Calibrating hyperelastic materials

Table of Contents

This reading is to develop a method for calibrating different hyperelastic materials using a set of stress-stretch data. The experimental data are collected from this work. Note, incompressibility of the materials is assumed.

```
clear; clc; close all;
```

Let's load the CSV data files.

```
s_x = importfile("Subject111_Sample1_YoungDorsal_OffbiaxialX.csv"); % Off-biaxial X
```

```
s_y = importfile("Subject111_Sample1_YoungDorsal_OffbiaxialY.csv"); % Off-biaxial Y
s_b = importfile("Subject111_Sample1_YoungDorsal_Equibiaxial.csv"); % Equibiaxial
```

Let's see how the data looks like.

```
% ss_plot(s_x,"Off-biaxial X")
% ss_plot(s_y,"Off-biaxial Y")
% ss_plot(s_b,"Equibiaxial")
```

Data preparation for equibiaxial dataset:

```
% Only eqibiaxial
ss_data_11 = table2array(s_b(:,["Lambda11","Sigma11MPa"]));
ss_data_22 = table2array(s_b(:,["Lambda22","Sigma22MPa"]));
lam11 = ss_data_11(:,1);
lam22 = ss_data_22(:,1);

% DATA for all sets
sb_data_11  = table2array(s_b(:,["Lambda11","Sigma11MPa"]));
sb_data_22  = table2array(s_b(:,["Lambda22","Sigma22MPa"]));
sb_lam11    = sb_data_11(:,1);
sb_lam22    = sb_data_22(:,1);

sx_data_11  = table2array(s_x(:,["Lambda11","Sigma11MPa"]));
sx_data_22  = table2array(s_x(:,["Lambda22","Sigma22MPa"]));
sx_lam11    = sx_data_11(:,1);
sx_lam22    = sx_data_22(:,1);

sy_data_11  = table2array(s_y(:,["Lambda11","Sigma11MPa"]));
sy_data_22  = table2array(s_y(:,["Lambda22","Sigma22MPa"]));
sy_lam11    = sy_data_11(:,1);
sy_lam22    = sy_data_22(:,1);
```

Neo-Hookean

Wikipedia page: https://en.wikipedia.org/wiki/Neo-Hookean_solid

This model has one parameter C_1, and for incompressible materials, it can be reduced to:

Uniaxial: $\sigma_{11} = 2C_1\left(\lambda^2 - \frac{1}{\lambda}\right)$ (assuming $\sigma_{22} = \sigma_{33} = 0$)

Equibiaxial: $\sigma_{11} = \sigma_{22} = 2C_1\left(\lambda^2 - \frac{1}{\lambda^4}\right)$ (assuming $\sigma_{33} = 0$)

As we don't have data for uniaxial extension, let's write a code to find C_1 using the equibiaxial equation. For that, we have sets of $(\sigma_{11}, \lambda_{11})$ and $(\sigma_{22}, \lambda_{22})$, which are not the same because the selected material is not isotropic. So, for now, let's focus only on the first set along the 11 direction, and start with a first guess of 1 for C_1.

Optimization:

```
% c1_0 = 1;
% [C1_11, nH_function] = nH_biaxial(ss_data_11, c1_0)
```

And, here is the results if we use dataset for the 22 direction.

```
% C1_22 = nH_biaxial(ss_data_22, c1_0)
```

Now, we can compare the optimizied solutions with the original data.

```
% s11_nH = nH_function(C1_11, lam11);
% s22_nH = nH_function(C1_22, lam22);
%
% ss_plot(s_b,"Equibiaxial")
% hold on;
% plot(lam11,s11_nH,'b-')
% plot(lam22,s22_nH,'r-')
% hold off
```

It is easy to see that nH model is not a suitable model for our material. The results suggests that for a limited deformation, e.g. $\lambda < 1.3$, the model may work well. Let's test it out.

```
% ss_data_11_test = ss_data_11(ss_data_11(:,1) < 1.3,:);
% ss_data_22_test = ss_data_11(ss_data_22(:,1) < 1.3,:);
% C1_11 = nH_biaxial(ss_data_11_test, c1_0);
% C1_22 = nH_biaxial(ss_data_22_test, c1_0);
%
% lam11_test = ss_data_11_test(:,1);
% lam22_test = ss_data_22_test(:,1);
% s11_nH = nH_function(C1_11, lam11_test);
% s22_nH = nH_function(C1_22, lam22_test);
%
% ss_plot(s_b,"Equibiaxial")
% hold on;
% plot(lam11_test,s11_nH,'b-')
% plot(lam22_test,s22_nH,'r-')
% hold off
```

Let's continue with a more advanced model that has at least two controlling parameters.

Mooney-Rivlin

Wikipedia page: https://www.wikiwand.com/en/Mooney-Rivlin_solid

This model has two parameters C_1 and C_2, and for the equibiaxial tension can be written as:

$$\sigma_{11} = \sigma_{22} = 2C_1\left(\lambda^2 - \frac{1}{\lambda^4}\right) - 2C_2\left(\lambda^2 - \frac{1}{\lambda^4}\right) \text{ (assuming } \sigma_{33} = 0)$$

Let's see whether this model can make better approximations.

```
% c0 = C1_11 * ones([1,2]);
% [c_11, fval_11, MR_function] = MR_biaxial(ss_data_11, c0);
% [c_22, fval_22, ~] = MR_biaxial(ss_data_22, c0);
% s11_MR = MR_function(c_11, lam11);
% s22_MR = MR_function(c_22, lam22);
%
% ss_plot(s_b,"Equibiaxial")
% ylim([0,0.08]);
% hold on;
% plot(lam11,s11_MR,'b-')
% plot(lam22,s22_MR,'r-')
% hold off
```

Ogden

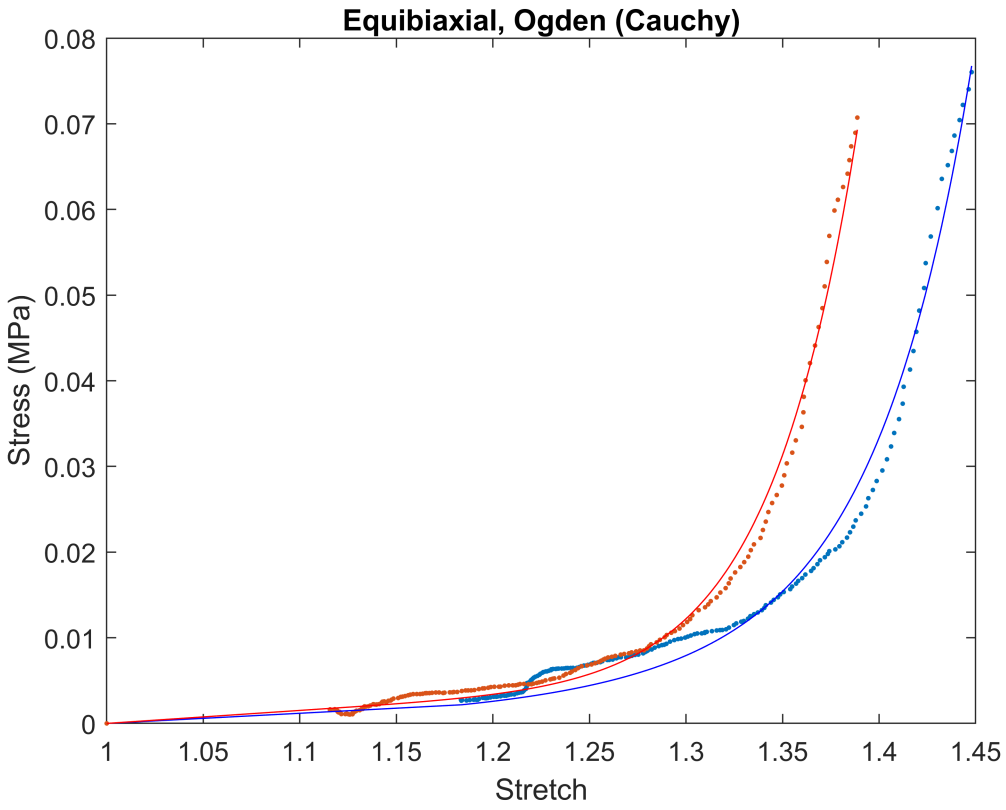
Wikipedia page: https://www.wikiwand.com/en/Ogden_(hyperelastic_model)

For quibiaxial tension, the model is written as:

$\sigma_{11} = \sigma_{22} = \sum_{p=1}^{N} \mu_p \left(\lambda^{\alpha_p} - \lambda^{-2\alpha_p} \right)$ (assuming $\sigma_{33} = 0$),

where N, μ_p and α_p are material constants.

```
N = 2;
mu0 = 1e-5 * ones([1,N]);
alpha0 = 10 * ones([1,N]);
c0 = [mu0,alpha0];

[optC_11, fval_11, Ogden_fun] = Ogden_biaxial(ss_data_11, N, c0);
[optC_22, fval_22, ~] = Ogden_biaxial(ss_data_22, N, c0);
s11_Og = Ogden_fun(N, optC_11, lam11);
s22_Og = Ogden_fun(N, optC_22, lam22);

ss_plot(s_b,"Equibiaxial, Ogden (Cauchy)")
ylim([0,0.08]);
hold on;
plot(lam11,s11_Og,'b-')
plot(lam22,s22_Og,'r-')
hold off
```




```
[fval_11 fval_22]
```

```
ans = 1×2
10^-3 ×
    0.8717    0.3634
```

This function can provides accurate approximations. Let's see how far the solutions are from each other:

```
optC_11 ./ optC_22
```

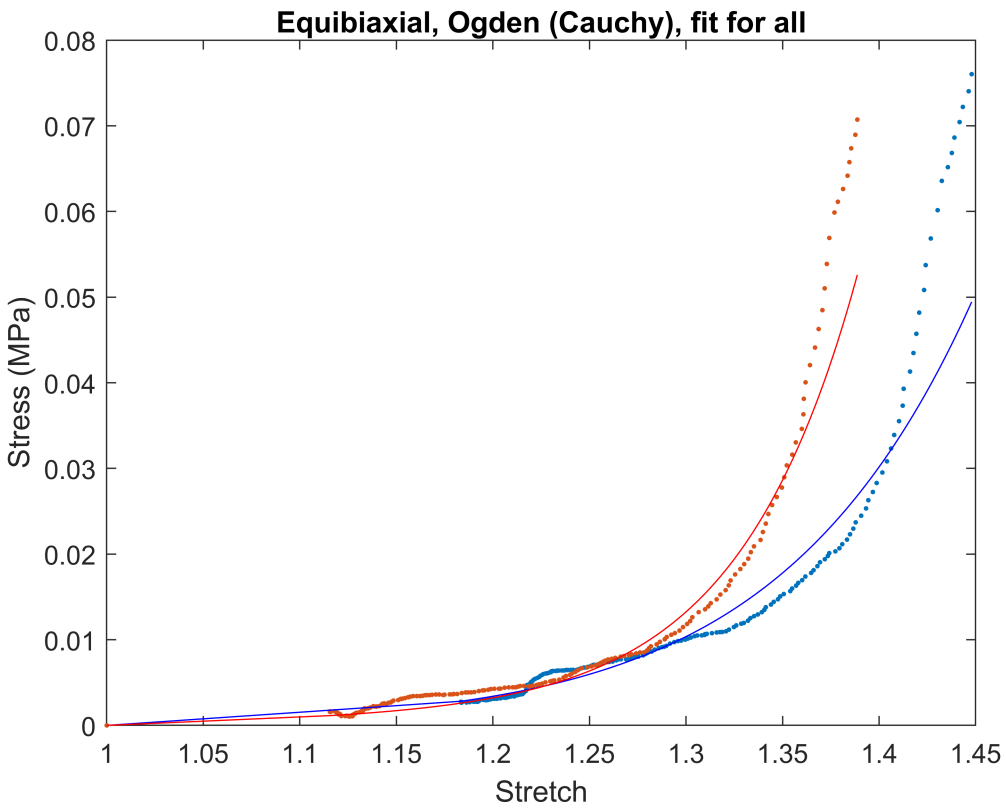
```
ans = 1×4
    0.1220    0.1507    4.9952    1.0396
```

It shows that only the values of α_1 are close to each other for both solutions. Therefore, we need to work with an anisotropic material model.

