Systems of Equations with Matrices; Data Fitting

Mitchell Miller

Physics 240

This assignment explores the applications and methods matrix manipulation. Matrices are used to solve complex groups of equations called systems. Using a package called LAPACK, very complex matrix manipulations can be performed with relative ease. In the course of this assignment, several applications of these manipulations were explored. First, very simple tasks commonly used in advanced calculations were explored. Next, the applications in data fitting were examined. The methods of Lagrange Interpolation and Spline fitting were applied to a data set in order to generate intermediate points. Next, the linear least squared method was applied to both exponential decay data and heat flow data. These data sets both effectively displayed the properties of this method.

1. Introduction

Matrix manipulations are a vital part of most complex problems in physics. The ability to effectively apply them is a vital tool for developing physicists. Using matrices, one can solve all sorts of problems; however, designing code for even the simplest methods can be extremely difficult. Fortunately, free code is readily available from professional computer scientists. Using packages like LAPACK and BLAS, one can implement relatively simple code to perform very complicated manipulations and solve very complex problems. One application of these packages is in solving systems of equations. Systems of equations are dependent equations that must be all solved simultaneously in order to reach a solution. These are commonly seen in kinematics and quantum problems.

This assignment also explores several methods of data fitting. Data fitting is incredibly important for modern experiments. Fitting allows one to predict expected values between measured data points. By generating a function that fits the measured points, one can calculate the value of the generated function at any point, allowing one to take advantage of nearly infinite data points.

2. Theory

Converting a linear system of equations to a matrix is a simple process. A matrix of coefficients is first created and multiplied by a variable vector of dimensions 1 x N, where N is the number of equations. This product is set equal to a vector of equal dimensions as the variable vector that is filled with the answers of each equation. A general system:

$$\begin{pmatrix} a_{11}x_1 + & \cdots & +a_{1N}x_N = b_1 \\ \vdots & \ddots & \vdots \\ a_{N1}x_1 + & \cdots & +a_{NN}x_N = b_N \end{pmatrix} \quad (1)$$

Is converted to the following matrix

$$\begin{pmatrix} a_{11} & \cdots & a_{1N} \\ \vdots & \ddots & \vdots \\ a_{N1} & \cdots & a_{NN} \end{pmatrix} * \begin{pmatrix} x_1 \\ \vdots \\ x_N \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_N \end{pmatrix} \quad (2)$$

In order to solve this system, values for x_1,x-$_2$,…,x_N that satisfy all equations must be found. This is relatively simple as long as N is very small, but as N gets large, completing a system by hand becomes nearly impossible. A non-matrix equation **A*x = b**, is a very simple equation to solve, because one can simply divide by **A**. One would assume then, that, in order to solve the same equation with matrices, one would simply invert all the terms in A and multiply it to both sides. This simple manipulation cannot be done with matrices, however. To invert a matrix, one must perform complicated process called ‘LU Decomposition.’ This method involves converting a matrix into a product of a lower and upper triangular matrix[1]. LAPACK can be used to both solve directly a system of equations in the form of equation (2) and to solve for the inverse of a matrix.

Another simple matrix concept explored in this assignment is the eigenvalues and eigenvector. Eigenvalues are unique sets of scalar multipliers that are characteristic of any given matrix[2]. Eigenvalues are used to describe matrices and follow the form:

$$AX = \lambda X \tag{3}$$

Where λ is the scalar eigenvalues of matrix A[3]. From this equation, the eigenvector X can also be found. The eigenvector is also characteristic of the matrix A and also used in many problems in physics. Both the eigenvalues and eigenvectors of a matrix can be found easily using LAPACK. These values often have imaginary parts, which is important to consider when implementing the functions in LAPACK and when trying to calculate them by hand.

The other focus of this assignment is data interpolation and fitting. Data fitting involves generating a function that matches a set of experimentally collected data points. This function is then used to predict or interpolate the values at points that were not measured. This is useful in areas such as X-ray diffraction and florescence where it is difficult to measure data at every possible energy value. The most common and simplest method of interpolation makes use of Lagrange polynomials. Lagrange interpolation fits data points with a Lagrange polynomial of the lowest degree possible[4]. A function, g(x), can be found using the following equations[5]:

$$g(x) \cong g_1\lambda_1(x) + g_2\lambda_2(x) + \cdots + g_N\lambda_N(x) \tag{4}$$

$$\lambda_i(x) = \prod_{J(\neq i)=1}^{N} \frac{x - x_j}{x_i - x_j} \tag{5}$$

Although this method is very easy to implement, it has a few drawbacks. First, the method provides no estimate of error. Additionally, it can frequently generate a function that, while it hits every provided point, represents nothing in between the data points, creating peaks and valleys where a smooth curve should be. This is usually due to the high degree of polynomial required to match every point. It is much more accurate when implemented in small sets of points along the data set. By generating individual polynomials for small sets of data, the curvature is kept much closer to the expect value and preventing unnecessary peaks or valleys in the function. Finally, this method is also very bad for extrapolating data because the values for points outside the data range are completely arbitrary, since this method only matches to the values provided.

The next method of data fitting is cubic spline interpolation. A cubic spline is a set of third order polynomials assembled to fit a set of control points[6]. For purposes in science, those control points are experimentally collected data. Spline interpolation requires the second derivative of the function at the specified endpoints to always be zero. This adds extra boundary conditions, which leaves fewer unknowns during calculation.

