**Summer Training**  
Undertaken at  
**Defence Laboratory, Jodhpur**  
**Defence Research and Development Organization**

**Project Report**on **“Software Development for Solving Linear Equation Using Kaczmarz Method”**

From: 21 May 2019  
To: 5 July 2019

**Guided by- Submitted by-**

**Mr. D.K. Tripathi Divyanshu Gupta Scientist ‘F’ 3rd Year B.E. (ECC)  
Defence Laboratory, Jodhpur Faculty of Engineering,  
DRDO M.B.M. Engineering College, Jodhpur  
 JNV University, Jodhpur**

**CERTIFICATE**

This is to certify that the project **“Software Development for Solving Linear Equation Using Kaczmarz Method”** is the bona fide work carried out by Mr. Divyanshu Gupta at Defence Laboratory, Jodhpur during summer training (21 May, 2019 to 5 July. 2019) in partial fulfilment of the requirements for the award of the degree of Bachelor of Engineering as per the norms prescribed by Faculty of Engineering, M.B.M. Engineering College, Jodhpur (JNVU, Jodhpur).

**D.K. Tripathi  
Scientist ‘F’  
Defence Laboratory, Jodhpur  
Defence Research and Development Organization   
(DRDO)**

**ACKNOWLEDGEMENT**

I take the opportunity to express my heartfelt gratitude to all those people who have provided their immense support and aspect their valuable time in guiding me. It is due to their support, guidance, encouragement and supervision that I have successfully completed summer training project on **“Software Development for Solving Linear Equation Using Kaczmarz Method”.**

I am highly indebted to my guide Mr. D.K. Tripathi, Scientist ‘F’ for his guidance and constant supervision as well as for providing necessary information regarding the project.

I would like to articulate my guidance towards the Director, Defence Laboratory, Jodhpur,   
DRDO for giving me opportunity to apply theoretical knowledge into practice.

In the end, I would like to extend my sincere thanks to my friends and family members for their support and positive criticisms.

**Divyanshu Gupta  
3rd Year (ECC)  
B.E.  
Faculty of Engineering,  
M.B.M. Engineering College, Jodhpur  
JNV University, Jodhpur**

**TABLE OF CONTENTS**

1. Introduction 1
2. Kaczmarz Method For Solving Linear Equation 2
3. Software Design And Development 9
4. Hardware Requirement
5. Software Requirement
6. Software Development Life Cycle
7. Software Testing 11

* Test Performed With Different Sets of Linear Equation

1. Computer Programme In C language 15
2. Conclusion 19
3. Reference 20

**1. Introduction**

An entirely different approach for tomographic imaging consists of assuming that the cross section consists of an array of unknowns, and then setting up algebraic equations for the unknowns in terms of the measured projection data. Although conceptually this approach is much simpler than the transform-based methods discussed in previous sections, for medical

applications it lacks the accuracy and the speed of implementation. However, there are situations where it is not possible to measure a large number of projections, or the projections are not uniformly distributed over 180° or 360°) both these conditions being necessary requirements for the transform based techniques to produce results with the accuracy desired in medical imaging. An example of such a situation is earth resources imaging using cross-borehole measurements. Problems of this type are sometimes more amenable to solution by algebraic techniques. Algebraic techniques are also useful when the energy propagation paths between the source and receiver positions are subject to ray bending on account of refraction, or when the energy propagation undergoes attenuation along ray paths as in emission CT. Unfortunately, many imaging problems where refraction is encountered also suffer from diffraction effects. As will be obvious from the discussion to follow, in algebraic methods it is essential to know ray paths that connect the corresponding transmitter and receiver positions. When refraction and diffraction effects are substantial, (medium inhomogeneities exceed 10% of the average background value and the correlation length of these inhomogeneities is comparable to a wavelength), it becomes impossible to predict these ray paths. If algebraic techniques are applied under these conditions, we often obtain meaningless results.

If the refraction and diffraction effects are small, (medium inhomogeneities are less than 2 to 3% of the average background value and the correlation width of these inhomogeneities is much greater than a wavelength), in some cases it is possible to combine algebraic

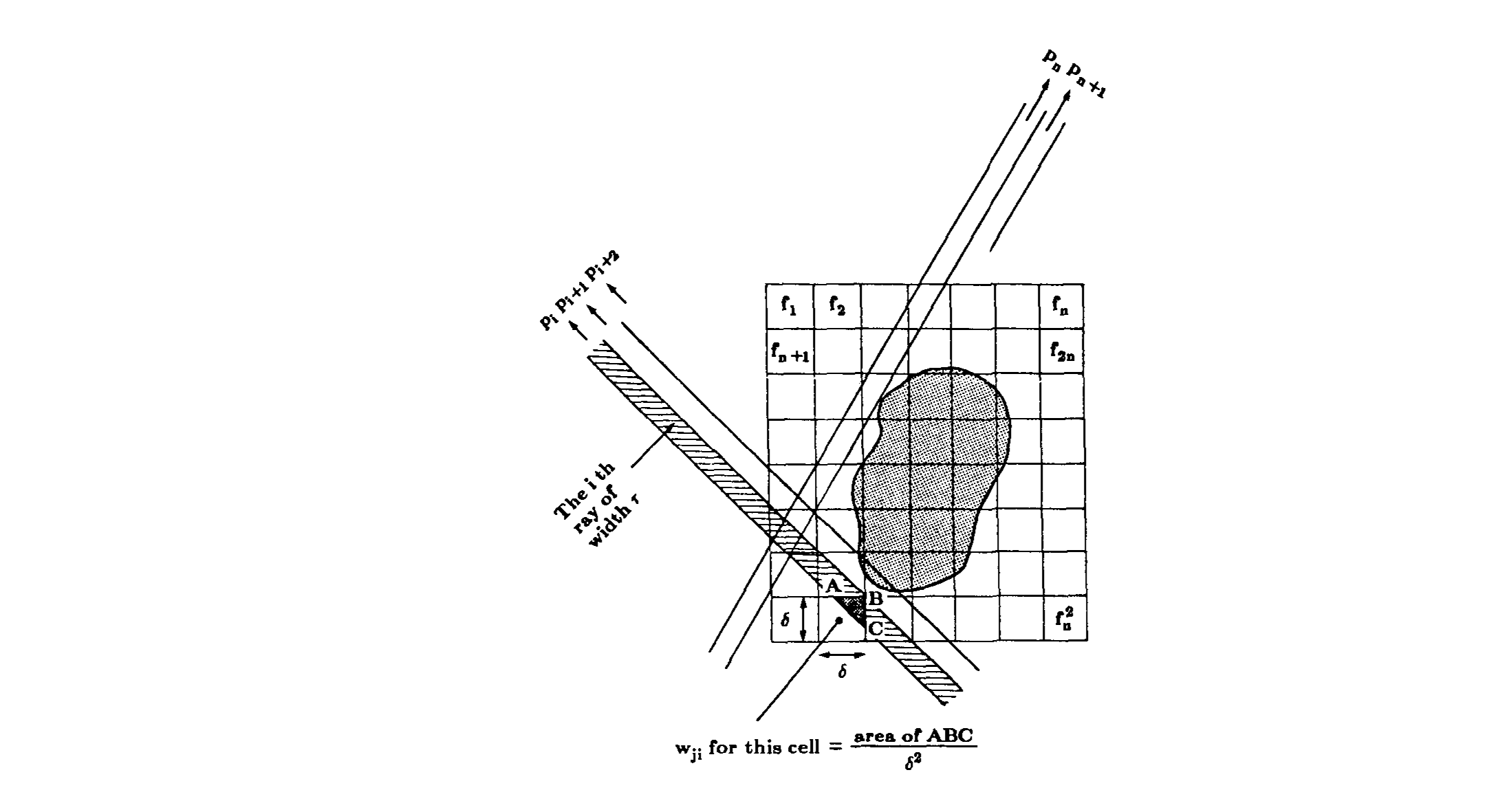
techniques with digital ray tracing techniques and devise iterative procedures in which we first construct an image ignoring refraction, then trace rays connecting the corresponding transmitter and receiver locations through this distribution, and finally use these rays to construct a more accurate set of ALGEBRAIC RECONSTRUCTION ALGORITHMS 275 algebraic equations. Experimental verification of this iterative procedure for weakly refracting objects has been obtained. Space limitations prevent us from discussing here the combined ray tracing and algebraic reconstruction algorithms. Our aim in this section is to merely introduce the reader to the algebraic approach for image reconstruction. First, we will show how we may construct a set of linear equations whose unknowns are elements of the object cross section. The Kaczmarz method for solving these equations will then be presented. This will be followed by the various approximations that are used in this method to speed up its computer implementation.

