Preconditioned Conjugate Gradient Methods for Serial and Parallel Computers

Andrew D. Pollard

September 1992

Submitted to the

Department of Mathematics,

University of Reading,

in partial fulfillment of the requirements for the

Degree of Master of Science

Abstract

Various Preconditioned Conjugate Gradient Methods are investigated for the solution of matrix systems arising from the discretisation of a simple two dimensional partial differential equation of interest to the Oil industry.

Consideration is given to the implementation of these methods on serial computers and efficient parallel implementation on a network of transputers.

Acknowledgements

I would like to thank Dr.N.K.Nichols, Dr.P.K.Sweby and Dr.M.J.Baines of the Mathematics Department, University of Reading, and Dr.J.J.Barley of BP Research, Sunbury, for their help, support and advice during this work.

I would also like to acknowledge the financial support of the SERC.

Contents

List of Figures

List of Tables

Notation

Symbol	**Meaning**
A	matrix arising from problem discretisation
$\tilde{A}$	preconditioned matrix $= M^{-1}A$
$\underline{a}^{(x)}$	diagonals of A
α	parameter in CG algorithm
$\underline{b}, \underline{f}$	right hand side of matrix equation
$\tilde{\underline{b}}$	preconditioned right hand side of matrix equation
β	parameter in CG algorithm
D	diagonal of matrix A
Δx	width of grid block
Δy	height of grid block
E_p	parallel efficiency for p processors
f_{ij}	approximation to right hand side of matrix problem
$\underline{g}$	gravitational acceleration
i,j	counters
k,K	permeability of medium
κ	condition number of matrix
L	lower triangular part of matrix, Cholesky factor of matrix
M	preconditioning matrix
n	dimension of A, $= nx \times ny$
nx	number of grid blocks in x-direction
ny	number of grid blocks in y-direction

Symbol	**Meaning**
$\underline{n}$	outward normal vector to region Ω
Ω	region in which problem is being solved
ω	SOR relaxation parameter
p	number of processors/subproblems, pressure of fluid
p_{ij}	approximation to pressure of fluid
p_{BH}	pressure at injection well
$\underline{p}$,$\underline{x}$	solution vector to matrix problem
$\underline{p}$	search direction vector in CG algorithm
ρ	density of fluid, spectral radius of matrix
q	production rate of fluid at production well
$\mathbb{R}$	set of real numbers
s,s_i	block size
S_p,S_p^*	parallel speedup with p processors
U	upper triangular part of matrix
$\underline{u}$	Darcy velocity of fluid
μ	viscosity of fluid
γ	transmissibility at injection well, parameters in polynomial approx.
Γ_{ij}	grid block region
ϕ	porosity of medium, functional to be minimised
λ	mobility of fluid, eigenvalues of matrix
λ_{ij}	approximations to mobilities

Chapter 1

Introduction

There are basically two approaches to obtaining the solution of a matrix system of equations – direct methods or iterative methods. Direct methods solve the system in a known (finite) number of steps, and any errors incurred arise only from the use of finite precision arithmetic. They are often impractical when the matrix is large and sparse, due to fill-in destroying the sparsity. Iterative methods, on the other hand, generate a sequence of approximate solutions that (should) converge to the solution of the problem. These methods are more applicable to large, sparse systems, essentially due to the fact that only a matrix-vector product is required, thus the matrix need not be explicitly stored.

This dissertation deals with the Conjugate Gradient and Preconditioned Conjugate Gradient iterative methods, with respect to large, sparse matrix systems arising from the discretisation of two dimensional fluid flow obeying Darcy's Law, such as in Oil Recovery. Various types of preconditioning are examined for implementation on serial computers and efficient implementation on a parallel system comprising of a network of transputers.

In Chapter 2, the governing pressure equation for Oil Recovery is presented, a simplified version derived for the purposes of the dissertation, the finite difference approximation used to generate the matrix system derived, and some sample problems given. In Chapter 3, the Conjugate Gradient and Preconditioned Conjugate Gradient algorithms are presented, with the description of a number of preconditioning strategies. Chapter 4 indicates, in more detail, the implementation of some of the preconditioning strategies on serial computers, and describes the metrics that are used to compare them. The architecture of the transputer system, and the parallel programming primitives needed to use it, are described in Chapter 5, as well as the parallel implementation of the Conjugate Gradient and Preconditioned Conjugate Gradient algorithms with their preconditioners, and the metrics for their performance. Finally, in Chapter 6, closing comments are made and some conclusions drawn about the optimum preconditioning strategies.

Appendix A describes some basic matrix theory that is used throughout the dissertation.

Chapter 2

Matrix System Generation

2.1 Oil Recovery [2]

Oil reservoirs are beds of porous rock saturated with various fluids and gases. There are three stages to the recovery of the oil. The primary stage involves boring into the rocks and letting the natural pressure of the fluids in the rocks to force the oil to the surface. The secondary stage, reached after the production from the primary stage has decreased substantially, has other fluids injected in an attempt to displace the oil towards the production well. Typical injection fluids are water or steam. There may still be a substantial amount of the original reservoir oil left in the reservoir after secondary recovery, and a tertiary recovery process, such as surfactant flooding [2], can be used to extract some of the remaining oil.

2.1.1 Modelling the Recovery Process

Here, the secondary stage of recovery is investigated, where a fluid is injected into the injection well to keep the pressure constant there. This allows the use of a single phase (one component) flow model [2]. This model gives the 'mass continuity equation', i.e. the equation representing the fact that the fluid cannot disappear,

$$-\underline{\nabla}.(\rho\underline{u}) = \frac{\partial}{\partial t}(\rho\phi) + \tilde{f} \qquad (2.1)$$

where ρ is the density of the fluid, ϕ is the porosity of the medium that the fluid is flowing in, i.e. a representative measure of how much space in the medium is available to hold the fluid, $\tilde{f}$ is the forcing function representing the effect of the injection and production wells, i.e. the rate of the fluid leaving the medium, and $\underline{u}$ is the velocity of the fluid.

As well as the mass continuity equation, a relationship between the flow rate and the pressure gradient is required. Such a relationship, albeit an empirical one, was developed by Darcy (1856) for a single phase flow, i.e.

$$\underline{u} = -\frac{K}{\mu}(\underline{\nabla}p + \rho\underline{g}) \qquad (2.2)$$

where $\underline{u}$ is the Darcy velocity, p is the pressure of the fluid, K is the absolute permeability tensor, i.e. the willingness of the medium to allow fluids to flow through it, μ is the viscosity of the fluid, i.e. the internal resistance of the fluid to flow within itself, and $\underline{g}$ is the gravitational acceleration. The permeability tensor has to be determined experimentally. In most practical problems it is possible (or necessary) to assume that K is a diagonal tensor, e.g. in two dimensions

$$K = \begin{bmatrix} k_x & 0 \\ 0 & k_y \end{bmatrix}$$

If $k_x = k_y$, the medium is called isotropic, otherwise it is anisotropic.

By combining (2.1) and (2.2), our fluid flow equation becomes

$$\underline{\nabla}.(\frac{\rho K}{\mu}(\underline{\nabla} p + \rho \underline{g})) = \frac{\partial}{\partial t}(\rho\phi) + \tilde{f} \tag{2.3}$$

Then, by making some simplifying assumptions; the fluid density ρ remains constant, i.e. there are no temperature dependent changes, there is no gravitational effect, i.e. $\underline{g} = 0$, no time dependence, i.e. steady state version, an isotropic medium, i.e. $K = kI$, and defining the mobility λ to be $\frac{k}{\mu}$, i.e. the ease with which the fluid moves through the medium, we change equation (2.3) into

$$\underline{\nabla}.(\lambda \underline{\nabla} p) = f \tag{2.4}$$

If we assume there is a constant pressure injection well and a constant rate production well, the forcing function f can be represented by

$$f = -\gamma(p_{bh} - p)\delta_I + q\delta_P \tag{2.5}$$

where δ_I and δ_P are Kronecker Deltas[1] for the injection and production wells respectively, γ is the transmissibility [2] of the injector, q is the production rate of the producer, and p_{bh} is the pressure of the injection well.

The region in which (2.4) is solved is called Ω. Assuming that the region Ω is finite, we require a boundary condition. The simplest boundary condition is a 'no flow' boundary, i.e. the region Ω is surrounded by an area of zero permeability, so that no fluid can flow out of Ω. This condition can be represented by

$$\underline{u}.\underline{n} = 0 \tag{2.6}$$

[1]If $Q \in \Omega$, then the Kronecker Delta δ_Q is defined by $\forall \underline{x} \in \Omega$, $\delta_Q(\underline{x}) = \begin{cases} 1 & \underline{x} = Q \\ 0 & \underline{x} \neq Q \end{cases}$

where $\underline{u}$ is the above Darcy velocity (2.2) and $\underline{n}$ is the outward normal to the region Ω. With the above assumptions, the boundary condition (2.6) can be written

$$\underline{u}.\underline{n} = -\lambda(\underline{\nabla} p.\underline{n}) = -\lambda\frac{\partial p}{\partial n} = 0 \tag{2.7}$$

and thus can be satisfied with either $\lambda = 0$ or $\frac{\partial p}{\partial n} = 0$ on the boundary $\partial\Omega$.

2.2 Discretisation

For the purposes of this dissertation, the equation (2.4) is solved on a two-dimensional rectangular region Ω, with dimension X in the x-direction, and Y in the y-direction.

The equation (2.4) is discretised using a finite difference approximation, as opposed to a finite element approximation.

2.2.1 Grid

The region, Ω, is divided up into a computational grid with nx grid blocks of width $\Delta x = X/nx$ in the x-direction, and ny grid blocks of height $\Delta y = Y/ny$ in the y-direction. Each grid block, Γ_{ij}, is defined by

$$\Gamma_{ij} = [(i-1)\Delta x, i\Delta x] \times [(j-1)\Delta y, j\Delta y]$$

where $i = 1, 2, \ldots, nx$ and $j = 1, 2, \ldots, ny$. On each grid block Γ_{ij}, the pressure p_{ij} is defined, at the centre of the block, to be the cell average over Γ_{ij}, i.e.

$$p_{ij} = \frac{1}{\Delta x \Delta y}\iint_{\Gamma_{ij}} p(x,y)\, d\Gamma_{ij}$$

The mobilities λ_{ij} are also defined at the centre of each block, although as is shown later, their values are needed on the block boundaries, i.e. $\lambda_{i-\frac{1}{2},j}$, $\lambda_{i+\frac{1}{2},j}$, $\lambda_{i,j-\frac{1}{2}}$, and $\lambda_{i,j+\frac{1}{2}}$ are required.

2.2.2 Finite Difference Operator

By considering each grid block Γ_{ij}, and integrating equation (2.4) over each grid block, then approximating the integrals, we obtain an approximation to the original partial differential equation (2.4). This is known as the 'integration method'. In two dimensions it becomes

$$\iint_{\Gamma_{ij}} \underline{\nabla}.(\lambda\underline{\nabla}p)\, d\Gamma_{ij} = \iint_{\Gamma_{ij}} \frac{\partial}{\partial x}(\lambda\frac{\partial p}{\partial x}) + \frac{\partial}{\partial y}(\lambda\frac{\partial p}{\partial y})\, d\Gamma_{ij} = \iint_{\Gamma_{ij}} f\, d\Gamma_{ij} \qquad (2.8)$$

Using Green's Theorem (in two dimensions),

$$\iint_R (\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y})\, dR = \int_{\partial R} -v\, dx + u\, dy$$

we can transform the differential equation side of (2.8) into a boundary integral around the grid block Γ_{ij}. Labelling the grid block as indicated in Figure 2.1, where $\partial\Gamma_{ij}$ is the path $ABCD$, we obtain

$$\begin{aligned} \iint_{\Gamma_{ij}} \frac{\partial}{\partial x}(\lambda\frac{\partial p}{\partial x}) + \frac{\partial}{\partial y}(\lambda\frac{\partial p}{\partial y})\, d\Gamma_{ij} &= \int_{\partial\Gamma_{ij}} -\lambda\frac{\partial p}{\partial y}\, dx + \lambda\frac{\partial p}{\partial x}\, dy \\ &= -\int_A^B \lambda\frac{\partial p}{\partial y}\, dx + \int_B^C \lambda\frac{\partial p}{\partial x}\, dy \qquad (2.9) \\ &\quad -\int_C^D \lambda\frac{\partial p}{\partial y}\, dx + \int_D^A \lambda\frac{\partial p}{\partial x}\, dy \end{aligned}$$

A very simple way of approximating $\int_A^B q\, dx$ is to give it the value of the length of the interval AB (Δ), multiplied by the value of q at the midpoint C (q_C) of the interval, e.g.

$$\int_A^B q\, dx \approx \Delta q_C \qquad (2.10)$$

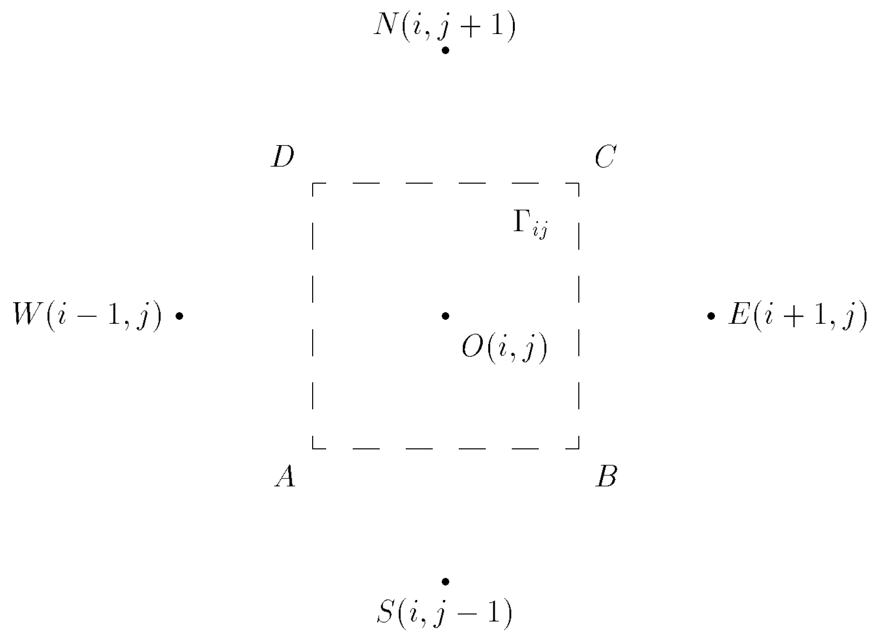


Figure 2.1: Labelling the Grid Block Γ_{ij}

One way approximate the value of a derivative at a point is to use central differences, e.g. the value of $\frac{\partial q}{\partial x}$ at a point C, the midpoint of AB, assuming that AB is in the direction of increasing x, is

$$\frac{\partial q}{\partial x}|_C = \frac{q_B - q_A}{\Delta} \tag{2.11}$$

Thus, by using equations (2.10) and (2.11), we have

$$\begin{array}{lclcl} \int_A^B \lambda \frac{\partial p}{\partial y}\, dx & \approx & (\lambda \frac{\partial p}{\partial y})|_{i,j-\frac{1}{2}} \Delta x & \approx & \lambda_{i,j-\frac{1}{2}} (\frac{p_O - p_S}{\Delta y}) \Delta x \\ \int_C^D \lambda \frac{\partial p}{\partial y}\, dx & \approx & -(\lambda \frac{\partial p}{\partial y})|_{i,j+\frac{1}{2}} \Delta x & \approx & -\lambda_{i,j+\frac{1}{2}} (\frac{p_N - p_O}{\Delta y}) \Delta x \\ \int_B^C \lambda \frac{\partial p}{\partial x}\, dy & \approx & (\lambda \frac{\partial p}{\partial x})|_{i+\frac{1}{2},j} \Delta y & \approx & \lambda_{i,j+\frac{1}{2}} (\frac{p_E - p_O}{\Delta x}) \Delta y \\ \int_D^A \lambda \frac{\partial p}{\partial x}\, dy & \approx & -(\lambda \frac{\partial p}{\partial x})|_{i-\frac{1}{2},j} \Delta y & \approx & -\lambda_{i,j-\frac{1}{2}} (\frac{p_O - p_W}{\Delta x}) \Delta y \end{array} \tag{2.12}$$

where p_* is the value of p at the point $*$. Combining equations (2.8), (2.9) and (2.12), we obtain

$$\iint_{\Gamma_{ij}} \frac{\partial}{\partial x}(\lambda \frac{\partial p}{\partial x}) + \frac{\partial}{\partial y}(\lambda \frac{\partial p}{\partial y})\, d\Gamma_{ij} \approx \lambda_N p_N + \lambda_E p_E - \lambda_O p_O + \lambda_W p_W + \lambda_S p_S \tag{2.13}$$

where

$$\begin{aligned}
\lambda_N &= \lambda_{i,j+\frac{1}{2}} \frac{\Delta x}{\Delta y} \\
\lambda_E &= \lambda_{i+\frac{1}{2},j} \frac{\Delta y}{\Delta x} \\
\lambda_O &= \lambda_{i,j+\frac{1}{2}} \frac{\Delta x}{\Delta y} + \lambda_{i+\frac{1}{2},j} \frac{\Delta y}{\Delta x} + \lambda_{i-\frac{1}{2},j} \frac{\Delta y}{\Delta x} + \lambda_{i,j-\frac{1}{2}} \frac{\Delta x}{\Delta y} \qquad (2.14) \\
\lambda_W &= \lambda_{i-\frac{1}{2},j} \frac{\Delta y}{\Delta x} \\
\lambda_S &= \lambda_{i,j-\frac{1}{2}} \frac{\Delta x}{\Delta y}
\end{aligned}$$

Note that $\lambda_O = \lambda_N + \lambda_E + \lambda_W + \lambda_S$. The right hand side of equation (2.8) is now approximated in a similar manner to that of equation (2.10), i.e. $\iint_{\Gamma_{ij}} f \, d\Gamma_{ij}$ is approximated by the area of Γ_{ij} multiplied by f evaluated at the midpoint of Γ_{ij},

$$\iint_{\Gamma_{ij}} f \, d\Gamma_{ij} \approx \Delta x \Delta y \, f_{ij} \qquad (2.15)$$

The equations (2.13), (2.14), and (2.15) are then combined into a matrix equation of the form $A\underline{p} = \underline{f}$ by writing all the equations simultaneously, mapping p_{ij} onto p_n and f_{ij} to f_n where $n = i + (j-1)\, nx$ (this is the natural ordering of the nodes p_{ij}), and then letting $\underline{p}$ be the vector of the p_n's and $\underline{f}$ be the vector of the $\Delta x \Delta y \, f_n$'s.

By negating equations (2.13) and (2.15), we obtain the relation

$$-\lambda_N p_N - \lambda_E p_E + \lambda_O p_O - \lambda_W p_W - \lambda_S p_S = -\Delta x \Delta y f_O \qquad (2.16)$$

with the λ's as before in (2.14). The above mapping for the p's and the f's leads to a matrix A whose diagonal entries are positive.