The final method of data fitting explored in the assignment is the least squares method. This method minimizes the square of the error in the data fit. By doing this, the estimation is very accurate. This is implemented with the following equation[7]:

$$\chi^2 = \sum_{i=1}^{N} (\frac{y_i - g(x)}{\sigma_i})^2 \tag{6}$$

This sum is computed over all the experimental points and the total sum is minimized by modifying g(x), until the fit is at an acceptable curve. This method is seen frequently in data fitting due to its simplicity and accuracy. Code to implement this method can be written very quickly and still provide an excellent fit to the data. It is more common, though, to see it used for linear data.

3. Experimental Method

The first section of this assignment explores simple matrix manipulations with LAPACK. The first task was to invert a 3x3 matrix. Next, a system of equations in the form of equation (2) was solved. Next, the eigenvalues and eigenvectors of another 3x3 matrix were computed. Finally, a special kind of matrix called the Hilbert matrix was solved. This matrix has the following form[8]:

$$A_{ij} = \frac{1}{i+j-1}, B_i = \frac{1}{i} \tag{7}$$

$$H = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \end{bmatrix}.$$

The Hilbert matrix solved in this assignment was a 100x100 matrix.

Next, a set of data from a simulated scattering experiment was fit using the Lagrange method. First the data was fit using a single Lagrange Polynomial. The data was then fit again using local fitting with a third order polynomial.

Finally, the linear least squares method was implemented in two different data sets. The first set was the data from an exponential decay experiment. By taking the natural log of the decay rate, a linear data set was obtained and fitted. The lifetime of a π meson was then calculated from the slope of this fit. Next, a data set from a heat flow experiment was fitted and the variances were found. Both these data sets were plotted and the linear fit was compared to the experimental data.

4. Data and Analysis

The first section of this lab involves testing a few simple matrix manipulations included in LAPACK. The first was to compute the inverse of the matrix:

$$A = \begin{pmatrix} 4 & -1 & 2 \\ 3 & 6 & -4 \\ 2 & 1 & 8 \end{pmatrix}$$

$$A^{-1} = \begin{pmatrix} 0.1997 & 0.0646 & 0.0076 \\ -0.1217 & 0.1141 & 0.0722 \\ -0.0342 & -0.0304 & 0.1141 \end{pmatrix}$$

Next, the solution, X, to a system of equations using A as the coefficient matrix was calculated with the following B matrices:

$$B_1 = \begin{pmatrix} 12 \\ -25 \\ 32 \end{pmatrix}, B_2 = \begin{pmatrix} 4 \\ -10 \\ 22 \end{pmatrix}, B_3 = \begin{pmatrix} 20 \\ -30 \\ 40 \end{pmatrix}$$

$$X_1 = \begin{pmatrix} 1 \\ -2 \\ 4 \end{pmatrix}, X_2 = \begin{pmatrix} 0.312 \\ -0.038 \\ 2.677 \end{pmatrix}, X_3 = \begin{pmatrix} 2.319 \\ -2.965 \\ 4.790 \end{pmatrix}$$

These are the same as the expect values predicted in the book. Next, the eigenvalues and eigenvectors of a different matrix were calculated. This matrix had eigenvalues of λ_1=5, λ_2=λ_3=3. Since two of the eigenvalues are a double root, the eigenvectors found for those two eigenvalues were degenerate. This means that they are linear combinations of a vector. The function in LAPACK successfully calculated eigenvectors for these, but the accuracy is unknown.

Next, a set of data from a scattering experiment provided in the textbook was fit using Lagrange polynomials. First, the entire energy spectrum was fit using a single, eight-degree polynomial. The results of this fit are displayed in Figure 2. This is clearly not an ideal fit. Although the polynomial passes through every data point, shown in Figure 1, there are unnecessary oscillations. It is unlikely that there are peaks in the actual data where the Lagrange polynomial predicts there to be. Figure 3 shows the results of a local interpolation using third order Lagrange polynomials. The order of the polynomial is reduced combining several polynomials that are generated by only considering a point and one point on either side of it. By combining several third order polynomials, a much smoother function is formed that is significantly more accurate than the eight-order polynomial shown in Figure 2.

Next, the exponential decay and heat flow data provided in the textbook were both fit. Figure 4 shows the plot of the log of the decay rate versus the time. Because no error was provided with the data, an experimental error of 5% was assumed when calculating the linear regression. Since the decay is exponential, the log of the decay rate is linear. From the slope of the linear regression, the lifetime of the meson can be calculated. From the data provided, the lifetime was found to be 2.73E-8 s. After comparing this to the accepted lifetime of 2.6E-8 s, an error of 4.87% was found. This is a very small error that is most likely due to the randomness of the decay process. One should also note that this error is less than the guessed experimental error used during the linear regression calculation. This error could be reduced by measuring the decay of a larger sample. The equation for the linear regression is the following:

$$y = -8 * 10^7 x + 23.305$$

The data from the heat flow experiment provided in the textbook is shown in Figure 5. Once again, this is clearly linear and the linear regression is plotted with the data. The equation for the regression is

$$y = 10.0733x + 0.889$$

As with the decay data, since no error was provided, 5% was assumed. Using this, the variances for the slope equation parts were calculated to be:

$$\sigma_{0.889} = 1.01 * 10^{-4}, \sigma_{10.07} = 1.52 * 10^{-5}$$

These are both very small values, indicating a very high degree of certainty in the calculated values. This indicates that the values calculated are certain to many more significant figures than the original data had.

