Notes on simulating the vibrating membrane

The vibrating membrane is an extension of the vibrating string; instead of a one-dimensional object vibrating in a second dimension, you will simulate a two-dimensional object vibrating in three dimensions.

The things you'll need to address are:

- Dealing with three dimensions: physics
- Dealing with three dimensions: animation
- Dealing with a two-dimensional mesh: coding challenges
- Dealing with a two-dimensional mesh: physics
- Initial conditions for the normal modes

1 Three dimensions: physics

Since you now have a collection of masses moving in three dimensions rather than two, you will need variables for the z-component of their positions and velocities.

Computing the force in the z-direction is done in precisely the same way as in the x- and y-directions; you will likely even use some cut-and-paste magic to write your code.

You will, however, need to add the z-component into the computation of the separation between points. Previously you used the Pythagorean theorem to write the separation between point i and point j as

$$r_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2};$$

in three dimensions, you can just use the three-dimensional analog of the Pythagorean theorem

$$r_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}.$$

2 Three dimensions: animation

You will need `anim` commands for drawing in three dimensions. They are:

- `l3` $x_1 y_1 z_1 x_2 y_2 z_2$ – draw a line from (x_1, y_1, z_1) to (x_2, y_2, z_2)
- `c3` $x_1 y_1 z_1 r$ – draw a circle at (x_1, y_1, z_1) with radius r

While `anim` is running, you can manipulate the viewport in a few new ways. The keys q, a, w, s, e, and d rotate your view along all three axes. Mouse control works as it always has. You can also use shift-minus and shift-equals to control the angle of view while keeping the size constant; minus and equals zoom in and out as before.

You will need to think about how to ensure your mesh gets completely drawn without drawing links outside of the square; I find this easiest with something like

```
for (i=0; i<=N; i++)
{
  for (j=0; j<=N; j++)
  {
    if (i>0) (draw from i,j to i-1,j)
    if (i<N) (draw from i,j to i+1,j)
    if (j>0) (draw from i,j to i,j-1)
    if (j<N) (draw from i,j to i,j+1)
  }
}
```

3 2D mesh: coding challenges

Since your membrane is a $N \times N$ (or $N+1 \times N+1$, depending on how you define N) two-dimensional mesh, you need to figure out how to store the six sets of N^2 or $(N+1)^2$ that are your dynamical variables.

There are two ways to do this: two-dimensional arrays or index calculation.

3.1 Two-dimensional arrays

You can make an array of arrays in C like this:

```
double x[N][N];
```

This does exactly what you want. You can then iterate over these elements with a nested `for` loop:

```
for (i=0; i<N; i++)
{
  for (j=0; j<N; j++)
  {
    (do whatever you wanted to do to each element, like an RK2 substep)
  }
}
```

This seems pretty simple, but there is one gotcha: **you cannot pass two-dimensional arrays to a function unless you specify the size explicitly, not as a variable, in the function header**.

This is legal:

```
double force_magnitude (double x[40][40], double y[40][40], double z[40][40])
```

This is not:

```
double force_magnitude (double x[N][N], double y[N][N], double z[N][N])
```

There *is* a hackish way to get around this that still lets you change N pretty easily. If you write

```
#define N 40
```

at the top of your code, then every occurrence of `N` in your program will be changed to `40`. (Note that there is no equals sign.) This is still legal, since this sort of "cut and paste" redefinition happens before the program is compiled; `#define` and `#include` are called *preprocessor directives*, since they happen before compilation.

So writing

```
double force_magnitude (double x[N][N], double y[N][N], double z[N][N])
```

is legal if you have the `#define N 40` line at the top of your code, since in that case `N` isn't a variable that contains the number 40; it actually gets replaced with the text `40`.

If you do all of your math inside `main` without needing to pass a two-dimensional array through a function call then this isn't a problem.

3.2 Index calculation

A way to avoid this is to just use one-dimensional arrays for everything and do a little mathematics to figure out the index that corresponds to each point in the mesh.

