TOC

Environment Building
Toolset preparation
Driver Compilation
Main Program **FAILED**
Jupyter Receiving Test
Pre-built Python RTL-SDR Wrapper
PSD of 4 Receiver using SciPy **SLOW**
MUSIC Algorithm Simplification
Practical Steps of Simplified MUSIC
FPGA Implementation of Modules
Module Architecture
Jacobi based EVD algorithm
Implementation of Single-Sided Jacobi Method
Helpful References

Compiling Kerberos Toolsets on PYNQ Ubuntu

2021.2.21 当前进度: 在 PYNQ 上安装 Kerberos FPGA 驱动与主程序

`git clone` Kerberos SDR 的驱动到 PYNQ 上然后直接编译安装:

```
1  git clone https://github.com/rtlsdrblog/rtl-sdr-kerberos
       ~/kerberos_sdr
2  cd ./kerberos_sdr && make -j4
3  sudo make install
4  sudo ldconfig
```

该步骤顺利完成, 未发生任何错误. 使用 build 得到的`rtl_test -d 0` 与`rtl_fm 104200000 -r` 指令测试, 能正常获取 Kerberos SDR 设备的 4 个 RTL-SDR 并正常收听 FM 广播.

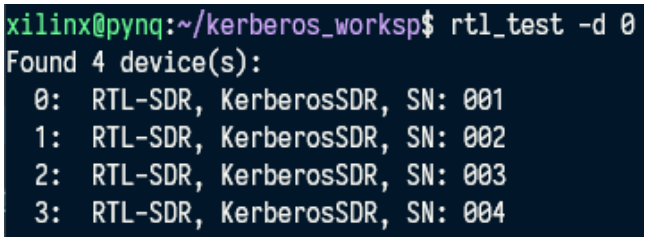

Figure: RTL-SDR Detection Successful

Compiling Main Toolset

为减小代码量，同时由于 PYNQ 无法安装 GTK 图形界面，将主程序中的接收机部分下载后通过 makefile 编译:

```
1        cd ./kerberos_sdr/recv/C
2        make -j4
```

编译后发现得到的可执行文件无法运行. 查看 makefile 中指定的编译选项与 Zynq 7020 芯片的 datasheet 后，作者认为编译选项没有任何问题. 可能是由于代码缺少必要依赖项导致编译链接得到的程序无法正常运行.

```
xilinx@pynq:~/kerberos_worksp$ ls ~/kerberos_worksp/
gate  gate.c  hydra_recv.py  Makefile  rtl_daq  rtl_daq.c
xilinx@pynq:~/kerberos_worksp$ ./rtl_daq
[ INFO ] Starting multichannel coherent RTL-SDR receiver
Segmentation fault (core dumped)
xilinx@pynq:~/kerberos_worksp$ ./sim
Hydra test started
Could not open file
Could not open file
Could not open file
Could not open file
xilinx@pynq:~/kerberos_worksp$ python ./hydra_recv.py
xilinx@pynq:~/kerberos_worksp$
```

Figure: Receiver SW Compilation Failed

Python Wrapper Library of RTL-SDR APIs

Use Jupyter Notebook and Python library `pyrtlsdr` to avoid invoking the RTL-SDR manually:

```
1          sudo pip3 install pyrtlsdr numpy scipy
```

这个库通过ctype 库调用 RTL-SDR 驱动编译得到的.so 和.a 库文件. 在 Notebook 中通过 pyrtlsdr 库获取 RTL-SDR 的设备地址并读取复基带流:

```
1 from rtlsdr import *
2 serial_number = RtlSdr.get_device_serial_addresses()
3 print(serial_number)
4 sdr_ar = []
5 for i in range(0, size(serial_number)):
6     sdri = RtlSdr(RtlSdr.get_device_index_by_serial(
          serial_number[i]))
7     sdr_ar.append(sdri)
```

PSD using Scipy Welch()

在 FFT 和抽样段数较大时 (如 1024 点 FFT,256/512 段采样),SciPy 库提供的 Welch 功率谱算法运行很慢, 几乎没有实时性. 但该算法的运行结果较好, 调用语法与 matlab 类似.

```
1 n_fft = 1024
2 num_samp = 256
3 samples0 = np.array(sdr_ar[0].read_samples(num_samp * n_fft))
4 sdr_ar[0].close()
5 [f1,Pxx1]=sgn.welch(samples0,sr,nperseg=num_samp,nfft=n_fft,
      detrend=False,return_onesided=False)
6 plot(f1/1e6+cf/1e6,np.log10(Pxx1))
```

```
1     executed in 19.7s, finished 20:04:31 2021-02-21
```

