We're passing in two matrices, B11 and B12, which have been computed ahead of time (they don't depend on the sRGB value being passed in). I'll describe how to compute them below, after the end of the ILSS function.

```
function rho=ILSS(B11,B12,sRGB)

% This is the Iterative Least Slope Squared (ILSS) algorithm for generating
% a "reasonable" reflectance curve from a given sRGB color triplet.
% The reflectance spans the wavelength range 380-730 nm in 10 nm increments.
% It solves min sum(rho_i+1 - rho_i)^2 s.t. T rho = rgb, K1 rho = 1, K0 rho = 0,
% using Lagrangian formulation and iteration to keep all rho (0-1].
% B11 is upper-left 36x36 part of inv([D,T';T,zeros(3)])
% B12 is upper-right 36x3 part of inv([D,T';T,zeros(3)])
% sRGB is a 3-element vector of target D65-referenced sRGB values in 0-255 range,
% rho is a 36x1 vector of reflectance values (0->1] over wavelengths 380-730 nm,
% Written by Scott Allen Burns, 4/26/15.
% Licensed under a Creative Commons Attribution-ShareAlike 4.0 International
% License (http://creativecommons.org/licenses/by-sa/4.0/).
% For more information, see http://scottburns.us/reflectance-curves-from-srgb/
```

This is Matlab-ese for creating a 36 x 1 column vector of all 0.5:

```
rho=ones(36,1)/2; % initialize output to 0.5
```

Set the lower bound on rho for the optimization

```
rhomin=0.00001; % smallest refl value
```

Edge cases:

```
% handle special case of (255,255,255)
if all(sRGB==255)
    rho=ones(36,1);
    return
end
% handle special case of (0,0,0)
if all(sRGB==0)
    rho=rhomin*ones(36,1);
    return
end

% compute target linear rgb values
```

The (:) converts sRGB to a 3x1 column vector, in case it was passed in as a 1x3 row vector. It is then divided by 255 to put it in the 0-1 range.

```
sRGB=sRGB(:)/255; % convert to 0-1 column vector
```

Initialize 3x1 column vector rgb and compute linear rgb by removing gamma correction.

```
rgb=zeros(3,1);
% remove gamma correction to get linear rgb
for i=1:3
    if sRGB(i)<0.04045
        rgb(i)=sRGB(i)/12.92;
    else
        rgb(i)=((sRGB(i)+0.055)/1.055)^2.4;
    end
end
```

Compute this matrix product for use later. No point in recomputing it over and over inside the loop below!
R is 36x1, B12 is 36x3, and rgb is 3x1.

```
R=B12*rgb;
```

Initialize:

```
% iteration to get all refl 0-1
maxit=10; % max iterations
count=0; % counter for iteration
```

Here's how to read the following Matlab code. The phrase "rho>1" returns a 36x1 logical vector of True or False values, corresponding to the 36 rho values. The function "any(logical_array)" returns a single logical True if any one or more of values in the logical array are True. The while statement loops until all rho values are between rhomin and 1, and the iteration count hasn't exceeded maxit. The edge case of count==0 is added because "((any(rho>1)||any(rho<rhomin))&&count<=maxit)" is true when first entering.

```
while ( (any(rho>1) || any(rho<rhomin)) && count<=maxit ) || count==0
```

This creates a 36x1 logical column vector that identifies the rho values >= 1.

```
    % create K1 matrix for fixed refl at 1
    fixed_upper_logical = rho>=1;
```

The "find(logical_array)" function identifies the indices of logical_array that have a True value.

```
    fixed_upper=find(fixed_upper_logical);
```

The length function counts the number of elements in a vector.

```
    num_upper=length(fixed_upper);
```

Here is an example of the above operations, applied to a random row vector:

```
>> a = rand(1,3)
a =
    0.9572    0.4854    0.8003
>> b = a>0.5
b =
     1     0     1
>> c = find(b)
c =
     1     3
>> d = length(c)
d =
     2
```

Initialize an array of zeros with "num_upper" rows and 36 columns.

```
    K1=zeros(num_upper,36);
```