2.2.3 Boundary Conditions

Equation (2.7) shows that the boundary conditions can be satisfied by letting $\frac{\partial p}{\partial n}$ and/or λ be zero on the boundary $\partial\Omega$.

At a boundary, the working of Section 2.2.2 goes ahead exactly as before, except if, for example, the line DA is on the boundary, then

$$\int_D^A \lambda \frac{\partial p}{\partial x}\, dy = \int_D^A \lambda \frac{\partial p}{\partial n}\, dy = 0$$

is substituted directly into equation (2.9), knocking out a term of the approximation.

An equivalent, but easier to implement, effect is obtained by setting $\lambda = 0$ on the boundary, since this also knocks out the same term in the approximation.

2.2.4 Forcing Term

As mentioned in Section 2.2.2, the f term is approximated by (2.15). At everywhere except the injection and production well, the forcing function

$$f = -\gamma(p_{BH} - p)\delta_I + q\delta_P$$

is zero, since both the δ_I and δ_P are zero there. At the production well, $\delta_P = 1$, and so $f = q$ there. At the injection well f is not constant, since it involves the pressure p, and thus

$$f = -\gamma(p_{BH} - p)$$

The p term is taken over to the differential side of the equation, and hence at the injection well, the problem turns into a Helmholtz equation. Following the notation of equation (2.16), λ_O is converted into $\lambda_O + \gamma$ and the right hand side term of equation (2.16) becomes $-\Delta x \Delta y\, \gamma\, p_{BH}$.

2.2.5 Mobilities

The mobility, λ, is defined in terms of permeability of the medium and the viscosity of the fluid (see Section 2.1.1). The permeability k is not usually a continuous function, but consists of a number of physical sample values at the centre of each grid block, thus λ is also only defined at the block centres. As shown previously, λ is required at the boundaries of the grid blocks, and hence the values have to be interpolated. The standard way of doing this is to use harmonic averaging [2], i.e.

$$\lambda_{i+\frac{1}{2},j} = 2\,(\frac{1}{\lambda_{ij}} + \frac{1}{\lambda_{i+1,j}})^{-1}$$

etc.

2.3 Structure of the Matrix Problem

The discretisation of Section 2.2 generates a matrix system

$$A\underline{p} = \underline{f}$$

where, due to the regular structure of the grid, A is a symmetric, 5-diagonal matrix. A has dimension $nx \times ny$. The diagonals of A are the main diagonal, the principle subdiagonals, and the subdiagonals nx rows below the main diagonal.

$$A = \begin{bmatrix} a_{11} & a_{12} & 0 & \cdots & a_{nx+1,1} & 0 & \cdots \\ a_{12} & a_{22} & a_{32} & \ddots & 0 & a_{nx+2,2} & 0 \\ 0 & a_{32} & \ddots & \ddots & 0 & & \ddots \\ \vdots & \ddots & \ddots & \ddots & & & \\ a_{nx+1,1} & 0 & 0 & & & & \\ 0 & a_{nx+2,2} & & & & & \\ \vdots & 0 & \ddots & & & & \end{bmatrix}$$

Due to the nature of the grid, this can be simplified slightly, and the matrix A is, in fact, block tridiagonal,

$$A = \begin{bmatrix} A_1 & B_1 & & & \\ B_1^T & A_2 & B_2 & & \\ & \ddots & \ddots & \ddots & \\ & & B_{ny-2}^T & A_{ny-1} & B_{ny-1} \\ & & & B_{ny-1}^T & A_{ny} \end{bmatrix}$$

where each of the ny blocks A_i is an nx by nx tridiagonal matrix, and each B_i is an nx by nx diagonal matrix. The off-diagonal elements are all non-positive, and the diagonal elements are all positive, except, possibly, the diagonal entry corresponding to the injection well, where the entry is $\lambda_O + \gamma$. If we insist that $\gamma \geq 0$, then this is also positive. If $\gamma \geq 0$, then equation (2.14) shows that A is diagonally dominant, and if $\gamma > 0$, then A is irreducibly diagonally dominant. If this is the case, then it is guaranteed that A is also positive definite, and that A^{-1} exists, and so the system can be solved by the Conjugate Gradient Method.

2.4 Sample Problems

For the purposes of this dissertation, it is assumed that the viscosity of the fluid is unity, i.e. $\mu = 1$, and so the mobility λ is the same as the permeability k. The forcing function (2.5) has values of its parameters set as

$$\begin{aligned} \gamma &= 1.0 \\ p_{BH} &= 2.5 \\ q &= -1.0 \end{aligned}$$

The region Ω is the unit square $[0,1] \times [0,1]$. The injection and production wells are at point $(0,0)$ and $(1,1)$ respectively. Two choices are given for the permeability function k.

2.4.1 Problem 1

The permeability is chosen so that it is constant everywhere

$$k(x,y) \equiv 1,\ (x,y) \in [0,1] \times [0,1]$$

On a 20 by 20 grid, scaling the output pressure so that it is in the range zero to one (the pressure at $(0,0)$ is 3.50973, and that at $(1,1)$ is 3.50000), the pressure distribution over Ω, as seen in Figure 2.2, is obtained.

The corresponding fluid flow field, obtained by calculating the gradient direction for each point in the pressure distribution, is shown in Figure 2.3, where the flow direction is shown by the arrow direction, and the magnitude of the flow shown by the arrow body lengths.

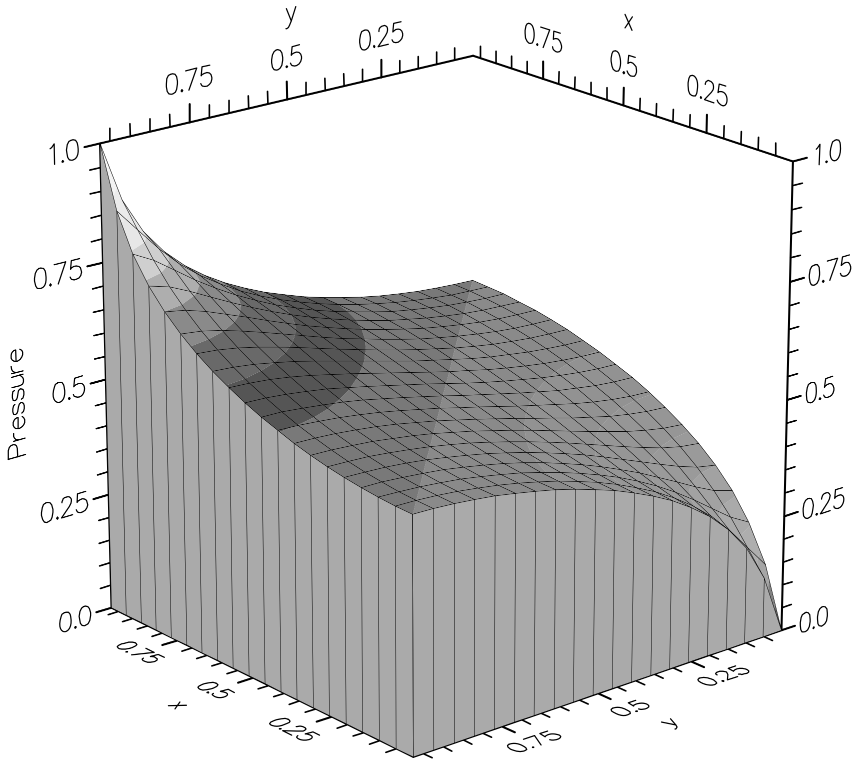


Figure 2.2: Problem 1 pressure distribution for a 20 by 20 grid.

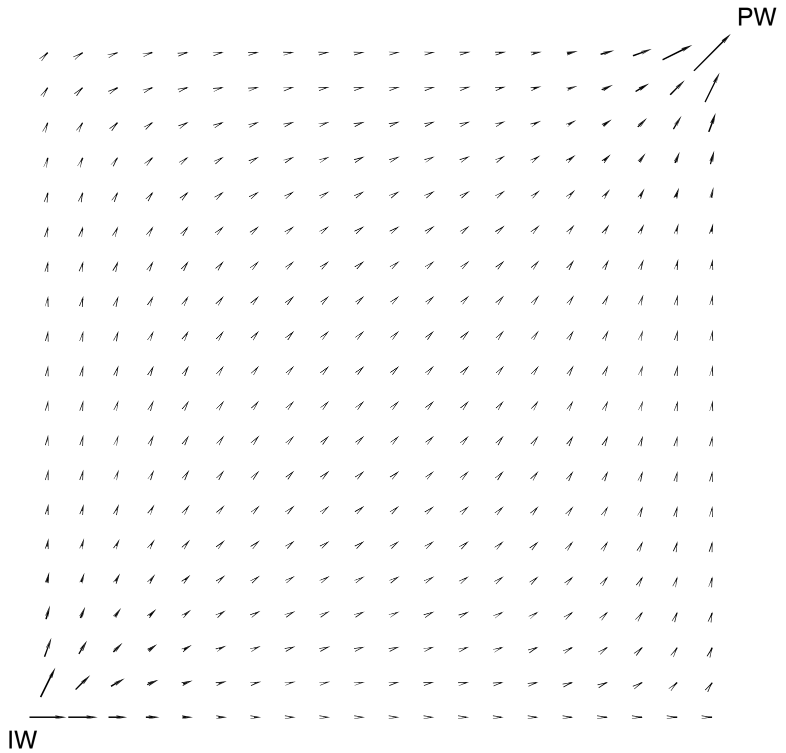


Figure 2.3: Problem 1 fluid flow field for a 20 by 20 grid.

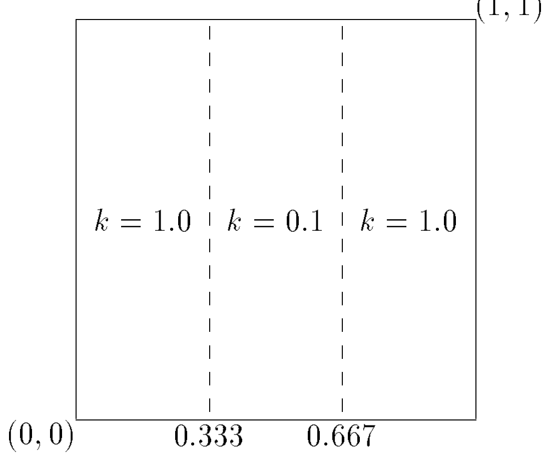


Figure 2.4: Diagram of $k(x, y)$ for problem 2

2.4.2 Problem 2

The permeability is chosen to be a non-uniform function, such that

$$k(x,y) = \begin{cases} 0.1 & 0.333 \leq x \leq 0.667 \\ 1.0 & \text{otherwise} \end{cases}$$

as in Figure 2.4. On a 20 by 20 grid, again scaling the output pressure so that it is in the range zero to one (the pressure at $(0, 0)$ is 3.51695, and that at $(1, 1)$ is 3.50000), the pressure distribution over Ω, as seen in Figure 2.5, is obtained.

The corresponding fluid flow field, is shown in Figure 2.6,

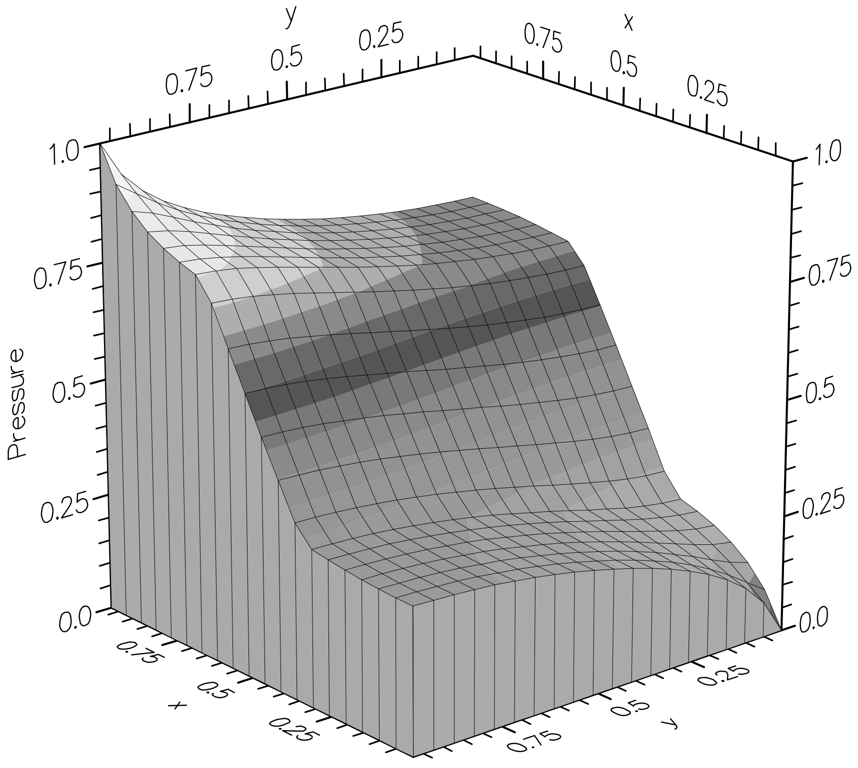


Figure 2.5: Problem 2 pressure distribution for a 20 by 20 grid.

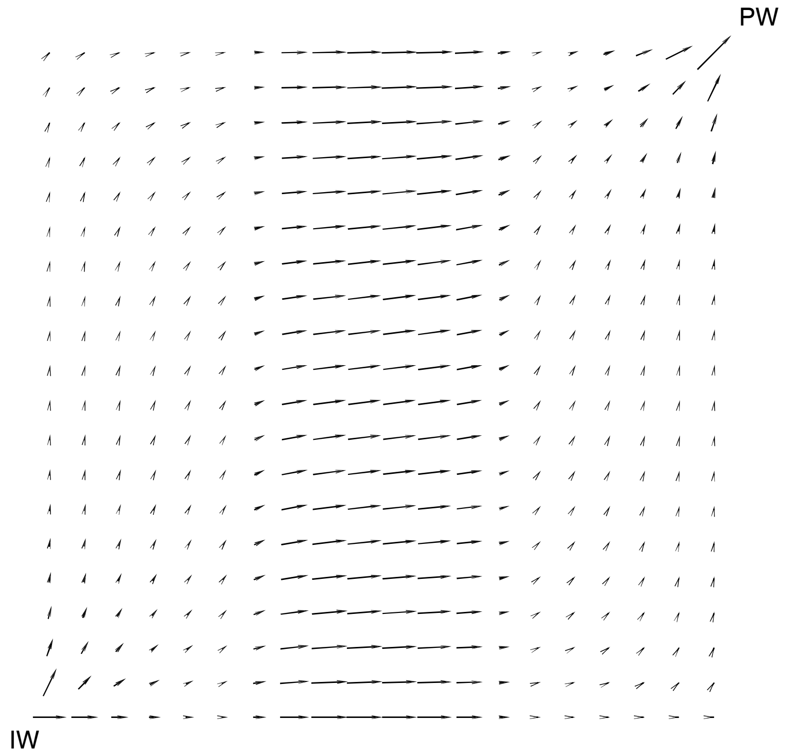


Figure 2.6: Problem 2 fluid flow field for a 20 by 20 grid.

Chapter 3

Conjugate Gradient Methods

First, this chapter briefly describes the standard iterative methods for the solution of the matrix equation

$$A\underline{x} = \underline{b}, \tag{3.1}$$

assuming A is symmetric and positive definite, conditions satisfied by the discretisation of the partial differential equation in Chapter 2. Next, the Conjugate Gradient Method is developed and some of its properties are given. Lastly, the idea of preconditioning is presented, and a number of preconditioning strategies are described.

3.1 Standard Iterative Methods

The standard iterative methods given in [7] are based on rearranging equation (3.1) by splitting the matrix A into $A = M - N$, and forming the iteration

$$\begin{aligned} M\underline{x}^{k+1} &= N\underline{x}^{k} + \underline{b} \\ \text{or } \underline{x}^{k+1} &= M^{-1}N\underline{x}^{k} + M^{-1}\underline{b} \end{aligned} \tag{3.2}$$

where k is the iteration number. The emphasis is on a choice of M so that an equation such as $M\underline{x} = \underline{b}$ is easy to solve at each iteration.

Partitioning A so that $A = L + D + U$, where

$$L = \begin{bmatrix} 0 & 0 & \cdots & \cdots & 0 \\ a_{21} & 0 & \cdots & & \vdots \\ a_{31} & a_{32} & \ddots & & 0 \\ \vdots & & & 0 & 0 \\ a_{n1} & a_{n2} & \cdots & a_{n,n-1} & 0 \end{bmatrix}$$

$$D = \operatorname{diag}(a_{11}, \cdots, a_{nn}) \tag{3.3}$$

$$U = \begin{bmatrix} 0 & a_{12} & \cdots & \cdots & a_{1n} \\ 0 & 0 & \cdots & & \vdots \\ 0 & 0 & \ddots & & a_{n-2,n} \\ \vdots & & & 0 & a_{n-1,n} \\ 0 & 0 & \cdots & 0 & 0 \end{bmatrix}$$

i.e. D is the diagonal part of A, L is the lower triangular part of A, and U is the upper triangular part of A, leads us to a number of standard iterative methods.

3.1.1 Convergence

The convergence of iteration (3.2) to the solution $\underline{x} = A^{-1}\underline{b}$ depends on the eigenvalues of the iteration matrix $M^{-1}N$. In particular, if the spectral radius, $\rho(M^{-1}N)$, is less than one then the iteration will converge [7].

3.1.2 Jacobi

Using iteration (3.2) and equation (3.3), with $M = D$ and $N = -(L+U)$ we obtain to the Jacobi Method. This method is easily parallelised since each equation in (3.2) is decoupled from the rest.

3.1.3 Gauss-Seidel

Alternatively, setting $M = (D+L)$ and $N = -U$ gives us the Gauss-Seidel (GS) method, where, essentially, at each iteration a lower triangular system has to be solved, although it is not implemented this way. The effect is that the value of each unknown requires the value of the previous one, and this leads to problems in parallelisation, due to the implied order for finding the unknowns.

3.1.4 Successive Over-Relaxation

A modification of the GS step, at each iteration combining the Gauss-Seidel new iterate with the old iterate, using a parameter ω,

$$\underline{x}_{SOR}^{k+1} = \omega \underline{x}_{GS}^{k+1} + (1-\omega)\underline{x}_{SOR}^{k}$$

leads to the Successive Over-Relaxation (SOR) method, and ω is called the relaxation parameter. In matrix form the iteration is

$$M_\omega \underline{x}^{k+1} = N_\omega \underline{x}^{k} + \omega \underline{b} \tag{3.4}$$

where $M_\omega = D + \omega L$ and $N_\omega = (1-\omega)D - \omega U$. With the correct choice of ω the rate of convergence can be significantly greater than that of GS. Unfortunately, in complicated problems, an eigenvalue analysis of the iteration matrix may need to be performed in order to find an optimum ω, although there do exist adaptive methods that will converge to the optimum ω as the iterations proceed.

