ECE 204 Project 5

Douglas Harder

March 27, 2024

1 Introduction

In this topic, we will approximate the solution to Laplace's equation in three dimensions using MATLAB. Laplace's equation finds the steady state solution assuming that the boundary values remain constant or insulated.

Laplace's equation says that the value of the field at every point of the interior must be the average of the non-insulated values surrounding that point, for if it is not, then there would be a force pressuring the value to move towards that average. This, of course, would occur simultaneously at each interior point.

For example, if $u(x, y)$ is the height of water in a pool (with the boundaries being insulated), then the steady state would be $u(x, y) = 0$, where 0 is assumed to be the surface of the water when the water is at rest. If there is any wave (where $u(x_0, y_0) > u(x, y)$ for all points in the immediate vicinity of (x_0, y_0), then the wave will either continue will move or spread out, but the wave will not remain stationary.

Similarly, if $u(x, y)$ is the temperature of a of a metal plate with all four sides in contact with surfaces that are at an ambient temperature, then the steady state would be that ambient temperature, and if one point on that metal plate was heated, then once the source of heat is removed, then the temperature would spread throughout the plate returning to the ambient temperature.

To find an approximation to a solution to Laplace's equation, we require a grid of points, some of which are boundary points, and others that are interior points for which we must approximate a solution. We will do this by passing a three-dimensional array where each entry of the array is either:

1. A fixed boundary value, represented by a finite real value.
2. An insulated boundary value, represented by `nan`.
3. An interior point, represented by $-\infty$ (or `-inf`).

Assuming that there are n interior points, we will create a system of n linear equations in n unknowns, the unknowns being the values of the approximations to the solution to Laplace's equation at those interior points. When we have found this approximation, we will then overwrite the `-inf` with the corresponding approximation.

Recall that Laplace's equation says that a real-valued function $u(x, y, z)$ is a solution to Laplace's equation if the value of the function at each interior point (x, y, z) is the average of all the values around it. For our finite-difference approximation of Laplace's equation, this says that, if there are no insulated boundaries in the vicinity, then

$$u(x,y,z) = \frac{u(x+h,y,z) + u(x-h,y,z) + u(x,y+h,z) + u(x,y-h,z) + u(x,y,z+h) + u(x+h,y,z-h)}{6},$$

although it is easier to write this equation as

$$6u(x,y,z) - u(x+h,y,z) - u(x-h,y,z) - u(x,y+h,z) - u(x,y-h,z) - u(x,y,z+h) - u(x+h,y,z-h) = 0.$$

If the point $(x, y+h, z)$ is a boundary value with a value of 5, then this term becomes a constant and is thus moved from the left-hand side to the right:

$$6u(x,y,z) - u(x+h,y,z) - u(x-h,y,z) - u(x,y-h,z) - u(x,y,z+h) - u(x,y,z-h) = 5.$$

If the point $(x, y, z - h)$ is an insulated boundary value, it does not count towards the average, so instead, we only take the average of the five surrounding non-insulated values, so we now have

$$5u(x, y, z) - u(x + h, y, z) - u(x - h, y, z) - u(x, y - h, z) - u(x, y, z + h) = 5.$$

Supposing that the remaining five values are interior points, and therefore unknown. In this case, each of them will be represented by an unknown variable, so for example:

- $u_{54} \approx u(x, y, z)$
- $u_{55} \approx u(x + h, y, z)$
- $u_{53} \approx u(x - h, y, z)$
- $u_{24} \approx u(x, y - h, z)$
- $u_{103} \approx u(x, y, z + h)$

Thus, our linear equation is now:

$$5u_{54} - u_{55} - u_{53} - u_{24} - u_{103} = 5.$$

This will be the 54th linear equation, and thus Row 54 will have column 54 set to 5, and columns 24, 53, 55 and 103 set to -1, while entry 54 of the target vector will be 5. Once we solve for u_{54}, then the entry in the matrix corresponding to (x, y, z) will be overwritten with this value.

2 The algorithm

We will walk through a number of algorithms to make this possible; you are not responsible to determine the algorithms necessary to create the matrix and vector that define this system of n linear equations in n unknowns. Instead, we will walk you through the process, and hopefully you will learn something about Matlab and interpreted languages (including, for example, Python).

