Hyperbolic Partial Differential Equations

Hakim Lachnani

Numerical Methods

Problem 1: Analytical Solution to 1-D Wave Equation

The standard 1 dimensional wave equation is presented as $U_{tt} = c^2U_{xx}$. To find an analytical solution, the following assumptions must first be made: the mass of the string per unit length is constant, gravity can be neglected, and all point motions are vertical. Having made these assumptions, a solution can be obtained using the method of Separation of Variables. Specifically, $U(x,t) = F(x)G(t)$. This allows the equation to be separated into two constant parts:

$F_{xx} - kF = 0$ and $G_{tt} - c^2kG = 0$

From here, boundary conditions must be applied to the theory of Fourier series to obtain a series solution. Since our initial velocity is zero, this series can be summed to

$U(x,t) = .5[f(x-ct)+f(x+ct)]$

Where f is the odd extension of the initial displacement.

The problem in question asks us to find period of a guitar string of length 80cm with tension 40 grams and mass of 1 gram. c^2 is given by $T*g/w$ where $T = 40000$, $g = 980$, and $w = 1/80$. This leads to $c = 56000$. Since the average frequency for a guitar string ranges anywhere from 100-350 Hz, we can expect the period to be between .0028 and .0100 seconds. The analytical solution confirms this.

t	0	10	20	30	40	50	60	70	80
0.0000	0.0000	0.3000	0.6000	0.5000	0.4000	0.3000	0.2000	0.1000	0.000
0.0005	0.0000	-0.0600	0.0400	0.1400	0.2400	0.3000	0.2000	0.1000	0.000
0.0010	0.0000	-0.1000	-0.2000	-0.3000	-0.3200	-0.2200	-0.1200	-0.0200	0.000
0.0015	0.0000	-0.1000	-0.2000	-0.3000	-0.4000	-0.5000	-0.5200	-0.3000	0.000
0.0020	0.0000	-0.1000	-0.2000	-0.2600	-0.1600	-0.0600	0.0400	0.1000	0.000
0.0025	0.0000	0.1000	0.2000	0.3000	0.4000	0.3000	0.2000	0.1000	0.000
0.0030	0.0000	0.3000	0.4400	0.5000	0.4000	0.3000	0.2000	0.1000	0.000
0.0035	0.0000	-0.1000	-0.1200	-0.0200	0.0800	0.1800	0.2000	0.1000	0.000
0.0040	0.0000	-0.1000	-0.2000	-0.3000	-0.4000	-0.3800	-0.2800	-0.1800	0.000
0.0045	0.0000	-0.1000	-0.2000	-0.3000	-0.4000	-0.4600	-0.3600	-0.2600	0.000
0.0050	0.0000	-0.1000	-0.2000	-0.1000	0.0000	0.1000	0.2000	0.1000	0.000

The analytical solution reaches a second maximum, indicating a full revolution, when t is approximately .003 seconds. We can use finer increments on a smaller interval to find a more accurate result.

t	0	10	20	30	40	50	60	70	80
0.002700	0.0000	0.3000	0.4240	0.5000	0.4000	0.3000	0.2000	0.1000	0.000
0.002750	0.0000	0.3000	0.4800	0.5000	0.4000	0.3000	0.2000	0.1000	0.000
0.002800	0.0000	0.3000	0.5360	0.5000	0.4000	0.3000	0.2000	0.1000	0.000
0.002850	0.0000	0.3000	0.5920	0.5000	0.4000	0.3000	0.2000	0.1000	0.000
0.002900	0.0000	0.3000	0.5520	0.5000	0.4000	0.3000	0.2000	0.1000	0.000
0.002950	0.0000	0.3000	0.4960	0.5000	0.4000	0.3000	0.2000	0.1000	0.000
0.003000	0.0000	0.3000	0.4400	0.5000	0.4000	0.3000	0.2000	0.1000	0.000
0.003050	0.0000	0.2840	0.3840	0.4840	0.4000	0.3000	0.2000	0.1000	0.000
0.003100	0.0000	0.2280	0.3280	0.4280	0.4000	0.3000	0.2000	0.1000	0.000
0.003150	0.0000	0.1720	0.2720	0.3720	0.4000	0.3000	0.2000	0.1000	0.000
0.003200	0.0000	0.1160	0.2160	0.3160	0.4000	0.3000	0.2000	0.1000	0.000

Now we have t ~= .00285 seconds. This indicates a frequency of 350Hz, which is in agreement with reality.