5. Conclusion

This lab successfully demonstrated the applications of matrix manipulations and data fitting. First, simple matrix manipulations were explored. Next, various experimental data sets were fit using the Lagrange Polynomials and linear least squares methods.

6. Bibliography

[1]"LU Decomposition - Wikipedia, the Free Encyclopedia." *Main Page - Wikipedia, the Free Encyclopedia*. Web. 16 Apr. 2010. <http://en.wikipedia.org/wiki/LU_decomposition>.

[2]"Eigenvalue, Eigenvector and Eigenspace - Wikipedia, the Free Encyclopedia." *Main Page - Wikipedia, the Free Encyclopedia*. Web. 16 Apr. 2010. <http://en.wikipedia.org/wiki/Eigenvalue,_eigenvector_and_eigenspace>.

[3]"Eigenvalue -- from Wolfram MathWorld." *Wolfram MathWorld: The Web's Most Extensive Mathematics Resource*. Web. 16 Apr. 2010. <http://mathworld.wolfram.com/Eigenvalue.html>.

[4]"Lagrange Polynomial - Wikipedia, the Free Encyclopedia." *Main Page - Wikipedia, the Free Encyclopedia*. Web. 16 Apr. 2010. <http://en.wikipedia.org/wiki/Lagrange_interpolation>.

[5]"Lagrange Interpolating Polynomial -- from Wolfram MathWorld." *Wolfram MathWorld: The Web's Most Extensive Mathematics Resource*. Web. 16 Apr. 2010. <http://mathworld.wolfram.com/LagrangeInterpolatingPolynomial.html>.

[6]"Cubic Spline -- from Wolfram MathWorld." *Wolfram MathWorld: The Web's Most Extensive Mathematics Resource*. Web. 16 Apr. 2010. <http://mathworld.wolfram.com/CubicSpline.html>.

[7]"Least Squares - Wikipedia, the Free Encyclopedia." *Main Page - Wikipedia, the Free Encyclopedia*. Web. 16 Apr. 2010. <http://en.wikipedia.org/wiki/Least_squares>.

[8]"Hilbert Matrix - Wikipedia, the Free Encyclopedia." *Main Page - Wikipedia, the Free Encyclopedia*. Web. 16 Apr. 2010. <http://en.wikipedia.org/wiki/Hilbert_matrix>.

APPENDIX A: TABLES AND FIGURES

Figure 1

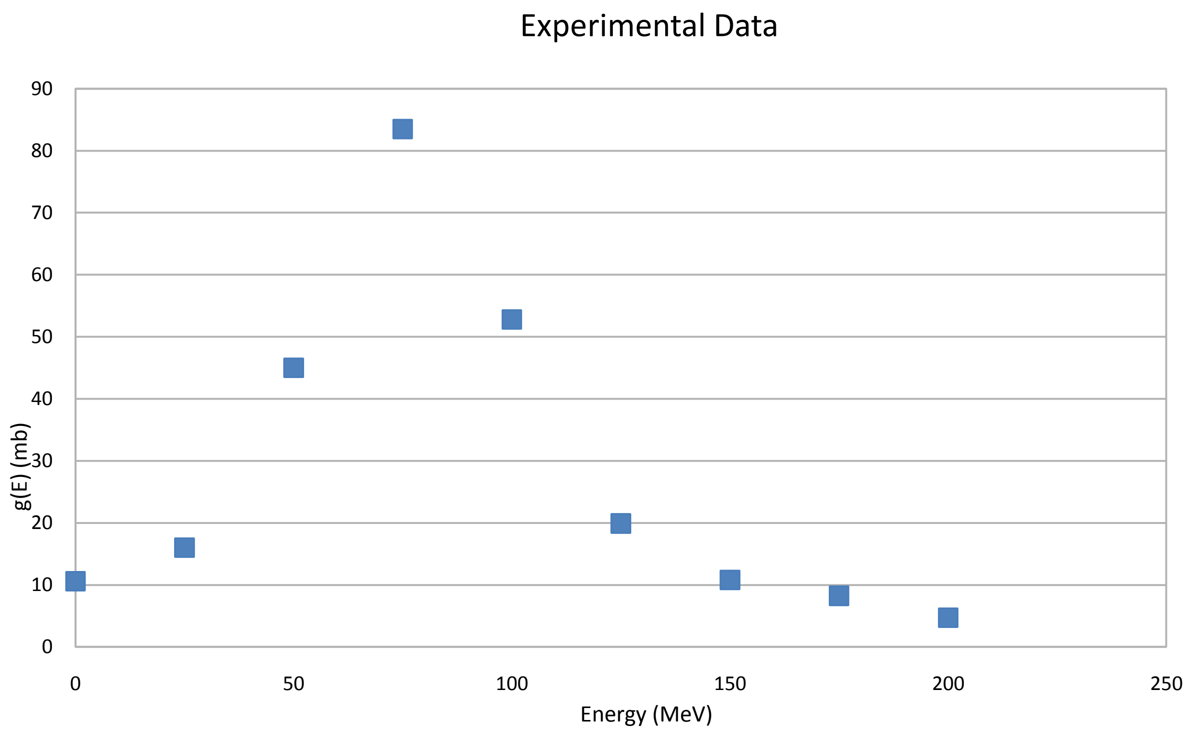

Figure 1 – This figure shows the plot of the cross-sectional scattering data provided in the textbook. The function g(E) represents the amount of energy scattered.

