1

Introduction

In this book I present an overview of a number of related iterative methods for the solution of linear systems of equations. These methods are so-called Krylov projection type methods and they include popular methods such as Conjugate Gradients, MINRES, SYMMLQ, Bi-Conjugate Gradients, QMR, Bi-CGSTAB, CGS, LSQR, and GMRES. I will show how these methods can be derived from simple basic iteration formulas and how they are related. My focus is on the ideas behind the derivation of these methods, rather than on a complete presentation of various aspects and theoretical properties.

In the text there are a large number of references for more detailed information. Iterative methods form a rich and lively area of research and it is not surprising that this has already led to a number of books. The first book devoted entirely to the subject was published by Varga [212], it contains much of the theory that is still relevant, but it does not deal with the Krylov subspace methods (which were not yet popular at the time).

Other books that should be mentioned in the context of Krylov subspace methods are the 'Templates' book [20] and Greenbaum's book [101]. The Templates are a good source of information on the algorithmic aspects of the iterative methods and Greenbaum's text can be seen as the theoretical background for the Templates.

Axelsson [10] published a book that gave much attention to preconditioning aspects, in particular all sorts of variants of (block and modified) incomplete decompositions. The book by Saad [168] is also a good source of information on preconditioners, with much inside experience for such methods as threshold ILU. Of course, GMRES receives much attention in [168], together with variants of the method. Kelley [126] considers a few of the most popular Krylov methods and discusses how to use them for nonlinear systems. Meurant [144] covers the

theory of most of the best algorithms so far known. It contains extensive material on domain decomposition and multilevel type preconditioners. Meurant's book is also very useful as a source text: it contains as many as 1368 references to literature. Brezinski's book [31] emphasizes the relation between (Krylov) subspace methods and extrapolation methods. He also considers various forms of hybrid methods and discusses different approaches for nonlinear systems. Implementation aspects for modern High-Performance computers are discussed in detail in [61].

For general background on linear algebra for numerical applications see [98, 181], and for the effects of finite precision, for general linear systems, I refer to [116] (as a modern successor of Wilkinson's book [222]).

Some useful state of the art papers have appeared; I mention papers on the history of iterative methods by Golub and van der Vorst [97], and Saad and van der Vorst [170]. An overview on parallelizable aspects of sparse matrix techniques is presented in [70]. A state-of-the-art overview for preconditioners is presented in [22].

Iterative methods are often used in combination with so-called preconditioning operators (easily invertible approximations for the operator of the system to be solved). I will give a brief overview of the various preconditioners that exist.

The purpose of this book is to make the reader familiar with the ideas and the usage of iterative methods. I expect that then a correct choice of method can be made for a particular class of problems. The book will also provide guidance on how to tune these methods, particularly for the selection or construction of effective preconditioners.

For the application of iterative schemes we usually have linear sparse systems in mind, for instance linear systems arising in the finite element or finite difference approximations of (systems of) partial differential equations. However, the structure of the operators plays no explicit role in any of these schemes, which may also be used successfully to solve certain large dense linear systems. Depending on the situation, this may be attractive in terms of numbers of floating point operations.

I will also pay some attention to the implementation aspects of these methods, especially for parallel computers.

Before I start the actual discussion of iterative methods, I will first give a motivation for their use. As we will see, iterative methods are not only great fun to play with and interesting objects for analysis, but they are really useful in many situations. For truly large problems they may sometimes offer the only way towards a solution, as we will see.

Figure 1.1. The computational grid for an ocean flow.

1.1 On the origin of iterative methods

In scientific computing most computational time is spent in solving systems of linear equations. These systems can be quite large, for instance as in computational fluid flow problems, where each equation describes how the value of a local unknown parameter (for example the local velocity of the flow) depends on (unknown) values in the near neighbourhood.

The actual computation is restricted to values on a previously constructed grid and the number of gridpoints determines the dimensions of the linear system. In Figure. 1.1 we see such a grid for the computation of two-dimensional ocean flows. Each gridpoint is associated with one or more unknowns and with equations that describe how these unknowns are related to unknowns for neighbouring gridpoints. These relations are dictated by the physical model. Because many gridpoints are necessary in order to have a realistic computational model, we will as a consequence have many equations. A nice property of these linear systems is that each equation contains only a few unknowns. The matrix of the system contains mainly zeros. This property will be of great importance for the efficiency of solution methods, as we will see later.

We see that the grid consists of four differently represented subgrids. The reason for this is that, in the actual computations for this problem, we had to do the work in parallel: in this case on four parallel computers. This made it possible to do the work in an acceptably short time, which is convenient for model studies. We will see that most of the methods that we will describe lend themselves to parallel computation.