**2. Kaczmarz Method for Solving Linear Equation**

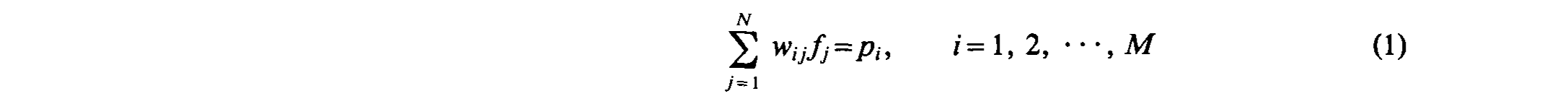
In Fig. 2.1, we have superimposed a square grid on the image f(x, y); we will assume that in each cell the function& y) is constant. Let fj denote this constant value in the jth cell, and let N be the total number of cells. For algebraic techniques, a ray is defined somewhat differently. A ray is now a “fat” line running through the (x, y)-plane. To illustrate this we have shaded the ith ray in Fig. 2.1, where each ray is of width r. In most cases, the ray width is approximately equal to the image cell width. A line integral will now be called a ray-sum.

Like the image, the projections will also be given a one-index representation

Fig. 2.1



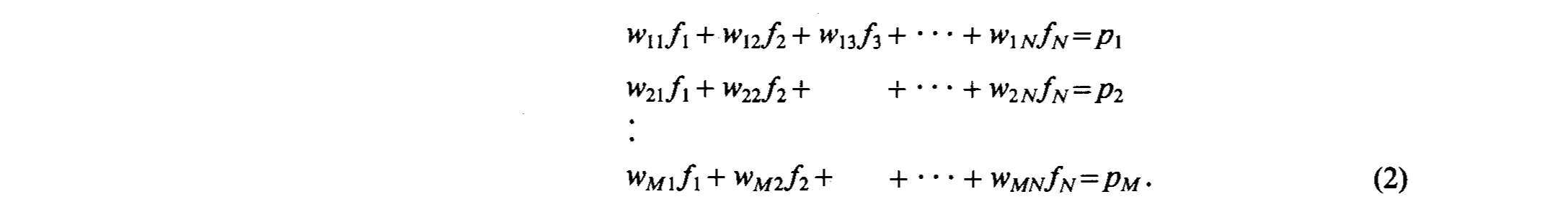
Let pi be the ray-sum measured with the ith ray as shown in Fig. 2.1. The relationship between the 4’s and pi’s may be expressed as



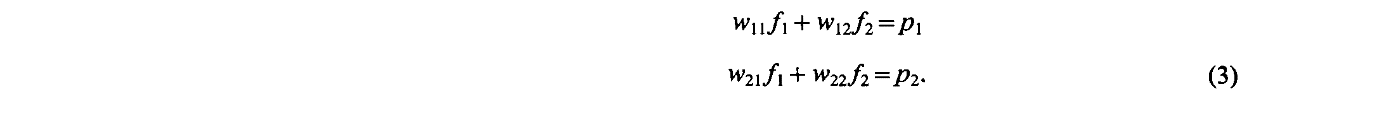
where M is the total number of rays (in all the projections) and Wij is the weighting factor that represents the contribution of the jth cell to the ith ray integral. The factor Wij is equal to the fractional area of the jth image cell intercepted by the ith ray as shown for one of the cells in Fig. 2.1. Note that most of the wij’s are zero since only a small number of cells contribute to any given ray-sum.

If M and N were small, we could use conventional matrix theory methods to invert the system of equations in (1). However, in practice N may be as large as 65,000 (for 256 x 256 images), and, in most cases for images of this size, M will also have the same magnitude. For these values of M and N the size of the matrix [ WijJ in (1) is 65,000 X 65,000 which precludes any possibility of direct matrix inversion. Of course, when noise is present in the measurement data and when A4 < N, even for small Nit is not possible to use direct matrix inversion, and some least squares method may have to be used. When both M and N are large, such methods are also computationally impractical.

For large values of M and N there exist very attractive iterative methods for solving (1). These are based on the “method of projections” as first proposed by Kaczmarz and later elucidated further by Tanabe. To explain the computational steps involved in these methods, we first write (1) in an expanded form:



A grid representation with N cells gives image N degrees of freedom. Therefore, an image, represented by (f1, f2, + ··· , fN), may be considered to be a single point in an N-dimensional space. In this space, each of the above equations represents a hyperplane. When a unique solution to these equations exists, the intersection of all these hyperplanes is a single point giving that solution. This concept is further illustrated in Fig. 2.2 where, for the purpose of display, we have considered the case of only two variables f, and f2 satisfying the following equations:



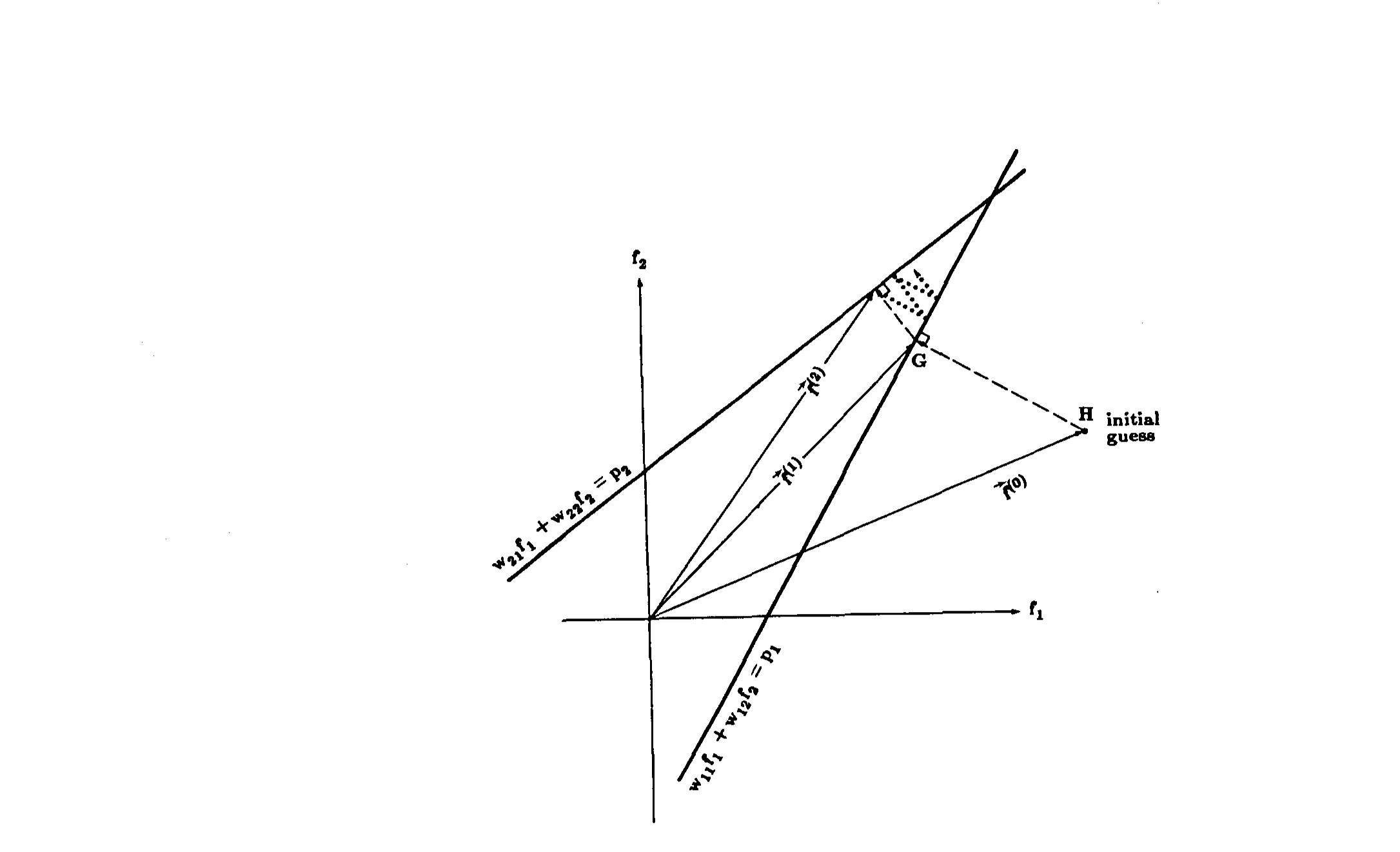
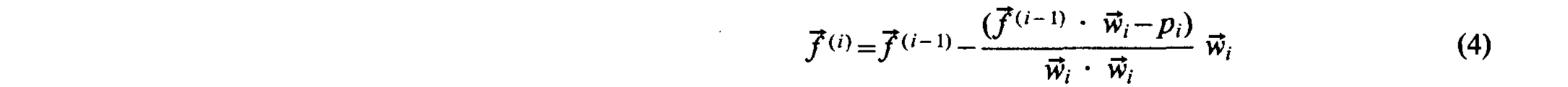


Fig. 2.2

The computational procedure for locating the solution in Fig. 2.2 consists of first starting with an initial guess, projecting this initial guess on the first line, re-projecting the resulting point on the second line, and then projecting back onto the first line, and so forth. If a unique solution exists, the iterations will always converge to that point.