3.1.5 Chebyshev Semi-Iterative

Another way to accelerate the convergence of an iterative method is to combine previous iterates $\underline{x}^1, \ldots, \underline{x}^k$ with coefficients $\nu_j(k),\ j = 0, 1, \ldots, k$ such that

$$\underline{y}^k = \sum_{j=0}^{k} \nu_j(k)\underline{x}^j$$

is an improvement over $\underline{x}^k$. The choice of suitable coefficients can be obtained from the Chebyshev polynomials, assuming the iteration matrix is symmetric [7].

3.1.6 Symmetric SOR

The assumption that Chebyshev requires a a symmetric iteration matrix rules out GS and SOR, but it is possible to symmetrise them by combining the standard method with a backward method in which the unknowns are generated in the reverse order. In the case of SOR, backward SOR is given by (cf. equation (3.4))

$$\tilde{M}_\omega \underline{x}^{k+1} = \tilde{N}_\omega \underline{x}^k + \omega \underline{b} \tag{3.5}$$

where $\tilde{M}_\omega = D + \omega U$ and $\tilde{N}_\omega = (1 - \omega)D - \omega L$. See [7] for details.

3.1.7 Block Versions

Block versions of the above iterations can be formed, and in this case D, L and U are the block diagonal, block lower triangular and block upper triangular parts of A respectively. The serial and parallel implementation of these methods has been investigated in [17].

3.2 The Conjugate Gradient Method

Although there are several ways to derive the Conjugate Gradient Method [5, 12], the derivation of the Conjugate Gradient method for the solution of equation (3.1), as given in [7], is presented.

The starting point is to consider the functional $\phi(\underline{x})$, defined by

$$\phi(\underline{x}) = \frac{1}{2}\underline{x}^T A\underline{x} - \underline{x}^T\underline{b}$$

where $\underline{b} \in \mathbb{R}^n$ and $A \in \mathbb{R}^{n\times n}$. It can be shown [17] that the minimum value of ϕ is $-\underline{b}^T A^{-1}\underline{b}/2$, at the point $\underline{x} = A^{-1}\underline{b}$. Thus, minimising ϕ and solving equation (3.1) are equivalent.

3.2.1 Steepest Descent

At a point $\underline{x}^k$, the functional ϕ^k $(= \phi(\underline{x}^k))$ decreases most rapidly in the direction of $-\underline{\nabla}\phi^k$, the residual $\underline{r}^k$ at the kth step, i.e.

$$-\underline{\nabla}\phi^k = \underline{r}^k = \underline{b} - A\underline{x}^k \tag{3.6}$$

A new iterate, $\underline{x}^{k+1}$, is defined by

$$\underline{x}^{k+1} = \underline{x}^k + \alpha\underline{r}^k, \ \ \alpha \in \mathbb{R}$$

where α is chosen to minimise ϕ^{k+1}. This is done by setting

$$\frac{\partial\phi^{k+1}}{\partial\alpha} = 0$$

which gives

$$\alpha = \frac{(\underline{r}^k)^T\underline{r}^k}{(\underline{r}^k)^T A\underline{r}^k}$$

The global convergence of the steepest descent method [7] is obtained from the inequality

$$\phi(\underline{x}^{k+1}) + \frac{1}{2}\underline{b}^T A^{-1}\underline{b} \leq \left(1 - \frac{1}{\kappa_2(A)}\right)\left(\phi(\underline{x}^k) + \frac{1}{2}\underline{b}^T A^{-1}\underline{b}\right)$$

As can be seen from this inequality, if $\kappa_2(A)$ is large, then the rate of convergence is slow. To overcome this problem, instead of minimising along the set of residual vectors $\{\underline{r}^0, \underline{r}^1, \ldots\}$, we minimise along a set of 'search directions' $\{\underline{p}^0, \underline{p}^1, \ldots\}$. Defining the new iterate this time as

$$\underline{x}^{k+1} = \underline{x}^k + \alpha \underline{p}^k \tag{3.7}$$

and, again, minimising ϕ^{k+1} by setting

$$\frac{\partial \phi^{k+1}}{\partial \alpha} = 0$$

gives

$$\alpha = \frac{(\underline{p}^k)^T \underline{r}^k}{(\underline{p}^k)^T A \underline{p}^k} \tag{3.8}$$

In order to ensure a reduction in the size of ϕ, $\underline{p}^k$ must not be orthogonal to $\underline{r}^k$, i.e. $(\underline{p}^k)^T \underline{r}^k \neq 0$, and the search directions $\underline{p}^k$ are taken as A-conjugate to all the previous search vectors [7], i.e. $P_{k-1}^T A \underline{p}^k = 0$ where $P_{k-1} = \{\underline{p}^0, \ldots, \underline{p}^{k-1}\} \in \mathbb{R}^{n \times k}$. This is the basis of the Conjugate Gradient (CG) method. From [7], the required search directions are given by

$$\underline{p}^k = \underline{r}^k + \beta \underline{p}^{k-1} \tag{3.9}$$

where

$$\beta = \frac{(\underline{r}^k)^T \underline{r}^k}{(\underline{r}^{k-1})^T A \underline{r}^{k-1}} \tag{3.10}$$

3.2.2 CG Method

Using equation

$$(\underline{p}^k)^T \underline{r}^k = (\underline{r}^k)^T \underline{r}^k$$

from [20] and equation

$$\underline{r}^{k+1} = \underline{b} - A\underline{x}^{k+1} = \underline{r}^k - \alpha A\underline{p}^k$$

from [5] we can rearrange the equations (3.6), (3.7), (3.8), (3.9), and (3.10) into the standard CG algorithm, as devised by Hestenes and Stiefel (1952):- Choose an initial guess $\underline{x}^0$, set $\underline{r}^0 = \underline{b} - A\underline{x}^0$ and $\underline{p}^0 = \underline{r}^0$ then do for $k = 0, 1, 2, \ldots, n$

$$\begin{aligned}
\alpha^k &= \frac{(\underline{r}^k)^T \underline{r}^k}{(\underline{p}^k)^T A\underline{p}^k} \\
\underline{x}^{k+1} &= \underline{x}^k + \alpha^k \underline{p}^k \\
\underline{r}^{k+1} &= \underline{r}^k - \alpha^k A\underline{p}^k \\
\beta^k &= \frac{(\underline{r}^{k+1})^T \underline{r}^{k+1}}{(\underline{r}^k)^T \underline{r}^k} \\
\underline{p}^{k+1} &= \underline{r}^{k+1} + \beta^k \underline{p}^k
\end{aligned} \tag{3.11}$$

3.2.3 Convergence of CG

If algorithm (3.11) is performed in exact arithmetic, the (exact) solution is obtained in at most n steps (and the method would be a direct one). However, when finite precision arithmetic is used, rounding errors lead to a gradual loss of orthogonality among the residuals, and the finite termination property is lost. This is not serious, since in 1971, Reid [20] showed that when CG is treated purely as an iterative method for large, sparse systems, convergence to a required accuracy usually occurs in many less than n steps. If the matrix A has m distinct eigenvalues, then CG will converge in at most m iterations. Thus, if A has a series of clustered eigenvalues, then the convergence will be more rapid. The termination criterion is based on a maximum number of iterations, k_{max}, and some measure of the size of $\underline{r}^k$.

From [7] an error bound on the CG method can be obtained in terms of the A-norm, $||.||_A$, and after k iterations

$$||\underline{x} - \underline{x}^k||_A \leq 2\,||\underline{x} - \underline{x}^0||_A \left(\frac{\sqrt{\kappa_2(A)} - 1}{\sqrt{\kappa_2(A)} + 1} \right)^k \tag{3.12}$$

This bound is usually far too pessimistic, and the accuracy of the $\{\underline{x}^k\}$ is often better than inequality (3.12) predicts. One useful consequence of inequality (3.12) is that the CG method converges very quickly in the A-norm if $\kappa_2(A) \approx 1$.

3.3 The Preconditioned CG Method

As noted above, if $\kappa_2(A) \approx 1$, then CG converges quickly. The idea of preconditioning is to convert equation (3.1) into an equivalent problem

$$\tilde{A}\tilde{\underline{x}} = \tilde{\underline{b}} \tag{3.13}$$

where $\tilde{A}$ is close to the identity matrix, so that $\kappa_2(\tilde{A}) \approx 1$. This can be achieved by pre-multiplying equation (3.1) by a the inverse of a symmetric, positive definite matrix $M \in \mathbb{R}^{n\times n}$ giving

$$M^{-1}A\underline{x} = M^{-1}\underline{b}$$

Comparing this with equation (3.13) gives us

$$\tilde{A} = M^{-1}A \qquad \tilde{\underline{x}} = \underline{x} \qquad \tilde{\underline{b}} = M^{-1}\underline{b}$$

and the basic CG method can be applied to equation (3.13).

In order to get $\tilde{A}$ close to the identity it is required that M is a good approximation for A, so that $M^{-1}A \approx I$.

Note that the basic CG algorithm is the preconditioned CG algorithm with the identity matrix I as the preconditioner.

3.3.1 Preconditioned CG Algorithm

Applying algorithm (3.11) to equation (3.13) we obtain the following Preconditioned CG (PCG) algorithm:- Choose an initial guess $\underline{x}^0$, set $\underline{r}^0 = \underline{b} - A\underline{x}^0$ and $\underline{p}^0 = M^{-1}\underline{r}^0$ then do for $k = 0, 1, 2, \ldots, n$

$$\begin{aligned}
\alpha^k &= \frac{(\underline{r}^k)^T M^{-1} \underline{r}^k}{(\underline{p}^k)^T A \underline{p}^k} \\
\underline{x}^{k+1} &= \underline{x}^k + \alpha^k \underline{p}^k \\
\underline{r}^{k+1} &= \underline{r}^k - \alpha^k A \underline{p}^k \qquad (3.14) \\
\beta^k &= \frac{(\underline{r}^{k+1})^T M^{-1} \underline{r}^{k+1}}{(\underline{r}^k)^T M^{-1} \underline{r}^k} \\
\underline{p}^{k+1} &= M^{-1}\underline{r}^{k+1} + \beta^k \underline{p}^k
\end{aligned}$$

The presence of M^{-1} in algorithm (3.14) places an extra restriction on the choice of M, that is M must be computationally simple and inexpensive to invert. The hope is that any extra work involved in preconditioning the equation is more than offset by the reduction in number of iterations required to reach convergence to a required tolerance. Unfortunately, as is shown later, there are choices of M that at first examination would appear to be good, but actually involve far more work than the basic CG algorithm does.

3.4 Preconditioners

This section details various types of preconditioner M. There are five main types considered:-

1. M based on splittings of A, i.e. $A = M - N$
2. Complete or incomplete factorisations of A, e.g. $A = LL^T + E$
3. Approximations of $M = A^{-1}$
4. Iterative solution of $M\underline{z} = \underline{r}$
5. Reordering of the equations and/or unknowns, e.g. Domain Decomposition

3.4.1 Splittings of A

Here M is chosen such that $A = M - N$, i.e. M is a part of the matrix A, and N is the remains of the matrix. M and N need not be disjoint matrices, although they are in this discussion.

Diagonal Preconditioning

The simplest choice for M is $D = \mathrm{diag}\{a_{11}, a_{22}, \ldots, a_{nn}\}$ and this is equivalent to a rescaling of all the equations, such that the diagonal of $D^{-1}A$ is all ones. D is very easy to invert. It can be shown that if A is 2-cyclic then D is optimal among all diagonal matrices, i.e. κ_2 is minimised.

Tridiagonal Preconditioning

Due to the structure of the matrix equations arising from the discretisation in Chapter 2, that is 5-diagonal matrices, another choice for M is the tridiagonal part of A, i.e.

$$T = \begin{bmatrix} a_{11} & a_{12} & 0 & \cdots & 0 \\ a_{21} & a_{22} & a_{23} & & \vdots \\ 0 & a_{32} & \ddots & \ddots & \\ \vdots & & \ddots & a_{n-1,n-1} & a_{n-1,n} \\ 0 & \cdots & 0 & a_{n,n-1} & a_{nn} \end{bmatrix}$$

This choice of T is relatively simple to invert, being just a forward and backward substitution. As mentioned in Section 2.3, the A obtained from the problem discretisation is actually block tridiagonal, and thus T consists of ny tridiagonal blocks each of size nx by nx.

Another tridiagonal preconditioner would be the tridiagonal part of A^{-1}, which should be more a accurate approximation than the inverse of the tridiagonal part of A, but, unfortunately, a way to compute this inexpensively could not be found.

Block Diagonal Preconditioning

Instead of taking just the diagonal elements of the matrix A, diagonal blocks of A can be used, e.g. s such blocks A_i, possibly of differing sizes

$$M = \begin{bmatrix} A_1 & 0 & \cdots & \cdots & 0 \\ 0 & A_2 & 0 & & 0 \\ \vdots & 0 & \ddots & \ddots & \vdots \\ \vdots & & \ddots & A_{s-1} & 0 \\ 0 & 0 & \cdots & 0 & A_s \end{bmatrix}$$

and thus, the solution of $M\underline{z} = \underline{r}$ in the PCG algorithm reduces to the solutions to a series of smaller independent problems.

In the discretisation in Chapter 2, a natural block size is nx, and in this case all the A_i's are tridiagonal. Another natural size is any integer multiple of nx, and here all the A_i's are 5-diagonal.

Blocked Tridiagonal Preconditioning

As above, tridiagonal preconditioning can be 'blocked' to introduce independent smaller subsystems. With a block size of nx, this reduces to the Block Diagonal Preconditioning with block size of nx, and incidentally, it is the same as Tridiagonal Preconditioning.

3.4.2 Factorisations of A

Standard techniques for the direct solution of equation (3.1), involve factorising A into the product of two matrices A_1 and A_2 such that systems $A_i\underline{x} = \underline{b}$, $i = 1, 2$ are trivial to solve.

Cholesky Factorisation

If A is symmetric and positive definite, then A can be written as

$$A = LL^T \tag{3.15}$$

where L is lower triangular [7]. Equation (3.15) then uniquely defines L, and since L is lower triangular $L^{-1}\underline{z}$ and $L^{-T}\underline{z}$ are easily calculated for any vector $\underline{z}$, and the solution to (3.1) is given as

$$\underline{x} = (L^{-T})(L^{-1}\underline{b})$$

Equation (3.15) can be solved column by column recursively to obtain L [7] as follows

```
do $i = 1, 2, \ldots, n$
    $L_{ii} = \sqrt{A_{ii} - \sum_{k=1}^{i-1} L_{jk}^2}$                (3.16)
    do $j = i+1, i+2, \ldots, n$
        $L_{ji} = \frac{A_{ji} - \sum_{k=1}^{i-1} L_{jk}L_{ik}}{L_{ii}}$
```

The square root operation is expensive to calculate and often inaccurate, so there exists an alternative statement of Cholesky which avoids it, where A is written as

$$A = LDL^T$$

where the diagonal of L consists of all ones.

Incomplete Cholesky Factorisation

For large sparse matrices A, the calculation of L is both very time and storage consuming, and the sparseness of the original matrix A is lost due to the massive amount of fill-in in L. Since only an approximation to A is required for the preconditioning (calculation of the complete decomposition would solve the problem immediately!), the Incomplete Cholesky Decomposition of A can be defined. First, a sparsity pattern is defined to impose on L, i.e. a set of matrix entries, P, are chosen which are forced to be zero in L. Algorithm (3.16) is still used, but whenever an L_{ij} arises with $(i,j) \in P$, it is set to zero and the algorithm continues. Thus, these elements of L are neither calculated or stored. The simplest choice for P is

$$P = \{\,(i,j) \mid A_{ij} = 0;\, i,j = 1,2,\ldots,n\,\}$$

that is, the sparsity pattern of A is enforced on L. In [15], this choice of P is referred to ICCG(0), one of a family of methods. This choice of P is used throughout this dissertation, and is referred to as ICCG. Thus

$$A = LL^T + E$$

where E is a small error matrix whose non-zero entries all lie in P, and the preconditioning matrix M can be taken as LL^T, which only requires a forward and backward substitution to invert.

In algorithm (3.16) it is crucial that all the L_{ii} are greater than zero. If $L_{ii} = 0$ then the algorithm breaks down, and if $L_{ii} < 0$, then LL^T is not positive definite [12] and the CG method cannot be used. It is shown in [15] that if A is an M-matrix[1] then the ICCG algorithm always gives $L_{ii} > 0$, but, unfortunately, ICCG on an arbitrary positive definite matrix is not guaranteed to work. A

[1] $A = (a_{ij})$ is an M-matrix if $a_{ij} \leq 0$ for $i \neq j$, A is non-singular, and $A^{-1} \geq 0$.

simple counter-example is given in [12],

$$\begin{bmatrix} 3 & -2 & 0 & 2 \\ -2 & 3 & -2 & 0 \\ 0 & -2 & 3 & -2 \\ 2 & 0 & -2 & 3 \end{bmatrix}$$

Complete Cholesky decomposition gives $L_{44} = \frac{1}{2}$, while ICCG gives $L_{44} = -5$. All is not lost though, since an exact decomposition of A is not required, and [12] suggests that if an L_{ii} arises such that $L_{ii} < 0$, it is simply set to some positive value[2] and the algorithm is continued. This only causes the ith diagonal element of E to be non-zero, all other non-zeros are still in P. If $L_{ii} < 0$ rarely occurs, i.e. if the decomposition is mostly stable, this adaptation should work quite well, and the experiences in [12] confirm this.

A small modification can be made to the ICCG algorithm, and that is to enforce the equality of the row-sums of A and LL^T by altering the diagonal elements. This leads to a slightly better approximation of A [10].

Blocked Incomplete Cholesky

In Section 3.4.1, the idea of Block Diagonal Preconditioning was introduced to introduce smaller subproblems which were more easily solved. The idea here is to solve each subproblem obtained with the Block Diagonal Preconditioner with the ICCG algorithm, instead of calculating the exact solution, for the same reasons that the ICCG algorithm was developed above.

[2]In [12] $L_{ii} = \sum_{j=1}^{i-1} |L_{ij}| + \sum_{j=i+1}^{n} |L_{ij}|$.