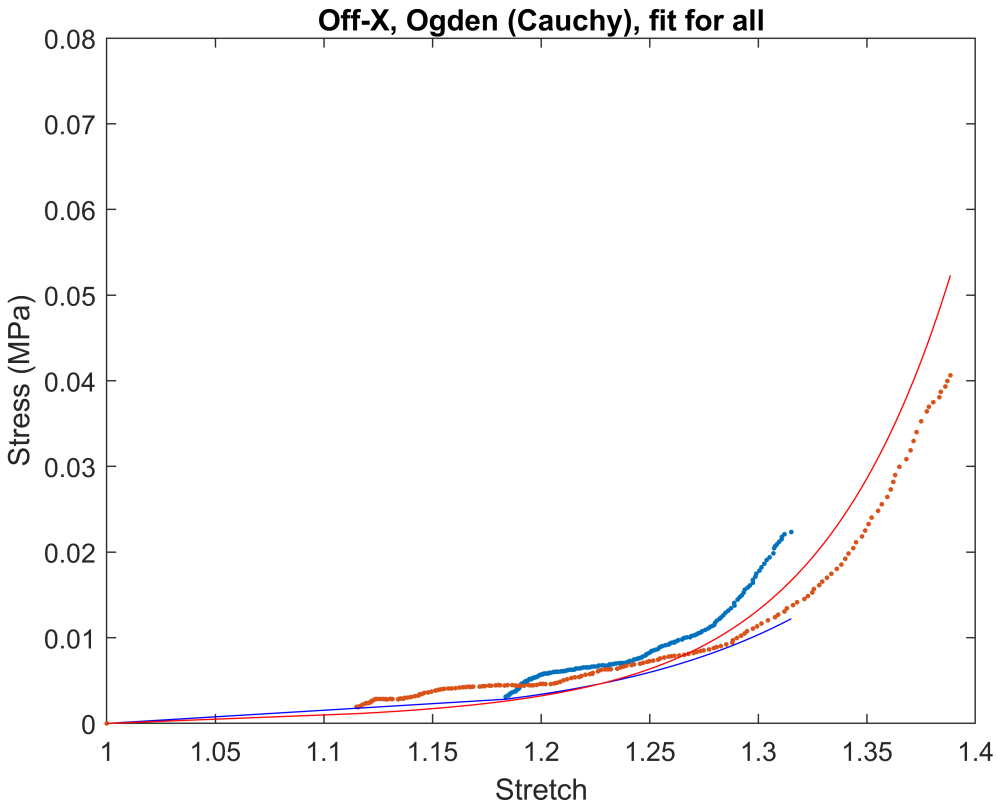
Ogden on all sets

```
[optC_11, ~, Yeoh_fun] = Ogden_biaxial([sb_data_11;sx_data_11;sy_data_11], N, c0);
[optC_22, ~, ~] = Ogden_biaxial([sb_data_22;sx_data_22;sy_data_22], N, c0);

sb_11_Yeoh = Ogden_fun(N, optC_11, sb_lam11);
sb_22_Yeoh = Ogden_fun(N, optC_22, sb_lam22);
ss_plot(s_b,"Equibiaxial, Ogden (Cauchy), fit for all")
ylim([0,0.08]);
hold on;
plot(sb_lam11,sb_11_Yeoh,'b-')
```

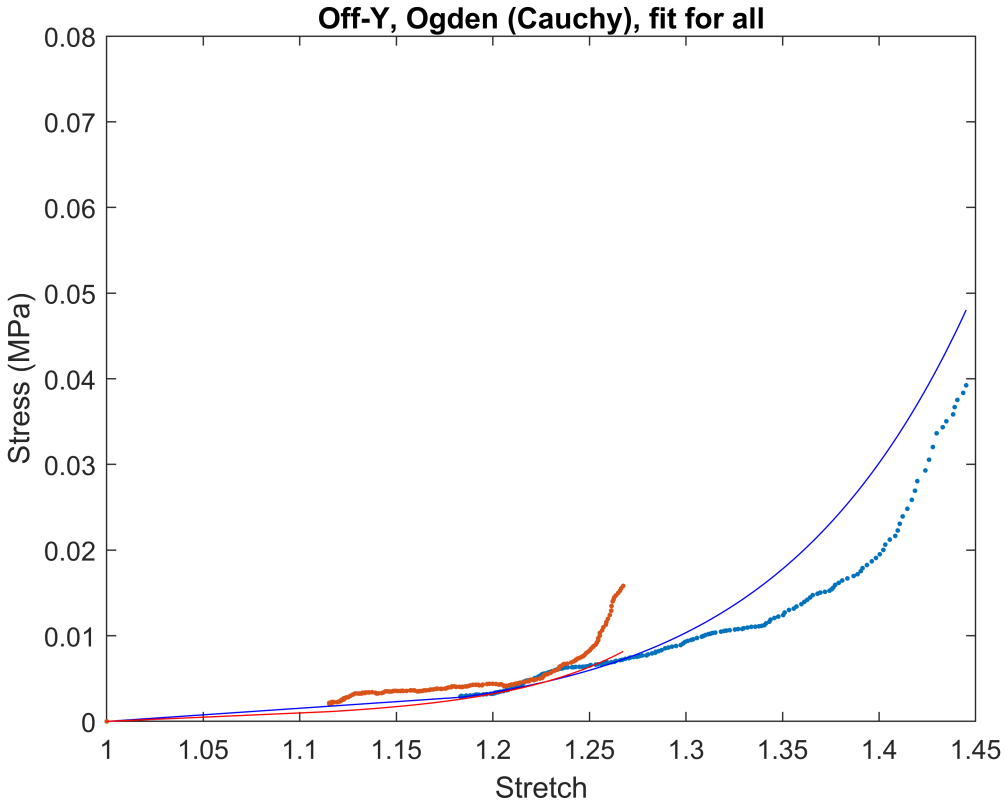
```
plot(sb_lam22,sb_22_Yeoh,'r-')
hold off
```




```
sx_11_Yeoh = Ogden_fun(N, optC_11, sx_lam11);
sx_22_Yeoh = Ogden_fun(N, optC_22, sx_lam22);
ss_plot(s_x,"Off-X, Ogden (Cauchy), fit for all")
ylim([0,0.08]);
hold on;
plot(sx_lam11,sx_11_Yeoh,'b-')
plot(sx_lam22,sx_22_Yeoh,'r-')
hold off
```




```
sy_11_Yeoh = Ogden_fun(N, optC_11, sy_lam11);
sy_22_Yeoh = Ogden_fun(N, optC_22, sy_lam22);
ss_plot(s_y,"Off-Y, Ogden (Cauchy), fit for all")
ylim([0,0.08]);
hold on;
plot(sy_lam11,sy_11_Yeoh,'b-')
plot(sy_lam22,sy_22_Yeoh,'r-')
hold off
```



Yeoh

Wikipedia page: https://www.wikiwand.com/en/Yeoh_(hyperelastic_model)

We have different types of hyperelastic materials, some are invariant-based such as neo-Hookean and Mooney-Rivlin, and some are stretch-based such as Ogden. We saw that Ogden worked well for predicting the stress-stretch dataset. But before going on to develop a more general approach for finding the parameters, let's see if we can work with an invariant-based model too. For that, I choose Yeoh model with a strain energy density of:

$W = \sum_{p=1}^{N} C_p (I_1 - 3)^p$. For the biaxial tension, it can be written as:

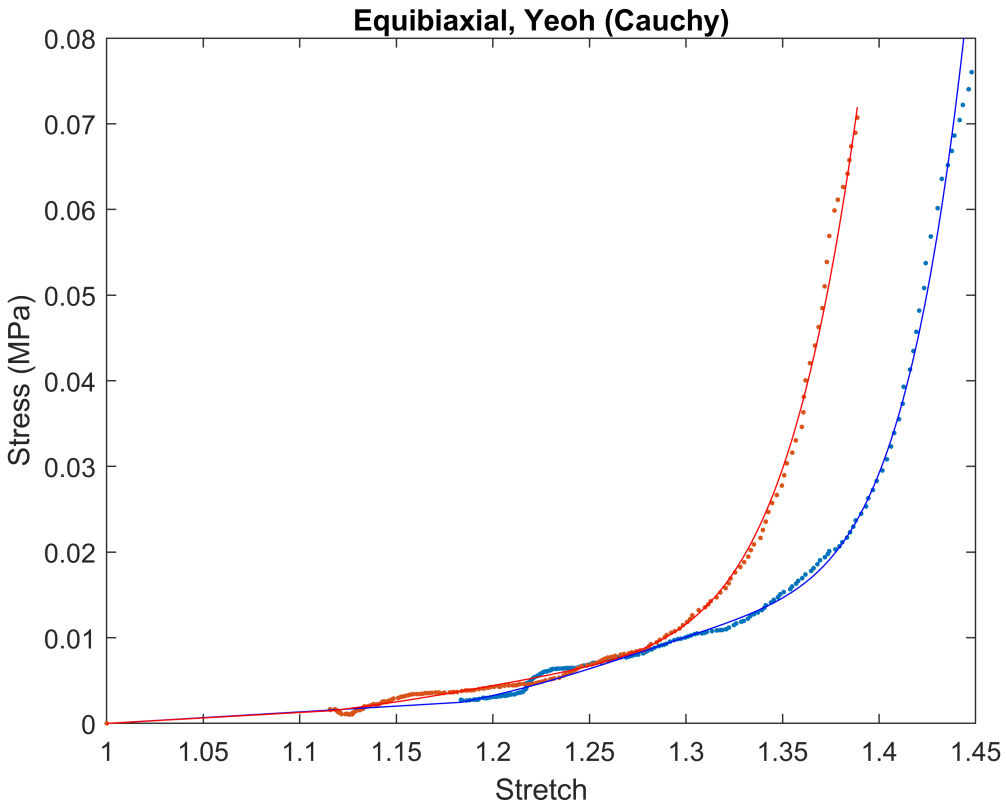
$\sigma_{11} = \sigma_{22} = 2(\lambda^2 - \lambda^{-4}) \frac{\partial W}{\partial I_1}$, with $\frac{\partial W}{\partial I_1} = \sum_{p=1}^{N} pC_p (I_1 - 3)^{p-1}$ and $I_1 = 2\lambda^2 + \lambda^{-4}$, or simply:

$$\sigma_{11} = \sigma_{22} = 2(\lambda^2 - \lambda^{-4}) \sum_{p=1}^{N} pC_p (2\lambda^2 + \lambda^{-4} - 3)^{p-1}$$

