Parallel Numerical Algorithms for the Solution of Diffusion Problems

David John Gavaghan

Linacre College

Thesis submitted for the degree of Doctor of Philosophy
at the University of Oxford

Trinity Term

1991

Parallel Numerical Algorithms for the Solution of Diffusion Problems

David John Gavaghan
Linacre College

Thesis submitted for the degree of Doctor of Philosophy
at the University of Oxford

Trinity Term 1991

Abstract

The purpose of this thesis is to determine the most effective parallel algorithm for the solution of the parabolic differential equations characteristic of diffusion problems. The primary aim is to apply the chosen algorithm to obtain solutions to the equations governing the operation of membrane–covered oxygen sensors, known as Clark electrodes, which are used for monitoring the oxygen concentration of blood. The boundary conditions of this problem require the development of a singularity correction technique.

A brief history of electrochemical sensors leading to the development of the Clark electrode is given, together with the two–dimensional equations and boundary conditions governing its operation. A locally valid series expansion is derived to take care of the boundary singularity, together with a robust method of matching this to the finite difference approximation. Parallel implementations of three representative numerical algorithms applied to a simple model problem are compared by extending Leland's parallel effectiveness model. The chosen parallel algorithm is combined with the singularity correction to obtain a solution to the Clark electrode problem. Numerical experiments show this solution to achieve the required accuracy. Previous one–dimensional models of the Clark electrode are shown to be inadequate before the two–dimensional model is used to examine the variation of operation with design. The understanding gained allows us to demonstrate the advantages of pulse amperometry over steady–state techniques, and to suggest the most appropriate method and design for use in *in vivo* clinical monitoring.

Contents

Acknowledgments

I would first like to thank my supervisor, Dr John Rollett, for all his helpful advice and ideas. His friendly encouragement has made my studies both rewarding and enjoyable.

Those studies would not have been possible without the generous financial assistance of my employers, the Nuffield Department of Anaesthetics, together with the SERC. I would particularly like to thank Dr Clive Hahn who initiated the project and contributed many helpful suggestions.

Along the way I have received help and encouragement from many sources. Particular thanks are due to Andy Mayfield and David Hagan for a multitude of sins, Peter Standen who got me started, Rob Leland, Richard Miller, John Mackenzie and Tom Murdoch who helped me to sort out some of my more muddled ideas, and Jim Davies, Ted and Isaac for many fruitful discussions. Many people have had to endure my sporting excesses, Claudio Mezzetti, Marcus von Krosigk, Mark Howe, Nils Christoffersen and Andy Weir more than most.

My time in Oxford has been enriched by the enlightened guidance of the boys from the *Ad Hox*, and I am indebted to Robert F. Dewey Jr., Dr Simon T. Bennett, and the intrepid John C. McArthy.

Finally I would like to thank the three people without whom this thesis could not have been completed, and to whom it is dedicated.

To

my parents

and

Elaine

Chapter 1

Introduction

From clinical monitoring to space flight, from manufacturing to communications, in just four decades computers have revolutionised almost every aspect of human existence. The desire of scientists and engineers to tackle problems of ever–increasing complexity has been matched by the willingness of computer scientists and manufacturers to build new architectures to meet their needs. The resulting impact on industrial design has created the modern technological world that we know today.

Each new architecture must undergo a maturing process during which the deficiencies and failings of its design can be ironed out before it gains widespread use in the scientific community. The latest generation of computers, parallel multiprocessors with shared or distributed memory, are currently undergoing this maturing process. The problems that we would like to solve in engineering and science today are much more complicated than would have been envisaged in the 1940s. They typically involve the solution of systems of partial differential equations governing fluid flow or structural mechanics, and efficient methods have been developed for their optimal solution on sequential computers. Equivalent algorithms for the new generation of parallel computers are, like the machines themselves, much more complex. To smooth the maturing process on its way so that these machines gain the more general use that their potential warrants, a greater understanding of these complexities is required. It is hoped that by studying the design of appropriate numerical algorithms and applying them to a problem of practical importance in clinical medicine, this thesis will contribute to that understanding.

1.1 Parallelism

Prior to the introduction of the first stored program computers in the late 1940s and early 1950s, arithmetic calculations were typically performed on a desk calculator working with 12 decimal digits in parallel. In addition it was well understood that a large and complex calculation could be performed more quickly by breaking it down into several smaller parts which could be completed independently by several operators working concurrently. By communicating intermediate steps in the calculation to each other, the final solution could be obtained. Providing the problem was sufficiently large, the greater the number of operators, the more rapid

the solution.

The earliest computers used electronic valves to perform bit–serial arithmetic which gave an improved performance of around 10^4 (in terms of speed) over the mechanical machines, and the simplicity of processing a single digit at a time was considered an advantage. However, designers soon realised the advantages of returning to parallel features, first re–considering bit–parallel arithmetic, then developing through functional parallelism and vector–pipelining to the modern day supercomputers with several processors sharing a large memory store. In 1985, with the announcement of the Inmos transputer, the development cycle came full circle. The transputer is a computer in its own right and can be programmed to solve completely any particular problem in the same manner as a conventional, sequential computer. However, like its calculator–operator predecessors, it can communicate with other transputers during its calculations, so that complex problems can also be decomposed and shared out amongst the processors. Again, providing the problem is sufficiently large, the greater the number of processors, the more rapid the solution.

1.2 Application

The particular class of problems that we will consider solving on this new architecture is the parabolic partial differential equations involved in diffusion processes. Our interest is stimulated by a particular practical problem originating in electrochemistry, that of modelling the design and operation of membrane–covered oxygen sensors, particularly those used in the *in vivo* monitoring of blood oxygen concentration. Whilst this means of oxygen monitoring was invented nearly forty years ago, its mathematical modelling has largely been limited to one–dimensional analytic treatments. This has in part been due to the unavailability of the necessary computing power required to solve the problem in higher dimensions to an accuracy sufficient to demonstrate the veracity of any new model. In developing a more generally valid two–dimensional model, we hope not only to gain better agreement with experimental results, but also to be able to suggest the best theoretical design for optimal results in clinical practice.

1.3 Outline

A brief history of the development of electrochemical oxygen sensors is given in Chapter 2, where we also describe the equations that we will use to model the operation of the currently used oxygen sensor, which is known as the Clark electrode [51]. Difficulties encountered in the practical use of the Clark electrode for *in vivo* monitoring are described, together with techniques developed in laboratory experiments to overcome them. Previous workers have favoured one–dimensional models with which we will eventually wish to compare our two–dimensional model, so that a brief description of these models is also given.

Before we consider parallel algorithms for the solution of the problem we first

describe, in Chapter 3, a method of treatment of the mathematical difficulty of a singularity caused by the boundary conditions of the problem. In Chapter 4 we introduce the parallel architecture with the transputer as its basic processor, and give a brief introduction to the programming language Occam. We generalise the work of Leland [56] on the parallel efficiency and effectiveness of elliptic partial differential equations to cover a wider class of problems including the parabolic partial differential equations of interest. This allows us to compare different numerical methods of solution of a simple test problem, before deciding upon the most appropriate method to use when modelling the Clark electrode. In Chapter 5, this work is combined with the singularity correction of Chapter 3 to obtain an efficient and accurate parallel solution of the equations governing the operation of the Clark electrode. The analysis of the complex interplay between the errors of the numerical finite difference solution and those of the singularity correction yields a set of conditions for a sufficiently accurate solution to be obtained. This is used in Chapter 6 to demonstrate the inadequacies of the one–dimensional models, and to study the operation of the Clark electrode as the governing parameters are varied. This allows us to go on to suggest a practical regime which, theoretically, can overcome each of the difficulties associated with the use of the Clark electrode for *in vivo* measurements. In the final chapter we discuss the lessons we have learnt from our experience of working with a parallel architecture, and make some suggestions for improvements from a user's point of view. Finally, we summarise our investigations into the operation and design of the Clark electrode, and indicate areas of future interest in the modelling of electrochemical sensors.

In the hope that the model will become more widely used in the investigation of the behaviour of the Clark electrode, we include copies of the text of some of the codes used. In Appendix A we give a brief description of the structure of the parallel code, the text of which is given in Appendix B. A version of the sequential Fortran code is given in Appendix C.

Chapter 2

Membrane–Covered Oxygen Sensors

Of the thousands of elements and compounds on which our lives depend, our continuing existence is most precariously reliant on oxygen. Cut off the oxygen supply, and our most important organ, the brain, will suffer irreparable damage within four minutes, rapidly followed by death. However, the body can neither feel, hear, taste nor smell oxygen. Our only built–in oxygen sensor is the excruciating pain we feel when certain tissues (cardiac, skeletal and smooth muscle) are deprived of their supply. In fact, as mountaineers will attest, the gradual onset of hypoxia is accompanied by a subtle, mind–numbing and potentially lethal euphoria.

In spite of this primary importance, prior to World War II the most reliable real–time indication of a patient's oxygenation status was the colour of their skin, since the only available means of measuring blood oxygen tension (P_{O_2}) was by removing all the gases from a blood sample via a vacuum pump. The introduction during the war of ear oximeters[1] improved the detection of hypoxia in pilots, but the limitations of these techniques led respiratory physiologists and clinicians to look for a direct means of measuring blood P_{O_2}.

2.1 Electrochemistry

The science of electrochemistry owes its birth to the strange discoveries of Luigi Galvani, an Italian physiologist. Whilst teaching anatomy in Bologna in about 1786, Galvani noticed that the legs of a dead frog twitched whenever a nearby electrostatic machine was revolving. He followed up this result by noticing the same effect whenever he touched the frog with a copper wire connected to an iron spike on which the frog was impaled. These discoveries led the physicist, Alessandro Volta, to observe that the twitching was very marked if two different metals were used, but nothing happened if they were the same. By experimenting with different metals, Volta was able to arrange them in an electromotive series, leading to his construction of the first battery in 1796 from sheets of silver and zinc separated by wet cloth. Nicholson and Carlisle, in 1800, connected wires to

[1]The colour of blood is a function of oxygen concentration, the oxygen–rich arterial blood being red, the oxygen–depleted veinous blood, blue. This colour change is due to the optical properties of the haemoglobin molecule which change as the blood deoxygenates, becoming increasingly less permeable to red light. By measuring the 'redness' of a patient's blood, the oximeter gives an indication of the oxygen saturation.

the poles of Volta's battery to demonstrate the dissociation of water into hydrogen and oxygen. It was not until 100 years later, at the suggestion of Walther Nernst, that Heinrich Danneel [15] demonstrated that the reverse reaction was also possible, i.e. oxygen in solution could be electrolysed to the hydroxyl ion. Using two large platinum electrodes with a potential difference of 20mV between them, Danneel also showed that the current obtained was directly proportional to the dissolved oxygen concentration. However, his attempts to use his discovery to measure the oxygen concentration in biological media were thwarted either by protein deposition or other poisoning of the cathode surface[2].

2.2 Electrochemical Oxygen Sensors

The first electrochemical oxygen sensor to be successfully used in biological media was the dropping mercury electrode (DME) developed by Jaroslav Heyrovsky [41] in Prague in the early 1920's. Heyrovsky happened upon his discovery by chance whilst investigating irregularities in the capillarity of mercury, which involved allowing the mercury to flow through a capillary and weighing the drops. Since the process was so slow, he decided to put the glass capillary in a solution with a potential difference between the mercury in the capillary and that collecting at the bottom. He noticed that in the presence of a reducible species, a current flowed in the system proportional to the concentration of the species, and therefore designed a device capable of plotting the relationship between the applied voltage and the current. It was built from an old phonograph motor which simultaneously turned a potentiometer and moved a photographic film behind a slit lit by a galvanometer mirror. Heyrovsky called his device a *polarograph*, and the electrochemical measurement of gases is still often referred to as *polarography*. Since the surface of each mercury drop acts as the cathode and is renewed with each drop, the problem of protein deposition is virtually eliminated. Heyrovsky's students could therefore use the polarograph to obtain oxygen concentrations in biological media, and by 1938 it had been used by Baumberger [5] to determine the P_{O_2} of blood plasma. However, the DME cannot be used *in vivo*, and by the time the oxygen concentration of anaerobically separated plasma from a patient has been measured, it might be too late.

Modern electrochemical oxygen sensors are descended from the platinum electrodes which were used by Davies, Bronk and Brink to analyse oxygen concentration in tissues at the University of Pennsylvania during World War II. Bare metal electrodes implanted in tissue are kept clean by scavenging white blood cells, so that Brink and Davies were able to build a nerve respirometer from small platinum wires, capable of making local measurements of oxygen metabolism and P_{O_2} in the mammalian cerebral cortex [16]. By covering the cathode with collodion, a

[2]A *reduction* reaction is one in which a species gains one or more electrons: $A + e^- \rightarrow A^-$, whilst in *oxidation*, a species loses one or more electrons: $B \rightarrow B^+ + e^-$. The electrode at which reduction takes place is termed the *cathode*, and that at which oxidation takes place the *anode*.

semi–permeable membrane permeable to both ions and oxygen, Davies and Brink were able to obtain more dependable results, particularly for electrodes embedded in tissue for long periods. However, such electrodes are still poisoned when trying to measure the P_{O_2} of whole blood.

The solution to the problem was provided by Leland Clark [51] between 1954 and 1956. Clark had invented the first blood bubble oxygenator for human use, which oxygenates the blood outside the body, allowing the heart and lungs to be bypassed and open heart surgery to take place. Clark found that his invention was so efficient that over–oxygenation became a worry, and since the haemoglobin became totally saturated, oximeters became useless in monitoring the P_{O_2}. Clark knew of the work of Davies and his colleagues and realised that the problems of protein deposition on bare electrodes might be overcome by protecting the cathode with a membrane permeable only to oxygen. From his work in oxygenation, he knew of several suitable materials. He polished to a flat surface a platinum bead sealed in the end of a glass rod, tied a piece of cellophane over the end, and placed a drop of potassium chloride solution as an electrolyte layer between the two. The result was the first membrane covered oxygen sensor or *Clark electrode.* Sensors of this type are still the main means of blood P_{O_2} measurement in clinical medicine. Since their invention, use of Clark electrodes has been extended to several areas of clinical research. These include transcutaneous oxygen monitoring in babies, haemoglobin dissociation and molecular genetics. They are also used in industries as diverse as sewage treatment, soil chemistry and beer and wine production.

2.3 The Clark Electrode

The modern version of the Clark–type oxygen sensor is shown in Figure 2.1. The cathode is usually made from an unreactive noble metal such as platinum or gold. If the reference anode is made from a more basic metal such as zinc or lead, then there will be a sufficient difference between the electrode potentials for spontaneous oxygen reduction to occur. More commonly, the anode is also made from an unreactive metal (such as silver/silver chloride) and an external voltage source is required before reduction takes place. The two types are referred to as *galvanic* and *non–galvanic* respectively, and in both, the electrode assembly is immersed in aqueous electrolyte solution and protected from poisoning by a tightly stretched plastic membrane which is permeable to oxygen.

The simplest form of the overall chemical reaction that takes place at the cathode when oxygen is reduced is[3]

$$O_2 + H_2O + 4e^- = 4OH^-, \tag{2.1}$$

which sets up an oxygen concentration gradient within the electrolyte, allowing oxygen to diffuse towards the cathode. By increasing the potential between the

[3]Fatt [21] describes this as the most likely reaction, although it can be the result of a more complicated series of reactions.

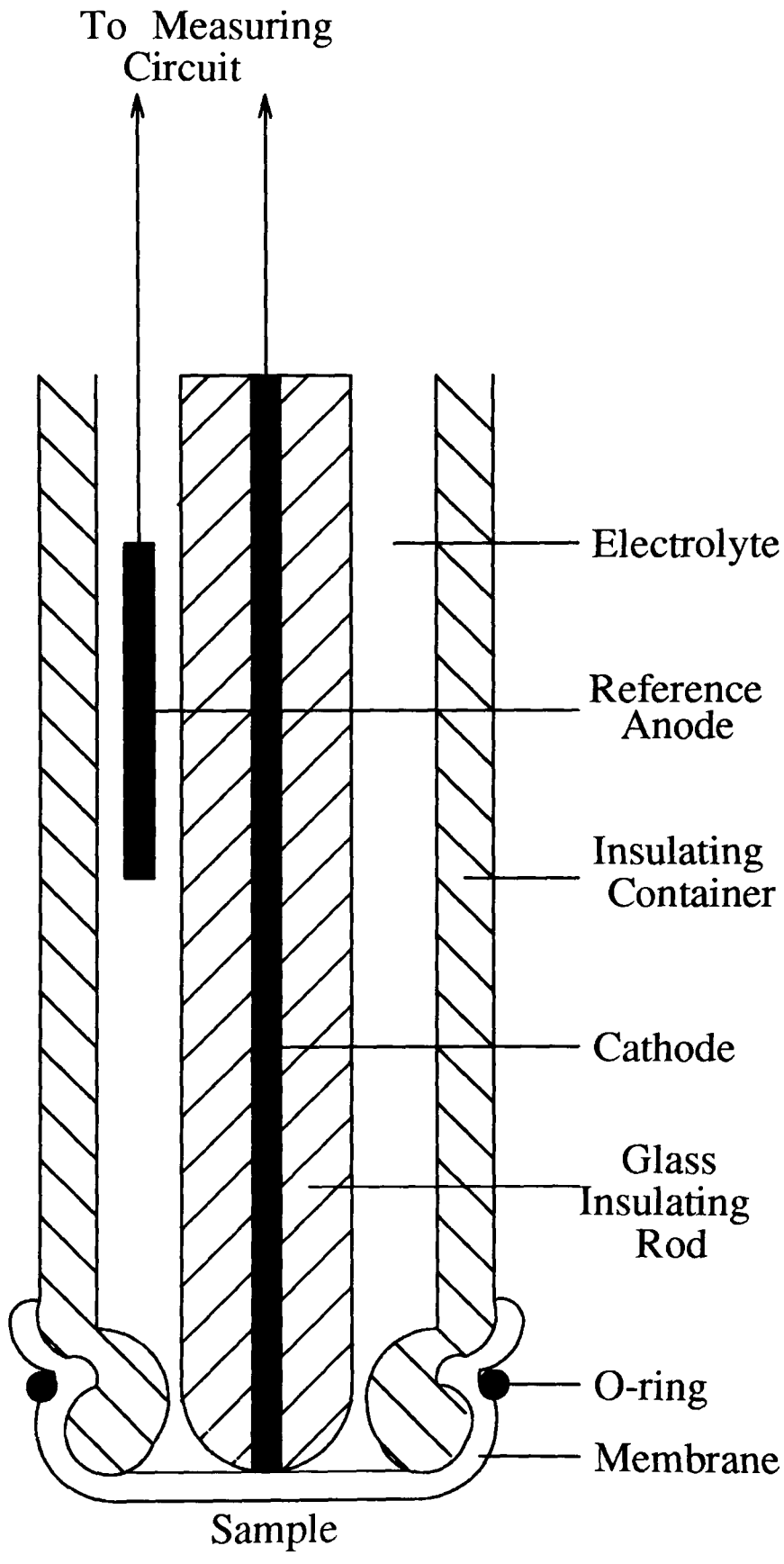


Fig. 2.1: Schematic outline of the main features of the membrane–covered oxygen sensor or *Clark electrode*.

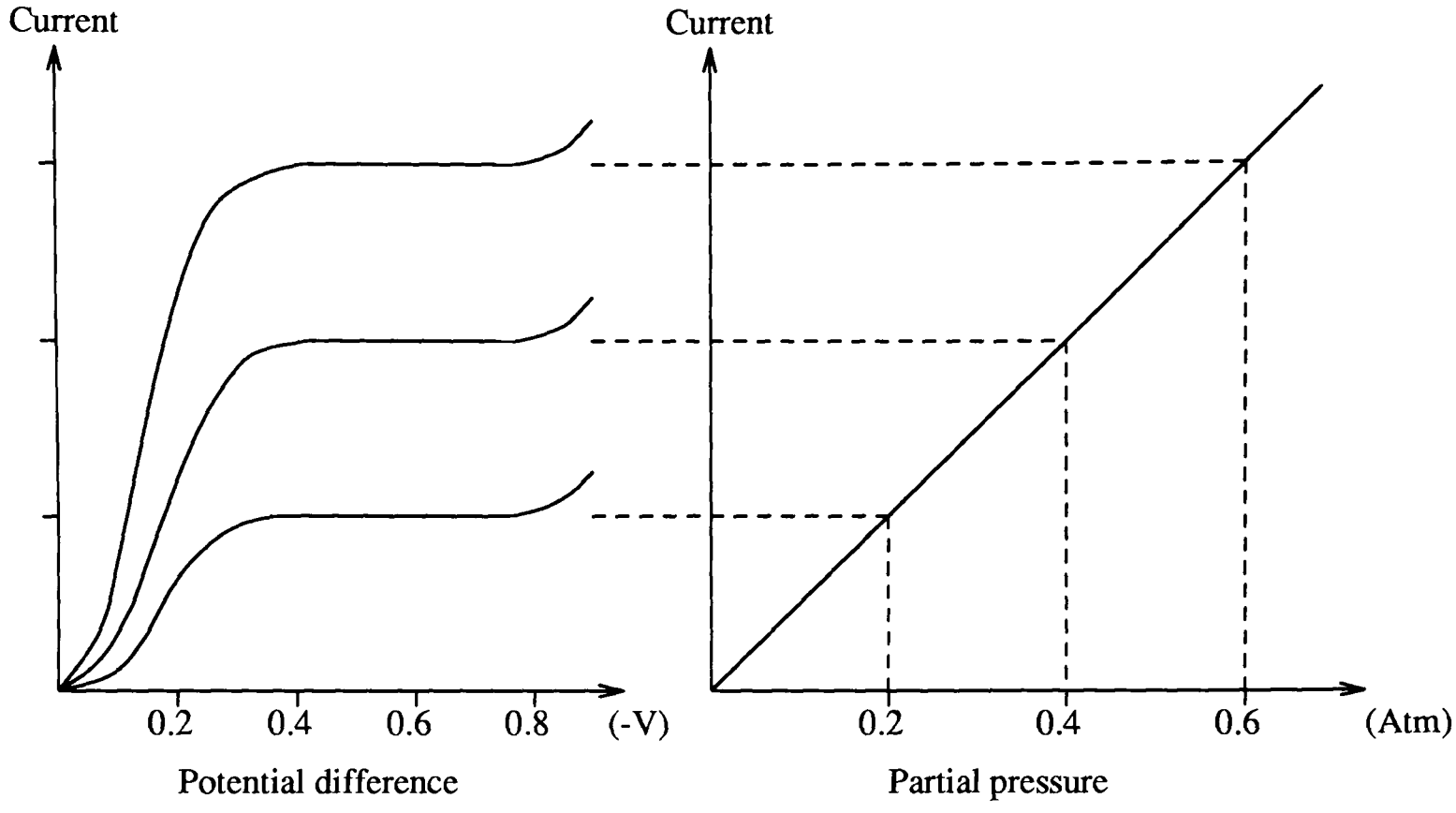


Fig. 2.2: Idealised current against voltage plot at various oxygen concentrations, and typical polarising voltage. The size of the measured current is dependent on several factors including cathode size, and membrane type and thickness, but is typically of the order of tens of nA.

electrodes, we can increase the current until all oxygen reaching the cathode is reduced. The current is now diffusion–rate limited and no further current increase takes place until the potential is sufficiently high for other electrode reactions to begin, giving another sharp rise in the current. Figure 2.2 shows a typical current against voltage plot for various oxygen concentrations. By fixing the potential in the diffusion limiting region, we obtain a measured current which is *directly proportional to oxygen concentration.*

2.4 Mathematical Analysis

Although such electrodes have been in use for over thirty years, the mathematical analysis of the dynamic behaviour of membrane–covered oxygen sensors (which we will in future refer to simply as the Clark electrode) has largely been confined to one–dimensional models. The first of these was due to Mancy *et al* [69], who were interested in modelling a galvanic oxygen sensor that they had built, based on the Clark electrode, for measuring P_{O_2} in natural waters and wastes. Their analysis (which contained some errors) applied only to large sensors for which the diffusion of oxygen could be considered one–dimensional and axial. It has since been corrected [39, 42, 73, 86] and many attempts have been made to apply a similar theory to the much smaller sensors used for analysing blood P_{O_2} [19, 20, 36, 37, 39, 75, 74, 83, 97]. In attempting to explain the differences between the currents predicted by the one–dimensional theory and those obtained

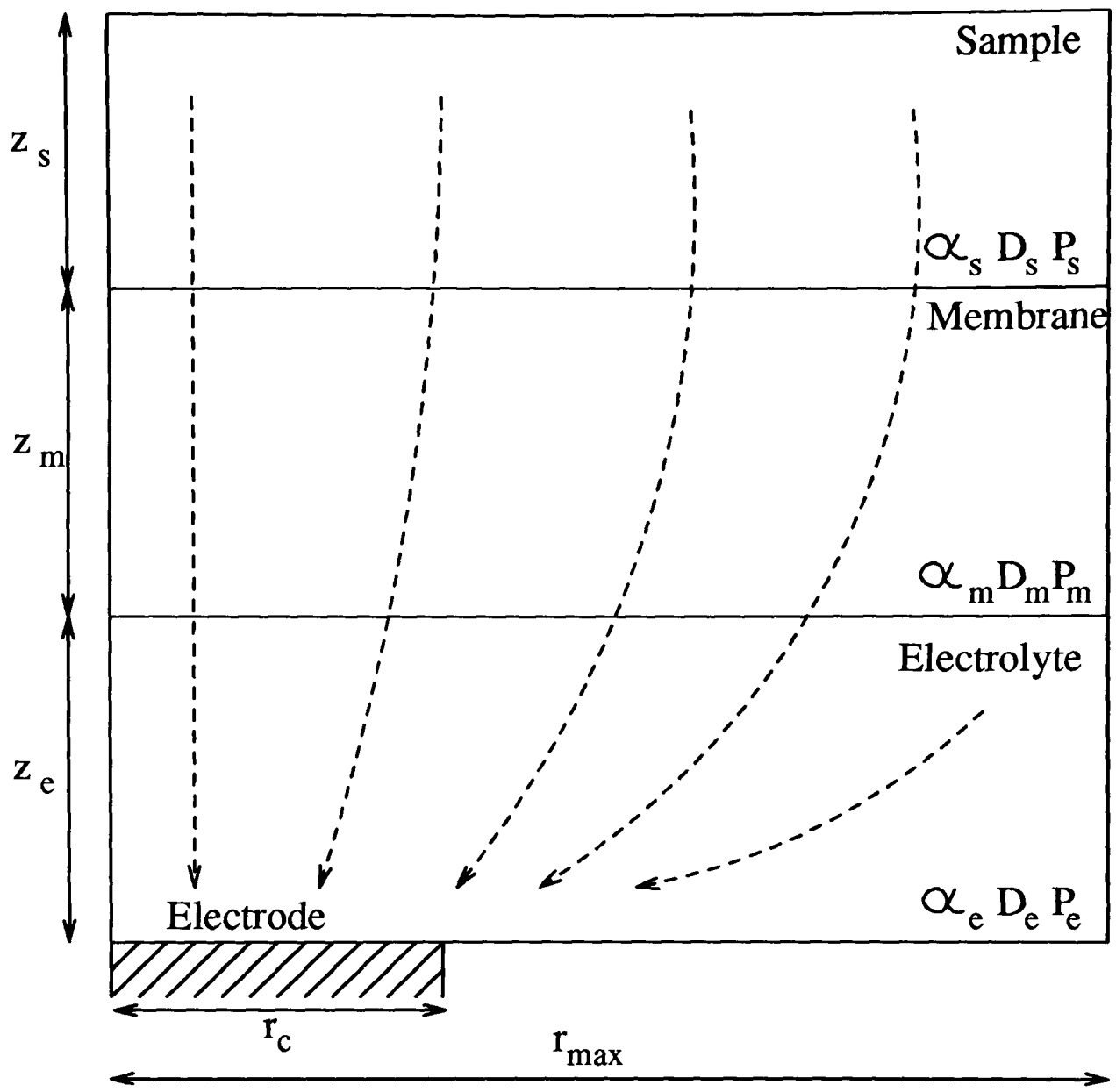


Fig. 2.3: The finite two–dimensional region in which cylindrical diffusion to the electrode takes place.

by experiment, various authors have extended Mancy's work to cover purely radial diffusion [63, 64, 94], and pseudo two–dimensional diffusion [48, 54].

Since the early papers of Clark and Mancy, a large number of articles have appeared concerning the theory and practice of oxygen sensors. Comprehensive reviews can be found in the books of Hitchman [42], Linek *et al* [64], Fatt [21], Gnaiger and Forstner [29], and in the articles by Grasshopf [47], Hoare [44], Kreuzer and Kimmich [55], and Hahn [35, 36, 37].

2.4.1 The two–dimensional model

Since oxygen sensors are used in practice with a suitably negative potential to give diffusion–limited current, we can treat the cathode as an oxygen sink and assume that all oxygen molecules reaching it will be reduced. The resulting concen-

tration gradient allows the transport of oxygen to the cathode to be modelled by the diffusion equation. Since one–dimensional models have been found to grossly underestimate the current for the practical cathode sizes used in blood–oxygen sensors (circa $10\mu m$ – $100\mu m$ diameter), the sensor is modelled as a disc cathode lying in the surface of an infinite planar insulating material, giving us an axial symmetry about the z–axis, so that we can reduce our problem to one in two–dimensional cylindrical coordinates as shown in Figure 2.3. Above the cathode, we have a thin film of electrolyte, separated from the medium to be measured by a thin, semi–permeable membrane, and we will model this as the three layers:

(a) An electrolyte layer (e) of thickness z_e. The thickness is not easy to measure, but is typically between $2\mu m$ and $10\mu m$, and the effect of varying this thickness is of practical interest;

(b) A membrane layer (m), thickness z_m typically between $5\mu m$ and $25\mu m$, which is porous to oxygen but has a diffusion coefficient at least an order of magnitude lower than that of the electrolyte;

(c) A layer of sample (s) taken to be infinitely thick, with diffusion coefficient of the same order of magnitude as that of the electrolyte.

The oxygen partial pressure p is related to the concentration c of oxygen via Henry's law $c = \alpha_l p$, where $l = e, m, s$ and α_l is the solubility in the relevant layer, so that it also obeys the cylindrical diffusion equation. In addition the partial pressure enjoys the advantage of being continuous across the material interfaces between the layers, and we will therefore use it in preference to the concentration. Within each layer we assume that the partial pressure $p(r, z, t)$ is governed by the cylindrical diffusion equation,

$$\frac{\partial p}{\partial t} = D_l \left(\frac{1}{r}\frac{\partial p}{\partial r} + \frac{\partial^2 p}{\partial r^2} + \frac{\partial^2 p}{\partial z^2} \right) \tag{2.2}$$

where $l = e, m, s$ and D_l is the diffusion coefficient in the relevant layer. Since it is assumed that all oxygen reaching the cathode is reduced, the current is directly proportional to the oxygen flux into the disc area and is given by [42]

$$i(t) = 2\pi n F P_e \int_0^{r_c} \left(\frac{\partial p}{\partial z} \right)_{z=0} r dr. \tag{2.3}$$

Here F is the Faraday constant, P_e is the permeability in the electrolyte (related to D_e through $P_e = \alpha_e D_e$), and n is the number of electrons per molecule transferred in the reduction reaction.[4] If the sample is stirred sufficiently vigorously then we can assume that the partial pressure at the outer surface of the membrane is continually replenished, allowing us to use a simpler, two–layer model. However in

[4]We will take n to be 4, which is equivalent to assuming that the reaction at the cathode can be wholly represented by the simple reaction 2.1, which is a single four electron step. In practice the reduction of oxygen also has another mechanism consisting of two two–electron steps with H_2O_2 as an intermediate, and n therefore lies between 2 and 4 (see for example Linek *et al* [64]).

practice, oxygen sensors often give a reduced response when used for long periods [35, 98] due to depletion of the sample. One of our aims is to obtain the conditions under which this depletion can be minimised, so that we assume that oxygen depletion takes place within all three layers. To render the problem finite it is further assumed that there exist distances $r_{\max}$, $z_{\max}$ $(= z_e + z_m + z_s)$ (effectively at infinity) beyond which no further depletion takes place, which can be determined by numerical experiment. We therefore wish to obtain a numerical solution of equation (2.2) within each layer for the dimensionless partial pressure $p' = p/p_0$, where p_0 is the bulk partial pressure, starting from the initial condition

$$t < 0 \quad p'(r,z,t) = 1 \quad \text{in } [0, r_{\max}] \times [0, z_{\max}]. \tag{2.4}$$

The boundary conditions on $z = 0$ are dependent on the mode of operation of the sensor. In an attempt to overcome the problem of sample depletion, the cathode can be pulsed i.e. a potential difference sufficient to reduce oxygen at the cathode can be applied for a brief period t_{on}, during which the current is measured, then the potential difference is removed, usually for a longer period t_{off}, and the system begins to relax back to its original state before the next pulse is applied. The boundary conditions on $z = 0$ are therefore

$$\begin{aligned} p' &= 0 \quad (0 \le r \le r_c) \\ \frac{\partial p'}{\partial z} &= 0 \quad (r_c < r \le r_{\max}) \end{aligned} \tag{2.5}$$

when a sufficient potential difference is applied to the cathode for oxygen to be reduced, and

$$\frac{\partial p'}{\partial z} = 0 \qquad (0 \le r \le r_{\max}) \tag{2.6}$$

when it is removed. The conditions on the other boundaries are given by

$$\begin{aligned} &\text{on } r = 0 && \frac{\partial p'}{\partial r} = 0 \quad (0 \le z \le z_{\max}) && \textit{(symmetry)} \\ &\text{on } z = z_{\max} && p' = 1 \quad (0 \le r \le r_{\max}) && \textit{(infinity)} \\ &\text{on } r = r_{\max} && p' = 1 \quad (0 \le z \le z_{\max}) && \textit{(infinity).} \end{aligned} \tag{2.7}$$

In addition we have internal conditions at the material interfaces that oxygen partial pressure and flux are equal,

$$\begin{aligned} &\text{on } z = z_e && p'_{z=z_{e-}} &&= p'_{z=z_{e+}} \\ & && P_e \left(\frac{\partial p'}{\partial z}\right)_{z=z_{e-}} &&= P_m \left(\frac{\partial p'}{\partial z}\right)_{z=z_{e+}} \\ &\text{on } z = z_e + z_m && p'_{z=z_e+z_{m-}} &&= p'_{z=z_e+z_{m+}} \\ & && P_m \left(\frac{\partial p'}{\partial z}\right)_{z=z_e+z_{m-}} &&= P_s \left(\frac{\partial p'}{\partial z}\right)_{z=z_e+z_{m+}}. \end{aligned} \tag{2.8}$$

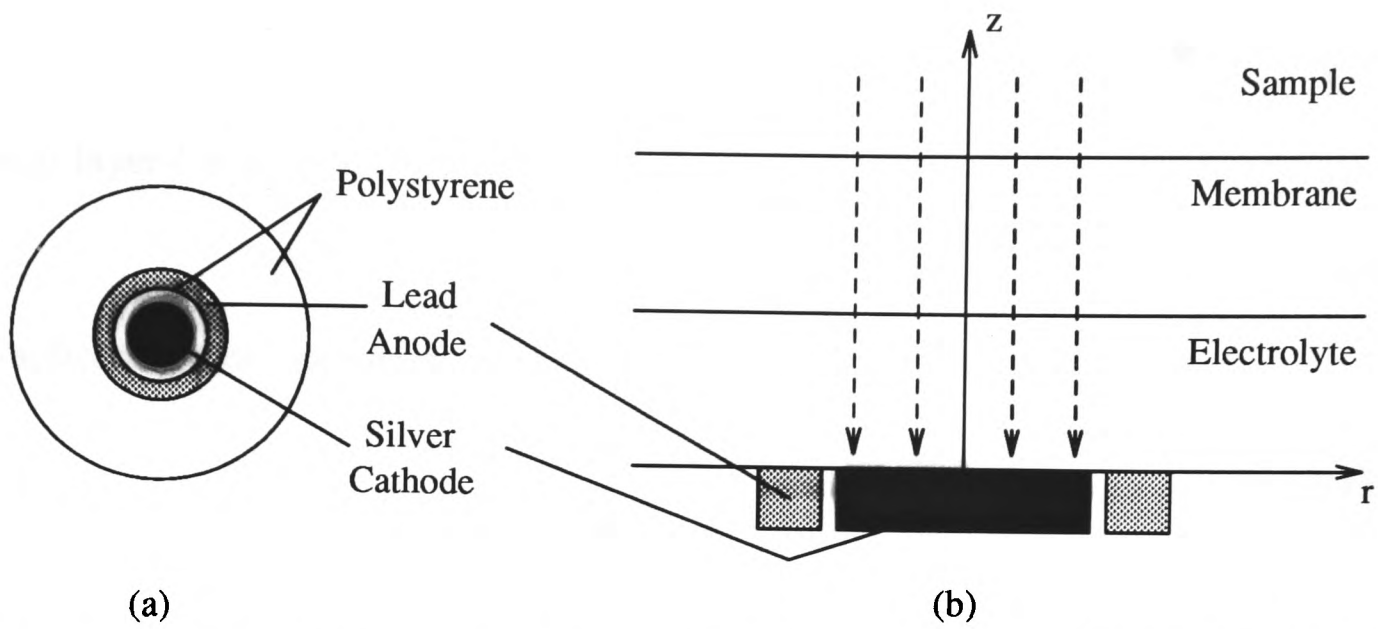


Fig. 2.4: (a) The electrode configuration of the Mancy sensor and (b) Linear diffusion parallel to the z–axis only.

Whenever a potential difference exists between the electrodes so that oxygen is being reduced at the cathode, the boundary conditions on $z = 0$ ensure that there is a discontinuity in the first derivative of p' at the point $(r_c, 0)$. This has long been known to cause problems when trying to obtain a numerical solution by standard finite difference techniques, since the difference approximations used to replace the partial derivatives are based on the first three terms of the Taylor expansion of the solution, and are therefore valid only for a continuously differentiable function. The problem was first described by Motz [72], whose work has since been extended by Fox and Sankar [26], Crank and Furzeland [12, 13] and Bell and Crank [6]. We will describe in the next chapter a method of overcoming these problems by combining the approaches of these authors, which will allow us to obtain a sufficiently accurate solution to the two–dimensional problem. Since we will then wish to compare our results to those obtained from the one–dimensional models of previous workers, we will briefly describe these models in the next section.

2.4.2 One–dimensional linear diffusion

Figure 2.4(a) shows the electrode assembly used by Mancy, Okun and Reilly. It consisted of a large ($0.3cm$ radius) silver cathode surrounded by an annular lead anode, the cathode being sufficiently large to give negligible radial diffusion and minimise what they termed the "edge effect". This allows them to model the operation of their sensor using the one–dimensional equation for diffusion parallel to the z–axis only (which we will refer to as *linear diffusion*), as shown in Figure 2.4(b).

The equation for linear diffusion is given by

$$\frac{\partial p}{\partial t} = D_l \left(\frac{\partial^2 p}{\partial z^2} \right) \tag{2.9}$$

in each layer $l = e, m, s$, with initial condition

$$t < 0 \qquad p(z,t) = p_0 \quad (0 < z \leq z_{\max}) \tag{2.10}$$

and subject to the boundary conditions

$$\begin{aligned} p &= 0 \quad \text{on } z = 0 \\ p &= p_0 \quad \text{on } z = z_{\max}. \end{aligned} \tag{2.11}$$

The interface conditions are again given by equations (2.8).

Since the general solution is extremely complicated, Mancy *et al* used Laplace transform methods to obtain solutions to the above equations under certain simplifying assumptions, including that of constant stirring of the sample so that the sample layer can be ignored. This allowed them to give solutions in the form of infinite series for the three cases:

$$\frac{z_e}{\sqrt{D_e}} >> \frac{z_m}{\sqrt{D_m}}, \quad \frac{z_e}{\sqrt{D_e}} << \frac{z_m}{\sqrt{D_m}}, \quad \frac{z_m}{\sqrt{D_m}} = \frac{z_e}{\sqrt{D_e}}, \tag{2.12}$$

which correspond to the characteristic times for diffusion through the electrolyte layer being very much greater than, very much less than and equal to that of diffusion through the membrane. The derivation of the simplified solutions involved several pages of complex algebra and, not surprisingly, contained several errors. These, together with some confusions of notation, have since been discussed and corrected by many authors (see, for example, Short and Shell [86]). More recently, Myland and Oldham [73] have given a more comprehensive solution to the two–layer linear diffusion equations which gives a clear account of both the analysis and the parameters used, and claims to be an exact treatment for the time–transient current under certain less limiting assumptions.

Since no analytic solution is presently available for the three–layer linear model we will obtain a numerical solution to the linear diffusion case using the implicit Crank–Nicholson scheme [14], and use the analytic treatment for linear diffusion in two layers given by Myland and Oldham to check for agreement with our numerical solution. When comparing linear diffusion to two–dimensional cylindrical diffusion in the three–layer model we will quote results obtained using our one–dimensional numerical code.

2.4.3 One–dimensional spherical diffusion

The theoretical case of one–dimensional spherical diffusion (i.e. purely radial diffusion to a hemi–spherical cathode) has been studied in the steady–state by Firouztale, Skerpon and Ultman [94, 22]. The time–dependent case has been the subject

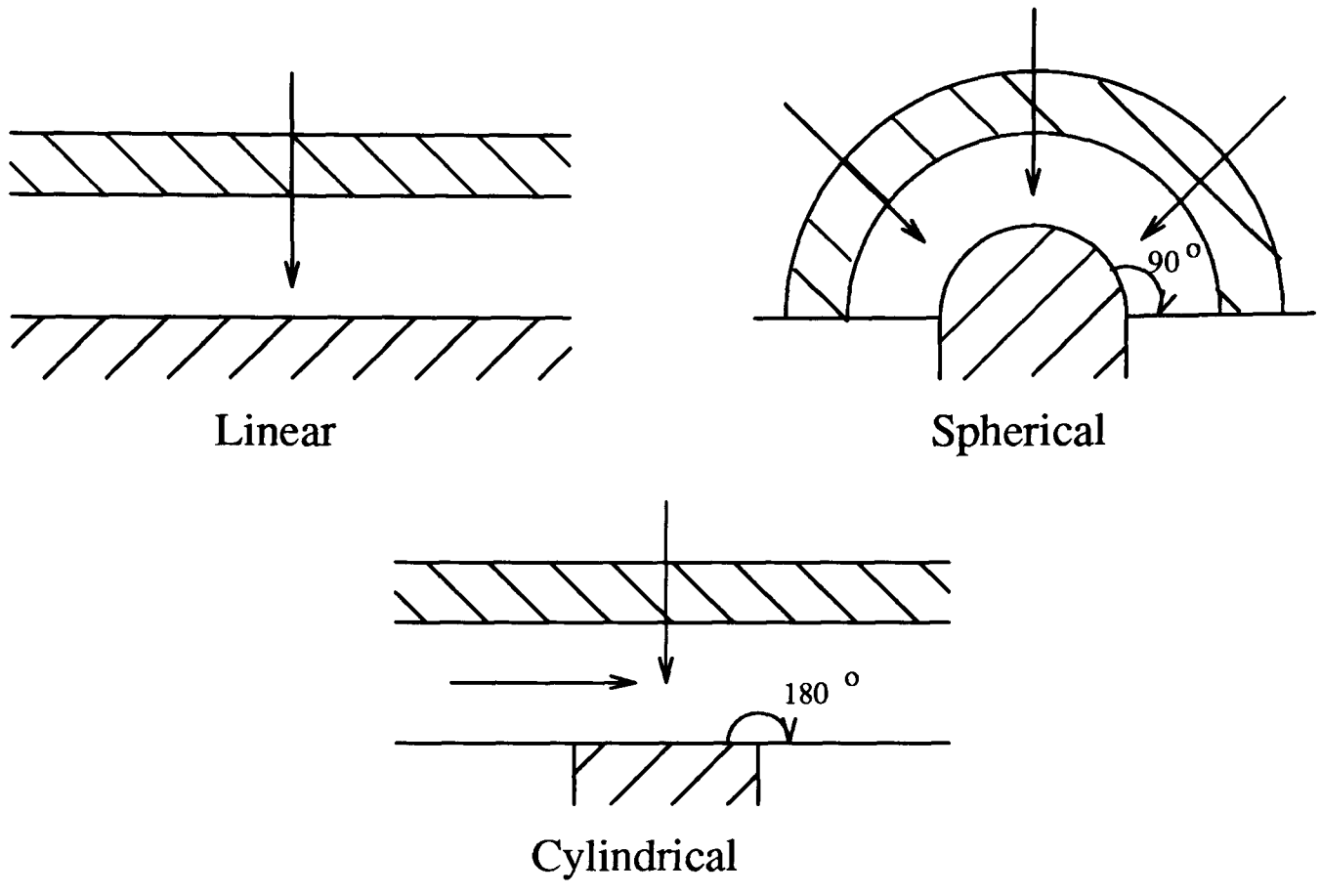


Fig. 2.5: Possible directions of diffusion for the linear, spherical, and cylindrical models. The two–dimensional cylindrical model has a singularity at the edge of the cathode.

of several papers and a book by Linek *et al* [63, 64] who consider a three–layer model of diffusion to a hemi–spherical cathode, but use the initial condition of zero P_{O_2} in both the membrane and electrolyte layers. We cannot therefore use their work for comparison with our solution for an electrode with all three layers initially in equilibrium, which is the more usual practical case when measuring oxygen concentrations in blood. Also, since we are dealing with a flat disc cathode, the nature of the boundary conditions at the electrode edge are different, all three cases being shown in Figure 2.5. The linear model assumes an infinite cathode and has a continuous boundary condition on $z = 0$, the spherical model has a 90^o angle between the insulation boundary and the cathode, and so avoids the singularity at the junction of the cathode and the insulating surface of the cylindrical model, which has a continuous boundary on $z = 0$.

The equation governing diffusion to a (hemi–)spherical cathode surrounded by a (hemi–)spherical membrane is

$$\frac{\partial p}{\partial t} = D_l \frac{1}{r^2} \frac{\partial}{\partial r} \left(\frac{1}{r^2} \frac{\partial p}{\partial r} \right) \tag{2.13}$$

in each layer $l = e, m, s$, with initial condition

$$t < 0 \qquad p(r,t) = p_0 \quad (0 < r \leq r_{\max}) \tag{2.14}$$

and subject to the boundary conditions

$$\begin{aligned} p &= 0 \quad \text{on } r = r_c \\ p &= p_0 \quad \text{on } r = r_{\max}. \end{aligned} \tag{2.15}$$

The interface conditions are again given by equation (2.8), but with r replacing z. The method of solution used by Linek *et al* is again Laplace transforms, although they give no details of the algebra. However, since the infinite series defining the time–dependent current is in the form

$$i(t) = 1 + C_1 \sum_{n=1}^{\infty} \frac{\alpha_n^2 \exp -C_2(\alpha_n^2 t)}{Q(\alpha_n)}, \tag{2.16}$$

for C_1, C_2 constants, where the α_n are the positive roots of a nonlinear equation with about 20 terms, and $Q(\alpha_n)$ is a nonlinear equation with about 40 terms, we imagine the analysis to be non–trivial.

Since these authors claim that the spherical model is a good approximation to the cylindrical model for sufficiently small cathodes, we will again obtain a numerical solution to equations (2.13)–(2.15) using a Crank–Nicholson type scheme, and compare it to the results of the two–dimensional model.

2.5 Practical Difficulties in Using the Clark Electrode

There are several practical difficulties, both *in vitro* and *in vivo*, encountered when attempting to measure the P_{O_2} of blood using the Clark electrode. In the case of a continuously applied potential where the current measurement is made at "steady–state", the primary problem is the depletion of oxygen in the sample by the reduction processes at the cathode. In *in vitro* measurements we have a sample of finite size, and whilst the problem can be overcome to some extent by stirring the sample, some oxygen will clearly be removed by the measuring process. In commercial *in vitro* analysers therefore, small cathodes of around $10\mu m$ radius are used to minimise the amount of oxygen removed, and further allowances are made by estimating an appropriate "correction factor" (Hahn [37])[5].

In *in vivo* measurements, the practical situation for a continuously applied potential is unclear. If there is a sufficient rate of blood flow within the artery for negligible reduction in the oxygen content at the external face of the membrane, then the diffusion process should reach a steady–state. However in practice this is found not to occur, and measurements are blood–flow dependent.

It was suggested as early as 1960 by Evans and Naylor [19, 20, 75, 74] that the problems of sample–depletion and blood flow dependence might be minimised by operating the Clark electrode in pulsed mode. The intention is to limit the diffusion

[5]The assumption that under a continuously applied potential the Clark electrode will reach a *steady–state* also assumes that there is sufficient motion or mixing within the sample for any oxygen removed by the electrode to be replenished. If this is not the case, then the current will continue to decrease with increasing time and, for a finite sample, will eventually reach zero when all the oxygen initially present has been reduced.

processes almost entirely to the electrolyte and membrane layers by applying the potential for a sufficiently short period, before removing it and allowing the system to relax back towards its initial state. Again this is further aided by making use of small cathodes of the order of $10\mu m$ radius. The advantages of the technique over waiting for the steady–state are clear: since the current is measured after a short time it is comparatively large; by using a sufficiently short pulse we remove only a fraction of the oxygen; and the response time to a step change in sample P_{O_2} can be reduced. As a result many workers have published laboratory results suggesting the most appropriate pulsing regime and electrode geometry (which we will summarise in Chapter 6). However, these suggestions vary widely and theoretical analysis has been limited to simple extrapolations from the one–dimensional models for a continuously applied potential. Despite its advantages, therefore, the pulsing technique is still not used in blood–oxygen measurements in clinical practice.

2.6 Modelling Objectives

The *current sensitivity* is the defining quantity in the operation of a particular design of electrode, and is usually defined as the current obtained per unit atmosphere of O_2 at steady–state, and is typically of the order of tens of nA[6]. Our main objective in modelling the Clark electrode will be to investigate the time–dependent development of the diffusion processes and the current sensitivity, in the hope of gaining a sufficient understanding of the operation of the electrode as the design parameters are altered, to be able to suggest ways of overcoming the practical difficulties in its use. The various theoretical models that have been developed for the operation of the Clark electrode are not simply an attempt to explain the current sensitivity obtained in experiments. This theoretical research has often been funded by manufacturers, since a reliable model which will accurately predict the measured current of a device with, say, a smaller cathode radius or a membrane of higher permeability, will save them the expense of producing and testing prototypes experimentally. We have listed below those design parameters which can be considered as variable. Our main objective is to determine the relationships between these and the current sensitivity as a function of time when a constant potential difference is applied, and the extent and nature of the sample depletion:

(1) Cathode radius (r_c)
(2) Electrolyte layer thickness (z_e)
(3) Membrane thickness (z_m)
(4) Membrane permeability (P_m)
(5) Membrane diffusivity (D_m).

[6]We will more often use the term current sensitivity to refer to the calculated current *per unit area of cathode per unit atmosphere* at a particular time after the potential is first applied, but the meaning will always be clear from the context.

Having determined these relationships, we will use them to demonstrate where the various one–dimensional models are inadequate.

In practical measurements in blood, it would clearly be advantageous to use the Clark electrode in the pulsed operation mode defined by equations (2.5) and (2.6). We will therefore attempt to use our understanding of the variation of the time–transient current with the above parameters to enable us to suggest the optimal theoretical geometry and pulsing regime, which combines blood flow independence with a rapid response time to changes in blood P_{O_2}.

Chapter 3

Numerical Solution for an Unshielded Electrode

In the previous chapter we mentioned the problems caused by the boundary singularity at the cathode edge when using standard finite difference techniques. These are not "serious" (Fox and Sankar [26]), since the true solution of the problem is perfectly "well–behaved" at all points in the solution region. It does, however, result in inaccuracies in the numerical solution in the neighbourhood of the singularity. In order to be able to tackle the problem without the additional difficulties introduced by the inclusion of the membrane, we first consider the case of an unshielded disc electrode, which in itself is an electrochemical problem which has received much interest .

The method of treatment described in this chapter was first suggested by Motz [72] for use in elliptic problems. It essentially uses the finite difference solution at points sufficiently distant from the singularity to be comparatively unaffected by it, to calculate the constants in a truncated series solution about the singular point. This approach was used by Crank and Furzeland [12, 13] to solve the electrode problem in the steady state. Their suggestions for the treatment of the time–dependent case will be combined with the work of Bell and Crank [6], who solved the time–dependent problem in Cartesian coordinates, to obtain a time–dependent solution for the electrode problem.

3.1 Previous Work

Modern measurements of oxygen tension in living tissues are usually carried out using *microelectrodes*, since electrodes used *in vivo* should be smaller than the unit size of the tissue of interest [2]. One of the most popular shapes both because of its ease of construction [7] and ease of insertion *in vivo*, is the circular disc electrode embedded in a coplanar insulating material, as illustrated in Figure 3.1. Such microelectrodes are also of more general interest in electrochemistry because of their low sample depletion, relatively small capacitive currents and geometry which achieves steady state current at short times [2]. The theoretical analysis of the time–transient current to unshielded disc electrodes has therefore received much attention in its own right, by a variety of analytical and numerical techniques

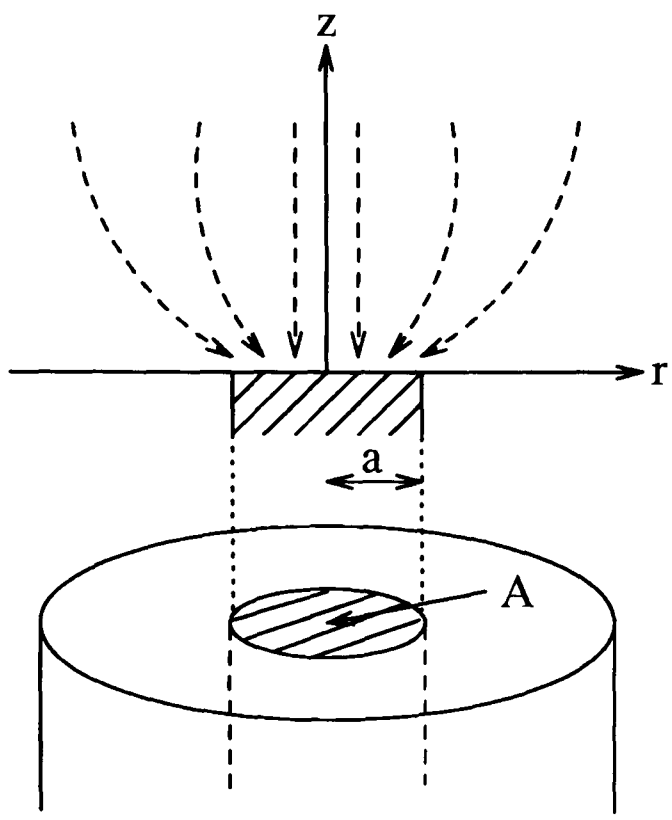


Fig. 3.1: Diffusion processes at an inlaid disc electrode.

[2, 3, 7, 23, 24, 40, 88, 89, 90, 92].

The analytic solution of the steady–state problem first given by Saito [81], has since been extended to cover a wider variety of electrochemical problems by Bond *et al* [7] and Oldham [77], and has been solved numerically by Crank and Furzeland [12] who obtained very good agreement with the analytic solution by correcting for the singularity.

No generally valid analytic solution to the time–dependent problem has been given. Cottrell [10], in 1903, solved the simpler time–dependent problem of semi–infinite linear diffusion to a planar electrode of area A using Laplace transforms. He obtained the error function expression

$$c(t) = c_0 \mathrm{erf}\left(\frac{z}{2\sqrt{Dt}}\right) \tag{3.1}$$

for the the concentration, $c(t)$, where c_0 is the bulk concentration, D the diffusion coefficient of the electroactive species (in our case, oxygen), and z the perpendicular distance from the electrode surface. He then gave a simple expression for the current as

$$i(t) = nAFD\left(\frac{\partial c}{\partial z}\right)_{z=0} = \frac{nAFc_0D^{\frac{1}{2}}}{\pi^{\frac{1}{2}}t^{\frac{1}{2}}}, \tag{3.2}$$

where n is again the number of electrons transferred in the reduction reaction and F the Faraday constant. The simplicity of this expression has appealed to more recent workers ([3, 2, 40, 76, 89]) (not least because it implies that both n and D for any electroactive species can be calculated from a single current–time plot [52, 53]),

and solutions to the two dimensional equations (both analytic and numerical) have usually attempted to augment equation (3.2) with an extra term or power series to explain the "edge effects" [3]. The most widely quoted are the analytic results of Aoki and Osteryoung [3], who obtained a series expansion for large t by taking the Laplace transform of the two–dimensional diffusion equation in polar coordinates to which they applied the Kantorovich–Lebedev transformation, allowing them to use the Wiener–Hopf method to obtain a power series in $t^{-\frac{1}{2}}$ for the current. At short times, they obtained an asymptotic series expansion in $t^{\frac{1}{2}}$ for the current by again applying the Wiener–Hopf method to the Laplace transform of the two–dimensional cylindrical diffusion equation after applying a Hankel transformation. This highly sophisticated solution was marred by a mathematical error which was corrected by Shoup and Szabo [88], who give an expression for the current which they claim is accurate to 0.6% at all times.

Solutions obtained by numerical techniques [40, 53, 88, 89, 90, 92] have relied upon the use of fine meshes to overcome to some extent the inaccuracies caused by the presence of the the singularity. Heinze [40] recognises the difficulty and uses a special treatment in calculating the flux at the electrode edge, but makes no attempt to correct concentration values in the neighbourhood of the singularity. More recently, Taylor *et al* [92] have used local mesh refinement in the neighbourhood of the singularity. Below we describe the numerical method that we use in the bulk of the solution region and then derive an analytic treatment in the region of the singularity. A robust method of combining these two approaches allows us to obtain a very accurate and economical solution method which agrees well with previous work.

3.2 Mathematical Formulation of Problem

We wish to model the diffusion of oxygen in an electrolytic solution to a circular disc electrode at which it is assumed all oxygen will be reduced. The axial symmetry of the problem allows us to use the two–dimensional diffusion equation in cylindrical coordinates i.e. we wish to solve

$$D\left(\frac{\partial^2\psi}{\partial r^2}+\frac{1}{r}\frac{\partial\psi}{\partial r}+\frac{\partial^2\psi}{\partial z^2}\right) = \frac{\partial\psi}{\partial t} \tag{3.3}$$

for the oxygen concentration $\psi(r,z,t)$ subject to the boundary conditions

$$\begin{aligned}
t<0 \qquad \psi &= \psi_0 \qquad \forall\, r,z \\
t\geq 0 \qquad \psi &= 0 \qquad \text{on } z=0,\ r\leq a \\
\frac{\partial\psi}{\partial z} &= 0 \qquad \text{on } z=0,\ r>a \\
\frac{\partial\psi}{\partial r} &= 0 \qquad \text{on } r=0 \\
\lim_{r,z\to\infty}\psi &= \psi_0
\end{aligned} \tag{3.4}$$

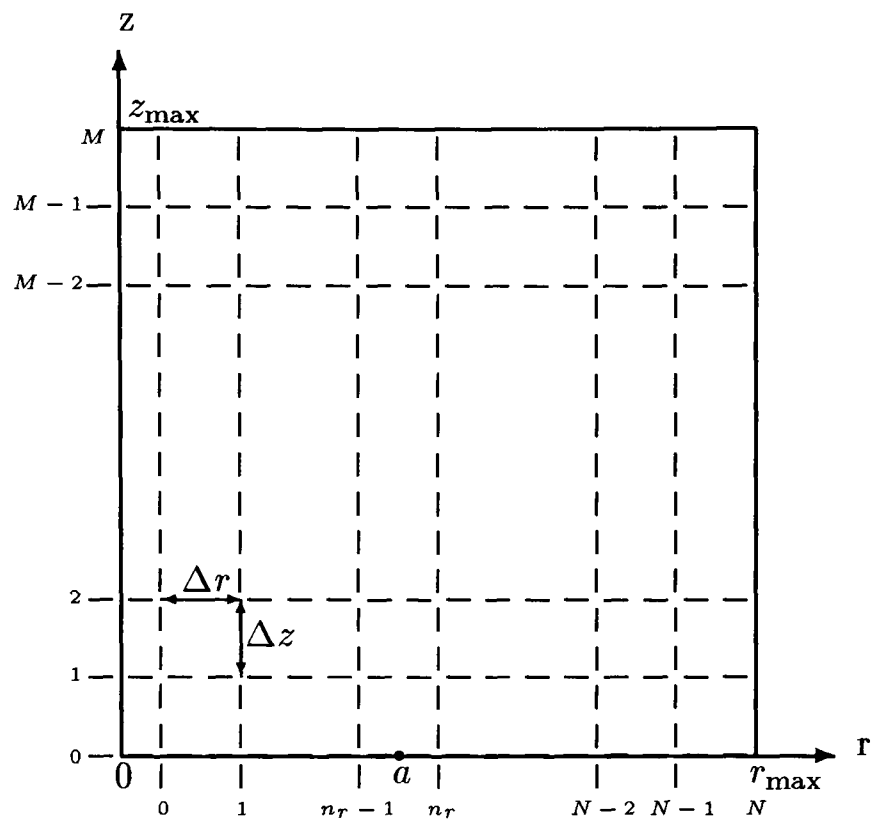


Fig. 3.2: Discretisation of the finite region for the finite difference solution of the problem.

where ψ_0 is the bulk concentration, and a is the electrode radius. The two boundary conditions on $z = 0$ for $t \geq 0$ ensure that there is a discontinuity in the first derivatives of ψ at the point $(a, 0)$, the *boundary singularity.* The quantity of interest is the current to the electrode caused by the reduction of oxygen at its surface given by

$$i(t) = 2\pi nFD \int_0^a \left(\frac{\partial \psi}{\partial z}\right)_{z=0} r\,dr. \tag{3.5}$$

3.3 Numerical Solution

To obtain a numerical solution to the problem the finite region of Figure 3.2 replaces the the semi–infinite region $[0 \leq r \leq \infty] \times [0 \leq z \leq \infty]$.

We therefore wish to solve equation (3.3) with boundary conditions (3.4) for a non–dimensional partial pressure $\phi = \psi/\psi_0$ in the region $[0 \leq r \leq r_{\max}] \times [0 \leq z \leq z_{\max}]$ with the boundary conditions *at infinity* replaced by

$$\phi = 1 \begin{cases} \text{on } r = r_{\max} & \forall \quad z \\ \text{on } z = z_{\max} & \forall \quad r \end{cases}. \tag{3.6}$$

The distances $r_{\max}$, $z_{\max}$ are chosen so that the diffusion front has not reached the boundary at the time of interest and are therefore time–dependent. To avoid the problem of the singularity of the term $\frac{1}{r}\frac{\partial \phi}{\partial r}$ at $r = 0$, we choose the mesh lines

$$\begin{aligned} r &= (i + 1/2)\,\Delta r \quad (i = 0, 1, \ldots, N) \\ z &= j\Delta z \quad (j = 0, 1, \ldots, M). \end{aligned} \tag{3.7}$$

The uniform spacings in the r– and z–directions are defined as

$$\Delta r = \frac{a}{n_r}, \qquad \Delta z = \frac{z_{\max}}{M} \tag{3.8}$$

where n_r is the number of mesh points above the electrode, ensuring that the singularity at $(a, 0)$ is placed half way between two mesh points.

Using a timestep Δt, the approximate non–dimensional solution $\phi_{i,j}^k$ is found at time $t_n = n\Delta t$ using the ADI method first suggested by Peaceman and Rachford [79] . This introduces an intermediate step at $(k + 1/2)$, $k = 0, \ldots, n-1$, and can be written in cylindrical coordinates as the pair of equations

$$\begin{aligned} \left[1 - \frac{\nu_r}{2}\left(\delta_r^2 + \frac{1}{i+\frac{1}{2}}\Delta_{0r}\right)\right]\phi_{i,j}^{k+\frac{1}{2}} &= \left[1 + \frac{\nu_z}{2}\delta_z^2\right]\phi_{i,j}^k \\ \left[1 - \frac{\nu_z}{2}\delta_z^2\right]\phi_{i,j}^{k+1} &= \left[1 + \frac{\nu_r}{2}\left(\delta_r^2 + \frac{1}{i+\frac{1}{2}}\Delta_{0r}\right)\right]\phi_{i,j}^{k+\frac{1}{2}} \end{aligned} \tag{3.9}$$

where Δ_{0r}, δ_r, δ_z are the usual central difference operators in the r– and z–directions and $\nu_r = D\frac{\Delta t}{\Delta r^2}$, $\nu_z = D\frac{\Delta t}{\Delta z^2}$. This results in the need to solve a tridiagonal system in the r–direction at $(k + 1/2)\Delta t$, and in the z–direction at $(k + 1)\Delta t$, with the benefit that it is unconditionally stable and second order accurate. The boundary conditions (3.4) are imposed via

$$\begin{array}{lllll} t < 0 & \phi_{i,j} & = & 1 & (i = 0, \ldots, N), (j = 0, \ldots, M) \\ t \geq 0 & \phi_{i,0} & = & 0 & (i = 0, \ldots, n_r - 1) \\ & \phi_{i,-1} & = & \phi_{i,1} & (i = n_r, \ldots, N) \\ & \phi_{-1,j} & = & \phi_{0,j} & (j = 0, \ldots, M) \\ & \phi_{i,M} & = & \phi_{N,j} = 1 & (i = 0, \ldots, N), (j = 0, \ldots, M) \end{array} \tag{3.10}$$

We will consider the ADI method in greater detail in the next chapter where we will describe both its sequential and parallel implementation, and in Chapter 5, where we will consider its local truncation error and stability properties. In this chapter we will concentrate on the derivation and implementation of the singularity correction.

