Problem 0.1. Watkins 1.1.9

Solution. When I ran the program, my ratios were 8.74, 4.02, and 4.29. Since matrix-vector multiplication is $O(n^2)$, I would expect doubling the matrix size to quadruple the computation time. Except for the first ratio, my numbers agree with that.

Problem 0.2. Watkins 1.1.10

Solution. With my code, I found that Octave’s built in multiplication algorithm was roughly 3800 times faster! (The Octave/Matlab languages are not meant to be super fast/optimized. They intend you program them at a high level, and let them take care of the nitty-gritty details of how to calculate matrix multiplications, decompositions, etc.)

Here is my code:

```plaintext
n=300;
A=randn(n);
x=randn(n,1);
t=cputime;
for rep = 1:100
    b=A*x;
end
their_time = cputime-t

t=cputime;
for rep = 1:10
    b=zeros(n,1);
    for j = 1:n
        for i = 1:n
            b(i)=b(i)+A(i,j)*x(j);
        end
    end
end
my_time = (cputime-t)*10
%times 10 since I did 10 times fewer repetitions
```
ratio = my_time/their_time

Problem 0.3. Watkins 1.2.4

Solution. Assume that \( A^{-1} \) exists and that \( Ay = 0 \). Then, \( A^{-1}Ay = A^{-1}0 \), which reduces to \( y = 0 \). Thus \( y \) must be zero, and so there can be no nonzero \( y \) such that \( Ay = 0 \). □

Problem 0.4. Watkins 1.2.16a

Solution. If \( m = 8 \), then \( h = 1/8 \), but otherwise the calculation is the same as in the book. The general equation reduces to

\[
(-64 + 4c)u_{i+1} + (128 + d)u_i + (-64 - 4c)u_{i-1} = f_i.
\]

This means the matrix problem is

\[
\begin{bmatrix}
128 + d & -64 + 4c & 0 & 0 & 0 & 0 & 0 \\
-64 - 4c & 128 + d & -64 + 4c & 0 & 0 & 0 & 0 \\
0 & -64 - 4c & 128 + d & -64 + 4c & 0 & 0 & 0 \\
0 & 0 & -64 - 4c & 128 + d & -64 + 4c & 0 & 0 \\
0 & 0 & 0 & -64 - 4c & 128 + d & -64 + 4c & 0 \\
0 & 0 & 0 & 0 & -64 - 4c & 128 + d & -64 + 4c \\
0 & 0 & 0 & 0 & 0 & -64 - 4c & 128 + d \\
0 & 0 & 0 & 0 & 0 & 0 & -64 - 4c & 128 + d
\end{bmatrix}
\begin{bmatrix}
u_1 \\
u_2 \\
u_3 \\
u_4 \\
u_5 \\
u_6 \\
u_7
\end{bmatrix}
= \begin{bmatrix}
f_1 \\
f_2 \\
f_3 \\
f_4 \\
f_5 \\
f_6 \\
f_7
\end{bmatrix}.
\]

Problem 0.5. Watkins 1.2.20

Solution. The generic equation is

\[
-k_i(x_i - x_{i-1}) + k_{i+1}(x_{i+1} - x_i) = -f_i,
\]

where the three terms are the force of the left spring, the force of the right spring, and the force you exert. Simplified, this is

\[
k_{i+1}x_{i+1} - (k_i + k_{i+1})x_i + k_ix_{i-1} = -f_i.
\]

As a matrix equation, this is

\[
\begin{bmatrix}
-k_i - k_{i+1} & k_{i+1} & 0 & \cdots & 0 \\
k_i & -k_i - k_{i+1} & k_{i+1} & \cdots & 0 \\
0 & k_i & -k_i - k_{i+1} & \cdots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & 0 & \cdots & -k_i - k_{i+1}
\end{bmatrix}
\begin{bmatrix}x_1 \\
x_2 \\
x_3 \\
\vdots \\
x_n\end{bmatrix}
= \begin{bmatrix}-f_1 \\
-f_2 \\
-f_3 \\
\vdots \\
-f_n\end{bmatrix}.
\]

The solution I found was

\[
\vec{x} \approx (0.52, 1.0, 1.6, 2.1, 2.6, 2.1, 1.7, 1.2, .71, .24, \\
-0.24, -.71, -1.2, -1.7, -2.1, -2.6, -2.1, -1.6, -1, -0.52).
\]
This seems reasonable since the left cars should move right from the positive force, while the right cars should move left from the negative force.

**Problem 0.6.** Watkins 1.2.21

*Solution.* (a) The symbol \(\lim_{h \to 0}\) just means “what happens as \(h\) is approximately 0?” Thus, if \(h \approx 0\), then \(u'(x) \approx \frac{u(x + h) - u(x)}{h}\). (More technically, you’d have to use the \(\epsilon - \delta\) definition of a limit, but I did not expect or intend for you to do that.)