3.4.3 Polynomial Approximation

If $\rho(J) < 1$, then from [22], the inverse of $I - J$ can be expressed as a power series in J, i.e.

$$(I-J)^{-1} = \sum_{k=0}^{\infty} J^k = I + J + J^2 + J^3 + \dots \tag{3.17}$$

If A is written as $A = D+L+U$, as in equation (3.3), then $J = -D^{-1}(L+U)$ is the Jacobi iteration matrix (see Section (3.1.2)), and [7] shows that $\rho(J) < 1$. Thus, if A is written as $A = D(I-J)$, and $(I-J)^{-1}$ expanded as in equation (3.17), the inverse of A can be expressed as

$$\begin{aligned} A^{-1} &= (D(I-J))^{-1} \\ &= (I-J)^{-1}D^{-1} \\ &= \{\sum_{k=0}^{\infty} J^k\}D^{-1} \\ &= \sum_{k=0}^{\infty}(-1)^k(D^{-1}(L+U))^k D^{-1} \end{aligned} \tag{3.18}$$

The idea of polynomial preconditioning is to truncate the power series (3.18) after m terms and obtain an approximation M_m^{-1} for A^{-1}. If the series is truncated after the $k = 1$ term, then

$$M_1^{-1} = D^{-1} - D^{-1}(L+U)D^{-1} \tag{3.19}$$

has the same sparsity pattern as A, hence no fill-in has occurred. The idea in [11] is instead of having a truncated power series of the form (3.18), a parameter is introduced for each term in the approximation, thus

$$M_m^{-1} \approx \sum_{k=0}^{m} \gamma_k(-1)^k(D^{-1}(L+U))^k D^{-1} \tag{3.20}$$

and then to choose the γ_i's such that the spectral radius, $\rho(M_m^{-1}A)$, is minimised, i.e. the convergence of the PCG method maximised. The observation for equation (3.19) still holds, i.e. M_1^{-1} still has the same sparsity as A. This method is also used in [4], although there it is for approximating the inverse of a tridiagonal matrix, and three sets of values are given for γ_0 and γ_1 with $m = 1$; approximately 0.9412 and -0.4706, 1 and -1, and 1.1429 and -1.1429. The values given in [11] for γ_0 and γ_1 are 1.0 and -1.0, and $\frac{7}{6}$ and $-\frac{5}{6}$. See [11] for details of how these values were obtained.

3.4.4 Iterative Methods

Since each iteration of the PCG algorithm involves the solution of a system of the form $M\underline{z} = \underline{r}$, an idea is to perform a number of iterations of one of the standard iterative methods as described in Section 3.1, where M is the required part of A. Hence, any iterative method based on the splitting of $A = M - N$ may be accelerated by the use of the CG method, as long as M is symmetric and positive definite. The case where one iteration of the Jacobi (Section 3.1.2) iteration is performed corresponds to the diagonal scaling method in Section 3.3.

3.4.5 Domain Decomposition

The idea of domain decomposition can be illustrated by dividing the region the problem is being solved on, Ω, into two non-overlapping subregions Ω_1 and Ω_2 ($\Omega = \Omega_1 \cup \Omega_2$). The mesh points are ordered so that those in Ω_1 come first, those in Ω_2 come next, and all those not in Ω_1 or Ω_2 (the mesh points on the interface $\partial\Omega_1 \cup \partial\Omega_2$ and those on the global boundary $\partial\Omega$) come last. The three groups of mesh points are called $\underline{x}_1$, $\underline{x}_2$ and $\underline{x}_b$ respectively, and the resulting matrix system is of the form

$$\begin{bmatrix} A_1 & 0 & B_1 \\ 0 & A_2 & B_2 \\ B_1^T & B_2^T & A_b \end{bmatrix} \begin{bmatrix} \underline{x}_1 \\ \underline{x}_2 \\ \underline{x}_b \end{bmatrix} = \begin{bmatrix} \underline{b}_1 \\ \underline{b}_2 \\ \underline{b}_b \end{bmatrix} \tag{3.21}$$

where $A_i,\ i = 1,2$ are symmetric positive definite, the $B_i,\ i = 1,2$ are sparse, and the third block row and column are much narrower than the first two. This is basically a rearrangement of the equations and unknowns for the original matrix system (3.1). Equation (3.21) can be expanded to the p subdomain case, where the matrix system becomes

$$\begin{bmatrix} A_1 & \cdots & 0 & B_1 \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & A_p & B_p \\ B_1^T & \cdots & B_p^T & A_b \end{bmatrix} \begin{bmatrix} \underline{x}_1 \\ \vdots \\ \underline{x}_p \\ \underline{x}_b \end{bmatrix} = \begin{bmatrix} \underline{b}_1 \\ \vdots \\ \underline{b}_p \\ \underline{b}_b \end{bmatrix} \tag{3.22}$$

Common preconditioners used with domain decomposition are either the iterative method type (Section 3.4.4) or a matrix splitting type (Section 3.4.1). The iterative type is used in [13] where one iteration of a symmetric Gauss-Seidel method is performed, i.e. for equation (3.21), with initial estimate $\underline{x}^{(0)} = 0$, the following applies

$$\begin{aligned} \text{Step 1. } A_1\underline{x}_1^{(*)} &= (\underline{b}_1 - B_1\underline{x}_b^{(0)}) = \underline{b}_1 \\ A_2\underline{x}_2^{(*)} &= (\underline{b}_2 - B_2\underline{x}_b^{(0)}) = \underline{b}_2 \\ \text{Step 2. } A_b\underline{x}_b^{(1)} &= \underline{b}_b - B_1^T\underline{x}_1^{(*)} - B_2^T\underline{x}_2^{(*)} \\ \text{Step 3. } A_1\underline{x}_1^{(1)} &= (\underline{b}_1 - B_1\underline{x}_b^{(1)}) \\ A_2\underline{x}_2^{(1)} &= (\underline{b}_2 - B_2\underline{x}_b^{(1)}) \end{aligned}$$

and similarly for equation (3.22) in the p subdomain case.

The splitting type is explored briefly in [7], e.g. with the $p = 2$ case (c.f. equation 3.21). Set

$$M = L \begin{bmatrix} M_1^{-1} & 0 & 0 \\ 0 & M_2^{-1} & 0 \\ 0 & 0 & S^{-1} \end{bmatrix} L^T$$

where

$$L = \begin{bmatrix} M_1 & 0 & 0 \\ 0 & M_2 & 0 \\ B_1^T & B_2^T & S \end{bmatrix}$$

then

$$M = \begin{bmatrix} M_1 & 0 & B_1 \\ 0 & M_2 & B_2 \\ B_1^T & B_2^T & S_* \end{bmatrix} \tag{3.23}$$

with $S_* = B_1^T M_1^{-1} B_1 + B_2^T M_2^{-1} B_2$. The block parameters M_1, M_2, and S are now chosen to give an effective preconditioner. The comparison of equations (3.21) and (3.23) shows that it makes sense to let M_i approximate A_i for $i = 1, 2$, and to let S_* approximate A_b. The latter can be achieved if $S \approx A_b - B_1^T M_1^{-1} B_1 - B_2^T M_2^{-1} B_2$. There are several ways to approximate S due to the fact that the $B_i^T M_i^{-1} B_i$, $i = 1, 2$, are dense and need to be approximated, e.g. a polynomial approximation to the M_i^{-1}'s as in Section 3.4.3.

Domain Decomposition ideas are not investigated in this dissertation due to the discretisation in Chapter 2, which always generates a 5-diagonal matrix and the fact that Domain Decomposition alters this structure. They do deserve some attention, though, due to the opportunities for parallel computation as described in [13].

Chapter 4

Serial Implementation

This chapter outlines the serial implementation of the PCG algorithm (3.14) with some of the preconditioners in Section 3, complete with any implementation peculiarities. The metrics used for the comparison of various preconditioners are then introduced, and finally, results for all of the serial preconditioners are presented.

4.1 Serial PCG Algorithm

Algorithm (3.14) is not the most efficient algorithm in terms of work and storage space; only one matrix-vector product $A\underline{p}^k$ and only one M^{-1} calculation (solution of $M\underline{z} = \underline{r}$) need be done per iteration. Values can be saved from previous iterations, and the indices of the vectors $\underline{x}$, $\underline{r}$, $\underline{p}$, etc. can be dropped, just by using them to represent the latest vectors. The following algorithm uses, apart from the storage of the problem (3.1) and the preconditioner matrix M, three vectors and five scalars of storage,

```
$\underline{z} = A\underline{x}$
$\underline{r} = \underline{b} - \underline{z}$
$\underline{p} = M^{-1}\underline{r}$
$\text{lip} = \underline{r}^T\underline{p}$
do $i = 0, 1, \ldots,$ {
        $\underline{z} = A\underline{p}$
        $\text{nip} = \underline{p}^T\underline{z}$                    (4.1)
        $\alpha = \frac{\text{nip}}{\text{lip}}$
        $\underline{x} = \underline{x} + \alpha\underline{p}$
        $\underline{r} = \underline{r} - \alpha\underline{z}$
        $\underline{z} = M^{-1}\underline{z}$
        $\text{nip} = \underline{r}^T\underline{z}$
        $\beta = \frac{\text{nip}}{\text{lip}}$
        $\underline{p} = \underline{z} + \beta\underline{p}$
        $\text{lip} = \text{nip}$
}
```

An extra vector of storage may be needed in the parts of algorithm (4.1) where the inverse of the preconditioner M^{-1} is involved. The iterating is stopped after a predetermined maximum number of iterations, or when a measure of the residual size is small enough, e.g.

$$||\underline{r}||_2^2 = \underline{r}^T\underline{r} < \text{tol}^2$$

4.2 Implementation Details

The nature of the problem being solved, and the types of preconditioner chosen to be implemented, allow several storage and time saving optimisations to be done. The preconditioners chosen to be implemented in serial were :-

- CG - Standard Conjugate Gradients (Algorithm 3.11)
- DCG - Diagonal Scaled CG (Section 3.4.1)
- DBCG - Diagonal Block Preconditioned CG (Section 3.4.1)
- ICCG - Incomplete Cholesky Preconditioned CG (Section 3.4.2)
- DBICCG - Diagonal Block Incomplete Cholesky Preconditioned CG (Section 3.4.2)
- DBTCG - Tridiagonal Diagonal Block Preconditioned CG (Section 3.4.1)
- POLCG - Polynomial Preconditioned CG (Section 3.4.3)

The tridiagonal preconditioner (Section 3.4.1) has not been included, because, as noted in Section 3.4.1, it is equivalent, for the problem discretisation in Chapter 2, to the DBCG method with block size nx, and also the DBTCG method. The Iterative accelerated methods (Section 3.4.4) were not considered due to lack of time. The Domain Decomposition methods (Section 3.4.5) were not considered due to the different matrix structure obtained from discretisation.

4.2.1 Matrix Storage

Since the discretisation in Chapter 2 generates a symmetric, 5-diagonal matrix A, it is easy to store A completely using just three vectors length n, $\underline{a}^{(d)}$, $\underline{a}^{(1)}$ and

$\underline{a}^{(2)}$, so

$$\begin{bmatrix} a_{11} & a_{12} & 0 & \cdots & a_{1,s+1} & 0 & \cdots \\ a_{21} & a_{22} & a_{23} & \ddots & 0 & a_{2,s+2} & \\ 0 & a_{32} & a_{33} & \ddots & & & \ddots \\ \vdots & \ddots & \ddots & \ddots & & & \\ a_{s+1,1} & 0 & & & \ddots & \ddots & 0 \\ 0 & a_{2,s+2} & & & \ddots & a_{n-1,n-1} & a_{n-1,n} \\ \vdots & & \ddots & & 0 & a_{n,n-1} & a_{nn} \end{bmatrix}$$

maps to

$$\begin{bmatrix} \underline{a}_1^{(d)} & \underline{a}_1^{(1)} & 0 & \cdots & \underline{a}_1^{(2)} & 0 & \cdots \\ \underline{a}_1^{(1)} & \underline{a}_2^{(d)} & \underline{a}_2^{(1)} & \ddots & 0 & \underline{a}_2^{(2)} & \\ 0 & \underline{a}_2^{(1)} & \underline{a}_3^{(d)} & \ddots & & & \ddots \\ \vdots & \ddots & \ddots & \ddots & & & \\ \underline{a}_1^{(2)} & 0 & & & \ddots & \ddots & 0 \\ 0 & \underline{a}_2^{(2)} & & & \ddots & \underline{a}_{n-1}^{(d)} & \underline{a}_{n-1}^{(1)} \\ \vdots & & \ddots & & 0 & \underline{a}_{n-1}^{(1)} & \underline{a}_n^{(d)} \end{bmatrix}$$

Similar types of storage mechanism can be used for most of the preconditioners, e.g. ICCG (Section 3.4.2) and polynomial approximation (Section 3.4.3) both have the same structure as A.

4.2.2 Preconditioner Implementation

Before the PCG algorithm (4.1) can start, the preconditioning matrix, M, has to be calculated, and during the iteration, its inverse, M^{-1}, has to be calculated (possibly implicitly).

CG

Since CG is PCG with the identity matrix, $M = I$, and thus M, or its inverse, M^{-1}, never need to be calculated, and the algorithm (4.1) simplifies slightly.

DCG

In DCG, the preconditioner is the diagonal of A, and hence to multiply by M^{-1}, division by the diagonal elements of A is performed.

DBCG

In this case M consists of diagonal blocks M_i of A, and each of these diagonal blocks is either tridiagonal or 5-diagonal. In the tridiagonal case, the solution of $M_i\underline{z} = \underline{r}$ is obtained by a simple forward and backward substitution to obtain the two factors of M_i, but in the 5-diagonal case a full[1] Cholesky decomposition has to be done to obtain the factors. This requires a problem dependent amount of extra storage for the factors, and, unfortunately, Fortran 77 cannot allocate dynamic storage, thus the arrays are just made very large at the start, hoping there is enough space for the factors given the required problem size.

In the preconditioning phase, the problem is split up into p subproblems (the size of each problem is automatically determined to make sure each one is an integer multiple of nx), and M is factored so that $M = LL^T$, i.e. each M_i is factored so that $M_i = L_iL_i^T$. During the iteration, each subproblem $M_i\underline{z} = \underline{r}$ is solved using the factor L_i by forward and backward substitution. As mentioned in Section 4.1, an extra vector of storage is required for this substitution phase.

[1]Actually, since A is banded, a banded Cholesky decomposition can be done, requiring less storage space and time.

The splitting of the problem such as above is unnatural, in that the code required to do the splitting is rather complex, but it has been done here more for the comparison with a parallel version of the method than to give an efficient serial implementation.

ICCG

The matrix A is factorised using the Cholesky decomposition (3.16) with the modifications in Section 3.4.2 for the incomplete decomposition, but by using the sparsity of A, the entire factorisation can be simplified because most of the summations in algorithm (3.16) only have one product in them. Reordering the equations allows the factorisation to be performed in a single loop. As above, the solution of $M\underline{z} = \underline{r}$ requires an extra vector for the forward and backward substitution.

DBICCG

Very similar to the DBCG method, but instead of having to do the full Cholesky decomposition for a 5-diagonal block, the Incomplete decomposition, as above, is done for each block. Thus, there are a fixed number of vectors required for the decomposition, unlike DBCG where the number is dependent on the problem size. As for ICCG, one extra vector is required for the forward and backward substitution phase.

DBTCG

This is exactly the same as the DBCG method, but with the block size enforced to be nx, i.e. the blocks are all tridiagonal.

POLCG

The polynomial approximation (3.20) M^{-1} to A^{-1} given in Section 3.4.3, with $m = 1$, has the same sparsity as A, thus only three vectors are needed to store it (Section 4.2.1). It is precalculated before the iteration starts, and the equation $M\underline{z} = \underline{r}$ is solved by a matrix multiplication $\underline{z} = M^{-1}\underline{r}$.

4.3 Performance Metrics

In order to compare preconditioning strategies, some metrics for determining efficiency, rate of convergence, etc. , are required.

4.3.1 Timing Metrics

The easiest thing to do is to time how long each part of the PCG algorithms take for a certain problem size, and count the number of iterations required to reach a certain accuracy. The four chosen here are

Preconditioning Time The time taken to calculate any preconditioning matrix, and perform the instructions up until the first iteration starts.

Iteration Time The time taken to perform one iteration of the PCG algorithm.

Number of Iterations The number of iterations required to reach the desired accuracy.

Total Time The total solution time for the PCG algorithm.

The total time metric is the most useful for comparing which are the best preconditioners, although the number of iterations required is interesting.

4.3.2 Iteration Matrix Metrics

The idea of preconditioning is to make the condition number of $\tilde{A} = M^{-1}A$ close to one, so that $\tilde{A} \approx I$. This also means that all of the eigenvalues of $\tilde{A}$ are required to be bunched around one, and hence the spectral radius of $\tilde{A}$ needs to be small.

Using the EISPACK [21] library of eigenvalue and eigenvector routines, a complete eigenvalue spectrum, $\lambda(\tilde{A})$, can be obtained from the $\tilde{A}$'s from each preconditioner. From $\lambda(\tilde{A})$ it is possible to obtain the condition number, $\kappa_2(\tilde{A})$, and the spectral radius, $\rho(\tilde{A})$.

4.4 Serial Implementation Results

The preconditioners in Section (4.2.2) were implemented in Fortran 77. All the timings were obtained by running the programs on a single transputer (See Chapter 5). All of the graphs were obtained using the UNIRAS graphics library, and the eigenvalue spectra were obtained using the EISPACK subroutines, both on Sun Sparc computers.

Since the DBCG and DBICCG methods require a block size, or alternatively, the number of subproblems to split the problem into, and the fact that the transputer system available (See Chapter 5) has five processors, it was decided, for both of these methods, to have the number of subproblems as one, two, three, four, or five (to allow direct comparisons with the parallel versions (See Section 5.7)). The number of subproblems is indicated by following DBCG or DBICCG by the number, e.g. DBCG2 for two subproblems. DBCG1 is equivalent to directly solv-

Method	γ_0	γ_1
POLCG1	1.0	-1.0
POLCG2	1.1429	-1.1429
POLCG3	0.9412	-0.4706
POLCG4	1.16666	-0.83333

Table 4.1: Parameters for the POLCG method

ing the problem by Cholesky Decomposition (3.16), ignoring the time spent in performing one iteration of the PCG method, and DBICCG1 is equivalent to ICCG. Four versions of the POLCG were used, with their γ_0 and γ_1 defined as in Table 4.1. Occasionally, the results show a TRICG method. This is just a full tridiagonal preconditioner, and in terms of performance is just about equivalent to DBTCG (the solution stage takes slightly longer, but the end result is the same).

4.4.1 Timings

Tables 4.2, 4.3, 4.4, and 4.5 show, in order of increasing total iteration time, the number of iterations required to reach convergence (in this case for the 2-norm of the residual to fall below 10^{-8}), the preconditioning time, the time per iteration, and the total time for each of the investigated methods on two problem sizes (10 by 10 and 20 by 20) and for both of the sample problems (Section 2.4). The times involved are measured in ticks of $\frac{1}{16525}$ths of a second.

