Least Squares Curve Fitting

Daniel Underwood

July 21, 2018

Linear Least Squares

Background

In linear least squares fitting, we have some set of data pairs $\{(x_1, y_1), ..., (x_n, y_n)\}$ that we want to fit to an arbitrary function of the form

$$\sum_i c_i f_i(x) \tag{1}$$

A common form of curve fitting is to fit the data to a polynomial of order m. That is,

$$P_m(x) = \sum_{k=0}^{m} c_k x^k \tag{2}$$

In this report, we will fit $\text{erf}\,(x)$, where $\text{erf}\,(x)$ is the error function:

$$\text{erf}\,(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \tag{3}$$

It should be noted that least squares fitting is typically used to fit collected data to some function rather than generating points from an already-known function and trying to fit them.

Full Polynomial Fit

First, we will fit $\text{erf}\,(x)$ with a full polynomial. That is, a polynomial with powers such that every natural number less than n is taken into account.

Plotted below are the errors of polynomial fits given by $\log |\text{erf}\,(x) - P_m(x)|$ for a polynomial of order m. Plotted in this figure are polynomials with $m = 1, ..., 10$.

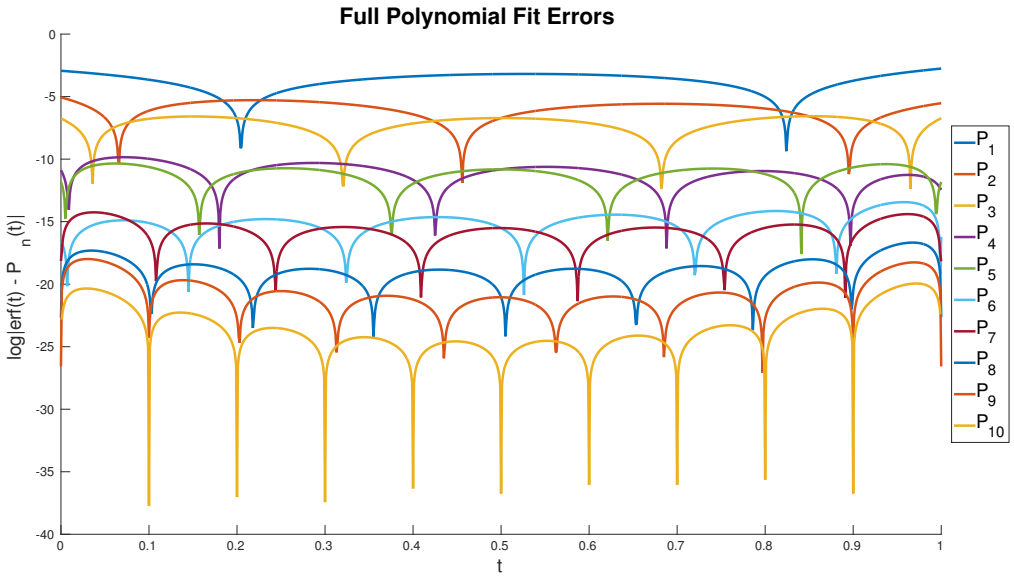

Figure 1: Full Polynomial Fit Error Plot

As can be seen in fig. 1, the error of the fits decreases as the order of the polynomial increases. This is expected as higher-order polynomials have more inflection points and can stay stable for a longer range before blowing up. It can also be noticed that all of the errors have approximate symmetry about $x = 0.5$. In addition, the polynomials seem to be grouped closely into pairs of even and odd functions with a common error at $x = 0.5$. Below is a table that tabulates the logarithm of the root mean squared error (RMSE) and the logarithm of the maximum absolute error for each degree of polynomial. Logarithms were used as the errors were too small on the interval $x \in [0, 1]$ to be useful as decimals.

	log(RMSE)	log(Max Absolute Error)
P_1	-3.4783	-2.7574
P_2	-5.7754	-5.0600
P_3	-6.9864	-6.5686
P_4	-10.7137	-9.8465
P_5	-10.9621	-10.3492
P_6	-14.5498	-13.4399
P_7	-15.2587	-14.2548
P_8	-18.0025	-16.6762
P_9	-19.2864	-17.9890
P_{10}	-21.3235	-19.9477

Table 1: Full Polynomial Fit Errors

In table 1, it is revealed that the error decreased as polynomial order increased, as seen in fig. 1. From this, we can see that the RMSE and maximum absolute errors are relatively close, which indicates that most points have similar errors. It can also be seen that for $P_{10}(x)$, the RMSE and maximum errors are approximately $e^{-36} \approx 10^{-16}$, which is the maximum precision that is typically dealt with for a data type.