3.4 Solution in the Neighbourhood of the Singularity

We obtain a truncated series solution about the point $(a, 0)$ by first transforming equation (3.3) to self–adjoint form using the transformation

$$u(r, z, t) = r^{\frac{1}{2}}\phi(r, z, t) \tag{3.11}$$

giving

$$D\left(\frac{\partial^2 u}{\partial r^2} + \frac{\partial^2 u}{\partial z^2} + \frac{u}{4r^2}\right) = \frac{\partial u}{\partial t}. \tag{3.12}$$

We may now transform to polar coordinates (ρ, θ) avoiding the complicated variable coefficients which arise when attempting to convert equation (3.3) directly to

polar coordinates. Following closely the work of Fox and Sankar [26] and Crank and Furzeland [12] we shift the origin to $(a,0)$ by setting $r' = r - a$, and transform to polar coordinates using $r' = \rho\cos\theta$, $z = \rho\sin\theta$. We then use the standard relations

$$\begin{aligned}
\frac{\partial^2 u}{\partial r^2} &= \cos^2\theta\frac{\partial^2 u}{\partial\rho^2} - \sin 2\theta\frac{\partial^2 u}{\partial\rho\partial\theta} + \sin^2\theta\frac{\partial^2 u}{\partial\theta^2} + \frac{\sin^2\theta}{\rho}\frac{\partial u}{\partial\rho} + \frac{\sin 2\theta}{\rho^2}\frac{\partial u}{\partial\theta} \\
\frac{\partial^2 u}{\partial z^2} &= \sin^2\theta\frac{\partial^2 u}{\partial\rho^2} + \sin 2\theta\frac{\partial^2 u}{\partial\rho\partial\theta} + \cos^2\theta\frac{\partial^2 u}{\partial\theta^2} + \frac{\cos^2\theta}{\rho}\frac{\partial u}{\partial\rho} - \frac{\sin 2\theta}{\rho^2}\frac{\partial u}{\partial\theta}
\end{aligned} \tag{3.13}$$

in equation (3.12) to give

$$\frac{\partial^2 u}{\partial\rho^2} + \frac{1}{\rho}\frac{\partial u}{\partial\rho} + \frac{1}{\rho^2}\frac{\partial^2 u}{\partial\theta^2} + \frac{u}{4(\rho\cos\theta + a)^2} = \frac{1}{D}\frac{\partial u}{\partial t}. \tag{3.14}$$

The boundary conditions on $z = 0$ become

$$\begin{aligned}
u &= 0 \qquad \text{on } \theta = \pi \\
\frac{\partial u}{\partial\theta} &= 0 \qquad \text{on } \theta = 0.
\end{aligned} \tag{3.15}$$

Writing

$$g(\rho,\theta) = \frac{1}{4a^2}\left(1 + \frac{\rho}{a}\cos\theta\right)^{-2} = \sum_{n=0}^{\infty}\rho^n\frac{(-1)^n(n+1)}{4a^{n+2}}\cos^n\theta = \sum_{n=0}^{\infty} g_n\rho^n \tag{3.16}$$

for $\rho < a$, we will attempt to find a separable solution of the form

$$u(\rho,\theta,t) = R(\rho)\Theta(\theta)T(t) + w(\rho,\theta) \tag{3.17}$$

where $w(\rho,\theta)$ is the separated solution of the steady–state problem as given by Crank and Furzeland. Substitution of (3.17) into equation (3.14) gives

$$\frac{R''}{R} + \frac{1}{\rho}\frac{R'}{R} + \frac{1}{\rho^2}\frac{\Theta''}{\Theta} + g(\rho,\theta) = \frac{1}{D}\frac{T'}{T} \tag{3.18}$$

and since the r.h.s. is a function of just one variable we may write

$$\frac{T'}{T} = -\alpha^2 D \quad \text{and} \quad T(t) \propto \exp(-\alpha^2 D t) \tag{3.19}$$

with α a real constant, to give the expected exponential decay with time. Setting

$$h(\rho,\theta) = \alpha^2 + g(\rho,\theta) = \sum_{m=0}^{\infty} h_m\rho^m \tag{3.20}$$

where $h_0 = \alpha^2 + g_0$, and $h_m = g_m$ for $m > 0$, we are left with the equation

$$\frac{R''}{R} + \frac{1}{\rho}\frac{R'}{R} + \frac{1}{\rho^2}\frac{\Theta''}{\Theta} + h(\rho,\theta) = 0 \tag{3.21}$$

for which we look for a solution in the form (in the notation of Fox and Sankar)

$$\sum_{j=0}^{\infty} R_j \Theta_j = \sum_{j=0}^{\infty} \rho^{\sigma+j} A_{\sigma,j}(\theta) \tag{3.22}$$

where σ is a real, positive constant. Substitution of (3.22) into (3.21) gives the equation

$$\begin{aligned} \left[A''_{\sigma,0} + \sigma^2 A_{\sigma,0}\right] \rho^{\sigma-2} + \left[A''_{\sigma,1} + (\sigma+1)^2 A_{\sigma,1}\right] \rho^{\sigma-1} + \\ \sum_{m=0}^{\infty} \left[A''_{\sigma,m+2} + (\sigma+m+2)^2 A_{\sigma,m+2}\right] \rho^{\sigma+m} = -\sum_{m=0}^{\infty} \sum_{j=0}^{m} h_{m-j} A_{\sigma,j} \rho^{\sigma+m} \end{aligned} \tag{3.23}$$

where primes denote differentiation w.r.t. θ. Comparison of powers of ρ gives the system of ordinary differential equations (ODE's)

$$\begin{aligned} A''_{\sigma,0} + \sigma^2 A_{\sigma,0} &= 0 \\ A''_{\sigma,1} + (\sigma+1)^2 A_{\sigma,1} &= 0 \\ A''_{\sigma,m+2} + (\sigma+m+2)^2 A_{\sigma,m+2} &= -\sum_{j=0}^{m} h_{m-j} A_{\sigma,j} \end{aligned} \tag{3.24}$$

for the $A_{\cdot,\cdot}$. The boundary conditions (3.15) can then be imposed via

$$\left.\begin{aligned} A_{\sigma,m}(\pi) &= 0 \\ A'_{\sigma,m}(0) &= 0 \end{aligned}\right\} m = 0, 1, \ldots. \tag{3.25}$$

Clearly to obtain non–trivial solutions $A_{\cdot,\cdot}$ satisfying these periodic boundary conditions we need

$$\sigma = k + 1/2 \ , \ k = 0, 1, \ldots. \tag{3.26}$$

Using simple linear ODE theory we may solve the associated homogeneous equations corresponding to (3.24) to obtain the general solution

$$A_{k+\frac{1}{2},m} = a_{m,k} \cos(k + \tfrac{1}{2} + m)\theta + p_{m,k}(\theta) \quad m = 0, 1, \ldots, \tag{3.27}$$

where $p_{m,k}(\theta)$ is the particular solution of the non–homogeneous problem.

Since equations (3.24) are already homogeneous for $m = 0, 1$ we have particular solutions $p_{0,k}(\theta) = p_{1,k}(\theta) = 0$. For $m = 2$ we have

$$A''_{k+\frac{1}{2},2} + \left(k + \frac{5}{2}\right)^2 A_{k+\frac{1}{2},2} = -\left(\alpha^2 + \frac{1}{4a^2}\right) a_{0,k} \cos(k + \frac{1}{2})\theta \tag{3.28}$$

suggesting that the particular solution has the form

$$p_{2,k}(\theta) = d_{0,k} \cos(k + \frac{1}{2})\theta, \tag{3.29}$$

and substitution into equation (3.28) gives

$$d_{0,k} = -a_{0,k}\left(\frac{\alpha^2 + 1/4a^2}{2(2k+3)}\right). \tag{3.30}$$

For $m = 3$ we have

$$\begin{aligned} A''_{k+\frac{1}{2},3} + \left(k+\frac{7}{2}\right)^2 A_{k+\frac{1}{2},3} &= a_{0,k}\frac{2}{4a^3}\cos\theta\cos(k+\frac{1}{2})\theta \\ &\quad -a_{1,k}\left(\alpha^2 + \frac{1}{4a^2}\right)\cos(k+\frac{3}{2})\theta, \end{aligned} \tag{3.31}$$

and we can use the relation

$$\cos\theta\cos(k+\frac{1}{2})\theta = \frac{1}{2}\left(\cos(k+\frac{3}{2})\theta + \cos(k-\frac{1}{2})\theta\right), \tag{3.32}$$

to obtain the form of the particular solution as

$$p_{3,k}(\theta) = d_{1,k}\cos(k-\frac{1}{2})\theta + d_{2,k}\cos(k+\frac{3}{2})\theta. \tag{3.33}$$

Substituting this into equation (3.31) and comparing coefficients of the cosine terms gives

$$d_{1,k} = \frac{a_{0,k}}{16a^3(2k+3)} \tag{3.34}$$

$$d_{2,k} = \frac{1}{2(2k+5)}\left(\frac{a_{0,k}}{4a^3} - a_{1,k}\left(\frac{1}{4a^2} + \alpha^2\right)\right). \tag{3.35}$$

Combining the above we have

$$\begin{aligned} A_{k+1/2,0} &= a_{0,k}\cos(k+1/2)\theta \\ A_{k+1/2,1} &= a_{1,k}\cos(k+3/2)\theta \\ A_{k+1/2,2} &= a_{2,k}\cos(k+5/2)\theta + d_{0,k}\cos(k+1/2)\theta \\ A_{k+1/2,3} &= a_{3,k}\cos(k+7/2)\theta + d_{1,k}\cos(k-1/2)\theta + \\ &\quad d_{2,k}\cos(k+3/2)\theta \\ &\vdots \end{aligned} \tag{3.36}$$

which together with equations (3.19) and (3.21) give the series solution about the singularity in the form

$$u(\rho,\theta,t) = \sum_{i=0}^{\infty}\exp(-\alpha_i^2 Dt)\sum_{j=0}^{\infty}\sum_{k=0}^{\infty}\rho^{k+\frac{1}{2}+j}A_{k+\frac{1}{2},j}(\theta) + w(\rho,\theta), \tag{3.37}$$

where the α_i's are determined by the conditions on the remaining parts of the boundary (Bell and Crank [6]).

Substituting from equation (3.36) and gathering together powers of ρ in equation (3.37) gives

$$\begin{aligned} u(\rho,\theta,t) &= \sum_{i=0}^{\infty}\exp(-\alpha_i^2 Dt)\left\{A_{1/2,0}\rho^{\frac{1}{2}}+\left(A_{1/2,0}+A_{1/2,0}\right)\rho^{\frac{3}{2}}+\right. \\ &\quad \left(A_{5/2,0}+A_{3/2,1}+A_{1/2,2}\right)\rho^{\frac{5}{2}}+\left(A_{7/2,0}+A_{5/2,1}+\right. \qquad (3.38) \\ &\quad \left.\left. A_{3/2,2}+A_{1/2,3})\rho^{\frac{7}{2}}+O(\rho^{\frac{9}{2}})\right\}+w(\rho,\theta), \\ &= \sum_{i=0}^{\infty}\exp(-\alpha_i^2 Dt)\left\{\rho^{\frac{1}{2}}\left[1-\left(\frac{\alpha_i^2}{6}+\frac{1}{24a^2}\right)\rho^2+\frac{1}{48a^3}\rho^3\right]a_{0,0}\cos\frac{1}{2}\theta\right. \\ &\quad +\rho^{\frac{3}{2}}\left[a_{0,1}+a_{1,0}-\left(\frac{\alpha_i^2}{8}+\frac{1}{32a^2}\right)\rho^2 a_{0,1}\right. \qquad (3.39) \\ &\quad \left.+\left(\frac{a_{0,0}}{40a^3}-\frac{a_{1,0}}{10}\left(\alpha_i^2+\frac{1}{4a^2}\right)\right)\rho^2\right]\cos\frac{3\theta}{2}+\rho^{\frac{5}{2}}\left[a_{0,2}+a_{1,1}+a_{2,0}\right]\cos\frac{5\theta}{2} \\ &\quad \left.+\rho^{\frac{7}{2}}\left[a_{0,3}+a_{1,2}+a_{2,1}+a_{3,0}\right]\cos\frac{7\theta}{2}+O(\rho^{\frac{9}{2}})\right\}. \end{aligned}$$

Since we now have to invert our original transformation (3.11) to obtain the solution for ϕ, this series looks a little intractable. However, since we will use our singularity correction only in the immediate neighbourhood of the singularity where $\rho << a$, we might hope that it will be sufficient to truncate the series at $O(\rho^{\frac{7}{2}})$ to obtain

$$\begin{aligned} u(\rho,\theta,t) &= \sum_{i=0}^{\infty}\exp(-\alpha_i^2 Dt)\left[b_1\left\{1-\left(\frac{\alpha_i^2}{6}+\frac{1}{24a^2}\right)\rho^2\right\}\rho^{\frac{1}{2}}\cos\frac{\theta}{2}+\right. \\ &\quad \left. b_2\rho^{\frac{3}{2}}\cos\frac{3\theta}{2}+b_3\rho^{\frac{5}{2}}\cos\frac{5\theta}{2}+O(\rho^{\frac{7}{2}})\right]+w(\rho,\theta) \qquad (3.40) \end{aligned}$$

for u, with $b_1 = a_{0,0}$, $b_2 = a_{0,1}+a_{1,0}$, $b_3 = a_{0,2}+a_{1,1}+a_{2,0}$. We wish to solve for our original function $\phi = r^{-1/2}u$, so

$$\begin{aligned} \phi(\rho,\theta,t) &= r^{-\frac{1}{2}}u = a^{-\frac{1}{2}}\left(1+\frac{\rho}{a}\cos\theta\right)^{-\frac{1}{2}}u(\rho,\theta,t) \qquad (3.41) \\ &= a^{-\frac{1}{2}}\left(1-\frac{1}{2}\frac{\rho}{a}\cos\theta+\frac{3}{8}\left(\frac{\rho}{a}\cos\theta\right)^2+O(\rho^3)\right)u(\rho,\theta,t) \end{aligned}$$

Substituting for $u(\rho,\theta,t)$ from equation (3.41) and taking the product gives

$$\begin{aligned} \phi(\rho,\theta,t) &= \sum_{i=0}^{\infty}\exp(-\alpha_i^2 Dt)\left\{b_1\left(\frac{\rho}{a}\right)^{\frac{1}{2}}\left[1-\left(\frac{\alpha_i^2}{6}+\frac{1}{24a^2}\right)\rho^2-\frac{1}{2}\frac{\rho}{a}\cos\theta\right.\right. \\ &\quad \left.+\frac{3}{8}\frac{\rho^2}{a^2}\cos^2\theta\right]\cos\frac{\theta}{2}+b_2\rho^{\frac{3}{2}}a^{-\frac{1}{2}}\left[1-\frac{1}{2}\frac{\rho}{a}\cos\theta\right]\cos\frac{3\theta}{2} \qquad (3.42) \\ &\quad \left.+b_3\rho^{\frac{5}{2}}a^{-\frac{1}{2}}\cos\frac{5\theta}{2}+O(\rho^{\frac{7}{2}})\right\}+w(\rho,\theta) \end{aligned}$$

We can now make use of the trigonometric relations

$$\begin{aligned}
\cos\theta\cos\frac{\theta}{2} &= \frac{1}{2}\left(\cos\frac{3\theta}{2}+\cos\frac{\theta}{2}\right)\\
\cos\theta\cos\frac{3\theta}{2} &= \frac{1}{2}\left(\cos\frac{5\theta}{2}+\cos\frac{\theta}{2}\right)\\
\cos^2\theta\cos\frac{\theta}{2} &= \frac{1}{2}\cos\frac{\theta}{2}+\frac{1}{4}\left(\cos\frac{5\theta}{2}+\cos\frac{3\theta}{2}\right)
\end{aligned} \tag{3.43}$$

to give the time–dependent form of the locally valid series expansion about the singular point as

$$\begin{aligned}
\phi(\rho,\theta,t) = \sum_{i=0}^{\infty}\exp(-\alpha_i^2 Dt)\Bigg[b_1\left\{\left(\frac{\rho}{a}\right)^{\frac{1}{2}}\cos\frac{\theta}{2}-\frac{1}{4}\left(\frac{\rho}{a}\right)^{\frac{3}{2}}\left(\cos\frac{3\theta}{2}+\cos\frac{5\theta}{2}\right)+\right.\\
\left.\frac{1}{96}\left(\frac{\rho}{a}\right)^{\frac{5}{2}}\left(\left(14-16a^2\alpha_i^2\right)\cos\frac{\theta}{2}+9\cos\frac{3\theta}{2}+9\cos\frac{5\theta}{2}\right)\right\}+\\
b_2 a\left\{\left(\frac{\rho}{a}\right)^{\frac{3}{2}}\cos\frac{3\theta}{2}-\frac{1}{4}\left(\frac{\rho}{a}\right)^{\frac{5}{2}}\left(\cos\frac{\theta}{2}+\cos\frac{5\theta}{2}\right)\right\}+\\
b_3 a^2\left(\frac{\rho}{a}\right)^{\frac{5}{2}}\cos\frac{5\theta}{2}+O\left(\rho^{\frac{7}{2}}\right)\Bigg]+w(\rho,\theta)
\end{aligned} \tag{3.44}$$

for $\rho << a$. The expression in square brackets is now in the same form as the steady–state solution given by Crank and Furzeland apart from the extra term in α_i^2. Therefore defining

$$\begin{aligned}
c_i(t) &= c_i^{'} + b_i\sum_{j=0}^{\infty}\exp(-\alpha_j^2 Dt)\ i=1,2,3\\
c_4(t) &= b_1\sum_{j=0}^{\infty}\frac{\alpha_j^2}{6}\exp(-\alpha_j^2 Dt)
\end{aligned} \tag{3.45}$$

where the $c_i^{'}$'s are the corresponding coefficients in the steady state solution, we can now write our solution

$$\phi(\rho,\theta,t)=\sum_{i=1}^{4}f_i(\rho,\theta)c_i(t)=\mathbf{f}^T\mathbf{c} \tag{3.46}$$

in an obvious notation, with the $f_i(\rho,\theta)$ defined by equation (3.44).

3.5 Implementation

To implement the steady–state series solution, Crank and Furzeland [12, 13] use it to derive modified finite–difference stencils for points N "near" the singularity (i.e. in some neighbourhood $N(a)$ to be determined), in terms of points F,"far" points, distant from it. In the time–dependent case this would involve (at least) the inversion of a 4×4 matrix at each timestep for each near point and is therefore undesirably costly.

Instead we modify the methods of Fox and Sankar [26] and Bell and Crank [6] by using a least squares approach to obtain approximations for the coefficients $c_i(t)$, $i = 1, \ldots, 4$, which involves finding only one pseudo–inverse of an $m \times 4$ matrix ($m \geq 4$) before any other calculations take place. If we take $N(a)$ to be an $2l \times k$ submesh (see Figure 3.3) (i.e. we correct for the effects of the singularity l mesh points to each side of the singularity in the r–direction, and k points above it in the z–direction), we have $m = 2(l + k) + 3$ immediate neighbours to use as far points. We have from equation (3.46) for each far point

$$\phi_F(\rho_i, \theta_i, t) = \mathbf{f}_F^T(\rho_i, \theta_i)\mathbf{c}(t) \quad i = 1, \ldots, m \tag{3.47}$$

where the subscript F indicates a far point. In matrix notation this becomes

$$\phi_F = \mathbf{F}\mathbf{c} \tag{3.48}$$

where $\mathbf{F}$ is an $m \times 4$ matrix with $[\mathbf{F}]_{ij} = f_j(\rho_i, \theta_i)$, which gives us an over–determined system for the unknowns $\mathbf{c}$. Rather than solve this least squares problem at each timestep, we find the pseudo–inverse[1] of $\mathbf{F}$, $\mathbf{F}^*$ which normally requires use of singular value decomposition (see, for example, Wilkinson [46]). However since all of our far points are distinct, $\mathbf{F}$ is of full rank, so that $\mathbf{F}^*$ is defined by

$$\mathbf{F}^* = (\mathbf{F}^T\mathbf{F})^{-1}\mathbf{F}^T. \tag{3.49}$$

When using a conventional sequential computer, problems such as finding the pseudo–inverse of a matrix are made simple by the availability of subroutine libraries (for example the original code to implement the singularity correction described above was written in Fortran (see Appendix C) and made use of routine F01 BLF from the NAg [66] library to find $\mathbf{F}^*$). However in Chapter 5 we will implement the singularity correction on an architecture for which standard libraries are not yet available, and we therefore outline briefly the procedure we will use there.

The matrix $\mathbf{F}$ is first factored into an $m \times n$ orthogonal matrix $\mathbf{Q}$ and an $n \times n$ upper triangular matrix $\mathbf{R}$ to give

$$\mathbf{F}\mathbf{c} \;=\; \mathbf{Q}\begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix}\mathbf{c} = \phi_F. \tag{3.50}$$

Multiplying both sides by $\mathbf{F}^T = \begin{bmatrix}\mathbf{R}^T\mathbf{0}^T\end{bmatrix}\mathbf{Q}^T$ gives

$$\begin{bmatrix}\mathbf{R}^T\mathbf{0}^T\end{bmatrix}\mathbf{Q}^T\mathbf{Q}\begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix}\mathbf{c} \;=\; \begin{bmatrix}\mathbf{R}^T\mathbf{0}^T\end{bmatrix}\mathbf{Q}^T\phi_F \tag{3.51}$$

[1]So defined since when required to find a solution to the problem $\mathbf{A}\mathbf{x} = \mathbf{b}$ for $\mathbf{x}$, with $\mathbf{A}$ $m \times n, m > n$, and rank($\mathbf{A}$)$< n$, the pseudo–inverse $\mathbf{A}^*$ gives the best least–squares fit, i.e. $\mathbf{x} = \mathbf{A}^*\mathbf{b}$ is such that $||\mathbf{x}||_2$ is minimised for all vectors which minimise $||\mathbf{A}\mathbf{x}-\mathbf{b}||_2$. The minimiser of $||\mathbf{A}\mathbf{x} - \mathbf{b}||_2$ is unique only if rank($\mathbf{A}$)$= n$

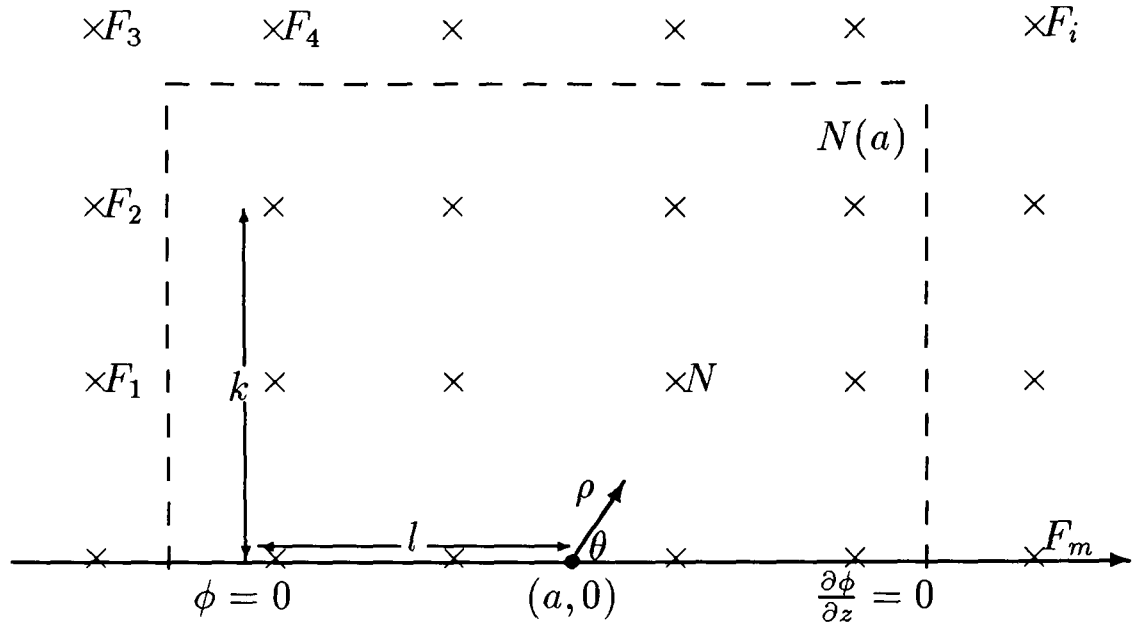


Fig. 3.3: The effects of the singularity are to be corrected in neighbourhood $N(a)$, which is an $2l \times k$ submesh about the point $(a, 0)$, in terms of far points F. (This is the case $k = l = 2$).

and since $\mathbf{Q}$ is orthogonal we have

$$\mathbf{R}^T\mathbf{R}\mathbf{c} = \left[\mathbf{R}^T\mathbf{0}^T\right]\mathbf{Q}^T\phi_F. \tag{3.52}$$

Since $\mathbf{R}$ is upper triangular and therefore invertible we can multiply both sides by $(\mathbf{R}^T)^{-1}$ and then by $(\mathbf{R})^{-1}$ to get

$$\begin{aligned} \mathbf{c} &= \left[\mathbf{R}^{-1}\mathbf{0}^T\right]\mathbf{Q}^T\phi_F \\ &= \mathbf{F}^*\phi_F, \end{aligned} \tag{3.53}$$

which defines $\mathbf{F}^*$. $\mathbf{R}$ and $\mathbf{Q}$ are found using Householder reflections (which is a common algorithm, we use the one given by Hager [34], since it is particularly efficient in both storage and computation), and since $\mathbf{R}$ is upper–triangular we can find its inverse very easily using back–substitution.

Having found $\mathbf{F}^*$ we may write

$$\mathbf{c}^* = \mathbf{F}^*\phi_F \tag{3.54}$$

where $\mathbf{c}^*(t)$ is now the best least–squares approximation to the coefficients of the series solution in terms of all neighbouring far points. If we now use equation (3.46) (with subscript N indicating a near point) again for the $l(2k+1)$ near points at which ϕ is non–zero, we have

$$\phi_N^i = \mathbf{f}_N^T\mathbf{c} \quad i = 1, \ldots, l(2k+1), \tag{3.55}$$

and substituting for $\mathbf{c}$ from (3.54) gives

$$\phi_N^i = \mathbf{f}_N^T \mathbf{F}^* \phi_F = \mu^T \phi_F \quad i = 1, \ldots, l(2k+1). \tag{3.56}$$

The m–vector $\mu^T = \mathbf{f}_N^T \mathbf{F}^*$ is time–independent and may therefore be calculated *a–priori*. Thus the corrected solution at each timestep can be obtained by doing the usual ADI step at all points in the mesh, then overwriting the values at the near points by values calculated using (3.56), which involves just m multiplications per near point.

It is possible to obtain the optimum approximation for $\mathbf{c}$ more simply by choosing the appropriate four far points. $\mathbf{F}$ is then a 4×4 matrix and we can find its inverse to give $\mathbf{c} = \mathbf{F}^{-1}\phi_F$. This approach, however, is not robust when extended to the shielded problem, since the nature of the solution in the region of the singularity varies greatly with the particular set of parameter values chosen.

3.6 Calculation of the Flux to the Electrode

Having corrected the concentration values, we must also take into account the nature of the singularity in calculating flux values. The current to the electrode is given by equation (3.5)

$$i(t) = 2\pi n F D \int_0^a \left(\frac{\partial \psi}{\partial z}\right)_{z=0} r dr \tag{3.57}$$

To facilitate comparison with previous work, we define a non–dimensional time

$$\tau = \frac{4Dt}{a^2}, \tag{3.58}$$

and replace the calculation of the current by the equivalent non–dimensional flux defined by

$$f(\tau) = \frac{\pi}{2a} \int_0^a \left(\frac{\partial \phi}{\partial z}\right)_{z=0} r dr. \tag{3.59}$$

We obtain an approximation for $f(\tau)$ by calculating the approximate flux Q_i into boxes $i = 0, \ldots, n_r$ (see Figure 3.4). Away from the singularity in boxes $i = 0, \ldots, i_s$, Q_i may be calculated using an interpolant over each box as described by Evans and Gourlay [18]. We define a local bilinear interpolant for ϕ in the form

$$\phi = \alpha z + \beta r z + \gamma r + \delta. \tag{3.60}$$

Differentiating we obtain $\left(\frac{\partial \phi}{\partial z}\right)_{z=0} = \alpha + \beta r$ and substitution into (3.59) gives

$$Q_i \propto \int_{box(i)} r(\alpha + \beta r) dr = \left[\frac{\alpha r^2}{2} + \frac{\beta r^3}{3}\right]_{box(i)}. \tag{3.61}$$

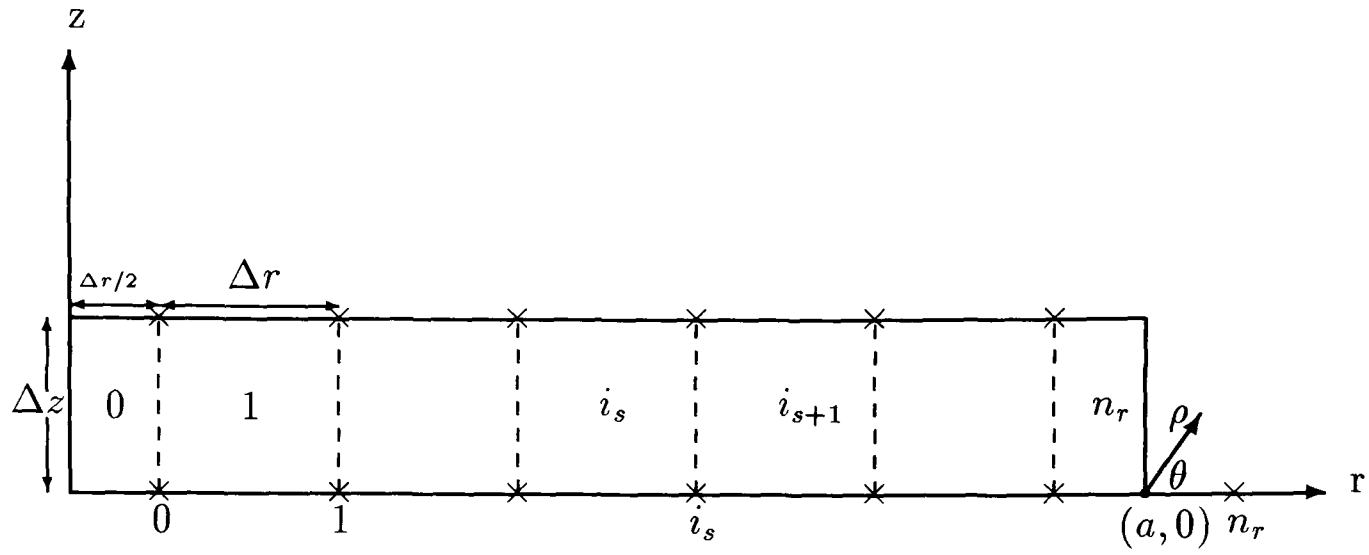


Fig. 3.4: Flux calculation. A local bilinear interpolant is used in boxes $0,\ldots,i_s$, and an interpolant taking into account the form of the singularity for $i > i_s$ near to the singularity at $(a,0)$.

Since we know the values of ϕ at the corners of each box, we have a simple linear system of equations for α, β, γ, and δ, which can be solved to give $\gamma = \delta = 0$ and

$$\alpha = \frac{1}{\Delta z}\left\{\left(i+\frac{1}{2}\right)\phi_{i-1,1} - \left(i-\frac{1}{2}\right)\phi_{i,1}\right\} \tag{3.62}$$

$$\beta = \frac{1}{(\Delta r \Delta z)}\left\{\phi_{i,1} - \phi_{i-1,1}\right\}, \tag{3.63}$$

for $i = 1,\ldots,i_s$. Substitution into equation (3.61) then gives the non–dimensional flux into each box as

$$Q_i = \frac{\pi}{4a}\frac{\Delta r^2}{\Delta z}\left\{\left(i-\frac{1}{6}\right)\phi_{i-1,1} - \left(i+\frac{1}{6}\right)\phi_{i,1}\right\} \tag{3.64}$$

For box 0 we make use of the symmetry condition $\phi_{0,1} = \phi_{-1,1}$ on $r = 0$, to give $\beta = 0, \alpha = \phi_{0,1}/\Delta z$ and

$$Q_0 = \frac{\pi}{4a}\frac{\Delta r^2}{4\Delta z}\phi_{0,1}. \tag{3.65}$$

For $i > i_s$ we must use a local interpolant which takes into account the form of the singularity. Rather than attempting to integrate equation (3.44), we return to equation (3.40) which we write in the form

$$\begin{aligned} u(\rho,\theta,t) = \Bigg[& c_1\left(1-\frac{\rho^2}{24a^2}\right)\rho^{\frac{1}{2}}\cos\frac{\theta}{2} + \\ & c_2\rho^{\frac{3}{2}}\cos\frac{3\theta}{2} + c_3\rho^{\frac{5}{2}}\cos\frac{5\theta}{2} + c_4\rho^{\frac{5}{2}}\cos\frac{\theta}{2} + O(\rho^{\frac{7}{2}})\Bigg]. \end{aligned} \tag{3.66}$$

We wish to approximate the flux given by equation (3.59), but since $\phi = r^{-\frac{1}{2}}u$ we have $r\frac{\partial\phi}{\partial z} = r^{\frac{1}{2}}\frac{\partial u}{\partial z}$ and

$$\begin{aligned} r\left(\frac{\partial\phi}{\partial z}\right)_{z=0} &= (a-\rho)^{\frac{1}{2}}\left\{\frac{\partial u}{\partial\rho}\frac{\partial\rho}{\partial z}+\frac{\partial u}{\partial\theta}\frac{\partial\theta}{\partial z}\right\}_{\theta=\pi} \\ &= (a-\rho)^{\frac{1}{2}}\left\{\sin\theta\frac{\partial u}{\partial\rho}+\frac{\cos\theta}{\rho}\frac{\partial u}{\partial\theta}\right\}_{\theta=\pi} \\ &= (a-\rho)^{\frac{1}{2}}\frac{1}{\rho}\left(\frac{\partial u}{\partial\theta}\right)_{\theta=\pi}. \end{aligned} \tag{3.67}$$

Differentiating equation (3.66) w.r.t. θ and setting $\theta = \pi$, equation (3.67) becomes

$$\begin{aligned} r\left(\frac{\partial\phi}{\partial z}\right)_{z=0} = \frac{a^{\frac{1}{2}}}{2}\left(1-\frac{\rho}{a}\right)^{\frac{1}{2}}\rho^{-\frac{1}{2}}\left(c_1\left(1-\frac{\rho^2}{24a^2}\right)+\right. \\ \left. 3c_2\rho+(5c_3+c_4)\rho^2+O(\rho^3)\right) \end{aligned} \tag{3.68}$$

We can expand the first bracket in powers of $(\frac{\rho}{a})^{\frac{1}{2}}$, take the product, and substitute into equation (3.59) to give

$$\begin{aligned} Q_{sing} = \frac{\pi}{a^{\frac{1}{2}}}\int_0^{\rho_{is}} c_1\rho^{-\frac{1}{2}}\left(1-\frac{\rho}{a}-\frac{\rho^2}{6a^2}\right)+ \\ 3c_2\rho^{\frac{1}{2}}\left(1-\frac{1}{2}\frac{\rho}{a}\right)+5(c_3+c_4)\rho^{\frac{5}{2}}+O(\rho^{\frac{7}{2}})d\rho. \end{aligned} \tag{3.69}$$

Finally, we can integrate to obtain the flux into the region of the electrode affected by the singularity as

$$\begin{aligned} Q_{sing} \approx \frac{\pi}{a^{\frac{1}{2}}}\left[c_1\rho^{\frac{1}{2}}\left(1-\frac{\rho}{6a}-\frac{\rho^2}{30a^2}\right)+\right. \\ \left. c_2\rho^{\frac{3}{2}}\left(1-\frac{3\rho}{10a}\right)+\left(c_3+\frac{c_4}{5}\right)\rho^{\frac{5}{2}}+O(\rho^{\frac{7}{2}})\right]_0^{\rho_{is}}. \end{aligned} \tag{3.70}$$

The coefficients c_i, $i = 1,\ldots,4$ are calculated from the far points using equation (3.54).

Our overall approximation to the non-dimensional flux $f(\tau)$ to the electrode is given by the sum of these contributions

$$f(\tau) \approx Q_0 + \sum_0^{i=i_s} Q_i + Q_{sing}. \tag{3.71}$$

3.7 Numerical Convergence

Previous attempts to solve the unshielded problem by both analytic and digital simulation techniques were summarised by Shoup and Szabo [88], who also corrected the analytic work of Aoki and Osteryoung [3]. They regarded as highly accurate the numerical solution of Heinze [40], who uses the ADI method to solve

τ	n_r	Mesh N × M	Uncorrected Hopscotch	Uncorrected ADI	This Work
0.0961	10	15 × 5	2.954	3.377	3.544
	20	30 × 10	3.468	3.544	3.616
	40	60 × 20	3.590	3.660	3.676
	80	120 × 40	3.646	3.676	3.677
1.0	10	30 × 30	1.586	1.665	1.753
	20	60 × 60	1.713	1.727	1.766
	40	120 × 120	1.746	1.752	1.767
	80	240 × 240	1.759	1.762	1.767

Table 3.1: Convergence of non–dimensional flux with mesh spacing.

for the concentration values through which he fits a cubic spline in the region of the singularity to obtain a more accurate approximation for the flux (he makes no attempt to correct for the singularity). Unfortunately, Heinze did not give details of the meshes that he used, but simply stated *"in the interest of accuracy, the number of elements covering the electrode in the radial direction should be at least 30"*. We cannot therefore compare his results with ours for convergence with mesh spacing. Shoup and Szabo [89] give full details of parameters used in testing the convergence of the hopscotch method (first described by Gourlay [31, 32]), without correcting for the singularity, at two non–dimensional times, $\tau = 0.0961$ and $\tau = 1.0$. These correspond to a very short and a very long time respectively, and allow comparison with the asymptotic results of Aoki and Osteryoung [3].

We have reproduced results consistent with theirs for the uncorrected hopscotch method at $\tau = 0.0961$. These are shown together with values obtained at $\tau = 1.0$ in Table 3.1, where we compare the results with those we have obtained using the ADI method both without and with correction for the singularity.

Since we are using non–dimensional variables for both the time and the flux, we can take the bulk partial pressure ϕ_0, the electrode radius a, and the diffusion coefficient D to be unity, and all results shown in Table 3.1 were calculated using $\Delta z = \Delta r = a/n_r$. At $\tau = 0.0961$, a *mesh ratio* ν_r $(= \nu_z = D\Delta t/\Delta r^2)$ value of 0.5 was used on the 15 × 5 and 30 × 10 meshes for both ADI methods and for all the meshes for the hopscotch method, whilst a value of 2.0 was used for the finer ADI meshes. For $\tau = 1.0$, $\nu_r = 2.5$ was used for both ADI methods on the 30 × 30 mesh, and $\nu_r = 10.0$ on all finer meshes. For the hopscotch method we were again forced to use a value of 0.5, since for larger values of ν_r results became progressively more inaccurate and began to oscillate wildly on the 60 × 60 mesh, and became unstable for values of $\nu_r > 1.2$ on the 120 × 120 mesh. In all cases, the neighbourhood of correction, $N(a)$, was one point to each side of the electrode and one row above it ($l = k = 1$ in Figure 3.4).

It can be seen that at $\tau = 0.0961$ the meshes required to give 1% accuracy are 120 $\times$ 40 for hopscotch, but 60 $\times$ 20 for ADI, with or without correction. The corresponding figures at $\tau = 1.0$ are 240 $\times$ 240 for hopscotch, 120 $\times$ 120 for uncorrected ADI, and 60 $\times$ 60 for corrected ADI. The tables stop at the point where $N(a)$ is of such a size that the truncation errors of the series and finite difference methods are well matched. If we, for example, go on to a 240 $\times$ 80 mesh with 160 elements covering the electrode at $\tau = 0.0961$, the 1 $\times$ 1 sub–mesh is too small and a lower flux of 3.673 is obtained. By increasing the size of $N(a)$ to a 2 $\times$ 2 sub–mesh, we obtain the value 3.676.

Correcting for the singularity enables us to obtain our required 1% accuracy, at least for the unshielded case, much more economically. The greater accuracy attained on coarser meshes at the later time is due to the decay of the time–dependent terms in the truncation errors of both the ADI method and the singularity correction. This is demonstrated further in the next section where we are able to obtain the required accuracy with progressively coarser meshes and longer timesteps as time advances.

3.7.1 Evolution of flux with time

The corrections to the analysis of Aoki and Osteryoung [3] made by Shoup and Szabo [88] allowed them to give the expression

$$f(\tau) = \frac{\pi}{4} + \frac{\pi^{\frac{1}{2}}}{2}\tau^{-\frac{1}{2}} + \left(1 - \frac{\pi}{4}\right)\exp\left(\frac{2\pi^{\frac{1}{2}}(1 - 8\pi^{-2})}{4 - \pi}\tau^{-\frac{1}{2}}\right) \qquad (3.72)$$

for the flux which they claim is accurate to 0.6% for all values of τ. In Table 3.2 we therefore use this expression to compare the analytic solution with our results obtained using the singularity correction (all values quoted have converged to better than 1% accuracy), and the numerical results of Heinze [40]. Again we have taken $\Delta r = \Delta z = a/n_r$ and $a = \phi_0 = D = 1$. The value of the flux decays with increasing τ, as do the time–dependent terms in the truncation error. As a result we can increase the step sizes in r and z with time. This effect is somewhat off–set by the need to increase both N and M so that the depletion zone does not reach the outer boundary (this would invalidate our boundary conditions). The overall effect is that the number of computing operations required to obtain the flux is more or less independent of τ for $0.04 < \tau < 25.0$. When the flux is to be found at large τ values, the concentration values calculated on the coarse meshes used are relatively inaccurate early in the calculation, but the effects of these early errors decay with time and good accuracy is obtained at the required time. For short τ values, the solution is changing very rapidly with time resulting in large truncation errors in both the finite difference and series solutions, hence the need for a relatively fine mesh to obtain good accuracy. The method can therefore be used quite economically over a wide range of times provided that the steplengths are adjusted appropriately.

Time τ	Mesh $M \times N$	n_τ	Δt	Flux $f(\tau)$		
				Analytic	Heinze	This work
0.01	80×400	320	3.13×10^{-6}	9.647	9.660	9.663
0.04	80×240	160	7.81×10^{-5}	5.221	5.237	5.239
0.09	80×160	80	3.13×10^{-4}	3.775	3.772	3.771
0.25	80×80	40	1.25×10^{-3}	2.603	2.609	2.609
0.64	100×100	40	2.50×10^{-3}	1.974	1.969	1.973
1.21	160×160	40	6.88×10^{-3}	1.696	1.688	1.691
2.25	100×100	20	1.25×10^{-2}	1.504	1.495	1.500
4.00	100×100	20	1.25×10^{-2}	1.374	1.367	1.372
6.76	60×60	10	5.00×10^{-2}	1.285	1.280	1.286
25.0	100×100	10	5.00×10^{-2}	1.146	1.141	1.146
100.	200×200	10	5.00×10^{-2}	1.072	1.071	1.075

Table 3.2: Comparison of the non–dimensional calculation of the flux by various methods with increasing time. The mesh sizes given are those used in our calculations.

3.7.2 Relevance to practical electrode calculations

In this chapter, we have dealt with the case of the unshielded electrode, and have demonstrated that by correcting for the singularity at the electrode edge we can economically obtain agreement of high accuracy with previous work and reveal the sources of error which limit accuracy. We will show in Chapter 5 that this can be straightforwardly extended to the shielded case where we will also obtain a parallel numerical solution to this problem. In the next chapter we therefore consider three sequential algorithms which are representative of the class of parabolic p.d.e. solvers, with the aim of discovering the most appropriate parallel numerical algorithm for the solution of problems of this type.

Chapter 4

Parallel Solution of Parabolic Partial Differential Equations

The limitations of sequential computing and the advent of parallelism have recently been comprehensively reviewed by Leland [56], who also develops a new performance model for parallel computers, the *parallel effectiveness* model. The aim of the model is to determine how effective a parallel implementation of a given algorithm might be on a particular parallel architecture, by allowing for the potential trade–off between the speed of the sequential algorithm and the efficiency of its parallel implementation. This allows Leland to compare directly various parallel algorithms for the solution of large, sparse, linear systems, making use of a new definition of parallel efficiency which he proposes for this class of problems. By extending this definition to cover a wider class of numerical algorithms, including those for the solution of the parabolic partial differential equations in which we are particularly interested, we are able to apply the new performance model to compare the parallel implementation of three algorithms for the solution of a simple model problem of this type, for which we know the exact solution.

4.1 The Parallel Architecture

Of the wide variety of parallel architectures that has become available over the last decades, three predominate; the SIMD machine (Single Instruction, Multiple Data stream, as defined by Flynn [25]), such as the ICL DAP, which has *distributed memory* (i.e. each processor has its own local memory) and each processor executes identical instructions in lock step; the shared memory machine, such as the Cray–2, in which each processor has MIMD (Multiple Instruction, Multiple Data stream) control and a small, fast, on–chip cache memory so that it can execute a distinct program asynchronously, but shares a single (large) off–chip memory; in the final category each processor has its own local memory together with MIMD control so that each can operate as a distinct computer in its own right, which we will refer to as the *multicomputer*. An example is the Inmos transputer in which we are primarily interested. The processors associated with each type of architecture are usually classified according to *grain size*, which refers to the computational power of each processor. Typical SIMD machines consist of thousands of small grain,

single bit processors, each with a few kilobytes of store. Shared memory machines consist of a few (2–16) very powerful, large grain processors, usually with vector pipelining capabilities each connected to a single, very large, shared memory ($\approx$ 1 Gbyte). Multicomputers then fall into the medium grain size category, each processor having a few Mbytes of local memory.

4.1.1 The Inmos T800 transputer

The parallel architecture with which we will be concerned is the multicomputer, with the Inmos IMS T800 transputer (or T8) as the basic processor. The T8 is a 32–bit CMOS microcomputer with a 64–bit floating point unit, and is provided with four purpose–built bi–directional communication links which allow networks of transputers to be constructed by direct point to point connections with no external logic. The manufacturers specifications [61] state that a processor will achieve 10 MIPS with a 20 MHz clock, whilst the floating point unit operating concurrently with the processor provides single and double length arithmetic to the ANSI–IEEE 754–1985 standard, at a peak rate of 1.5 Mflops (32 bit) or 1.1 Mflops (64 bit). Each processor has 4Kbytes of cache memory, the interface to off–chip memory being 32 bits wide giving a maximum data–transfer rate of about 25 Mbytes/sec for a 20 MHz clock, whilst each bi–directional link has a maximum data–transfer rate of 2.35 Mbytes/sec.

Figure 4.1 shows our parallel architecture which is a software–reconfigurable NiCHE [67] transputer board, with which we communicate via a Sun workstation. It is a multi–user system capable of supporting up to four users simultaneously. Each user site consists of one T8 with 2 Mbytes of local memory, the "host" transputer, and four T8's each with 1 Mbyte of local memory. The processors within each site are linked in a fixed chain, giving each two free links which can be connected via NiCHE software commands to any processor on the board. Since we wish to use one processor as a control, the maximum network size that we will use is a 4×4 array of processors.

4.2 The Model Problem

Having described the parallel architecture available to us, we wish to consider the most appropriate parallel algorithms for the solution of the electrode problem on such a machine. In Chapter 2, we modelled the operation of the Clark electrode by a parabolic partial differential equation (p.d.e.), with complex boundary and interface conditions, for which we have no analytic solution. Rather than attempting to compare algorithms for solving this comparatively difficult problem, we will consider the parallel implementations of three well known algorithms: the simple explicit method; the odd–even hopscotch method of Gourlay [31, 32]; and the ADI (Alternating Direction Implicit) method of Peaceman and Rachford [79], for the solution of a simple model problem for which we know the exact solution. This will allow us to demonstrate clearly the accuracy and computational efficiency

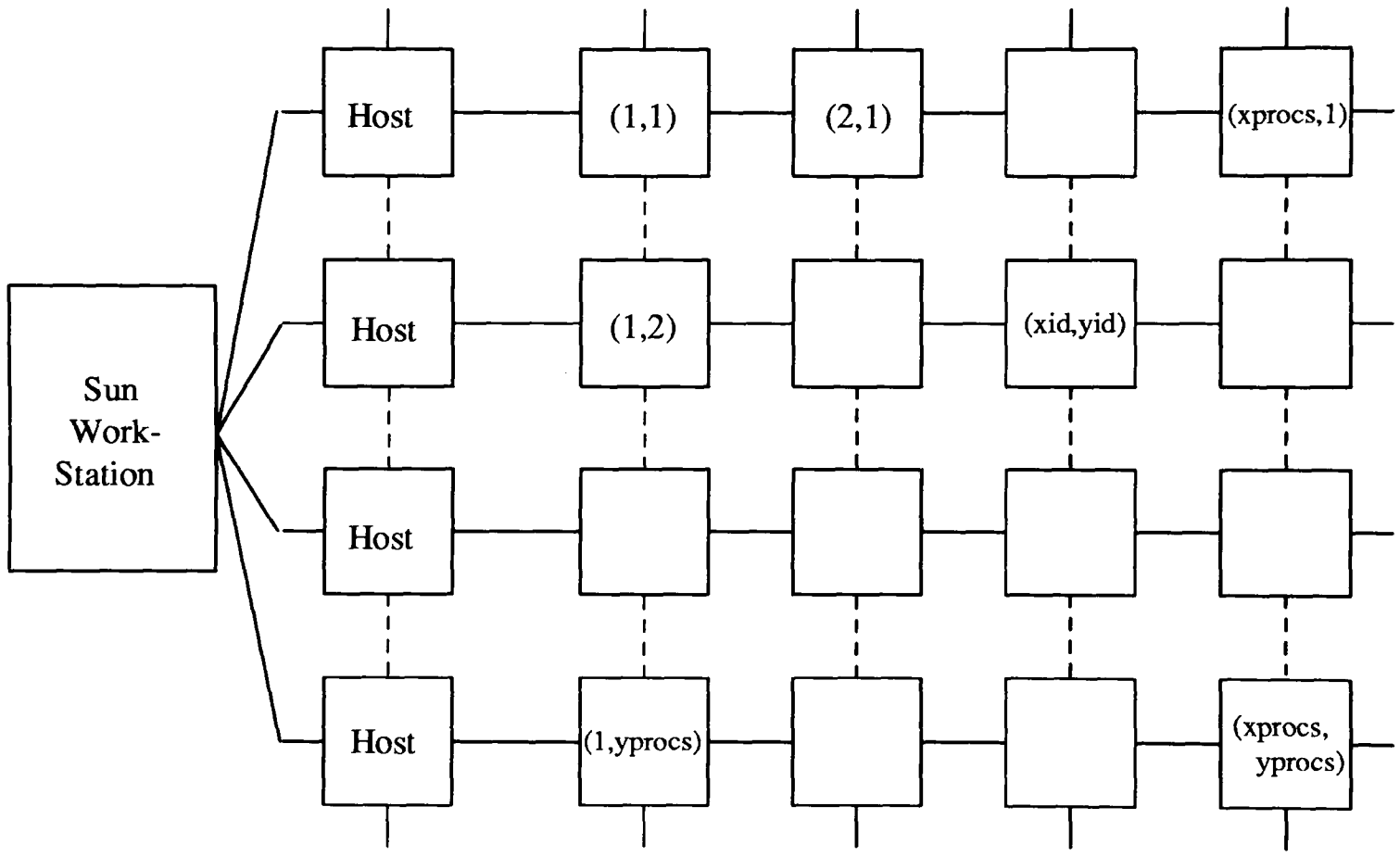


Fig. 4.1: The reconfigurable array of Inmos T800 transputers, where ——— represents a fixed link, - - - - a re–programmable link.

of each of the methods, and to highlight the difficulties involved in their parallel implementation.

The simplest two–dimensional parabolic equation that we can consider is the heat equation on the unit square with Dirichlet boundary conditions,

$$v_t = \nabla^2 v = \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) \quad (x,y)\, \epsilon\, \Omega\, (= [0,1] \times [0,1]) \tag{4.1}$$

with $v = f(x,y)$ on the boundary, $\delta\Omega$, and some initial condition $v(x,y,0) = g(x,y)$. We can solve the homogeneous problem straightforwardly by separation of variables to give

$$v(x,y,t) \;=\; \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} A_{mn} \sin(m\pi x) \sin(n\pi y) \exp(-(m^2+n^2)\pi^2 t) \tag{4.2}$$

If we take as our initial condition

$$g(x,y) = x(1-x)y(1-y), \tag{4.3}$$

we must take the constants A_{mn} in equation (4.2) to be the coefficients of the series expansion of $g(x,y)$ in terms of the eigenfunctions $\sin(m\pi x)\sin(n\pi y)$. This gives

$$v(x,y,t) = \sum_{p=0}^{\infty} \sum_{q=0}^{\infty} \left[\frac{64}{[(2p+1)(2q+1)\pi^2]^3} \sin((2p+1)\pi x) \sin((2q+1)\pi y) \right. \tag{4.4}$$
$$\left. \exp\left(-((2p+1)^2 + (2q+1)^2)\pi^2 t\right)\right]$$

as the solution of the homogeneous problem with initial condition $g(x,y)$. Since any bilinear function $h(x,y)$ satisfies $\nabla^2 h = 0$, we may obtain non–zero boundary conditions and a non–symmetric solution by adding any such function to the homogeneous solution. We therefore choose to solve the problem

$$\frac{\partial u}{\partial t} = \nabla^2 u \tag{4.5}$$

with initial condition

$$u(x,y,0) = 1 - y(1-x) - x(1-x)y(1-y) \tag{4.6}$$

and boundary conditions

$$\begin{aligned} u(0,y,t) &= 1-y \\ u(1,y,t) &= 1 \\ u(x,0,t) &= 1 \\ u(x,1,t) &= x \end{aligned} \tag{4.7}$$

in $\Omega \times [0 \leq t \leq T]$, as shown in Figure 4.2. This has exact solution

$$u(x,y,t) = 1 - y(1-x) - v(x,y,t), \tag{4.8}$$

chosen so that the boundary conditions are similar to those of the electrode problem.

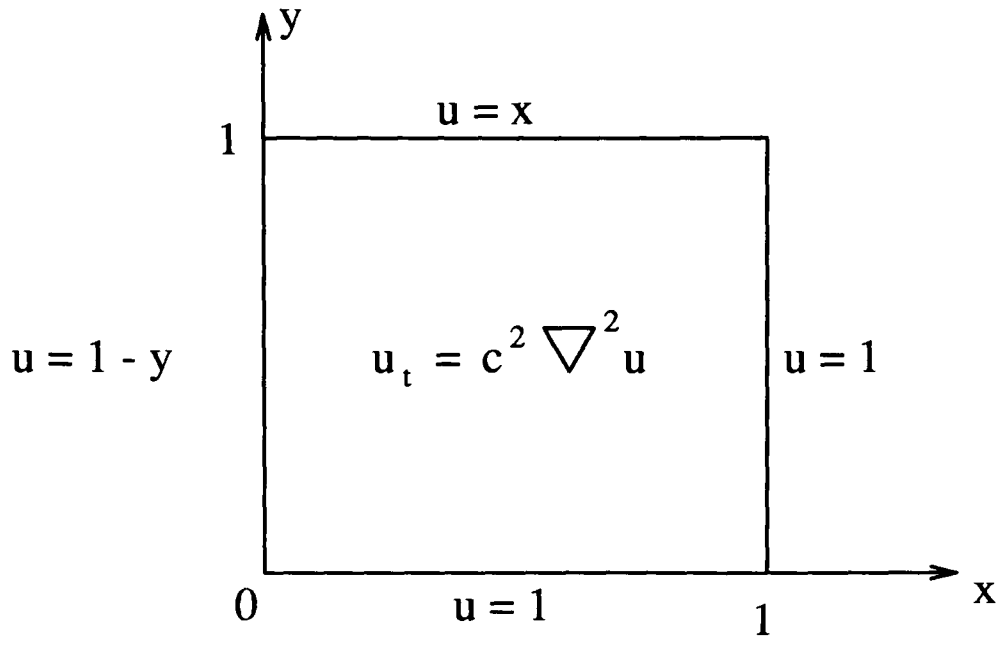


Fig. 4.2: The solution region and boundary conditions for the model problem.

4.3 The Sequential Algorithms

Superimposing a uniform $(\kappa+2)\times(\kappa+2)$ square mesh on Ω, with spacing $h = 1/\kappa$, we obtain the set of points $(ih, jh)\epsilon\Omega_h$, for i, j integers, and defining a timestep Δt, we require a numerical solution to equation (4.5) at the mesh points $\Omega_h \times \{n\Delta t\}$, where $n = 1, \ldots, N$ and $T = N\Delta t$.

In approximating the time derivatives in equation (4.5) we will use the forward difference formula

$$\frac{\partial u}{\partial t} \approx \frac{u_{i,j}^{n+1} - u_{i,j}^{n}}{\Delta t}. \tag{4.9}$$

The spatial derivatives are approximated using the three–point central difference formula

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{\delta_x^2}{h^2} = \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2}. \tag{4.10}$$

and similarly for y.

By choosing to evaluate the approximation to the spatial derivatives at the old (explicit) or new (implicit) time level, or a combination of both, we can obtain methods with differing degrees of complexity and numerical properties.

4.3.1 The explicit method

The explicit method simply uses the old values at time level n to approximate the new values at level $n + 1$ from

$$u_{i,j}^{n+1} = \left\{1 + \nu(\delta_x^2 + \delta_y^2)\right\} u_{i,j}^{n} \tag{4.11}$$

where $\nu = \Delta t/h^2$ is the *mesh ratio*. This requires 2(mult)+4(add) per mesh point. It is easy to show via Taylor series expansions that this finite difference scheme is

second order accurate (in space), but substitution of the Fourier mode gives the restrictive stability condition on the mesh ratio of

$$\nu \leq \frac{1}{4}. \tag{4.12}$$

By introducing varying degrees of implicitness it is possible to overcome this restriction.

4.3.2 The hopscotch method

The odd–even hopscotch method is usually associated with Gourlay [31] who extended an idea of Gordon [30]. Gordon's method involves splitting the mesh according to whether $(i+j+n)$ is odd or even. It is then possible to use the simple explicit formula (4.11) to update the "even" mesh points allowing the implicit formula

$$\begin{aligned} u_{i,j}^{n+1} &= u_{i,j}^{n} + \nu\left(\delta_x^2 + \delta_y^2\right) u_{i,j}^{n+1} \\ &= \frac{1}{1+4\nu}\left\{u_{i,j}^{n} + \nu(u_{i+1,j}^{n+1} + u_{i-1,j}^{n+1} + u_{i,j+1}^{n+1} + u_{i,j-1}^{n+1})\right\} \end{aligned} \tag{4.13}$$

to be used at the "odd" points. By introducing the mesh function

$$\theta_{ij}^{n} = \begin{cases} 1 \text{ if } (n+i+j) \text{ is odd} \\ 0 \text{ if } (n+i+j) \text{ is even} \end{cases} \tag{4.14}$$

Gourlay was able to write this scheme as

$$\left\{1 - \theta_{ij}^{n+1}\nu(\delta_x^2 + \delta_y^2)\right\} u_{i,j}^{n+1} = \left\{1 + \theta_{ij}^{n}\nu(\delta_x^2 + \delta_y^2)\right\} u_{i,j}^{n}. \tag{4.15}$$

He pointed out that by writing this equation at two successive time levels in the form

$$\begin{aligned} u_{i,j}^{n+1} - \nu\theta_{ij}^{n+1}(\delta_x^2 + \delta_y^2)u_{i,j}^{n+1} &= u_{i,j}^{n} + \nu\theta_{ij}^{n}(\delta_x^2 + \delta_y^2)u_{i,j}^{n}. \\ u_{i,j}^{n+2} - \nu\theta_{ij}^{n+2}(\delta_x^2 + \delta_y^2)u_{i,j}^{n+2} &= u_{i,j}^{n+1} + \nu\theta_{ij}^{n}(\delta_x^2 + \delta_y^2)u_{i,j}^{n+1} \\ &= 2u_{i,j}^{n+1} - \left\{u_{i,j}^{n} + \nu\theta_{ij}^{n}(\delta_x^2 + \delta_y^2)u_{i,j}^{n}\right\}, \end{aligned} \tag{4.16}$$

it can be seen that for $(n+i+j)$ an even integer equation (4.16) reduces to

$$u_{i,j}^{n+2} = 2u_{i,j}^{n+1} - u_{i,j}^{n}. \tag{4.17}$$

This allows him to define the following very fast computational algorithm:

(1)As startup step, overwrite $u_{i,j}^{n}$ for which $(i+j+n)$ is odd with $u_{i,j}^{n+1}$ using equation (4.11).

(2)If the values at each point at level $n+1$ are required, then overwrite $u_{i,j}^{n}$ for which $(i+j+n+1)$ is odd with $u_{i,j}^{n+1}$ using equation (4.13), otherwise go to step (3).

(3)The general step calculates $u_{i,j}^{n+1}$ for fixed (i,j) with $(i+j+n+1)$ odd using equation (4.13), then overwrites the value of $u_{i,j}^{n}$ with $u_{i,j}^{n+2}$ using equation (4.17).

If we now increment n by 1, we either return to step (2) if we require values at all mesh points at level $n+1$, or to the start of step (3) if not, and repeat.

The general step requires 2(mult)+4(add) altogether, but only calculates for half the mesh points. However we have the additional overhead of either a conditional or two integer operations to determine whether a mesh point is odd or even.

In a later paper [32], Gourlay points out that the odd–even hopscotch method is equivalent to the Dufort–Frankel scheme [17]. It is well known [91] that whilst such schemes are unconditionally stable, they become inconsistent as the mesh is refined unless the mesh ratio ν is kept constant. We will see that in practice this results in a loss of accuracy as the timestep is increased for a fixed mesh size.

4.3.3 The ADI method

The usual Crank–Nicholson [14] finite difference scheme which uses the average value at the old and new time levels to approximate the spatial derivatives is

$$u_{i,j}^{n+1} = u_{i,j}^{n} + \nu(\delta_x^2 + \delta_y^2)\frac{1}{2}(u_{i,j}^{n+1} + u_{i,j}^{n}). \tag{4.18}$$

This is an implicit scheme and requires the inversion of a matrix of large order at each timestep. Writing this as

$$\left\{1 - \frac{\nu}{2}(\delta_x^2 + \delta_y^2)\right\} u_{i,j}^{n+1} = \left\{1 + \frac{\nu}{2}(\delta_x^2 + \delta_y^2)\right\} u_{i,j}^{n}, \tag{4.19}$$

Peaceman and Rachford [79] suggested that this could be factored as

$$\left(1 - \frac{\nu}{2}\delta_x^2\right)\left(1 - \frac{\nu}{2}\delta_y^2\right) u_{i,j}^{n+1} = \left(1 + \frac{\nu}{2}\delta_x^2\right)\left(1 + \frac{\nu}{2}\delta_y^2\right) u_{i,j}^{n} \tag{4.20}$$

which introduces only an extra fourth order spatial difference term and leaves the method second order accurate overall. Introducing an intermediate time level at $n+1/2$ we can write equation (4.20) as

$$\begin{aligned} \left(1 - \frac{\nu}{2}\delta_x^2\right) u_{i,j}^{n+\frac{1}{2}} &= \left(1 + \frac{\nu}{2}\delta_y^2\right) u_{i,j}^{n} \\ \left(1 - \frac{\nu}{2}\delta_y^2\right) u_{i,j}^{n+1} &= \left(1 + \frac{\nu}{2}\delta_x^2\right) u_{i,j}^{n+\frac{1}{2}}. \end{aligned} \tag{4.21}$$

To obtain our solution at level $n+1$, we need now solve only tridiagonal systems in alternating directions at each half step. If we write either of equations (4.21) in the form

$$a_\nu x_{\nu-1} + b_\nu x_\nu + c_\nu x_{\nu+1} = d_\nu \text{ for } \nu = 1, \ldots, \kappa, \tag{4.22}$$

then these become $A\mathbf{x} = \mathbf{d}$ with

$$A = \begin{pmatrix} b_1 & c_1 & & 0 \\ a_2 & \ddots & \ddots & \\ & \ddots & \ddots & c_{\kappa-1} \\ 0 & & a_\kappa & b_\kappa \end{pmatrix}. \tag{4.23}$$

We could solve each tridiagonal system, using for example the Thomas algorithm since A is diagonally dominant. However A is a constant matrix in time, so that it is much more efficient to decompose each tridiagonal matrix into upper (U) and lower (L) triangular matrices such that $A = LU$ just once at the start of the calculation, and use them at each half time step to obtain the solution. It can easily be verified that

$$L = \begin{pmatrix} 1 & & & 0 \\ e_2 & \ddots & & \\ & \ddots & \ddots & \\ 0 & & e_\kappa & 1 \end{pmatrix} \quad U = \begin{pmatrix} f_1 & g_1 & & 0 \\ & \ddots & \ddots & \\ & & \ddots & g_{\kappa-1} \\ 0 & & & f_\kappa \end{pmatrix} \tag{4.24}$$

with

$$\left.\begin{array}{l} f_1 = b_1 \\ f_i = b_i - e_i c_{i-1} \\ e_i = a_i / f_{i-1} \end{array}\right\} i = (1, \ldots, \kappa) \ . \qquad \begin{array}{l} \\ \\ \\ g_i = c_i \quad i = (1, \ldots, \kappa - 1) \end{array} \tag{4.25}$$

It is important to note that since the mesh ratio ν is constant for all mesh points we obtain the same tridiagonal matrix for all mesh lines in either the x– or y–directions. This means that we need to do just one decomposition and store only three vectors $\mathbf{e}, \mathbf{f}, \mathbf{g}$ of length κ to calculate the solution for all mesh lines (although in practice we do one decomposition in each direction so that the code is more easily adapted to the more difficult electrode problem in which we are interested).

Once in the form $LU\mathbf{x} = \mathbf{d}$, by writing $\mathbf{z} = U\mathbf{x}$ we may solve the system $L\mathbf{z} = \mathbf{d}$ by forward substitution using

$$\begin{array}{rcll} z_1 = & d_1 & & \\ z_i = & d_i - e_i z_{i-1} & \text{for } i = 2, \kappa & \end{array} \tag{4.26}$$

and then obtain our solution $\mathbf{x}$ from $U\mathbf{x} = \mathbf{z}$ by backward substitution, using

$$\begin{array}{rcll} x_\kappa = & z_\kappa / f_\kappa & & \\ x_{\kappa-i} = & z_{\kappa-i} - g_{\kappa-i} x_{\kappa-i+1} / f_{\kappa-i} & \text{for } i = 1, \kappa - 1 & \end{array} \tag{4.27}$$

The forward sweep requires just 1(add)+1(mult) per mesh point, the backward sweep requires just 1(add)+1(mult)+1(div) per mesh point, and the right hand sides in equations (4.21) need 2(add)+4(mult), resulting in a highly efficient algorithm both in terms of calculation and storage. However, by virtue of the sweeping backwards and forwards across the whole mesh in different directions at each half time step, this algorithm appears to be inherently sequential.

We will describe later in this chapter how we can obtain an efficient parallel implementation of the ADI algorithm by re–ordering the calculations to solve the tridiagonal systems, whilst keeping the mathematical algorithm identical. We will illustrate the structure and nature of the parallel implementations of all three

methods by reference to parts of the Occam code. We will therefore first give a brief introduction to Occam to define those additional Occam constructs which would not be familiar to a programmer with experience of another high level language. This is based on examples and definitions given in the *Occam 2 Reference Manual* [58].

4.4 Introduction to Occam

The development of the Inmos Transputer and the Occam programming language have been closely related. Occam is a high level language specifically designed to allow the implementation of parallel algorithms on a network of processors, and is based on the mathematical concepts of *Communicating Sequential Processes* developed by Hoare [43].

Occam can also be regarded as a model which allows an application to be described as a collection of processes which execute concurrently and communicate with each other only over *channels*. This provides a simple and clearly defined program structure which can exploit the performance of a multi–processor system in an obvious way.

Since Occam and the transputer were designed and implemented concurrently, programming in Occam is nearly as efficient as programming in assembler on a conventional computer. However, compilers are now available for the transputer for other high level languages such as Fortran, Pascal and C. Whilst these compilers are much improved over earlier versions, the cost of programming in these familiar languages rather than Occam is still about a factor of two.

4.4.1 Occam constructs

The programming language Occam has been specifically designed to allow a set of distinct processes to run concurrently whilst providing them with the means to communicate with one another. The language syntax would look very familiar to a Pascal programmer, but it has several additional constructs and data types to support this concurrency.

An Occam program is made up of processes which act upon *variables*, *channels* and *timers*. Variables are assigned values by an *input* or *assignmment*, and values are communicated over channels. A timer produces a value representative of the time.

A value is classified by its *data type*, of which there are eight: **BOOL**, Boolean values; **BYTE**, integer values from 0 to 255; **INT**, signed integer values using the most efficient word size; **INT16**, 16 bit signed integer values (-2^{15} to $2^{15}-1$); **INT32**, 32 bit signed integer values; **INT64**, 64 bit signed integer values; **REAL32**, 32 bit or single length reals; and **REAL64**, 64 bit or double length reals.

Communication takes place over *channels*, of type **CHAN**, which may be used to communicate between processes running on the same processor, or between processes running on different processors. There are two types: *input*

keyboard ? char

which receives a value on the channel called **keyboard** and assigns it to the variable **char**; and *output*

screen ! char

which transmits the value of the variable **char** to the channel **screen**. The format and data type of the values communicated over a channel are defined by the channel *protocol*, which is specified when the channel is first declared. The simplest type of protocol consists of a primitive data type or array of data types, for example

```
CHAN OF [xnodes][ynodes]REAL64 arraychan:
```

so that the output

arraychan ! u

communicates a rectangular array of double length reals. More complex *variant* protocols can be declared to allow different data types to be communicated over the same channel, for example

```
PROTOCOL DATA
  CASE
    real; REAL64
    array; [xnodes][ynodes]REAL64
    int; INT
    :
```

This is then used to ensure the correct data type is being communicated, for example

```
CHAN OF DATA input,output:
  SEQ
    output ! real; x
    input ? CASE array; v
    :
```

declares two channels with protocol data, outputs the *tag* **real** to inform the receiving process of the format of the rest of the message, then outputs the real value **x**. The input process first receives the **array** tag using the **CASE** construct, and then receives the array **v**.

A channel can also be declared as type **ANY**, which can receive or transmit variables of any type as a byte stream, and converts the value to the type of the receiving variable. This type of protocol will be sufficient for the simpler communication structure required for the model problem, but for the more complex electrode problem we will use variant protocols.

All communications are synchronised and are therefore not completed until a transmitted value has been received.

A *procedure* definition defines a name for an Occam process. For example:

```
PROC increment(CHAN OF ANY input, output, INT x)
  SEQ
    input ? x
    output ! x+1
:
```

The procedure **increment** has three *formal parameters*, two channels of type **ANY**, and an integer variable **x**. **increment** inputs a value **x** on channel **input**, and transmits the value **x+1** on channel **output**.

