目录

指南 1：在深入 GNU Radio 之前，你应该…

摘要

GNU Radio 不但需要很强的计算机能力，而且也需要渊博的通信与数字信号处理知识。本文列举了一些有用的资源，包括书本、网页链接、在线指南。本文的目的是帮助 GNU Radio 爱好者们准备好这些有用工具。

我干脆认为你已经对 GNU Radio 产生了浓厚的兴趣，而且渴望去玩转它。不幸的是除了兴趣以外，这个过程中还包含许多其它的挑战。你需要很多领域的知识，包括通信（无线）系统，数字信号处理，基本硬件和电路设计，面向对象编程等。但是，你的兴趣与激情可以是这些变得更加容易。在本文中，我列举了一些有用的文章和资源。这些文章和资源在你深入 GNU Radio 之前是非常重要的。本文中也包含了非常好的 GNU Radio 社区建议阅读的资源，这些都是非常有用的信息。

1. 对 GNU Radio 有一个清晰的轮廓…

如果你对 GNU Radio 还没有一个清晰的轮廓的话，请首先阅读 Eric Blossom 的在线文章《开发 GNU Radio》。这本书对软件无线电做了非常简洁而准确的介绍，Eric 是整个 GNU Radio 项目的发起者。你确定已经理解了数模转换器是怎样工作的和为什么需要射频前端，回忆一下《信号与系统》课上学的抽样定理，然后再仔细看《泛在软件无线电外围设备(USRP)》和《FPGA 中做了些什么》这两部分。本文也提供了两个例子：一个简单拨号音输出和调频接收机，你至少应该理解第一个例子。不能理解“调频接收机”这个例子？没关系，读 Eric Blossom 的第二篇文章《一步一步实现用软件收听调频无线电台》。你不需要一行一行去理解代码，但是你应该了解信号流是怎样从空中到达声卡的。然后，你最好尽可能多的了解 USRP 所进行的处理工作，“USRP 维基”和“USRP 用户向导”这两页将会非常有用。假设你已经阅读了上面的所有文章，你还可以去“GNU Radio 维基”网页上寻找更多的信息。

2.GNU Radio 中的编程…

为了真正地“玩转”GNU Radio，你应该有能力自己写代码。从《开发 GNU Radio》这篇文章中，你应该已经了解到 GNU Radio 的软件架构包含两层，所有的信号处理模块是用 C++写的，Python 用来创建一个网络或图和将这些模块粘合在一起。所以，在这种特殊的场景中，Python 是一种高层语言。GNU Radio 项目提供了许多有用的和经常使用的模块，所以，在许多情形下，你不需要接触 C++，仅仅用 Python 就可以完成你的任务。但是，为了做更复杂的工作，你不得不用 C++去创建你自己的模块。在这种情况下，Eric Blossom 的一篇在线文章《怎样去写一个模块》就是你所需的。你可能想知道哪些模块已经提供给我

们了？不幸的是，不像其它开发工具，如 TinyOS，GNU Radio 在这点上的记录文档很糟糕，但是你还是有两个非常有用的文章，这两个文档是在安装完“gnuradio-core”和”usrp”模块后用 Doxygen 生成的，你可以找到两个 html 包，这两个包位于

/usr/local/share/doc/gnuradio-core-x.xcvs/html/index.html

/usr/local/share/doc/usrp-x.xcvs/html/index.html

我把它们放在了我的浏览器的收藏夹中。尽管这些不是足够的清晰，但是它们可以告诉我们很多信息。第一个文档可以在这里在线获得。

如果你以前没有机会去使用的 Python 的话，请阅读 Python 在线指南。最重要的部分是：

- 第二部分：使用 Python 解释器
- 第三部分：对 Python 的非正式简介
- 第六部分：模块
- 第七部分：输入和输出
- 第九部分：类

这些将在 GNU Radio 的编程中经常使用。如果面向对象编程（OOP）听起来不熟悉的话，你应该更加仔细的读第九部分。下面这些链接也可以帮助教你领会面向对象的本质。

- 课程：面向对象编程的概念
- C++面向对象编程简介
- 面向对象编程网页

不管怎样，在这阶段 Python 看起来是非常重要的，所以必须确保你能很好的掌握它。

3.数字信号处理（DSP）

我想我们中的大部分人都已经上过《信号与系统》这门课，我们在那门课中学到的东西在这儿非常有用。但是，仅仅这些是不够的。如果我们将信号变到模拟域或数字域、变到时域或频域，你必须确保不会迷失。底线是你应该知道什么是抽样定律、什么是 Z 变化、怎样获得信号的频谱和有限冲击响应滤波器（FIR）与无限冲击响应滤波器的概念。我在这儿推荐几本经典的书。

- 《信号与系统（第二版）》——Alan V. Oppenheim, Alan S. Willsky
- 《离散时间信号处理（第二版）》——Alan V. Oppenheim, Ronald W. Schafer, John R.
- 《数字信号处理：原理、算法和应用（第三版）》——John G. Proakis, Dimitris M.

读关于离散时间傅立叶变化和有限冲击响应滤波器（FIR）、无线冲击响应滤波器（IIR）的章节。我知道这些书都非常贵而且乏味，这儿有一些有用的在线资源：

- 数字信号处理指南

- 科技人员的数字信号处理向导

4·通信

我们知道我们发送或接受的真实信号不处于基带，它们需要调制和解调。我认为你已经在一些课程上学习了调幅和跳频无线电的概念，这两个概念都属于模拟世界。为了开发更好和更有吸引力的方法，我们需要数字通信。在这点上，特别重要且有趣的是数字调制与解调、同步，这些知识在你的高年级课程“通信系统”中会讲述。更重要的是，我推荐你读：

- 第四章和第五章，《数字通信（第四版）》——John G· Proakis

这些知识在这两章中介绍得非常准确。这儿还有一本经典的书：

- 《通信系统中的数字信号处理》——Marvin E· Frerking

这本书是实际工程关注的，它包含大量例子。Frerking 对一个给定的发射机或接收机设计问题常常给出多种解决方法，在这本书中实际算法的提出多于纯理论讨论。可以把它当作一本通信系统设计的字典。

5·准备好开始了？

以上我列举了四个主题，这并不意味着在你可以用 GNU Radio 做些事情之前，你必须的把这四个主题一个接一个的学习完。你诚然可以通过修课来学习它们，但是至少你应该读在上面第一部分和第二部分提到的第一篇文章。然后，你可以试着去做这些练习：

在’gnuradio-example’模块中，你可以在’gnuradio-example/python/usrp/’这个文件夹中找到许多示例代码。你可以一行一行地读懂并理解下面两个程序中的代码吗？

- gnuradio-examples/python/usrp/am_rcv·py
- gnuradio-examples/python/usrp/wfm_rcv_gui·py

如果是这样的话，我可以说你已经前进了一大步。

参考

所有本文中提到的书、在线资源。

指南 2：进入 GNU 软件无线电世界

摘要

本文简单介绍的用 GNU Radio 工具包建立软件无线电。这个指南是 Eric 的文章《开发 GNU Radio》的修改版本。本文将会介绍软件无线电的概念和信号处理的一些基础知识。我们将进入那令人激动的 GNU 软件无线电世界。

1· 介绍 GNU 软件无线电

软件无线电是一种让代码尽可能靠近天线的技术，它将无线电硬件问题转换为软件问题。软件无线电的基本特征是软件定义发射波形和软件解调接收波形，这跟大多数无线电系统是相反的，现存的大多数无线电用模拟电路或联合使用数字芯片的模拟电路来完成这些处理。

软件无线电由于具有实现可变无线电和为用户提供更多选择的能力，因此，软件无线电是无线电设计领域的一场革命。最基本的是软件无线电可以做许多传统无线电无法做的事情。软件无线电中令人激动的一部分是软件可以给我们提供灵活性。用一台计算机和一些必须的硬件去玩无线电应该是一件非常简单、有趣和吸引人的事情。

GNU Radio 是一个免费学习、建立和配置软件无线电的软件工具包。GNU Radio 提供一个信号处理模块库和将信号处理模块连在一起的“粘贴剂”。它是免费和开源的，任何人都可以查看完整的源代码和了解系统是怎样建立的。

2·软件无线电系统的架构

2.1 块图

这是一个典型的软件无线电接收流程图：

天线 -> 接收射频端 -> 模数转换器 -> 软件代码

这是一个典型的软件无线电发射流程图：

软件代码 -> 数模转换器 -> 发射射频端 -> 天线

为了理解无线电的软件部分，我们首先了解一点相关的硬件知识。在接收流程图中，我们看到天线、神秘的射频前端、模数转换器和一串代码。

2.2 模数转换器（ADC）

模数转换器是连接物理世界中的连续模拟信号与可被软件处理的离散数字信号世界的桥梁。

模数转换器有两个主要特征参数：抽样率和动态幅度。抽样率是模数转换器在测量模拟信号时每秒的次数。动态幅度是指可以识别的最大信号与最小信号之间的差，它是数模转换器数字输出量位数的函数。例如：一个 8 位的转换器至多可以表示 256 个不同信号，然而一个 16 位的转换器则可以至多表示 65536 个不同信号。一般来说，器件的复杂度、代价和抽样率、动态幅度间存在一个折中。

在经历过模数转换器后，连续信号变成离散序列进入计算机，序列在软件中可以作为一个阵列来被数字化处理。因此，如果我们想“玩”GNU Radio 的话，哪种模数转换器可能是我们的一个好的选择呢？最佳答案应该是 Matt 和 Eric 开发的 USRP 板，随后我们将来讲讲 USRP 板。

2.3 射频前端

为了要理解射频前端所扮演的角色，我们需要讲点理论。乃奎斯特理论告诉我们，模拟信号在转换成数字信号的时候，为了避免混叠，模数转换器的抽样率必须是信号最高频率的两倍，这样才能正确的保存所有谱信息。在西方的老电影中，马车轮看起来向后转就是一种混叠现象。这是由于电影摄像机的抽样速率不够高以至于不能正确的现实轮辐的位置。

假设我们处理的是低通信号，信号的带宽是从 0 到 f_{MAX}，乃奎斯特准则显示我们抽样速率至少是 2*fMAX。但是如果我们的模数转换器的采样速率是 20MHz，我们怎样才能收听 92·1MHz 的调频广播呢？答案就是射频前端。接收射频前端将输入的射频信号转换成输出的中频信号，例如，我们可以可虑一个射频前端将一个频带范围为 90 – 100MHz 的射频信号转换到 0 – 10MHz 的中频信号。

通常我们可以将射频前端视为一个只受中频输出控制的黑盒。一个具体例子，一个有线调制解调器的调谐模块将一个带宽为 6MHz，中心频率位于 50MHz 和 800MHz 之间的信号转换为中心频率为 5·75MHz 的输出信号。输出信号的中心频率叫做中频，或者 IF。

什么可以被用来做射频前端？如果你想用元器件来做一个射频前端，那你应该买一些微电路元件。一个典型的架构如下：

天线 -> 低噪声放大器（LNA）-> 低通滤波器（LPF）-> 混频器 -> 低通滤波器 -> 模数转换器 本地振荡器 ->

低噪声放大器和低通滤波陪用来选择感兴趣的带宽和放大信号。例如，为了接收调调频电台，你可能会用一个低噪声放大器和一个低通滤波陪来截取一个带宽为 120MHz 的信号。混频器可以看作一个两个输入、一个输出的乘法器。本地振荡器用来生成固定频率的正弦波，频率为 RF-IF，你可以用一个压控振荡器来实现这个目的。在混频器的输出端，你可以获得两个信号，它们的中心频率分别是 IF 和 2*RF-IF。因此，混频器接下来应该是低通滤波器，用来移除中心频率为 2*RF-IF 这部分信号，只保留 IF 这部分。最后，中频信号就可以满足乃奎斯特准则进入模数转换器。如果你认为这些微电路部分使你的实验台变得很混乱的话，你也可以买一些集成模块，像 mc4020 有线调制解调模块。如果你使用的是我们推荐的 USRP 的话，一些新的接收子板将会为你带来更多的方便。

3·软件

数字信号最后进入了计算机，等待它们的是我们的代码，也被称作软件。GNU Radio 提供一个信号处理模块库和将这些模块粘贴在一起的“粘贴剂”。我们将在写一章中讨论 GNU Radio 的编程细节，现在我们大概的讲点。

在 GNU Radio 中，编程者通过创建一个图（图论中的概念）来建立无线电，图中的节点是信号处理模块，边代表这些信号处理模块之间的数据流。信号处理模块用 C++实现，理论上，信号处理块处理无限的数据流，这些数据流从输入端流入，输出端流出。信号处理快的属性包括输入、输出端的数目，以及流经这些端口的流的数据类型。最经常使用的类型是短整型、浮点型和复数类型。

有些信号处理模块就只有输出端口或输入端口，这些模块在图中用来做信号源或信号的接收器。信号来自读取文件中的数据或者来自模数转换器，信号接收要么写入文件，要么输入数模转换器或者是图形显示端。GNU Radio 已经写好的模块大概有 100 个，写一个新模块也并不困难。

图是用 Python 构造和运行的。这里有一个 GNU Radio 中的’Hello World’程序，它生成两个正弦波，并将它们输出到声卡，一个左声道，一个右声道。

Hello World Example: Dial Tone Output

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


from gnuradio import gr
from gnuradio import audio
```

```
def build_graph():
    sampling_freq=48000
    ampl=0.1

    fg=gr.flow_graph()
    src0=gr.sig_source_f(sampling_freq, gr.GR_SIN_WAVE, 350, ampl)
    src1=gr.sig_source_f(sampling_freq, gr.GR_SIN_WAVE, 440, ampl)
    dst=audio.sink(sampling_freq)
    fg.connect((src0, 0), (dst, 0))
    fg.connect((src1, 0), (dst, 1))

    return fg

if __name__=='__main__':
    fg=build_graph()
    fg.start
    raw_input('Press Enter to quit: ')
    fg.stop()
```

我们开始用这些模块模块创建流图，通过调用 gr.sig_source_f 生成两个正弦波。后缀 f 表示信号源产生的信号数据类型为浮点型，一个正弦波的频率为 350Hz，另一个正弦波的

频率为 440Hz。合在一起，声音听起来像美国的拨号音。

Audio·sink 是一个将信号输入到声卡的信号接收器模块。输入流的数据类型为浮点型，幅度从-1 到 1。我们用流图中的 connect()方法来连接这三个模块。

Connect()带两个参数：源端和目的端，它的作用是在源端和目的端之间创建一个连接。一端点有两个组成部分：信号处理模块和端口号。端口号是指将被连接的是模块的那个输入或输出端口。最一般的形式，端点用 Python 表示如下：（block，port_number）。当端口号是零时，则这模块就只能被单独使用。

下面两种表示形式是等效的：

```
fg·connect ((src1, 0), (dst, 1))