The function we will author has the signature:

```
function [U] = laplace3d( U )
  % ...
end
```

This must be saved in a file called `laplace3d.m`. In Matlab, there is no way to pass a matrix by reference: when a matrix is passed, it is passed by value, meaning the entire matrix is copied. In the function, we will make changes to the parameter `U`, and that matrix will be returned.

2.1 Ensuring the array is three-dimensional

First, the argument `U` must be three-dimensional, which we can check using the `ndims( U )` command, which returns a positive integer. If this value is not equal to 3, then you should throw an exception:

```
    throw( MException( ...
      MATLAB:invalid_argument', ...
      'expecting a 3-dimensional array, but got one that was %d dimensional', ...
      ndims( U ) ...
    ) );
```

Remember: in Matlab, instead of using `!=`, you need to use `~=`.

If you are using Octave, you will have to replace this conditional statement with an assertion:

```
assert( ndims( U ) == 3, 'MATLAB:invalid_argument', ...
  'expecting a 3-dimensional array, but got one that was %d dimensional', ...
  ndims( U ) ...
);
```

As in C++, exceptions can be thrown and caught, while assertions simply terminate the execution of the program. Octave, being an open-source version of Matlab, does not have exceptions implemented as of yet.

2.2 Determining the capacities of the three dimensions of the array

Next, we will have to extract the capacity of the three dimensions. This can be done with a simple assignment statement:

```
[Nx, Ny, Nz] = size( U );
```

The function `size(...)` returns a vector with as many entries as there are dimensions, so because there are three dimensions, we can either assign the output to a single variable and access the capacities

```
Ns = size( U )     % Access the entries with Nvec(1), Nvec(2), Nvec(3)
     Nvec = 3   5   4

[Nx, Ny, Nz] = size( U );
     Nx = 3
     Ny = 5
     Nz = 4
```

For this project, it will be easier to access `Nx`, `Ny` and `Nz` individually, as it will make the code more comprehensible to the reader.
Remember: if you want to see the result of any one statement while executing a function, you simply have to remove the semicolon at the end of that line. Then, when you run the function, the result of executing that line will be printed to the screen.

2.3 Checking the edges

To approximate a solution to Laplace's equation, we cannot have any unknown values on the boundary. In this case, the boundary are the size sides of the cube defined by the three-dimensional array. The slow way of performing this check is to use a for loop:

```
% Make sure that any entry (1, j, k) or (Nx, j, k)
% is not equal to -inf
for j = 1:Ny
    for k = 1:Nz
        if U(1,j,k) == -inf
            % Throw an exception like described above,
        end

        if U(end,j,k) == -inf
            % Throw an exception like described above,
        end
        % Note that you could also check
        %     if U(Nx,j,k) == -inf
        % but 'end' is clearer: it explicitly tells Matlab
        % to access the last entry in that dimension,
        % and the reader also understands immediately
        % what is meant.
```

```
            % If you are in Octave, you must replace these
            % two with an assertion:
            %   assert( (U(1,j,k) ~= -inf) & (U(end,j,k) ~= -inf), 'error-message-here...' );
        end
    end
```

You would have to do this also for the four other sides:

```
    for i = 1:Nx
        for k = 1:Nz
            % Check U(i,1,k) and U(i,end,k)
        end
    end

    for i = 1:Nx
        for j = 1:Ny
            % Check U(i,j,1) and U(i,j,end)
        end
    end
```

This is what you might do in C++, but Matlab is an interpreted language, so each loop requires that the statements are first interpreted before they are executed.

Instead, observe the following:

```
    A = [ 1  2  3  4  5  6
          7  8  9 10 11 12
         13 14 15 16 17 18];

    A(1,:)   % all entries in Row 1
        ans = 1 2 3 4 5 6

    A(end,:)   % all entries in Row 3
        ans = 13 14 15 16 17 18

    A(:,1)    % all entries in Column 1
        ans =  1
               7
              13

    A(:,end)  % all entries in Column 6
        ans =  6
              12
              18
```

The colon (:) says to access everything in that index. We can do even more:

```
    A = [ 1  2  3  4  5  6
          7  8  9 10 11 12
         13 14 15 16 17 18];

    A([1,end],:)   % all entries in Rows 1 and 3
        ans =  1  2  3  4  5  6
              13 14 15 16 17 18
    A(:,[1,end])  % all entries in Columns 1 and 6
        ans =  1   6
               7  12
              13  18
```

