INVERTING A POSITIVE DEFINITE MATRIX

A COMPARATIVE STUDY OF TWO ALGORITHIMS

A THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQIREMENTS

FOR THE DEGREE OF MASTER OF SCIENCE

IN THE GRADUATE SCHOOL OF THE

TEXAS WOMAN'S UNIVERSITY

COLLEGE OF ARTS AND SCIENCE

BY

TEGE RADCLIFFE, B.S.

DENTON, TEXAS

DECEMBER 1999

TEXAS WOMAN'S UNIVERSITY
DENTON, TEXAS

Nov. 12, 1999
Date

To the Associate Vice President for Research and Dean of the Graduate School:

I am submitting herewith a thesis written by Tege Radcliffe entitled "Inverting A Positive Definite Matrix. A Comparative Study of Two Algorithms." I have examined this thesis for form and content and recommended that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in mathematics.

W. J. Zimmermann
Dr. Wayne J. Zimmermann, Major Professor

We have read this thesis
and recommend its acceptance:

Department Chair

Accepted:

Leslie M Thompson
Associate Vice President for Research
and Dean of the Graduate School

ABSTRACT

INVERTING A POSITIVE DEFINITE MATRIX
A COMPARATIVE STUDY OF
TWO ALGORITHMS

TEGE RADCLIFFE

DECEMBER 1999

The purpose of the paper is to investigate the efficiency of two matrix-inverting algorithms that are restricted to positive definite matrices. The selected algorithms are a Guass-Jordan method and a gradient projection optimization scheme.

To investigate the efficiency, we will be comparing the two algorithms and the two hardware systems. The primary system is a vonNeumann machine, specifically a PC. The other system is a parallel processor we have designed specifically for matrix inversion and implemented by simulation.

The paper begins with a review of some basic results concerning positive definite matrices. Following this review, the paper describes the two mathematical algorithms used to determine the inverse and how they will be implemented on the PC. Next we describe the architecture of the parallel processor that will be used to simulate matrix inversion. Following the description of the parallel processor, the implementation of each algorithm on the parallel system is described. To test the efficiency of the algorithms we apply each to several large positive definite matrices and measure the time.

The paper concludes with the results of the efficiency study in which each algorithm is compared on the selected system and each system is compared for the selected algorithm.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Introduction

When selecting a numerical method for matrix inversion, the speed and accuracy of the algorithm should be taken in to account when trying to determine which method to select, especially since matrix inversion is very costly in time and storage when the matrices are large. This paper compares two methods for matrix inversion hoping to determine which algorithm and architecture is more efficient. The two methods we have chosen for this paper are a Gauss-Jordan elimination method and a gradient projection iterative method.

Matrix inversion has many applications, as in solving the systems of the form $Ax = b$, where x is an unknown vector, A the coefficient matrix, and b the unknown vector. The system of linear equations can be solved by $x = A^{-1}b$. Once the inverse is found this equation can be solved for any number of known vectors b.

The computer's architecture can play an important role in increasing the efficiency of an algorithm. Today's computers follow the same basic design principle formulated in the late 1940's and attributed to John Von Neumann. The architecture and mode of operation allow only one instruction to be executed at a time. With the need for faster computers, another type of architecture that is growing in popularity is the parallel

processor. A parallel system consists of multiple processing units that join together to solve a problem by working concurrently on different parts of the problem, thus allowing many instructions to be executed at one time. The two algorithms for matrix inversion will be modified to execute on a hypothetical parallel processor.

Although the problem of finding the inverse of a positive definite matrix is highly restrictive relative to that of finding the inverse of an arbitrary matrix, it is important since there are many applications that require that the matrix be positive definite. For example, the learning algorithm involving steepest descent and the LMS algorithm due to Widrow[4] require that the associated matrix be positive definite. The two learning algorithms are used to solve problems related to adaptive signal processing. But the application of positive definite matrices is not limited to these problems. Shampire, Allen, and Pruess [3] indicate that positive definite matrices appear in problems related to least squares fitting of data and finite element solution of partial differential equations. The most important property of the positive definite matrix is that the hypersurface $x^T Ax$ has a unique optimal point.

Statement of Problem

The purpose of the paper is to investigate the efficiency of two matrix inverting algorithms restricted to positive definite matrices. The two algorithms will be implemented on two types of architecture. The two types of architecture being a Von Neumann machine and a hypothetical parallel system specifically designed for matrix inversion.

There are many problems to address in this thesis but all relate to the task of inverting positive definite matrices. The first problem is to design a parallel system that can be used to implement the two methods for matrix inversion. The parallel system we have designed for this paper is specifically designed for the Gauss-Jordan algorithm. Once the system has been designed the algorithms must be modified in order to implement them on the parallel system. In order to test the speed of each algorithm on a PC, a computer program is developed to generate matrices of the following sizes: 32 x 32, 64 x 64, 128 x 128, 256 x 256, and 512 x 512. Some Basic results of positive definite matrices were used to generate these matrices. The generated matrices are then inverted on a Von Neumann machine using both algorithms. Since the time needed to invert a matrix varies due to background work being done by the system, the two methods are tested multiple times (5) on each size in order to obtain an average speed. A comparison of the speed of the two algorithms is made on the Von Neumann machine. A comparison for speed of the two algorithms is then made on the parallel system. Finally, a comparison of each environment and for each method is made. The Gauss-Jordan algorithm is compared on the PC and parallel system for efficiency. Then the gradient projection method is compared on both environments for efficiency.

Overview

Chapter I provides an overview of the thesis which includes a statement of the problem and a rationale for studying the problem. Since the paper will be restricted to positive definite matrices, Chapter II provides some basic results concerning positive

definite matrices. In addition, Chapter II also provides two methods for determining the inverse of positive definite matrices. Chapter III looks at some general hardware specifications of the parallel system we designed specifically for matrix inversion. Also Chapter III also looks at the architecture of the PC we will use for matrix inversion. Chapter IV looks at the implementation of the Gauss-Jordan and gradient projection methods on the hypothetical parallel system. Chapter V provides the results of the efficiency study. Chapter V hopes to determine which algorithm is more efficient at finding the inverse of large matrices on a PC and a parallel system.

CHAPTER II

METHODS FOR MATRIX INVERSION

Since this paper is restricted to symmetric positive definite matrices, this chapter reviews some basic results concerning positive definite matrices. This chapter will also review the two methods that will be used for matrix inversion. A step by step process of each algorithm will be given followed by an example.

As we mentioned in the introduction positive definite matrices are a very important class of matrices with many applications. Our review of positive definite matrices will include results given in [2] and [3]. The results found in the review of positive definite matrices are used in this paper in order to generate large positive definite matrices and modify the two algorithms for matrix inversion.

Definition 2.1 A matrix M is said to be positive definite if and only if

$x^T Mx > 0$ for all x while $x \neq 0$.

Definition 2.2 A matrix M is said to be semi-positive definite if for

$x^T Mx \geq 0$ for all x while $x \neq 0$.

In order to generate positive definite matrices on the PC we used:

Theorem 2.1 If M is symmetric, diagonally dominant, and has positive diagonal elements, then M is positive definite.

A matrix is said to be diagonally dominant if for each column

$$|Aii| \geq \sum_{i \neq j} |Aij| . \tag{2.1}$$

Equation (2.1) indicates that the largest element in the column will always be the pivot when using the Gauss-Jordan algorithm. Therefore no row interchanges will be needed to minimize round-off error or avoid division by zero. Since this paper is restricted to symmetric positive definite matrices we will not have to be concerned with matrix inversion failing. Hageman and Young[6], indicate that if A is symmetric and positive definite, then A is nonsingular therefore an inverse exists. As was mentioned in the introduction the most important property of a positive definite matrix is the hypersurface $x^T Ax$ has a unique optimal point. This property assures us that the gradient projection method will always converge to the optimal point. The code for the matrix generator can be found in Appendix A, program A.1.

Gauss-Jordan Method of Inversion

Our review of the Gauss-Jordan method will include the results given in [5]. The Gauss-Jordan method is an elimination method. It consists of multiplying the various rows of a matrix by the appropriate constants and subtracting from the other rows of a matrix in order to obtain zero elements in some locations until eventually you are left with a diagonal matrix. In finding the inverse the following transformation (2.2) is applied again and again.

$$A' ij = Aij - Aik \frac{Akj}{Akk} \tag{2.2}$$

If the pivot row is divided by Akk then $\frac{Akj}{Akk}$ would be $A'\,kj$ with $A'\,kk = 1$ and (2.2) becomes

$$A'\,ij = Aij - Aik\,A'\,kj \tag{2.3}$$

Given A an nxn matrix create its augmented matrix [A | I] by appending the identity matrix.

$$\left[\begin{array}{cccc|cccc} a_{11} & a_{12} & a_{13} & \dots a_{1n} & 1 & 0 & 0 & \dots 0 \\ a_{21} & a_{22} & a_{23} & \dots a_{2n} & 0 & 1 & 0 & \dots 0 \\ a_{31} & a_{32} & a_{33} & \dots a_{3n} & 0 & 0 & 1 & \dots 0 \\ \vdots & & & & \vdots & & & \\ a_{n1} & a_{n2} & a_{n3} & \dots a_{nn} & 0 & 0 & 0 & \dots 1 \end{array}\right]$$

Usually the initial step would be to find the first nonzero in column one but since the matrix is diagonally dominant we know that every diagonal element will not be zero. Therefore the next step would be to divide the first equation by a_{11} to make the element in the a_{11} position become 1 and obtain the rest of the elements for row one.

$$\left[\begin{array}{cccc|cccc} 1 & \frac{a_{12}}{a_{11}} & \frac{a_{13}}{a_{11}} & \dots \frac{a_{1n}}{a_{11}} & \frac{1}{a_{11}} & \frac{0}{a_{11}} & \frac{0}{a_{11}} & \dots \frac{0}{a_{11}} \\ a_{21} & a_{22} & a_{23} & a_{2n} & 0 & 1 & 0 & \dots 0 \\ a_{31} & a_{32} & a_{33} & a_{3n} & 0 & 0 & 1 & \dots 0 \\ \vdots & & & & \vdots & & & \\ a_{n1} & a_{n2} & a_{n3} & a_{nn} & 0 & 0 & 0 & \dots 1 \end{array}\right]$$