fg·connect (src1, (dst, 1))
```

一旦图建好后，我们就可以开始运行它了。调用 start()激发一个或多个线程去运行图中所描述的这些计算，并立即将控制权返还给调用者。在这个例子中，我们简单地等待任何击键动作。

如果你还对代码中的某些部分感到困惑的话，请不要担心。我们虽然将进行更加深入的讨论。在这个阶段，理解好典型 GNU Radio 程序的框架就足够了。

在 GNU Radio 中也可以用图形用户接口(GUI)，如软件示波器和软件频谱分析仪，它们是用 wxPython 构造的。这些好的 GUI 工具为 GNU Radio 系统添光增彩，也给我们带来了方便和灵活性。

4·硬件

GNU Radio 独立于硬件，当今日用上 GHz、超大规模的 CPU 拥有单独的浮点单元，这些意味着用台式机处理复杂的数字信号处理成为了可能。一个 3 GHz 的 Pentium 或 Athlon 的 CPU 可以每秒计算 30 亿个浮点的 FIR 系数。我们现在可以用软件来构建通信系统，这在几年之前是不可想象的。

你对计算能力的需求决定于你想做什么。但是，通常来说，一个 1 GHz 或 2GHz，RAM 至少为 256MB 的机器应该足够了。你也应该有些方法去建立模拟世界与计算机之间的联系。低成本的包括用自带的声卡。

关于射频前端和模数/数模转换器，这里有很多选择。但是，泛在软件无线电外围电路（USRP）板是专门为 GNU Radio 设计的，我们强力推荐。事实上，大部分 GNU Radio 玩家都使用 USRP 板。不仅因为它好和使用方便，更重要的是你可以从 GNU Radio 社区获得更多技术支持。因此，为了使你的生活变得简单点，就选用 USRP。在下一篇指南中，我们将研究在 USRP 板中究竟发生了些什么。

5·总结

软件无线电是一个引人入胜的领域，GNU Radio 为开发软件无线电提供了开发工具，GNU Radio 绝对是一个很好的系统。欢迎来到 GNU Radio 的世界。

指南 3：USRP 板

摘要

本文介绍泛在软件无线电外围设备（USRP）板，USRP 板是 GNU Radio 的硬件部分。

1. USRP 板简介

这些天，当我们讨论 GNU Radio 时，泛在软件无线电外围设备（USRP）板已经成为一个必不可少的硬件组成部分。它是 Matt 完全为 GNU Radio 用户而开发的。基本上，USRP 是一个由包含 AD/DA 转换器、一些射频前端和一个 FPGA 组成，其中 FPGA 主要是做一些非常重要的但是计算量又非常大的输入信号的预处理工作。USRP 板便宜而且又高速，是 GNU Radio 用户实现一些实施应用的最好选择。我们可以从 Ettus 公司购买到 USRP 板，一块 USRP 板由一块母板和最多四块子板组成，母板的价格是 *450* 美元，每块子板的价格是 *50* 美元。图 *1* 显示的是插入四块子板（*2* 块接收子板和 *2* 块发射子板）USRP 板。

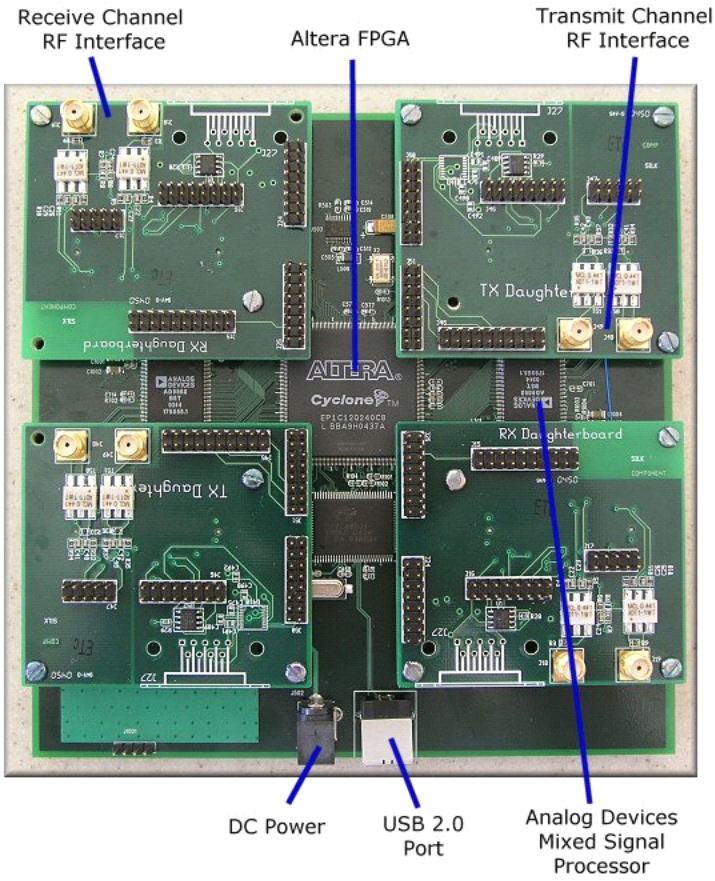

图 1：USRP 板：一块母板，两块接收/两块发射子板

2·USRP 板的“数据表单”

这部分介绍 USRP 板的硬件组成部分。需要强调的是硬件组成部分的特点是非常重要的。它们将影响广泛地影响你的无线电的设计和软件编程。你不得不遵从由于硬件而带来的约束，记住一些约束参数将会非常有用。所以，请仔细阅读这部分。

USRP 板的典型设置由一块母板和最多四块子板组成，如图 2 所示。

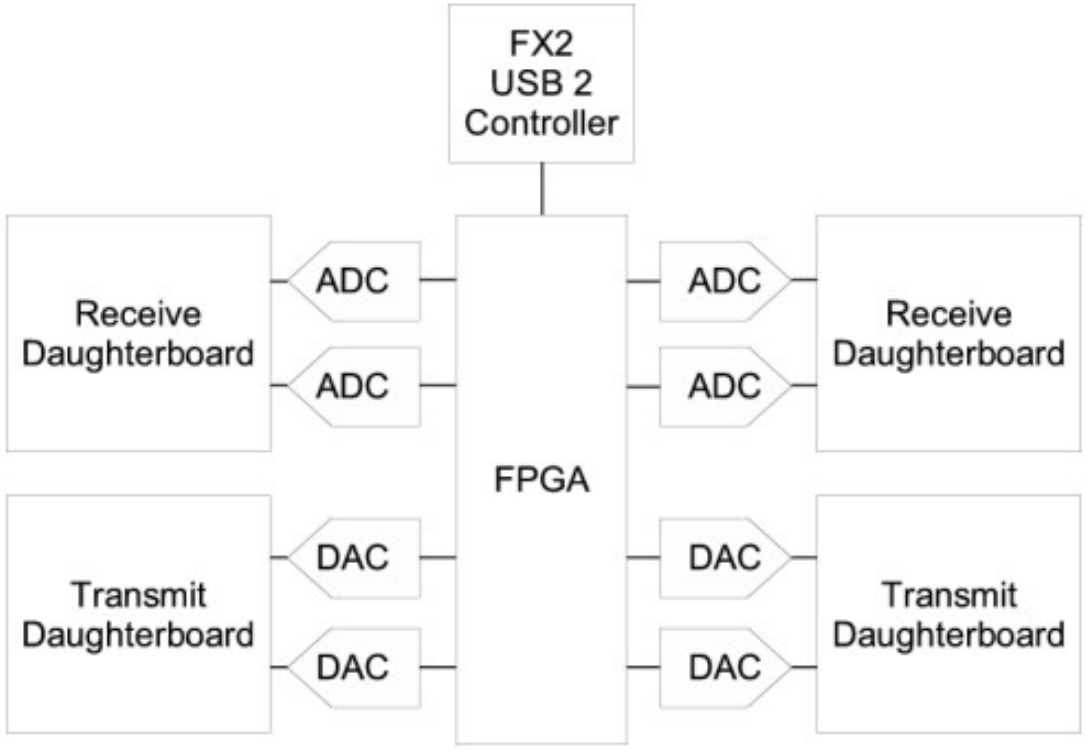

图 2：USRP 板

在母板上，我们可以看到直流输入电源和 USB2·0 端口。眼下不支持 USB1·x，所以必须确保你的 PC 机有 USB2·0 接口。

2.1 AD/DA 转换器

USRP 板上有四个高速的 12 位 AD 转换器。抽样速率是 64M 符号/秒。理论上它可以数字化 32MHz 的带宽。模数转换器可以带通抽样最高 150MHz 的信号。如果我们抽样一

个信号的中频高于 32MHz 的话，那将会带来混叠现象，实际上我们通常将所感兴趣的信号映射到-32MHz～32MHz 这个范围之内。有时这些是有用的，例如，我们可以在没有任何射频前端的情况下收听调频电台。抽样信号的频率越高，由于误差带来的 SNR 性能下降越严重。100 MHz 是推荐的最高限。

模数转换器的幅度峰峰值是 2V，输入的差分电阻是 50 欧姆，也就是 40mw 或者 16dbm。模数转换器后面有一个可编程的增益放大器，用来放大输入信号，达到全部利用模数转换器输入信号幅度范围的目的，以防输入信号非常微弱。可编程增益放大器的放大能力高达 20dB。

注意，如果我们想的话，我们也可以使用其它抽样速率，所有可用的抽样速率都是 128MHz 的约数，如 64MS/s，42·66MS/s，32MS/s，25·6MS/s 和 21·33MS/s。

在发射处理流程中，有 4 个高速 14 位数模转换器。数模转换器的时钟频率是 128MS/s，所以乃奎斯特频率是 64MHz。但是，为了使滤波器实现简单，我们通常希望其乃奎斯特频率小于或等于 50MHz。因此，有用的输出频率范围为直流到 50MHz。数模转换器输出信号的幅度峰峰值可达 1V，差分负载为 50 欧姆或者 10mw（10dbm）。数模转换器之后也有一个增益放大器，放大倍数高度 20dB。注意，接收处理流程中的增益放大器和发射处理流程中的增益放大器均是可以编程的。

因此理论上我们将有 4 条实抽样输入通路和 4 条是抽样输出通路。但是，如果我们用复抽样（IQ）的话，我们将会有更大的灵活性。如果是复抽样的话，我们就需要将它们配对使用，所以，我们就只能获得 2 条复抽样输入通路和 2 条复抽样输出通路。

2.2 子板

母板上有四个插槽，你可以在这四个插槽中插入两块接收子板和两块发射子板。子板被用做射频接收机或射频发射机。

母板上有两个卡槽用来插发射子板，标记为 TXA 和 TXB，有两个卡槽用来插接收子板，标记为 RXA 和 RXB。每块子板与两个高速 AD/DA 转换器相连（模数转换器用在发射处理流程中的输出，数模转换器用在接收处理流程中的输入）。这使得每个子板在采用事抽样时，

相当于两个独立的射频部分和 2 根天线。如果采用的是复抽样的话，每块子板只能支持一个射频部分，那整个系统一共有两个射频部分。

我们可以看到每块子板上有两个 SMA 转接头。我们常规地使用它们去连接输入或输出信号。

当前有几种类型的子板可用：

基本子板：它没有什么特别的，两个 SMA 转接头用来连接外部调谐器或信号发生器。我们可以将它视为信号的入口或出口，还需要一些额外的射频前端。

电视接收子板：该子板内嵌微调谐 4937 有线调制解调器。它只是一个接收子板，射频频率范围从 50MHz 到 800MHz，中频频率为 6MHz，如果你的无线电应用是在这个范围之内，如调频电台或者电视接收，那你所需要的仅仅是一根天线。

DBSRX 子板：与电视接收子板相似，它也只能接收。射频频率范围从 800MHz 到 2·4GHz。

一些新的子板正在开发之中，特别是包括一些发射子板。它们很快就有销售，将给我们带来很大的方便。

2.3 FPGA

对 GNU Radio 用户来说，也许理解 FPGA 中进行的处理是最重要的部分。如图 2 所示，所有的模数转换器和数模转换器都被连接到 FPGA 上。FPGA 在 GNU Radio 系统中扮演着关键角色。

大体上 FPGA 完成的工作包括高速的数学计算和为了适应 USB2·0 端口速率而进行的降低数据速率的操作。FPGA 连接到一个 USB2·0 端口芯片 Cypress FX2。加载在 USB2·0 总线上的所有东西（FPGA 电路和 USB 微控制器）都是可以编程的。

我们标准的 FPGA 配置包括用级联积分梳滤波器（CIC）实现的数字下变频转换器（DDC）。CIC 滤波器是非常高性能的滤波器，它只由加法器和延时器构成。FPGA 实现四个数字下变频转换器（DDC）。这使得可以分别产生 1，2，4 条接收通路。在接收流程中，有 4 个模数转换器和 4 个数字下变频转换器。每个数字下变频转换器有两路输入，分别是同相支路（I）和正交支路（Q）。4 个模数转换器中的任意一个都可以被路由到 4 个数字下变频转换器中的任何一个 I 路或 Q 路输入信号上。这就使得对于同一个 ADC 抽样流的输出可以有多个通路选择。

在发射侧的数字上变频转换器（DUC）实际上被包含在 AD9862 CODEC 芯片里，不

在 FPGA 中。发射流程中信号在 FPGA 中的唯一处理模块是上采样。上采样的输出可以被路由到 4 个 CODEC 输入中的任何一个。

多接收通路（1，2 或者 4）必须具有相同的数据速率（也就是相同的抽值速率）。对于发射通路也有相同的要求。

图 3 所示的是 USRP 接收流程的块图和数字下变频转换器图。

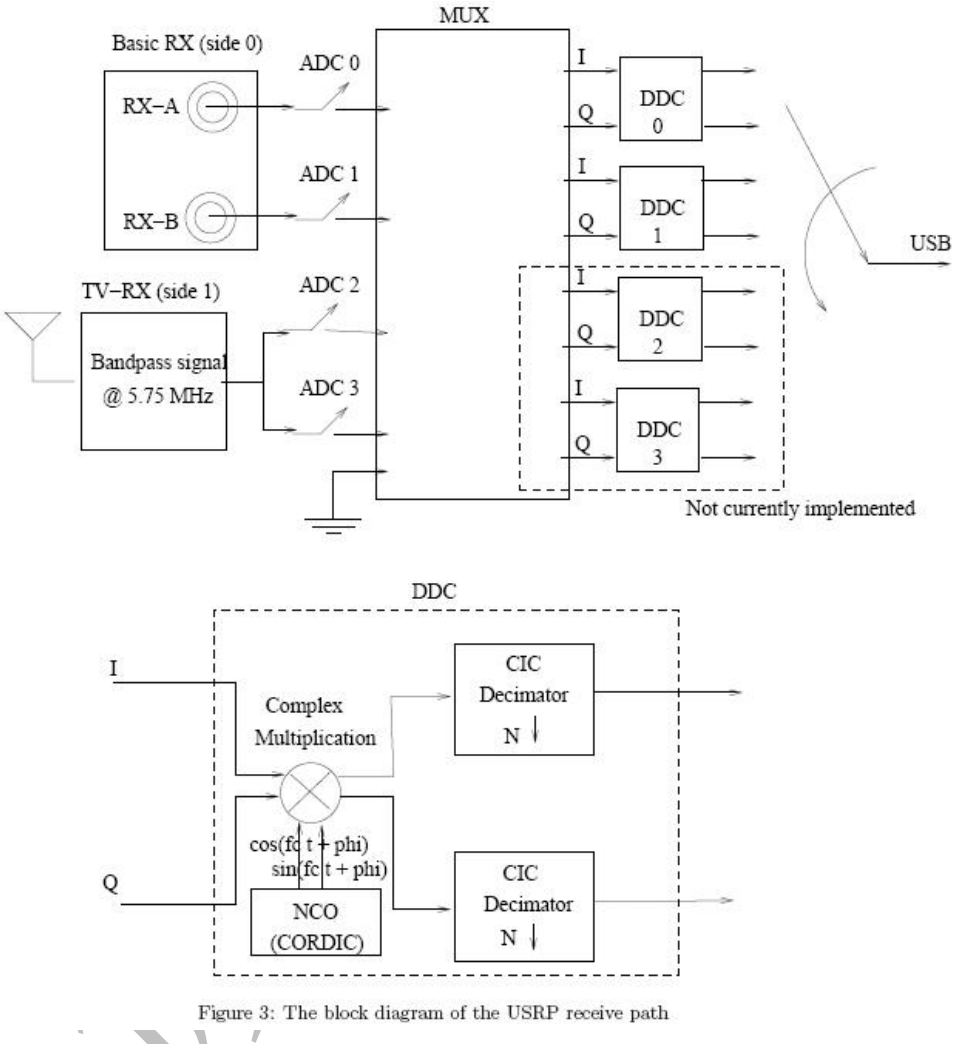

Figure 3: The block diagram of the USRP receive path

图 3：USRP 接收流程中的模块图

复用器（MUX）像是一个路由器或者一个电路交换机。它决定哪个模数转换器（ADC）与哪个数字下变频转换器（DDC）的输入相连。FPGA 中有四个数字下变频转换器（DDC），每个有两个输入。我们可以在 Python 中用 usrp·set_mux()方法来控制复用。我们将在下一章中讨论有关 GNU Radio 的编程问题，但是现在值得提前看一下。我们用 4 位来表示 ADC 被连接到哪个输入（I0，Q0，I1 … I3，Q3），因此，一个 32 位的整数完全能够表示 ADC

是连接到8个输入中的具体哪一个。

指南 4：通过逐行阅读 FM 接收代码来为 GNU Radio 中的 Python 做准备—第一部分

摘要

Python 在 GNU Radio 编程中扮演了一个关键角色。GNU Radio 为构造软件无线电提供了一种架构。这种信号处理应用，通过高层组织、策略、GUI 的 Python 代码和其他非关键性能的函数构造形成，同时性能的关键信号处理模块使用 C++编写。以 Python 的视角来看，GNU Radio 提供了一种数据流抽象。本文主要关注 Python 层，介绍了 Python 的基本使用以及 Python 是如何粘合所有的信号处理块(block)并控制数字数据流的。这里使用了一个通俗且经典的例子：实现一个带有 GUI 的 FM 接收器。这里对代码进行了逐行分析。同时，亦介绍了 Python 的基本语法和软件无线电的概念。所以本文可以同时看作一个简短的 Python 指南和软件无线电指导。如何使用 C++编写模块，以及其他高级话题将会在接下来的章节中涉及。

1. 回顾

GNU Radio 的软件组织采用两层结构。所有关键性能的信号处理块使用 C++编写，而更高层次的组织、连接和粘合由 Python 完成。很多经常使用的信号处理块(block)已经作为 GNU Radio 的一部分提供给了我们。

这种结构与 OSI 七层网络结构有一定的相似性。底层向高层提供服务，高层只关心必需接口和函数调用，而不关心底层的具体执行细节。在 GNU Radio 中，这种层的透传也以一种相似的方式存在。从 Python 的视角来看，它(Python)所做的仅仅是选择信号源、信号接收端和信号处理块，设置正确的参数，然后把它们连接起来以形成一个完整的应用。事实上，所有的这些信号源、接收端和块都是以 C++类的形式实现的。参数设置、连接操作与 C++中一些复杂的函数或类方法相对应。然而，Python 却看不到 C++工作的如何勤勉。一段冗长的、复杂的、强大的 C++代码在 Python 看来仅仅是一个接口(interface)而已。

结果就是，不管应用是如何的复杂，Python 代码都是一如既往的短小简洁。真正沉重的负担已经扔给 C++了。一条经验法则需要牢记：对于任何应用，我们在 Python 层所需要

做的都仅仅是，先在脑子里画一个信号从源端流动到目的端的图，然后使用“可爱的笔”—Python，来找到并连接它们，有时还带有 GUI 支持。

显然，在 GNU Radio 学习中 Python 至关重要。Python 是一种有力而又灵活的编程语言，它本身也有一段很长的故事。但是如果你拥有足够的 C/C++背景，这不过是小菜一碟。

考虑到 Python 应用在 GNU Radio 中时，它本身的一些特殊特性和花哨特征都不是必需的，本文着力于把 Python 编程技术和软件无线电概念相结合。我始终相信若要理解一些新知识的本质，就要通过实例学习而不是枯燥的学习句法或语义。所以我们的讨论将会紧紧地围绕一个通俗且经典的例子：实现一个带有 GUI 的 FM 接收器。在此，我们将会逐行分析代码，同时也会讨论 Python 编程、信号处理技术、软件无线电概念和一些硬件配置。

本例实现了一个带有图形用户接口的FM接收器。空中的FM信号通过USRP板接收，然后USRP板和计算机对其进行处理。最后解调的信号通过声卡播放出来。不要求有特殊的天线。你仅仅通过把一根铜线插入basic RX子板就可以听到非常高质量的FM信号。代码可以在‘gnuradio-examples/python/usrp/wfm_rcv_gui.py‘找到。请对照附录A比对来确保我们看到的是同一版本的代码。更深一步的话题，如GUI工具(wxPython)和一些Python内建包(built-in packages)的使用，我们将会在指南 8 中讨论。

2.第一行

如果你还阅读了其他例子的代码，那么你会发现这些程序的第一行往往都是：

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

如果我们在脚本的开头输入这一行，同时给予该文件可执行的权限，那么该 Python script 就能够直接执行。文件的开头必需是“#！”这两个字符。使用“chmod”命令可以使 script 获得可执行权限：

```
$ chmod +x wfm_rcv_gui.py
```

现在 wfm_rcv_gui.py 已经有了可执行权限。你可以使用如下指令在 shell 中执行程序：

```
$ ./wfm_rcv_gui.py arguments
```

这时 Python 解释器将会被调用，而代码则会顺序地逐行执行。Python 是像 Matlab 一样的解释性语言。不需要编译和连接。

这里还有几种方法来调用 Python 解释器：你可以使用

```
$ python ./wfm_rcv_gui.py arguments
```

而无须赋予它可执行权限。

你也可以使用交互模式，通过在 shell 中输入命令：

```
$ python
```

然后 Python 解释器将会调用，你可以一行一行的输入你的代码。但这显然一点儿也不方便。

我们很少交互式使用解释器，除非我们写一些一次性(throw-away)程序、测试函数或是把它用作桌面计算器。多数情况下，使我们更便捷的是在.py 文件里写代码，并且使其自执行(self-executable)。

3.导入必要模块

接下来，我们会看到很多导入的东西：

```
from gnuradio import gr, eng_notation
from gnuradio import audio
from gnuradio import usrp
from gnuradio import blks
from gnuradio.eng_option import eng_option
from optparse import OptionParser
import sys
import math
from gnuradio.wxgui import stdgui, fftsink
import wx
```

理解这些语句需要 Python 中“模块”(module)和“包”(package)的概念。学习它们最好的途径是浏览 "http://www.python.org/doc/current/tut/node8.html" the Chapter 6 of the Python tutorials.。

这里做一个简要介绍。如果我们退出 Python 解释器再重新进入，那么我们已经定义的所有函数和变量都会丢失。因此，我们希望写一个稍长的包含函数和变量定义甚至包含可执行语句的程序，然后把它保存为一个“脚本”(script)。这个脚本可以用作 Python 解释器的输入。我们可能还想使用一些已经写在一些程序中的特殊的函数，同时还不想把它们的定义复制到每个程序中去。

为了支持这些，Python 提供了模块/包(module/package)的组织结构。一个模块是一个包含 Python 定义和声明的以后缀“.py”结尾的文件。模块中，模块的名字（作为一个字符串）是全局变量“__name__”的值。模块中的定义可以被导入到其他模块或顶层模块。一个包则是那些有相似功能并且通常被放置于相同目录下的模块的集合。Python 把那些目录当做包处理时需要“__init__.py”文件。一个 package 可以包含,modules 和 sub-packages(可以包含 sub-sub-package)。我们使用“.模块名”来构造 Python 的模块命

名空间(namespace)。比如，模块名 A·B 指明名为 A 的包中的子模块 B。

一个模块可以包含可执行声明(statment)和函数定义。这些声明在该模块第一次导入某处时执行，然后该模块得以初始化。每个模块都有其私有符号表(symbol table)，而私有符号表在定义在该模块里的函数中是作为全局符号表来使用的。模块的作者(author)可以使用模块中的全局变量，而无需担心与模块用户的全局变量发生意外冲突。作为模块的使用者，我们可以通过使用"模块名·条目名(itemname)"来获取模块的函数和全局变量。这里，"条目名"既可以是函数，也可以是变量。

模块可以通过使用"import"命令来导入其他模块。习惯上，所有的"import"语句都会放在模块的最开始部分。既然导入操作非常的灵活，(那么)我们可以导入包、模块，或仅仅是模块中的定义。当我们从一个包中导出一个模块的时候，我们既可以使用"import packageA·moduleB'"，也可以使用"from package A import module B"。当我们使用"from package import item"时，item 可以是一个包中的模块／附属包，也可以是定义在包中的其他名字，比如函数、类或变量。

花一番时间来介绍这个例子中使用的模块是值得的，因为这些模块或包我们在 GNU Radio 中将会频繁地遇到。GNU Radio 顶层的包是"gnuradio"，它包含有所有 GNU Radio 相关的模块。它的位置在

/usr/local/lib/python2.4/site-packages

默认情况下，这个目录是不包含于 Python 的寻找路径里的，我们需要把这个路径导出到环境变量"PATHONPATH"中。所有我们通常把下面这一行添加到用户的"·bash_profile"文件中·

$export PATHONPATH=/usr/local/lib/python2.4/site-packages

这样就可以确保 Python 解释器可以找到 gnuradio 包·

gr 是 gnuradio 一个重要的子包(sub-package)，是 gnuradio 软件的核心。'flow graph'类的类型定义在 gr 中，并且对调度信号流发挥关键作用。eng_notation 是一个专为工程师记法(notation)便捷而设计的一个模块，其中的很多字和字符都根据工程惯例赋予了新的常数值。audio 模块提供了通向声卡的接口，usrp 提供控制 USRP 板子的接口。audio 和 usrp 通常被用作信号源端和宿端。我们在本文的后面会看到有关它们的细节。blks 是一个子包，

如果你查看它的目录的话，会发现它的文件夹几乎是空的。它基本上把它所有的任务都交给了另一个 gnuradio 中的子包 blksimpl，就像在__init__·py 文件中描述的那样。blksimpl 提供了一些有用的应用的实现，比如 FM receiver, GMSK,等等。在这个例子中，FM receiver 真正的信号处理部分就在这个包中完成。

我们来看下一行，这行更加有趣：

from gnuradio.eng_option import eng_option

这就是刚刚提到过的，我们既可以导入一个完整的模块/子包，或仅仅是这个模块中的函数、类以及变量定义。在这个例子中，eng_option 是定义在 gnuradio·eng_option 模块中的一个类。我们需要的不是这个完整的类来导入，而仅仅是单个的一个类定义。gnuradio·eng_option 模块做的不过是为 optparse·OptionParser 添加对工程记法(engineering notation)的支持。

这一行似乎有相似的格式：

from gnuradio.wxgui import stdgui, fftsink

但是其含义有一点点不同，gnuradio·wxgui 是一个子包，不是一个模块，而 stdgui 和 fftsink 则是这个子包的两个模块。由于不需要导入整个子包，所以只要把我们明确需要的导入就可以了。gnuradio·wxgui 为 GNU Radio 提供了可视化工具，它是基于 wxPython 的。Python 中的导入操作为我们提供了极大的灵活和便利。

最后，optparse、 math、 sys 和 wx 都是 Python 或 wxPython 的内建模块或子包，而不是 GNU Radio 的一部分。

这时候，我们再强调一遍，上面这些导入的模块可以包含可执行声明(statements)和函数或类定义。这些声明在该模块被导入后立即执行。在导入这些模块和包之后，定义在其中的很多的变量、类和模块被初始化。所以不要认为什么都没做。事实上进行了很多幕后工作。很多家伙(guys)在工作区中等待着你的命令。

OK！到目前为止我们已经快速浏览了 GNU Radio 中经常使用的模块，也看到了它们是如何组织在一起的。也许这些听起来太简略，你仍然对它们的使用感到困惑。没关系，我们很快会再看到它们。

4·wfm_rx_graph 类的故事

熟悉面向对象编程(OOP)对理解这部分内容很重要。面向对象编程本身是一个很长的故事，我们现在不去关注它。但我们同时会讨论一些 OOP 的概念。如果我们遇到相关的代码，一些数字信号处理技术也将会一并讨论。

4.1 类定义

在这个例子中，代码的很大一部分都是‘wfm_rx_graph’类的定义。声明

class wfm_rx_graph (stdgui.gui_flow_graph):

定义了一个新的类‘wfm_rx_graph’，它是从基类或是说父类----‘gui_flow_graph’继承过来的。父类 gui_flow_graph 是定义在我们刚刚从 gnuradio 中导入的 stdgui 模块中的。根据命名空间的规则，我们提及它的时候使用 stdgui·gui_flow_graph。

4.2 'FLOW GRAPH'类的家族

‘flow graph’类是类的一个非常重要的种类，它在 GNU Radio 中扮演着关键的角色，应该引起我们特别的注意。定义在 GNU Radio 中还有一系列与‘GRAPH’相关的类。我们可以发现 stdgui·gui_flow_graph 继承自 gr·flow_graph，而后者定义在子包 gr 中。更进一步，gr·flow_graph 又是从根类(root class)gr·basic_flow_class 继承来的。在 GNU Radio 中，还有很多类继承自 gr·basic_flow_graph。这个庞大的 GRAPH 家族使得 GNU Radio 编程整洁和简单，同时也使得信号处理的时序清晰易懂。

这些’graphs’到底是做什么的？假设你正尝试用一些 Pspice 这样的商业软件来设计电路。也许你会先打开一个原理图，或是一个“画布(canvas)”，然后再在这张画布上放上所有必需的电路组成部分，如电阻，放大器或是一些直流电源。最后，你会把这些部分用线连接起来，完成电路设计。这个场景可以很好的映射到 GNU Radio 中。一个’graph’就像是原理图或者画布。电路元件被 GNU Radio 中的信号源端，信号宿端和信号处理块所代替。最后，这些线--对应于 graph 类中的’connect’，负责把这些块连接在一起。可以肯定的是，有时一个完整的电路在其他原理图中是作为一部分的，叫做子电路(sub-circuit)。这在 GNU Radio 中也是一样的，一个子图(sub-graph)在其它图中可以用作一个完整的块(block)。

在我们的例子中，wfm_rx_graph 就是一个属于这个家族的带有 GUI 支持的图类(graph class)。稍后我们会使用’connect’来把 FM 接收器中的块粘在一起。

4.3 初始化：__init__

接下来我们实现 wfm_rx_graph 类的'__init__'方法(method)/函数(function)。定义一个新的方法的语法是

def funcname(arg1 arg2 ...)

__init__对任何类来说都是一个重要的方法。在类的定义之后，我们可能会使用这个类来实例化一个例子。__init__这个特殊的方法就是用来在已知的初始状态创建一个对象。类的例示(class instantiation)会为新创建的类的实例自动调用__init__。其实在这个例子中，__init__是定义在 wfm_rx_graph 类中的唯一的一个方法。

在讨论__init__函数的细节之前，我们需要关注 Python 的一个重要特性。我们注意到在这段代码中，不存在明确的标识来说明类或函数的定义在哪里开始和结束。而通常在其他如 C++或 Pasca1 编程语言中，我们会使用一对'begin'和'end'或是一对'{'和'}'来明确的表征声明的两端。然而在 Python 中，这种情况就不再存在了。这里没有这样的标识。在 Python 中，声明的分组(grouping)是由缩进来完成的，而不是开始和结束的括弧。所以当你用 Python 编程的时候，要对你代码的编辑和定位(layout)非常小心。

现在我们来看看__init__函数中到底发生了什么。

def __init__(self,frame,panel,vbox,argv):

声明了初始化函数__init__带有四个参数。照惯例，所有方法的第一个参数都叫做'self'。这仅仅是一个惯例而已：self 这个名字对 Python 绝对没有任何特殊的含义。然而，方法可能会使用 self 参数的方法属性(method attributes)来调用其他方法，如我们稍后会遇到的'self·connect()'。

__init__所作的第一件事，就是调用方法 stdgui·gui_flow_graph 的初始化，它的父函数，带有同样的四个参数。

stdgui.gui_flow_graph.__init__ (self,frame,panel,vbox,argv)

你可能想要看一眼 stdgui·gui_flow_graph 的初始化方法。由于 wfm_rx_graph 继承自它，我们可以有把握的把 wfm_rx_graph 当做一个特殊的 gui_flow_graph。毕竟这个子类(son class)应该做些像它老爸的事情，然后再做些特别的使他自己有些不同。

4.4 构建一个带有信源、信宿和信号处理块的图(graph)

4.4.1 定义的部分

从下一行起，我们就开始看到真实的信号了，也就少了一些枯燥