Yeoh on Equibiaxial

```
N = 6;
c0 = 0 * ones([1,N]);

[optC_11, ~, Yeoh_fun] = Yeoh_biaxial(ss_data_11, N, c0);
[optC_22, ~, ~] = Yeoh_biaxial(ss_data_22, N, c0);
```

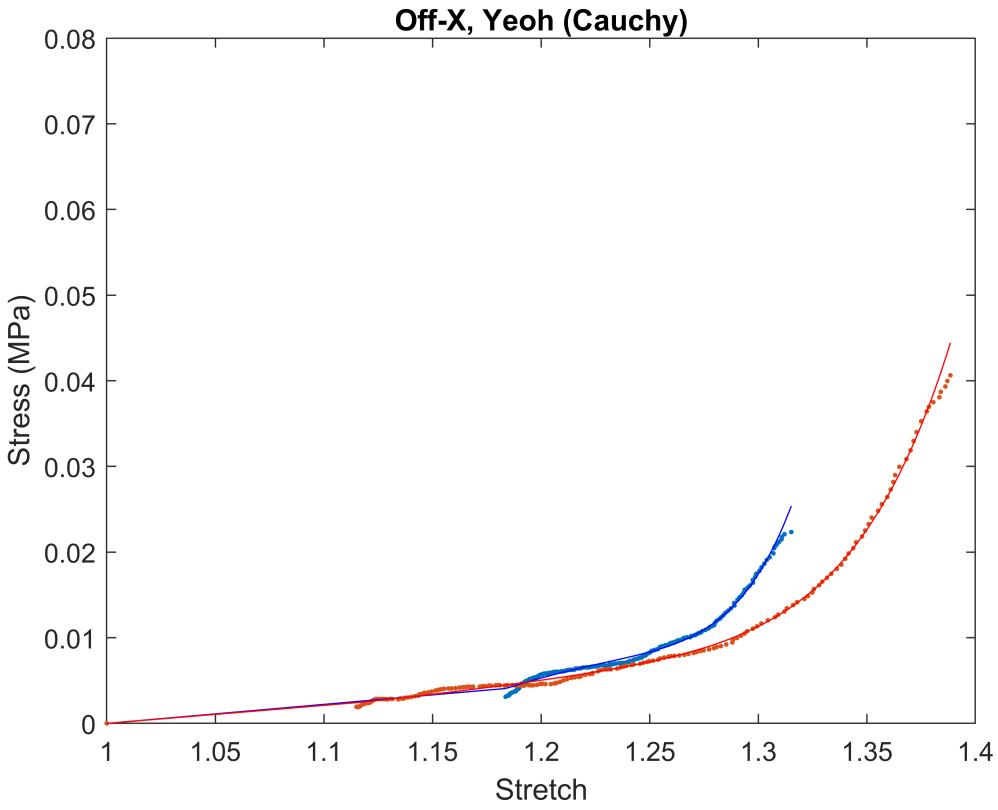
```
s11_Yeoh = Yeoh_fun(N, optC_11, lam11);
s22_Yeoh = Yeoh_fun(N, optC_22, lam22);

ss_plot(s_b,"Equibiaxial, Yeoh (Cauchy)")
ylim([0,0.08]);
hold on;
plot(lam11,s11_Yeoh,'b-')
plot(lam22,s22_Yeoh,'r-')
hold off
```




```
% [fval_11 fval_22]
```

Yeoh on Off-X

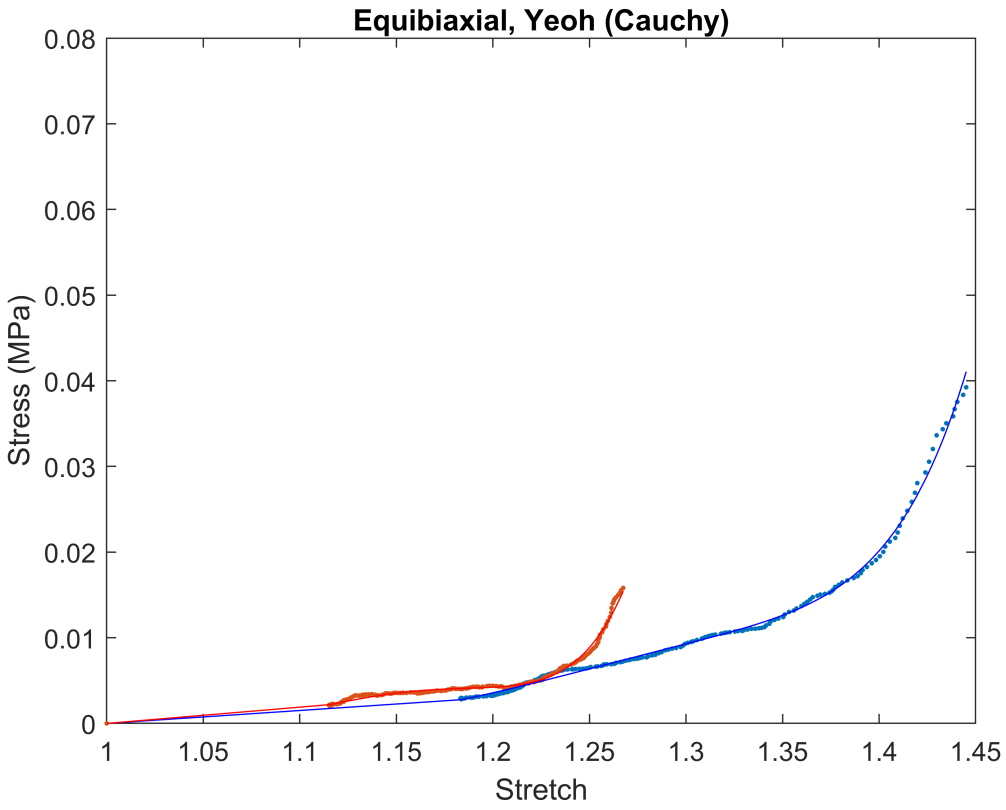
```
[optC_11, ~, Yeoh_fun] = Yeoh_biaxial(sx_data_11, N, c0);
[optC_22, ~, ~] = Yeoh_biaxial(sx_data_22, N, c0);


s11_Yeoh = Yeoh_fun(N, optC_11, sx_lam11);
s22_Yeoh = Yeoh_fun(N, optC_22, sx_lam22);

ss_plot(s_x,"Off-X, Yeoh (Cauchy)")
ylim([0,0.08]);
hold on;
plot(sx_lam11,s11_Yeoh,'b-')
plot(sx_lam22,s22_Yeoh,'r-')
hold off
```




```
% [fval_11 fval_22]
```

Yeoh on Off-Y

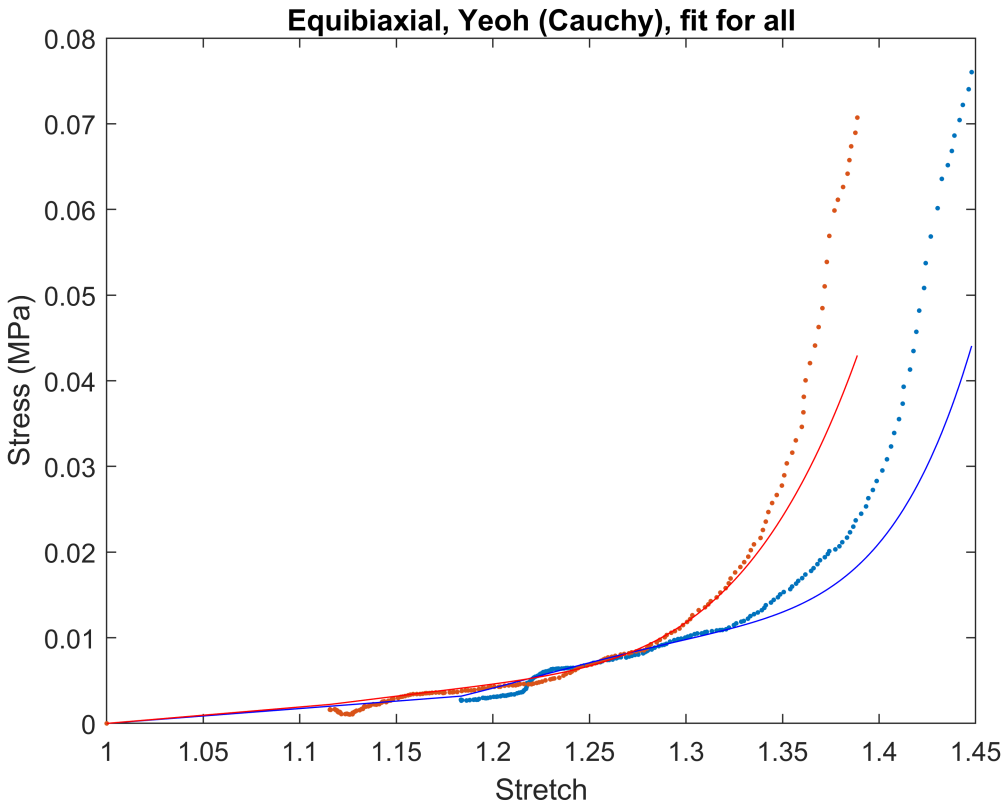
```
[optC_11, ~, Yeoh_fun] = Yeoh_biaxial(sy_data_11, N, c0);
[optC_22, ~, ~] = Yeoh_biaxial(sy_data_22, N, c0);


s11_Yeoh = Yeoh_fun(N, optC_11, sy_lam11);
s22_Yeoh = Yeoh_fun(N, optC_22, sy_lam22);

ss_plot(s_y,"Equibiaxial, Yeoh (Cauchy)")
ylim([0,0.08]);
hold on;
plot(sy_lam11,s11_Yeoh,'b-')
plot(sy_lam22,s22_Yeoh,'r-')
hold off
```



