Analysis of a Spring-Mass System at Equilibrium

Scott Myers
April 13th, 2015

Abstract

Springs-mass systems are used in a great number of situations throughout the field of engineering. They have potential in scenarios such as observing simple harmonic motion or studying forces with gravity. In this case, we are evaluating the displacement of 3 masses that are hanging from springs at equilibrium. We will be utilizing Cramer's Rule, L-U decomposition, and the inversion of a matrix to do this.

1. Introduction

In this paper, we will be using matrices to solve a system of linear algebraic equations. We got these equations by observing the system when it reached its steady state, or equilibrium. These equations can be solved when using the ordinary differential equations which are satisfied by the system. We then use this to generate a stiffness matrix, which can be used to determine the displacement of the masses in the system. The equations and numbers that were generated from this process were solved using three different computational methods: Cramer's Rule, L-U decomposition, and the inversion of a matrix. Our analysis determined that while Cramer's Rule was the most simplistic, it had more situations where it did not work, as well as having increasing difficult as more equations were added. Conversely, L-U decomposition and matrix inversions yield similar and satisfactory results.

2. Physical Analysis

This project is based on the following spring system:

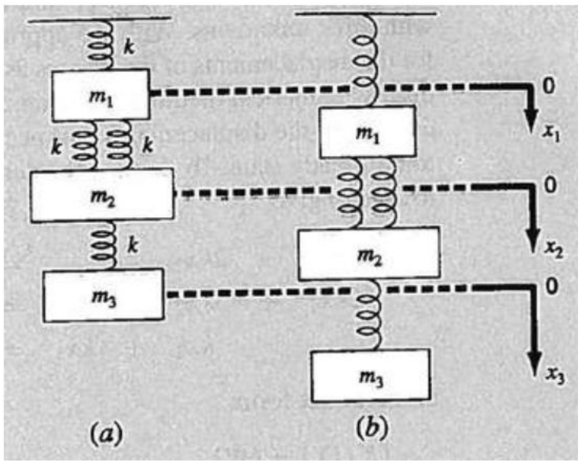

Figure 1

In this system, part (a) represents the system before it's released, while part (b) represents it after it is released and has reached equilibrium. By analyzing this system, we can come up with the following equations:

$$m_1 \frac{d^2 x_1}{dt^2} = 2k(x_2 - x_1) + m_1 g - k x_1 \quad (1)$$

$$m_2 \frac{d^2 x_2}{dt^2} = k(x_3 - x_2) + m_2 g - 2k(x_2 - x_1) \quad (2)$$

$$m_3 \frac{d^2 x_3}{dt^2} = m_3 g - k(x_3 - x_2) \quad (3)$$

In each equation 'k' represents a constant that is dependent on the spring's material and physical properties. The variable 'g' refers to the gravitational constant, and the variables 'x_1', 'x_2', and 'x_3' refer to the displacements of mass 1, mass 2, and mass 3, respectively. The equations in their entirety represent the force acting on a given mass. It is important to note that each equation features gravity, which is acting on all parts of the system. Equation (1) shows how 'm_1' is being held up by a single spring, and has the rest of the system pulling down on it. Equation (2) shows how 'm_2' and 'm_3' are pulling down, while having have the springs above it pull it upward. Equation (3) shows that in that particular spot, the only forces are gravity acting down and the springs pulling up. This can all be observed by referencing Figure 1.

After a time, the system will reach equilibrium. This means that there will be a change in the previous equations. This change is due to the net force changing to zero, which means the left side of equations (1), (2), and (3) will become zero. This gives us a set of 3 new equations, which are as follows:

$$3kx_1 - 2kx_2 = m_1 g \qquad (4)$$

$$-2kx_1 + 3kx_2 - kx_3 = m_2 g \qquad (5)$$

$$-kx_2 + kx_3 = m_3 g \qquad (6)$$

This system of linear algebraic equations represents how the gravitational force pulling down on a given mass is equal to the springs that are pulling up and down on the mass. These equations can then be turned into a matrix, which can then be analyzed. This system can be seen in part (b) of Figure 1.