For the computer implementation of this method, we first make an initial guess at the solution. This guess, denoted by f(0)1, f(0)2, … , f(0)N, is represented vectorially by f(0). In the N-dimensional space. In most cases, we simply assign a value of zero to all the fi’s. This initial guess is projected on the hyperplane represented by the first equation in (2) giving vector f(1), as illustrated in Fig. 2.2 for the two-dimensional case. Vector f(1) is projected on the hyperplane represented by the second equation in (2) to yield Vector f(2) and so on. When vector f(i-1) is projected on the hyperplane represented by the ith equation to yield vector f(i), the process can be mathematically described by



where Vector Wi = (Wi1, Wi2, ···, WiN), and Vector Wi DOT Vector Wi is the dot product of Vector Wi with itself. To see how (4) comes about we first write the first equation of (2) as

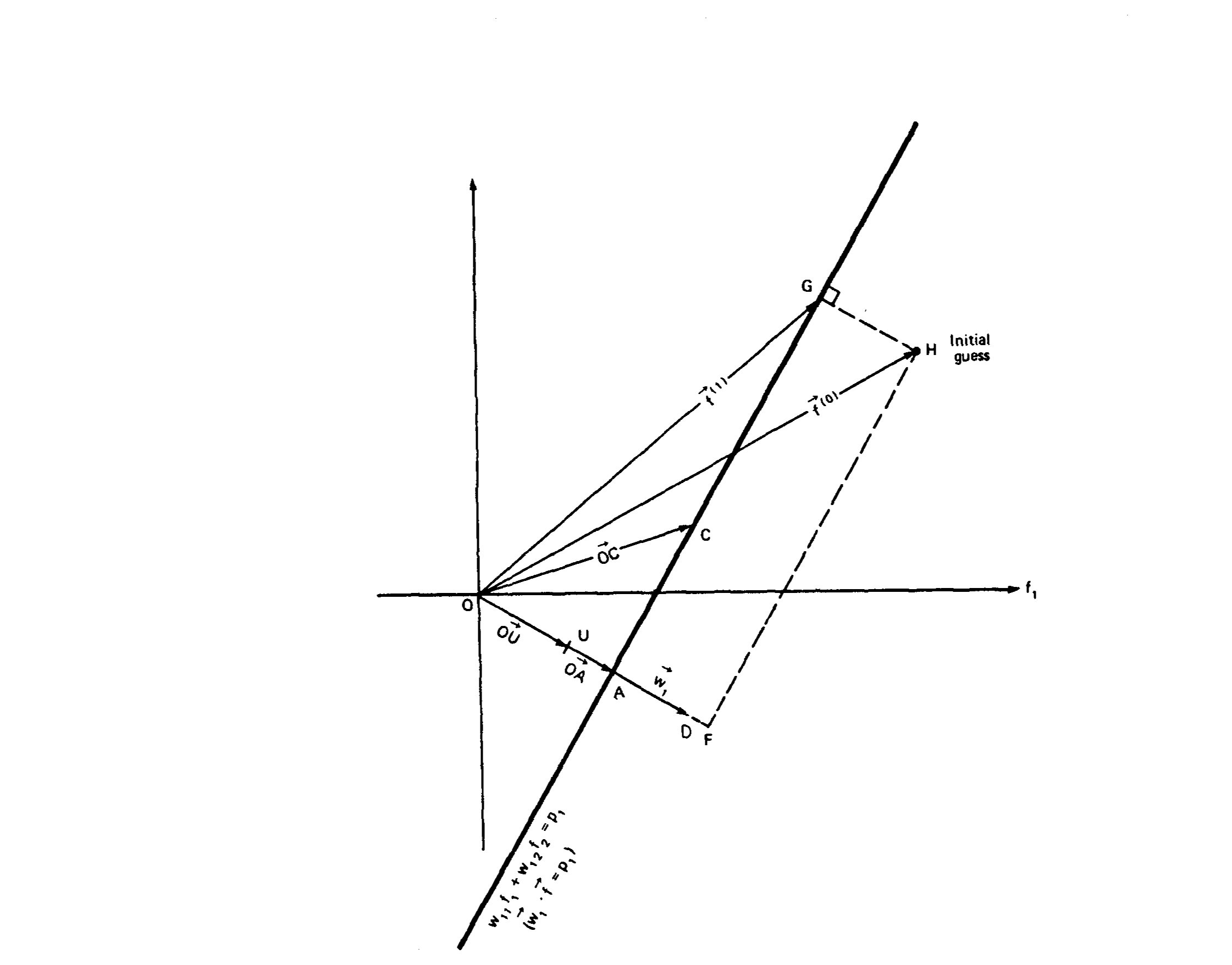
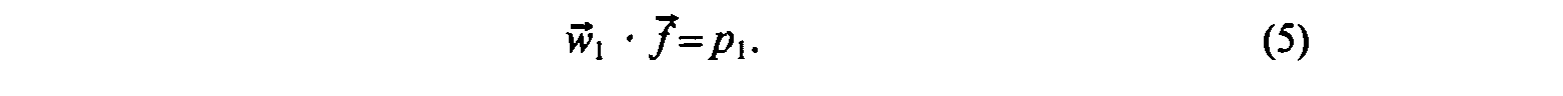
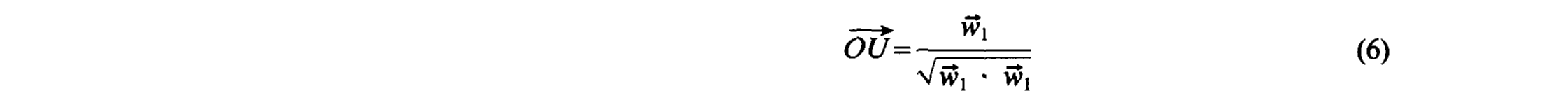


Fig. 2.3

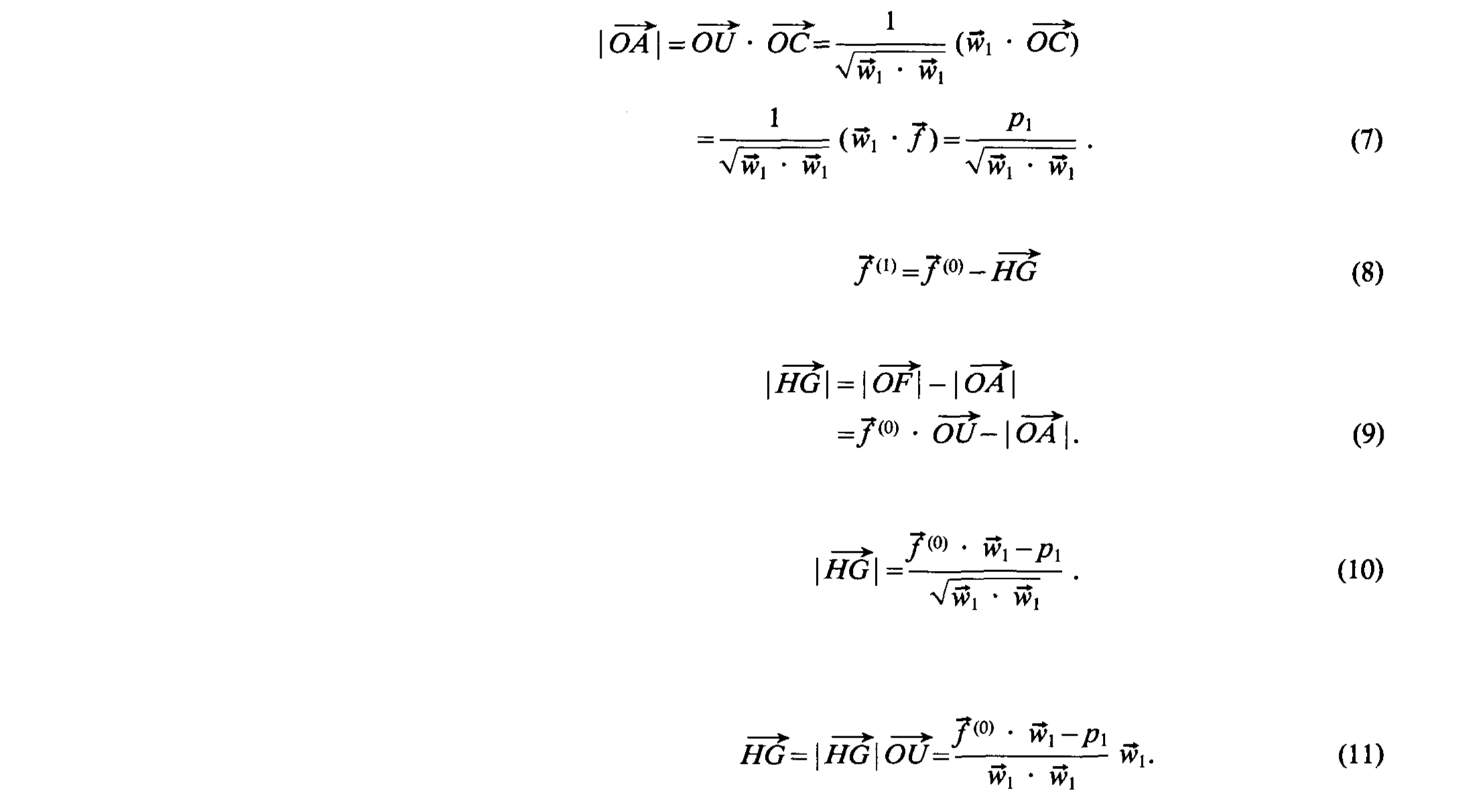
Follows:



The hyperplane represented by this equation is perpendicular to the vector W1. This is illustrated in Fig. 2.3, where the vector OD represents W1. This equation simply says that the projection of a vector OC (for any point C on the hyperplane) on the vector W1 is of constant length. The unit vector OU along W1, is given by



and the perpendicular distance of the hyperplane from the origin, which is equal to the length of OA in Fig. 2.3, is given by Vector OC DOT Vector OU:



Substituting (11) in (8), we get (4).

As mentioned before, the computational procedure for algebraic reconstruction consists of starting with an initial guess for the solution, taking successive projections on the hyperplanes represented by the equations in (2), eventually yielding Vector f(M). In the next iteration, Vector f(M) is projected on the hyperplane represented by the first equation in (2), and then successively onto the rest of the hyperplanes in (2), to yield Vector f(2M), and so on. Tanabe has shown that if there exists a unique solution Vector fs to the system of equations (2), then



A few comments about the convergence of the algorithm are in order here. If in Fig. 2.2, the two hyperplanes are perpendicular to each other, the reader may easily show that given for an initial guess any point in the (f1, f2)-plane, it is possible to arrive at the correct solution in only two steps like (4). On the other hand, if the two hyperplanes have only a very small angle between them, k in (12) may acquire a large value (depending upon the initial guess) before the correct solution is reached. Clearly the angles between the hyperplanes considerably influence the rate of convergence to the solution. If the M hyperplanes in (2) could be made orthogonal with respect to one another, the correct solution would be arrived at with only one pass through the A4 equations (assuming a unique solution does exist). Although theoretically such orthogonalization is possible using, for example, the Gram-Schmidt procedure, in practice it is computationally not feasible. Full orthogonalization will also tend to enhance the effects of the ever present measurement noise in the final solution. Ramakrishnan et al. [Ram791] have suggested a pairwise orthogonalization scheme, which is computationally easier to implement and at the same time considerably increases the speed of convergence. A simpler technique,is to carefully choose the order in which the hyperplanes are considered. Since each hyperplane represents a distinct ray integral, it is quite likely that adjacent ray integrals (and thus hyperplanes) will be nearly parallel. By choosing hyperplanes representing widely separated ray integrals, it is possible to improve the rate of convergence of the Kaczmarz approach.

A not uncommon situation in image reconstruction is that of an overdetermined system in the presence of measurement noise. That is, we may have M > N in (2) and p1, p2, ..., pm corrupted by noise. No unique solution exists in this case. In Fig. 2.4 we have shown a two-variable system represented by three “noisy” hyperplanes. The broken line represents the course of the solution as we successively implement (4). Now the “solution” does not converge to a unique point, but will oscillate in the neighbourhood of the intersections of the hyperplanes.

When M < N a unique solution of the set of linear equations in (2) does not exist, and, in fact, an infinite number of solutions are possible. For example, suppose we have only the first of the two equations in (3) to use for calculating the two unknowns f1, and f2; then the solution can be anywhere on the line corresponding to this equation. Given the initial guess vector f(0) (see Fig. 2.3), the best one could probably do under the circumstances would be to draw a projection from vector f(0) on this line, and call the resulting vector f(1) a solution. Note that the solution obtained in this manner corresponds to that point on the line, which is closest to the initial guess. This result has been rigorously proved by Tanabe who has shown that when M < N, the iterative approach described above converges to a solution, call it vector f’s, such that |Vector f(0) – Vector f’s | is minimized.

Besides its computational efficiency, another attractive feature of the iterative approach presented here is that it is now possible to incorporate into the solution some types of a priori information about the image one is reconstructing. For example, if it is known a priori that the image f (x, y) is nonnegative, then in each of the solutions Vector f(k), successively obtained by using (4), one may set the negative components equal to zero. One may similarly incorporate the information that f (x, y) is zero outside a certain area, if this is known.

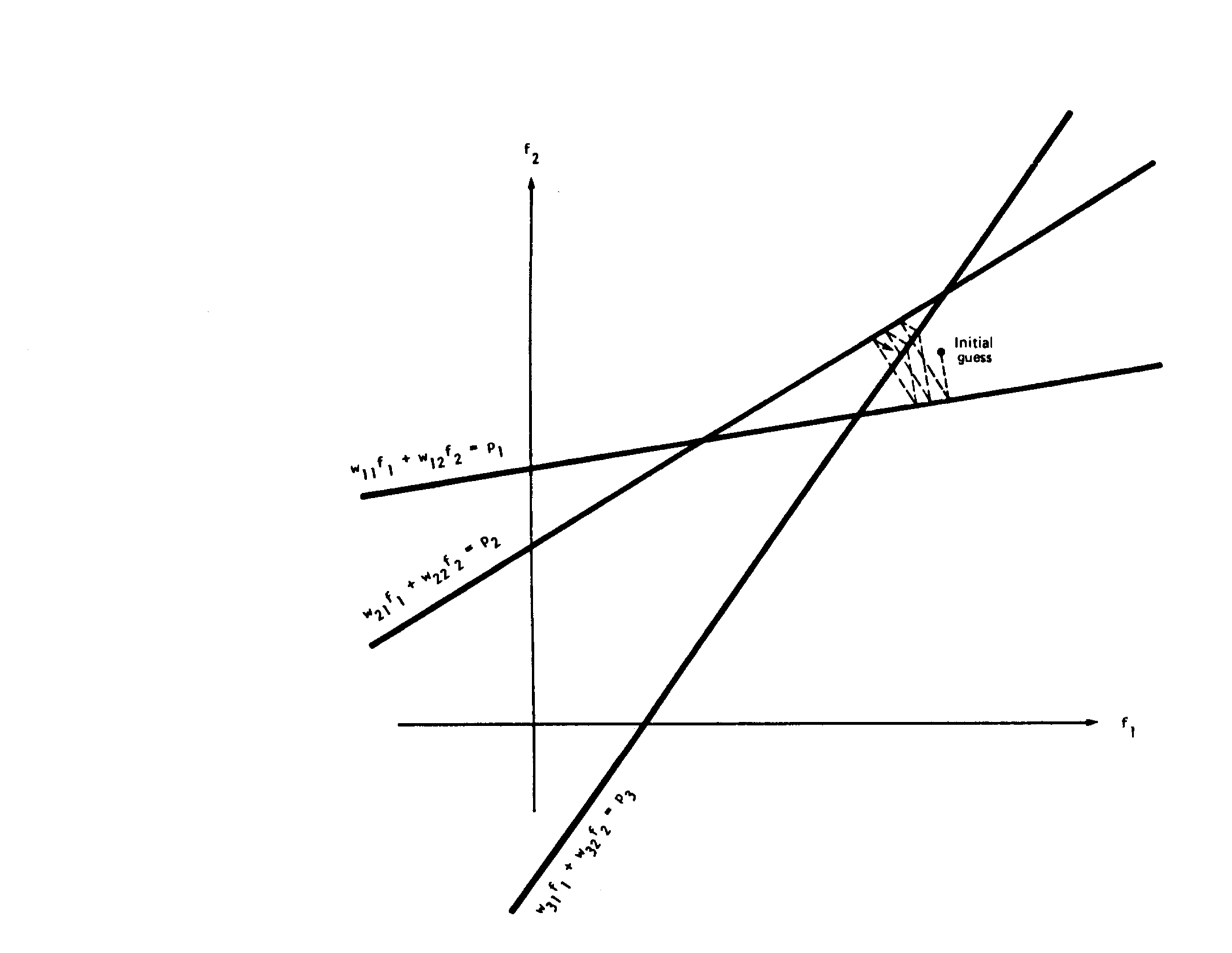
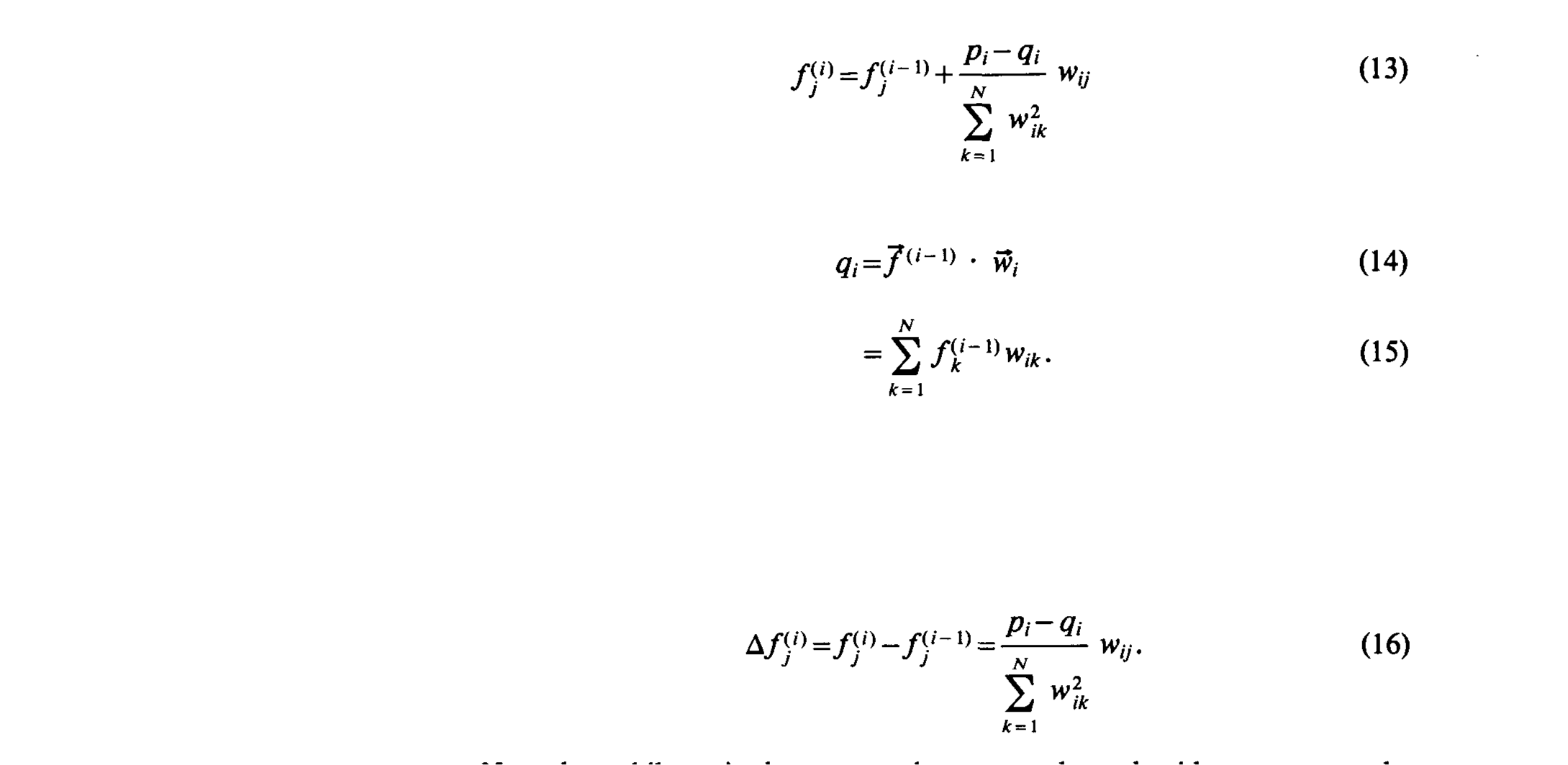