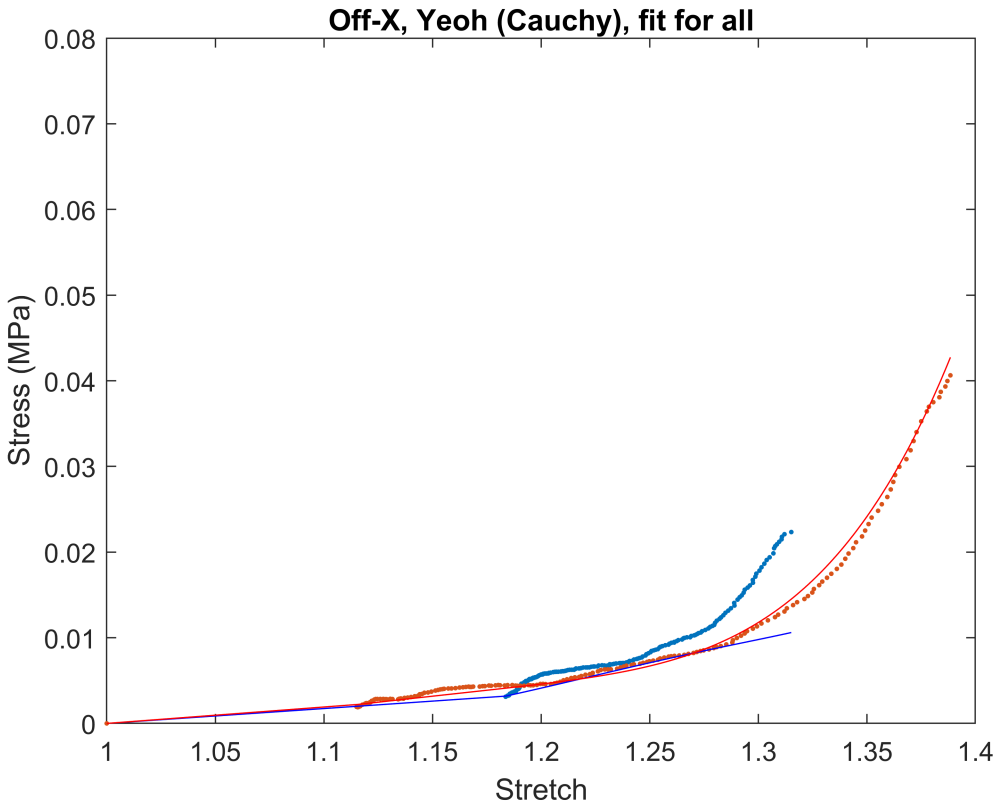

```
% [fval_11 fval_22]
```

Yeoh on all sets

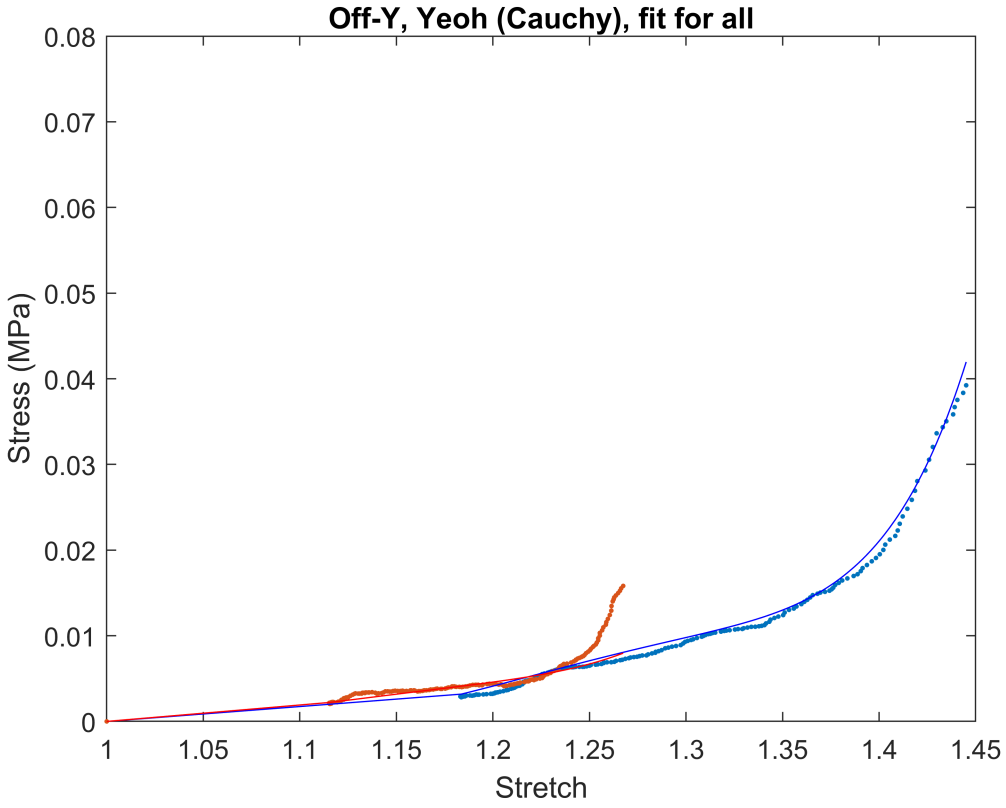
```
[optC_11, ~, Yeoh_fun] = Yeoh_biaxial([sb_data_11;sx_data_11;sy_data_11], N, c0);
[optC_22, ~, ~] = Yeoh_biaxial([sb_data_22;sx_data_22;sy_data_22], N, c0);

sb_11_Yeoh = Yeoh_fun(N, optC_11, sb_lam11);
sb_22_Yeoh = Yeoh_fun(N, optC_22, sb_lam22);
ss_plot(s_b,"Equibiaxial, Yeoh (Cauchy), fit for all")
ylim([0,0.08]);
hold on;
plot(sb_lam11,sb_11_Yeoh,'b-')
plot(sb_lam22,sb_22_Yeoh,'r-')
hold off
```




```
sx_11_Yeoh = Yeoh_fun(N, optC_11, sx_lam11);
sx_22_Yeoh = Yeoh_fun(N, optC_22, sx_lam22);
ss_plot(s_x,"Off-X, Yeoh (Cauchy), fit for all")
ylim([0,0.08]);
hold on;
plot(sx_lam11,sx_11_Yeoh,'b-')
plot(sx_lam22,sx_22_Yeoh,'r-')
hold off
```




```
sy_11_Yeoh = Yeoh_fun(N, optC_11, sy_lam11);
sy_22_Yeoh = Yeoh_fun(N, optC_22, sy_lam22);
ss_plot(s_y,"Off-Y, Yeoh (Cauchy), fit for all")
ylim([0,0.08]);
hold on;
plot(sy_lam11,sy_11_Yeoh,'b-')
plot(sy_lam22,sy_22_Yeoh,'r-')
hold off
```



2nd Piola-Kirchhoff stress method

For building a general solution, we should find the constants of a strain energy only by providing stress-stretch data and the formulation of the strain energy. The method then should calculate the corresponding relationship between stress and stretch by itself without providing any further relationships. To do so, the program should take the following steps:

Some assumptions

- The material is incompressible: $I_3 = \det(\mathrm{F}) = J = 1$
- Deformations are only due to $\lambda_1 = \lambda_x$ and $\lambda_2 = \lambda_y$; therefore, if we provide stretches for both of these directions, the stretch of the third direction can be estimated as $\lambda_3 = (\lambda_1\lambda_2)^{-1}$. For the available dataset, we should provide both λ_1 and λ_2; however, because samples were anisotropic, we cannot directly use the data for the models we have discussed so far. So, we need to pick a set of data (e.g. (λ_1, σ_1)) and approximate λ_2 based on the test from $\lambda_2 = \sqrt{\lambda_1}$ (off-biaxial) or $\lambda_2 = \lambda_1$ (equibiaxial). Note, for the discussed models, we can use only the equibiaxial dataset. Moreover, aside from changing λ_2, we should set $\sigma_2 = \sigma_1$.
- The boundary condition required for calculating p is $\sigma_{33} = \sigma_z = 0$.

Steps for $W = W(I)$

1. Takes stress and stretches in a tensor format, so it can build the deformation gradient tensor F
2. Calculates $\mathrm{C} = \mathrm{F}^T\mathrm{F}$.
3. Calculates the required invariants of $\mathrm{C} \rightarrow I_1 = tr(\mathrm{C}),\ I_2 = \frac{1}{2}(tr(\mathrm{C}^2) - tr(\mathrm{C})^2) = (\lambda_1\lambda_2)^2 + (\lambda_2\lambda_3)^2 + (\lambda_3\lambda_1)^2$
4. Calculates the derivatives of I_i wrt $\mathrm{C} \rightarrow \frac{\partial I_1}{\partial \mathrm{C}} = \mathrm{I},\ \frac{\partial I_2}{\partial \mathrm{C}} = tr(\mathrm{C})\mathrm{I} - \mathrm{C}^T,\ \frac{\partial I_3}{\partial \mathrm{C}} = \det(\mathrm{C})\mathrm{C}^{-T} = 0$
5. Calculates $\frac{\partial W}{\partial I_i}$ for $i = 1, 2, \ldots$ (if $W = W(I_i)$)
6. Calculates $\mathrm{S}' = 2\frac{\partial W}{\partial \mathrm{C}} = 2\frac{\partial W}{\partial I_i}\frac{\partial I_i}{\partial \mathrm{C}}$. For an incompressible isotropic material, it can be reduced to $\mathrm{S}' = 2\left(\frac{\partial W}{\partial I_1}\mathrm{I} + \frac{\partial W}{\partial I_2}(I_1\mathrm{I} - \mathrm{C})\right)$
7. Uses a boundary condition to calculates p for a relevant stress obtained from $\sigma = \frac{1}{J}\mathrm{F}\mathrm{S}'\mathrm{F}^T - p$ (note, $J = 1$)
8. Insert p in the σ formulation and calculates stress for the dataset.

Steps for $W = W(\lambda)$

1. Takes stress and stretches in a tensor format, so it can build the deformation gradient tensor F
2. Calculates $\frac{\partial W}{\partial \lambda_i}$ for $i = 1, 2, 3$ (if $W = W(\lambda_i)$
3. Calculates $\mathrm{S}' = \sum_{i=1}^{3}\frac{\partial W}{\partial \lambda_i}\frac{1}{\lambda_i}\mathrm{N}_i \otimes \mathrm{N}_i = \mathrm{diag}\left(\frac{1}{\lambda_1}\frac{\partial W}{\partial \lambda_1}, \frac{1}{\lambda_2}\frac{\partial W}{\partial \lambda_2}, \frac{1}{\lambda_3}\frac{\partial W}{\partial \lambda_3}\right)$
4. Uses a boundary condition to calculates p for a relevant stress obtained from $\sigma = \frac{1}{J}\mathrm{F}\mathrm{S}'\mathrm{F}^T - p$ (note, $J = 1$)
5. Insert p in the σ formulation and calculates stress for the dataset.