The **SEQ** construct combines processes which are to be performed sequentially, for example

```
SEQ
  process1
  process2
  process3
```

in which **process3** is performed after the completion of **process2** after **process1**.

The **PAR** construct allows processes to be combined which are to run concurrently, for example

```
PAR
  westin ? new
  westout ! old
```

This receives a value on channel **westin** and assigns it to the variable **new**, whilst simultaneously transmitting the value **old** on channel **westout**. Both the **SEQ** and **PAR** constructs can be replicated and nested to form do–loops, for example

```
SEQ i=1 FOR xnodes
  SEQ j=1 FOR ynodes
    x[i][j] := x[i][j] + y[i]*z[j]
```

Whilst the **PAR** contruct allows processes to run concurrently within the same processor, the **PLACED PAR** construct assigns a process for execution on a specified processor. For example

```
PLACED PAR
  PROCESSOR 0
    master(input,output)
  PROCESSOR 1
    slave(input,output)
```

in which process **master** executes concurrently on processor 0 with process **slave** running on processor 1. Each process uses local memory and communicates with the other via channels **input** and **output**.

Each of the communication links on a transputer is associated with an integer value (0–7) as shown in Figure 4.3. A particular channel name may be associated with a physical link using the **PLACE command**,

```
VAL INT rightlinkin IS 6:
VAL INT rightlinkout IS 2:
PLACE eastin AT rightlinkin
PLACE eastout AT rightlinkout
```

where the **VAL** construct is used to specify a name for a constant value.

4.4.2 The decomposition

In order to obtain a parallel implementation of the three algorithms described above, we must first distribute the work load in some way amongst the processors. Since we calculate the solution at a square array of mesh points of size $(\kappa + 2) \times (\kappa + 2)$, there is an obvious geometrical decomposition of the problem onto a rectangular $p \times q$ network of processors by placing a $(\kappa+2)/p \times (\kappa+2)/q$ submesh on each.

Each of the algorithms updates using a five–point difference stencil, so that any point on the edge of a submesh requires the value of its immediate neighbour in the global mesh which is stored on the adjacent processor. Rather than communicate each of these values as they are required, we store an additional *halo* of points around the edge of each submesh. We can then communicate the required values from the edge of the submesh on the adjacent processor at the end of each timestep or half timestep, as illustrated in Figure 4.4. At the start of every timestep, each processor then has all the information it requires to update each mesh point at the new time level, in the case of the explicit and hopscotch schemes, or to form the right hand sides of equation (4.21) for each half step of the ADI method. We will refer to this process as *halo communication.*

4.4.3 The communications harness

We will make use of the **PAR** construct only when communicating between processors, where it greatly simplifies the programmers task. To implement the halo communication, as illustrated in Figure 4.4, we first assign the values along the edges of a submesh into four vectors **vn, vs, ve, vw** which are then concurrently broadcast using

```
PAR
  northout ! vn
  southout ! vs
  eastout  ! ve
  westout  ! vw
  northin  ? un
  southin  ? us
  westin   ? uw
  eastin   ? ue
```

which also simultaneously receives the corresponding values **un, us, ue, uw** from all four neighbours. Leland proves (Theorem 3.1 of [56]) by formal methods that

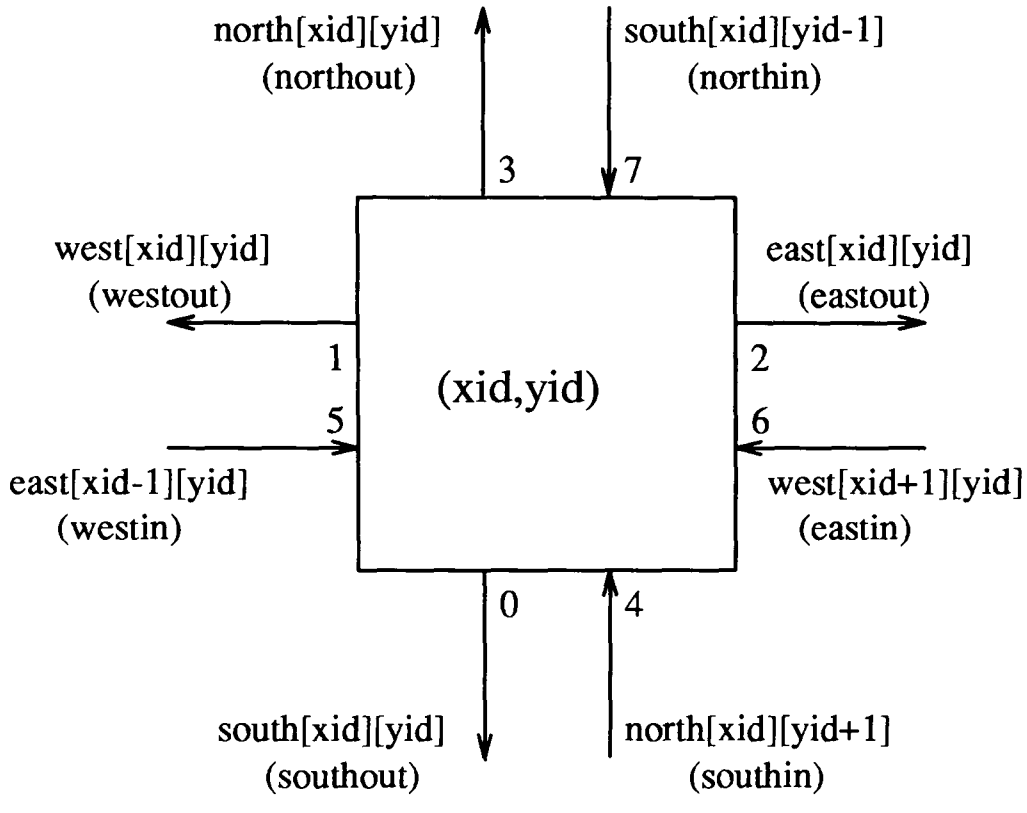


Fig. 4.3: The global and local (in brackets) names of the channels associated with each of the communication links (0–7) of a transputer in the network.

such a communication process is deadlock free in the context of communication between individual mesh points. Since we may consider the entire update process on a submesh as a similar nodal process provided it uses only the halo process to communicate (the four–way communication between individual mesh points and between complete submeshes are topologically equivalent), we may assume that the proof of deadlock freedom holds. In practice, the halo procedure also contains several guarding conditionals to deal with processors at the corners or edges of the network.

4.4.4 Configuration

The parallel implementation of all three algorithms takes the form of a single, identical Occam code which is loaded onto each processor. Each processor is supplied with its **(xid,yid)** coordinate in the **(xprocs** $\times$ **yprocs)** network (see Figure 4.1), so that the code can calculate the position of its submesh in the global mesh, and decide with which of its neighbours it needs to communicate. Each of these codes is regarded as one of a set of Occam processes (usually consisting of a number of sub–processes), which make up the components of an Occam *program.* The association of each of these component processes with a physical processor, and the mapping of the channels which interconnect the distributed processes onto the physical links between processors, is termed *configuration.*

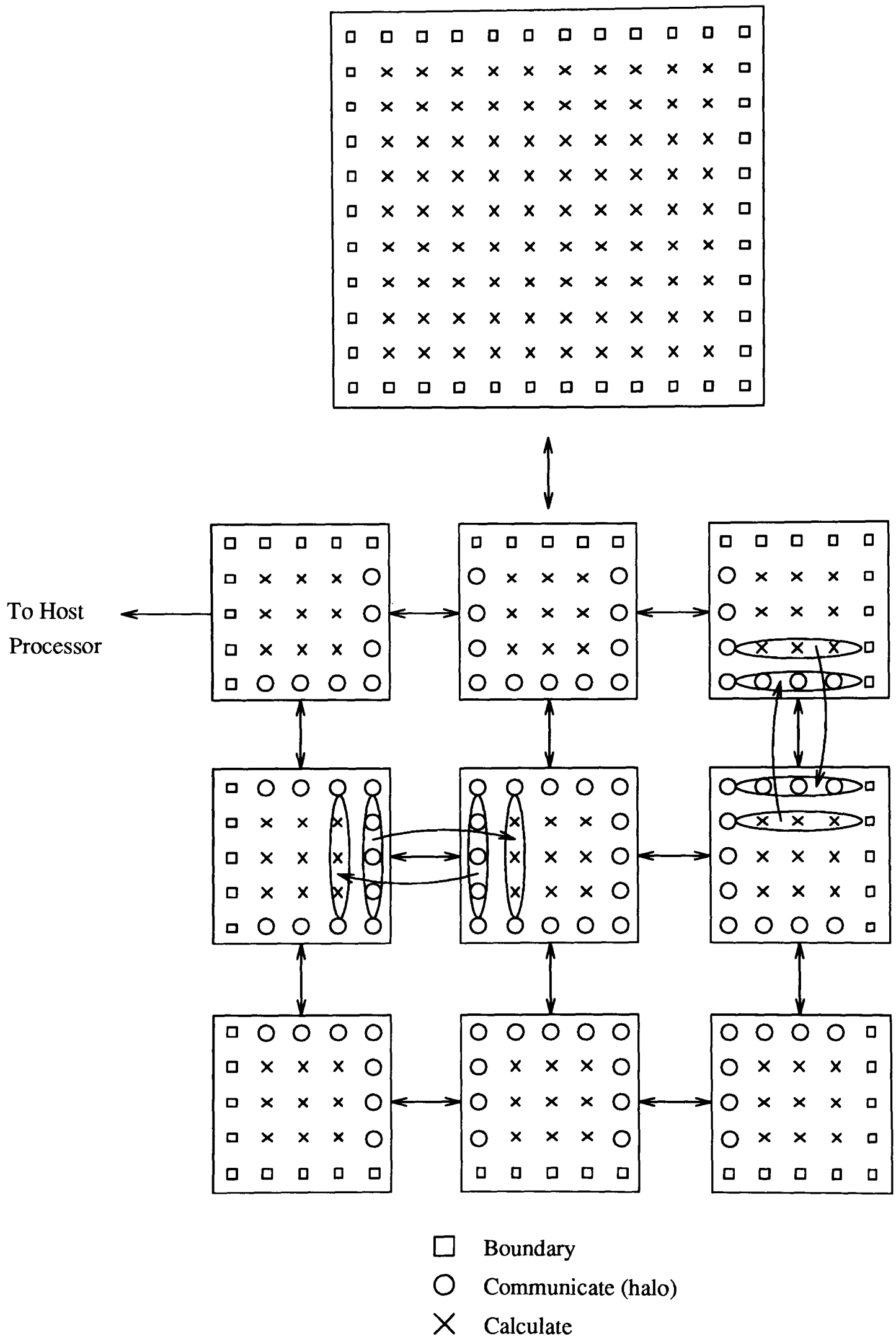


Fig. 4.4: The decomposition of a 9×9 global mesh onto a 3×3 array of processors, with an illustration of the *halo* communication process.

The geometrical decomposition of our problem onto the network requires that each processor can communicate bi–directionally with its four immediate neighbours. By assigning to each processor a unique identity

prid=(yid-1)*xprocs + (xid-1)

and defining an **(xprocs+1)** $\times$ **(yprocs+1)** array of (global) channels in each of the four directions **north, south, east, west**, we may configure our network of processors in the required manner using the replicated **PLACED PAR** construct below

```
[xprocs+1][yprocs+1]CHAN OF ANY north,south,east,west:
  PLACED PAR xid=1 FOR xprocs
    PLACED PAR yid=1 FOR yprocs
      VAL INT prid IS ((yid-1)*xprocs)+(xid-1):
      PROCESSOR prid T8
        PLACE west[xid][yid] AT 1:
        PLACE east[xid-1][yid] AT 5:
        PLACE east[xid][yid] AT 2:
        PLACE west[xid+1][yid] AT 6:
        PLACE south[xid][yid] AT 0:
        PLACE north[xid][yid+1] AT 4:
        PLACE north[xid][yid] AT 3:
        PLACE south[xid][yid-1] AT 7:
        adi(south[xid][yid-1],north[xid][yid],north[xid][yid+1],
        south[xid][yid],west[xid+1][yid],east[xid][yid],
        east[xid-1][yid],west[xid][yid],xid,yid)
```

This configuration statement places the procedure **adi** with ten formal parameters (the eight required channels and the processor coordinates) onto each processor in a rectangular array of any size, and maps channels onto links as shown in Figure 4.3. It is worth noting that like other high–level languages, Occam allows local names for the formal parameters of a procedure to be used. Within the procedure for each of our algorithms we therefore choose the local names **northin, northout, southin,...** etc. with correspondence to the global channels as in Figure 4.3.

4.5 Parallel Implementations

In implementing all three algorithms in parallel we load a single, *sequential* Occam process onto each processor in the network, the **PAR** construct being used only within the halo procedure. These execute concurrently with a monitoring process on the host transputer which receives results from the network and handles i/o with the host system (in our case a network of Sun workstations). All three programs set up and initialise the appropriate submesh within the global mesh, communicate the halo of initial values and begin the timestepping. All real values are double length. Both the explicit and hopscotch methods can then be fully implemented by

the insertion of the halo process at appropriate points. The inherently sequential ADI algorithm, however, must be completely redesigned and requires additional communications before an efficient parallel implementation can be obtained (we will leave until the next section any explanation of what we mean by "efficient").

4.5.1 The explicit method

Since the explicit method calculates the value at the new time level using only values at the old, its parallel implementation is very straightforward. We simply write a sequential code to execute the sequential algorithm defined by equation (4.11) on the submesh contained on each processor, then communicate the halo values at the end of each timestep. The Occam code loaded onto each processor is of the form

```
PROC expl(CHAN OF ANY northin,northout,southin,southout,
          eastin,eastout,westin,westout,VAL INT xid,yid)
  ... declare.variables
  ... PROCEDURES
  SEQ
    initialise()
    SEQ n=1 FOR onsteps
      SEQ
        explicit.node()
        halo(u)
    output.results
:
```

where **halo** is the name of our communications procedure and has as its one formal parameter **u**, of type **[xnodes+2][ynodes+2]REAL64**, the array containing the values in the halo and at each point in the submesh. All mesh points are updated by procedure **explicit.node()** at each timestep, which implements equation (4.11).

4.5.2 The hopscotch method

The implementation of the hopscotch algorithm is very similar, except that we must perform the startup step (1) of the algorithm before beginning the timestepping, and communicate the halo after this step, and after each part of the update procedure defined by step (3) of the algorithm. We must then perform step (2) of the algorithm before outputting the results. The odd–even relationship between adjacent mesh points must be maintained between all points, including those held on different processors. The initialisation procedure therefore assigns the value 1 or 0 to the integer variable **skip** according to whether the first node in the submesh is odd or even. We could then decide the nature of a mesh point on any particular timestep when implementing either part of the general update step (3), by using a conditional which determines whether $(i + j + n + skip)$ is odd or even. However, since conditionals are comparatively expensive,

```
PROC hops(CHAN OF ANY northin,northout,southin,southout,
          eastin,eastout,westin,westout,VAL INT xid,yid)
  ... declare.variables
  ... PROCEDURES
  SEQ
    initialise()
    explicit.node()              (startup step (1))
    halo(u)
    SEQ n=1 FOR onsteps          (general step (3))
      SEQ
        implicit.node()
        halo(u)
        update.node()
        halo(u)
        skip:=1-skip             (switch the global value skip)
    last.implicit.node()         (step (2))
    output.results
:
```

we explicitly operate on only half of the (sub–) mesh points in any column using integer operations. The procedure shown below implements the second half of step (3) of the algorithm.

```
PROC update(INT skip)
  INT procskip:
  SEQ i=1 FOR xnodes
    SEQ
      procskip:=skip
      SEQ jj=1 FOR halfynodes+(odd*procskip)
        j:=(2*jj)-procskip
        u[i][j]:=(2.0(REAL64)*v[i][j])-u[i][j]
      procskip:=(1-procskip)     (switch the local value procskip)
:
```

The global integer variable **odd** is given the value 1 or 0 according to **ynodes** being odd or even. By flipping the value of the local variable **procskip** between 1 and 0 for alternate columns of mesh points, we can ensure that we update only the odd points at a particular timestep at which we hold the global variable at **skip**=1. Switching the global value to **skip**=0 on the subsequent timestep ensures that we update the other half of the mesh points, the even points.

4.5.3 The ADI method

At first inspection the sequential ADI algorithm allows no obvious method of decomposition into a parallel algorithm. The solution of the tridiagonal system for each mesh line in the x–direction requires that we sweep forwards and then backwards across each whole mesh line, then we switch direction and sweep forwards and backwards for each mesh line in the y–direction. We do not wish to go to a substructuring algorithm such as that described by Wang [95] or Johnsson *et*

al [49], since the efficiency of both storage and calculation in the sequential algorithm is achieved by performing the decompositions only once. Nor is an attempt to decompose the mesh onto the transputers as groups of rows or columns feasible since communication costs would be prohibitive. Instead we use the same geometrical decomposition of the global mesh as for the explicit and hopscotch methods, but re–order the sequence of the calculations in the ADI algorithm defined in Section 4.2, to obtain an efficient parallel implementation.

The inherently sequential parts of our algorithm are defined by equations (4.25), (4.26), and (4.27), each of which calculates each of the components of a set of vectors in turn from the values of the immediately preceding components. The parallel implementation of equations (4.25) which calculate the three vectors $\mathbf{e}, \mathbf{f}, \mathbf{g}$ making up the decomposed matrices L and U in the x–direction, is illustrated in the Occam code below.

```
IF
  xid=1        (processor in first column)
    SEQ
      fx[1]:=bx[1]
      gx[1]:=cx[1]
      SEQ i=2 FOR xnodes-1
        SEQ
          ex[i]:=ax[i]/fx[i-1]
          fx[i]:=bx[i]-(ex[i]*cx[i-1])
          gx[i]:=cx[i]
      IF
        xid=xprocs                    (processor in last column)
          SKIP                        (do nothing)
        TRUE                          (otherwise or else)
          eastout ! fx[xnodes]        (output last f cpt)
TRUE      (processor not in first column)
  SEQ
    westin ? fx[0]      (input first f cpt)
    SEQ i=1 FOR xnodes
      SEQ
        ex[i]:=ax[i]/fx[i-1]
        fx[i]:=bx[i]-(ex[i]*cx[i-1])
        gx[i]:=cx[i]
    IF
      xid=xprocs
        SKIP
      TRUE
        eastout ! fx[xnodes]
```

The decompositions start in the x–direction on the first processor in each column (**xid=1**) of the network, sweeping forwards across the processor array, communicating the required last value of **f** (**fx[xnodes]** in the code) on a processor,

Proc.																
(1,1)	F	F	F	w	w	B	w	B	w	B	D	D	D	w	w	...
(1,2)	w	F	F	F	B	w	B	w	B	D	D	D	w	w	U	...
(1,3)	w	w	F	B	F	B	F	B	D	D	D	w	w	U	w	...
(2,1)	F	F	F	w	w	B	w	B	w	B	w	D	D	D	U	...
(2,2)	w	F	F	F	B	w	B	w	B	w	D	D	D	U	w	...
⋮																

Table 4.1: Summary of work of processors in a 3×3 processor array, each with a 3×3 submesh, as illustrated in Figure 4.5. F=forwards sweep, B=backwards, D=downwards, U=upwards and w=wait. The steps at which the processors are working clearly dominate, even for this small submesh.

which becomes the first value of **f** (**fx[0]**) on the adjacent processor until the decomposition is complete. The process is then repeated in the y–direction sweeping down the network starting from the first processor in each row. Each processor stores only those parts of the vectors making up L and U that it requires to calculate the solution on its own submesh. Whilst this process can proceed on only one row or column of processors, it need be done only once at the start of the calculation, and processors can begin the timestepping algorithm once their portion of the decomposition is complete.

The timestepping algorithm uses a similar idea, as illustrated in Figure 4.5. We wish to obtain the solution at time level $n + \frac{1}{2}$ from that at level n by solving a tridiagonal system for each mesh line in the x–direction. At the first half step we begin the forward sweep to find **z** for the first (sub-) mesh line in the x–direction on each processor in the first processor column using equations (4.26). When these processors reach the end of their first x–mesh line, they each communicate their final value of **z** to the adjacent processor in the second column, which then begin their first mesh line in the x–direction. This is exactly the same as when finding **f** except that the first processors can now start the forward sweep on their second mesh line and are not therefore idle.

The calculation proceeds across the network until all processor columns are working (after (**xprocs-1**) stages). The first (**xprocs-1**) processors in each row of the network calculate the forward sweep for **z** for every x–mesh line (storing it over the old solution), whilst the last processor sweeps forwards to calculate **z** then immediately backwards along each x–line in turn to calculate the solution using equations (4.27). By counting the number of completed x–lines on the last processor in each processor row and buffering the information transmitted by the next to last, we can ensure that this last processor is always ready to receive a **z** value or transmit a solution value as required.

Once the next to last processor in each row has finished all of its mesh lines in

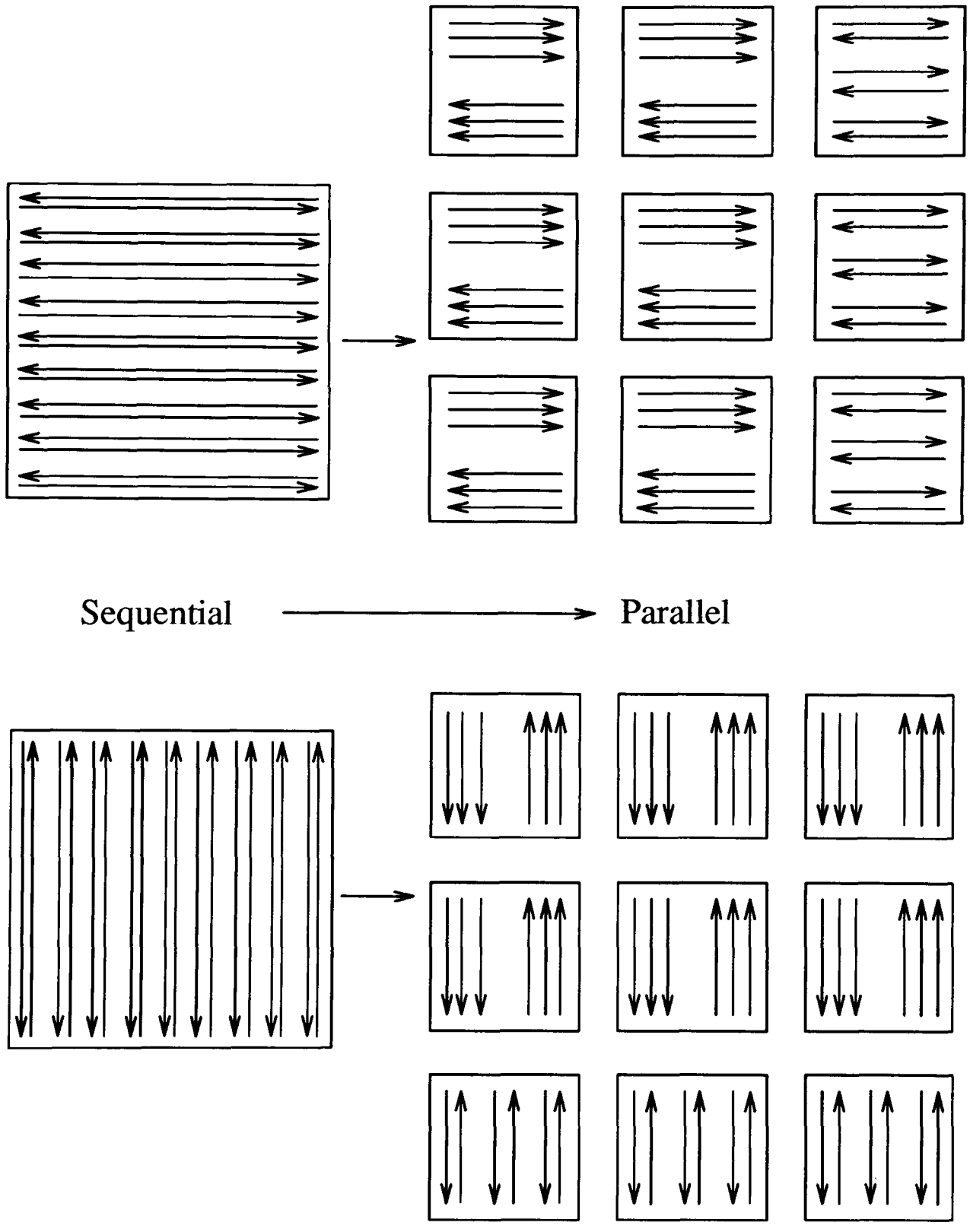


Fig. 4.5: The sequential algorithm sweeps forwards then backwards across each row in turn, then down and up each column. The parallel algorithm sweeps forwards for all rows on each processor except the last, which treats its submesh as the sequential case, sweeping forwards then immediately backwards, then the other processors sweep back across each row. The process is repeated to obtain the solution in the y–direction.

the x–direction, it inputs the last element of the solution vector stored on the last processor, and begins the backwards sweep across each mesh line in turn, until all lines in the x–direction are completed. Each processor then communicates the newly updated values via the halo process, and repeats the entire process in the y–direction. We therefore maintain $O(\kappa)$ communication whilst calculation is $O(\kappa^2)$, so that provided we have a large enough sub mesh, the algorithm should be efficient in parallel, an efficiency which is gained not by replacing the sequential tridiagonal solver by a slower "parallel" one, but by pipelining the solutions of successive tridiagonal systems. Table 4.1 summarises the work load of each processor as the calculation progresses.

4.6 Parallel Efficiency

Throughout the foregoing we have used terms such as "expensive" and "efficient" in a very loose sense. In this section we will clearly define and quantify what we mean by these expressions and demonstrate the greater validity of Leland's new parallel *efficiency* model. In the next section we will extend his parallel *effectiveness* model to cover a wider class of numerical algorithms and then use it to determine which of the above algorithms is most suitable for a given set of circumstances.

The term *parallel efficiency* is usually used to refer to the *speedup gained per processor applied* in a multicomputer. Whilst it is clearly desirable that it should be as close to unity as possible, it is normally understood that it can never exceed unity. Leland showed that this need not necessarily be the case.

Following the notation introduced by Leland, we define $T_{\text{exec}}(P)$ to be the execution time for a particular problem on a P–processor multicomputer. Each individual processor spends its time either calculating, communicating or idling so that

$$T_{\text{exec}}(P) = T_{\text{calc}}(P) + T_{\text{com}}(P) + T_{\text{idle}}(P) \tag{4.28}$$

and, if we neglect any i/o with the host system, $T_{\text{exec}}(1) = T_{\text{calc}}(1)$. If we now assume that

$$T_{\text{calc}}(P) = \frac{T_{\text{calc}}(1)}{P} \tag{4.29}$$

then we obtain

$$\begin{aligned} T_{\text{exec}}(P) &= \frac{T_{\text{exec}}(1)}{P}\left(1 + \frac{T_{\text{com}}(P)}{T_{\text{calc}}} + \frac{T_{\text{idle}}(P)}{T_{\text{calc}}}\right) \\ &= \frac{T_{\text{exec}}(1)}{P}\left(1 + f_{\text{com}} + f_{\text{idle}}\right), \end{aligned} \tag{4.30}$$

where f_{com} and f_{idle} are the overhead factors. We may now define the gain in speed in going from 1 to P processors to be

$$S(P) = \frac{T_{\text{exec}}(1)}{T_{\text{exec}}(P)} \tag{4.31}$$

giving

$$\epsilon' = \frac{S(P)}{P} \tag{4.32}$$

as the *measured* parallel efficiency. Substituting into equation (4.31) for the execution times we obtain

$$\epsilon_{\text{old}} = \frac{1}{1 + f_{\text{com}} + f_{\text{idle}}} \leq 1 \tag{4.33}$$

as a simple *model* for the parallel efficiency.

Leland pointed out that the assumption that the time spent in calculating the solution of a given problem on one processor must be P times that on P processors, is not in general true. He therefore replaces equation (4.29) with

$$T_{\text{calc}}(1) = \eta_{\text{calc}} P T_{\text{calc}}(P) \tag{4.34}$$

where η_{calc} is the *calculation efficiency*, and gives three factors which might cause η_{calc} to differ from unity.

4.6.1 Sweep efficiency

The sweep efficiency is a measure of the additional complexity introduced into an algorithm by its parallel implementation. In general when numerically solving a partial differential equation, we must perform a certain number of operations ($o(\kappa^2)$) per mesh point, additional operations ($o(\kappa)$) for points next to a boundary or interface or due to line solvers in methods such as ADI, plus the number of overhead operations ($o(1)$) required to set up the calculation, which is independent of mesh size. Leland therefore defines a weighted complexity per iteration (or timestep) Φ on one processor as

$$\Phi(1, \kappa^2) = a\kappa^2 + b\kappa + c \tag{4.35}$$

which gives the actual cost of the calculation, not just the operation count.

On a network of P processors, each submesh will contain κ^2/P points and therefore

$$\Phi(P, \kappa^2) = a\frac{\kappa^2}{P} + b\sqrt{\frac{\kappa^2}{P}} + c + d, \tag{4.36}$$

where d is the additional overhead incurred in obtaining the parallel solution (for instance we must determine the odd or even nature of the first node of the submesh when using the hopscotch method).

Combining these we obtain

$$\eta_{\text{sweep}} = \frac{T_{\text{calc}}(1)}{P T_{\text{calc}}(P)} = \frac{a\kappa^2 + b\kappa + c}{P(a\frac{\kappa^2}{P} + b\sqrt{\frac{\kappa^2}{P}} + c + d)} \tag{4.37}$$

so that $\eta_{\text{sweep}} < 1$, unless $b = c = d = 0$ which is not in general true.

4.6.2 Convergence efficiency

The software system used by Leland, which he called ONDE (Occam Nodal Declaration Environment), attempts to create an environment in which parallel algorithms for a particular class of problems can be implemented and tested in a

manner which is true to the Occam *model* of parallelism. This allows certain theorems about deadlock freedom within the Environment to be stated and proved using formal methods of computer science. The class of problems which Leland considers is the iterative solution of large, sparse, linear systems, particularly those occurring in the solution of elliptic partial differential equations. Within ONDE, the calculation to update each mesh point (or *node*) is treated as a separate Occam process, which communicates with the processes to update neighbouring nodes over channels. The software allows each of these nodal processes to communicate with its immediate neighbours in exactly the same manner, regardless of whether the neighbour is held on the same processor, freeing the user from concern about the physical structure of the processor network. Each complete update of the global mesh is then achieved with each nodal process executing concurrently with all others, the calculation being synchronised at the end of each update.

One of the consequences of this approach is that the programmer has no control over the order in which the calculations take place for a given nodal update, which may affect the convergence rate for the type of iterative method that Leland considers. He therefore defines convergence efficiency to be

$$\eta_{\text{conv}} = \frac{\text{serial convergence}}{\text{parallel convergence}} = \frac{\log \rho(1, \kappa^2)}{\log \rho(P, \kappa^2)}, \tag{4.38}$$

where ρ is the spectral radius of the relevant iterative algorithm. As Leland points out, if the spectral radii of the methods differ, the algorithms are mathematically distinct. This is one of the drawbacks of the ONDE system, the others being programming complexity and the cost of using concurrent processes within a single processor (a cost which will be quantified later), which must be weighed against the advantages of being able to use formal methods in the analysis of the test algorithms.

The parallel implementation of the methods described in the previous section uses the halo communication process with a sequential update procedure for all mesh points within a single processor, so that the calculations are ordered in the same manner as in the sequential algorithms. Therefore whilst we will define a measure of the work required to obtain a given accuracy for parabolic partial differential equations which corresponds to the spectral radius, this parameter will be assumed to have the same value for both the sequential and parallel implementations, so that η_{conv} is taken to be unity.

4.6.3 Memory efficiency

The access time for a particular memory location is dependent on memory size. Simplistically, if the memory on a chip has M bits arranged in a two–dimensional array, then the length of the data/address lines might be expected to be proportional to $\sqrt{M}$. Leland therefore defines t_{mem} to be the conversion factor between length and time, and t_{arith} the time required to apply a floating point algorithm to

Method	$T_{\text{exec}}(1)$ (s)	$T_{\text{exec}}(16)$ (s)	$S(16)$	ϵ'
Explicit	260.8	16.0	16.33	1.02
Hopscotch	202.3	13.1	15.44	0.96
ADI	755.5	62.3	12.12	0.76

Table 4.2: Comparison of execution time, measured speedup, and measured efficiency of the three algorithms.

operands already placed in registers, so that the relative time required to do the same floating point calculation can be modelled by

$$\frac{t_{\text{calc}}(1)}{Pt_{\text{calc}}(P)} = \frac{t_{\text{arith}} + 2\kappa t_{\text{mem}}}{t_{\text{arith}} + 2\sqrt{\frac{\kappa^2}{P}}t_{\text{mem}}}. \tag{4.39}$$

t_{arith} and t_{mem} are usually of comparable order so that for large P

$$\eta_{\text{mem}} = \frac{t_{\text{calc}}(1)}{Pt_{\text{calc}}(P)} \sim \sqrt{P}. \tag{4.40}$$

This implies that for a sufficiently large number of processors we might hope for superlinear speedup.

Combining these three types of calculation efficiency gives

$$\eta_{\text{calc}} = \eta_{\text{sweep}}.\eta_{\text{conv}}.\eta_{\text{mem}} \tag{4.41}$$

and replacing equation (4.29) with equation (4.34) leads to Leland's new model for ϵ_{new}

$$\epsilon_{\text{new}} = \left(\frac{1}{\eta_{\text{calc}}} + f\right)^{-1} \tag{4.42}$$

where $f = f_{\text{com}} + f_{\text{idle}}$. The condition for the superlinear speedup ($\epsilon_{\text{new}} > 1$) that we have proposed is

$$\eta_{\text{calc}} > \frac{1}{1-f}.$$

4.6.4 Efficiency of the three methods

Due to the large software overheads and the amount of memory required by the code, Leland did not observe superlinear speedups using the ONDE system. Indeed, in general, the memory efficiency effect is obscured and is usually swamped by the communication overheads, since each processor has the same amount of memory and the hardware cycle time of the under full memories cannot be reduced. However, for algorithms with minimal communications overhead such as the explicit algorithm, it is achievable even on our 4×4 network of processors. Table 4.2 shows the times taken, the speedup and the measured efficiency for the three algorithms in solving the model problem on a 240×240 mesh (this is the largest problem that will fit into the memory of a single processor) using a mesh

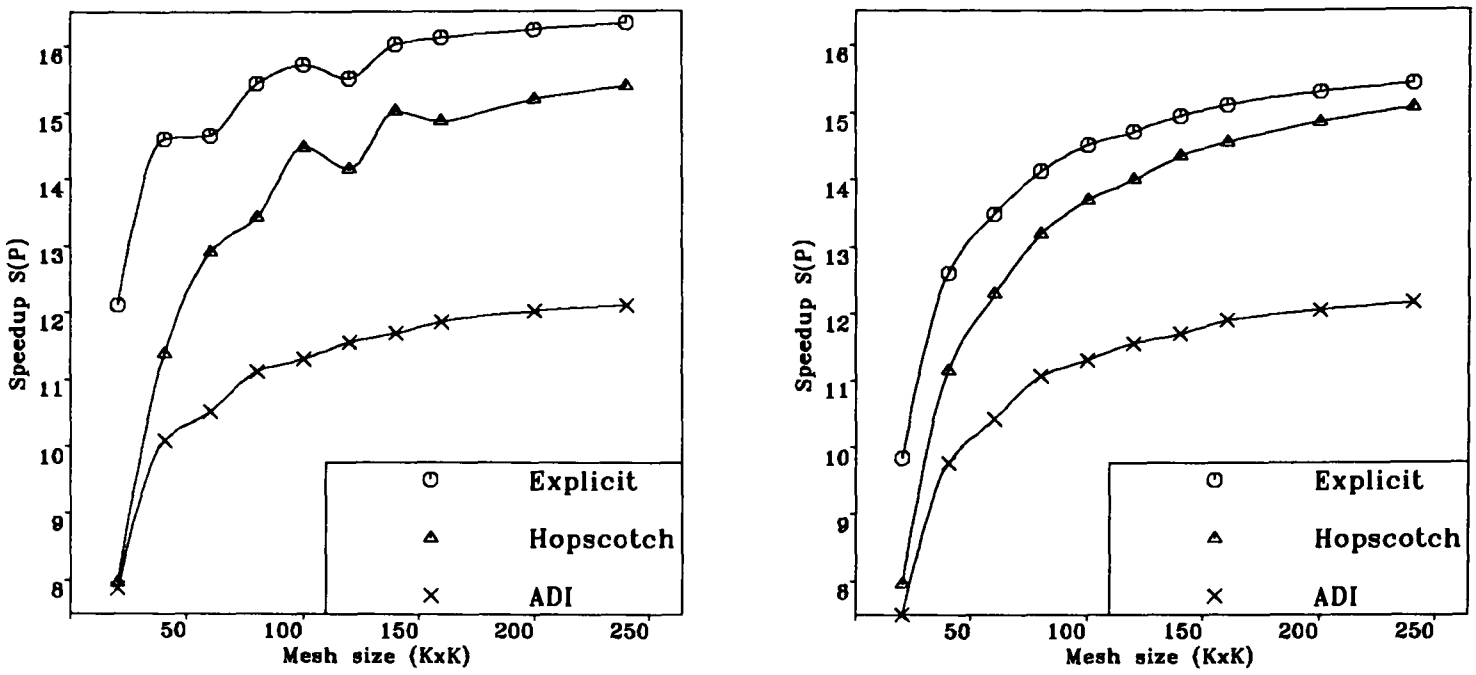


Fig. 4.6: Increase in speedup with mesh size (a) Without using abbreviations to save on memory access time and (b) With abbreviations.

ratio $\nu = 0.25$ (the maximum allowable for stability for the explicit method) after 256 timesteps on one and sixteen processors respectively. The explicit method achieves a speedup of just over 16, whilst both the hopscotch and ADI methods with their greater communication loads, give sublinear speedups.

The variation of the measured speedup $S(16)$ with mesh size κ in going from 1 to 16 processors is illustrated in Figure 4.6(a) and Table 4.3. The times for both inter–processor communications and off–chip memory access are expected to increase smoothly with mesh size, so that we would expect a smooth rise in

Mesh	Explicit			Hopscotch			ADI		
$\kappa \times \kappa$	T_{exec}	$S(16)$	ϵ'	T_{exec}	$S(16)$	ϵ'	T_{exec}	$S(16)$	ϵ'
20	0.14	12.1	0.757	0.17	7.96	0.497	0.67	7.87	0.492
40	0.47	14.6	0.914	0.48	11.4	0.712	2.07	10.1	0.630
80	1.83	15.4	0.965	1.65	13.5	0.841	7.49	11.1	0.630
120	4.13	15.5	0.970	3.53	14.2	0.886	16.3	11.6	0.723
160	7.20	16.1	1.006	6.05	14.9	0.930	28.3	11.9	0.742
200	11.2	16.2	1.014	9.26	15.2	0.949	43.7	12.0	0.751
240	16.0	16.3	1.021	13.1	15.4	0.963	62.3	12.1	0.758

Table 4.3: Increase in speedup and efficiency on 16 processors with increasing mesh size for the three algorithms, together with the execution time in seconds on 16 processors.

$S(16)$ with increasing κ. This is clearly not the case, particularly for the explicit and hopscotch methods. The reason for this can be seen by examining in greater detail the nodal processes described in Section 4.5, concentrating for clarity on the simpler explicit method. The update procedure for the explicit method at each timestep is given by equation (4.11) and is written in Occam as

```
PROC exp.node()
  SEQ
    nu1:=1-nu
    SEQ i=1 FOR xnodes
      SEQ j=1 FOR ynodes
        SEQ
          v[i][j]:=(nu1*u[i][j])+(nu*((u[i+1][j]+
          u[i-1][j])+(u[i][j+1]+u[i][j-1])))
    SEQ i=1 FOR xnodes
      SEQ j=1 FOR ynodes
        SEQ
          u[i][j]:=v[i][j]
:
```

which uses the minimum number of floating point operations, and is immediately transcribable into any of the other common high–level languages. However, the update of each node requires 8 memory locations to be addressed in the two two–dimensional arrays **u, v** which contain the old and new values, respectively, at each point in the mesh. The calculation of each address requires an integer multiply in the CPU which can be done using the fast multiply **TIMES** since array indices in Occam are non–negative. The time taken for this operation, and therefore of the address calculation, is dependent on the size of the array index in bits ([60, 62]). The explicit method has an (*address calculation*) : (*floating point calculation*) ratio of 4 : 3, so that we might expect a significant increase in the time taken for lookup in off–chip memory each time the storage of the indices requires an extra bit of memory. This would be displayed as an increase in the execution time at $(\kappa+2) =$ 16, 32, 64 and 128 (for $\kappa < 16$ we can fit both arrays into the 4096 Kbytes of cache memory). This effect can be investigated by considering a model of the overall execution time on one processor, $T_{\text{exec}}(1)$, for various meshes. In modelling $T_{\text{exec}}(1)$ we include a term to account for the time spent updating each mesh point at each timestep ($a\kappa^2 t_{\text{steps}}$), one for the overhead per timestep (bt_{steps}), one for the initialisation of each mesh point ($c\kappa^2$), plus the cost of the global setup (d), to obtain

$$\tau_{\text{exec}}(1) = t_{\text{steps}}(a\kappa^2 + b) + c\kappa^2 + d, \tag{4.44}$$

as our simple model for the execution times.

Estimates of the values of the constants a, b, c, and d from a least squares fit of the execution times (measured in clock cycles of $64\mu s$) obtained on 30 different

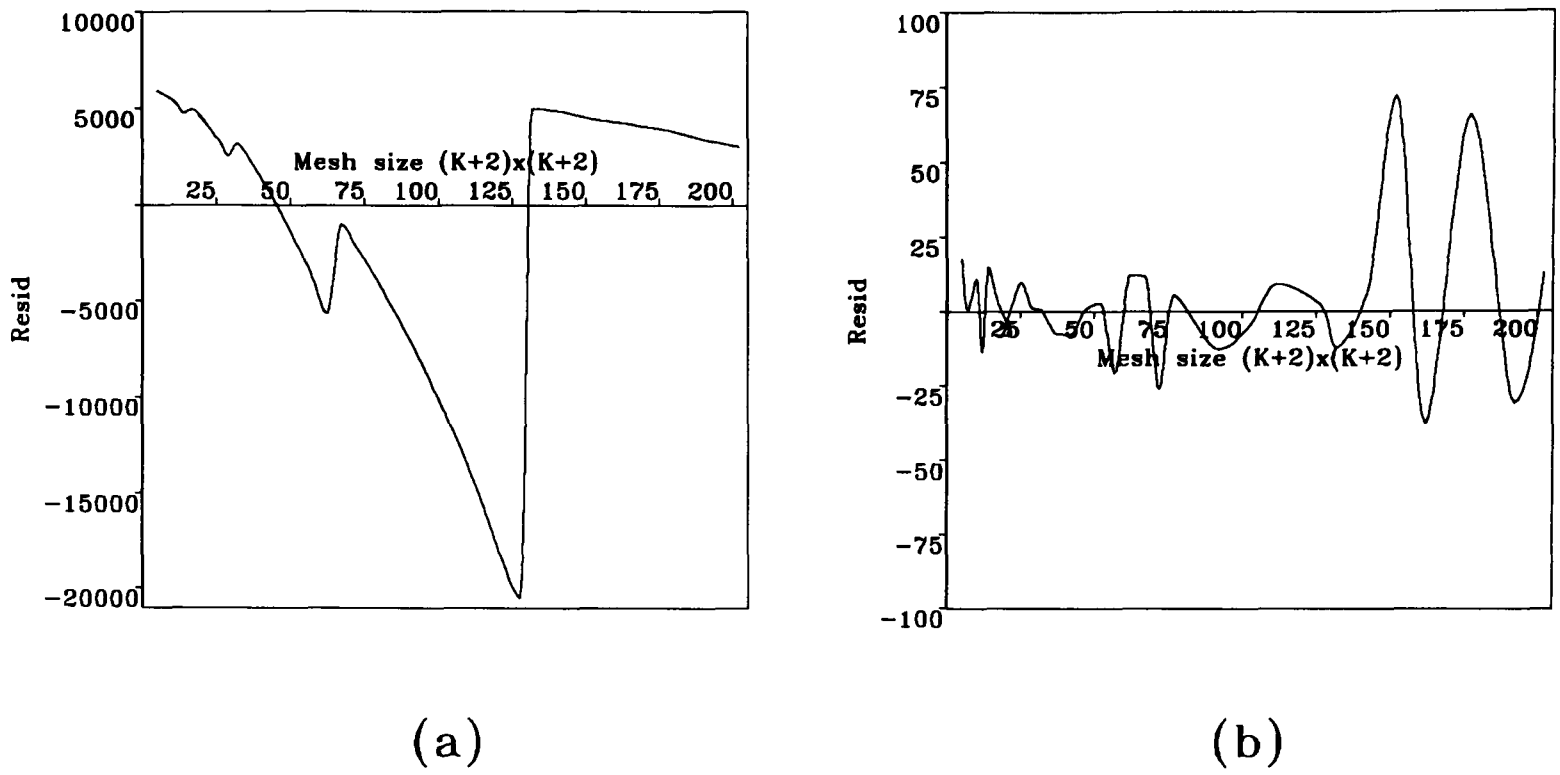


Fig. 4.7: Fitted residuals of least squares fit of execution times on increasing mesh sizes after 256 timesteps, (a) Data unpartitioned and (b) Data partitioned in powers of two as defined in equation (4.45).

	estimate	s.e.	parameter
d	68.58	255.6	1
c	0.3541	0.01841	κ^2
b	-23.55	2.318	t_{steps}
a	0.2753	0.0001691	$\kappa^2 t_{\text{steps}}$

Table 4.4: Least squares estimates of the constants and standard errors (s.e.) from the fit of the execution times (in clock cycles) for the explicit method using the simple model defined by equation (4.44).

	estimate	s.e.	parameter
d	11.85	1.948	1
c_1	0.2740	0.0304	$M(1).\kappa^2$
c_2	0.3443	0.0048	$M(2).\kappa^2$
c_3	0.3533	0.0010	$M(3).\kappa^2$
c_4	0.3546	0.00025	$M(4).\kappa^2$
c_5	0.3563	0.00009	$M(5).\kappa^2$
b_1	0.03611	0.0285	$M(1).t_{\text{steps}}$
b_2	0.3250	0.03956	$M(2).t_{\text{steps}}$
b_3	0.6039	0.0349	$M(3).t_{\text{steps}}$
b_4	1.048	0.0335	$M(4).t_{\text{steps}}$
b_5	1.805	0.0436	$M(5).t_{\text{steps}}$
a_1	0.2433	0.00036	$M(1).\kappa^2 t_{\text{steps}}$
a_2	0.2566	0.000075	$M(2).\kappa^2 t_{\text{steps}}$
a_3	0.2626	0.000015	$M(3).\kappa^2 t_{\text{steps}}$
a_4	0.2686	3.58e-06	$M(4).\kappa^2 t_{\text{steps}}$
a_5	0.2750	1.65e-06	$M(5).\kappa^2 t_{\text{steps}}$

Table 4.5: Least squares estimates of the constants and standard errors (s.e.) from the fit of the execution times (in clock cycles) for the explicit method. M is a factor with five levels defined according to equation (4.45).

meshes varying from 3×3 up to 240×240, and 16 values of t_{steps} from 2 to 384, giving 288 degrees of freedom in all, are shown in Table 4.4.

Figure 4.7(a) shows the fitted residuals $(T_{\text{exec}}(1) - \tau_{\text{exec}}(1))$ plotted against mesh size after 256 timesteps, with a clear effect visible at $(\kappa + 2)$ =16, 32, 64 and 128, as expected. The maximum fitted residual is about 20000 clock cycles (which is approximately a 2% error), and the model clearly does not give a good fit.

The least squares fit was calculated using the GLIM [1] statistical package, which also allows us to quantify the cost of calculating the memory addresses by declaring a factor M which partitions the data according to the size of the mesh in powers of two, so that

$$\begin{aligned} M(1) &= \left\{ T_{\text{exec}}(1) \text{ for } 3 \leq \kappa < 2^4 \right\} \\ M(i) &= \left\{ T_{\text{exec}}(1) \text{ for } 2^{i+2} \leq \kappa < 2^{i+3} \right\} \quad i \geq 2 \end{aligned} \tag{4.45}$$

Repeating the least squares fit on the partitioned data set gives the estimates for a_i, b_i, c_i, d, shown in Table 4.5. The fitted residuals after 256 timesteps shown in Figure 4.7(b) are now extremely small (the maximum difference between observed and fitted execution times is less than 0.01%).

The differences between successive estimates of a_i, b_i and c_i give a measure of the increase in the memory overheads with mesh size, with the differences between the a_i's being the main determinant since this term makes up virtually all of the

overall execution time. Therefore $(a_2 - a_1) \approx 0.013$ clock cycles $= 0.83\mu s$, gives a measure of the increase in cost per mesh point per timestep in going from on–chip to external memory, which is about twice that of the 0.006 clock cycles $= 0.38\mu s$ increase incurred each time the size of the array index increases by an extra bit. To give some idea of the global cost of this memory access problem, on our largest mesh after 256 timesteps the fitted execution times if we were to use estimates a_1, b_1 and c_1 would be 231 seconds, as compared to 261 seconds using a_5, b_5 and c_5.

Having analysed and quantified the cost of the increase in memory access times, we can make use of some of the assembler–like features of Occam to reduce this cost. Occam supports both *abbreviation* and the *block–copying* of vectors. These may be used to take advantage of the block address nature of the external memory to rename or to re–assign parts of the large global arrays into smaller local ones very quickly. We use the general term "abbreviation" here to cover both techniques. For example we can rewrite the nodal process for the explicit method as

```
PROC exp.node()
  [ynodes+2]REAL64 u1,u2,u3:
  SEQ
    nu1:=1-nu
    SEQ i=1 FOR xnodes
      SEQ
        u1:=u[i-1]
        u2:=u[i]
        u3:=u[i+1]
        SEQ j=1 FOR ynodes
          SEQ
            v[i][j]:=(nu1*u2[j])+
            (nu*((u3[j]+u1[j])+(u2[j+1]+u2[j-1])))
    u:=v
:
```

By assigning by block copy the relevant three rows of the global array to one–dimensional local arrays **u1, u2, u3**, and overwriting the old values **u** with the new values **v** using a 2D block copy (the block copy operates at about 12 Mbytes per second), we have not only removed most of our memory address calculation costs, but also greatly reduced the length of the data/address lines. This technique of code optimisation which makes use of the block storage of data will be very familiar to programmers of a previous generation or to those used to programming at a lower level. It results in greater programming complexity and codes in which the underlying mathematical algorithm is less transparent (particularly for already complex algorithms such as ADI). However, the improvement in performance can be quite substantial as shown in Table 4.6, which summarises this improvement for all three methods.

As can be seen, the explicit method is now 40% faster, with considerable savings in execution times for both of the other methods. However we no longer achieve superlinear speedup since the calculation to communication ratio has been reduced. Figure 4.6(b) shows the increase in speedup with mesh size for the faster

Method	$T_{\text{exec}}(1)$ (Δ) (s)	$T_{\text{exec}}(16)$ (Δ) (s)	$S(16)(\Delta)$	ϵ' (Δ)
Explicit	160.9 (-100.2)	10.40 (-5.6)	15.43 (-0.9)	0.96 (-0.6)
Hopscotch	194.1 (-8.3)	12.87 (-0.23)	15.07 (-0.33)	0.94 (-0.2)
ADI	684.9 (-70.6)	56.29 (-6.0)	12.16 (-0.04)	0.76 (0.0)

Table 4.6: Comparison of measured speedup of the three algorithms on a 240 $\times$ 240 mesh after 256 timesteps using abbreviation. Figures in brackets show the relevant change (Δ) as compared to Table 4.2.

implementations, which is much smoother now that we have greatly reduced the cost when either the global mesh or sub mesh size increases by a power of two.

In order to compare the parallel efficiency and effectiveness on the 16 processor network, we wish to consider mesh sizes up to 800 $\times$ 800, which will not fit into the memory of a single processor. We cannot therefore obtain the measured speedup and efficiency directly, but we can use a similar statistical model of the one–processor execution times to obtain good estimates of them. Since we are primarily interested in obtaining a fast solution, we will make use of abbreviation to optimise our code, but whilst this greatly reduces the address calculation overhead it does not totally remove it. We therefore model the execution times on one processor using equation (4.44) as our model, with the data partitioned according to equation (4.45), to obtain estimates for a_i, b_i, c_i, and d for $i = 1,\ldots,5$, for the three methods as shown in Table 4.7. As can be seen, the memory address calculation overhead has now been concentrated into the cost per iteration term (the b_i's) which roughly doubles for each partition, whilst for larger meshes, the cost per mesh point per timestep (a_3 to a_5) and overhead per mesh point (c_3 to c_5) remain approximately constant. We can therefore estimate, with some hope of reasonable accuracy, the coefficients at the successive, unmeasurable memory partitions i.e. for $M(6)$ ($256 \leq \kappa < 512$) and $M(7)$ ($512 \leq \kappa < 1024$), we use $a_7 = a_6 = a_5$, $c_7 = c_6 = c_5$, and $b_7 = 2b_6$, $b_6 = 2b_5$, to obtain the estimates of the execution times, measured speedup and efficiency shown in Tables 4.8 and 4.9. We will use these values in the next section in estimating the parallel effectiveness.[1]

The simpler explicit and hopscotch algorithms are of roughly equivalent speed, but the more complex nature of the ADI algorithm, together with the time spent by the processors idling whilst the calculation spreads across the processor network

[1]Since we have introduced assignment into one–dimensional vectors on each timestep to overcome the memory effect, we should also now include a term in κt_{steps} in the fit. However, whilst we found that this term gave a significant improvement in the fit for all three methods, it was strongly correlated with both the $\kappa^2 t_{\text{steps}}$ and t_{steps} terms, resulting in a loss of the systematic relationship between successive memory levels, and hence our predictive capability. This term was therefore omitted, with a resulting increase in the maximum error of the fitted residuals on larger meshes of $< 0.1\%$, together with poor standard errors for some of the less important terms in Table 4.7.

Method	Explicit		Hopscotch		ADI	
	estimate	s.e.	estimate	s.e.	estimate	s.e.
d	9.652	6.262	11.93	23.87	21.15	99.34
c_1	0.3055	0.075	0.3769	0.287	0.2689	1.19
c_2	0.3490	0.014	0.4282	0.0549	0.3547	0.229
c_3	0.3543	0.0035	0.4326	0.0132	0.3599	0.055
c_4	0.3547	0.00078	0.4319	0.0030	0.3581	0.012
c_5	0.3563	0.00031	0.4320	0.0012	0.3587	0.0049
b_1	0.7748	0.086	1.398	0.327	2.040	1.36
b_2	1.973	0.115	3.891	0.439	6.287	1.83
b_3	4.293	0.115	7.970	0.440	14.78	1.83
b_4	8.820	0.124	16.39	0.474	29.84	1.97
b_5	16.98	0.163	30.01	0.620	55.80	2.58
a_1	0.1739	0.00087	0.2157	0.0033	0.7315	0.0138
a_2	0.1706	0.00021	0.2073	0.00080	0.7269	0.0034
a_3	0.1686	0.000051	0.2042	0.00019	0.7212	0.00081
a_4	0.1681	0.000012	0.2030	0.000045	0.7212	0.00019
a_5	0.1685	5.9e–06	0.2034	0.000023	0.7235	0.000094

Table 4.7: Least squares fit of the execution times (in clock cycles) for the explicit, hopscotch and ADI methods where the Occam code makes use of abbreviation. Again, M is a factor with five levels defined by equations (4.45).

Mesh	Explicit		Hopscotch		ADI	
$\kappa \times \kappa$	$T_{exec}(1)$	$T_{exec}(16)$	$T_{exec}(1)$	$T_{exec}(16)$	$T_{exec}(1)$	$T_{exec}(16)$
200	111.6	7.26	134.9	9.08	476.0	39.5
400	447.2*	28.4	537.6*	34.8	1903.*	154.0
800	1788.*	112.6	2148.*	136.9	7610.*	608.7

Table 4.8: Measured execution times in seconds on 1 and 16 processors (those times denoted * have been estimated using equation (4.44) and the values of the coefficients given in Table 4.7).

Mesh	Explicit		Hopscotch		ADI	
$\kappa \times \kappa$	$S(16)$	ϵ'	$S(16)$	ϵ'	$S(16)$	ϵ'
200	15.30	0.956	14.86	0.923	12.05	0.753
400	15.73 *	0.983*	15.43*	0.964*	12.36*	0.772*
800	15.88*	0.992*	15.69*	0.981*	12.50*	0.781*

Table 4.9: Increase in speedup and efficiency with increasing mesh size for the three algorithms (* again represents estimates).

(which appears to be about 20%), results in the ADI algorithm running at about one fifth the speed on 16 processors.

4.7 Parallel Effectiveness

The central question that Leland attempts to answer in his thesis is

> " *How much would it cost to solve the same problem to the same accuracy on a given machine with the proposed algorithm,* "

and the *parallel effectiveness model* was developed in order to facilitate a quantitative answer to this question. Leland took the term "cost" to mean speed (or more exactly $(\text{speed})^{-1}$), and since we have moved to the multicomputer environment primarily to obtain a solution to the electrode problem, this will also be our main concern: which of our parallel algorithms will take the least time to solve our model problem to a specified accuracy? However, in other situations, we may wish to determine cost in financial terms or in terms of time for a completed project (from algorithm design through software development to completed solution), although since these may be programmer–dependent they would be more difficult to quantify.

4.7.1 The model

Leland defines his model of parallel effectiveness recursively to allow both qualitative and quantitative comparisons of methods using any choice of cost function. Since any chosen algorithm must be mapped onto our parallel architecture, he defines γ to be the *mapping* effectiveness. He also takes into consideration how fast a sequential solution the algorithm provides by defining an *algorithmic* effectiveness, ϕ, to obtain the overall *parallel* effectiveness,

$$\xi = \gamma\phi. \tag{4.46}$$

With speed as the cost function, the obvious measure of the mapping effectiveness γ is the measured parallel efficiency, ϵ', which we have already obtained for all three methods in Table 4.8. The algorithmic effectiveness must measure the cost of obtaining a solution to a specified accuracy. In solving by finite difference methods partial differential equations (p.d.e.'s) of any type, the spatial accuracy for a discretisation Ω_h is determined by the order of the finite difference approximation, the measure of the work involved for a particular algorithm being given by the sweep efficiency, $\Phi(1, \kappa^2)$. The most suitable measure of the overall algorithmic efficiency is then determined by the type of p.d.e. we are considering: for elliptic problems we use iterative methods and can measure the relative rates of convergence by comparing the spectral radius, ρ; for time–dependent hyperbolic and parabolic p.d.e.'s we require a measure of the maximum timestep that we can use whilst retaining stability and accuracy for the specified discretisation, determined by the CFL number (see, for example, Smith [91]) for hyperbolic p.d.e.'s, and the

mesh ratio ν for parabolic p.d.e.'s; for mixed hyperbolic–parabolic p.d.e.'s we use a combination of the two.

To compare our three methods for parabolic p.d.e.'s we therefore define $\kappa_{\min}$ and $\nu_{\max}$ to be the coarsest discretisation and maximum mesh ratio that we can use to obtain some specified accuracy δ. The algorithmic effectiveness, ϕ, is then given by

$$\phi = \Phi(1, \kappa_{\min}^2)\nu_{\max} \qquad \text{such that} \qquad |u_{i,j}^n - u(x,y,t)| < \delta, \tag{4.47}$$

where $u(x,y,t)$ is the exact solution (4.8) at a mesh point and $u_{i,j}^n$ the numerical approximation at that mesh point. Any two of the methods can then be compared to each other using

$$\frac{\xi_1}{\xi_2} = \frac{\gamma_1\phi_1}{\gamma_2\phi_2} = \frac{\epsilon_1'}{\epsilon_2'}\frac{\Phi(1,\kappa_{1_{\min}}^2)}{\Phi(1,\kappa_{2_{\min}}^2)}\frac{\nu_{1_{\max}}}{\nu_{2_{\max}}} \tag{4.48}$$

which gives our measure of the relative parallel effectiveness.

4.7.2 Comparison of the three methods

For the problem that is our main concern, the electrode problem, we would like to beat the 1% accuracy attainable by experiment. However, this accuracy refers to the measurement of the current, which we must approximate by numerically differentiating the partial pressure values. Since all three of our methods are second order accurate in space, by obtaining a numerical estimate of the spatial derivative we lose an order of accuracy i.e. whilst the partial pressure values are $O(h^2)$, the current values are only $O(h)$, and to obtain our 1% accuracy for the current, we must demand 0.01% accuracy in the partial pressure values. In using the model problem solution to compare our algorithms, we therefore choose $\delta = 0.01\%$ and require that at a given time $t = n\Delta t$

$$\max_{i,j}\left|\frac{u_{i,j}^n - u(x,y,t)}{u(x,y,t)}\right| < 10^{-4}. \tag{4.49}$$

The simple model problem, which we have chosen because we know its analytic solution, is much easier to solve numerically than the electrode problem, since it has no internal interfaces and no boundary singularity. If we fix the mesh ratio at 0.25 (the maximum possible for the explicit method) then we can exceed the required accuracy for both the explicit and ADI methods on a 20 $\times$20 mesh, and on an 80$\times$80 mesh for the hopscotch method, at all times. These are comparatively coarse meshes, and we already know that we need much finer meshes to solve the unshielded electrode problem to the same accuracy, and we expect the shielded case to be more difficult. We therefore assume that all three methods will achieve the required spatial accuracy, and look for the maximum possible timestep that we can use on meshes in the range of interest (200$\times$200 up to 800$\times$800).

Rewriting the values given in Table 4.8 as the relative speeds given in Table 4.10, we obtain a measure of the relative algorithmic complexity, relative efficiency

Mesh	$\frac{\Phi_E}{\Phi_H}$	$\frac{\epsilon'_E}{\epsilon'_H}$	$\frac{S_E(16)}{S_H(16)}$	$\frac{\Phi_E}{\Phi_A}$	$\frac{\epsilon'_E}{\epsilon'_A}$	$\frac{S_E(16)}{S_A(16)}$	$\frac{\Phi_H}{\Phi_A}$	$\frac{\epsilon'_H}{\epsilon'_A}$	$\frac{S_H(16)}{S_A(16)}$
200	1.21	1.04	1.24	4.27	1.27	5.42	3.53	1.23	4.35
400	1.20	1.02	1.23	4.26	1.27	5.42	3.54	1.25	4.42
800	1.20	1.01	1.22	4.26	1.27	5.41	3.54	1.26	4.45

Table 4.10: Relative algorithmic complexity, measured efficiency, and their product, the measured speedup, of the three methods on meshes of practical size, in going from one to sixteen processors.

and relative measured speedup of the three methods. If parallel efficiency were our main criterion in choosing our algorithm, then there would be little to choose between the explicit and hopscotch methods, and we would probably make our decision based on ease of programming. However, these measurements were all made at the fixed mesh ratio $\nu = 0.25$ to allow the explicit method to compete. Leland's parallel effectiveness model will now allow us to take account of the greater stability of the other two methods, before deciding which of the three is the most effective, with the ADI method needing to use a timestep at least 5 times that of the other methods to stay in the running.

We are interested in the time evolution of the current and we require that our chosen solution method should give the required accuracy at all times. Since both the model problem and electrode problem involve exponential decay, we choose to consider the solution of the model problem at six times at which the exponential has decayed by approximately 1%, 10%, 50%, 90% and 99% of its initial value, and at steady–state. The explicit method will achieve the required accuracy at all times with $\nu_{\max} = 0.25$, the number of timesteps required to reach each time being approximately 100, 1000, 4000, 20000, 40000, and 100000. Table 4.11 summarises the results on a 200×200 mesh and 800×800 mesh for the ADI and hopscotch methods. The ADI method is so stable that it requires very few timesteps (with a correspondingly large mesh ratio) to reach a given time, so that for both methods we give two values for each parameter, one giving slightly more than the required accuracy, and one giving slightly less. The ADI method remains both stable and sufficiently accurate using a timestep between 20 and 400 times the maximum possible with the hopscotch method on a 200×200 mesh, and between 80 and 2000 times on an 800×800 mesh. We can calculate the relative measured parallel effectiveness of the three algorithms using equation (4.48) and these are given in Table 4.12.

Over most of the range of interest, the ADI method is at least two orders of magnitude faster, and therefore more effective, than its nearest rival the hopscotch method. In the next chapter where we will describe our numerical solution to the shielded electrode problem, we will therefore use the ADI method. It is worth

Decay	ADI			Hopscotch		
	$\nu_{\max}$	t_{steps}	% error	$\nu_{\max}$	t_{steps}	% error
			200×200	Mesh		
~1%	25.0	1	.006	1.25	20	.006
				2.5	10	.011
~ 10%	50.0	5	.002	1.25	200	.004
	125.0	2	.011	2.0	125	.011
~50%	250.0	5	.0098	1.25	1000	.005
	312.5	4	.012	2.0	625	.011
~90%	500.	10	.003	2.0	2500	.008
	1000.	5	.014	2.5	2000	.014
~99%	2000.	5	.008	5.0	2000	.0084
	2500.	4	.012	10.0	1000	.32
~St.St.	5000.	5	.006	32.0	1000	.0097
	6250	4	.012	40.0	625	0.23
			800×800	Mesh		
~1%	400.0	1	.006	5.0	80	.0041
				10.0	40	.020
~ 10%	1000.	4	.00032	5.0	800	.0042
	2000.	2	.011	10.0	400	.017
~50%	4000.	5	.0085	5.0	4000	.0048
	5000.	4	.012	10.0	2000	.019
~90%	10000.	8	.0048	5.0	16000	.0053
	20000.	4	.018	10.0	8000	.021
~99%	20000.	8	.0029	20.0	8000	.0082
	40000.	4	.011	25.0	6400	.13
~St.St.	80000.	5	.0064	125.0	3200	.0072
	100000.	4	.012	200.0	2000	0.78

Table 4.11: Values of the mesh ratio and associated number of timesteps to obtain the specified error (as given by equation (4.49)) for the hopscotch and ADI methods at increasing times. Two values of each parameter are given, one giving an error slightly lower than our specified accuracy, the other slightly higher.

Mesh	$\frac{\xi_A}{\xi_E}$	$\frac{\xi_A}{\xi_H}$	$\frac{\xi_H}{\xi_E}$
200	$18.5 - 1.85 \times 10^4$	4.5 – 92.0	4.0 – 80.6
800	$296. - 2.96 \times 10^5$	18.0 – 449.	16.4 – 410.

Table 4.12: The range of the relative parallel effectiveness of the three methods calculated using equation (4.48).

Operation	Fortran (μs)	Occam (μs)	Ratio
Addition	3.35	1.35	2.48
Multiplication	4.14	2.10	1.97
Division	4.88	2.70	1.81
Assignment	3.16	0.80	3.95
Flop	4.61	2.48	1.86
Communication			
Initialise	8.19	6.30	1.3
Transmit	4.50	4.50	1.0

Table 4.13: Comparison of times for arithmetic operations and communication on double length reals. Flop = add + mult.

noting, however, that had we chosen a different cost function such as software development time, then both the hopscotch and explicit methods would have beaten the ADI method by similar orders of magnitude.

4.7.3 Language and software considerations

Since the transputer and Occam were designed for each other, it is not surprising that the floating point code generated by a Fortran compiler is not as efficient as the equivalent Occam code. Table 4.13[2] compares the times taken for various arithmetic operations and communications in both Fortran and Occam, but perhaps a more relevant comparison can be obtained from the execution time for the simple implementation of the explicit method (for a 240 $\times$ 240 mesh after 256 timesteps) on a single processor in each of the languages: the Occam code without abbreviation takes (from Table 4.2) 260 seconds; that with abbreviation takes 160 seconds; whilst a simple Fortran code compiled using the 3L Fortran compiler [65] takes 386 seconds. The Fortran code, which implements exactly the same algorithm as the original Occam code, takes 1.5 times as long, but 2.4 times as long as the optimised Occam code which uses abbreviation.

