Problem 0.1. Watkins 8.3.10

Solution. The k-th error is \( e^{(k)} = G^k e^{(0)} \). As discussed before, that means that \( \| e^{(k+j)} \| \approx \rho(G)^k \), i.e., the norm of the error is approximately multiplied by \( \rho(G)^k \). Thus, I want \( \rho(G)^k = 1/e \). Solving this equation for \( k \), the number of iterations, I get \( k \approx -1/\ln(\rho(G)) = 1/R_\infty(G) \). Thus it takes about \( 1/R_\infty(G) \) iterations to reduce the error by a factor of \( e \).

Problem 0.2. Watkins 8.3.12

Solution. We assume that \( \rho(G_{GS}) = \rho(G_J)^2 \). Then

\[
R_\infty(G_{GS}) = -\ln(\rho(G_{GS})) = -\ln(\rho(G_J)^2) = -2\ln(\rho(G_J)) = 2R_\infty(G_J),
\]

as desired.

Problem 0.3. Watkins 8.3.17a

Solution. For \( h = 1/10 \), \( \rho(G) = \cos(\pi/10) \approx 0.951 \) and \( R_\infty(G) \approx -\ln(\cos(\pi/10)) \approx 0.0502 \).

Since the error after \( k \) steps is multiplied by about \( \rho(G)^k \), I want \( \rho(G)^k = 1/100 \). Thus I want \( k \approx 91.66 \), so it takes about 92 iterations to reduce my error by a factor of 100.

Problem 0.4. Watkins 8.3.19a,b

Solution. (a) If the eigenvalues of \( A \) are \( \lambda_1, \ldots, \lambda_n \), then the eigenvalues of \( I - \omega A \) are \( 1 - \omega \lambda_1, \ldots, 1 - \omega \lambda_n \). (This is similar to what we talked about for the shift-and-invert method for finding eigenvalues.) Since all the \( \lambda_i > 0 \), then clearly \( 1 - \omega \lambda_i < 1 \), as desired.

(b) For Richardson’s method, \( G = I - \omega A \). Thus the eigenvalues we found above must all have norm less than 1 in order for Richardson’s method to converge. Since we already know \( 1 - \omega \lambda_i < 1 \), we just need to show that \( 1 - \omega \lambda_i > -1 \) as well. Rearranging that equation, we find that the method converges if and only if \( \omega < 2/\lambda_i \). Since \( \lambda_n \) is the smallest eigenvalue, that shows that we actually just need to make sure that \( \omega < 2/\lambda_n \).

Problem 0.5. Watkins 8.3.20
Solution. Using what the proposed iteration is, we can calculate
\[ x^{(k+1)} = x^{(k)} + M^{-1}r^{(k)} \]
\[ = x^{(k)} + M^{-1}(b - Ax^{(k)}) \]
\[ = x^{(k)} + M^{-1}b - M^{-1}(M - N)x^{(k)} \]
\[ = x^{(k)} + M^{-1}b - x^{(k)} + M^{-1}Nx^{(k)} \]
\[ = M^{-1}Nx^{(k)} + M^{-1}b, \]
which is the standard iteration for any splitting.

Since \( M = \frac{1}{\omega} I \) for Richardson’s method, this last result is immediate.  \( \square \)

Problem 0.6. Watkins 8.3.22

Solution. If we consider the preconditioned problem \( M^{-1}Ax = M^{-1}b \), Richardson’s method with \( \omega = 1 \) had \( \bar{M} = I \) and \( \bar{N} = I - M^{-1}A \), and uses \( \bar{b} = M^{-1}b \).

The iteration for Richardson’s method is \( \bar{M}x^{(k+1)} = \bar{N}x^{(k)} + \bar{b} \). Using the calculated quantities,
\[ x^{(k+1)} = (I - M^{-1}A)x^{(k)} + M^{-1}b \]
\[ = x^{(k)} - M^{-1}Ax^{(k)} + M^{-1}b \]
\[ = x^{(k)} - M^{-1}(M - N)x^{(k)} + M^{-1}b \]
\[ = x^{(k)} - x^{(k)} + M^{-1}Nx^{(k)} + M^{-1}b \]
\[ = M^{-1}Nx^{(k)} + M^{-1}b \]
which is the standard iteration. (There are other ways to do this calculation.)  \( \square \)

Problem 0.7. Watkins 8.4.7a

Solution. Consider the very first step. Our direction is \( e_1 \), i.e., the “x”-direction. The exact line search produces
\[ \alpha_1 = \frac{e_1^T r^{(1)}}{e_1^T Ae_1} = \frac{r_1^{(1)}}{a_{11}}. \]

Thus \( x^{(2)} \) is the starting guess plus \( r_1^{(1)}/a_{11} \) in the first entry only. But \( x_1^{(1)} + r_1^{(1)}/a_{11} = \frac{1}{a_{11}} (b_1 - \sum_{j \neq 1} a_{1j}x_j) \), which is the standard step for Gauss-Seidel.

