Matlab Tools for RF Pulse Design and Simulation

Amir Schricker
amirs@bme.jhu.edu

Final Project
BME 580.473: Magnetic Resonance in Medicine
The Johns Hopkins University
Fall 2001

1 Introduction

This project is a package of Matlab routines that design radiofrequency (RF) pulses used in magnetic resonance imaging (MRI). The package allows users to design and simulate the slice profiles of 90° excitation pulses. Most importantly, both the design and simulation are computed using the Shinnar Le-Roux algorithm for RF pulse design.

2 Overview of the Shinnar-Le Roux Algorithm

The Shinnar-Le Roux (SLR) pulse design algorithm, developed by Pauly *et al.*, is a method for designing slice-selective RF pulses based on discrete approximations of the Bloch equations [1]. It transforms the problem of pulse design to the problem of digital filter design, for which well-known algorithms exist; the digital filter is then converted back to an RF pulse. Specifically, using the SLR transform, the effect of an RF pulse can be described by a pair of complex-valued polynomials $\alpha_n(z)$ and $\beta_n(z)$.

$$B_1(t) \Longleftrightarrow \big(\alpha_n(z), \beta_n(z)\big)$$

The forward and inverse SLR transform will now each be considered individually.

2.1 Forward SLR Transform

The forward direction of the transform maps the RF pulse to a pair of polynomials: $B_1(t) \rightarrow (\alpha_n(z), \beta_n(z))$, for an RF pulse $B_1(t)$ consisting of n samples. The polynomials can be calculated with the following recursive formula:

$$\begin{pmatrix}\alpha_j \\ \beta_j\end{pmatrix} = \begin{pmatrix} a_j & -b_j^* \\ b_j & a_j \end{pmatrix} \begin{pmatrix}\alpha_{j-1} \\ \beta_{j-1}\end{pmatrix}$$

where a_j and b_j are functions of the RF pulse, flip angle, and axis of rotation. Once $\alpha_n(z)$ and $\beta_n(z)$ have been calculated, the slice profile after an excitation is

$$\frac{M_{xy}}{M_0} = 2\alpha^*\beta$$

2.2 Inverse SLR Transform

The inverse direction of the transform maps the pair of polynomials back into an RF pulse: $(\alpha_n(z), \beta_n(z)) \rightarrow B_1(t)$, which allows RF pulse design to be mapped to a digital filter design problem. The recursion matrix is simply the inverse of that of the forward direction.

3 Matlab Tools

This project implements the SLR algorithm in both directions, and the following table lists all the Matlab routines (`.m` files) included in this package.

Pulse Design	Slice Profile Simulation
`makerf`	`rf2ab`
`beta2alpha`	`profile`
`ab2rf`	
`ht`	
`dinf`	

3.1 Pulse Design

Since the $\beta(z)$ polynomial is proportional to sine of half of the tip angle, it is used to approximate the desired slice profile. The Parks-McClellan algorithm, a powerful tool for approximating polynomials, can be used to design the ideal $\beta(z)$. $\alpha(z)$ can then be determined uniquely from $\beta(z)$, and together, the RF pulse can be created using the inverse SLR transform.

The front-end routine in this package for designing pulses is `makerf`. Its usage is as follows:

```
rf = makerf(points,time,band,d1,d2)
```

where `points` is the number of points in the pulse, `time` is the time duration of the pulse (in msec), `band` is the bandwidth of the pulse (in kHz), and `d1` and `d2` are the in-slice and out-of-slice ripple amplitudes, respectively.

The `makerf` routine performs the following steps:

1. Take input parameters and create a digital filter.
2. Design $\alpha(z)$.
3. Create the RF pulse.

3.1.1 Digital Filter Design to Produce $\beta(z)$

The first step to creating the β polynomial is to create a digital filter using the Parks-McClellan (PM) algorithm. The built-in `remez` function in Matlab uses the PM algorithm to create an equiripple digital filter. It requires as inputs the number of taps, a vector of frequency band edges, the amplitudes of the band, and the tolerable ripple of the bands. Each of these inputs to `remez` can be computed from the parameters given to `makerf`.