Problem 1: Solve the problem using Finite Differences

6 interior nodes splits the string into 7 even partitions. Using the explicit method, we can create a linear system that outputs the next time iteration from the previous 2 time iterations. The system is of the form Ax + b = c where A is a matrix, x is the previous iteration, b is the iteration before the previous iteration, and c is the time we are interested in. The matrix A takes the form

```
A =

    1.5200    0.2400         0         0         0         0
    0.2400    1.5200    0.2400         0         0         0
         0    0.2400    1.5200    0.2400         0         0
         0         0    0.2400    1.5200    0.2400         0
         0         0         0    0.2400    1.5200    0.2400
         0         0         0         0    0.2400    1.5200
```

Given that this matrix is tridiagonal, it is easier to solve using an iterative process, which is what I have elected to do. In playing around with this explicit method, two things have been made clear. First, the method is very quick for one that does not require the analytical solution. Even with 80 partitions in x and 10,000 steps in time, the method quickly delivers results. However, the steps must be chosen such that $\rho = (c*k/g)^2 < 1$. If not, the method will not converge to a true solution. This is what is dictated by the theory, and it is what I found in my results. In order to accommodate this, I let c = 56 rather than 56000. This allowed ρ to stay less than 1 and for the method to converge. However, this can only be done with good knowledge of the true solution. Indeed, the 6 node code gives a solution at approximately 2.85, which is exactly 3 orders of magnitude higher than the true solution.

```
     t          0        11.43      22.86      34.29      45.71      57.14      68.57      80

  0.2000     0.0000     0.2847     0.4233     0.4473     0.3429     0.2560     0.2880     0.0000
  0.3000     0.0000     0.2051     0.2887     0.4065     0.3471     0.3120     0.2706     0.0000
  0.4000     0.0000     0.0964     0.1624     0.3233     0.3571     0.3664     0.1981     0.0000
  0.5000     0.0000    -0.0196     0.0589     0.2095     0.3613     0.3782     0.1185     0.0000
  0.6000     0.0000    -0.1121    -0.0273     0.0960     0.3331     0.3236     0.0728     0.0000
  0.7000     0.0000    -0.1573    -0.1043     0.0099     0.2457     0.2111     0.0699     0.0000
  0.8000     0.0000    -0.1520    -0.1666    -0.0471     0.0934     0.0729     0.0840     0.0000
  0.9000     0.0000    -0.1137    -0.1967    -0.0990    -0.0976    -0.0577     0.0753     0.0000
  1.0000     0.0000    -0.0681    -0.1834    -0.1740    -0.2793    -0.1659     0.0166     0.0000
  1.1000     0.0000    -0.0337    -0.1402    -0.2766    -0.4085    -0.2575    -0.0899     0.0000
  1.2000     0.0000    -0.0169    -0.1042    -0.3781    -0.4698    -0.3452    -0.2151     0.0000
  1.3000     0.0000    -0.0169    -0.1130    -0.4358    -0.4791    -0.4315    -0.3199     0.0000
  1.4000     0.0000    -0.0360    -0.1762    -0.4264    -0.4666    -0.5024    -0.3747     0.0000
  1.5000     0.0000    -0.0801    -0.2659    -0.3666    -0.4531    -0.5341    -0.3702     0.0000
  1.6000     0.0000    -0.1495    -0.3351    -0.3034    -0.4382    -0.5070    -0.3162     0.0000
  1.7000     0.0000    -0.2276    -0.3522    -0.2801    -0.4075    -0.4175    -0.2321     0.0000
  1.8000     0.0000    -0.2810    -0.3220    -0.3048    -0.3486    -0.2811    -0.1368     0.0000
  1.9000     0.0000    -0.2767    -0.2779    -0.3441    -0.2630    -0.1263    -0.0433     0.0000
  2.0000     0.0000    -0.2063    -0.2494    -0.3480    -0.1640     0.0157     0.0407     0.0000
  2.1000     0.0000    -0.0967    -0.2342    -0.2841    -0.0660     0.1205     0.1088     0.0000
  2.2000     0.0000     0.0031    -0.1980    -0.1559     0.0243     0.1777     0.1537     0.0000
  2.3000     0.0000     0.0539    -0.1034     0.0055     0.1083     0.1924     0.1674     0.0000
  2.4000     0.0000     0.0540     0.0552     0.1654     0.1877     0.1808     0.1469     0.0000
  2.5000     0.0000     0.0414     0.2399     0.3042     0.2602     0.1628     0.0993     0.0000
  2.6000     0.0000     0.0665     0.3924     0.4170     0.3198     0.1529     0.0431     0.0000
  2.7000     0.0000     0.1539     0.4726     0.5005     0.3627     0.1567     0.0029     0.0000
  2.8000     0.0000     0.2809     0.4829     0.5442     0.3892     0.1731    -0.0011     0.0000
  2.9000     0.0000     0.3889     0.4595     0.5360     0.4011     0.1995     0.0371     0.0000
  3.0000     0.0000     0.4205     0.4375     0.4771     0.3969     0.2353     0.1053     0.0000
```

Von Neumann stability analysis for the explicit method forces ρ < 1. The implicit method has no such requirement. It is absolutely stable. However, I was unable to produce a program which converged using the implicit method. Furthermore, the two implicit programs failed in completely different ways.