Figure 2

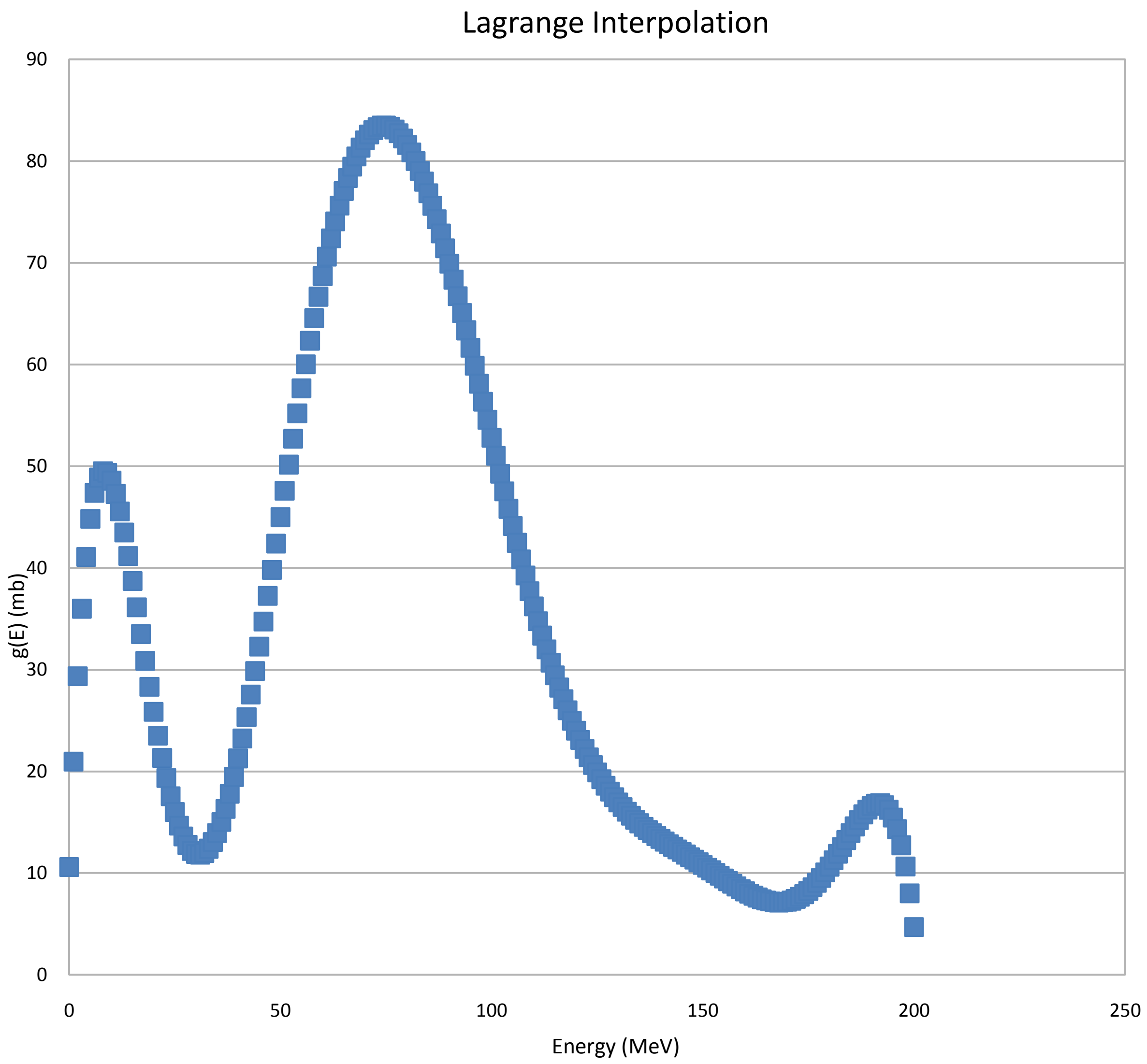

Figure 2 – This figure shows the Lagrange polynomial fit to the data in Figure 1 using only one polynomial. This figure shows the large oscillations that can occur when using Lagrange polynomials.