3.1.2 Designing $\alpha_n(z)$

Once $\beta_n(z)$ is known, the $\alpha_n(z)$ polynomial can be found. See [1] for a detailed explanation on its derivation. Given the $\beta_n(z)$ polynomial, `b`, the `beta2alpha` function performs this step:

```
alf = beta2alpha(b)
```

3.1.3 Creating the RF Pulse

Once $\alpha_n(z)$ and $\beta_n(z)$ are both known, the inverse SLR algorithm is used to create the RF pulse. The function in this package that performs this inverse transform is `ab2rf` and has the following usage:

```
rf = ab2rf(a,b)
```

where `a` and `b` are the $\alpha(z)$ and $\beta(z)$ polynomials, respectively.

Therefore, the pulse design process can be summarized with the following flowchart:

$$\begin{tikzcd} \textit{input parameters} \arrow[r] & \boxed{\texttt{remez}} \arrow[r, "\beta"] & \boxed{\texttt{beta2alpha}} \arrow[r, "\alpha{,}\beta"] & \boxed{\texttt{ab2rf}} \arrow[r] & \textit{RF pulse} \end{tikzcd}$$

3.2 Pulse Simulation

Simulating the effect of a pulse involves the forward SLR transform. The $\alpha(z)$ and $\beta(z)$ polynomials are calculated from the pulse, and the excitation profile is simply $2\alpha^*\beta$.

The function `rf2ab` computes the forward transform, and it has the following usage:

```
[a b] = rf2ab(rf,x)
```

where `rf` is the RF pulse and `x` is the position vector of spatial positions. The output, `a` and `b`, are two separate vectors containing the α and β polynomials. The profile is finally computed with the `profile` routine:

```
mxy = profile(alpha,beta)
```

and can be plotted simply as follows:

```
plot(abs(mxy))
```

3.3 Examples

Two pulse design examples are now presented. The first one is a slice-selective 90° pulse. For an 4 ms long 90° pulse that will excite a 1.0 cm slice, at a gradient strength of 1.0 G/cm, the bandwidth of the pulse should be $B = \gamma\, G\, \Delta x = 4.3kHz$. There will be 128 points in the pulse. To design this pulse, enter:

```
>> rf = makerf(128, 4, 4.3);
```

Since no ripple amplitudes were entered, the default values of 0.01 (= 1%) were used, and the pulse is now stored in `rf`. To simulate its slice profile, the α and β polynomials corresponding to this pulse must be calculated, and a vector of spatial positions, `x`, must be specified:

```
>> x = [-10:0.1:10];
>> [a b] = rf2ab(rf,x);
```

To plot the slice profile, use the `profile` function, and then plot it versus frequency. (The `freq = x/4` converts the normalized vector to frequency (in kHz)). The RF waveform is also plotted.

```
>> mxy = profile(a,b);
>> freq = x/4;
>> plot(rf);
>> plot(freq, abs(mxy));
```

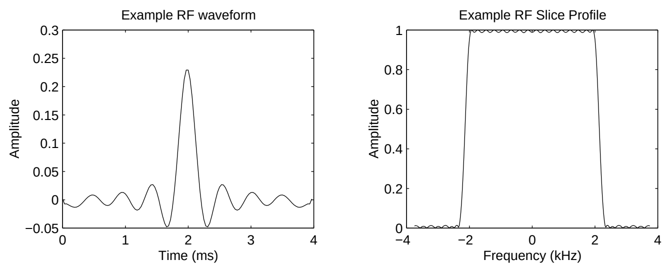


Figure 1: Plot of an example 90° pulse (a) waveform and (b) its slice profile.

The results of this plot is displayed in Figure 1.

To extend this example, another 90° pulse will be designed but with less restriction on the ripples in the passband and stopband. The same parameters as before will be used, but with 10% tolerable ripple in both bands. Design the pulse with the following command:

```
>> rf = makerf(128, 4, 4.3, 0.1, 0.1);
```

To simulate and plot its slice profile, use the same series of commands as above. The pulse waveform and slice profile are plotted in Figure 2. The large ripple amplitude is now very noticeable.