The Figures 4.1 and 4.2 show the number of iterations and total time required as the problem size increases (from 5 by 5, i.e. 25 unknowns, to 30 by 30, i.e. 900

Method	Iterations	Preconditioning Time	Iteration Time	Total Time
DBCG1	1	1511	376	1886
ICCG	17	156	137	2493
POLCG4	27	120	135	3764
DBICCG2	25	166	147	3848
POLCG3	28	120	135	3899
CG	44	55	91	4090
DCG	42	58	100	4282
DBICCG3	27	170	154	4351
DBICCG4	29	178	164	4952
POLCG2	37	120	135	5119
DBTCG	43	96	120	5281
TRICG	43	103	128	5647
DBICCG5	32	186	176	5827
POLCG1	37	120	135	5827
DBCG2	15	1409	347	6625
DBCG3	23	1306	327	8841
DBCG4	27	1205	307	9514
DBCG5	30	1109	287	9740

Table 4.2: Iterations and timings for Problem 1 (10 by 10)

Method	Iterations	Preconditioning Time	Iteration Time	Total Time
ICCG	30	634	554	17279
DBCG1	1	17904	2510	20414
DBICCG2	43	673	596	26338
POLCG4	52	484	546	28785
DBICCG3	46	700	634	29885
POLCG3	58	484	546	32080
DBICCG4	48	740	680	33427
CG	93	223	368	34476
DBICCG5	50	775	722	36899
DCG	91	234	405	36962
DBTCG	88	386	484	43033
TRICG	88	417	517	45998
POLCG1	86	484	546	47454
POLCG2	86	484	546	47454
DBCG2	18	17249	2398	60428
DBCG3	31	16581	2318	88469
DBCG4	38	15955	2239	101066
DBCG5	43	15333	2160	108230

Table 4.3: Iterations and timings for Problem 1 (20 by 20)

Method	Iterations	Preconditioning Time	Iteration Time	Total Time
DBCG1	1	1510	376	1886
ICCG	21	157	137	3042
DBICCG2	25	166	147	3848
DBICCG3	28	170	154	4505
DBICCG4	30	178	164	5116
DBTCG	44	96	120	5402
DBICCG5	31	186	176	5651
DCG	56	58	100	5706
TRICG	44	104	128	5776
POLCG3	42	120	135	5798
DBCG2	15	1408	347	6624
POLCG4	57	120	135	7830
CG	87	55	92	8063
DBCG3	23	1306	327	8841
DBCG4	27	1206	307	9515
DBCG5	31	1110	287	10029
POLCG2	85	120	135	11627
POLCG1	86	120	135	11763

Table 4.4: Iterations and timings for Problem 2 (10 by 10)

Method	Iterations	Preconditioning Time	Iteration Time	Total Time
DBCG1	1	17904	2510	20414
ICCG	38	635	554	21709
DBICCG2	43	672	596	26338
DBICCG3	46	700	634	29886
DBICCG4	48	740	680	33427
DBICCG5	51	774	722	37621
DBTCG	88	385	484	43032
POLCG3	83	484	547	45807
TRICG	88	417	517	45998
DCG	120	234	405	48846
DBCG2	19	17249	2398	62825
CG	188	223	370	69837
POLCG4	128	484	547	70517
DBCG3	31	16582	2319	88471
DBCG4	38	15955	2239	101067
DBCG5	43	15335	2160	108233
POLCG1	218	484	547	119937
POLCG2	218	484	547	119937

Table 4.5: Iterations and timings for Problem 2 (20 by 20)

unknowns).

A number of observations can be made from these tables and figures,

- In terms of iteration count the ICCG method is only beaten by the DBCG methods with a small number of subproblems.
- In terms of total time the ICCG method is only ever beaten by the DBCG1 method (the direct solution), and as the problem size increases the difference between them decreases, and for large problem sizes in problem 2, ICCG is better.
- The DBCG2-5 methods, although keeping the iteration count down, have large iteration times, and the more subproblems there are, the longer the method takes.
- The DBICCG2-5 methods also keep the iteration count down, but also have smaller iteration times, thus decreasing the total time. DBICCG2 is usually the next best after ICCG.
- POLCG1 and POLCG2 are always worse than CG in straight iteration time, and in problem 2, have far many more iterations. POLCG3 and POLCG4 are better than CG, and on problem 1 they perform well (POLCG4 being better), but on problem 2 POLCG3 is better, and the performance of POLCG4 falls off with increasing problem size.
- DCG for problem 1, although decreasing the iteration count, takes more time than CG, while on problem 2 DCG is a great improvement over CG.

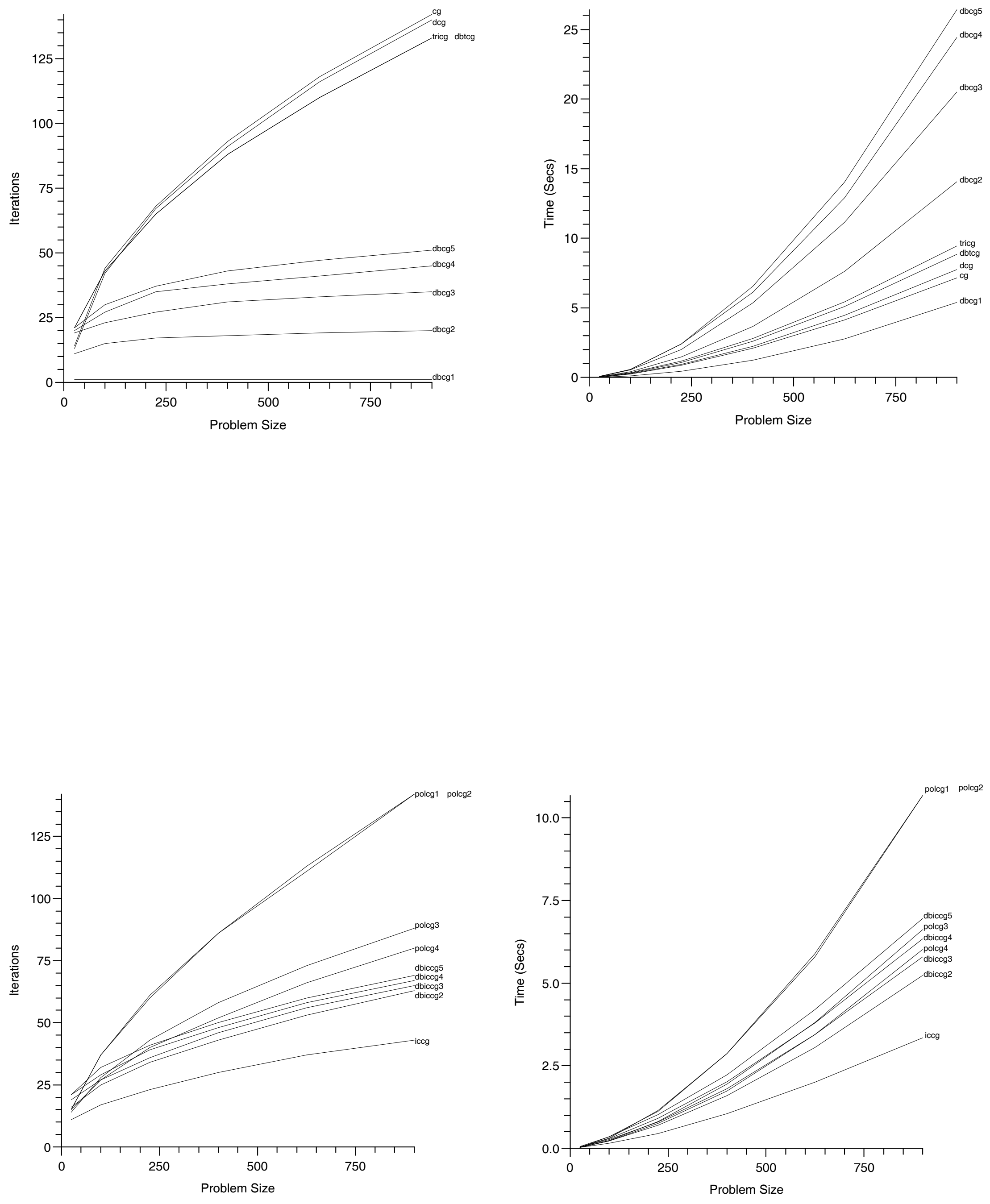


Figure 4.1: Work involved in problem 1 as problem size increases

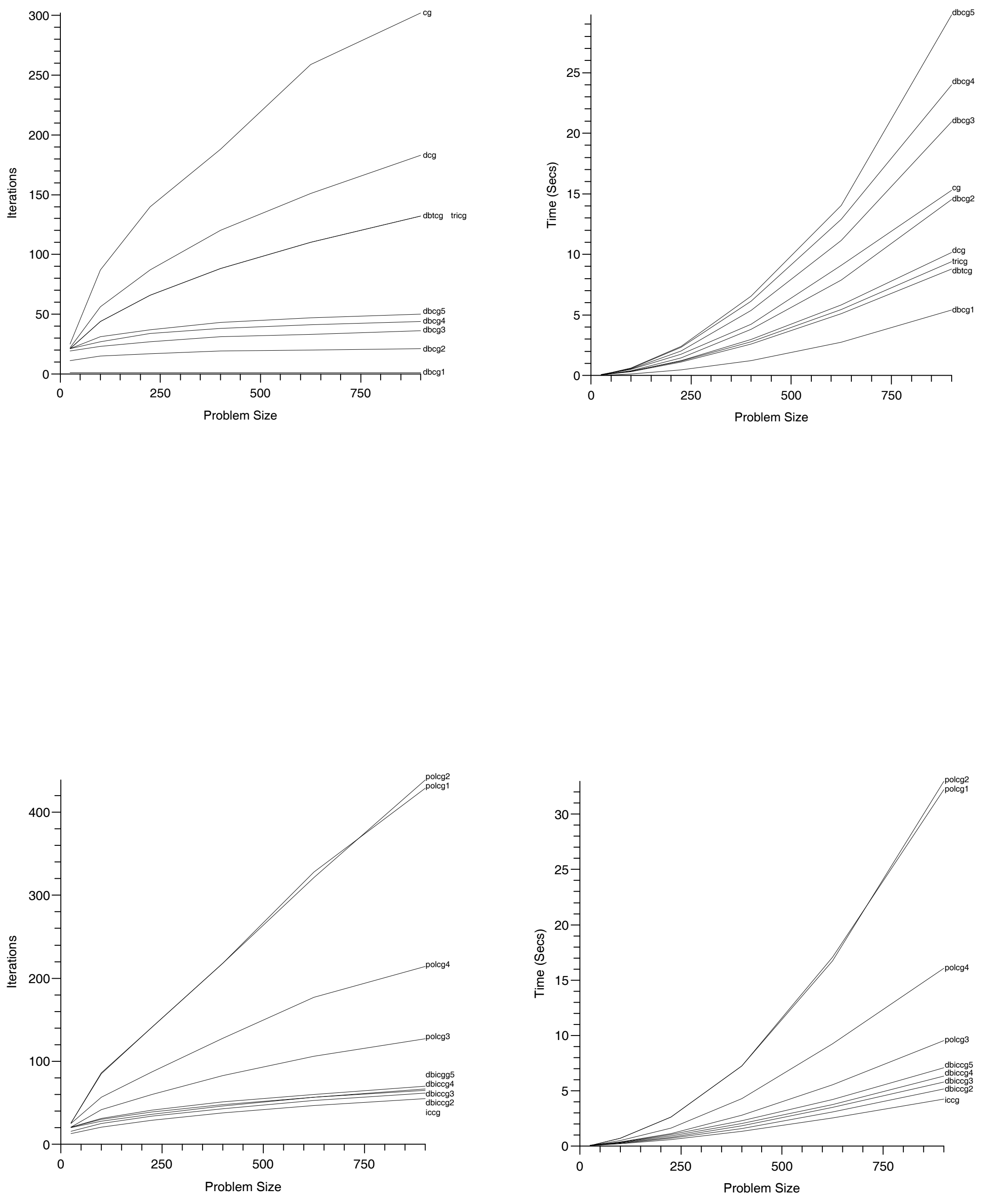


Figure 4.2: Work involved in problem 2 as problem size increases

- DBTCG(TRICG) for problem 1 has slightly less iterations than DCG, but slightly more work. For problem 2 there is less work and iterations.

- On problem 1, quite a few of the preconditioners involve more work than CG, even though they have less iterations, whereas in problem 2 only the DBCG3-5, POLCG1, and POLCG2 (also POLCG4 for larger problem sizes) are worse than CG.

4.4.2 Convergence

Figures 4.3 and 4.4 show the rates of convergence for the various preconditioners for the problems with a grid size of 20 by 20. The $\log_{10}$ of the 2-norm of the residual is plotted against the number of iterations and, alternatively, the accumulated time. A number of observations can be made from these figures,

- All preconditioners have the characteristic initial oscillations associated with the Conjugate Gradient methods, followed by the rapid decrease in residual norm, usually at a constant rate.

- The CG method for problem 2 has a very oscillatory behaviour, as do POLCG1, POLCG2 and POLCG4, though to a lesser degree. All others have the (practically) constant rate.

4.4.3 Eigenvalue Spectra

Figures 4.5, 4.6, 4.7, and 4.8 show the complete eigenvalue spectrum for each iteration matrix $\tilde{A} = M^{-1}A$ for problems 1 and 2 with size of 10 by 10. Each of the iteration matrices has 100 eigenvalues. With reference to the theory in

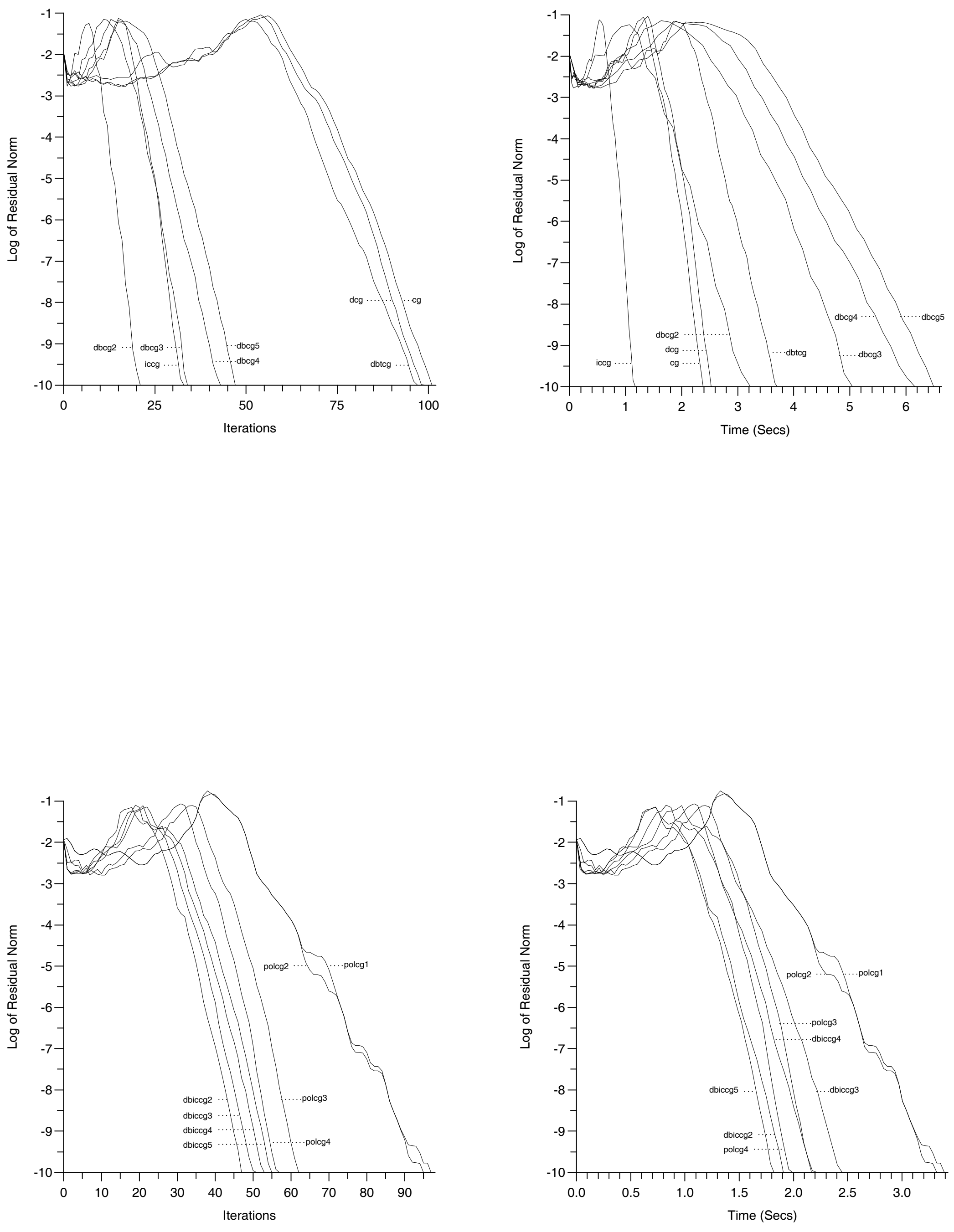
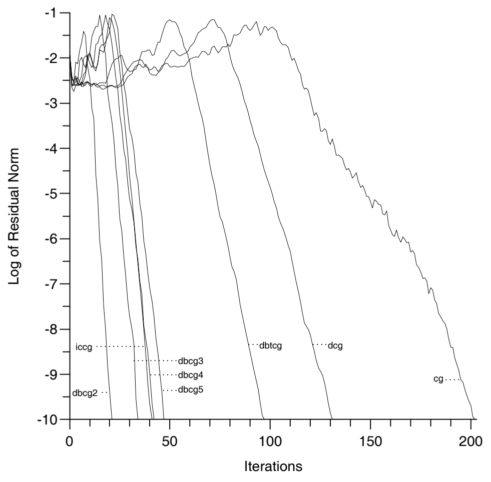
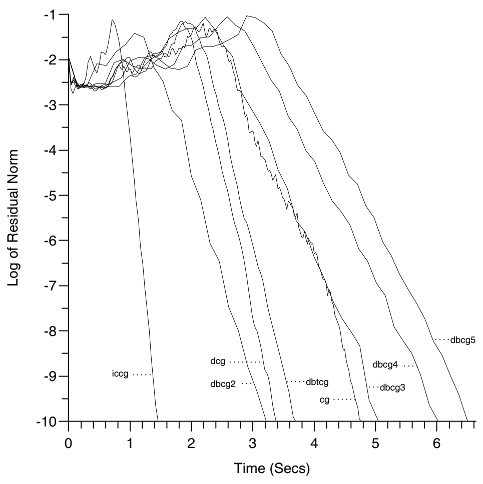
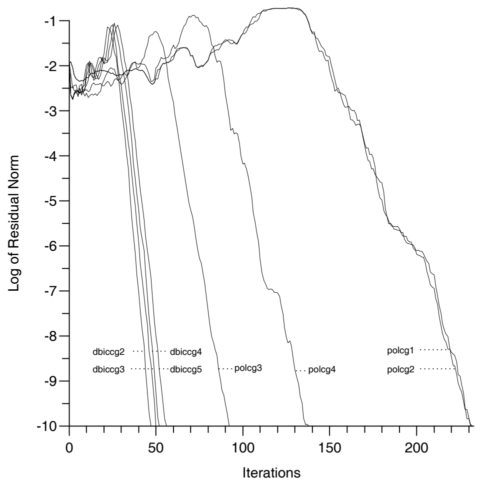


Figure 4.3: Convergence Rates for problem 1 (20 by 20)







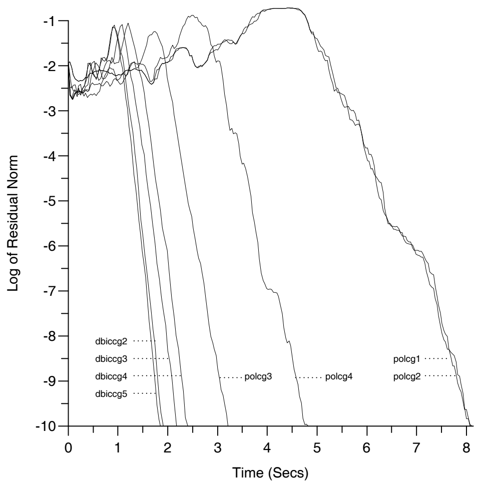


Figure 4.4: Convergence Rates for problem 2 (20 by 20)

Section 3.2, the Figures 4.1 and 4.2, the Tables 4.2, 4.3, 4.4, and 4.5, and the above eigenvalue spectrum figures, a number of observations can be made;

- Unfortunately, the condition number $\kappa_2(\tilde{A})$ does not seem to have a direct effect on convergence speed, although the smaller it is the more likely a faster convergence (in number of iterations).