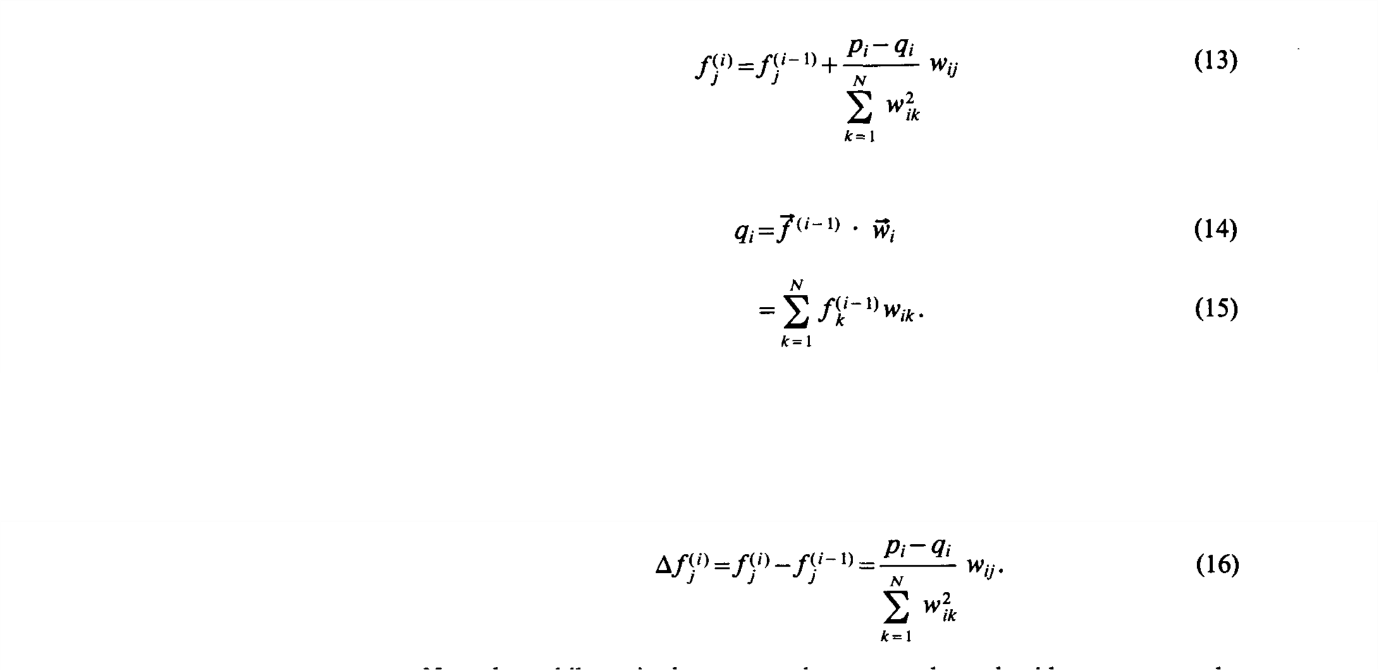
Fig. 2.4

In applications requiring a large number of views and where large-sized reconstructions are made, the difficulty with using (4) can be in the calculation, storage, and fast retrieval of the weight coefficients w,. Consider the case where we wish to reconstruct an image on a 100 x 100 grid from 100 projections with 150 rays in each projection. The total number of weights, Wij, needed in this case is 108, which is an enormous number and can pose problems in fast storage and retrieval in applications where reconstruction speed is important. This problem is somewhat eased by making approximations, such as considering Wij, to be only a function of the perpendicular distance between the centre of the ith ray and the centre of the jth cell. This perpendicular distance can then be computed at run time.

To get around the implementation difficulties caused by the weight coefficients, a myriad of other algebraic approaches have also been suggested, many of which are approximations to (4). To discuss some of the more implementable approximations, we first recast (4) in a slightly different



These equations say that when we project the (i - 1)th solution onto the ith hyperplane [ ith equation in (2)] the gray level of the jth element, whose current value is f(i-1) is obtained by correcting its current value by Δfj(i), where



Note that while pi is the measured ray-sum along the ith ray, qi may be considered to be the computed ray-sum for the same ray based on the (i -1)th solution for the image gray levels. The correction Δfj to the jth cell is obtained by first calculating the difference between the measured ray-sum and the computed ray-sum, normalizing this difference by ΣNk=1 W2ik , and then assigning this value to all the image cells in the ith ray, each assignment being weighted by the corresponding Wij.

With the preliminaries presented above, we will now discuss three different computer implementations of algebraic algorithms. These are represented by the acronyms ART, SIRT, and SART.

**3. Software Design and Development**

**3.1. Hardware Requirements**Processor: Intel® Core™ i5-8250U CPU @ 1.60 GHz to 1.80 GHz  
Installed RAM: 8.00 GB  
System Type: 64-bit Operating System

**3.2. Software Requirements**Operating System: Windows 10  
Editor :

**3.3. Software Development Life Cycle (SDLC)**It is a well-defined, structured sequence of stages in software engineering to develop the intended software product.