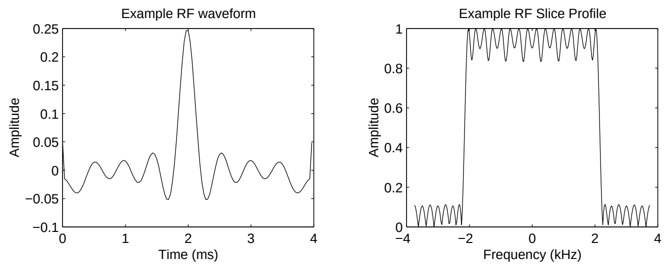


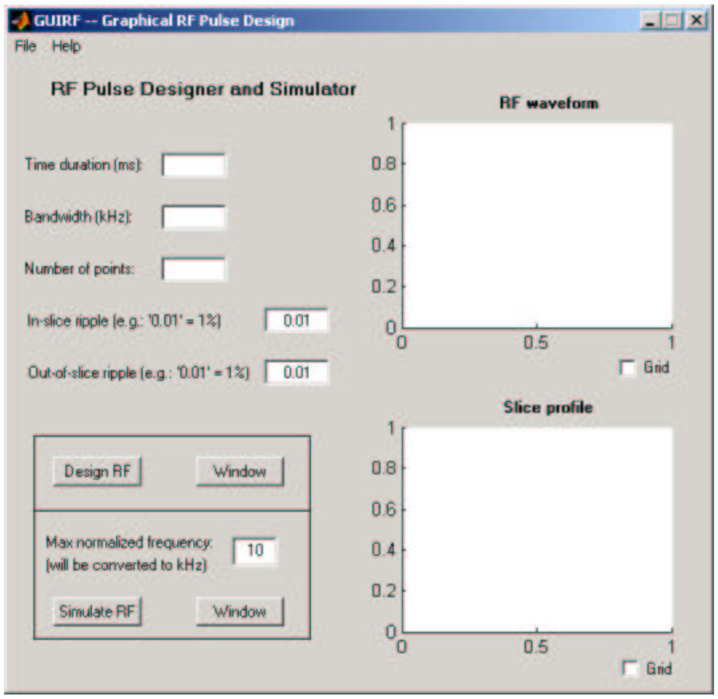
Figure 2: Plot of the second example 90° pulse (a) waveform and (b) slice profile.

4 RF Tools with the GUI

These pulse design tools can be run from the command line, but there is also a graphical user interface (GUI) that can be used to perform the same functions. The GUI can be invoked at a Matlab prompt by typing `guirf`. The following screen should appear (Fig. 3):

Pulse design is carried out using the same functions as before. Simply enter text in all the following boxes:

- *Time duration* (in msec)
- *Bandwidth* (in kHz)
- *In-slice ripple* (by default, set at 0.01 (= 1%))



Time duration (ms):

Bandwidth (kHz):

Number of points:

In-slice ripple (e.g.: '0.01' = 1%) 0.01

Out-of-slice ripple (e.g.: '0.01' = 1%) 0.01

Max normalized frequency: (will be converted to kHz) 10

Figure 3: Screen shot of `guirf`, the GUI for RF pulse design, upon startup.

- *Out-of-slice ripple* (by default, set at 0.01 (= 1%))

