Problem 0.1. Consider a grid on $[0, 1] \times [0, 1]$ with mesh size $h = 1/4$. Set up the matrix equation $Ax = b$ that you can solve in order to approximate the solution to $-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = f$, where $f$ is 1 at $(1/2, 1/2)$, and zero elsewhere. Use the boundary conditions that $u = 0$ on the boundary of $[0, 1] \times [0, 1]$.

Solution. Since $h = 1/4$, there are 9 points at which we want to know the value of $u$. (The boundary values are all 0, so we don’t need to calculate them.) Our matrix is thus $9 \times 9$. It would be acceptable to use the description of the matrix in the book, but let me do it more directly.

Think of $u$ as a function of $x$ and $y$, so $u(x, y)$. Then $u_{xx}(x, y) \approx \frac{u(x + h, y) - 2u(x, y) + u(x - h, y)}{h^2}$, and similarly for $u_{yy}$. Thus, for instance, at $(x, y) = (1/2, 1/2)$, the equation becomes, after multiplying through by $h^2$,

$$4u(1/2, 1/2) - u(3/4, 1/2) - u(1/2, 3/4) - u(1/4, 1/2) - u(1/2, 1/4) = 16f(1/2, 1/2).$$

Of course, this is in terms of coordinates, but the point $(x, y) = (1/2, 1/2)$ is the point $x_5$ in the usual per-row ordering. Also, $(x, y) = (3/4, 1/2)$ is $x_6$, and similarly for the others. For the other equations, at least one of the values used will be a boundary point, and so we can set that term to zero.

Putting this ordering and these equations together, we get this system

$$\begin{bmatrix}
4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
-1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & -1 & 4 & 0 & 0 & -1 & 0 & 0 & 0 \\
-1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 \\
0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\
0 & 0 & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\
0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4
\end{bmatrix}
\begin{bmatrix}
u(x_1) \\
u(x_2) \\
u(x_3) \\
u(x_4) \\
u(x_5) \\
u(x_6) \\
u(x_7) \\
u(x_8) \\
u(x_9)
\end{bmatrix}
= \begin{bmatrix}
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0
\end{bmatrix}$$

Problem 0.2. Watkins 8.1.12
Solution. Using the same kind of notation as (8.1.8), though with three indices, the general equation will be
\[-u_{i-1,j,k} - u_{i,j-1,k} - u_{i,j,k-1} + 6u_{i,j,k} - u_{i+1,j,k} - u_{i,j+1,k} - u_{i,j,k+1} = h^2 f_{i,j,k}\]
for \(i, j, k = 1, \ldots, m - 1\). As there are \((m - 1)^3\) total grid points and thus that many unknowns. In each equation, as above, there are at most 7 unknowns, perhaps fewer near the boundary points. □

Problem 0.3. Watkins 8.2.12

Solution. (a) This is easiest to see by looking at one vector at a time. Recall that the formula for Gauss-Seidel is
\[x^{(k+1)}_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x^{(s)}_j \right).\]
Notice that for each \(i\), this is one row of the formula given in the book. The use of \(E\) on the previous guess but \(F\) on the most recent guess guarantees that the correct guess is used for each term in the sum, and note that the signs of \(E\) and \(F\) are opposite of what you might think.

(b) Multiply both sides by \(D\) to get
\[Dx^{(k+1)} = b + Ex^{(k+1)} + Fx^{(k)}.\]
Then, subtract over the \(E\) term and factor out the \(x^{(k+1)}\):
\[(D - E)x^{(k+1)} = b + Fx^{(k)}.\]
Finally, multiply both sides by \((D - E)^{-1}\).