You want an array with N^2 points, so you declare it in the usual way:

```
double x[N*N];
```

Then, however, you will need to do a quick bit of algebra every time you need to access one of them. If you number them consecutively, going along rows one at a time, you might do something resembling this:

```
for (i=0; i<N; i++)
{
  for (j=0; j<N; j++)
  {
    x[i*N+j] = x[i*N+j] + vx[i*N+j] * dt;
  }
}
```

This is not much more complicated than the other method; whenever you'd write `x[i][j]` you just write `x[i*N+j]` instead. This is, in my opinion, a small price to pay for not having to mess with the limitations on passing multidimensional arrays to functions.

4 2D mesh: physics

4.1 Dynamics

The only real difference in the dynamics comes from the geometry of the mesh itself: each mass is connected to four neighbors rather than two, so you have four forces to evaluate.

A node at position (i, j) in the mesh feels forces from:

- Its neighbor to the left, at $(i-1, j)$
- Its neighbor to the right, at $(i+1, j)$
- Its neighbor above it, at $(i, j-1)$
- Its neighbor below it, at $(i, j+1)$

4.2 Initial conditions

You will once again want to stretch your membrane; remember that you had to put some tension on your string for it to vibrate in the expected way, and the membrane is no different.

You'll need to assign initial x, y, and z positions and velocities to each point. This is done, like everything else, with a nested `for` loop.

I'll let you figure out the x and y coordinates on your own. The normal modes are products of sine waves in the x and y directions:

$$z(x, y) = A \sin\left(\frac{n_x \pi x}{L}\right) \sin\left(\frac{n_y \pi y}{L}\right)$$

4.3 Deriving the microscopic quantities from the macroscopic ones

The macroscopic quantities of the membrane are:

- Thickness T
- Young's modulus E
- Equilibrium length of a side L
- Mass density σ

As before, you have to derive your microscopic quantities (that your simulation is concerned with) from the number of masses on a side N and these macroscopic quantities. These are:

- Equilibrium spring length r_0 (easy)

- Mass m (easy)
- Single-spring spring constant k' (hard)

The equilibrium length is just L/N or $L/(N-1)$ as before, depending on how you define N.

The mass is also not bad; your whole membrane has a mass σL^2, so each of the N^2 masses has a mass $\sigma L^2/N^2$.

The spring constant is tricky. We'll figure it out by imagining that we stretch the membrane by a distance Δx along one axis and calculating the restoring force using both macroscopic and microscopic pictures, and figuring out what k' must be to make them match.

Using our macroscopic quantities, we treat the membrane as a "spring" extending in one direction, and calculate its spring constant from the definition of Young's modulus. Suppose we stretch the membrane along the x-axis; the cross-sectional area is thus the thickness times the extent in the y-direction (which is just L). This gives us:

$$k = \frac{EA}{L} = \frac{ELT}{L} = ET$$

This is unexpected; the spring constant *doesn't depend on the length at all!* This cancellation didn't happen for the spring. We can again define a stiffness γ (I use a different letter because the dimensions are different for the membrane), but this isn't really needed, since it's just equal to the spring constant of the whole thing! (This is only true if it's square at equilibrium, since the cancellation of the factors of L involved both horizontal and vertical lengths).

$$\gamma = ET$$

Thus the restoring force is, by Hooke's law,

$$F = ET\Delta x.$$

In the microscopic picture, we imagine stretching a lattice of $N \times N$ springs by a distance Δx. Obviously we need only consider the springs in the x-direction.

Consider first a single row of springs, just as we did before. If the total stretch is Δx, then each one stretches by $\Delta x/N$. Then the force exerted on the endpoints is just

$$F_{1\,\mathrm{row}} = k'\Delta x/N$$

However, there are N rows, each exerting this much force. This means that the total restoring force is N times as much:

$$F = k'\Delta x.$$

As in the macroscopic case, we see that all the factors of N cancel!

This required a bit of thinking, but in the end we derive that $k' = ET \equiv \gamma$: rather than having to derive the microscopic spring constant from the stiffness, N, and L, in this case the stiffness *is* just the microscopic spring constant.

This means you can just choose k' directly and get the same physics regardless of N. Neat!