The cost of choosing a familiar high–level language such as Fortran has therefore been much reduced since the first such compilers became available (these generated floating point code that was about 6 times slower than its Occam equivalent). Since Occam is available only for the transputer, it seems likely that the advantages of portability between different hardware enjoyed by the traditional high–level languages will outweigh the by–now marginal gain in speed achievable using Occam, with Occam therefore reduced to a subsidiary role.

However, Occam's greater speed is not its only advantage. We have skipped over, to some extent, the mathematical basis of the Occam language and model, mentioning only briefly that it enabled Leland to prove certain properties about his ONDE system using formal methods. We can use Leland's system to give a

[2]I would like to thank Paul Wesson for providing the Fortran timings.

guide to the cost of remaining faithful to the Occam model when designing code, by comparing the performance attainable using ONDE with that achieved using a simple spatial decomposition and the halo communication process for the same problem. Leland's model problem was the solution of Laplace's equation on the unit square using various iterative algorithms of which he found Red–Black Successive Over Relaxation (RBSOR) to be the most effective. Comparison of the speed of the ONDE implementation of RBSOR with a simple Occam implementation using the halo process shows that such faithfulness costs a factor of about 6. Therefore, programming in Occam in the manner in which its designers intended, results in code which would run at about a quarter the speed of the equivalent Fortran code. The equivalent Fortran code, however, would only be *equivalent* in the sense that it would be a Fortran implementation of the same mathematical algorithm. We would not, for instance be able to give a formal proof that it would do the job intended for it. If, rather than solving Laplace's equation, we were attempting to write a fail–safe code for, for example, the control of an aircraft, then the formal mathematical basis of Occam would become invaluable in proving program–correctness. It seems likely, however, that for scientific applications such as the numerical solution of p.d.e.'s, scientists will continue to opt for the more familiar languages.

4.8 Summary

We have discussed in this chapter the parallel architecture and software available to us, and further developed the parallel efficiency and effectiveness models of Leland to incorporate a wider class of problems. This allowed us to analyse, optimise and compare the parallel implementations of three methods of solution for the parabolic p.d.e.'s in which we are interested. The use of statistical techniques to investigate run times and memory efficiency allowed us to gain substantial improvements in performance. In the next chapter we will use the knowledge gained in solving the simple model problem when finding an efficient and effective parallel solution to the electrode problem.

Chapter 5

Solution for the Clark Electrode

In Chapter 2 when we first described the modelling of the Clark electrode, we pointed out the difficulties caused by the presence of the singularity at the electrode edge for which an effective treatment in the case of the unshielded electrode was derived in Chapter 3. A few test runs of a sequential code to solve for the shielded electrode making use of the singularity correction led us to realise that we would be unable to demonstrate the required accuracy in a reasonable time on the conventional sequential computers available to us [27]. In this chapter we will therefore combine the work of the preceding chapter on parallel algorithms for problems of this type with the work on the singularity correction, and obtain an efficient parallel numerical solution to the equations governing the operation of the Clark electrode. We will investigate the manner in which the errors of the two approaches combine together, allowing us to demonstrate that we can achieve the required accuracy using this method.

5.1 Numerical Methods

The transport of oxygen from the sample, through the membrane and electrolyte, to the cathode can be modelled by the cylindrical diffusion equation, as described in Section 2.2. A very similar numerical technique to that described for the unshielded electrode of Chapter 3 can be used, although the inclusion of the different layers and resulting interface conditions adds an extra degree of complexity to the problem. The finite region in which we wish to obtain a solution was illustrated in Figure 2.3, and to obtain a numerical solution we must again discretise the region by superimposing on the domain the rectangular mesh shown in Figure 5.1. In the r–direction we again choose the mesh lines

$$r_i = (i + 1/2)\,\Delta r \quad (i = 0, 1, \ldots, N) \tag{5.1}$$

with uniform spacing

$$\Delta r \;=\; \frac{r_c}{n_r}. \tag{5.2}$$

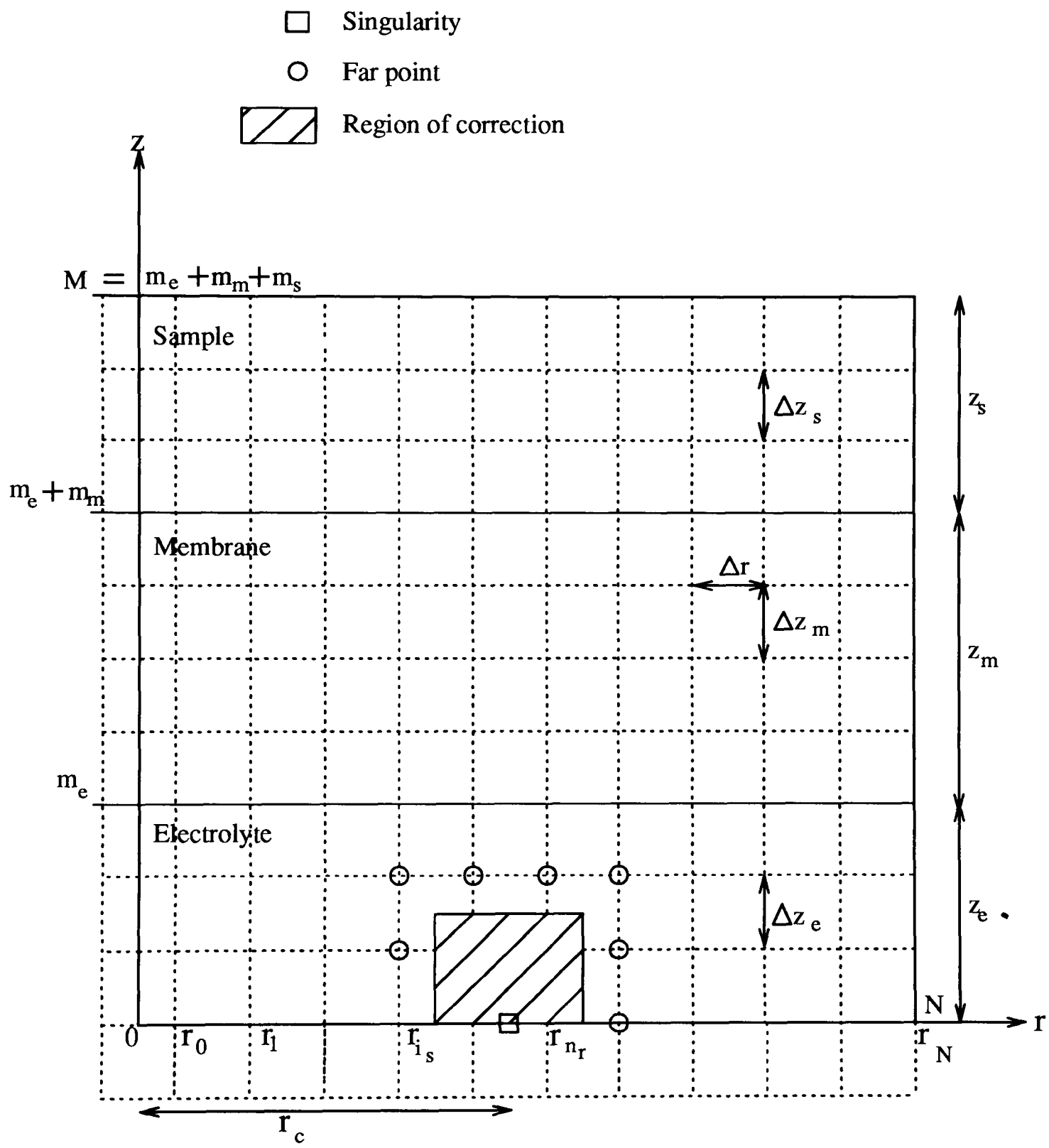


Fig. 5.1: The rectangular mesh which is superimposed over the domain to obtain the numerical solution.

In the z–direction we may wish to allow the the mesh spacing Δz_l to differ in the three layers $l = e, m, s$, and therefore choose mesh lines at

$$\begin{array}{lll} \text{electrolyte} & z = j\Delta z_e & (0 \le j \le m_e) \\ \text{membrane} & z = z_e + j\Delta z_m & (0 \le j \le m_m) \\ \text{sample} & z = z_e + z_m + j\Delta z_s & (0 \le j \le m_s), \end{array} \tag{5.3}$$

where $m_l = z_l/\Delta z_l \quad l = e, m, s$, which also ensures that each of the material interfaces lies on a meshline. We therefore wish to obtain a numerical solution at these mesh points to equation (2.2) subject to the initial, boundary and interface conditions given by equations (2.4) to (2.8).

5.1.1 The ADI method

In the previous chapter we determined that the most *effective* method of parallel solution for the simple diffusion equation was the pipelined implementation of the ADI algorithm. We therefore adapt this algorithm in finding a parallel solution for the Clark electrode problem. From equation (3.9), the method is written in cylindrical coordinates as

$$\begin{aligned} \left[1 - \frac{\nu_r^{(l)}}{2}\left(\delta_r^2 + \frac{1}{i+\frac{1}{2}}\Delta_{0r}\right)\right] u_{i,j}^{k+\frac{1}{2}} &= \left[1 + \frac{\nu_z^{(l)}}{2}\delta_z^2\right] u_{i,j}^{k} \\ \left[1 - \frac{\nu_z^{(l)}}{2}\delta_z^2\right] u_{i,j}^{k+1} &= \left[1 + \frac{\nu_r^{(l)}}{2}\left(\delta_r^2 + \frac{1}{i+\frac{1}{2}}\Delta_{0r}\right)\right] u_{i,j}^{k+\frac{1}{2}}, \end{aligned} \tag{5.4}$$

where the mesh ratios, $\nu_r^{(l)} = D_l\frac{\Delta t}{\Delta r^2}$, $\nu_z^{(l)} = D_l\frac{\Delta t}{\Delta z_l^2}$, can now differ in the three layers. u_{ij}^k is our approximate solution at the mesh points to the true partial pressure $p'((i+\frac{1}{2})\Delta r, j'\Delta z', k\Delta t)$ where $i = 0, \ldots, N$ and $j = 0, \ldots, M$, and $j'\Delta z'$ is a convenient notation for the non–uniform z–mesh.

By incorporating the interface conditions in the correct manner (see below), we can maintain the simple tridiagonal form of these equations so that we need solve only tridiagonal systems in the r–direction at $(k+1/2)\Delta t$, and in the z–direction at $(k+1)\Delta t$, whilst maintaining the stability and second order accuracy described in Chapter 4.

5.1.2 Discretisation of boundary conditions

The Neumann boundary conditions along $r = 0$ and $z = 0$ for $r > r_c$ are dealt with by introducing fictitious mesh lines at $z = -\Delta z_e$, $r = -\frac{1}{2}\Delta r$ respectively. The boundary conditions (2.7) are then imposed by

$$
\begin{array}{lllll}
\text{On } z = 0 & 0 \le r \le r_c & u_{i,0} = 0 & (i = 0, \ldots, n_r) & \\
\text{On } z = 0 & r_c \le r \le r_{\max} & u_{i,1} = u_{i,-1} & (i = n_{r+1}, \ldots, N) & \\
\text{On } r = 0 & z \ge 0 & u_{-1,j} = u_{0,j} & (j = 0, \ldots, M) & (5.5) \\
\text{On } r = r_{\max} & & u_{N,j} = 1 & (j = 0, \ldots, M) & \\
\text{On } z = z_{\max} & & u_{i,M} = 1 & (i = 0, \ldots, N), &
\end{array}
$$

when a constant polarising potential sufficient to reduce oxygen at the cathode is being applied between the electrodes. If the sensor is being used in the pulsed mode, then we use the first of equations (5.5) whilst the potential is applied, but use $u_{i,1} = u_{i,-1}$ for $0 \le r \le r_{\max}$ on $z = 0$ when it is removed.

The Neumann conditions described by the second and third of equations (5.5) can be incorporated directly into the tridiagonal systems, with the values at the fictitious mesh lines corresponding to $i = -1$ and $j = -1$ being overwritten at the end of each timestep. The Dirichlet conditions are imposed using the first of equations (5.5) at the cathode, and the fourth and fifth "at infinity".

5.1.3 Discretisation of interface conditions

The internal conditions at the material interfaces ensure that oxygen partial pressure and flux are equal. They are given (from equation (2.8)) by

$$
\begin{array}{lll}
\text{on } z = z_e & p'_{z=z_{e-}} = p'_{z=z_{e+}}, & \\
& P_e \left(\frac{\partial p'}{\partial z}\right)_{z=z_{e-}} = P_m \left(\frac{\partial p'}{\partial z}\right)_{z=z_{e+}}, & (5.6) \\
\text{and on } z = z_e + z_m & p'_{z=z_e+z_{m-}} = p'_{z=z_e+z_{m+}}, & \\
& P_m \left(\frac{\partial p'}{\partial z}\right)_{z=z_e+z_{m-}} = P_s \left(\frac{\partial p'}{\partial z}\right)_{z=z_e+z_{m+}}. &
\end{array}
$$

We can approximate each of the $\partial p'/\partial z$ terms using simple first order backward and forward difference approximations on either side of the membrane to to get

$$
\begin{aligned}
P_e \left(\frac{u_{i,m_e} - u_{i,m_e-1}}{\Delta z_e}\right) &= P_m \left(\frac{u_{i,m_e+1} - u_{i,m_e}}{\Delta z_m}\right), \\
P_m \left(\frac{u_{i,m_e+m_m} - u_{i,m_e+m_m-1}}{\Delta z_m}\right) &= P_s \left(\frac{u_{i,m_e+m_m+1} - u_{i,m_e+m_m}}{\Delta z_s}\right),
\end{aligned} \tag{5.7}
$$

for $i = 0, \ldots, N$ on $z = z_e$, $z_e + z_m$ respectively. In the r–direction, we first solve the tridiagonal systems for all mesh lines other than those along the interfaces,

and then use equation (5.7) in the form

$$\begin{aligned} u_{i,m_e} &= \left(\frac{(P_e/\Delta z_e)u_{i,m_e-1} + (P_m/\Delta z_m)u_{i,m_e+1}}{P_e/\Delta z_e + P_m/\Delta z_m} \right), \\ u_{i,m_e+m_m} &= \left(\frac{(P_m/\Delta z_m)u_{i,m_e+m_m-1} + (P_s/\Delta z_s)u_{i,m_e+m_m+1}}{P_m/\Delta z_m + P_s/\Delta z_s} \right), \end{aligned} \tag{5.8}$$

on $z = z_e$ and $z_e + z_m$ respectively. When solving for lines in the z–direction, we rewrite (5.7) in the form

$$\begin{aligned} \frac{P_e}{\Delta z_e}u_{i,m_e-1} - \left(\frac{P_e}{\Delta z_e} + \frac{P_m}{\Delta z_m}\right)u_{i,m_e} + \frac{P_m}{\Delta z_m}u_{i,m_e+1} &= 0, \\ \frac{P_m}{\Delta z_m}u_{i,m_e+m_m-1} - \left(\frac{P_m}{\Delta z_m} + \frac{P_s}{\Delta z_s}\right)u_{i,m_e+m_m} + \frac{P_s}{\Delta z_s}u_{i,m_e+m_m+1} &= 0, \end{aligned} \tag{5.9}$$

on $z = z_e, z_e + z_m$ respectively, which can then be incorporated directly into the tridiagonal systems.

5.1.4 *LU* Decomposition of tridiagonal matrices

Whilst handling the boundary and interface conditions in this manner allows us to maintain the tridiagonal form of the equations, we no longer obtain the same tridiagonal matrix for all mesh lines in a particular direction. Figure 5.2 summarises the different types of mesh lines. We must decompose two tridiagonal matrices to solve for mesh lines parallel to the z–direction, of length $(M-1)$ (above the cathode), and of length M (above the insulation boundary). In the r–direction we have four types of unknown vector, one of length $N - n_r$ along $z = 0$ away from the cathode (if a potential difference is being applied), together with one in each layer of length N. Each of these six matrices can be decomposed into its LU factors and stored in three vectors in exactly the same manner as in Chapter 4, with the forward and backward sweeps for any particular mesh line performed using the appropriate set of vectors. This will, however, greatly complicate the parallel implementation of the ADI method.

5.2 Correction of Boundary Singularity

The derivation and implementation of the truncated series solution of equation (2.2) about the singularity at the cathode edge was described in Chapter 3. It can be implemented in the same manner for the shielded case, i.e. we use the solution calculated by the ADI method at points sufficiently distant from the singularity as to be unaffected by it (the "far points" in Figure 5.1), to obtain the coefficients of the locally valid series expansion. This is then used to overwrite values near the singularity. The added proviso in the shielded case is that the neighbourhood of correction is wholly contained within the electrolyte layer.

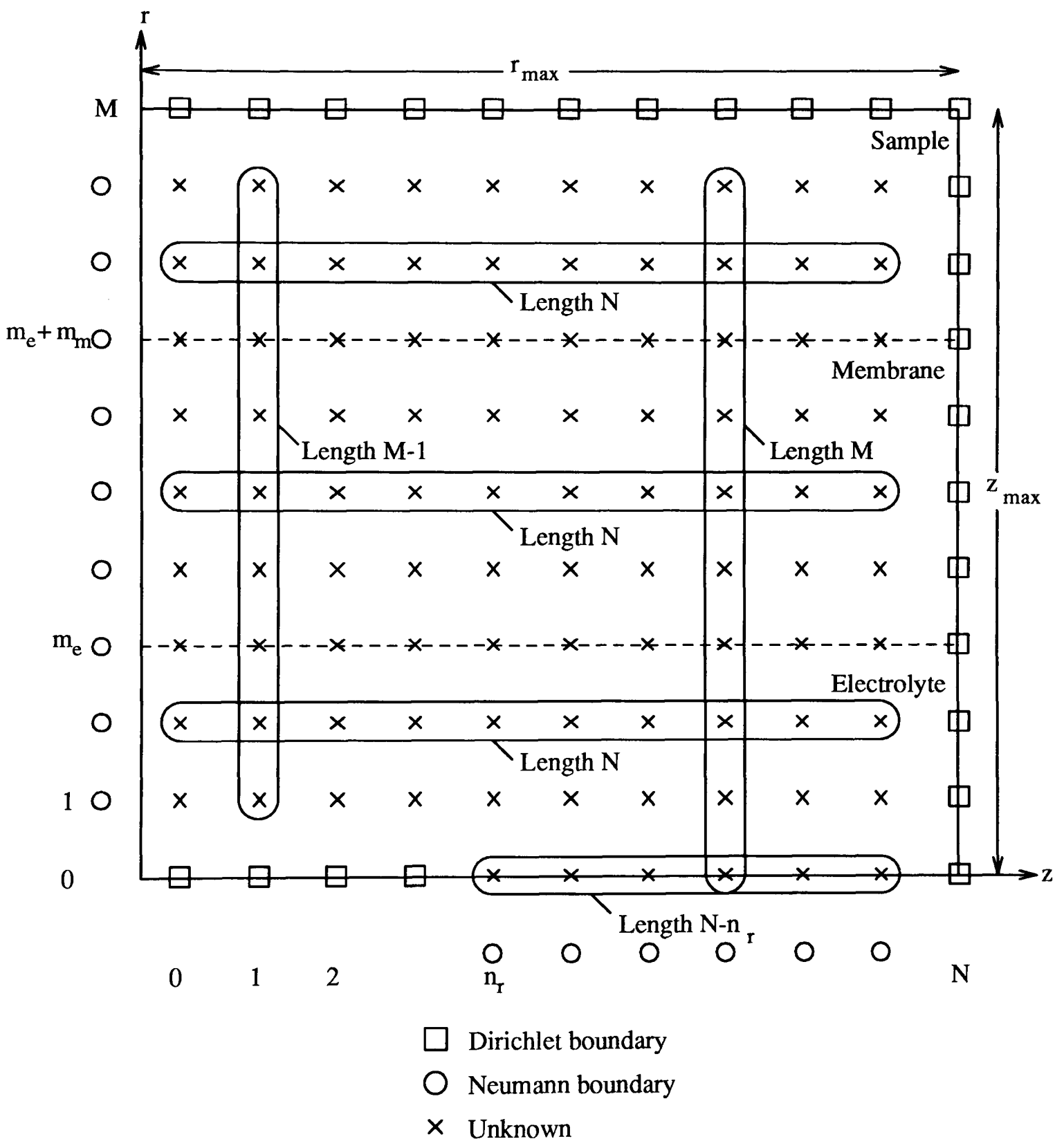


Fig. 5.2: Discretisation used to solve the shielded electrode problem. Using the ADI method we solve for complete mesh lines of unknowns in the r–direction of length N or $N - n_r$ at step $k + \frac{1}{2}$, then for complete mesh lines in the z–direction of length M or $M - 1$ at step $k + 1$.

5.3 Calculation of the Current to the Electrode

The current to the cathode is given by equation (2.3). For ease of notation we define a flux function

$$Q(t) = \frac{i(t)}{2\pi nFP_e} = \int_0^{r_c} \left(\frac{\partial p}{\partial z}\right)_{z=0} r dr.$$

which we will use in describing our numerical method of approximating the current. In calculating the flux we must again take into account the nature of the singularity at the cathode edge. We therefore consider the region $r \leq r_{i_s}$ which is sufficiently distant from the singularity as to be unaffected by it, and $r \geq r_{i_s}$ separately, where $r_{i_s} = (i_s + 1/2)\Delta r$ (see Figure 5.1).

5.3.1 $r \leq r_{i_s}$

In Chapter 3, for $r \leq r_{i_s}$ we followed Evans and Gourlay [18] in using a bilinear interpolant over mesh squares adjacent to the cathode. The introduction of the membrane can introduce greater curvature into the solution in the z–direction (see Chapter 6), with the result that the first order bilinear form is insufficiently accurate, and we must move to a second order method.

This can be done by using the Taylor series expansion of $p(r, z, t)$ about $z = 0$ to give a second order approximation to $\partial p/\partial z$ on $z = 0$. In terms of the calculated approximate values of p, $u_{i,1}, u_{i,2}$ at $z = \Delta z_e, 2\Delta z_e$, this gives

$$\left(\frac{\partial p}{\partial z}\right)_{z=0} \approx u_{z_i} = \frac{4u_{i,1} - u_{i,2}}{2\Delta z_e} \quad i = 0, \ldots, i_s. \tag{5.10}$$

With r_i defined as in equation (5.1), we subdivide the region $[0, r_{i_s}]$ into intervals $[r_{2j}, r_{2j+2}]$, $j = 0, (i_s/2-1)$, of length $2\Delta r$. We then use the usual divided difference formula (see, for example, Chapter 2 of [9]) to obtain the constants $\alpha_j, \beta_j, \gamma_j$ in each interval which give the quadratic approximation to $\partial p/\partial z$ along $z = 0$ in the form

$$\frac{\partial p\,(r, 0, t)}{\partial z} \approx g_j\,(r) = \alpha_j r^2 + \beta_j r + \gamma_j, \quad r \epsilon [r_{2j}, r_{2j+2}] \tag{5.11}$$

such that

$$g_j\,(r_{2j+k}) = u_{z_{2j+k}} \quad (k = 0, 1, 2), \quad (j = 0, i_s/2 - 1). \tag{5.12}$$

Defining

$$Q_j = \int_{r_{2j}}^{r_{2j+2}} \left(\frac{\partial p}{\partial z}\right)_{z=0} r dr, \tag{5.13}$$

we may substitute from equation (5.11) for $\partial p/\partial z$ and integrate to obtain

$$Q_j \approx \left[\frac{\alpha_j r^4}{4} + \frac{\beta_j r^3}{3} + \frac{\gamma_j r^2}{2}\right]_{r_{2j}}^{r_{2j+2}} \tag{5.14}$$

as the approximate flux into each interval. In the first interval, $r\epsilon[0, \Delta r/2]$, we assume that $\partial p/\partial z$ is constant and use

$$Q_0 \approx \frac{\Delta r^2}{8} u_{z_0}. \tag{5.15}$$

5.3.2 $r \geq r_{i_s}$

For $r \geq r_{i_s}$, we again make use of the locally valid series expansion about the singularity to give the oxygen flux to the cathode. Defining ρ to be the distance from the singularity and Q_{sing} to be the flux into the region $[r_{i_s}, r_c]$, we may use the expression derived in Chapter 3 (equation (3.70)) in the form

$$\begin{aligned} Q_{sing} \approx\ & r_c^{\frac{1}{2}} \left[c_1 \rho^{\frac{1}{2}} \left(1 - \frac{\rho}{6r_c} - \frac{\rho^2}{30r_c^2} \right) + \right. \\ & \left. c_2 \rho^{\frac{3}{2}} \left(1 - \frac{3\rho}{10r_c} \right) + \left(c_3 + \frac{c_4}{5} \right) \rho^{\frac{5}{2}} \right]_{\rho_{i_s}}^{0}. \end{aligned} \tag{5.16}$$

The constants c_i, $i = 1, \ldots, 4$, are the time–dependent coefficients of the locally valid series expansion, and ρ is the distance from the singularity so that $\rho_{i_s} = r_c - r_{i_s}$.

Our total approximation of the current at the cathode is then given by the sum of these contributions

$$i(t) = 2\pi n F P_e Q(t) \approx 2\pi n F P_e \left\{ Q_0 + \sum_{j=1}^{\frac{i_s}{2}-1} Q_j + Q_{sing} \right\}. \tag{5.17}$$

5.4 Parallel Implementation

The Occam code that implements a parallel version of the mathematical algorithm defined by equations (5.4)–(5.17) is given in the appendices, together with a description of the structure of the code. Here we will describe only the major features of the parallel implementation, and in particular those areas where it differs from the simpler algorithm described in Chapter 4. There we showed that the parallel implementation of the the ADI method to solve the model problem is 70–80% efficient on meshes of practical size. For the shielded electrode problem we will use a similar geometric decomposition of the rectangular $N \times M$ mesh of Figure 5.1, onto our **xprocs**$\times$ **yprocs** network of processors as $\frac{N}{\textbf{xprocs}} \times \frac{M}{\textbf{yprocs}}$ rectangular submeshes. The simple model problem used only Dirichlet boundary conditions so that we could use the **(xid,yid)** coordinate of a processor in the network to determine and initialise its submesh. At each timestep we then initiated the forward sweep on the first processor in each row (or column if solving in the opposite direction), the backward sweep on the last, with all intermediate processors running the same piece of code. The initiation of any sweep could be handled using conditionals on the processor coordinates so that each processor ran a single, identical Occam code, which is highly desirable, since it ensures that if more processors

become available we can make immediate use of them just by changing the values of **xprocs** and **yprocs** in the configuration statement described in Section 4.4.4.

For the shielded electrode problem we have a much more complicated mesh resulting in a great variety of possible submeshes, onto which we wish to map our parallel ADI algorithm in such a way that we again obtain a single identical Occam code which runs on every processor. This can be done using a series of conditionals, of which as many as possible are kept outside the timestepping loop (and are therefore done only once) to maintain computational efficiency. The resulting decision structure is illustrated in Figure 5.3 together with four of the 162[1] possible submeshes.

5.4.1 *LU* Decompositions

Once each processor has determined and initialised its submesh, it calculates only those portions of the *LU* decompositions that it will need when updating these unknowns at each half–step. The possibilities are best illustrated by considering a few examples such as those given in Figure 5.3. The submesh (a) requires two decompositions in the r–direction (one along $z = 0$ which it initiates and one away from $z = 0$ in the electrolyte layer) and two in the z–direction (above the cathode and above the insulation boundary), (b) requires two in the r–direction (one in each layer[2]) and one or two in the z–direction (dependent on whether the submesh is above the singularity or not), (c) requires just one in the r–direction and one or two in the z–direction, whilst (d) requires four in the r–direction and two in the z–direction.

5.4.2 Forward and backward sweeps

Once the *LU* decompositions are complete each processor in the first column can begin the timestepping with the forward sweep along its first r–meshline as described in Section 4.5.3, except for processors in the first processor row which contain the boundary along $z = 0$. The forward sweep for these processors is initiated by whichever of them contains the singularity, with all subsequent r–meshlines being started on the first processor in each row as before. The last processor in each row again sweeps forwards and backwards, buffering information until the penultimate processors have completed their forward sweeps. The backward sweep across each meshline then proceeds as before, except on $z = 0$ where it terminates once it has reached the singularity. The solution in the opposite direction for z–meshlines is identical to that of section 4.5.3 except that different

[1]Referring to Figure 5.3, this figure is obtained from: (left branch (18)) plus (right branch (9)) of the decision tree times the six possibilities (first, middle, or last processor in a row or column) from the parallel implementation of the ADI method. In practice this figure can be slightly reduced since some of the information derived from the decision tree can be used in the ADI algorithm (see Appendix A).

[2]These are, of course, simply scalar multiples of each other, but are calculated separately for simplicity.

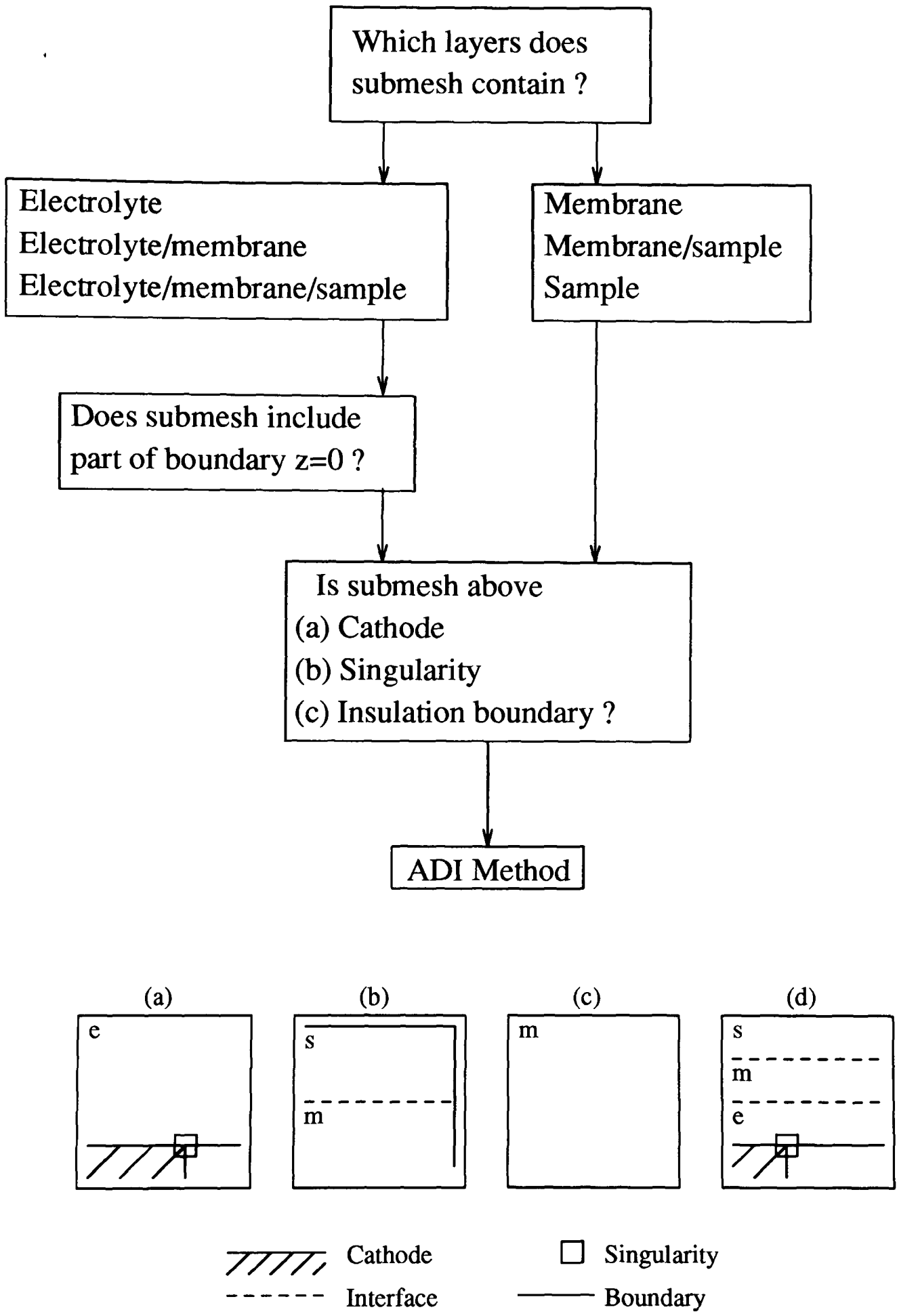


Fig. 5.3: Decision structure to determine the nature of the submesh of any particular processor in the network, together with just four of the possible submeshes.

Mesh	Efficiency				
	No. Procs.	2	4	8	16
50 × 50		71%	63%	55%	46%
100 × 100		74%	66%	61%	55%
200 × 200		75%	68%	66%	62%
400 × 400		–	69%	68%	66%
800 × 800		–	–	–	68%

Table 5.1: Variation of the efficiency of the parallel code with number of processors and mesh size.

decomposed vectors must be used when solving for meshlines above the cathode and above the insulation boundary.

5.4.3 Singularity correction

Once a complete timestep has been completed, the processor whose submesh contains the singularity carries out the singularity correction in the manner described in Section 3.5. This part of the code is completely sequential and it is left to the user to ensure that the region of correction and the associated far points are entirely contained both within one submesh and within the electrolyte layer. The calculation of the pseudo–inverse $\mathbf{F}^*$ and the vector μ (defined by equations (3.49) and (3.56)) are carried out once just after the decompositions, and thereafter the values at the near points are overwritten by the values calculated from the series expansion, a process which requires just m (the number of far points) multiplications per near point. This process should not therefore substantially alter the efficiency of the parallel ADI algorithm.

5.5 Efficiency

In Chapter 4, we showed that our parallel implementation of the ADI method was about 70–80% efficient on problems of practical size. Table 5.1 shows the efficiency figures for various mesh sizes and numbers of processors when solving the electrode problem. Since we use the same algorithm as the sequential case, these figures are a true indication of the speedup we can obtain by moving to a parallel machine. On one processor the calculation takes approximately $5 \times 10^{-5} s$ per mesh point per complete timestep, which is about 0.5 Mflops, with about 5 Mflops on 16 processors. On the largest mesh the efficiency is now around 68%, the reduction being due to the more complicated decision structure and the need to do the singularity correction on just one of the processors at each timestep. These figures also suggest that provided the problem size increases with the number of processors, the efficiency will remain acceptable.

Layer	Diffusion Coefft m^2s^{-1}	Solubility $molm^{-3}atm^{-1}$	Layer thickness (μm)	Radius (μm)
Electrolyte	$\approx 2 \times 10^{-9}$	≈ 1.0	2.0–10.0	
Membrane	10^{-10} to 10^{-12}	1.0 to 20.0	5.0 to 20.0	5.0 to 100.0
Sample	1.0 to 5.0 $\times 10^{-9}$	≈ 1.0	0.0 to 100.0	

Table 5.2: Parameter values.

5.6 Error Analysis and Convergence

The parallel implementation gives a sufficiently fast solution to allow us to examine the numerical properties of the overall solution technique as the mesh is refined. In obtaining the solution method for the partial pressure, we have coupled a locally valid analytic treatment of the singularity via a least squares approximation to the implicit finite difference method in the rest of the region, whilst also incorporating the various boundary and interface conditions. The approximation to the quantity of interest, the current, is then obtained from these values via a further combination of analytical and numerical techniques. Whilst it is possible to analyse the error introduced by these successive approximations individually, their combined effect can only be dealt with empirically and is, not surprisingly, quite complex. We stated at the outset that our aim was to achieve 1% accuracy. We here define this to mean that as we refine our spatial mesh, we will say that we have achieved 1% numerical accuracy if on consideration of a sequence of successively finer meshes, it *appears* that further refinement will not alter the calculated value of the current by more than 1%. We emphasise *appears* since on a computer with finite memory, we will not be able to prove that this is the case.

We have already seen in Chapter 3 that the errors introduced in truncating the series expansion about the singularity were $O(\rho^{\frac{7}{2}})$, where ρ is the distance from the singularity. However this is in turn dependent on the errors of the finite difference approximation from which it is calculated. We therefore consider first the local truncation error and stability of the ADI method. We will make use of this, together with the effects of the singularity correction, to explain the pattern of convergence of the current along various refinement paths for several sets of parameter values. Based on this practical experience of running the code, we are able to give a set of rules under which best results can be obtained.

5.6.1 Parameter values

So that we might use realistic test cases in considering the convergence we examined the electrochemical literature to obtain appropriate values for the various parameters of interest. Table 5.2 was compiled using values from [8, 22, 39, 48, 73, 84, 94] and shows the range of values for each parameter.

For any particular membrane material there is much disparity between the

Layer	Diffusion Coefft $m^2 s^{-1}$	Solubility $mol\ m^{-3}\ atm^{-1}$
Electrolyte	2.1×10^{-9}	1.28
Membrane	1.14×10^{-11}	9.6
Sample	$1.0\text{–}5.0 \times 10^{-9}$	2.0–5.0

Table 5.3: Parameter values that we will assume for the electrolyte, PTFE membrane and sample, taken from Myland and Oldham [73].

values of the diffusion (D_m), permeability (P_m) and solubility (α_m) coefficients quoted by different authors. For example, for the most commonly used membrane material, polytetrafluoroethylene (PTFE), we obtained the range (1.1 to 5.6) $\times 10^{-11} m^2 s^{-1}$ for the diffusion coefficient of oxygen at 27° C.

Myland and Oldham [73] give a similar wide range of values for the permeability of PTFE. However quoting Hitchman [42], they do give a complete set of values in SI units for both the electrolyte layer and the membrane when giving their solution to the one–dimensional, two–layer linear diffusion problem. We therefore adopt their values here when considering the numerical convergence of our method. These values are given in Table 5.3.

5.6.2 Stability

In Cartesian coordinates on a rectangular region as used in Chapter 4, we can analyse the stability of the ADI method very straightforwardly by Fourier analysis (see, for example, [71]) and show that it is unconditionally stable, and since it gives a consistent approximation we are ensured of convergence to the true solution by the Lax equivalence theorem [80]. In cylindrical coordinates, however, we have a non–constant coefficient (there is a term in $1/r$) so that Fourier analysis is not possible. In both cases a maximum principle clearly holds for $\max\{\nu_r, \nu_z\} < 1$, which is a sufficient condition for stability (but not necessary), and the consistency of the cylindrical ADI approximation again guarantees convergence along any refinement path obeying this restrictive condition on the mesh ratios. In practice we find, as expected, that the perturbation to the p.d.e. introduced by the $1/r$ term does not affect the stability of the ADI method within the limit s of the mesh ratio that we will use. However, as we will show, in the interest of accuracy we will be forced to use a much smaller timestep, particularly on coarse meshes, than for the simple model problem of Chapter 4.

5.6.3 Local truncation error

We can analyse the local truncation error (LTE) of the ADI method in the interior of a particular layer in the usual way by substituting the exact solution of the problem, $p'(r, z, t)$ into the finite difference equations (5.4). At any point $(r, z) = ((i + \frac{1}{2})\Delta r, j'\Delta z)$ in the mesh, the LTE, T_{ij}^k, in going from timestep k to $k + 1$ is

then given by

$$T_{ij}^k = \frac{1}{\Delta t}\left\{\left[1+\frac{\nu_r^{(l)}}{2}\left(\delta_r^2+\frac{1}{i+\frac{1}{2}}\Delta_{0r}\right)\right]\left[1+\frac{\nu_z^{(l)}}{2}\delta_{z_l}^2\right]p'(r,z,t+\Delta t)- \left[1-\frac{\nu_r^{(l)}}{2}\left(\delta_r^2+\frac{1}{i+\frac{1}{2}}\Delta_{0r}\right)\right]\left[1-\frac{\nu_z^{(l)}}{2}\delta_{z_l}^2\right]p'(r,z,t)\right\}. \quad (5.18)$$

We can expand each term of this expression in Taylor series in each variable about the point (r,z,t). Making use of the differential equation $p'_t - D(p'_{rr}+\frac{1}{r}p'_r+p'_{zz})$ together with the mesh ratios $\nu_r^{(l)} = D_l\Delta t/\Delta r^2$ and $\nu_z^{(l)} = D_l\Delta t/\Delta z^2$ we obtain

$$T_{ij}^k = \frac{\Delta t^2}{12}\left\{2p'_{3t}-3D_l\left(\frac{1}{r}p'_{rtt}+p'_{rrtt}+p'_{zztt}\right)\right\}- \frac{D_l}{12}\left\{\Delta r^2\left(p'_{4r}+\frac{2}{r}p'_{3r}\right)+\Delta z_l^2 p'_{4z}\right\}+O(\Delta t^3,\Delta r^3,\Delta z_l^3). \quad (5.19)$$

As expected this expression is second order in both space and time, with the time–terms in the first bracket decaying rapidly with increasing time.

5.6.4 Global error and convergence

The global error in the partial pressure values at any particular time is made up of the sum of the LTE at each step, together with the errors propagated from all the other approximations. Since we must numerically differentiate these values to obtain the current, we expect to lose an order of accuracy and obtain first order convergence in the current. Table 5.4 shows the convergence of the current[3] along various refinement paths for cathode radii of $5\mu m$, $10\mu m$, $20\mu m$, and $40\mu m$. Each case uses a $12\mu m$ PTFE membrane protecting an $8\mu m$ electrolyte layer at time $t = 0.1s$, with the other parameters values as given in Table 5.3. This time period is of the same order as that used when pulsing Clark electrodes, and corresponds to the diffusion front beginning to eat into the membrane layer as shown in Figures 5.4 and 5.5.

We choose the refinement path for each case by first fixing both the aspect ratio $\Delta r : \Delta z$ of the mesh and the mesh ratio $\nu = \nu_r + \nu_z$. We then refine the mesh by successively doubling the number of points in each direction whilst keeping the mesh ratio constant by quartering the timestep. For example, the first case shown in Table 5.4 for the $5\mu m$ radius cathode has an aspect ratio of $1:2$ and a timestep of $6.25\times10^{-4}s$, giving a mesh ratio $\nu = 13.125$ and a current of 249.5. Referring to Figure 5.1 this value is calculated using $n_r = 10$, $m_e = 8$, $m_m = 12$, $N = 100$, and $M = 24$, each of which is doubled to obtain the second current value of 254.7, whilst the timestep is quartered to $1.5625\times10^{-4}s$. However, no points on successive meshes lie exactly on top of one another, since we shift the mesh by

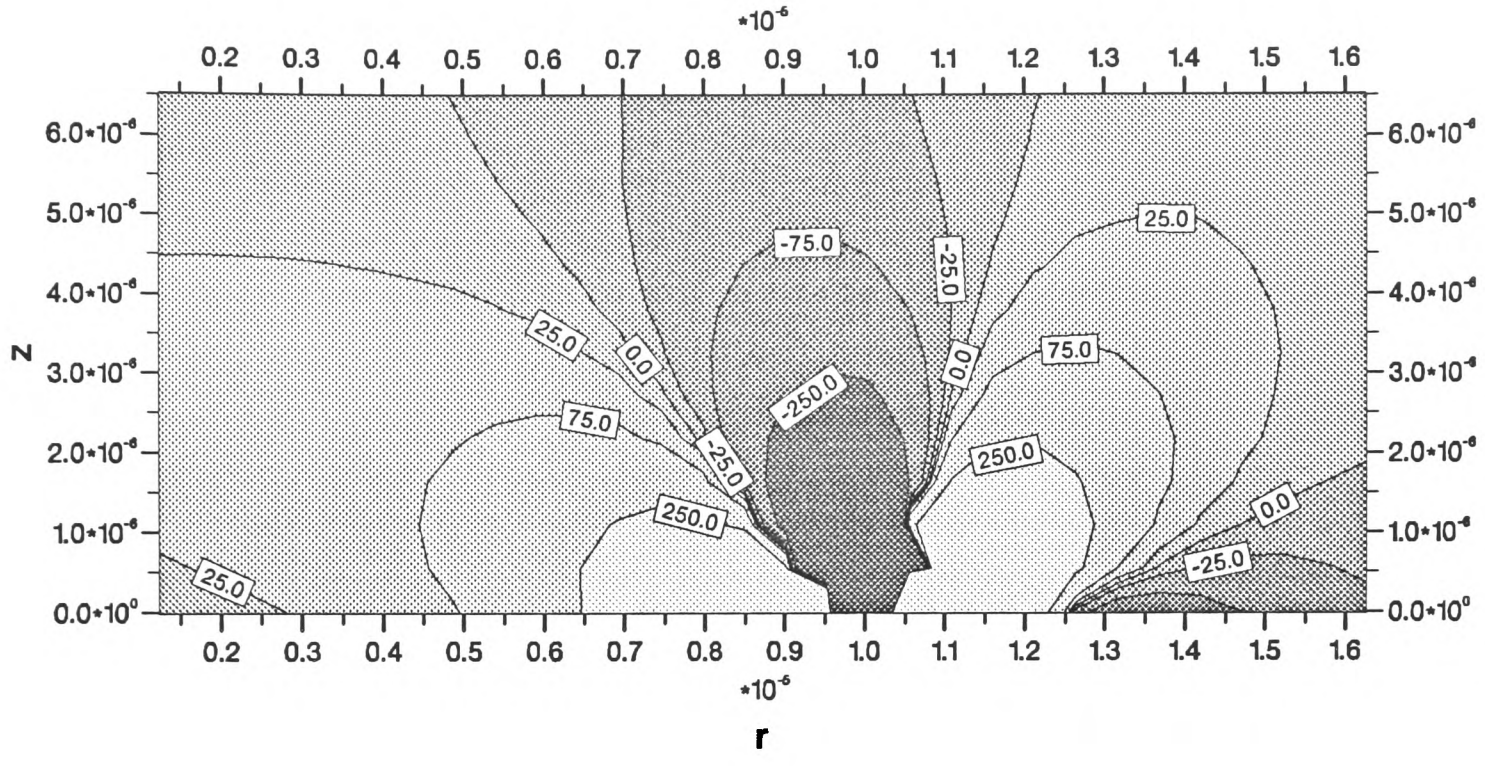
[3]Whenever we quote values of the current it will always be the current per unit area of cathode per unit atmosphere, in units of $Am^{-2}atm^{-1}$, so that we can more easily compare the currents obtained when the cathode radius is varied.

r_c	n_r	$\Delta r = \frac{1}{2}\Delta z$ ($\nu = 13.1$)	$\Delta r = \frac{3}{4}\Delta z$ ($\nu = 8.2$)	$\Delta r = \Delta z$ ($\nu = 10.5$)	$\Delta r = 2\Delta z$ ($\nu = 13.1$)
5 μm	10	249.5	260.9	267.3	279.5
	20	254.7	262.0	266.1	274.2
	40	257.1	261.9	264.6	270.2
	80	258.0	261.3	263.2	267.0
		$\Delta r = \frac{1}{2}\Delta z$ ($\nu = 13.1$)	$\Delta r = \Delta z$ ($\nu = 10.5$)	$\Delta r = 2\Delta z$ ($\nu = 13.1$)	$\Delta r = 3\Delta z$ ($\nu = 13.1$)
10 μm	10	114.6	130.4	140.5	145.3
	20	123.5	133.0	139.1	142.2
	40	128.1	133.9	137.7	139.7
	80	130.3	134.0	136.5	137.8
		$\Delta r = \frac{1}{2}\Delta z$ ($\nu = 13.1$)	$\Delta r = \Delta z$ ($\nu = 10.5$)	$\Delta r = 2\Delta z$ ($\nu = 13.1$)	$\Delta r = 3\Delta z$ ($\nu = 13.1$)
20 μm	20	58.6	68.8	74.8	77.5
	40	65.9	71.7	75.2	76.8
	80	69.6	73.0	75.0	76.0
	160	71.5	73.5	74.7	75.4
		$\Delta r = \Delta z$ ($\nu = 10.5$)	$\Delta r = 2\Delta z$ ($\nu = 13.1$)	$\Delta r = 3\Delta z$ ($\nu = 13.1$)	$\Delta r = 4\Delta z$ ($\nu = 11.2$)
40 μm	40	42.02	46.2	47.8	48.7
	80	44.8	47.0	48.0	48.5
	160	46.0	47.3	47.9	48.2
	320	46.6	47.4	47.8	47.9

Table 5.4: Convergence of current per unit area along various refinement paths for increasing sizes of cathode radius, where the mesh ratio $\nu = \nu_r + \nu_z$.

$\frac{1}{2}\Delta r$ in the r–direction to overcome the singularity of $\frac{1}{r}\frac{\partial p}{\partial r}$ along $r = 0$. In all cases shown in Table 5.4, we choose N and M to be sufficiently large that the diffusion front does not reach the boundary.

As can be seen, for each radius the global error changes sign as the aspect ratio of the mesh is altered. For example, for the $5\mu m$ cathode radius, the current is underestimated on meshes with $\Delta r : \Delta z$ ratio of $1 : 2$, the error changes sign at a ratio of about $3 : 4$ and overestimates the current for ratios greater than $1 : 1$ in favour of z. This pattern is repeated for all four of the radii considered, although the switch in the sign of the error occurs at roughly $1 : 1$ for the $10\mu m$ cathode, $2 : 1$ for the $20\mu m$ cathode, and $3 : 1$ for the $40\mu m$ cathode. In each case, when we are converging to the solution from below, the error is roughly halved on successive meshes i.e. we have first order convergence. However, once the error has switched sign and the current is converging from above, the values on the coarser meshes are too small for convergence to be first order.



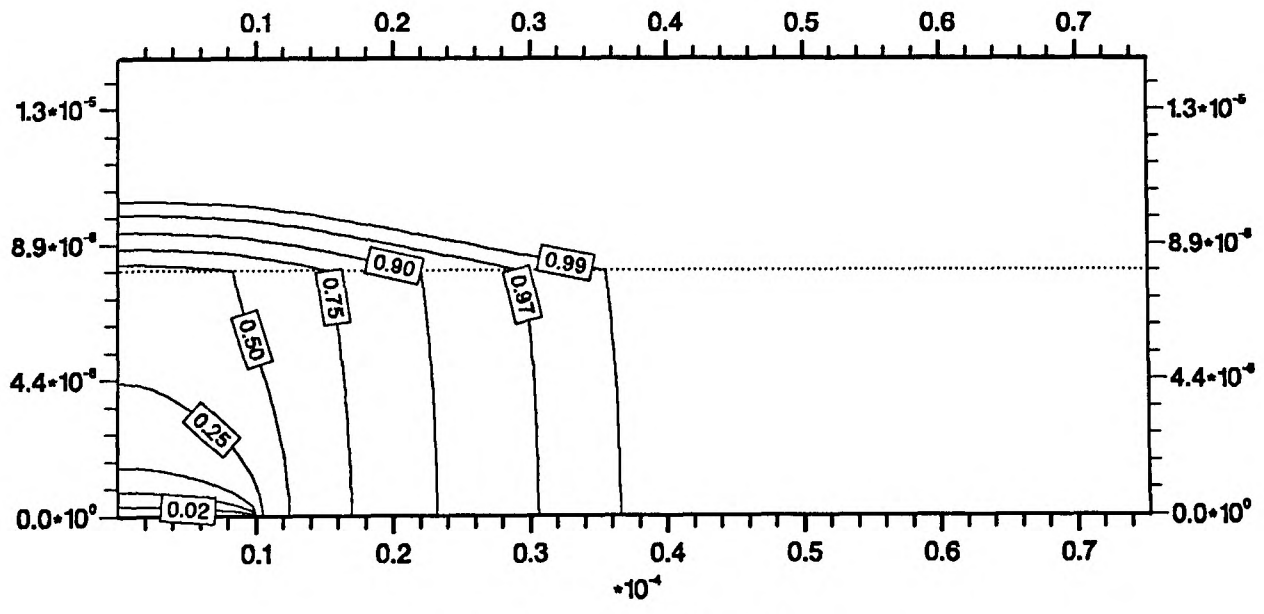


Fig. 5.4: Contour plot of spatial truncation error estimate within the electrolyte in the region of the singularity for a $10\mu m$ radius cathode, $8\mu m$ electrolyte layer, $12\mu m$ PTFE membrane, at $t = 0.1s$. The corresponding contour plot of the solution for the partial pressure is shown in both the electrolyte and membrane layers.

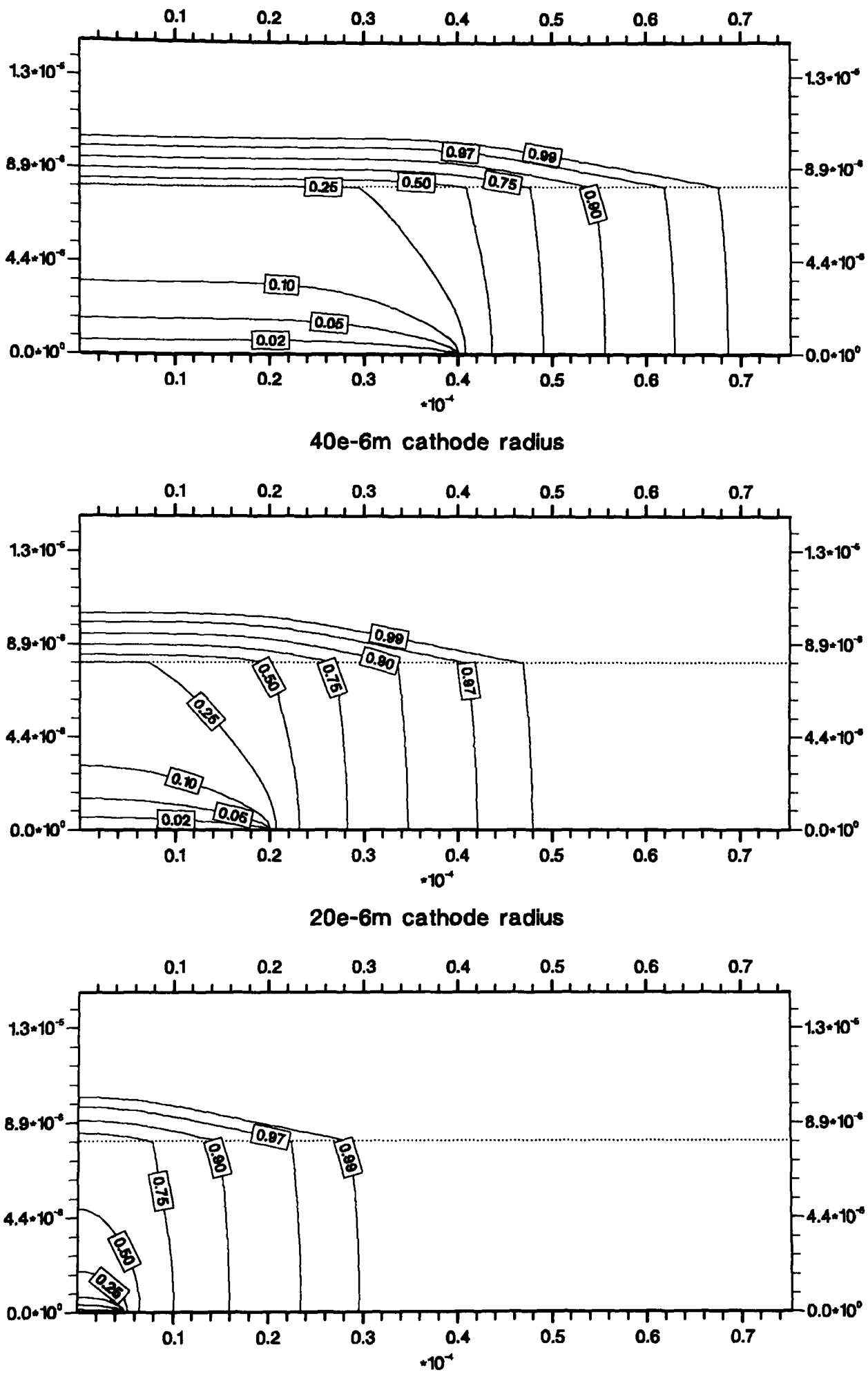


Fig. 5.5: Contour plots of equal partial pressure for the other three cathode radii considered in Table 5.4. As the cathode radius increases, the dominant diffusive component moves from radial to linear.

We will show that these effects are due to the combination of errors from the LTE of the ADI method (when using too large a timestep), and those from the singularity correction (when using too *small* a timestep), together with the changing nature of the solution for the partial pressure with cathode radius. This will result in the need to use comparatively fine meshes to remove both types of error for some refinement paths.

5.6.5 Aspect ratio dependent error effects

To get a feel for why the global error might change in sign, we return to the truncation error. Since the time terms in equation (5.19) decay with time, we can obtain an indication of the pattern of the LTE over the whole mesh by considering only the spatial derivatives in this expression. We can obtain rough approximations to each term by numerically differentiating the approximate solution at each mesh point. The approximations to both the third and fourth derivatives in a particular direction require the values of the two immediate neighbours to either side in that direction. Figure 5.4 shows a contour plot of the spatial terms of the LTE obtained in this manner. It is shown only in the electrolyte since the higher derivatives are not continuous across the interfaces. The particular case shown is for the $10\mu m$ cathode radius with $\Delta r = \Delta z$ and $n_r = 20$ giving a current of 133.0 in Table 5.4. The LTE is a corrugated function of r and z with two positive lobes separated by two negative lobes, the lines of separation originating at the singularity. The contribution to the global error in the current results from the propagation of the LTE incurred at each mesh point at each timestep. By altering the aspect ratio of the mesh, we change the way in which the errors at each mesh point interact, and over many timesteps (640 in the case shown in Figure 5.4) this results in the refinement path dependent errors of Table 5.4. The areas where the LTE is large correspond, as expected, to the areas of high curvature in the solution which is shown in Figure 5.4.

The shift in the aspect ratio at which the sign of the error switches as the cathode radius is increased corresponds to a shift from dominant radial diffusion for the smaller cathodes, to dominant linear diffusion for the larger cathodes, as shown in Figures 5.4 and 5.5. This means that as the cathode radius increases we can obtain good answers over the range of interest without having to use too fine a mesh in the r–direction. The number of mesh points in the r–direction covering the cathode can therefore be kept comparable to the number in the electrolyte in the z–direction.

5.6.6 Timestep dependent error effects

Two distinct error effects are due to the size of the timestep. The first is caused by using too small a timestep and is due to the singularity correction, the second from using too large a timestep is due to the finite difference approximation. Table 5.5 shows the current obtained at $t = 0.1s$ for a $20\mu m$ radius cathode with $\Delta r = \Delta z$

for different values of the mesh ratio $\nu = \nu_r + \nu_z$, a large value of ν corresponding to a large timestep. On the coarsest mesh, with $n_r = 20$, $\Delta r = \Delta z = 1.0 \times 10^{-6} m$, the current is timestep dependent at all mesh ratios considered, wildly overestimating the current for very large timesteps before beginning to oscillate[4]. As the mesh is refined (again keeping the mesh ratio constant), the region over which the calculated current is timestep–independent grows, until on the finest mesh with $n_r = 80$, $\Delta r = \Delta z = 1.25 \times 10^{-7} m$, the calculation of the current remains constant for all *sufficiently large* timesteps.

In order to keep the mesh ratio ν constant between successively finer meshes it is necessary to quarter the timestep each time we double the number of mesh points. Since the error due to the time terms of the LTE in equation (5.19) is also second order it decreases much more rapidly than that due to the spatial terms as the mesh is refined. Since the overestimation of the current observed in Table 5.4 on coarse meshes for large ν values is not carried through onto the fine meshes (which have the same ν value but a comparatively small timestep), it is likely that this error is introduced by the time–terms in the LTE. This is confirmed by the very large currents we observe if we increase the mesh ratio still further in Table 5.5, with the current eventually oscillating between positive and negative for very large mesh ratios. This means that on the finer meshes on which we will need to do most of our calculations, we can maintain the time accuracy whilst using a relatively large timestep, needing only to halve the timestep between successively halved spatial meshes to maintain the same time accuracy.

The cause of the underestimation of the current when a small value of ν (and therefore of the timestep) is used is illustrated in Figure 5.6. Since the singularity correction itself does not take account of time explicitly but only through the variation of the values at the far points, whenever the diffusion front has not reached any of the far points on the first timestep (Case 1 in Figure 5.6), their value remains unchanged and the singularity correction will always give the value corresponding to this at the near points. The correction is based on a truncated series expansion so that the practical effect in this case is to underestimate the values at the near points. This error is then transmitted to the next timestep when the ADI method uses the corrected values at the near points. The underestimation of the current therefore continues throughout the calculation, decreasing too slowly to have vanished at all times of interest. If the timestep is lowered still further, this effect is heightened since the diffusion front may not in reality reach the far points for several timesteps, but the correction is still done after each timestep, with the resulting error passed on to the next. For example continuing the sequence given in Table 5.5 on the coarsest mesh gives a value of 67.53 at $\nu = 1.32$ and 66.93 at $\nu = 0.66$. In Case 2 of Figure 5.6 the diffusion front has not passed all of the far

[4]The method does not go unstable, however, and at longer times these time–dependent effects decay away and we can obtain an accurate solution using a large mesh ratio.

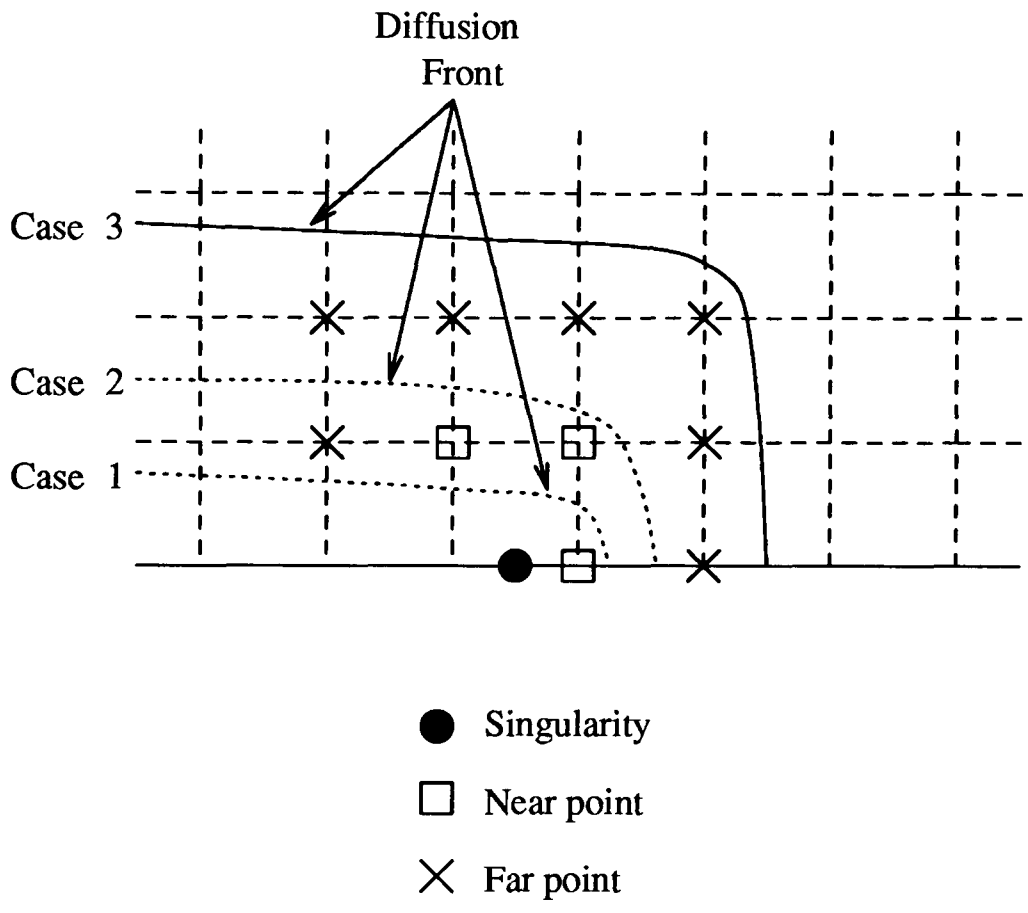


Fig. 5.6: Position of the diffusion front at the end of the first timestep. In Case 1 the current will be underestimated since the diffusion front has not yet reached any of the far points. In Case 2 the correction at the near points will be based only on the change in value of one of the far points and will not be robust. We must therefore choose a sufficiently large timestep so that we obtain Case 3.

n_r	$\nu = 2.63$	$\nu = 5.25$	$\nu = 10.5$	$\nu = 21.0$	$\nu = 42.0$	$\nu = 84.0$
20	68.16	68.60	68.78	71.98	133.5	−223.8
40	71.22	71.56	71.72	71.73	79.21	240.1
80	72.60	72.85	72.97	73.00	72.98	91.76
160	73.18	73.36	73.45	73.47	73.47	73.47

Table 5.5: Timestep–dependent errors in the current for the $20\mu m$ cathode, using an aspect ratio of 1:1 so that $\nu_r = \nu_z = \frac{1}{2}\nu$. The timestep is $\Delta t = 6.25 \times 10^{-4} s$ and the mesh spacing $\Delta r = \Delta z = 1.0 \times 10^{-6} m$ for the first value in the table.

points and the correction will not be robust. We therefore place the condition on the timestep that it be large enough for Case 3 to hold, where the diffusion has moved past all the far points after the first timestep.

The first two columns of Table 5.5 show that this effect is present on all meshes, although less prominent on the finer meshes. This is because there is again a square relationship between space and time, with the time taken for the diffusion front to move a given distance being proportional to the square of the distance. Quartering the timestep whilst halving the spatial mesh length simply ensures that on successive meshes the diffusion front is in an identical position relative to the far points, and the effect is maintained at a fixed mesh ratio however fine the mesh. The effect is reduced on the finer meshes only because the region of correction becomes smaller compared to the overall solution region.

We can get an idea of how large a timestep we require to alleviate this problem using the simple relationship between characteristic transport time and distance (see Myland and Oldham [73]) to give

$$\Delta t_{\min} > \frac{\rho_{\max}^2}{D_e}, \tag{5.20}$$

where $\rho_{\max}$ is the distance of the furthest far point from the singularity. For all values given in Table 5.5, we used $\Delta r = \Delta z$ so that $\rho_{\max} = 2.5\Delta r$ and $\nu = 2D_e\Delta t/\Delta r^2$, which together give the condition $\nu > 12.5$ as a rough guide, which agrees well with the results obtained in practice.

If we now return to the problem of convergence from above in Table 5.4, we can see that by choosing the timestep small enough to remove time–dependent errors in the LTE, we have violated equation (5.20). For example, for the $5\mu m$ cathode radius with $\Delta r = 2\Delta z$, equation (5.20) requires that $\nu > 16.2$, and it is not possible to avoid both types of time–dependent error on our coarsest mesh. Since the distance to the far point furthest from the singularity increases with the aspect ratio of the mesh, the minimum allowable mesh ratio becomes quite large and it is not possible to obtain a solution unaffected by either of the timestep errors until the mesh is sufficiently fine. The worst case in Table 5.4 is for the

r_c	n_r	$\Delta r = \frac{1}{2}\Delta z$	$\Delta r = \Delta z$	$\Delta r = 2\Delta z$	$\Delta r = 3\Delta z$
20 μm	40	65.85	71.73	—	—
	80	69.62	73.00	74.76	—
	160	71.45	73.47	74.57	75.49
	320	72.45	73.64	74.43	74.86
Extrapolated		73.45	73.81	74.29	74.23

Table 5.6: Convergence of values of current for the $20\mu m$ cathode radius following the rules given in the text for minimising errors. — represents refinement paths where we could not remove both types of timestep–dependent error.

$40\mu m$ radius with $\Delta r = 4\Delta z$ for which equation (5.20) requires that $\nu > 42.5$, and obeying this condition would introduce errors due to the LTE of the ADI method on all but the two finest meshes used.

5.6.7 Conclusions

Throughout this section we have described how the amalgamation of the various analytical and numerical techniques that we use to obtain a value for the current leads to a complex error pattern. In analysing each of the contributions we have suggested certain rules and restrictions to ensure reliable results, which are summarised below:

(1) To avoid a timestep–dependent error for the cathode radii we have so far considered, the number of mesh points covering the cathode should be at least 20, with a similar number in the electrolyte layer;
(2) The minimum timestep can be calculated from equation (5.20);
(3) The upper limit on the timestep can be obtained straightforwardly by numerical experiment;
(4) Once in the timestep–error free region, the timestep need only be halved when the space steps are halved;
(5) The refinement path should be chosen
    (i) To converge from below to give timestep–error free values on coarser meshes, whilst
    (ii) Being close to the switch from underestimation to overestimation so that most of the errors due to the LTE of the ADI method cancel out.