```
IF_freq = parseargs(argv[1:])
```

这一行的意思是把 parseargs 函数的返回值设置为 IF 频率。

IF 频率是什么？IF 是'intermediate frequency'的缩写。粗略的说，它是我们感兴趣频段的中间频率。我们把通常非常高的真实的频段，即 RF 频段，转换到 ADC 可以依照奈奎斯特准则工作的中频。这不是本文的关键点。我希望大家在这一点上已经有足够的通信和 DSP 背景。

parseargs 函数定义在本例的后面，就在 wfm_rx_graph 类定义之后。当程序在 shell 中运行时，它接收用户的输入参数。我们稍后会谈到它。

另外，你可能已经注意到了，Python 不需要进行变量或参数的声明。这和'使用前要声明'的 C 有着非常大的不同。

```
adc_rate = 64e6
```

这一行定义了 AD 转换器的采样率，对于 USRP 用户来说，这个值应该被设为 64M 赫兹。根据奈奎斯特准则，目标信号所含的最大频率不能超过 32M 赫兹，以防在采样后丢失频谱信息。

```
decim = 250
quad_rate = adc_rate / decim            # 256 kHz
```

抽取(decimation)是 DSP 中的概念。在对模拟信号采样后，我们得到一个数字信号，它非常高的速率对于 CPU 和存储器来说都是很大的负担。一般，我们可以对数字序列降低采样(抽取)而不会丢失频谱信息。在这个例子中，抽取率是 250，因此最后的数据速率是 256Ksamples/second，这个速率对我们的 CPU 来说就非常合理。quad_rate 表示正交数据速率(quadrature data rate)。为什么称其为正交速率我们稍后会解释。

256 kHz'是注释。在 Python 中，注释跟在'#'符号后面。每行的在'#'后面的语句都会被 Python 解释器忽略。

```
audio_decimation = 8
audio_rate = quad_rate / audio_decimation          # 32 kHz
```

在对数字 FM 信号处理之后，我们希望能够通过计算机的声卡把声音播放出来。然而，声卡可以接受的数据速率非常有限。一般来说 256K 过于高了，并且毫无必要。因此我们需要进一步降低数据速率。对于大多数的声卡来说，32K 是一个不错的选择。

4.4.2 信号源

下面几行是对信号处理过程的高度提炼(abstraction)，基本上包含三个部分：信号源、信宿，和一系列的信号处理模块。这个例子给了这些信号处理模块一个非常好的名字：guts。本例中 FM 接收器的信号源是 USRP

```
 # usrp is data source
src = usrp.source_c (0, decim)
src.set_rx_freq (0, IF_freq)
src.set_pga(0,20)
```

USRP 板子从空中通过 RX 子板接收到模拟的 FM 信号，并且使用采样率为 64M 的 AD 转换器对信号进行采样。然后得到的数字序列进入了 USRP 板子上的 FPGA 芯片中。在这里根据用户设定的抽取率（本例中是 250）进行下采样。另一个重要的数字信号处理过程也是在 FPGA 中完成的—实射频信号转换为带有 I/Q 两路正交部分的复数基带信号。这也是为什么把抽取后的数据速率称作正交速率的原因。当然，这后面的事情要复杂很多，我们把细节放在之后的章节里来讲。最后，复基带信号通过计算机上的 USB2.0 口被送至软件模块。复数通过实部/虚部来表示，事实上是需要两个实数值的。

OK，继续看代码。USRP 模块是我们一开始就导入的。它的位置在：

/usr/local/lib/python2.4/site-packages/gnuradio/usrp.py

它告诉我们对于 USRP 信宿(发送)和信源(接收)，USRP 是已经封装(wrapper)好的。当一个信源或信宿 instantiated，usrp 模块会先通过 USB 端口来获得所需板子的编号。source_c 是定义在 usrp 模块中的函数。它返回一个表示数据源的类对象。后缀-c 表示信号的数据类型是'complex',原因就是进入计算机的信号是复数(实际上是以实部/虚部的形式)。相应的，在 usrp 模块中，还有方法 source_s，用以 16 位短整型数据类型。

例子中，source_c 有两个参数。'0',指定打开哪个 usrp 板子。如果我们只使用唯一一块 USRP 板子的话，这个值就设为 0。第二个参数是把抽取率告诉 USRP 板。set_rx_freq 和

set_pga 是 src 源的两个方法。set_rx_freq 把中频告诉 USRP 板。刚才我们提到过，USRP 板把实中频信号转化为带 I/Q 两路正交部分的复基带信号。为了做到这一点， USRP 板需要知道中频。pga 是`Programmable Gain Amplifier'的缩写。我们可以通过方法 set_pga 来设置它的值(单位是 db)，我们例子中是 20db。

这些方法是定义在哪的？在这个层面上，Python 是不足以解释这些东西的。事实上，所有这些方法都是使用 C++实现的。SWIG 提供了 C++和 Python 间的接口，因此我们可以直接在 Python 中调用这些函数，而无需关注 C++的细节实现。Boost，一个智能指针系统，也在这里使用了，以使得 C++和 Python 的交互更加容易。我们在这里跳过这些实现的细节。让我们在 Python 中快乐的使用这些方法！

一个用 Doxygen 产生的关于 USRP 的重要文档位于：

/usr/local/share/doc/usrp-x.xcvs/html

在这个文档里，我们可以所有的由 USRP 提供的方法。USRP 的顶层接口是 usrp_standard_rx 和 usrp_standard_tx。我们也应该看一眼它们的基类，usrp_basic_rx, usrp_basic_tx 和 usrp_basic。这里还有很多其他的方法，用以 USRP 板的控制和交互。如果你还在担心 Python 和 C++之间的接口问题，那现在松一口气吧，我们将会在讨论到后面的 how to use C++ to write blocks 时再提及。

4.4.3 核心信号处理块'gut'

guts = blks.wfm_rcv (self, quad_rate, audio_decimation)

在它背后有很长的一段故事。

'guts'是这个 FM 接收器的核心处理块。所有的信号处理块，如去加重、非相干解调都在这个'gut'中粘合在一起了。这个非常有意思的故事可以在下面这个文件中找到：

/usr/local/lib/python2.4/site-packages/gnuradio/blksimpl/wfm_rcv.py

FM 接收技术的细节我们将会在指南 7 中讨论。现在我们仅从高层的观点看一眼。wfm_rcv 是一个定义在模块 blksimpl 中的类。它的基类是 hier_block，定义在另一个模块 gr·hier_block 中。gr·hier_block 可以被视作一个子图(sub-graph)，包含一些信号处理块，而这些块在另一个更大的图中用作一个复杂的块。在这个声明(statement)中，我们创建了

一个对象'gut'作为 wfm_rcv 的实例。所有实(real)的信号都在这个大块中处理。

4.4.4 信号源

最后，我们要使用声卡来播放解调的 FM 信号。所有这个例子中音频设备室信号接收端：

```
# sound card as final sink
audio_sink = audio.sink (int (audio_rate))
```

sink 是一个定义在 audio 模块中的全局函数。它返回一个对象作为信号接收块。audio_rate 是描述进入声卡信号的数据速率的参数，在我们这个例子是 32K 赫兹。

4.4.5 把它们粘在一起

下面的两行最后完成了我们这个信号流图：

```
# now wire it all together
self.connect (src, guts)
self.connect (guts, (audio_sink, 0))
```

刚才我们讨论了流图类的家族，新的类 wfm_rx_graph 也属于这里。所有的流图类都继承自'根'类 gr.basic_flow_graph。connect 是定义在 gr.basic_flow_graph 中的方法。这个方法是用来在流图中把所有的块绑定在一起。我们将会在指南 6 中探索更多的流图类和它们的方法。

此时此刻，这个信号流图就大功告成了！

5 结论

OK！我们停在这里休息一下。剩下的代码需要深一些的知识才能理解。我们将会为 FM 接收器添加非常酷而且有吸引力的 GUI 支持。它是基于 gr-wxgui 建立的，而后者又是基于 wxPython 和 FFTW 的。我们还没有讨论如何从用户界面接收参数，以及如何开始流图。在这时，一些通过类和函数传递的参数也许看起来仍然很困惑。

本文主要聚焦于 Python 的基本语法，和 Python 在 GNU Radio 中如何扮演它的角色。同时，也提及了一些软件无线电的概念和信号处理技术。我希望在阅读这篇文章后，大家能够对如何在 GNU Radio 中编程能有一个粗略的认识。

附录：源代码

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

```
from gnuradio import gr, eng_notation
from gnuradio import audio
from gnuradio import usrp
from gnuradio import blks
from gnuradio.eng_option import eng_option
from optparse import OptionParser
import sys
import math

from gnuradio.wxgui import stdgui, fftsink
import wx


class wfm_rx_graph (stdgui.gui_flow_graph):
   def __init__(self,frame,panel,vbox,argv):
      stdgui.gui_flow_graph.__init__ (self,frame,panel,vbox,argv)

      IF_freq = parseargs(argv[1:])
      adc_rate = 64e6

      decim = 250
      quad_rate = adc_rate / decim            # 256 kHz
      audio_decimation = 8
      audio_rate = quad_rate / audio_decimation  # 32 kHz

      # usrp is data source
      src = usrp.source_c (0, decim)
      src.set_rx_freq (0, IF_freq)
      src.set_pga(0,20)

      guts = blks.wfm_rcv (self, quad_rate, audio_decimation)

      # sound card as final sink
      audio_sink = audio.sink (int (audio_rate))

      # now wire it all together
```

```
        self.connect (src, guts)
        self.connect (guts, (audio_sink, 0))

        if 1:
            pre_demod, fft_win1 = \
                    fftsink.make_fft_sink_c (self, panel, "Pre-Demodulation",
                                             512, quad_rate)
            self.connect (src, pre_demod)
            vbox.Add (fft_win1, 1, wx.EXPAND)

        if 1:
            post_deemph, fft_win3 = \
                     fftsink.make_fft_sink_f (self, panel, "With Deemph",
                                              512, quad_rate, -60, 20)
            self.connect (guts.deemph, post_deemph)
            vbox.Add (fft_win3, 1, wx.EXPAND)

        if 1:
            post_filt, fft_win4 = \
                    fftsink.make_fft_sink_f (self, panel, "Post Filter",
                                             512, audio_rate, -60, 20)
            self.connect (guts.audio_filter, post_filt)
            vbox.Add (fft_win4, 1, wx.EXPAND)

def parseargs (args):
    nargs = len (args)
    if nargs == 1:
        freq1 = float (args[0]) * 1e6
    else:
        sys.stderr.write ('usage: wfm_rcv freq1\n')
        sys.exit (1)

    return freq1 - 128e6

if __name__ == '__main__':
    app = stdgui.stdapp (wfm_rx_graph, "WFM RX")
    app.MainLoop ()
```

参考文献

[1] **Python on-line tutorials**, http://www.python.org/doc/current/tut/
[2] Eric Blossom, **Exploring GNURadio**,
http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html

指南 5：图、块和连接

摘要

在指南 5 中，我们已经看到流图家族为高层的信号处理调度和组织提供了极大地便利性和灵活性。基本上，一个流图可以容纳几个信号源、接收端和信号处理块，然后把它们连接在一起来形成一个完整的应用。在本文中，我们将会在更深的层次中针对更多的微妙之处(subtleties)进行探索。

1. 回顾

在指南 5 中，我们已经看到流图家族为高层的信号处理调度和组织提供了极大地便利性和灵活性。基本上，一个流图可以容纳几个信号源、接收端和信号处理块，然后把它们连接在一起来形成一个完整的应用。在本文中，我们将会在更深的层次中针对更多的微妙之处(subtleties)进行探索。

另一个例子，"dial tone"用来说明上述情况。这是个非常简单的"Hello World!"式的例子。但它也足够用来描绘 GNU Radio 中图机制的强健和优美。在这个例子中，我们简单的产生两个不同频的正弦波，然后通过声卡把它的音调播放出来。只需要信源和信宿，而实信号处理时不需要的。

这个例子的源码位于'gnuradio-examples/python/audio/dial_tone.py'。请参照附录 A 以确保我们的代码是同一版本。运行这个例子不需要 USRP 板，需要的是你计算机上的声卡。在你的终端里简单的输入$./dial_tone.py，你就可以从喇叭中听到清晰的声音。

一些 Python 编程技巧与此同时也会提及。

2. 定义函数 build_graph()

gr 和 audio 两个模块先被导入。然后定义了不带任何参数的函数 build_graph()：

```
def build_graph ():
```

build_graph()是 dial_tone.py 模块的一个全局函数，与我们上次提到的__init__()只有细微的不同，后者是属于 wfm_rx_graph 类的方法。

接下来两个定义了常量。提醒一下：不要忘了缩进！

sampling_freq = 32000

ampl = 0.1

在上个指南中我们讨论音频抽取时，已经见过了 sampling_freq。它是进入声卡的数字信号的数据速率。对于我们用的大多数的声卡来说，32K 赫兹是一个不错的选择。采样率是我们稍后就会看到的信号源 gr·sig_source_f 的一个参数。另一个参数就是正弦波的幅度，ampl。在这个例子里我们把它设为 0·1v。

3· 创建一个图

图来了！

fg = gr.flow_graph ()

flow_graph 是定义在 gr·flow_graph·py 模块中的类。

这里有个小技巧。为什么我们可以直接用 gr·flow_graph 引用(refer to)这个类，而不是用 gr·flow_graph·flow_graph？答案就在 gr 包的初始化文件`__init__·py'中。我们来看看它的内容：

from gnuradio_swig_python import *

from basic_flow_graph import *

from flow_graph import *

from exceptions import *

from hier_block import *

我们知道每个目录(derectory)必须包含一个__init__·py 文件使 Python 能够把它作为一个包处理。当一个包被导入到某处时__init__·py 中的语句(statement)就会立即执行。本例中，当 gr 被导入时，flow_graph 的所有定义同时也被导出到 gr 包的全局符号(symbol)表里。这里''*'表示''所有东西'。所以我们可以直接使用''gr·item'而不是'gr·flow_graph·item'来获取定义在 flow_graph·py 中的类或函数。

Flow_graph 继承自它的父类 basic_flow_graph，所有'图'相关类的根(root)。basic_flow_graph 概念上描述了块之间的连接。让我们来看一眼它的源文件，位于：

/usr/local/python2.4/site-packages/gnuradio/gr/basic_flow_graph.py

按常理，图应该包括顶点，以及连接这些顶点的边。这两个基本元素在 basic_flow_graph·py

中被定义为 endpoint 类和 edge 类。
endpoint 的部分定义是这样的：

```
class endpoint (object):
  __slots__ = ['block', 'port']
  def __init__ (self, block, port):
    self.block = block
    self.port = port
```

在 Python 中，’__slot__’是类的一种特殊的变量。默认的，类的所有实例都有一个’’字典’来存储属性：变量’__dict__’,它的数据类型是’’Dictionary’。这种机制使得拥有非常少变量的对象浪费了一些空间。在创建大量实例时这种空间浪费问题变得非常严重。__slots__的声明获取(take)一系列的实例变量，并且在每个实例中保存仅够的空间，用以为每个变量保留值。由于不是所有的实例都创建__dict__，所有空间节省了下来。

所以从__slot__中，我们可以看到 endpoint 类有两个属性：block 和 port。这个两元组可以在图中指定一个端点(顶点)。block 可以是一个源、接收或信号处理块。一个 block 可以有多个端口，对应图中的不同顶点。

下面是 edge 类的部分定义：

```
class edge (object):
  __slots__ = ['src', 'dst']
  def __init__ (self, src_endpoint, dst_endpoint):
    self.src = src_endpoint
    self.dst = dst_endpoint
```

edge 提供了两个端点之间的直连。显然，edge 有两个属性：src 和 dst，表示信号流从何处开始到何处结束。src 和 dst 的数据类型就是刚刚我们定义的’endpoint’。

基于 endpoint 和 edge，我们定义了 basic_flow_graph 类。部分定义：

```
class basic_flow_graph (object):
  '''basic_flow_graph — describe connections between blocks'''
  __slots__ = ['edge_list']
  def __init__ (self):
    self.edge_list = []
```

basic_flow_graph 只有一个属性’’edge_list’，edge_list 是一个保存图中所有边的列表(list)。当一个流图实例建立的时候，这个列表初始化为空。只有当我们调用定义在 basic_flow_graph 类里的方法的时候，比如’’connect’’’或’disconnect’，这个列表才会修改。我们稍后再讨论这些方法。

我们例子中的 flow_graph 是继承自 basic_flow_graph。定义在：

/usr/local/python2.4/site-packages/gnuradio/gr/flow_graph.py

下面是它部分的定义：

```
class flow_graph (basic_flow_graph):
    """add physical connection info to simple_flow_graph
    """
    __slots__ = ['blocks', 'scheduler']

    def __init__ (self):
        basic_flow_graph.__init__ (self);
        self.blocks = None
        self.scheduler = None
```

注释解释了所有。与它的父类 basic_flow_graph 相比，flow_graph 添加了到它的物理连接。basic_flow_graph 仅仅简要的定义了流图，保存(preserve)了图中使用哪种''端点'以及端点中的哪些是通过'边'直连的这样的信息。你可以把它想做是画在你的草稿纸上的图表，在真实世界里没有发生。当然你也不可以运行这个图。在 flow_graph 中，我们'物理地'连接了这些块，比如，所有块的输入都连接到了它们的上游缓存—计算机内存中的真实缓存。为了彻底理解'物理连接'，我们来看下列定义在 flow_graph 类中的方法：

```
def _setup_connections (self):
    """given the basic flow graph, setup all the physical connections"""
    self.validate ()
    self.blocks = self.all_blocks ()
    self._assign_details ()
    self._assign_buffers ()
    self._connect_inputs ()

def _assign_details (self):
    ...
def _assign_buffers (self):
    """determine the buffer sizes to use, allocate them and attach to detail"""
    ...
def _connect_inputs (self):
    """connect all block inputs to appropriate upstream buffers"""
    ...
```

当一个图实例调用它的 start()方法来运行程序的时候，这个图首先会调用它的 _setup_connections()来建立所有物理连接。它实际上是顺序调用了三个方法：_assign_details()、_assign_buffers()和_connect_inputs()。这三个方法完成了物理层的实际连接。也分配和计算了实际需要的缓存大小。最后，所有块的输入连接到了相应的上游缓存。这些方法的实现细节对于我们来说就过于琐碎了，而且我们进行 GNU Radio 编程时很少会用到。我们使用图的时候也从来不明确调用这些方法。然而，我相信知道一些背后的故事会帮助大家对 GNU Radio 的工作机制有一个更好的理解。

flow_graph 又添加了两个属性：'blocks'和'scheduler'。block 是一个列表(list)变量，保存图中存在的所有块的列表。它使用定义在 basic_flow_graph 中的 all_blocks()方法来得到列表。相同的块但是不同端口的端点只会计算一次。scheduler 是一个控制流图运行的重要变量。它的数据类型是 scheduler 类，定义在模块 gr·scheduler 中。gr·scheduler 使用 Python 的内建模块 threading，来控制图的''开始'、'停止'或'等待'操作。

flow_graph 添加到 basic_flow_graph 的最有区别的部分是控制流图进行的类方法，如 start(),stop(),wait(),run()等待。从名字上就可以看出来，这些方法是用来实际控制信号流在图中的移动或停止。在这些类方法中，scheduler 扮演了一个关键角色。

我们可以想象到，大部分我们想创建图实例或派生一个图类的时候，我们应该使用 flow_graph，而不是 basic_flow_graph。

OK！单单从 fg = gr·flow_graph ()这一行我们就讲了这么多故事。愿你们不会忘记，我们创建了一个图！这是 flow_graph 类的一个实例。

最后再说一下，在上面的介绍中，我们提到了一些 Python 的新的数据类型，如'List''tuple'和'Dictionary'。它们确实也是为什么 Python 这么有力、灵活又有效的部分原因。不幸的是，我们不能在这个指南中讲述所有的这些细节。请参阅 Python 指南来得到更多的它们的信息。

4· 信源和信宿

4.1 信源

我们继续。下面我们产生两个数据源。在上一个指南中，我们已经看到了一种源的类型:usrp。它把 USRP 板当做输入端。在这个例子中，源就更加简单了：计算机产生的正弦波：

src0 = gr.sig_source_f (sampling_freq, gr.GR_SIN_WAVE, 350, ampl)

src1 = gr.sig_source_f (sampling_freq, gr.GR_SIN_WAVE, 440, ampl)

严格来讲，这个源也属于'''blocks'的范畴。就像以前提到的，所有的块都是由 C++实现的。SWIG 提供 Python 和 C++之间的接口，因此你在 Python 中可以直接使用由 C++实现的类定义或函数。我们将会在指南 10 中讨论如何用 C++来写块，以及如何用 SWIG 来创立接口。那将会是理解整件事的最好时机。现在，我们就忘掉那些恼人的技巧(trick)，在 Python 中安全的使用这些块吧。

学习 GNU Radio 中绑定和实现的块的最佳途径就是浏览由 Doxygen 产生的 GNU Radio 文档。GNU Radio 安装之后，它的位置应该在：

/usr/local/share/doc/gnuradio-core-2.5cvs/html/index.html
点击这里也可以。

点击`Class Hierarchy'标签，然后查找 gr_sig_source_f 类。你找到后，打开它的类参照，你可以看到 gr_sig_source_f 的层次(hierarchy)关系。'根'类是 gr_block。所有的信号处理块都是继承自 gr_block。gr_sync_block 是一个继承自 gr_block 的重要的类。它实现了具有可选函数关系(history)和做了一定简化的 1:1 块。信源和信宿，如 gr_sig_source_f，都是继承自 gr_sync_block。它们之间唯一的不同就是源没有输入而宿没有输出。所有的这些块都是由 gr 前缀开头的，并且是由 C++实现的。当我们使用 SWIG 提供 C++和 Python 之间的接口时，背后有一些神奇魔术，使我们能够通过在 Python 中的 gr·sig_source_f()连接到 gr_sig_source_f。这里的后缀 f，表示我们这个例子中的源是 float 数据类型。我们讨论使用 C++来写块的时候，所有的这些神奇魔术都会再次遇到，所以现在你感到困惑的话不用担心。

无论如何，结果就是，我们已经通过 gr·sig_source_f()的返回值创建了两个源的实例。它们可以由我们刚才创建出的图画出来！

4.2 信宿

我们在 FM 接收器的例子中就见过这个接收端：音频设备。我们要使用声卡播放产生的音乐。

dst = audio.sink (sampling_freq)

音频模块的位置是：

/usr/local/lib/python2.4/site-packages/gnuradio/audio.py

音频模块跟 usrp 非常相似。它对音频源和接收端进行了封装(wrapper)。再强调一遍，这背后有些精巧的魔术，以致我们在 Python 中可以直接使用 audio·sink()来创建音频接收端。'sink()'和'source()'都是定义在 audio 模块中的函数，它们都连接到了 OSS 或 ALSA 来为声卡提供支持，前提是你已经安装了 gr-audio-oss 或 gr-audio-alsa 模块。这两个类函数会返回一个类实例作为音频接收端或源端。

从你的观点来看，背后的事情可能很复杂，但是在 Python 中建立一个音频接收端真的很容易。我们已经完成了！就把那些恼人的细节忽略掉吧。

5·连接

最后，我们把已经定义的块连接在一起来完成我们的流图：