As we will see, the process of solving the unknowns from these large linear systems involves much computational work. The obvious approach via direct Gaussian elimination is often not attractive. This was already recognized by the great Gauss himself, in 1823, albeit for different reasons to those in the present circumstances [93]. In that year he proposed an iterative method for the solution of four equations with four unknowns, arising from triangular measurements.

In order to appreciate his way of computing, we start with the familiar (Gaussian) elimination process. As an example we consider the small linear system

$$\begin{bmatrix} 10 & 0 & 1 \\ \frac{1}{2} & 7 & 1 \\ 1 & 0 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 21 \\ 9 \\ 8 \end{bmatrix}.$$

The elimination process is as follows. We subtract $\frac{1}{20}$ times the first row from the second row and then $\frac{1}{10}$ times the first row from the third row. After this we have zeros in the first column below the diagonal and the system becomes

$$\begin{bmatrix} 10 & 0 & 1 \\ 0 & 7 & 1 - \frac{1}{20} \\ 0 & 0 & 6 - \frac{1}{10} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 21 \\ 9 - \frac{21}{20} \\ 8 - \frac{21}{10} \end{bmatrix}.$$

As a coincidence we also have a zero element in the second column below the diagonal, and now we can solve the system without much effort. It leads to the solution $x_3 = 1, x_2 = 1$, and $x_1 = 2$. Note that we have used exact computation. This is not a problem in this case, which has been designed to have a 'nice' solution. However, in more realistic situations, we may have non-integer values and then exact computation may lead to significant computational effort for a human being. It is not so easy to avoid human errors and after checking that the computed erroneous solution does not satisfy the initial system, it is not easy to find the place where the error occurred. Gauss suffered from this in his computations. He had a good physical intuition and he knew that the solution of his system should have components of about the same order of magnitude. Because his matrices had strongly dominating elements on the diagonal, he knew that the main contribution in the right-hand side came from

the components in the solution that had been multiplied by a diagonal element: in our example $10x_1$, $7x_2$, and $6x_3$. This implies that if we neglect the off-diagonal elements in our system,

$$\begin{bmatrix} 10 & 0 & 0 \\ 0 & 7 & 0 \\ 0 & 0 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 21 \\ 9 \\ 8 \end{bmatrix},$$

we may still expect, from this perturbed system, a fairly good approximation for the solution of the unperturbed system; in our example: $x_1 = 2.1$, $x_2 = \frac{9}{7}$, and $x_3 = \frac{8}{6}$. Indeed, this is a crude approximation for the solution that we want. This way of approximation is still popular; it is known as a Gauss–Jacobi approximation, because the mathematician-astronomer Jacobi used it for the computation of perturbations in the orbits of the planets in our solar system.

Gauss made another intelligent improvement. He observed that we can approximate the original system better if we only replace nonzero elements in the strict upper triangular part. This leads to

$$\begin{bmatrix} 10 & 0 & 0 \\ \frac{1}{2} & 7 & 0 \\ 1 & 0 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 21 \\ 9 \\ 8 \end{bmatrix}.$$

This system has the solution $x_1 = 2.1$, $x_2 = \frac{7.95}{7}$, and $x_3 = \frac{5.9}{6}$. Indeed, this leads to an improvement (it should be noted that this is not always the case; there are situations where this approach does not lead to an improvement). The approach is known as the Gauss–Seidel approximation.

Altogether, we have obtained a crude approximated solution for our small system for only a small reduction in the computational costs. At this point it is good to discuss the computational complexity. For a system with n equations and n unknowns we need $2(n-1)^2$ operations to create zeros in the first column (if we ignore possible, already present, zeros). Then for the second column we need $2(n-2)^2$ operations. From this we conclude that for the elimination of all elements in the lower triangular part, we need about $\frac{2}{3}n^3$ operations. The cost of solving the resulting upper triangular system again requires roughly n^2 operations, which is a relatively minor cost for larger values of n. We may conclude that the cost of obtaining the exact solution is proportional to n^3. It is easy to see that the cost of computing only the Gauss–Seidel approximation is proportional to n^2 and it may be seen that this promises great advantages for larger systems.