Next we can eliminate each element in column one not in row one by multiplying the first row by a_{21} and subtracting from the second row and multiplying the first row by a_{31} and subtracting from the third row making everything below a_{11} zero. Giving

$$\left[\begin{array}{cccc|cccc} 1 & \frac{a_{12}}{a_{11}} & \frac{a_{13}}{a_{11}} & \dots \frac{a_{1n}}{a_{11}} & \frac{1}{a_{11}} & 0 & 0 & \dots 0 \\ 0 & a_{22} - a_{21}\left(\frac{a_{12}}{a_{11}}\right) & a_{23} - a_{21}\left(\frac{a_{13}}{a_{11}}\right) & \dots a_{2n} - a_{21}\left(\frac{a_{1n}}{a_{11}}\right) & 0 - a_{21}\left(\frac{1}{a_{11}}\right) & 1 - a_{21}(0) & 0 - a_{21}(0) & \dots 0 - a_{21}(0) \\ 0 & a_{32} - a_{31}\left(\frac{a_{12}}{a_{11}}\right) & a_{33} - a_{31}\left(\frac{a_{13}}{a_{11}}\right) & \dots a_{3n} - a_{31}\left(\frac{a_{1n}}{a_{11}}\right) & 0 - a_{31}\left(\frac{1}{a_{11}}\right) & 0 - a_{31}(0) & 1 - a_{31}(0) & \dots 0 - a_{31}(0) \\ \vdots & & & & \vdots & & & \\ 0 & a_{n2} - a_{n1}\left(\frac{a_{12}}{a_{11}}\right) & a_{n3} - a_{n1}\left(\frac{a_{13}}{a_{11}}\right) & \dots a_{nn} - a_{n1}\left(\frac{a_{1n}}{a_{11}}\right) & 0 - a_{n1}\left(\frac{1}{a_{11}}\right) & 0 - a_{n1}(0) & 0 - a_{n1}(0) & \dots 1 - a_{n1}(0) \end{array}\right]$$

Next we need to eliminate everything above and below the a_{22} pivot by the same process. This process is continued for all rows leaving us with the identity matrix on the left side and the inverse on the right as follows:

$$\left[\begin{array}{cccc|cccc} 1 & 0 & 0 & \dots 0 & a_{11} & a_{12} & a_{13} & \dots a_{1n} \\ 0 & 1 & 0 & \dots 0 & a_{21} & a_{22} & a_{23} & \dots a_{2n} \\ 0 & 0 & 1 & \dots 0 & a_{31} & a_{32} & a_{33} & \dots a_{3n} \\ \vdots & & & & \vdots & & & \\ 0 & 0 & 0 & \dots 1 & a_{n1} & a_{n2} & a_{n3} & \dots a_{nn} \end{array}\right]$$

Example of the Gauss-Jordan Method

Given the 3 x 3 matrix augmented with identity

$$\left[\begin{array}{ccc|ccc} 16 & 8 & 4 & 1 & 0 & 0 \\ 8 & 35 & 17.5 & 0 & 1 & 0 \\ 4 & 17.5 & 63 & 0 & 0 & 1 \end{array}\right]$$

Next divide row 1 by a_{11}

$$\left[\begin{array}{ccc|ccc} 1 & .5 & .25 & .0625 & 0 & 0 \\ 8 & 35 & 17.5 & 0 & 1 & 0 \\ 4 & 17.5 & 63 & 0 & 0 & 1 \end{array}\right]$$

Next eliminate the elements below a_{11} by multiplying the first row by a_{21} subtracting from the second row and do the same for row 3 by multiplying the first row by a_{31} and subtracting from the third row.

$$\left[\begin{array}{ccc|ccc} 1 & .5 & .25 & .0625 & 0 & 0 \\ 0 & 31 & 15.5 & -.05 & 1 & 0 \\ 0 & 0 & 62 & -.25 & 0 & 1 \end{array}\right]$$

Next divide row 2 by a_{22} and eliminate the elements above and below by multiplying the second row by a_{32} and subtracting from the third row. Then multiply the second row by a_{12} and subtract from the second row.

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & .0706 & -.0162 & 0 \\ 0 & 1 & .5 & -.0162 & .0323 & 0 \\ 0 & 0 & 54.25 & -.0005 & -.5007 & 1 \end{array}\right]$$

The last step is to divide row 3 by a_{33} and eliminate the elements above by multiplying the third row by a_{23} and subtracting from the second row. Then multiply the first row by a_{13} and subtract from the first row.

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & .0706 & -.0162 & 0 \\ 0 & 1 & 0 & -.0162 & .0323 & -.0092 \\ 0 & 0 & 1 & 0 & -.0092 & .0184 \end{array}\right]$$

Code for this algorithm can be found in appendix A program A.2.

Gradient Projection Method

Our review of the gradient projection method will include the results given in [1]. Throughout the rest of the paper gradient projection method will be referred to as GPM. The GPM is an iterative method used to solve linear systems. The method uses functional gradients as a choice of minimization direction. This method uses the fact that

the direction opposite to the direction of the functional gradient guarantees the most rapid decrease of the functional. Therefore this method is sometimes referred to as method of steepest descent. The gradient method constructs a line of steepest descent starting from an arbitrary point and running to the point giving a minimum for the functional. Therefore constructing a line whose direction at each point is opposite the direction of the gradient.

Given a linear system

$$Ax = F \tag{2.4}$$

The solution of the system is found by finding the vector that is a minimum for the functional

$$H(x) = (Ax, x) - 2(F, x) \tag{2.5}$$

First we select an arbitrary vector x_0. Next we calculate the direction opposite of the gradient of the functional $H(x)$ at this point by calculating the residual $r_0 = F - Ax_0$ for the initial arbitrary vector. From the point x_0 we move to the point x_1 at which the functional $H(x)$ becomes minimal.

Since

$$\begin{aligned} H(x_0 + \alpha r_0) &= (Ax_0 + \alpha Ar_0, x_0 + \alpha r_0) - 2(F, x_0 + \alpha r_0) \\ &= (Ax_0, x_0) + 2\alpha (Ax_0, r_0) + \alpha^2(Ar_0, r_0) - 2(F, x_0) - 2\alpha(F, r_0) \\ &= H(x_0) - 2\alpha(r_0, r_0) + \alpha^2(Ar_0, r_0) \\ &= H(x_0) - \frac{(r_0, r_0)^2}{(Ar_0, r_0)} + (Ar_0, r_0)\left[\alpha - \frac{(r_0, r_0)}{(Ar_0, r_0)}\right]^2 \end{aligned}$$

this expression obtains a minimum for

$$\alpha = \alpha_0 = \frac{(r_0, r_0)}{(Ar_0, r_0)} \tag{2.6}$$

this minimum is equal to

$$H(x_0) - \frac{(r_0, r_0)^2}{(Ar_0, r_0)} \tag{2.7}$$

Therefore

$$x_1 = x_0 + \alpha_0 r_0$$

In general compute $x_{k+1} = \alpha_k r_k$ where $r_k = F - Ax_k$ and $\alpha_k = \dfrac{(r_k, r_k)}{(Ar_k, r_k)}$

Terminate computation if $|x_{k+1} - x_k| < e$ where e is the user-selected error.

Theorem 2.2: The successive approximation of $x_0, x_1, x_2, \ldots$ converges to the solution of the system $Ax = F$ at a geometric rate.

Example of Gradient Projection Method

Solve the linear system:

$2x_1 + x_2 = 1$

$x_1 + 2x_2 = 0$

Exact solution for system: $(x_1, x_2) = (2/3., -1/3)$

Using gradient projection method:

Let the arbitrary vector $x_0 = (.5,.5)$

Next calculate $r_0 = F - Ax_0$

$$r_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} - \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} .5 \\ .5 \end{bmatrix} = \begin{bmatrix} -.5 \\ -1.5 \end{bmatrix}$$

Next calculate $\alpha_0 = \dfrac{(r_0, r_0)}{(Ar_0, r_0)}$

$$\alpha_0 = \frac{\begin{bmatrix} -.5 & -1.5 \end{bmatrix} \begin{bmatrix} -.5 \\ -1.5 \end{bmatrix}}{\begin{bmatrix} -.5 & -1.5 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} -.5 \\ -1.5 \end{bmatrix}} = .384615$$

Next calculate $x_1 = x_0 + \alpha_0 r_0$

$$x_1 = \begin{bmatrix} .5 \\ .5 \end{bmatrix} + .384615 \begin{bmatrix} -.5 \\ -1.5 \end{bmatrix} = \begin{bmatrix} .3076925 \\ -.0769225 \end{bmatrix}$$

Next calculate $r_1 = F - Ax_1$

$$r_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} - \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} .3076925 \\ -.0769225 \end{bmatrix} = \begin{bmatrix} .461537 \\ -.153848 \end{bmatrix}$$

Next calculate $\alpha_1 = \dfrac{(r_1, r_1)}{(Ar_1, r_1)}$

$$\alpha_1 = \frac{\begin{bmatrix} .461537 & -.153848 \end{bmatrix} \begin{bmatrix} .461537 \\ -.153848 \end{bmatrix}}{\begin{bmatrix} .461537 & -.153848 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} .461537 \\ -.153848 \end{bmatrix}} = .714291$$

Next calculate $x_2 = x_1 + \alpha_1 r_1$

$$x_2 = \begin{bmatrix} .3076925 \\ -.0769225 \end{bmatrix} + .714291 \begin{bmatrix} .461537 \\ -.153848 \end{bmatrix} = \begin{bmatrix} .637364 \\ -.186815 \end{bmatrix}$$

This process is continued until x converges to the solution. You can begin so see the desired result at x_7.

$$x_7 = \begin{bmatrix} .668003 \\ -.333309 \end{bmatrix} + .346154 \begin{bmatrix} -.002697 \\ -.001385 \end{bmatrix} = \begin{bmatrix} .667069 \\ -.333788 \end{bmatrix}$$

For the purpose of this paper computation is terminated if $|x_{k+1} - x_k| < .000001$

Code for this algorithm can be found in Appendix A program A.3.

This chapter has described two methods for matrix inversion on a PC. In Chapter IV we will modify the algorithms in order to implement them on the hypothetical parallel processor.

CHAPTER III

COMPUTER ARCHITECTURES

For the purpose of this paper we will use a simulated parallel system. In order to simulate a parallel system, a system had to be designed. In this chapter we will present a device which enhances the speed for finding the inverse of a matrix. This chapter will also discuss the code used to access the hypothetical parallel processor. The chapter will describe the general hardware specifications and how the code is used to implement the parallel processor. This chapter will also briefly describe the hardware of the PC used in inverting matrices for this paper. Throughout the rest of the paper HPP will be used to denote hypothetical parallel processor.