3. Numerical Analysis

Equations (4), (5), and (6), as previously mentioned, can now be transformed into a matrix equation. This equation takes the form of Ax=b, and looks like:

$$(3k \ -2k \ 0 \ -2k \ 3k \ -k \ 0 \ -k \ k \ )\cdot(x_1 \ x_2 \ x_3 \ ) = (m_1 g \ m_2 g \ m_3 g \ )$$

Matrix ‘A’ is known as a stiffness matrix, while matrix ‘x’ represents the displacement of each mass. Matrix ‘b’ represents the gravitational force of each mass. In this paper, we used Cramer’s rule, L-U decomposition, and inverting a matrix to solve for matrix ‘x’, the displacement.

Cramer’s rule is a computational method that uses determinants to solve for a series of unknowns, in this case being ‘x_1’,’x_2’, and ‘x_3’. The core of Cramer’s rule starts out by taking the determinant of matrix A. This is then used along with three other determinants, which are determined by making substitutions of matrix b into the columns of matrix A. For example,

$$x_1 = \frac{1}{det(A)}\cdot det(m_1 g \ -2k \ 0 \ m_2 g \ 3k \ -k \ m_3 g \ -k \ k \ )$$

In this, we can see that matrix b is substituted in for the first column of matrix A. The determinant of this is taken, and divided by the determinant of the original matrix A. This solves for x_1. The same is to be repeated for x_2, except with matrix b being substituted in for the second column. Once again, x_3 can be found by substituting matrix b in for the third column.

L-U decomposition is a more advanced method, and it works disassembling matrix A into two separate matrices; the two parts being called L and U. These matrices typically look like the following:

$$L = (1\ 0\ 0\ L_{21}\ 1\ 0\ L_{31}\ L_{32}\ 1\ \ )$$

$$U = (U_{11}\ U_{12}\ U_{13}\ 0\ U_{22}\ U_{23}\ 0\ 0\ U_{33}\ \ )$$

These originate from the lower and upper portions of matrix A, and when multiplied together will reform it. Once this point is reached, a forward and backward substitution is needed.

First, we solve the equation:

$$L \cdot y = b$$

This represents the forward substitution.

Next, we solve for matrix x using:

$$U \cdot x = y$$

This represents the backward substitution and gives us our values for matrix x.

Finally, the inverse of matrix A can be solved for by using the equation:

$$A \cdot A^{-1} = (1\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 1\ \ )$$

This entire process is done using an L-U decomposition to solve for A^{-1}. Multiplying this matrix with matrix b will then give us our desired values, matrix x.

4. Results

All three methods shared a few common features. In the beginning, they all had established values for matrix A. These values, however, could easily be changed for application in another problem. They also all used dynamic memory, which was established in the beginning of the program and was used to save memory. Additionally, all of the programs had arrays which were defined by user input. These arrays were then used for the rest of the calculations. At the conclusion of each program, the values for matrix X are presented, as well as the dynamic memory is closed and deleted.

Cramer's rule was analyzed by setting up the calculations for the determinants of each of the newly created matrices. The values for these matrices are using those defined at the beginning of the program. The code solves for these determinants, applies Cramer's rule, and then displays the displacement.

L-U Decomposition was a bit more complicated, requiring many for-loops to do its computation. After starting out as previously mentioned, the program creates two matrices which are based off of matrix A, called L and U. This is done by using a series of nested for-loops.

At this time, a "dummy" array is created. This is to prevent the code from overwriting itself, and allows the computations to be run smoothly. Two for-loops are then started. The first loop helps establish the dummy array. The second loop solves for a sum. This is used to create one of our matrices and is the forward substitution portion of our code.

Next, we establish a value for our temporary displacement. Once this is done, we can begin by starting two nested for-loops again. The first loop resets sum to zero. It then starts the second loop, which updates sum by scanning through matrix A and the temporary displacement matrix. Computations are done, and the real displacement is found. These values are then presented to the user.

Finally, the inverse of matrix A is found using an almost identical method to that of L-U decomposition. This time, however, during the backward substation there is a slight difference: a new matrix, A^{-1} is constructed based off of the decomposed values of matrix A. This is computed, and then matrix A^{-1} is displayed.