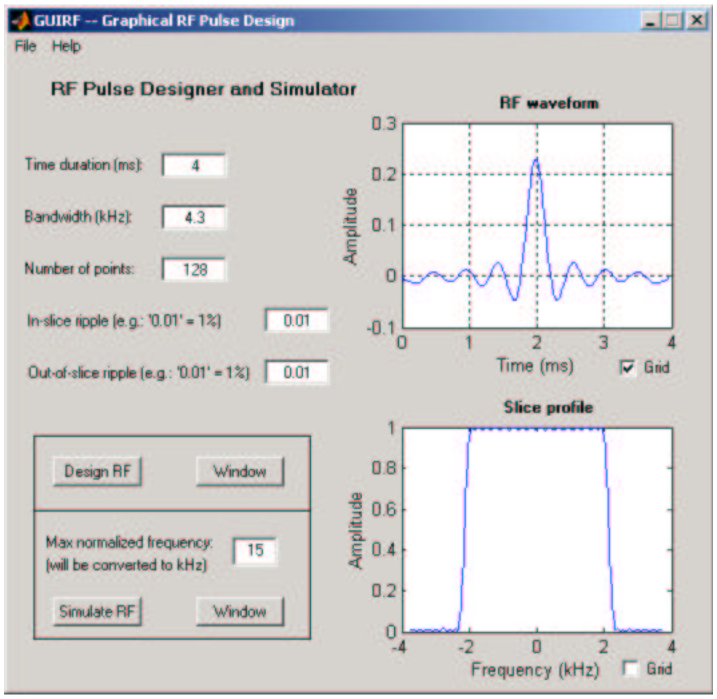
After all the fields have been selected and filled in, click the "Design RF" button; the pulse will be designed and its waveform will be plotted on the top set of axes. All of the fields, including the ripples, must be filled in with a valid number before the "Design RF" button is clicked, otherwise an error message will pop up.

To simulate this pulse, first enter a maximum frequency value (`max`, in normalized frequency units. The x-axis will then range from $-$`max:max` divided by the time duration (in ms) of the pulse; this yields the appropriate frequency axis (in kHz). The values of this maximum normalized frequency will typically be between 10 and 20. Finally, click the "Simulate RF" button and its slice profile will be plotted on the bottom set of axes.

Figure 4 displays what the above pulse design example would output after its design and simulation.

Some miscellaneous functions:

- Each of the "Window" buttons opens up a new figure window and plots either the waveform graph or slice profile graph in it. This is useful if you want to look at the plots in more detail or if you want to do other operations on the plots (such as print them, export them, etc.)
- Grids can be added to each set of axes by checking the "Grid" checkbox underneath the individual graphs.
- Go to the "Help" menu to learn some fascinating information about this GUI.
- To quit the GUI, go to the "File" menu and select "Exit".



Time duration (ms): 4

Bandwidth (kHz): 4.3

Number of points: 128

In-slice ripple (e.g.: '0.01' = 1%) 0.01

Out-of-slice ripple (e.g.: '0.01' = 1%) 0.01

Max normalized frequency: (will be converted to kHz) 15

Figure 4: The second example pulse waveform and slice profile from above.

5 Final Words

This document has described all aspects of my Matlab package for RF pulse design and simulation. For your convenience, the Matlab code (the `.m` files) of all the routines have been included in the appendix at the end.

Finally, many thanks go to John Pauly for his help with this oftentimes confusing algorithm.

Happy pulsing!

References

[1] J. Pauly, P. Le Roux, D. Nishimura, A. Macovski, Parameter relations for the Shinnar-Le Roux selective excitation pulse design algorithm. *IEEE Trans. Med. Imaging* **10**, 53-65 (1991).

Appendix: Matlab m-files

This appendix contains the Matlab code for all the `.m` files in this project. All the files are listed below and briefly explained.

1. `makerf` The front-end function to create RF pulses.
2. `ab2rf` Performs the inverse SLR recursion.
3. `beta2alpha` Calculates the α polynomial from the β polynomial. It is used in the pulse design.
4. `ht` Computes the Hilbert Transform. It is used to calculate the α polynomial.
5. `dinf` Calculates D_∞, which is used in the pulse design.
6. `rf2ab` Performs the forward SLR recursion.
7. `profile` Calculates the slice profile of an excitation pulse, given the α and β polynomials.
8. `guirf` Invokes the graphical user interface for pulse design and simulation.