```
fg.connect (src0, (dst, 0))
fg.connect (src1, (dst, 1))
```

connect()类函数的使用非常简单。但还有几点需要注意。我们再来看一下 basic_flow_graph 类的定义。这是方法 connect()的定义：

```
def connect (self, *points):
    '''connect requires two or more arguments that can be coerced to
    endpoints. If more than two arguments are provided, they are
    connected together successively.
    '''
    if len (points) < 2:
        raise ValueError, ("connect requires at least two endpoints;
        %d provided." % (len (points),))
    for i in range (1, len (points)):
        self._connect (points[i-1], points[i])
```

point 参数前面的'*'是 Python 中函数的一个特点。表示这个函数可以带任意数量的参数被调用。这些参数会被包装在一个组里。在可变数量的参数前面，可以出现 0 或者更多的正常的参数。所以 connect()可以接受 2 个以上的参数。如果两个以上的块作为 connect()的参数，它们会被依次连接，这种代码十分清晰。

另外隐藏的一点是，如果我们连接点(points)，这些点会先被‘强制(coerced)’转换成‘端点’实例。把参数首先改做为与 endpoint 类一致是很必要的。我们来看看

coerce_endpoint()方法的定义：

```
def coerce_endpoint (x):
    if isinstance (x, endpoint):
        return x
    elif isinstance (x, types.TupleType) and len (x) == 2:
        return endpoint (x[0], x[1])
    elif hasattr (x, 'block'):          # assume it's a block
        return endpoint (x, 0)
    elif isinstance(x, hier_block.hier_block_base):
        return endpoint (x, 0)
    else:
        raise ValueError, "Not coercible to endpoint: %s" % (x,)
```

显然我们不需要为 connect()方法提供一个完整的 endpoint 实例（事实上这样做还会引起代码的混乱）。我们可以简单的提供一个 2 元祖来作为参数，例如(块名，端口号)。coerce_endpoint()方法会把它转换成一个 endpoint 实例。就像我们的例子中，我们引用端点时使用(dst,0)，(dst,1)。如果我们只提供了一个块实例而没有给出端口号的话，coerce_endpoint()方法将会添加 0 作为默认的端口号，并且返回一个 endpoint 实例，如同我们例子中的 src。它就相当于(src，0)。对于只有一个端口的块，直接使用块名作为 connect()的参数是十分方便的。

注意到上面提到的这些，这时，图仅仅是逻辑的创建和连接了，并不是物理的。我们在草稿纸上刚刚有了一个简图(diagram)而已。

最后，全局函数 build_graph()返回了我们刚刚创建的流图实例 fg。

6. 运行程序

到了运行这个流图的时候了。

```
if __name__ == '__main__':
    fg = build_graph ()
    fg.start ()
    raw_input ('Press Enter to quit: ')
    fg.stop ()
```

第一行是什么意思？每个模块都有一个’__name__’属性，表示当前模块的名字。很多模块都保护一段这样的代码：

```
if __name__ == '__main__':
    The testing code
    ...
```

用来测试的。当一个模块被导出到某处时，’__name__’将会在模块的命名空间中建立，来保存模块文件的名字。所以当一个模块导出时，测试代码将会被解释器忽略，因为模块的名字__name__永远都不可能是’’__main__’。然而，当我们把这个模块作为一个可执行文件直接运行的时候，如·/dial_tone·py，或者使用 python dial_tone·py 来运行脚本，不是导出，那么模块的命名空间是一个全局命名空间，Python 会自动的把全局命名空间的__name__设为__main__。这种情况下，测试的代码就会生效。

fg = build_graph()调用我们刚刚定义的函数并且把返回的流图实例设为 fg。下一行使流图开始运行：fg·start()。`start()‘和`stop()‘的定义添加在 flow_graph 类中：

```
def start (self):
    '''start graph, forking thread(s), return immediately'''
    if self.scheduler:
        raise RuntimeError, "Scheduler already running"
    self._setup_connections ()

    # cast down to gr_module_sptr
    # t = [x.block () for x in self.topological_sort (self.blocks)]
    self.scheduler = scheduler (self)
    self.scheduler.start ()


def stop (self):
    '''tells scheduler to stop and waits for it to happen'''

    if self.scheduler:
        self.scheduler.stop ()
        self.scheduler = None
```

当我们开始一个图时，start()做的第一件事，就是通过调用 self·_setup_connections()方法来“物理的”连接图，这个方法我在上面讨论过。然后它会创建一个调度器(scheduler)的实例，并且用这个调度器来控制图的开始和停止。我们在上面已经讨论过 scheduler 了。

它在这里扮演一个关键角色。但是它的工作原理对我们来说可能过于琐碎了。我们应该对在 Python 中使用 start()和 stop()控制流图的运行感到安全。

raw_input()是 Python 的内建模块，就像 print。它读取用户经标准输入设备的输入。例如 password = raw_input("Please input your password:")，将会在屏幕上打印出"please input your password:"，然后等待用户的输入。然后输入的字符串会保存在'password'变量中。在我们的例子中，程序等待用户的输入来停止运行。否则它会永远运行下去。

7. 结论

在本文中，我们对一个非常简单的例子进行了非常多的评论。我们分析了为何流图的机制能够为 GNU Radio 编程带来极大的便利性和灵活性。也探索了一些场景背后的故事来表现结构(mechanism)是怎么组织到一起的。

本文涉及的一些资料对于我们在 GNU Radio 的 Python 级编程来说可能是不必须的。但它们对我们理解整个框架应该是有帮助的。

附录：源码

```
from gnuradio import gr
from gnuradio import audio

def build_graph ():
    sampling_freq = 32000
    ampl = 0.1

    fg = gr.flow_graph ()
    src0 = gr.sig_source_f (sampling_freq, gr.GR_SIN_WAVE, 350, ampl)
    src1 = gr.sig_source_f (sampling_freq, gr.GR_SIN_WAVE, 440, ampl)
    dst = audio.sink (sampling_freq)
    fg.connect (src0, (dst, 0))
    fg.connect (src1, (dst, 1))

    return fg

if __name__ == '__main__':
    fg = build_graph ()
```

```
fg.start ()
raw_input ('Press Enter to quit: ')
fg.stop ()
```

参考资料

*[1]***Python on-line tutorials**, http://www.python.org/doc/current/tut/

*[2]*Eric Blossom, **Exploring GNU Radio**,

http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html

指南 6：探索 FM 接收器

摘要

在指南 5 中，我们跳过了 FM 信号如何解调的讨论，把'guts'当做了一个黑盒子。本文中，介绍了 FM 信号解调的实信号处理技术。我们将会打开黑盒子，看看软件的世界中信号时如何处理的。

1· 回顾

在之前的指南中，我们已经介绍 GNU Radio 的硬件建立，和一些编程的技巧，这些形成了“软世界”中真实的应用的基础。本文中，我们将会探索广播 FM 信号是如何解调的。FM 接收器是 GNU Radio 中典型的例子。你只要用一小段电线就可以听到很强的电台。在指南 5 中，我们跳过了黑盒子'guts'的介绍，那是 FM 信号探测真正神奇的地方。在本文中我们将要展示信号从空中到声卡是如何处理的。

2·从空中到计算机，从实信号到复信号

在指南 3 和 4 中，我们已经讨论了 USRP 的操作，尤其是数字下变频器 DDC 的角色。基本上，USRP 做的就是选择我们需要的频段，然后通过系数 N 来抽取数字序列。得到的是带有 I/Q 两路的复信号(gr_complex)。所以在我们写完了这些行以后

```
src = usrp.source_c (0, decim)    # decim = 250, so data rate (quad_rate) is 256kHz
src.set_rx_freq (0, IF_freq)      # IF_freq = our input - 128MHz
src.set_pga(0,20)
```

我们会得到一个“复”信号，其数据速率是 256kS/s。我们称其为“正交速率”- quad_rate 因为复信号有 I/Q 正交部分。

我们选择的中频在这里是一个非常有趣而且有用的点。IF_freq 等于用户的输入减去 128M 赫兹，就像 parseargs()函数中表示的：

```
return freq1 - 128e6
```

写 wfm_rcv_gui 时假设这里没有 RF 前端来降低频率。A/D 采样率是 64M。因此奈奎斯特区域是：

[0, 32M] normal

[32M, 64M] inverted

[64M, 96M] normal

[96M, 128M] inverted

问题就是 96M 是一个交叠点，发生在广播频段的中间。这就意味着 95.0 和 97.0 都混叠了到一个频率并且不能区分出来。如果 freq1>=96M,那么 freq1 - 128M 将会给出一个有效的频率。底线是：它是一个 kludge 式的工作。（kludge：笨拙地用不相配的硬件和软件来给问题提供尚可的解决）

3.获得瞬时频率，从复信号到实信号

到了挖掘'guts'核心的时候了，我们在指南 5 中把它当做了一个黑盒子。

guts = blks.wfm_rcv (self, quad_rate, audio_decimation)

blks 是 gnuradio 中的一个包。它除了与另一个包 blksimpl 关联几乎什么也不做。wfm_rcv 是一个定义在/gnuradio/blksimpl/wfm_rcv.py 中的类，是 FM 接收器中一个真实的“处理器”。源码添加到本文的最后了。

wfm_rcv 继承自 gr.hier_block。gr.hier_block 描述了流图中一系列块的串联(tandem)。它假设了链条的开头和结尾都至多有一个单独(single)块。头或者尾都可以是空的，表示只有一个信源或信宿。hier_block 可以被识做一个包含几个相连块的子图。为了构建 hier_block，我们需要指定流图，这个流图包含信号处理链条中这种分级的、开始和最后的块。分级的块可以作为普通的块处理，也就是可以在流图中放置(place)和连接，就像下面这些行论述的：

self.connect (src, guts)

self.connect (guts, (audio_sink, 0))

在链条中的“头”块是 fm_demod，它是 gr.quadrature_demod_cf 的实例。为了理解其中的实际的工作，我们应该对 FM 信号如何产生有一定的了解。作为输入信号的函数，带有 FM 的传输波形的瞬时频率时刻在变。

任意时刻的瞬时频率由下面这个公式给出：

f(t) = k * m(t) + fc

m(t)是输入信号，k 控制频率灵敏度的常数，fc 是载波频率(比如 100.1MHz)。所以为了恢复 m(t)，需要两步。第一步我们需要移除载频 fc，这样我们就有了一个带有与原始消

息 m(t)成比例的瞬时频率的基带信号。第二步显然是计算基带信号的瞬时频率。这样，我们的挑战就是找到一种方法来移除载频以及计算瞬时频率。在指南 3 和 4 中介绍过了，通过 FPGA 中的 DDC 可以移除载频。在前一部分，我们也解释了为什么需要调频到(fc – 128MHz)。所以进入到'guts'的信号就已经是基带信号了，我们剩下的任务就是计算瞬时频率。如果我们对频率积分(integrate)，那么会得到相位(phase)或相角(angle)。相对的，对相位的时间积分也会得到频率。这就是我们创建接收器的关键点。

我们使用 gr·quadrature_demod_cf 来计算基带信号的瞬时频率。我们通过确定相邻样点之间的相角来近似区分相位。这时调用 DDC 来输出复数。多使用一点三角学，我们就能确定两个相邻样点之间的相角，主要是通过用另一个的复共轭来乘以这一个，然后对其再取反正切(arc tangent)。只要你弄明白你要做什么，代码是不用编写太多的。gr_quadrature_demod_cf·cc 包含了这个块的 C++实现。我们将在指南 10 和 11 中详细讨论如何用 C++来编写信号处理块。不过现在看看代码也是有用的。大量的信号处理是在 sync_work()函数的循环中完成的。

```
Part of gr_quadrature_demod_cf.cc
...
int
gr_quadrature_demod_cf::sync_work (
   int noutput_items,
   gr_vector_const_void_star &input_items,
   gr_vector_void_star &output_items)
{
  gr_complex *in = (gr_complex *) input_items[0];
  float *out = (float *) output_items[0];
  in++;        // ensure that in[-1] is valid
  for (int i = 0; i < noutput_items; i++){
    gr_complex product = in[i] * conj (in[i-1]);
    out[i] = d_gain * arg (product);
  }
  return noutput_items;
}
```

一个 FM 接收器有用的图表可以在这里找到。

fm_demod_gain = quad_rate/(2*math.pi*max_dev)

注意 fm_demod_gain 可用用作控制音量的常数。它实际的值无关紧要。这里 arg (product)就是相邻样点间的相位不同，如果我们用采样间隔来除它，比如，乘以数据速率 quad_rate,我们可以得到角频率ω，如果我们再用 2π除ω，那么我们就可以得到瞬时频率 f。最后我们使用 max_dev 来把频率标准化。

4·去加重

链条中的第二个块就是去加重 deemph，fm_deemph 类的一个实例。fm_deemph 定义在 fm_emph·py 中，同样位于 blksimpl 包中。

什么是去加重？我们简要介绍一下。理论上已经证明，在 FM 探测中，输出噪声的功率是随着频率平方(quadratically)增长的。然而，对于大部分实际的信号，比如人类的声音和音乐，信号的功率随着频率增长而大幅下降。结果就是，高频端的信噪比通常是不可忍受的。为了规避这种效应，人们在 FM 系统中引入了“预加重”和“去加重”。发送端，我们使用恰当的预加重电路来人工的加大高频部分，并且在接收端进行相反的操作来恢复原来的信号功率分布。因此，我们有效的提高了信噪比。

在模拟的世界里，一个简单的一阶 RLC 电路通常就足够进行预加重和去加重。这里是一个有关它们传输函数的漂亮的图。在我们的数字信号处理中，一个一阶 IIR 滤波器是一个不错的选择。

Part of fm_emph.py

```
...

#         1
# H(s) = -------
#        1 + s
#
# tau is the RC time constant.
# critical frequency: w_p = 1/tau
#
# We prewarp and use the bilinear z-transform to get our IIR coefficients.
# See "Digital Signal Processing: A Practical Approach" by Ifeachor and Jervis
#
class fm_deemph(gr.hier_block):
    """
    FM Deemphasis IIR filter.
    """
```

```
def __init__(self, fg, fs, tau=75e-6):
    """
    @type fs: float
    @param tau: Time constant in seconds (75us in US, 50us in EUR)
    @type tau: float
    """
    w_p = 1/tau
    w_pp = math.tan (w_p / (fs * 2)) # prewarped analog freq

    a1 = (w_pp - 1)/(w_pp + 1)
    b0 = w_pp/(1 + w_pp)
    b1 = b0

    btaps = [b0, b1]
    ataps = [1, a1]

    deemph = gr.iir_filter_ffd(btaps, ataps)
    gr.hier_block.__init__(self, fg, deemph, deemph)
```

我们把模拟滤波器的传输函数作为原型，并使用双线性变换来得到数字 IIR 滤波器。注意到 gr·iir_filter_ffd 就是带有 float 输入、float 输出和 double 抽头(taps)的 IIR 滤波器。

5·音频 FIR 抽取滤波器

在通过去加重后，现在的信号是什么样子？首先，它是一个数据率为 256KHz 的实信号。第二，它是一个基带信号，有效的频率范围是 0 到 100kHz，包含了 FM 电台的所有频率分量。

补充说明一下，FM 电台的的带宽通常在 2*100kHz 左右。这也解释了为什么选择 256kHz 作为正交速率（USRP 的抽取率选为 250）。256kHz 的采样率对于 200kHz 的带宽是刚好合适的，不会丢失任何频谱信息。可能你已经注意到了，在 FM 接收机中，我们从不使用任何低通滤波操作来“挑捡出”我们的目标 FM 电台。实际上这个工作已经由 USRP 的 DDC 暗中完成了。DDC 可以当作是一个低通 FIR 滤波器，紧跟一个下采样器。因此，目标电台已经挑选出来，然后抽取后扩展到整个数字频域。因为我们选择了正确的抽取率，所以我们终于完成了很多工作！底线是，我们是在信号进入 USRP 后操作在单一的 FM 电台信号上。

现在我们需要解决两个问题。首先，信号速率已经是 256kHz，比声卡能接受的高出很

多。计算机的声卡通常采样的上限是 96,000Hz。第二，100kHz 的频谱包含几个通道，L+R，L-R，导频音，等待。为使事情简单一些，我们仅设计一个只能接收 L+R 信号的单声道接收机。所以很明显，我们想要的正是一个 FIR 抽取滤波器。

为了更清楚，这里是简要介绍一下调频信号的频带。从 0 到 16kHz 是左加右(L+R)音频。19kHz 的峰值是立体声导频音。左减右(L-R)立体声信息以 2 倍导频(38kHz)为中点，并且它是 FM 上的 AM 调制。其他的子载波有时会在 57kHz – 96kHz 区域中。我们可以使用 GNU Radio 的带有 GUI 支持的内建 fft 块来看解调后的频谱（指南 8 介绍了细节）。这里是一个由 Eric 给出的不错的图。这里是 FM 波段一个很好的插图。

OK，我们现在来设计 FIR 抽取滤波器。GNU Radio 的 gr·fir_filter_fff 块给了我们一个带有 float 输入、float 输出和 float 抽头的 FIR 滤波器。它的构造函数有两个参数：第一个是抽取因子，第二个是滤波器系数(抽头)向量。

```
self.audio_filter = gr.fir_filter_fff (audio_decimation, audio_coeffs)
```

如果我们需要 FIR 滤波器又不想改变数据速率，那么我们把抽取率简单的设为 1 就好了。如果我们需要的是一个插值滤波器而不是抽取滤波器，那么应该选择 GNU Radio 的 gr·interp_fir_filter_xxx 块。

使用 FIR 滤波器设计 gr·firdes 块可以获得滤波器系数 audio_coeffs。low_pass()是一个定义在 gr_firdes 类中的静态公共函数。相似的，它也有 high_pass(), band_pass(), band_reject()函数。我们使用这些函数来设计 FIR 滤波器抽头，提供滤波器参数和详细说明。例如，设计一个低通滤波器的语法是：

```
vector< float > gr_firdes::low_pass ( double      gain,
                                      double      sampling_freq,
                                      double      cutoff_freq,
                                      double      transition_width,
                                      win_type    window = WIN_HAMMING,
                                      double      beta = 6.76
                                    )  [static]
```

每个参数的意义十分明显。注意到 beta 是 Kaiser 窗的参数。在我们的例子中，选择音频抽取因子(audio_decimation)为 8，所以得到的进入声卡的数据速率为 32kHz。我们只

对从 0 到 16kHz 的 L+R 音频感兴趣，因此我们对带有 16kHz 截止频率的正交解调器低通。这给了我们一个连接到声卡的单声道输出。在我们的例子中，我们选择截止频率为 15kHz，传输带宽为 1kHz，这是合理的。

```
width_of_transition_band = audio_rate / 32
audio_coeffs = gr.firdes.low_pass (volume,        # gain                (20)
                                   quad_rate,     # sampling rate       (32kHz)
                                   audio_rate/2 - width_of_transition_band, (15kHz)
                                   width_of_transition_band,           (16kHz)
                                   gr.firdes.WIN_HAMMING)
```

OK！我们的 FM 接收机完成了！声卡上的信号已经可以播放了。注意到在数字信号处理中，FIR 滤波器的使用，和多速率处理一样是非常重要的。

最后，我们把这些块连起来，并调用 gr·hier_block 的__init__()方法来完成 wfm_rcv 类的__init__()方法。这里我们需要指出通道(pipeline)的头和尾。

```
fg.connect (self.fm_demod, self.deemph, self.audio_filter)
gr.hier_block.__init__(self,
                       fg,
                       self.fm_demod,      # head of the pipeline
                       self.audio_filter)  # tail of the pipeline
```

6·结论

本文中，我们介绍了 FM 探测技术以及它们在 GNU Radio 中是如何实现的。现在我们可以看出 GNU Radio 真是一个不错的系统，给我们提供了这么多有用的工具和控制应用的灵活方法。在下个指南中，我们将要总结处理 wfm_rcv_gui·py 的解说，尤其会强调 GNU Radio 的 GUI 工具。

附录 A：源码

```
from gnuradio import gr
from gnuradio.blksimpl.fm_emph import fm_deemph
import math
```

```
class wfm_rcv(gr.hier_block):
    def __init__ (self, fg, quad_rate, audio_decimation):
        """
        Hierarchical block for demodulating a broadcast FM signal.

        The input is the downconverted complex baseband signal (gr_complex).
        The output is the demodulated audio (float).