Next, we can now perform some sort of comparison operation on each of these entries:

```
    A(:,[1,end]) <= 10  % all entries in Columns 1 and 6
        ans =  1   1
               1   0
               0   0
```

You will see that all entries less than or equal to 10 now 1, and all other entries are 0. You can now ask are any of the entries equal to 1 (that is, `true`)?

```
    any( A(:,[1,end]) <= 10 )
        ans = 1  1
```

The result is a row vector, where each says that there was at least one entry that was 1 in that column. We can ask if any of those entries are 1 by calling `any(...)` again. To see this work:

```
    A = randn( 3, 3 )
        A =
           0.157384  -0.047897   0.573332
           0.179019   1.245516  -0.930445
           1.034134   0.497195   0.307113

    any( A < 0.0 )   % are any entries negative in each column?
        ans =  0  1  1

    any( any( A < 0.0 ) )   % are any negative entries?
        ans = 1

    any( A < -1.0 )  % are any entries less than -1? (No)
        ans =  0  0  0

    any( any( A < -1.0 ) ) % Are any entries in the matrix less than -1?
        ans = 0
```

Thus, we can check:

```
    % Are any of the entries at either ends of the cube
    % in the first dimension equal to -inf?
    if any( any( any( U([1,end],:,:) == -inf ) ) )
        % throw an exception
    end

    % Are any of the entries at either ends of the cube
    % in the second dimension equal to -inf?
    if any( any( any( U(:,[1,end],:) == -inf ) ) )
        % throw an exception
    end

    % Repeat for the third dimension...
    % Are any of the entries at either ends of the cube
    % in the third dimension equal to -inf?
```

Again, if you are using Octave, you must use an assertion instead of these conditional statements; for example:

```
assert( ~any( any( any( U([1,end],:,:) == -inf ) ) ), ...
     'some appropriate error message...' );
```

2.4 Counting the number of -inf

Next, we need to count how many of the entries are equal to `-inf`, as this is the number of unknown variables. You could, if you wish, perform a for-loop:

```
u_count = 0;

for i = 1:Nx
    for j = 1:Ny
        for k = 1:Nz
            if U(i,j,k) == -inf
                % Matlab does not have ++u_count
                u_count = u_count + 1;
            end
        end
    end
end
```

Instead, you could try the following:

```
u_count = sum( sum( sum( U == -inf ) ) );
```

The comparison `U == -inf` returns a matrix that is either 0 or 1, with a 1 each time the condition is true. You can try this as follows:

```
V = randn( 4, 4 )
    V =
      -0.541873  -0.536664   0.780389  -0.144424
       0.026593   0.052658  -0.959457  -1.730366
      -0.101185   0.964140  -2.086544   1.267127
      -1.010140  -1.022619   2.039456  -0.255638

Vpos = (V > 0.0)
    Vpos =
      0  0  1  0
      1  1  0  0
      0  1  0  1
      0  0  1  0

sum( Vpos )
    ans =
       1   2   2   1

u_count = sum( sum( Vpos ) )
    u_count =  6
```

The last command tells you how many entries in `V` were positive. Because `U` is a three-dimensional array, you must call `sum(...)` three times:

```
u_count = sum( sum( sum( U == -inf ) ) );
```

Because Matlab is an interpreted language, the for loop must be repeatedly interpreted, so this is exceptionally slow. If the three-dimensional array `U` has one million entries, the for loop takes 15 seconds to execute, while the single statement using built-in functions executes in 10 milliseconds; a factor of 1500 faster. It is always important in an interpreted language to use built-in functions when possible.