---

9. `guirf.fig` The Matlab generated file which defines the look and layout of the GUI. This is NOT a `.m` file or even a text file.

makerf.m

```
function rf = makerf(points, time, band, d1, d2)
% rf = makerf(points,time,band,pulsetype,d1,d2)
%
% Top-level function for designing RF pulses.
%   Only designs small-tip angle pulses
%   and 90 degree excitation pulses.
%
% rf = makerf(points, time, band, d1, d2)
%       points - number of points in the pulse
%       time - time duration of pulse (ms)
%       band - bandwidth of pulse (kHz)
%       d1 - in-slice ripple (default = 0.01 (1%))
%       d2 - out-of-slice ripple (default = 0.01 (1%))
%
% Written by Amir Schricker, 2001.
% amirs@csua.berkeley.edu

% If not enough input arguments, use default values.
if (nargin == 3)
    d1 = 0.01; d2 = 0.01;
elseif (nargin ~= 5)
    disp('Usage: rf = makerf(points, time, band, d1, d2)');
    disp(' ');
end

% Need to scale the ripples for the remez algorithm
% (see SLR paper for details)
d1 = sqrt(d1/2);
d2 = d2/sqrt(2);

% Calculate the transition band lengths and sampling frequency.
d = dinf(d1,d2);
tb = time*band;
w = d/tb;
bw = band*w;
fsamp = points/time;

% Calculate parameters for the 'remez' algorithm:
% number of points, vector of frequency edges, amplitudes, and ripple
n = points;
f = [0 band-bw band+bw fsamp] / fsamp;
a = [1 1 0 0];
w = [1 d1/d2];

% Make the digital filter and scale it for a 90-degree pulse
% scaling factor is sin(flipangle/2) = sin(45)
% This scaled digital filter is the beta polynomial
% (see SLR paper for details)
b = remez(n-1,f,a,w);
b = sqrt(1/2) * b;

% Get alpha polynomial and make RF
alf = beta2alpha(b);
bet = b;
rf = ab2rf(alf,bet);
rf = -imag(rf);
```

ab2rf.m

```
function rf = ab2rf(a,b)
% rf = ab2rf(a,b)
%
% Performs inverse SLR transform:
%  Creates an RF pulse from the complex
%  alpha and beta polynomials.
%
% rf = ab2rf(a,b)
%       a - alpha polynomial
%       b - beta polynomial
%
% Amir Schricker
% amirs@csua.berkeley.edu

if(length(a) ~= length(b))
    disp('Lengths of alpha and beta not equal');
end

n = length(a);
rf = zeros(1,n);

for j=n:-1:1,

    % Get lowest order coefficients
    b0 = b(1);
    a0 = a(1);

    % Calculate RF
    phi = 2 * atan2(abs(b0),abs(a0));
    theta = angle(-i*b0/a0);
    rf(j) = phi * exp(i*theta);

    % Now get the coefficients of one-lower A and B polynomials
    C = cos(abs(rf(j)/2));
    S = i * exp(i*angle(rf(j))) * sin(abs(rf(j)/2));
    len = length(a);
    a_1 = C*a + conj(S)*b;
    b_1 = conv( [-S zeros(1,len)], a) + conv( [ C zeros(1,len)], b);

    % Cut out leading term (lowest power) of A
    % and low-order term of B (highest power).
    a = a_1(1:len-1);
    b = b_1(2:len);
end
```

beta2alpha.m

```
function alf = beta2alpha(b)
% alf = beta2alpha(b)
%
% Calculates a minimum phase alpha polynomial
% from the beta polynomial using the Hilbert
% Transform.
%
% Amir Schricker
% amirs@csua.berkeley.edu

n = length(b);

% Details of this are, again, in the SLR paper.

% Zero-pad b up to 1024 and get its profile.
bz = [b zeros(1,1024-n)];
bet = fft(bz);

% Compute magnitude of alpha
a_mag = sqrt(1-bet.*conj(bet));

% Compute alpha profile
a_total = exp(ifft(ht(fft(log(a_mag)))));
alf = fft(a_total)/1024;    % N-pt FFT amplifies by N.

% Get back original points
alf = real(alf(1:n));
```