For the purpose of this paper we designed the HPP specifically for inverting matrices using the Gauss-Jordan method. Since the Gauss-Jordan method involves the adding of rows and the multiplication by a scalar, the HPP was designed with many multipliers and adders in parallel. In finding the inverse, the following transformation (3.1) is applied again and again.

$$A'\,ij = Aij - Aik\,Akj \,/\, Akk \qquad (3.1)$$

If the pivot row is divided by Akk then $Akj \,/\, Akk$ would be $A'kj$ with $A'kk = 1$ and (3.1) becomes

$$A'\,ij = Aij - Aik\,A'\,kj \qquad (3.2)$$

Because the system we have designed is limited to addition, equation (3.2) can be transformed into

$$A' ij = Aij + (-Aik)\, A' kj \qquad (3.3)$$

Based on (3.3) we can design a device with four, six, eight, or as many elements as needed that provide the work defined by the transformation given in (3.3). An element defined by (3.3) is:

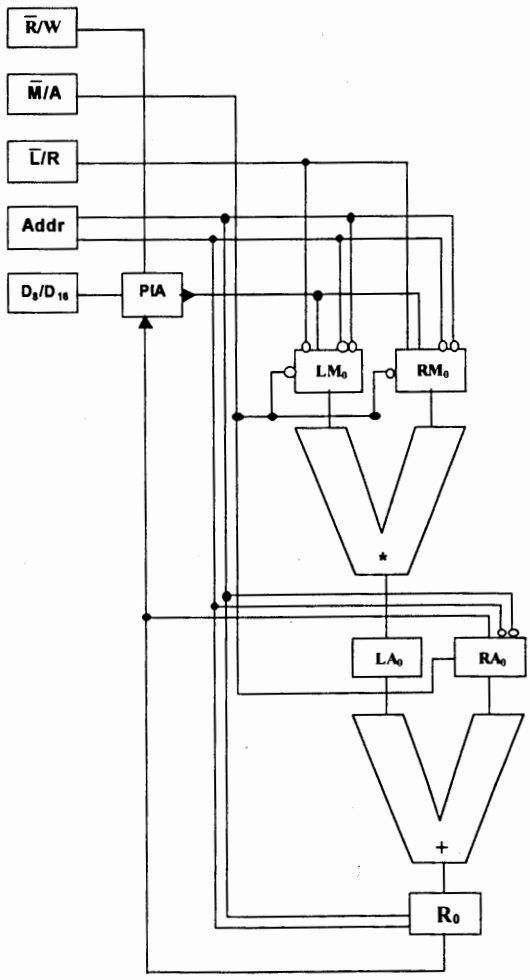

Figure 3.1 Single Element

Using N of these elements in parallel we can configure a device which will enhance the speed of matrix inversion using the Guass-Jordan method. In Figure 3.2 we have a 4-element configuration that we will use for the purpose of simulating the two methods for matrix inversion.

The main advantage of the HPP is in eliminating the elements above and below the pivot row as mentioned in Chapter II using (3.3). The 4 multipliers and 4 adders allow the HPP to work on four elements at a time, therefore making the HPP more efficient in inverting matrices as opposed to inverting matrices on a PC where only one element can be processed at a time. This HPP is restricted to processing four elements at a time but could easily be designed with more processors. As far as adding more processors to the unit, it would depend on cost and effective increase in speed accomplished by adding an element. So one must determine which design is most suited for a particular problem and whether the cost is worth the increase in speed.

Some of the general hardware specifications of the HPP are: a 1 bit multiplication and adder bus denoted as $\overline{M}/A$, a 1 bit left and right bus denoted as $\overline{L}/R$, a 1 bit read or write bus denoted as $\overline{R}/W$, a 2 bit address bus denoted as Addr, a peripheral interface adapter denoted as PIA, and a 16 bit data bus denoted as D_{16}. The $\overline{M}/A$ bus is used to determine whether the data is loaded into the multiplication unit or the adder unit. The $\overline{L}/R$ is used to determine which side of the multiplication unit the data is loaded. Since the adder unit can only be loaded on the right side there is no need for a $\overline{L}/R$ bus going

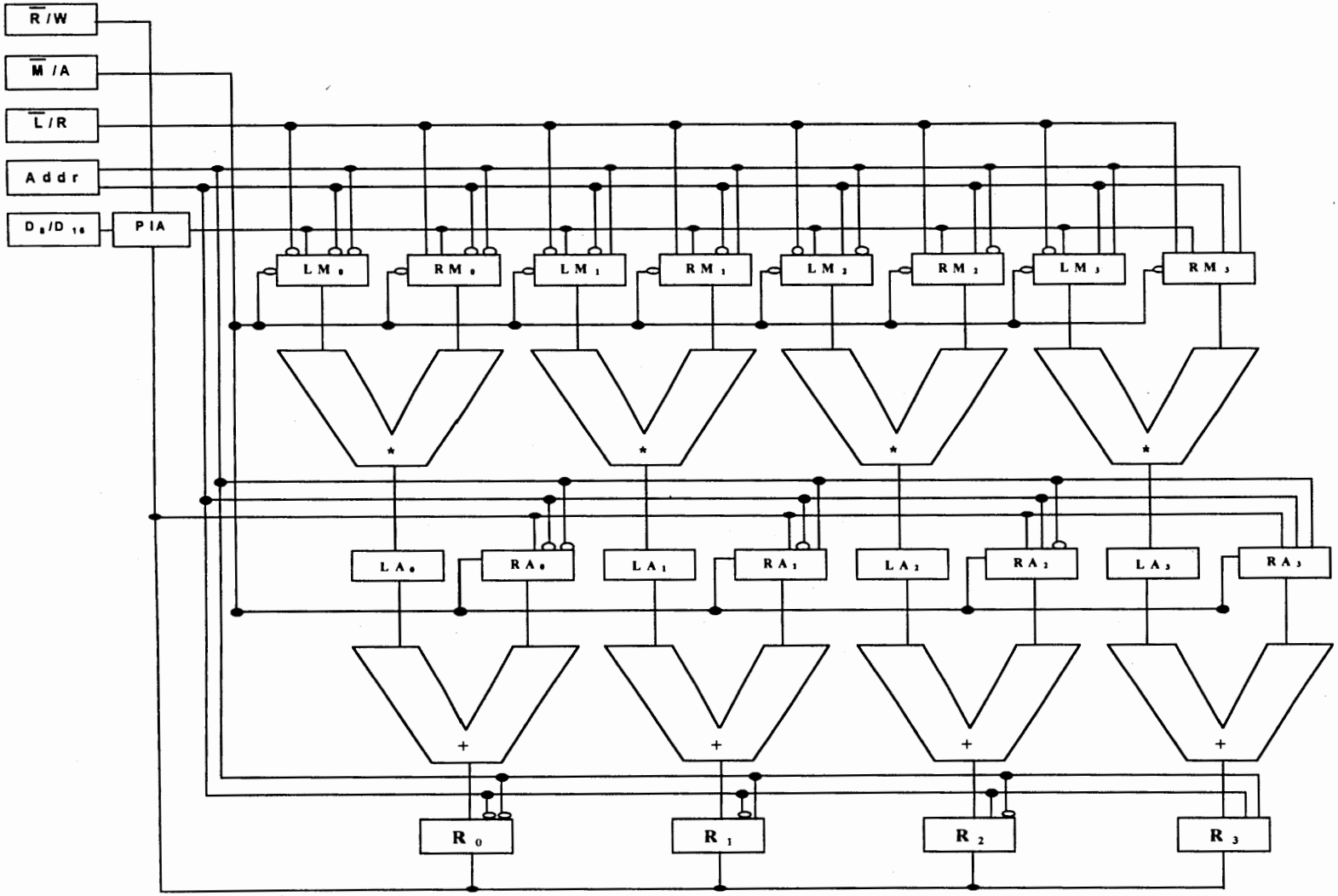

Figure 3.2 4-element

to the adder units. The $\overline{R}/W$ bus is used determine if the HPP is reading data or writing data. The Addr bus determines which processor the data is loaded into. The PIA determines if the HPP is reading or writing data to the data bus. The D_{16} reads in data 8 bits at a time and writes data 16 bits at a time.

With the implementation of the two methods for matrix inversion on the HPP, there is a need for code to access the HPP. There are four commands that will be used for accessing the HPP. The four commands are LOADMULT, LOADADD, READ, and EXECUTE. The LOADMULT command contains two parameters, one for loading vector from memory and one for selecting left or right side of multiplier unit. An example of the implementation of the code would be LOADMULT(vector,L). The LOADADD command contains one parameter for loading vector from memory. An example would be LOADADD(vector). The READ command contains one parameter and puts the result back into its memory location. An example would be READ(vector). The last command is the EXECUTE command which executes the HPP system. These commands will be used in the implementation of the algorithms for matrix inversion on the HPP.

Next we will go into detail of the process for loading a 4-element vector into the multiplier unit. When the LOADMULT(vector,L) command is executed several events occur. The first event is selecting either the multiplication unit or the adder unit. In this case the multiplication unit is selected. Therefore the $\overline{M}/A$ bus activates all 4 multiplier units. Next, the left or right side of the multiplier unit is selected. In this case the left

side is selected therefore opening access to the left side of the multiplier units $LM_0 \ldots LM_3$. If the command LOADMULT(vector,R) had been used then access to the right side of the multiplier units would have been made available. Since we are using a LOAD command the $\overline{R}/W$ bus will send signal to the PIA so the data bus will be read into the HPP. The next event is the retrieving of data from memory. The first parameter of the LOADMULT command is the address location of the 4-element vector. When the first element is retrieved it is sent through the D_{16} bus to the multiplier unit. The first element is assigned the address 00. This address is sent to the Addr bus therefore placing the first element in the left side of the first multiplier unit LM_0. This process is carried out for each element until all four elements have been loaded into the multiplier unit.

The process of loading the adder unit is very similar with some slight differences. When the LOADADD(vector) command is executed similar events will occur as in loading the multiplier unit. In this case the adder unit is selected. Therefore the $\overline{M}/A$ bus activates all 4-adder units. Unlike the multiplier unit the adder unit can only be loaded on the right side. Therefore there is no need for a $\overline{L}/R$ bus connection to the adder unit. The left side of the adder unit receives its data from the result of the multiplier unit. Once again since we are using a LOAD command the $\overline{R}/W$ bus will send signal to the PIA so the data bus will be read into the HPP. Then the first element is retrieved from memory and is sent through the D_{16} bus to the adder unit. Like the multiplier unit the first element is assigned the address 00. This address is sent to the Addr bus therefore

placing the first element in the right side of the adder unit RA_0. This process is carried out for each element until all four elements have been loaded into the adder unit.

Once both the multiplier unit and the adder unit have been loaded, the EXECUTE command is used to execute the HPP. Once the HPP has been executed the result is written back into a memory location by using the command READ(vector). When the READ command is executed the $\overline{R}/W$ bus sends signal to the PIA selecting the data bus to write data. Therefore the results are read from $R_0 \ldots R_3$ and written into the memory location for vector.

The HPP does require that you load both the multiplier unit and the adder unit in order for it to execute. But you may ask what if you only needed to multiply two vectors? This question can be answered by loading a zero vector into the adder unit. If you needed to add two vectors you would have to load a unity vector into one side of the multiplier unit and one vector into the other side. Then you would load the other vector into the adder unit then execute the HPP. So the HPP can be used for other calculations but it is most efficient at executing (3.3).

One might have noticed that there are four processors. But what if you were working with a matrix that was not a multiple of 4? This can be overcome by padding with zeros in order to fill the extra processors. So if you had a matrix of size $3x6$ you would pad an extra two columns of zeros to the end in order for it to implement of the HPP.

Since we used a PC to time the two methods of matrix inversion, we will briefly discuss the architecture of the PC. The typical PC has a single floating point multiplier, which requires 296 cycles and a floating-point adder, which requires about 14 cycles. The PC used for testing is an IBM Aptiva Series S with Intel MMX processor. The processor speed for the computer is 200mhz. The PC was used to compare the speed of the two algorithms for matrix inversion.

This chapter has described in detail the architecture of the HPP and the code for accessing the HPP. We have omitted the handshaking between the device and the computer. The two algorithms for matrix inversion from Chapter II will be modified in Chapter IV in order to implement them on the HPP. Keep in mind that the HPP was specifically designed for matrix inversion using the Gauss-Jordan method. Therefore making it more difficult to implement the gradient projection method.