        @param fg: flow graph.
        @type fg: flow graph
        @param quad_rate: input sample rate of complex baseband input.
        @type quad_rate: float
        @param audio_decimation: how much to decimate quad_rate to get to aud
io.
        @type audio_decimation: integer
        """
        volume = 20.

        max_dev = 75e3
        fm_demod_gain = quad_rate/(2*math.pi*max_dev)
        audio_rate = quad_rate / audio_decimation


        # We assign to self so that outsiders can grab the demodulator
        # if they need to.  E.g., to plot its output.
        #
        # input: complex; output: float
        self.fm_demod = gr.quadrature_demod_cf (fm_demod_gain)

        # input: float; output: float
        self.deemph = fm_deemph (fg, quad_rate)

        # compute FIR filter taps for audio filter
        width_of_transition_band = audio_rate / 32
        audio_coeffs = gr.firdes.low_pass (volume,            # gain
                                           quad_rate,         # sampling rate
                                           audio_rate/2 - width_of_transition_
band,
```

```
                                        width_of_transition_band,
                                        gr.firdes.WIN_HAMMING)
    # input: float; output: float
    self.audio_filter = gr.fir_filter_fff (audio_decimation, audio_coeffs)

    fg.connect (self.fm_demod, self.deemph, self.audio_filter)

    gr.hier_block.__init__(self,
                           fg,
                           self.fm_demod,        # head of the pipeline
                           self.audio_filter)    # tail of the pipeline
```

参考文献

[1] Eric Blossom, **Listening to FM Radio in Software, Step by Step**,

http://www.linuxjournal.com/article/7505

指南 7：通过逐行阅读 FM 接收代码来为 GNU Radio 中的 Python 做准备—第二部分

摘要

我们继续讨论 FM 例子 wfm_rcv_gui·py。GNU Radio 中基于 wxPython 的 GUI 工具的使用，也将会在本文出现。我们还会介绍一些参数分析(parsing)的实用编程技巧。如果你有兴趣使用甚至开发 GUI 工具，本文将会非常有用。

1 回顾

本文中，我们将会完成对 FM 接收器的代码 wfm_rcv_gui·py 的讨论。GNU Radio 有一个非常激动人心的特点，就是集成了强大的 GUI 工具用以显示和分析信号，模拟了真实的频谱分析器和示波器。我们将要介绍 GUI 工具的使用，它们是基于 wxPython 的。下一步我们将要讨论在 Python 中如何处理命令行参数。

2 GNU Radio 中的 GUI 工具

最直观和直接的分析信号的方式就是把它用图像表示出来，包括时域的和频域的。对于真实世界里的应用，我们有频谱分析器和示波器来方便自己。幸运的是，在软件无线电领域里，我们也有这样的好工具，多亏了 wxPython，提供了一种灵活的方式来构造 GUI 工具。

2.1 频谱分析器—ffi_sink

我们继续阅读代码：

```
if 1:
    pre_demod, fft_win1 = \
            fftsink.make_fft_sink_c (self, panel, "Pre-Demodulation",
                                     512, quad_rate)
    self.connect (src, pre_demod)
    vbox.Add (fft_win1, 1, wx.EXPAND)
```

这就是“软示波器”，基于对数字序列的快速傅里叶变换(FFT)。这个“软示波器”用作信号接收端。这也是为什么它的名字是“fftsink”。它定义在 wxgui·fftsink·py 模块中。

make_fft_sink_c()函数用作一个公共接口来创建一个 fft sink 的实例：

```
/site-packages/gnuradio/wxgui/fftsink.py
...
def make_fft_sink_c(fg, parent, title, fft_size, input_rate, ymin=0, ymax=100):
    block = fft_sink_c(fg, parent, title=title, fft_size=fft_size,
                       sample_rate=input_rate, y_per_div=(ymax - ymin)/8, ref_level=ymax)
    return (block, block.win)
```

首先，我们应该指出，在 Python 中，一个函数可以返回多个值。make_fft_sink_c()返回两个值：block 是 fft_sink_c 的一个实例，定义在相同的模块 wxgui·fftsink·py 中。另一个 Python 的特点需要指出：Python 支持多重继承。fft_sink_c 继承自两个类：gr·hier_block 和 fft_sink_base。作为 gr·hier_block 的一个子类，表示着 fft_sink_c 作为一个正常的块被处理，也就是可以在流图中放置和连接，就像下一行写的：

```
self.connect (src, pre_demod)
```

block·win 显然是 block 的一个属性。在 fft_sink_c 的定义中，我们可以找到它的数据类型是 fft_window 类，wx·Window 的一个子类，同样也定义在 wxgui·fftsink·py 中。我们可以把它想作是一个即将在你的屏幕挂起的窗口。这个窗口会被用作 vbox·Add 的一个参数。

2.2 wxPython 是怎么扮演它的角色的

为了充分理解其它部分，我们需要指点一点关于 wxPython 的知识，它是 Python 的一个 GUI 工具包。有兴趣的读者可以浏览 wxPython 的网站或是指南来获取更多信息。可能浏览 wxPython 的所有技术细节会非常耗时间。GNU Radio 的好消息就是，我们可以使用频谱分析器和示波器像真的一样。你修改一点点就可以把这些行复制到任何地方。

让我们来花费一些时间来研究 wxPython 的组织。第一件要做的事当然是导出 wxPython 的所有组件到工作区。就像是`import wx'这一行做的。每个 wxPython 应用到需要从 wx·App 继承一个类，并且为其提供一个 OnInit()方法。在 wx·App·__init()__中，系统调用了这个方法作为它的启动/初始化序列的一部分。OnInit()的主要作用就是创建框架、窗口等等这些程序开始操作需要的东西。定义完这样一个类之后，我们需要实例化这个类的一个对象，并且通过它的 MainLoop()方法来开始应用，这个方法用以处理事件。在我们 FM 接收机的例子中，这个类是在哪定义的？让我们来看最后三行：

```
if __name__ == '__main__':
    app = stdgui.stdapp (wfm_rx_graph, "WFM RX")
    app.MainLoop ()
```

实际上，这样一个 stdapp 类，在我们导出 stdgui 模块时就已经创建了。

```
from gnuradio.wxgui import stdgui, fftsink
```

最后两行，有可能对于所有的 wxPython 应用来说都是一样的。我们简单的创建一个我们应用的类的实例，并且调用它的 MainLoop()方法。MainLoop()是这个应用的核心，也是事件被处理并被分发(dispatched)到各个窗口的地方。这个场景背后有一些技巧。不必担心这个。

我们来看看在/gnuradio/wxgui/stugui·py 中的 stdapp 的定义：

```
class stdapp (wx.App):
    def __init__ (self, flow_graph_maker, title="GNU Radio"):
        self.flow_graph_maker = flow_graph_maker
        self.title = title
        # All our initialization must come before calling wx.App.__init__.
        # OnInit is called from somewhere in the guts of __init__.
        wx.App.__init__ (self)

    def OnInit (self):
        frame = stdframe (self.flow_graph_maker, self.title)
        frame.Show (True)
        self.SetTopWindow (frame)
        return True
```

stdapp 正是继承自 wx·App 的类。它的初始化方法__init__()有两个参数：flow_graph_maker,一个属于流图家族的类（记不记得我们创建的最大的类 wfm_rx_graph 继承自 gr·flow_graph）；title，整个应用的名称（在我们的例子中是 WFM RX）。在 OnInit()方法里，这两个参数进一步用来创建 stdframe 的对象。

```
class stdframe (wx.Frame):
    def __init__ (self, flow_graph_maker, title="GNU Radio"):
        # print "stdframe.__init__"
        wx.Frame.__init__(self, None, -1, title)

        self.CreateStatusBar (2)
        mainmenu = wx.MenuBar ()
```

```
        menu = wx.Menu ()
        item = menu.Append (200, 'E&xit', 'Exit')
        self.Bind (wx.EVT_MENU, self.OnCloseWindow, item)
        mainmenu.Append (menu, "&File")
        self.SetMenuBar (mainmenu)
        self.Bind (wx.EVT_CLOSE, self.OnCloseWindow)
        self.panel = stdpanel (self, self, flow_graph_maker)
        vbox = wx.BoxSizer(wx.VERTICAL)
        vbox.Add(self.panel, 1, wx.EXPAND)
        self.SetSizer(vbox)
        self.SetAutoLayout(True)
        vbox.Fit(self)
    def OnCloseWindow (self, event):
        self.flow_graph().stop()
        self.Destroy ()

    def flow_graph (self):
        return self.panel.fg
```

有必要来花一些时间理解 wxPython GUI 的布置(layout)。在 wxPython 中，一个 wx·Window 就是能够占用计算机屏幕可视空间的任意东西。因此，wx·Window 类是一个基类，从它这里，可以派生所有可视元素-包括输入域、下拉菜单、按钮，等等。wx·Window 定义了与所有的可视化 GUI 元素共有的所有的行为，包括位置、大小、外观、突出重点，等等。如果我们要找到一个对象来代表计算机屏幕上的窗口，不要看 wx·Window，看 wx·Frame。wx·Frame 继承自 wx·Window。它实现了计算机屏幕的窗口的所有行为。一个“Frame”就是一个大小和位置可以由用户改变的窗口。它通常有厚厚的边界和标题栏，可以包含一个菜单栏，工具栏和状态栏。一个“Frame” 可以包含任何窗口而不是一个框架或对话框。所以要创建一个“窗口”在计算机屏幕上的话，你需要创建一个 wx.Frame（或者是它的一个子类，如 wx.Dialog）而不是一个 wx.Window。

有了框架，你将会使用若干 wx.Window 的子类来充实框架的内容，比如 wx.MenuBar、wx.StatusBar 、 wx.ToolBar 、 wx.Control 的 子 类 （ 比 如 ，wx.Button,wx.StaticText,wx.TextCtrl,wx.ComboBox，等等），或者 wx.Panel，这是一个可以保存你的多样的 wx.Control 对象的容器。所有的可视元素（wx.Window 对象和他们的子类）可以有子元素。一个 wx.Frame 可以有一定数量的 wx.Panel 对象，而后者又有一些 wx.Button,wx.StaticText 和 wx.TextCtrl 对象。

在我们的例子中，stdframe，wx.Frame 的子类，用以创建“frame”。我们通过使用 frame.Show(True)来把这个框架（frame）展示出来。SetTopWindow()方法简单的告诉应用

这个框架是主框架（本例中唯一的一个）中的一个。注意到 wx.Frame 构造函数的形状是这样：

```
wx.Frame(Parent, Id, "title")
```

wxPython 中大部分的构造函数都有这种形状：一个父类对象作为第一个参数，一个 Id 作为第二个参数。就像例子里表现的那样，分别使用 None 和*-1* 作为默认参数是可以的，意味着此对象没有父类对象和系统定义的 Id。

在 stdframe·__init__中，我们创建了一个 panel 并且把其放在框架里。

```
self.panel = stdpanel (self, self, flow_graph_maker)
```

stdpanel 类继承自 wx·Panel*:*

```
class stdpanel (wx.Panel):
  def __init__ (self, parent, frame, flow_graph_maker):
    wx.Panel.__init__ (self, parent, -1)
    self.frame = frame

    vbox = wx.BoxSizer (wx.VERTICAL)
    self.fg = flow_graph_maker (frame, self, vbox, sys.argv)
    self.SetSizer (vbox)
    self.SetAutoLayout (True)
    vbox.Fit (self)

    self.fg.start ()
```

注意到 panel 的 parent 是我们刚刚创建的 frame 对象，意味着这个面板是框架的一个子成员。frame 通过使用 wx·BoxSizer,vbox 来把 panel 放置其中。box sizer 背后的基本思想是，窗口通常被放置为非常简单的基本几何图形，典型的就是一排或是一列，或是彼此嵌套。一个 wx·BoxSizer 会把它的条目都在一排或一列摆开，根据传递给构造函数的方位*(*orientation*)*参数。在我们的例子中，vbox = wx·BoxSizer*(*wx·VERTICAL*)*告诉构造函数所有的条目*(*item*)*将会被垂直摆放。SetSizer*()*告诉框架使用的哪种大小。调用 SetAutoLayout*()*告诉框架使用规划器（sizer）来定位你的组件并规范其大小。最后，调用 sizer·Fit*()*告诉规划器为其所有的元素计算初始大小和位置。如果你正在使用规划器，这是你在首次展示前设置你的窗口和框架的内容的常规流程。对于规划器来说最重要最有用的方法就是 add*()*，它会把一个条目添加到规划器里面。它的语法是这样：

```
Add(self, item, proportion=0, flag=0)
```

“item”是你希望添加到规划器的条目，通常是一个 wx·Window 对象。它也可以是一个子规划器。proportion 与规划器的一个子类是否可以在 wx·BoxSizer 的主方位中改变它的大小有关。有几个标志定义在 wx 中，它们用以确定规划器的条目如何运转以及窗口的边沿。wx·EXPAND 在我们的例子中表示这个条目将会被扩大来填充分配给其的空间。参阅这里获得完整的描述。

你有没有过这种困惑：我们定义了一个大类 wfm_rcv_graphf，但是我们在哪使用的呢？为什么我们没有创建过这个类的实例？这个秘密就在 stdpanel·__init__*()*中揭晓。wfm_rcv_graph 的实例是在这里创建的，流图也是在这里开始的。

```
class stdpanel (wx.Panel):
  def __init__ (self, parent, frame, flow_graph_maker):
    .....
    vbox = wx.BoxSizer (wx.VERTICAL)
    self.fg = flow_graph_maker (frame, self, vbox, sys.argv)
    self.SetSizer (vbox)
    self.SetAutoLayout (True)
    vbox.Fit (self)
    self.fg.start ()
```

我们在框架里放置了一个面板，但我们在面板里放了什么？我们首先为面板创建了一个新的规划器 vbox。注意到 vbox 和我们在 stdfame 中定义的不一样。然后我们创建了 flow_graph_maker*(*wfm_rcv_graph*)*类的实例，同时 vbox 作为一个参数（同时还有 frame,panel）传递给它。在 wfm_rcv_graph·__init__*()*中，vbox 通过使用 vbox·Add*()*来为规划器添加几个频谱分析器或示波器*(*wx·Window 的对象*)*。然后面板是同规划器 vbox 来定位和规范所有这些子窗口的位置和大小。最后，我们开始流图： self·fg·start*()*，相关的数据将会动态的显示在屏幕上。

我们回过头来再看一看我们的 FM 接收器例子的代码：

```
if 1:
    pre_demod, fft_win1 = \
        fftsink.make_fft_sink_c (self, panel, "Pre-Demodulation",
                                 512, quad_rate)
    self.connect (src, pre_demod)
    vbox.Add (fft_win1, 1, wx.EXPAND)
```

以及 make_fft_sink_c*()*方法的定义：

/site-packages/gnuradio/wxgui/fftsink.py

...

```
def make_fft_sink_c(fg, parent, title, fft_size, input_rate, ymin=0, ymax=100):
    block = fft_sink_c(fg, parent, title=title, fft_size=fft_size,
                       sample_rate=input_rate, y_per_div=(ymax - ymin)/8, ref_level=ymax)
    return (block, block.win)
```

所有的事情现在都很清楚了，对吧？panel 传递给 make_fft_sink_c*()*作为这个 fft 接收端的“父亲”（block·win，一个 wx·Window 对象）。返回值 block·win 保存在 fft_win*1* 中，然后附加到 vbox 中。

make_fft_sink_c*()*需要你个参数。fft_size 是 FFT 变换时使用的样点数。input_rate 是采样率。ymin 和 ymax 提供了你画图时希望显示在屏幕上的纵轴范围。

考虑到在 fft_sink_c 背后的复杂的故事。我们不讨论快速傅里叶变换时如何进行并用作一个块的，我们只考虑它到 Python 和 wxPython 的接口。事实上，另一个叫做“Numeric”的 Python 包在这里起了很大作用。然而，我们不需要知道所有的细节。理解了在 Python 层它是如何与 wxPython 及其它块交互的就足够了。

2.3 “示波器”—scope-sink

在 GNU Radio 中另一个重要的 GUI 工具就是“软示波器”-scope_sink。在我们的例子中没有使用它，但是如果你希望看到时域的波形，那它会是非常有帮助的。fft_sink 的用法是非常简单的：

```
if 1:
    scope_input, scope_win1 = \
        scopesink.make_scope_sink_f (self, panel, "Title", self.fs)
    self.connect (signal, scope_input)
    vbox.Add (scope_win1, 1, wx.EXPAND)
```

在这里要注意到 signal 是一个实浮点信号。如果你希望表示带有 I/Q 路的复信号，make_scope_sink_c*()*是个正确的选择。把这几行拷贝到你认为示波器需要出现的地方，然后把它作为一个块连接到信号。参阅/site-packages/gnuradio/wxgui/scopesink·py 获取更多的细节。

3 处理命令行参数

Python 支持创建可以在命令行中运行的程序，带有命令行参数或是短-长-类型的标志来指定多样的选项。还记得我们 stdpanel·__init__()中创建了一个 wfm_rcv_graph 实例，我们使用：

self.fg = flow_graph_maker (frame, self, vbox, sys.argv)

每个传递给程序的命令行参数将会保存在 sys·argv 中，后者是一个"列表"。在这个列表中，sys·argv[0]就是命令本身（在我们的例子中是 wfm_rcv_gui·py）。因此实际上所有的参数都保存在 sys·argv[1:]中。这也就解释了为什么我们使用 IF_freq = parseargs(argv[1:])来处理参数。

你可能想要使用短-长-形式的标志来添加多样的选项，如“-v”或“--help”。然后你就会使用 optpatse 模块了。optparse 是一个强大而又灵活的命令行解释器。你可以看这里来学习它。另一个例子，位于

gnuradio-examples/python/usrp/fsk_r(t)x·py

这个例子对于如何使用这个分析器进行了一个非常好的论述。

4 结论

本指南结束了对 FM 接收机例子的讨论。我们主要讨论了 wxPython 在 GNU Radio 中如何扮演它的角色。这里可能涉及了一些对于这些类如何组织到一起的理解，但如果我们有足够的耐心这个是不难的。另外，一个好消息就是我们可以简单的使用这些代码作为模板，而无须对实现的细节过多的担心。

参考资料

[1] **Python on-line tutorials**, http://www·python·org/doc/current/tut/

[2] **Python Library Reference - optparse**,

http://www·python·org/doc/2·3/lib/module-optparse·html

*[3]***wxPython on-line tutorials**, http://wxpython.org/tutorial.php

*[4]***wxPython wiki, Getting started**,

http://wiki.wxpython.org/index.cgi/Getting_20Started

指南 8：GNU Radio 块的字典

摘要

一个 GNU Radio 块的字典。本文介绍了大部分频繁使用的块，解释了它们的语法以及如何使用它们。

1 介绍

当我们使用 Matlab 来仿真的时候，为了代码简洁有效，我们需要很好的记住大量的 Matlab 内建的函数和工具箱，并且能够熟练的使用。这在 GNU Radio 也是一样的。GNU Radio 大约有 100 个经常使用的块。对于一定数量的应用，我们可以使用这些已经存在的块来完成设计，编程也只在 Python 层面，不需要编写我们自己的块。所以在本文中，我们来看看这些 GNU Radio 块。

浏览这些块的一个很好的途径是学习由 Doxygen 产生的两个 GNU Radio 文档。当你安装 gnuradio-core 和 usrp 包时，它们就产生了，假设你已经安装了 Doxygen。它们位于

/usr/local/share/doc/gnuradio-core-x.xcvs/html/index.html

/usr/local/share/doc/usrp-x.xcvs/html/index.html

一个块实际就是一个由 C++实现的类。比如，在 FM 接收机的例子中，我们使用了 gr.quadrature_demod_cf 块。这个块对应于由 C++实现的 gr_quadrature_demod_cf。SWIG 生成了它到 Python 的接口。SWIG 做了一些神奇的事情，因此从 Python 的视角来看，这个块就成了一个定义在模块 gr 中的名为 quadrature_demod_cf 的类，所以我们可以在 Python 中使用 gr.quadrature_demod_cf 来使用这个块。当你看 Doxygen 文档时，像 gr、qa、usrp 这样的前缀，对应于 Python 中的模块名字，前缀后面的部分是那个模块中的真实的块名，比如 quadrature_demod_cf, fir_filter_fff。所以当我谈论名为`A_B_C_...'的块名时，在 Python 中我们通过`A.B_C_...'来使用它。

2 信号源

2.1 正弦和余弦源

块：gr·sig_source_X

用法：

```
gr.sig_source_c [f, i, s]  ( double sampling_freq,
                             gr_waveform_t waveform,
                             double frequency,
                             double amplitude,
```

gr_complex [float, integer, short] offset)

说明：后缀 x 表示的是信号源的数据类型。它可以是 c(complex)，f(float)，i(4 字节整型)或 s(2 字节短整型)。offset 参数的类型同信号一样。waveform 参数表示的是波形的类型。gr_waveform_t 是一个定义在 gr_sig_source_waveform 中的枚举类型。它的值可以是：

gr.GR_CONST_WAVE

gr.GR_COS_WAVE

gr.GR_SIN_WAVE

当你使用 gr·GR_CONST_WAVE 时，输出将会是一个常量电压，也就是幅度加偏移量。

2.2 噪声源

块：gr·noise_source_X

用法：

gr.noise_source_c [f, i, s] (gr_noise_type_t type,

float amplitude,

long seed)

说明：后缀 x 表示的是信号源的数据类型。它可以是 c(complex)，f(float)，i(4 字节整型)或 s(2 字节短整型)。type 参数表示的是噪声的类型。gr_noise_type_t 是一个定义在 gr_noise_type.h 中的枚举类型。它的值可以是：

GR_UNIFORM

GR_GAUSSIAN

GR_LAPLACIAN

GR_IMPULSE

选择 GR_UNIFORM 会产生均匀分布的离散信号，范围在[-amplitude, amplitude]。GR_GAUSSIAN 产生了零均值和方差 amplitude2 的偏差(deviate)。相似的，GR_LAPLACIAN 和 GR_IMPLUSE 则分别代表了拉普拉斯分布和脉冲分布。所有这些噪声源的块都是基于伪随机数产生块 gr.random。你可以在/gnuradio-core/src/lib/general/gr_random.h(cc)中看一看在多种分布下如何产生随机数。

2.3 Null 源

块：gr·null_source

用法：

gr.null_source (size_t sizeof_stream_item)

说明：gr.null_source 产生了零常量。参数 sizeof_stream_item 指定了零流(zero stream)的数据类型，如 gr_complex,float,integer 等等。

2.4 向量源

块：gr·vector_source_X

用法：

gr.vector_source_c [f, i, s, b] (const std::vector< gr_complex > & data,
bool repeat = false)

(gr_complex can be replaced by float, integer, short, unsigned char)

说明：后缀 X 暗示着信号源的数据类型。它可以是 c（复），f（浮点），i（4 字节整型），s（2 字节短整型），或是 b（1 字节无符号字符型）。这些源从一个向量得到他们的数据。参数 repeat 决定了向量中的数据是否重复发送。举个例子，我们可以在 Python 中这样使用这个块：