Applying these rules to the $20\mu m$ case given in Tables 5.4 and 5.5 we obtain the results given in Table 5.6, where the extrapolated values are calculated assuming simple first order convergence on the finer meshes. The extrapolated values along all the refinement paths are well within 1% of their combined mean. The mesh of choice according to these rules would be the 1 : 1 aspect ratio in column 2 of Table 5.6.

5.7 Summary

In examining the convergence of the solution for the current with mesh refinement we have shown that whilst the various errors incurred interact in a complex manner, provided that we choose the refinement path with sufficient care we can obtain sufficiently accurate results. This examination of the likely sources of error has been made possible by the speed of the parallel processing facilities with the largest possible mesh (800×800) requiring about $2.8s.$ per complete timestep to calculate the solution on all 16 processors. Since the current is derived from the partial pressure values by numerical differentiation, we obtain much greater accuracy for the approximate partial pressure solution and we will make use of this fact when graphically examining the diffusion processes in the next chapter.

Chapter 6

Operation and Design of the Clark Electrode

The method of solution described in Chapter 5 was shown to be sufficiently accurate that we can now use it as a numerical tool for investigating the variation in the operation of the Clark electrode with the governing parameters, and to examine the resulting implications for its design. However, we first make use of it to demonstrate where and why the one–dimensional models of previous workers have proved inadequate, by comparison of the variation with cathode radius and electrolyte layer thickness. By considering cathode radii of differing magnitudes, we then go on to examine the time–dependent variation of the current for diffusion processes dominated by radial diffusion, linear diffusion, and a combination of both. This allows us to choose appropriate times at which to investigate the variation of the current with the parameters of interest listed at the end of Chapter 2. Whilst the time–dependent current is our primary interest for a continuously applied potential, we can infer physical explanations for its variation by using surface and contour plots of the solution for the partial pressure, and gain insight into the likely speed of response to step changes in sample P_{O_2}. The understanding that we gain allows us to suggest an appropriate geometry for optimal results under pulsed operation. This has been shown in the laboratory to overcome many of the practical difficulties caused by the requirements of rapid response time and flow–independence, requirements that are mutually incompatible when using the steady–state technique for *in vivo* measurements.

6.1 Comparison with One–Dimensional Theory

In Chapter 2 we briefly described the previous work that had been done in using one–dimensional models to investigate the operation of the Clark electrode. These assumed the diffusion process to be either purely axial (linear diffusion) or purely radial (spherical diffusion), but still did not yield a generally valid analytic solution. It was found in practice that both of these models, but particularly the linear model, grossly underestimated the current measured by experiment for all but very large cathode radii. Since we wish to compare these models with our two–dimensional model, we obtain numerical solutions to equations (2.9)

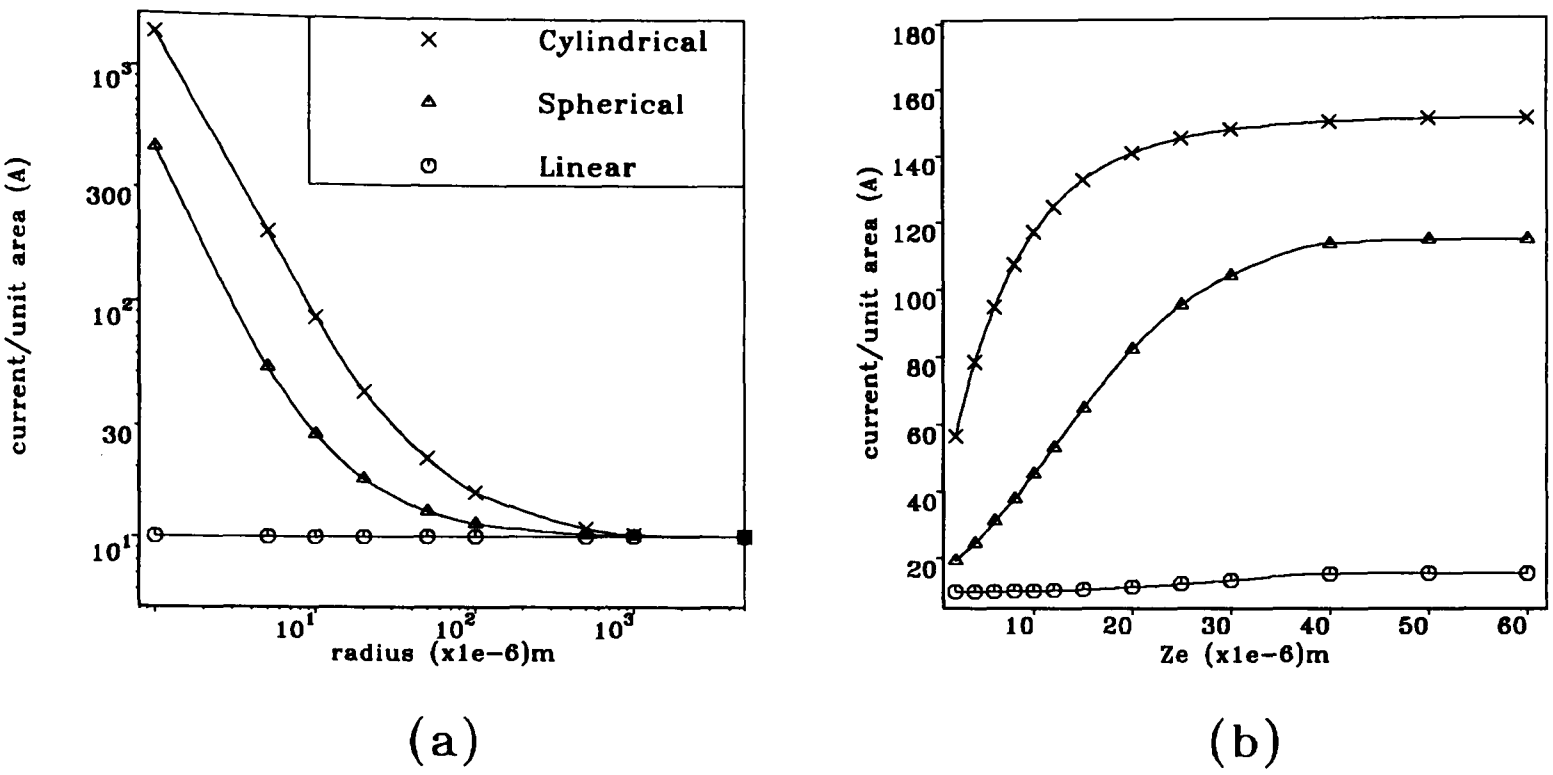


Fig. 6.1: Variation of current with (a) Cathode radius and (b) Electrolyte layer thickness for a $10\mu m$ radius, both at time t=0.5s.

and (2.13) subject to their respective boundary conditions via implicit Crank–Nicholson finite–difference approximations [14]. The Crank–Nicholson method again approximates the partial derivatives using central differences, and since both models are one–dimensional we need solve only one tridiagonal system at each timestep to update at all mesh points, which can be done as in Chapter 4 when solving for a particular mesh line. Both boundary conditions are now Dirichlet and are therefore imposed at the start of the calculation, whilst the interface conditions are incorporated into the tridiagonal systems using equation (2.8). Since the methods are one–dimensional, conventional sequential computing resources facilitate a sufficiently accurate numerical solution.

6.1.1 Variation with cathode radius

To investigate whether there is any region in which diffusion to the cathode can be adequately modelled in one–dimension, we examine the variation of the current over the range of cathode radii of $1\mu m$ to $5000\mu m$. So that we can consider diffusion in all three layers at a reasonably short time, we choose a thin $5\mu m$ PTFE membrane over a $5\mu m$ electrolyte layer (which is in the middle of the practical range) at time $t = 0.5s$. Table 6.1(a) and Figure 6.1(a) show the numerically calculated values of the current obtained from the linear, spherical and cylindrical models respectively.

As expected, both of the one–dimensional models underestimate the current for all sizes of cathode. The linear diffusion model assumes an infinite cathode radius

r_c (μm)	Current Linear	Spherical	Cylindrical	z_e (μm)	Current Linear	Spherical	Cylindrical
1	10.05	445.0	1335.0	2	9.99	19.4	56.7
5	10.05	53.21	199.3	4	10.03	24.6	78.5
10	10.05	27.78	87.15	6	10.08	31.1	94.8
20	10.05	17.93	42.25	8	10.17	37.9	107.3
50	10.05	12.97	22.00	10	10.25	45.3	116.9
100	10.05	11.47	15.66	12	10.39	53.2	124.3
500	10.05	10.32	10.93	15	10.69	64.8	132.4
1000	10.05	10.18	10.24	20	11.38	82.3	140.6
5000	10.05	10.07	10.11	25	12.36	95.4	145.1
				30	13.44	104.0	147.7
				40	15.55	113.4	150.1
				50	15.60	114.4	150.9
				60	15.62	114.5	151.1
	(a)				(b)		

Table 6.1: (a) Variation of current (a) with cathode radius and (b) with electrolyte layer thickness at time $t = 0.5s$ for a PTFE membrane with $z_e = z_m = 5\mu m$. Columns 2 and 6 give the solution of equation (2.9), columns 3 and 7 the solution of equation (2.13), and columns 4 and 8 the solution of equation (2.2).

and therefore gives the same value, 10.05[1], for all radii. Whilst it can be seen that both the spherical and cylindrical models tend to this value as the cathode radius becomes large, it is not until the radius is greater than $1mm$ that they are within 1% of it. For very small cathode radii, the strength of the radial component of diffusion in the two–dimensional model is evident in the very large values of current per unit area. However, the value of 1335.0 given for the $1\mu m$ cathode radius in Table 6.1 corresponds to a current of only $4.2nAatm^{-1}$. This compares to a value of $27nAatm^{-1}$ and $490nAatm^{-1}$ for the $10\mu m$ and $100\mu m$ radii respectively.

It has been suggested [22, 64, 94] that the spherical model is a good approximation to the cylindrical model for sufficiently small cathode radii. Table 6.1(a) demonstrates that this is not the case, with the spherical model also greatly underestimating the current. We can gain an understanding of this failure by considering the nature of the diffusion to the cathode which is shown in Figure 6.2 for $1\mu m$, $10\mu m$, and $100\mu m$ cathode radii, in each case protected by a $5\mu m$ PTFE membrane over a $5\mu m$ electrolyte layer. In modelling the diffusion of oxygen as purely radial by taking the cathode and membrane to be hemi–spherical, in addition to changing the geometry at the electrode edge, we have limited the volume of the electrolyte layer and the inner surface area of the membrane over which diffusion

[1]Throughout this chapter we will again assume, unless otherwise stated, that p_0=1.0, so that all current values quoted are *per unit area per unit atmosphere* in units of $Am^{-2}atm^{-1}$

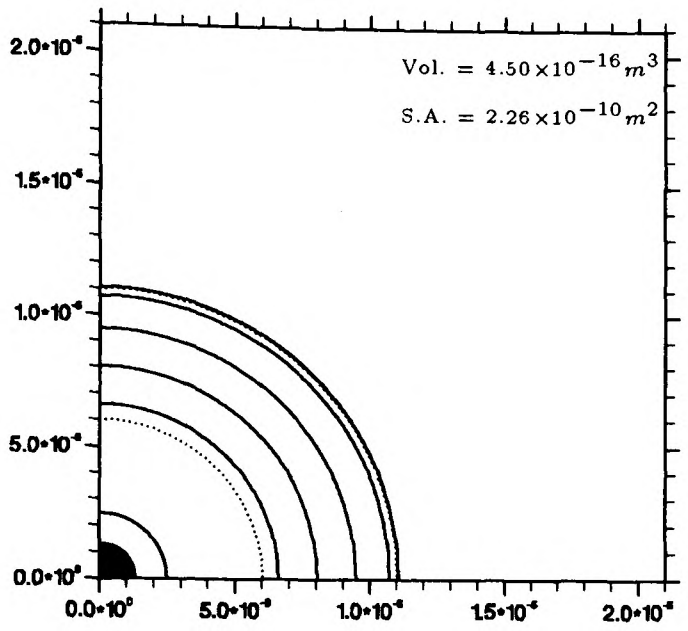
can take place. If we consider the three cases of cylindrical diffusion shown in Figure 6.2 we see that the contours of equi–partial pressure spread out radially in the electrolyte layer, allowing oxygen to reach the cathode from a large surface area of the membrane and a large volume of electrolyte. On each contour plot we give the volume of electrolyte and the inner surface area of the membrane that are enclosed by the last contour at $0.99p_0$.

For all three cases the electrolyte has been depleted to a much greater extent using the cylindrical model than is possible with the spherical model where the cathode is completely enclosed. Paradoxically, the problem becomes worse as the cathode radius is reduced since the greater radial component of diffusion allows oxygen to be drawn in to the cathode from further and further into the electrolyte with the cylindrical model. For example for the $1\mu m$ cathode the diffusion front has eaten into the electrolyte to over 20 times its radius. Oxygen is therefore reaching the cathode through a volume of electrolyte about 20 times and a surface area of membrane about 15 times the maximum possible with the spherical model. It is not surprising that with the spherical model which allows no mechanism of lateral replenishment of oxygen to the cathode, the small volume of electrolyte is stripped of oxygen much more rapidly, with the consequent underestimation of the current, which will become worse for longer times as the diffusion front eats further radially into the electrolyte layer. It is clear, therefore, that the spherical model is only valid for a truly hemispherical cathode protected by a hemispherical membrane. Both the spherical and linear models will give a good approximation to the current with a very large ($> 1mm$) flat disc cathode, but the approximation deteriorates rapidly as the radius is reduced.

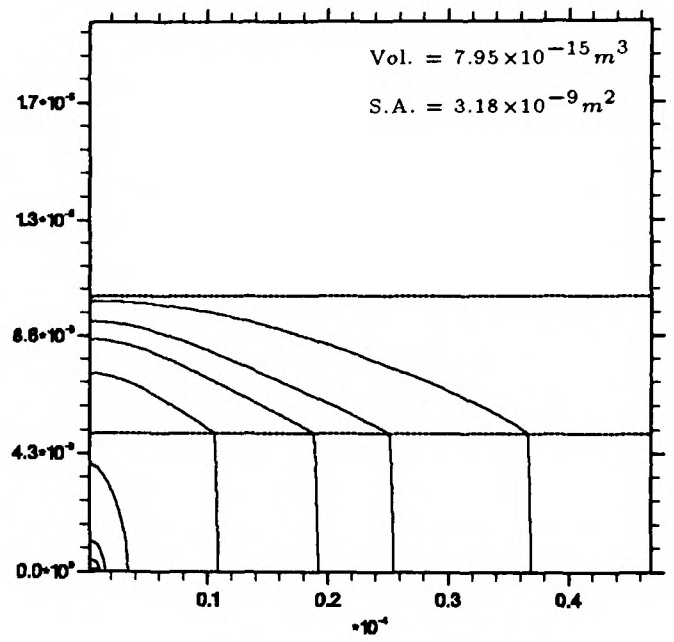
6.1.2 Variation with electrolyte layer thickness

Since two of the three special cases considered by Mancy *et al* correspond to very thick electrolyte layers ($z_e/\sqrt{D_e} = z_m/\sqrt{D_m}$ and $z_e/\sqrt{D_e} >> z_m/\sqrt{D_m}$), the variation of the current for such cases has attracted much attention [8, 39, 73]. The major concern has been the extent to which the time–dependent variation of the current can be considered "Cottrellian" i.e. directly proportional to $t^{-\frac{1}{2}}$ (see section 3.1), a consideration with which we will deal in the next section. In Table 6.1(b) and Figure 6.1(b) we therefore show the values obtained from the three models as the electrolyte layer thickness is varied from $2\mu m$ to $60\mu m$, going well beyond the usually quoted practical limit of $10\mu m$, for a 10μm cathode radius and a 5μm PTFE membrane at time $t = 0.5s$. Once again both one–dimensional models underestimate the current, the linear model grossly. In this case, the curves for the spherical and cylindrical models are a different shape as the electrolyte layer thickness is increased, with the cylindrical model giving a much more rapid initial increase with electrolyte layer thickness, before tending towards the unshielded current at a lower value of z_e than the spherical model.

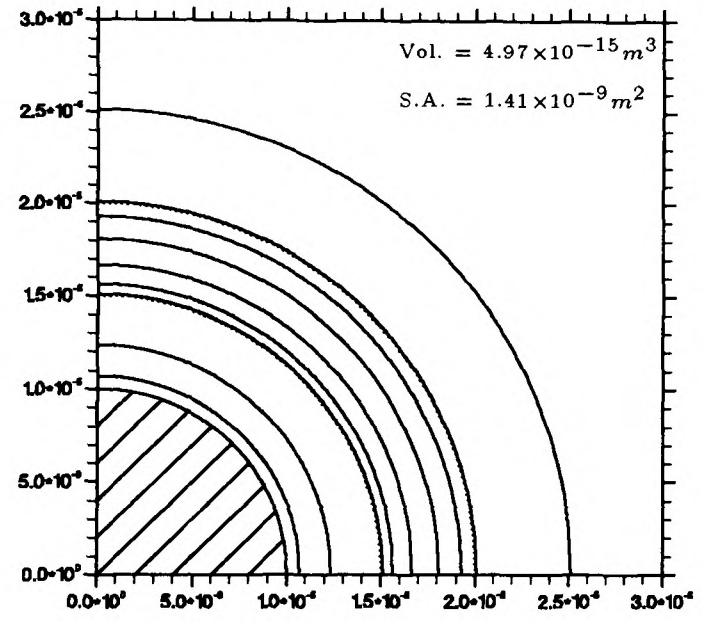
Figure 6.3 compares the diffusion patterns obtained from the two models for a



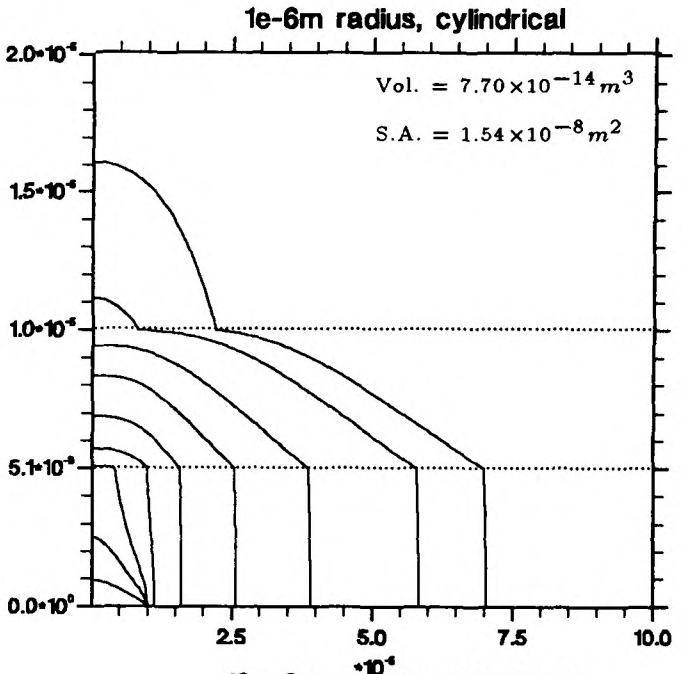
1e-6m radius, spherical



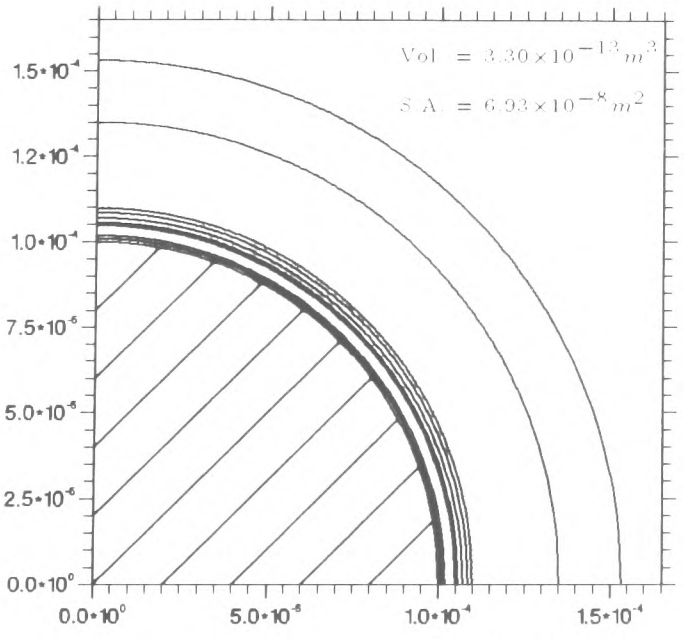
1e-6m radius, cylindrical



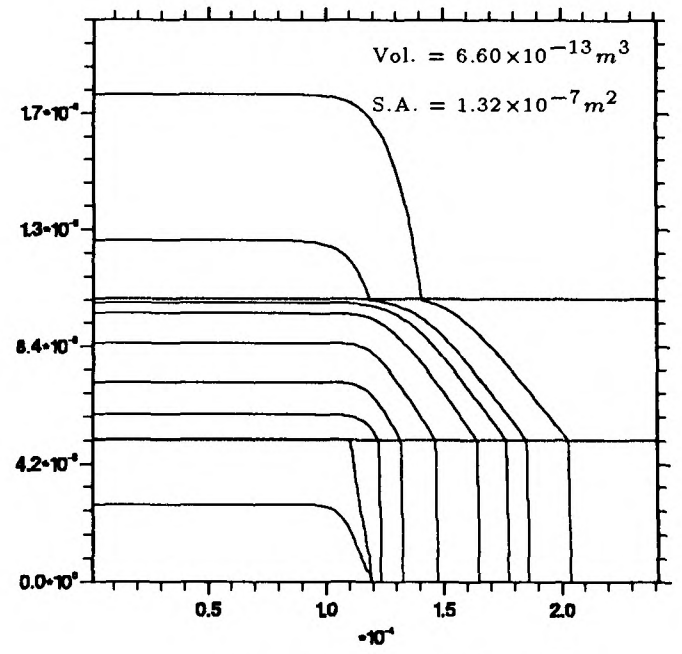
10e-6m radius, spherical



10e-6m radius, cylindrical

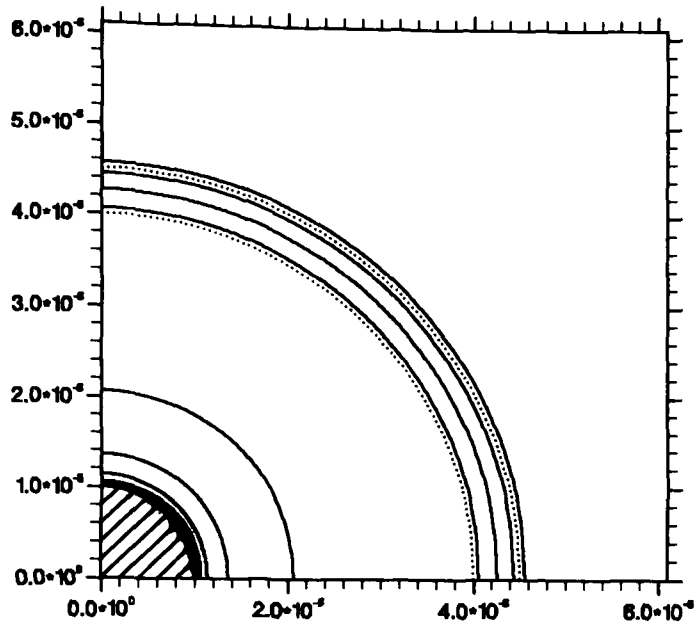


100e-6m radius, spherical

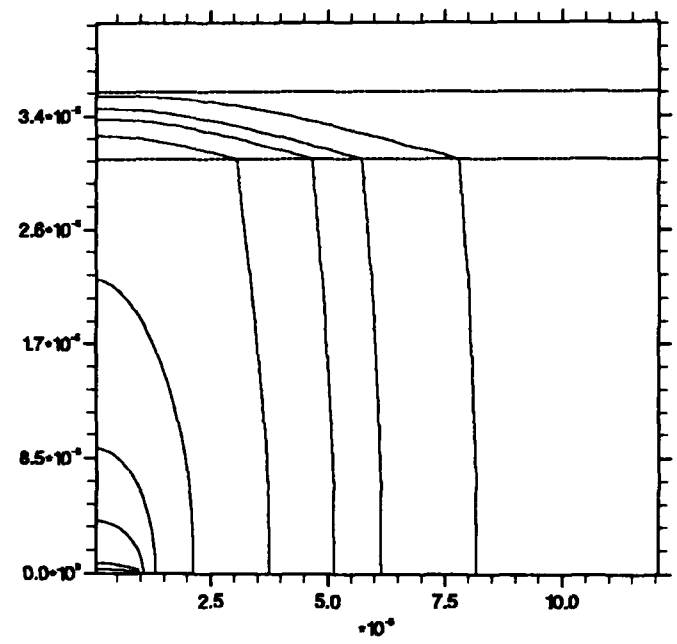


100e-6m radius, cylindrical

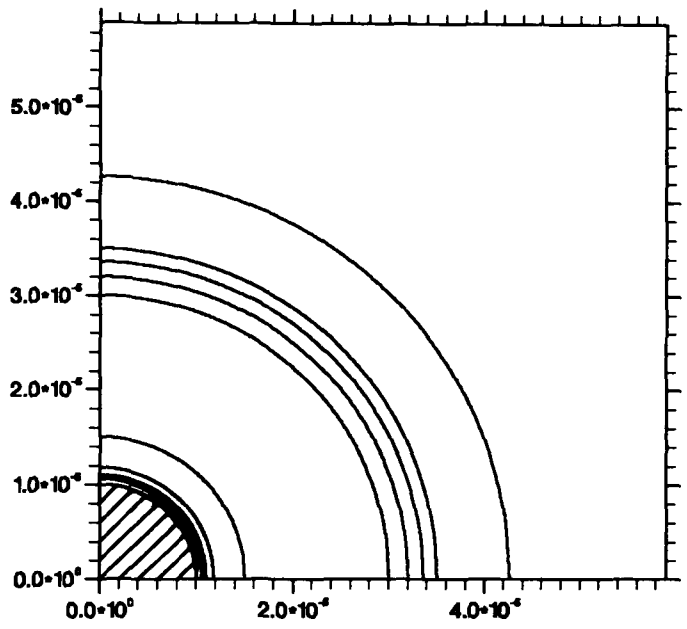
Fig. 6.2: Contour plots of equal partial pressure for $1\mu m$, $10\mu m$ and $100\mu m$ radius cathodes using the one-dimensional spherical model and the two–dimensional cylindrical model at time $t = 0.5s$. The contours are at $p/p_0 = 0.02$, 0.05, 0.1, 0.25, 0.5, 0.75, 0.9, 0.97, and 0.99. The values of the volume of electrolyte(Vol.) and inner surface area of membrane(S.A.) refer to that quantity contained within the 0.99 contour line.



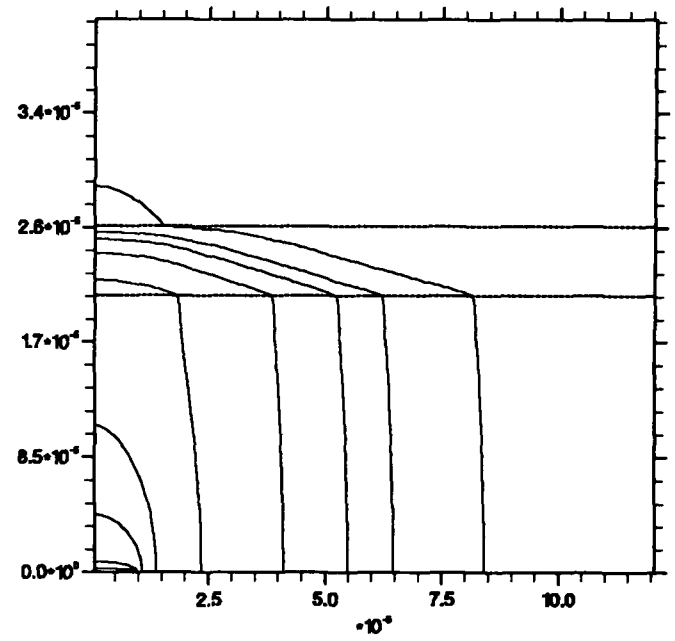
Spherical, 30e-6m electrolyte layer



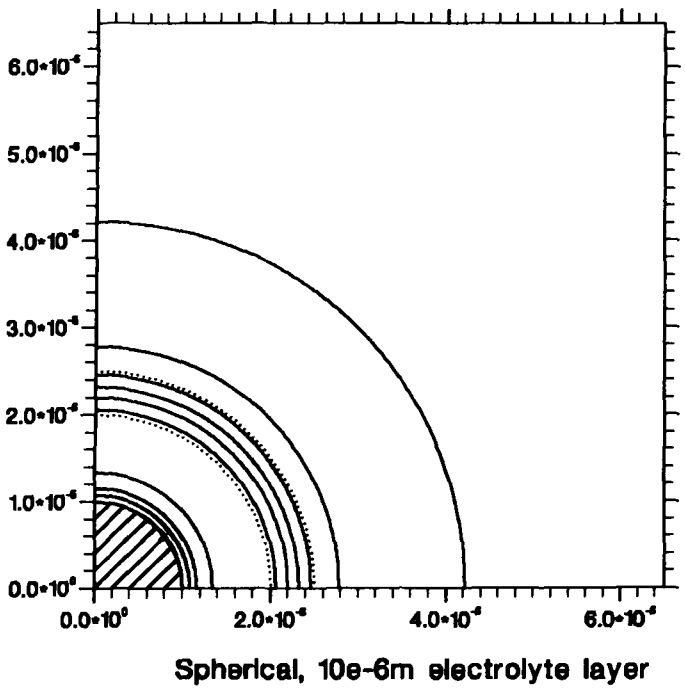
Cylindrical, 30e-6m electrolyte layer



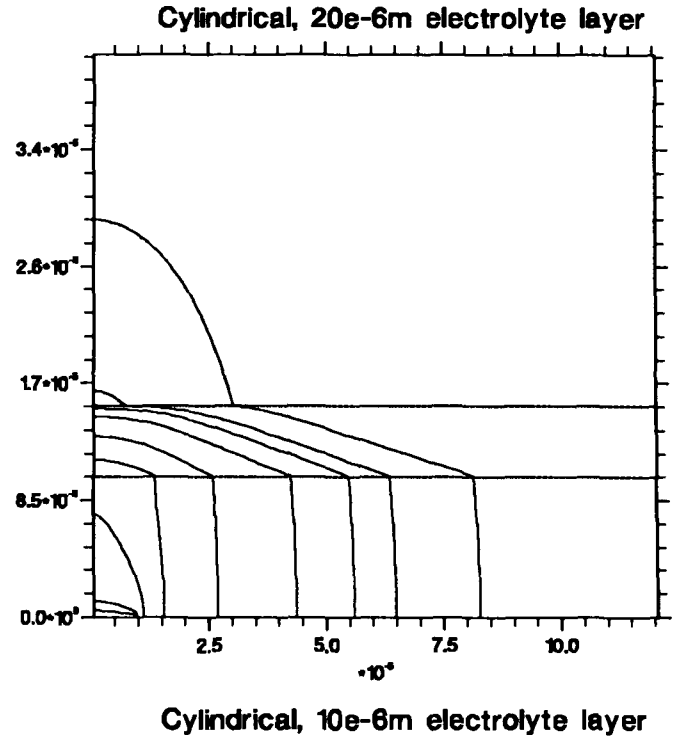
Spherical, 20e-6m electrolyte layer



Cylindrical, 20e-6m electrolyte layer



Spherical, 10e-6m electrolyte layer



Cylindrical, 10e-6m electrolyte layer

Fig. 6.3: Contour plots of equal partial pressure for a $10\mu m$ radius cathode using the one-dimensional spherical model and the two–dimensional cylindrical model at electrolyte layer thicknesses of $10\mu m$, $20\mu m$ and $30\mu m$ at time $t = 0.5s$. The contours are at $p/p_0 = 0.02$, 0.05, 0.1, 0.25, 0.5, 0.75, 0.9, 0.97, and 0.99.

$10\mu m$ cathode radius with a $10\mu m$, $20\mu m$, and 30 μm electrolyte layer, which can also be compared to the corresponding $5\mu m$ electrolyte layer shown in Figure 6.2. For the spherical model which allows diffusion perpendicular to the membrane only, the diffusion pattern changes sharply between all four thicknesses. For example the $0.5p_0$ contour moves from well into the membrane for the $5\mu m$ case, to the inner surface of the membrane in the $20\mu m$ case, to well inside the electrolyte layer for the $30\mu m$ case. Since diffusion in the electrolyte is comparatively rapid, these changes are mirrored in the increase in the current. For the cylindrical model, there is a large change in the pattern between the $5\mu m$ and $10\mu m$ cases, but this is less dramatic from then on, as the increased volume of electrolyte can again replenish the area around the cathode by diffusion parallel to the membrane.

In modelling the changes in the current sensitivity with electrolyte layer thickness the spherical model does not give a true indication of either the nature or the magnitude of the variation, whilst the linear model is clearly totally inappropriate for a cathode of this size. In the remainder of this chapter, when describing the current variation with the other parameters of interest, we will therefore concentrate only on results obtained from the two–dimensional cylindrical model.

6.2 Variation of the Governing Parameters

At the end of Chapter 2 we listed the parameters whose effect on the operation of the Clark electrode we would like to investigate for a continuously applied potential. We have already seen in Figure 6.2 that by altering the cathode radius we can completely change the nature of the diffusion of oxygen to the cathode. Therefore, throughout this section where we consider the variation of the diffusion processes and the current obtained with time, electrolyte layer thickness and membrane characteristics, we do so for the three cathode radii $1\mu m$, $10\mu m$ and $100\mu m$, which we have already seen are predominantly governed by radial, both radial and linear, and linear diffusion respectively. The most commonly used cathode radii in practical measurements of blood oxygen concentration are of the order of $10\mu m$ (Hahn [37]), with larger cathodes more common in industrial applications. Radii of the order of $1\mu m$ are difficult to build, and, since we are measuring the bulk partial pressure using a surface effect, a very small cathode is particularly susceptible to small fluctuations in P_{O_2}. In general, we will therefore consider only the qualitative changes in the nature of the diffusion for very small cathodes. When presenting contour plots of the variation of the partial pressure, we again choose to do so at the time $t = 0.5s.$, at which we will show that the nature of the diffusion process is well established, and the effect of the variation of the particular parameter has become clear. Where we do not state otherwise, we protect the cathode by a PTFE membrane using the parameter values given in Table 5.3.

6.2.1 Variation of the diffusion with time

The build up of the partial pressure gradients over the first second for each

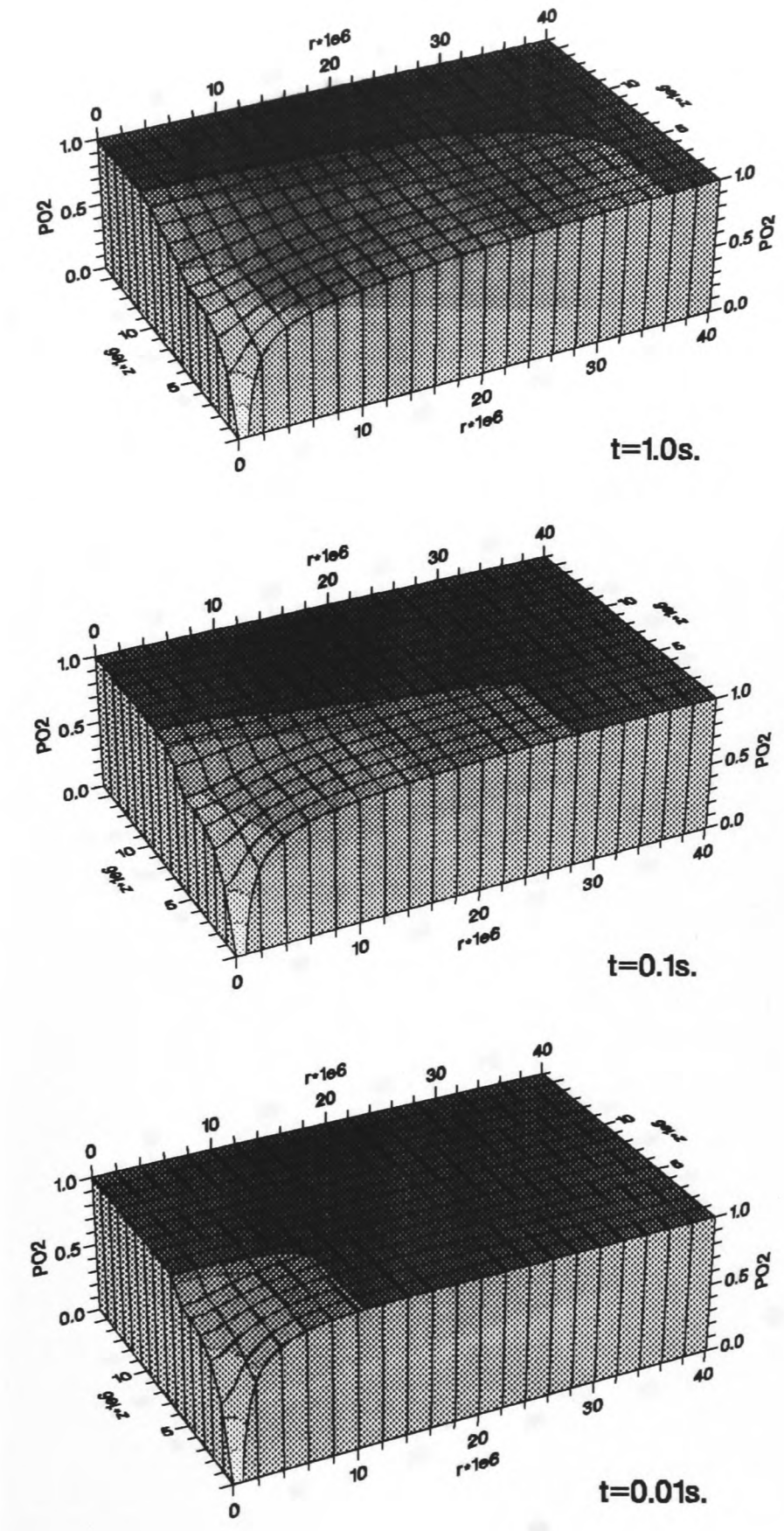


Fig. 6.4: Change in partial pressure gradient with increasing time for a 1 μm cathode radius protected by a 5μm PTFE membrane over a 5μm electrolyte layer.

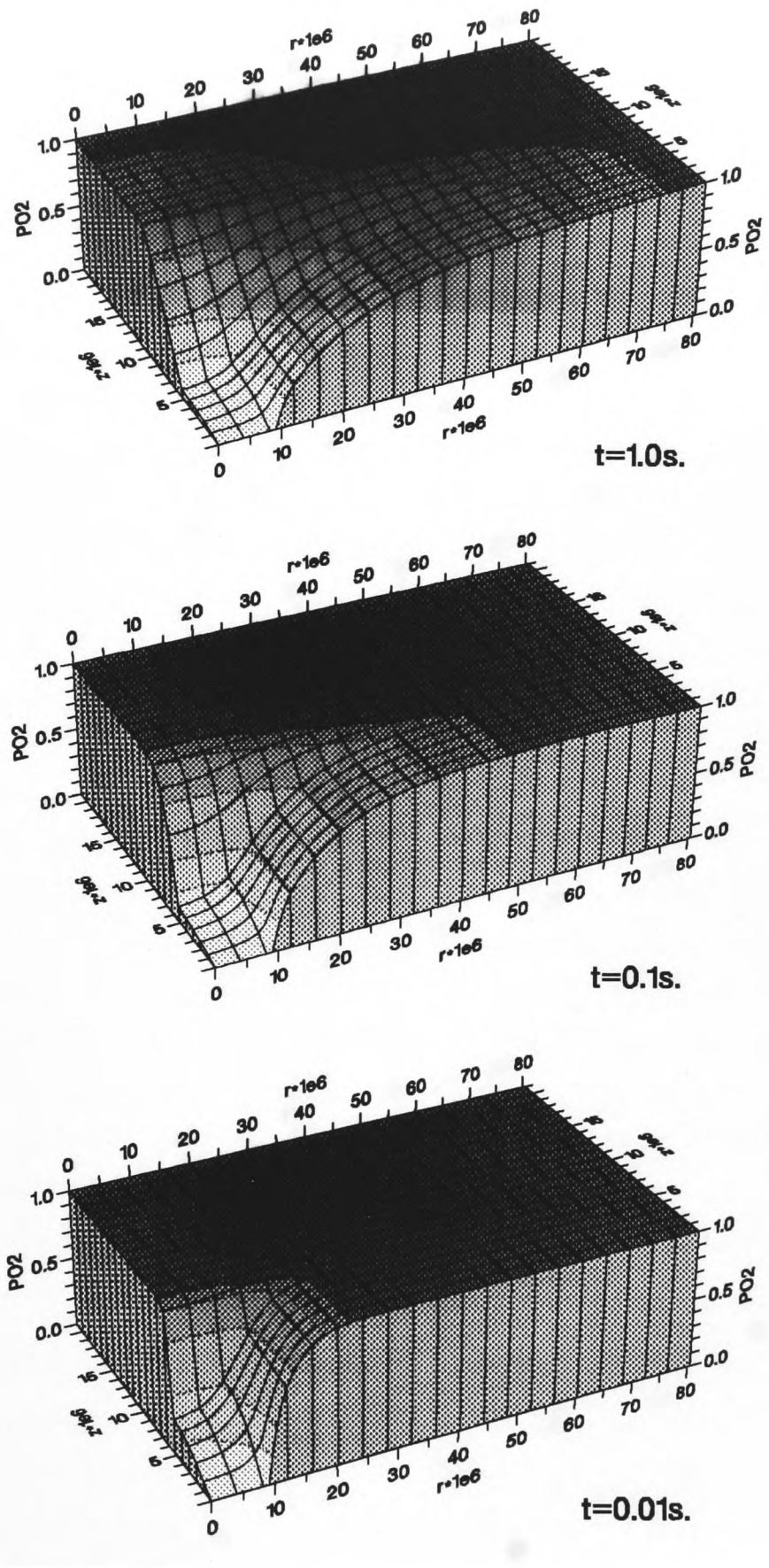


Fig. 6.5: Change in partial pressure gradient with increasing time for a 10 μm cathode radius protected by a 5μm PTFE membrane over a 5μm electrolyte layer.

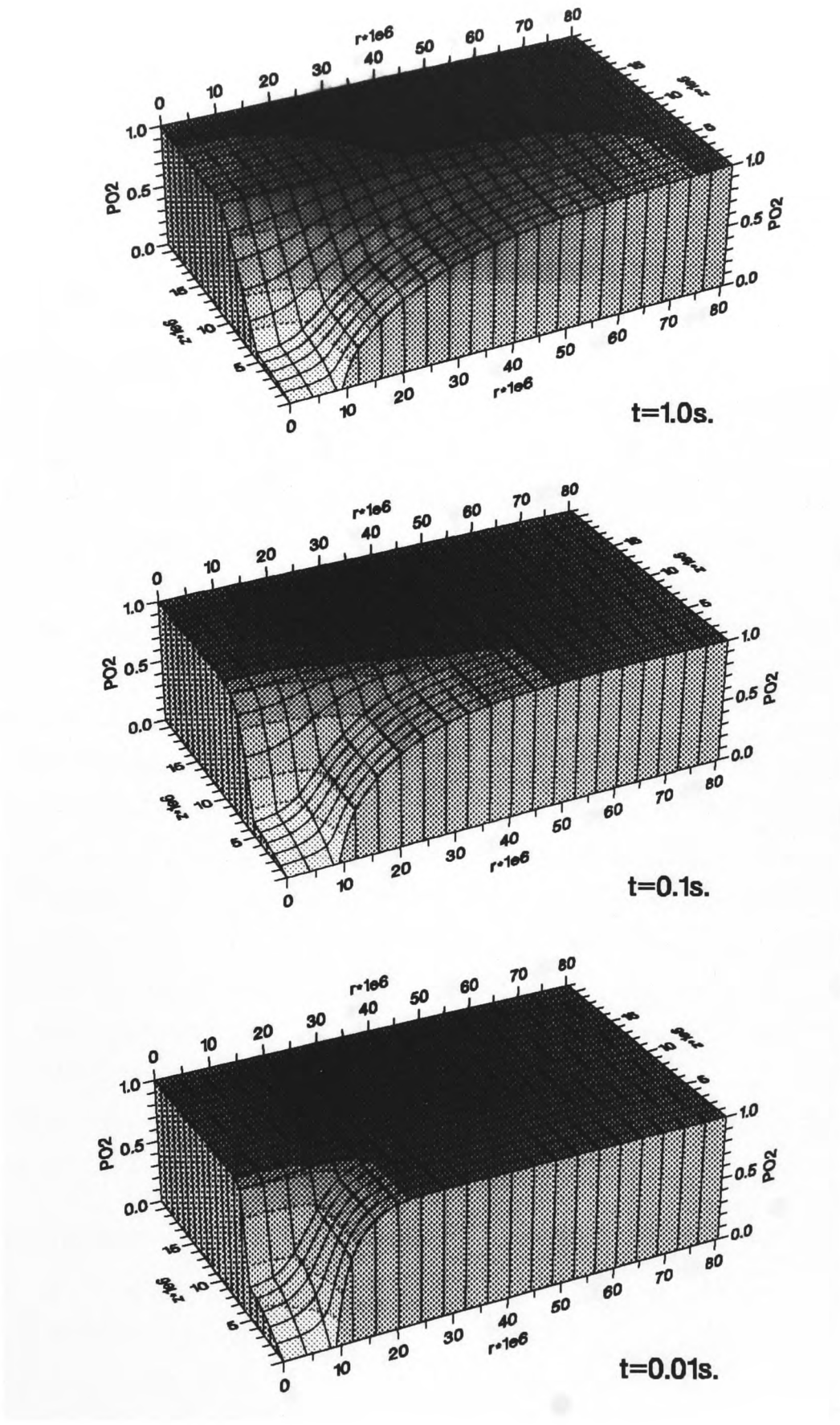


Fig. 6.6: Change in partial pressure gradient with increasing time for a 100 μm cathode radius protected by a 5μm PTFE membrane over a 5μm electrolyte layer.

of the three cathode radii is shown in Figures 6.4, 6.5 and 6.6. The very small $1\mu m$ radius cathode acts almost as a point sink in the corner of the region, only removing oxygen from the electrolyte to any degree in its immediate vicinity. The resulting concentration gradient is therefore very weak in both directions, so that the less diffusive membrane effectively acts as a barrier in the z–direction, with most of the diffusion taking place radially through the more diffusive electrolyte. The $10\mu m$ cathode radius of Figure 6.5 is sufficiently large to set up a strong concentration gradient in both directions. By $0.01s$. it has already largely depleted the electrolyte of oxygen in the z–direction out to the membrane and radially to about twice its radius. The gradient remains strong in both directions, so that at time $1.0s$. oxygen is reaching the cathode down a steep concentration gradient through the membrane immediately above the cathode, and along a smooth one eating out radially into the electrolyte. In Figure 6.6, the very large $100\mu m$ cathode radius sets up a very strong initial gradient in the z–direction and has removed virtually all of the oxygen from the electrolyte immediately above its surface by $0.01s$. From this time on, however, since diffusion through the membrane is so slow, radial diffusion through the electrolyte begins to build up, with a smooth gradient forming out to about twice the radius by $1.0s$. For both of the larger cathodes, the linear component of diffusion is sufficient to move through into the sample by $t = 1.0s$. In all three cases, the nature of the singularity at the cathode edge can be seen clearly.

For the larger cathodes, particularly the $100\mu m$ case, we can view the diffusion pattern as having several regions of both time and space which are predominantly one–dimensional in nature. Over the first few milliseconds, rapid linear diffusion parallel to the z–axis sets up a region of low oxygen partial pressure between the cathode and the inner surface of the membrane. From this time on, we have two regions where diffusion is approximately linear coupled together by diffusion in the membrane. The first is again parallel to the z–axis in the membrane, the other perpendicular to this from the cathode edge out radially parallel to the r–axis in the electrolyte layer. We will make use of this later in the chapter when considering operating the sensor in pulsed mode.

6.2.2 Time–transient behaviour of the current

Figure 6.7(a) shows the decay of the current for cathode radii varying from $1\mu m$ to $100\mu m$ radius cathodes over the first half second. As we have seen, a $1\mu m$ cathode radius is sufficiently small that diffusion is primarily limited to the electrolyte with very little depletion of the oxygen supply. The current decays very little from its initial value, and is little–affected by changes in membrane characteristics. The current–sensitivity therefore varies very little as membrane parameters are altered, so that we will consider only the qualitative variation in the diffusion patterns for the $1\mu m$ radius with membrane characteristics.

For the larger cathode radii, the current decays very rapidly initially as the

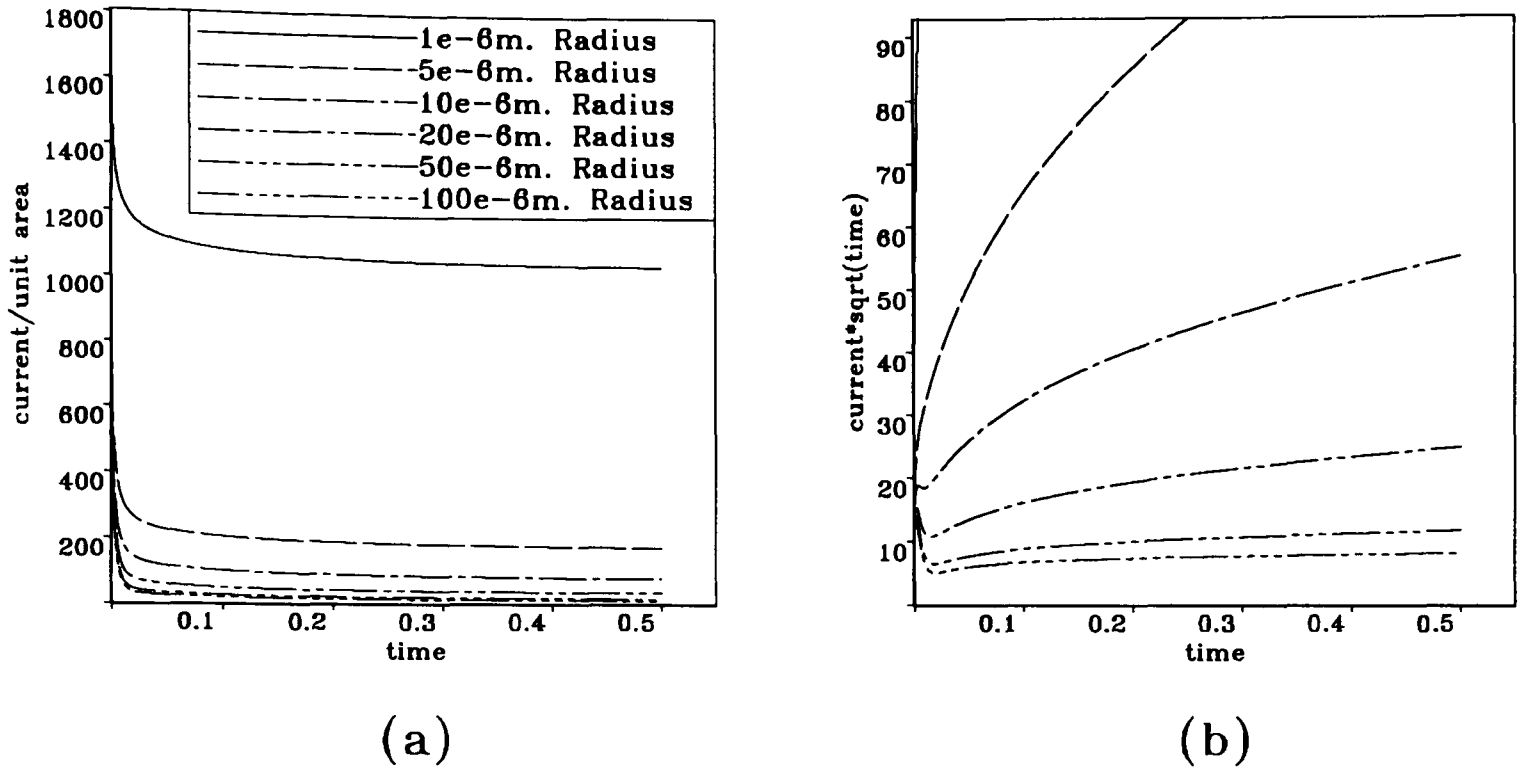


Fig. 6.7: Variation of (a) Current with time and (b) $it^{1/2}$ against time for various cathode radii, for a 5μm electrolyte layer and a 5μm PTFE membrane.

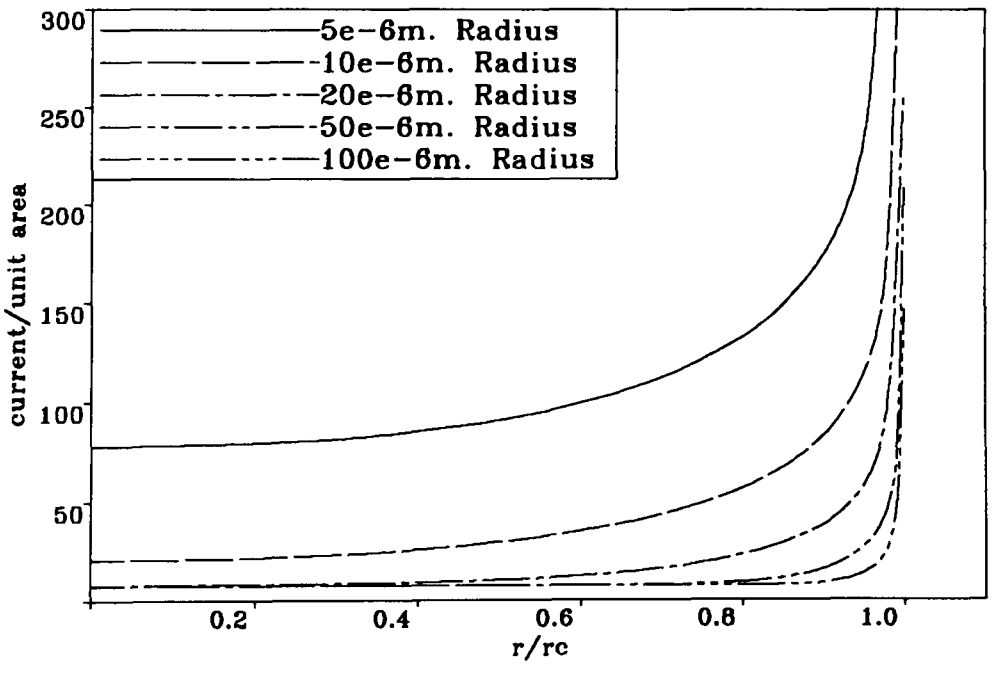


Fig. 6.8: Plot of the current per unit area against distance along the cathode for increasing cathode radii at time $t = 0.5s$.

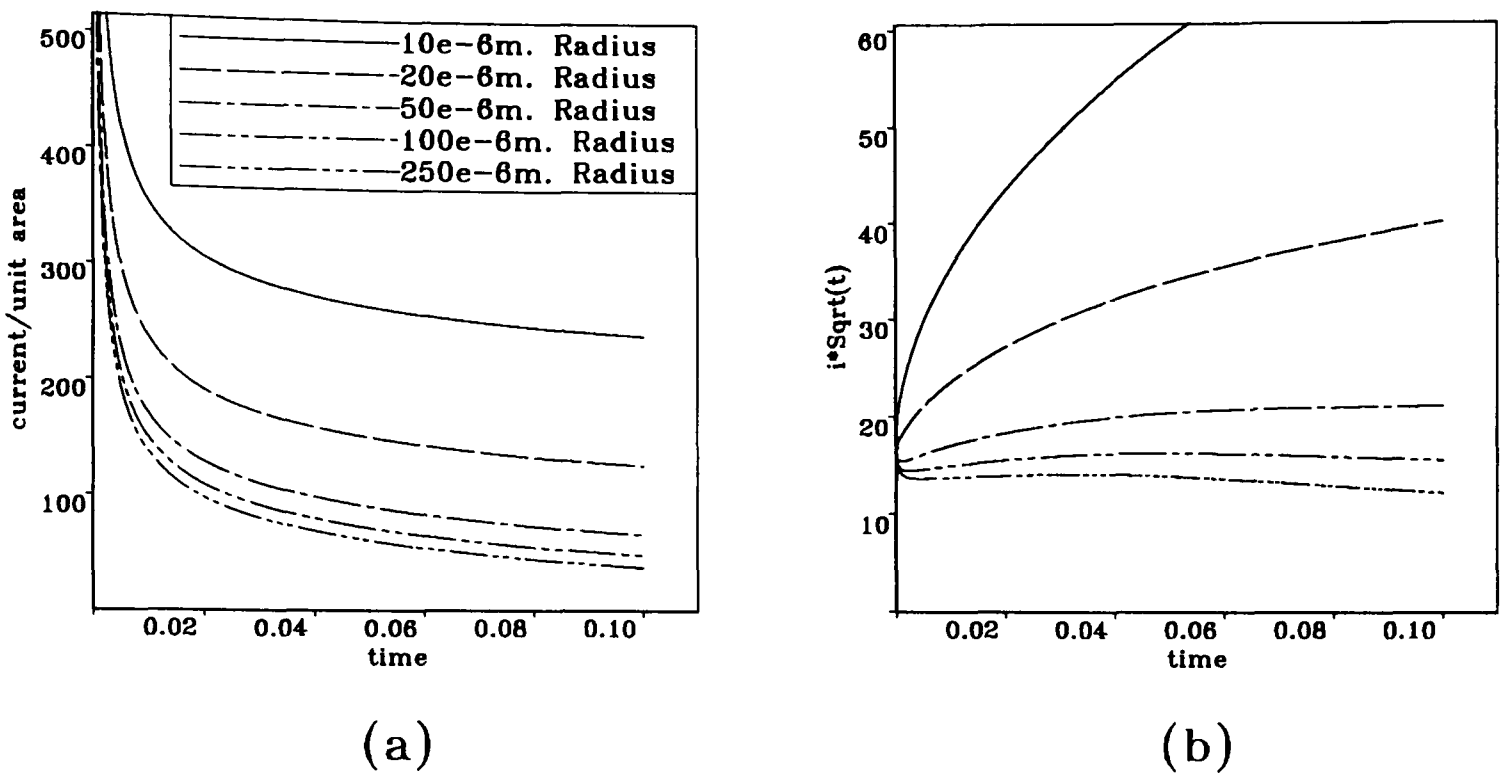


Fig. 6.9: Variation of (a) Current with time and (b) $it^{1/2}$ against time for various cathode radii, for a 20μm electrolyte layer and a 5μm PTFE membrane.

readily available oxygen store in the electrolyte immediately above the cathode is used up. The current per unit area obtained for the smaller cathodes decays to a higher value since the radial component of the diffusion is able to deliver oxygen to the whole cathode surface, as illustrated in Figure 6.8, which shows the current per unit area as a function of distance along the cathode (with all radii normalised to 1.0). It can be seen that for the $100\mu m$ cathode radius, radial diffusion only affects the oxygen flux to the outermost 10–20% (or 10–$20\mu m$) of the cathode radius. This corresponds well with the situation for the other radii shown, so that radial diffusion affects a proportion inversely with cathode radius until for the $10\mu m$ radius, the flux to the entire cathode is increased by radial diffusion. This emphasises the point made earlier about regions of one–dimensional diffusion with a cathode radius of about $10\mu m$ being a transition point. All smaller cathode radii are affected by two–dimensional diffusion over the whole cathode surface, whilst larger radii have a region of one–dimensional diffusion to the inner part of the cathode.

6.2.3 Cottrellian behaviour

Figure 6.7(b) shows the variation of $it^{1/2}$ (where i is the current) with time. Since the electrolyte layer is so thin ($z_e = 5\mu m$), the membrane begins to influence the current almost immediately. The value of $it^{1/2}$ falls sharply initially for all but the $1\mu m$ and $5\mu m$ cathode radii, as the readily available supply of oxygen in the electrolyte is rapidly depleted. From this time on, the diffusion has two

components in all cases, a radial one through the highly diffusive electrolyte, and a linear one through the depleted electrolyte immediately above the surface and the less diffusive membrane. If diffusion were purely linear, then we would observe a current proportional to $t^{\frac{1}{2}}$ whose magnitude would be determined by the diffusion coefficient of the membrane. However, in all cases in Figure 6.7(b) the radial component of diffusion in the electrolyte layer is sufficient to give an upward trend in $it^{1/2}$, although for the $100\mu m$ cathode the radial component of diffusion is almost negligible, and we observe a current which is almost proportional to $t^{-1/2}$ once the transport of oxygen is mainly limited to the membrane.

Figure 6.9(b) shows the same cases as Figure 6.7(b) but this time with $z_e = 20\mu m$, so that the membrane will not influence the current to any degree until $t \approx 0.05s$ (we omit the $1\mu m$ radius to obtain a more appropriate scale). Again none of the smaller cathodes shows Cottrellian behaviour over the timescale shown. For the larger cathode radii we have one diffusion process (linear) dominant, and within the accuracy of our calculations the current is roughly proportional to $t^{-1/2}$ for the $100\mu m$ and $250\mu m$ cathode radius over this period.

It is interesting to note the similarity between Figures 6.7(b) and 6.9(b) for the larger cathode radii and Figures 2 and 3 in Myland and Oldham [73] which are obtained from the analytic solution to the linear problem. Our results reinforce their observation that Cottrellian behaviour will only be observed for large cathode radii with a sufficiently thick electrolyte layer to mimic the behaviour of an unshielded cathode.

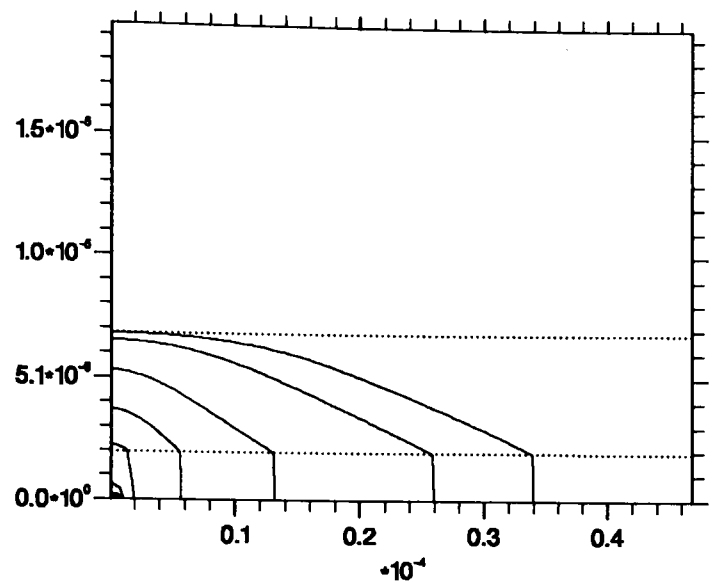
6.2.4 Variation with electrolyte layer thickness

We have already considered the variation with electrolyte layer thickness in some detail when comparing the one– and two–dimensional models. Here we consider in Figures 6.10 and 6.11 the variation for our three chosen cathode radii.

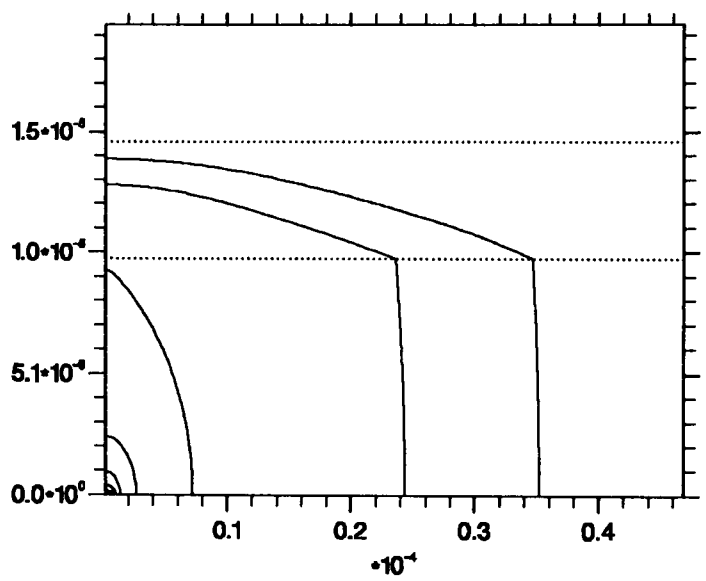
In Figure 6.10 we again see that increasing the thickness of the electrolyte layer allows oxygen to diffuse to the cathode from a greater volume of electrolyte, with the contours able to spread out further radially through the more diffusive electrolyte layer for all three cathode radii. This results in the higher values of the current shown in Figure 6.11. Whilst increasing the electrolyte thickness reduces the amount of sample depletion, it also increases the response time to changes in sample P_{O_2}.

6.2.5 Variation with membrane characteristics

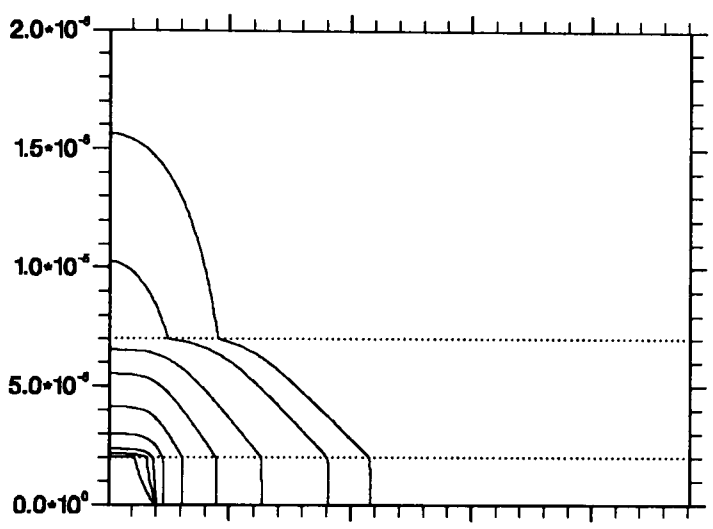
Figures 6.12 and 6.13 give a quantitative measure of the current variation with membrane thickness, diffusion coefficient and solubility for the $10\mu m$ and $100\mu m$ cathode radii at increasing times. Figure 6.12 shows the variation of the current with membrane thickness for a PTFE membrane. It can be seen that for a very thin membrane the diffusion front has moved through the membrane into the sample after a short time, and the more rapid diffusion of oxygen in the sample layer



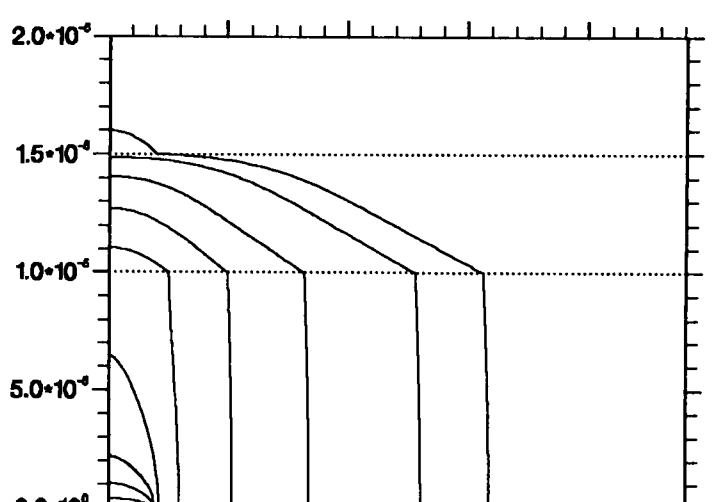
1e-6m radius, 2e-6m electrolyte layer



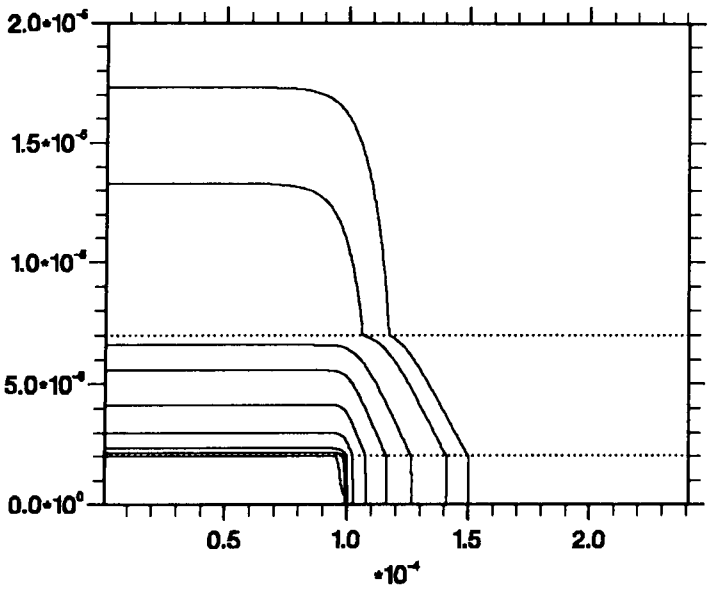
1e-6m radius, 10e-6m electrolyte layer



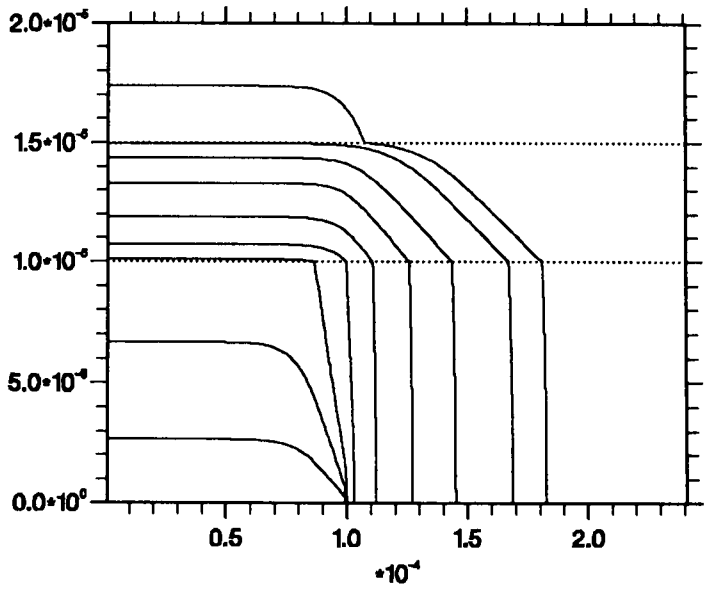
10e-6m radius, 2e-6m electrolyte layer



10e-6m radius, 10e-6m electrolyte layer



100e-6m radius, 2e-6m electrolyte layer



100e-6m radius, 10e-6m electrolyte layer

Fig. 6.10: Contour plots of the partial pressure using a $2\mu m$ and $10\mu m$ electrolyte layer for a $1\mu m$, $10\mu m$, and $100\mu m$ cathode radius at $t = 0.5s$ calculated using the cylindrical model. The contours are at $p/p_0 = 0.02$, 0.05, 0.1, 0.25, 0.5, 0.75, 0.9, 0.97, and 0.99.

