먼저 신호처리 이론에 대해서 견고한 백그라운드를 습득하셔야 할 것 입니다. 무엇인가를 배우는 가장 좋은 길은 해보는 것 입니다. 비록 이 튜토리얼이 여러분에게 GNU Radio에 대해서 가능한 쉽게 소개하려고 디자인됐지만, 완벽한 가이드는 아닐 겁니다. 사실, 저는 때때로 단순히 더 쉽게 설명하기 위해 진실을 말하지 않을 것 입니다. 저는 또 이후의 챕터들에서 스스로에게 모순된 말을 할 것 입니다. GNU Radio application들을 개발하기 위해서는 여전히 머리를 좀 쓰셔야 할 것 입니다.

Preliminaries

이 튜토리얼을 시작하기 전에 GNU Radio가 설치되어 동작하는지 확인하십시오. USRP는 반드시 필요하지 않지만, 어떤 종류의 쏘스(입력원)과 sink (USRP, 오디오나 다른 하드웨어)가, 비록 엄격히 요구되는건 아니지만, 도움이 될 것입니다. GNU Radio 예제들(gr-audio/examples/python 안의 dial_tone.py과 같은)이 동작한다면 시작할 준비가 되신 것 입니다. 여러분은 프로그래밍에 대한 백그라운드를 가지고 있어야 합니다. 하지만 여러분이 파이썬으로 프로그램을 해본적이 없더라도 걱정하지 마십시오. 파이썬은 배우기에 매우 쉬운 언어입니다.

Understanding flow graphs

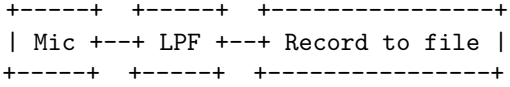
우리는 코드를 폭발적으로 작성하기 전에 먼저 GNU Radio에 대한 가장 기본적인 개념(그래프 이론에 따르는 흐름 그래프와 블록들)을 이해할 필요가 있습니다. 많은 GNU Radio application들은 흐름 그래프 외엔 아무것도 포함하고 있지 않습니다. 이런 그래프의 노드들을 블록들이라고 부르고 데이터는 그 경계를 따라 흐릅니다. 어떤 실제 신호 처리는 블록으로 완성됩니다. 이상적으로 모든 블록은 정확히 하나의 작업을 수행합니다. - 이 방법으로, GNU Radio는 모듈화 되고 유연하게 유지됩니다. 블록들은 C++로 작성되는데 새로운 블록을 작성하는 것은 매우 어렵지는 않지만 다른 곳에서 설명됩니다.

블록들 사이를 통과하는 데이터는 모든 종류들(실제로 여러분이 C++에서 정의할 수 있는 모든 데이터형) 일 수 있습니다. 사실상, 하나의 블록에서 다음 블록으로 통과되는 가장 오랜 시간동안 샘플과 비트가 될 데이터로서 가장 흔한 데이터형들은 복소수와 실수, short 또는 long형 정수, 그리고 부동소수점형 값 입니다.

Examples

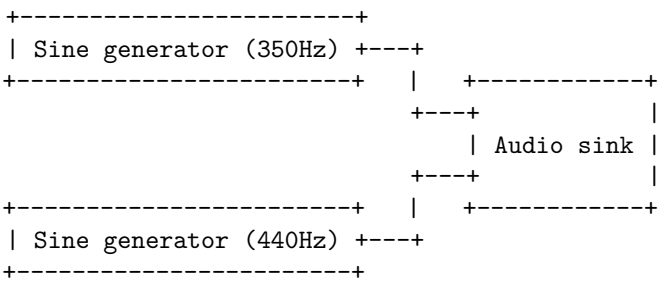
이 산만한 주제를 조금 명확히 하기 위해, 몇 개의 예제를 시작해 봅시다.

Low-pass filtered audio recorder



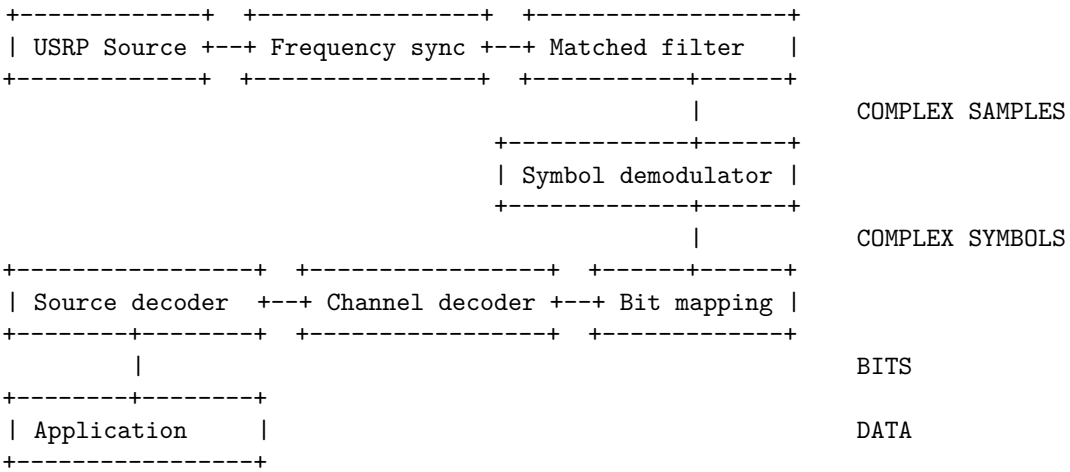
첫번째로, 마이크에서 어떤 오디오 신호가 여러분 PC의 사운드 카드에 의해 녹음되고 디지털 신호로 변환됩니다. 그 샘플들은 어떤 FIR filter와 같은 것으로 구현된 low pass filter(LPF)인 다음 블록으로 스트림 됩니다. 그 필터된 신호는 필터된 오디오 신호를 파일로 저장하는 마지막 블록으로 전달됩니다. 이 예제는 단순하지만 완전한 흐름 그래프입니다. 첫번째와 마지막 블록은 특별한 목적을 제공합니다: 그것들은 쏘스(입력원)과 싱크로서 운영됩니다. 모든 흐름 그래프는 적어도 하나의 기능하는 쏘스(입력원)과 싱크를 필요로 합니다.

Dial tone generator



이 간단한 예제는 종종 "Hello World of GNU Radio"라고 불립니다. 첫번째 예제와 달리, 두 개의 쏘스(입력원)를 가지고 있습니다. 반면에 싱크는 두 개의 입력(이 경우는 사운드 카드의 왼쪽과 오른쪽 채널)을 가지고 있습니다. 이 예제의 소스코드는 gr-audio/examples/python/dial_tone.py에 있습니다.

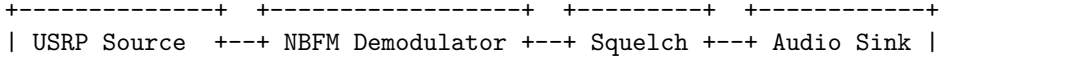
QPSK Demodulator

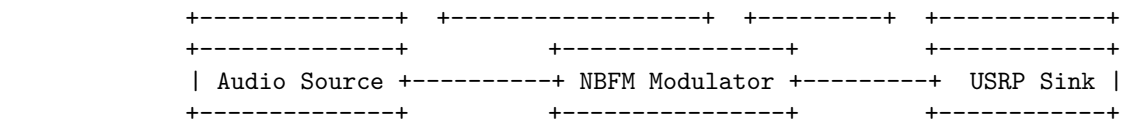


이 예제는 조금 더 복잡합니다. 하지만 RF 엔지니어에게는 꽤 익숙할 것 입니다. 이 경우는 쏘스가 하나의 안테나에 연결된 USRP 입니다. 이런 종류의 쏘스는 복소수 샘플들을 이어지는 블록에 보냅니다.