Figure 3

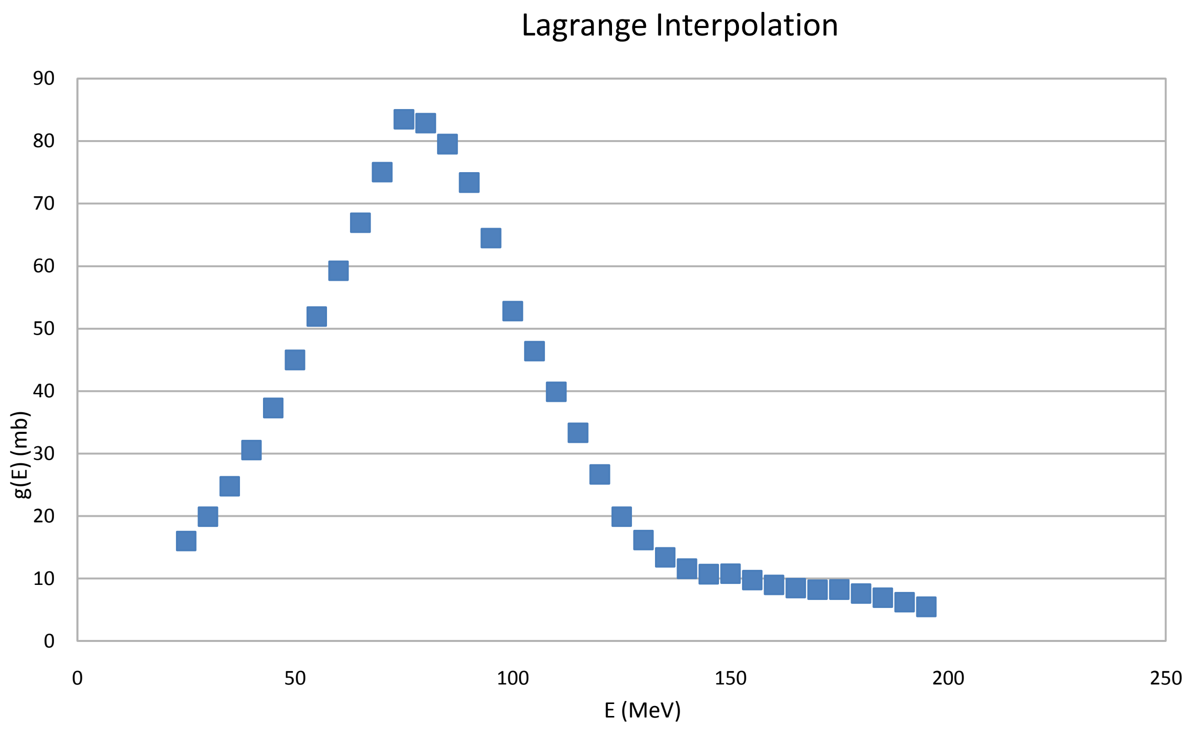

Figure 3 – This figure displays the fit of the data in Figure 1 using local Lagrange interpolation. By fitting points based on the two points on either side of it, the oscillations are eliminated.