```
src_data = (-3, 4, -5.5, 2, 3)
src = gr.vector_source_f (src_data)
```

2.5 文件源

块：gr·file_source

用法：

gr.file_source (size_t itemsize,
const char * filename,
bool repeat)

说明：gr·file_source 从文件中读取数据流。通过 filename 来指定文件名字。第一个参数 itemsize 决定了这个流的数据类型，如 gr_complex,float,unsigned char·参数 repeat 决定了这个文件中的数据是否重复发送。举个例子，我们在 Python 中可以这样使用这个块：

```
src = gr.file_source (gr.sizeof_char, "/home/dshen/payload.dat", TRUE)
```

2.6 音频源

块：gr·audio_source

用法：

gr.audio_source (int sampling_rate)

说明：audio_source 从音频输入读入数据。参数 sampling_rate 指定了源的数据率，单

位是样点/秒。

2.7 USRP 源

块：usrp.source_c [s]
用法：

```
usrp.source_c (s) (int           which_board,
                   unsigned int  decim_rate,
                   int           nchan = 1,
                   int           mux = -1,
                   int           mode = 0 )
```

说明：当你使用 GNU Radio 设计一个接收机的时候，比如，工作在 RX 路径，可能你就会需要 USRP 作为一个数据源。后缀 c(complex),或 s(short)指定了从 USRP 出来的流的数据类型。复信号源（DDC 输出的 I/Q 正交信号）使用的更为频繁。一些参数在指南 4 里介绍过了。which_board 指定了打开哪个 USRP 板，数值为 0 表示只有一个 USRP 板。decim_rate 告诉 DDC 抽取因子 D。nchan 指定了通道(channels)数量，1，2 或 4。mux 设定输入 MUX 的配置，也就是决定了哪个 ADC（或 0 常量）连接到了哪个 DDC（参阅指南 4 获取更多细节）输入。'-1'表示我们保存默认设置。mode 设定 FPGA 模式，我们很少用到。默认值为 0 表示常规模式。通常我们只会指定前两个参数，其他的都使用默认值。

例如：

```
usrp_decim = 250
src = usrp.source_c (0, usrp_decim)
```

3 信号接收端

3.1 Null 接收端

块：gr·null_sink

用法：

```
gr.null_sink ( size_t  sizeof_stream_item )
```

说明：gr·null_sink 除了耗尽（eat up）你的流以外什么也没有做。参数

sizeof_stream_item 指定了零流的数据类型，比如 gr_complex,float,integer 等等。

3.2 向量接收端

块：gr·vector_sink_X

用法：

gr.vector_sink_c [f, i, s, b] ()

说明：后缀 x 表示信号接收的数据类型。它可以是 c(复)，f（浮点），i（4 字节整型），s（2 字节短整型），或 b（1 字节无符号字符）。这些接收端将数据写入了一个向量。举个例子，我们在 Python 中可以这样使用这个块：

dst = gr.vector_sink_f ()

3.3 文件接收端

块：gr·file_sink

用法：

```
gr.file_sink ( size_t       itemsize,
               const char *   filename )
```

说明：gr·file_source 将数据流写入一个文件。文件名通过 filename 来指定。第一个参数 itemsize 决定了输入流的数据类型，比如 gr_complex,float,unsigned char·举个例子，我们在 Python 中可以这样使用这个块：

src = gr.file_source (gr.sizeof_char, "/home/dshen/rx.dat")

3.4 音频接收端

块：gr·audio_sink

用法：

gr.audio_source (int sampling_rate)

说明：当你完成信号处理并希望把这个信号通过喇叭播放出来，你将会使用这个音频接收端。audio_sink 将会把数据以采样率 sampling_rate 输出到声卡。

3.5 USRP 接收端

块：usrp·sink_c*[s]*

用法：

usrp.sink_c (s) (int　　which_board,
　　unsigned int　interp_rate,
　　int　　nchan = 1,
　　int　　mux = -1)

说明：当你使用 GNU Radio 设计一个发送器的时候，比如，工作在 TX 路径，你就可能会需要 USRP 作为数据接收端。后缀 c*(*complex*)*,或 s*(*short*)*指定了进入 USRP 板的流的数据类型。复信号接收端使用的更为频繁。一些参数在指南 *4* 中已经介绍过了。which_board 指定了打开哪个 USRP 板，*0* 表示只有一个 USRP 板。interp_rate 告诉 FPGA 上的插值器插值因子 I。注意插值率 I 必须在*[4,512]*范围内，并且一定要是 *4* 的倍数。nchan 指定了通道*(*channels*)*数量，*1* 或者 *2* 。mux 设定了输出 MUX 配置，决定了哪个 DAC 要连接哪个插值器输出（或不可用）。*'-1'*表示我们保存默认设置。通常我们只会指定前两个参数，其他的都使用默认值。例如：

usrp_interp = 256
src = usrp.sink_c (0, usrp_interp)

4 简单操作

4.1 添加一个常量

块：gr·add_const_XX

用法：

gr.add_const_cc [ff, ii, ss, sf] (gr_complex [float, integer, short] k)

说明：后缀 xx 包含两个字符。第一个字符表示输入流的数据类型，第二个表示输出流的数据类型。它可以是 cc（复数到复数），ff（浮点到浮点），ii（*4* 字节整型到整型），ss（*2* 字节短整型到短整型），或者 sf（短整型到浮点型）。gr·add_const_XX 块为输入流添加了一个常量，是信号有了偏移。注意 gr·add_const_sf,参数是 k 浮点型。我们添加了一个浮点常

量到短整型流上，输出流变为浮点型。

4.2 加法器

块：gr.add_XX
用法：

gr.add_cc [ff, ii, ss] ( )

说明：后缀 xx 表示输入和输出流的数据类型。gr·add_XX 把所有输入流添加到一起。

所有输入流的类型必须是一样的。当我们在 Python 中使用它时，多个上游（upstream）块可以连接到它。例如：

```
adder = gr.add_cc ()
fg.connect (stream1, (adder, 0))
fg.connect (stream2, (adder, 1))
fg.connect (stream3, (adder, 2))
fg.connect (adder, outputstream)
```

adder 的输出就是 stream*1*+stream*2*+stream*3*

4.3 减法器

块：gr·sub_XX

用法：

gr.sub_cc [ff, ii, ss] ( )

说明：后缀 xx 表示输入和输出流的数据类型。gr·sub_XX 减去了所有通过的输入流。

output = input_*0* - input_*1* - input_*2* …。所有的输入流的类型都必须是一样的。当我们在 Python 中使用它的时候，多个上游（upstream）块可以连接到它。例如：

```
subtractor = gr.sub_cc ()
fg.connect (stream1, (subtractor, 0))
fg.connect (stream2, (subtractor, 1))
fg.connect (stream3, (subtractor, 2))
fg.connect (subtractor, outputstream)
```

此时 subtractor 的输入就是 stream*1*-stream*2*-stream*3*·

4.4 乘以一个常量

块：gr.multiply_const_XX
用法：

gr.add_const_cc [ff, ii, ss] (gr_complex [float, integer, short] k)

说明：后缀 xx 表示输入和输出流的数据类型。gr·multiply_const_XX 块将输入流乘了一个常数 k。

4.5 乘法器

块：gr·multiply_XX

用法：

gr.multiply_cc [ff, ii, ss] ( )

说明：后缀 xx 表示输入和输出流的数据类型。gr·multiply_XX 计算所有输入流（相乘）的结果。所有的输入流的类型都必须是一样的。当我们在 Python 中使用它的时候，多个上游（upstream）块可以连接到它。例如：

```
multiplier = gr.multiply_cc ()
fg.connect (stream1, (multiplier, 0))
fg.connect (stream2, (multiplier, 1))
fg.connect (stream3, (multiplier, 2))
fg.connect (multiplier, outputstream)
```

此时 multiplier 的输出是 stream*1*×stream*2*×stream*3*·

4.6 除法器

块：gr·divide_XX

用法：

gr.divide_cc [ff, ii, ss] ( )

说明：后缀 xx 表示输入和输出流的数据类型。gr·divide_XX 计算所有输入流（相除）的结果。output = input_*0*／input_*1*／input_*2*···· 所有的输入流的类型都必须是一样的。当我们在 Python 中使用它的时候，多个上游（upstream）块可以连接到它。例如：

```
divider = gr.divide_cc ()
fg.connect (stream1, (divider, 0))
```

fg.connect (stream2, (divider, 1))

fg.connect (stream3, (divider, 2))

fg.connect (divider, outputstream)

此时 subtractor 的输出就是 stream*1*÷stream*2*÷stream*3*·

4.7 Log 函数

块：gr·nlog*10*_ff

用法：

gr.nlog10_ff (float n,

unsigned vlen)

说明：gr·nlog*10*_ff 计算了 n ×log*10(*input*)*·输入和输出都是浮点数。忽略参数 vlen，向量长度，使用其默认值 *1*·

5 类型转换

5.1 复数转换

块：

gr.complex_to_float

gr.complex_to_real

gr.complex_to_imag

gr.complex_to_mag

gr.complex_to_arg

用法：

gr.complex_to_float(unsigned int vlen)

gr.complex_to_real(unsigned int vlen)

gr.complex_to_imag(unsigned int vlen)

gr.complex_to_mag(unsigned int vlen)

gr.complex_to_arg(unsigned int vlen)

说明：参数 vlen 是向量长度，我们通常使用其默认值 *1*·所有就忽略这个参数吧。这些块将一个复信号转换为单独的实&虚浮点流，实部，虚部，复信号的幅度以及复信号的相角。

注意 gr·complex_to_float 可以有 *1* 或 *2* 的输出。如果只连接了一路输出，那么输出就是复信号的实部。它的作用等效于 gr·complex_to_real·

5.2 浮点型转换

块：

gr.float_to_complex

gr.float_to_short

gr.short_to_float

用法：

gr.float_to_complex ( )

gr.float_to_short ( )

gr.short_to_float ( )

说明：这些块就像是“适配器”，为两个不同类型的块提供接口。注意 gr·float_to_complex 块可以有 *1* 或 *2* 输入。如果这里只有 *1* 个，那么输入信号就是输出信号的实部，而虚部就是常数 *0*·如果有两路输入，那么它们就分别是输出信号的实部和虚部。

6 滤波器

6.1 滤波器设计

块：gr·firdes

说明：gr·firdes 有几个静态公共成员函数用来设计不同类型的 FIR 滤波器。这些函数返回一个包含 FIR 系数的向量。返回的向量经常用作其他 FIR 滤波器块的参数。

6·1·1 低通滤波器

用法：

```
vector< float > gr.firdes::low_pass ( double      gain,
                                      double      sampling_freq,
                                      double      cutoff_freq,
                                      double      transition_width,
```

win_type window = WIN_HAMMING,
double beta = 6.76) [static]

说明：low_pass 是 gr·firdes 类的一个静态公共成员函数。它设计的 FIR 滤波器系数（抽头（taps））指定了滤波器。这里参数 window 是 FIR 滤波器设计中用的窗的类型。有效值包含：

WIN_HAMMING
WIN_HANN
WIN_BLACKMAN
WIN_RECTANGULAR

6·1·2 高通滤波器

用法：

vector< float > gr.firdes::high_pass (double gain,
double sampling_freq,
double cutoff_freq,
double transition_width,
win_type window = WIN_HAMMING,
double beta = 6.76) [static]

6·1·3 带通滤波器

用法：

vector< float > gr.firdes::band_pass (double gain,
double sampling_freq,
double low_cutoff_freq,
double high_cutoff_freq,
double transition_width,
win_type window = WIN_HAMMING,
double beta = 6.76) [static]

6·1·4 带阻滤波器

用法：

vector< float > gr.firdes::band_reject (double gain,
double sampling_freq,
double low_cutoff_freq,

```
                double      high_cutoff_freq,
                double      transition_width,
                win_type    window = WIN_HAMMING,
                double      beta = 6.76)   [static]
```

6·1·5 希尔伯特滤波器

用法：

```
vector< float > gr.firdes::hilbert ( unsigned int     ntaps,
                                     win_type    windowtype = WIN_RECTANGULAR,
                                     double      beta = 6.76 )   [static]
```

说明：gr·firdes*::*hilbert 设计了希尔伯特变换滤波器。ntaps 是抽头的数量，它必须是奇数。

6·1·6 升余弦滤波器

用法：

```
vector< float > gr.firdes::root_raised_cosine ( double      gain,
                                                double      sampling_freq,
                                                double      symbol_rate,
                                                double      alpha,
                                                int         ntaps )   [static]
```

说明：gr·firdes*::*root_raised_cosine 设计了一个根升余弦滤波器。alpha 参数是截止*(*excess*)*带宽因子。ntaps 是抽头的数量。

6·1·7 高斯滤波器

用法：

```
vector< float > gr.firdes::gaussian ( double      gain,
                                      double      sampling_freq,
                                      double      symbol_rate,
                                      double      bt,
                                      int         ntaps )   [static]
```

说明：gr·firdes*::*gaussian 设计了一个高斯滤波器。参数 ntaps 是抽头的数量。

6.2 FIR 抽取滤波器

块：

gr.fir_filter_ccc

gr.fir_filter_ccf

gr.fir_filter_fcc

gr.fir_filter_fff

gr.fir_filter_fsf

gr.fir_filter_scc

用法：

```
gr.fir_filter_  cc (int decimation,
                    const std::vector< gr_complex > & taps )
gr.fir_filter_ccf (int decimation,
                    const std::vector< float > & taps )
gr.fir_filter_fcc (int decimation,
                    const std::vector< gr_complex > & taps )
gr.fir_filter_fff (int decimation,
                    const std::vector< float > & taps )
gr.fir_filter_fsf (int decimation,
                    const std::vector< float > & taps )
gr.fir_filter_scc (int decimation,
                    const std::vector< gr_complex > & taps )
```

说明：这些块都有 *3* 个字符的后缀。第一个表示输入流的数据类型，第二个是输出流的数据类型，最后一个代表了 FIR 滤波器抽头的数据类型。

每个块都有两个参数。第一个是 FIR 滤波器的抽取率。如果是 *1*，那么就是常规的 *1*:*1*FIR 滤波器。第二个参数 taps 是 FIR 系数的向量，也就是我们从 FIR 滤波器设计块 gr·firdes 块中得到的。

6.3 FIR 插值滤波器

块：

gr.interp_fir_filter_ccc

gr.interp_fir_filter_ccf

gr.interp_fir_filter_fcc

gr.interp_fir_filter_fff

gr.interp_fir_filter_fsf

gr.interp_fir_filter_scc

用法：

```
gr.interp_fir_filter_ccc (unsigned interpolation,
                          const std::vector< gr_complex > & taps )
gr.interp_fir_filter_ccf (unsigned interpolation,
                          const std::vector< float > & taps )
gr.interp_fir_filter_fcc (unsigned interpolation,
                          const std::vector< gr_complex > & taps )
gr.interp_fir_filter_fff (unsigned interpolation,
                          const std::vector< float > & taps )
gr.interp_fir_filter_fsf (unsigned interpolation,
                          const std::vector< float > & taps )
gr.interp_fir_filter_scc (unsigned interpolation,
                          const std::vector< gr_complex > & taps )
```

说明：这些块都有 *3* 个字符的后缀。第一个表示输入流的数据类型，第二个是输出流的数据类型，最后一个代表了 FIR 滤波器抽头的数据类型。

每个块都有两个参数。第一个是 FIR 滤波器的插值率。第二个参数 taps 是 FIR 系数的向量，也就是我们从 FIR 滤波器设计块 gr·firdes 块中得到的。

6.4 带有 FIR 抽取滤波器的数字下变频转换器

块：

gr.freq_xlating_fir_filter_ccc
gr.freq_xlating_fir_filter_ccf
gr.freq_xlating_fir_filter_fcc
gr.freq_xlating_fir_filter_fcf
gr.freq_xlating_fir_filter_scc
gr.freq_xlating_fir_filter_scf

用法：

```
gr.freq_xlating_fir_filter_ccc [ccf, fcc, fcf, scc, scf]
                        ( int      decimation,
                          const std::vector< gr_complex [float] > &   taps,
                          double     center_freq,
                          double     sampling_freq )
```

说明：这些块都有 *3* 个字符的后缀。第一个表示输入流的数据类型，第二个是输出流的数据类型，最后一个代表了 FIR 滤波器抽头的数据类型。

这些块被用作结合了频率变换的 FIR 滤波器。回顾指南 *4*，在 FPGA 中进行了 DDC 使信号从 IF 变换到基带。然后抽取器对信号进行下采样并通过低通滤波器选择了一个窄带。这些块除了使用软件做的是一样的事情。这些类有效地结合了频率变换（通常是下变换）和 FIR 滤波器（典型的是低通）和抽取。它是理想的“通道选择滤波器”并且可以有效的用作选择抽取一个出自宽带宽输入的窄带信号。

6.5 希尔伯特变换滤波器

块：gr·hilbert_fc

用法：

gr.hilbert_fc(unsigned int ntaps)

说明：gr·hilbert_fc 是一个希尔伯特变换。实部输出是输入的恰当延迟。虚部输出是对输入的希尔伯特变换（*90*°相位转换）值。我们使用这个块时只需要指定抽头数量 ntaps。

希尔伯特滤波器设计 gr·firdes*::*hilbert 暗含在 gr·hilbert_fc 块的实现中。

6.6 滤波延迟组合滤波器

块：gr·filter_delay_fc

用法：

gr.filter_delay_fc(const std::vector< float > & taps)

说明：这是一个滤波延迟组合块。这个块接受一路或两路浮点型流并输出复信号流。如果只输入了一个浮点流，那么实部输出就是这个输入的延迟并且虚部输出就是滤波输出。如果输入连接了两路浮点流，那么实部就是第一路输入的延迟，虚部就是滤波输出。计算实部一路的延迟是因为虚部一路的滤波引入了组延迟。在初始化这个块之前滤波器抽头需要用 gr·firdes 来计算。

6.7 IIR 滤波器

块：gr·iir_filter_ffd

用法：

```
gr.iir_filter_ffd ( const std::vector< double > &  fftaps,
                    const std::vector< double > &  fbtaps)
```

说明：后缀 ffd 表示浮点输入，浮点输出，double 的抽头。这个 IIR 滤波器使用直接 I

实现，这里 ffttaps 包含前馈抽头，fbtaps 是反馈抽头。fftyaps 和 fbtaps 必须有相同数量的抽头。输入和输出要满足下列差分方程：

$$y[n]-\sum_{k=1}^{N}a_k y[n-k]=\sum_{k=0}^{M}b_k x[n-k]$$

带有相应的合理的系统函数：

$$H(z)=\frac{\sum_{k=0}^{M}b_k z^{-k}}{1-\sum_{k=1}^{N}a_k z^{-k}}$$

注意有些文章定义系统函数是在分母上带有一个 ‘+’。如果你使用那种惯例，你需要取消反馈抽头。

6.8 单级 IIR 滤波器

块：gr·single_pole_iir_filter_ff

用法：

gr.single_pole_iir_filter_ff (double alpha,
unsigned int vlen)

说明：这是一个带有浮点输入浮点输出的单级 IIR 滤波器。输入和输出满足差分方程：

$$y[n]-(1-\alpha)y[n-1]=\alpha x[n]$$

带有相应合理的系统函数：

$$H(z)=\frac{\alpha}{1-(1-\alpha)z^{-1}}$$

注意有些文章定义系统函数是在分母上带有一个 ‘+’。如果你使用那种惯例，你需要取消反馈抽头。参数 vlen 是向量长度。我们通常使用其默认值 *1*，所以忽略掉它吧。

7 FFT

块：

gr.fft_vcc

gr.fft_vfc

用法：

gr.fft_vcc (int fft_size,

bool forward,
bool window)

gr.fft_vfc (int fft_size,
bool forward,
bool window)

说明：这些用来计算输入序列的傅里叶变换。对 gr·fft_vcc 来说，它计算正向或反向 FFT，复向量输入，复向量输出。对 gr·fft_vfc 来说，它计算正向 FFT，浮点向量输入，复向量输出。

8 其他有用的块

8.1 FM 调制和解调

块：

gr·frequency_modulator_fc

gr·quadrature_demod_cf

用法：

gr.frequency_modulator_fc (double sensitivity)

gr.quadrature_demod_cf (float gain)

说明：gr·frequency_modulator_fc 块是 FM 调制器。输出复信号的瞬时频率对应于输入浮点信号。我们在 FM 接收机例子中已经见过 gr·quardrature_demod_cf 了。它计算输入信号的瞬时频率。

8.2 数控振荡器

块：gr·fxpt_nco

用法：这个块用来产生正弦波。我们可以通过这个类的几个公共成员函数来设置或调整振荡器的相位或频率。这些函数包括：

void set_phase (float angle)

void adjust_phase (float delta_phase)

void set_freq (float angle_rate)

void adjust_freq (float delta_angle_rate)

void step ()

void step (int n)

float get_phase () const

float get_freq () const

void sincos (float *sinx, float *cosx) const

float cos () const

float sin () const

我们使用 cos()或者 sin()函数来得到正弦样点。它们根据当前相位来计算 sin 和 cos 值。freq 就是连续样点之间的相位差。当我们使用 step()方法是，当前相位会通过 freq 来提高。

8.3 数字传输块

块：

gr.bytes_to_syms

gr.simple_framer

gr.simple_correlator

用法：

gr.bytes_to_syms ( )

gr.simple_framer (int payload_bytesize)

gr.simple_correlator (int payload_bytesize)

说明：gr·bytes_to_syms()把一字节序列（例如，一个无符号字符流）转换为一个±序列，比如，数字二进制符号。gr·simple_framer 把一字节流打包为一个包，长度是 payload_bytesize。然后必要的同步和命令字节添加到头。gr·simple_correlator 是一个数字检测器，可以对符号和帧同步，最后检测出正确的数字信息。这些块的真正实现部分有点复杂。请先学习 FSK 例子 fsk_r[t]x·py 和这些块的源码。当我们想要设计数字传输方案时这些块会非常有用。

9 结合起来

本指南回顾了一些 GNU Radio 中频繁使用的块。熟练的使用这些块是非常重要和有用的。当然我们只讨论了可用的块的一小部分。还有一些更深的，较少使用的块我们没有讨论。随着 GNU Radio 的流行（也许有一天会有你写的块）也有越来越多的块加入进来。我们的介绍时非常简单的。如果你想知道一个块的所有细节，请直接去看它的文档页然后阅读其源

码。源代码是理解块里发生什么的最好地方。通过学习一些例子来看一个块是如何使用也是一个非常好的途径。

参考文献

[1] **GNU Radio 2.x Documentation** http://www.gnu.org/software/gnuradio/doc/index.html

指南 9：为 GNU Radio 写一个信号处理模块（第一部分）

摘要

本章解释了如何为 GNUradio 编写一个信号处理模块。具体来说，我们要讨论如何应用 C++中的 gr_block 模块派生出的一个类，命名习惯以及如何用 SWIG 来产生 Python 和 C++中的接口。最后我们将得到一个“产品”，它是 GNUradio 软件包中的一个 Python 模块，并且我们可以很轻易地应用此模块。

1. 回顾

这篇教程中我们解释如何为 GNU Radio 编写一个信号处理模块。本篇教程实际上是 Eric Blossom 编写的网络教程`How to Write a Signal Processing Block' 的扩展版本。我们将采用相同的材料并且应用相同的例子。然而，为了增强可读性我们也增加了更多的内容和细节。

在前面的教程中，我们已经介绍了 GNU Radio 中由 Python 到 C++的连接结构。我们也在例子中碰到了一些模块，例如 gr_sig_source_f，gr_quadrature_demod_cf。当时我们跳

过了这些模块是怎样被应用的以及他们是怎样连接到 Python 的这些细节问题。在本篇教程中，我们将揭开这些秘密。

我们将从一些最简单的信号处理模块的结构入手并解释其中应用到的方法和习惯用法。

2. 三万米高的视角

从 Python 的角度来看，GNU Radio 提供了一个抽象的数据流。基本的概念是信号处理模块以及他们之间的连接。正如我们在指南 *6* 中所讨论的一样，这样的抽象是由 Python 中的类 gr_flow_graph 实现的。每一个模块都有一些输入端口和输出端口。每一个端口都有一个联合数据类型。最常见的端口类型是浮点类型和复数型(等同于 std::complex<float>)

从高层的角度来看，无穷的数据流经过了这些端口。在 C++层面，数据流以简洁的数据块的方式被处理，表示为下一层类型的相邻数据。

当我们编写一个模块的时候，我们要以便于共享的方式构造它，也就是说可以应用“接口”机制自动的接入 Python。SWIG-简单的打包器和接口形成器，被用来产生能从 Python 的角度利用我们所写的代码的连接。编写一个新的信号处理模块需要创建三个文件：·h 和·cc 文件用来定义新的模块类，·i 文件告诉 SWIG 怎样产生接通类到 Python 的连接。新的类必须继承于 gr_block 或是它的一个子类。

3·所有信号处理模块的基类：gr_block

C++类 gr_block 是 GNU Radio 中所有信号处理模块的基础。我们尝试编写的一个新模块必须衍生自 gr_block 或者它的某个子类。所以我们先通过 gr_block·h 文件来了解学习它。找到 gr_block·h 的源代码有两种方法：一是打开 GNU Radio 文件夹，选择‘文件列表’标签，之后搜索 gr_block·h，另一种方法是通过地址/src/lib/runtime 来找到它。附录 A 中有 gr_block·h 的源代码。

让我们首先浏览一下 gr_block 中定义的成员变量：