이런 종류의 흐름 그래프에서 흥미로운 부분은 흐름 그래프 동안 데이터 형들이 변한다는 것입니다: 첫번째에서 복소수 베이스밴드 샘플들이 통과됩니다. 그러면 복소수 심볼들이 신호로부터 모여집니다. 다음에는 이 심볼들이 다음 번 처리를 위해 비트로 변환됩니다. 마지막으로 디코딩 된 비트들은 그 데이터를 이용하는 어떤 응용으로 전달됩니다.

Walkie Talkie





이 applicaiton은 두 개의 분리되고 서로 병렬로 동작하는 흐름 그래프들로 구성되어 있습니다. 그것들 중 하나는 Tx(송신) 경로로 다루어지고, 다른 하나는 Rx(수신) 경로로 다루어집니다. 이런 종류의 application은 하나가 동작될 때 다른 하나를 중지시키기 위해 흐름 그래프 외부에 추가적인 코드를 요구할 것 입니다. 두 흐름 그래프는 적어도 하나의 쏘스(입력원)와 싱크를 서로 간에 요구합니다. 여러분은 조금더 복잡하지만 gr-uhd/examples/python/usrp_nbfm_ptt.py에서 이것을 수행하는 GNU Radio application을 찾을 수 있습니다.

Summary

흐름 그래프에 대한 결론을 정리합니다. 여기에 여러분이 정말로 알아야만 하는 중대한 요점을 빠르게 요약합니다.

- GNU Radio에서 모든 신호처리는 흐름 그래프로 완성됩니다.
- 흐름 그래프는 블록들로 구성됩니다. 하나의 블록은 필터링, 신호 추가, 변환, 디코딩, 하드웨어 접근 외의 많은 동작들과 같은 신호처리 운용의 하나를 수행합니다.
- 데이터는 블록 간에 복소수, 실수 정수, 부동소수점 수, 또는 여러분이 정의할 수 있는 기본적인 데이터형 등의 다양한 형태로 전달됩니다.
- 모든 흐름 그래프는 적어도 하나의 싱크와 쏘스(입력원)를 필요로 합니다.

A first working code example

다음 단계는 이런 흐름 그래프들을 실제 파이썬에서 어떻게 작성할 수 있는지 아는 것 입니다. 약간의 코드를 한 줄 한 줄 분석하는 것으로 시작해 보도록 하겠습니다. 여러분이 파이썬을 잘 알고 있다면 몇몇 설명은 아마 건너뛰어도 좋을 것 입니다만, 그렇지 않다면 아직 다음 절로 달려가시면 안됩니다. 이 설명은 파이썬과 GNU Radio 초보자 모두를 위한 것 입니다. 아래의 코드 예제는 Dial tone generator의 흐름 그래프를 표현한 것 입니다. 실제로 이것은 여러분이 gr-audio/examples/python/dial_tone.py에서 찾을 수 있는 코드 예제를 약간 수정한 버전입니다.

```
#!/usr/bin/env python

from gnuradio import gr
from gnuradio import audio, analog

class my_top_block(gr.top_block):
    def __init__(self):
    gr.top_block.__init__(self)

    sample_rate = 32000
    ampl = 0.1

    src0 = analog.sig_source_f(sample_rate, analog.GR_SIN_WAVE, 350, ampl)
    src1 = analog.sig_source_f(sample_rate, analog.GR_SIN_WAVE, 440, ampl)
    dst = audio.sink(sample_rate, "")
    self.connect(src0, (dst, 0))
    self.connect(src1, (dst, 1))

if __name__ == '__main__':
```

```
try:
my_top_block().run()
except [[KeyboardInterrupt]]:
pass
```

첫번째 줄은 유닉스, 리눅스 백그라운드를 가진 분들이 보시기에 익숙합니다: 이것은 이 파일은 파이썬 파일이고 이 파일을 실행하기 위해 파이썬 번역기를 이용해야 한다고 알려주는 쉘 스크립트입니다. 이 파일이 명령줄에서 직접 실행되길 원하신다면 이 줄이 필요합니다.

세번째, 네번째 줄은 GNU Radio를 실행하기 위해 필요한 파이썬 모듈을 포함 시킵니다. *import* 명령은 C/C++에서 *#include* 지시자와 비슷합니다. 여기, gnuradio에서 온 3개의 모듈이 추가되었습니다: gr, audio, 그리고 analog. 첫번째 모듈인 gr은 기본적인 GNU Radio 모듈입니다. 여러분은 GNU Radio application을 실행하기 위해 이것을 항상 포함시켜야만 합니다. 두번째 모듈은 audio device block들을 불러오고, 세번째 모듈은 analog signal functionality와 modulation을 포함시킵니다. 많은 GNU Radio 모듈들이 있고, 그 모듈들의 짧은 목록을 후에 정리하겠습니다.

6번째 줄부터 17번째 줄까지는 또 다른 클래스인 gr.top_block으로부터 상속받은 my_top_block이라는 클래스를 정의합니다. 이 클래스는 기본적으로 흐름 그래프를 위한 컨테이너입니다. gr.top_block 클래스를 상속받음으로써, 여러분은 블록을 추가하고 그것들을 연결하기 위해 필요한 모든 hooks과 함수들을 얻을 수 있습니다.

오직 한 멤버 함수만이 이 클래스에 정의되어 있습니다. *___init___()*는 이 클래스의 생성자입니다. 이 함수의 첫번째 줄에서(8번째 줄), 부호 생성자가 호출됩니다(파이썬에서, 이것은 명시적으로 요구됩니다. 파이썬의 대부분이 명시적으로 작성되도록 요구됩니다. 사실, 이것이 파이썬의 주요 원칙입니다.). 다음으로, 두개의 변수가 정의되었습니다. *sample_rate* 와 *ample*. 이들이 신호 생성기의 sampling rate와 amplitute를 제어할 것 입니다.

다음 줄을 설명하기 전에, 이전 섹션에 세 개의 블록과 두 개의 경계로 구성된 흐름 그래프 차트를 보셨으면 합니다. 블록들은 13번째 줄부터 15번째 줄까지 정의되어 있습니다: 두 개의 신호원이 생성됩니다(*src0*과 *src1*). 이들 입력원은 주어진 주파수(350과 440Hz)와 주어진 sampling rate(32kHz)로 지속적으로 사인파를 생성합니다. 진폭은 *ampl* 변수로 조절되고 0.1로 셋팅되었습니다. 블록 형 *analog.sig_source_f*의 접미사 "f"는 출력이 *float* 이라는 것을 지시합니다. 이것은 오디오 싱크가 -1에서 1 사이의 범위에서 부동소수점 샘플을 허용하기 때문에 좋습니다. 이런 종류의 것들은 프로그래머가 주의해서 다루어야 합니다: 비록 GNU Radio가 연결이 올바른지 확인하지만, 수동으로 주의해서 다루어야 하는 것이 여전히 있습니다. 예를 들어, 여러분이 정수 샘플들을 *audio.sink*로 보내고 싶다면, GNU Radio는 에러를 낼 것 입니다. 하지만 여러분이 위의 예제에서 진폭을 1보다 큰 어떤 값으로 셋팅한다면 에러가 아닌 왜곡된 신호를 얻을 것 입니다.