Odd Polynomial Fit

In an effort to improve the fit to the error function points, other models are investigated. One thing quickly noticed is that since $\text{erf}\,(x)$ is an odd function, it makes sense to fit the data to a purely odd polynomial. We will denote a purely odd polynomial as

$$O_j(x) = \sum_{k=1}^{j} c_k x^k \tag{4}$$

Where k must be odd. For this report, we will look at $j = 1, ..., 5$, resulting in polynomial degrees of 1 to 9.

Plotted below are the absolute errors of the purely odd polynomials. Once again, a logarithmic scale is used due to the low size of the errors.

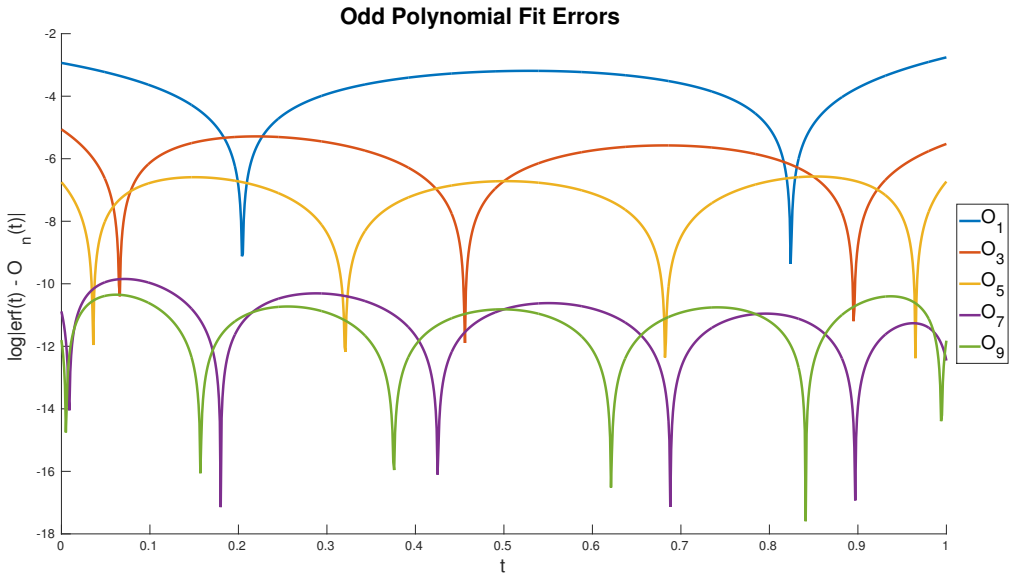

Figure 2: Odd Polynomial Fit Error Plot

It can be seen that the errors of the purely odd polynomials are unexpectedly larger than the full polynomials. While the exact cause of this is unknown, it is likely that the higher error is due to only looking at the error function evaluated on the interval $x \in [0, 1]$ rather than an interval that includes negative values where the parity of the function would matter. The purely odd function would likely be an improvement if we looked at an interval such as $x \in [-1, 1]$, although we will not examine such an interval here.

	log(RMSE)	log(Max Absolute Error)
O_1	-3.4783	-2.7574
O_3	-5.7754	-5.0600
O_5	-6.9864	-6.5686
O_7	-10.7137	-9.8465
O_9	-10.9621	-10.3492

Table 2: Odd Polynomial Fit Errors

In table 2, the values of the logarithm of the RMSE and the max absolute error are listed for the purely odd polynomial. An interesting feature revealed is that the errors are similar to the corresponding full polynomial of order $\frac{j+1}{2}$ and being slightly more erroneous at lower orders and less so at higher orders. What this likely means is that since we are not looking at an interval where the function parity matters, only the number of times the function can "twist" matters, which is related to the number of terms in the polynomial. With a full polynomial, each term allows the function to change direction and become stable around a point. With the odd polynomial, there are only half as many terms, so there are fewer points at which the function can change from increasing to decreasing or decreasing to increasing.