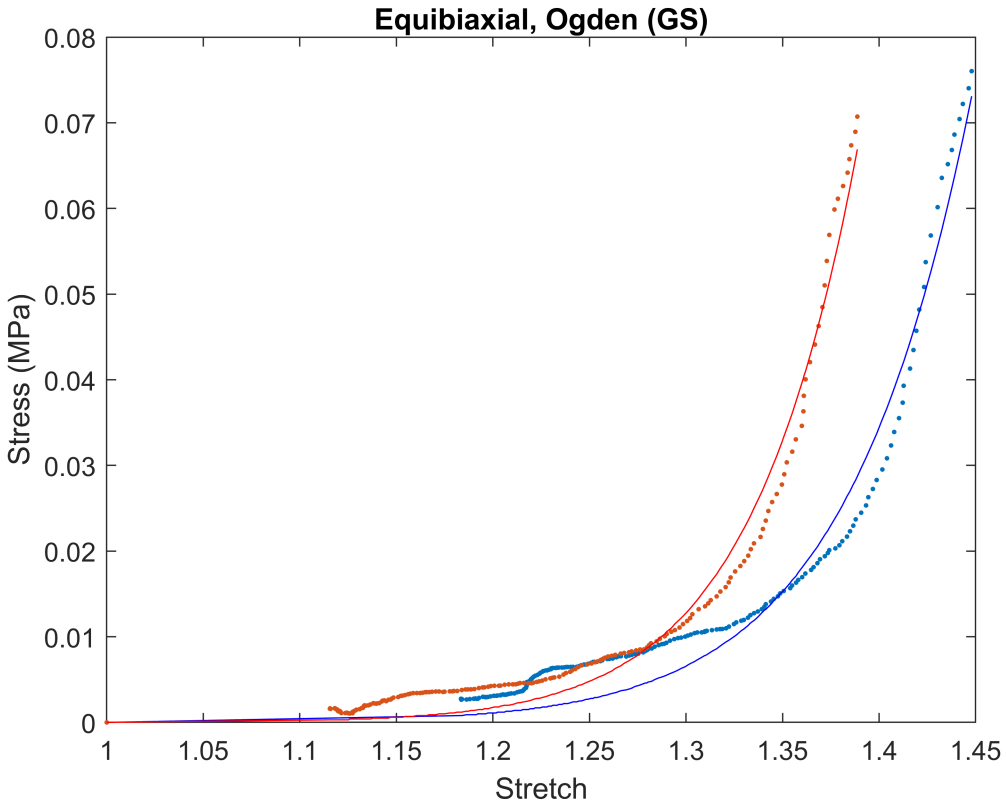
Data preparation

```
ss_data_11 = table2array(s_b(:,["Lambda11","Sigma11MPa"]));
ss_data_22 = table2array(s_b(:,["Lambda22","Sigma22MPa"]));
lam11 = ss_data_11(:,1);
lam22 = ss_data_22(:,1);
```

Implementing the steps for $W = W(\lambda)$ (Ogden)

```
N = 2;
mu0 = 1e-5 * ones([1,N]);
alpha0 = 10 * ones([1,N]);
c0 = [mu0,alpha0];
CF = 1;
[optC_11_GS, ~, W_func]  = W_calibrator([ss_data_11 ss_data_11],  'Ogden', c0, CF);
[optC_22_GS, ~, ~]       = W_calibrator([ss_data_22 ss_data_22],  'Ogden', c0, CF);
```

```
s11_Yeoh_GS = W_func([lam11 lam11], 'Ogden', optC_11_GS);
s22_Yeoh_GS = W_func([lam22 lam22], 'Ogden', optC_22_GS);

ss_plot(s_b,"Equibiaxial, Ogden (GS)")
ylim([0,0.08]);
hold on;
plot(lam11,s11_Yeoh_GS(:,1),'b-')
plot(lam22,s22_Yeoh_GS(:,1),'r-')
hold off
```

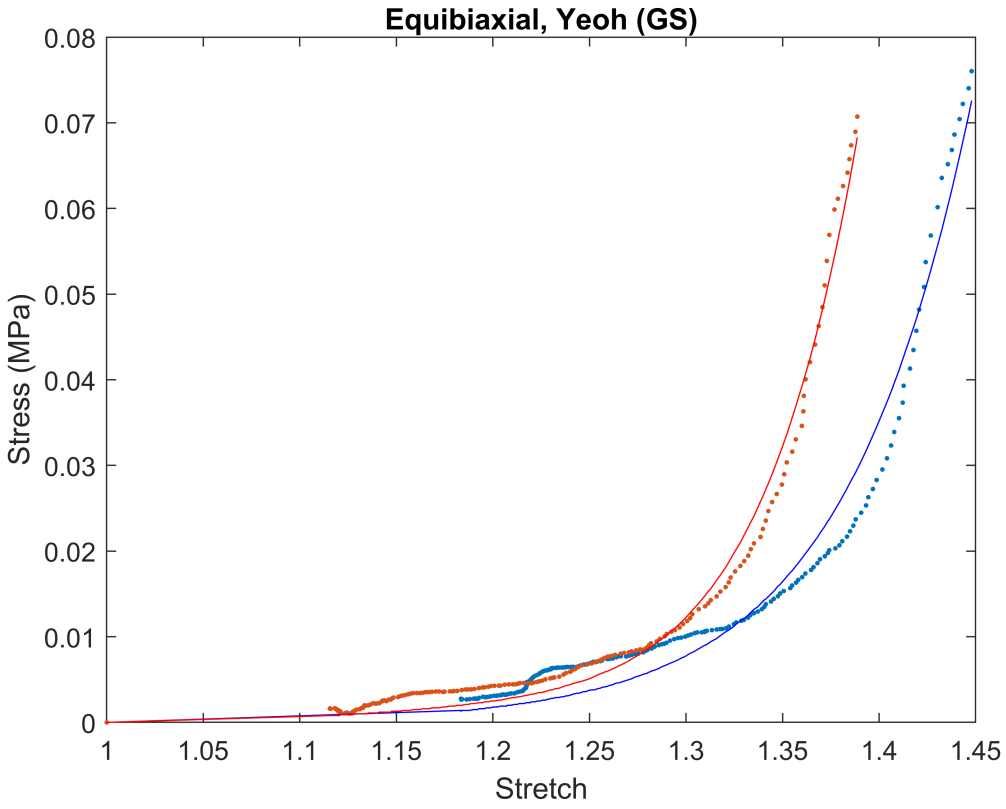



```
% [fval_11 fval_22]
```

Implementing the steps for $W = W(I)$ (Yeoh)

```
N = 6;
c0 = 0 * ones([1,N]);
CF = 1;
[optC_11_GS, fval_11, W_func]  = W_calibrator([ss_data_11 ss_data_11],  'Yeoh', c0, CF);
[optC_22_GS, fval_22, ~     ]  = W_calibrator([ss_data_22 ss_data_22],  'Yeoh', c0, CF);

s11_Yeoh_GS = W_func([lam11 lam11], 'Yeoh', optC_11_GS);
s22_Yeoh_GS = W_func([lam22 lam22], 'Yeoh', optC_22_GS);

ss_plot(s_b,"Equibiaxial, Yeoh (GS)")
```

```
ylim([0,0.08]);
hold on;
plot(lam11,s11_Yeoh_GS(:,1),'b-')
plot(lam22,s22_Yeoh_GS(:,2),'r-')
hold off
```




```
% [fval_11 fval_22]
```

The case for Gasser-Ogden-Holzapfel (GOH) model

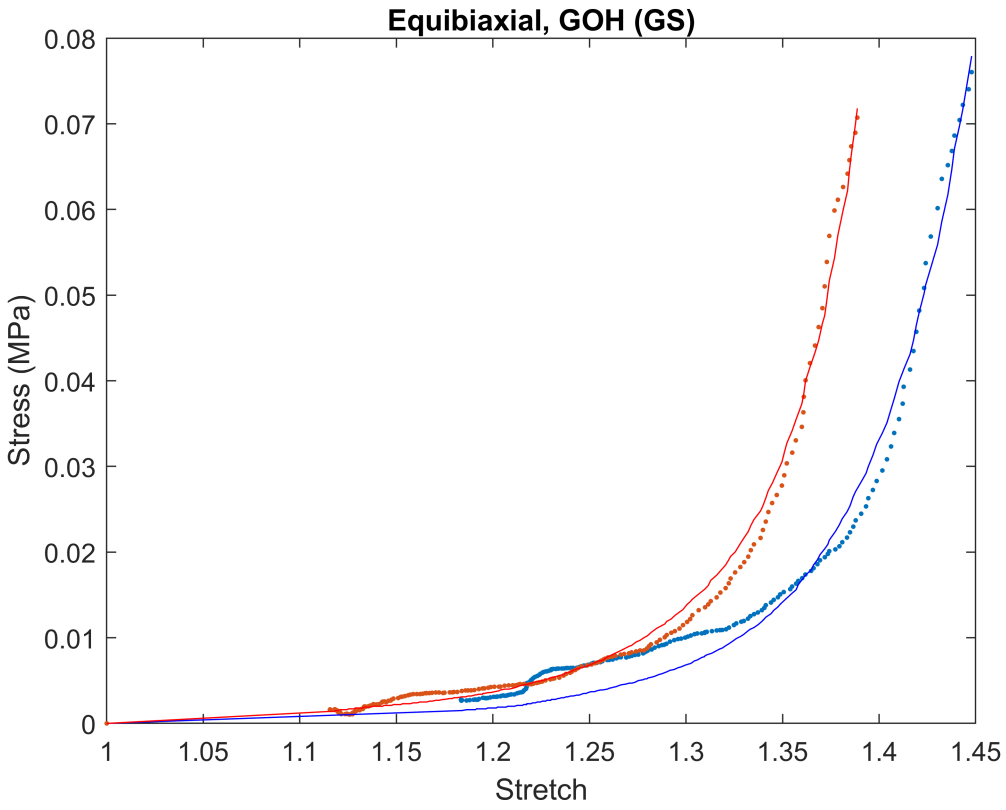
Equibiaxial

```
% DATA
ss_data_11 = table2array(s_b(:,["Lambda11","Sigma11MPa"]));
ss_data_22 = table2array(s_b(:,["Lambda22","Sigma22MPa"]));
lam11 = ss_data_11(:,1);
lam22 = ss_data_22(:,1);

% c: C10, k1, k2, kappa, [theta(degree)]
c0  = [0, 0, 0, 0, 0, 90];
CF = 1;
[optC_GS, fval, W_func]  = W_calibrator([ss_data_11 ss_data_22], 'GOH', c0, CF);
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 4).
```

```
% c0
% optC_GS
```