신호 싱크는 15번째 줄에 정의되어 있습니다: *audio.sink()*는 soundcard control로서 행동하며 그것에 어떤 샘플을 전송하여 재생하는 블록을 리턴합니다. 먼저 블록 안에서 비록 신호원에 의해 이미 셋팅되었더라도 sampling rate이 명시적으로 셋팅될 필요가 있습니다. GNU Radio는 (그것이 블록들 사이의 정보 흐름의 부분이 아니므로) 올바른 sampling rate를 문맥으로부터 추정할 수 없습니다.

16번째와 17번째 줄은 블록들을 연결합니다. 블록들을 연결하는 일반적인 문법은 *self.connect(block1, block2, block3, ...)*입니다. 이것은 *block1*의 출력과 *block2*의 입력을 열견하고, *block2*의 출력과 *block3*의 입력을 연결하는 식으로 계속 연결합니다. 여러분은 *connect()*를 한 번 호출함으로써 여러분이 원하는 만큼의 block들을 연결할 수 있습니다. 여기, 우리는 *src0*와 *dst*의 첫번째 입력과 연결하고 *src1*과 *dst*의 두번째 입력에 연결하기를 원하므로 특별한 문법이 필요합니다. *self.connect (src0, (dst, 0))*이 바로 그것을 수행합니다: 그것은 명백하게 *src0*와 *dst*의 port 0를 연결합니다. *(dst, 0)*은 파이썬 용어로 "tuple"이라고 불립니다. 그것은 *self.connect()*를 호출할 때 port 번호를 명세 합니다. port 번호가 0일 때 block은 port 번호를 안쓸 수 있습니다. 그러므로 16번째 줄은 다음과 같이 쓸 수 있습니다.

```
self.connect((src0, 0), (dst, 0))
```

흐름 그래프를 만드는 모든 것을 다 했습니다. 마지막 5 줄(22번째 줄)은 흐름 그래프를 시작하는데 관련이 없습니다. 이 *try*와 *except* 구문은 단순히 흐름 그래프가 Ctrl+C가 눌렸을 때(이것은 *KeyboardInterrupt* 파이썬 예외를 발생합니다) (안그랬으면 무한히 돌아갔을) 흐름 그래프가 멈춰지는 것을 확인하는 것 입니다.

파이썬 초보자들을 위해, 두 가지 언급을 더 하지 않을 수 없습니다: 여러분은 *my_top_block* 클래스가 이전에 인스턴스를 생성하지 않고 실행되었다는 것을 알아차렸을지 모르겠습니다. 파이썬에서, 특별히 여러분이 어떻게든 오직 하나의 인스턴스만을 얻으려는 클래스를 갖고 있다면 이것은 꽤 상식적인 행동입니다. 하지만, 여러분이

하나의 인스턴스를 생성하든 그 이상의 클래스 인스턴스를 생성하든 인스턴스에서 *run()* 메소드를 호출해야 합니다.

두번째로, 들여쓰기는 코드의 한 부분이고 C++처럼 단순히 프로그래머의 편리를 위한게 아닙니다. 여러분께서 이 코드를 실행해 보고 수정해 볼 때 탭과 스페이스를 섞어쓰지 않아야 합니다. 모든 단계는 일관적으로 들여써져야 합니다.

여러분께서 이 튜토리얼로 계속 진행하시길 바란다면, 좀더 견고한 파이썬 지식을 먼저 얻어야 합니다. 파이썬 문서는 파이썬 웹사이트 http://www.python.org 또는 여러분이 선택한 라이브러리에서 찾을 수 있습니다. 이전에 프로그래밍 경험이 있던 사람들을 위한 가장 좋은 곳은 http://wiki.python.org/moin/BeginnersGuide/Programmers 입니다.

Summary

- 필수적인 GNU Radio 모듈들을 추가하기 위해서 *from gnuradio import* 명령을 써야 합니다. 항상 *gr* 모듈이 필요합니다.
- 흐름 그래프는 *gr.top_block*으로부터 상속받은 클래스에 포함됩니다.
- 블록들은 *analog.sig_source_f()*와 같은 함수를 호출함으로써 그리고 변수에 반환값을 저장함으로써 생성됩니다.
- 블록들은 흐름 그래프 클래스 내에서 *self.connect()*를 호출함으로써 연결됩니다.
- 여러분께서 지금 기초 파이썬 코드를 작성하는데 편안함을 느끼지 못하신다면, 몇몇 파이썬 튜토리얼을 다 끝낼때까지 잠시 멈추십시오.

Coding Python GNU Radio Applications

위의 예제에서 파이썬 GNU Radio 응용프로그램 작성법에 대해 이미 꽤 많은 부분을 다루었습니다. 이 챕터와 다음 챕터에서 GNU Radio 응용프로그램의 가능성과 어떻게 그것을 이용할지 보여드리려 합니다. 지금부터, 순차적으로 읽으실 필요가 없습니다. 소제목을 보면서 여러분이 먼저 보고 싶으신 것부터 보십시오.

GNU Radio Modules

GNU Radio에는 매우 많은 라이브러리들과 모듈들이 있습니다. 여러분은 대게 다음의 문법으로 모듈들을 추가할 것 입니다.

```
from gnuradio import MODULENAME
```

모듈들은 좀 다르게 동작합니다. 다음은 가장 흔히 사용되는 모듈들입니다.

MODULE	설명
gr	Main GNU Radio 라이브러리. 거의 항상 이것을 포함해야 합니다.
analog	아날로그 신호나 아날로그 변조와 관련된 어떤 것들
audio	사운드 카드 제어(신호원들, 싱크들). 이것으로 사운드 카드에 오디오를 보내거나 받을 수 있지만, 확장 RF 프론트엔드와 협대역 리시버로서 사운드카드를 이용할 수도 있습니다.
blocks	기타 블록들. 만약 어떤 카테고리에 들어가 있지 않다면 이곳에 있을겁니다.
channels	시뮬레이션을 위한 채널 모델들
digital	디지탈 변조와 관련된 것들
fec	Forward Error Correction(FEC)와 관련된 것들

MODULE	설명
fft	FFT와 관련된 것들
filter	firdes이나 optfir과 같은 필터 블록들과 디자인 툴들
plot_data	Matplotlib로 데이터를 그래프로 그리는(plot) 기능들
qtgui	QT library를 사용하여 데이터를 그래프(time, frequency, spectrogram, constellation, histogram, time raster)로 그리는(plot) 그래픽 툴
trellis	trellis, trellis coded modulation , FSM(finite state machine)를 구축하기 위한 블록들과 도구들
vocoder	음성 코딩/디코딩을 다루는 것들
wavelet	wavelets을 다루는 것들
wxgui	여러분의 흐름 그래프로 빠르게 GUI를 만들 수 있는 유틸리티를 포함한 하위 모듈. submodule에 모든 것을 추가하기 위해서는 *from gnuradio.wxgui import ** 사용하십시오 . 특별한 컴포넌트들을 추가하기 위해서는 *from gnuradio.wxgui import stdgui2, fftsink2*와 같이 사용하십시오. Graphical User Interfaces에서 더 많은 정보를 다룹니다.
eng_notation	예를 들어 100 * 10^6'를 표시하기 위해 @100M'를 쓰는 것처럼 공학 표기법을 위해 기능들을 추가합니다.
eng_options	이 기능을 추가하기 위해 *from gnuradio.eng_options import eng_options*를 사용하십시오. 이 모듈은 공학 표기법을 이해하기 위해 파이썬 *optparse* 모듈을 확장합니다.
gru	잡다한 유틸리티들, 수학적인 것, 기타 등등

이것은 아주 완벽한 리스트도 아니고 그 자체로 매우 유용한 모듈 설명도 아닙니다. GNU Radio code는 자주 변경되어서 정적인 문서를 만드는 것은 매우 현명하지 못한 일일 겁니다. GNU Radio는 동적으로 API 문서를 생성하기 위해 Doxygen과 Sphinx를 씁니다.

