Finishing 1.2. Last time we talked about numerically solving ODEs by writing the associated linear system and solving it. A few more things about the system are worth saying (and even repeating).

What does the matrix in that system look like? Recall that the ODE is
\[ u''(x) + au'(x) + bu(x) = f(x), \text{ for every } x \in [0,1], \text{ and } u(0) = u(1) = 0. \]

We have constructed an equipartition of \([0,1]\) into \(m\) equal intervals of length \(h = 1/m\), and \(u_i\) is designed to approximate \(u(x_i) = u(i \cdot h);\) we write \(f_i = f(x_i)\). The \(i\)th equation of the linear system will look like this:
\[ \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + a \frac{u_{i+1} - u_{i-1}}{2h} + bu_i = f_i. \]

Note \(u_0 = u_m = 0\).

From the \(i\)th equation we see that the only non-zero coefficients are those of \(u_{i-1}, u_i,\) and \(u_{i+1}\). This means that the matrix is **banded**, that is, it only has non-zero entries in a band around the main diagonal (any system arising from a homogeneous, constant-coefficient ODE is going to be banded; in this case the band length is 3, as the system is tridiagonal). Rewrite the equation:
\[ \left( \frac{1}{h^2} - \frac{a}{h} \right) u_{i-1} + \left( -\frac{2}{h^2} + b \right) u_i + \left( \frac{1}{h^2} + \frac{a}{2h} \right) u_{i+1} = f_i; \]

now we see that the matrix looks like
\[
A_h = \begin{bmatrix}
\left( \frac{1}{h^2} - \frac{a}{h} \right) & \left( -\frac{2}{h^2} + b \right) & 0 & \cdots & 0 & 0 \\
\left( \frac{1}{h^2} - \frac{a}{h} \right) & \left( \frac{1}{h^2} + \frac{a}{2h} \right) & \left( \frac{1}{h^2} + \frac{a}{2h} \right) & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & \left( -\frac{2}{h^2} + b \right) & \left( \frac{1}{h^2} + \frac{a}{2h} \right)
\end{bmatrix},
\]

The rows of this matrix are independent, and a very well-known theorem from linear algebra tells us that the system will have exactly one solution for each right hand side.

**Theorem 1.** Let \(A\) be an \(n \times n\) matrix with real (or complex) entries. The following are equivalent:

1. \(A^{-1}\) exists (the matrix is invertible or non-singular);
2. there is no \(y \neq 0\) such that \(Ay = 0\) (meaning, if \(Ay = 0\) then \(y = 0\));
3. the columns of \(A\) are linearly independent;
4. the rows of \(A\) are linearly independent;
5. for any column vector \(b\) of length \(n\), there exists precisely one column vector \(y\) such that \(Ay = b\).
For Homework 1 you will have to set up, or to set up and solve, systems of linear equations (the former for an ODE, the latter for a mass-spring system, which you can also read more about in Section 1.2 of the textbook).

**Section 1.3.** We will now start talking about how one may approach solving linear systems; one of the simplest systems to solve is a triangular one, so called because the 0-structure of the matrix makes it look like a triangle (if all entries above the main diagonal are 0, we call it lower-triangular, and if all entries below the main diagonal are 0, we call it upper-triangular).

Let \( G y = b \) be an example of a lower-triangular system (the awful notation simply reflects the textbook notation); the system looks like

\[
\begin{align*}
g_{11} y_1 & = b_1 \\
g_{12} y_1 + g_{22} y_2 & = b_2 \\
& \quad \vdots \\
g_{n1} y_1 + g_{n2} y_2 + \ldots + g_{nn} y_n & = b_n
\end{align*}
\]

The obvious strategy is to do **forward substitution**; solve for \( y_1 \) from the first equation, substitute into the second equation and solve for \( y_2 \), then substitute the values you have found for \( y_1, y_2 \) in the third equation and solve for \( y_3 \), and continue this way down until you solved all the equations.

This relies on the following formula for \( y_i \), with \( i = 1, 2, \ldots, n \), which can be evaluated sequentially from 1 through \( n \), using the values we have already found:

\[
y_i = \frac{b_i - g_{i1} y_1 - g_{i2} y_2 - \cdots - g_{i(i-1)} y_{i-1}}{g_{ii}}.
\]

A simple way to code forward substitution into MATLAB can be seen below.

```matlab
function y = lowtriangsolve1(G,b);
y = b;
for i = 1 : n
    for j = 1 : (i - 1)
        y_i = y_i - g_ij * y_j;
    end
    if g_ii = 0, error('matrix is singular'), end
    y_i = y_i / g_ii;
end
```

**NOTE:** recall that the determinant of a triangular matrix is the product of the diagonal entries (Exercise: try to think why). So if the matrix is non-singular, none of the diagonal entries can be 0. Conversely, if any diagonal entry is 0, the matrix must be singular (and the system cannot be uniquely solved).

Often, once \( y \) is found, \( b \) is no longer needed and so it is overwritten. In the textbook, the pseudocode algorithms for triangular solve overwrite \( b \). This helps with space-saving if the matrices are huge, but it will not impact things much for the kinds of matrices we will be dealing with in this class.

**Flop count.** Consider the floating point operations (a.k.a. *flop*) (+, −, *, /) count for forward substitution. The inner loop effectuates 2 flops each time it is run, and it is run \((i - 1)\) times, for
a total of $2i - 2$ flops. The division at the end adds one more flop. So for each $i$ from 1 to $n$, the work done inside the first for loop is proportional to $2i - 1$.

The flop count is

$$\sum_{i=1}^{n} (2i - 1) = 2 \sum_{i=1}^{n} i - \sum_{i=1}^{n} 1 = n(n + 1) - n = n^2.$$ 

Each time we double $n$, the algorithm will take 4 times as long to run.

**Backward Substitution.** So far we have seen how to solve lower triangular systems, but what if the system is upper triangular, i.e., looks like

$$u_{11}x_1 + u_{12}x_2 + \ldots + u_{1n}x_n = b_1$$
$$u_{22}x_2 + \ldots + u_{2n}x_n = b_2$$
$$\vdots$$
$$u_{nn}x_n = b_n,$$

how does the strategy change? Answer: we start from the bottom, from $x_n$, and work our way backwards to $x_1$. All the rest is similar; the strategy is named *backward substitution*. You can work out the new algorithm on your own; it is very similar to the old one and you should expect to see the same flop count as before.