Cramer's rule and L-U decomposition both produced values of displacement as follows:

$$x = (20\ 25\ 30\ )$$

The inverse of A was calculated to be:

$$A^{-1} = (0.5\ 0.5\ 0.5\ 0.5\ 0.75\ 0.75\ 0.5\ 0.75\ 1.25\ )$$

After working out the math on paper and comparing, we have determined that our methods are accurate.

5. Conclusion

In this paper, we analyzed three methods of solving linear algebraic equations using matrices. By using Cramer’s Rule and L-U decomposition we found the displacement of three masses after they are released and reached equilibrium. Additionally, we found the inverse of the matrix representing the corresponding stress field. Some of the issues that arose during this project are values overwriting themselves and erroneous computations due to syntax errors. However, after diligent debugging, these issues were resolved.

References

Various users, cplusplus forums, 2014.

Corina Drapaca, Lecture notes on matrix algebra and solving systems of linear algebraic equations

Appendix

Cramer's Rule:

Psuedo-code:

```
Set up dynamic memory

Establish constants and variables

User input matrix dimensions

Establish matrix K and matrix W

detK = (mathematical implementation of determinant of K)

detX = (mathematical implementation of determinant of K with W in column 1)

detY = (mathematical implementation of determinant of K with W in column 2)

detZ = (mathematical implementation of determinant of K with W in column 3)

x = detx/detK

y = dety/detK

z = detz/detK

f1 = (2 * kk * (y - x)) + (m1 * g) - (kk * x)
f2 = (kk * (z - y)) + (m2 * g) - (2 * kk * (y - x))
f3 = (m3 * g) - (kk * (z - y))

display displacements and forces
close dynamic memory
```

C++:

```
#include <iostream>
#include <cmath>
using namespace std;

typedef double* DArrayPtr;

int main()
{
	const int m1 = 2, m2 = 1, m3 = 1, g = 10, kk = 2;
	double f1, f2, f3, x, y, z;
	int d1, d2;

	cout << "Enter the matrix dimensions:\n";
	cin >> d1 >> d2;

	//setting up dynamic memory
	DArrayPtr *K = new DArrayPtr[d1];

	for (int i = 0; i < d1; i++)
		K[i] = new double[d2];

	//Setting values for dyanmic memory
	K[0][0] = 3 * kk;
	K[0][1] = -2 * kk;
	K[0][2] = 0;
	K[1][0] = -2 * kk;
	K[1][1] = 3 * kk;
	K[1][2] = -kk;
	K[2][0] = 0;
	K[2][1] = -kk;
	K[2][2] = kk;

	int W[3][1];

	W[0][0] = m1*g;
	W[1][0] = m2*g;
	W[2][0] = m3*g;

	//Determinants
	double detK = (K[0][0] * K[1][1] * K[2][2]) + (K[0][1] * K[1][2] * K[2][0]) +
(K[0][2] * K[1][0] * K[2][1]) - (K[0][2] * K[1][1] * K[2][0]) - (K[0][1] * K[1][0] *
K[2][2]) - (K[0][0] * K[1][2] * K[2][1]);
	double detx = (W[0][0] * K[1][1] * K[2][2]) + (K[0][1] * K[1][2] * W[2][0]) +
(K[0][2] * W[1][0] * K[2][1]) - (K[0][2] * K[1][1] * W[2][0]) - (K[0][1] * W[1][0] *
K[2][2]) - (W[0][0] * K[1][2] * K[2][1]);
	double dety = (K[0][0] * W[1][0] * K[2][2]) + (W[0][0] * K[1][2] * K[2][0]) +
(K[0][2] * K[1][0] * W[2][0]) - (K[0][2] * W[1][0] * K[2][0]) - (W[0][0] * K[1][0] *
K[2][2]) - (K[0][0] * K[1][2] * W[2][0]);
	double detz = (K[0][0] * K[1][1] * W[2][0]) + (K[0][1] * W[1][0] * K[2][0]) +
(W[0][0] * K[1][0] * K[2][1]) - (W[0][0] * K[1][1] * K[2][0]) - (K[0][1] * K[1][0] *
W[2][0]) - (K[0][0] * W[1][0] * K[2][1]);

	//Cramer's Rule
	x = detx / detK;
	y = dety / detK;
	z = detz / detK;

	//Calculating force
	f1 = (2 * kk * (y - x)) + (m1 * g) - (kk * x);
	f2 = (kk * (z - y)) + (m2 * g) - (2 * kk * (y - x));
	f3 = (m3 * g) - (kk * (z - y));

	cout << "Displacement of mass m1: " << x << endl;
	cout << "Displacement of mass m2: " << y << endl;
	cout << "Displacement of mass m3: " << z << endl;
	cout << "Force on mass 1:\n" << f1 << endl;
	cout << "Force on mass 2:\n" << f2 << endl;
	cout << "Force on mass 3:\n" << f3 << endl;

	//Close dynamic memory
	for (int i = 0; i < d1; i++)
		delete[] K[i];

	delete[] K;
	system("pause");
	return 0;
}
```