Exponential Fit

In search of a function that better fits the error function, it is necessary to look at some of the properties of the error function. One such property is the horizontal asymptote that the error function has. We know that $\lim_{x\to\pm\infty} \operatorname{erf}(x) = \pm 1$. This causes an issue with the models that we have been using as is is necessary for a polynomial of order greater than zero to have the property $\left|\lim_{x\to\pm\infty} P_m(x)\right| = \infty$. While the exact signs can vary based on the order of polynomial and the signs of its terms, a polynomial will most certainly never have a horizontal asymptote as $x \to \pm\infty$. To fit this property of the error function, we should choose a model that has a horizontal asymptote.

Of all elementary functions, the only nicely-behaved function that has a horizontal asymptote is the exponential decay. The exponential decay has a horizontal asymptote at 0, although that can be changed to a constant by simply adding the constant as another term in the model. The problem arises that the rest of the exponential does not have the behavior that we see in the error function. We can solve this issue by using the product of the exponential decay and another function, which can also have a horizontal asymptote as the exponential term usually has more effect than other terms. The model that was chosen for this report is the following

$$c_1 + e^{-t^2}\left(c_2 + c_3 z + c_4 z^2 + c_5 z^3\right) \tag{5}$$

where $z = (1+t)^{-1}$.

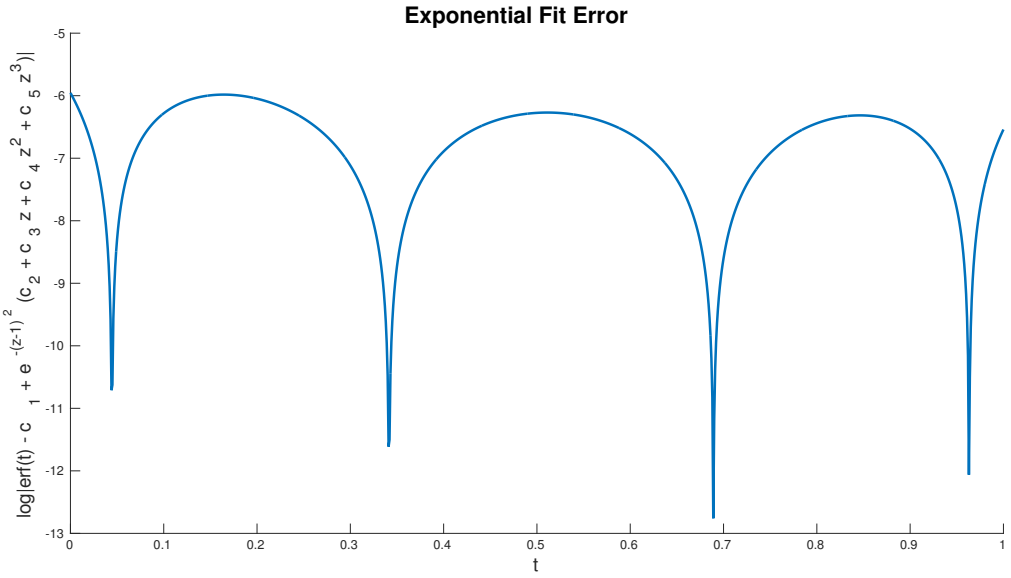

Figure 3: Exponential Fit Error Plot

Plotted in fig. 3 is the logarithm of the absolute error of the exponential fit to the error function. This is due to the asymptotic nature of the error function and the exponential; as x increases, the error will decrease, with the hope that the error quickly becomes close to 0.

In addition, with this model, we found the RMSE ≈ 0.0015 and $\log(\text{RMSE}) \approx -5.9549$. This model also resulted in a max absolute error of 0.0026 with a logarithm of -5.9549. Compared to the polynomials, these error values aren't great in the region of interest. However, the main feature of this model is the horizontal asymtote, which is important in the limits of the error function.

Additional improvements may be made by fitting a nonlinear model, resulting in parameters in places such as exponents.

Nonlinear Least Squares

Background

In order to explore nonlinear least squares, we must have a nonlinear system. One such system is involved with determining the location of a GPS transmitter by satellite locations and signal transmission times using the navigation equation

$$\sqrt{(x-x_i)^2+(y-y_i)^2+(z-z_i)^2}-c\,(t_i-d)=0 \tag{6}$$

where x_i, y_i, z_i, and t_i are respectively the x, y, and z coordinates as well as transmittance time to satellite i. x, y, and z are the spatial coordinates of the GPS transmitter and d is the time error that the transmitter has. c is the speed of light, which is approximately 299792.458 m/s.