- A similar thing happens with the spectral radius $\rho(\tilde{A})$, e.g. POLCG1 has a very small spectral radius for problem 1, but the number of required iterations is large.

- All of the methods, apart from POLCG1 and POLCG2 (and POLCG4 for problem 2) have the smallest eigenvalue just above zero. This accounts for the large condition numbers found (maximum/minimum eigenvalues). The exceptions have at least one negative eigenvalue, thus the iteration matrix $\tilde{A}$ is not positive definite. This does not seem to make a difference in problem 1, but it may help to explain the awful convergence properties for problem 2.

- POLCG3 for problem 1 seems to have a good clustering of eigenvalues, although the convergence is not as good as this should indicate.

- The behaviour of the POLCG methods in problem 2 is far worse than that in problem 1, although the eigenvalue spectra (when the matrix stays positive definite) seems to have a similar spread.

- For problem 1, CG and DCG have a similar spacing of their eigenvalues, which matches their similar convergence behaviour, whereas in problem 2, CG has a much less even spread, and consequently worse behaviour.

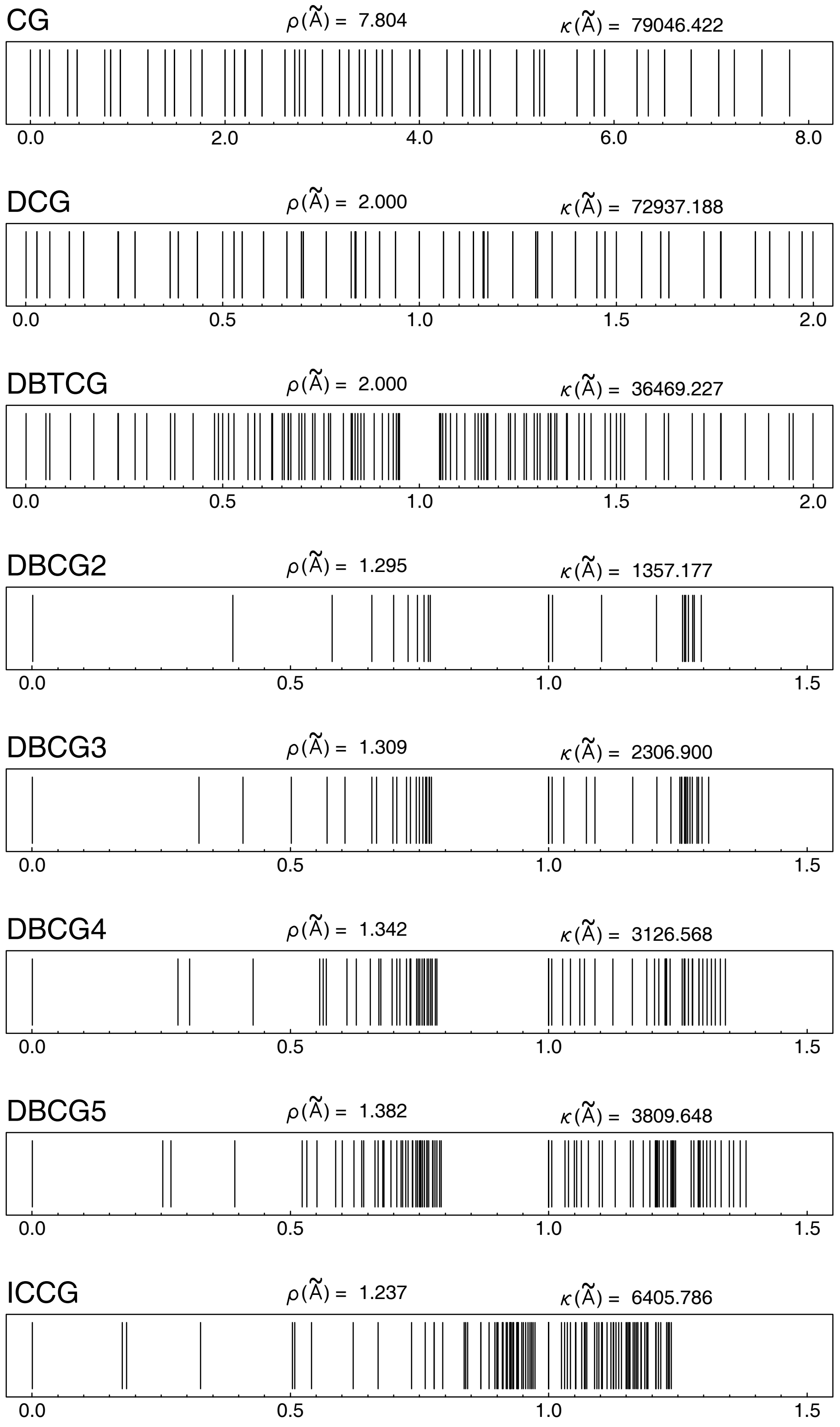


Figure 4.5: Problem 1 Iteration Matrix Eigenvalue Spectra

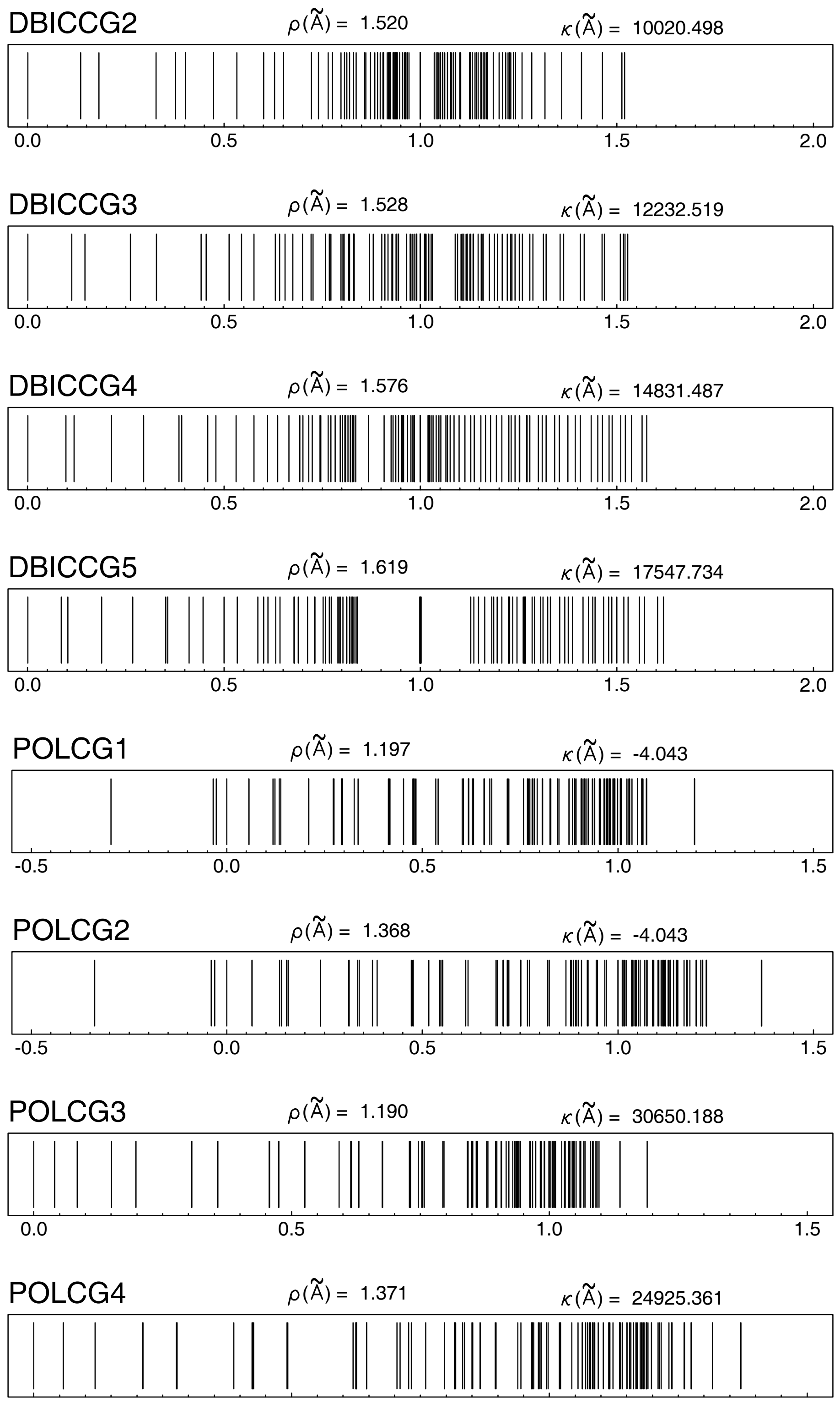


Figure 4.6: Problem 1 Iteration Matrix Eigenvalue Spectra (contd.)

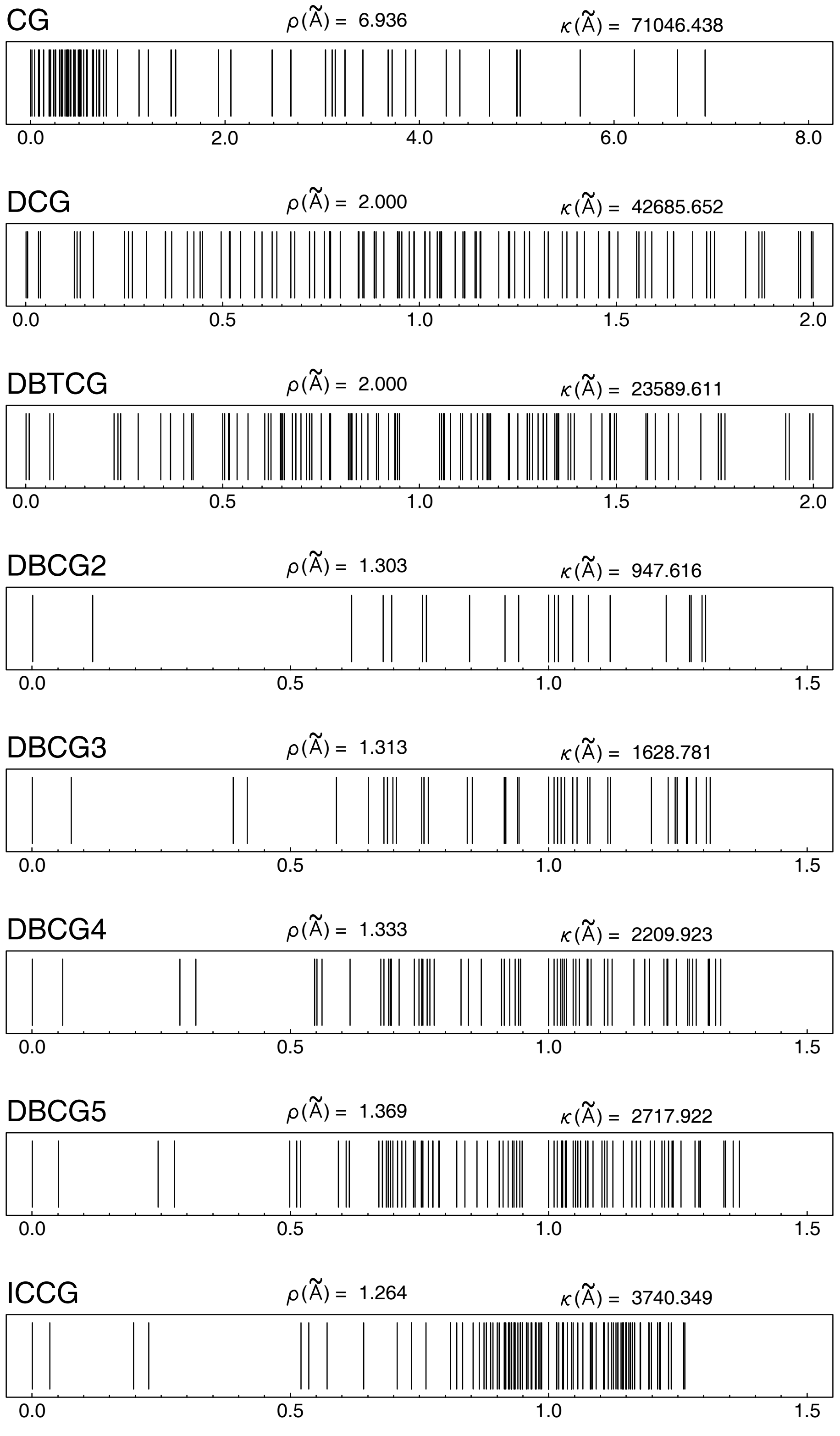


Figure 4.7: Problem 2 Iteration Matrix Eigenvalue Spectra

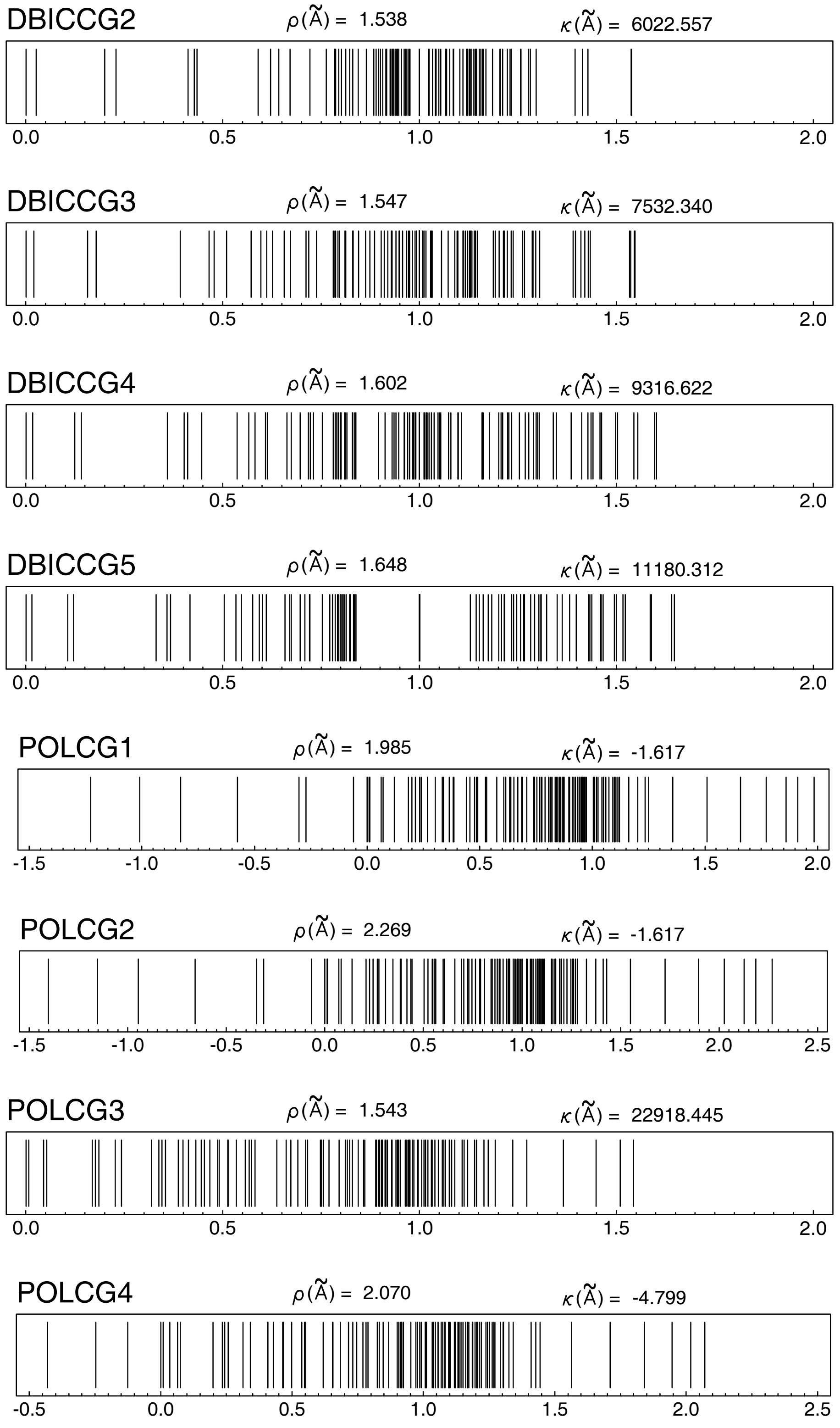


Figure 4.8: Problem 2 Iteration Matrix Eigenvalue Spectra (contd.)

- DBTCG has a better bunching of its eigenvalues around one, and, as predicted by theory, better convergence than DCG (in terms of iterations).

- The DBCG methods all have similar spectra, and although not shown, DBCG1 has all its eigenvalues equal to one, and converges in one iteration! DBCG2 has a lot of repeated eigenvalues, and DBCG3-5 just have these repeated eigenvalues spreading out. All of them are pretty well bunched, and their convergence (in iterations) echoes this.

- The ICCG and DBICCG methods have similar properties to the above DBCG methods.

Chapter 5

Parallel Implementation

First, this chapter describes the hardware and software of the transputer system in the Mathematics Department[1]. Next, the difficulties in parallelising a serial algorithm are presented, and an alternative PCG algorithm, more suited to a parallel programming environment, is described. The problem discretisation, as given in Chapter 2, is then investigated for use in parallel, and some of the preconditioners introduced in Chapter 3 parallelised. Lastly, metrics are introduced to allow comparison of the various parallel PCG algorithms, complete with the results of these comparisons.