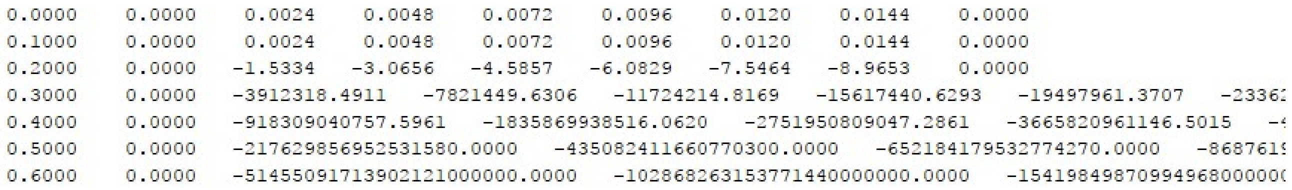
The first program suffered from an inability to dip below zero. The values are always non-negative. Therefore, there is no way to see when a period is complete. The code was originally written with a tridiagonal solver for speed. I substituted the 'Naïve_Gauss' class code for reliability, but it did not solve the problem. The first few lines of results were as follows.

```
0.0000     0.0000     0.3429     0.5714     0.4571     0.3429     0.2286     0.1143     0.0000
0.1000     0.0000     0.3429     0.5166     0.4571     0.3429     0.2286     0.1143     0.0000
0.2000     0.0000     0.5647     0.4123     0.4388     0.3417     0.2299     0.1349     0.0000
0.3000     0.0000     0.8668     0.3599     0.3902     0.3352     0.2410     0.2378     0.0000
0.4000     0.0000     1.0770     0.4460     0.3271     0.3193     0.2855     0.4797     0.0000
0.5000     0.0000     1.1039     0.6611     0.3030     0.3038     0.3999     0.8394     0.0000
0.6000     0.0000     0.9840     0.8961     0.3801     0.3275     0.6102     1.2019     0.0000
0.7000     0.0000     0.8372     1.0217     0.5826     0.4561     0.9018     1.4235     0.0000
0.8000     0.0000     0.7738     0.9897     0.8700     0.7467     1.2103     1.4280     0.0000
```

These results were with c set at 56 as with the explicit method. However, putting the true value for c only compounded the errors:

```
0.3000     0.0000    -2.2852    -1.1419    -1.7130    -2.2843    -2.8558    -3.4274     0.0000
0.4000     0.0000     5.5984     3.9973     4.7965     5.5961     6.3962     7.1968     0.0000
0.5000     0.0000    -4.9109    -2.8514    -3.8783    -4.9060    -5.9347    -6.9646     0.0000
0.6000     0.0000     8.2224     5.7038     6.9578     8.2133     9.4707    10.7303     0.0000
0.7000     0.0000    -7.5327    -4.5541    -6.0346    -7.5177    -9.0038   -10.4935     0.0000
0.8000     0.0000    10.8414     7.4019     9.1082    10.8184    12.5333    14.2537     0.0000
```

In the second code, I hard iterated the values of b in the Ax = b linear system in order to have greater control. I also implemented a tridiagonal system that was simpler than Naïve_Gauss. Unfortunately, this failed as well.



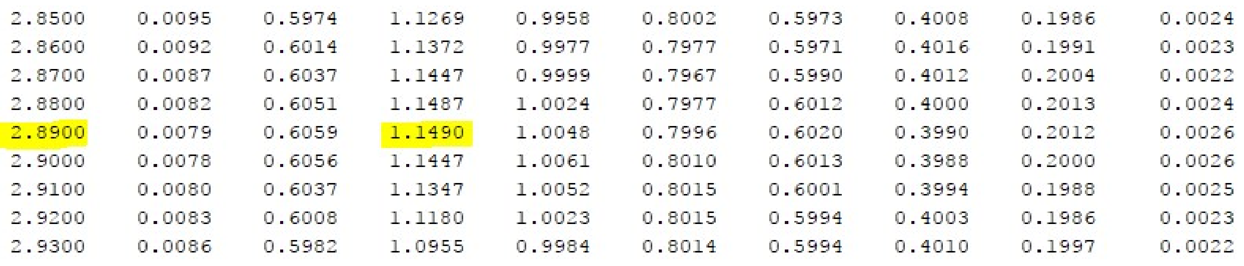
```
0.0000    0.0000    0.0024    0.0048    0.0072    0.0096    0.0120    0.0144    0.0000
0.1000    0.0000    0.0024    0.0048    0.0072    0.0096    0.0120    0.0144    0.0000
0.2000    0.0000    -1.5334   -3.0656   -4.5857   -6.0829   -7.5464   -8.9653   0.0000
0.3000    0.0000    -3912318.4911    -7821449.6306    -11724214.8169    -15617440.6293    -19497961.3707    -2336
0.4000    0.0000    -918309040757.5961    -1835869938516.0620    -2751950809047.2861    -3665820961146.5015    -
0.5000    0.0000    -2176298569525315 80.0000    -4350824116607703 00.0000    -6521841795327742 70.0000    -868761
0.6000    0.0000    -514550917139021210 00000.0000    -102868263153771440000000.0000    -15419849870994968000000
```

The values quickly grew too large. This problem was compounded with each iteration. In trying to fix these codes, I attempted to modify the implicit method used for the parabolic PDE, but this did not work due to dependence on two previous times rather than one. The pseudocode in Cheney and Kincaid's book iterates on a tridiagonal solution that cannot work for the hyperbolic PDE. The implicit method for the hyperbolic PDE involves subtracting *U(x,t-k)* from the solution vector at each iteration. While simple in principle, I could not make the code function properly.