4-Satellite GPS Problem

Solving eq. (6) requires at least 4 satellites to avoid having an infinite number of solutions. In fact, the case of four satellites results in a square system that can be reduced to a linear system. We will first solve for a system with the four satellites enumerated in the following table

i	x_i	y_i	z_i	t_i
1	15600	7540	20140	0.07074
2	18760	2750	18610	0.07220
3	17610	14630	13480	0.07690
4	19170	610	18390	0.07242

Table 3: Satellites

This system may be solved relatively simply with MATLAB's built-in fsolve function. This results in GPS transmitter values of

$$(x,y,z,d)=(-401.8,-16.8,6370.1,0)$$

This location is very nearly the north pole, and is plotted in the following figure

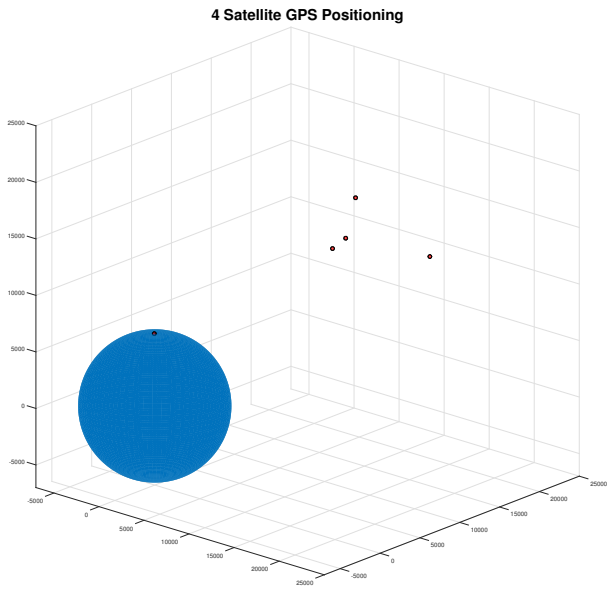

Figure 4: 4-Satellite GPS System

6-Satellite GPS Problem

The GPS problem mentioned in the preceding sections may be made more interesting by involving more satellites. By increasing the number of satellites, the problem is no longer square and may be overdetermined; therefore, it is necessary to find a least-squares solution as an exact solution may not exist.

This case may be solved by MATLAB's lsqnonlin toolbox, which yields GPS transmitter values of

$$(x, y, z, d) = (-10829, -4087, 1957, 0)$$

As seen in the following plot, this would indicate that the transmitter is on the opposite side of the planet and is a good bit in the air

Figure 5: 6-Satellite GPS System

The results shown in fig. 5 is quite interesting as one would think that increasing the number of satellites would increase the accuracy of the measurement. This is in fact not the case since a least-squares result must be used. The reason for the error is inconsistent data from the satellites. The difference in Δt between the satellites may be small, but it is large when the used with distance units $c\Delta t$. This combined with the close grouping of the satellites leads to issues with the solution of the system.

Source Code Listing

In addition to the following source code listings, the newest versions of the code may be found at `https://github.com/danielunderwood/least-squares-curve-fitting` and the latest compiled version of this report may be found at `https://www.sharelatex.com/github/repos/danielunderwood/least-squares-curve-fitting/builds/latest/output.pdf`.