PSD using SciPy Welch()

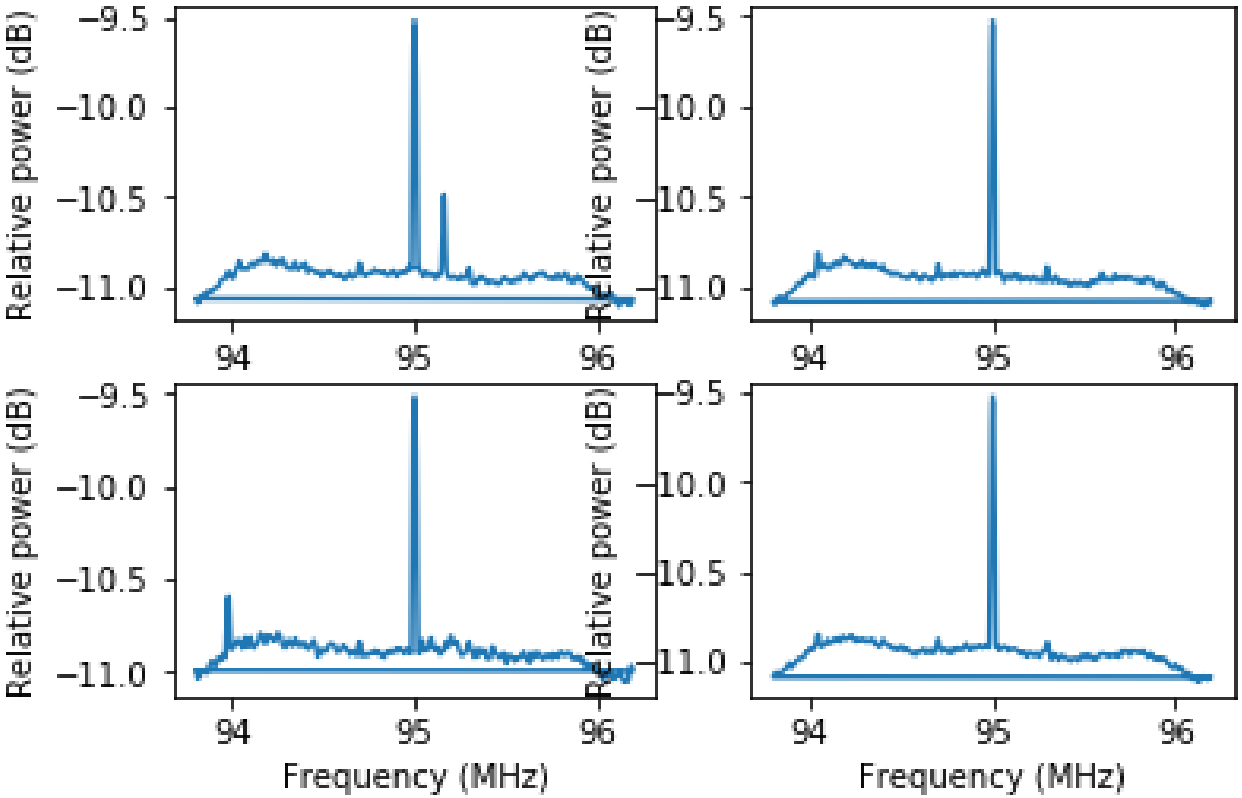

Figure: Welch PSD Estimation Result at NFFT=1024, NSegments=512

MUSIC Algorithm Simplification

在实际的 MUSIC 估计中, 有效信号的数量往往要远少于天线阵元的数量. 利用这一点, 可以大大降低信号子空间的维度, 避开繁琐的 SVD 分解 [1] [2]. 对于 M 元线阵, L 个入射信号的情形, 接收到的信号为

$$X_{out} = AS(t) + N(t) = (a(\theta_1), \ldots, a(\theta_L)) \begin{pmatrix} s_1(t) \\ \vdots \\ s_L(t) \end{pmatrix} + \begin{pmatrix} n_1(t) \\ \vdots \\ n_L(t) \end{pmatrix} \quad (1)$$

对于任意的入射信号数量 $L < M/2$, 信号矩阵可做如下分解:

$$X_{out} = (X_s(t), X'_s(t)) = (x_1(t), \cdots, x_L(t), x_{L+1}(t), \cdots, x_M(t))$$

$$a(\theta_n) = (a_{sub}(\theta_n), \cdots, a'_{sub}(\theta_n)) = \exp\left(-j\frac{2\pi d}{\lambda}(0, \cdots, L, \cdots, M-1)\sin \right. \quad (2)$$