I found success with the Crank-Nicolson method. By combining the explicit and implicit methods, we have,

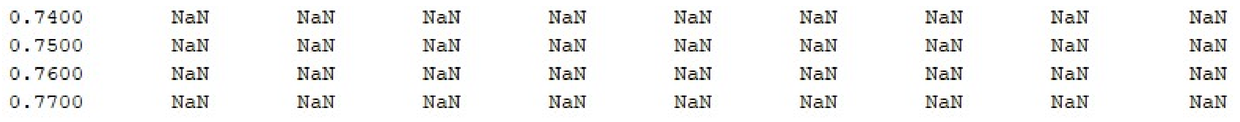
$$(1+\rho)U(x,t+k) - .5\rho U(x+h,t+k) - .5\rho U(x-h,t+k) = 2U(x,t) + .5\rho U(x+h,t-k) - \rho U(x,t-k) + .5\rho U(x-h,t-k)$$

This method led to a solution (adjusting from c = 56) of .00289 for the period of the string. Interestingly, the values for the position of the string are incorrect, but they do reach a maximum. Near t = .003,



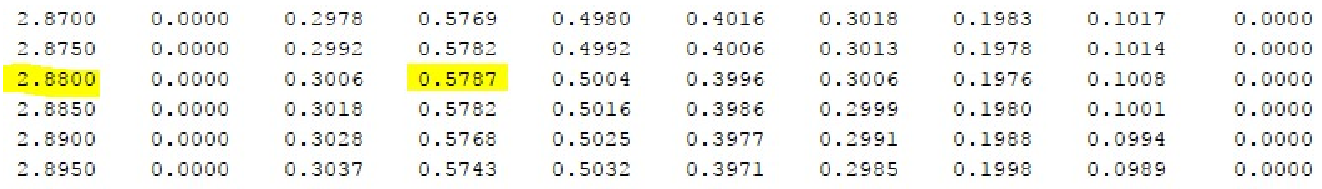
```
2.8500    0.0095    0.5974    1.1269    0.9958    0.8002    0.5973    0.4008    0.1986    0.0024
2.8600    0.0092    0.6014    1.1372    0.9977    0.7977    0.5971    0.4016    0.1991    0.0023
2.8700    0.0087    0.6037    1.1447    0.9999    0.7967    0.5990    0.4012    0.2004    0.0022
2.8800    0.0082    0.6051    1.1487    1.0024    0.7977    0.6012    0.4000    0.2013    0.0024
2.8900    0.0079    0.6059    1.1490    1.0048    0.7996    0.6020    0.3990    0.2012    0.0026
2.9000    0.0078    0.6056    1.1447    1.0061    0.8010    0.6013    0.3988    0.2000    0.0026
2.9100    0.0080    0.6037    1.1347    1.0052    0.8015    0.6001    0.3994    0.1988    0.0025
2.9200    0.0083    0.6008    1.1180    1.0023    0.8015    0.5994    0.4003    0.1986    0.0023
2.9300    0.0086    0.5982    1.0955    0.9984    0.8014    0.5994    0.4010    0.1997    0.0022
```

Having played with the values of c, it is clear that that is the change causing the problem. However, if c = 56000 is entered, there results are undefined.



```
0.7400       NaN       NaN       NaN       NaN       NaN       NaN       NaN       NaN        NaN
0.7500       NaN       NaN       NaN       NaN       NaN       NaN       NaN       NaN        NaN
0.7600       NaN       NaN       NaN       NaN       NaN       NaN       NaN       NaN        NaN
0.7700       NaN       NaN       NaN       NaN       NaN       NaN       NaN       NaN        NaN
```

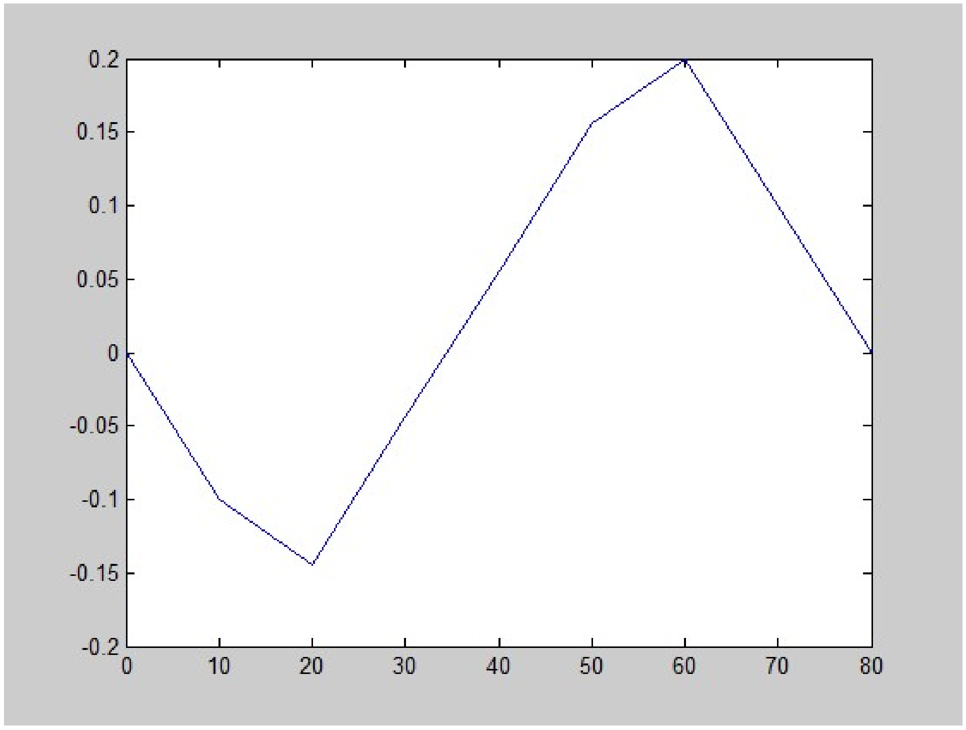
The fact that the Crank-Nicolson method still reveals a solution shows that the method preserves something integral about the function. Since maxima and minima are preserved, this connection likely comes through the derivative. Despite having a solution, the Crank-Nicolson method is slower than the explicit method. Other than the analytic solution, a program using the explicit method with many partitions on x seems to be best. The following was produced using the same general code as the 6 node code, but with h = 1 and k = .0005. The Crank-Nicolson method took over 300 seconds to reach t = 5 with k = .001. The explicit method took 0.64 seconds with twice the accuracy.