Listing 1: erfFitting.m

```
function erfFitting()
%ERFFITTING Uses linlsqfit to fit error function

% Create test points
t = 0:0.1:1;

% Evaluate error function at points for fitting
yFit = erf(t);

% Evaluate error function more generally for error testing
tErrors = 0:0.001:1;
erfEval = erf(tErrors);

% Generate full polynomials order 1-10
for order=1:10
    fullPoly{order} = buildpolyfunc(order, 0, 1);
end

% Generate odd polynomials order 1-10
for order=1:2:10
    oddPoly{(order + 1) / 2} = buildpolyfunc(order, 1, 2);
end

% Generate exponential model
expModel = @(z) 1 + exp(-(z-1).^2) * 1  + exp(-(z-1).^2) .* z.^2 + ...
    exp(-(z-1).^2) .* z.^2 + exp(-(z-1).^2) .* z.^3;

% --- Full Polynomials --- %

% Create figure and axes
fullFig = figure;
fullAx = axes;
hold(fullAx);
for order=1:10
    polyFit{order} = linlsqfit(t, yFit, fullPoly{order});

    % Evaluate at error testing points
    polyTerms = splitfunction(fullPoly{order});
    evaluatedValue = 0;
    for term=1:order+1
        evaluatedValue = evaluatedValue + polyFit{order}(term) * ...
            polyTerms{term}(tErrors);
    end

    evaluatedFullPoly{order} = evaluatedValue;

    % Get errors
    fullPolyError{order} = evaluatedFullPoly{order}  - erfEval;
```

```
49
50      % Max and rms error
51      fullRMSE{order} = rms(fullPolyError{order});
52      fullMaxAbsError{order} = max(abs(fullPolyError{order}));
53
54      % Plot log of absolute error
55      plot(fullAx, tErrors, log(abs(fullPolyError{order})), ...
56          'LineWidth', 4, 'DisplayName', sprintf('P_{%d}', order));
57  end
58
59  fullLegend = legend('-DynamicLegend');
60  set(fullLegend, 'FontSize', 30);
61  set(fullLegend, 'Location', 'eastoutside');
62
63  fullTitle = title('Full Polynomial Fit Errors');
64  set(fullTitle, 'FontSize', 36);
65
66  fullXLabel = xlabel('t');
67  fullYLabel = ylabel('log|erf(t) - P_n(t)|');
68  set(fullXLabel, 'FontSize', 30);
69  set(fullYLabel, 'FontSize', 30);
70
71  % Stop holding axes
72  hold(fullAx);
73
74  % --- Odd Polynomials --- %
75
76  % Create figure and axes
77  oddFig = figure;
78  oddAx = axes;
79  hold(oddAx);
80  for order=1:2:10
81      oddFit{(order + 1) / 2} = linlsqfit(t, yFit, oddPoly{(order + 1) / 2});
82
83      % Evaluate at error testing points
84      oddTerms = splitfunction(oddPoly{(order + 1) / 2});
85      evaluatedValue = 0;
86      for term=1:((order + 1) / 2)
87          evaluatedValue = evaluatedValue + ...
88              oddFit{(order + 1) / 2}(term) * oddTerms{term}(tErrors);
89      end
90
91      evaluatedOddPoly{(order + 1) / 2} = evaluatedValue;
92
93      % Get errors
94      oddPolyError{(order + 1) / 2} = ...
95          evaluatedFullPoly{( order + 1) / 2} - erfEval;
96
97      % Max and rms error
98      oddRMSE{(order + 1) / 2} = rms(oddPolyError{(order + 1) / 2});
99      oddMaxAbsError{(order + 1) / 2} = max(abs(oddPolyError{(order + 1) /
            2}));
100
101     % Plot log of absolute error
102     plot(oddAx, tErrors, log(abs(oddPolyError{(order + 1) / 2})), ...
103         'LineWidth', 4, 'DisplayName', sprintf('O_{%d}', order));
```

```
104 end
105
106 oddLegend = legend('-DynamicLegend');
107 set(oddLegend, 'FontSize', 30);
108 set(oddLegend, 'Location', 'eastoutside');
109
110 oddTitle = title('Odd Polynomial Fit Errors');
111 set(oddTitle, 'FontSize', 36);
112
113 oddXLabel = xlabel('t');
114 oddYLabel = ylabel('log|erf(t) - O_n(t)|');
115 set(oddXLabel, 'FontSize', 30);
116 set(oddYLabel, 'FontSize', 30);
117
118 % Toggle hold
119 hold(oddAx);
120
121 % --- Exponential Model --- %
122 expFig = figure;
123 expAx = axes;
124 hold(expAx);
125
126 expFit = linlsqfit(t, yFit, expModel);
127
128 % Evaluate at error testing points
129 expTerms = splitfunction(expModel);
130 evaluatedValue = 0;
131 for term=1:5
132     evaluatedValue = evaluatedValue + ...
133         expFit(term) * expTerms{term}(tErrors);
134 end
135
136 evaluatedExpModel = evaluatedValue;
137
138 % Get error
139 expError = evaluatedExpModel - erfEval;
140
141 expRMSE = rms(expError);
142 expMaxAbsError = max(abs(expError));
143
144 plot(expAx, tErrors, log(abs(expError)), 'LineWidth', 4);
145
146 expTitle = title('Exponential Fit Error');
147 set(expTitle, 'FontSize', 36);
148
149 expXLabel = xlabel('t');
150 expYLabel = ylabel(['log|erf(t) - c_1 + e^{-(z-1)^2} ' ...
151     '(c_2 + c_3 z + c_4 z^2 + c_5 z^3)|']);
152 set(expXLabel, 'FontSize', 30);
153 set(expYLabel, 'FontSize', 30);
154
155 % Toggle hold
156 hold(expAx);
157
158
159 end
```