CHAPTER IV

IMPLEMENTATION ON THE PARALLEL SYSTEM(HPP)

In Chapter II we discussed the two algorithms that we will use for matrix inversion. Chapter IV discusses the modifications that will be made to the two algorithms in order to implement them on the HPP. As we have mentioned earlier the HPP has been designed specifically for matrix inversion. Since the HPP was specifically designed for the Gauss-Jordan method of matrix inversion, this chapter begins with the modifications of the Gauss-Jordan method. This is followed by the modifications of the gradient projection method. Each algorithm is followed by a diagram of the implementation on HPP.

Gauss-Jordan Implementation on HPP

The method for Gauss-Jordan on the HPP is very similar to that of the Gauss-Jordan method used on the PC. Given A, an nxn matrix create its augmented matrix [A | I] by appending the identity matrix.

$$\left[\begin{array}{cccc|cccc} a_{11} & a_{12} & a_{13} & \dots a_{1n} & 1 & 0 & 0 & \dots 0 \\ a_{21} & a_{22} & a_{23} & \dots a_{2n} & 0 & 1 & 0 & \dots 0 \\ a_{31} & a_{32} & a_{33} & \dots a_{3n} & 0 & 0 & 1 & \dots 0 \\ \vdots & & & & \vdots & & & \\ a_{n1} & a_{n2} & a_{n3} & \dots a_{nn} & 0 & 0 & 0 & \dots 1 \end{array}\right]$$

In Chapter III the diagram of the architecture was given. One will notice that the HPP can only handle four elements at a time. The architecture does not limit the HPP to only handling matrices that are a multiple of four. If the matrix being processed is not a multiple of four the matrix can be padded with zeros at the end in order to make it work. An example would be a 3x3 matrix appended with the identity matrix therefore making it a 3x6.

$$\left[\begin{array}{ccc|ccccc} a_{11} & a_{12} & a_{13} & 1 & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 1 & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & 1 & 0 & 0 \end{array}\right]$$

In this example the matrix is padded with an extra 2 columns of zeros. Now we have eight rows which makes it a multiple of four. So the HPP can process any size of matrix by just padding the back of the matrix with columns of zeros.

In the Chapter II discussion of Gauss-Jordan it was mentioned that there is no need to check for nonzero elements in the pivot position or no need for pivoting, given the fact that the matrices are restricted to symmetric positive definite. The same holds true for the implementation on the HPP.

Step 1 of Gauss-Jordan

The first step in the process would be to divide the first row by a_{11} to make the pivot element in the a_{11} position become 1 and obtain the rest of the elements for row 1. Since the HPP only has a multiplier unit this step is carried out by multiplying $\frac{1}{a_{11}}$ by ever element in row 1. With the HPP this process is applied to four elements at a time.

In order to carry out the process on the HPP the command LOADMULT(vector-4,L) is used to load the first four elements of row 1. The next command that would be used is LOADMULT(NORMAL,R). This command loads the four element vector containing $\frac{1}{a_{11}}$. Next the command LOADADD(ZERO) is used to load a vector of zeros into the adders. Next the command EXECUTE is used in order to execute the HPP. After the HPP has been executed the READ command reads in data from the output bus of the adders and then stores them into row one of input matrix. The command LOADMULT(vector-4,L) is carried out $\frac{n}{4}$ times. Below is a diagram of the HPP with Step 1 applied.

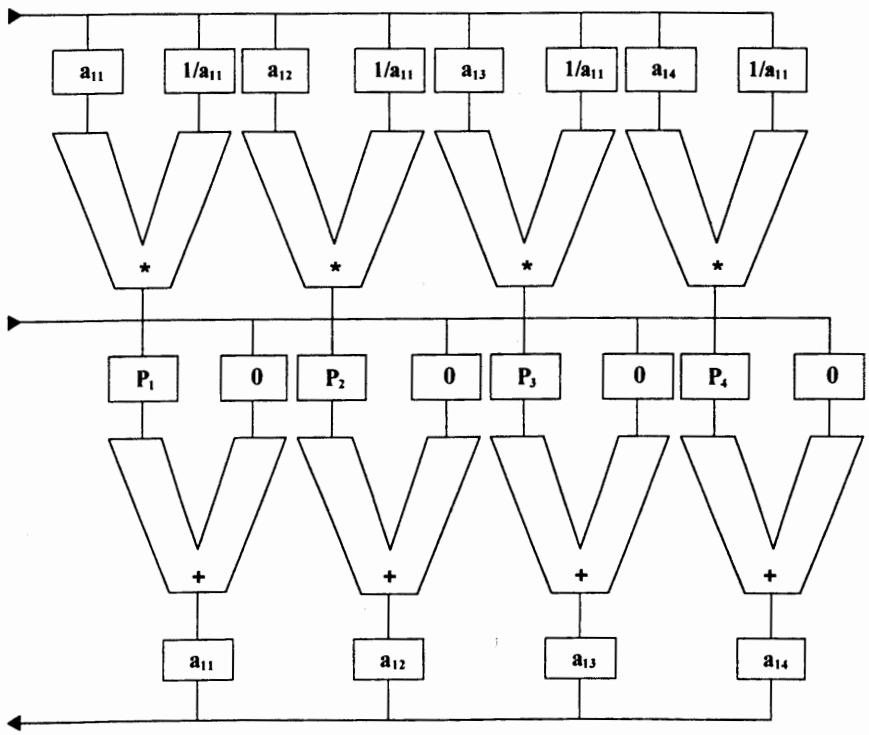

Figure 4.1 HPP Gauss-Jordan Step 1

Step 2 of Gauss-Jordan

Step 2 in the process is to eliminate each element in column one and not in row one. a_{21} can be eliminated by multiplying the first row by a_{21} and subtracting from the second row and a_{31} can be eliminated by multiplying the first row by a_{31} and subtracting from the third row. This process is continued for each row making everything below a_{11} zero. In order to implement this step on the HPP a CONSTANT vector is created consisting of elements equal to the negative of the first element of row two. First the CONSTANT vector is loaded into the multiplier unit (LOADMULT(CONSTANT, R)). Next the first four elements of the input matrix are loaded into the HPP multiplier unit (LOADMULT(vector-4, L)). Then the first four elements of row two are loaded into the adders (LOADADD(vector-4)). Next the HPP is executed and the data of the output bus are read into the row two. The LOADMULT(vector-4, L) and LOADADD(vector-4) commands are carried out $\frac{n}{4}$ times with n equal to the total number of columns. The LOADMULT(CONSTANT, R) command does not need to be executed since it remains the same for the row being processed. After processing row two, the process is continued for rows three and so forth.

Both Step 1 and step 2 are repeated for each row. With each repeated process the a_{ii} element becoming the pivot. So the next loop would use a_{22} as the pivot and eliminate all elements below and above. This process is carried out the same for a_{31} and so on for each row.

Step 2 applied

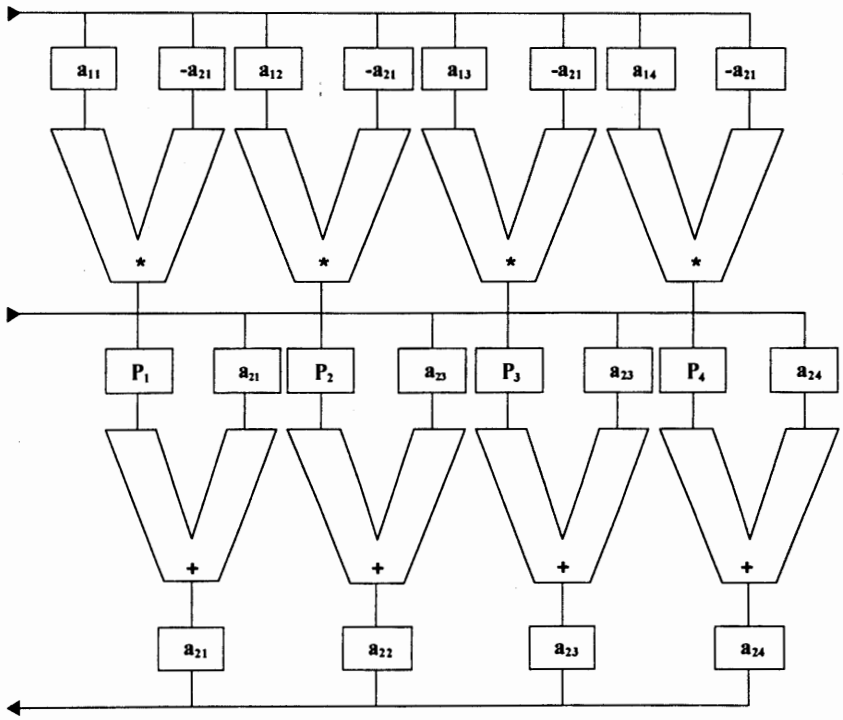

Figure 4.2 HPP Gauss-Jordan Step 2

Summarization of Steps for Gauss-Jordan

Loop i times. (i denotes number of rows)

LOADADD(ZERO)

LOADMULT(NORMAL,R)

Loop $\frac{n}{4}$ times (n denotes number of columns)

LOADMULT(row i vector ,L)

EXECUTE

READ(row i vector)

End Loop. (A 4x8 requires looping 2 times)

Loop k For rows $\neq i$:

LOADMULT(CONSTANT,R)

Loop $\frac{n}{4}$ times (n denotes number of columns):

LOADMULT(row i vector ,L)

LOADADD(row k vector)

EXECUTE

READ(row k vector)

End Loop.

End Loop k

End Loop i

Gradient Projection Method

Since the HPP was specifically designed for matrix inversion using the Gauss-Jordan method, the GPM was more difficult to implement on the HPP. The HPP would be well suited for any method that uses row reduction to find the inverses of matrices, such as Gaussian elimination. Since the GPM is not a row reduction scheme the GPM was not well suited for the HPP. So in fact it would be wise to develop another HPP that would be more efficient in implementing the GPM. For the purpose of this paper we will restrict the implementation to one HPP.

With the implementation of the GPM the previous commands for the HPP used in Gauss-Jordan will also be used for the GPM. Once again the matrix or vector does not

have to be a multiple of four. They can both be padded with zeros in order to implement them on the HPP.

The GPM is used to solve the linear system $Ax = F$. In order to use the GPM for matrix inversion we solve the linear system $Ax = F$ where F is the jth column of the identity matrix, A is an *nxn* matrix, and x is the solution vector. So when using the GPM for matrix inversion we will solve for each column of the identity. Therefore having a solution vector for each column of the identity. At the end the solution vectors will make up the matrix inverse. As for the initial arbitrary vector x_0, for the purpose of this paper it will also be the jth column of the identity matrix. We must also create a ZERO vector whose elements will be made up of zeros. The ZERO vector is used to load into adder unit when adding is not need in the calculation. Lastly we create a 4-element vector called VECTOR4 used to hold data read in from output bus.