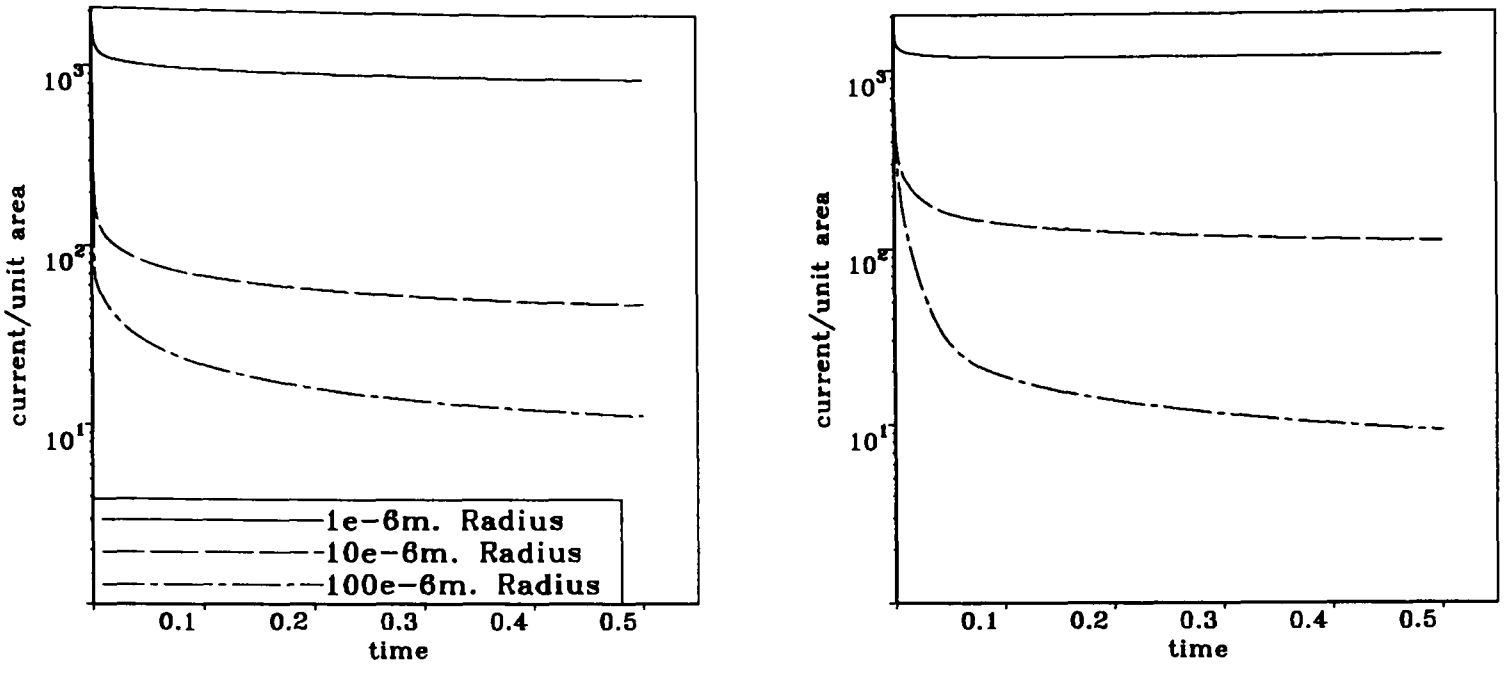


Fig. 6.11: Variation of Current with time (a) With a $2\mu m$ electrolyte layer and (b) With a 10μm electrolyte layer both protected by a 5μm PTFE membrane.

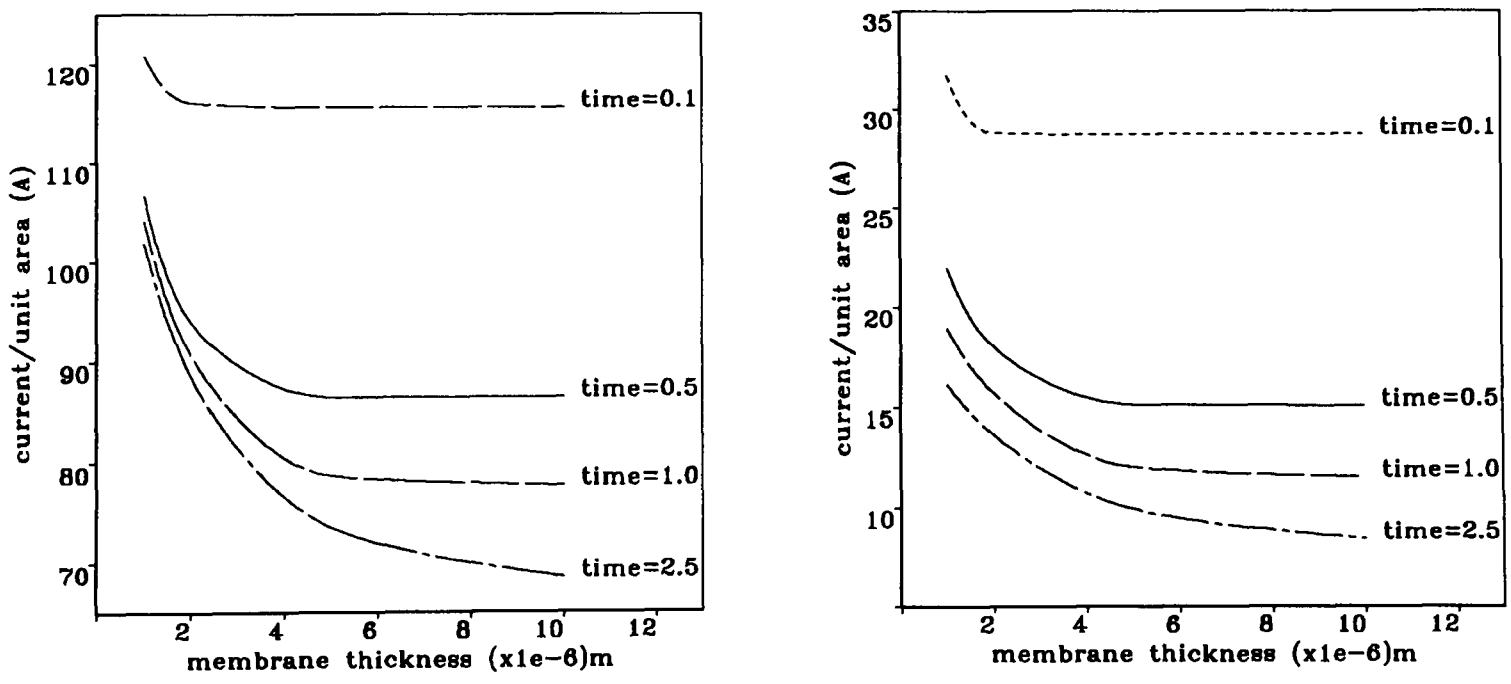


Fig. 6.12: Variation of current per unit area with membrane thickness at various times.

results in an increased current. This is more marked for the smaller cathode since the volume of sample depleted relative to cathode area is inversely proportional to cathode area as can be seen in the diffusion patterns shown in Figure 6.14.

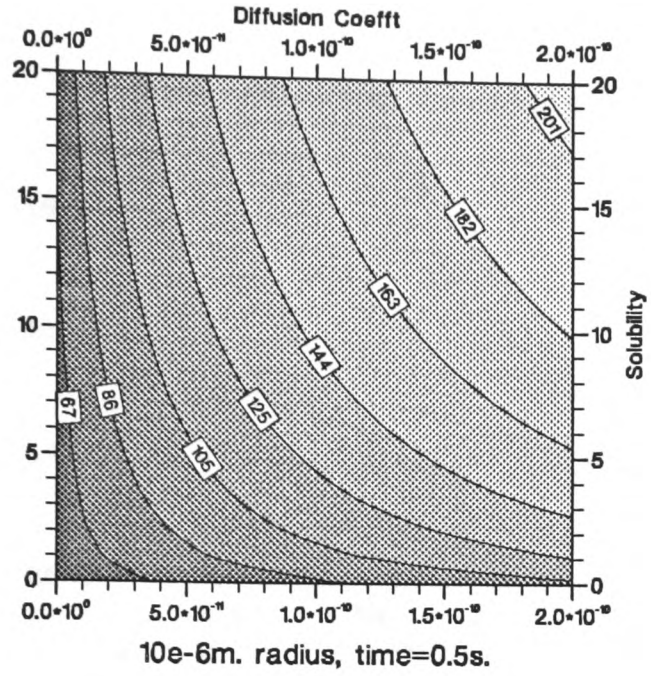
Figure 6.13 shows the current variation with both diffusion coefficient and solubility, the contour lines being those of equal current per unit area. For shorter times the plots are skewed in favour of solubility in that we obtain large current values for a comparatively high diffusivity and low solubility. A high value of diffusivity allows early breakthrough into the sample, and more rapid transport through the membrane. This is more obvious for the larger electrode for which most of the current results from linear diffusion through the membrane. At longer times, this effect is reduced by depletion of the sample, so that the oxygen reaching the cathode through the membrane has diffused from a large distance into the sample layer.

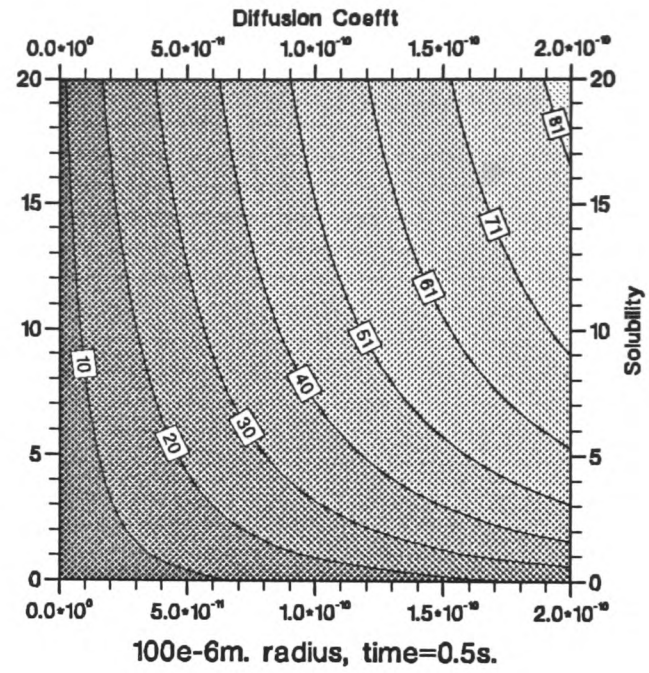
Since solubility is a measure of the storage of oxygen in the membrane, a very high value of solubility will also raise the initial current even with a small diffusion coefficient, since there will be a large store of oxygen in the membrane close to the cathode. Again, this effect is reduced with time as oxygen is transported only slowly from deeper into the membrane.

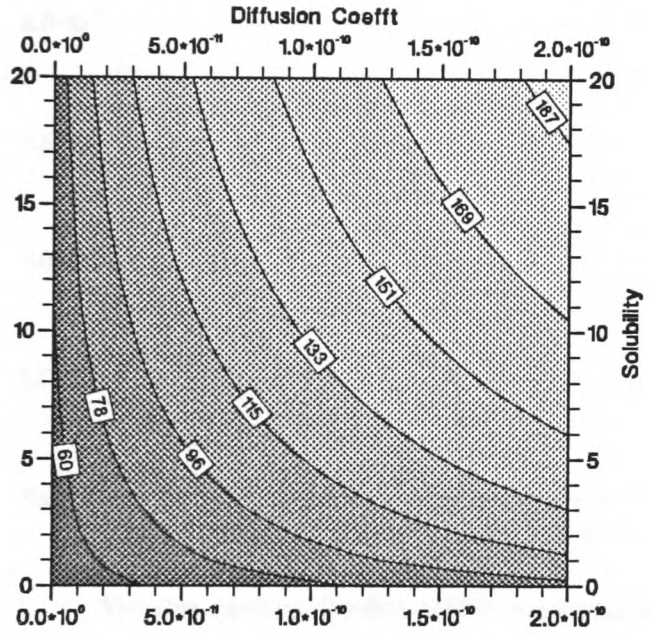
At longer times as the oxygen supply close to the cathode is depleted, the contour plots tend to symmetric rectangular hyperbole i.e. we obtain the same current provided that the product of solubility and diffusion coefficient, the permeability, is constant. We can therefore interpret the permeability as a measure of the rate of mass transport of oxygen through the membrane.

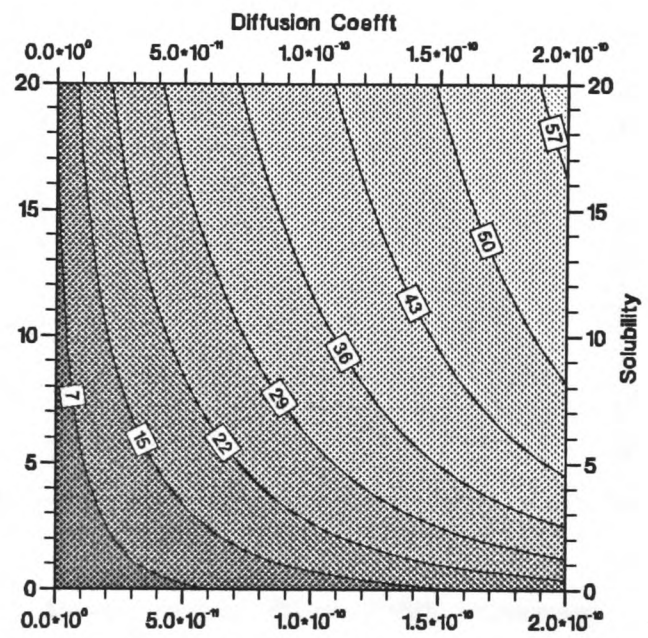
Each of these effects can be seen in the contour plots of the diffusion patterns shown in Figures 6.15 and 6.16. In Figure 6.15 we keep the solubility constant whilst using a diffusion coefficient four times larger, and one order of magnitude smaller (which roughly corresponds to a polypropylene membrane) than that of PTFE. The permeability is therefore also changed correspondingly. Lowering the diffusion coefficient lowers the oxygen supply through the membrane, the P_{O_2} immediately above the cathode becomes very low, and the strength of the radial diffusion gradient in the electrolyte layer is increased. As a result the diffusion front is forced out further in the radial direction.

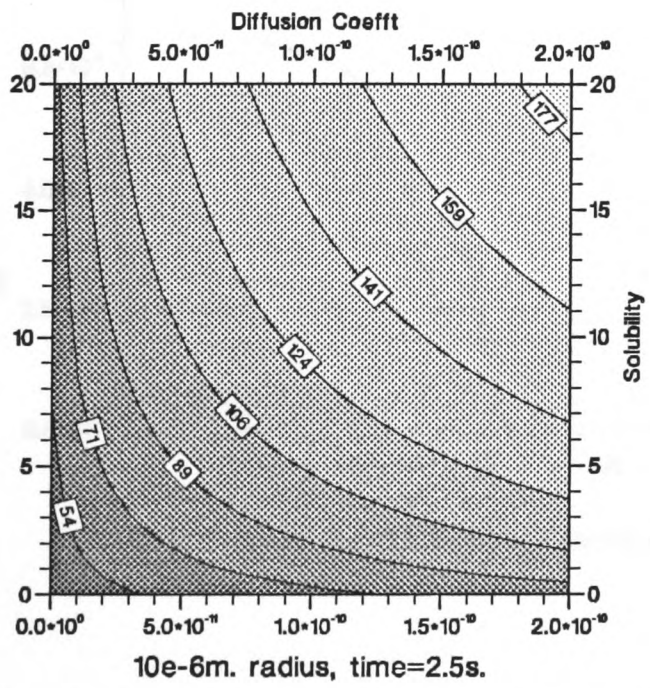
Figure 6.16 uses the same diffusion coefficient for all cases (that of PTFE membrane) with comparatively high and low values of membrane solubility. With high solubility there is a large store of oxygen in the membrane which has not been drained by $t = 0.5s$ so that there is less depletion of oxygen in the electrolyte, and the concentration gradient in the electrolyte layer is steeper in both directions, with oxygen in the vicinity of the cathode being replenished from the large store in the membrane. Again with low solubility, there is little oxygen available from the membrane so that the region of electrolyte immediately above the cathode is rapidly depleted setting up a stronger radial concentration gradient and diffusion











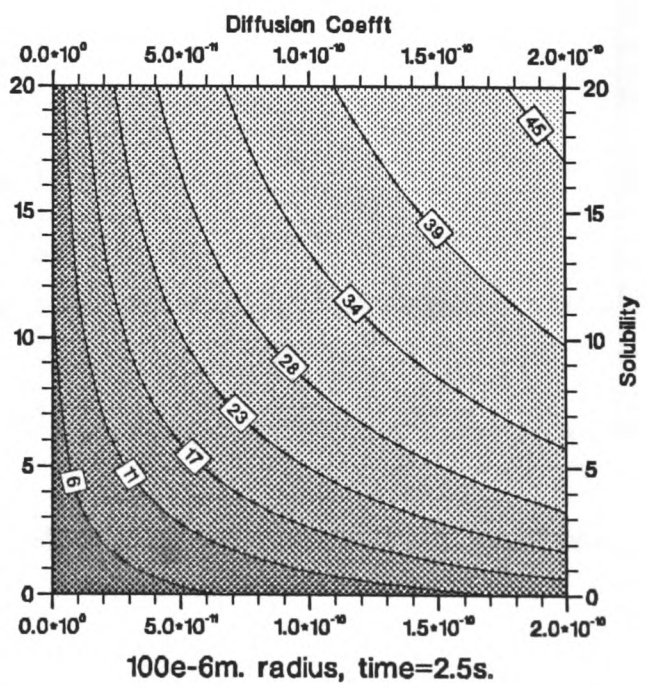
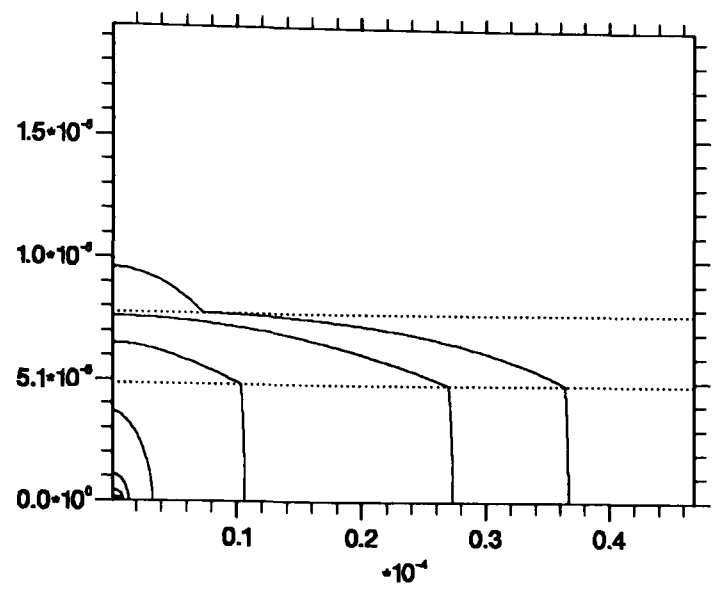
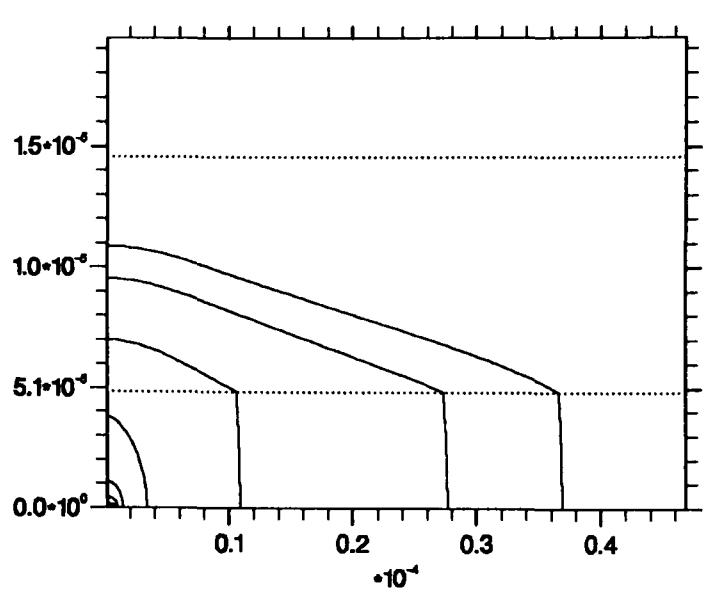


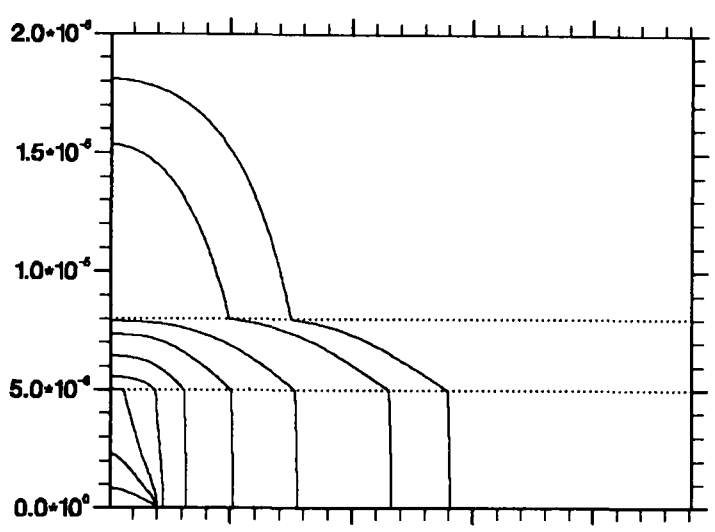
Fig. 6.13: Contour plots showing variation of current per unit area with membrane diffusion coefficient and solubility at increasing times. All cases use a 10 μm PTFE membrane and a 5 μm electrolyte layer.



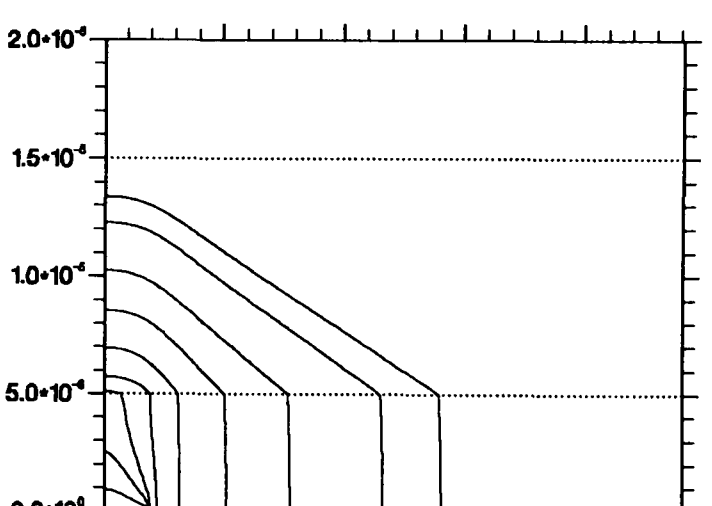
1e-6m radius, 3e-6m PTFE membrane



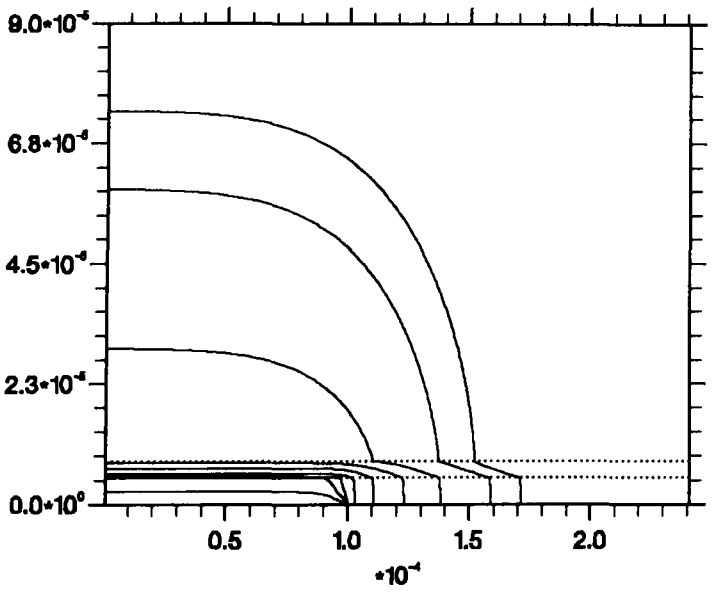
1e-6m radius, 10e-6m PTFE membrane



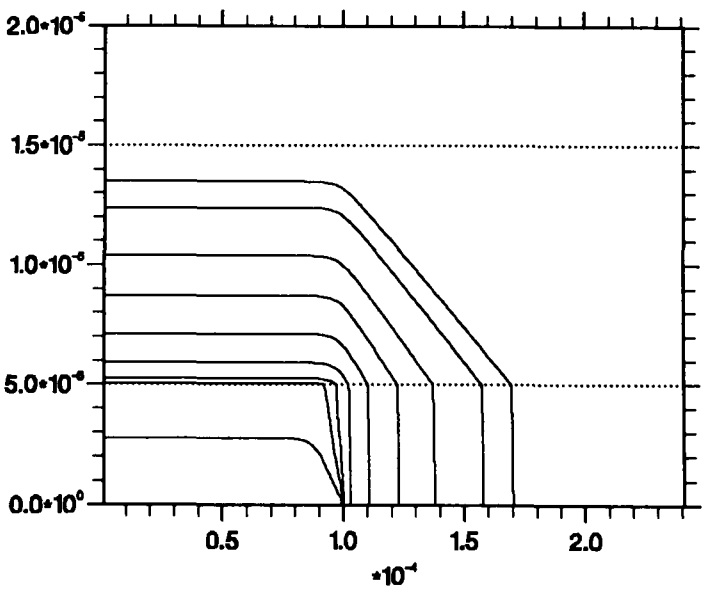
10e-6m radius, 3e-6m PTFE membrane



10e-6m radius, 10e-6m PTFE membrane

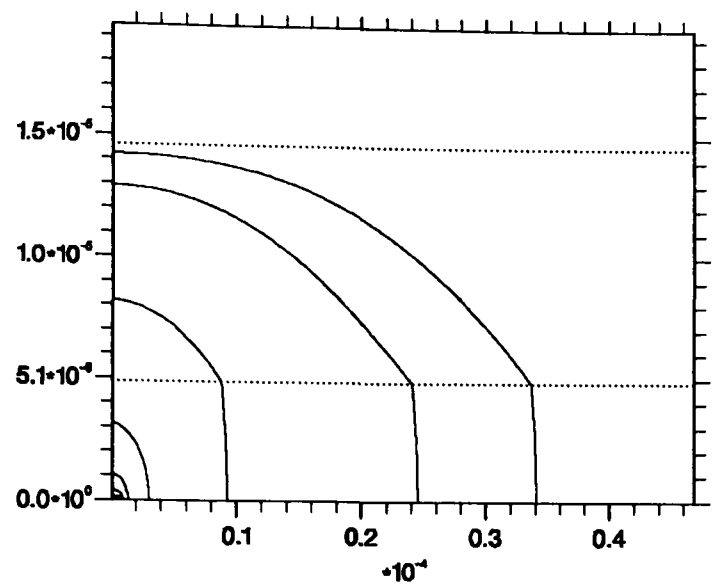


100e-6m radius, 3e-6m PTFE membrane

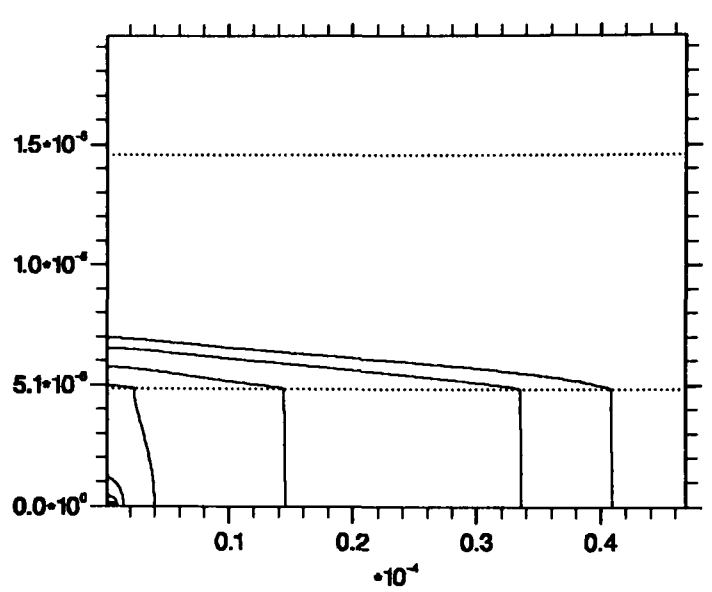


100e-6m radius, 10e-6m PTFE membrane

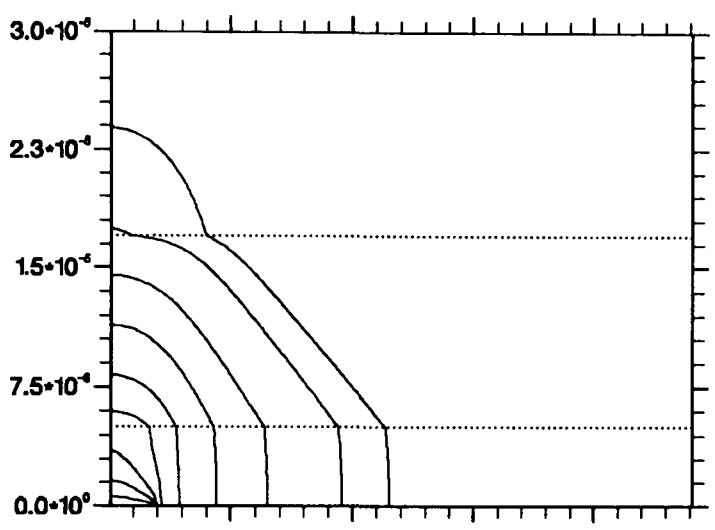
Fig. 6.14: Contour plots of the partial pressure using a $3\mu m$ and $10\mu m$ PTFE membrane for a $1\mu m$, $10\mu m$ and $100\mu m$ cathode radius at $t = 0.5s$ calculated using the cylindrical model. The contours are at $p/p_0 = 0.02$, 0.05, 0.1, 0.25, 0.5, 0.75, 0.9, 0.97, and 0.99.



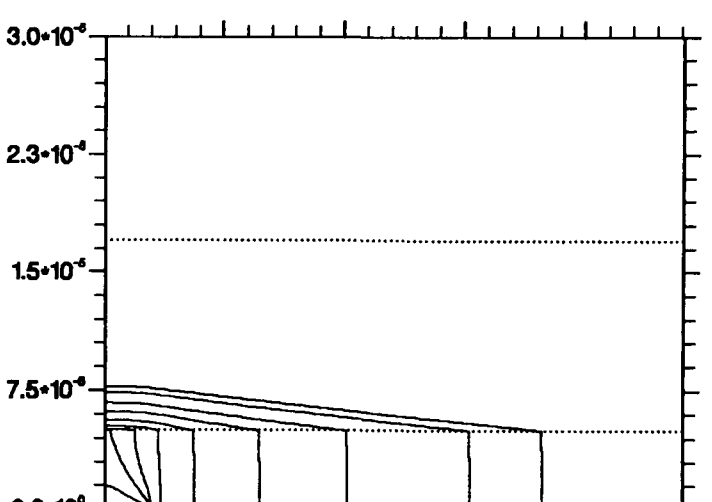
1e-6m radius, high diffusivity



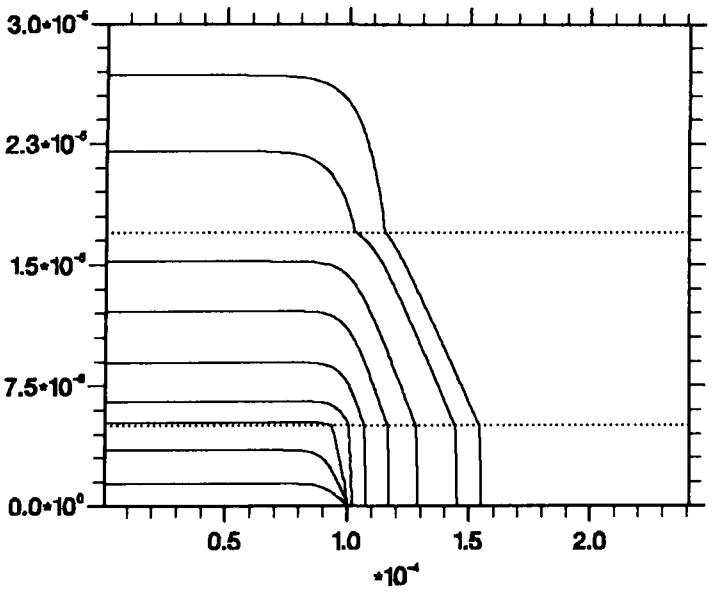
1e-6m radius, low diffusivity



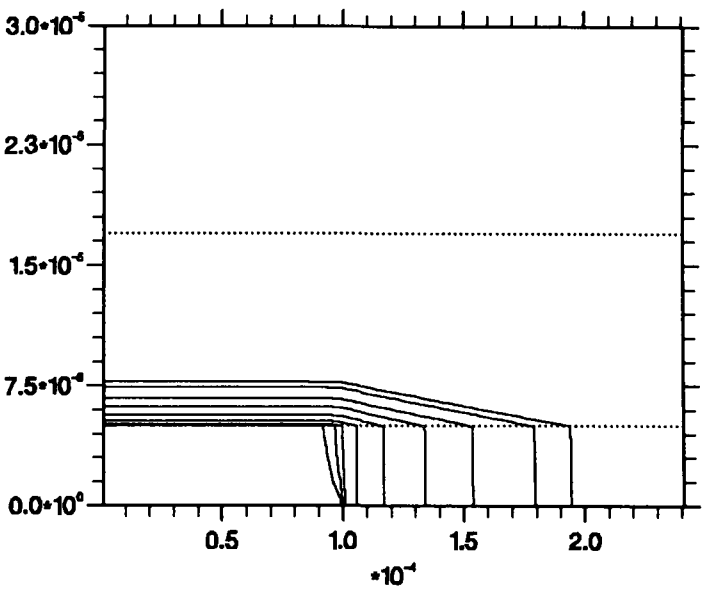
10e-6m radius, high diffusivity



10e-6m radius, low diffusivity



100e-6m radius, high diffusivity



100e-6m radius, low diffusivity

Fig. 6.15: Contour plots of the partial pressure for a $1\mu m$, $10\mu m$ and $100\mu m$ cathode radius protected by membranes with diffusion coefficients $D_m = 4.56 \times 10^{-11} m^2 s^{-1}$ and $1.14 \times 10^{-12} m^2 s^{-1}$ respectively at $t = 0.5s$. The contours are at $p/p_0 = 0.02$, 0.05, 0.1, 0.25, 0.5, 0.75, 0.9, 0.97, and 0.99.

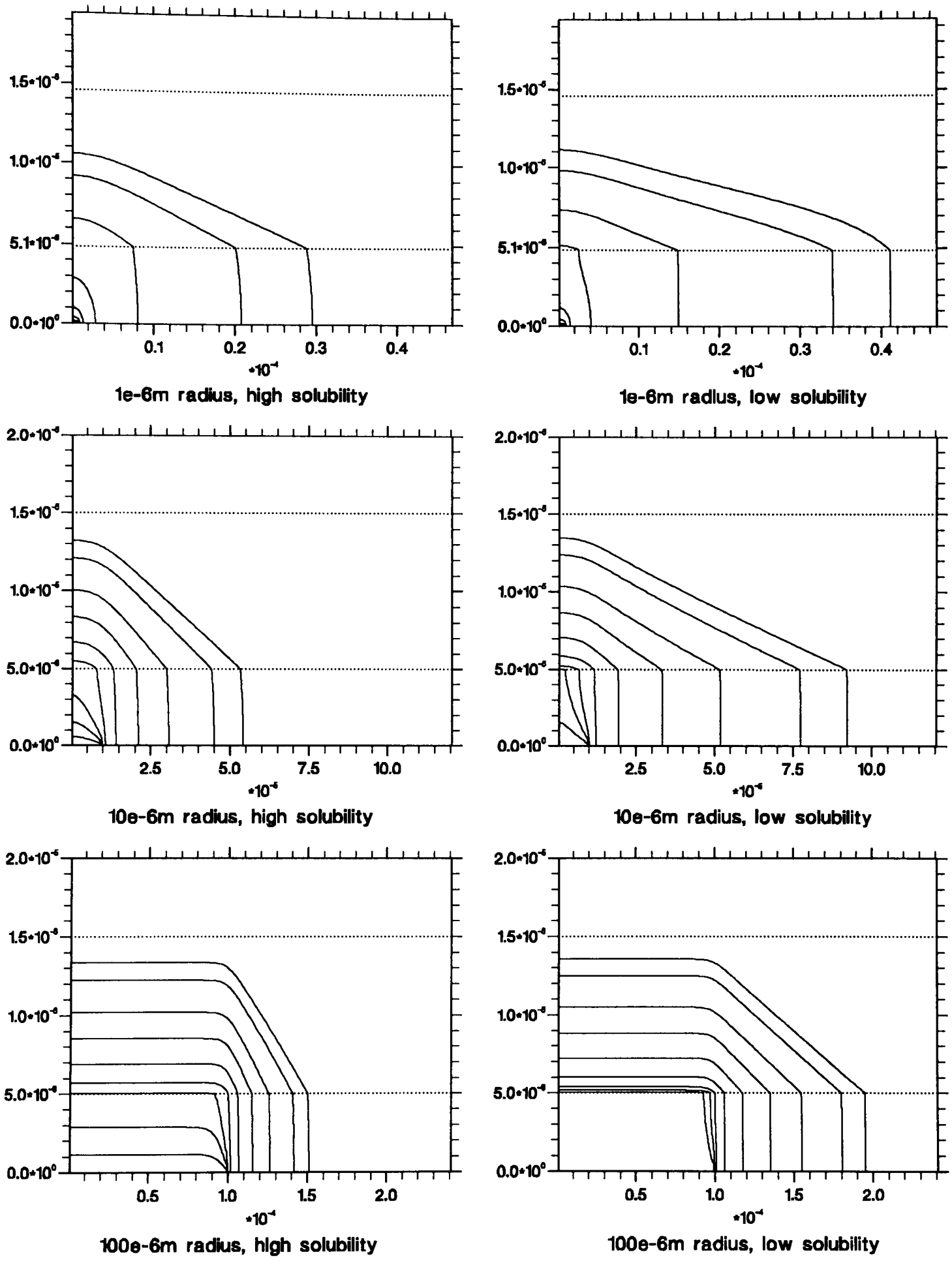


Fig. 6.16: Contour plots of the partial pressure for a $1\mu m$, $10\mu m$ and $100\mu m$ cathode radius protected by membranes with solubility coefficients $\alpha_m = 2.0$ and 20.0 respectively at $t = 0.5s$. The contours are at $p/p_0 = 0.02$, 0.05, 0.1, 0.25, 0.5, 0.75, 0.9, 0.97, and 0.99.

spreads out further into the electrolyte.

In all cases, the nett effect of reducing solubility or diffusivity is to deplete the available supply of oxygen in the vicinity of the cathode more rapidly. Oxygen must then be drawn in to the cathode from a greater distance and the current decays more quickly.

6.2.6 Summary

We have now used the cylindrical model to investigate the variation of both the current sensitivity and the diffusion patterns for a continuously applied potential with each of the parameters of interest. In all cases it has given physically sensible results. We have seen that for cathodes of practical size ($\geq 10\mu m$ radius), a strong linear diffusion gradient through the membrane is set up and eventual sample depletion is inevitable. However, we have gained sufficient information on the time–dependent variation of the diffusion processes to help to place constraints on the electrode geometry most likely to give optimal results under pulsed operation.

6.3 Pulsed Operation

The three major difficulties of using electrochemical sensors to measure oxygen in a biological environment have been summarised by Zick [98] as: 1) instability; 2) flow sensitivity; and 3) inadequate response time. The first of these difficulties is largely overcome by the design of the Clark electrode and, as was described in Chapter 2, is its major advantage over competing oxygen sensors. An ideal *in vivo* oxygen sensor will be sensitive only to changes in P_{O_2} in the artery, and not to changes in blood flow. If we simply switch on our oxygen sensor, we have seen that after a time the sample becomes depleted in a small region immediately above the cathode. The response of the system is then highly dependent on whether there is motion within the sample to produce partial or total replenishment of the P_{O_2} to this small region. We can overcome this problem of flow sensitivity to some extent by choosing a very thick, or a very impermeable protective membrane. However, this would exacerbate the third problem of inadequate response time of the sensor to any changes in P_{O_2} in the sample.

In practical *in vivo* measurements, therefore, it would be advantageous if Clark electrodes were used in the pulsed mode of operation which has been shown to overcome these difficulties in the laboratory. The technique involves setting up a pulsing regime whereby a sufficient potential is applied to the cathode to reduce oxygen for a fixed period, say t_{on}. The potential is then either removed, or altered so that oxygen is no longer reduced, for a period t_{off}, and the system relaxes back towards its initial state. By fixing a suitable ratio of t_{on} to t_{off}, a *pseudo-steady-state* can be reached with the measured current at the end of the "on" pulse, say, becoming constant. From equation (2.3), the oxygen concentration is then directly proportional to the current at this time, and the sensor can be calibrated and used to measure oxygen concentration. To be compatible with our requirement of rapid

response time, our chosen pulsing regime should also achieve this pseudo–steady–state quickly. The advantage of this technique is that by choosing a sufficiently short on–pulse, we can ensure that the resulting diffusion processes are primarily restricted to the electrolyte and membrane layers, even for a very thin membrane. The measured current at the end of each pulse should then be virtually unaffected by motion in the sample, but still give a rapid response to any changes in P_{O_2} in the sample.

The diffusion mechanism by which any change in P_{O_2} in the sample is then transmitted to the electrolyte takes place over a wide area of membrane and is effectively decoupled from the diffusion process at the cathode. The response time of the electrode should therefore be determined only by the transport time through the membrane, which is a function of membrane diffusivity and thickness.

6.3.1 Previous experimental work

The advantages of using the Clark electrode in pulsed mode, usually referred to as *pulse amperometry*, rather than waiting for steady–state to be reached, were soon realised. A series of early papers reviewing the possible "applied voltage waveforms" were published by Evans and Naylor [20, 19, 75, 74] in 1960. They again built their own sensor, and used a pulsing regime of "applying 0.7 or 0.8v between the anode and cathode for a period of about 0.5 seconds every 17 seconds". Since then there have been many papers published espousing various theories for the most appropriate geometry and pulsing regime [38, 50, 54, 57, 68, 82, 83, 87, 96, 97, 99], which together cover most of the possible range of parameter values. Amongst the more influential of these (in that they are more widely quoted) are the works of Lilley et al [57], Mancy together with several co-workers [68, 83, 97], Hahn et al [38], Zick [98, 99], and more recently Short and Shell [87].

6.3.2 The pulsing regime

The most appropriate pulsing regime has been the subject of much debate, both amongst theoreticians and experimentalists. Suggested times for the on–pulse width have varied from a few milliseconds [98] to several seconds [57], and for the off–pulse width from tens of milliseconds up to several minutes. No general consensus has emerged, although most authors suggest that a very short on–pulse width (of the order of a few milliseconds) will be affected by a capacitative current due to charging at the cathode/electrolyte interface (Hitchman [42]).

6.3.3 Previous theoretical work

Many of the above authors have made simple attempts at a theoretical analysis of pulsed amperometry, largely based on the one–dimensional theory of Mancy *et al* for a continuously applied potential, using their work for diffusion limited to the electrolyte layer to try to determine conditions for membrane–independent current measurements. As described earlier, the disadvantage of the one-dimensional analysis is that it allows no lateral means of replenishment parallel to the membrane in

the electrolyte layer. This is of primary importance when considering pulse amperometry, since it implies that whatever pulsing regime is chosen, sample depletion is inevitable since oxygen only reaches the cathode in the direction through the membrane. Once the store of oxygen within the membrane and electrolyte has been depleted after a few pulses, oxygen must be taken from the sample. As a result, some quite complicated one–dimensional models, which also incorporate the required fluid flow within the artery, have been proposed [22, 84].

Both the theoretical and practical use of pulse amperometry for the measurement of oxygen have been comprehensively reviewed by Hahn [36, 37]. He points out in his later paper that whilst the advantages outlined above make pulse amperometry very attractive, the failure of the one–dimensional theory to predict practical behaviour, together with the lack of consensus on the optimal pulsing regime and the effects of capacitative charging, have prevented the development of the technique for use with commercial *in vivo* sensors.

6.3.4 Two-dimensional modelling

Since the parameter space which governs the operation of the Clark electrode is so wide, we will first make our task easier by using the knowledge we gained earlier in the chapter to suggest the electrode geometry and pulsing regime most likely to satisfy our requirements of fast response and rapid attainment of the pseudo–steady–state, whilst also minimising sample depletion. Our first choice is to consider only the commonly used PTFE membrane, and since the response time to a step change in sample P_{O_2} is a function of transport time through the membrane only, we will use the minimum membrane thickness of $5\mu m$ (due to practical considerations of strength), and the $5\mu m$ electrolyte layer which is in the middle of our range. Optimal conditions for any other electrode geometry can easily be determined in similar fashion.

We saw in Figures 6.4, 6.5, and 6.6 the extent of the depletion within the electrolyte with increasing time. We have already described the difficulties associated with very small cathodes of the order of $1\mu m$, so that even though such a small cathode would clearly fulfil our requirement of low sample depletion, we are interested in improving *practical* performance and therefore will not consider further radii of this order.

The two–dimensional model allows replenishment of oxygen to the region of the cathode by rapid radial diffusion in the electrolyte layer, and assuming the volume of electrolyte is sufficient, it should be possible to choose a pulsing regime which ensures that no significant depletion of the sample need take place. To limit the possible choices of pulsing regime we make use of the one–dimensional solution to the unshielded problem of Cottrell which we gave in equation (3.1) as equivalent to

$$p(t) = p_0 \mathrm{erf}\left(\frac{z}{2\sqrt{Dt}}\right) \tag{6.1}$$

with partial pressure replacing concentration since they are directly proportional in a given layer. This will allow us to estimate the order of magnitude of the times taken for the one–dimensional components of the overall diffusion process to take place. To estimate the appropriate on–pulse width, we treat the initial diffusion within the electrolyte as purely linear. We can then use the values of the error function from tables to calculate that the P_{O_2} in the electrolyte at a distance of $5\mu m$ above an unshielded cathode of infinite radius would fall by 1% after $1ms$, 50% after $10ms$ and 90% after $250ms$. If a PTFE membrane were in direct contact with the cathode, the times would be much longer, with the P_{O_2} at a distance of $5\mu m$ falling by 1% only after about $170ms$. In practice for a finite cathode, the time for depletion in the membrane will be longer than this since the P_{O_2} in the electrolyte will never actually reach zero, and oxygen in the electrolyte is replenished by lateral diffusion. We therefore choose to try a pulse width of the order of $100ms$ which should be short enough to avoid substantial depletion in the membrane (and therefore the sample), and long enough to overcome practical difficulties with measurement.

During an off–pulse, it is more difficult to estimate the combined effects of replenishment to the region of the cathode from both the electrolyte and the membrane. We therefore investigate the process by applying a $100ms$ pulse to the cathode and then allowing the system to relax. Figure 6.17 shows the surface plot of oxygen partial pressure for the $100\mu m$ cathode radius at the end of such a pulse, then $500ms$ and $1000ms$ later with the potential removed. Even during this very short pulse, the large surface area of the cathode results in the reduction of so much oxygen from the electrolyte that substantial depletion of the sample takes place before the system relaxes back to its initial state. We experienced the same difficulty even with an on–pulse width as short as $10ms$. We could solve the problem by using a thicker or less permeable membrane, but this would increase the response time to changes in the sample layer. Alternatively, we could leave a very long time between on–pulses to allow the system to return to its initial state, but we would then be unable to take measurements during this period which would again result in a long response time.

Figure 6.18 shows similar surface plots for the $10\mu m$ cathode radius after a $100ms$ on–pulse, and then $100ms$ and $500ms$ later with the potential removed. The very rapid diffusion in the electrolyte replenishes virtually all the depleted oxygen by the later time (just above the cathode the P_{O_2} is $0.97p_0$, with the lowest level in the membrane at $0.947p_0$), whilst the oxygen P_{O_2} in the sample is nowhere depleted by more than 1%. Choosing an on–pulse width of $100ms$ and an off–pulse of $500ms$ for this electrode geometry should therefore satisfy all three of our requirements: rapid achievement of pseudo–steady–state; minimal sample depletion; and rapid response time to a change of partial pressure in the sample.

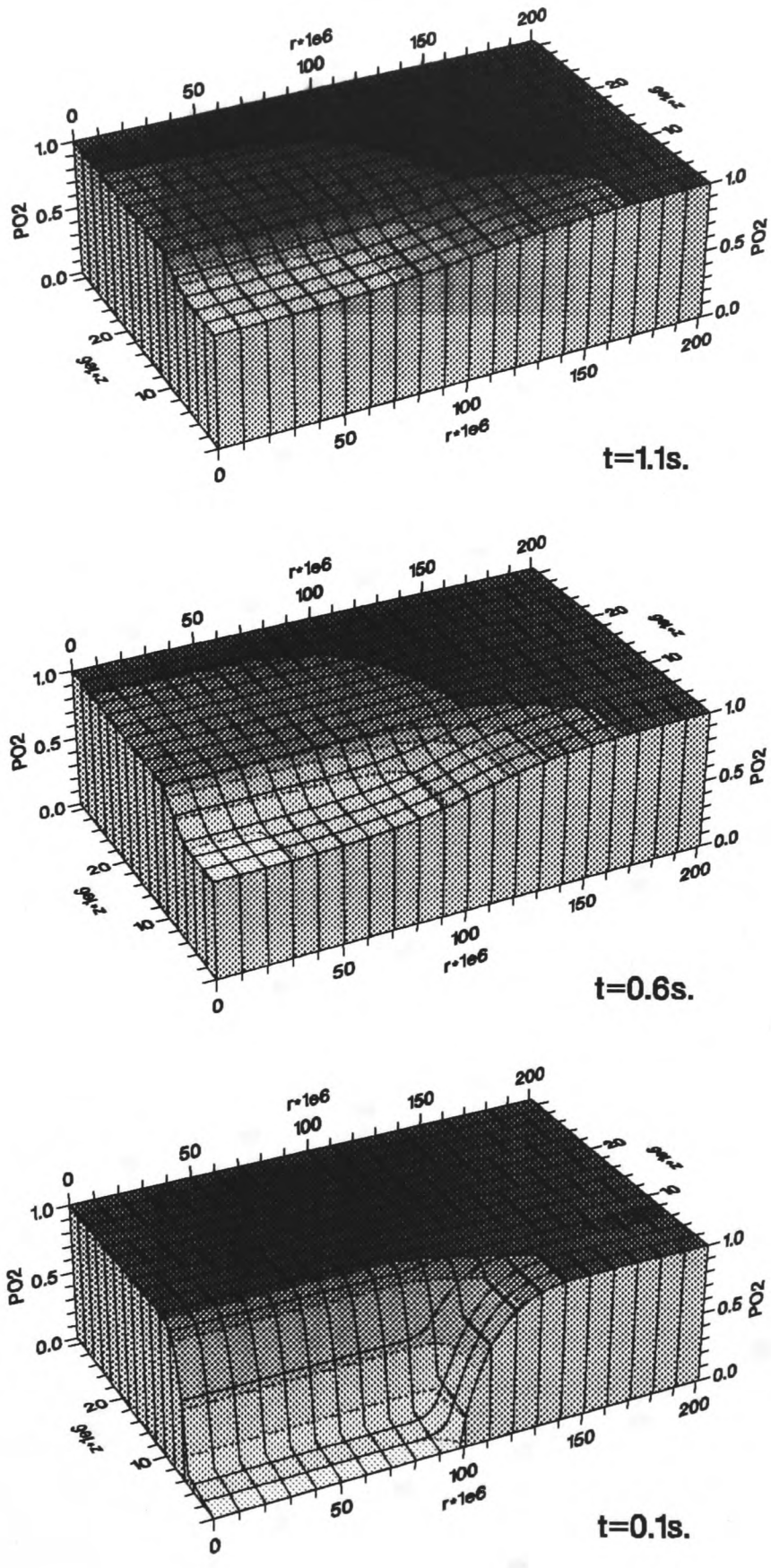


Fig. 6.17: Partial pressure surface at the end of a $100ms$ pulse, and $500ms$ and $1000ms$ later with the potential removed, for a $100\mu m$ cathode radius protected by a $5\mu m$ PTFE membrane.

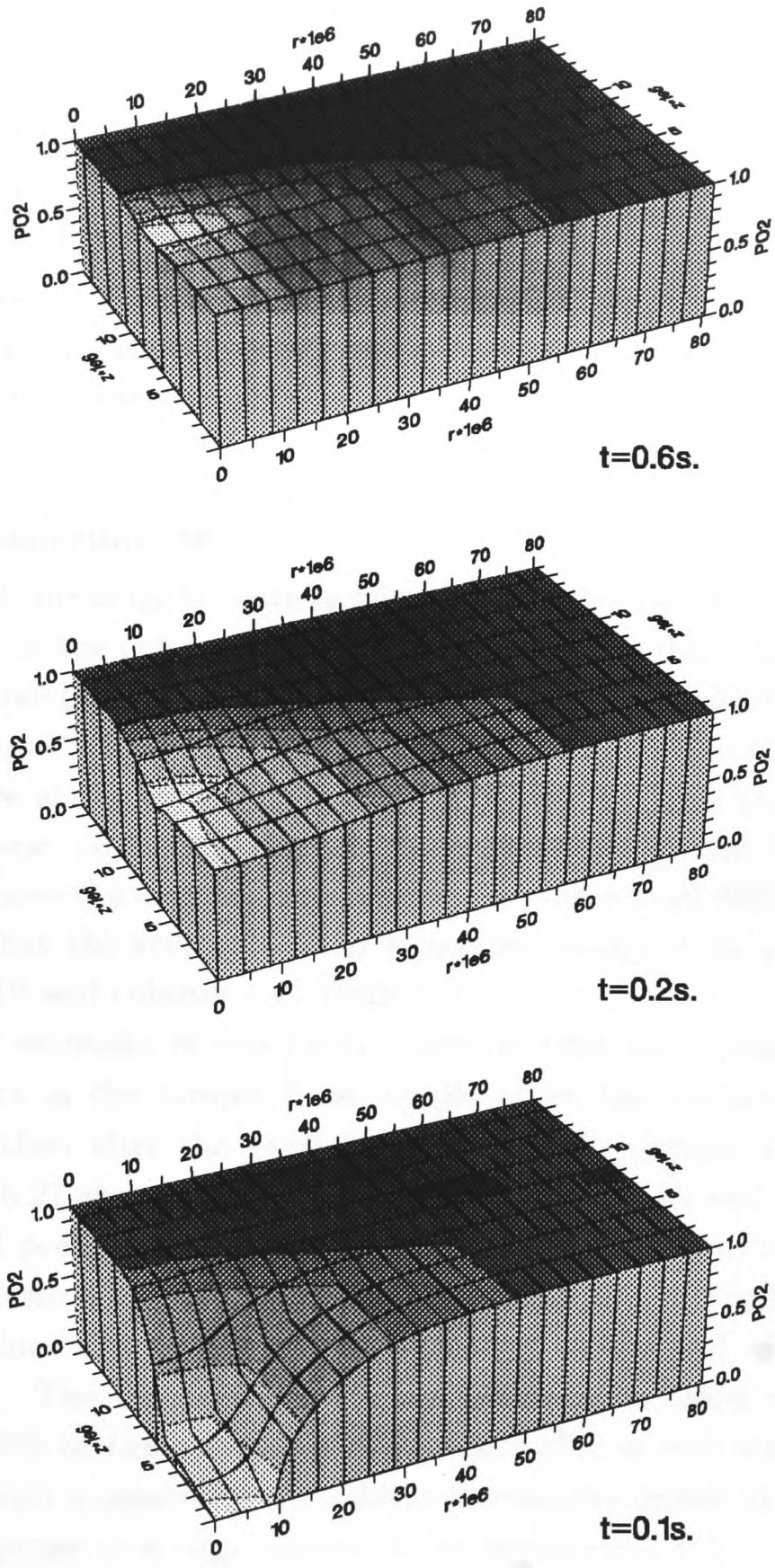


Fig. 6.18: Partial pressure surface at the end of a $100ms$ pulse, and $100ms$ and $500ms$ later with the potential removed for a $10\mu m$ cathode radius protected by a $5\mu m$ PTFE membrane.

Pulse	$P_{O_2} = 1.0$	$p/p_0 = 2.0$	$p/p_0 = 1.5$
1	110.61	103.07	205.68
2	105.83	137.20	188.64
3	104.21	165.29	174.61
4	103.55	181.64	166.44
⋮	⋮	⋮	⋮
12	102.92	205.39	154.49
13	102.91	205.56	154.44
14	102.91	205.66	154.41
15	102.91	205.72	154.39

Table 6.2: Response of the measured current per unit area at the end of each on–pulse to step changes in P_{O_2} in the sample.

6.3.5 Numerical experiments

In this section we will investigate numerically the time to pseudo–steady–state and the response time of the optimal geometry selected above: the $10\mu m$ cathode radius, using a $100ms$ on–pulse and a $500ms$ off–pulse, assuming a $5\mu m$ electrolyte layer protected by a $5\mu m$ PTFE membrane. Figure 6.20 shows the contour plots of equi–partial pressure at the end of the first, second and tenth on and off–pulses using this pulsing regime. As hoped, the sample is nowhere depleted by 1%. The region of electrolyte above the cathode is replenished to more than 90% by the end of each off–pulse, so that the system reaches a pseudo–steady state very quickly, as shown in Figure 6.19 and column 1 of Table 6.2.

We can obtain an estimate of the likely response time to a step change in oxygen partial pressure in the sample if we simply allow the system to reach a pseudo–steady–state, then alter the partial pressure in the sample and monitor the response[2]. Figure 6.21 shows the current per unit area at the end of each on–pulse, with the partial pressure doubled to $p/p_0 = 2.0$ at the end of the fifteenth off–pulse, and then reduced to $p/p_0 = 1.5$ at the end of the thirtieth off–pulse. The corresponding values of the current per unit area at the end of each pulse are given in Table 6.2. The response time of the sensor from being switched on initially to reaching 99% of the pseudo–steady–state value is just four complete pulses or about $2.5s$, with a requirement of about 9 complete pulses or $5s$ to make the same degree of response to a step–change in the sample P_{O_2} (the values at the end of the ninth pulse after each of the the step changes being 203.9 and 155.32, respectively).

The way in which the partial pressure in the sample equilibrates with that

[2]This requires a slight alteration to the boundary conditions given in Chapter 2, since we no longer wish to fix the value on the boundary at $r = r_{\max}$ (see Figure 2.3). We therefore replace the condition $p/p_0 = 1$ with $\frac{\partial p}{\partial r} = 0$ on $r = r_{\max} \forall z$.

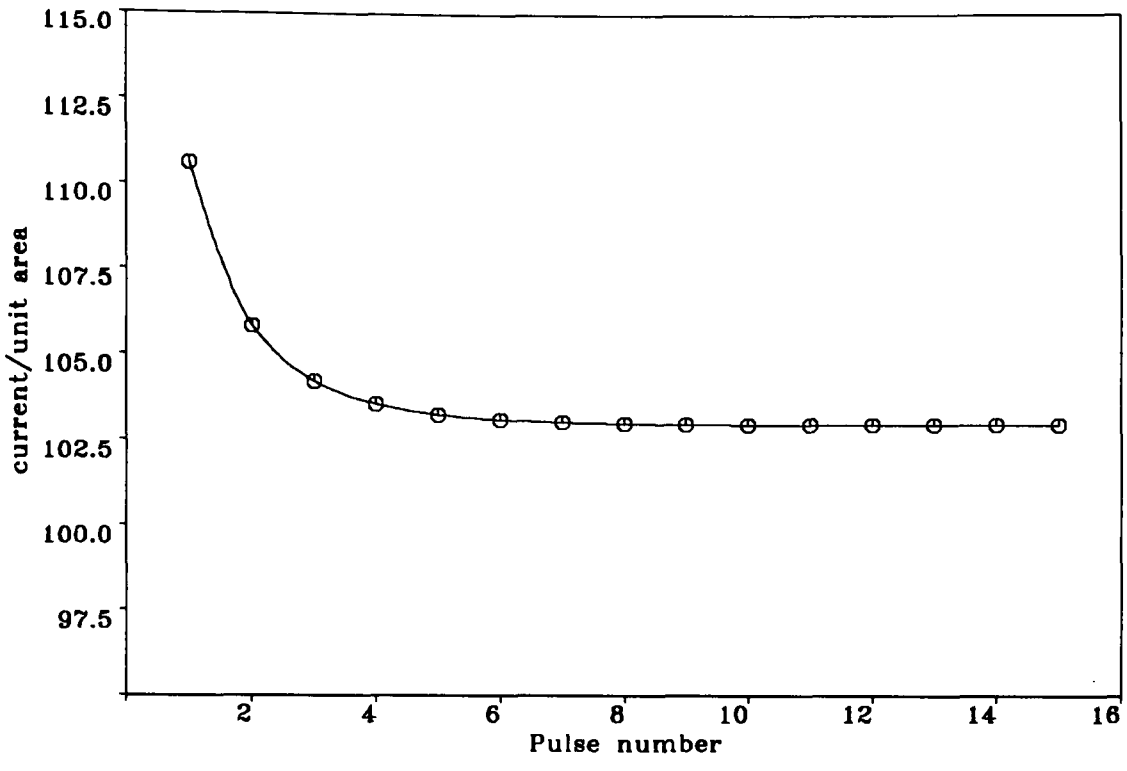


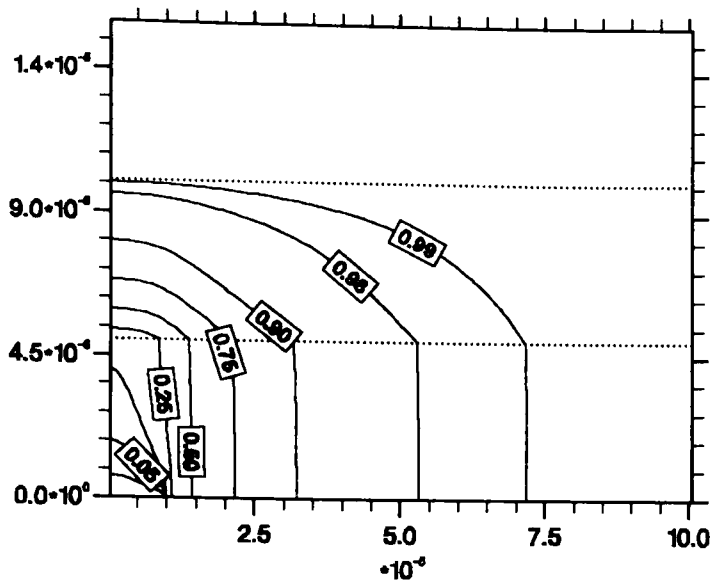
Fig. 6.19: Approach of calculated current per unit area at the end of each on–pulse to a pseudo–steady–state.

in the membrane and electrolyte after the first step–change from $p/p_0 = 1.0$ to $p/p_0 = 2.0$, is illustrated in Figure 6.22. By the end of the first on–pulse after the step–change, the partial pressure in the membrane has begun to increase. Since we are using a very thin PTFE membrane, by the end of the third on–pulse the partial pressure in the electrolyte has increased by about 50%, so that by the end of the tenth off–pulse, the diffusion pattern has almost returned to its state prior to the step change, but with each of the contour heights doubled.

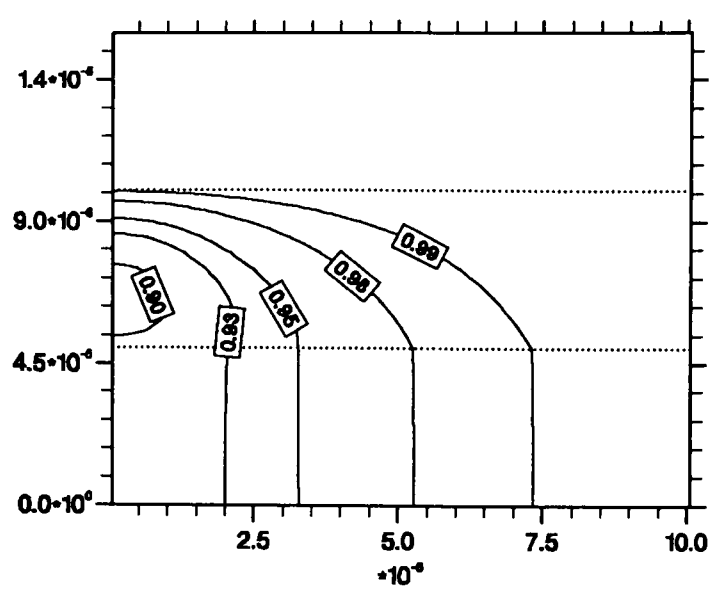
In Figure 6.23 we give the contour plots of equi–partial pressure at the end of the off–pulse immediately prior to each of the step changes, the end of the on–pulse immediately after each of the step changes, and finally after the system has been allowed to equilibrate with the $p/p_0 = 1.5$ for a further fifteen pulses (or $9s$). At this final time, it can be seen that the system has again returned to a pseudo–steady–state in which the effects of pulsing at the cathode have very little effect on the partial pressure in the sample. The immediate effects of the step changes are more graphically illustrated in Figure 6.24, which shows surface plots of the partial pressure at the end of the on–pulse immediately after each of the step changes.