Listing 2: linlsqfit.m

```
function [ fitParams ] = linlsqfit(x, y, modelfun)
%linlsqfit Fits data to a model function with linear parameters

% Make x and y column vectors
x = x(:);
y = y(:);

% Split modelfun into terms
funcTerms = splitfunction(modelfun);

% Get number of terms and points
numTerms = numel(funcTerms);
numPoints = numel(x);

% Preallocate A
A = zeros(numPoints, numTerms);

% Generate matrix A
for m=1:numPoints
    for n=1:numTerms
        A(m,n) = funcTerms{n}(x(m));
    end
end

% Solve for fit parameters
fitParams = A \ y;
end
```

Listing 3: splitfunction.m

```
function funcArray = splitfunction(funcHandle)
%splitfunction Splits function into cell array
%   Splits function into cell array based on terms.
%   i.e., 1 + x + x^2 -> {1, x, x^2}

F = functions(funcHandle);

% Get string from function handle
str = F.function;

% pref is the '@(...)' part. Split by regex
[pref, body] = regexp(str, '@\(.+?\)', 'match', 'split');

% body is the main part
body = body{2};

% Indices for splitting
ind = cumsum((body=='(')-(body==')'))==0 & body=='+';

% '?' will be used as split marker
body(ind) = '?';

% split, and then add prefix
funcStrArray = strcat(pref, strsplit(body, '?'));

```

```
% Convert strings to functions
funcArray = cellfun(@str2func, funcStrArray, 'uniformoutput', 0);

end
```

Listing 4: navigationequation4.m

```
function F =navigationequation4(x)
%distanceequation 4-satellite Navigation Equation for GPS Problem

% Hardcode Satellite Values
xi = [ 15600 18760 17610 19170 ];
yi = [ 7540 2750 14630 610 ];
zi = [ 20140 18610 13480 18390 ];
ti = [ 0.07074 0.07220 0.07690 0.07242 ];
c = 299792.458;

% Distance Equation
F = [ (x(1) - xi(1)).^2 + (x(2) - yi(1)).^2 + (x(3) - zi(1)).^2 - (c*(ti(1)
    - x(4))).^2;
    (x(1) - xi(2)).^2 + (x(2) - yi(2)).^2 + (x(3) - zi(2)).^2 - (c*(ti(2) -
        x(4))).^2;
    (x(1) - xi(3)).^2 + (x(2) - yi(3)).^2 + (x(3) - zi(3)).^2 - (c*(ti(3) -
        x(4))).^2;
    (x(1) - xi(4)).^2 + (x(2) - yi(4)).^2 + (x(3) - zi(4)).^2 - (c*(ti(4) -
        x(4))).^2 ];
end
```

Listing 5: navigationequation6.m

```
function F =navigationequation6(x)
%distanceequation 6-satellite Navigation Equation for GPS Problem

% Hardcode Satellite Values
xi = [ 15600 18760 17610 19170 17800 17500 ];
yi = [ 7540 2750 14630 610 6400 7590 ];
zi = [ 20140 18610 13480 18390 18660 18490 ];
ti = [ 0.07074 0.07220 0.07690 0.07242 0.0732 0.0735 ];
c = 299792.458;

% Distance Equation
F = [ (x(1) - xi(1)).^2 + (x(2) - yi(1)).^2 + (x(3) - zi(1)).^2 - (c*(ti(1)
    - x(4))).^2;
    (x(1) - xi(2)).^2 + (x(2) - yi(2)).^2 + (x(3) - zi(2)).^2 - (c*(ti(2) -
        x(4))).^2;
    (x(1) - xi(3)).^2 + (x(2) - yi(3)).^2 + (x(3) - zi(3)).^2 - (c*(ti(3) -
        x(4))).^2;
    (x(1) - xi(4)).^2 + (x(2) - yi(4)).^2 + (x(3) - zi(4)).^2 - (c*(ti(4) -
        x(4))).^2
    (x(1) - xi(5)).^2 + (x(2) - yi(5)).^2 + (x(3) - zi(5)).^2 - (c*(ti(5) -
        x(4))).^2;
    (x(1) - xi(6)).^2 + (x(2) - yi(6)).^2 + (x(3) - zi(6)).^2 - (c*(ti(6) -
        x(4))).^2; ];
end
```