The second and so forth steps proceed similarly. Thus each group of \( n \) steps is one iteration of Gauss-Seidel.  \( \square \)

Problem 0.8. Watkins 8.4.12
Solution.

\[ x^{(k+1)} = x^{(k)} + \alpha_k p^{(k)} \]
\[ Ax^{(k+1)} = Ax^{(k)} + \alpha_k A p^{(k)} \]
\[ b - Ax^{(k+1)} = b - (Ax^{(k)} + \alpha_k A p^{(k)}) \]
\[ r^{(k+1)} = r^{(k)} - \alpha_k A p^{(k)} \]

Problem 0.9. Usually it doesn’t matter which stopping criterion you use. Explain briefly why sometimes it does.

Solution. Though usually it doesn’t matter, sometimes the least stringent condition

\[ \|r^{(k+1)}\|_2 < \epsilon (\|b\|_2 + \|A\|_2 \|x^{(k+1)}\|_2) \]

will allow the iteration to converge more quickly, while still showing that the iteration was backward stable. In fact, sometimes this condition will allow it to converge, while the other standard choices will never converge at all!

Problem 0.10. Briefly and intuitively explain why preconditioners help descent methods converge more quickly.

Solution. Descent methods struggle when the level sets of \( J(y) \) form long, narrow “canyons,” and this happens when the matrix has a large condition number. A preconditioner effectively “widens” the canyons before the descent method is begun, which makes the descent methods converge much more quickly.

Problem 0.11. Watkins 8.4.8(a,b). Use a sketch like Figure 8.4 to explain why part b makes sense graphically, and why overrelaxation performs better than standard Gauss-Seidel, but only for a good choice of \( \omega \). (This picture, to be clear, only applies when \( A \) is positive definite.)

Solution. (a) This solution works essentially the same as for 8.4.7, which I did above, so I won’t repeat it.

(b) Let’s just consider \( p = e_i \). (I’ll drop the superscript \( (k) \)’s for simplicity.) Then

\[ \alpha = \frac{e_i^T r}{e_i^T A e_i} = \frac{r_i}{a_{ii}}. \]

Note that \( a_{ii} > 0 \) since \( A \) is positive definite.
Next, consider $J(x + \omega \alpha e_i)$.

\[
J(x + \omega \alpha e_i) = \frac{1}{2} (x + \omega \alpha e_i)^T A (x + \omega \alpha e_i) - (x + \omega \alpha e_i)^T b \\
= J(x) + \omega \alpha e_i^T A x + \frac{1}{2} \omega^2 \alpha^2 a_{ii} - \omega \alpha e_i^T b \\
= J(x) + \frac{1}{2} \omega^2 \alpha^2 a_{ii} + \omega \alpha e_i^T (A x - b) \\
= J(x) + \frac{1}{2} \omega^2 \alpha^2 a_{ii} - \omega r_i \\
= J(x) + \frac{1}{2} \omega^2 a_{ii} \frac{r_i^2}{a_{ii}} - \omega \frac{r_i}{a_{ii}} r_i \\
= J(x) + \frac{r_i^2}{a_{ii}} \left( \frac{1}{2} \omega^2 - \omega \right).
\]

Thus $J(x^{(k+1)})$ is bigger than $J(x^{(k)})$ if $\frac{r_i^2}{a_{ii}} \left( \frac{1}{2} \omega^2 - \omega \right) > 0$ and smaller if it is less than zero. Since $\frac{r_i^2}{a_{ii}} > 0$, this depends only on the sign of $\frac{\omega^2}{2} - \omega$, and so the desired results are easy to show.