MATLAB:

```
%Cramer's Rule

m1 = 2;

m2 = 1;

m3 = 1;

g = 10;

kk = 2; %setting constants



K =[3*kk, -2*kk, 0; -2*kk, 3*kk, -kk; 0, -kk, kk];

W =[m1*g; m2*g; m3*g]; %making initial arrays



detK= (K(1) * K(5) * K(9)) + (K(2) * K(6) * K(7)) + (K(3) * K(4) * K(8)) - (K(3) * K(5) *
K(7)) - (K(2) * K(4) * K(9)) - (K(1) * K(6) * K(8));

detx = (W(1) * K(5) * K(9)) + (K(2) * K(6) * W(3)) + (K(3) * W(2) * K(8)) - (K(3) * K(5)
* W(3)) - (K(2) * W(2) * K(9)) - (W(1) * K(6) * K(8));
```

```
dety = (K(1) * W(2) * K(9)) + (W(1) * K(6) * K(7)) + (K(3) * K(4) * W(3)) - (K(3) * W(2)
* K(7)) - (W(1) * K(4) * K(9)) - (K(1) * K(6) * W(3));

detz = (K(1) * K(5) * W(3)) + (K(2) * W(2) * K(7)) + (W(1) * K(4) * K(8)) - (W(1) * K(5)
* K(7)) - (K(2) * K(4) * W(3)) - (K(1) * W(2) * K(8));


x = detx/detK;

y = dety/detK;

z = detz/detK; %Cramer's Rule


f1 = (2 * kk *( y - x)) + (m1 * g) - (kk * x);

f2 = (kk * (z-y)) +(m2*g)-(2 * kk * (y-x));

f3 = (m3 * g) - (kk * (z - y));


fprintf('The force on mass 1 is: %f\n.',f1);

fprintf('The force on mass 2 is: %f\n.',f2);

fprintf('The force on mass 3 is: %f\n.',f3);

fprintf('The displacement of mass 1 is: %f\n.',x);

fprintf('The displacement of mass 2 is: %f\n.',y);

fprintf('The displacement of mass 3 is: %f\n.',z);
```

L-U Decomposition:

Psuedo-code:

```
set variables (arrays, constants, initialize variables to be used later)

create dynamic memory

for c=1; c < d -1; c++

for a = c + 1; a<d; a++
        factor = $k_{ik}/k_{kk}$
        $k_{ik}$ = factor
        for b = c + 1; b<d; b++
            $a_{ij} = a_{ij}$ – factor * $a_{kj}$
        end for
    end for
end for
y[0]=W[0]
for (a = 1; a < d; a++)
    sum = W[a]
    for (b = 0; b <= a - 1; b++)
    end for
    sum = sum – $k_{ij}$ * $b_j$
    $b_i$ = sum
end for
$x_n = b_n / k_{nn}$
for a = d - 1; a >= 0; a—
    sum = 0
    for b = a + 1; b < d; b++
        sum = sum + $k_{ij}$ * $x_j$
    end for
    $x_i = (b_i – \text{sum}) / k_{ii}$
```

```
end for

display $x_i$
```