대신에, 여러분은 모듈들의 세세한 것을 더 탐구하기 위해 오래된 스타워즈의 모토를 따르실 수 있습니다. "쏘스를 이용하라!" GNU Radio에 여러분이 원하는 기능이 이미 있다고 여기신다면 파이썬을 사용할 모듈 디렉토리를 훑어보시거나 GNU Radio의 쏘스 디렉토리를 찾아보십시오. 특별히 쏘스 디렉토리에서 gr-trellis와 같이 *gr-*로 시작하는 디렉토리들에 주의를 기울이십시오. 이것들은 그 자신의 코드와 결과적으로 그 자신의 모듈들을 갖고 있습니다.

물론 파이썬 스스로도 GNU Radio application을 만들기 위해 몇몇은 필요없을지라도 엄청나게 유용한 많은 모듈들을 갖고 있습니다. 더 많은 정보를 확인하기 위해 파이썬 문서와 SciPy 웹사이트를 찾아보십시오.

Choosing, defining, and configuring blocks

GNU Radio는 미리 정의된 다양한 블록들로 구성되어 있어서 초보자들이 응용프로그램을 만들기 위해 올바른 블록을 찾거나 올바르게 셋팅하는데 자주 어려움을 느끼게 됩니다. Doxygen과 Sphinx는 C++과 파이썬 API의 문서 자동화를 위해 이용됩니다.(C++ Manual, Python Manual) 여러분도 이 문서를 생성할 수 있어서 항상 여러분이 설치한 버전에 맞을 것 입니다. 여러분이 Doxygen과 Sphinx를 설치했다면 CMake로 이것을 자동화 할 수 있습니다.

이 문서화 방법을 배우는 것은 GNU Radio를 이용하는 법을 배우는데 가장 중요한 부분입니다.

실습을 해보겠습니다. 이전 예제로부터 블록을 정의하는 세 개의 줄이 있습니다.

```
src0 = analog.sig_source_f (sample_rate, analog.GR_SIN_WAVE, 350, ampl)
src1 = analog.sig_source_f (sample_rate, analog.GR_SIN_WAVE, 440, ampl)
dst = audio.sink (sample_rate, "")
```

이 코드가 실행될 때 무슨 일이 일어나는지 간단히 보겠습니다. 첫번째로, *analog* 모듈의 *sig_source_f* 함수가 실행됩니다. 이 함수는 4개의 인자를 받습니다.

- 파이썬 변수, sample_rate
- *analog* 모듈에서 정의되어 있는(여기) 상수, analog.GR_SIN_WAVE
- sample rate에 비례하는 sine wave의 주파수인 보통의 문자열, 상수 350
- sine wave의 진폭으로 설정할 변수, ampl

이 함수는 이어서 *src0*에 할당될 클래스를 생성합니다. 비록 sink는 다른 모듈(audio)에서 가져왔지만 다른 두 줄에서도 동일한 일이 생깁니다.

어떤 블록을 사용할지, *analog.sig_source_f()*에 무엇을 전달할지 어떻게 알 수 있겠습니까? 문서를 보면 됩니다. 여러분이 Sphinx에서 생성한 문서를 보신다면, "gnuradio.analog"를 클릭 하십시오. "Signal Sources"로 가십시오.

(역주)*analog.sig_source_f()*가 GNU Radio 3.7.4에는 없습니다. 하지만 GNU Radio 3.7.3에서 그 클래스를 확인할 수 있습니다. (steveyoon@telechips.com)

여러분은 *sig_source_* 족을 포함한 신호 발생기 목록을 찾을 수 있을 겁니다. 접미사는 출력 자료형을 정의합니다.

- f = float
- c = complex float
- i = int
- s = short int
- b = bits (정수 형)

이 접미사들은 블록들의 모든 형 정의를 위해 사용됩니다. 예를 들어, *filter.fir_filter_ccf()*는 복소수 입력, 복소수 출력, 부동소수점 Tap으로 FIR 필터를 정의하고, *blocks.add_const_ss()*는 들어오는 short int를 다른 short int 상수에 더하는 블록을 정의합니다.

비록 여러분이 C++은 건드리고 싶지 않을지라도 대부분의 블록들이 사실 C++로 만들어진 다음 파이썬으로 export된 것이기 때문에 Doxygen으로 만들어진 문서를 보는 것은 가치가 있습니다.

이 시점에서, GNU Radio 커튼 뒤를 좀 가까이 바라보는 것이 좋은 것 같습니다. 여러분이 C++로 작성된 블록을 파이썬에서 쉽게 이용할 수 있는 이유는 GNU Radio가 SWIG이라는 툴을 파이썬과 C++ 간의 인터페이스를 생성하는데 쓰기 때문입니다. 앞에서 언급된 예제에서 *gr::analog::sig_source_f::make()*처럼 *gr::component::block::make(***)*와 같이 C++에서의 모든 블록들은 생성하는 함수로부터 시작합니다. 이 함수는 항상 그것에 맞는 클래스와 같은 페이지에 문서화 되어 있습니다. 그리고 이 함수가 파이썬에 export 되었습니다. 그래서 파이썬에서 *analog.sig_source_f()*는 C++에서 *gr::analog::sig_source_f::make()*를 호출합니다. 같은 이유 때문에, 같은 argument(인수)를 갖습니다. - 그것이 여러분이 파이썬에서 블록을 초기화 하는 법을 아는 것입니다.

(역주)파이썬 모듈은 C++로 먼저 작성된 뒤 파이썬으로 export 된 것이기 때문에 C++ API 문서인 Doxygen 문서를 보면 파이썬에서의 모듈 블록을 초기화 할 때 인자가 무엇인지 알 수 있게 된다는 말 입니다. (steveyoon@telechips.com)

여러분이 Doxygen 문서에서 클래스 *gr::analog::sig_source_f*를 불러올 때, *set_frequency()*같은 많은 다른 클래스 메소드를 보게 될 겁니다. 이 함수들은 파이썬으로도 export되었습니다. 그래서 여러분이 만약 한 신호원을 생성했었고 주파수를 변경하기를 원한다면 이 메소드를 여러분의 파이썬으로 정의된 블록에 사용할 수 있습니다.

```
# We're in some cool application here
src0 = analog.sig_source_f (sample_rate, analog.GR_SIN_WAVE, 350, ampl)
# Other, fantastic things happen here
src0.set_frequency (880) # Change frequency
```

이 코드는 첫번째 신호발생기의 주파수를 880Hz로 변경할 것 입니다.

희망스럽게도, GNU Radio 문서는 점점 완성도 높아지고 있습니다. 그러나 블록의 상세한 동작을 완전히 이해하기 위해서는, 얼마나 좋은 문서를 갖고 있든지 간에 바로 코드를 보시거나 나중에라도 보셔야 할 것 입니다.

Connecting blocks

*gr.top_block*의 메소드인 *connect()*를 블록들을 연결하기 위해 사용하십시오. 유용한 언급들을 하겠습니다.

- 여러분은 오직 자료형이 맞는 경우에만 입력과 출력을 연결하실 수 있습니다. 만약에 부동소숫점 출력과 복소수형 입력을 연결하려고 하면 에러를 낼 겁니다.
- 하나의 출력은 여러개의 입력과 연결될 수 있습니다. 그래서 신호 경로를 복사하기 위한 추가적인 블록이 필요 없습니다.