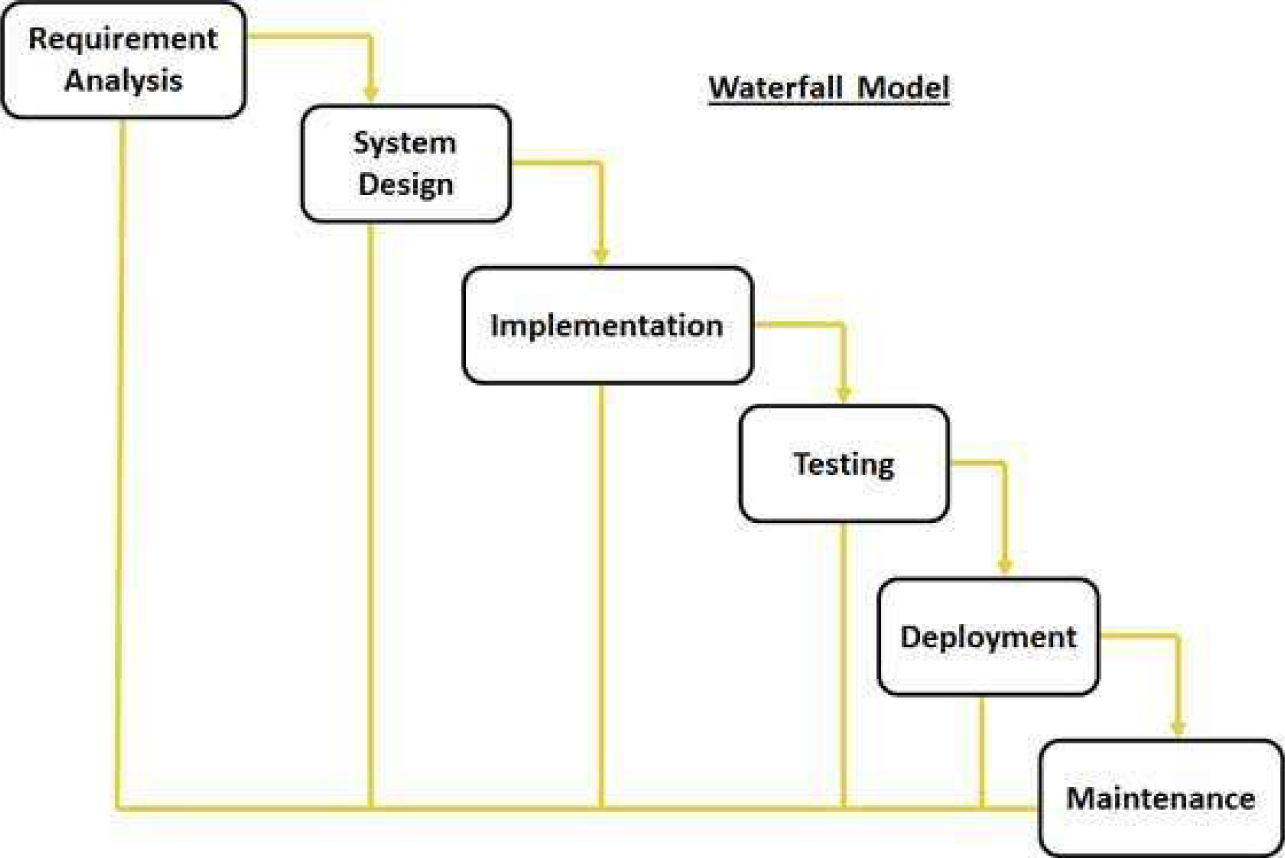
The waterfall model was the first process model to be introduced. It is also referred to as a linear-sequential life cycle model. It is very simple to understand and use. In a waterfall model, each phase must be completed before the next phase can begin and there is no overlapping in the phases.

The waterfall model illustrates the software development process in a linear sequential flow. This means that any phase in the development process begins only if the previous phase is complete. In this waterfall model, the phases do not overlap.

**Waterfall Model – Design**

Waterfall approach was first SDLC model to be used widely in Software Engineering to ensure success of the project. In “The Waterfall” approach, the whole process of software development is divided into separate phases. In this waterfall model, typically, the outcome of one phase acts as the input for the next phase sequentially.

The following illustration is a representation of the different phases of the waterfall model.



**Requirement gathering and analysis** – All possible requirements of the system to be developed are captured in this phase and documented in a requirement specification document.

**System Design** – The requirement specifications from first phase are studied in this phase and the system design is prepared. This system design helps in specifying hardware and system requirements and helps in defining the overall system architecture.

**Implementation** – With inputs from the system design, the system is first developed in small programs called units, which are integrated in the next phase. Each unit is developed and tested for its functionality, which is referred to as Unit Testing.

**Integration and Testing** – All the units developed in the implementation phase are integrated into a system after testing of each unit. Post integration the entire system is tested for any faults and failures.

**Deployment of system** – Once the functional and non-functional testing is done; the product is deployed in the customer environment or released into the market.

**Maintenance** – There are some issues which come up in the client environment. To fix those issues, patches are released. Also to enhance the product some better versions are released. Maintenance is done to deliver these changes in the customer environment.

All these phases are cascaded to each other in which progress is seen as flowing steadily downwards (like a waterfall) through the phases.

The next phases is started only after the defined set of goals are achieved for previous phase and it is signed off, so the name “Waterfall Model”. In this model, phases do not overlap.

**Application** – Every software developed is different and requires a suitable SDLC approach to be followed based on the internal and external factors. Some situations where the use of waterfall model is most appropriate are –  
 **·** Requirements are very well documented, clear and fixed  
 **·** Product definition is stable.  
 **·** Technology is understood and is not dynamic.  
 **·** There are no ambiguous requirements.  
 **·** Ample resources with required expertise are there to support the product.  
 **·** The product is short.

**4. Software Testing**

Software testing is an investigation conducted to provide stakeholders with information about the quality of the software product or service under test. Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. Test techniques include the process of executing a program or application with the intent of finding software bugs (errors or other defects), and verifying that the software product is fit for use.

As per the tasks given to me during my part of a summer trainee, I was assigned with the task to develop a software that could solve linear equation using Kaczmarz method.

**Types of Testing**

**Alpha testing**

Alpha testing is simulated or actual operational testing by potential users/customers or an independent test team at the developers' site. Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing before the software goes to beta testing.

**Beta testing**

Beta testing comes after alpha testing and can be considered a form of external user acceptance testing. Versions of the software, known as beta versions, are released to a limited audience outside of the programming team known as beta testers. The software is released to groups of people so that further testing can ensure the product has few faults or bugs. Beta versions can be made available to the open public to increase the feedback field to a maximal number of future users and to deliver value earlier, for an extended or even indefinite period of time (perpetual beta).

**Testing Levels**

**Unit testing**

Unit testing refers to tests that verify the functionality of a specific section of code, usually at the function level. In an object-oriented environment, this is usually at the class level, and the minimal unit tests include the constructors and destructors.

Developers usually write these types of tests as they work on code (white-box style), to ensure that the specific function is working as expected. One function might have multiple tests, to catch corner cases or other branches in the code. Unit testing alone cannot verify the functionality of a piece of software, but rather is used to ensure that the building blocks of the software work independently from each other.

Unit testing is a software development process that involves a synchronized application of a broad spectrum of defect prevention and detection strategies in order to reduce software development risks, time, and costs. The software developer or engineer performs it during the construction phase of the software development life cycle. Unit testing aims to eliminate construction errors before code is promoted to additional testing; this strategy is intended to increase the quality of the resulting software as well as the efficiency of the overall development process.

Depending on the organization's expectations for software development, unit testing might include static code analysis, data-flow analysis, metrics analysis, peer code reviews, code coverage analysis and other software testing practices.

**Integration testing**

Integration testing is any type of software testing that seeks to verify the interfaces between components against a software design. Software components may be integrated in an iterative way or all together ("big bang"). Normally the former is considered a better practice since it allows interface issues to be located more quickly and fixed.

Integration testing works to expose defects in the interfaces and interaction between integrated components (modules). Progressively larger groups of tested software components corresponding to elements of the architectural design are integrated and tested until the software works as a system.

**Tests performed with different set of given equations**

Problem 1.

X1+X2+X3 = 6

X1+2X2+3X3 = 14

3X1-X2+X3 = 4

Executed Solution

X1=1, X2=2, X3 =3

Problem 2.

X1+X2+X3+X4 = 2

X1+X2-2X3+3X4 = 3

X1+2X2+X3-6X4 = 11

X1+2X2+8X3+7X4 = -9

Executed Solution

X1 = 2, X2 = 2, X3 = -1, X4 = -1

Problem 3.

X1+X2+X3+X4+X5 = 1

X1+4X2+X3-3X4+X5 = -10

X1-3X2+7X3+X4+2X5 = 12

X1-3X2+X3+X4-5X5 = -9

4X1+2X2+5X3+5X4+2X5 = 3

Executed Solution

X1 = -1, X2 = -2, X3 = 0, X4 = 1, X5 = 3

Problem 4.