ht.m

```
function h = ht(a)
% h = ht(a)
%
% Amir's Hilbert transform operator
%
%  Assumes data is not fftshift'ed
%  (i.e. DC is the first sample)
%
% Written by Amir Schricker, 2001.
% amirs@csua.berkeley.edu

n = length(a);

% Double the amplitude of all frequencies
% (but we'll only keep the positives ones, see next step)
% Don't touch DC.
a(2:n) = 2*a(2:n);

% Zero out the negative frequencies
if mod(n,2) == 0        % even length
    a(n/2+1:n) = 0;
else                    % odd length
    a(ceil(n/2)+1:n) = 0;
end

h = a;
```

dinf.m

```
function d = dinf(d1,d2)
% d = dinf(d1,d2)
%
% Calculates D_infinity, the performance measure of the filter,
%  as a function of the in-slice and out-of-slice ripple.
%
% Written by Amir Schricker, 2001.
% amirs@csua.berkeley.edu


% The D_infinity value is a performance of a digital
% filter as a function of the ripples, and it is
% empirically derived. See SLR paper for details.

a1 = 5.309e-3;
a2 = 7.114e-2;
a3 = -4.761e-1;
a4 = -2.66e-3;
a5 = -5.941e-1;
a6 = -4.278e-1;

L1 = log10(d1);
L2 = log10(d2);

part1 = a1*L1*L1 + a2*L1 + a3;
part2 = a4*L1*L1 + a5*L1 + a6;

d = part1*L2 + part2;
```

rf2ab.m

```
function [a,b] = rf2ab(rf,x)
% [a b] = rf2ab(rf,x)
%
% Performs forward SLR transform:
%  Calculates the complex alpha and beta from the RF pulse.
%
% [a b] = rf2ab(rf,x)
%       rf - RF pulse
%       x  - position vector where a and b will be calculated
%
% Written by Amir Schricker, 2001.
% amirs@csua.berkeley.edu

n = length(rf);
g = ones(1,n)/n*(2*pi);

% "initialize" a and b vectors
a = zeros(1,n);
b = zeros(1,n);

for k=1:length(x)     % for each x location figure out a and b
    xx = x(k);
    g = g(1);   % gradient constant everywhere

    % initialize alpha and beta
    alpha = 1;
    beta = 0;

    for j=1:1:n     % loop to find a_j from a_(j-1)

        % Calculate all the necessary values for the algorithm.
        phi = -sqrt( abs(rf(j))^2 + (g*xx)^2 );
        const = 1/abs(phi);
        nx = const*rf(j);
        ny = 0;         % assumes B1 is only in x-direction
        nz = const*g*xx;

        %calculate rotation matrix of a's and b's here.
        a_j = cos(phi/2) - i*nz*sin(phi/2);
        b_j = -i*(nx + i*ny) * sin(phi/2);
        rotMatrix = [a_j -conj(b_j); b_j conj(a_j)];

        product =  rotMatrix * [alpha; beta];

        % Iterate to compute the next alpha and beta.
        alpha = product(1);
        beta = product(2);
    end

    a(k) = alpha;
    b(k) = beta;

end
```

profile.m

```
function mxy = profile(alpha, beta)
% mxy = profile(alpha, beta)
%
% Calculates the slice profile of an excitation RF pulse
% based on alpha and beta. Mxy = 2*conj(a)*b
%
% Written by Amir Schricker, 2001.
% amirs@csua.berkeley.edu

mxy = 2 * conj(alpha) .* beta;
```