6.4 Conclusions

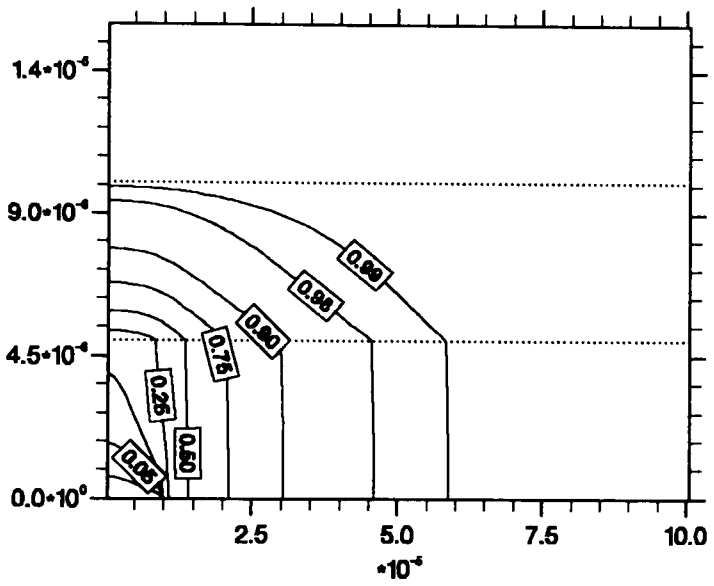
In the early part of this chapter we were able to demonstrate that the truly



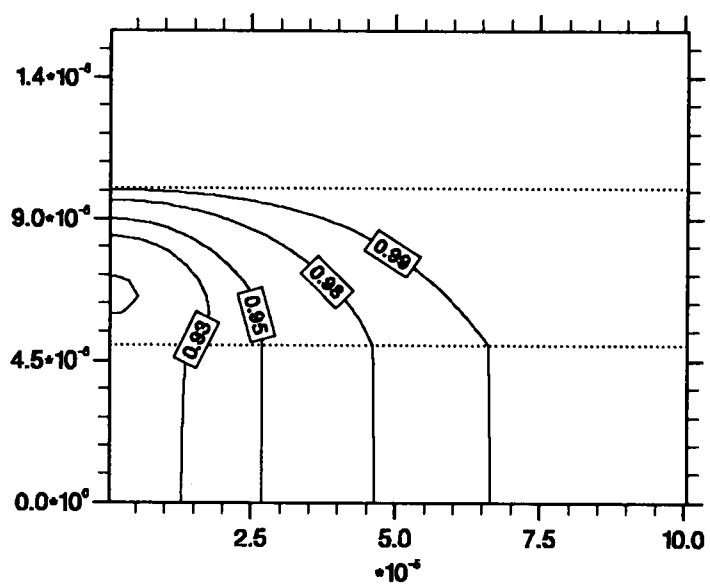
On pulse 10, t=5.5s.



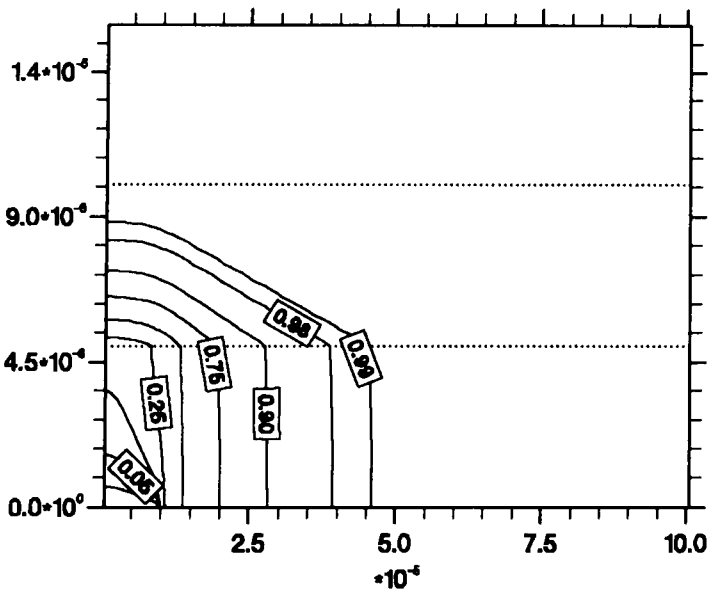
Off pulse 10, t=6.0s.



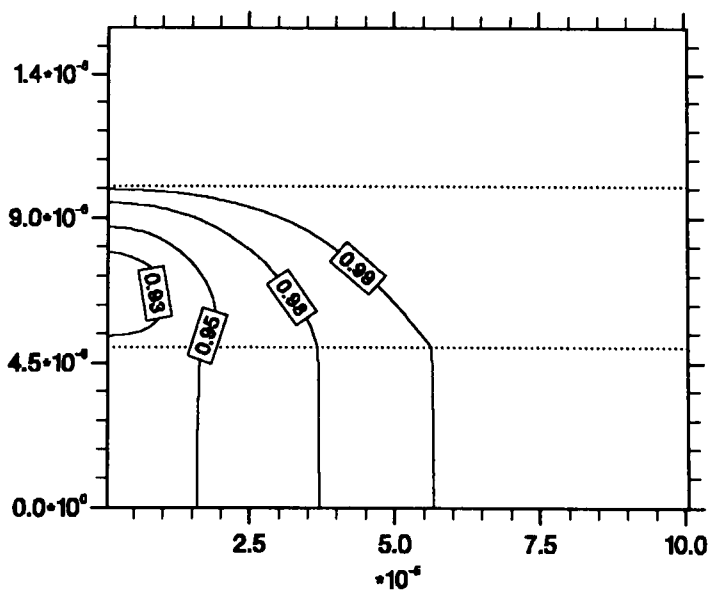
On pulse 2, t=0.7s.



Off pulse 2, t=1.2s.



On pulse 1, t=0.1s.



Off pulse 1, t=0.6s.

Fig. 6.20: Contours of equi–partial pressure at the end of the first, second and tenth on and off–pulses for a $10\mu m$ cathode radius protected by a $5\mu m$ PTFE membrane, using an on–pulse width of $100ms$, and an off–pulse width of $500ms$.

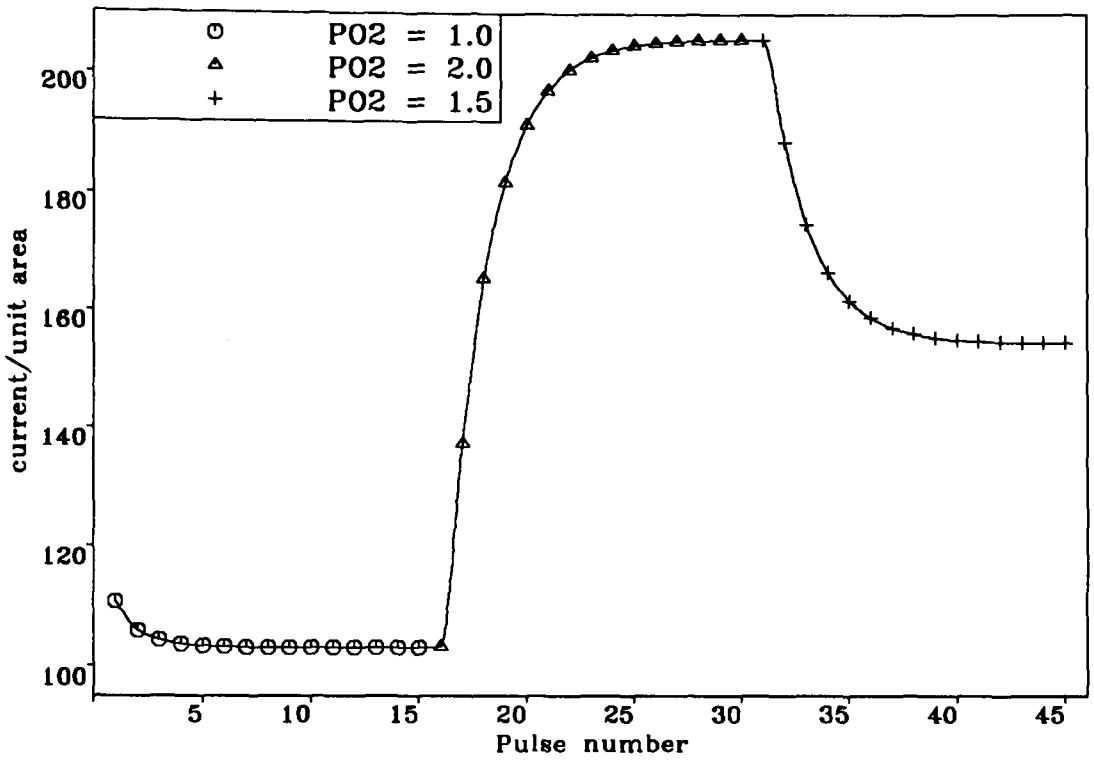
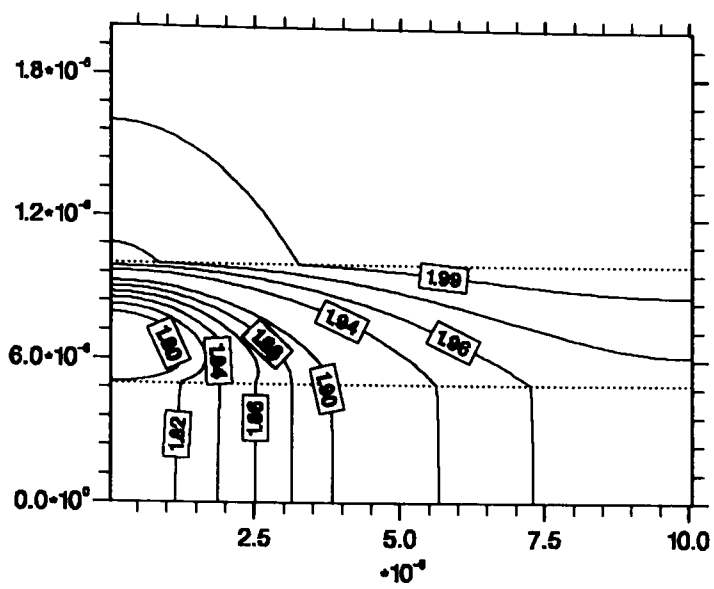


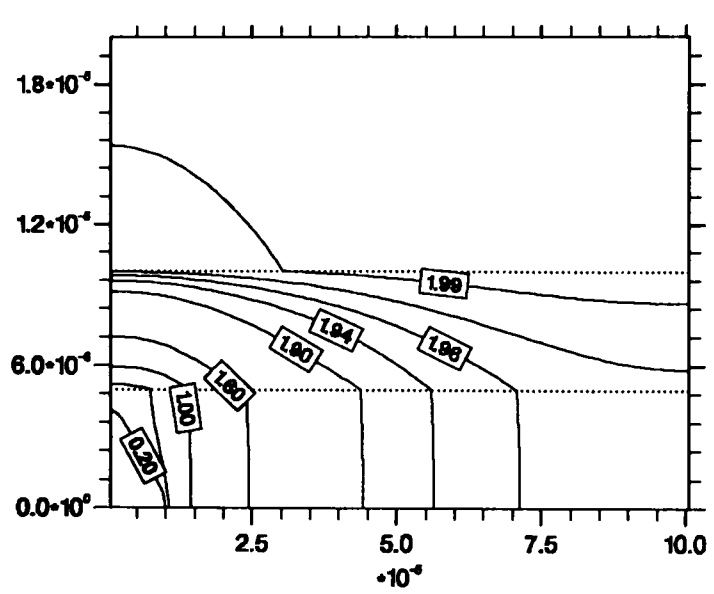
Fig. 6.21: Response time of calculated current per unit area at the end of each on–pulse to a step change of bulk P_{O_2} in the sample to $p/p_0 = 2.0$ at the end of the 15th off–pulse, and to $p/p_0 = 1.5$ at the end of the 30th off–pulse.

Cathode radius	Membrane thickness	On–pulse width	Off–pulse width	Time for 99% pseudo–st.–st.	Time for 99% response
$10\mu m$	$5\mu m$	$100ms$	$500ms$	$< 2.5s.$	$< 5s.$

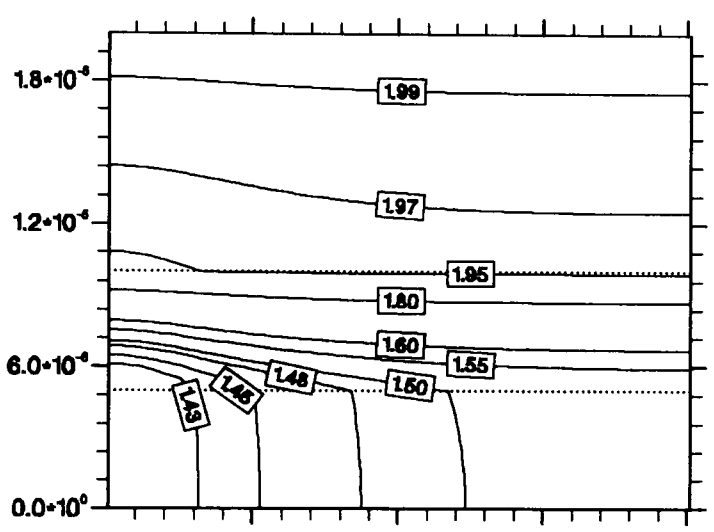
Table 6.3: Optimal theoretical electrode geometry and pulsing regime, with the times to reach 99% of the pseudo–steady–state and to give a 99% response to a step change in sample P_{O_2}.



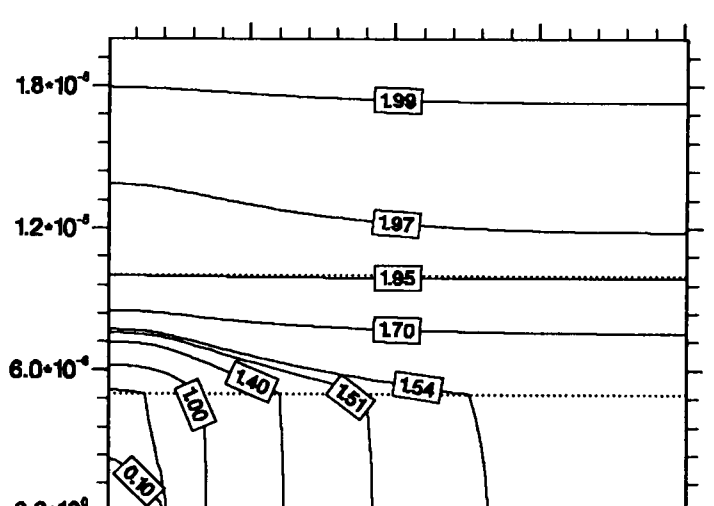
End of off-pulse. p0=2.0,t=15.0s.



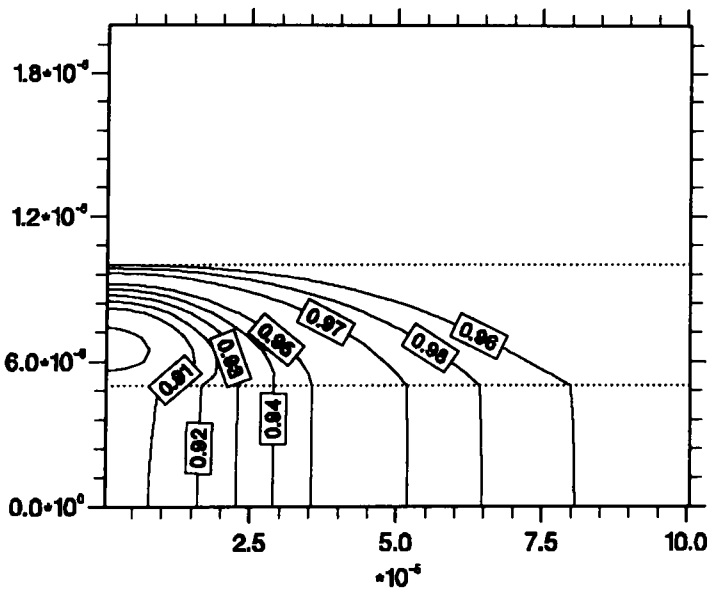
End of on-pulse. p0=2.0,t=15.1s.



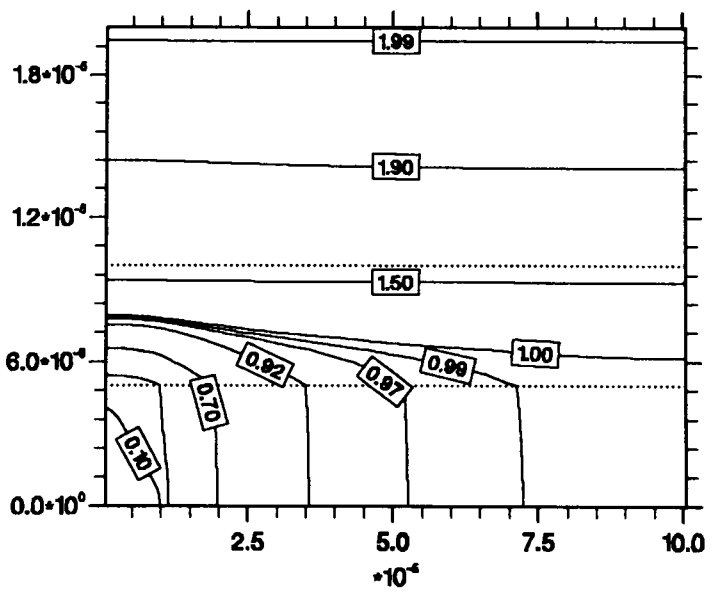
End of off-pulse. p0=2.0,t=10.2s.



End of on-pulse. p0=2.0,t=10.3s.

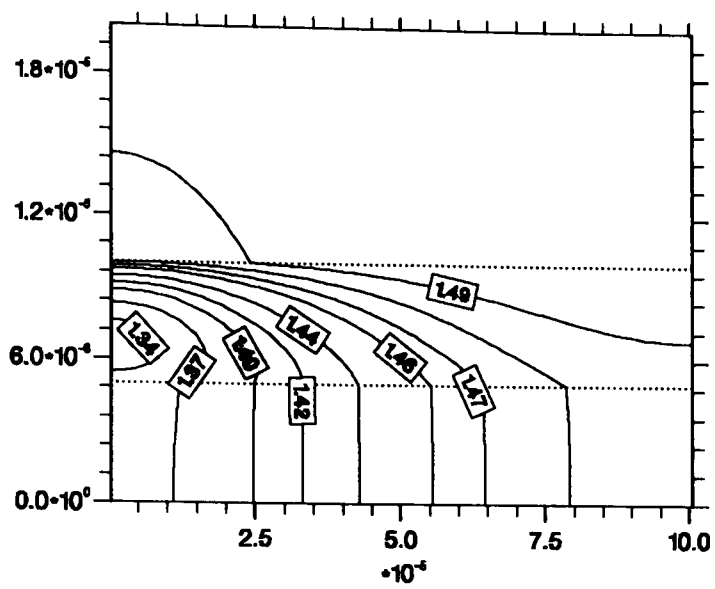


End of off-pulse. p0=1.0,t=9.0s.

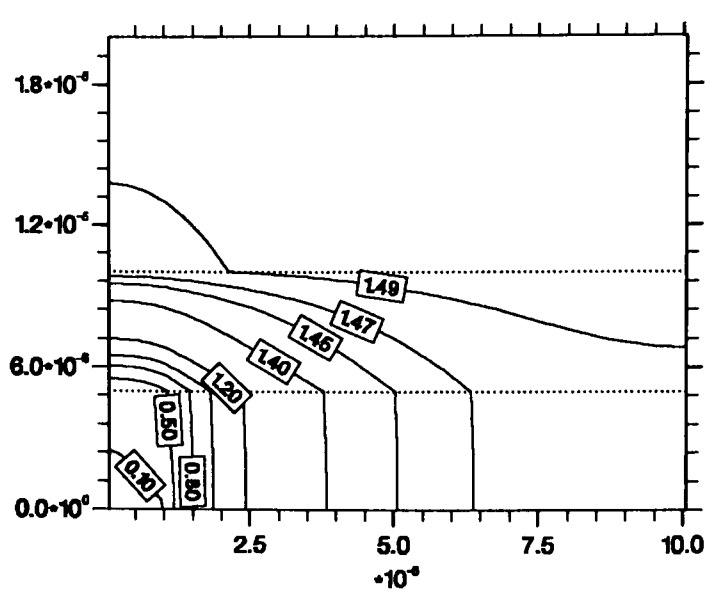


End of on-pulse. p0=2.0,t=9.1s.

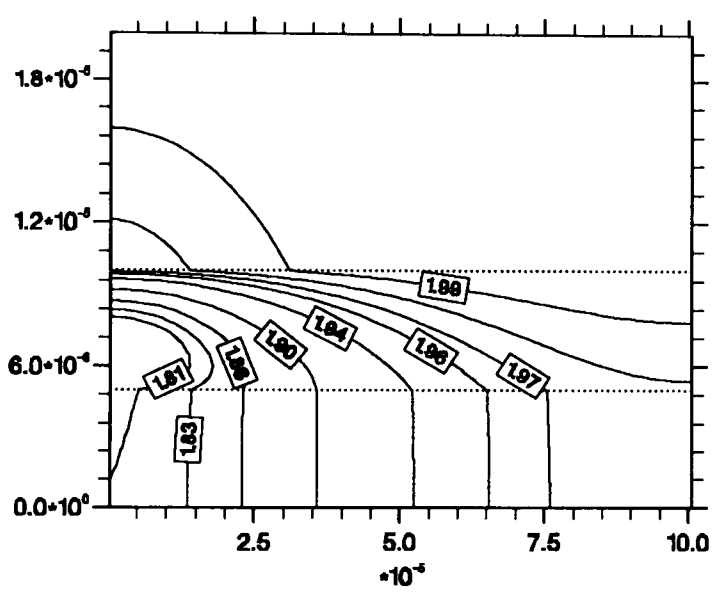
Fig. 6.22: Contours of equi–partial pressure at the end of the off–pulse prior to the step change from $p/p_0 = 1.0$ to $p/p_0 = 2.0$, the end of the first on–pulse after the step change, then at the end of the subsequent third and tenth on– and off–pulses.



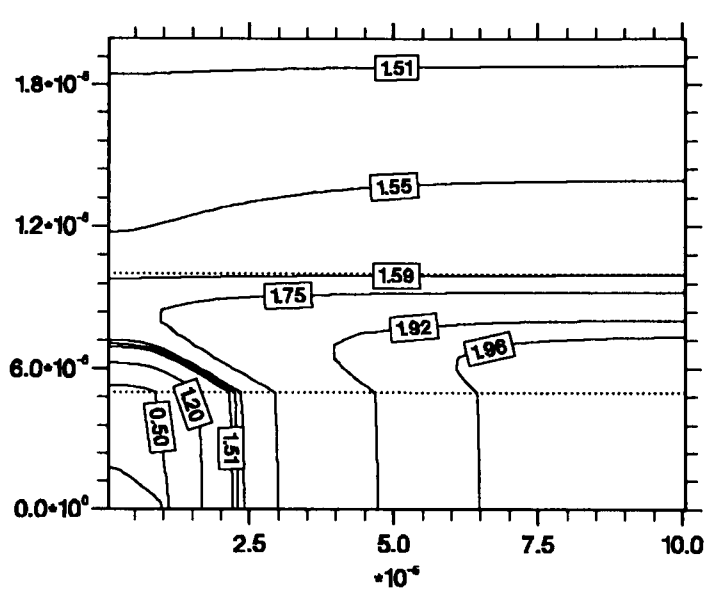
End of off-pulse. p0=1.5,t=27.0s.



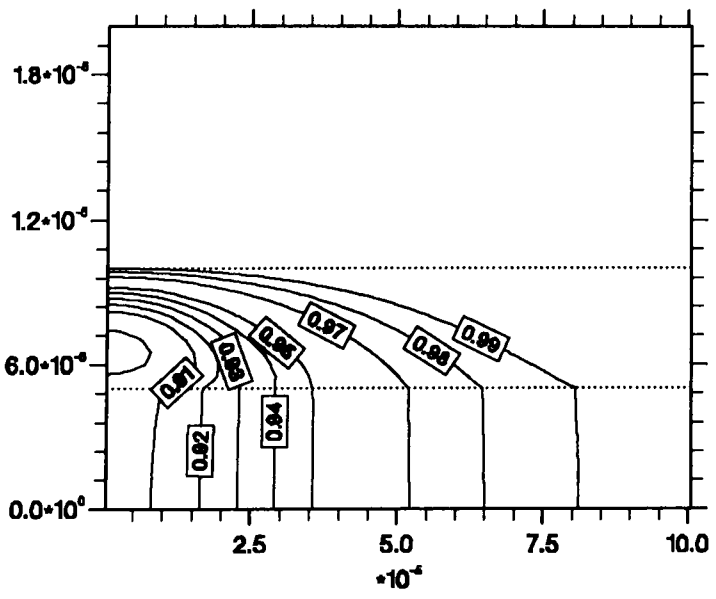
End of on-pulse. p0=1.5,t=27.1s.



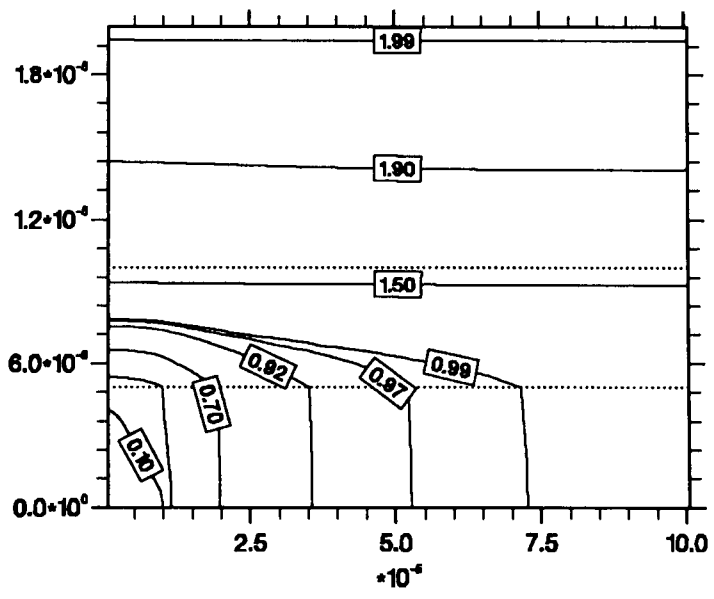
End of off-pulse. p0=2.0,t=18.0s.



End of on-pulse. p0=1.5,t=18.1s.



End of off-pulse. p0=1.0,t=9.0s.



End of on-pulse. p0=2.0,t=9.1s.

Fig. 6.23: Contours of equi–partial pressure at the end of each off–pulse immediately prior each of the step changes in sample P_{O_2}, at the end of the first on–pulse immediately after each step change, and finally at the end of the subsequent fifteenth off–pulse and sixteenth on–pulse with the sample P_{O_2} held fixed.

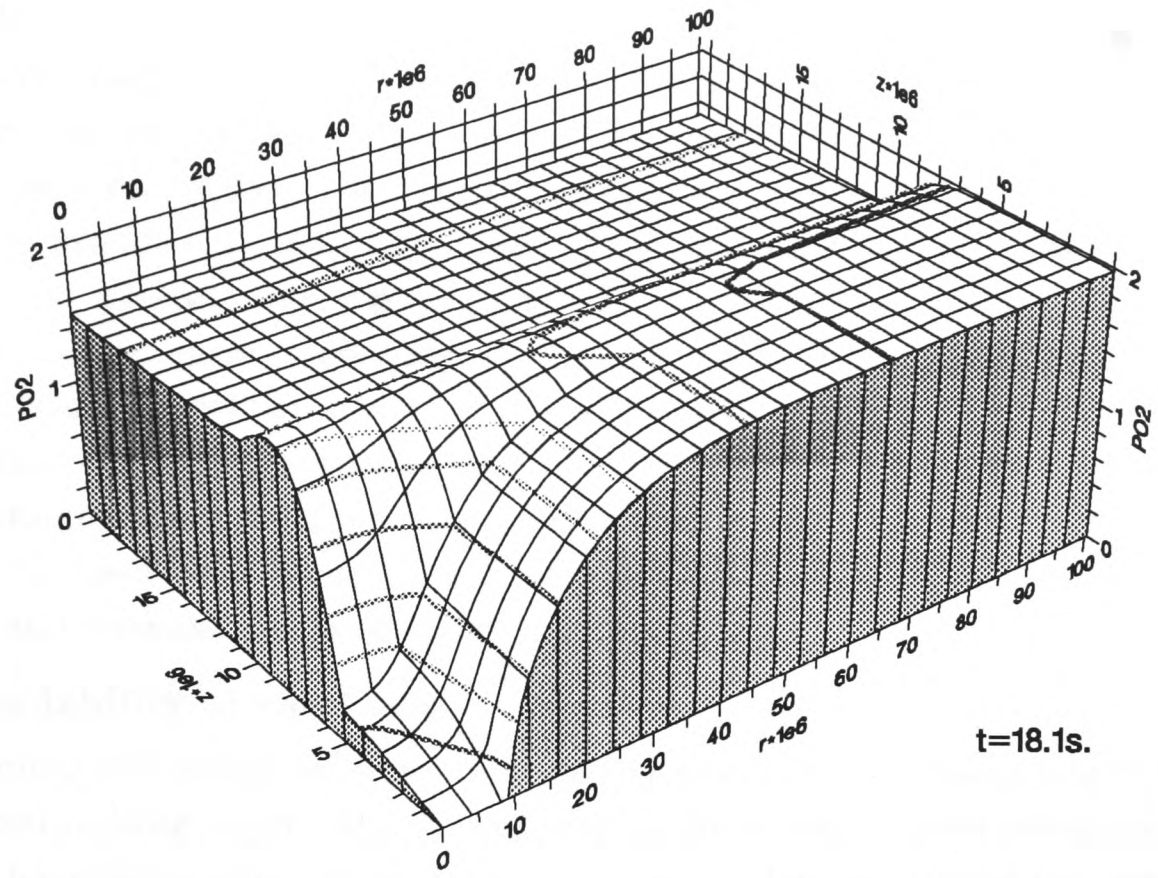


Fig. 6.24: Partial pressure surface at the end of the on–pulse immediately after the step change from $P_{O_2} = 2.0$ to $P_{O_2} = 1.5$ and at the end of the on–pulse immediately after the step change from $P_{O_2} = 1.0$ to $P_{O_2} = 2.0$

two–dimensional nature of the diffusion processes to a Clark electrode could not be adequately modelled in one–dimension for most practical situations. In investigating the time–dependent response of the two–dimensional model to changes in the governing parameters under a constantly applied reducing potential, we were able to gain sufficient understanding of the combined effects of the differing diffusion processes within each layer to go on to make numerical experiments to investigate practical *in vivo* situations. Based on practical limitations, we were able to select an electrode geometry and pulsing regime which is summarised in Table 6.3, and in theory, alleviates each of the difficulties of flow sensitivity and slow response time, whilst achieving a pseudo–steady–state very rapidly. We hope that by developing a theoretical model which predicts and more clearly defines the advantages displayed by pulse amperometry in the laboratory, the technique will soon be used with a commercial *in vivo* sensor. This should lead to a more accurate and reliable means of monitoring blood P_{O_2}.

6.5 Availability of the Code

The techniques we employed to investigate this particular geometry and to discover the optimal pulsing regime are equally applicable to any chosen electrode design. Since we have demonstrated earlier in the chapter that the model can adequately deal with all likely developments in sensor design, we hope that the code will become more widely used amongst both experimentalists and manufacturers. For this reason we include a sequential version of the Fortran code in Appendix C.

Chapter 7

Conclusions

Our aim at the outset of the research reported in this thesis was to investigate some of the difficulties involved in developing parallel numerical algorithms for the solution of parabolic partial differential equations. In mind, we had a particular application for the methods that we hoped to develop, that of modelling the Clark electrode for measuring oxygen concentration in blood. We achieved our parallel computing goals of determining which of the parallel algorithms that we developed was the most effective by extending and applying previous work. Before applying the chosen algorithm to our practical problem, it was necessary to first derive an analytic treatment for a boundary singularity, which in itself proved to be a novel solution to a problem in electrochemistry. In combining these numerical and analytical approaches, we obtained a tool with which we could interrogate the parameter space governing the operation of the Clark electrode in a relevant way, and then investigate the optimal theoretical geometry and mode of operation to satisfy the criteria laid down in the literature.

Along the way we have made some discoveries, and learnt many lessons, which we now take the opportunity of summarising.

7.1 Mathematical Methods

When using finite difference techniques, it is necessary to treat correctly any boundary singularities if the code is to be efficient. In addition to developing such a treatment in cylindrical coordinates, the nature of the internal boundary conditions for the shielded electrode problem also required the introduction of a more robust method of implementation when combining the finite difference and truncated series solutions. The underlying problem that we also solved in obtaining our method of correction, is that of the solution of the electrochemical problem of time–dependent diffusion to an unshielded disc electrode, for which there is also no generally valid analytic solution. The problem has therefore received much attention and we were able to show good agreement with previous numerical and analytic work, with the correction of the boundary singularity allowing us to obtain our solution much more economically. A less detailed account of the work described in Chapter 3 has appeared in the *Journal of Electroanalytical Chemistry* [28].

The inclusion of the singularity correction affected the truncation and global errors of the finite difference method in a complicated and case–dependent manner. Numerical experiments varying the mesh ratio and timestep for various cathode radii led us to determine appropriate refinement paths along which sufficiently accurate solutions could be extrapolated.

7.2 Computational Methods

Of the discoveries arising from our investigations of parallel numerical algorithms for a transputer based MIMD computer architecture, we will first consider those which are of specific interest for diffusion problems. However, we also encountered many difficulties primarily due to the inadequacy of the software tools available for such systems. These are of more general importance, and enable us to make several suggestions for helpful innovations from a user's point of view.

7.2.1 Parallel algorithms

Of the three algorithms that we chose to investigate as being representative of the class of parabolic partial differential equation solvers, two, the explicit and hopscotch methods, were effectively explicit in nature and could therefore be implemented on our parallel architecture with few changes to the basic sequential algorithm. The third, the ADI method, is an implicit method involving a tridiagonal solver. Previous parallel implementations of ADI have tended to move away from the most efficient sequential algorithm tuned to sequential computers, towards a substructuring technique more suited to a parallel architecture. However, since the partial differential equation that we wish to solve has time–independent coefficients, by changing the order of the calculations and effectively pipelining the calculations for successive rows of the mesh, we were able to obtain a 70% to 80% efficient parallel implementation of the ADI method. The resulting algorithm is mathematically identical to the most efficient sequential algorithm. Whilst our implementation may be slightly less *efficient* than the best substructuring algorithms, we avoid the associated penalty of using twice the number of operations and therefore obtain a more *effective* algorithm. Moreover, the associated parallel tridiagonal solver is more widely applicable, and has been used by Gwilliam [27, 33] to obtain a parallel solution method for the Navier–Stokes equations. For two–dimensional problems such as those that we have considered, the pipelined algorithm is likely to remain more effective than the substructuring approach. However, the method does not scale indefinitely, since as the mesh size increases, so does the number of processors needed to deal with a complete mesh line. Ultimately, we have more processors along a line than we have lines in a processor, and the efficiency of the pipelined algorithm begins to fall off. When it has fallen below 50%, the substructuring algorithm can become the more effective since it remains efficient on many processors.

Of the three methods considered for the solution of parabolic partial differential

equations, the ADI method was shown to be at least an order of magnitude more effective in terms of speed, than either of the other methods. However, it is implicit in nature, making its parallel implementation much less straightforward, so that in situations where software development time is of greater importance the hopscotch method would be more appropriate.

The statistical methods employed to analyse the run–times of the parallel implementations gave us greater understanding of the relationship between memory addressing and floating point calculation, and resulted in changes to the algorithm structure giving dramatic improvements in performance. These techniques are likely to become more valuable in optimising performance on more recent machines which combine processors made by different manufacturers, where the relationship will be much more complex.

A simple halo communication process was used for all three methods. Although we did not give a formal correctness proof for any of the algorithms, the simplicity of this process together with validity checks on the results for the simple model problem used to compare the methods, are adequate demonstration of the software's robustness. When numerically solving partial differential equations in this way, the simplicity of the halo process recommends it over more formally developed systems such as Leland's ONDE.

In modelling the Clark electrode, the fast and cheap processing that we required to gain a broad understanding of the wide range of parameters affecting design and performance were provided most economically by the transputer system. The development of the parallel code combined the complicated implementation of the ADI method with the boundary singularity, mixed boundary conditions and internal interface conditions arising from our modelling, resulting in a very complex parallel algorithm structure. Since we required our code to be scalable, we insisted that an identical code should run on every processor, exacerbating the difficulties. Whilst the effort involved in this code development was considerable, it was less than that required to understand the problem itself. Distributed memory computing did not, therefore, significantly degrade our ability to progress. However, our task would have been simplified by a more "user–friendly" programming environment. Since no libraries of standard numerical methods were available, we had to develop our own code to carry out standard numerical procedures. We found, more generally, that the tools provided by the manufacturers to help overcome the inherent difficulties were inadequate and personal debugging techniques were developed. In the next section we therefore describe some of the difficulties inherent in parallel computing, and make some suggestions for innovations which would help overcome them, particularly in the development of parallel numerical algorithms.

7.2.2 Parallel lessons

Over the last six years since the introduction of the transputer, there has been very rapid growth in the development of parallel hardware and a wide variety of systems is now available. Whilst the parallel codes that we have developed are written in Occam and therefore useful only on a transputer–based system, the lessons that we have learnt have a wider application, and allow us to suggest further developments which we feel are necessary in the operating systems and software tools to allow these systems to be used widely and to their full potential.

When we began our work on the electrode problem, the best software tools available to us were the software to support the Occam language which was designed specifically for the transputer. Since then, compilers have been developed for the transputer in the more traditional scientific languages of Fortran and C, for which the penalty for not using Occam is now only a factor of about 1.5. It seems likely that the advantages of familiarity and portability of these languages will tend to outweigh the elegance and theoretical rigour of Occam, which will probably be relegated, at best, to the role of an intermediate language.

In general, the porting of a sequential numerical code to a distributed memory system demands changes at several levels. In most cases, storage arrays must be divided up between processors and basic units of algorithms need to be redesigned or even replaced to allow an efficient parallel implementation. Some of the lower level changes may soon be handled painlessly by use of appropriate libraries, as these become available, and by (partially) parallelising compilers. Higher level changes (such as the redesign of the ADI method) will continue to demand the re–thinking of algorithm structure and research to ensure that the revised algorithms are as effective as those tuned to sequential systems.

At present, the environment for program development on MIMD machines is hostile, and it is usually advisable to test an algorithm on a sequential machine before moving it to a parallel system. This is partly due to the novelty and variety of the parallel architectures currently available, partly to the small scale on which such architectures are produced and supplied, and partly intrinsic in the complexity of parallel work. Since, particularly for numerical algorithms, the scalable nature of distributed memory systems gives them the potential to handle cases too large for the storage of any feasible sequential machine, part of the development work will need to be done on the parallel system. Currently available MIMD systems usually consist of a network of processors (configured as a rectangular array, tree, hypercube etc.) connected to a host machine. Difficulties in communication between different parts of the network and the host machine appear to be entirely due to the present state of parallel operating systems, and standardisation of communication procedures for all systems is necessary if parallel codes are to become portable between different types of hardware, or even between different operating systems for the same hardware. Generally there is a need for recognition of the

inherent and peculiar difficulties of locating errors in parallel codes and for the appearance of tools for program testing which reflect these difficulties, as adequately as do the equivalent tools for sequential systems. Until such tools are provided there will be little incentive for programmers to abandon the more familiar and straightforward architectures, and parallel processing will remain the preserve of a few dedicated specialists.

7.3 Modelling the Clark Electrode

The inability of one–dimensional models to agree with experimental observations has led to much confusion over the most appropriate design and operation of the Clark electrode. By making use of the cheap processing power made available through the parallel architecture, we have developed a two–dimensional model and demonstrated its validity to better than 1% precision. This has allowed us not only to illustrate how and why the previous one–dimensional models failed, but also to predict the behaviour of the Clark electrode more accurately than is currently possible experimentally. The understanding we have gained of the dependence of the operation of the electrode on its dimensions and membrane characteristics are summarised below.

7.3.1 Governing parameters

Assuming that the diffusion coefficients for all likely choices of electrolyte will be roughly equal, the cathode radius is the most critical parameter in determining the nature of the diffusion processes governing transport of oxygen. Using the widely quoted figure of $2.1 \times 10^{-9} m^2 s^{-1}$ for the diffusion coefficient of an aqueous electrolyte, we were able to show that radial diffusion is the dominant process for radii of the order of $1\mu m$, with a smooth transition through to dominant linear diffusion for radii of order $100\mu m$.

Making use of graphical software, we were able to gain many insights into the time–dependent variation of the solution for the diffusion processes. This led us to realise that for electrodes of practical size, no single one–dimensional process gives an acceptable approximation to electrode behaviour. However, it is possible to use a simpler model of two perpendicular regions of one–dimensional diffusion (coupled by subsidiary two–dimensional processes) to estimate the order of the timescale for diffusion processes to take place within the different layers. The graphical representations could be generated on comparatively coarse meshes both because the solution for the partial pressure is more accurate than that for the numerically derived current, and because less accuracy is required to make such qualitative assessments.

Consideration of a continuously applied potential to the cathode allowed us to consider the likely causes of the practical difficulties that we described in Chapter 2. The effects of varying electrolyte layer thickness and membrane characteristics were illustrated both quantitatively in terms of current sensitivity, and qualitatively

through consideration of diffusion patterns. In each case we were able to infer physically realistic explanations for the variations predicted by the model. It was shown that the ratio of the electrolyte layer thickness (which is difficult to ascertain in practice) to the cathode radius had a significant effect on the nature of the diffusion processes. At the longer times at which the "steady–state" measurements are made, sample depletion was shown to take place locally above the cathode surface. In addition to giving a reduced current, this also makes the electrode susceptible to small fluctuations in P_{O_2} in this region, such as might be caused by pulsatile blood flow. Increasing membrane thickness or decreasing its diffusivity were shown to lessen this problem, but only at the expense of a longer response time.

7.3.2 Practical operation

By considering the use of Clark electrodes in pulsed amperometry, we were able to give theoretical explanations for the advantages displayed by the technique in laboratory experiments in overcoming many of the practical difficulties involved in their use for *in vivo* measurements. This method succeeds by measuring the current during a short interval of applied potential, giving a large current sensitivity with low oxygen depletion. The primary advantage of the two–dimensional model over either of the one–dimensional models, that it allows lateral replenishment of oxygen to the vicinity of the cathode by diffusion parallel to the membrane surface, now becomes of paramount importance for understanding the situation. By choosing a cathode of appropriate size (of order $10\mu m$) we can ensure that the potential is applied for a sufficiently short period that significant oxygen depletion is largely limited to the electrolyte, spreading out radially over a wide area of inner membrane surface. During the relaxation period, this same rapid radial diffusion replaces the oxygen in the heavily depleted region immediately above the cathode surface, whilst the slow vertical diffusion through a large area of the membrane gives immunity from localised fluctuations in the sample. By making use of our simpler model of two regions of perpendicular one–dimensional diffusion, we were able to constrain the possible parameter space before making detailed numerical experiments. This allowed us to demonstrate the efficacy of the optimal theoretical geometry and pulsing regime that we determined in meeting the prescribed criteria of blood–flow insensitivity, large current sensitivity, and short response time.

7.4 Future Work

The application of the ADI method together with a correction procedure for the boundary singularity have provided an effective practical tool for evaluating the design of the Clark electrode. However, the technique is computationally expensive, and it has only been with the aid of the powerful, parallel computing resources provided by a transputer based system that we were able to proceed. It may be of interest to investigate the applicability of finite volume or finite element methods

to the problem, particularly if a special element can be designed to correct for the singularity, since these methods tend to give higher order accuracy. Whilst for the two–dimensional problem the savings in computing time are likely to be exceeded by the method development cost, if electrode design changes necessitate a three–dimensional model, their development may become worthwhile.

Experimental work to confirm our theoretical results has commenced, and so far gives good qualitative agreement for a constantly applied potential, with measured currents of the correct order of magnitude. A more quantitative check is hindered by uncertainty over the precise values of the various diffusion and solubility coefficients, and we have contacted two manufacturers in the hope of obtaining more precise values. There is a need for a procedure of accurately determining the electrolyte layer thickness, and recent reports of uncertainty over surface topography [78] suggest the possible need for a statistical model of the electrode surface.

The two–dimensional model that we have developed has advanced insight into the performance and narrowed down the range of uncertainty in the design of the Clark electrode. The theory is now ahead of experimental work, and we are in a position to predict the likely effects of design changes. In modelling the operation of the sensor using pulse amperometry, we were able to account for the advantages displayed in the laboratory, and to suggest the most appropriate practical technique for use *in vivo*. We hope that this will contribute to the development of a commercial *in vivo* sensor making use of the technique, and that our model will become widely used to enhance significantly the performance and reliability of the Clark electrode for the clinical monitoring of blood oxygen concentration.

Appendix A

The Structure of the Parallel Code

As we described in Chapters 6 and 7, we do not expect the Occam programming language to become widely used since it is available only for transputer–based systems. However, the underlying structure of the parallel code that we have developed can be used on any similar multiprocessor system, and is likely to be a valuable starting point for code development in any programming language.

A.1 The Operating System and the Associated Editor

The transputer system supplied by Inmos comes with its own operating system called the Transputer Development System(TDS) [59], specifically designed to aid the development of Occam programs. Programs can be edited, compiled and run either on a single processor, or loaded onto a network of processors, all within the system. The associated editor makes use of *folds* to clarify the program structure. These are analogous to folds in a piece of paper, in that each fold hides part of the program text. Folds can be displayed in two ways. An "open" fold reveals its contents displayed between two marker lines called *creases*, the top crease being marked by the symbol {{{ , and the bottom crease by }}}. A "closed" fold occupies only one line of text, which is marked with the symbol ... followed by suitable text to indicate the contents of the fold.

Folds may be nested to give the code a tree–like structure to a maximum depth of 50, so that the program can be folded in such a way that most of the folds are shorter than the length of the screen. Moving through the fold structure then becomes the main means of traversing the code. The crease markers may also be removed and the text contained within the fold is then placed in sequence with the surrounding lines of code. Each fold has an associated indentation at which the crease markers begin. The indentation is significant in an Occam program. We described in Chapter 4 how the name of a particular data type could be declared. Associated with each name is the region of the program in which the name is valid, called the *scope*, outside of which the name has no meaning. The scope of the name is indicated by the level of program indentation at which it is declared, and encompasses all parts of the code up to the point at which the indentation next becomes less than the level at declaration.

A.2 Code Structure

By using two levels of nested folds, we are able to reveal the underlying decision structure of the code (which we illustrated in Figure 5.3) in an obvious way.

A.2.1 Top level fold

The Occam code shown below is the top level fold containing the entire code for the Clark electrode problem.

```
#USE adilib
#USE dblmath
PROC adi(CHAN OF COMMS northin,northout,southin,southout,
                    eastin,eastout,westin,westout,VAL INT xid,yid)
  ... declare.variables
  ... PROCEDURES
  SEQ
    clock ? start
    initialise()
    setup()
    bc(u)
    bc(v)
    IF
      membrane = 1
        ... solve in electrolyte
      membrane = 2
        ... solve in electrolyte/membrane
      membrane = 3
        ... solve in electrolyte/membrane/sample
      membrane = 4
        ... solve in membrane
      membrane = 5
        ... solve in membrane/sample
      membrane = 6
        ... solve in sample
      TRUE
        SKIP
  clock ? finish
  time:=finish-start
  ... output time
  IF
    printflag = 1
      SEQ
        ... output.results
    TRUE
      SKIP
:
```

The code makes use of two libraries: the user–defined library **adilib** which contains all the parameter values defining the electrode geometry (number of processors in

each direction, mesh size, cathode size, thickness of each layer, time of run, neighbourhood of singularity correction); and the libraries of additional mathematical functions for double length arithmetic, **dblmath**, provided with the operating system. The entire procedure which is loaded onto each processor is called **adi**, which has as its ten formal parameters the eight required channels that we illustrated in Figure 4.3, together with the processor coordinates within the network. The **declare.variables** fold contains all the global variables, i.e. all variables whose scope is the entire program. The names of any variables declared at this level are also valid throughout each of the procedures which are contained in the following fold **PROCEDURES**. The text of all the procedures can be found in Appendix B. The procedures contained within the top–level fold are: **initialise** which initialises all of the global variables to 1.0; **setup**, which calculates the mesh ratios in both directions in all three layers, the position of the submesh in the global mesh, and calculates the value of the integer variable **membrane** according to the types of layer contained within the submesh; and **bc** which imposes the boundary conditions (2.5) to (2.7). The construct **clock ? start** uses **clock**, which has been declared as having type **timer**, to give a value representative of the time to the integer variable **start**. The conditional statement which follows then uses the value of **membrane** to determine the appropriate number of layers in the submesh for the processor with coordinates **(xid,yid)** in the processor network.

A.2.2 Second level folds

Each of the six "**membrane**" folds contains code to determine the position of the submesh relative to the singularity, and is of the same form as that shown below. This is the most complex possibility, with **membrane = 3**, that of the submesh containing all or part of all three layers.

```
{{{ solve in electrolyte/membrane/sample
IF
  singularity = 0 - sub mesh above cathode
    ... code for sing=0
  singularity = 1 - sub mesh above singularity
    ... code for sing=1
  singularity = 2 - sub mesh above boundary
    ... code for sing=2
  TRUE
    SKIP
}}}
```

A.2.3 Third level folds

Within each of the singularity folds is the appropriate timestepping algorithm for that particular submesh. The code shown below is contained in the second fold, **... code for sing = 2**. It calculates the solution at each timestep for the most complicated of all possible submeshes, which contains all or part of all three layers, and either contains or is above, the singularity.

```
{{{code sing = 1
SEQ
   xsetup(exe,fxe,gxe,re1,re2,re3)
   xsetup(exm,fxm,gxm,rm1,rm2,rm3)
   xsetup(exs,fxs,gxs,rs1,rs2,rs3)
   IF
      yid=1
        SEQ
        ysetup0(ey0,fy0,gy0)
        singcoeff()
        SEQ
           SEQ k=1 FOR onsteps
              SEQ
                 bcon(u)
                 solvev13()
                 interface(v,rate,ratm,me)
                 interface(v,ratm,rats,(me+mm))
                 halo(v)
                 bc(v)
                 solveforu11()
                 halo(u)
                 singcorr(u)
        calcc()
        currsing()
      TRUE
        SEQ
           ysetup(ey0,fy0,gy0)
           SEQ
              SEQ k=1 FOR onsteps
              SEQ
              bc(u)
              solvevgt13layer()
              interface(v,rate,ratm,me)
              interface(v,ratm,rats,(me+mm))
              halo(v)
              bc(v)
              solveforu1gt1()
              halo(u)
}}}
```

Each of the procedures used within the folds is defined within the top level fold marked **... PROCEDURES**. The procedures **xsetup** and **ysetup**[1] implement all of the LU decompositions (defined by equations (4.25)) required for that particular submesh, and are therefore carried out just once before the timestepping algorithm begins. The pseudo–inverse $\mathbf{F}^*$ and the associated vector μ^T (defined

[1]Since the code for the electrode problem was developed from the simpler ADI code to solve for the model problem of Chapter 4, the Cartesian notation using x and y was retained, and now correspond to the r and z directions respectively.

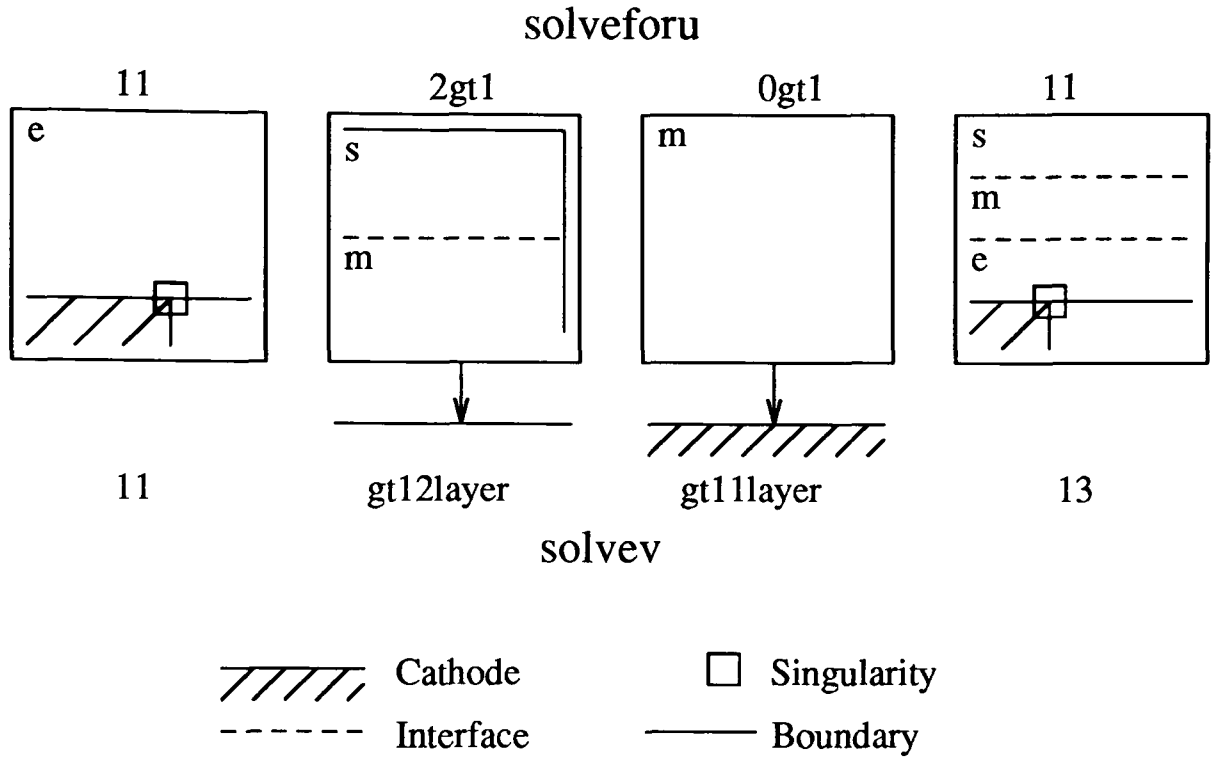


Fig. A.1: Four of the possible procedures to update the mesh lines on particular submeshes parallel to the z–axis (**solveforu**), and r–axis (**solvev**).

by equations(3.49) and (3.56)) are found by the procedure **singcoeff**, whilst **calcc** calculates the time–dependent coefficients $\mathbf{c}(t)$ of the truncated series solution (defined by equation (3.54)), and **currsing** calculates the current at the cathode (defined by equation (5.17)).

At each half timestep of the ADI method, meshlines parallel to the z–axis are updated using procedures whose names are of the form **solveforu**, whilst meshlines parallel to the r–axis use procedures whose names are of the form **solvev**. Both procedures calculate the right hand sides of equations(5.4) and then carry out the parallel solution algorithm described in Chapter 4. The full procedure name is determined by the nature of the submesh. For the **solveforu** procedures we have six cases: first we append a **0**, **1**, or **2** according to the value of **singularity**; this is followed by **1** if the submesh contains the first r–meshline (i.e. the boundary $z = 0$), or by **gt1** if not. For the **solvev** procedure we again have six possibilities: first we append a **1** if the submesh contains the first r–meshline; this is followed by **1** if the submesh is wholly contained within the electrolyte layer, **2** if it contains the electrolyte/membrane interface, and **3** if it contains the electrolyte/membrane interface and the membrane/sample interface; if the submesh does not contain the first r–meshline, then we first append **gt1** to **solvev**, followed by **1layer** if the submesh is wholly contained within just one layer, **2layer** if it contains either of the interfaces, and **3layer** if it contains both of the interfaces. In the code given above

we have **solvev13** and **solvevgt13layer**, and **solveforu11** and **solveforu1gt1**. Four further examples are illustrated in Figure A.1.

The names of the other procedures are hopefully self–evident from the description given in Chapters 4 and 5. The complete text of the Occam code is given in Appendix B.

A.3 The Sequential Code

Whilst the parallel Occam code is unlikely to be useful in its own right, the sequential Fortran code is easily ported between different sequential machines. A version (without any graphical routines) is therfore given in Appendix C. This code, together with the appropriate data files and associated codes, are available in machine–readable form on application to the author or his department.