X1+X2+X3+X4+X5+X6 = -1

X1+X2+X3-X4-X5-X6 = 1

X1-X2+X3-X4+X5+X6 = 5

X1+X2+X3+X4+2X5+X6 = -2

X1+2X2+X3+3X4+2X5+4X6 = -1

X1+2X2+X3+X4+X5-X6 = -6

Executed Solution

X1 = 1, X2 = -1, X3 = 0, X4 = -2, X5 = -1, X6 = 2

Problem 5.

X1+X2+X3+X4+X5+X6+X7 = 4

X1+X2+X3-X4+X5+X6-X7 = -6

X1-X2+X3+X4-X5+X6+X7 = 6

X1+X2+4X3+X4+X5+2X6+X7 = 2

5X1+X2+X3+2X4+2X5+X6+3X7 = 9

X1+2X2+2X3+X4+3X5+3X6+2X7 = 4

X1+X2-X3+X4+X5+X6-3X7 = -2

Executed Solution

X1 = 0, X2 = 1, X3 = -1, X4 = 3, X5 = -2, X6 = 1, X7 = 2

Problem 6.

X1+X2+X3+X4+X5+X6+X7+X8 = 36

X1+X2+X3+X4+X5+X6+X7-X8 = 20

X1+X2+X3+X4+X5+X6-X7-X8 = 6

X1+X2+X3+X4+X5-X6-X7-X8 = -6

X1+X2+X3+X4-X5-X6-X7-X8 = -16

X1+X2+X3-X4-X5-X6-X7-X8 = -24

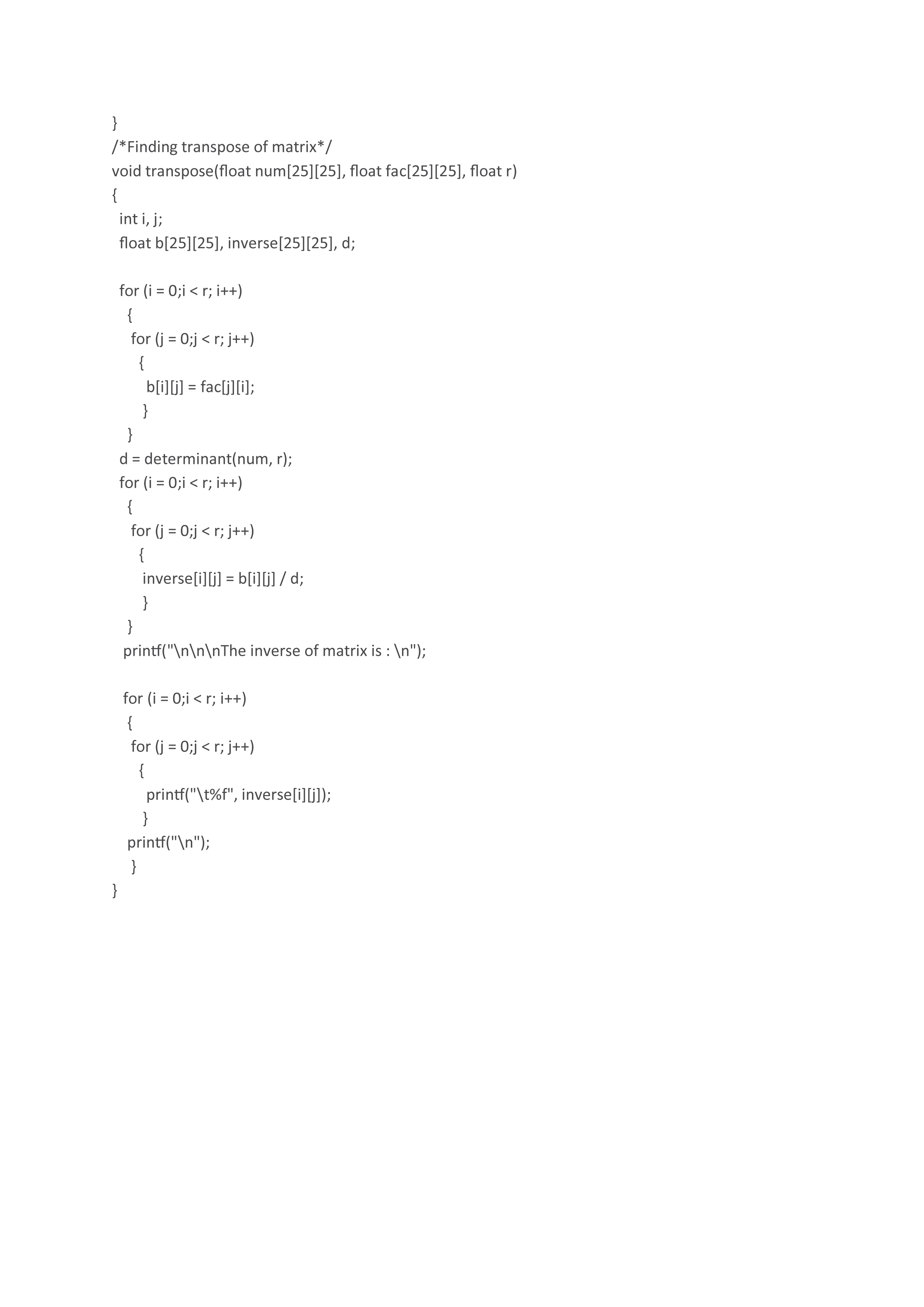
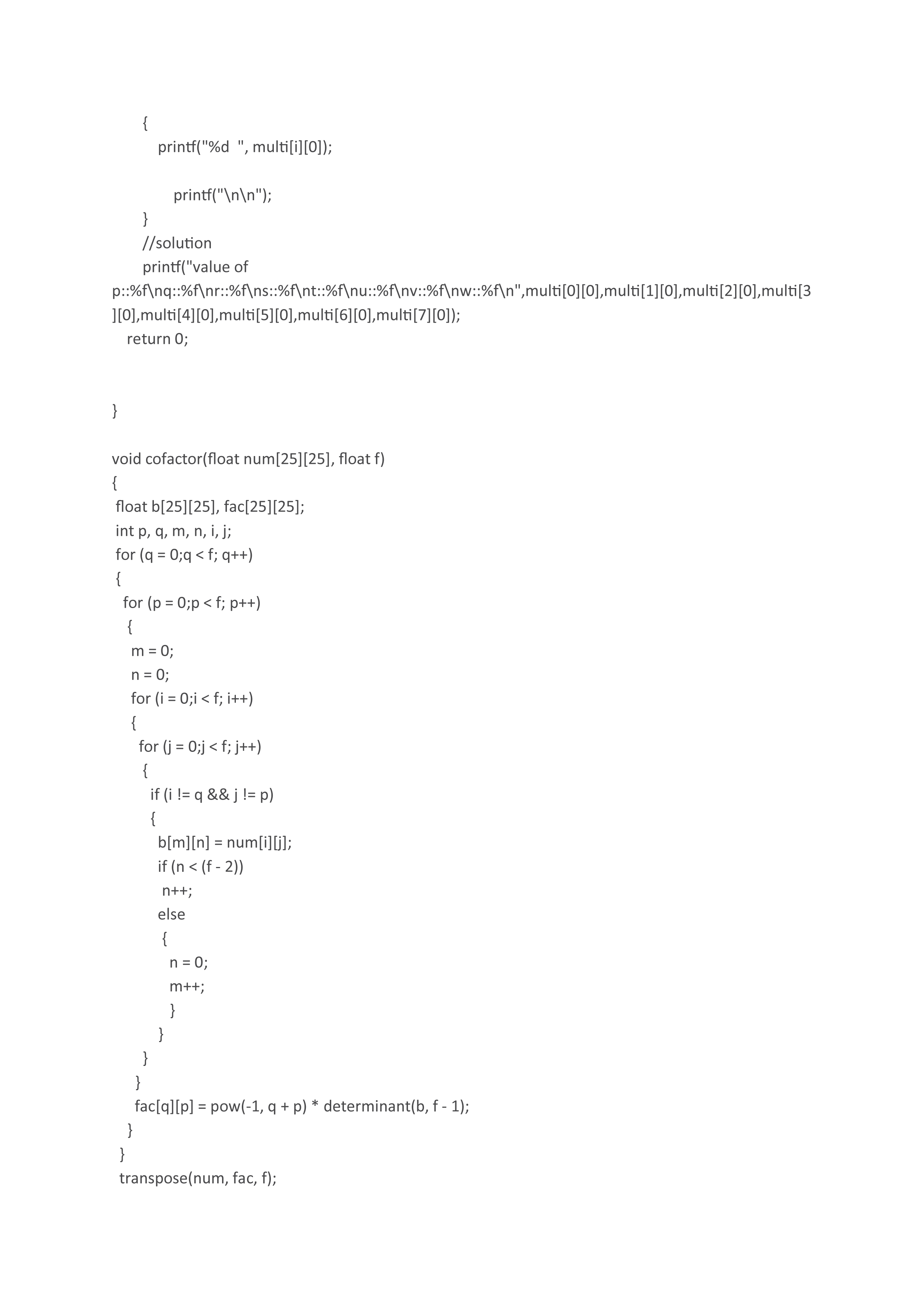
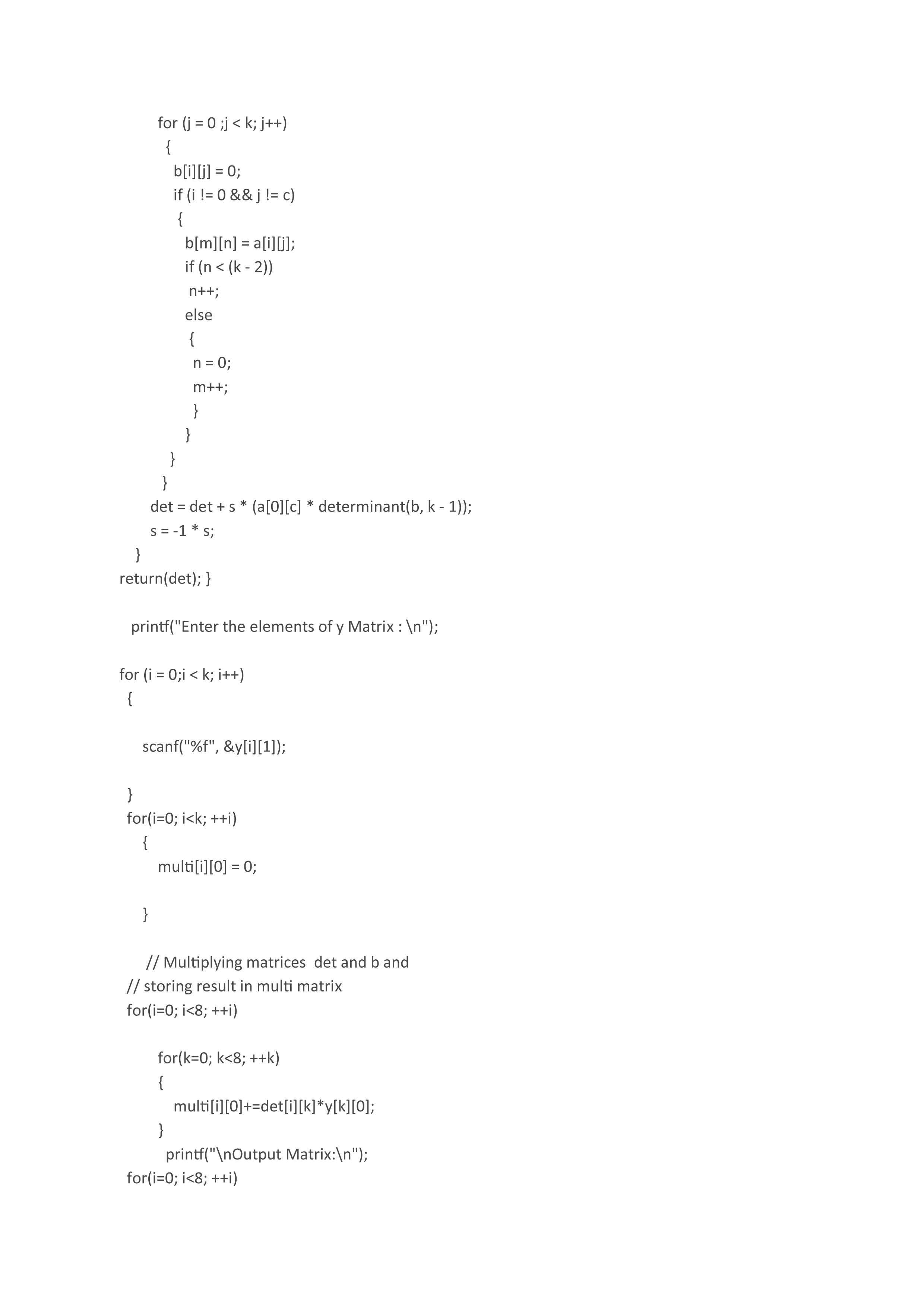