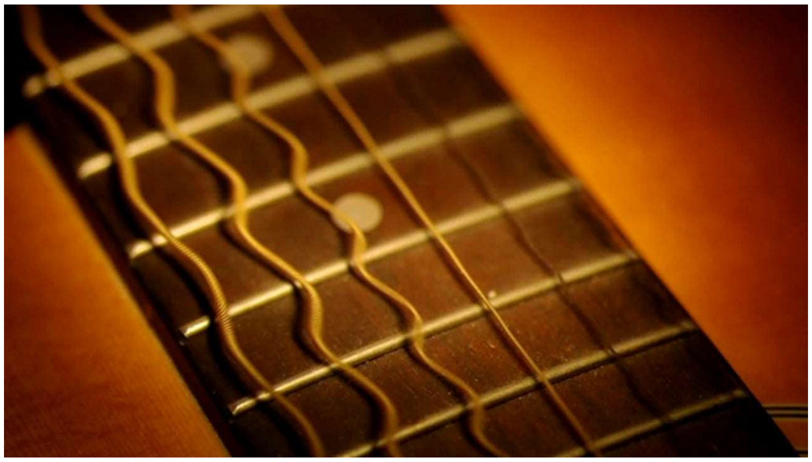
```
    2.8700    0.0000    0.2978    0.5769    0.4980    0.4016    0.3018    0.1983    0.1017     0.0000
    2.8750    0.0000    0.2992    0.5782    0.4992    0.4006    0.3013    0.1978    0.1014     0.0000
    2.8800    0.0000    0.3006    0.5787    0.5004    0.3996    0.3006    0.1976    0.1008     0.0000
    2.8850    0.0000    0.3018    0.5782    0.5016    0.3986    0.2999    0.1980    0.1001     0.0000
    2.8900    0.0000    0.3028    0.5768    0.5025    0.3977    0.2991    0.1988    0.0994     0.0000
    2.8950    0.0000    0.3037    0.5743    0.5032    0.3971    0.2985    0.1998    0.0989     0.0000
```

Furthermore, the explicit method was closer to the analytical solution.

The most interesting discovery while playing with the codes was the behavior of the string. I had imagined that it would almost immediately round off and move up and down as a unit. Observing the graph of the analytical solution made it clear that this is not the case.



Upon further research, I discovered that this is not the case for real guitar strings either. Slow motion photography shows that guitar strings do indeed act as waves.



Problem 2: Analytical Solution to 2D Wave Equation

Just as with the one dimensional case, this problem requires some assumptions: uniform mass distribution, uniform tension, and vertical movement. Also like its one dimensional counterpart, the equation is tackled using separation of variables. This time,

$G_{tt}+\lambda^2 G = 0$ and $F_{xx}+F_{yy}+v^2F = 0$

Once again ODEs are formed, which lead to boundary conditions. As with the one dimensional case, the solution is a Fourier series. However, it cannot be simplified as easily as the one dimensional case. The general form of the analytic solution with no initial velocity is

$$U(x,y,t) = \sum_{m=1}^{\infty}\sum_{n=1}^{\infty} B_{mn}\cos\lambda_{mn}t\left(\sin\frac{m\pi x}{a}\sin\frac{n\pi y}{b}\right)$$

Where B_{nm} is given by

$$B_{mn} = \frac{4}{ab}\int_0^a\int_0^b f(x,y)\sin\frac{m\pi x}{a}\sin\frac{n\pi y}{b}\,dxdy$$

In the question, we are given $f(x,y) = x(2-x)(2-y)$y, a = b = 2, and c = $3^{1/3}$

We must first determine the values of the coefficients B_{mn}. Substituting the given function and integrating gives

