for some real $\alpha_k$. The process of choosing $\alpha_k$ from among all $\alpha \in \mathbb{R}$ is the line search. We want to choose $\alpha_k$ in such a way that $J(x^{(k+1)}) \leq J(x^{(k)})$. One way to ensure this is to choose $\alpha_k$ so that $J(x^{(k+1)}) = \min_{\alpha \in \mathbb{R}} J(x^{(k)} + \alpha p^{(k)})$. If $\alpha_k$ is chosen in this way, we say that the line search is exact. Otherwise, we say that it is inexact.

For some types of functions an exact line search can be a formidable task, but for quadratic functions like (7.4.1), line searches are trivial. The following theorem shows that the correct value of $\alpha$ can be obtained from a formula.

**Theorem 7.4.5** Let $x^{(k+1)} = x^{(k)} + \alpha_k p^{(k)}$ be obtained by an exact line search. Then

$$\alpha_k = \frac{p^{(k)}T \tau^{(k)}}{p^{(k)}T A p^{(k)}},$$

where $\tau^{(k)} = b - Ax^{(k)}$.

**Proof.** Let $g(\alpha) = J(x^{(k)} + \alpha p^{(k)})$. The minimizer of $g$ is $\alpha_k$. A routine computation shows that $g(\alpha) = J(x^{(k)}) - \alpha p^{(k)}T \tau^{(k)} + \frac{1}{2} \alpha^2 p^{(k)}T A p^{(k)}$. This is a quadratic polynomial in $\alpha$ whose unique minimum can be found by solving the equation $g'(\alpha) = 0$. Since $g'(\alpha) = -p^{(k)}T \tau^{(k)} + \alpha p^{(k)}T A p^{(k)}$, we have $\alpha(\alpha) = p^{(k)}T \tau^{(k)} / p^{(k)}T A p^{(k)}$. \hfill $\square$

**Remark 7.4.6** Notice that $\alpha_k = 0$ if and only if $p^{(k)}T \tau^{(k)} = 0$. We want to avoid this situation because we would rather have $x^{(k+1)} \neq x^{(k)}$. The equation $p^{(k)}T \tau^{(k)} = 0$ just says that the search direction is orthogonal to the residual. Thus we would normally want to choose $p^{(k)}$ so that it is not orthogonal to $\tau^{(k)}$. This is always possible unless $\tau^{(k)} = 0$ (in which case $x^{(k)}$ is the solution). If we always choose $p^{(k)}$ so that $p^{(k)}T \tau^{(k)} \neq 0$, we will always have the strict inequality $J(x^{(k+1)}) < J(x^{(k)})$. \hfill $\square$

Let us now consider some examples of descent methods.

**Exercise 7.4.7** In this exercise the initial guess will be denoted $x^{(1)}$ for notational simplicity. Similarly the first search direction will be denoted $p^{(1)}$. Consider a method in which the first $n$ search directions $p^{(1)}, \ldots, p^{(n)}$ are taken to be the standard unit vectors $e_1, \ldots, e_n$, the next $n$ search directions $p^{(n+1)}, \ldots, p^{(2n)}$ are $e_1, \ldots, e_n$ again, so are $p^{(2n+1)}, \ldots, p^{(3n)}$, and so on. Suppose an exact line search is performed at each step.

(a) Show that each group of $n$ steps is one iteration of the Gauss-Seidel method.

(b) Show that it can happen that $J(x^{(k+1)}) = J(x^{(k)})$ even if $x^{(k)} \neq x$.

(c) Prove that the situation described in part (b) cannot persist for $n$ consecutive steps. In other words, show that if $x^{(k)} \neq x$, then $J(x^{(k+n)}) < J(x^{(k)})$.

Thus each Gauss-Seidel iteration lowers the potential energy. This idea leads to a proof that the Gauss-Seidel method applied to a positive definite matrix always converges. \hfill $\square$
Notice also that \( \tilde{p}^T \tilde{r} = p^T r \) and \( \tilde{p}^T \tilde{q} = p^T q \), so the tildes can be dropped from the computation of \( \alpha \).

So far the transformation process appears quite boring. We started with Algorithm 7.4.14 in the transformed coordinate system, involving \( \tilde{p}, \tilde{q}, \tilde{r}, \) etc., then defined \( p, q, \) and \( r \) in such a way that we could rewrite the equations with the tildes removed. The result is the original Algorithm 7.4.14, or so it appears at first.