Step 1 of GPM

In step 1 we use the formula $r = F - Ax$ to calculate the residual. First we load the zero vector into the adder. The command would be LOADADD(ZERO). Next we load row 1 of matrix A and the vector x four elements at a time into the multiplier unit. The command would be LOADMULT(row 1, L) and LOADMULT(x, R). The HPP is then executed by EXECUTE command. The data from the adder output bus is read into a 4-element vector called VECTOR4. Then the sum of the VECTOR4 is calculated and stored into the variable sum. This process is carried out $\frac{n}{4}$ times where n is the number of elements in the vector x or number of rows in matrix A. Once row one has been

summed then the first element of the r vector is calculated by $r[1] = F[1] - \text{sum}$. This process is continued for each row of matrix A. Step 1 applied to HPP.

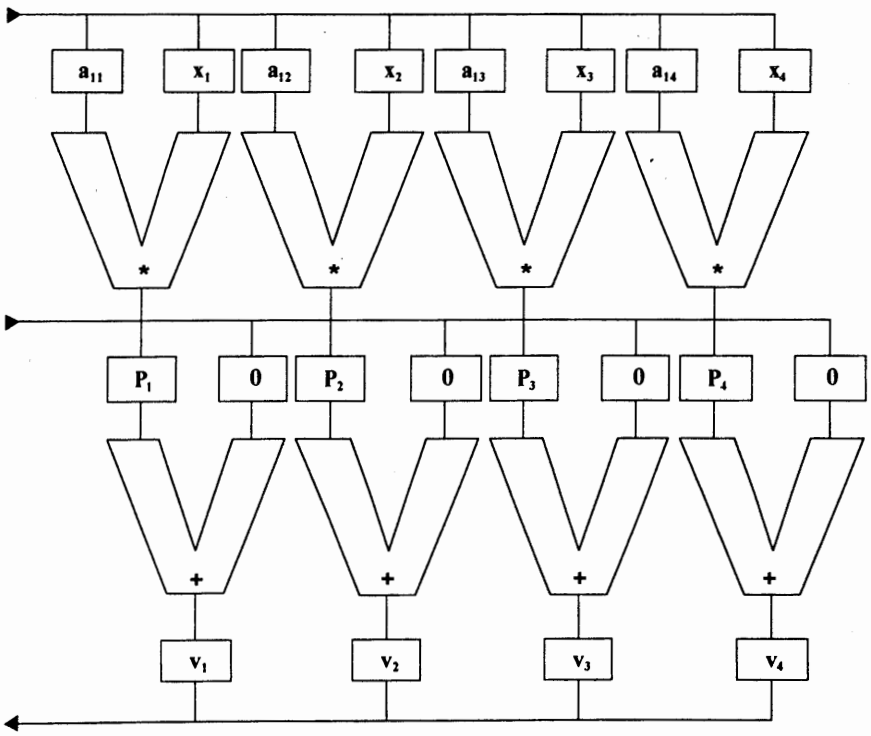

Figure 4.3 HPP Gradient Projection Step 1

Step 2 of GPM

In step 2 we calculate $\alpha = \frac{(r, r)}{(Ar, r)}$. To generate the factor α we first calculate $R = (r, r)$ by loading r into the multiplier unit. The command would be as follows: LOADMULT(r,L) and LOADMULT(r,R). Next the HPP would be executed. The results would be read from the adder output bus into the vector VECTOR4. The elements of VECTOR4 would then be summed and stored into R. This process is carried out $\frac{n}{4}$ times. Next we calculate $T = Ar$ by loading vector r and row 1 of matrix A into the

multiplier unit. The command would be as follows: LOADMULT(r ,L) and LOADMULT(row 1, R). The HPP is executed. The results are read into VECTOR4. VECTOR4 is summed and the value is stored in the vector T. This process is carried out $\frac{n}{4}$ times. The whole process is carried out for each row of A. Next we calculate $S = (T, r)$ by loading vector T and r into the multiplier unit. The command would be as follows: LOADMULT(T,L) and LOADMULT(r ,R). The HPP is executed. The results are read into VECTOR4 and summed. The result is stored in S. This process is also carried out $\frac{n}{4}$ times. α is then calculated by $\frac{R}{S}$. Step 2 applied to HPP.

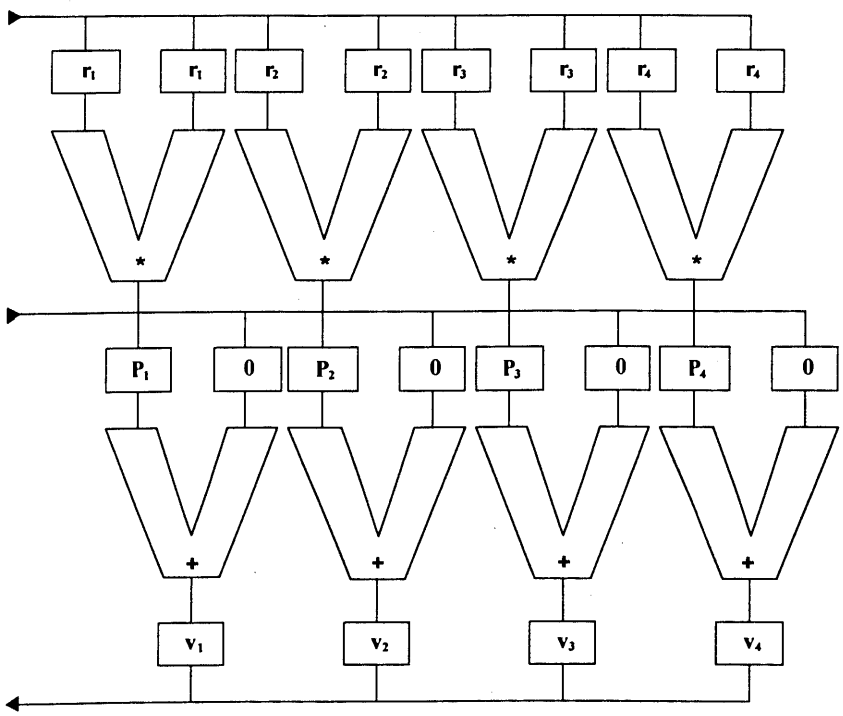

Figure 4.4 HPP Gradient Projection Step 2

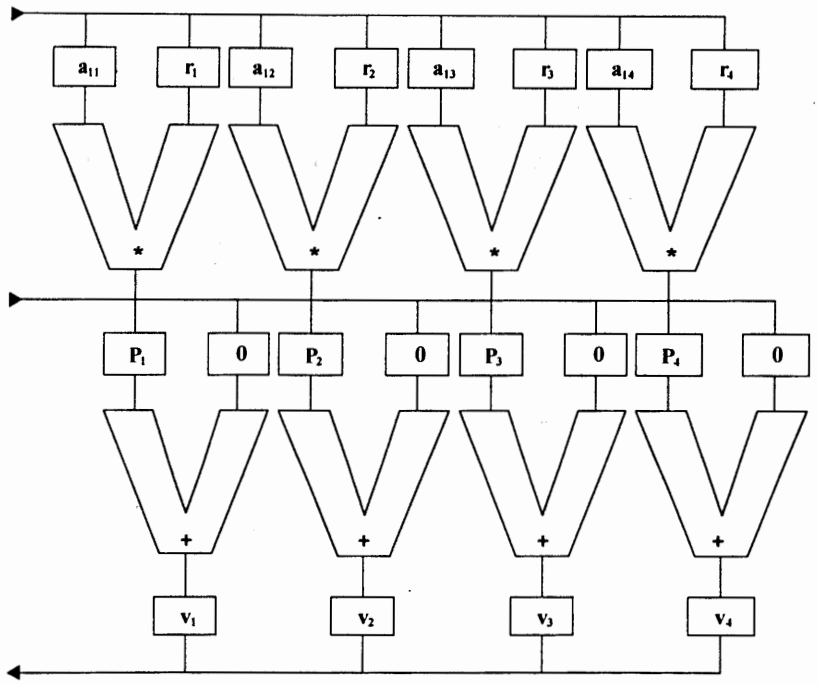

Figure 4.5 HPP Gradient Projection Step 2

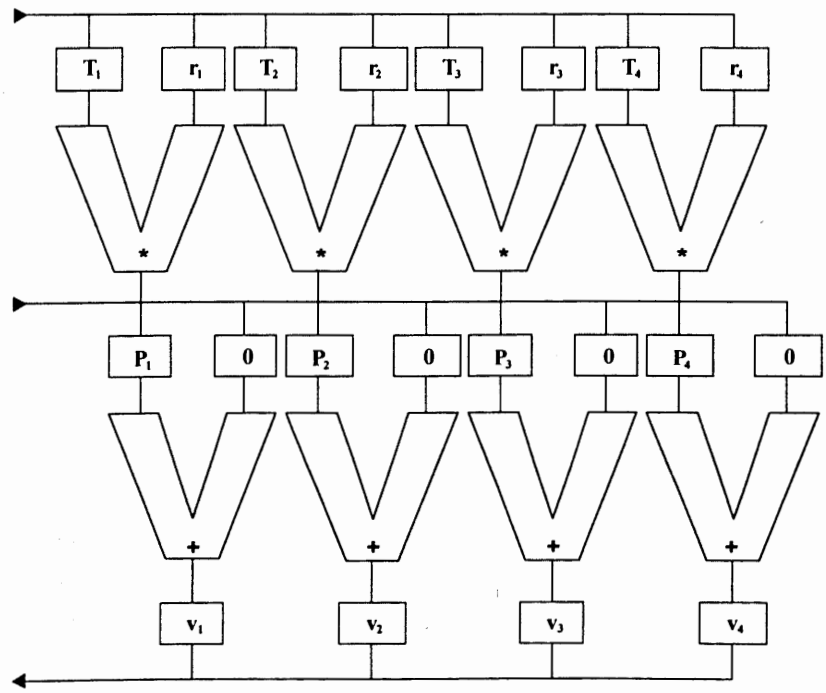

Figure 4.6 HPP Gradient Projection Step 2

Step 3 of GPM

The α is loaded into VECTOR4. The value for x is stored in previous x. Next $x = x + \alpha r$ is calculated by loading α and r into the multiplier unit. Then x is loaded into the adder. The commands are as follows: LOADMULT(α, L), LOADMULT(r,R), and LOADADD(x). This process is carried out $\frac{n}{4}$ times. Step 3 applied to HPP.