5.1 Transputer Systems

5.1.1 The Transputer

The transputer (INMOS 1985) is a single chip microprocessor designed for the construction of parallel systems [9]. To allow this, the transputer has on chip

[1]The system is currently on loan from the Rutherford Appleton Laboratory as part of the DTI/SERC Initiative in the Engineering Application of Transputers.

memory, a RISC[2] processor, a hardware process queue to allow concurrent running of multiple processes on a single transputer, and four bi-directional serial communication links to connect to other transputers, all on a single VLSI[3] chip. The newer transputers (T800 series) have 4Kb of internal memory and also have a built in floating point unit, whereas the older ones (T414 series) only have 2Kb of memory and are only integer based. Both transputers usually have external access to larger ($\approx$1Mb) memory, but this takes longer to access than the internal memory. All components of the transputer operate concurrently, so that communication, floating point computations, and other processor work can occur simultaneously. The communication between units within a single transputer is much faster than communication between transputers. During communication, both processes must be synchronised for the transfer, i.e. they both must pause while the transfer takes place. If both processes are on the same transputer, the communication is performed via the local memory, otherwise the communication hardware is used. The transputer communication hardware has a very small message start up time, the same order as the time to transfer a byte of data, thus even small packets of data are transferred efficiently. There is no great need to send very large data packets as in other multiprocessor systems.

5.1.2 Transputer Networks

The four communication links allow various interconnection topologies to be implemented directly, e.g. pipelines, rings, two dimensional arrays, trees (See Fig-

[2]Reduced Instruction Set Computer, the idea being it is easier to make a very fast simple processor than a complex slower one.

[3]Very Large Scale Integration.

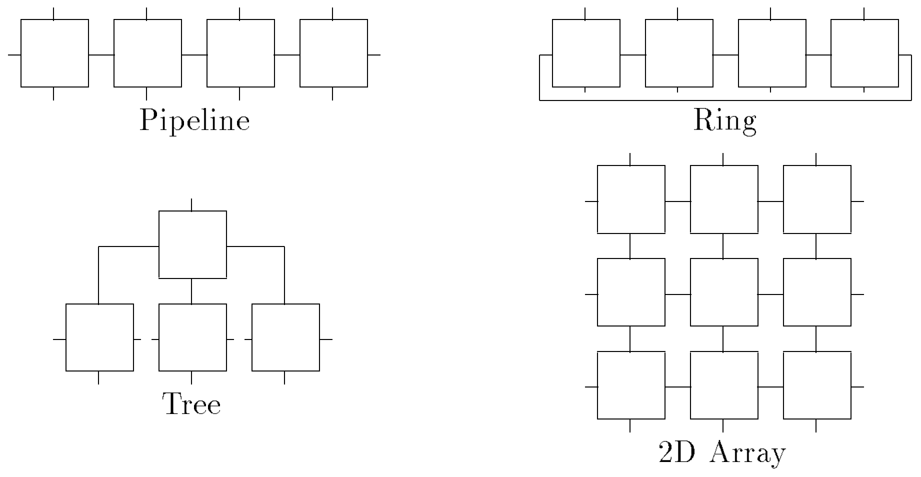


Figure 5.1: Transputer Networks

ure 5.1), and possibly p-dimensional hypercubes with 2^p processors, each one connected to p other processors ($p \leq 4$). More complex interconnection networks can be implemented if switching nodes are used, where basically some (or all) of the communications links go into a large switch that can be programmed to define the network topology. An example of this is the Meiko Computing Surface[4]. The more connected a network is, the more different types of network can be embedded in it, i.e. in a two dimensional array, a ring or a pipeline also exist.

[4]There is a Meiko Surface in the Computer Science Department, Reading, but by the time access to it became available it was too late to investigate.

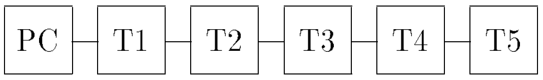


Figure 5.2: Reading Transputer System Network

transputer	T1	T2	T3	T4	T5
processor type	T800	T800	T800	T800	T800
clock speed (MHz)	17.5	20.0	20.0	20.0	25.0
external memory (Kb)	2048	1024	1024	1024	1024
memory access delay	3 cycles	2 cycles	2 cycles	2 cycles	4 cycles

Table 5.1: Reading Network Transputer Information

The network for a particular transputer system is usually a hardware feature (the Meiko is an exception), and in the system available in the Mathematics Department, it consists of five transputers linked in a pipeline (see Figure 5.2), with one end of the pipeline connected to a PC compatible. The types of transputers in this system are described in Table 5.1. The transputer $T1$ is termed the master or root processor, and the others are slave processors. Unfortunately, a p processor pipeline has $2p + 1$ unused communication links, and, therefore, does not make optimum use of the communication facilities, since if two non-adjacent processors want to communicate, they have to pass information through all of the processors in between.

5.1.3 Classification of Parallel Systems

The macroscopic structure of a parallel computer can be classified according to Flynn's Taxonomy [9], this system being based on how the machine relates its instructions to the data it is processing. Since a transputer system is built up from a number of independent processors, each with their own memory, a transputer system is a member of the MIMD (Multiple Instruction stream, Multiple Data stream) class of parallel computers. If, however, each transputer executes the same program, but on its own data, then it can be considered to be a SIMD (Single Instruction stream, Multiple Data stream) system. This is how it has been chosen to use the Reading System, due to the nature of the problem to be solved, i.e. give each processor part of the matrix system.

5.1.4 Software Model

The transputer is designed to efficiently implement the OCCAM language. However since the standard numerical analysts' language is FORTRAN 77, parallel FORTRAN 77 is available, e.g. Reading has the 3L parallel FORTRAN 77 package [1].

There are two main types of software parallelisation available in 3L FORTRAN, tasks and threads.

Tasks

A task is a FORTRAN program. The structure of tasks within a system is static, i.e. tasks are not created and destroyed dynamically during execution, and no hierarchy exists, e.g. tasks cannot have subtasks. The structure of tasks is controlled in the 3L system by a configuration file that indicates the distribution of tasks among the various available processors and describes the links between them [1, 16]. Each task is a separate entity, no data is shared between them, even if they are on the same transputer. They can only communicate via channels which are fixed one way communication paths. A complete transputer application consists of a collection of one or more tasks. A task may contain several concurrent threads.

Threads

A thread is a FORTRAN subroutine. They can be created and destroyed dynamically and a thread my be created within another thread. Starting a thread is like making a subroutine call, except that the calling program regains control immediately, and the thread runs concurrently with the calling program. Threads can share FORTRAN common blocks, and thus can communicate directly.

Communications

The 3L FORTRAN runtime library provides a simple interface to allow tasks to communicate. Basically, it consists of send message and receive message function calls which send and receive a packet of data to or from a channel. The possible channels are determined indirectly by the configuration file [16].

5.2 Parallelising Serial Algorithms

The most obvious approach to parallelise an algorithm is to examine it and convert it into a procedure that operates on composite mathematical objects, such as vectors or matrices, and to split these objects over all of the processors. However, the latest and most efficient serial method is not always suited to this adaptation, and often an older, less efficient serial method may possess a higher degree of inherent parallelism, and be more adaptable.

When a problem is split into a number of subproblems, it is desired that each of the subproblems are independent from the others. This would allow each problem to be solved on a separate processor. Unfortunately, very few algorithms split up into totally independent subproblems, and some global communication needs to be done, e.g. calculation of an inner product where both vectors are distributed over the network. During such a global communication, all processors have to wait to receive their information from the others before they can continue processing[5]. This global synchronisation limits the amount of work that can be done in parallel, and thus affects the performance of the algorithm, thus, each transputer should do a large amount of computation with respect to communication.

5.2.1 Alternative parallel PCG algorithm

The serial PCG algorithm (4.1) given in Section 4 has two vector inner products (three depending on the termination condition) requiring global communication between the processors, and they do not appear in the same position in the algorithm. An alternative algorithm, proposed by Meurant, described in [13], has an extra inner product and two extra vectors of storage, but all the inner products appear at the same point, and thus they can all be calculated at the

[5]In SIMD or shared memory MIMD systems each processor can have access to all of the memory, so the problems are not as serious.

same time.

```
$\underline{w} = A\underline{x}$
$\underline{r} = \underline{b} - \underline{w}$
$\underline{z} = M^{-1}\underline{r}$
$\underline{p} = \underline{z}$
do $i = 0, 1, \ldots, \{$
    $\underline{w} = A\underline{p}$
    $\underline{v} = M^{-1}\underline{w}$
    $\mathrm{ip1} = \underline{v}^T\underline{w}$
    $\mathrm{ip2} = \underline{w}^T\underline{p}$                    (5.1)
    $\mathrm{ip3} = \underline{r}^T\underline{z}$
    $\alpha = \frac{(\mathrm{ip3})}{(\mathrm{ip2})}$
    $s = \alpha^2(\mathrm{ip1}) - (\mathrm{ip3})$
    $\beta = \frac{s}{(\mathrm{ip3})}$
    $\underline{x} = \underline{x} + \alpha\underline{p}$
    $\underline{r} = \underline{r} - \alpha\underline{w}$
    $\underline{z} = \underline{z} - \alpha\underline{v}$
    $\underline{p} = \underline{z} + \beta\underline{p}$
}
```

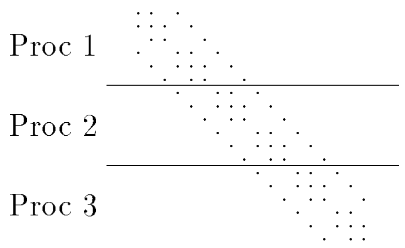


Figure 5.3: Sample split of A over three processors

5.3 Parallelising the Matrix Problem

The n by n matrix problem, where $n = nx \times ny$,

$$A\underline{x} = \underline{b}$$

now has to be split up so that it can be solved on p processors. It is sensible to allocate similar size parts of the problem to each processor, to make sure that each processor does a similar amount of work. Noting from Section 3.4.1 that the block preconditioners require the block size, s_i to be an integer multiple of nx, for the ease of programming, we enforce these restrictions for all the preconditioners. The matrix A is split up into p parts of s_i contiguous rows of A, where $i = 1, 2, \ldots, p$. The same happens for all vectors on each processor. Each processor i stores its part of A in the same way as Section 4.2.1 for the serial case, e.g. for a 18 by 18 matrix with $nx = 3$ and $ny = 6$, with three processors, A is split as in Figure 5.3, where, as in Section 4.2.1, processor two stores $\underline{a}^d_7$ to $\underline{a}^d_{12}$, $\underline{a}^1_6$ to $\underline{a}^1_{12}$, and $\underline{a}^2_4$ to $\underline{a}^2_{12}$. In general, if a block starts at equation i, with block size s, we store $\underline{a}^d_i$ to $\underline{a}^d_{i+s-1}$, $\underline{a}^1_{i-1}$ to $\underline{a}^1_{i+s-1}$, and $\underline{a}^2_{i-nx}$ to $\underline{a}^2_{i+s-1}$.

For a matrix-vector multiplication, $A\underline{x}$, most of the required information is available in the current processor. If i and s are defined as above, then we need the elements of $\underline{x}$ from the 'above' processor $\underline{x}_{i-nx}$ to $\underline{x}_{i-1}$, and the elements of $\underline{x}$

from the 'below' processor $\underline{x}_{i+s}$ to $\underline{x}_{i+s+nx-1}$.

5.4 Communication Harnesses and Libraries

In order to effectively implement an algorithm on an arbitrary number of processors connected in an arbitrary communications network, it is necessary to have an effective communications harness [3] that takes out the need for the user to know the details of the network. This harness provides a transparent message routing system that allows any processor to send a message to any other, irrespective of the network topology. The harness runs in parallel with the processes on each transputer. The harness is implemented as a library of function calls, which the user links in when compiling their program. The main process on each transputer calls an initialise function which sets up all the required information about the network, and then all communication is done via the harness library calls. The main advantage of harnesses is that if, say, an algorithm was developed on a pipeline transputer system, to move it to a two dimensional array network would only require the writing of an array harness, the actual algorithm would remain unchanged.

In a similar vein, it is a good idea to write a library of subroutines for doing various useful things, such as matrix-vector multiplication or vector inner products, where the matrices or vectors are distributed over the network (See [14] for the implementation of a complete matrix and vector package on a Hypercube). The library is written in terms of the harness subroutine calls, so it is instantly portable, and again independent of the underlying network topology.

It was decided, due to the lack of time, to implement a single library to do the function of both of the above, tailored for a pipeline of transputers and the problem in hand. It assumes that the configuration file for the transputers has been set up correctly. It provides the following services

- An initialisation subroutine to assign a unique identifier to each transputer in the pipeline.
- A vector inner product subroutine to distribute local parts of the inner product over the network, and to return with the global inner product.
- A matrix-vector multiplication support subroutine to obtain the parts of the vector not stored in the current transputer, as in Section 5.3.
- A subroutine to collect up all the partial solutions and deposit them in the master processor.

5.4.1 Initialisation

Although it is possible to determine the network topology of a system at run-time [3], here it is assumed that the network is a pipeline, and that the connections between the processors never change.

The master processor calls the initialisation subroutine with details of the number of processors, p, required to use and the dimensions of the matrix system, nx and ny. The slave processors just call the routine and wait for the return value. The subroutine returns to each processor a unique identifier, the block size s_i it has been assigned and the index of the first equation assigned. From these the processor can work out what entries of the matrix A and the right hand side $\underline{b}$ to generate.

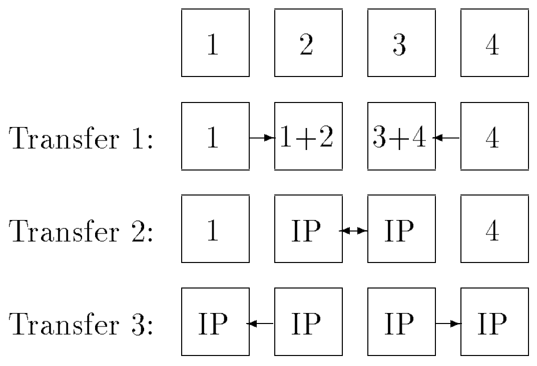


Figure 5.4: Inner Product transfers for four processors

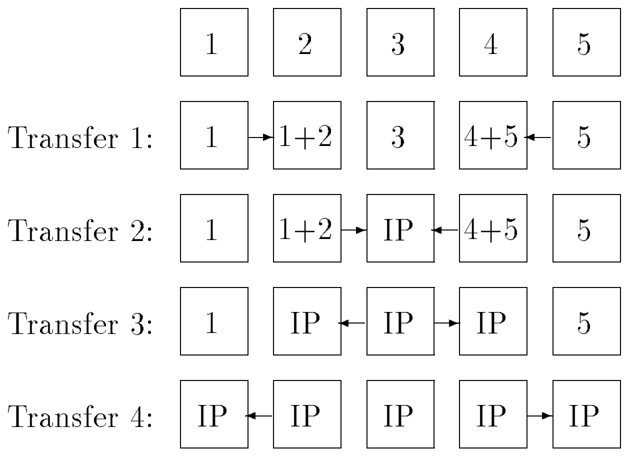


Figure 5.5: Inner Product transfers for five processors

5.4.2 Inner Product

Each processor works out the local inner product using the partial vectors stored locally, then all processors call the inner product subroutine. This subroutine calculates the global inner product IP in $p-1$ communications, e.g. with four processors the sequence of transfers is shown in Figure 5.4, and that for five processors in Figure 5.5. and similarly for any other number of processors.

This method of passing the partial inner product from the outside to the centre and back again is the most efficient way to calculate an inner product on a pipeline, but it is not the best way over all networks. The fastest requires a binary tree network, where the number of communications required is $O(\log p)$, as opposed to $O(p)$ for the pipeline [14].

5.4.3 Matrix-vector Multiplication

This subroutine is called before the transputer does its local matrix-vector multiplication to obtain the parts of the vector that are stored in the 'above' and 'below' transputers (See Section 5.3).

5.4.4 Solution Collection

This subroutine collects all of the partial solutions $\underline{x}_i$ and assembles them into the solution, $\underline{x}$, on the master transputer.

5.5 Parallel Preconditioner Implementation

The preconditioners investigated in parallel are the same ones that were looked at in Section 4. All of those, except the ICCG algorithm and the POLCG algorithm, are inherently parallel, that is the work involved in putting the problem on a set of processors and the implementation of the harness suffices to allow the parallel implementation to be done.

5.5.1 POLCG Preconditioner

The POLCG algorithm, can be easily implemented in parallel if the storage of A on each processor (Section 5.3) is modified slightly. The diagonal of A, $\underline{a}^d$, has to extend nx positions above the first equation in the block, i.e. as in Section 5.3 we need $\underline{a}^d_{i-nx}$ to $\underline{a}^d_{i+s-1}$, $\underline{a}^1_{i-1}$ to $\underline{a}^1_{i+s-1}$, and $\underline{a}^2_{i-nx}$ to $\underline{a}^2_{i+s-1}$. This modification allows the storage of the equivalent part of the A^{-1} approximation as that of A in each processor, thus the $A\underline{x}$ and the approximate $A^{-1}\underline{x}$ multiplications have the same form.

5.5.2 ICCG Preconditioner

There are two problems involved in parallelising ICCG, one is doing a parallel Incomplete Cholesky factorising of the matrix A, and the other is the solution of the $LL^T\underline{z} = \underline{r}$ equation, which involves a forward and backward substitution across the network of processors. Unfortunately, the detailed study of these two problems could not be performed, due to a lack of time.

Parallel Cholesky Decomposition

The sparse Cholesky Decomposition in parallel is investigated in [6], and this notices that the Cholesky algorithm (3.16) can be written as the following

```
do $i = 1, 2, \ldots, n$
    do $j = 1, 2, \ldots, i-1$
        cmod$(i, j)$
    cdiv$(i)$
```

where task cmod(i, j) modifies column i by column j where $j < i$, and task cdiv(i) divides column i by a scalar. The task tcol(i) computes the ith column of the Cholesky factor. Note that the first cdiv(i) cannot start until cmod(i, j) has been completed for all $i < j$, and column i can only be used to modify subsequent columns after cdiv(i) has been completed. However, there is no restriction on the order of the cmod tasks. When A is sparse, the order restrictions become less, and, by constructing an elimination tree [6], the order of possible execution can be determined.

In the serial case, with the A generated from the discretisation in Chapter 2, it was possible to drastically simplify the incomplete Cholesky algorithm. A similar feat should also be possible in parallel.

Parallel Solution of $LL^T\underline{z} = \underline{r}$

The standard way of solving a triangular system $L\underline{z} = \underline{r}$ is to use forward (or backward) substitution. The problem with this, is that it is inherently serial, since the value of one unknown depends on the value of the previous one. This problem is compounded with the fact that ICCG requires a forward and a backward substitution pass for the solution of $LL^T\underline{z} = \underline{r}$. Several parallel methods have been proposed in [8],

- Fan-in and Fan-out algorithms
- Wavefront algorithms
- Cyclic algorithms

but these all require a ring network topology, and a different mapping of the matrices and vectors onto the processors, both of which conflict with the hardware available and the parallel decomposition proposed in Section 5.3.