The one thing we still need to discuss is the transformation of the lines \( \tilde{p} \leftarrow \tilde{q} \). Now let us focus our attention specifically on the steepest descent method. In the original steepest descent method we had \( p = r \), so preconditioned steepest descent will have \( \tilde{p} = \tilde{r} \). Now here is the important point. When we transform the equation \( \tilde{p} = \tilde{r} \) back to the original coordinate system, we do not get \( p = r \), because \( p \) and \( r \) obey different transformation laws.\(^4\) Since \( \tilde{p} = Rp \) and \( \tilde{r} = R^{-T} r \), we have \( p = R^{-1} R^{-T} r \).

Recalling that \( M = R^T R \), we see that \( p = M^{-1} r \). With this transformation, the symbol \( R \) disappears from the algorithm. Thus the preconditioned steepest descent algorithm with preconditioner \( M \) is as follows.

**Preconditioned Steepest Descent Algorithm** (with preconditioner \( M \)) for solving \( Ax = b \). On entry the initial guess is stored in \( x \), and the vector \( b \) is stored in \( r \). The algorithm returns in \( x \) its best estimate of the solution and a flag that indicates whether or not the specified tolerance was achieved.

\[
\begin{align*}
\dot{r} &\leftarrow r - Ax \\
p &\leftarrow M^{-1} r \\
k &\leftarrow 0 \\
\text{do until converged or } k &\leftarrow l \\
q &\leftarrow Ap \\
\alpha &\leftarrow p^T r / p^T q \\
x &\leftarrow x + \alpha p \\
r &\leftarrow r - \alpha q \\
p &\leftarrow M^{-1} r \\
k &\leftarrow k + 1 \\
\text{if not converged, set flag}
\end{align*}
\]

Algorithm 7.5.2 is identical to Algorithm 7.4.14, except for the lines \( p \leftarrow M^{-1} r \).

Indeed, Algorithm 7.5.2 is an instance of Algorithm 7.4.14. The only effect of applying a preconditioner to the steepest descent algorithm is that the search direction is changed from \( r \) to \( M^{-1} r \).

**Exercise 7.5.3** Check all of the details of the foregoing development. Verify that Algorithm 7.5.2 is indeed the correct translation of the steepest descent algorithm for \( \tilde{A} \tilde{x} = \tilde{b} \) where \( \tilde{A} = R^{-T} AR^{-1} \).

\[ \square \]

\( ^4 \) There are two transformation laws. In the language of duality, the vectors \( x \) and \( p \), which satisfy \( z = R^{-1} \tilde{z} \), are primal, and the vectors \( b, r, \) and \( q \), which satisfy \( z = R^T \tilde{z} \), are dual.
Examples of Preconditioners

Our derivation assumed that $M$ is positive definite. We thus search for preconditioners among those iterative methods that have positive definite splitting matrices.

Example 7.5.4 Jacobi’s method has $M = D$, the diagonal part of $A$. This $M$ is positive definite if $A$ is. The Jacobi preconditioner, which is also known as the diagonal preconditioner, is particularly easy to apply. The operation $p \leftarrow M^{-1} r$ amounts to $p_i \leftarrow r_i / a_{ii}$, $i = 1, \ldots, n$. Unfortunately it is not a very powerful preconditioner.

Exercise 7.5.5 Show that the convergence of the steepest descent algorithm on the model problem (7.1.8) is not accelerated at all by the Jacobi preconditioner.

In problems for which the main-diagonal entries of the coefficient matrix vary markedly in magnitude, the Jacobi preconditioner is often effective.

Exercise 7.5.6 Apply the steepest descent algorithm to the system

$$
\begin{bmatrix}
10 & 1 \\
1 & 1
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2
\end{bmatrix} =
\begin{bmatrix}
13 \\
4
\end{bmatrix}.
$$

Starting with $x^{(0)} = 0$, perform three iteration using (a) no preconditioner, (b) the Jacobi preconditioner. You may do this by hand or, better yet, write a simple MATLAB script. Note the extreme effectiveness of the preconditioner in this case. If you are using MATLAB, you might like to compare the condition numbers of $A$ and $\tilde{A} = D^{-1/2} AD^{-1/2}$.

On small problems like the one in Exercise 7.5.6, there is no harm in forming the matrix $M^{-1}$ explicitly and using it to perform the preconditioning step $p \leftarrow M^{-1} r$. On big problems it is wasteful to do so. Normally the preconditioning operation is handled by a subroutine that takes $r$ as input and returns $p$, and normally the subroutine does this without forming $M$ or $M^{-1}$. For example, a Jacobi preconditioning subroutine will just take each entry of $r$ and divide it by the corresponding main diagonal entry of $A$.