(c) Obviously, if \(M = D - E \) and \(A = D - E - F\), then, \(F = M - A\). Then, starting with the previous equation,
\[x^{(k+1)} = M^{-1}(b + (M - A)x^{(k)})\]
\[x^{(k+1)} = M^{-1}(b + Mx^{(k)} - Ax^{(k)})\]
\[x^{(k+1)} = M^{-1}Mx^{(k)} + M^{-1}(b - Ax^{(k)})\]
\[x^{(k+1)} = x^{(k)} + M^{-1}r^{(k)}.\]
□

Problem 0.4. Write a function in MATLAB that takes as input a matrix \(f\), a mesh size \(h\), a maximum number of iterations, a tolerance for convergence and the relaxation constant \(\omega\). The matrix \(f\) represents the function on the right hand side of the Poisson equation (8.1.6). Its values give the value of \(f\) at the corresponding grid point. Your program should use SOR to calculate the solution \(u\) of the Poisson equation (with \(u = 0\) on the boundary), using that max number of iterations and tolerance. Use only basic programming. Return the solution \(u\) as a matrix, and graph the solution. (Hint: Be careful of the boundary condition. To graph the solution, if \(u\) is your solution matrix, you can use the command
surf(u). If correct, your program should output a graph that looks something like the \( f \) you input, but smoothed out, and the graph should go to zero at the boundaries. It’s probably easiest and fastest to use something like (8.2.20), rather than the full matrix code.)

Solution. Here is my code. I did some non-obvious things with how I stored \( u \) to simplify my coding.

```matlab
function u=poisson(f,h,iter,tolerance,w)

[n,m]=size(f);

%To represent my function, I made my matrix for u
%two bigger than "necessary." This way, my
%boundary condition is represented by the zeros
%in the outside entries of u, and so I don't need
%to modify my code to deal with the boundary.
u=zeros(n+2,m+2);

for k=1:iter
    %Each k is an iteration
    oldu=u;
    for i=1:n
        for j=1:m
            %Because the first row/column (and last) represent
            %the zero boundary condition, I have to refer
            %to one higher index for each index.
            %Thus, I use j+2 instead of j+1, etc., but only for u.
            uhat=(u(i,j+1)+u(i+1,j)+u(i+1,j+2)+u(i+2,j+1)+h^2*f(i,j))/4;
            delta=uhat-u(i+1,j+1);
            u(i+1,j+1)=u(i+1,j+1)+w*delta;
        end
    end
    %check tolerance using the 2-norm.
    %but since stored as a matrix, not a vector,
    %the 2-vector norm is the Frobenius matrix norm.
    if norm(oldu-u,'fro')/norm(u,'fro')<tolerance
        %display number of iterations.
        k
        break
    end
end
surf(u)
```
Problem 0.5. A good guess can greatly reduce the number of iterations required to solve a system using iterative methods. As you may have noticed while playing with the previous problem, if $f$ is overall positive, so is $u$. Thus a good guess for $u$ might be a matrix with all entries that are the average value of the matrix $f$. Since $u$ peaks where $f$ does, it might help to add a small multiple of $f$ to your original guess for $u$ as well. Play around with this using your Poisson equation solver and report your findings in an informal way. (Hints: The command `mean(mean(f))` will return the average value of $f$. A fun matrix to play with is $f=rand(20)-.47$, for instance. A tolerance of $10^{-2}$, measured in the way the book does, is more than sufficient. Compare different $\omega$’s as well.)

Solution. This problem was, essentially, a very poor attempt at “multigrid” methods. A better attempt would be to find $u$ on a very coarse grid, then use that to choose our approximate solution on a fine grid. For instance, you could run the problem on a $5 \times 5$ grid, then use that solution to get an approximate solution on a $100 \times 100$ grid, and use that approximate as the starting guess on that problem. The $5 \times 5$ solve is incredibly cheap, and would provide a better initial guess than the “average of $f$.”

To my above code, after the line where I initialized $u$, I added the following line:

```matlab
u(2:n+1,2:m+1)=ones(n,m)*mean(mean(f))*n*h+3*f*h;
```

(The weird left hand side is because, for instance, the first row and column of $u$ are 0 to represent the boundary conditions.) I multiply by $n$ since it seemed that bigger multiples of the average matrix was better for bigger matrices. I multiply by $h$ so that this is a good guess regardless of my choice of $h$ in my above code.

I tried this on matrices like $f=rand(20)-.47$. With $\omega=1$ and tolerance $10^{-2}$, I got 50, 16 and 14 iterations for an initial guess of 0, an initial guess of just the average matrix, and an initial guess of the average with the additional $f$’s. The good guess clearly improved my convergence. At a tolerance of $10^{-5}$, I instead got 343, 264, and 175 iterations.