The question now arises – how is the obtained approximated solution improved at relatively low costs? Of course, Gauss had also considered this aspect. In order to explain his approach, I will use matrix notation. This was not yet invented in Gauss's time and the lack of it makes the reading of his original description not so easy. We will write the system as

$$Ax = b,$$

with

$$A = \begin{bmatrix} 10 & 0 & 1 \\ \frac{1}{2} & 7 & 1 \\ 1 & 0 & 6 \end{bmatrix},$$

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 21 \\ 9 \\ 8 \end{bmatrix}.$$

The lower triangular part of A is denoted by L:

$$L = \begin{bmatrix} 10 & 0 & 0 \\ \frac{1}{2} & 7 & 0 \\ 1 & 0 & 6 \end{bmatrix}.$$

The Gauss–Seidel approximation is then obtained by solving the system

$$L\widetilde{x} = b.$$

For a correction to this solution we look for the 'missing' part Δx:

$$A(\widetilde{x} + \Delta x) = b,$$

and this missing part satisfies the equation

$$A\Delta x = b - A\widetilde{x} \equiv r.$$

It is now an obvious idea to compute an approximation for Δx again with a Gauss–Seidel approximation, that is we compute $\widetilde{\Delta x}$ from

$$L\widetilde{\Delta x} = r,$$

and we correct our first approximation with this approximated correction $\widetilde{x}$:

$$\widetilde{\widetilde{x}} = \widetilde{x} + \widetilde{\Delta x}.$$

Table 1.1. Results for three Gauss–Seidel iterations

iteration	1	2	3
x_1	2.1000	2.0017	2.000028
x_2	1.1357	1.0023	1.000038
x_3	0.9833	0.9997	0.999995

Of course, we can repeat this trick and that leads to the following simple iteration procedure:

$$x^{(i+1)} = x^{(i)} + L^{-1}(b - Ax^{(i)}),$$

where the vector $y = L^{-1}(b - Ax^{(i)})$ is computed by solving

$$Ly = b - Ax^{(i)}.$$

We try this process for our little linear system. In the absence of further information on the solution, we start with $x^{(0)} = 0$. In Table 1.1 we display the results for the first three iteration steps.

We observe that in this case we improve the solution by about two decimals per iteration. Of course, this is not always the case. It depends on how strongly the diagonal elements dominate. For instance, for the ocean flow problems we have almost no diagonal dominance and Gauss–Seidel iteration is so slow that it is not practical in this bare form.

The computational costs per iteration step amount to roughly $2n^2$ operations (additions, subtractions, multiplications) for the computation of $Ax^{(i)}$, plus n^2 operations for the solution of the lower triangular system with L: in total $\approx 3n^2$ operations per iteration step. Solution via the direct Gaussian elimination process takes $\approx \frac{2}{3}n^3$ operations. This implies a gain in efficiency if we are satisfied with the approximations and if these are obtained after less than

$$\left(\frac{2}{3}n^3\right)/(3n^2) = \frac{2}{9}n$$

iterations.

Computation with this iteration method was very attractive for Gauss, not because of efficiency reasons but mainly because he did not have to compute the approximated solutions accurately. A few decimal places were sufficient. Unintentional errors and rounding errors 'correct themselves' in the later iterations. Another advantage is that the residual $b - Ax^{(i)}$ has to be computed in each step, so that we can see at a glance how well the computed approximation

satisfies the system. In a letter to his colleague Gerling, Gauss was elated over this process and mentioned that the computations could be undertaken even when a person is half asleep or thinking of more important things.

The linear systems that Gauss had to solve were strongly diagonally dominant and for that reason he could observe fast convergence. The Gauss–Seidel iteration process is much too slow for the very large linear systems that we see in many applications. For this reason there has been much research into faster methods and we will see the results of this later in this text.

1.2 Further arguments for iterative methods

For the solution of a linear system $Ax = b$, with A a nonsingular n by n matrix, we have the choice between direct and iterative methods.

The usual pro-arguments for iterative methods are based on economy of computer storage and (sometimes) CPU time. On the con side, it should be noted that the usage of iterative methods requires some expertise. If CPU-time and computer storage are not really at stake, then it would be unwise to consider iterative methods for the solution of a given linear system. The question remains whether there are situations where iterative solution methods are really preferable. In this section I will try to substantiate the pro-arguments; the con-arguments will appear in my more detailed presentation of iterative methods. I hope that the reader will feel sufficiently familiar, after reading these notes, with some of the more popular iterative methods in order to make a proper choice for the solving of classes of linear systems.

Dense linear systems, and sparse systems with a suitable nonzero structure, are most often solved by a so-called direct method, such as Gaussian elimination. A direct method leads, in the absence of rounding errors, to the exact solution of the given linear system in a finite and fixed amount of work. Rounding errors can be handled fairly well by pivoting strategies. Problems arise when the direct solution scheme becomes too expensive for the task. For instance, the elimination steps in Gaussian elimination may cause some zero entries of a sparse matrix to become nonzero entries, and nonzero entries require storage as well as CPU time. This is what may make Gaussian elimination, even with strategies for the reduction of the so-called fill-in, expensive.