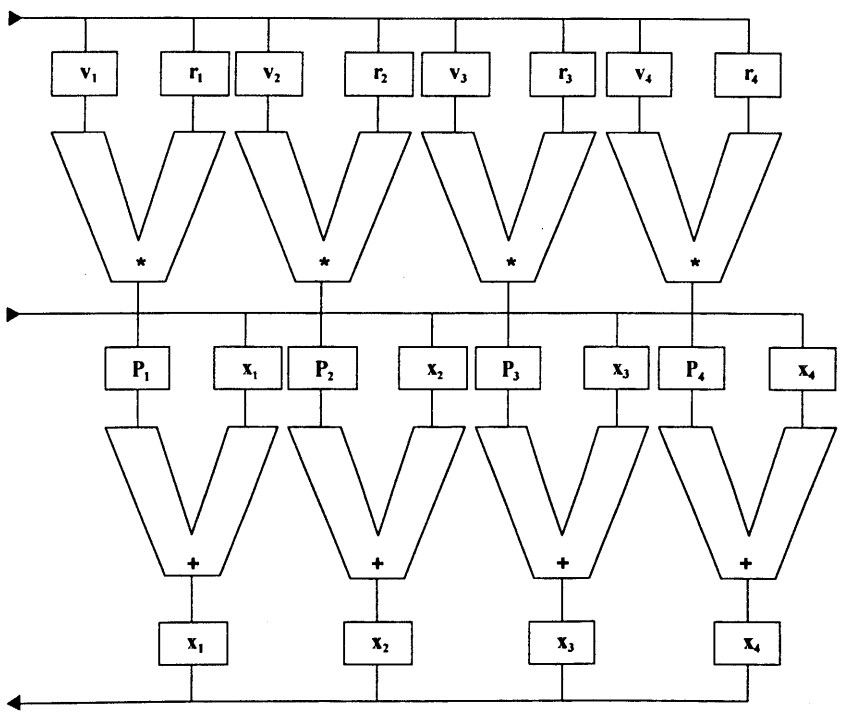

Figure 4.7 HPP Gradient Projection Step 3

Steps 1-3 are carried out until $|x - previous x| < e$ with e being the user-selected error. After designated error has been reached column 1 of the inverse of matrix A is the result of the solution x. Steps 1-3 are then carried out for each column of the identity matrix. Solving the linear system $Ax = F$ for each column of the identity matrix.

Summarization of Steps for GPM

Loop k times. (k denotes number of columns in identity)

Loop until $|x - previousx| < e$ where $e = .000001$

Loop i times(i denotes number of rows in matrix A)

LOADADD(ZERO)

Loop $\frac{n}{4}$ times(n denotes number columns)

LOADMULT(row i vector , L)

LOADMULT(x, R)

EXECUTE

READ(VECTOR4)

For (t = 0; t < 4; t++)

sum += VECTOR4[t];

End Loop

$R[i] = F[i] - sum$

End Loop i

LOADADD(ZERO)

Loop $\frac{n}{4}$ times(n denotes number columns)

LOADMULT(r, L)

LOADMULT(r, R)

EXECUTE

READ(VECTOR4)

For(t = 0; t < 4; t++)

R = VECTOR4[t];

End Loop

Loop *i* times(*i* denotes number of rows in matrix *A*)

LOADADD(ZERO)

Loop $\frac{n}{4}$ times(n denotes number columns)

LOADMULT(row *i* vector, L)

LOADMULT(*r*, R)

EXECUTE

READ(VECTOR4)

For (t = 0; t < 4; t++)

T [*i*] += VECTOR4[t];

End Loop

End Loop *i*

Loop $\frac{n}{4}$ times(n denotes number columns)

LOAD(*r*, R)

LOAD(*T*, L)

EXECUTE

READ(VECTOR4)

for(t = 0; t < 4; t++)

S = VECTOR4[t];

End Loop

$\alpha = R / S$;

for(t = 0; t < 4; t++)

VECTOR4[t] = α;

Loop $\frac{n}{4}$ times(n denotes number columns)

LOADMULT(α,L)

LOADMULT(r, R)

previous$x$ = x

LOADADD(x)

READ(x)

End Loop.

End Loop (when $|x - \text{previous}x| < e$)

End Loop k

This chapter has explained the needed modifications in order to implement the two methods for matrix inversion on the HPP. From the length of the GPM algorithm it can be seen that the GPM is not well suited for the HPP. On the other hand the Gauss-Jordan method is well suited for implementation on the HPP.

CHAPTER V

CONCLUSIONS

As we mentioned in the introduction, the efficiency of an algorithm is important when choosing a method to solve a specific problem. Since this paper deals with inverting positive definite matrices, we hope to conclude which of the two methods is more efficient and what system is best suited for the problem and algorithm.

In testing the two methods for efficiency on a PC, we applied each algorithm to different sizes of matrices. The sizes tested are 32 x 32, 64 x 64, 128 x 128, 256 x 256, and 512 x 512. Each matrix was tested multiple times for speed and an average was calculated. The reason for testing multiple times had to do with the fact that the clocked times varied due to background work being done by the computer. We also tested different matrices of the same size. The reason for this is that the Gradient Projection Method converged on some matrices at a faster rate than other matrices. This had to do with the initial guess being closer to the solution. As far as the Gauss-Jordan method there was no need to test on different matrices of the same size because the Gauss-Jordan method is an exact method, therefore the same number of steps executed is the same for any matrix of the same size. Although there is some difference in time due to the background work being done by the computer. In Figure 5.1 the results of our efficiency test on a PC are given. The time is measured in seconds.

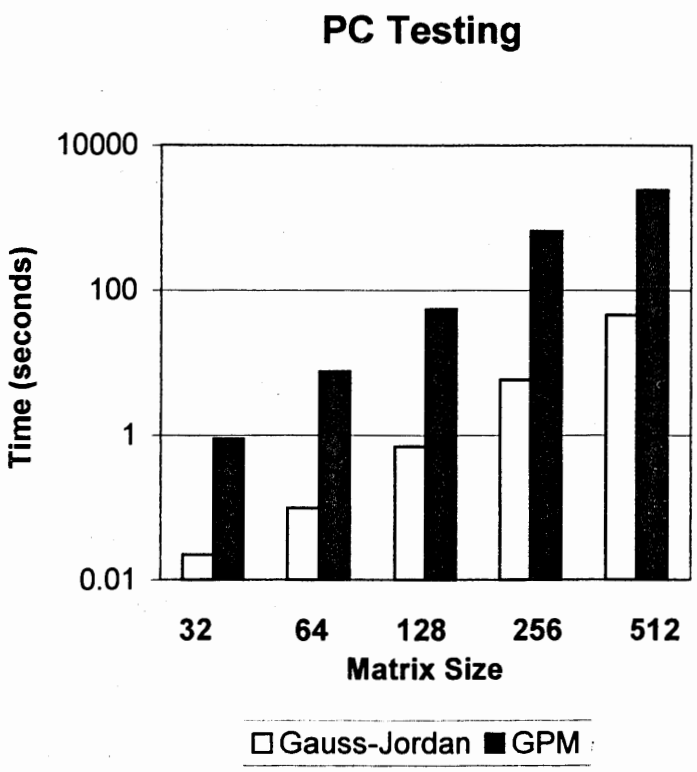

Figure 5.1 PC Testing

The comparison of the two methods on a PC found that the Gauss-Jordan out performed the Gradient Projection Method for each size of matrix. The GPM had a selected error of .000001.

In comparing the two methods, one must remember that the Gauss-Jordan method is an exact method and the Gradient Projection method is an iterative method. So you can not make a real comparison of the two. But from the results it can be seen that the Gauss-Jordan method outperforms the gradient projection method in every case. So one could draw the conclusion that the Gauss-Jordan method is more efficient at inverting

matrices. The gradient projection method could be made more efficient by increasing the error from .000001 to .00001. We tested five of the matrices at .00001 and found that even though there was an increase in speed, the Gauss-Jordan still outperformed the gradient projection method. So you say why not increase the error even more? Well by increasing the error you are loosing accuracy. Therefore one must determine what kind of accuracy is best suited for the problem.

Is it necessary to test the algorithms on a machine? The answer is yes and no. The answer is yes if we are concerned with overhead. As we mentioned earlier when testing the algorithms on the PC the time varied slightly from run to run due to the background work being done by the system. These variations can vary with the demands of the system. The answer on the other hand would be no if we are not concerned with overhead. We can simply determine the number of operations for performing the algorithm. By counting the number of multiplication's and additions for a matrix of size n when using the Gauss-Jordan method we have:

Size	Multiplication	Additions
2	14	7
3	45	30
4	104	78
⋮	⋮	⋮
n	$\frac{3n^3 + n^2}{2}$	$\frac{3n^3 - 2n^2 - n}{2}$

Figure 5.2 Gauss-Jordan Work

If a multiplication requires 102 cycles and an addition requires 11 cycles then the total number of cycles required to find the inverse of an nxn matrix is given by (5.1)

$$Total\ Arithmetic\ Cycles = 102 * \frac{3n^3 + n^2}{2} + 11 * \frac{3n^3 - 2n^2 - n}{2} \quad (5.1)$$

Equation (5.1) gives the total cycles for arithmetic operations but there is also a need to calculate the time to move the data during these operations. If a move requires 1 cycle and there are three moves in addition and multiplication then the total number of move cycles required to find the inverse of an nxn matrix is given by (5.2)

$$Total\ Move\ Cycles = 3 * \frac{3n^3 + n^2}{2} + 3 * \frac{3n^3 - 2n^2 - n}{2} \quad (5.2)$$

The total cycles to calculate the inverse of an nxn matrix is given by equation (5.3)

$$Total\ Cycles = Total\ Arithmetic\ Cycles + Total\ Move\ Cycles \quad (5.3)$$

It is not possible to do the same for the gradient projection method since it is an iterative method. One could determine the number of operations for one iteration but you could not determine how many iterations would be needed to reach a solution. In Figure 5.3 the multiplications and additions are counted for the GPM. When counting we are assuming that one iteration is needed to find the solution of each column of the identity.

Size	Multiplication	Additions
2	30	32
3	84	90
4	180	192
⋮	⋮	⋮
n	$2n^3 + 3n^2 + n$	$2n^3 + 4n^2$

Figure 5.3 Gradient Projection Work for 1 Iteration

We next simulated the two methods for matrix inversion on the HPP. In simulating the Gauss-Jordan on the HPP we used equation (5.1) to obtain (5.4)

$$Total\ Arithmetic\ Cycles\ HPP = \frac{102 * \frac{3n^3 + n^2}{2} + 11 * \frac{3n^3 - 2n^2 - n}{2}}{4} \qquad (5.4)$$

By dividing equation (5.1) by 4 we obtain the total arithmetic cycles for the HPP. We divide by 4 because there are 4 elements. Equation (5.1) is divided by the number of elements that the HPP contains. The total move cycles will remain the same as in equation (5.2). The total cycles to compute the inverse of an nxn matrix on the HPP is given by equation (5.5)

$$Total\ Cycles\ HPP = Total\ Arithmetic\ Cycles\ HPP + Total\ Move\ Cycles \qquad (5.5)$$

By using (5.1), the total arithmetic cycles for a 32x32 on a PC would be 5594960 cycles. By using (5.2), the total move cycles for a 32x32 on a PC would be 293328 cycles. The total cycles on a PC would be 5888288 by applying (5.3). Applying (5.4), the total arithmetic cycles for a 32x32 on the HPP would be 1398740 cycles. The total move cycles would remain the same. Therefore the total cycles for the HPP would be 1692068 by applying (5.5). With the HPP you see a 71.2% decrease in time. As you can see the HPP would be well suited for inverting matrices. One must determine if this increase in efficiency is worth the cost of a HPP. Figure 5.4 list some results found in comparing the Gauss-Jordan on the HPP and PC.