The procedure for applying the Jacobi preconditioner is perfectly obvious. Now let us consider how to apply a more complicated one. Consider any iterative method that is generated by a splitting matrix $M$. Recall that the iteration is defined by $M x^{(k+1)} = N x^{(k)} + b$, where $A = M - N$. If we perform one iteration to get from $x^{(0)}$ to $x^{(1)}$, we have

$$x^{(1)} = x^{(0)} + M^{-1} r^{(0)}, \quad \text{(7.5.7)}$$

where $r^{(0)} = b - Ax^{(0)}$. This was shown in general in Exercise 7.3.20 and for special cases in Exercises 7.2.4, 7.2.12, 7.2.18, 7.2.24, and 7.2.26. For example, if we perform one iteration of SOR, getting from $x^{(0)}$ to $x^{(1)}$, the two iterates are related by (7.7.5.7), where $M = \frac{1}{2} D - E$. But the SOR iteration is executed by sweeping through the $x$ vector, correcting the entries one after the other; we never form or even
Exercise 7.5.11  Our development of Algorithm 7.5.2 assumes that the splitting matrix is positive definite. Nevertheless, there is nothing to stop us from running Algorithm 7.5.2 with a nonsymmetric $M$. In this case we have no justification for the choice of search direction $p = M^{-1}r$, but we are still performing a descent algorithm with exact line searches.

(a) Try using SOR as a preconditioner for steepest descent on the model problem. How does it compare with SSOR? with no preconditioner?

(b) Try a preconditioner that does $m$ sweeps of SOR instead of one. Try $m = 2$, 3, and 4, at least. Notice that as $m$ is increased, the preconditioner gets better. Of course, it also consumes more time.

Exercise 7.5.12  Given a preconditioner or splitting matrix $M$, show that the preconditioned steepest descent algorithm is the same as the iteration generated by the splitting matrix $\frac{1}{\alpha} M$, except that the damping parameter varies from one iteration to the next. (See Exercises 7.3.20 and 7.3.23.)

Exercise 7.5.13  Let $A$ and $M$ be positive definite matrices, and let $M = R^T R$. Define $A_1 = M^{-1} A$ and $A_2 = R^{-T} A R^{-1}$.

(a) Show that $A_1$ and $A_2$ are similar. That is, $A_2 = S^{-1} A_1 S$ for some nonsingular matrix $S$.

(b) Show that Richardson’s method applied to either $A_1$ or $A_2$ will converge if the damping parameter $\omega$ is small enough, and the convergence rate is the same for both. (See Exercise 7.3.19.)

7.6  THE CONJUGATE-GRADIENT METHOD

All of the iterative methods that we have discussed to this point are limited by their lack of memory. Each uses only information about $x^{(k)}$ to get to $x^{(k+1)}$. All information from earlier iterations is forgotten. The conjugate-gradient (CG) method [39] is a simple variation on steepest descent that performs better because it has a memory.

Our approach will be to introduce the algorithm right away, compare and contrast it with the steepest descent method, and observe how well it performs. Once we have done that, we will derive the algorithm and study its theoretical properties.

We begin with the basic CG algorithm with no preconditioner, which is shown in (7.6.1), below. In appearance this algorithm differs only slightly from steepest descent. The computation of $\alpha$ is organized a bit differently, but this difference is cosmetic. The line searches are still exact; the CG algorithm is an instance of Algorithm 7.4.14. Initially $p \leftarrow r$, so the first step is steepest descent. On subsequent
Additional Exercises

Exercise 7.6.6 Write a program that applies the conjugate-gradient algorithm with the SSOR preconditioner to the model problem (7.1.8). The instructions in Exercises 7.4.19 and 7.5.9 for executing the steps \( p \leftarrow Aq \) and \( s \leftarrow M^{-1}r \) are also applicable here. To test your program, use the same \( f \), \( g \), and \( u^{(0)} \) as in Exercise 7.4.19. Terminate iterations when \( s^{(k+1)}T^{(k+1)}r^{(k+1)} < \epsilon^2 s^{(0)}T^{(0)}r^{(0)} \), where \( \epsilon = 10^{-8} \) (see Exercise 7.5.10).

(a) Try out your code under the conditions of Example 7.6.5. You should get similar, but not identical, results, since you are using a different stopping criterion.

(b) Consider the model problem with \( h = 1/160 \). What is the dimension of the associated matrix? What is its bandwidth (assuming the natural row ordering)? Approximately what fraction of the matrix entries are nonzero?