A less technical way is this: Consider the line defined by your search direction. Along that line, $J(y)$ is a parabola. An exact line search will find the bottom of the parabola. If you double the distance to the bottom from where you started, you will end at the same height as you started, because of symmetry. Thus, $\omega = 2$ makes $J(y)$ the same as before. Between where you started ($\omega = 0$) and $\omega = 2$, you decrease $J$, thanks to the shape of the parabola. Similarly, outside that range, $J$ increases, thanks to the shape.

The following figure has three pictures. These are exact graphs for SOR with, respectively, $\omega = 1, 1.5, \text{ and } 2$. The problem I was solving was $\begin{bmatrix} 1.1 & 1 \\ 1 & 1.05 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 7 \\ 5 \end{bmatrix}$, with initial guess $[0; 0]$.

![Figure 1. SOR with various $\omega$.](image)

As you can see, part (b) makes sense because $\omega = 1$ goes straight to the lowest part, and $\omega = 2$ goes to the location with the same height as you started. The
reason overrelaxation performs better is that standard Gauss-Seidel gets stuck in the canyon very early, and goes back and forth in it a lot. But overrelaxation allows you to run down the length of the canyon more quickly, so you don’t have so many tiny steps.

For fun, I have included the equivalent graphs for steepest descent and conjugate gradient. Remember, conjugate gradient converges in two steps, so it has converged already.

\[ \square \]

**Figure 2.** Steepest descent and conjugate gradient methods.

**Problem 0.12.** Write a function in MATLAB that takes as input a symmetric positive definite matrix \( A \), the right hand side vector \( b \), a maximum number of iterations, and a tolerance for convergence and returns as output the solution of \( Ax = b \) as found by performing the conjugate-gradient method as found in (8.7.1). It will be easiest to program the exact algorithm found there, though you will have to change notation and names of variables so that MATLAB can understand it, and check the tolerance.

**Solution.** Here is my code, which is very, very similar to the algorithm in the book.

\[
\begin{aligned}
&\text{function } x = \text{CG}(A, b, \text{iter}, \text{tolerance}) \\
&n = \text{length}(A); \\
normA = \text{normest}(A); \\
% \text{estimate norm, since full norm is too slow.} \\
normb = \text{norm}(b); \\
x = \text{zeros}(n, 1); \\
\quad r = b - A * x; \\
\quad p = r; \\
\quad v = r' * r; \\
\end{aligned}
\]
for i=1:iter
    q=A*p;
    mu=p'*q;
    a=v/mu;
    x=x+a*p;
    r=r-a*q;
    vp=r'*r;
    b=vp/v;
    p=r+b*p;
    v=vp;
    if norm(r)/(normb+normA*norm(x))<tolerance
        %display number of iterations
        break
    end
end

Problem 0.13. Compare your SOR solver with your conjugate-gradient method. Pick a large matrix (such as the one for the Poisson equation), and compare the number of iterations each method uses to solve $Ax = b$ up to the desired precision. (Compare with several $\omega$.)

Solution. I used this code to generate the Laplacian matrix. It’s not the most efficient, but it works.

function A = lap(m)
    r=zeros(m^2,1);
    r(1)=4; r(2)=-1; r(m+1)=-1;
    A=sparse(toeplitz(r));
    %Making it sparse is super important. It saves room
    %and makes it much faster.

    r(1)=4; r(2)=-1; r(m+1)=-1;
    A=sparse(toeplitz(r));
    %Making it sparse is super important. It saves room
    %and makes it much faster.

    %This is NOT quite the right A.
    % There need to be some extra zeros.
    for i=1:m-1
        A(i*m,i*m+1)=0;
        A(i*m+1,i*m)=0;
    end
I then ran it in the above CG code, with the $m = 100$ version, $f$ the vector of 10000 zeros, except $f(123) = 1$, $f(666) = 24$ and $f(3243) = -3$. (Any $f$ other than zero would have worked.) Using that code and stopping condition, it took 288 iterations to converge to $10^{-8}$ precision, and only 53 to converge to $10^{-2}$ precision. (I changed the stopping condition to $\text{norm(oldu-u)}/\text{norm(u)}$ to be consistent with my SOR code.)

I then reshaped $f$ using $\text{reshape}(f,[100,100])$, and used my efficient SOR code that was specifically for the Poisson problem. For $\omega = 1$, and $10^{-2}$ tolerance, it took 50 iterations, while with $\omega = 1.9$, it took only 35. If I increased the tolerance to $10^{-8}$, it took a (very slow) 643 iterations. I didn’t bother trying it with $\omega = 1$, since I didn’t want to sit here all day...