Figure 4

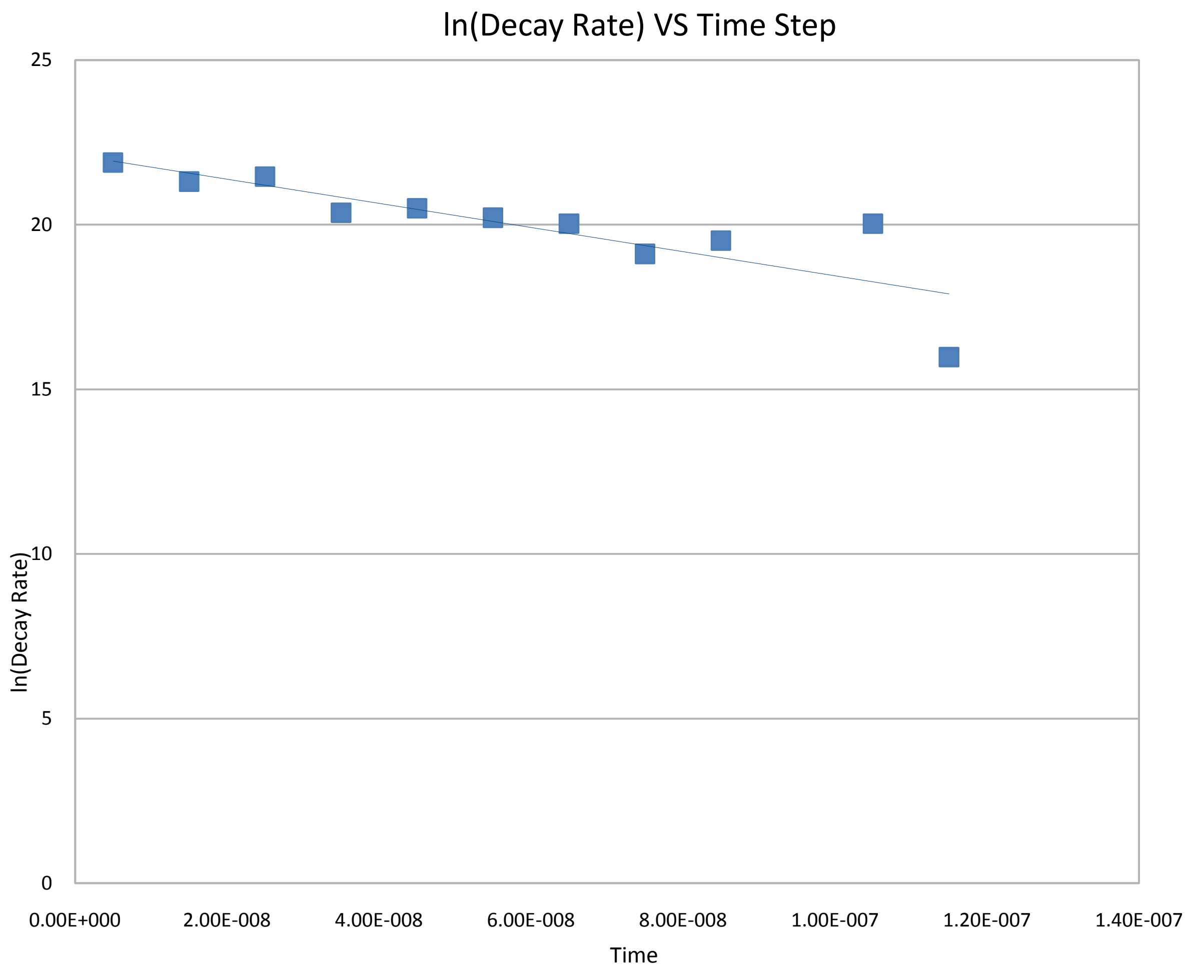

Figure 4 – This figure displays the plot and linear regression of the natural log of decay rate versus time. This data was gathered from Figure 8.4 in the textbook.

Figure 5

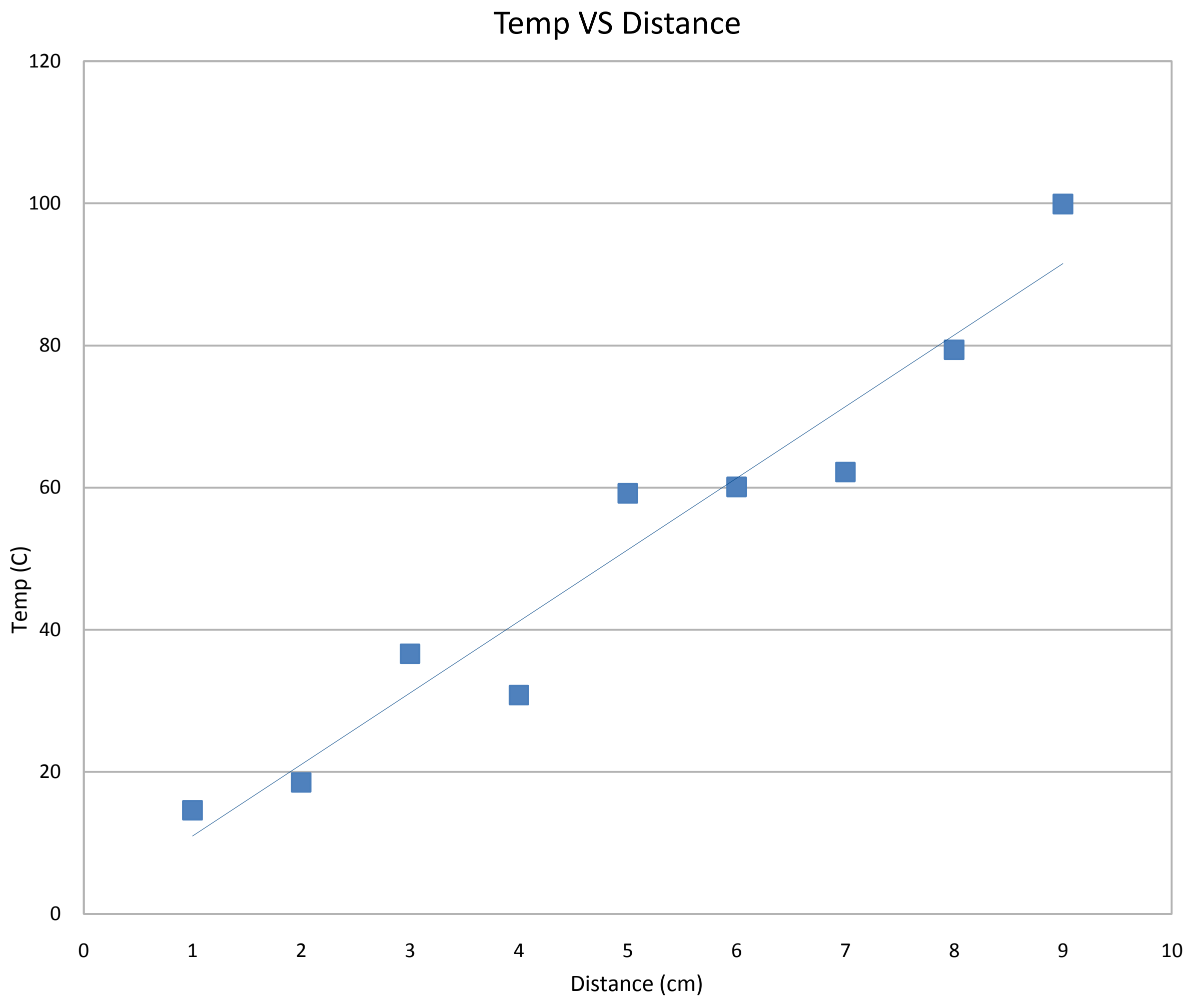

Figure 5 – This plot shows the temperature in °C versus distance in a heat flow experiment. The linear regression is also plotted.