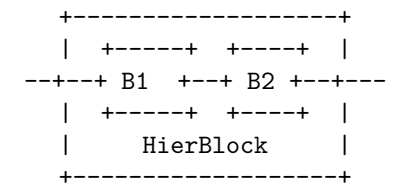
이것들은 블록들을 연결하는 기본적인 규칙입니다. 그리고 대부분의 경우 잘 동작합니다. 그러나 혼합 자료형을 쓸 때 유용한 주의 사항을 언급하겠습니다.

- GNU Radio는 입력과 출력의 데이터형이 짝을 이루는지 그 크기를 체크해서 확인합니다. 만약 여러분이 같은 크기이지만 서로 다른 데이터형의 포트들을 연결하려 한다면, 여러분은 분명히 쓰레기 데이터를 얻게 될 것 입니다.
- 싱글 비트들을 처리할 때에는 주의를 하십시오. 어떤 경우에는, 보통의 감각으로 바이너리 데이터로 작업해야 할 것 입니다. 어떤 시점의 특정 개수의 비트들을 다루고 싶어지는 경우도 생깁니다. 이것을 위해 *packed_to_unpacked**와 *unpacked_to_packed** 블록을 보십시오.
- 동적 범위를 다룰 때는 주의를 기울이십시오. 부동소수점형이나 복소수형을 사용할 때, 고려하는 기계에서 필요로 하는 것보다 더 큰 범위를 가집니다. 그러나 어떤 싱크와 쏘스들은 여러분이 고정되길 필요로 하는 특별한 범위를 갖기도 합니다. 예를 들어, 오디오 싱크들은 +-1 사이의 샘플들을 필요로 하고 이 범위를 벗어나면 잘려집니다. 반면에 USRP 싱크는 DAC의 동적 범위 때문에 -32768부터 32767까지의 범위를 갖습니다.

Hierarchical blocks

때때로 몇몇 블록을 하나의 블록으로 합치기도 합니다. 여러분이 모두 몇 개의 블록으로 구성된 공통 신호처리 컴포넌트를 가진 몇 개의 응용프로그램을 가졌다고 말해봅시다. 이 블록들은 보통의 GNU Radio 블록이라면 여러분의 응용프로그램에서 사용될 수 있는 새로운 블록으로 결합될 수 있습니다.

예제: 여러분이 FG1과 FG2라는 두 개의 다른 흐름 그래프를 갖고 있다고 합시다. 이 둘은 다른 것들 중에서 블록 B1과 B2를 이용합니다. 여러분은 그것들을 *HierBlock*이라는 계층 구조의 블록에 결합하고 싶어 합니다.



이것이 여러분이 할 것입니다. *gr.hier_block2*로부터 상속받은 흐름 그래프를 만들고, 쏘스와 싱크로서 *self*를 이용합니다.

```
class HierBlock(gr.hier_block2):
    def __init__(self, audio_rate, if_rate):
        gr.hier_block2.__init__(self, "HierBlock",
                        gr.io_signature(1, 1, gr.sizeof_float),
                        gr.io_signature(1, 2, gr.sizeof_gr_complex))

        B1 = blocks.block1(...) # Put in proper code here!
        B2 = blocks.block2(...)

        self.connect(self, B1, B2, self)
```

보시다시피, 계층 구조의 블록을 만드는 것은 *gr.top_block*으로 흐름 그래프를 만드는 것과 매우 비슷합니다. 쏘스와 싱크로서 *self*를 이용하는 것과 별개로, 또 다른 차이가 있습니다. (세번째 줄에서 호출되는) 부모 클래스의 생성자가 추가적인 정보를 받도록 합니다. *gr.hier_block2.___init___()* 호출에 4개의 파라미터가 필요합니다. * self (항상 첫번째 인수로 생성자를 전달합니다.) * 계층 구조 블록을 위한 식별자로서 문자열 (편리한대로 바꾸십시오) * 입력 서명 * 출력 서명

마지막 두 개는 여러분이 이미 C++로 자신의 블록들 작성했을지라도 추가적인 설명이 요구됩니다. GNU Radio는 블록에서 사용하는 입력과 출력의 자료형이 무엇인지 알아야만 합니다. 입출력 서명을 만드는 것은 *gr.io_signature()*를 호출함으로서 완료될 수 있습니다. 이 함수는 3개의 인수를 가지고 호출됩니다. * 포트의 최소 개수 * 최대 포트의 개수 * 입출력 요소의 크기

계층 구조 블록 *HierBlock*에서 여러분은 정확히 하나의 입력과 하나 또는 두 개의 출력을 볼 수 있습니다. 들어오는 객체는 부동소수점 크기를 가지고 있어서, 블록은 들어온 부동소수점수 실수를 처리합니다. B1 또는 B2의 어딘가에서 데이터는 복소 부동소수점 수 값으로 변환됩니다. 그래서 출력 서명이 나가는 객체의 크기를 *gr.sizeof_gr_complex*의 크기로 선언됩니다. *gr.sizeof_float*와 *gr.sizeof_gr_complex*는 C++에서 *sizeof()를 호출하여 얻은 반환값과 같습니다. 다른 선정의된 상수들은* gr.sizeof_int * gr.sizeof_short * gr_sizeof_char 입니다.

특별히 쏘스와 싱크로서 계층 구조 블록을 정의하는 것으로서, null IO 서명을 생성하기 위해 gr.io_signature(0,0,0)을 이용합니다.

그것이 다 입니다. 이제 여러분은 보통 블록을 사용하여 *HierBlock* 쓸 수 있습니다. 예를 들면, 같은 파일에서 이 코드를 넣을 수 있습니다.

```
class FG1(gr.top_block):
  def __init__(self):
      gr.top_block.__init__(self)

  ... # Sources and other blocks are defined here
  other_block1 = blocks.other_block()
  hierblock    = HierBlock()
  other_block2 = blocks.other_block()

  self.connect(other_block1, hierblock, other_block2)

  ... # Define rest of FG1
```

물론 파이썬 모듈화를 하기 위해 *HierBlock*을 위한 코드를 *hier_lock.py*라는 외부 파일에 넣을 수도 있습니다. 다른 파일로부터 이 블록을 이용하기 위해 간단히 여러분의 코드에 import 지시자를 추가하시면 됩니다.

```
from hier_block import HierBlock
```

그리고 앞서 언겁된것처럼 *HierBlock*을 이용하실 수 있습니다.

Hierarchical 블록들의 예는 다음과 같습니다.

```
gr-uhd/examples/python/fm_tx_2_daughterboards.py
gr-uhd/examples/python/fm_tx4.py
gr-digital/examples/narrowband/tx_voice.py
```

Multiple flow graphs

어떤 경우에는 예를 들어 송수신 path(워키토키 예제 참조)와 같이 완전히 분리된 흐름 그래프가 필요할 때가 있습니다. 2008년 6월 현재, 동시에 동작하는 다중 top_blocks는 불가능합니다. 그러나 여러분이 하려는 것은

앞선 장에서와 같이 *gr.hier_block2*를 이용한 계층 구조 블록으로 전체 흐름 그래프를 생성할 수 있습니다. 그럼, 흐름 그래프들을 붙잡는 top_block을 만들어 보겠습니다.

