Parametric Modeling

Parametric modeling techniques find the parameters for a mathematical model describing a signal, system, or process. These techniques use known information about the system to determine the model. Applications for parametric modeling include speech and music synthesis, data compression, high-resolution spectral estimation, communications, manufacturing, and simulation.

The toolbox parametric modeling functions operate with the rational transfer function model. Given appropriate information about an unknown system (impulse or frequency response data, or input and output sequences), these functions find the coefficients of a linear system that models the system.

One important application of the parametric modeling functions is in the design of filters that have a prescribed time or frequency response. These functions provide a data-oriented alternative to the IIR and FIR filter design functions discussed in Chapter 2, “Filter Design and Implementation.”
Here is a summary of the parametric modeling functions in this toolbox. Note that the System Identification Toolbox provides a more extensive collection of parametric modeling functions.

<table>
<thead>
<tr>
<th>Domain</th>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>arburg</td>
<td>Generate all-pole filter coefficients that model an input data sequence using the Levinson-Durbin algorithm.</td>
</tr>
<tr>
<td></td>
<td>arcov</td>
<td>Generate all-pole filter coefficients that model an input data sequence by minimizing the forward prediction error.</td>
</tr>
<tr>
<td></td>
<td>armcov</td>
<td>Generate all-pole filter coefficients that model an input data sequence by minimizing the forward and backward prediction errors.</td>
</tr>
<tr>
<td></td>
<td>aryule</td>
<td>Generate all-pole filter coefficients that model an input data sequence using an estimate of the autocorrelation function.</td>
</tr>
<tr>
<td></td>
<td>lpc,</td>
<td>Linear Predictive Coding. Generate all-pole recursive filter whose impulse response matches a given sequence.</td>
</tr>
<tr>
<td></td>
<td>levinson</td>
<td>Generated IIR filter whose impulse response matches a given sequence.</td>
</tr>
<tr>
<td></td>
<td>prony</td>
<td>Find IIR filter whose output, given a specified input sequence, matches a given output sequence.</td>
</tr>
<tr>
<td></td>
<td>stmcb</td>
<td>Generated digital or analog filter coefficients given complex frequency response data.</td>
</tr>
</tbody>
</table>

Because yulewalk is geared explicitly toward ARMA filter design, it is discussed in Chapter 2, “Filter Design and Implementation”.

pburg and pyulear are discussed in Chapter 3, “Statistical Signal Processing” along with the other (nonparametric) spectral estimation methods.
Time-Domain Based Modeling

The \texttt{lpc}, \texttt{prony}, and \texttt{stmcb} functions find the coefficients of a digital rational transfer function that approximates a given time-domain impulse response. The algorithms differ in complexity and accuracy of the resulting model.

Linear Prediction

Linear prediction modeling assumes that each output sample of a signal, $x(k)$, is a linear combination of the past $n$ outputs (that is, it can be “linearly predicted” from these outputs), and that the coefficients are constant from sample to sample:

$$x(k) = -a(2)x(k-1) - a(3)x(k-2) - \ldots - a(n+1)x(k-n)$$

An $n$th-order all-pole model of a signal $x$ is

$$a = \texttt{lpc}(x,n)$$

To illustrate \texttt{lpc}, create a sample signal that is the impulse response of an all-pole filter with additive white noise:

```matlab
randn('state',0);
x = impz(1,[1 0.1 0.1 0.1 0.1],10) + randn(10,1)/10;
```

The coefficients for a fourth-order all-pole filter that models the system are

$$a = \texttt{lpc}(x,4)$$

$$a = \begin{bmatrix} 1.0000 & 0.2574 & 0.1666 & 0.1203 & 0.2598 \end{bmatrix}$$

\texttt{lpc} first calls \texttt{xcorr} to find a biased estimate of the correlation function of $x$, and then uses the Levinson-Durbin recursion, implemented in the \texttt{levinson} function, to find the model coefficients $a$. The Levinson-Durbin recursion is a fast algorithm for solving a system of symmetric Toeplitz linear equations. \texttt{lpc}'s entire algorithm for $n=4$ is

```matlab
r = xcorr(x);
r(1:length(x)-1) = [];
% Remove corr. at negative lags
a = levinson(r,4)
```

$$a = \begin{bmatrix} 1.0000 & 0.2574 & 0.1666 & 0.1203 & 0.2598 \end{bmatrix}$$
You could form the linear prediction coefficients with other assumptions by passing a different correlation estimate to `levinson`, such as the biased correlation estimate:

```matlab
r = xcorr(x,'biased');
r(1:length(x)-1) = [] ; % Remove corr. at negative lags
a = levinson(r,4)
```

```
a =
   1.0000    0.2574    0.1666    0.1203    0.2598
```

**Prony’s Method (ARMA Modeling)**

The `prony` function models a signal using a specified number of poles and zeros. Given a sequence `x` and numerator and denominator orders `n` and `m`, respectively, the statement

```
[b,a] = prony(x,n,m)
```

finds the numerator and denominator coefficients of an IIR filter whose impulse response approximates the sequence `x`.