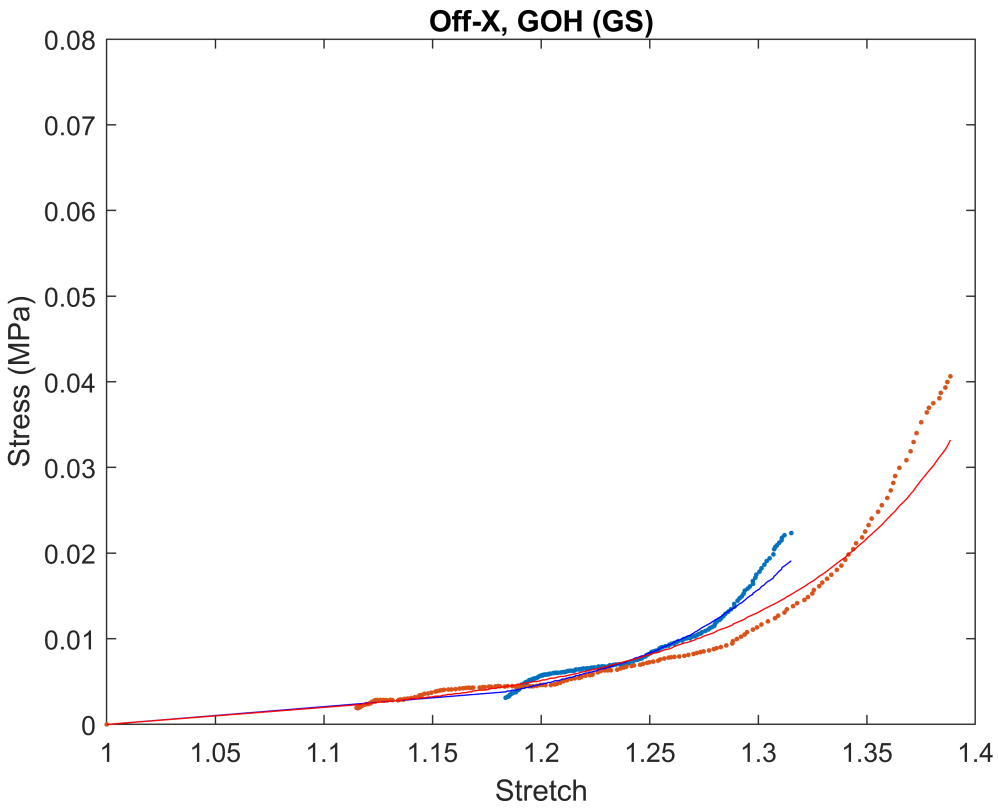
```
% fval

s_GOH_GS = W_func([lam11 lam22], 'GOH', optC_GS);
ss_plot(s_b,"Equibiaxial, GOH (GS)")
ylim([0,0.08]);
hold on;
plot(lam11,s_GOH_GS(:,1),'b-')
plot(lam22,s_GOH_GS(:,2),'r-')
hold off
```



Off-X

```
% DATA
ss_data_11 = table2array(s_x(:,["Lambda11","Sigma11MPa"]));
ss_data_22 = table2array(s_x(:,["Lambda22","Sigma22MPa"]));
lam11 = ss_data_11(:,1);
lam22 = ss_data_22(:,1);

% c: C10, k1, k2, kappa, [theta(degree)]
c0  = [0, 0, 0, 0, 0, 90];
CF = 2;
[optC_GS, fval, W_func]  = W_calibrator([ss_data_11 ss_data_22], 'GOH', c0, CF);
% c0
% optC_GS
% fval
```

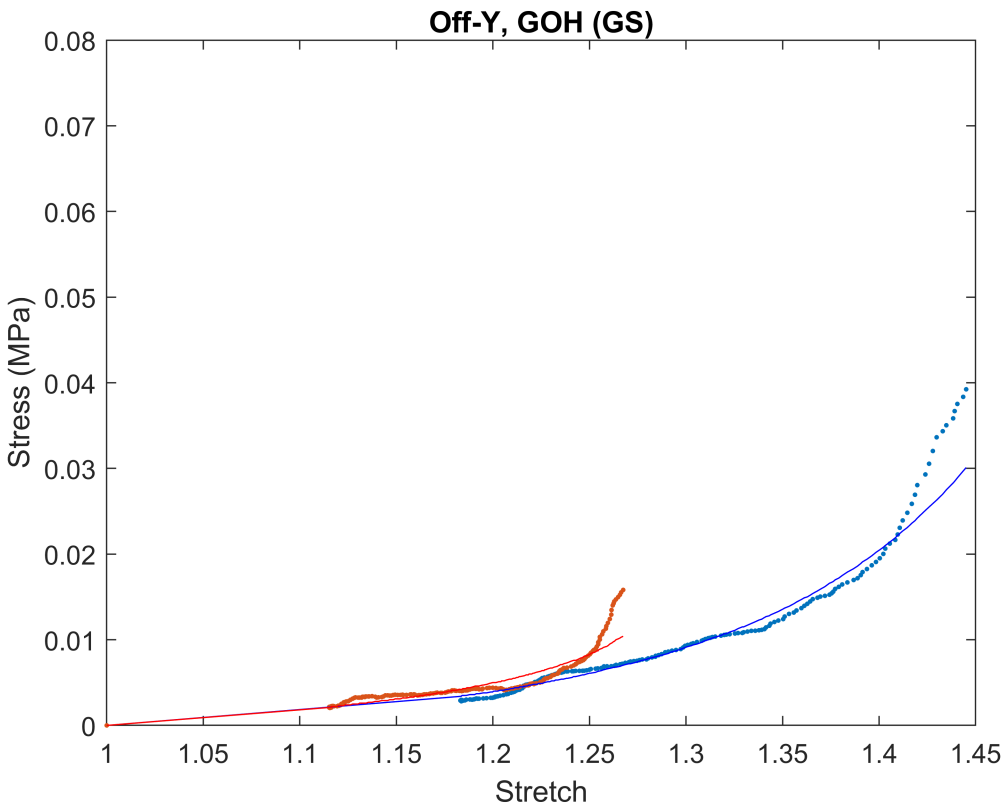
```
s_GOH_GS = W_func([lam11 lam22], 'GOH', optC_GS);
ss_plot(s_x,"Off-X, GOH (GS)")
ylim([0,0.08]);
hold on;
plot(lam11,s_GOH_GS(:,1),'b-')
plot(lam22,s_GOH_GS(:,2),'r-')
hold off
```



Off-Y

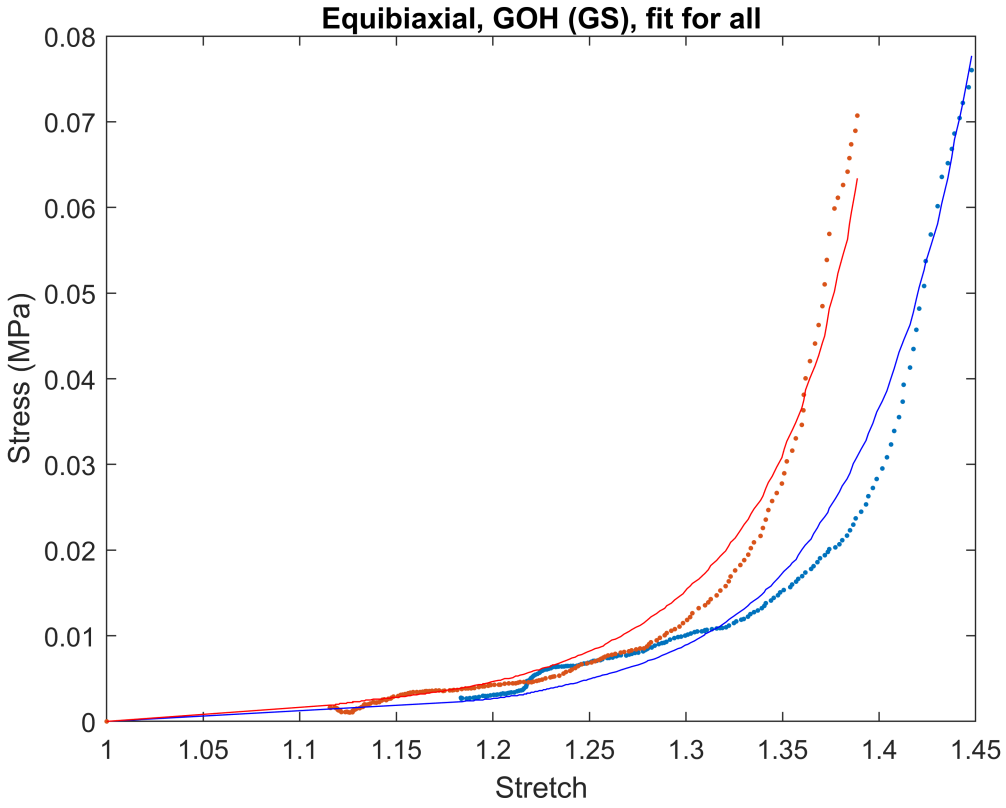
```
% DATA
ss_data_11 = table2array(s_y(:,["Lambda11","Sigma11MPa"]));
ss_data_22 = table2array(s_y(:,["Lambda22","Sigma22MPa"]));
lam11 = ss_data_11(:,1);
lam22 = ss_data_22(:,1);

% c: C10, k1, k2, kappa, [theta(degree)]
c0  = [0, 0, 0, 0, 0, 90];
CF = 2;
[optC_GS, fval, W_func]  = W_calibrator([ss_data_11 ss_data_22], 'GOH', c0, CF);
% c0
% optC_GS
% fval

s_GOH_GS = W_func([lam11 lam22], 'GOH', optC_GS);
ss_plot(s_y,"Off-Y, GOH (GS)")
```

```
ylim([0,0.08]);
hold on;
plot(lam11,s_GOH_GS(:,1),'b-')
plot(lam22,s_GOH_GS(:,2),'r-')
hold off
```