```
private:

  std::string                  d_name;
```

```
    gr_io_signature_sptr      d_input_signature;
    gr_io_signature_sptr      d_output_signature;
    int                       d_output_multiple;
    double                    d_relative_rate;        // approx output_rate / inp
ut_rate
    gr_block_detail_sptr      d_detail;               // implementation details
    long                      d_unique_id;            // convenient for debuggin
g
```

d_name 是用来保存模块名的字符串。d_unique_id 是长整型，它定义了此模块的‘ID’，这是为了方便调试。

什么是 d_input_signature 和 d_output_signature 呢？数据类型 gr_io_signature_sptr 又是什么呢？问题很快就变得复杂起来。在这里我们必须要解释两个问题：一个是 类 gr_io_signature；另一个是灵活的指针 boost。现在，让我们先做点准备工作。

3.1 用于输入输出的类：gr_io_signature

类 gr_io_signature 定义在/src/lib/runtime/gr_io_signature.h 中。就像它的名字所指出的，gr_io_signature 就像对模块输入输出流的一种认可，指出了输入输出流的基本信息。现在让我们来看看 gr_io_signature.h 文件中关于定义的一部分：

```
class gr_io_signature {
public:
    ~gr_io_signature ();
    int min_streams () const {return d_min_streams ;}
    int max_streams () const {return d_max_streams ;}
    size_t sizeof_stream_item (int index) const {return d_sizeof_stream_item ;}
private:
```

```
    int             d_min_streams;

    int             d_max_streams;

    size_t          d_sizeof_stream_item;

    gr_io_signature (int min_streams, int max_streams, size_t sizeof_stream_item);

    friend gr_io_signature_sptr gr_make_io_signature (int min_streams,
                                                      int max_streams,
                                                      size_t sizeof_stream_item);
};
```

对于一个模块的输入或是输出，类 gr_io_signature 定义了数据流中的最大值（d_max_streams）和最小值（d_min_streams）作为最高和最低的边界。d_sizeof_stream_item 表示了数据流中一个数据块的大小（所占的字节数），它是 size_t 类型的成员变量。

当我们要生成一个模块的时候，我们必须为输入和输出流指明两个‘signatures’。

3.2 灵活的指针：Boost

数据类型 gr_io_signature_sptr 是什么呢？让我们对它做一个深入的学习。gr_runtime·h 包含在 gr_block·h 中。在 gr_runtime·h 中，我们可以看到数据类型 gr_io_signature_sptr 是这样定义的：

```
typedef boost::shared_ptr<gr_io_signature>     gr_io_signature_sptr;
```

更进一步的，gr_runtime·h 首先包含了 gr_type·h 文件。在 gr_type·h 中，我们引入了一个有趣的头文件：

```
#include <boost/shared_ptr·hpp>
```

GNU Radio 吸取了灵活的指针包 Boost 的优点。

3·2·1 什么是 **Boost**?

Boost 是一个 C++库文件的一个集合包，在安装 GNU Radio 软件包之前你要提前安装它，Boost 从很多方面对 C++提供了强大的扩展，例如算法的应用，数学/数字，输入/输出，

迭代，等等。如果想要获得更多你所感兴趣的程序可以查看网站 http://www.boost.org/。

3.2.2 什么是灵活的指针

GNU Radio 从 Boost 中引入一种有用的特性：smart_ptr 库，因此称为灵活的指针包，它是能够动态分配指针存储到特定对象的一个对象包。它们的作用类似于内置的 C++指针，但是它们还能在适当的时间自动的删除指针对象。灵活的指针包具有的这一特性在动态分配的对象具有合适的自动析构能力在异常条件下特别的有用。他们还能被用来跟踪被多个用户共享的动态分配对象。概念上来说，灵活的指针包在拥有被指针所指的对象时是可见的，相应的在不再需要此指针所指的对象时可以删除这个对象。

实际上，灵活的指针包是作为模版类来定义的。Smart_ptr 库提供了五个灵活指针包的模版类，但是在 GNU Radio 中，我们只用了其中的一个：shared_ptr，它定义在<boost/shared_ptr.hpp>中。shared_ptr 被用于这样的场合：此时被指针所指的对象被多个指针共享。

3.2.3 怎样应用 GNU Radio 中的灵活指针

应用 GNU Radio 中的灵活的指针包 Smart_ptr，首先我们需要包括头文件<boost/shared_ptr.hpp>。然后我们利用它去定义一个新的灵活的指针，就像这样：

```
boost::shared_ptr<T>  pointer_name
```

这些灵活的指针类模版都有一个模版参数：T，这个参数表示由这个灵活变量所指对象的类型。例如：语句 boost::shared_ptr<gr_io_signature>定义了一个灵活的指针，它指向一个类型为 gr_io_signature 的对象。至此，我们就已经解释了 gr_io_signature_sptr 的含义。

不管怎样，我们可以简单的理解 Boost 灵活指针是一种内置的 C++指针，只是它具有一些很实用的功能。除此之外它们没有明显的区别。

现在让我们回过头去看看类 gr_block 中定义的成员变量。我们已经可以把 d_input_signature 和 d_output_signature 理解为两个出自对象 gr_io_signature 的灵活指针，它们是一个模块中最重要的信息用来指代输入和输出数据流。让我们暂时放下其它的几个成员变量，首先来看看两个很重要的方法，它们定义在 gr_block：forecast()和 general_work()中。介绍它们之前，我们还应该首先熟悉一下 GNU Radio 中定义的数据类型。

3.3 GNU Radio 中的数据类型

GNU Radio 中定义的特定的数据类型可以在 gr_types.h 和 gr_complex.h 这两个文件中找到:

```
typedef std::complex<float>            gr_complex;

typedef std::complex<double>           gr_complexd;


typedef std::vector<int>               gr_vector_int;

typedef std::vector<float>             gr_vector_float;

typedef std::vector<double>            gr_vector_double;

typedef std::vector<void *>            gr_vector_void_star;

typedef std::vector<const void *>      gr_vector_const_void_star;


typedef short                          gr_int16;

typedef int                            gr_int32;

typedef unsigned short                 gr_uint16;

typedef unsigned int                   gr_uint32;
```

就像我们所看到的，一种数据类型只是 C++内置数据类型的一个新名字加上‘gr’这样的前缀。我们只是采用这样具有一致性的便利的方法来定义它们。Vector 和 complex 都是 C++标准类型，它们在 GNU Radio 中是十分常用的。

3.4 一个模块的核心：general_work()方法

general_work()方法是完全虚拟的方法，我们绝对要了解这一点。该方法完成实际上的信号流程，这是一个模块的核心部分。

```
virtual int general_work (int                  noutput_items,
```

```
        gr_vector_int             &ninput_items,
        gr_vector_const_void_star  &input_items,
        gr_vector_void_star       &output_items) = 0;
```

简单的说，general_work()方法由输入流计算出输出流。首先让我们介绍一下这四个参数。

noutput_items 是写入每路输出流数据的数目，ninput_items 则给出了每路输入数据可用的数据数目。一个模块可以有 x 个输入流和 y 个输出流。ninput_items 是一个长为 x 的整型向量，其中第 i 个元素给出了第 i 个输入流数据的可用符号数目。然而，对于输出流来说，为什么 noutput_items 仅仅是一个整数而不是向量呢？这是因为出于一些技术因素，当前的 GNU Radio 版本只支持这样的模块，它对所有的输出流采用相同的数据速率，也就是说写给每一个输出流的数据数目和写给所有输出流的数目是一样的。输入数据流的速率可以不一样。

我们在上一个子目录下已经介绍了数据类型 gr_vector_const_void_star 和 gr_vector_void_star。input_items 是指向输入数据的指针向量，每个输入数据流只有一个入口。output_items 是指向输出数据的指针向量，每个输出数据流同样只有一个入口。我们实际上利用这些指针得到输入的数据并且把计算得到的输出数据写入合适的数据流。

可以注意到刚刚介绍的三个成员 ninput_items，input_items 和 output_items 只是简要的定义的，由符号‘&’指明的。这样它们在 general_work()方法中就可以被改变了。

general_work()方法的返回值是实际上写入每个输出数据流的数据数目，或者是-1 在 EOF 中。返回一个小于 noutput_items 的值也是可以的。

最后也是最重要的当我们应用 general_work()方法写我们自己的模块的时候，我们必须调用 consume()或是 consume_each()方法来指明每个输入数据流有多少数据量被消耗了。

```
void consume (int which_input, int how_many_items);
```

consume()方法告诉调度器第 i 路输入流中多少数据（由‘how_many_items’给出）已经被消耗了。如果每个输入数据消耗了相同数目的数据，我们可以用 consume_each() 方法来替代。

```
void consume_each (int how_many_items);
```

它告诉调度器每个输入数据流已经消耗了‘how_many_items’的数据。

我们之所以必须调用 consume()或是 consume_each()方法是因为我们必须告诉调度器这个输入流中的多少数据已经被消耗了，这样调度表就能根据情况相应的重新排列数据流

缓存和指针的安排。这两个方法背后的细节对我们来说太琐碎了。我们只需要记住它们被GNU Radio很好的应用了并且在每次我们应用general_work()方法的时候记得调用它们就可以了。

3.5 简单介绍其它方法和成员变量

3.5.1 forecast()方法

```
virtual void forecast ( int          noutput_items,
                        gr_vector_int  &ninput_items_required);
```

forecast()方法是用来估计对于一个输出请求所需要的输入数据的。

第一个参数 noutput_items 已经在 general_work()方法中介绍过了，它是每一个输出流所产生的数据的数目。第二个参数 ninput_items_required 是一个整型向量，用来保存每路输入流所需的输入数据的个数。

当我们应用 forecast()方法时，我们需要估计每个输入数据流需要的数据数目，把这一请求给每一输出流用来产生‘ noutput_items’的数据。估计并不需要很精确，但是应该是接近的。参数 ninput_items_required 通过返回传递，这样计划中的估计值就能立即保存进去了。

我们可以在‘/src/lib/runtime/gr_block.cc’中的‘stub 1:1 implementation’看到它的定义，其中每个输入流需要的输入数据的数目只是简单的赋值给了 noutput_items。这一点对很多正规的模块是很有效的，但是对于插入器，十进制取值器以及那些输出数据数目和所需的输入之间的关系很复杂的模块效果明显却不是适当的。

3.5.2 d_output_multiple 和 set_output_multiple()方法

现在让我们介绍在 gr_block 中定义的第四个成员变量：d_output_multiple。它是用来驱使 noutput_items 参数传递给 forecast()和 general_work()的。

调度表必须确定传递给 forecast()和 general_work()的 noutput_items 数目是 d_output_multiple 的整数倍。d_output_multiple 所得缺省值是 *1*。

这是一个值得强调的严格的指针。假设我们准备设计一个模块类，在我们执行了 forecast()和 general_work()方法后，谁将会用或者调用这些方法，又怎么调用呢？特别重要的是，什么数值会被传递给 noutput_items，又是谁做这项工作？一个简单的回答是：调度表会利用合适的理由来调用这些方法。当我们应用 forecast()和 general_work()，我们常常假想 noutput_items 以及其他的参数都已经被概念上地提供了。实际上，我们从来没有调用这些方法并且明确的设定这些数值。调度表会了解一切事情并且根据更高层的原则和缓存分配策略来调用这些方法。表面现象背后的小技巧涉及到了太多的细节并且这这些细节是远远没有必要知晓的。在我们设计自己的模块的时候大可以不要操心它们。

我们对于 noutput_items 有一些层次的控制。d_output_multiple 变量告诉调度表 noutput_items 必须是 d_output_multiple 的整数倍。

我们可以用方法 set_output_multiple*()* 设定 d_output_multiple 的值，也可以利用 output_multiple*()*方法得到它的值。

```
void    gr_block::set_output_multiple (int multiple)

{

   if (multiple < 1)

      throw std::invalid_argument ("gr_block::set_output_multiple");

   d_output_multiple = multiple;

}

int     output_multiple () const {return d_output_multiple; }
```

3·5·3 d_relative_rate 和 set_relative_rate()方法

第五个成员变量 d_relative_rate 给出了合适的与速率相关的信息，也就是说给出了合适的输入输出速率。

这样的信息给缓存分配和调度表提供了一种提示，使得它们能相应的分配缓存和调整参数。d_relative_rate 的缺省值是 1.0，大多是信号处理模块会默认这样的值。很明显，十进制取值器的 d_relative_rate 应该小于 1.0，插入器的 d_relative_rate 应该大于 1.0。

我们可以运用 set_relative_rate()方法来设定 d_relative_rate 的值，也可以用 relative_rate()得到它的值。

```
void    gr_block::set_relative_rate (double relative_rate)

{

   if (relative_rate < 0·0)

      throw std::invalid_argument ("gr_block::set_relative_rate");

   d_relative_rate = relative_rate;

}
```

```
double  relative_rate ()  const {return d_relative_rate;}
```

好的！至此，我们已经逐条介绍了类 gr_block。可以注意到我们跳过了对成员变量 d_detail 及其相关方法的介绍。它们同样被用到了，不过只用于内部应用。在我们设计我们自己的信号处理模块的时候很少会遇到它们。我们强烈推荐读者通过仔细的阅读源代码对这些东西有一个完整的理解。

4 命名惯例

在学习了类 gr_block 并且阅读了相关的源文件后，我们应该整理一下 GNU Radio 中采用的命名惯例。这些惯例会帮助我们理解基本代码，以及更好的连接 C++语言和 Python 语言。

在 GNU Radio 中，没有宏和其他的常量值，所有的标识符有类似的形式：‘words_separated_like_this’。宏和常量值应该在 UPPER_CASE 中。

4.1 包前缀

所有的看得见的全局的名字（类型，函数，变量，常量等等）都应该以一个‘包前缀’开始，紧跟着的是下划线。GNU Radio 中大部分的代码属于前缀为‘gr’的包，因此名字会类似为这样的：gr_open_file。

大的结合紧密的代码可能会有其他的前缀。这里给出一些常见到的：

gr_: Almost everything in GNU Radio

usrp_: Universal Software Radio Peripheral (USRP) related packages

qa_: Quality Assurance, used for our testing code

4.2 类数据成员（实例变量）

就像我们看到的，所有的类数据成员应该以前缀‘d_’开始。

这样做的一大好处是当你看一块代码的时候，可以很明显的看出哪些是分配在块以外的。这也能让你不必在采用创建新方法和构想的时候为参数设置新的名字。你只需要采用和成员变量一样的名字，只是去掉‘d_’就可以了。

```
class gr_wonderfulness {

  std::string    d_name;

  double         d_wonderfulness_factor;
```

```
public:

  gr_wonderfulness (std::string name, double wonderfulness_factor)

  : d_name (name), d_wonderfulness_factor (wonderfulness_factor)


  ...

};
```

所有的类静态数据成员应该以‘s_’开始，我们目前为止还没有碰到这样的数据。

4.3 文件命名

每个重要的类都应该包含在它自己的文件中。例如，类 gr_foo 的声明应该在文件 gr_foo·h 中，它的定义应该在 gr_foo·cc 中。

这是真的。
呵呵。

4.4 后缀

按照惯例，我们应用后缀编辑信号处理模块中输入输出类型的名字。这样的后缀一般长度为一两个字符。源和接收端会有单独的特别的后缀。那些有输入和输出规范的模块会有两个这样的特别后缀。第一个后缀指明了输入流的数据类型，第二个后缀指明了输出流的数据类型。FIR 滤波器模块有三个这样的特别后缀，分别指明了输入，输出和窗口（taps）的类型。

这有几个特别的后缀以及他们的解释：

f - single precision floating point

c - complex<float>

s - short (16-bit integer)

i - integer (32-bit integer)

此外，对于一个处理数据流向量的模块这样的情况，我们用特性 v 作为第一个后缀的特性。这样应用的一个例子是 gr_fft_vcc。这个 FFT 模块将一个复数向量作为它的输入并且在输出端产生一个复数向量。

5 我们的第一个模块：howto_square_ff

让我们尝试完成我们的第一个模块：howto_square_ff。它简单的计算输入数据的平方：y[n]=x2[n]。

我们直接引申自 gr_block 来得到这样的一个模块并且特别关注像 general_work()这样的方法。源代码可以在以下的地址找到：

/gr-howto-write-a-block/src/lib/howto_square_ff.h (.cc)

也可以从附录 B 和附录 C 中找到源代码。
让我们着重研究代码中几个关键点。

5.1 构造函数

首先让我们看一看它的构造函数，其中有一些小把戏：

在 howto_square_ff.h 中：

```
…

typedef boost::shared_ptr<howto_square_ff> howto_square_ff_sptr;

howto_square_ff_sptr howto_make_square_ff ();

class howto_square_ff : public gr_block

{

private:

   friend howto_square_ff_sptr howto_make_square_ff ();

   howto_square_ff ();

   …

}

…
```

在 howto_square_ff.cc 中：

```
…

howto_square_ff_sptr howto_make_square_ff ()

{

  return howto_square_ff_sptr (new howto_square_ff ());

}

…

static const int MIN_IN = 1;   // mininum number of input streams

static const int MAX_IN = 1;   // maximum number of input streams

static const int MIN_OUT = 1;  // minimum number of output streams

static const int MAX_OUT = 1;  // maximum number of output streams


howto_square_ff::howto_square_ff ()

  : gr_block ("square_ff",

        gr_make_io_signature (MIN_IN, MAX_IN, sizeof (float)),

        gr_make_io_signature (MIN_OUT, MAX_OUT, sizeof (float)))

{

  // nothing else required in this example

}

…
```

我们刚刚讨论过灵活的指针 Boost。在 GNU Radio 中，我们仅仅使用‘boost::shared_ptr’这样一种灵活指针而不是生涩的 C++指针作为‘gr_’开头模块（还有其他的数据结构）的接入口。所以为了不小心使用了 C++的指针我们定义 howto_square_ff 的构造函数为私有的。私有的构造函数在类之外是无法被调用的。所以下面这样应用

howto_square_ff 的例子是非法的：

howto_square_ff* test_block = new howto_square_ff()

因为 howto_square_ff 的构造函数不是公共的。这样做我们可以避免一个模块被一个 C++指针指向的可能性。

此外，我们应用函数 howto_make_square_ff()作为公共的接口去产生新的例子。首先，它被定义为类 howto_square_ff 的友函数，这样就可以接入 howto_square_ff 定义的所有私有成员变量和方法。

```
friend howto_square_ff_sptr howto_make_square_ff ();
```

之后在函数 howto_make_square_ff()中，我们调用类 howto_square_ff 的私有构造函数，用 new 命令产生一个对象。然而，我们将返回指针的数据类型从 c++的指针变为灵活的指针 howto_square_ff_sptr：

```
howto_square_ff_sptr howto_make_square_ff ()
{
  return howto_square_ff_sptr (new howto_square_ff ());
}
```

做了这些小把戏后，我们实际上可以确保所有的‘gr_’的模块在最初定义时就被灵活的指针 boost::shared_ptr 指向了。函数 howto_make_square_ff()被用来作为产生其他对象的公共接口。

这样的小把戏存在于大多数‘gr_’模块和很多其他的数据结构中。让我们看看 howto_square_ff 的私有构造函数：

```
static const int MIN_IN = 1;  // minimum number of input streams
static const int MAX_IN = 1;  // maximum number of input streams
static const int MIN_OUT = 1;  // minimum number of output streams
static const int MAX_OUT = 1;  // maximum number of output streams

howto_square_ff::howto_square_ff ()
```

```
: gr_block ("square_ff",
      gr_make_io_signature (MIN_IN, MAX_IN, sizeof (float)),
      gr_make_io_signature (MIN_OUT, MAX_OUT, sizeof (float))) {}
```

我们调用 gr_block 的构造函数来要求类的名字作为第一个参数，输入输出作为第二个和三个参数。第二个和第三个参数实际上都是灵活的指针类型 gr_io_signature_sptr，指向了 gr_io_signature 的实例。相似的，gr_io_signature 的构造函数也是私有的。友函数 gr_make_io_signature（）被用作公共的接口来产生新的‘signatures’，并返回指向这些 signatures 的灵活指针 gr_io_signature_sptr。按照惯例，后缀‘_sptr’指明了是灵活的指针 boost::shared_ptr。

在我们的模块 howto_square_ff 中，我们仅仅需要一个输入流和一个输出流。所以输入输出流数目的最大值和最小值都被设置为 1·输入输出流中的数据的数据类型都是浮点型。这一点我们类名字的后缀‘_ff’也可以反映出来。

5.2 突出核心的方法：general_work ()