The `prony` function implements the method described in [3] Parks and Burrus (pgs. 226-228). This method uses a variation of the covariance method of AR modeling to find the denominator coefficients `a`, and then finds the numerator coefficients `b` for which the resulting filter’s impulse response matches exactly the first `n + 1` samples of `x`. The filter is not necessarily stable, but it can potentially recover the coefficients exactly if the data sequence is truly an autoregressive moving-average (ARMA) process of the correct order.

**Note** The functions `prony` and `stmcb` (described next) are more accurately described as ARX models in system identification terminology. ARMA modeling assumes noise only at the inputs, while ARX assumes an external input. `prony` and `stmcb` know the input signal: it is an impulse for `prony` and is arbitrary for `stmcb`. 
A model for the test sequence \( x \) (from the earlier \( \text{lp}c \) example) using a third-order IIR filter is

\[
[b,a] = \text{prony}(x,3,3)
\]

\[
b = \\
0.9567 \ -0.3351 \ 0.1866 \ -0.3782
\]

\[
a = \\
1.0000 \ -0.0716 \ 0.2560 \ -0.2752
\]

The \( \text{impz} \) command shows how well this filter’s impulse response matches the original sequence:

\[
\text{format long}
\]

\[
[x \ \text{impz}(b,a,10)]
\]

\[
\text{ans} = \\
0.95674351884718 \ 0.95674351884718 \\
-0.26655843782381 \ -0.26655843782381 \\
-0.07746676935252 \ -0.07746676935252 \\
-0.05223235796415 \ -0.05223235796415 \\
-0.18754713506815 \ -0.05726777015121 \\
0.15348154656430 \ -0.01204969926150 \\
0.13986742016521 \ -0.00057632797226 \\
0.00609257234067 \ -0.01271681570687 \\
0.03349954614087 \ -0.00407967053863 \\
0.01086719328209 \ 0.00280486049427
\]

Notice that the first four samples match exactly. For an example of exact recovery, recover the coefficients of a Butterworth filter from its impulse response:

\[
[b,a] = \text{butter}(4,.2);
\]

\[
h = \text{impz}(b,a,26);
\]

\[
[bb,aa] = \text{prony}(h,4,4);
\]

Try this example; you’ll see that \( bb \) and \( aa \) match the original filter coefficients to within a tolerance of \( 10^{-13} \).
**Steiglitz-McBride Method (ARMA Modeling)**

The `stmcb` function determines the coefficients for the system $b(z)/a(z)$ given an approximate impulse response $x$, as well as the desired number of zeros and poles. This function identifies an unknown system based on both input and output sequences that describe the system's behavior, or just the impulse response of the system. In its default mode, `stmcb` works like `prony`.

```matlab
[b,a] = stmcb(x,3,3)
b =
  0.9567  -0.5181   0.5702  -0.5471
a =
  1.0000  -0.2384   0.5234  -0.3065
```

`stmcb` also finds systems that match given input and output sequences:

```matlab
y = filter(1,[1 1],x); % Create an output signal.
[b,a] = stmcb(y,x,0,1)
b =
  1.0000
a =
     1     1
```

In this example, `stmcb` correctly identifies the system used to create $y$ from $x$.

The Steiglitz-McBride method is a fast iterative algorithm that solves for the numerator and denominator coefficients simultaneously in an attempt to minimize the signal error between the filter output and the given output signal. This algorithm usually converges rapidly, but might not converge if the model order is too large. As for `prony`, `stmcb`'s resulting filter is not necessarily stable due to its exact modeling approach.

`stmcb` provides control over several important algorithmic parameters; modify these parameters if you are having trouble modeling the data. To change the number of iterations from the default of five and provide an initial estimate for the denominator coefficients:

```matlab
n = 10;            % Number of iterations
a = lpc(x,3);      % Initial estimates for denominator
[b,a] = stmcb(x,3,3,n,a);
```
The function uses an all-pole model created with `prony` as an initial estimate when you do not provide one of your own.