In simulating the gradient projection on the HPP, once again we run into the problem of simulating due to the fact that the GPM is iterative. But

one could still see that the GPM would be more efficient than the GPM on the PC. Since the HPP can

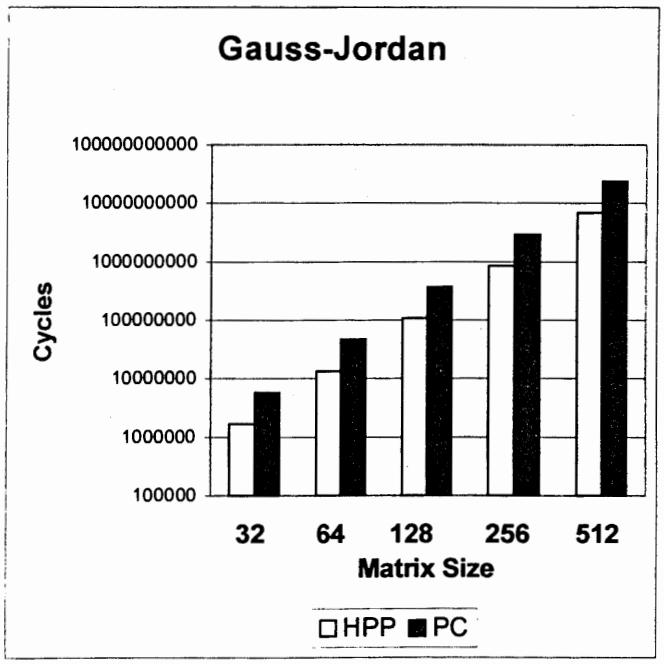

Figure 5.4 Gauss-Jordan Comparison

perform 4 operations at a time, without a corresponding increase in data movement, there will always be an increase in speed. When comparing the two algorithms on the HPP one can already see that the Gauss-Jordan will outperform the GPM due to the fact that the HPP was specifically designed for Gauss-Jordan. In a sense the HPP is job specific. One could design a HPP specifically for the GPM but that is beyond the scope of this paper.

In conclusion we have determined that the two methods for matrix inversion are more efficient when implemented on the HPP. In comparing the two methods on the PC we found that the Gauss-Jordan outperforms the GPM for any matrix size tested. When comparing the two methods on the HPP we found that the Gauss-Jordan method would

outperform the GPM due to the fact that the HPP was specifically designed for the Gauss-Jordan method. The main advantage of the HPP is to calculate (5.6)

$$A' ij = Aij + (-Aik)\, A' kj \tag{5.6}$$

which does not implement well with the GPM. In the results above one must remember that you really can not compare the two methods since one is exact and the other is iterative.

REFERENCES

[1] Faddeev, D.K., Faddeeva, V.N., Computational Methods of Linear Algebra, W.H. Freeman and Company, San Francisco, Ca., 1963

[2] Shampire, L.F., Allen, R.C., Pruess, S., Fundamentals of Numerical Computing, John Wiley & Sons, Inc., New York, 1997

[3] Stewart, G.W., Introduction to Matrix Computations, Academic Press, Inc., New York, 1973

[4] Widrow, B., Stearns, S., Adaptive Signal Processing, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1985

[5] Pennington, Ralph H., Computer Methods and Numerical Analysis, The Macmillian Company, London, 1970

[6] Hageman, Louis A., Young, David M., Applied Iterative Methods, Academic Press, NewYork, 1981

APPENDIX A

List of Programs

COMPUTER PROGRAM A.1

```
//Tege Radcliffe
//Fall 1999
//Matrix Generator
//Program Generates Symmetric Positive Definite Matrices
//Then Saves To Output File

#include<iostream.h>
#include<math.h>
#include<fstream.h>
#include<stdlib.h>
#include<string.h>
#include<iomanip.h>
#include<time.h>

//Maximum Size of Matrix
int const MAX_ROWS = 2048;
int const MAX_COLUMNS = 2048;

//Declare Global Variables
int size = 0;
ofstream outFile;
double MatrixA[MAX_ROWS][MAX_COLUMNS];
int numberLength = 0;

typedef char string[20];

//Functions
void OpenOutFile();
void GenerateMatrix();
void OutputMatrix();

int main()
{
        //Ask user for size of Matrix
        cout << "What size of square Matrix would you like to create?" << endl;
        cin  >> size;
```

```
        //Open output File
        OpenOutFile();

        //Generate Matrix
        GenerateMatrix();

        //Ouput Matrix to Screen and File
        OutputMatrix();

        return 0;
}

/*************************************************************************/
/*************************************************************************/

void GenerateMatrix()
{
        int row,column,k;
        double x;
        double offset = 0;
        double total = 0;

        //Initialize random number generator
        srand( (unsigned)time( NULL ) );

        //Generate Matrix
        for(row = 0; row < size; row++)
        {
                for(column = row; column <size; column++)
                {
                        //Generate random number for the pivot position
                        if(row = = column)
                        {
                                //offset + 1 is used to ensure next pivot position is greater
                                //than previous
                                x = rand()%30 + offset + 1;
                                MatrixA[row][column] = x ;
                                offset = MatrixA[row][column];
                        }
                        else
                        {
```

```
                        //Generate values for the next position and its symmetric
                        //postion
                        do
                        {
                                total = 0;
                                x = x/2;
                                MatrixA[row][column] = x;
                                MatrixA[column][row] = x;

                                //Total up every element in row besides pivot
                                for(k = 0; k <= column; k++)
                                {
                                        if(row != k)
                                                total = MatrixA[row][k] + total;
                                }

                        //Test for diagonally dominant
                        }while (MatrixA[row][row]  < total + 1 );

                }
        }
}
}

/*************************************************************************/
/*************************************************************************/

void OpenOutFile()
{
        string dataOut;

        //Set for output
        outFile.setf(ios::fixed, ios::floatfield);
        outFile.setf(ios::showpoint);
        cout.setf(ios::fixed, ios::floatfield);
        cout.setf(ios::showpoint);

        cout << "Enter name of output File" << endl;
        cin  >> dataOut;

        cin.ignore(100,'\n');
```

```
        outFile.open(dataOut,ios::out);

        if(!outFile)
        {
                cout << "Can not open File" << endl;
                exit( 1);
        }
}

/*************************************************************************/
/*************************************************************************/

void OutputMatrix()
{       int row,column;
        int count = 0;
        double value ;

        //Used to find length of biggest integer part in Matrix
        // For Formating in output
        numberLength = 0;
        value = 1;
        while(MatrixA[size-1][size-1] > value )
        {
                value = value * 10;
                numberLength++;
        }

        //Output Matrix Size
        outFile << size << endl;

        //Output Matrix
        for(row = 0; row < size; row++)
        {
                for(column = 0; column < size; column++)
                {
                        //output to screen and file
                        cout << setprecision(5) << setw(9 + numberLength )
                             << MatrixA[row][column] ;

                        outFile << setprecision(5) << setw(9 + numberLength
                                << MatrixA[row][column] ;
                        count++;
```

```
                    //To output four columns
                    if(count = = 4)
                    {
                            cout << endl;
                            outFile << endl;
                            count = 0;
                    }
            }
            cout << endl;
            outFile << endl ;
        }

}

/*************************************************************************/
/*************************************************************************/
```

COMPUTER PROGRAM A.2

```
//Tege Radcliffe
//Fall 1999
//Gauss-Jordan Method
//Program Uses Gauss Jordan Method to Find Inverse Of Matrix
//Then Outputs Inverse and clocked times to screen and file

#include<iostream.h>
#include<iomanip.h>
#include<fstream.h>
#include<time.h>
#include<ctype.h>
#include<string.h>
#include <stdlib.h>

typedef char string[60];

//Declare Global variables
int size,numberRows, numberColumns;;
ofstream outFile;
ifstream inFile;

//Marix Size
const MAX_ROWS = 2048;
const MAX_COLUMNS = 2048 ;

//Define Functions
void WriteToFile(double[MAX_ROWS][MAX_COLUMNS],ofstream&,int,int);
void WriteToScreen(double Matrix [MAX_ROWS][MAX_COLUMNS],int,int);
void WriteHeader(int,ofstream&);
void ReadInMatrix(double Matrix[MAX_ROWS][MAX_COLUMNS]);
void OpenOutFile();
void RowReduction(double Matrix[MAX_ROWS][MAX_COLUMNS]);
void OutputTime(clock_t, clock_t);
```

```
int main()
	{
		//Declare Matrix and clock
		double NewMatrix[MAX_ROWS][MAX_COLUMNS];
		clock_t start,end;

		//Read In Matrix From File
		ReadInMatrix(NewMatrix);

		//Open OutputFile
		OpenOutFile();

		//Write header to time file
		WriteHeader(size,outFile);

		//Call Clock and Output Start Time
		start = clock();

		//Use Row Reduction to Find Inverse
		RowReduction(NewMatrix);

		//Call Clock for End Time and Calculate total Time
		end = clock();

		//Output time results to file
		OutputTime(start,end);

		//Close File
		outFile.close();

		//Open output file for Inverse
		OpenOutFile();

		//Write Inverse to screen
		WriteToScreen(NewMatrix,numberRows,numberColumns);

		//Write Inverse to File
		WriteToFile(NewMatrix,outFile,numberRows,numberColumns);


	return 0;
	}
```

```
/*************************************************************************/
/*************************************************************************/

void WriteToScreen(double Matrix [MAX_ROWS][MAX_COLUMNS],int
                        numberRows,int numberColumns)
{
        cout.setf(ios::fixed, ios::floatfield);
        cout.setf(ios::showpoint);
        int row,column;

        for(row = 0; row < numberRows; row++)
        {
                for(column = numberColumns/2 ; column < numberColumns; column++)
                {
                        if(column % 4 = = 0)
                                cout << endl;

                        cout << setprecision(5) << setw(10 ) << Matrix[row][column] ;
                }

                cout << endl;
        }
}

/*************************************************************************/
/*************************************************************************/

void WriteToFile(double Matrix [MAX_ROWS][MAX_COLUMNS],ofstream& outFile,
                        int numberRows,int numberColumns)
{
        outFile.setf(ios::fixed, ios::floatfield);
        outFile.setf(ios::showpoint);
        outFile << setfill(' ');
        int row,column;

        for(row = 0; row < numberRows; row++)
        {
                for(column = numberColumns/2; column < numberColumns; column++)
                {
                        if(column % 4 = = 0)
                                outFile << endl;
```

```
                    outFile << setprecision(5) << setw(10 )
                            << Matrix[row][column] ;
                }

                outFile << endl;
        }
}

/*************************************************************************/
/*************************************************************************/

void WriteHeader(int size,ofstream& outFile)
{
        int count = 10;
        int width = 0;

        while(size > count)
        {
                count = count * 10;
                width++;
        }

        width = width * 2;

        outFile << setfill('*') << setw(45+width)  << endl;
        outFile << setfill('*') << setw(45 + width) << endl;
        outFile << "**"       << setfill(' ') << setw(42+width)  << "**" << endl;
        outFile << "**    TIME RESULTS FOR A " << size << "X" << size
                << " MATRIX   " << setw(3) << "**" << endl;
        outFile << "**       ROW REDUCTION METHOD " << setfill(' ')
                << setw(11+width) << "**" << endl;
        outFile << "**" << setfill(' ') << setw(42 + width) << "**" << endl;
        outFile << setfill('*') << setw(45 + width) << endl;
        outFile << setfill('*') << setw(45 + width) << endl;
        outFile << endl;
}

/*************************************************************************/
/*************************************************************************/

void ReadInMatrix(double Matrix [MAX_ROWS][MAX_COLUMNS])
{
```

```
int i,j;
double number;
string dataFile;

cout << "Enter name of data file" << endl;
cin  >> dataFile;

inFile.open(dataFile,ios::nocreate);

if(!inFile)
{
        cout << "Can not open File" << endl;
        exit( 1) ;
}

inFile >> size;

//Read in and Augment Matrix with Identity Matrix
for(i = 0; i < size; i++)
{
        for(j = 0; j < size*2; j++)
        {
                if(j >= size )
                {
                        if(i = = j  - size)
                        {
                                Matrix[i][j] = 1;
                        }
                        else
                                Matrix[i][j] = 0;
                }
                else
                {
                        inFile >> number;
                        Matrix[i][j] = number;
                }
        }
}

inFile.close();
numberRows = size;
numberColumns = size * 2;
```

```
}

/*************************************************************************/
/*************************************************************************/

void OpenOutFile()
{
        string dataOut;

        cout << "Enter name of output File" << endl;
        cin  >> dataOut;

        outFile.open(dataOut,ios::out);

        if(!outFile)
        {
                cout << "Can not open File" << endl;
                exit( 1);
        }
}

/*************************************************************************/
/*************************************************************************/

void RowReduction(double Matrix [MAX_ROWS][MAX_COLUMNS])
{
        int row,column,row2;
        double scalar;

        for(row = 0; row < numberRows ; row++)
        {
                //Get unity elements on the main diagonal
                scalar = 1/Matrix[row][row];

                for(column = 0; column < numberColumns; column++)
                {
                        Matrix[row][column] =  Matrix[row][column] * scalar;
                }

                //Reduce elements not in row to aij = aij - arowj * airow
                for(row2 = 0; row2 < numberRows; row2++)
                {
```

```
                    if(row2 != row )
                    {
                        scalar =  Matrix[row2][row];

                        for(column = row; column < numberColumns; column++)
                        {
                            Matrix[row2][column] = Matrix[row2][column] -
                            scalar * Matrix[row][column];
                        }
                    }
                }
            }
}

/*************************************************************************/
/*************************************************************************/

void OutputTime(    clock_t start,   clock_t  end)
{
        double startSecs,endSecs;

        //Convert processor time to seconds
        startSecs  = double(double(start)/CLK_TCK);
        endSecs = double(double(end)/CLOCKS_PER_SEC);

        cout << endl;
        cout << "------------------------------" << endl;
        cout << setw(14) << "START TIME: " << '\t' << startSecs << endl;
        cout << setw(14) << "THE END TIME: " << '\t' << endSecs << endl;
        cout << setw(14) << "TOTAL TIME: " <<  '\t' << endSecs-startSecs
             << endl;
        cout << "------------------------------" << endl;
        cout << endl;
        outFile << "Time Test " << ": " << '\t' << endSecs-startSecs << endl;
}

/*************************************************************************/
/*************************************************************************/
```