2.5 Enumerating the unknowns

Each `-inf` represents an interior point, and therefore a separate unknown. Thus, we must associate with each `-inf` a unique integer from `1` to `u_count`. For example, if we were approximating a solution to Laplace's equation in two dimensions, then if the 2-dimensional array `U` was the following,

```
U = [1    2    2    2    2  3
     0 -inf -inf -inf -inf  4
     0 -inf -inf -inf -inf  4
     0 -inf    6 -inf -inf  4
     2    4    4    4    4  4]
```

we would want to associate each interior value with a unique unknown, so

$$\begin{pmatrix} - & - & - & - & - & - \\ - & u_1 & u_2 & u_3 & u_4 & - \\ - & u_5 & u_6 & u_7 & u_8 & - \\ - & u_9 & - & u_{10} & u_{11} & - \\ - & - & - & - & - & - \end{pmatrix}$$

. Now, given that we have 11 unknowns, We have two goals:

1. Given a position with `-inf`, we want to associate each entry with a unique integer from 1 to 11 and be able to access that integer in $O(1)$ time. For example, in the above 2-dimensional array, we want to associate $(3, 4)$ with the index 7.

2. Given one of the eleven integers, determine the entry in the matrix corresponding to that integer, and we want to do this in $O(1)$ time. For example, in the above 2-dimensional array, we would want to associate the index 10 with $(4, 4)$.

We will do this as follows: a 2-dimensional array `To_index` will be initialized as one being of the same size, but of all zeros:

```
To_index = zeros( 5, 6 );
```

We also want to go from the integer from 1 to 11 to indicate where that entry is in `U`. For this, we will create an 11×2 2-dimensional array where each column contains the indices of the corresponding column number:

```
To_index = zeros( size( U ) ); % Create a 5 x 6 2-dimensional array
From_index = zeros( 11, 2 );   % Create a 11 x 2 2-dimensional array

index = 0;

for i = 1:5
    for j = 1:6
        if U(i, j) == -inf
            index = index + 1;
            To_index(i, j) = index;
            From_index( index, : ) = [i j];
        end
    end
end
```

Upon executing this, these two arrays now look like the following:

```
    To_index =
        0  0  0  0  0  0
        0  1  2  3  4  0
        0  5  6  7  8  0
        0  9  0 10 11  0
        0  0  0  0  0  0

    From_index =
        2  2
        2  3
        2  4
        2  5
        3  2
        3  3
        3  4
        3  5
        4  2
        4  4
        4  5
```

Thus, because `To_index( 3, 5 )` is not zero, that means it is an unknown, and because `To_index( 3, 5 ) == 8`, this means we are associating with this unknown the index 8, so the unknown u_8. Therefore, `From_index( 8, : )` must contain the entries `3  5`.

You will need to do this in three dimensions, though:

```
    To_index = zeros( Nx, Ny, Nz );
    From_index = zeros( u_count, 3 );   % Create a 'u_count' x 3 2-dimensional array

    index = 0;

    for i = 1:Nx
        for j = 1:Ny
            for k = 1:Nz
                if U(i, j, k) == -inf
                    index = index + 1;
                    To_index(i, j, k) = index;
                    From_index( index, : ) = [i j k];
                end
            end
        end
    end

    assert( index == u_count );
```

2.6 Creating the sparse matrix and target vector

We will create and solve $A\mathbf{u} = \mathbf{b}$ for the unknown vector $\mathbf{u}$.

We now must create a `u_count` $\times$ `u_count` matrix A. We now refer to this as a matrix and not a two-dimensional array because it is indeed used to store a system of linear equations. We will also create a `u_count`-dimensional vector `b`:

```
    % Allocate a u_count x u_count sparse matrix
    A = spalloc( u_count, u_count, 7*u_count );
```

```
% Allocate a u_count-dimensional array
b = zeros( u_count, 1 );
```

The third argument ensures enough memory is allocated for `7×u_count` entries, which is the maximum number of non-zero entries we could have.

2.7 Populating the matrix and vector

Our next goal is to correctly initialize each of the entries of the matrix, so for each unknown `1` through `u_count`, we need to create the corresponding linear equation based on the six surrounding entries:

```
for ell = 1:u_count
    % Determine the location (i, j, k) of the
    % entry u    using the 'From_index' 2-d array
    %       ell
    v = From_index( ell, : );
    i = v(1);
    j = v(2);
    k = v(3);

    % Determine the six surrounding entries to
    % update both 'A' and 'b'
    %  - follow the description below...
end
```

Once you determine that u_ℓ is associated with the position (i, j, k), we will now check all six neighboring entries:

$$(i+1, j, k), (i-1, j, k), (i, j+1, k), (i, j-1, k), (i, j, k+1), (i, j, k-1).$$