예제:

```
class transmit_path(gr.hier_block2):
    def __init__(self):
                gr.hier_block2.__init__(self, "transmit_path",
                                gr.io_signature(0, 0, 0), # Null signature
                                gr.io_signature(0, 0, 0))

        source_block = blocks.source()
        signal_proc  = blocks.other_block()
        sink_block   = blocks.sink()

        self.connect(source_block, signal_proc, sink_block)

class receive_path(gr.hier_block2):
    def __init__(self):
                gr.hier_block2.__init__(self, "receive_path",
                                gr.io_signature(0, 0, 0), # Null signature
                                gr.io_signature(0, 0, 0))

        source_block = blocks.source()
        signal_proc  = blocks.other_block()
        sink_block   = blocks.sink()

        self.connect(source_block, signal_proc, sink_block)

class my_top_block(gr.top_block):
    def __init__(self):
        gr.top_block.__init__(self)

    tx_path = transmit_path()

    rx_path = receive_path()

    self.connect(tx_path)
    self.connect(rx_path)
```

이제, *my_top_block*이 시작할 때 두 흐름 그래프가 병렬로 시작됩니다. 계층 구조 블록들이 명시적으로 정의된 입출력이 없다는 것에 주목하십시오. 그 블록들은 null IO signauture를 썼습니다. 결과적으로 그것들은 쏘스나 싱크로서 *self*에 연결하지 않습니다. 그 블록들은 자신만의 쏘스와 싱크를 정의하는 것이 더 낫습니다(계층 구조 블록을 정의하고 있는 그 때, 여러분이 정의하려는 만큼의 쏘스와 싱크). 최상위 블록은 단순히 계층 구조 블록들을 스스로에게 연결하지만 그것들에 어떤 방법으로든 접속하지는 않습니다.

다중 흐름 그래프의 예:

```
gr-uhd/examples/python/usrp_nbfm_ptt.py
```

GNU Radio extensions and tools

GNU Radio는 블록들과 흐름 그래프들에 더해 많은 도구들과 여러분이 DSP 응용 프로그램을 작성하기에 도움이 되는 코드들이 딸려 있습니다. 여러분을 돕도록 설계된 유용한 GNU Radio 응용 프로그램의 집합은 gr-util/

안에 있습니다. 필터 설계 코드나 변조 유틸리트 등과 같이 여러분의 파이썬 코드에서 이용할 수 있는 유틸리티를 찾기 위해 gnuradio-runtime/python/gnuradio에서 (그리고 다른 gr-/python 디렉토리에서도) 소스 코드를 찾아보십시오.

Controlling flow graphs

여러분이 여기까지 튜토리얼을 따라왔다면 흐름 그래프는 항상 *gr.top_block*에서 상속받은 클래스로 구현된다는 것을 알아차렸을 것 입니다. 이 클래스를 어떻게 제어하느냐는 질문이 남아있습니다.

이전에 언급한것처럼, *gr.top_block*으로부터 상속받을 클래스는 여러분이 필요로 할 모든 기능들을 가져옵니다. 현재의 흐름 그래프를 실행하거나 멈추기 위해, 아래의 메소드들을 사용하십시오:

run()	흐름 그래프를 실행하는 가장 간단한 방법입니다. *start()*를 호출한 뒤에 *wait()*를 호출하십시오. 스스로 멈추거나 SIGINT를 받을 때까지 막연히 실행되어야 할 흐름그래프를 실행할 때 사용되곤 합니다.
start()	정지 상태의 흐름 그래프를 시작합니다. 스레드들이 생성되면 호출자에게 반환됩니다.
stop()	실행 중인 흐름 그래프를 멈춥니다. 스케쥴러에 의해 생성된 스레드들이 종료되도록 알리고 호출자에게 반환됩니다.
wait()	흐름 그래프가 완료되기까지 기다리게 합니다. 흐름 그래프는 (1)모든 블록이 완료되었거나 (2) 종료 요청을 받아서 멈추었을 때 완료가 됩니다.
lock()	재설정을 준비하는 동안 흐름 그래프를 잠급니다.
unlock()	재설정을 준비하는 동안 흐름 그래프의 잠금을 풉니다. 같은 수의 *lock()*과 *unlock()*이 있을 때, 흐름 그래프는 자동으로 재시작할 것 입니다.

더 자세한 내용을 보려면 gr::top_block 문서를 찾으십시오.

예제:

```
class my_top_block(gr.top_block):
    def __init__(self):
        gr.top_block.__init__(self)
    ... # Define blocks etc. here

if __name__ == '__main__':
    my_top_block().start()
    sleep(5) # Wait 5 secs (assuming sleep was imported!)
    my_top_block().stop()
    my_top_block().wait() # If the graph is needed to run again, wait() must be called after stop
    ... # Reconfigure the graph or modify it
    my_top_block().start() # start it again
    sleep(5) # Wait 5 secs (assuming sleep was imported!)
    my_top_block().stop()  # since (assuming) the graph will not run again, no need for wait() to be cal
```

이 메소드들은 여러분이 흐름 그래프를 밖에서 제어하도록 도와줍니다. 하지만 많은 문제들 때문에 이것은 충분치 않을 것 입니다: 여러분은 단순히 흐름 그래프를 시작하거나 멈추게 하는 것 말고 행동을 재설정 하고 싶을 겁니다. 예를 들어, 여러분의 응용 프로그램이 흐름 그래프 안 어딘가에 볼륨 제어를 한다고 상상해 봅시다. 이 볼륨 제어는 샘플 스트림에 곱셈기를 삽입함으로써 구현됩니다. 이 곱셈기는 *blocks.multiply_const_ff* 타입입니다. 이 블록을 문서에서 찾아보면, 곱셈 인수를 셋팅하는 *blocks.multiply_const_ff.set_k()*를 볼 수 있습니다. 여러분은 블록을 제어하기 위해 셋팅 값들을 밖에서 볼 수 있게 해야 합니다. 가장 간단한 방법은 블록을 흐름 그래프 클래스의 속성(attribute)으로 만드는 것 입니다.

예제:

```
class my_top_block(gr.top_block):
    def __init__(self):
        gr.top_block.__init__(self)
    ... # Define some blocks
    self.amp = blocks.multiply_const_ff(1) # Define multiplier block
    ... # Define more blocks

        self.connect(..., self.amp, ...) # Connect all blocks

    def set_volume(self, volume):
    self.amp.set_k(volume)

if __name__ == '__main__':
    my_top_block().start()
    sleep(2) # Wait 2 secs (assuming sleep was imported!)
    my_top_block.set_volume(2) # Pump up the volume (by factor 2)
    sleep(2) # Wait 2 secs (assuming sleep was imported!)
    my_top_block().stop()
```

이 예제는 2 초 동안 흐름 그래프를 실행한 뒤, 멤버 함수인 *set_volume()*을 호출함으로써 *amp* 블록에 접속하여 볼륨을 두 배로 만듭니다. 물론, 멤머 함수를 생략하고 *amp* 속성(attribute)에 직접 접근할 수도 있었습니다.

힌트: 블록에 흐름 그래프의 속성(attribute)를 만드는 것은 일반적으로 추가적인 멤버 함수로 더 쉽게 흐름 그래프를 확장하는 것으로서 좋은 방법입니다.

Non-flow graph centered applications

지금까지, 이 튜토리얼에서 GNU Radio 응용프로그램들은 항상 *gr.top_block*에서 상속받은 하나의 클래스에 집중되었습니다. 하지만, GNU Radio를 이용하기 위해 꼭 이럴 필요는 없습니다. GNU Radio는 파이썬으로 DSP 응용 프로그램을 개발하기 위해 설계 되었습니다. 그래서 GNU Radio를 사용할 때 파이썬의 모든 힘을 쓰지 않을 이유가 없습니다.

파이썬은 무지막지하게 강력한 언어입니다. 그리고 새로운 라이브러리들과 기능들이 연속적으로 추가되고 있습니다.