C++:

```
#include <iostream>
using namespace std;

//creates dynamic memory space
typedef double* DArrayPtr;

int main()
{
	double W[3], x[3], y[3];
	int a, b, c, d1, d2, d(3);
	const int m1 = 2, m2 = 1, m3 = 1, g = 10, kk = 2;
	double factor, sum;

	cout << "Enter the dimensions of the system's matrix:\n";
	cin >> d1 >> d2;

	//sets up dynamic memory
	DArrayPtr *K = new DArrayPtr[d1];

	for (int i = 0; i < d1; i++)
		K[i] = new double[d2];

	//establishes dynamic memory
	K[0][0] = 3 * kk;
	K[0][1] = -2 * kk;
	K[0][2] = 0;
	K[1][0] = -2 * kk;
	K[1][1] = 3 * kk;
	K[1][2] = -kk;
	K[2][0] = 0;
	K[2][1] = -kk;
	K[2][2] = kk;

	W[0] = m1*g;
	W[1] = m2*g;
	W[2] = m3*g;

	//L-U decomposition
	for (c = 1; c< d - 1; c++)
	{
		for (a = c + 1; a<d; a++)
		{
			factor = K[a][c] / K[c][c];
			K[a][c] = factor;
			for (b = c + 1; b<d; b++)
			{
				K[a][b] = K[a][b] - (factor*K[c][b]);
			}
		}
	}

	// forward substitution
	y[0] = W[0]; //stops W from overwriting itself
	for (a = 1; a < d; a++)
	{
		sum = W[a];
		for (b = 0; b <= a - 1; b++)
		{
			sum = sum - (K[a][b] * y[b]);
		}
		y[a] = sum;
	}

	// backward substitution
	x[d - 1] = y[d - 1] / K[d - 1][d - 1];
	for (a = d - 1; a >= 0; a--)
	{
		sum = 0;
		for (b = a + 1; b < d; b++)
		{
			sum = sum + (K[a][b] * x[b]);
		}
		x[a] = (y[a] - sum) / K[a][a];
	}

	//displays displacement of masses
	cout << "Displacement of mass m1 is: " << x[0] << endl;
	cout << "Displacement of mass m2 is: " << x[1] << endl;
	cout << "Displacement of mass m3 is: " << x[2] << endl;

	//free dynamic memory
	for (int i = 0; i < d1; i++)
		delete[] K[i];

	delete[] K;
	system("pause");
	return 0;
}
```

MATLAB:

```
m1=2;

m2=1;

m3=1;

g=10;

kk=2;
```

```
a = [3*kk, -2*kk, 0; -2*kk, 3*kk, -kk; 0, -kk, kk];

b = [m1*g; m2*g; m3*g];

n=input('Please enter the dimensions of the matrix\n');

    %LU decomposition

    for k=1:n-1

        for i = k+1:n

            factor = a(i,k)/a(k,k);

            a(i,k) = factor;

            for j= k+1:n

                a(i,j) = a(i,j) - (factor * a(k,j));

            end

        end

    end

%forward substitution

    for i=2:n

        sum = b(i);

        for j=1:i-1

            sum = sum - (a(i,j)*b(j));

        end

        b(i) = sum;

    end

displacement = [x(1);x(2);x(3)];
```

Inversion of a Matrix:

Psuedo-code:

```
set variables (arrays, constants, initialize variables to be used later)
create dynamic memory
for c=1; c < d -1; c++
    for a = c + 1; a<d; a++
        factor = $k_{ik}/k_{kk}$
        $k_{ik}$ = factor
        for b = c + 1; b<d; b++
            $a_{ij} = a_{ij} – \text{factor} * a_{kj}$
        end for
    end for
end for
y[0]=W[0]
for (a = 1; a < d; a++)
    sum = W[a]
    for (b = 0; b <= a - 1; b++)
    end for
    sum = sum – $k_{ij} * b_j$
    $b_i$ = sum
```

```
end for

$x_n = b_n / k_{nn}$

for a = d - 1; a >= 0; a—

sum = 0

for b = a + 1; b < d; b++

sum = sum + $k_{ij} * x_j$

end for

for int j=0; j<n; j++
$ai_{ji} = x_j$

end for

end for

for int i = 1; i<n; i++

for int j = 0; j<n; j++

display $A^{-1}$

end for

end for
```