guirf.m

```
function varargout = guirf(varargin)
% Graphical User Interface for
%  Amir's RF Pulse Design Tools
%
% Written by Amir Schricker, 2001.
% amirs@csua.berkeley.edu

% GUIRF Application M-file for guirf.fig
%    FIG = GUIRF launch guirf GUI.
%    GUIRF('callback_name', ...) invoke the named callback.

% Last Modified by GUIDE v2.0 16-Dec-2001 00:55:53

if nargin == 0  % LAUNCH GUI

fig = openfig(mfilename,'reuse');

% Use system color scheme for figure:
set(fig,'Color',get(0,'defaultUicontrolBackgroundColor'),...
        'name','GUIRF -- Graphical RF Pulse Design');

% Generate a structure of handles to pass to callbacks, and store it.
handles = guihandles(fig);
guidata(fig, handles);

if nargout > 0
varargout{1} = fig;
end

elseif ischar(varargin{1}) % INVOKE NAMED SUBFUNCTION OR CALLBACK

try
if (nargout)
[varargout{1:nargout}] = feval(varargin{:}); % FEVAL switchyard
else
feval(varargin{:}); % FEVAL switchyard
end
catch
disp(lasterr);
end

end


%| ABOUT CALLBACKS:
%| GUIDE automatically appends subfunction prototypes to this file, and
%| sets objects' callback properties to call them through the FEVAL
%| switchyard above. This comment describes that mechanism.
%|
%| Each callback subfunction declaration has the following form:
%| <SUBFUNCTION_NAME>(H, EVENTDATA, HANDLES, VARARGIN)
%|
%| The subfunction name is composed using the object's Tag and the
%| callback type separated by '_', e.g. 'slider2_Callback',
%| 'figure1_CloseRequestFcn', 'axis1_ButtondownFcn'.
%|
%| H is the callback object's handle (obtained using GCBO).
%|
%| EVENTDATA is empty, but reserved for future use.
%|
%| HANDLES is a structure containing handles of components in GUI using
%| tags as fieldnames, e.g. handles.figure1, handles.slider2. This
%| structure is created at GUI startup using GUIHANDLES and stored in
%| the figure's application data using GUIDATA. A copy of the structure
%| is passed to each callback.  You can store additional information in
%| this structure at GUI startup, and you can change the structure
%| during callbacks.  Call guidata(h, handles) after changing your
%| copy to replace the stored original so that subsequent callbacks see
%| the updates. Type "help guihandles" and "help guidata" for more
%| information.
%|
%| VARARGIN contains any extra arguments you have passed to the
%| callback. Specify the extra arguments by editing the callback
%| property in the inspector. By default, GUIDE sets the property to:
%| <MFILENAME>('<SUBFUNCTION_NAME>', gcbo, [], guidata(gcbo))
%| Add any extra arguments after the last argument, before the final
%| closing parenthesis.


% --------------------------------------------------------------------
function varargout = time_duration_Callback(h, eventdata, handles, varargin)
user_entry = str2double(get(h,'string'));
if isnan(user_entry)
    errordlg('Hey buddy, enter a number!','Bad Input','modal')
end


% --------------------------------------------------------------------
function varargout = bandwidth_Callback(h, eventdata, handles, varargin)
user_entry = str2double(get(h,'string'));
if isnan(user_entry)
    errordlg('Hey buddy, enter a number!','Bad Input','modal')
end
```

```
% --------------------------------------------------------------------
function varargout = num_points_Callback(h, eventdata, handles, varargin)
user_entry = str2double(get(h,'string'));
if isnan(user_entry)
    errordlg('Hey buddy, enter a number!','Bad Input','modal')
end


% --------------------------------------------------------------------
function varargout = ripple1_Callback(h, eventdata, handles, varargin)
user_entry = str2double(get(h,'string'));
if isnan(user_entry)
    errordlg('Hey buddy, enter a number!','Bad Input','modal')
end


% --------------------------------------------------------------------
function varargout = ripple2_Callback(h, eventdata, handles, varargin)
user_entry = str2double(get(h,'string'));
if isnan(user_entry)
    errordlg('Hey buddy, enter a number!','Bad Input','modal')
end


% --------------------------------------------------------------------
function varargout = design_rf_Callback(h, eventdata, handles, varargin)

% Plot to the top axes
axes(handles.axes1)