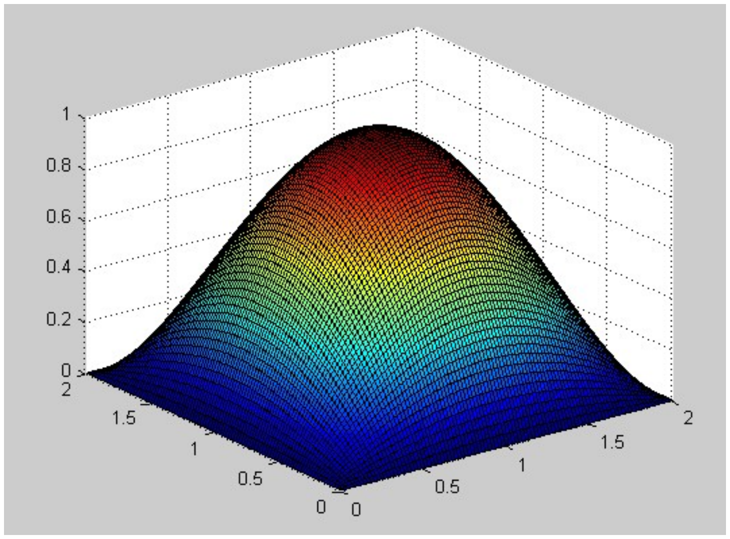
$$B_{mn} = \frac{16(1+(-1)^{m+1})}{\pi^3 m^3} * \frac{16(1+(-1)^{n+1})}{\pi^3 n^3}$$

This is simplified by the fact that B_{mn} = 0 if m or n is odd. With a = b = 2, λ_{mn} is given by

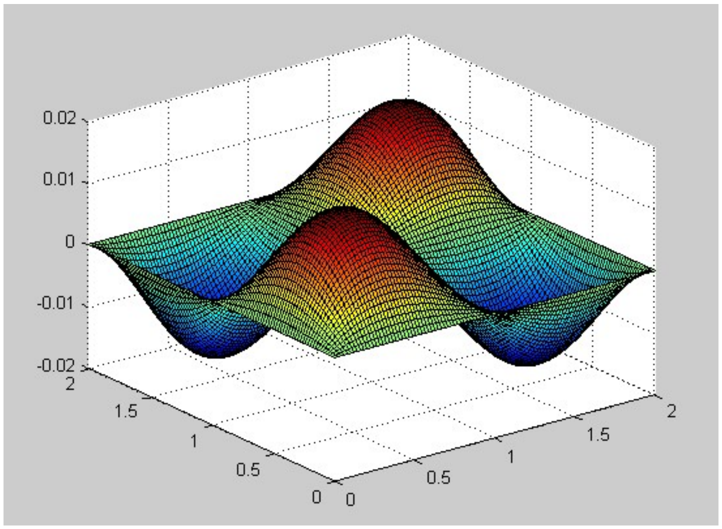
$$\lambda_{mn} = \sqrt{3}\pi\sqrt{\frac{m^2+n^2}{4}}$$

Since it is difficult to find a compact selection of points that vividly describes what is going on, and there is no specific question asked about the equations, here are graphs for the analytic solution at several values of t.