어떤면에서, GNU Radio는 파이썬을 강력하고 실시간 가능한 DSP 라이브러리로 확장합니다. 이것과 다른 라이브러리들을 결합함으로써 굉장한 기능을 여러분의 손 끝 바로 거기에 둘 수 있습니다. 예를 들어, GNU Radio와 파이썬 과학계산 라이브러리 모음인 SciPy를 결합함으로써, 여러분은 하나의 응용 프로그램에서 실시간으로 RF 신호를 녹취할 수 있고 오프라인으로 확장된 수학 연산을 할 수 있고 통계값을 데이터베이스에 저장할 수 있는 등의 일을 할 수 있습니다. 여러분이 이런 모든 라이브러리들을 결합한다면 Matlab 같은 비싼 공학용 소프트웨어 조차도 쓸 필요가 없게 될 것 입니다.

http://www.scipy.org

Advanced Topics

이전 섹션들을 읽으셨다면, 이미 첫번째 파이썬 GNU Radio 응용 프로그램을 작성할만큼 충분히 알게 되셨을 겁니다. 이 섹션은 파이썬 GNU Radio 응용 프로그램을 위한 고급 기능들을 조금 가볍게 이야기 하려고 합니다.

Dynamic flow graph creation

대부분의 경우에, 흐름 그래프를 정의하기 위해 앞서 이야기 한 방법은 전적으로 충분합니다. 만약에 응용 프로그램을 더 유연하게 만들려고 한다면, 클래스 밖에서 흐름 그래프를 한층 더 관리하고 싶어질 것 입니다. 이것은

코드를 *___init___()* 함수 밖으로 가져오고 간단히 *gr.top_block*을 컨테이너로서 이용하는 것으로써 할 수 있습니다.

예: ... # We are inside some application tb = gr.top_block() # Define the container

```
block1 = blocks.some_other_block()
block2 = blocks.yet_another_block()

tb.connect(block1, block2)

... # The application does some wonderful things here

tb.start() # Start the flow graph

... # Do some more incredible and fascinating stuff here
```

여러분이 흐름그래프를 (재설정하거나 재시작하는 등) 동적으로 멈추는 어떤 응용 프로그램을 작성하고 있다면 이것이 좀더 실용적인 방법일 것입니다.

이런 종류의 흐름 그래프 셋업의 예제:

gr-uhd/apps/hf_explorer/hfx.py

Command Line Options

파이썬에 명령줄 옵션을 파싱하는 라이브러리가 있습니다. 어떻게 사용할지에 대해서 *optparse* 모듈 문서를 찾아보십시오.

GNU Radio는 새 명령줄 옵션 자료형으로 optparse를 확장합니다. 이 확장을 추가하기 위해 *from gnuradio.eng_option import eng_option* 사용하십시오. eng_option에서 아래의 자료형을 쓸 수 있습니다.

eng_float	원래의 부동소수점 옵션과 비슷하지만 101.8M 같은 공학적 표시법도 허용합니다.
subdev	USRP에 도터보트를 명세하기 위해 A:0 처럼 오직 가용한 부장치(subdevice) 설명자들만 허용합니다.
intx	오직 정수만 허용합니다.

여러분의 응용 프로그램이 명령줄 옵션들을 제공한다면, 명령줄 옵션으로 GNU Radio 관습을 따르는 것이 더 훌륭할 것 같습니다. (개발자들을 위한 힌트로) 이것들을 README.hancking에서 찾아볼 수 있습니다.

거의 모든 GNU Radio 예제가 이 기능을 사용합니다. 쉬운 예제로 dial_tone.py를 보십시오.

Graphical User Interfaces

여러분이 파이썬 전문가이고 (무슨 GUI 툴킷을 즐겨 사용하건 간에) 파이썬 GUI를 작성해본 경험이 있다면, 이 섹션을 볼 필요가 없습니다. 이전에 이야기 한 것처럼, GNU Radio는 단지 파이썬에 DSP 루틴들을 확장한 것입니다. 그냥 GUI 응용 프로그램을 작성하십시오. 그리고 GNU Radio 흐름 그래프를 추가하고 GNU Radio 정보를 여러분의 응용 프로그램에 가져올 인터페이스들을 정의하십시오. 아니면 반대로 하셔도 됩니다. 여러분의 데이터를 그래프로 그리고 싶으시다면 Matplotlib이나 Qwt를 사용하실 수 있습니다.

그러나 때때로 여러분은 성가신 widget을 설정하거나 메뉴를 정의하는 등의 일 없이 단순히 빠르게 GUI 응용 프로그램을 작성하고 싶을 때도 있을 것 입니다. GNU Radio에는 여러분이 그래픽으로 된 GNU Radio 응용 프로그램을 작성하는데 도움이 되는 미리 정의된 클래스들이 딸려 있습니다.

이 모듈들은 플랫폼 독립적인 GUI 툴킷인 wxWidgets(정확히 말해 wxPython) 기반입니다. wxPython에 대한 백그라운드가 필요합니다만 걱정하지 마십시오. 별로 복잡하지도 않고 인터넷에 몇몇 튜토리얼이 있습니다. wxPython 웹사이트에서 문서를 찾아 보십시오.

GNU Radio wxWidget 도구를 사용하기 위해 몇개의 모듈들을 추가해야 합니다.

from gnuradio.wxgui import stdgui2, fftsink2, slider, form

이 네 개의 컴포넌트들은 gnuradio.wxgui 하위 모듈로부터 추가됩니다. 여기에 모듈들의 빠른 목록이 있습니다. (다시 말하지만, 결코 완전하지 않습니다. 여러분은 gr-wxgui/python에서 모듈이나 소스를 찾아보아야만 합니다.)

stdgui2	기본 GUI 도구, 항상 써야 합니다.
fftsink2	스펙트럼 분석기나 다른 것을 만들기 위해 데이터의 FFT 그래프를 그립니다.
scopesink2	오실로스코프 출력
waterfallsink2	폭포수 출력
numbersink2	입력 데이터의 숫자값 출력
form	종종 입력 폼으로 사용됩니다. (아래를 보십시오)

다음으로, 새로운 흐름 그래프를 정의하겠습니다. 지금은, *gr.top_block*이 아니라 *stdgui2.std_top_block*을 상속받겠습니다.

```
class my_gui_flow_graph(stdgui2.std_top_block):
        def __init__(self, frame, panel, vbox, argv):
                stdgui2.std_top_block.__init__ (self, frame, panel, vbox, argv)
```

보시는 바대로, 또 다른 차이가 있습니다. 생성자는 몇 개의 새로운 인자들을 받습니다. 이것은 *stdgui2.std_top_block*이 (gr.top_block*만을 상속받을 때처럼) 흐름 그래프를 추가할 뿐 아니라 직접 메뉴와 같은 다소 기본적인 컴포넌트로 윈도우를 생성하기 때문입니다. 이것은 빨리 GUI 응용 프로그램을 개발하려는 사람들에게 좋은 소식입니다. GNU Radio는 윈도우와 widget에 추가하려는 모든 것을 생성합니다. 이 새로운 객체로 여러분이 할 수 있는 것들의 목록이 아래에 있습니다. (만약에 여러분에게 GUI 프로그래밍에 대한 생각이 없다면 아마도 아무 의미 없을지도 모릅니다.)

frame	윈도우의 wx.Frame입니다. frame.GetMenuBar()로 미리 정의된 메뉴를 얻을 수 있습니다.
panel	모든 wxControl widget을 고정하는 'frame'에 위치된 panel
vbox	panel에 배치되곤 하는 widget으로 수직 box sizer. (wx.BoxSizer(wx.VERTICAL)이 정의하는 법입니다)
argv	명령줄 인수

이제 여러분은 GUI 응응 프로그램을 작성하기 위한 모든 것을 갖췄습니다. 단순히 vbox에 새 box sizer와 widget을 추가하고 메뉴를 변경하거나 다른 무엇이든 할 수 있습니다. 어떤 일반적인 기능들은 GNU Radio GUI 라이브러리 *form*에서 더욱 단순화 되었습니다.

form은 대단히 많은 입력 widget을 갖고 있습니다: 정적 텍스트 필드(표시 전용)를 위해 *from.static_text_field()*, 부동소수점 값 입력을 위해 *form.float_field()*, 텍스트 입력을 위해 *form.text_field()*, 체크 박스를 위해 *form.checkbox_field()*, 라디오 박스를 위해 *form.radiobox_field()* 등. 완전한 목록을 위해 gr-wxgui/python/form.py의 소스 코드를 찾아 보십시오. 대부분의 이 호출들은 대부분의 인자를 알맞는 wxPython 객체로 전달합니다. 그래서 그 함수의 인자들은 꽤 스스로 설명하도록 되어 있습니다. form을 사용하여 widget을 추가하는 법을 아래에서 이야기 하는 예제들 중 하나에서 보겠습니다.

