Code to simulate xLS methods

- You have now seen derivations of different capacity estimators
- Now time to see how to code in Octave: `xLSalgos.m`

```
% Tests the recursive performance of the xLS algorithms on a particular dataset
% [Qhat,SigmaQ] = xLSalgos(measX,measY,SigmaX,SigmaY,Gamma,Qnom,SigmaY0)
%   - measX   = noisy z(2)-z(1)
%   - measY   = noisy integral(i(t)/3600 dt)
%   - SigmaX  = variance of X
%   - SigmaY  = variance of Y
%   - Gamma   = geometric forgetting factor (Gamma = 1 for perfect memory)
%   - Qnom    = nominal value of Q: if nonzero, used to initialize recursions
%   - SigmaY0 = variance of uncertainty of nominal capacity (optional param.)
%
%   - Qhat = estimate of capacity at every time step
%     - column 1 = WLS - weighted, recursive
%     - column 2 = WTLS - weighted, but not recursive
%     - column 3 = TLS - recursive TLS when SigmaX and SigmaY related by
%                  constant multiple, sqrt(SigmaX(1)/SigmaY(1))
%     - column 4 = AWTLS - recursive and weighted
%   - SigmaQ = variance of Q, columns correspond to methods same as for Qhat
```


Initializing `xLSalgos` variables

- Code begins by reserving memory and initializing recursive parameters if `Qnom` is not set to zero

```
function [Qhat,SigmaQ]=xLSalgos(measX,measY,SigmaX,SigmaY,Gamma,Qnom,varargin)
  measX = measX(:); measY = measY(:); SigmaX = SigmaX(:); SigmaY = SigmaY(:);
  Qhat = zeros(length(measX),4); SigmaQ = Qhat; % Reserve some memory
  K = sqrt(SigmaX(1)/SigmaY(1));

  % Initialize some variables used for the recursive methods
  c1 = 0; c2 =0; c3 = 0; % unscaled
  C1 = 0; C2 =0; C3 = 0; C4 = 0; C5 = 0; C6 = 0; % scaled by K
  if Qnom ~= 0,
    if ~isempty(varargin), SigmaY0=varargin{1}; else SigmaY0=SigmaY(1); end
    c1 = 1/SigmaY(1); c2 = Qnom/SigmaY(1); c3 = Qnom^2/SigmaY(1);
    C1 = 1/(K^2*SigmaY(1)); C2 = K*Qnom/(K^2*SigmaY(1));
    C3 = K^2*Qnom^2/(K^2*SigmaY(1)); C4 = C1; C5 = C2; C6 = C3;
  end
```


Updating recursive parameters

- Function then loops through input data, updating recursive parameter values each iteration

```
  for iter = 1:length(measX),
    % update unscaled recursive parameter values
    c1 = Gamma*c1 + measX(iter)^2/SigmaY(iter);
    c2 = Gamma*c2 + measX(iter)*measY(iter)/SigmaY(iter);
    c3 = Gamma*c3 + measY(iter)^2/SigmaY(iter);

    % update scaled recursive parameter values
    C1 = Gamma*C1 + measX(iter)^2/(K^2*SigmaY(iter));
    C2 = Gamma*C2 + K*measX(iter)*measY(iter)/(K^2*SigmaY(iter));
    C3 = Gamma*C3 + K^2*measY(iter)^2/(K^2*SigmaY(iter));
    C4 = Gamma*C4 + measX(iter)^2/SigmaX(iter);
    C5 = Gamma*C5 + K*measX(iter)*measY(iter)/SigmaX(iter);
    C6 = Gamma*C6 + K^2*measY(iter)^2/SigmaX(iter);
```

Implementing the WLS method

- Code next computes update to total-capacity estimate for this time step using each method
- Start by computing the WLS estimate
  - `Q` is the estimate of total capacity, stored in column 1 of `Qhat`
  - `H` is the Hessian, used to compute bounds

```
% Method 1: WLS
Q = c2./c1; Qhat(iter,1) = Q;
H = 2*c1;   SigmaQ(iter,1) = 2/H;
```


Implementing the WTLS method

- Next, implement WTLS method, which is not recursive so uses all data from 1 to `k` to compute estimate at iteration `k`

```
% Method 2: WTLS -- not recursive
x = measX(1:iter); SX = SigmaX(1:iter); % x, variance of x, already "squared"
y = measY(1:iter); SY = SigmaY(1:iter); % y, variance of y, already "squared"
g = gamma.^((iter-1):-1:0)';
Q = Qhat(iter,1); % initialize with WLS solution
for k = 1:5,
  J = sum(g.*(2*(Q*x-y).*(Q*y.*SX+x.*SY))./((Q^2*SX+SY).^2));
  H = sum(g.*(2*SY.^2.*x.^2+SX.^2.*(6*Q^2*y.^2-4*Q^3*x.*y) - ...
               SX.*SY.*(6*Q^2*x.^2-12*Q*x.*y+2*y.^2))./((Q^2*SX+SY).^3));
  Q = Q - J/H;
end
Qhat(iter,2)=Q;
SigmaQ(iter,2)=2/H;
if SigmaQ(iter,2)<0 || Q<0, % Sometimes WTLS fails to converge to soln
  Qhat(iter,2) = NaN; SigmaQ(iter,2)=0;
end
```


Implementing the TLS method

- Next, implement TLS method using recursive parameters
- Stores results much like WLS did

```
% Method 3: TLS
Q = (-c1+K^2*c3+sqrt((c1-K^2*c3)^2+4*K^2*c2^2))/(2*K^2*c2);
Qhat(iter,3) = Q;
H = ((-4*K^4*c2)*Q^3+(6*K^4*c3-6*c1*K^2)*Q^2+...
    12*c2*K^2*Q+2*(c1-K^2*c3))/(Q^2*K^2+1)^3;
SigmaQ(iter,3) = 2/H;
```

Implementing the AWTLS method

- Finally, implement the AWTLS using recursive parameters
- Uses "`roots`" (eigenvalue) method instead of Ferrari method

```
    % Method 4: AWTLS with pre-scaling
    r = roots([C5 (-C1+2*C4-C6) (3*C2-3*C5) (C1-2*C3+C6) -C2]);
    r = r(r==conj(r)); % discard complex-conjugate roots
    r = r(r>0); % discard negative roots
    Jr = ((1./(r.^2+1).^2).*(r.^4*C4-2*C5*r.^3+(C1+C6)*r.^2-2*C2*r+C3))';
    J = min(Jr);
    Q = r(Jr==J); % keep Q that minimizes cost function
    H = (2/(Q^2+1)^4)*(-2*C5*Q^5+(3*C1-6*C4+3*C6)*Q^4+(-12*C2+16*C5)*Q^3 ...
          +(-8*C1+10*C3+6*C4-8*C6)*Q^2+(12*C2-6*C5)*Q+(C1-2*C3+C6));
    Qhat(iter,4) = Q/K;
    SigmaQ(iter,4) = 2/H/K^2;
  end
return
```


Summary

- Have now learned how to implement each total-capacity estimation algorithm in Octave
  - WLS straightforward using recursive parameters
  - WTLS not recursive, so all data must be re-evaluated every time a new data point becomes available
  - TLS recursive, and just as simple to implement as WLS
  - AWTLS recursive, requires finding roots of a quartic (more challenging in embedded system but simple in Octave), requires evaluating cost function to select root