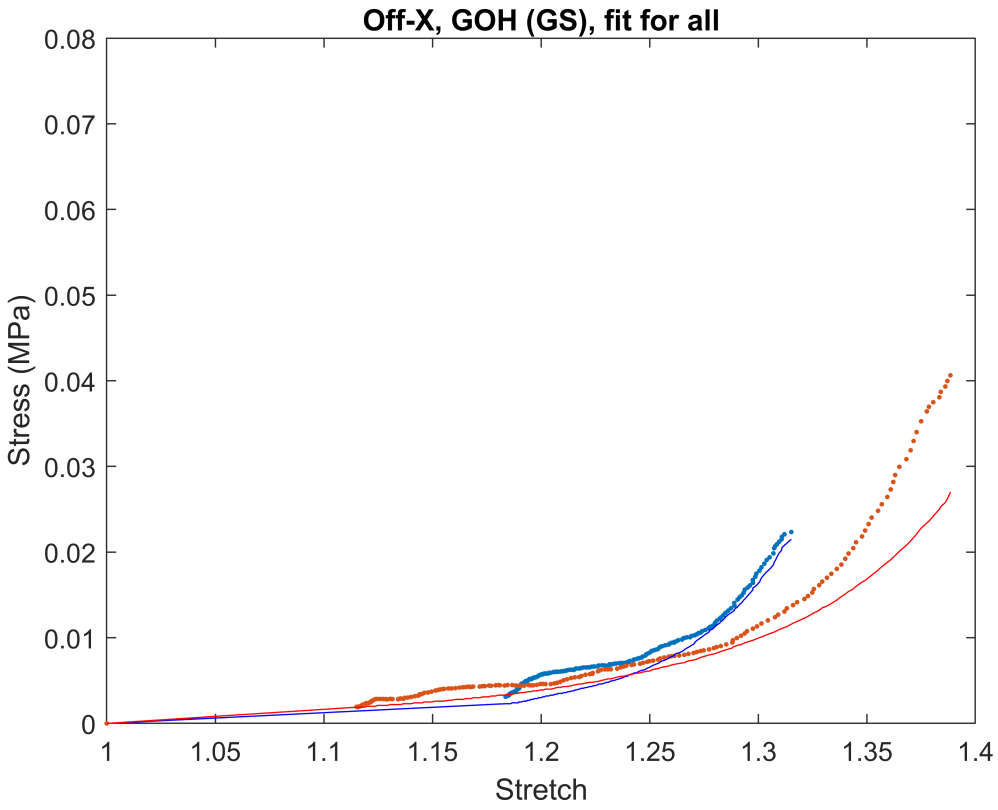


All 3 sets of data

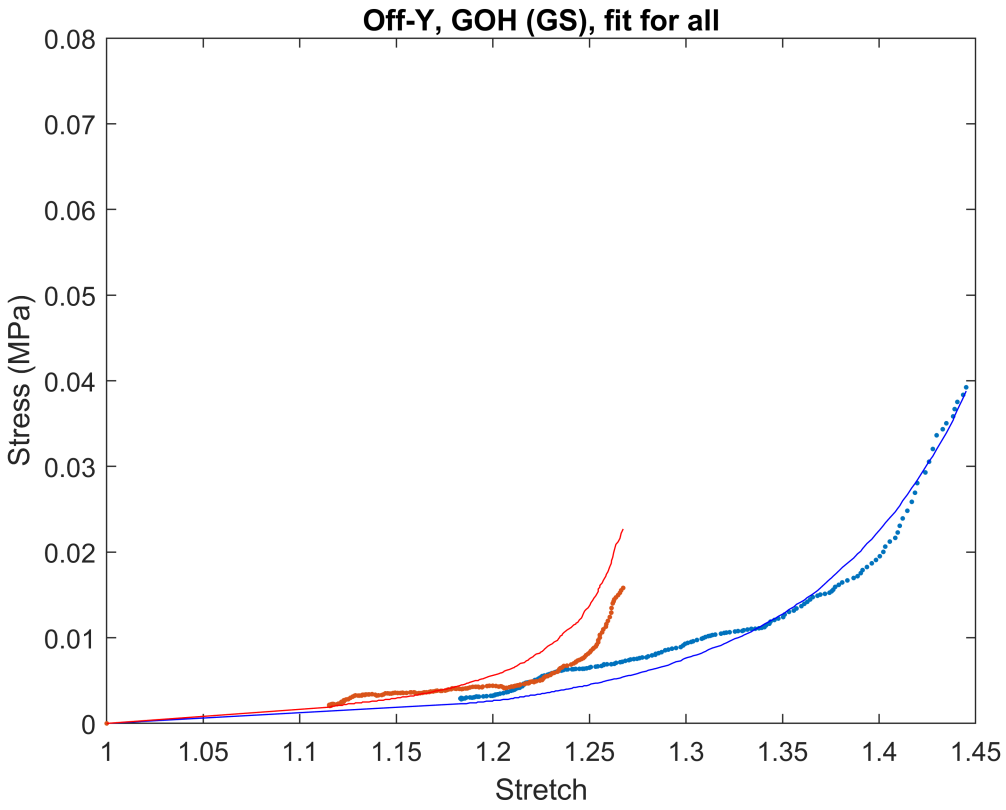
```
ss_data_11 = [sb_data_11; sx_data_11; sy_data_11];
ss_data_22 = [sb_data_22; sx_data_22; sy_data_22];

% c: C10, k1, k2, kappa, [theta(degree)]
c0  = [0, 0, 0, 0, 0, 90];
CF = 1;
[optC_GS, fval, W_func]  = W_calibrator([ss_data_11 ss_data_22], 'GOH', c0, CF);
% c0
% optC_GS
% fval

sb_GOH_GS = W_func([sb_lam11 sb_lam22], 'GOH', optC_GS);
ss_plot(s_b,"Equibiaxial, GOH (GS), fit for all")
ylim([0,0.08]);
hold on;
plot(sb_lam11,sb_GOH_GS(:,1),'b-')
plot(sb_lam22,sb_GOH_GS(:,2),'r-')
hold off
```




```
sx_GOH_GS = W_func([sx_lam11 sx_lam22], 'GOH', optC_GS);
ss_plot(s_x,"Off-X, GOH (GS), fit for all")
ylim([0,0.08]);
hold on;
plot(sx_lam11,sx_GOH_GS(:,1),'b-')
plot(sx_lam22,sx_GOH_GS(:,2),'r-')
hold off
```




```
sy_GOH_GS = W_func([sy_lam11 sy_lam22], 'GOH', optC_GS);
ss_plot(s_y,"Off-Y, GOH (GS), fit for all")
ylim([0,0.08]);
hold on;
plot(sy_lam11,sy_GOH_GS(:,1),'b-')
plot(sy_lam22,sy_GOH_GS(:,2),'r-')
hold off
```



Functions

ss_plot

```
function ss_plot(dataset,name)
figure
plot(dataset.Lambda11,dataset.Sigma11MPa,'.')
hold on
plot(dataset.Lambda22,dataset.Sigma22MPa,'.')

title(name);
xlabel('Stretch')
ylabel('Stress (MPa)')

end
```

Neo-Hookean, Cauchy stress method

$$\sigma_{11} = \sigma_{22} = 2C_1\left(\lambda^2 - \frac{1}{\lambda^4}\right)$$

```
function [optC, nH_fun] = nH_biaxial(ss_data, c0)
% ss_data : stretch | stress
t = ss_data(:,1); % stretch
s = ss_data(:,2); % stress
```

```
% nH function
syms lam
c       = sym('c', [1]);
nH_sym  = 2*c(1) * (lam.^2 - lam.^-4);
nH_fun  = matlabFunction(nH_sym, 'Vars', {c, lam});

% Sum-Squared-Error Cost Function
ssecf = @(c) sum((nH_fun(c, t) - s).^2);

% Minimization
[optC,~] = fminsearch(ssecf,c0);
end
```

Mooney-Rivlin, Cauchy stress method

$$\sigma_{11} = \sigma_{22} = 2C_1\left(\lambda^2 - \frac{1}{\lambda^4}\right) - 2C_2\left(\lambda^2 - \frac{1}{\lambda^4}\right)$$

```
function [optC, fval, MR_fun] = MR_biaxial(ss_data, c0)
% ss_data : stretch | stress
t = ss_data(:,1); % stretch
s = ss_data(:,2); % stress

% MR function
syms lam
c       = sym('c', [1,2]);
MR_sym  = 2*c(1) * (lam.^2 - lam.^-4) - 2*c(2) * (lam.^2 - lam.^-4);
MR_fun  = matlabFunction(MR_sym, 'Vars', {c, lam});

% Sum-Squared-Error Cost Function
ssecf = @(c) sum((MR_fun(c, t) - s).^2);

% Minimization
[optC,fval] = fminsearch(ssecf,c0);
end
```

Ogden, Cauchy stress method

$$\sigma_{11} = \sigma_{22} = \sum_{p=1}^{N} \mu_p\left(\lambda^{\alpha_p} - \lambda^{-2\alpha_p}\right)$$

```
function stress = Ogden_sigma_biaxial(N,c,lam)
%OGDEN_SIGMA_BIAXIAL calculates stress for incompressible Ogden material
% Data collection
mu      = c(1:N);
alpha   = c(1+N:2*N);
% Stress calculation
stress = zeros(size(lam));
for p = 1:N
    stress = stress + mu(p) * (lam.^alpha(p) - lam.^(-2*alpha(p)));
```

```
end
end
```

```
function [optC, fval, Ogden_fun] = Ogden_biaxial(ss_data, N, c0)
% ss_data : stretch | stress
t = ss_data(:,1); % stretch
s = ss_data(:,2); % stress

% Ogden function
% c = [N, mu, alpha]
% size(mu) = size(alph) = 1 x N
Ogden_fun = @(N,c,lam) Ogden_sigma_biaxial(N,c,lam);

% Sum-Squared-Error Cost Function
ssecf = @(c) sum((Ogden_fun(N, c, t) - s).^2);

% Minimization
options = optimset('Display','none');
[optC,fval] = fminsearch(ssecf,c0,options);
end
```

Yeoh, Cauchy stress method

$$\sigma_{11} = \sigma_{22} = 2(\lambda^2 - \lambda^{-4}) \sum_{p=1}^{N} pC_p(2\lambda^2 + \lambda^{-4} - 3)^{p-1}$$

```
function stress = Yeoh_sigma_biaxial(N,c,lam)
%YEOH_SIGMA_BIAXIAL calculates stress for incompressible Yeoh material
% Check data validity
classes = {'numeric'};
attributes = {'size',[1,N]};
validateattributes(c,classes,attributes);
% Stress calculation
stress = zeros(size(lam));
lam_p2 = lam.^2;
lam_n4 = lam.^-4;

for p = 1:N
    stress = stress + p * c(p) * (2*lam_p2 + lam_n4 - 3) .^ (p-1);
end
stress = stress .* (lam_p2 - lam_n4) * 2;

end
```

```
function [optC, fval, Yeoh_fun] = Yeoh_biaxial(ss_data, N, c0)
% ss_data : stretch | stress
```

```
t = ss_data(:,1); % stretch
s = ss_data(:,2); % stress

% Ogden function
% size(c) = 1 x N
Yeoh_fun = @(N,c,lam) Yeoh_sigma_biaxial(N,c,lam);

% Sum-Squared-Error Cost Function
% ssecf = @(c) sum((Yeoh_fun(N, c, t) - s).^2);
mre = @(e,p) abs((e-p)./max(prot0(e),prot0(p))); % Modified Relative Error
smrecf = @(c) 100 * sum(mre(Yeoh_fun(N, c, t) , s));

% Minimization
% [optC,fval] = fminsearch(ssecf,c0);
[optC,fval] = fminsearch(smrecf,c0);

end
```

zero-correction for mRE cost function

```
function x = prot0(x)
eps = 1e-12;
x(x==0) = eps;
end
```

General Solver

Yeoh strain energy

```
function W = Yeoh_energy(c,I)
I1 = I(1);
W = 0;
for i = 1:numel(c)
    W = W + c(i) * (I1 - 3).^i;
end
W = sum(W);
end
```

Derivative of Yeoh strain energy wrt I_1 (only for test)

```
function dWI1 = Yeoh_dI1(N,c,I1)
dWI1 = 0;
for i = 1:N
    dWI1 = dWI1 + i * c(i) * (I1 - 3).^(i-1);
end
end
```

Ogden strain energy

```
function W = Ogden_energy(c,lambda)
N       = numel(c) / 2;
mu      = c(1:N);
```

```
a       = c(1+N:2*N);
lam1    = lambda(:,1);
lam2    = lambda(:,2);
lam3    = lambda(:,3);
W = 0;
for i = 1:N
    W = W + mu(i)/a(i) * (lam1.^a(i) + lam2.^a(i) + lam3.^a(i) - 3);
end
end
```