(c) With \( h = 1/160 \), try \( \omega = 1.7, 1.75, 1.8, 1.85, 1.9, \) and \( 1.95 \). Observe that the preconditioner performs satisfactorily over this range of \( \omega \) values.

(d) Apply SOR with \( h = 1/160 \) for comparison against preconditioned CG. (Use your code from Exercise 7.2.23.) Try \( \omega = 1.9, 1.95, \) and \( 1.962 \) (optimal for \( h = 1/160 \)). Note the sensitivity to the choice of \( \omega \). How does SOR compare with CG preconditioned by SSOR?

(e) Choose a value of \( \omega \), say \( 1.8 \) or \( 1.9 \), and run your preconditioned CG code with that chosen \( \omega \) and \( h = 1/10, 1/20, 1/40, 1/80 \), and \( 1/160 \). Record the number of iterations \( i_h \) in each case. Notice that \( i_h \) grows as \( h \) is decreased, but the growth is moderate. (Optimally we would like to see no growth. There is still room for improvement.)

Exercise 7.6.7 Try replacing SSOR by SOR as a preconditioner for CG. How does it work?

Exercise 7.6.8 A conjugate-gradient routine \texttt{pcg} is supplied with MATLAB. Read about \texttt{pcg} in the MATLAB help facilities, and try it out on a discrete Laplacian matrix. Use commands like the following:

\[
\begin{align*}
m &= 40; \\
A &= 	ext{delsq}(	ext{numgrid}('N',m+1)); \\
n &= 	ext{size}(A,1) \\
sol &= 	ext{ones}(n,1); \% 'all ones' solution vector. \\
b &= A*sol; \% right-hand side vector. \\
tol &= 1e-12; \text{maxit} = 1000; \\
[x,flag,relres,iter] &= \text{pcg}(A,b,tol,maxit); \\
err &= \text{norm}(\text{sol}-x)
\end{align*}
\]

Th
The matrix $A$ generated here is exactly the matrix of our model problem (7.1.8) with $h = 1/m$. A "nested dissection" ordering is used [30]. The most common way to use pcg is to supply the matrix $A$ explicitly in MATLAB's sparse matrix format, and that is what we are doing here. This is far from the most efficient way to use this matrix to execute the CG algorithm, but it is convenient and works reasonably well on moderate-sized problems. Here we are running pcg with a residual tolerance of $10^{-12}$, a maximum of 1000 on the number of iterations, and no preconditioner. Try this out with various values of $m$, e.g. $m = 40, 80, 160$, and check how many iterations are needed in each case.

Exercise 7.6.9 MATLAB also provides a routine cholinc for computing incomplete Cholesky preconditioners, (see Section 7.5). Read about cholinc in the MATLAB help facilities. Repeat Exercise 7.6.8 using an incomplete Cholesky preconditioner $M = R^T R$ based on a drop tolerance of $10^{-2}$:

```matlab
droptol = 1e-2;
R = cholinc(A,droptol);
spy(A), spy(R), spy(R'*R),
[x,flag,relres,iter] = pcg(A,b,tol,maxit,R',R);
```

Compare your results with those of Exercise 7.6.8.

Exercise 7.6.10 Repeat Exercise 7.6.9 with fixed $m$, say $m = 80$, varying the drop tolerance. Try, for example, $10^0, 10^{-1}, 10^{-2},$ and $10^{-3}$. Check the effect on the number of CG iterations needed and the number of nonzeros in the incomplete Cholesky factor. Compare with the number of zeros in the complete Cholesky factor of $A$, as computed by chol.

Exercise 7.6.11 Repeat Exercises 7.6.8, 7.6.9, and 7.6.10, using Wathen matrices of various sizes. Type $A = gallery(\text{\textquoteleft wathen\textquoteleft},20,15)$, for example. For a bigger matrix replace the 20 and 15 by larger numbers. Type help private/wathen for information about Wathen matrices.

### 7.7 DERIVATION OF THE CG ALGORITHM

The conjugate-gradient algorithm is a descent method that performs exact line searches. To complete the formal definition of the algorithm, we need only specify how the search directions are chosen. After some preparation we will be in a position to derive the search directions in a natural way.

In Chapter 3 we introduced the inner product

$$ (x,y) = \sum_{i=1}^{n} x_i y_i = y^T x = x^T y $$

(7.7.1)

and used it extensively in our discussion of the least-squares problem. It was the only inner product we needed at that time, and we spoke of it as if it were the only inner