APPENDIX B: SOURCE CODE

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "/usr/include/clapack.h"
#include <cblas.h>

//Mitchell Miller
//Assignment 6
//PHYS 240 4/16/10

//http://www.netlib.org/lapack/double/


static long
calldgetrf(long M, long N, double *matrixA, long lda, long *matrixPivot){

  long info;

  dgetrf_(&M, &N, matrixA, &lda, matrixPivot, &info);
  return info;

}

static long
calldgetri(long N, double *matrixA, long lda, long *matrixPivot, double *matrixWork, long
lwork){

        long info;

        dgetri_(&N, matrixA, &lda, matrixPivot, matrixWork, &lwork, &info);
        return info;

}

static long
calldgesv(long N, long NRHS, double *matrixA, long lda, long *matrixPivot, double *matrixB,
long ldb){

        long info;

        dgesv_(&N, &NRHS, matrixA, &lda, matrixPivot, matrixB, &ldb, &info);
        return info;
```

```
}

static long
calldgeev(char jobL, char jobR, long N, double *matrixA, long lda, double *eigenvalueRealArray,
double *eigenvalueImaginaryArray, double *eigenvectorL, long ldvl, double *eigenvectorR, long
ldvr, double *matrixWork, long lwork){

        long info;

        dgeev_(&jobL,&jobR,&N,matrixA,&lda,eigenvalueRealArray,eigenvalueImaginaryArray,e
igenvectorL,&ldvl,eigenvectorR,&ldvr,matrixWork,&lwork,info);
        return info;

}

int main(){

        //Initialize Variables
        double matrixATranspose[] = { 4, 3, 2,
                                -2, 6, 1,
                                1, -4, 8};
        double matrixAInverse[] = { 4,-2, 1,
                                3, 6,-4,
                                2, 1, 8};
        double matrixAEigen[] = {-2,2,-3,
                                    2,1,-6,
                                   -1,-2,0};
        double eigenvalueRealArray[3];
        double eigenvalueImaginaryArray[3];
        double eigenvectorArrayL[9];
        double eigenvectorArrayR[9];
        long matrixPivot[9];
        int counter1,counter2;
        int three=3,info=0,stupid=12;
        double matrixWork[3];

        //Calculate inverse of A
        info = calldgetrf(3,3,matrixAInverse,3,matrixPivot);
        printf("INFO: %d\n", info);
        info = calldgetri(3,matrixAInverse,3,matrixPivot,matrixWork,3);
        printf("INFO: %d\n", info);

        //Prinf inverse of A
```

```
	printf("****A INVERSE SOLUTION****\n");
	for(counter1=0;counter1<3;counter1++){

		for(counter2=0;counter2<3;counter2++)
printf("%lf\t",matrixAInverse[3*counter1+counter2]);
		printf("\n");

	}

	//Initialize Matricies for part 2
	double matrixB[] = {12.,-25.,32.};
	//double matrixB[] = {4,-10,22};
	//double matrixB[] = {20,-30,40};

	//Calculate solutions for A*x = B
	info = calldgesv(3,1,matrixATranspose,3,matrixPivot,matrixB,3);

	//Print solutions
	printf("****X MATRIX SOLUTION****\n");
	printf("INFO: %d\n", info);
	for(counter1=0;counter1<3;counter1++)	printf("%lf\n", matrixB[counter1]);

	//Calculate eignevalues and eigenvectors
	calldgeev('V','V',3,matrixAEigen,3,eigenvalueRealArray,eigenvalueImaginaryArray,eigenv
ectorArrayL,3,eigenvectorArrayR,3,matrixWork,12);

	//Print soluionts
	printf("****EIGENVALUE SOLUTIONS****\n");
	printf("INFO: %d\n", info);

	for(counter1=0;counter1<3;counter1++)	printf("%lf %lf\n",
eigenvalueRealArray[counter1],eigenvalueImaginaryArray[counter1]);

	for(counter2=0;counter2<3;counter2++){

		printf("****EIGENVECTOR %d SOLUTIONS****\n",counter2+1);
		for(counter1=0;counter1<3;counter1++){

			printf("%lf
%lf\n",eigenvectorArrayL[counter1+3*counter2],eigenvectorArrayR[counter1+3*counter2]);

		}

	}
	//Initialize Hilbert matrix
	double matrixHilbert[10000];
	double matrixBHilbert[100];
	for(counter1=1;counter1<101;counter1++){

		for(counter2=1;counter2<101;counter2++){

			matrixHilbert[100*(counter1-1)+(counter2-1)] =
1/((double)counter1+(double)counter2-1.);
			//printf("Position = %d, %lf\n",100*(counter1-
1)+counter2,matrixHilbert[100*(counter1-1)+counter2]);

		}
		matrixBHilbert[counter1-1] = (1/(double)counter1);
		//printf("%lf\n",matrixBHilbert[counter1-1]);

	}

	//Compute solution for Hilbert Matrix
	info = calldgesv(100,1,matrixHilbert,100,matrixPivot,matrixBHilbert,100);
	printf("****HILBERT MATRIX SOLUTION****\n");
	printf("INFO: %d\n",info);
	for(counter1=0;counter1<100;counter1++)
printf("%d\t%lf\n",counter1,matrixBHilbert[counter1]);
}


#include <stdio.h>
#include <stdlib.h>
#include <math.h>

//Mitchell Miller
//Assignment 6
//PHYS 240 4/16/10

double lagrangeInterpolation(int numberPoints, double *positionMatrix, double *valueMatrix,
double interpolationPoint){

	int counter1,counter2;
	double product,sum=0;

	for(counter1=0;counter1<numberPoints;counter1++){
		product = 1;
		for(counter2=0;counter2<numberPoints;counter2++){

			if(counter1!=counter2){

				//Calculate lambda
				product = product * ((interpolationPoint - 
positionMatrix[3*counter2]) / (positionMatrix[3*counter1] - positionMatrix[3*counter2]));