<u>C++:</u>

```
#include <iostream>

using namespace std;


//creating dynamic memory

typedef double* DArrayPtr;
```

```
int main()

{

        double W[3], x[3], y[3], ai[3][3];

        int a, b, c, d1, d2, d(3);

        const int m1 = 2, m2 = 1, m3 = 1, g = 10, k = 2;

        double factor, sum;



        cout << "Enter matrix dimensions:\n";

        cin >> d1 >> d2;



        //gives exact space for dynamic memory

        DArrayPtr *A = new DArrayPtr[d1];



        for (int i = 0; i < d1; i++)

                A[i] = new double[d2];



        //setting up dynamic memory

        A[0][0] = 3 * k;

        A[0][1] = -2 * k;

        A[0][2] = 0;

        A[1][0] = -2 * k;
```

```
A[1][1] = 3 * k;

A[1][2] = -k;

A[2][0] = 0;

A[2][1] = -k;

A[2][2] = k;


//L-U decomposition

for (c = 0; c< d - 1; c++)

{

        for (a = c + 1; a<d; a++)

        {

                factor = A[a][c] / A[c][c];

                A[a][c] = factor;

                for (b = c + 1; b<d; b++)

                {

                        A[a][b] = A[a][b] - (factor*A[c][b]);

                }

        }

}

for (int i = 0; i < d; i++)

{

        for (int j = 0; j<d; j++)
```

```
{
        if (i == j)
        {
                W[j] = 1;
        }
        else
        {
                W[j] = 0;
        }
}
// forward substitution
y[0] = W[0];
for (a = 1; a < d; a++)
{
        sum = W[a];

        for (b = 0; b <= a - 1; b++)
        {
                sum = sum - (A[a][b] * y[b]);
        }
        y[a] = sum;
}
```

```
        // backward substitution

        x[d - 1] = y[d - 1] / A[d - 1][d - 1];

        for (a = d - 1; a >= 0; a--)

        {

                sum = 0;

                for (b = a + 1; b < d; b++)

                {

                        sum = sum + (A[a][b] * x[b]);

                }

                x[a] = (y[a] - sum) / A[a][a];

        }

        for (int j = 0; j<d; j++)

        {

                ai[j][i] = x[j];

        }

}


//displays inverse

        cout << "A^(-1): \n";

        for (int i = 0; i<d; i++){

        for (int j = 0; j<3; j++){
                cout << ai[i][j] << "  ";
            }
            cout << endl;
        }

        //free dynamic memory
        for (int i = 0; i < d1; i++)
            delete[] A[i];

        delete[] A;
        system("pause");
        return 0;
}
```

<u>MATLAB:</u>

```
m1=2;
m2=1;
m3=1;
g=10;
kk=2;
a = [3*kk, -2*kk, 0; -2*kk, 3*kk, -kk; 0, -kk, kk];
b = [m1*g; m2*g; m3*g];
n=input('Please enter the dimensions of the matrix\n');
```

```
%LU decomposition

for k=1:n-1

    for i = k+1:n

        factor = a(i,k)/a(k,k);

        a(i,k) = factor;

        for j= k+1:n

            a(i,j) = a(i,j) - (factor * a(k,j));

        end

    end

end


%inverse

for i=1:n

    for j=1:n

        if i == j

            b(j) = 1;

        else b(j) = 0;

        end

    end

    %forward substitution

for i=2:n

    sum = b(i);

    for j=1:i-1

        sum = sum - (a(i,j)*b(j));

    end

    b(i) = sum;
    end


%backward substitution

       x(n) = b(n) / a(n,n);

        for i=n:-1:1

            sum=0;

               for j=i+1:n

                sum = sum + (a(i,j) * x(j));

               end

                x(i) = (b(i) - sum)/a(i,i);

        end

        for j=1:n

        for i=1:n

            ai(j,i)=x(j);

        end

        end

    end

    ai = [ai(1,1),ai(1,2),ai(1,3);ai(2,1),ai(2,2),ai(2,3);ai(3,1),ai(3,2),ai(3,3)];
```