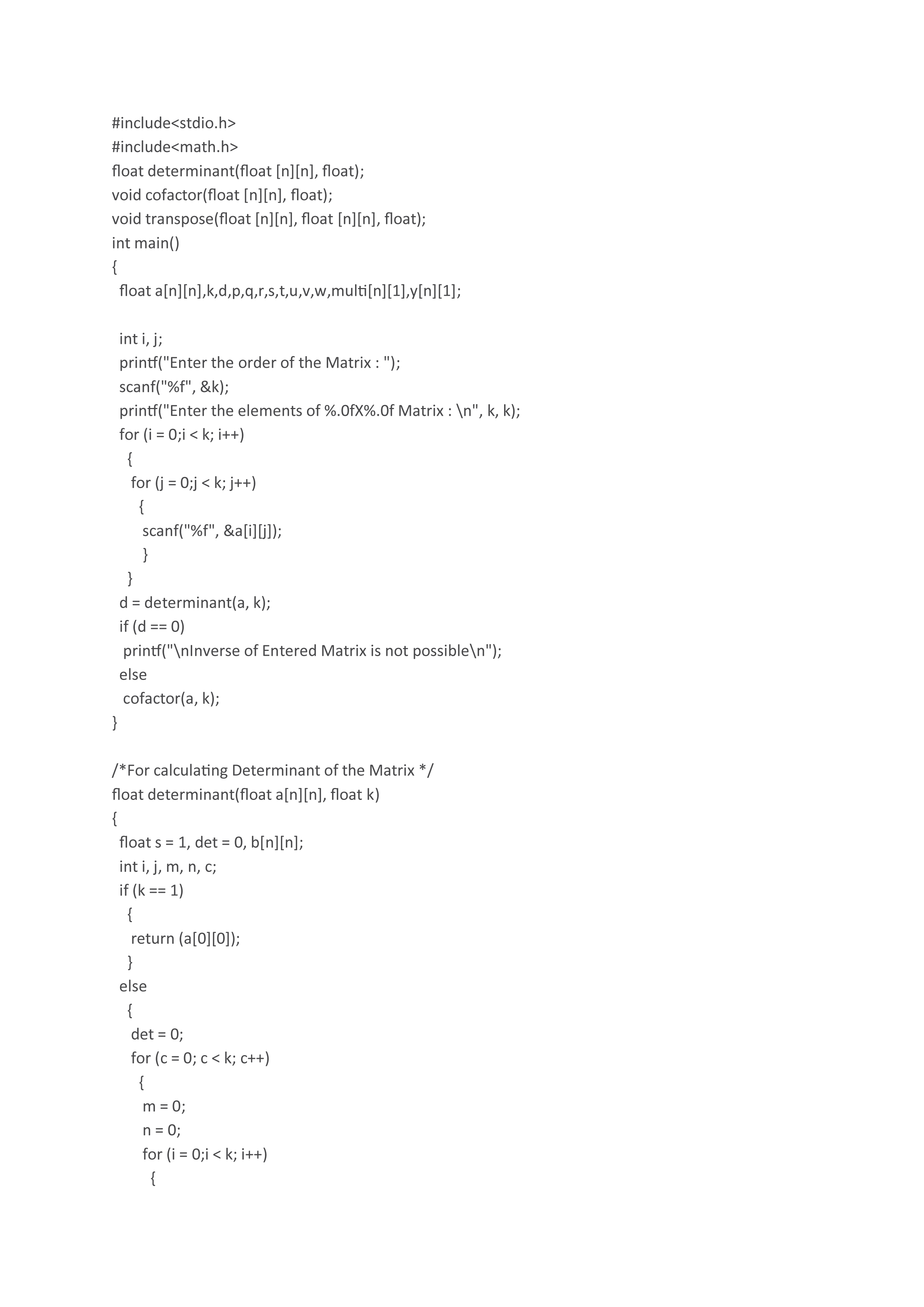
X1+X2-X3-X4-X5-X6-X7-X8 = -30

X1-X2-X3-X4-X5-X6-X7-X8 = -34

Executed Solution

X1 = 1, X2 = 2, X3 = 3, X4 = 4, X5 = 5, X6 = 6, X7 = 7, X8 = 8

**5. Computer Program in C Language**



**6. Conclusion**

The computation of solution linear equations through a computer programme using Kaczmarz Method produces result fast and efficiently.

This report presents the implementation of the software for calculation of results of real life linear equation solving issues by interpreting in a matrix form to get the desired results.

**7.** **References**

1. Programming with C – E Balaguruswamy

2. Elements of Discrete Mathematica – C. L. Liu

3. Software Engineering – Roger S Pressman

4. www.sanfoundry.com

5. www.techdifferences.com

6. Raygun.com C blog

7. www.programiz.com for images