MUSIC Algorithm Simplification

因而简化后的输出信号为

$$\begin{pmatrix} X_s \\ X'_s \end{pmatrix} = \begin{pmatrix} a_s(\theta_1) & \dots & a_s(\theta_L) \\ a'_s(\theta_1) & \dots & a'_s(\theta_L) \end{pmatrix} S + \begin{pmatrix} N_s(t) \\ N'_s(t) \end{pmatrix} \tag{3}$$

将 X_s 的系数矩阵

$$A_s = (a_s(\theta_1), \dots, a_s(\theta_L)) \tag{4}$$

扩展为较为普遍的形式

$$A_s(m) = \begin{pmatrix} \exp\left(-j\frac{2\pi d}{\lambda} 0 \sin\theta_1\right) & \dots & \exp\left(-j\frac{2\pi d}{\lambda} 0 \sin\theta_L\right) \\ \vdots & \dots & \vdots \\ \exp\left(-j\frac{2\pi d}{\lambda} m \sin\theta_1\right) & \dots & \exp\left(-j\frac{2\pi d}{\lambda} m \sin\theta_1\right) \end{pmatrix} \tag{5}$$

MUSIC Algorithm Simplification

可见 $A_s(M-L)$ 与 A'_s 存在简单的系数关系:

$$A'_s = A_s(M-L)\Lambda^L \tag{6}$$

, 其中

$$\Lambda^L = \operatorname{diag}\exp\left(-j\frac{2\pi dL}{\lambda}\sin(\theta_1,\cdots,\theta_L)\right) \tag{7}$$

求协方差去掉随机噪声, 将信号和噪声分离到正交的子空间中, 然后提取信号子空间的输出. 选择同时含有 $A_s(M-L)$ 与 $A_s(L)$ 的子矩阵:

$$\begin{aligned} R &= E((AS+N)(AS+N)^H) \\ \Rightarrow R_s &= E((A_s(M-L)\Lambda^L S+N)(A_s(L)S+N)^H) \\ &= A_s(M-L)\Lambda^L E(SS^H)A_s(L) \end{aligned} \tag{8}$$

MUSIC Algorithm Simplification

由于 rk $A_s(L) \leqslant L$, 因此上述协方差子矩阵的列空间就是 $M-L$ 维信号矢量张成的信号子空间, 对 $R_s = R(L:M, 1:L)$ 进行施密特正交化即可非常方便的获得信号子空间.
空间谱峰则通过对该信号子空间进行峰搜索得出:

$$P(\theta) = \frac{1}{a^H(\theta)(I - U_S U_S^H)a(\theta)} = \frac{1}{\|a^H\|^2 - \|a^H U_s\|^2} \tag{9}$$

$a(\theta)$ 是可以查表获得的常数, 计算得到的空间谱表则可以传回 Zynq 的 PS 端, 利用成熟的 Python 算法进行快速排序. 但是这个算法需要**预先指定**入射信号的数量, **不能对未知数量的入射信号进行盲检测**.

Architecture of System

根据之前分析得到的简化算法, MUSIC 测向系统可以被简化为一个快速协方差/自相关模块, 一个 QR 分解模块和一个模长模块. 查阅 Xilinx 官方的文档 (现已知最好的 FPGA 教程之一) 后, 发现这些模块均可通过 MATLAB coder 导出 C++ 到 HLS 实现 (4), 无需手写 Verilog RTL.

Comparison between SVD and EVD based MUSIC algorithm

由奇异值分解 (SVD) 与本征值分解 (SVD) 的关系

$$A = U\Sigma V^H \leftrightarrow KR = AA^H = U(\Sigma\Sigma^*)U^H \tag{10}$$

可知 MUSIC 算法中的本征值和本征矢只需要用 SVD 算法提取 A 矩阵的 4*4 左奇异向量 $U/\sqrt{K}$ 和奇异值中的非零对角元 $\Lambda = \|\operatorname{diag}(\text{nonzero}(\Sigma))\|^2$, 省去了矩阵乘法环节, 进一步降低了矩阵计算需要的存储器数量.
对于复信号矩阵 $X = (x_1, x_2, x_3, x_4)^T$, 可以选择 Lanczos 法和 Jacobi 法等方法实现 EVD, 而直接分解非方阵的 SVD 算法只有 Cholsky 方法. 由于 Lanczos 法效率最高,Jacobi 法最为简单, 因此考虑使用 Lanczos 法或 Jacobi 法对协方差矩阵进行 EVD 分解:

$$[U, \Sigma] = \text{Lanczos}\left(XX^H\right)/K \tag{11}$$

Converting Complex EVD to Real EVD

经考察 Lanczos 算法和 Jacobi 算法的现有 C++ 代码, 并且评估 MATLAB coder 编译生成的 Cholsky 法 C++ 代码后, 采取代码量最小、应用也最为广泛的 Jacobi 法, 以减轻片上资源负担.
由于待分解的 4*4 复矩阵为厄密矩阵 $R = XX^H/K$, 因此其所有本征值 λ_k 均为非负实数. 据此可将待分解的复矩阵和它的复本征矢拆分为实部和虚部

$$\boldsymbol{R} = \boldsymbol{A} + j\boldsymbol{B}, v_k = u_k + j\sigma_k \tag{12}$$

则对于每个 λ_k 有

$$(\boldsymbol{A} + j\boldsymbol{B})(u_k + j\sigma_k) = \lambda_k(u_k + j\sigma_k) \tag{13}$$

亦即

$$\begin{pmatrix} A & -B \\ B & A \end{pmatrix} \begin{pmatrix} u_k \\ \sigma_k \end{pmatrix} = \lambda_k \begin{pmatrix} u_k \\ \sigma_k \end{pmatrix} \tag{14}$$

这样便把 4*4 复矩阵的 EVD 转化为了便于 FPGA 处理的 8*8 实矩阵 EVD.

Single Side Jacobi Algorithm

考察特征值分解的定义式

$$R = U\Lambda U^H \Rightarrow RU = U\Lambda \tag{15}$$

由于本征矢 U 为酉矩阵,Λ 为对角矩阵, 因此

$$Q = U\Lambda \leftrightarrow q_i q_j = \lambda_{ij}^2 \delta_{ij} \tag{16}$$

可见右边为正交矩阵, 将 A 正交化的变换矩阵 U 即为欲求的本征矢,$\Lambda = QQ^H$ 即为特征值.

Single Side Jacobi Algorithm

寻找变换矩阵/本征矢 U 可以分解为一系列迭代问题. 每次正交化 A 的两列 $(a_i^T, a_j^T) \subset A = (a_1^T, a_2^T, \ldots, a_k^T)$, 变换矩阵也应当是一个酉矩阵 (以保证本征矢的单位性):

$$(a_i^T, a_j^T) \begin{pmatrix} c & -s \\ s & c \end{pmatrix} = (q_i^T, q_j^T) \tag{17}$$

由 $(q_i^T, q_j^T) \subset Q = (q_1^T, q_2^T, \ldots, q_k^T)$ 的正交性, 可得

$$\begin{aligned} 0 &= b_i^T b_j \\ &= (ca_i^T + sa_j^T)(sa_i + ca_j) \\ &= (c^2 - s^2) a_i^T a_j + cs(-a_i^T a_i + a_j^T a_j) \end{aligned} \tag{18}$$

Single Side Jacobi Algorithm

由于 $i \neq j, c^2 + s^2 = 1$, 因此上式可求得唯一解

$$c = \frac{1}{\sqrt{1+t^2}}, s = ct \tag{19}$$

其中

$$t = \frac{\text{sign}(\tau)}{\tau + \sqrt{1+\tau^2}}, \tau = \frac{\|a_i\|^2 - \|a_j\|^2}{2a_i^T a_j} \tag{20}$$

综上

$$U = \prod_{i,j<k} Q_0(i,j) = \prod_{i,j<k} \begin{pmatrix} c & -s \\ s & c \end{pmatrix} \tag{21}$$

Recovery from Real to Imaginary

实数化后分解出的 U 为 8*8 实矩阵. 对于具有同一本征值 λ_k 的两个本征矢, 经过多次 MATLAB 计算后归纳得出

$$(u_2, \sigma_2) = (-\sigma_1, u_1) \tag{22}$$

这可以由直接代入本征值的定义式验证. 据此便可以恢复出原复矩阵的基向量

$$v_k = u_k + j\sigma_k \tag{23}$$

由于特征子空间的基向量存无穷多组解, 基向量个数等于特征值个数, 因此 Cholsky 算法或 Jacobi 算法获得的基向量可以收敛到相近的任何一组.