To compare the functions `lpc`, `prony`, and `stmcb`, compute the signal error in each case:

```matlab
a1 = lpc(x,3);
[b2,a2] = prony(x,3,3);
[b3,a3] = stmcb(x,3,3);
[x-impz(1,a1,10)  x-impz(b2,a2,10)  x-impz(b3,a3,10)]
ans =
-0.0433   0  -0.0000
-0.0240   0  0.0234
-0.0040   0  -0.0778
-0.0448  -0.0000  0.0498
-0.2130  -0.1303  -0.0742
 0.1545   0.1655  0.1270
 0.1426   0.1404  0.1055
 0.0068   0.0188  0.0465
 0.0329   0.0376  0.0530
 0.0108   0.0081 -0.0162
```

```matlab
sum(ans.^2)
ans =
 0.0953   0.0659   0.0471
```

In comparing modeling capabilities for a given order IIR model, the last result shows that for this example, `stmcb` performs best, followed by `prony`, then `lpc`. This relative performance is typical of the modeling functions.
**Frequency-Domain Based Modeling**

The `invfreqs` and `invfreqz` functions implement the inverse operations of `freqs` and `freqz`; they find an analog or digital transfer function of a specified order that matches a given complex frequency response. Though the following examples demonstrate `invfreqz`, the discussion also applies to `invfreqs`.

To recover the original filter coefficients from the frequency response of a simple digital filter:

```matlab
[b,a] = butter(4,0.4)    % Design Butterworth lowpass
b =
   0.0466    0.1863    0.2795    0.1863    0.0466
a =
   1.0000   -0.7821    0.6800   -0.1827    0.0301
[h,w] = freqz(b,a,64);         % Compute frequency response
[b4,a4] = invfreqz(h,w,4,4)    % Model: n = 4, m = 4
b4 =
   0.0466    0.1863    0.2795    0.1863    0.0466
a4 =
   1.0000   -0.7821    0.6800   -0.1827    0.0301
```

The vector of frequencies `w` has the units in rad/sample, and the frequencies need not be equally spaced. `invfreqz` finds a filter of any order to fit the frequency data; a third-order example is

```matlab
[b4,a4] = invfreqz(h,w,3,3)    % Find third-order IIR
b4 =
   0.0464    0.1785    0.2446    0.1276
a4 =
   1.0000   -0.9502    0.7382   -0.2006
```

Both `invfreqs` and `invfreqz` design filters with real coefficients; for a data point at positive frequency `f`, the functions fit the frequency response at both `f` and `-f`.

```
[4 22]
```
By default `invfreqz` uses an equation error method to identify the best model from the data. This finds $b$ and $a$ in

$$
\min_{b,a} \sum_{k=1}^{n} wt(k) |h(k)A(w(k)) - B(w(k))|^2
$$

by creating a system of linear equations and solving them with the MATLAB \ operator. Here $A(w(k))$ and $B(w(k))$ are the Fourier transforms of the polynomials $a$ and $b$ respectively at the frequency $w(k)$, and $n$ is the number of frequency points (the length of $h$ and $w$). $wt(k)$ weights the error relative to the error at different frequencies. The syntax

```matlab
invfreqz(h,w,n,m,wt)
```

includes a weighting vector. In this mode, the filter resulting from `invfreqz` is not guaranteed to be stable.

`invfreqz` provides a superior (“output-error”) algorithm that solves the direct problem of minimizing the weighted sum of the squared error between the actual frequency response points and the desired response

$$
\min_{b,a} \sum_{k=1}^{n} wt(k) \left| h(k) \frac{B(w(k))}{A(w(k))} \right|^2
$$

To use this algorithm, specify a parameter for the iteration count after the weight vector parameter:

```matlab
wt = ones(size(w)); % Create unity weighting vector
[b30,a30] = invfreqz(h,w,3,3,wt,30) % 30 iterations
```

```matlab
b30 =
0.0464 0.1829 0.2572 0.1549
```

```matlab
a30 =
1.0000 -0.8664 0.6630 -0.1614
```

The resulting filter is always stable.
Graphically compare the results of the first and second algorithms to the original Butterworth filter with FVTool (and select the Magnitude and Phase Responses):

\[ \text{fvtool}(b,a,b4,a4,b30,a30) \]

To verify the superiority of the fit numerically, type

\[ \text{sum}(\text{abs}(h-\text{freqz}(b4,a4,w)).^2) \quad \% \text{Total error, algorithm 1} \]

\[ \text{ans} = 0.0200 \]
\[
\text{sum(abs(h-freqz(b30,a30,w)).^2) \% Total error, algorithm 2}
\]
\[
\text{ans} = \quad 0.0096
\]