			}

		}

		//Calcualte g(x)
		sum += product*valueMatrix[counter1];

	}

	printf("g(x) = %lf\n",sum);
	return sum;

}

int main(){

	int numberPoints = 9;
	int counter1;
	double positionMatrix[9] = {0.,25.,50.,75.,100.,125.,150.,175.,200.};
	double valueMatrix[9] = {10.6,16.0,45.0,83.5,52.8,19.9,10.8,8.25,4.7};
	double pointValue;
	FILE *outfile;
	outfile = fopen("8.5.2.txt","w");

	//Calculate interpolation at points between 0 and 200
	for(counter1=0;counter1<=200;counter1++){

		pointValue = 
lagrangeInterpolation(numberPoints,positionMatrix,valueMatrix,(double)counter1);
		fprintf(outfile,"%d\t%lf\n",counter1,pointValue);

	}
	fclose(outfile);

}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

//Mitchell Miller
//Assignment 6
//PHYS 240 4/16/10

double lagrangeInterpolation(int numberPoints, double *positionMatrix, double *valueMatrix,
double interpolationPoint){

	int counter1,counter2;
	double product,sum=0;
	int shift = (int)interpolationPoint/25 - 1;

	for(counter1=0;counter1<numberPoints;counter1++){

		product = 1;
		for(counter2=0;counter2<numberPoints;counter2++){

			if(counter1!=counter2){

				//Calculate lambda
				product = product * ((interpolationPoint -
positionMatrix[counter2+shift]) / (positionMatrix[counter1+shift] -
positionMatrix[counter2+shift]));

			}

		}

		//Calcualte g(x)
		sum += product*valueMatrix[counter1+shift];

	}

	printf("g(x) = %lf, shift = %d\n",sum,shift);
	return sum;

}

int main(){

	int numberPoints = 3;
	int counter1;
	double positionMatrix[9] = {0.,25.,50.,75.,100.,125.,150.,175.,200.};
	double valueMatrix[9] = {10.6,16.0,45.0,83.5,52.8,19.9,10.8,8.25,4.7};
	double pointValue;
	FILE *outfile;
	outfile = fopen("8.5.2.2.txt","w");

	//Calculate interpolation at points between 25 and 200
	for(counter1=25;counter1<200;counter1+=5){

		pointValue =
lagrangeInterpolation(numberPoints,positionMatrix,valueMatrix,(double)counter1);
		fprintf(outfile,"%d\t%lf\n",counter1,pointValue);

	}
	fclose(outfile);

}

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

//Mitchell Miller
//Assignment 6
//PHYS 240 4/16/10

int main(){

	//Intialize variables
	//double xMatrix[11] = {15, 25, 35, 45, 55, 65, 75, 85, 105, 115}; //T **EXPONENTIAL
DECAY 8.7.2
	double xMatrix[9] = {1,2,3,4,5,6,7,8,9};    //X **HEAT FLOW 8.7.3
	//double yMatrix[11] = {3.2,-1.8,2.1,-0.7,0.8,0.6,0.5,0.2,0.3,0.5,0.1};		//dN/dt
	**EXPONENTIAL DECAY 8.7.2
	//double yMatrix2[11] = {3.47,2.89,3.04,1.95,2.08,1.79,1.61,0.69,1.1,1.61,0};
	//ln(dN/dt) **EXPONENTIAL DECAY 8.7.2
	double yMatrix2[9] = {14.6,18.5,36.6,30.8,59.2,60.1,62.2,79.4,99.9};		//T_i **HEAT
FLOW 8.7.3
	double errorMatrix[11] = {.05,.05,.05,.05,.05,.05,.05,.05,.05,.05,.05};
	double slope, yIntercept, xAvg=0, yAvg=0, sXX=0, sXY=0;
	double varianceA,varianceB,s=0,sX=0,delta;
	//int nPoints=11;			//**EXPONENTIAL DECAY 8.7.2
	int nPoints=9;		//**HEAT FLOW 8.7.3
	int counter;

	//Calculate Averages
	for(counter=0;counter<nPoints;counter++)	xAvg +=
xMatrix[counter]*(1/(double)nPoints);
	for(counter=0;counter<nPoints;counter++)	yAvg +=
yMatrix2[counter]*(1/(double)nPoints);

	//Calculate sXX and sXY
	for(counter=0;counter<nPoints;counter++){

		sXX += pow((xMatrix[counter]-xAvg),2) / pow(errorMatrix[counter],2);
		sXY += (xMatrix[counter]-xAvg)*(yMatrix2[counter]-yAvg) /
pow(errorMatrix[counter],2);

	}

	slope = sXY/sXX;
	yIntercept = yAvg - slope*xAvg;
	printf("Y = %lfx + %lf\n",slope,yIntercept);

	//Calculate variance
	for(counter=0;counter<nPoints;counter++){

		s += 1 / pow(errorMatrix[counter],2);
		sX += xMatrix[counter] / pow(errorMatrix[counter],2);

	}

	delta = s*sXX-pow(sX,2);
	varianceA = sXX / delta;
	varianceB = s / delta;
	printf("Variance A = %e, Variance B = %e\n",varianceA,varianceB);
  getchar();

}
```