Recovery from Real to Imaginary

这一点可以通过如下的 MATLAB 代码进行验证:

```
C1=randn([3,2])+1j*randn([3,2]);
C=C1*C1';
[EV,L]=eig(C);
A=real(C);B=imag(C);
D=[[A,-B];[B,A]];
writematrix(reshape(D,1,[]),"matD.txt","Delimiter",',')
[EV1,L1]=cpp_eig(D); % 用Jacobi程序编译成mex文件计算特征值
F=EV1(:,1:2:length(EV1)-1);
F=F(1:(length(EV1)/2),:)+1j*F((length(EV1)/2+1):length(EV1),:);
rref([EV;F])
^^I
```

对实矩阵分解还原的特征向量矩阵 F 和直接分解复矩阵得到的复特征向量矩阵 EV 进行高斯消元, 发现矩阵 (E;F) 的秩为 4. 任意一个特征向量矩阵的秩均为 4, 因此可以判定 E 和 F 均为同一个特征子空间的不同基矢量.

C++ Implementation Using Existing Code

依据现有的代码 (https://blog.csdn.net/zhangchaoyangsun/article/details/8470267) 进行改进. 现有的这个实现采用了 Matrix.h 封装的矩阵类, 并且利用 CORDIC 方法计算变换矩阵的两个独立矩阵元.
定义矩阵类的 Matrix.h 头文件中有大量未使用的冗余方法和较为复杂的封装, 因此在 Vitis HLS 中编译综合后占用了大量的片上资源. 因此, 重新编写之前测试通过的 Jacobi EVD 代码, 自行编写一个仅包含 2-范数、点乘和元素数量的矩阵类.

Optimization of SSJM under HLS Environment

https://blog.csdn.net/chenaiyanmie/article/details/80011244(使用 memcpy 进行优化的 C++ 代码)
在一般的 CPU 上有高度优化的 memcpy 指令和 SIMD 运算指令, 可以快速进行 4 个双精度数的乘加运算. 而在 Vitis HLS 综合中, 则需要通过 HLS 指令 (Directives) 指定编译器将不适合 FPGA 处理的循环等结构转换为适合 FPGA 处理的流水线结构.
同时, 传入的数据类型也需要修改成行主序存储的一维 vector, 便于 python 端进行不间断调用. 这一点要求彻底重构当前的 C++ 代码.(正在进行中)

Xilinx Step-to-step Manuals

LDU Inversion Using HLS[3]: XAPP-1317
Fast Float Multiplication[4]: XAPP-1332
HLS and MATLAB coder coupling[5]: WP-452

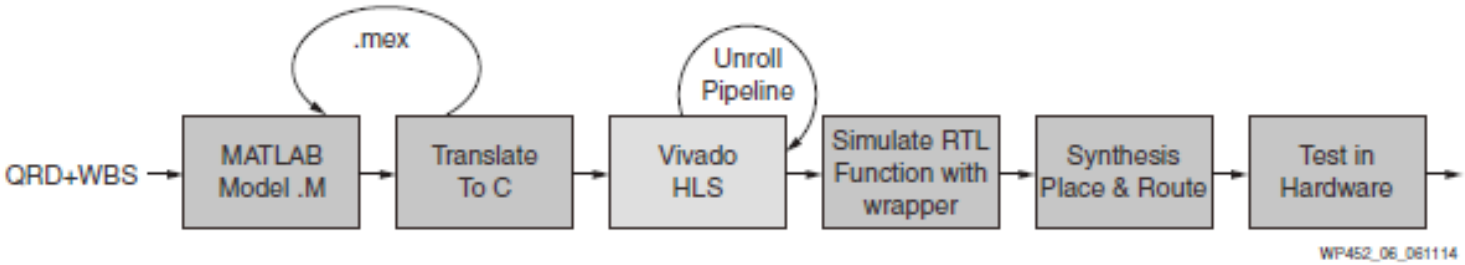

Figure 6: **Adaptive Beamformer Design Flow Using HLS**

Figure: MATLAB to HLS workflow

References


宋树田; 秦建存;, "一种改进 music 算法的 fpga 实现," 无线电工程, vol. 39, no. 09, pp. 61–64, 2009.


——, "一种快速 doa 估计算法," 无线电通信技术, vol. 35, no. 05, pp. 58–61, 2009.


M. Ruan, *Scalable Floating-Point Matrix Inversion Design Using Vivado High-Level Synthesis*. Xilinx Inc.


X. Inc., *FPGA Acceleration of Matrix Multiplication for Neural Networks*. Xilinx Inc.


L. Miller, *Adaptive Beamforming for Radar:Floating-Point QRD+WBS in an FPGA*. Xilinx Inc.