```
int howto_square_ff::general_work (int                  noutput_items,
                   gr_vector_int        &ninput_items,
                   gr_vector_const_void_star  &input_items,
                   gr_vector_void_star       &output_items)
{
  const float *in = (const float *) input_items[0];
  float *out = (float *) output_items[0];

  for (int i = 0; i < noutput_items; i++) {
    out[i] = in[i] * in[i];
  }
```

```
// Tell runtime system how many input items we consumed on each input stream·

  consume_each (noutput_items);


// Tell runtime system how many output items we produced·

  return noutput_items;

}
```

这是真实的信号处理部分。它的应用是很简单和直接的。因为我们的模块只有一个输入流和一个输出流，指针向量 input_items 和 output_items 只包含一个入口。我们逐个计算输出数据流中的数据。

```
for (int i = 0; i < noutput_items; i++) {

    out[i] = in[i] * in[i];

}
```

不要忘记调用 consume（）方法或 comsume_each（）方法，它们告诉运行时间系统每个输入流中的多少输入数据已经被处理了。在我们一入一出的情况下，输入流中处理的数据只是简单的等于输出流产生的数据（noutput_items）就可以了。

最后，我们返回产生的输出数据的数目。注意到在这个例子中，我们没有强调 forecast（）方法。这是因为缺省的一入一出的应用对于我们的模块是合理的。

好的！我们已经做完了！我们已经得到了自己的第一个模块 hwoto_square_ff。

6 结论

现在剩下的工作就是完成 C++于 Python 直接的连接。让我们休息一下把这项工作留给下一章节。

在本章中，我们细致的模拟分析了所有类中最基础的类：gr_block。顺道我们还介绍了灵活的指针 Boost 和 GNU Radio 中的命名习惯。这些东西是创造一个信号处理模块的基础。在下一章教程中，我们会谈一谈连接 C++与 Python 的技巧，这样本文中我们创造的模块就能在 Python 中以一种简单的方式被利用。

附录 A：gr_block·h 的源代码

```
#ifndef INCLUDED_GR_BLOCK_H

#define INCLUDED_GR_BLOCK_H


#include <gr_runtime·h>

#include <string>


class gr_block {


public:


   virtual ~gr_block ();


   std::string name () const {return d_name ;}

   gr_io_signature_sptr input_signature () const {return d_input_signature;}

   gr_io_signature_sptr output_signature () const {return d_output_signature;}

   long unique_id () const {return d_unique_id; }


   virtual void forecast (int            noutput_items,

                          gr_vector_int   &ninput_items_required);
```

```
    virtual int general_work (int                       noutput_items,
                              gr_vector_int             &ninput_items,
                              gr_vector_const_void_star  &input_items,
                              gr_vector_void_star       &output_items) = 0;

    virtual bool check_topology (int ninputs, int noutputs);

    void set_output_multiple (int multiple);
    int  output_multiple () const {return d_output_multiple ;}

    void consume (int which_input, int how_many_items);

    void consume_each (int how_many_items);

    void  set_relative_rate (double relative_rate);

    double relative_rate () const { return d_relative_rate; }

private:

    std::string            d_name;
    gr_io_signature_sptr    d_input_signature;
```

```
    gr_io_signature_sptr    d_output_signature;
    int                     d_output_multiple;
    double                  d_relative_rate;    // approx output_rate / input_rate
    gr_block_detail_sptr    d_detail;       // implementation details
    long                    d_unique_id;        // convenient for debugging

protected:

    gr_block (const std::string &name,
              gr_io_signature_sptr input_signature,
              gr_io_signature_sptr output_signature);

    void set_input_signature (gr_io_signature_sptr iosig){
         d_input_signature = iosig;
         }

    void set_output_signature (gr_io_signature_sptr iosig){
         d_output_signature = iosig;
         }

public:

    gr_block_detail_sptr detail () const {return d_detail ;}
```

```
    void set_detail (gr_block_detail_sptr detail) {d_detail = detail ;}

};


    long gr_block_ncurrently_allocated ();


#endif /* INCLUDED_GR_BLOCK_H */
```


附录 B：howto_square_ff.h 的源代码

```
#ifndef INCLUDED_HOWTO_SQUARE_FF_H

#define INCLUDED_HOWTO_SQUARE_FF_H
```

```
#include <gr_block.h>

class howto_square_ff;

typedef boost::shared_ptr<howto_square_ff> howto_square_ff_sptr;

howto_square_ff_sptr howto_make_square_ff ();

class howto_square_ff: public gr_block
{
private:
  friend howto_square_ff_sptr howto_make_square_ff ();

  howto_square_ff ();  // private constructor

public:
  ~howto_square_ff ();  // public destructor
  int general_work (int                       noutput_items,
                    gr_vector_int             &ninput_items,
                    gr_vector_const_void_star  &input_items,
                    gr_vector_void_star       &output_items);
```

```
};

#endif /* INCLUDED_HOWTO_SQUARE_FF_H */
```

附录 C：howto_square_ff·cc 的源代码

```
#ifdef HAVE_CONFIG_H
#include "config·h"
#endif

#include <howto_square_ff·h>
#include <gr_io_signature·h>

howto_square_ff_sptr
howto_make_square_ff ()
{
   return howto_square_ff_sptr (new howto_square_ff ());
}

static const int MIN_IN = 1;  // mininum number of input streams
static const int MAX_IN = 1;  // maximum number of input streams
static const int MIN_OUT = 1;  // minimum number of output streams
static const int MAX_OUT = 1;  // maximum number of output streams

howto_square_ff::howto_square_ff ()
  : gr_block ("square_ff",
        gr_make_io_signature (MIN_IN, MAX_IN, sizeof (float)),
```

```
        gr_make_io_signature (MIN_OUT, MAX_OUT, sizeof (float)))
{
   // nothing else required in this example
}

howto_square_ff::~howto_square_ff ()
{
   // nothing else required in this example
}

int  howto_square_ff::general_work (int                  noutput_items,
                        gr_vector_int          &ninput_items,
                        gr_vector_const_void_star   &input_items,
                        gr_vector_void_star        &output_items)
{
   const float *in = (const float *) input_items[0];
   float *out = (float *) output_items[0];

   for (int i = 0; i < noutput_items; i++) {
      out[i] = in[i] * in[i];
   }
```

```
    consume_each (noutput_items);

    // Tell runtime system how many output items we produced.
    return noutput_items;
}
```

参考文献

[1] Eric Blossom, How to Write a Signal Processing Block, http://www.gnu.org/software/gnuradio/doc/howto-write-a-block.html

[2] Python on-line tutorials, http://www.python.org/doc/current/tut/

[3] Eric Blossom, Exploring GNU Radio, http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html

脚注

作者来自巴黎圣母院大学电子工程系网络通信与信号处理实验室。欢迎发送你的评论和要求到如下地址：Email: dshen@nd.edu。

指南 10：为 GNU Radio 写一个信号处理模块（第二部分）

摘要

本章继续讨论如何为 GNU Radio 编写一个信号处理模块，集中讨论如何连接 C++程序和 Python 程序。很有在 GNU Radio 中很有用的小细节也会被介绍到。

1·回顾

在上一章教程中，我们已经提到了编写一个新的信号处理模块需要涉及到三个文件：定义新模块的·h 和·cc 文件以及告诉 SWIG 如何产生 C++类到 Python 连接的·i 文件。我们已经完成了 howto_square_ff·h 和 howto_square_ff·cc，现在我们要完成‘·i’文件。之后，我们应该重新正确的安排目录布局，把这些文件放到正确的位置上。最后，我们需要一个叫‘Makefile·am’的文件构建所有这些东西。

我们都很讨厌编写文件时出现大量的非法访问数据库和系统的活动。幸运的是，在 GNU 中我们可以用一些自动工具降低这样的复杂度。好消息是 GNU Radio 提供了能够被灵活应用的模版文件。

当你正在阅读本教程时，我们建议你下载下教程 gr-howto-write-a-block 并看一看我们将要介绍的那些文件。

2·SWIG 文件：how·i

SWIG，简单的打包器和接口形成器，用来产生一个连接，使我们的代码可以在 Python 中被利用。我们需要写一个·i 文件告诉 SWIG 连接的产生步骤。

一个·i 文件可以看作是一个·h 文件的缩减版本，除了多一点让 Python 和指针 boost::shared_ptr 合作的内容。为了使代码不至太臃肿，我们仅仅涉及那些我们想从 Python 接入的方法。

我们即将调用·i 文件 howto·i，并用它建立 SWIG 声明给所有前缀为‘howto_’的类，使这些类可以从 Python 进入。‘howto_’类之后在 Python 中构成一个包名称。这个·i 文件是很短小的：

```
1 /* -*- c++ -*- */
2
3 %feature ("autodoc", "1");      // generate python docstrings
4
5 %include "exception·i"
6 %import "gnuradio·i"           // the common stuff
7
8 %{
9 #include "gnuradio_swig_bug_workaround·h"  // mandatory bug fix

10 #include "howto_square_ff·h"          // the header file


11 #include <stdexcept>
12 %}


13 // ----------------------------------------------------------------
14 /*
```

```
15 * GR_SWIG_BLOCK_MAGIC does some behind-the-scenes magic so we can
16 * access howto_square_ff from Python as howto.square_ff()
17 * First argument "howto" is the package prefix.
18 * Second argument "square_ff" is the name of the class minus the prefix.
19 */
20 GR_SWIG_BLOCK_MAGIC(howto, square_ff);
21
22 /*
23  * howto_make_square_ff is the friend function of class howto_square_ff
24  * It's the public interface for creating new instances
25  */
26 howto_square_ff_sptr howto_make_square_ff();
27
28 /*class declaration*/
29 class howto_square_ff : public gr_block
30 {
31 private:
32   howto_square_ff();        //private constructor
33 };
```

这样就把这个文件作为一个模版并且忽略所涉及到的细节。红色的部分不要改动，蓝色的部分按照相同的公式产生合适的内容代替。注释已经作为内容加入代码中了。请仔细的阅读它们。

注意到 GR_SWIG_BLOCK_MAGIC 做了一些有魔力的事，因此我们可以在 Python 中通过 howto.square_ff()接入 howto.square_ff。从 Python 的角度看，howto 是一个包，

square_ff()是 howto 定义的一个函数。调用这个函数会返回一个指向新产生的实例 howto·square_ff 的灵活的指针 howto_square_ff_sptr。

3. 目录布局

接下来，我们需要正确的认识这些文件，把它们放到正确的位置。下表一：目录布局展现了目录的布局和我们常常会用到的文件。在重命名 topdir 目录后。我们就能在自己的工程中应用它们了。

表一：目录布局

File/Dir Name	Comments
topdir/Makefile·am (u)	Top level Makefile·am
topdir/Makefile·common (u)	Common fragment included in sub-Makefiles
topdir/bootstrap (u)	Runs autoconf, automake, libtool first time through
topdir/config (u)	Directory of m4 macros used by configure·ac
topdir/configure·ac (s)	Input to autoconf
topdir/src	
topdir/src/Makefile·am (u)	
topdir/src/lib	C++ code goes here, including ·h ·cc and ·i files
topdir/src/lib/Makefile·am (m)	
topdir/src/python	Python code goes here, for testing and `make check'

topdir/src/python/Makefile·am (s)	
topdir/src/python/run_tests (u)	Script to run tests in the build tree

为了减少我们要做的工作，拷贝整个文件夹作为我们自己的工作区间，只按照我们的需要修改相应的文件是一个不错的主意。此外，如果你觉得生成文件时的那些把戏很烦人的话，大可以把这些文件看成是模版，只在正确的地方用相应的代码替换就可以了。

在上表一中，符号‘u’的意思是‘未改变的’。不需要去修改这些地方。深蓝色的文件以‘S’表示的需要轻微的改动，可能只需要改动一两个小地方。唯一的以‘m’标识的文件/src/lib/Makefile·am，是我们真正的工作所在。稍后我们会看看这些文件。

在此之前我们需要重点了解几件事情。

3.1 构建需要的自动工具

让我们简单谈一谈整个的构建环境，并且解释一下我们即将用到的目录布局中列出的文件中的函数。

为了减少我们不得不做的一些生成文件的技巧，并且促使可移植性的接入各种系统，我们需要用到 GNU 系统中的 autoconf，automake 和 libtool 工具。它们被集中地看成是 autotools，并且一旦你克服了最初的困难，它们会成为你的得力助手的。好消息是我们可以把上面列出来的文件当作模版，不需要太多的改动。

3·1·1 Automake

Automake 和 configure 合作产生确保 GNU 按照相应的 Makefile·am 文件从很高的描述层次上产生顺从的构造文件。Makefile·am 用来指定要构建的库文件和程序以及要组合它们需要的源文件。Automake 读取 Makefile·am 中的内容后产生文件 Makefile·in。Configure 读取 Makefile·in 之后产生 Makefile。最后产生的 Makefile 中包含极多的规则来确保在构建，检测，安装你的代码的时候做出绝对正确的事。如果最后产生的 Makefile 是 Makefile·am 的五到六倍大也不是不正常的。

3.1.2 Autoconf

Autoconf 读取 configure.ac 文件并产生配置脚本文件。Configure 自动的检测其下系统的功能，在 Makefile 和你的 C++代码中找到一大堆会用到的变量和定义来制约构建过程。如果某个需要的功能没有被找到，配置过程会暂停并弹出错误信息。

3.1.3 Libtool

Libtool 在背后工作并提供可以在各种不同的系统间共享库文件的手段。

3.2 构建树和安装树

构建树是 topdir（包含 configure.ac 的文件夹）中的所有东西，表一中的目录布局实际上形成了构建树。安装树的路径是：prefix/lib/python2.x/site-packages。其中 prefix 是配置文件（缺省的是/usr/local）的后缀，2.x 是 Python 的安装版本。一个安装树的典型路径是：/usr/local/lib/python2.3/site-packages。

我们通常设置我们的 PYTHONPATH 环境变量指向安装树，并且在~/.bash_profile 里面做这个工作。这就允许 Python 接入所有的标准库中，除了我们本地安装的像 GNU Radio 的东西。

在完成了编写我们自己的信号处理模块代码的工作后，我们的工作应该按照安装 GNU Radio 时所做的步骤：

```
$ ./bootstrap

$ make

$ make check

$ make install
```

注意到要想执行./bootstrap 成功，你必须安装上面提到的自动工具：Automake，Autoconf 和 Libtool。Bootstrap 仅仅是一个用于调用这些工具去执行配置和生成文件的脚本。

3.3 检查

我们编写自己的应用是为了接入安装树中的代码和库文件的。另一方面，我们想让我们的测试代码能在构建树中执行，由此我们能在安装前发现问题所在。这就是 'make check' 真正所做的。

为了做这些，我们需要编写一小段后缀为‘qa_’的 Python 测试代码，并把它放在目录/src/python 下。‘qa’代表了‘Quality Assurance’。在这个 Python 脚本中我们可以利用`howto·square_ff()'来检测我们刚刚编写的模块，看看它能否正确工作。

之后我们用 make check 来执行我们的测试。Make check 调用这个/src/python/run_tests 脚本文件，它能安装 PYTHONPATH 环境变量并使我们的测试可以利用我们代码和库文件中相应的构建树版本。之后会运行所有类似 qa_*·py 格式名字的文件，并汇报完全正确或者出错。只有很少的背后的动作会要求用到我们代码中未安装的部分，你可以查看 run_tests 得到直接的结果。

4·改变配置和生成文件

4.1 /src/lib/Makefile.am

这个文件是需要最多工作量的地方，这里有一个模版，你也可以在这里/gr-howto-write-a-block/src/lib/Makefile·am 找到。

```
1 include $(top_srcdir)/Makefile·common
2
3 # Install this stuff so that it ends up as the gnuradio·howto module
4 # This usually ends up at:
5 #   ${prefix}/lib/python${python_version}/site-packages/gnuradio
6
7 ourpythondir = $(grpythondir)
8 ourlibdir    = $(grpyexecdir)
```

```
9
10 INCLUDES = $(STD_DEFINES_AND_INCLUDES) $(PYTHON_CPPFLAGS)
11
12 SWIGCPPPYTHONARGS = -noruntime -c++ -python $(PYTHON_CPPFLAGS)
\
13         -I$(swigincludedir) -I$(grincludedir)
14
15 ALL_IFILES =                       \
16         $(LOCAL_IFILES)             \
17         $(NON_LOCAL_IFILES)
18
19 NON_LOCAL_IFILES =                 \
20         $(GNURADIO_CORE_INCLUDEDIR)/swig/gnuradio·i
21
22 # The ·i file comes here
23 LOCAL_IFILES =                     \
24         howto·i
25
26 # These files are built and by SWIG·
27 # The first is the C++ glue·
28 # The second is the python wrapper that loads the _howto shared library
```

```
29 # and knows how to call our extensions·
30
31 BUILT_SOURCES =                    \
32         howto·cc                    \
33         howto·py
34
35 # This gets howto·py installed in the right place
36 ourpython_PYTHON =                  \
37         howto·py
38
39 ourlib_LTLIBRARIES = _howto·la
40
41 # These are the source files that go into the shared library
42 _howto_la_SOURCES =                 \
43         howto·cc                    \
44         howto_square_ff·cc
45
46 # magic flags
47 _howto_la_LDFLAGS = -module -avoid-version
48
49 # link the library against some common swig runtime code and the
50 # c++ standard library
```

```
_howto_la_LIBADD =                  \
        -lgrswigrunpy               \
        -lstdc++

howto.cc howto.py: howto.i $(ALL_IFILES)
        $(SWIG) $(SWIGCPPPYTHONARGS) -module howto -o howto.cc $<

# These headers get installed in ${prefix}/include/gnuradio
grinclude_HEADERS =                 \
        howto_square_ff.h

# These swig headers get installed in ${prefix}/include/gnuradio/swig
swiginclude_HEADERS =               \
        $(LOCAL_IFILES)

MOSTLYCLEANFILES = $(BUILT_SOURCES) *.pyc
```

这个 Makefile.am 文件构建了所有事情并且将从源中构建共享库。它也合并了 SWIG，为了共享库添加了连接。特别注意一下包含‘howto’的部分，在你构建其他模块的时候把它替换为合适的内容就可以了。

4.2 topdir/configure.ac

只有最初的几行需要小心的修改：

```
AC_INIT
```

```
2 AC_PREREQ(2.57)


3 AC_CONFIG_SRCDIR([src/lib/howto.i])


4 AM_CONFIG_HEADER(config.h)

5 AC_CANONICAL_TARGET([])


6 AM_INIT_AUTOMAKE(gr-howto-write-a-block,0.3)
```

仅仅修改蓝色的部分就可以了。

4.3 /src/python/Makefile.am

这是一个十分简单的文件

```
1 include $(top_srcdir)/Makefile.common

2

3 EXTRA_DIST = run_tests.in

4

5 TESTS =               \

6     run_tests


7 noinst_PYTHON =           \

qa_howto.py
```

所有在‘noinst_PYTHON’列出来的文件都不会被编译。这里仅仅列出你的 Python 测试文件。Qa_howto.py 是我们例子中的 Python 测试脚本，稍后会提到它。

5·Python 测试脚本：qa_howto·py

现在让我们编写一个 Python 测试脚本，它应该在/src/python/中。文件名应该以_qa 起头。在 make check 时它会被执行。

```
1 #!/usr/bin/env python
2
3 from gnuradio import gr, gr_unittest
4 import howto
5
6 class qa_howto (gr_unittest·TestCase):
7
8     def setUp (self):
9         self·fg = gr·flow_graph ()
10
11     def tearDown (self):
12         self·fg = None
13
14     def test_001_square_ff (self):
15         src_data = (-3, 4, -5·5, 2, 3)
16         expected_result = (9, 16, 30·25, 4, 9)
17         src = gr·vector_source_f (src_data)
18         sqr = howto·square_ff ()
19         dst = gr·vector_sink_f ()
```

```
20        self·fg·connect (src, sqr)
21        self·fg·connect (sqr, dst)
22        self·fg·run ()
23        result_data = dst·data ()
24        self·assertFloatTuplesAlmostEqual (expected_result, result_data, 6)
25
26  if __name__ == '__main__':
gr_unittest·main ()
```

gr_unittest 是标准 Python 模块 Unittest 的延伸，gr_unittest 增加了对检查相近的浮点复数元组的支持。Unittest 利用 Python 的映射机制找到所有以 test_开头的方法并执行它们。Unittest 用匹配的调用 setUp 和 tearDown 把每一个调用包装成 test_*。可以查看 Python 文档 Unittest 或是 Python 文件 sitepackages/gnuradio/gr_unittest·py 得到更多这方面的细节。

当我们执行测试时，gr_unittest·main ()就将准备调用 setUp，test_001_squre_ff 和 tearDown。

test_001_squre_ff 建立了一个包含三个节点的小的图。gr·vector_source_f(src_data)将提供 src_data 元素的源并且在完成时给出告知。howto·square_ff 是我们将要测试的模块。gr·vector_sink_f 聚集了 howto·square_ff 的输出结果。

Run 方法直到所有模块告知完成之后才执行图。最后，我们检查所有在 src_data 上执行的 square_ff 的结果是否和我们的预想一致。

Qa_howto·py 文件为编写测试脚本提供了一个非常好框架。我们常常可以充分发挥 gr_unittest 的优点去设计我们自己的 Python 测试文件。

6 我们完成了！

好的，这就是我们需要的所有东西了。·/bootstrap，configure 然后 make，所有的事

情应该成功的构建。我们会得到一些警告不过是正常的。把路径改为 src/python，之后试着 make check，我们应该能通过这个测试。

```
sachi@dshen:~/gr-howto-write-a-block/src/python$ make check
make  check-TESTS
make[1]: Entering directory `/home/sachi/gr-howto-write-a-block/src/python'
..
----------------------------------------------------------------------
Ran 2 tests in 0.025s

OK
PASS: run_tests
==================
All 1 tests passed
==================
make[1]: Leaving directory `/home/sachi/gr-howto-write-a-block/src/python'
```

好极了！我们已经有一个新的工作中的模块了。

7.结论

让我们再歇一歇，现在这个时候，你应该已经可以编写自己的模块并且知道怎么去测试它，之后把它构建到 GNU Radio 树中。在下一个教程中，我们会介绍 gr_block 的一些子类并且访问再次我们自己的例子 howto_square_ff()。

参考文献

[1] Eric Blossom, How to Write a Signal Processing Block,

http://www.gnu.org/software/gnuradio/doc/howto-write-a-block.html

[2] Python on-line tutorials, http://www.python.org/doc/current/tut/

[3] Eric Blossom, Exploring GNU Radio,

http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html

脚注

作者来自巴黎圣母院大学电子工程系网络通信与信号处理实验室。欢迎发送你的评论和要求到如下地址：Email: dshen@nd.edu。