The other one is the same, but with \(x\) replaced by \(x - h\). Both represent approximate slopes of \(u(x)\) near \(x\). The first one is the approximate slope to the right of \(x\), while the second is the approximate slope to the left of \(x\).

(b) The calculation is simple: add the two fractions and divide by 2. This new fraction represents the average slope between \(x - h\) and \(x + h\). The picture could look like this:

![Diagram](image)

**Problem 0.7.** Using basic programming only (i.e., not using MATLAB’s built in matrix multiplication), write a MATLAB function that takes as input two matrices, and outputs their product.

*Solution.* Here is one possible way to program it.

```matlab
function prod = matmult(A,B)

%First, let's make sure the sizes are right to multiply.
%I didn't expect you to do this, but it is good practice
%to always check that your inputs make sense.
if size(B,1) ~= size(A,2)
    error("Your matrices are not the right sizes.")
end

%next, create an empty matrix of the right size.
```

```
It should have the same number of rows as \( A \), and the same number of columns as \( B \):

\[
\text{prod} = \text{zeros}(\text{size}(A,1), \text{size}(B,2));
\]

\[
\begin{array}{c}
\text{for } i = 1: \text{size}(B,2) \% \text{for each column of } B \\
\quad \text{for } j = 1: \text{size}(A,1) \% \text{for each row in } A \\
\quad\quad \text{for } k = 1: \text{size}(A,2) \% \text{do a dot product} \\
\quad\quad\quad \text{prod}(j, i) = \text{prod}(j, i) + A(j, k) \times B(k, i); \\
\quad\end{array}
\]

\]

\]

Problem 0.8. What are the two main ways we can measure the speed of an algorithm/program? What are their respective strengths and weaknesses?

Solution. The first way is by measuring time. The strength is that this is what, in the end, we really care about. Time on a computer costs money. One weakness is that it depends heavily on the computer, and so it is impossible to compare time results from two different computers. It is also affected by random blips during computing. Measuring time also depends heavily not only on the overall algorithm’s speed, but also on the nitty-gritty details of how data is stored and accessed and what language you program in and a million other important factors. Similarly, it’s not a theoretical tool; it’s hard to extrapolate what will happen in other circumstances.

The second way is by measuring flops (number of calculations). The weakness is that we don’t really care about the number of calculations, since time is what’s important. On the other hand, things like memory accesses, etc. are roughly proportional to the number of calculations, so it gives us a good idea of time. One strength is that it is dependent only on the algorithm, and so it is easy to compare different algorithms. Also, since we calculate it based on the size of the matrices, it is a good theoretical tool that tells us how much more difficult a problem is if we, say, double the size of the matrices.

Problem 0.9. Consider the differential equation \( f''(x) - 2f(x) = x \) on the interval \([0, 3]\). Set up and solve a matrix problem to estimate the solution to this equation with \( f(0) = 0 \) and \( f(3) = 0 \). Use 6 subintervals, i.e., use a “mesh size” of \( 1/2 \). Turn in 1. The \( A\vec{x} = \vec{b} \) you set up, 2. The solution vector \( \vec{x} \) (You may use MATLAB to find it.), 3. A sketch of the graph of your approximate solution \( f(x) \). (Hint: Your matrix should be 5x5.)

Solution. Using the approximation for the second derivative, and \( h = 1/2 \), the general equation is

\[
4f(x_{i+1}) - 10f(x_i) + 4(x_{i-1}) = x_i,
\]
where the $x_i$ are the mesh points 0, 1/2, 1, $\cdots$, 3. Using the boundary conditions $f(x_0) = 0$ and $f(x_7) = 0$, the matrix equation is

$$
\begin{bmatrix}
-10 & 4 & 0 & 0 & 0 \\
4 & -10 & 4 & 0 & 0 \\
0 & 4 & -10 & 4 & 0 \\
0 & 0 & 4 & -10 & 4 \\
0 & 0 & 0 & 4 & -10 \\
\end{bmatrix}
\begin{bmatrix}
f(x_1) \\
f(x_2) \\
f(x_3) \\
f(x_4) \\
f(x_5) \\
\end{bmatrix}
= 
\begin{bmatrix}
1/2 \\
1 \\
3/2 \\
2 \\
5/2 \\
\end{bmatrix}.
$$

The solution to that, according to MATLAB, is

$$
\begin{bmatrix}
f(x_1) \\
f(x_2) \\
f(x_3) \\
f(x_4) \\
f(x_5) \\
\end{bmatrix}
= 
\begin{bmatrix}
-0.21484 \\
-0.41209 \\
-0.56538 \\
-0.62637 \\
-0.50055 \\
\end{bmatrix}.
$$

The approximate solution is plotted as the blue line, which connects the points we found. The red line was not part of the homework assignment. It is the true solution to the differential equation.