time = str2double(get(handles.time_duration,'string'));
band = str2double(get(handles.bandwidth,'string'));
npts = str2double(get(handles.num_points,'string'));
rip1 = str2double(get(handles.ripple1,'string'));
rip2 = str2double(get(handles.ripple2,'string'));

% Error check: make sure all the parameters
% are numbers, otherwise don't plot anything.
if (isnan(time) | isnan(band) | isnan(npts) | isnan(rip1) | isnan(rip2))
    errordlg('Hey, all the values have to be numbers!','Bad Input','modal')
else
    % The real stuff:
    rf = makerf(npts, time, band, rip1, rip2);
    incr = time/npts;
    t = [0:incr:time-incr];
    plot(t,rf);
    xlabel('Time (ms)');
    ylabel('Amplitude');
    handles.RF_PULSE = rf;
    handles.axis_time = t;
    guidata(h,handles);
end


% --------------------------------------------------------------------
function varargout = norm_freq_Callback(h, eventdata, handles, varargin)

user_entry = str2double(get(h,'string'));
if isnan(user_entry)
    errordlg('Hey buddy, enter a number!','Bad Input','modal')
end


% --------------------------------------------------------------------
function varargout = simulate_rf_Callback(h, eventdata, handles, varargin)

% Plot to the bottom axes
axes(handles.axes2)

time = str2double(get(handles.time_duration,'string'));
band = str2double(get(handles.bandwidth,'string'));
npts = str2double(get(handles.num_points,'string'));
rip1 = str2double(get(handles.ripple1,'string'));
rip2 = str2double(get(handles.ripple2,'string'));
nfreq = str2double(get(handles.norm_freq,'string'));

% Error check: make sure all the parameters
% are numbers, otherwise don't plot anything.
if (isnan(time) | isnan(band) | isnan(npts) | isnan(rip1) | isnan(rip2))
    errordlg('Hey, all the values have to be numbers!','Bad Input','modal')
else
    % The real stuff...
    rf = handles.RF_PULSE;
    x = [-nfreq:0.1:nfreq];
    [a b] = rf2ab(rf,x);
    xfreq = x/time;
    plot(xfreq, abs(profile(a,b)));
    xlabel('Frequency (kHz)');
    ylabel('Amplitude');
    disp('Simulation DONE');
    handles.a_poly = a;
    handles.b_poly = b;
    handles.axis_xfreq = xfreq;
    guidata(h,handles);
end
```

```
% --------------------------------------------------------------------
function varargout = design_window_Callback(h, eventdata, handles, varargin)

rf = handles.RF_PULSE;
t = handles.axis_time;
figure(10);
plot(t,rf);
xlabel('Time (ms)');
ylabel('Amplitude');
title('RF Waveform');


% --------------------------------------------------------------------
function varargout = simulate_window_Callback(h, eventdata, handles, varargin)

% This function plots the same slice profile
% but just in a separate figure window.
a = handles.a_poly;
b = handles.b_poly;
xfreq = handles.axis_xfreq;
figure(11);
plot(xfreq, abs(profile(a,b)));
xlabel('Frequency (kHz)');
ylabel('Amplitude');
title('Mxy Slice Profile');


% --------------------------------------------------------------------
function varargout = wave_grid_Callback(h, eventdata, handles, varargin)

% Toggle the grid on the top axes.
axes(handles.axes1)
if (get(h,'Value') == get(h,'Max'))
    grid on;
else
    grid off;
end


% --------------------------------------------------------------------
function varargout = slice_grid_Callback(h, eventdata, handles, varargin)

% Toggle the grid on the bottom axes.
axes(handles.axes2)
if (get(h,'Value') == get(h,'Max'))
    grid on;
else
```

```
    grid off;
end


% ----------------
% Menu definitions
% ----------------

% --------------------------------------------------------------------
function varargout = file_menu_Callback(h, eventdata, handles, varargin)

% --------------------------------------------------------------------
function varargout = exit_submenu_Callback(h, eventdata, handles, varargin)
delete(handles.figure1)

% --------------------------------------------------------------------
function varargout = help_menu_Callback(h, eventdata, handles, varargin)

% --------------------------------------------------------------------
function varargout = about_submenu_Callback(h, eventdata, handles, varargin)

% This message box displays some data about me.
msgbox({'Written by','Amir Schricker','amirs@csua.berkeley.edu'},'About GUIRF');
```