5.5.3 TRICG Preconditioner

Although this preconditioner was not considered in serial due to the structure of the matrix A from the problem discretisation, an efficient tridiagonal solver for a pipeline network was discovered in [19]. It used the technique of cyclic reduction of the tridiagonal system followed by a backward unfolding scheme. See [19] for the details.

5.6 Performance Metrics

As for the serial case, the timing metrics (Section 4.3) preconditioning time, iteration time, and total time are recorded, but in this case only for the master processor. Since no extra preconditioners have been considered in this Chapter, the iteration matrix metrics (Section 4.3) remain the same as those in Chapter 4. here.

5.6.1 Serial/Parallel Comparisons

It is natural, after parallelising an algorithm, to measure its performance related to the serial algorithm in some way. The most commonly accepted measure is speedup [18], S_p, where

$$S_p = \frac{\text{execution time on one processor}}{\text{execution time on } p \text{ processors}}$$

This definition includes any overheads involved in the parallel algorithm, such as communication, but one flaw is that one is usually more interested in how much faster a problem can be solved with p processors. Thus a modified definition exists,

$$S_p^* = \frac{\text{execution time on one processor with best serial algorithm}}{\text{execution time on } p \text{ processors with parallel algorithm}}$$

The best serial algorithm in this case is the ICCG algorithm. There are other modifications that can be done (See [18]) which model the work done in the parallel algorithm more accurately, but for the purposes of this dissertation, the above two are sufficient. Once the speedup has been determined, it is natural to ask how efficiently the parallel system is being used. The efficiency of a parallel algorithm is defined as

$$E_p = \frac{S_p}{p}$$

For the perfect parallel algorithm, $S_p = p$ and $E_p = 1$, but as this is very unlikely, a parallel algorithm is usually considered efficient if $E_p \geq 0.9$, i.e. 90% efficient.

5.7 Parallel Implementation Results

Unfortunately, due to unforeseen problems with programming the transputer system, only the CG and DCG methods were programmed in parallel. The other preconditioners were implemented, but either they did not converge, or gave errors during execution. Luckily, since most of the preconditioners are readily divisible into independent subproblems, the speedups obtained for CG and DCG are applicable to the rest.

Tables 5.2, 5.3, 5.4, and 5.5 show the time taken for the CG and DCG preconditioners for one, two, three, four, and five processors, the speedup S_p, and the parallel efficiency E_p (as a percentage). Since only the two preconditioners were implemented, and their performance is no where near that of the best serial

ICCG algorithm, the S_p^* speedup is not calculated. The number of iterations is obviously the same as for the serial case. The times given when one processor is being used are not the same as those obtained when using the serial algorithm, they are slightly larger. This is due to the fact that the modified PCG algorithm is being used, not the basic PCG algorithm.

- All the results show a decrease in iteration time for increasing the processors from one to five. As the results for the 5x5 systems show, this decrease may not last for much longer after five processors.

- The rate of decline of the parallel efficiency drops as the problem size gets larger.

- The smaller the problem size and the larger the number of processors, the less efficient the method will be, due to the increase in communication traffic compared to computation.

- Parallel efficiencies of over 90% are obtained when the problem size is greater than or equal to 20 by 20.

- Some of the entries in the E_p column are greater than 100% due to the unequal splitting of the problem over the transputers. As shown in Table 5.1, not all the transputers are the same speed, and, for example, a 20 by 20 problem on 3 processors will put 120 equations on the first, and 140 each on the second and third, and since transputers 2 and 3 are faster than the first, the equations get solved in the time that it should take for 360.

Problem Size	Processors	Time	S_p	E_p
5x5	1	553	1.00	100
	2	298	1.85	92.8
	3	240	2.30	76.8
	4	237	2.33	58.3
	5	206	2.68	53.7
10x10	1	6128	1.00	100
	2	3257	1.88	94.1
	3	2092	2.93	97.6
	4	1628	3.76	94.1
	5	1542	3.97	79.5
20x20	1	50172	1.00	100
	2	25487	1.97	98.4
	3	15658	3.20	107
	4	13237	3.79	94.8
	5	10816	4.63	92.8
30x30	1	175373	1.00	100
	2	88135	1.99	99.5
	3	59045	2.97	99.0
	4	41870	4.18	104
	5	36191	4.84	97.0

Table 5.2: CG for problem 1 with increasing number of processors

Problem Size	Processors	Time	S_p	E_p
5x5	1	999	1.00	100
	2	514	1.94	97.2
	3	431	2.32	77.3
	4	424	2.36	58.9
	5	355	2.81	56.3
10x10	1	12330	1.00	100
	2	6384	1.93	96.6
	3	4093	3.01	100
	4	3149	3.92	97.9
	5	3003	4.11	82.1
20x20	1	103638	1.00	100
	2	52344	1.98	98.9
	3	32136	3.22	107
	4	27132	3.82	95.5
	5	22146	4.67	93.6
30x30	1	372720	1.00	100
	2	187323	1.99	99.5
	3	125432	2.97	99.0
	4	88924	4.19	105
	5	76854	4.84	97.0

Table 5.3: CG for problem 2 with increasing number of processors

Problem Size	Processors	Time	S_p	E_p
5x5	1	534	1.00	100
	2	283	1.89	94.3
	3	229	2.33	77.7
	4	224	2.38	59.6
	5	196	2.72	54.5
10x10	1	6119	1.00	100
	2	3309	1.85	92.5
	3	2114	2.89	96.5
	4	1631	3.75	93.8
	5	1552	3.94	78.9
20x20	1	51429	1.00	100
	2	26129	1.97	98.4
	3	16031	3.21	107
	4	13537	3.80	95.0
	5	11041	4.66	93.2
30x30	1	177309	1.00	100
	2	89224	1.98	99.4
	3	59843	2.96	98.8
	4	42423	4.17	104
	5	36661	4.84	96.7

Table 5.4: DCG for problem 1 with increasing number of processors

Problem Size	Processors	Time	S_p	E_p
5x5	1	874	1.00	100
	2	465	1.88	94.0
	3	374	2.34	77.9
	4	369	2.37	59.2
	5	318	2.75	55.0
10x10	1	8118	1.00	100
	2	4290	1.89	94.6
	3	2742	2.96	98.7
	4	2114	3.84	96.0
	5	2010	4.03	80.8
20x20	1	67727	1.00	100
	2	34124	1.98	99.2
	3	20934	3.24	108
	4	17677	3.83	95.8
	5	11423	4.68	93.9
30x30	1	231713	1.00	100
	2	116610	1.99	99.3
	3	78211	2.96	98.8
	4	55433	4.18	104
	5	47902	4.84	96.7

Table 5.5: DCG for problem 2 with increasing number of processors

Chapter 6

Conclusions

Of all the serial preconditioning strategies investigated in Chapter 4, the ICCG method was found to outperform all the rest, and on the larger problem sizes, outperformed a direct Cholesky Decomposition of the matrix A. For problem 1, a surprising number of the preconditioning strategies were outperformed by the standard CG method, although for problem 2, most did outperform CG. The only other methods, apart from ICCG, that performed well over both the sample problems were the DBICCG methods, although some success was had with the POLCG methods used on problem 1. The overall performance of the POLCG methods was not as good as expected, especially on problem 2. The same applied to the DBTCG method, although this was better on problem 2. Since most physical problems would be far closer to problem 2 than problem 1, we would want to investigate further the methods which work well on problem 2.

No concrete evidence was obtained for direct links between condition number of the iteration matrix and the CG convergence, or spectral radius and CG convergence, although the results in Chapter 4 did seem to indicate a fair correlation. The clustering of eigenvalues appears to have a greater effect, although no quantitative measure of eigenvalue clustering could be found in order to examine the effect in detail.

A transputer library for the Reading transputer system was successfully implemented for the pipeline of processors, but, unfortunately, only the CG and DCG methods could be implemented in parallel, and further work is required to debug the remaining preconditioners. This has not stopped the investigation of the parallelisation of the CG and DCG methods, and Chapter 5 has shown that where problem size was far greater than number of processors, parallel efficiencies of over 90% can be achieved. Since most of the remaining preconditioners are all of a similar structure to CG and DCG (they are all inherently 'blocked' into independent subproblems), the high efficiencies should also be obtainable. The ease of obtaining these high parallel efficiencies was surprising considering that the communication network was a pipeline, but the possible overheads involved with a pipeline probably would not appear until a far larger number of processors were used. Further work is required on the parallelisation of the ICCG method, although, as shown in the serial case, the DBICCG methods can perform well over both sample problems (and, in theory, they are easy methods to parallelise).

Overall, there is a lot more work that can be done on the parallel PCG methods. More specifically, different communication networks should be investigated, and a more complex communication harness and a library of frequently used mathematical functions should be implemented.

Appendix A

Basic Matrix Theory

This appendix describes some of the basic matrix theory associated with this dissertation. Unless otherwise stated, the theory applies to n by n real, square matrices. The elements of the matrix A are denoted a_{ij}, where i and j range from 1 to n.

Symmetry

The matrix A is symmetric if

$$A = A^T$$

Positive Definite

The matrix A is positive definite if

$$\forall \underline{x} \neq 0, \quad \underline{x}^T A \underline{x} > 0$$

Diagonal Dominance

The matrix A is diagonally dominant if

$$\forall i, \qquad |a_{ii}| \geq \sum_{j=1}^{i-1} |a_{ij}| + \sum_{j=i+1}^{n} |a_{ij}|$$

It is strictly diagonally dominant if the inequality $\geq$ is a strict inequality $>$.

Irreducible

The matrix A is irreducible if

$$\nexists P \text{ such that } P^T A P = \begin{bmatrix} A_1 & A_2 \\ 0 & A_3 \end{bmatrix}$$

i.e. the matrix A cannot be partitioned so that there are two or more independent groups of equations.

Irreducible Diagonal Dominance

The matrix A is irreducibly diagonally dominant if it is diagonally dominant, with a strict inequality for at least one i, and it is irreducible.

Existence of A^{-1}

If A is irreducibly diagonally dominant, $a_{ii} > 0$, and $a_{ij} \leq 0$ for $i \neq j$, then A is positive definite and A^{-1} exists.

Eigenvalues

The n eigenvalues, λ_i, of the matrix A are the solutions of

$$\det(A - \lambda I) = 0$$

They are all real if A is symmetric, and all positive if A is positive definite.

Spectrum

The spectrum of the matrix A, $\lambda(A)$, is the set of all the eigenvalues of A

$$\lambda(A) = \{\, \lambda_i \mid i = 1, 2, \ldots, n \,\}$$

Spectral Radius

The spectral radius, $\rho(A)$, of the matrix A is defined by

$$\rho(A) = \max_{1 \le i \le n} (|\lambda_i|)$$

Vector Norms

A vector norm on $\mathbb{R}^n$ is a function $f : \mathbb{R}^n \mapsto \mathbb{R}$, denoted $f(\underline{x}) = ||\underline{x}||$, such that

$$\begin{aligned} f(\underline{x}) &\geq 0 && \underline{x} \in \mathbb{R}^n,\ f(\underline{x}) = 0 \text{ iff } \underline{x} = 0 \\ f(\underline{x} + \underline{y}) &\leq f(\underline{x}) + f(\underline{y}) && \underline{x},\ \underline{y} \in \mathbb{R}^n \\ f(\alpha \underline{x}) &= |\alpha| f(\underline{x}) && \alpha \in \mathbb{R},\ \underline{x} \in \mathbb{R}^n \end{aligned}$$

Some useful norms are the p-norms, where

$$||\underline{x}||_p = \sqrt[p]{\sum_{i=1}^{n} |\underline{x}_i|^p}, \qquad p \geq 1$$

of which the 1, 2, and ∞ norms are the most important

$$\begin{aligned} ||\underline{x}||_1 &= \sum_{i=1}^{n} |\underline{x}_i| \\ ||\underline{x}||_2 &= \sqrt{\sum_{i=1}^{n} |\underline{x}_i|^2} = \sqrt{\underline{x}^T \underline{x}} \\ ||\underline{x}||_\infty &= \max_{1 \le i \le n} |\underline{x}_i| \end{aligned}$$

Another useful norm is the A-norm

$$||\underline{x}||_A = \sqrt{\underline{x}^T A \underline{x}}$$

Matrix Norms

Similar to the vector norms, matrix norms are a function $f : \mathbb{R}^{m\times n} \mapsto \mathbb{R}$, denoted $f(A) = ||A||$, such that

$$\begin{array}{rcll} f(A) & \geq & 0 & A \in \mathbb{R}^{m\times n},\ f(A) = 0 \text{ iff } A = 0 \\ f(A+B) & \leq & f(A) + f(B) & A,\ B \in \mathbb{R}^{m\times n} \\ f(\alpha A) & = & |\alpha| f(A) & \alpha \in \mathbb{R},\ A \in \mathbb{R}^{m\times n} \end{array}$$

The frequently used norms are the Frobenius norm

$$||A||_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2}$$

and the p-norms

$$||A||_p = \sup_{\underline{x}\neq 0} \frac{||A\underline{x}||_p}{||\underline{x}||_p}$$

The 1, 2, and ∞ matrix norms have alternative expressions

$$\begin{aligned} ||A||_1 &= \max_{1\leq j\leq n} \sum_{i=1}^{m} |a_{ij}| \\ ||A||_2 &= \sqrt{\rho(A^T A)} \\ ||A||_\infty &= \max_{1\leq i\leq m} \sum_{j=1}^{n} |a_{ij}| \end{aligned}$$

Condition Number

The condition number of a matrix A, $\kappa(A)$, is defined as

$$\kappa(A) = ||A||\,||A^{-1}||$$

with the convention that $\kappa(A) = \infty$ if A is singular. It is a measure of how sensitive the $A\underline{x} = \underline{b}$ problem is to numerical solution, the larger the condition number the more sensitive the problem is to rounding errors. Note that $\kappa(A)$

depends on the underlying matrix norm, and when this norm is to be stressed, subscripts are used, e.g.

$$\kappa_2(A) = ||A||_2 \, ||A^{-1}||_2$$

The $\kappa_2(A)$ norm can also be expressed as

$$\kappa_2(A) = \frac{\lambda_{max}}{\lambda_{min}}$$

where λ_{max} and λ_{min} are the maximum and minimum eigenvalues of A.

p-cyclic Matrix

The matrix A is weakly cyclic index k (> 1) if there exists a permutation matrix P such that PAP^T is of the form

$$PAP^T = \begin{bmatrix} 0 & 0 & \cdots & \cdots & 0 & A_{1,k} \\ A_{21} & 0 & & & 0 & 0 \\ 0 & A_{32} & \ddots & & & \vdots \\ \vdots & 0 & \ddots & \ddots & & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & A_{k,k-1} & 0 \end{bmatrix}$$

If the Jacobi matrix, $J = D^{-1}(L + U)$, where D is the diagonal of A, L is the lower triangular part of A, and U is the upper triangular part of A, is weakly cyclic index p (≥ 2), then A is a p-cyclic matrix. It can be shown that a block tridiagonal matrix is a 2-cyclic matrix [22].

Bibliography

[1] 3L Ltd. *Parallel FORTRAN User Guide*, December 1990. Software version 2.1.3.

[2] Khalid Aziz and Antonin Settari. *Petroleum Reservoir Simulation.* Applied Science Publishers Ltd., London, 1979.

[3] M.A. Baker, K.C. Bowler, and R.D. Kenway. MIMD implementations of linear solvers for oil reservoir simulation. *Parallel Computing*, 16:313–334, 1990.

[4] Paul Concus, Gene H. Golub, and G. Meurant. Block preconditioning for the Conjugate Gradient method, July 1982. LBL-14856.

[5] Paul Concus, Gene H. Golub, and Dianne P. O'Leary. A generalized Conjugate Gradient method for the numerical solution of elliptic partial differential equations. In J.R. Bunch and D.J. Rose, editors, *Sparse Matrix Computations*, pages 309–332. Academic Press, New York, 1976.

[6] Alan George, Michael T. Heath, Joseph Liu, and Esmond Ng. Sparse Cholesky factorization on a local-memory multiprocessor. *SIAM Journal of Scientific and Statistical Computing*, 9(2):327–340, March 1988.

[7] Gene H. Golub and Charles F. van Loan. *Matrix Computations.* Johns Hopkins University Press, 2nd edition, 1989.

[8] Michael T. Heath and Charles H. Romine. Parallel solution of triangular systems on distributed-memory multiprocessors. *SIAM Journal of Scientific and Statistical Computing*, 9(3):558–588, May 1988.

[9] R.W. Hockney and C.R. Jesshope. *Parallel Computers 2: Architecture, Programming and Algorithms.* Adam Hilger, Bristol and Philadelphia, 2nd edition, 1988.

[10] D.A.H. Jacobs. Preconditioned Conjugate Gradient methods for solving systems of algebraic equations. Technical Report RD/L/N 193/80, Central Electricity Research Laboratories, October 1981.

[11] Olin G. Johnson, Charles A. Micchelli, and George Paul. Polynomial preconditioners for Conjugate Gradient calculations. *SIAM Journal of Numerical Analysis*, 20(2):362–376, April 1983.

[12] David S. Kershaw. The Incomplete Cholesky–Conjugate Gradient method for the iterative solution of systems of linear equations. *Journal of Computational Physics*, 26:43–65, 1978.

[13] P. Lockey and R.W. Thatcher. Efficient implementation of preconditioned Conjugate Gradients on a transputer network. Technical Report Numerical Analysis Report No. 217, Department of Mathematics, University of Manchester, June 1992.

[14] Oliver A. McBryan and Eric F. van de Velde. Hypercube algorithms and implementations. *SIAM Journal of Scientific and Statistical Computing*, 8:227–287, 1987.

[15] J.A. Meijerink and H.A. van der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Mathematics of Computation*, 31(137):148–162, 1977.

[16] Keiran J. Neylon. Guide to using the transputer system. Department of Mathematics, University of Reading, November 1991.

[17] Kieran J. Neylon. Block iterative methods for three-dimensional groundwater flow models. Master's thesis, Department of Mathematics, University of Reading, September 1991.

[18] James M. Ortega and Robert G. Voigt. Solution of partial differential equations on vector and parallel computers. *SIAM Review*, 27(2):149–240, 1985.

[19] Fabio Reale. A tridiagonal solver for massively parallel computer systems. *Parallel Computing*, 16:361–368, 1990.

[20] J.K. Reid. On the method of Conjugate Gradients for the solution of large sparse systems of linear equations. In J.K. Reid, editor, *Proceedings of the Conference on Large Sparse Systems of Linear Equations*, pages 231–254. Academic Press, New York, 1971.

[21] B.T. Smith, Y. Ikebe Boyle, V.C. Klema, and C.B. Moler. *Matrix Eigensystem Routines: EISPACK Guide*. Springer Verlag, New York, 2nd edition edition, 1970.

[22] R.S. Varga. *Matrix Iterative Analysis.* Prentice Hall, 1962.