Derivative of Ogden strain energy wrt λ_i (only for test)

```
function dWL = Ogden_dL(N,c,lambda)
mu      = c(1:N);
a       = c(1+N:2*N);
% lam1    = lambda(:,1);
% lam2    = lambda(:,2);
% lam3    = lambda(:,3);
dWL     = 0;
for i = 1:N
    dWL = dWL + mu(i) * lambda.^(a(i)-1);
end
end
```

GOH strain energy

```
function W = GOH_energy(c,I)
% c: C10, k1, k2, kappa
% I: I1, [I4]
C1  = c(1);
k1  = c(2);
k2  = c(3);
kap = c(4);
n_I4= numel(I) - 2;

W_iso   = C1 * (I(1)-3);
% W_an    = 0;
E(n_I4) = 0;
% for i = 1:n_I4
%     E(i) = kap * (I(1)-3) + (1-3*kap) * (I(i+1)-1);
% end
E = kap * (I(1)-3) + (1-3*kap) * (I(3:3+n_I4-1)-1);
E = (abs(E) + E)/2;
W_an = k1 / (2 * k2) * sum(exp((k2 * E.^2)-1),'all');

W = W_iso + W_an;
end
```

Stress calculator for $W(I)$

```
function sigma = W_I_stress(lambda,  W_model, W_par)
```

```
% List of parameters
% lambda: stretch data in the format of [stretch_1|stretch_2]
% W     : strain energy function
% W_par : parameters of W

del_I   = 1e-6; % delta_I for calculating derivative of W wrt I

% Assigining W and W_par
switch W_model
    case 'Yeoh'
        I2      = false;
        I4      = false;
        c0      = W_par;
%         N       = numel(c0);
        n_I     = 1 + 1; % number of dependent invariants: I1 + I2(reserved)
        I_list  = 1; % list of dependent invariants

        W       = @(c,I1) Yeoh_energy(c,I1);

    case 'GOH'
        I2      = false;
        I4      = true;
        c0      = W_par;
        n_I4    = (numel(c0) - 4);
        n_I     = 1 + 1 + n_I4; % I1 + I2(reserved) + I4(i)
        n_I4_ls = 2;
        I_list  = [1 3:n_I]; % list of dependent invariants

        W       = @(c,I) GOH_energy(c,I);

        for i = 1:n_I4
            g(i,:) = [cosd(c0(4+i)), sind(c0(4+i)), 0];
            g(i,:) = g(i,:)./norm(g(i,:));
        end

    otherwise
        error(['Error: the selected model (', W_model, ') is not available.']);
end

% Construct F
[F, data_size] = F_construct(lambda);

% pre-allocations
C       = zeros(3,3,data_size);
I       = zeros(data_size,n_I);
dWI     = zeros(data_size,n_I);
sigma   = zeros(data_size,2);
p       = zeros(data_size,1);
```

```
% Stress calculation
% if algo == 1
% Calculate C, dev(W,I), S', p, s1, s2
for i = 1:data_size
    %     Calculate C
    C(:,:,i) = F(:,:,i) .* transpose(F(:,:,i));
    %     Calculate the invariants
    inv = 1;
    I(i,inv) = sum(diag(C(:,:,i)));
    if I2
        inv = inv + 1;
        I(i,2) = ...
            (F(1,1,i) * F(2,2,i)) ^ 2 ...
            + (F(2,2,i) * F(3,3,i)) ^ 2 ...
            + (F(3,3,i) * F(1,1,i)) ^ 2;
    end
    if I4
        for j = 1:n_I4
            inv = I_list(n_I4_ls+j-1);
%           inv = inv + 1;
            I(i,inv) = g(j,:) * C(:,:,i) * g(j,:)';
        end
    end
    %     Calculate dW/dI
    for inv = I_list
        I_p         = I(i,:);
        I_p(inv)    = I_p(inv) + del_I;
        I_n         = I(i,:);
        I_n(inv)    = I_n(inv) - del_I;

        dWI(i,inv)  = (W(c0,I_p) -  W(c0,I_n)) / (2 * del_I);
    end
    %     Calculate S'
    S_PK2 = 2* (dWI(i,1)*eye(3) + dWI(i,2)*(I(i,1)*eye(3) - C(:,:,i)));
    if I4
        for j = 1:n_I4
            inv = I_list(n_I4_ls+j-1);
            S_PK2 = S_PK2 + 2 * dWI(i,inv) * (g(j,:)'*g(j,:));
        end
    end
    %     Calculate pressure, using BC: sigma_33 = 0
    p(i,1)      = F(3,3,i)^2 * S_PK2(3,3);
    %     Calculate sigma_11 = sigma_22
    sigma(i,1)  = round( F(1,1,i)^2 * S_PK2(1,1) + p(i,1) , 4);
    sigma(i,2)  = round( F(2,2,i)^2 * S_PK2(2,2) + p(i,1) , 4);
end
% end
end
```

Stress calculator for $W(\lambda)$

```
function sigma = W_L_stress(lambda,  W_model, W_par)

% List of parameters
% lambda: stretch data in the format of [stretch_1|stretch_2]
% W     : strain energy function
% W_par : parameters of W

lambda(:,3) = (lambda(:,1).*lambda(:,2)).^-1;
del_L   = 1e-6; % delta_I for calculating derivative of W wrt I

% Assigining W, W_par, and algo
switch W_model
    case 'Ogden'
        W   = @(c,lambda) Ogden_energy(c,lambda);
        c0  = W_par;
        N   = numel(c0) / 2;

    otherwise
        error(['Error: the selected model (', W_model, ') is not available.']);
end

% Construct F
[F, data_size] = F_construct(lambda);

% pre-allocations
dWL     = zeros(data_size,3);
sigma   = zeros(data_size,2);
p       = zeros(data_size,1);

% Stress calculation
% Calculate dev(W,lambda), S', p, s1, s2
for i = 1:data_size
    %     Calculate dW/dL
%     dWL(i,:)   = Ogden_dL(N,c0,lambda(i,:));
    for dir = 1:3
        lam_p       = lambda(i,:);
        lam_p(dir)  = lam_p(dir) + del_L;
        lam_n       = lambda(i,:);
        lam_n(dir)  = lam_n(dir) - del_L;

        dWL(i,dir)    = (W(c0,lam_p) -  W(c0,lam_n)) / (2 * del_L);
    end
    %     Calculate S'
    S_PK2       = diag(dWL(i,:)./lambda(i,:));
    %     Calculate pressure, using BC: sigma_33 = 0
    p(i,1)      = F(3,3,i)^2 * S_PK2(3,3);
    %     Calculate sigma_11 = sigma_22
    sigma(i,1)  = round( F(1,1,i)^2 * S_PK2(1,1) + p(i,1) , 4);
    sigma(i,2)  = round( F(2,2,i)^2 * S_PK2(2,2) + p(i,1) , 4);
end
```

```
end
```

Construct F

```
function [F,data_size] = F_construct(lambda)
data_size = size(lambda,1);
F       = zeros(3,3,data_size); % pre-allocation
i       = 1;
loop    = true;
while loop
    F(:,:,i) = zeros(3);
    F(1,1,i) = lambda(i,1);
    F(2,2,i) = lambda(i,2);
    F(3,3,i) = 1/(lambda(i,1) * lambda(i,2));

%     if F(3,3,i) == inf
%         lambda(i,:) = [];
%         F(:,:,end)  = [];
%         data_size   = data_size - 1;
%         i           = i - 1;
%     end
    i = i + 1;
    if i > data_size
        loop = false;
    end
end
end
```

Optimizer

```
function [optC, fval, W_func] = W_calibrator(ss_data, W_model, W_par,CF_option)

% s0_weight = 10; % weight of enforcing stress(stretch = 0) = 0 in CF

% classes = {'numeric'};
% Assigining type of algorithm and check parameters' validity
min_const = false;
switch W_model
    case 'Yeoh'
%         algo    = 1;
        W_func = @(lambda,  W_model, W_parameters) W_I_stress(lambda,  W_model, W_parameters);

    case 'GOH'
%         algo    = 1;
        W_func = @(lambda,  W_model, W_parameters) W_I_stress(lambda,  W_model, [W_parameters W
        W_par(5:end) = [];
        min_const = true;
        lb = zeros([1,numel(W_par)]);
        ub = 5 * ones([1,numel(W_par)]);
        ub([3 4]) = [15 1/3];
```

```
    case 'Ogden'
%         algo    = 2;
        W_func = @(lambda,  W_model, W_parameters) W_L_stress(lambda,  W_model, W_parameters);

    otherwise
        error(['Error: the selected model (', W_model, ') is not available.']);
end

% switch W_model
%     case 'GOH'
%         min_const = true;
%         lb = -inf * ones([1,numel(W_par)]);
%         ub = -lb;
%         lb(4) = 0;
%         ub(4) = 1/3;
% end


lambda  = [ss_data(:,1) ss_data(:,3)];
stress  = [ss_data(:,2) ss_data(:,4)];

% switch algo
%     case 1
%         W_func = @(lambda,  W_model, W_parameters) W_I_stress(lambda,  W_model, W_parameters)
%     case 2
%         W_func = @(lambda,  W_model, W_parameters) W_L_stress(lambda,  W_model, W_parameters)
% end


% Fitting Functions
sseff   = @(c) sum((W_func(lambda,  W_model, c) - stress).^2, 'all');

mre     = @(e,p) abs((e-p)./min(prot0(e),prot0(p))); % Modified Relative Error
smreff  = @(c) sum(mre(W_func(lambda,  W_model, c), stress),'all');

re     = @(e,p) abs(mean(e-p)./min(mean(e),mean(p))); % Modified Relative Error
sreff  = @(c) sum(re(W_func(lambda,  W_model, c), stress),'all');

at_00   = @(c) sum(abs(W_func([1 1],  W_model, c) - [0 0]),'all');

% Cost Functions
switch CF_option
    case 1
        CF = @(c) sseff(c) + 10 * at_00(c);
    case 2
        CF = @(c) smreff(c);% + at_00(c);
    case 3
        CF = @(c) sreff(c) + 10 * at_00(c);
end
```

```
% Minimization
if min_const
%     options = optimoptions('fmincon','Algorithm','sqp');
%     [optC,fval] = fmincon(CF,W_par,[],[],[],[],lb,ub,[],options);

    options = optimoptions('particleswarm','HybridFcn',@fmincon,'MinNeighborsFraction',0.5,'Dis
    [optC,fval] = particleswarm(CF,numel(W_par),lb,ub,options);

else
    [optC,fval] = fminsearch(CF,W_par);
end

% options = optimoptions('particleswarm','SwarmSize',3*numel(W_par),'HybridFcn',@fminsearch,'Di
% [optC,fval] = particleswarm(ssecf,numel(W_par),[],[],options);

% ub = ones(size(W_par));
% lb = -ub;
% options = optimoptions('particleswarm','HybridFcn',@fmincon,'Display','none');
% [optC,fval] = particleswarm(ssecf,numel(W_par),lb,ub,options);
end
```