In array K1, assign 1 to num_upper of its elements. There will be one 1 on each row, and they will appear in the columns contained in fixed_upper.

```
    for i=1:num_upper
        K1(i,fixed_upper(i))=1;
    end
```

Do the same thing we just did above, but for rho values <= rhomin, creating matrix K0.

```
    % create K0 matrix for fixed refl at rhomin
    fixed_lower_logical = rho<=rhomin;
    fixed_lower=find(fixed_lower_logical);
    num_lower=length(fixed_lower);
    K0=zeros(num_lower,36);
    for i=1:num_lower
        K0(i,fixed_lower(i))=1;
    end
```

As an example, if rho values 3 and 5 are >1 and rho values 1 and 4 are < rhomin, then K1 and K0 would be:

$$K_1 = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & \dots & 0 \end{bmatrix}_{2\text{x}36}$$

$$K_0 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & \dots & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & \dots & 0 \end{bmatrix}_{3\text{x}36}$$

```
    % set up linear system
```

Create matrix K, which has (num_upper + num_lower) rows and 36 columns, by stacking K1 on top of K0, i.e.,

$$K = \begin{bmatrix} K_1 \\ K_0 \end{bmatrix}$$

```
    K=[K1;K0];
```

Next is the meat of the computation. The divide symbol / has special meaning in Matlab when applied to matrices. In Matlab, the statement "x = B/A" solves the system of linear equations x*A = B for x. So in the following statement, we are solving C*(K*B11*K') = (B11*K') for C.

In other words, C = (B11*K') * inverse(K*B11*K'), but done without actually computing the inverse of that matrix, which would be very inefficient. For more info, see https://www.mathworks.com/help/matlab/ref/mrdivide.html. The linear equation solver used by Matlab is a very robust one, called LDL decomposition. See https://en.wikipedia.org/wiki/Cholesky_decomposition#LDL_decomposition.

(BTW, K' is the matrix transpose of K.)

```
    C=B11*K'/(K*B11*K');
```

The next line performs the computation

$$\rho = R - B_{11}K^{\mathsf{T}}\left[KB_{11}K^{\mathsf{T}}\right]^{-1}\left(KR - \begin{Bmatrix} 1 \\ \rho_{min} \end{Bmatrix}\right)$$

where

$$B_{11}K^{\mathsf{T}}\left[KB_{11}K^{\mathsf{T}}\right]^{-1}$$

has been replaced with C:

```
    rho=R-C*(K*R-[ones(num_upper,1);rhomin*ones(num_lower,1)]);
```

The next two statements add stability to the algorithm by removing the floating-point noise for those rho values that are constrained at 1 or rhomin.

```
    rho(fixed_upper_logical)=1; % eliminate FP noise
    rho(fixed_lower_logical)=rhomin; % eliminate FP noise

    count=count+1;
end
```

Check to make sure we exited the loop before maxit iterations.

```
if count>=maxit
    disp(['No solution found after ',num2str(maxit),' iterations.'])
end
```

Computing B11 and B12:
======================

B11 and B12 are portions of a larger 39x39 matrix, B:

$$B = \begin{bmatrix} D & T^{\mathsf{T}} \\ T & 0 \end{bmatrix}^{-1} = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

B11 is the upper-left 36x36 part of B, and B12 is the upper-right 36x3 part of B.

D is a 36x36 tri-diagonal matrix:

$$D = \begin{bmatrix} 2 & -2 & & & & \\ -2 & 4 & -2 & & & \\ & -2 & 4 & -2 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -2 & 4 & -2 \\ & & & & -2 & 2 \end{bmatrix}$$

T is a 3x36 array converting rho to linear, D65-weighted rgb, rgb = T*rho:

$$T = M^{-1}\, A' \,\mathrm{diag}(W)/w$$

M^-1 is a 3x3 matrix that converts tristimulus values to linear rgb:

3.243063328	-1.538376194	-0.49893282
-0.968963091	1.875424508	0.041543029
0.055683923	-0.204174384	1.057994536

A′ are the 3x36 color matching functions (I used the 1931 2-degree observer), W is the D65 illuminant, and w is a weighting factor, computed as the dot product of the second row of A′ and W.