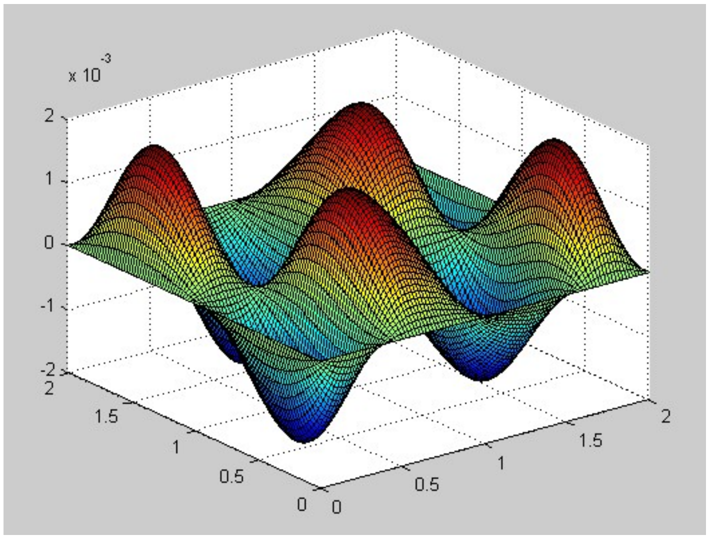
t = 0



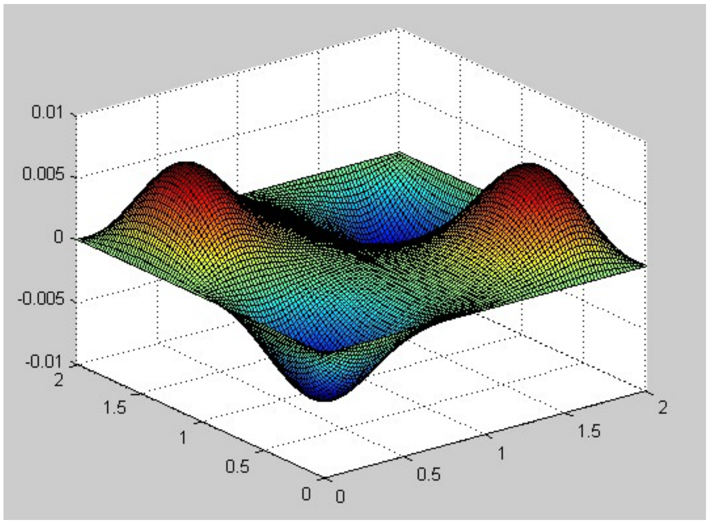
t = .003


0.02
0.01
0
-0.01
-0.02
2
1.5
1
0.5
0
0
0.5
1
1.5
2

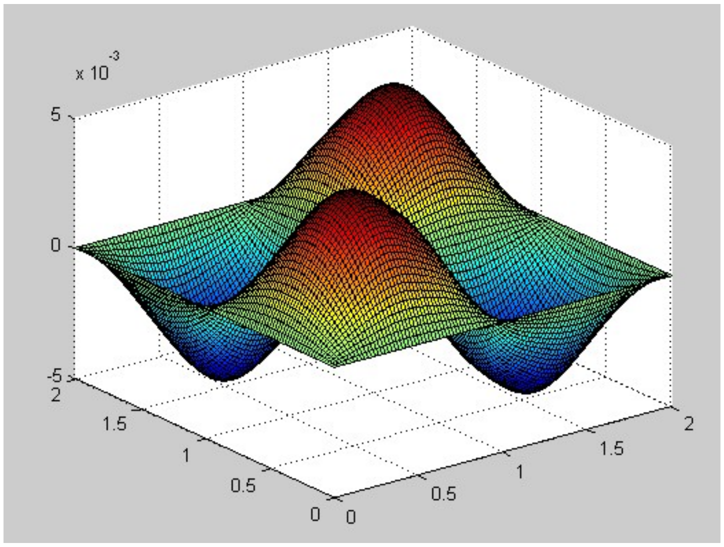
t = .2


x 10^-3
2
1
0
-1
-2
2
1.5
1
0.5
0
0
0.5
1
1.5
2

t = .25


0.01
0.005
0
-0.005
-0.01
2
1.5
1
0.5
0
0
0.5
1
1.5
2

t = .65



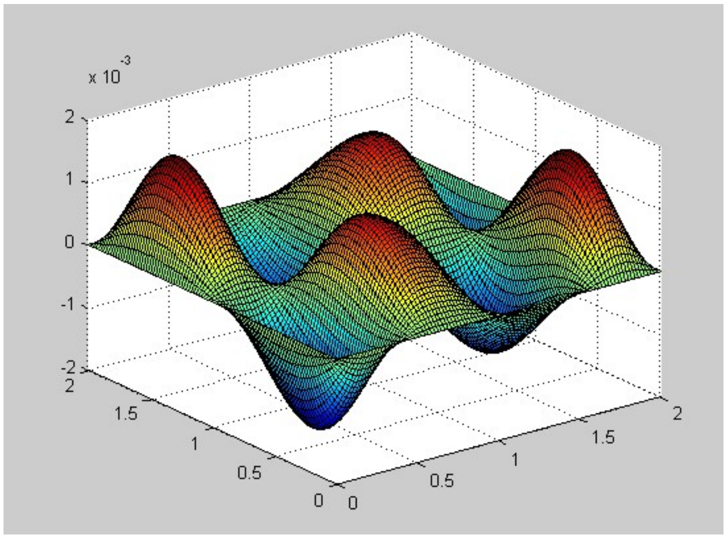
It is clear from the graphs that the period of the function is no less than .65 seconds. Since there is not a real world object to compare this membrane to, one cannot know whether this is a reasonable length of time. That being said, I have spent much time trying to uncover a smaller period with no success. If a smaller period does exist, it is less than .0001 seconds. The function I created to analyze this problem is extremely detailed. It examines 100 points in each the x and y direction for 10000 total mesh points. This is to say that it is unlikely that a shift a radical as a change in period would slip through unnoticed. Unfortunately, this is very slow due to the computation of the series through 10 on 10000 points every time step. Unlike, the one dimensional case, the numerical code has certain advantages.

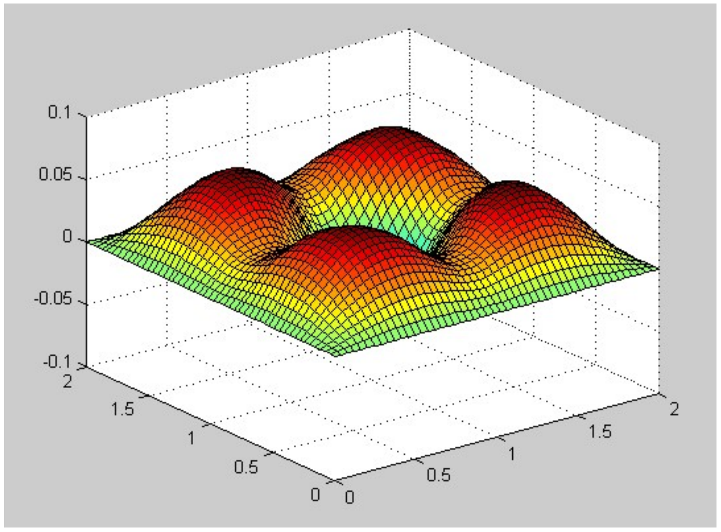
Problem 2: Solve the Problem Using Finite Differences