아마도 *gnuradio.wxgui*의 가장 유용한 부분은 입력 데이터를 직접 그래프로 그릴 가능성입니다. 이것을 위해, *gnuradio.wxgui*에 딸려 있는 싱크들 중 하나가 필요합니다. 이 싱크들은 단지 다른 GNU Radio 싱크들처럼 동작합니다. 하지만 또한 wxPython에서 사용하기 위해 요구되는 속성들을 갖고 있습니다.

예제:

```
from gnuradio.wxgui import stdgui2, fftsink2

# App gets defined here ...

# FFT display (pseudo-spectrum analyzer)
my_fft = fftsink2.fft_sink_f(panel, title="FFT of some Signal", fft_size=512,
             sample_rate=sample_rate, ref_level=0, y_per_div=20)
self.connect(source_block, my_fft)
vbox.Add(my_fft.win, 1, wx.EXPAND)
```

처음에, 블록이 정의됩니다 (*fftsink2.fft_sink_f*). sampling rate와 같은 일반적인 DSP 인자들은 제쳐놓고, 생성자에 전달되는 *panel* 객체도 필요로 합니다. 다음으로, 블록은 쏘스에 연결됩니다. 마침내, FFT 윈도(my_fft.win)가 실제로 표시되면서 *vbox* BoxSizer 내에 위치됩니다. 신호 블록 출력이 다른 많은 입력들에 연결될 수 있다는 것을 기억하십시오.

마지막으로, 전부가 시작되어야 합니다. GUI를 실행할 *wx.App()*가 실행되길 원하므로 시동 코드는 보통의 흐름 그래프와는 약간 다릅니다.

```
if __name__ == '__main__':
    app = stdgui2.stdapp(my_gui_flow_graph, "GUI GNU Radio Application")
    app.MainLoop()
```

*stdgui2.stdapp()*는 *wx.App*를 *my_gui_flow_graph*(첫번째 인자)로 만듭니다. 윈도 제목은 “GUI GNU Radio Application”으로 설정하였습니다.

간단한 GNU Radio GUI 들의 예제입니다.

gr-uhd/apps/uhd_fft gr-audio/examples/python/audio_fft.py ./gr-uhd/examples/python/usrp_am_mw_rcv.py

그리고 많이 더 있습니다.

What next?

파이썬으로 GNU Radio 응용 프로그램을 작성하는데 질문이 더 생긴다면, 이용할 리쏘스들이 여전히 많이 있습니다.

- 소스 코드를 이용하십시오. 특별히 gr-/examples와 gr-utils에 있는 예제는 매우 도움이 될 것 입니다.
- 메일링 리스트 저장소를 확인하십시오. 여러분의 매우 수준 높은 문제들이 이전에 질문되었을 수 있습니다.

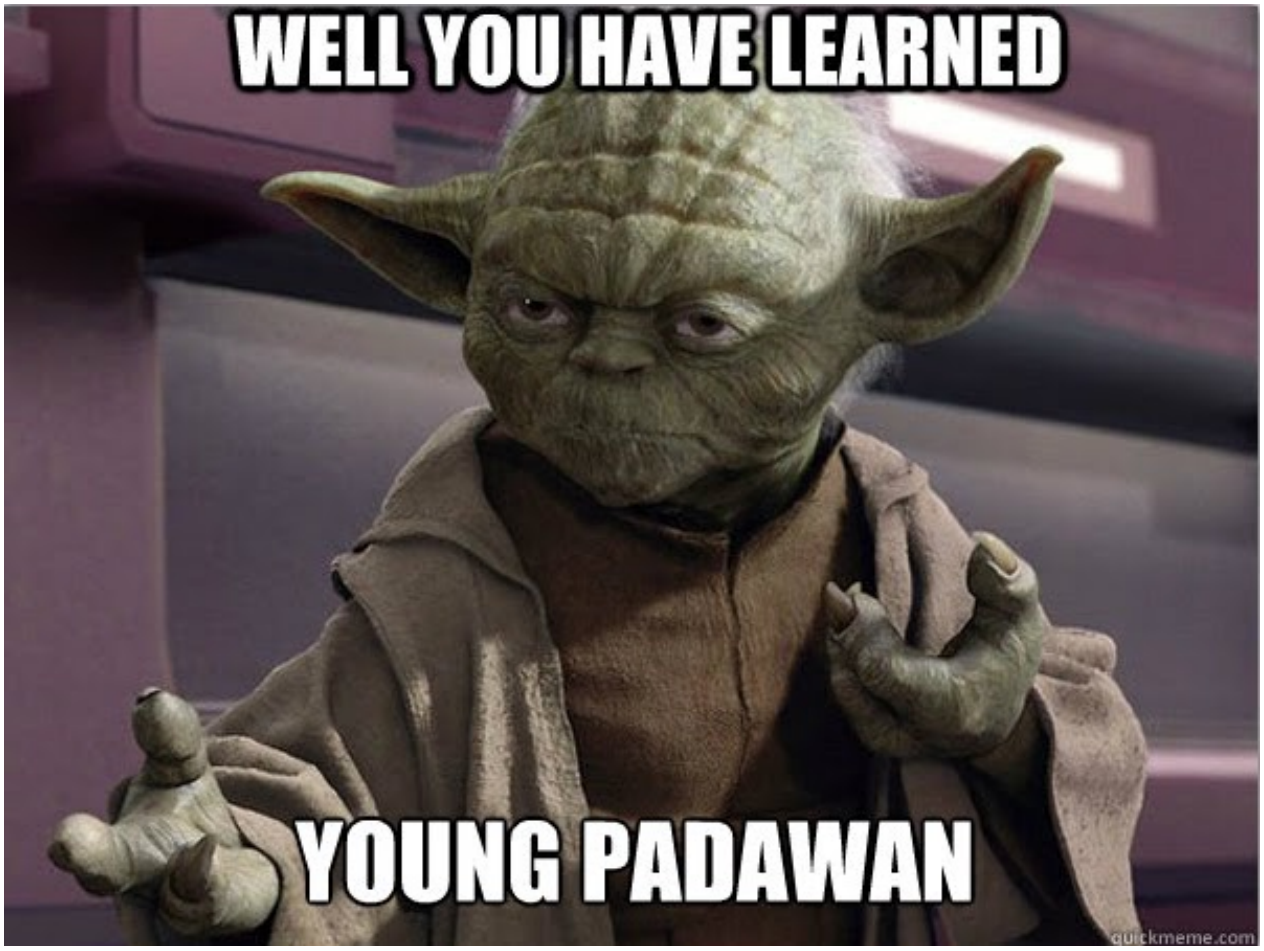


Figure 1: 이제 그만 하산하도록 하십시오