Each of these will be either a boundary condition (fixed or insulated) or associated with another unknown variable. We will go through an example. Let us assume that index $\ell = 17$ is associated with the location $(5, 7, 3)$ in the 3-dimensional array U. Thus, we must check

$$(4, 7, 3), (6, 7, 3), (5, 6, 3), (5, 8, 3), (5, 7, 2), (5, 7, 4).$$

We will update the matrix A or the vector $\mathbf{b}$ depending on whether the entry is an unknown or a fixed boundary value, respectively, and we do nothing if the entry is an insulated boundary value. So for example, while $(5, 7, 3)$ is may be associated with index 17, when we check $(5, 6, 3)$, and if `U(5, 6, 3) == -inf`, this may be associated with index 39. We will then proceed as follows:

1. If `U(5, 6, 3) == -inf`, then we will increment (add one to) the value at `A(17, 17)`, and we will decrement (subtract one from) the value at `A(17, 39)`.
2. If `U(5, 6, 3)` is a scalar (`isfinite(U(5,6,3))`), then we will increment (add one to) the value at `A(17, 17)` and add the value at `U(5,6,3)` to `b(17)`.
3. Finally, if `U(5, 6, 3)` is not-a-number (`isnan(U(5,6,3))`), then we do nothing: the insulated boundary does not affect the solution at $(5, 6, 3)$.

If, when this loop is finished, you want to see the matrix `A` or the vector `b`, all you have to do is have a single line in your code:

```
[A b]
```

without a semi-colon at the end, and this will print the augmented matrix. To remove this output, you can either put a semicolon at the end of the line or just comment it out. This is much easier than C++ where you must place an explicit `std::cout` statement.

2.8 Solve the matrix and substitute back the entries

We must now solve the system of linear equations:

```
u = A \ b;
```

We next must substitute these values back into U:

```
for ell = 1:u_count
    % Determine the location (i, j, k) associated with u
    %                                                   ell

    % Let us make sure it is an unknown entry
    assert( U( i, j, k ) == -inf );

    % Overwrite the -inf at that index with
    % the value stored in u(ell).

    % Finally, check that the entry is finite:
    assert( U( i, j, k ) ~= -inf );
end
```

2.9 Final check

We should make sure that there are no more entries in U that are equal to -inf:

```
assert( ~any( any( any( U == -inf ) ) ) );
```

If any of the entries are -inf, this assertion will fail, because one of the entries will be true (1):

```
A = zeros( 10, 32, 15 );
any( any( any( A ) ) )
    ans = 0
A( 8, 21, 14 ) = 1;  % Set one and only one entry to 'true' (1)
any( any( any( A ) ) )
    ans = 1
```

Alternatively, you could equivalently check:

```
% Check that all entries are not -inf
assert( all( all( all( U ~= -inf ) ) ) );
```

There will be examples on Crowdmark to test your code.

3 Sparse matrices

We will use both dense and sparse matrices. To describe the problem, we will use dense three-dimensional arrays. However, from this, we will need to generate a system of n linear equations in n unknowns, where n could be 100 or 1000 or even greater. With 1000 unknowns, this would theoretically require a matrix with one million entries, but at most seven entries per row will be non-zero, so less than 1% of the entries of the matrix will be non-zero. If you were to try to use a dense matrix, this would require billions of floating-point operations, most of which would be adding a multiple of zero onto an already existing zero.

To create a dense matrix of all zeros, use the command

```
A = zeros( m, n ); % 'm' and 'n' are positive integers.
```

To create a sparse matrix, use the command

```
B = sparse( m, n ); % 'm' and 'n' are positive integers.
```

Accessing or assigning to these matrices is identical, and a system of linear equations can be solved using the \ operator.

For example, the following creates a 100×100 matrix with entries 2 on the diagonal, 1 on both sub-diagonals, and zero everywhere else:

```
A = sparse( 100, 100 );
A = A + spdiags( 2.0*ones( 100, 1 ),  0, 100, 100 );
A = A + spdiags(     ones( 100, 1 ),  1, 100, 100 );
A = A + spdiags(     ones( 100, 1 ), -1, 100, 100 );
A \ (1:100)'
```

If you examine the solution, it is very close to the integer vector

$$(0, 1, 0, 2, 0, 3, 0, 4, 0, 5, \ldots, 0, 49, 0, 50),$$

which is the correct solution.