COMPUTER PROGRAM A.3

```
//Tege Radcliffe
//Fall 1999
//Gradient Projection Method
//Program uses Gradient Projection Method to find Inverse Of Matrix
//Then Outputs Time and Inverse to File and Screen

#include <stdio.h>
#include <io.h>
#include<iostream.h>
#include<fstream.h>
#include<iomanip.h>
#include<math.h>
#include<ctype.h>
#include<time.h>
#include<string.h>
#include <stdlib.h>

//Max Matrix Size
const MAX_ROWS  = 1024;
const MAX_COLUMNS = 1024;

//Global Variables
typedef char string[50];
ifstream inFile;
ofstream outFile;
ofstream outFile2;
int size;
double I[MAX_ROWS][MAX_COLUMNS];
double A[MAX_ROWS][MAX_COLUMNS];
double Inv[MAX_ROWS][MAX_COLUMNS];
double start2,end2;
int numExecutions = 5;
int execution;
double times[5];
```

```
//Functions
void OpenInFile();
void OpenOutFile(int);
void PrintHeader();
void CreateIdentity();
void OutputTime(clock_t,clock_t);
void OutputAvgTime();
void GradientProjection();
void OutputInverse();
void ReadInMatrix();

int main()
{
        clock_t start,end;
        int testNumber = 1;

        OpenInFile();

        OpenOutFile(0);

        ReadInMatrix();

        CreateIdentity();

        inFile.close();

        PrintHeader();

        //Test Matrix 5 times for time
        for(execution = 0; execution < numExecutions;execution++)
        {
                start2 = 0;
                end2 = 0;

                start = clock();

                GradientProjection();

                end = clock();

                OutputTime(start,end);
        }
```

```
        OutputAvgTime();

        outFile.close();

        OpenOutFile(1);

        OutputInverse();

        outFile.close();

        return 0;
}

/*************************************************************************/
/*************************************************************************/
void OpenInFile()
{
        string dataIn;

        cout << "Enter name of matrix File" << endl;
        cin  >> dataIn;

        inFile.open(dataIn,ios::out);

        if(!inFile)
        {
                cout << "Can not open File" << endl;
                exit( 1);
        }
}

/*************************************************************************/
/*************************************************************************/
void OpenOutFile(int type)
{
        string dataOut;

        if(type = = 0)
                cout << "Enter Name of Timed Results Output File " << endl;
        else
                cout << "Enter Name of Inverse Output File " << endl;
```

```
        cin >> dataOut;

        outFile.open(dataOut,ios::out);

        if(!outFile)
        {
                cout << "Can not open File" << endl;
                exit( 1);
        }
}

/*************************************************************************/
/*************************************************************************/
void PrintHeader()
{
        int count = 10;
        int width = 0;

        while(size > count)
        {
                count = count * 10;
                width++;
        }

        width = width * 2;

        outFile << setfill('*') << setw(45+width) << endl;
        outFile << setfill('*') << setw(45 + width) << endl;
        outFile << "**"       << setfill(' ') << setw(42+width) << "**" << endl;
        outFile << "**    TIME RESULTS FOR A " << size << "X" << size
                << " MATRIX   " << setw(4) << "**" << endl;
        outFile << "**     Gradient Projection Method " << setfill(' ')
                << setw(9+width) << "**" << endl;
        outFile << "**" << setfill(' ') << setw(42 + width) << "**" << endl;
        outFile << setfill('*') << setw(45 + width) << endl;
        outFile << setfill('*') << setw(45 + width) << endl;
        outFile << endl;
}

/*************************************************************************/
/*************************************************************************/
void CreateIdentity()
```

```
{
        int row;
        int column;

        for(row=0;row<size;row++)
        {
                for(column=0;column<size;column++)
                {
                        if(row = = column)
                                I[row][column] = 1;
                        else
                                I[row][column] = 0;
                }
        }
}

/*************************************************************************/
/*************************************************************************/
void ReadInMatrix()
{
        int row,column;

        inFile >> size;

        for(row=0;row<size;row++)
        {
                for(column=0;column<size;column++)
                {
                        inFile >> A[row][column];
                }
        }
}

/*************************************************************************/
/*************************************************************************/
void OutputTime(clock_t start, clock_t end)
{

        start2  = double(double(start)/CLK_TCK);

        cout << setw(13) << size << "X" << size << endl << endl;
        cout << setw(14) << "START TIME: " << '\t' << start2 << endl;
```

```
        end2 = double(double(end)/CLOCKS_PER_SEC);

        cout << setw(14) << "THE END TIME: " << '\t' << end2 << endl;
        cout << setw(14) << "TOTAL TIME: " << '\t' << end2-start2 << endl;
        cout << "------------------------------" << endl;

        times[execution] = end2-start2;

        outFile << "Time Test " << execution+1 << ": " << times[execution]
                << endl;

}

/*************************************************************************/
/*************************************************************************/
void GradientProjection()
{
        double F[MAX_ROWS];
        double X[MAX_ROWS];
        double R[MAX_ROWS];
        double K[MAX_ROWS];
        double RR;
        double PreviousX[MAX_ROWS];
        int row, column;
        double error;
        error = .000001;
        int count;
        int i,k,l,m;
        double T;
        bool test;

        // Compute x1 = x0 + α0r0
        for(column = 0; column < size;column++)
        {
                test = false;

                for(i=0;i<size;i++)
                {
                        //Load column of Identity
                        F[i] = I[i][column];

                        //Use column of Identity for initial guess
```

```
            X[i] = I[i][column];

}
do
{
            T = 0;
            RR = 0;

            for(k = 0; k < size; k++)
            {
                        R[k] = 0;
                        PreviousX[k] = X[k];
            }

            //Compute Ax0
            for(k = 0; k < size; k++)
            {
                        for(l = 0; l < size; l++)
                        {
                                    R[k] = A[k][l] * X[l] + R[k];

                        }
            }

            //Compute F - Ax0
            for(k = 0; k < size; k++)
            {
                        R[k] = F[k] - R[k];
            }

            //Compute r0r0
            for(k = 0; k < size; k++)
            {
                        RR = R[k] * R[k] + RR;
                        K[k] = 0;
            }

            //Compute Ar0, r0
            for(k = 0; k < size; k++)
            {
                        for(l = 0; l < size; l++)
                        {
```

```
                        K[k] = R[l] * A[k][l] + K[k];
                }
        }

        for(k = 0; k < size; k++)
        {
                T = K[k] * R[k] + T;
        }

        //Compute α0
        T = RR/T;

        //Store X in Previous
        for(k = 0; k < size; k++)
        {
                PreviousX[k] = X[k];
        }

        //Calculate x1
        for(k = 0; k < size; k++)
        {
                X[k] = R[k] * T + X[k];
        }

        count = 0;

        //Test X and PreviousX for error
        for(m = 0; m < size; m++)
        {
                if(fabs(X[m] - PreviousX[m]) < error)
                        count++;
        }

        if(count = = size)
                test = true;

}while(!test);

//Store column of inverse
for(row = 0;row<size;row++)
{
        Inv[row][column] = X[row];
```

```
                }
        }
}

/*************************************************************************/
/*************************************************************************/
void OutputAvgTime()
{
        double sum = 0;
        int g;

        for(g = 0;g<numExecutions;g++)
        {
                sum = sum + times[g];
        }

        double average = sum/numExecutions;

        outFile << "The average time was: " << average << endl;
}

/*************************************************************************/
/*************************************************************************/
void OutputInverse()
{
        int i,j;
        cout.setf(ios::fixed, ios::floatfield);
        cout.setf(ios::showpoint);
        outFile.setf(ios::fixed, ios::floatfield);
        outFile.setf(ios::showpoint);
        outFile << setfill(' ');

        for(i = 0; i < size; i++)
        {
                for( j = 0; j < size; j++)
                {
                        if(j % 4 = = 0)
                                outFile << endl;

                        outFile << setprecision(5) << setw(10) << Inv[i][j];
                        cout << endl;
                        cout << setprecision(5) << setw(10) << Inv[i][j];
```

```
            }

            cout << endl;
            outFile << endl;
        }
}
```