In order to get a more quantitative impression of this, we consider a sparse system related to discretization of a second order PDE over a (not necessarily regular) grid, with about m unknowns per dimension. Think, for instance, of a finite element discretization over an irregular grid [159]. In a 3D situation

this leads typically to a bandwidth $\sim n^{\frac{2}{3}}$ ($\approx m^2$ and $m^3 \approx n$, where $1/m$ is the (average) gridsize).

Gaussian elimination is carried out in two steps: first the matrix A is factored into a lower triangular matrix L, and an upper triangular matrix U (after suitable permutations of rows and columns):

$$A = LU.$$

When taking proper account of the band structure, the number of flops is then usually $\mathcal{O}(nm^4) \sim n^{2\frac{1}{3}}$ [98, 67]. We make the caveat 'usually', because it may happen that fill-in is very limited when the sparsity pattern of the matrix is special.

For 2D problems the bandwidth is $\sim n^{\frac{1}{2}}$, so that the number of flops for a direct method then varies with n^2.

Then, in the second step, we have to solve x from $LUx = b$, which, again, is done in two steps:

(a) first solve y from $Ly = b$,
(b) then solve x from $Ux = y$.

The LU factorization is the expensive part of the computational process; the solution of the two triangular systems is usually a minor cost item. If many systems with different right-hand sides have to be solved, then the matrix has to be factored only once, after which the cost for solving each system will vary with $n^{\frac{5}{3}}$ for 3D problems, and with $n^{\frac{3}{2}}$ for 2D problems.

In order to be able to quantify the amount of work for iterative methods, we have to be a little more specific. Let us assume that the given matrix is symmetric positive definite, in which case we may use the Conjugate Gradient (CG) method. The error reduction per iteration step of CG is $\sim \frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}$, with $\kappa = \|A\|_2\|A^{-1}\|_2$ [44, 8, 98].

For discretized second order PDEs over grids with gridsize $\frac{1}{m}$, it can be shown that $\kappa \sim m^2$ (see, for instance, [159]). Hence, for 3D problems we have that $\kappa \sim n^{\frac{2}{3}}$, and for 2D problems: $\kappa \sim n$. In order to have an error reduction by a factor of ϵ, the number j of iteration steps must satisfy

$$\left(\frac{1-\frac{1}{\sqrt{\kappa}}}{1+\frac{1}{\sqrt{\kappa}}}\right)^j \approx \left(1-\frac{2}{\sqrt{\kappa}}\right)^j \approx \mathrm{e}^{-\frac{2j}{\sqrt{\kappa}}} < \epsilon.$$

For 3D problems, it follows that

$$j \sim -\frac{\log\epsilon}{2}\sqrt{\kappa} \approx -\frac{\log\epsilon}{2}n^{\frac{1}{3}},$$

whereas for 2D problems,

$$j \approx -\frac{\log \epsilon}{2} n^{\frac{1}{2}}.$$

If we assume the number of flops per iteration to be $\sim fn$ (f stands for the average number of nonzeros per row of the matrix and the overhead per unknown introduced by the iterative scheme), then the required number of flops for a reduction of the initial error with ϵ is

(a) for 3D problems: $\sim -fn^{\frac{4}{3}} \log \epsilon$, and
(b) for 2D problems: $\sim -fn^{\frac{3}{2}} \log \epsilon$.

f is typically a modest number, say of order 10–15.

From comparing the flops counts for the direct scheme with those for the iterative CG method we conclude that the CG method may be preferable if we have to solve one system at a time, and if n is large, or f is small, or ϵ is modest.

If we have to solve many systems $Ax = b_k$ with different right-hand sides b_k, and if we assume their number to be so large that the costs for constructing the LU factorization of A is relatively small per system, then it seems likely that direct methods will be more efficient for 2D problems. For 3D problems this is unlikely, because the flops counts for the two triangular solves associated with a direct solution method are proportional to $n^{\frac{5}{3}}$, whereas the number of flops for the iterative solver (for the model situation) varies in the same way as $n^{\frac{4}{3}}$.

1.3 An example

The above arguments are quite nicely illustrated by observations made by Horst Simon [173]. He predicted that by now we will have to solve routinely linear problems with some 5×10^9 unknowns. From extrapolation of the CPU times observed for a characteristic model problem, he estimated the CPU time for the most efficient direct method as 520 040 years, provided that the computation can be carried out at a speed of 1 TFLOPS.

On the other hand, the extrapolated guess for the CPU time with preconditioned conjugate gradients, still assuming a processing speed of 1 TFLOPS, is 575 seconds. As we will see, the processing speed for iterative methods may be a factor lower than for direct methods, but, nevertheless, it is obvious that the differences in CPU time requirements are gigantic. The ratio of the two times is of order n, just as we might have expected from our previous arguments.

Also the requirements for memory space for the iterative methods are typically smaller by orders of magnitude. This is often the argument for the use