Once again, there is no 'problem' so solve per se, but creating a function which accurately captures the nature of the function at a higher speed is a challenge in and of itself. We begin with a simple example of 9 interior nodes. Just as with the one dimensional equation, the system of linear equations has been reduced to an iterative process for speed. Here are the results for the first two seconds:

t	(.5,.5)	(1,.5)	(1.5,.5)	(.5,1)	(1,1)	(1.5,1)	(.5,1.5)	(1,1.5)	(1.5,1.5)
0.0000	0.5625	0.7500	0.5625	0.7500	1.0000	0.7500	0.5625	0.7500	0.5625
0.1000	0.5175	0.6975	0.5625	0.6975	0.9400	0.7500	0.5625	0.7500	0.5625
0.2000	0.4365	0.6051	0.5625	0.6051	0.8362	0.7500	0.5625	0.7500	0.5625
0.3000	0.2912	0.4425	0.5625	0.4425	0.6562	0.7500	0.5625	0.7500	0.5625
0.4000	0.1123	0.2487	0.5625	0.2487	0.4475	0.7500	0.5625	0.7500	0.5625
0.5000	-0.0608	0.0702	0.5625	0.0702	0.2636	0.7500	0.5625	0.7500	0.5625
0.6000	-0.1879	-0.0502	0.5625	-0.0502	0.1500	0.7500	0.5625	0.7500	0.5625
0.7000	-0.2368	-0.0835	0.5625	-0.0835	0.1324	0.7500	0.5625	0.7500	0.5625
0.8000	-0.1921	-0.0217	0.5625	-0.0217	0.2112	0.7500	0.5625	0.7500	0.5625
0.9000	-0.0604	0.1202	0.5625	0.1202	0.3634	0.7500	0.5625	0.7500	0.5625
1.0000	0.1291	0.3083	0.5625	0.3083	0.5500	0.7500	0.5625	0.7500	0.5625
1.1000	0.3307	0.4974	0.5625	0.4974	0.7266	0.7500	0.5625	0.7500	0.5625
1.2000	0.4929	0.6421	0.5625	0.6421	0.8538	0.7500	0.5625	0.7500	0.5625
1.3000	0.5727	0.7077	0.5625	0.7077	0.9053	0.7500	0.5625	0.7500	0.5625
1.4000	0.5474	0.6785	0.5625	0.6785	0.8721	0.7500	0.5625	0.7500	0.5625
1.5000	0.4222	0.5614	0.5625	0.5614	0.7631	0.7500	0.5625	0.7500	0.5625
1.6000	0.2291	0.3846	0.5625	0.3846	0.6026	0.7500	0.5625	0.7500	0.5625
1.7000	0.0183	0.1904	0.5625	0.1904	0.4251	0.7500	0.5625	0.7500	0.5625
1.8000	-0.1555	0.0256	0.5625	0.0256	0.2692	0.7500	0.5625	0.7500	0.5625
1.9000	-0.2486	-0.0704	0.5625	-0.0704	0.1703	0.7500	0.5625	0.7500	0.5625
2.0000	-0.2392	-0.0745	0.5625	-0.0745	0.1528	0.7500	0.5625	0.7500	0.5625

This can be expanded into a more robust explicit method with various number of nodes. Like the 9 node program, this code forgoes matrices for point wise iteration. The results are similar to the Crank-Nicolson formula in the one dimensional case in the sense that they it provides a good qualitative assessment without being exact. For instance, consider both graphs as t = 10.





Clearly, the method (or its implementation) is flawed. However, it has captured some of the main features of the true solution. There are four peaks. They are positioned towards the corners. The middle is below 0 on both. All the sides are set at 0.

Unfortunately, I was not able to set up a functioning program for either the implicit or Crank-Nicolson methods. For the implicit method, there was no functioning 1 dimensional method to modify. I attempted to reverse engineer the explicit method formula, but this failed due to an inability to separate ΔU into x and y terms. Likewise for the Crank-Nicolson method. If this was the heat equation, the x and y components could be solved separately. However, this is not the case of the membrane.

Conclusion

I experienced much success with the analytical solutions. By performing iterations with small intervals, the programs can reduce errors down to the limits of MATLAB, so long as the k/h ratio is kept manageable. Likewise, I was able to create explicit schemes which provided which accuracy while increasing the speed of the program. In particular, the explicit 1 dimensional code can evaluate thousands of data points in less than a second. For the 1 dimensional equation, I created a Crank-Nicolson program which maintains the periodicity of the code despite not being very accurate.

However, I was unable to create a functioning implicit function for either the 1 or 2 dimensional case. I created 2 programs in the 1 dimensional case which failed in very different ways. In the failures, I realized how much more complex the wave equation is when compared to the head equation, despite having only a partial derivative of difference. The dependence of terms from 2 iterations ago made it very difficult to code. Having failed to complete the codes, I now see how small changes such as these create large differences in the code.

Note on codes

The codes are included in the order that they were introduced in the report. If a function was called in another function, I have included it first. Failed codes have also been included.