Using the optimum $\omega$, however, my guess did not help. At $10^{-2}$ tolerance, I got 21, 24, and 35 iterations! For $10^{-5}$, I got 53, 54, and 63. Using a larger than optimum $\omega = 1.95$, I got similar results for $10^{-5}$ of 208, 214, and 250 iterations. Apparently, my guess was not that great for SOR, even though it was good for Gauss-Seidel. I’m not really sure why. It did work better if I halved the guess.

For $f$ being zero except for a 1 in the middle, my guess helped, but not much. For $\omega = 1$, tolerance $10^{-5}$, I got 341, 318, and 307 iterations. For $\omega = 1.8$, I got 49, 51, and 61.

For larger matrices, like `rand(40)-.47`, it continued to work well. For $\omega = 1$, I got 56, 20, and 16. For the optimum $\omega = 1.9$, though, I got 33, 33, and 56. For a $80 \times 80$, I got 86, 13, and 12, and for a $160 \times 160$, I got 90, 8, and 8.
In summary, my guesses worked well for Gauss-Seidel, but for some reason were not very successful in SOR. As I mentioned in the first paragraph, solving a simple version of the problem would probably provide a much better initial guess. My “average” matrix is vaguely like solving the problem on a $1 \times 1$ grid!

Problem 0.6. What are some of the main advantages of iterative methods over direct methods?

Solution. In iterative methods, you can exploit a good guess, since your method just improves any guess you have. In direct methods, even if you have an excellent guess, you just have to do all the calculations anyway.

In iterative methods, you can stop early (after a few iterations) if you only need a rough solution. For direct methods, if you stop early, the method tells you nothing.

Iterative methods, for many large, extremely sparse problems, such as the Poisson equation, are only $O(n)$ (though with a large $C$), while the best direct methods are $O(n^{3/2})$. (The book says this in a footnote.) That means that for these large problems, iterative methods will simply be faster.

Problem 0.7. Write a function in MATLAB that takes as input a matrix $A$, a vector $b$, a maximum number of iterations, a tolerance for convergence and the relaxation constant $\omega$. Have this function use SOR to calculate the solution $x$ of $Ax=b$, using that max number of iterations and tolerance. Assume a starting guess of $x=0$. (You can test your program against the solution of any $Ax=b$.)

Solution. I will include two versions of this program, with different ways of finding $\hat{x}$. The first is very similar to how I programmed Jacobi’s method in class.

```matlab
function x=SOR(A,b,maxIter,tolerance,w)

%maxIter is maximum number of iterations
%tolerance is desired accuracy

n=length(b);
x=zeros(n,1);
iterations=maxIter;

for k=1:maxIter
    oldx=x;
    for i=1:n
        xhat=b(i);
        for j=1:i-1
            xhat=xhat-A(i,j)*x(j);
        end
        for j=i+1:n
            xhat=xhat-A(i,j)*x(j);
        end
        x=xhat;
    end
    if norm(x-oldx)/norm(x)<tolerance
        break;
    else
        oldx=x;
    end
end
```


end

xhat=xhat/A(i,i);
delta=xhat−x(i);
x(i)=x(i)+w*delta;
end
%check if residual is small
if norm(b−A*x)/norm(b)<tolerance
    iterations=k
    break
end
end

The second method has a faster and shorter way to program it. The command

\texttt{tempA=A(i,:);} extracts the \texttt{i-th row of A.}

\begin{verbatim}
function x=SOR(A,b,maxIter,tolerance,w)
%maxIter is maximum number of iterations
%tolerance is desired accuracy

n=length(b);
x=zeros(n,1);
iterations=maxIter;

for k=1:maxIter
    oldx=x;
    for i=1:n
        xhat=b(i);
        tempA=A(i,:);
        tempA(i)=0;
        xhat = xhat−tempA*x;

        xhat=xhat/A(i,i);
        delta=xhat−x(i);
        x(i)=x(i)+w*delta;
    end
    %check if residual is small
    if norm(b−A*x)/norm(b)<tolerance
        iterations=k
        break
    end
end
end
\end{verbatim}