Appendix B

The Occam Code to Solve the Clark Electrode Problem

```
#USE adilib
#USE dblmath
PROC adi(CHAN OF COMMS northin,northout,southin,southout,eastin,eastout,
                       westin,westout,VAL INT xid,yid)
  REAL64 taux,taux1,taux2,taux3,taux4,tauy,tauy1,tauy2,tauy3,tauy4,pi,total:
  REAL64 re,re1,re2,re3,re4,rm,rm1,rm2,rm3,rm4,rs,rs1,rs2,rs3,rs4,current,
  ze,ze1,ze2,ze3,ze4,zm,zm1,zm2,zm3,zm4,zs,zs1,zs2,zs3,zs4,rate,ratm,rats,
  currcomm,a1,b1,c1:
  [xnodes+2][ynodes+2]REAL64 u,v:
  [xnodes+2]REAL64 ax,bx,cx,ex0,fx0,gx0,exe,fxe,gxe,
  exm,fxm,gxm,exs,fxs,gxs,ex,fx,gx,solnx1,
  dx,zx0,zx,solnx,cft1,cft2:
  [ynodes+2]REAL64 ey0,fy0,gy0,ay,by,cy,ey,fy,gy,
  dy,zy0,zy,solny:
  [xnodes]REAL64 us,un,vs,vn:
  [ynodes]REAL64 ue,uw,ve,vw:
  [xnodes+2]REAL64 box:
  INT i,j,k,start,finish,time,xposn,yposn,sposn,membrane,
    singularity,in,iw,ie,zcount,now,corrtime,layer,singid:
  TIMER clock,delayclock:
  [fpt][npt]REAL64 f:
  [npt][fpt]REAL64 finv:
  [2*lpt][kpt+1][fpt]REAL64 coeff:
  [npt]REAL64 c:
  PROC setup()
    SEQ
      pi:=4.0(REAL64)*DATAN(1.0(REAL64))
      --Define nur's and nuz's (=pe*dt/dr/dr,...etc)
      re=(de*(dt/(dr*dr)))
      rm:=(dm*(dt/(dr*dr)))
      rs:=(ds*(dt/(dr*dr)))
      ze:=(de*(dt/(dze*dze)))
      zm:=(dm*(dt/(dzm*dzm)))
      zs:=(ds*(dt/(dzs*dzs)))
      re1:=-(0.5(REAL64)*re)
      re2:=1.0(REAL64)+re
      re3:=re1
      re4:=1.0(REAL64)-re
      rm1:=-(0.5(REAL64)*rm)
      rm2:=1.0(REAL64)+rm
      rm3:=rm1
      rm4:=1.0(REAL64)-rm
      rs1:=-(0.5(REAL64)*rs)
      rs2:=1.0(REAL64)+rs
      rs3:=rs1
      rs4:=1.0(REAL64)-rs
      ze1:=-(0.5(REAL64)*ze)
      ze2:=1.0(REAL64)+ze
      ze3:=ze1
      ze4:=1.0(REAL64)-ze
      zm1:=-(0.5(REAL64)*zm)
      zm2:=1.0(REAL64)+zm
      zm3:=zm1
      zm4:=1.0(REAL64)-zm
      zs1:=-(0.5(REAL64)*zs)
      zs2:=1.0(REAL64)+zs
      zs3:=zs1
      zs4:=1.0(REAL64)-zs
      rate:=pe/dze
      ratm:=pm/dzm
      rats:=ps/dzs
      xposn:=(xid-1)*xnodes
      yposn:=(yid-1)*ynodes
      SEQ i=0 FOR xnodes+2
        SEQ
          cft2[i]:=(REAL64 ROUND (2*(i+xposn)))/(REAL64 ROUND
          ((2*(i+xposn))-1))
          cft1[i]:=(REAL64 ROUND ((2*(i+xposn))-2))/
          (REAL64 ROUND ((2*(i+xposn))-1))
      IF
        xposn <= nr
          IF
            (xposn+xnodes) >= nr
              SEQ
                singularity := 1
                sposn:=(nr-xposn)
            TRUE
              singularity := 0
        TRUE
          singularity := 2
      IF
        yposn <= me
          IF
            (yposn+ynodes) <= me
              membrane := 1                --electrolyte
            TRUE
              IF
                (yposn+ynodes) <= (me+mm)
```

```
              membrane := 2            --electrolyte/membrane
            TRUE
              membrane := 3            --electrolye/membrane/sample
      TRUE
        IF
          yposn <= (me+mm)
            IF
              (yposn+ynodes) <= (me+mm)
                membrane := 4          --membrane
              TRUE
                membrane := 5          --membrane/sample
          TRUE
            membrane := 6              --sample
:
PROC initialise()
  SEQ
    SEQ i=0 FOR xnodes+2
      SEQ j=0 FOR ynodes+2
        SEQ
          u[i][j]:=1.0(REAL64)
          v[i][j]:=1.0(REAL64)
    SEQ i=0 FOR xnodes+2
      SEQ                              --xvectors
        ex0[i]:=1.0(REAL64)
        fx0[i]:=1.0(REAL64)
        gx0[i]:=1.0(REAL64)
        zx0[i]:=1.0(REAL64)
        ax[i]:=1.0(REAL64)
        bx[i]:=1.0(REAL64)
        cx[i]:=1.0(REAL64)
        ex[i]:=1.0(REAL64)
        fx[i]:=1.0(REAL64)
        gx[i]:=1.0(REAL64)
        zx[i]:=1.0(REAL64)
        dx[i]:=1.0(REAL64)
        solnx[i]:=1.0(REAL64)
        solnx1[i]:=1.0(REAL64)
        cft1[i]:=1.0(REAL64)
        cft2[i]:=1.0(REAL64)
    SEQ i=0 FOR ynodes+2
      SEQ                              --y vectors
        ey0[i]:=1.0(REAL64)
        fy0[i]:=1.0(REAL64)
        gy0[i]:=1.0(REAL64)
        zy0[i]:=1.0(REAL64)
        ay[i]:=1.0(REAL64)
        by[i]:=1.0(REAL64)
        cy[i]:=1.0(REAL64)
        ey[i]:=1.0(REAL64)
        fy[i]:=1.0(REAL64)
        gy[i]:=1.0(REAL64)
        zy[i]:=1.0(REAL64)
        dy[i]:=1.0(REAL64)
        solny[i]:=1.0(REAL64)
:
PROC bcon([xnodes+2][ynodes+2]REAL64 u)
  SEQ
    IF
      yid = 1
        IF
          (xposn >= nr)
            SEQ i=1 FOR xnodes
              u[i][0]:=u[i][2]
          TRUE
            IF
              ((xposn+xnodes) < nr)
                SEQ i=1 FOR xnodes
                  SEQ j=0 FOR 2
                    u[i][j]:=0.0(REAL64)
              TRUE
                SEQ
                  SEQ i=1 FOR (nr-xposn)-1
                    SEQ j=0 FOR 2
                      u[i][j]:=0.0(REAL64)
                  SEQ i=(nr-xposn) FOR ((xnodes+1)-(nr-xposn))
                    u[i][0]:=u[i][2]
      TRUE
        SKIP
    IF
      xid = 1
        SEQ j=1 FOR ynodes
          u[0][j]:=u[1][j]
      TRUE
        SKIP
:
PROC xrhs([xnodes+2]REAL64 dx,VAL INT j,REAL64 tauy1,tauy3,tauy4,taux3)
  SEQ
    SEQ i=1 FOR xnodes
```

```
      dx[i]:=((tauy4*u[i][j])-(tauy1*u[i][j-1]))-
      (tauy3*u[i][j+1])
    IF
      xid=xprocs
        dx[xnodes]:=dx[xnodes]-((cft2[xnodes]*taux3)*v[xnodes+1][j])
      TRUE
        SKIP
:
--solve in electrolyte/membrane/sample
PROC yrhs([ynodes+2]REAL64 dy,VAL INT i)
  SEQ
    --Calculate dy according to layer
    IF
      membrane = 1
        --solve in electrolyte
        SEQ
          SEQ j=1 FOR ynodes
            dy[j]:=((re4*v[i][j])-(cft1[i]*(re1*v[i-1][j])))-
            (cft2[i]*(re3*v[i+1][j]))
          IF
            yid=yprocs
              dy[ynodes]:=dy[ynodes]-(ze3*u[i][ynodes+1])
            TRUE
              SKIP
      membrane = 2
        --solve in electrolyte/membrane
        SEQ
          SEQ
            SEQ j=1 FOR (me-yposn)
              SEQ
                dy[j]:=((re4*v[i][j])-(cft1[i]*(re1*v[i-1][j])))-
                (cft2[i]*(re3*v[i+1][j]))                    --electrolyte
            dy[me-yposn]:=0.0(REAL64)
            SEQ j=(me-yposn)+1 FOR (ynodes-(me-yposn))
              SEQ
                dy[j]:=((rm4*v[i][j])-(cft1[i]*(rm1*v[i-1][j])))-
                (cft2[i]*(rm3*v[i+1][j]))                    --membrane
          IF
            yid=yprocs
              dy[ynodes]:=dy[ynodes]-(zm3*u[i][ynodes+1])
            TRUE
              SKIP
      membrane = 3
        --solve in electrolyte/membrane/sample
          SEQ
            SEQ j=1 FOR (me-yposn)
              SEQ
                dy[j]:=((re4*v[i][j])-(cft1[i]*(re1*v[i-1][j])))-
                (cft2[i]*(re3*v[i+1][j]))                    --electrolyte
            dy[me-yposn]:=0.0(REAL64)
            SEQ j=(me-yposn)+1 FOR (mm-1)
              SEQ
                dy[j]:=((rm4*v[i][j])-(cft1[i]*(rm1*v[i-1][j])))-
                (cft2[i]*(rm3*v[i+1][j]))                    --membrane
            dy[(me+mm)-yposn]:=0.0(REAL64)
            SEQ j=((me+mm)-yposn)+1 FOR ynodes-((me+mm)-yposn)
              SEQ
                dy[j]:=((rs4*v[i][j])-(cft1[i]*(rs1*v[i-1][j])))-
                (cft2[i]*(rs3*v[i+1][j]))                    --sample
            IF
              yid=yprocs
                dy[ynodes]:=dy[ynodes]-(zs3*u[i][ynodes+1])
              TRUE
                SKIP
      membrane = 4
        --solve in membrane
        SEQ
          SEQ j=1 FOR ynodes
            dy[j]:=((rm4*v[i][j])-(cft1[i]*(rm1*v[i-1][j])))-
            (cft2[i]*(rm3*v[i+1][j]))                        --membrane
          IF
            yid=yprocs
              dy[ynodes]:=dy[ynodes]-(zm3*u[i][ynodes+1])
            TRUE
              SKIP
      membrane = 5
        --solve in membrane/sample
        SEQ
          SEQ j=1 FOR ((me+mm)-yposn)
            SEQ
              dy[j]:=((rm4*v[i][j])-(cft1[i]*(rm1*v[i-1][j])))-
              (cft2[i]*(rm3*v[i+1][j]))                      --membrane
          dy[(me+mm)-yposn]:=0.0(REAL64)
          SEQ j=((me+mm)-yposn)+1 FOR (ynodes-((me+mm)-yposn))
            SEQ
              dy[j]:=((rs4*v[i][j])-(cft1[i]*(rs1*v[i-1][j])))-
              (cft2[i]*(rs3*v[i+1][j]))                      --sample
          IF
```

```
            yid=yprocs
              dy[ynodes]:=dy[ynodes]-(zs3*u[i][ynodes+1])
            TRUE
              SKIP
      membrane = 6
        --solve in sample
        SEQ
          SEQ j=1 FOR ynodes
            dy[j]:=((rs4*v[i][j])-(cft1[i]*(rs1*v[i-1][j])))-
            (cft2[i]*(rs3*v[i+1][j]))                          --sample
          IF
            yid=yprocs
              dy[ynodes]:=dy[ynodes]-(zs3*u[i][ynodes+1])
            TRUE
              SKIP
:
PROC xsetup([xnodes+2]REAL64 ex,fx,gx,REAL64 taux1,taux2,taux3)
  SEQ
    SEQ i=0 FOR xnodes+1
      SEQ
        ax[i]:=taux1*cft1[i]
        bx[i]:=taux2
        cx[i]:=taux3*cft2[i]
    IF
      xid=1
        SEQ
          bx[1]:=bx[1]+ax[1]
          fx[1]:=bx[1]
          gx[1]:=cx[1]
          SEQ i=2 FOR xnodes-1
            SEQ
              ex[i]:=ax[i]/fx[i-1]
              fx[i]:=bx[i]-(ex[i]*cx[i-1])
              gx[i]:=cx[i]
          IF
            xid=xprocs
              SKIP
            TRUE
              eastout ! real; fx[xnodes]
      TRUE
        SEQ
          westin ? CASE  real ;  fx[0]
          SEQ i=1 FOR xnodes
            SEQ
              ex[i]:=ax[i]/fx[i-1]
              fx[i]:=bx[i]-(ex[i]*cx[i-1])
              gx[i]:=cx[i]
          IF
            xid=xprocs
              SKIP
            TRUE
              eastout ! real ;  fx[xnodes]
:
PROC xsetup0([xnodes+2]REAL64 ex,fx,gx)
  SEQ
    SEQ i=sposn FOR (xnodes+1)-sposn
      SEQ
        ax[i]:=rel*cft1[i]
        bx[i]:=re2
        cx[i]:=re3*cft2[i]
    SEQ
      fx[sposn]:=bx[sposn]
      gx[sposn]:=cx[sposn]
      SEQ i=sposn+1 FOR (xnodes-sposn)
        SEQ
          ex[i]:=ax[i]/fx[i-1]
          fx[i]:=bx[i]-(ex[i]*cx[i-1])
          gx[i]:=cx[i]
      IF
        xid=xprocs
          SKIP
        TRUE
          eastout ! real ;  fx[xnodes]
:
PROC ysetup([ynodes+2]REAL64 ey,fy,gy)
  SEQ
    --Calculate ay,by,cy according to layer
    IF
      membrane = 1
        SEQ i=0 FOR ynodes+1
          SEQ
            ay[i]:=ze1
            by[i]:=ze2          --electrolyte
            cy[i]:=ze3
      membrane = 2
        SEQ
          SEQ i=0 FOR (me-yposn)
            SEQ
```

```
          ay[i]:=ze1
          by[i]:=ze2          --electrolyte
          cy[i]:=ze3
        ay[me-yposn]:=-((pe/dze)*(dt/dze))
        by[me-yposn]:=(((pe/dze)+(pm/dzm))*(dt/dze))
        cy[me-yposn]:=-((pm/dzm)*(dt/dze))
        SEQ i=(me-yposn)+1 FOR (ynodes-(me-yposn))
          SEQ
            ay[i]:=zm1
            by[i]:=zm2          --membrane
            cy[i]:=zm3
membrane = 3
  SEQ
    SEQ i=0 FOR (me-yposn)
      SEQ
        ay[i]:=ze1
        by[i]:=ze2          --electrolyte
        cy[i]:=ze3
    ay[me-yposn]:=-((pe/dze)*(dt/dze))
    by[me-yposn]:=(((pe/dze)+(pm/dzm))*(dt/dze))
    cy[me-yposn]:=-((pm/dzm)*(dt/dze))
    SEQ i=(me-yposn)+1 FOR (mm-1)
      SEQ
        ay[i]:=zm1
        by[i]:=zm2          --membrane
        cy[i]:=zm3
    ay[(me+mm)-yposn]:=-((pm/dzm)*(dt/dze))
    by[(me+mm)-yposn]:=(((pm/dzm)+(ps/dzs))*(dt/dze))
    cy[(me+mm)-yposn]:=-((ps/dzs)*(dt/dze))
    SEQ i=((me+mm)-yposn)+1 FOR ynodes-((me+mm)-yposn)
      SEQ
        ay[i]:=zs1
        by[i]:=zs2
        cy[i]:=zs3
membrane = 4
  SEQ i=0 FOR ynodes+1
    SEQ
      ay[i]:=zm1
      by[i]:=zm2          --membrane
      cy[i]:=zm3
membrane = 5
  SEQ
    SEQ i=0 FOR ((me+mm)-yposn)
      SEQ
```

```
          ay[i]:=zm1
          by[i]:=zm2          --membrane
          cy[i]:=zm3
        ay[(me+mm)-yposn]:=-((pm/dzm)*(dt/dze))
        by[(me+mm)-yposn]:=(((pm/dzm)+(ps/dzs))*(dt/dze))
        cy[(me+mm)-yposn]:=-((ps/dzs)*(dt/dze))
        SEQ i=((me+mm)-yposn)+1 FOR (ynodes-((me+mm)-yposn))
          SEQ
            ay[i]:=zs1
            by[i]:=zs2          --sample
            cy[i]:=zs3
  membrane = 6
    SEQ i=0 FOR ynodes+1
      SEQ
        ay[i]:=zs1
        by[i]:=zs2          --sample
        cy[i]:=zs3
--Calculate ey,fy,gy
IF
  yid=1
    SEQ
      cy[1]:=cy[1]+ay[1]
      fy[1]:=by[1]
      gy[1]:=cy[1]
      SEQ i=2 FOR ynodes-1
        SEQ
          ey[i]:=ay[i]/fy[i-1]
          fy[i]:=by[i]-(ey[i]*cy[i-1])
          gy[i]:=cy[i]
      IF
        yid=yprocs
          SKIP
        TRUE
          southout ! real ;  fy[ynodes]
TRUE
  SEQ
    northin ? CASE  real ;  fy[0]
    SEQ i=1 FOR ynodes
      SEQ
        ey[i]:=ay[i]/fy[i-1]
        fy[i]:=by[i]-(ey[i]*cy[i-1])
        gy[i]:=cy[i]
    IF
      yid=yprocs
```

```
            SKIP
          TRUE
            southout ! real ;  fy[ynodes]
:
PROC ysetup0([ynodes+2]REAL64 ey,fy,gy)
  SEQ
    --Calculate ay,by,cy according to layer
    IF
      membrane = 1
        SEQ i=0 FOR ynodes+1
          SEQ
            ay[i]:=ze1
            by[i]:=ze2        --electrolyte
            cy[i]:=ze3
      membrane = 2
        SEQ
          SEQ i=0 FOR (me-yposn)
            SEQ
              ay[i]:=ze1
              by[i]:=ze2        --electrolyte
              cy[i]:=ze3
          ay[me-yposn]:=-((pe/dze)*(dt/dze))
          by[me-yposn]:=(((pe/dze)+(pm/dzm))*(dt/dze))
          cy[me-yposn]:=-((pm/dzm)*(dt/dze))
          SEQ i=(me-yposn)+1 FOR (ynodes-(me-yposn))
            SEQ
              ay[i]:=zm1
              by[i]:=zm2        --membrane
              cy[i]:=zm3
      membrane = 3
        SEQ
          SEQ i=0 FOR (me-yposn)
            SEQ
              ay[i]:=ze1
              by[i]:=ze2        --electrolyte
              cy[i]:=ze3
          ay[me-yposn]:=-((pe/dze)*(dt/dze))
          by[me-yposn]:=(((pe/dze)+(pm/dzm))*(dt/dze))
          cy[me-yposn]:=-((pm/dzm)*(dt/dze))
          SEQ i=(me-yposn)+1 FOR (mm-1)
            SEQ
              ay[i]:=zm1
              by[i]:=zm2        --membrane
              cy[i]:=zm3
          ay[(me+mm)-yposn]:=-((pm/dzm)*(dt/dze))
          by[(me+mm)-yposn]:=(((pm/dzm)+(ps/dzs))*(dt/dze))
          cy[(me+mm)-yposn]:=-((ps/dzs)*(dt/dze))
          SEQ i=((me+mm)-yposn)+1 FOR ynodes-((me+mm)-yposn)
            SEQ
              ay[i]:=zs1
              by[i]:=zs2
              cy[i]:=zs3
      membrane = 4
        SEQ i=0 FOR ynodes+1
          SEQ
            ay[i]:=zm1
            by[i]:=zm2          --membrane
            cy[i]:=zm3
      membrane = 5
        SEQ
          SEQ i=0 FOR ((me+mm)-yposn)
            SEQ
              ay[i]:=zm1
              by[i]:=zm2        --membrane
              cy[i]:=zm3
          ay[(me+mm)-yposn]:=-((pm/dzm)*(dt/dze))
          by[(me+mm)-yposn]:=(((pm/dzm)+(ps/dzs))*(dt/dze))
          cy[(me+mm)-yposn]:=-((ps/dzs)*(dt/dze))
          SEQ i=((me+mm)-yposn)+1 FOR (ynodes-((me+mm)-yposn))
            SEQ
              ay[i]:=zs1
              by[i]:=zs2        --sample
              cy[i]:=zs3
      membrane = 6
        SEQ i=0 FOR ynodes+1
          SEQ
            ay[i]:=zs1
            by[i]:=zs2          --sample
            cy[i]:=zs3
    fy[2]:=by[2]
    gy[2]:=cy[2]
    SEQ i=3 FOR ynodes-2
      SEQ
        ey[i]:=ay[i]/fy[i-1]
        fy[i]:=by[i]-(ey[i]*cy[i-1])
        gy[i]:=cy[i]
    IF
      yid=yprocs
```

```
        SKIP
      TRUE
        southout ! real ;  fy[ynodes]
:
PROC win(REAL64 end)
  SEQ
    IF
      xid = 1
        SKIP
      TRUE
        westin ? CASE  real ; end
:
PROC wout(REAL64 end)
  SEQ
    IF
      xid = 1
        SKIP
      TRUE
        westout !  real ; end
:
PROC calczx([xnodes+2]REAL64 ex,zx)
  SEQ
    IF
      xid=1
        SEQ
          zx[1]:=dx[1]
          SEQ i=2 FOR xnodes-1
            SEQ
              zx[i]:=dx[i]-(ex[i]*zx[i-1])
      TRUE
        SEQ
          SEQ i=1 FOR xnodes
            SEQ
              zx[i]:=dx[i]-(ex[i]*zx[i-1])
:
PROC calczx0([xnodes+2]REAL64 ex,zx)
  SEQ
    zx[sposn]:=dx[sposn]
    SEQ i=sposn+1 FOR (xnodes-sposn)
      SEQ
        zx[i]:=dx[i]-(ex[i]*zx[i-1])
:
PROC calczy([ynodes+2]REAL64 ey,zy)
  SEQ
    IF
      yid=1
        SEQ
          zy[1]:=dy[1]
          SEQ j=2 FOR ynodes-1
            SEQ
              zy[j]:=dy[j]-(ey[j]*zy[j-1])
      TRUE
        SEQ
          SEQ j=1 FOR ynodes
            SEQ
              zy[j]:=dy[j]-(ey[j]*zy[j-1])
:
PROC calczy0([ynodes+2]REAL64 ey,zy)
  SEQ
    zy[2]:=dy[2]
    SEQ j=3 FOR ynodes-2
      SEQ
        zy[j]:=dy[j]-(ey[j]*zy[j-1])
:
PROC calcsolnx([xnodes+2]REAL64 fx,gx,zx,solnx)
  SEQ
    IF
      xid=xprocs
        SEQ
          solnx[xnodes]:=zx[xnodes]/fx[xnodes]
          SEQ i=1 FOR xnodes-1
            SEQ
              solnx[xnodes-i]:=(zx[xnodes-i]-(gx[xnodes-i]*
              solnx[xnodes-(i-1)]))/fx[xnodes-i]
      TRUE
        SEQ
          SEQ i=0 FOR xnodes
            SEQ
              solnx[xnodes-i]:=(zx[xnodes-i]-(gx[xnodes-i]*
              solnx[xnodes-(i-1)]))/fx[xnodes-i]
:
PROC calcsolnx0([xnodes+2]REAL64 fx,gx,zx,solnx)
  SEQ
    IF
      xid=xprocs
        SEQ
          solnx[xnodes]:=zx[xnodes]/fx[xnodes]
          SEQ i=1 FOR xnodes-sposn
```

```
            SEQ
              solnx[xnodes-i]:=(zx[xnodes-i]-(gx[xnodes-i]*
              solnx[xnodes-(i-1)]))/fx[xnodes-i]
        TRUE
          SEQ
            SEQ i=0 FOR (xnodes-sposn)+1
              SEQ
                solnx[xnodes-i]:=(zx[xnodes-i]-(gx[xnodes-i]*
                solnx[xnodes-(i-1)]))/fx[xnodes-i]
:
PROC calcsolny([ynodes+2]REAL64 fy,gy,zy,solny)
  SEQ
    IF
      yid=yprocs
        SEQ
          solny[ynodes]:=zy[ynodes]/fy[ynodes]
          SEQ j=1 FOR ynodes-1
            SEQ
              solny[ynodes-j]:=(zy[ynodes-j]-(gy[ynodes-j]*
              solny[ynodes-(j-1)]))/fy[ynodes-j]
      TRUE
        SEQ
          SEQ j=0 FOR ynodes
            SEQ
              solny[ynodes-j]:=(zy[ynodes-j]-(gy[ynodes-j]*
              solny[ynodes-(j-1)]))/fy[ynodes-j]
:
PROC calcsolny0([ynodes+2]REAL64 fy,gy,zy,solny)
  SEQ
    IF
      yid=yprocs
        SEQ
          solny[ynodes]:=zy[ynodes]/fy[ynodes]
          SEQ j=1 FOR ynodes-2
            SEQ
              solny[ynodes-j]:=(zy[ynodes-j]-(gy[ynodes-j]*
              solny[ynodes-(j-1)]))/fy[ynodes-j]
      TRUE
        SEQ j=0 FOR ynodes-1
          SEQ
            solny[ynodes-j]:=(zy[ynodes-j]-(gy[ynodes-j]*
            solny[ynodes-(j-1)]))/fy[ynodes-j]
:
PROC interface([xnodes+2][ynodes+2]REAL64 v,REAL64 c1,c2,VAL INT layer)
  SEQ
    SEQ i=1 FOR xnodes
      v[i][layer-yposn]:=((c1*v[i][(layer-yposn)-1])+
                          (c2*v[i][(layer-yposn)+1]))/(c1+c2)
:
PROC halo([xnodes+2][ynodes+2]REAL64 v)
  SEQ
    PAR
      IF
        xid=1
          IF
            xid=xprocs
              SKIP
            TRUE
              SEQ
                --east
                SEQ j=1 FOR ynodes
                  ve[j-1]:=v[xnodes][j]
                PAR
                  eastout ! yvector ;  ve
                  eastin ? CASE  yvector ;  ue
                SEQ j=1 FOR ynodes
                  v[xnodes+1][j]:=ue[j-1]
        xid=xprocs
          SEQ
            --west
            SEQ j=1 FOR ynodes
              SEQ
                vw[j-1]:=v[1][j]
            PAR
              westout ! yvector ;  vw
              westin ? CASE  yvector ;  uw
            SEQ j=1 FOR ynodes
              SEQ
                v[0][j]:=uw[j-1]
        TRUE
          SEQ
            --east/west
            SEQ j=1 FOR ynodes
              SEQ
                ve[j-1]:=v[xnodes][j]
                vw[j-1]:=v[1][j]
            PAR
              eastout ! yvector ;  ve
```

```
          westout ! yvector ;  vw
          westin ? CASE  yvector ;  uw
          eastin ? CASE  yvector ;  ue
        SEQ j=1 FOR ynodes
          SEQ
            v[xnodes+1][j]:=ue[j-1]
            v[0][j]:=uw[j-1]
IF
  yid=1
    IF
      yid=yprocs
        SKIP
      TRUE
        SEQ
          --south
          SEQ i=1 FOR xnodes
            vs[i-1]:=v[i][ynodes]
          PAR
            southout ! xvector ;  vs
            southin ? CASE  xvector ;  us
          SEQ i=1 FOR xnodes
            v[i][ynodes+1]:=us[i-1]
  yid=yprocs
    SEQ
      --north
      SEQ i=1 FOR xnodes
        SEQ
          vn[i-1]:=v[i][1]
      PAR
        northout ! xvector ;  vn
        northin ? CASE  xvector ;  un
      SEQ i=1 FOR xnodes
        SEQ
          v[i][0]:=un[i-1]
  TRUE
    SEQ
      --north/south
      SEQ i=1 FOR xnodes
        SEQ
          vs[i-1]:=v[i][ynodes]
          vn[i-1]:=v[i][1]
      PAR
        southout ! xvector ;  vs
        northout ! xvector ;  vn
          southin ? CASE  xvector ;  us
          northin ? CASE  xvector ;  un
        SEQ i=1 FOR xnodes
          SEQ
            v[i][ynodes+1]:=us[i-1]
            v[i][0]:=un[i-1]
:
PROC solveforu01()
  SEQ
    IF
      yid=yprocs
        SEQ
          i:=1
          zcount:=1
          WHILE i <= (2*xnodes)
            SEQ
              yrhs(dy,zcount)
              zy0[0]:=u[zcount][0]
              calczy0(ey0,zy0)
              i:=i+1
              calcsolny0(fy0,gy0,zy0,solny)
              SEQ j=2 FOR ynodes-1
                u[zcount][j]:=solny[j]
              zcount:=zcount+1
              i:=i+1
      TRUE
        SEQ
          SEQ i=1 FOR xnodes
            SEQ
              yrhs(dy,i)
              calczy0(ey0,zy0)
              southout ! real ;  zy0[ynodes]
              SEQ j=2 FOR ynodes-1
                u[i][j]:=zy0[j]
          SEQ i=1 FOR xnodes
            SEQ
              SEQ j=2 FOR ynodes-1
                zy0[j]:=u[i][j]
              southin ? CASE   real ; solny[ynodes+1]
              calcsolny0(fy0,gy0,zy0,solny)
              SEQ j=2 FOR ynodes-1
                u[i][j]:=solny[j]
:
PROC solveforu0gt1()
```

```
SEQ
  IF
    yid = yprocs
      SEQ
        i:=1
        zcount:=1
        WHILE i <= (2*xnodes)
          SEQ
            yrhs(dy,zcount)
            IF
              i <= xnodes
                northin ? CASE  real ;  u[i][0]
              TRUE
                northout ! real ;  u[i-xnodes][1]
            zy0[0]:=u[zcount][0]
            calczy(ey0,zy0)
            i:=i+1
            calcsolny(fy0,gy0,zy0,solny)
            SEQ j=1 FOR ynodes
              u[zcount][j]:=solny[j]
            zcount:=zcount+1
            IF
              i <= xnodes
                northin ? CASE  real ;  u[i][0]
              TRUE
                northout ! real ;  u[i-xnodes][1]
            i:=i+1
TRUE
  SEQ
    SEQ i=1 FOR xnodes
      SEQ
        yrhs(dy,i)
        northin ? CASE  real ;  zy0[0]
        calczy(ey0,zy0)
        southout ! real ;  zy0[ynodes]
        SEQ j=1 FOR ynodes
          u[i][j]:=zy0[j]
    SEQ i=1 FOR xnodes
      SEQ
        SEQ j=1 FOR ynodes
          zy0[j]:=u[i][j]
        southin ? CASE  real ;  solny[ynodes+1]
        calcsolny(fy0,gy0,zy0,solny)
        northout ! real ;  solny[1]
        SEQ j=1 FOR ynodes
          u[i][j]:=solny[j]
:
PROC solveforul1()
  SEQ
    IF
      yid=yprocs
        SEQ
          i:=1
          zcount:=1
          WHILE i <= (2*(sposn-1))
            SEQ
              yrhs(dy,zcount)
              zy0[0]:=u[zcount][0]
              calczy0(ey0,zy0)
              i:=i+1
              calcsolny0(fy0,gy0,zy0,solny)
              SEQ j=2 FOR ynodes-1
                u[zcount][j]:=solny[j]
              zcount:=zcount+1
              i:=i+1
          WHILE i <= (2*xnodes)
            SEQ
              yrhs(dy,zcount)
              zy[0]:=u[zcount][0]
              calczy(ey,zy)
              i:=i+1
              calcsolny(fy,gy,zy,solny)
              SEQ j=1 FOR ynodes
                u[zcount][j]:=solny[j]
              zcount:=zcount+1
              i:=i+1
    TRUE
      SEQ
        SEQ i=1 FOR sposn-1
          SEQ
            yrhs(dy,i)
            calczy0(ey0,zy0)
            southout ! real ;  zy0[ynodes]
            SEQ j=2 FOR ynodes-1
              u[i][j]:=zy0[j]
        SEQ i=sposn FOR xnodes-(sposn-1)
          SEQ
            yrhs(dy,i)
```

```
            calczy(ey,zy)
            southout ! real ;  zy[ynodes]
            SEQ j=1 FOR ynodes
              u[i][j]:=zy[j]
          SEQ i=1 FOR sposn-1
            SEQ
              SEQ j=2 FOR ynodes-1
                zy0[j]:=u[i][j]
              southin ? CASE  real ;  solny[ynodes+1]
              calcsolny0(fy0,gy0,zy0,solny)
              SEQ j=2 FOR ynodes-1
                u[i][j]:=solny[j]
          SEQ i=sposn FOR xnodes-(sposn-1)
            SEQ
              SEQ j=1 FOR ynodes
                zy[j]:=u[i][j]
              southin ? CASE  real ;  solny[ynodes+1]
              calcsolny(fy,gy,zy,solny)
              SEQ j=1 FOR ynodes
                u[i][j]:=solny[j]
:
PROC solveforulgt1()
  SEQ
    IF
      yid = yprocs
        SEQ
          i:=1
          zcount:=1
          WHILE i <= (2*(sposn-1))
            SEQ
              yrhs(dy,zcount)
              IF
                i <= xnodes
                  northin ? CASE  real ;  u[i][0]
                TRUE
                  northout ! real ;  u[i-xnodes][1]
              zy0[0]:=u[zcount][0]
              calczy(ey0,zy0)
              i:=i+1
              calcsolny(fy0,gy0,zy0,solny)
              SEQ j=1 FOR ynodes
                u[zcount][j]:=solny[j]
              zcount:=zcount+1
              IF
                i <= xnodes
                  northin ? CASE  real ;  u[i][0]
                TRUE
                  northout ! real ;  u[i-xnodes][1]
              i:=i+1
          WHILE i <= (2*xnodes)
            SEQ
              yrhs(dy,zcount)
              IF
                i <= xnodes
                  northin ? CASE  real ;  u[i][0]
                TRUE
                  northout ! real ;  u[i-xnodes][1]
              zy[0]:=u[zcount][0]
              calczy(ey,zy)
              i:=i+1
              calcsolny(fy,gy,zy,solny)
              SEQ j=1 FOR ynodes
                u[zcount][j]:=solny[j]
              zcount:=zcount+1
              IF
                i <= xnodes
                  northin ? CASE  real ;  u[i][0]
                TRUE
                  northout ! real ;  u[i-xnodes][1]
              i:=i+1
      TRUE
        SEQ
          SEQ i=1 FOR sposn-1
            SEQ
              yrhs(dy,i)
              northin ? CASE  real ;  zy0[0]
              calczy(ey0,zy0)
              southout ! real ;  zy0[ynodes]
              SEQ j=1 FOR ynodes
                u[i][j]:=zy0[j]
          SEQ i=sposn FOR xnodes-(sposn-1)
            SEQ
              yrhs(dy,i)
              northin ? CASE  real ;  zy[0]
              calczy(ey,zy)
              southout ! real ;  zy[ynodes]
              SEQ j=1 FOR ynodes
                u[i][j]:=zy[j]
```

```
          SEQ i=1 FOR sposn-1
            SEQ
              SEQ j=1 FOR ynodes
                zy0[j]:=u[i][j]
              southin ? CASE  real ;  solny[ynodes+1]
              calcsolny(fy0,gy0,zy0,solny)
              northout ! real ;  solny[1]
              SEQ j=1 FOR ynodes
                u[i][j]:=solny[j]
          SEQ i=sposn FOR xnodes-(sposn-1)
            SEQ
              SEQ j=1 FOR ynodes
                zy[j]:=u[i][j]
              southin ? CASE  real ;  solny[ynodes+1]
              calcsolny(fy,gy,zy,solny)
              northout ! real ;  solny[1]
              SEQ j=1 FOR ynodes
                u[i][j]:=solny[j]
:
PROC solveforu21()
  SEQ
    IF
  SEQ
    IF
      yid=yprocs
        SEQ
          i:=1
          zcount:=1
          WHILE i <= (2*xnodes)
            SEQ
              yrhs(dy,zcount)
              zy[0]:=u[zcount][0]
              calczy(ey,zy)
              i:=i+1
              calcsolny(fy,gy,zy,solny)
              SEQ j=1 FOR ynodes
                u[zcount][j]:=solny[j]
              zcount:=zcount+1
              i:=i+1
      TRUE
        SEQ
          SEQ i=1 FOR xnodes
            SEQ
              yrhs(dy,i)
              calczy(ey,zy)
              southout ! real ;  zy[ynodes]
              SEQ j=1 FOR ynodes
                u[i][j]:=zy[j]
            SEQ i=1 FOR xnodes
              SEQ
                SEQ j=1 FOR ynodes
                  zy[j]:=u[i][j]
                southin ? CASE  real ;  solny[ynodes+1]
                calcsolny(fy,gy,zy,solny)
                SEQ j=1 FOR ynodes
                  u[i][j]:=solny[j]
:
PROC solveforu2gt1()
  SEQ
    IF
      yid = yprocs
        SEQ
          i:=1
          zcount:=1
          WHILE i <= (2*xnodes)
            SEQ
              yrhs(dy,zcount)
              IF
                i <= xnodes
                  northin ? CASE  real ;  u[i][0]
                TRUE
                  northout ! real ;  u[i-xnodes][1]
              zy[0]:=u[zcount][0]
              calczy(ey,zy)
              i:=i+1
              calcsolny(fy,gy,zy,solny)
              SEQ j=1 FOR ynodes
                u[zcount][j]:=solny[j]
              zcount:=zcount+1
              IF
                i <= xnodes
                  northin ? CASE  real ;  u[i][0]
                TRUE
                  northout ! real ;  u[i-xnodes][1]
              i:=i+1
      TRUE
        SEQ
          SEQ i=1 FOR xnodes
            SEQ
              yrhs(dy,i)
```

```
            northin ? CASE  real ;  zy[0]
            calczy(ey,zy)
            southout ! real ;  zy[ynodes]
            SEQ j=1 FOR ynodes
              u[i][j]:=zy[j]
          SEQ i=1 FOR xnodes
            SEQ
            SEQ j=1 FOR ynodes
              zy[j]:=u[i][j]
            southin ? CASE  real ;  solny[ynodes+1]
            calcsolny(fy,gy,zy,solny)
            northout ! real ;  solny[1]
            SEQ j=1 FOR ynodes
              u[i][j]:=solny[j]
:
PROC solvev11()
  INT trick:
  SEQ
    IF
      xid=xprocs
        SEQ
          IF
SEQ
  IF
    singularity = 0
      SEQ
        j:=3
        zcount:=2
        trick:=0
    singularity = 1
      SEQ
        j:=1
        zcount:=1
        xrhs(dx,zcount,ze1,ze3,ze4,re3)
        zx[0]:=v[0][zcount]
        calczx0(ex0,zx0)
        j:=j+1
        calcsolnx0(fx0,gx0,zx0,solnx)
        SEQ i=sposn FOR xnodes-(sposn-1)
          v[i][zcount]:=solnx[i]
        zcount:=zcount+1
        j:=j+1
        trick:=1
    singularity = 2
      SEQ
        j:=1
        zcount:=1
        xrhs(dx,zcount,ze1,ze3,ze4,re3)
        IF
          j <= ynodes
            win(v[0][j])
          TRUE
            wout(v[1][j-ynodes])
        zx0[0]:=v[0][zcount]
        calczx(ex0,zx0)
        j:=j+1
        calcsolnx(fx0,gx0,zx0,solnx)
        SEQ i=1 FOR xnodes
          v[i][zcount]:=solnx[i]
        zcount:=zcount+1
        IF
          j <= ynodes
            win(v[0][j])
          TRUE
            wout(v[1][j-ynodes])
        j:=j+1
        trick:=0
      TRUE
        SKIP
    WHILE j <= (2*ynodes)
      SEQ
        xrhs(dx,zcount,ze1,ze3,ze4,re3)
        IF
          j <= (ynodes+trick)
            win(v[0][j])
          TRUE
            wout(v[1][j-ynodes])
        zx[0]:=v[0][zcount]
        calczx(exe,zx)
        j:=j+1
        calcsolnx(fxe,gxe,zx,solnx)
        SEQ i=1 FOR xnodes
          v[i][zcount]:=solnx[i]
        zcount:=zcount+1
        IF
          j <= (ynodes+trick)
            win(v[0][j])
          TRUE
            wout(v[1][j-ynodes])
        j:=j+1
```

```
TRUE
  SEQ
    IF
      singularity = 0
        SKIP
      singularity = 1
        SEQ
          SEQ j=1 FOR 1
            SEQ
              xrhs(dx,j,ze1,ze3,ze4,re3)
              calczx0(ex0,zx0)
              eastout ! real ;  zx0[xnodes]
              SEQ i=sposn FOR xnodes-(sposn-1)
                v[i][j]:=zx0[i]
      singularity = 2
        SEQ
          SEQ j=1 FOR 1
            SEQ
              xrhs(dx,j,ze1,ze3,ze4,re3)
              win(zx0[0])
              calczx(ex0,zx0)
              eastout ! real ;  zx0[xnodes]
              SEQ i=1 FOR xnodes
                v[i][j]:=zx0[i]
    TRUE
      SKIP
  SEQ j=2 FOR ynodes-1
    SEQ
      xrhs(dx,j,ze1,ze3,ze4,re3)
      win(zx[0])
      calczx(exe,zx)
      eastout ! real ;  zx[xnodes]
      SEQ i=1 FOR xnodes
        v[i][j]:=zx[i]
  IF
    singularity = 0
      SKIP
    singularity = 1
      SEQ
        SEQ j=1 FOR 1
          SEQ
            SEQ i=sposn FOR xnodes-(sposn-1)
              zx0[i]:=v[i][j]
            eastin ? CASE  real ;  solnx[xnodes+1]
            calcsolnx0(fx0,gx0,zx0,solnx)
            SEQ i=sposn FOR xnodes-(sposn-1)
              v[i][j]:=solnx[i]
      singularity = 2
        SEQ
          SEQ j=1 FOR 1
            SEQ
              SEQ i=1 FOR xnodes
                zx0[i]:=v[i][j]
              eastin ? CASE  real ;  solnx[xnodes+1]
              calcsolnx(fx0,gx0,zx0,solnx)
              wout(solnx[1])
              SEQ i=1 FOR xnodes
                v[i][j]:=solnx[i]
    TRUE
      SKIP
  SEQ j=2 FOR ynodes-1
    SEQ
      SEQ i=1 FOR xnodes
        zx[i]:=v[i][j]
      eastin ? CASE  real ;  solnx[xnodes+1]
      calcsolnx(fxe,gxe,zx,solnx)
      wout(solnx[1])
      SEQ i=1 FOR xnodes
        v[i][j]:=solnx[i]
:
PROC solvev12()
  INT trick:
  SEQ
    IF
      xid=xprocs
        SEQ
          IF
            singularity = 0
              SEQ
                j:=3
                zcount:=2
                trick:=0
            singularity = 1
              SEQ
                j:=1
                zcount:=1
                xrhs(dx,zcount,ze1,ze3,ze4,re3)
                zx[0]:=v[0][zcount]
```

```
      calczx0(ex0,zx0)
      j:=j+1
      calcsolnx0(fx0,gx0,zx0,solnx)
      SEQ i=sposn FOR xnodes-(sposn-1)
        v[i][zcount]:=solnx[i]
      zcount:=zcount+1
      j:=j+1
      trick:=1
singularity = 2
  SEQ
    j:=1
    zcount:=1
    xrhs(dx,zcount,ze1,ze3,ze4,re3)
    IF
      j <= ynodes
        win(v[0][j])
      TRUE
        wout(v[1][j-ynodes])
    zx0[0]:=v[0][zcount]
    calczx(ex0,zx0)
    j:=j+1
    calcsolnx(fx0,gx0,zx0,solnx)
    SEQ i=1 FOR xnodes
      v[i][zcount]:=solnx[i]
    zcount:=zcount+1
    IF
      j <= ynodes
        win(v[0][j])
      TRUE
        wout(v[1][j-ynodes])
    j:=j+1
    trick:=0
  TRUE
    SKIP
WHILE j <= (2*ynodes)
  SEQ
    IF
      j <= (ynodes+trick)
        win(v[0][j])
      TRUE
        wout(v[1][j-ynodes])
    zx[0]:=v[0][zcount]
    IF
      zcount < (me-yposn)
        SEQ
          xrhs(dx,zcount,ze1,ze3,ze4,re3)
          calczx(exe,zx)
          j:=j+1
          calcsolnx(fxe,gxe,zx,solnx)
      zcount > (me-yposn)
        SEQ
          xrhs(dx,zcount,zm1,zm3,zm4,rm3)
          calczx(exm,zx)
          j:=j+1
          calcsolnx(fxm,gxm,zx,solnx)
      TRUE
        j:=j+1
    SEQ i=1 FOR xnodes
      v[i][zcount]:=solnx[i]
    zcount:=zcount+1
    IF
      j <= (ynodes+trick)
        win(v[0][j])
      TRUE
        wout(v[1][j-ynodes])
    j:=j+1
TRUE
  SEQ
    IF
      singularity = 0
        SKIP
      singularity = 1
        SEQ
          SEQ j=1 FOR 1
            SEQ
              xrhs(dx,j,ze1,ze3,ze4,re3)
              calczx0(ex0,zx0)
              eastout ! real ;  zx0[xnodes]
              SEQ i=sposn FOR xnodes-(sposn-1)
                v[i][j]:=zx0[i]
      singularity = 2
        SEQ
          SEQ j=1 FOR 1
            SEQ
              xrhs(dx,j,ze1,ze3,ze4,re3)
              win(zx0[0])
              calczx(ex0,zx0)
              eastout ! real ;  zx0[xnodes]
```

```
            SEQ i=1 FOR xnodes
              v[i][j]:=zx0[i]
      TRUE
        SKIP
    SEQ j=2 FOR ynodes-1
      SEQ
        win(zx[0])
        IF
          j < (me-yposn)
            SEQ
              xrhs(dx,j,ze1,ze3,ze4,re3)
              calczx(exe,zx)
          j > (me-yposn)
            SEQ
              xrhs(dx,j,zm1,zm3,zm4,rm3)
              calczx(exm,zx)
          TRUE
            SKIP
        eastout ! real ;  zx[xnodes]
        SEQ i=1 FOR xnodes
          v[i][j]:=zx[i]
IF
  singularity = 0
    SKIP
  singularity = 1
    SEQ
      SEQ j=1 FOR 1
        SEQ
          SEQ i=sposn FOR xnodes-(sposn-1)
            zx0[i]:=v[i][j]
          eastin ? CASE  real ;  solnx[xnodes+1]
          calcsolnx0(fx0,gx0,zx0,solnx)
          SEQ i=sposn FOR xnodes-(sposn-1)
            v[i][j]:=solnx[i]
  singularity = 2
    SEQ
      SEQ j=1 FOR 1
        SEQ
          SEQ i=1 FOR xnodes
            zx0[i]:=v[i][j]
          eastin ? CASE  real ;  solnx[xnodes+1]
          calcsolnx(fx0,gx0,zx0,solnx)
          wout(solnx[1])
          SEQ i=1 FOR xnodes
                v[i][j]:=solnx[i]
      TRUE
        SKIP
    SEQ j=2 FOR ynodes-1
      SEQ
        SEQ i=1 FOR xnodes
          zx[i]:=v[i][j]
        eastin ? CASE  real ;  solnx[xnodes+1]
        IF
          j < (me-yposn)
            SEQ
              calcsolnx(fxe,gxe,zx,solnx)
          j > (me-yposn)
            SEQ
              calcsolnx(fxm,gxm,zx,solnx)
          TRUE
            SKIP
        wout(solnx[1])
        SEQ i=1 FOR xnodes
          v[i][j]:=solnx[i]
:
PROC solvev13()
  INT trick:
  SEQ
  INT trick:
  SEQ
    IF
      xid=xprocs
        SEQ
          IF
            singularity = 0
              SEQ
                j:=3
                zcount:=2
                trick:=0
            singularity = 1
              SEQ
                j:=1
                zcount:=1
                xrhs(dx,zcount,ze1,ze3,ze4,re3)
                zx[0]:=v[0][zcount]
                calczx0(ex0,zx0)
                j:=j+1
                calcsolnx0(fx0,gx0,zx0,solnx)
                SEQ i=sposn FOR xnodes-(sposn-1)
                  v[i][zcount]:=solnx[i]
```

```
      zcount:=zcount+1
      j:=j+1
      trick:=1
singularity = 2
  SEQ
    j:=1
    zcount:=1
    xrhs(dx,zcount,ze1,ze3,ze4,re3)
    IF
      j <= ynodes
        win(v[0][j])
      TRUE
        wout(v[1][j-ynodes])
    zx0[0]:=v[0][zcount]
    calczx(ex0,zx0)
    j:=j+1
    calcsolnx(fx0,gx0,zx0,solnx)
    SEQ i=1 FOR xnodes
      v[i][zcount]:=solnx[i]
    zcount:=zcount+1
    IF
      j <= ynodes
        win(v[0][j])
      TRUE
        wout(v[1][j-ynodes])
      j:=j+1
      trick:=0
  TRUE
    SKIP
WHILE j <= (2*ynodes)
  SEQ
    IF
      j <= (ynodes+trick)
        win(v[0][j])
      TRUE
        wout(v[1][j-ynodes])
    zx[0]:=v[0][zcount]
    IF
      zcount < (me-yposn)
        SEQ
          xrhs(dx,zcount,ze1,ze3,ze4,re3)
          calczx(exe,zx)
          j:=j+1
          calcsolnx(fxe,gxe,zx,solnx)
      zcount = (me-yposn)
        j:=j+1
      zcount < ((me+mm)-yposn)
        SEQ
          xrhs(dx,zcount,zm1,zm3,zm4,rm3)
          calczx(exm,zx)
          j:=j+1
          calcsolnx(fxm,gxm,zx,solnx)
      zcount > ((me+mm)-yposn)
        SEQ
          xrhs(dx,zcount,zs1,zs3,zs4,rs3)
          calczx(exs,zx)
          j:=j+1
          calcsolnx(fxs,gxs,zx,solnx)
      TRUE
        j:=j+1
    SEQ i=1 FOR xnodes
      v[i][zcount]:=solnx[i]
    zcount:=zcount+1
    IF
      j <= (ynodes+trick)
        win(v[0][j])
      TRUE
        wout(v[1][j-ynodes])
      j:=j+1
TRUE
  SEQ
    IF
      singularity = 0
        SKIP
      singularity = 1
        SEQ
          SEQ j=1 FOR 1
            SEQ
              xrhs(dx,j,ze1,ze3,ze4,re3)
              calczx0(ex0,zx0)
              eastout ! real ;  zx0[xnodes]
              SEQ i=sposn FOR xnodes-(sposn-1)
                v[i][j]:=zx0[i]
      singularity = 2
        SEQ
          SEQ j=1 FOR 1
            SEQ
              xrhs(dx,j,ze1,ze3,ze4,re3)
```

```
            westin ? CASE  real ;  zx0[0]
            calczx(ex0,zx0)
            eastout ! real ;  zx0[xnodes]
            SEQ i=1 FOR xnodes
              v[i][j]:=zx0[i]
      TRUE
        SKIP
    SEQ j=2 FOR ynodes-1
      SEQ
        win(zx[0])
        IF
          j < (me-yposn)
            SEQ
              xrhs(dx,j,ze1,ze3,ze4,re3)
              calczx(exe,zx)
          j = (me-yposn)
            SKIP
          j < ((me+mm)-yposn)
            SEQ
              xrhs(dx,j,zm1,zm3,zm4,rm3)
              calczx(exm,zx)
          j > ((me+mm)-yposn)
            SEQ
              xrhs(dx,j,zs1,zs3,zs4,rs3)
              calczx(exs,zx)
          TRUE
            SKIP
        eastout ! real ;  zx[xnodes]
        SEQ i=1 FOR xnodes
          v[i][j]:=zx[i]
    IF
      singularity = 0
        SKIP
      singularity = 1
        SEQ
          SEQ j=1 FOR 1
            SEQ
              SEQ i=sposn FOR xnodes-(sposn-1)
                zx0[i]:=v[i][j]
              eastin ? CASE  real ;  solnx[xnodes+1]
              calcsolnx0(fx0,gx0,zx0,solnx)
              SEQ i=sposn FOR xnodes-(sposn-1)
                v[i][j]:=solnx[i]
      singularity = 2
        SEQ
          SEQ j=1 FOR 1
            SEQ
              SEQ i=1 FOR xnodes
                zx0[i]:=v[i][j]
              eastin ? CASE  real ;  solnx[xnodes+1]
              calcsolnx(fx0,gx0,zx0,solnx)
              westout ! real ;  solnx[1]
              SEQ i=1 FOR xnodes
                v[i][j]:=solnx[i]
      TRUE
        SKIP
    SEQ j=2 FOR ynodes-1
      SEQ
        SEQ i=1 FOR xnodes
          zx[i]:=v[i][j]
        eastin ? CASE  real ;  solnx[xnodes+1]
        IF
          j < (me-yposn)
            SEQ
              calcsolnx(fxe,gxe,zx,solnx)
          j = (me-yposn)
            SKIP
          j < ((me+mm)-yposn)
            SEQ
              calcsolnx(fxm,gxm,zx,solnx)
          j > ((me+mm)-yposn)
            SEQ
              calcsolnx(fxs,gxs,zx,solnx)
          TRUE
            SKIP
        wout(solnx[1])
        SEQ i=1 FOR xnodes
          v[i][j]:=solnx[i]
:
PROC solvevgt11layer([xnodes+2]REAL64 ex,fx,gx,REAL64 z1,z3,z4,r3)
  SEQ
    IF
      xid=xprocs
        SEQ
          j:=1
          zcount:=1
          WHILE j <= (2*ynodes)
            SEQ
```

```
          xrhs(dx,zcount,z1,z3,z4,r3)
          IF
            j <= ynodes
              win(v[0][j])
            TRUE
              wout(v[1][j-ynodes])
          zx[0]:=v[0][zcount]
          calczx(ex,zx)
          j:=j+1
          calcsolnx(fx,gx,zx,solnx)
          SEQ i=1 FOR xnodes
            v[i][zcount]:=solnx[i]
          zcount:=zcount+1
          IF
            j <= ynodes
              win(v[0][j])
            TRUE
              wout(v[1][j-ynodes])
          j:=j+1
TRUE
  SEQ
    SEQ j=1 FOR ynodes
      SEQ
        xrhs(dx,j,z1,z3,z4,r3)
        SEQ
          xrhs(dx,j,z1,z3,z4,r3)
          win(zx[0])
          calczx(ex,zx)
          eastout ! real ;  zx[xnodes]
          SEQ i=1 FOR xnodes
            v[i][j]:=zx[i]
        SEQ j=1 FOR ynodes
          SEQ
            SEQ i=1 FOR xnodes
              zx[i]:=v[i][j]
            eastin ? CASE  real ;  solnx[xnodes+1]
            calcsolnx(fx,gx,zx,solnx)
            wout(solnx[1])
            SEQ i=1 FOR xnodes
              v[i][j]:=solnx[i]
:
PROC solvevgt12layer([xnodes+2]REAL64 ex1,fx1,gx1,ex2,fx2,gx2,
          REAL64 z11,z13,z14,r13,z21,z23,z24,r23,rat1,rat2,VAL INT posn)
  SEQ
    IF
      xid=xprocs
```

```
      SEQ
        j:=1
        zcount:=1
        WHILE j <= (2*ynodes)
          SEQ
            IF
              j <= ynodes
                win(v[0][j])
              TRUE
                wout(v[1][j-ynodes])
            zx[0]:=v[0][zcount]
            IF
              zcount < posn
                SEQ
                  xrhs(dx,zcount,z11,z13,z14,r13)
                  calczx(ex1,zx)
                  j:=j+1
                  calcsolnx(fx1,gx1,zx,solnx)
              zcount > posn
                SEQ
                  xrhs(dx,zcount,z21,z23,z24,r23)
                  calczx(ex2,zx)
                  j:=j+1
                  calcsolnx(fx2,gx2,zx,solnx)
              TRUE
                j:=j+1
            SEQ i=1 FOR xnodes
              v[i][zcount]:=solnx[i]
            zcount:=zcount+1
            IF
              j <= ynodes
                win(v[0][j])
              TRUE
                wout(v[1][j-ynodes])
            j:=j+1
    TRUE
      SEQ
        SEQ j=1 FOR ynodes
          SEQ
            win(zx[0])
            IF
              j < posn
                SEQ
                  xrhs(dx,j,z11,z13,z14,r13)
```

```
            calczx(ex1,zx)
          j > posn
            SEQ
              xrhs(dx,j,z21,z23,z24,r23)
              calczx(ex2,zx)
          TRUE
            SKIP
        eastout ! real ;  zx[xnodes]
        SEQ i=1 FOR xnodes
          v[i][j]:=zx[i]
    SEQ j=1 FOR ynodes
      SEQ
        SEQ i=1 FOR xnodes
          zx[i]:=v[i][j]
        eastin ? CASE  real ;  solnx[xnodes+1]
        IF
          j < posn
            SEQ
              calcsolnx(fx1,gx1,zx,solnx)
          j > posn
            SEQ
              calcsolnx(fx2,gx2,zx,solnx)
          TRUE
            SKIP
        wout(solnx[1])
        SEQ i=1 FOR xnodes
          v[i][j]:=solnx[i]
:
PROC solvevgt13layer()
  SEQ
    IF
      xid=xprocs
        SEQ
          j:=1
          zcount:=1
          WHILE j <= (2*ynodes)
            SEQ
              IF
                j <= ynodes
                  win(v[0][j])
                TRUE
                  wout(v[1][j-ynodes])
              zx[0]:=v[0][zcount]
              IF
                zcount < (me-yposn)
                  SEQ
                    xrhs(dx,zcount,ze1,ze3,ze4,re3)
                    calczx(exe,zx)
                    j:=j+1
                    calcsolnx(fxe,gxe,zx,solnx)
                zcount = (me-yposn)
                  j:=j+1
                zcount < ((me+mm)-yposn)
                  SEQ
                    xrhs(dx,zcount,zm1,zm3,zm4,rm3)
                    calczx(exm,zx)
                    j:=j+1
                    calcsolnx(fxm,gxm,zx,solnx)
                zcount > ((me+mm)-yposn)
                  SEQ
                    xrhs(dx,zcount,zs1,zs3,zs4,rs3)
                    calczx(exs,zx)
                    j:=j+1
                    calcsolnx(fxs,gxs,zx,solnx)
                TRUE
                  j:=j+1
              SEQ i=1 FOR xnodes
                v[i][zcount]:=solnx[i]
              zcount:=zcount+1
              IF
                j <= ynodes
                  win(v[0][j])
                TRUE
                  wout(v[1][j-ynodes])
              j:=j+1
      TRUE
        SEQ
          SEQ j=1 FOR ynodes
            SEQ
              win(zx[0])
              IF
                j < (me-yposn)
                  SEQ
                    xrhs(dx,j,ze1,ze3,ze4,re3)
                    calczx(exe,zx)
                j = (me-yposn)
                  SKIP
                j < ((me+mm)-yposn)
```

```
            SEQ
              xrhs(dx,j,zm1,zm3,zm4,rm3)
              calczx(exm,zx)
          j > ((me+mm)-yposn)
            SEQ
              xrhs(dx,j,zs1,zs3,zs4,rs3)
              calczx(exs,zx)
          TRUE
            SKIP
        eastout ! real ;  zx[xnodes]
        SEQ i=1 FOR xnodes
          v[i][j]:=zx[i]
    SEQ j=1 FOR ynodes
      SEQ
        SEQ i=1 FOR xnodes
          zx[i]:=v[i][j]
        eastin ? CASE  real ;  solnx[xnodes+1]
        IF
          j < (me-yposn)
            SEQ
              calcsolnx(fxe,gxe,zx,solnx)
          j = (me-yposn)
            SKIP
          j < ((me+mm)-yposn)
            SEQ
              calcsolnx(fxm,gxm,zx,solnx)
          j > ((me+mm)-yposn)
            SEQ
              calcsolnx(fxs,gxs,zx,solnx)
          TRUE
            SKIP
        wout(solnx[1])
        SEQ i=1 FOR xnodes
          v[i][j]:=solnx[i]
:
REAL64 FUNCTION ux (VAL INT i,VAL REAL64 pp,t)
  REAL64 result,p1,p2,p3,c1,c2,c3,p:
  VALOF
    SEQ
      p:=pp/rc
      p1:=DPOWER(p,0.5(REAL64))
      c1:=DCOS(t/2.0(REAL64))
      p2:=DPOWER(p,1.5(REAL64))
      c2:=DCOS((3.0(REAL64)*t)/2.0(REAL64))
      p3:=DPOWER(p,2.5(REAL64))
      c3:=DCOS((5.0(REAL64)*t)/2.0(REAL64))
      IF
        i = 1
          result:=((p1*c1)-(p2*((c1+c2)/4.0(REAL64))))+
          (p3*(((14.0(REAL64)*c1)+(9.0(REAL64)*(c2+c3)))/96.0(REAL64)))
        i=2
          result:=(p2*c2)-(p3*((c1+c3)/4.0(REAL64)))
        i=3
          result:=p3*c3
        i=4
          result:=p3*c1
        TRUE
          SKIP
    RESULT result
:
REAL64 FUNCTION quad (VAL REAL64 rr,a1,b1,c1)
  REAL64 r2,r3,r4,result:
  VALOF
    SEQ
      r2:=DPOWER(rr,2.0(REAL64))
      r3:=DPOWER(rr,3.0(REAL64))
      r4:=DPOWER(rr,4.0(REAL64))
      result:=(((a1/4.0(REAL64))*r4)+((b1/3.0(REAL64))*r3))+
      ((c1/2.0(REAL64))*r2)
    RESULT result
:
PROC pseudoinv([fpt][npt]REAL64 a)
  [fpt][npt]REAL64 r:
  [fpt][fpt]REAL64 q:
  [npt]REAL64 d:
  [npt][npt]REAL64 rinv:
  REAL64 sum,t,r1:
  INT i,j,k,l,m,n,kk:
  SEQ
    --overwrite a with r and the householder reflection vectors
    k:=-1
    m:=fpt
    n:=npt
    SEQ l=0 FOR n
      SEQ
        k:=k+1
        sum:=0.0(REAL64)
        SEQ i=k FOR m-k
```

```
    sum:=sum+(a[i][l]*a[i][l])
  IF
    sum = 0.0(REAL64)
      d[l]:=0.0(REAL64)
    TRUE
      SEQ
        sum:=DSQRT(sum)
        t:=a[k][l]
        r1:=1.0(REAL64)/(DSQRT(sum*(sum+DABS(t))))
        IF
          t < 0.0(REAL64)
            sum:=-sum
          TRUE
            SKIP
        d[l]:=-sum
        a[k][k]:=r1*(t+sum)
        SEQ i=k+1 FOR m-(k+1)
          a[i][k]:=r1*a[i][l]
        SEQ j=l+1 FOR n-(l+1)
          SEQ
            t:=0.0(REAL64)
            SEQ i=k FOR m-k
              t:=t+(a[i][k]*a[i][j])
            SEQ i=k FOR m-k
              a[i][j]:=a[i][j]-(t*a[i][k])
--calculate q from householder vectors
SEQ j=0 FOR n
  SEQ
    SEQ i=0 FOR m
      q[i][j]:=0.0(REAL64)
    q[j][j]:=1.0(REAL64)
    SEQ kk=2 FOR j+1
      SEQ
        l:=(j+2)-kk
        t:=0.0(REAL64)
        SEQ i=l FOR m-l
          t:=t+(a[i][l]*q[i][j])
        SEQ i=l FOR m-l
          q[i][j]:=q[i][j]-(t*a[i][l])
--retrieve r from a and d
SEQ i=0 FOR m
  SEQ
    SEQ j=0 FOR n
      SEQ
        r[i][j]:=0.0(REAL64)
        IF
          j > i
            r[i][j]:=a[i][j]
          TRUE
            SKIP
SEQ i=0 FOR n
  r[i][i]:=d[i]
--calculate rinv from r
SEQ i=0 FOR n
  SEQ j=0 FOR n
    rinv[i][j]:=0.0(REAL64)
rinv[0][0]:=1.0(REAL64)/r[0][0]
SEQ j=1 FOR n-1
  SEQ
    rinv[j][j]:=1.0(REAL64)/r[j][j]
    SEQ kk=2 FOR j
      SEQ
        i:=(j+1)-kk
        sum:=0.0(REAL64)
        SEQ k=i+1 FOR j-i
          sum:=sum-(r[i][k]*rinv[k][j])
        rinv[i][j]:=sum/r[i][i]
  --calculate pseudo inverse (=rinv*q)
  SEQ i=0 FOR n
    SEQ j=0 FOR m
      SEQ
        sum:=0.0(REAL64)
        SEQ k=0 FOR n
          SEQ
            sum:=sum+(rinv[i][k]*q[j][k])
        finv[i][j]:=sum
:
PROC singcoeff()
  [fpt+1]REAL64 px,tx:
  REAL64 p,t,r,z,dz:
  INT k,l,temp:
  SEQ
    k:=kpt
    l:=lpt
    dz:=dze
    --set up matrix F of far points
    SEQ i=1 FOR k
      SEQ
```

```
      r:=((REAL64 ROUND l) + 0.5(REAL64))*dr
      z:=(REAL64 ROUND i)*dz
      px[i]:=DSQRT((r*r)+(z*z))
      tx[i]:=pi-DATAN(z/r)
    SEQ i=k+1 FOR 2*(l+1)
      SEQ
        temp:=i-(k+1)
        r:=((REAL64 ROUND temp)-1.5(REAL64))*dr
        temp:=k+1
        z:=(REAL64 ROUND temp)*dz
        px[i]:=DSQRT((r*r)+(z*z))
        IF
          i < ((k+1)+2)
            tx[i]:=pi-DATAN((DABS(z/r)))
          TRUE
            tx[i]:=DATAN(z/r)
    SEQ i=(fpt-k) FOR k+1
      SEQ
        r:=((REAL64 ROUND l)+0.5(REAL64))*dr
        z:=(REAL64 ROUND (fpt-i))*dz
        px[i]:=DSQRT((r*r)+(z*z))
        tx[i]:=DATAN(z/r)
    SEQ i=1 FOR fpt
      SEQ j=1 FOR npt
        f[i-1][j-1]:=ux(j,px[i],tx[i])
    pseudoinv(f)
    --use pseudo-inverse to calculate coeff
    SEQ i=0 FOR 2*l
      SEQ j=0 FOR k+1
        SEQ
          IF
            i < l
              SEQ
                r:=((REAL64 ROUND (l-i))-0.5(REAL64))*dr
                z:=(REAL64 ROUND j)*dz
                p:=DSQRT((r*r)+(z*z))
                t:=pi-DATAN(z/r)
            TRUE
              SEQ
                r:=((REAL64 ROUND (i-l))+0.5(REAL64))*dr
                z:=(REAL64 ROUND j)*dz
                p:=DSQRT((r*r)+(z*z))
                t:=DATAN(z/r)
          SEQ k1=0 FOR fpt
            SEQ
              coeff[i][j][k1]:=0.0(REAL64)
              SEQ k2=0 FOR npt
                coeff[i][j][k1]:=coeff[i][j][k1]+(ux(k2+1,p,t)*finv[k2][k1])
:
PROC singcorr([xnodes+2][ynodes+2]REAL64 v)
  INT k,l:
  [fpt]REAL64 pt:
  SEQ
    k:=kpt
    l:=lpt
    --pick out appropriate far points
    SEQ i=0 FOR k
      pt[i]:=v[sposn-(l+1)][i+2]
    SEQ i=0 FOR 2*(l+1)
      pt[i+k]:=v[(sposn+i)-(l+1)][k+2]
    SEQ i=0 FOR (k+1)
      pt[(i+k)+(2*(l+1))]:=v[sposn+l][(k+1)-i]
    --overwrite solution at near points with series solution
    SEQ i=1 FOR 2*l
      SEQ j=1 FOR k+1
        IF
          i < (l+1)
            IF
              j = 1
                SKIP
              TRUE
                SEQ
                  v[sposn-((l+1)-i)][j]:=0.0(REAL64)
                  SEQ k2=0 FOR fpt
                    v[sposn-((l+1)-i)][j]:=v[sposn-((l+1)-i)][j]+
                    (coeff[i-1][j-1][k2]*pt[k2])
          TRUE
            SEQ
              v[sposn+(i-(l+1))][j]:=0.0(REAL64)
              SEQ k2=0 FOR fpt
                v[sposn+(i-(l+1))][j]:=v[sposn+(i-(l+1))][j]+
                (coeff[i-1][j-1][k2]*pt[k2])
:
PROC currinterp()
  REAL64 const,sum,temp1,temp2,r0,r1,r2,itemp:
  [nr]REAL64 phi:
  SEQ
    SEQ i=0 FOR nr
```

```
    phi[i]:=((4.0(REAL64)*u[i][2])-u[i][3])/(2.0(REAL64)*dze)
  const:=(dr*dr)*(pi)
  IF
    xid = 1
      SEQ
        box[0]:=(phi[0]*const)/4.0(REAL64)
        current:=box[0]
        SEQ i=1 FOR (is-1)/2
          SEQ
            temp1:=((phi[2*i]-(2.0(REAL64)*phi[(2*i)-1]))+phi[(2*i)-2])/
            ((2.0(REAL64)*dr)*dr)
            temp2:=(phi[(2*i)-1]-phi[(2*i)-2])/dr
            itemp:=REAL64 ROUND(2*i)
            r0:=(itemp-1.5(REAL64))/dr
            r1:=(itemp-0.5(REAL64))/dr
            r2:=(itemp+0.5(REAL64))/dr
            a1:=temp1
            b1:=temp2-((r0+r1)*temp1)
            c1:=(((r0*r1)*temp1)-(r0*temp2))+phi[(2*i)-2]
            current:=current+box[i]
        eastin ? CASE real; sum
        current:=current+sum
      TRUE
        SEQ
          current:=0.0(REAL64)
          SEQ i=0 FOR xnodes
            SEQ
              box[i]:=((((REAL64 ROUND (i+xposn))-(1.0(REAL64)/
              6.0(REAL64)))*u[i][2])+
              (((REAL64 ROUND (i+xposn))+(1.0(REAL64)/6.0(REAL64)))*
              u[i+1][2]))*const
              current:=current+box[i]
          eastin ? CASE real; sum
          current:=current+sum
          westout ! real; current
:
PROC calcc()
  INT k,l:
  [fpt]REAL64 pt:
  SEQ
    k:=kpt
    l:=lpt
    --pick out appropriate far points
    SEQ i=0 FOR k
      pt[i]:=u[sposn-(l+1)][i+2]
    SEQ i=0 FOR 2*(l+1)
      pt[i+k]:=u[(sposn+i)-(l+1)][k+2]
    SEQ i=0 FOR (k+1)
      pt[(i+k)+(2*(l+1))]:=u[sposn+l][(k+1)-i]
    --use pseudo-inverse to calculate c
    SEQ i=0 FOR npt
      SEQ
        c[i]:=0.0(REAL64)
        SEQ j=0 FOR fpt
          c[i]:=c[i]+(finv[i][j]*pt[j])
:
REAL64 FUNCTION q (VAL REAL64 pp)
  REAL64 result,p1,p2,p3,p4,p5,p:
  VALOF
    SEQ
      p:=pp
      p1:=DSQRT(p)
      p2:=DPOWER(p,0.5(REAL64))
      p3:=DPOWER(p,1.5(REAL64))
      p4:=DPOWER(p,2.5(REAL64))
      p5:=DPOWER((p/rc),2.0(REAL64))
      result:=(1.0(REAL64)-(p/(6.0(REAL64)*rc)))-(p5/30.0(REAL64))
      result:=result*(p1*c[0])
      result:=result+((c[1]*p3)*((0.3(REAL64)*p)-1.0(REAL64)))
      result:=result+(p4*(c[2]+(c[3]/5.0(REAL64))))
    RESULT result
:
PROC currsing()
  REAL64 temp1,temp2,r0,r1,r2,jtemp:
  [nr]REAL64 phi:
  INT itemp:
  REAL64 const,temp,sixth,sum:
  SEQ
    current:=0.0(REAL64)
    const:=(dr*dr)*(pi)
    sixth:=(1.0(REAL64)/6.0(REAL64))
    IF
      xid = 1
        SEQ
          SEQ i=1 FOR nr
            phi[i-1]:=((4.0(REAL64)*u[i][2])-u[i][3])/(2.0(REAL64)*dze)
          const:=(dr*dr)*(pi)
          IF
```

```
        xid = 1
          SEQ
            box[0]:=(phi[0]*const)/4.0(REAL64)
            current:=box[0]
            SEQ i=1 FOR (is-1)/2
              SEQ
                temp1:=((phi[2*i]-(2.0(REAL64)*phi[(2*i)-1]))+
                phi[(2*i)-2])/((2.0(REAL64)*dr)*dr)
                temp2:=(phi[(2*i)-1]-phi[(2*i)-2])/dr
                jtemp:=REAL64 ROUND(2*i)
                r0:=(jtemp-1.5(REAL64))*dr
                r1:=(jtemp-0.5(REAL64))*dr
                r2:=(jtemp+0.5(REAL64))*dr
                a1:=temp1
                b1:=temp2-((r0+r1)*temp1)
                c1:=(((r0*r1)*temp1)-(r0*temp2))+phi[(2*i)-2]
                box[i]:=(quad(r2,a1,b1,c1)-quad(r0,a1,b1,c1))*
                (2.0(REAL64)*pi)
                current:=current+box[i]
    TRUE
      SEQ
        current:=0.0(REAL64)
        SEQ i=0 FOR is-xposn
          SEQ
            itemp:=i+xposn
            temp:=(REAL64 ROUND itemp)
            box[i]:=(((temp-sixth)*u[i][2])+
            ((temp+sixth)*u[i+1][2]))*const
            current:=current+box[i]
    IF
      ((nr-1)-is) <= 0
        SKIP
      TRUE
        SEQ
          SEQ i=(is-xposn) FOR ((nr-1)-is)
            SEQ
              itemp:=(nr-1)-i
              temp:=(REAL64 ROUND itemp)
              box[i]:=(SQRT(rc)*(q((temp+0.5(REAL64))*dr)-
              q((temp-0.5(REAL64))*dr)))*(2.0(REAL64)*pi)
              current:=current+box[i]
    box[nr-(xposn+1)]:=(SQRT(rc)*(2.0(REAL64)*pi))*(q(0.5(REAL64)*dr)-
    q(0.0(REAL64)))
    current:=current+box[(nr-xposn)-1]
        IF
          xid = 1
            SKIP
          TRUE
            SEQ
              westout ! real; current
    :
    PROC currsing()
      INT itemp:
      REAL64 const,temp,sixth,sum:
      SEQ
        const:=(dr*dr)*(pi/dze)
        sixth:=(1.0(REAL64)/6.0(REAL64))
        IF
          xid = 1
            SEQ
              box[0]:=(u[1][2]*const)/4.0(REAL64)
              current:=box[0]
              SEQ i=1 FOR is-1
                SEQ
                  temp:=(REAL64 ROUND i)
                  box[i]:=(((temp-sixth)*u[i][2])+
                  ((temp+sixth)*u[i+1][2]))*const
                  current:=current+box[i]
          TRUE
            SEQ
              current:=0.0(REAL64)
              SEQ i=0 FOR is-xposn
                SEQ
                  itemp:=i+xposn
                  temp:=(REAL64 ROUND itemp)
                  box[i]:=(((temp-sixth)*u[i][2])+
                  ((temp+sixth)*u[i+1][2]))*const
                  current:=current+box[i]
        IF
          ((nr-1)-is) <= 0
            SKIP
          TRUE
            SEQ
              SEQ i=(is-xposn) FOR ((nr-1)-is)
                SEQ
                  itemp:=(nr-1)-i
                  temp:=(REAL64 ROUND itemp)
                  box[i]:=(SQRT(rc)*(q((temp+0.5(REAL64))*dr)-
```

```
            q((temp-0.5(REAL64))*dr)))*(2.0(REAL64)*pi)
            current:=current+box[i]
    box[nr-(xposn+1)]:=(SQRT(rc)*(2.0(REAL64)*pi))*(q(0.5(REAL64)*dr)-
    q(0.0(REAL64)))
    current:=current+box[(nr-xposn)-1]
    IF
      xid = 1
        SKIP
      TRUE
        SEQ
          westout ! real; current
:
SEQ
  clock ?  start
  initialise()
  setup()
  bcon(u)
  bcon(v)
  IF
    membrane = 1
      --solve in electrolyte
      IF
        singularity = 0     -- sub mesh above electrode
          SEQ
            xsetup(exe,fxe,gxe,re1,re2,re3)
            IF
              yid=1
                SEQ
                  ysetup0(ey0,fy0,gy0)
                  SEQ
                    SEQ k=1 FOR onsteps
                      SEQ
                        bcon(u)
                        solvev11()
                        halo(v)
                        bcon(v)
                        solveforu01()
                        halo(u)
                  --currinterp()
                  singid:=nr/xnodes
                  westout ! array;u
                  SEQ i=1 FOR singid-(xid-1)
                    SEQ
                      eastin ? CASE array;u
                      westout ! array;u
              TRUE
                SEQ
                  ysetup(ey0,fy0,gy0)
                  SEQ
                    SEQ k=1 FOR onsteps
                      SEQ
                        bcon(u)
                        solvevgt11layer(exe,fxe,gxe,ze1,ze3,ze4,re3)
                        halo(v)
                        bcon(v)
                        solveforu0gt1()
                        halo(u)
        singularity = 1     -- sub mesh above singularity
          SEQ
            xsetup(exe,fxe,gxe,re1,re2,re3)
            IF
              yid=1
                SEQ
                  xsetup0(ex0,fx0,gx0)
                  ysetup0(ey0,fy0,gy0)
                  ysetup(ey,fy,gy)
                  SEQ
                    singcoeff()
                    clock ? now
                    SEQ k=1 FOR onsteps
                      SEQ
                        bcon(u)
                        solvev11()
                        halo(v)
                        bcon(v)
                        solveforu11()
                        singcorr(u)
                        halo(u)
                  --calcc()
                  --currsing()
                  westout ! array;u
              TRUE
                SEQ
                  ysetup(ey0,fy0,gy0)
                  ysetup(ey,fy,gy)
                  SEQ
                    SEQ k=1 FOR onsteps
                      SEQ
```

```
                bcon(u)
                solvevgt1llayer(exe,fxe,gxe,ze1,ze3,ze4,re3)
                halo(v)
                bcon(v)
                solveforulgt1()
                halo(u)
singularity = 2    -- sub mesh above boundary
  SEQ
    xsetup(exe,fxe,gxe,re1,re2,re3)
    IF
      yid=1
        SEQ
          xsetup(ex0,fx0,gx0,re1,re2,re3)
          ysetup(ey,fy,gy)
          SEQ
            SEQ k=1 FOR onsteps
              SEQ
                bcon(u)
                solvev11()
                halo(v)
                bcon(v)
                solveforu21()
                halo(u)
      TRUE
        SEQ
          ysetup(ey,fy,gy)
          SEQ
            SEQ k=1 FOR onsteps
              SEQ
                bcon(u)
                solvevgt11layer(exe,fxe,gxe,ze1,ze3,ze4,re3)
                halo(v)
                bcon(v)
                solveforu2gt1()
                halo(u)
    TRUE
      SKIP
membrane = 2
  --solve in electrolyte/membrane
  IF
    singularity = 0    -- sub mesh above electrode
      SEQ
        xsetup(exe,fxe,gxe,re1,re2,re3)
        xsetup(exm,fxm,gxm,rm1,rm2,rm3)
        IF
          yid=1
            SEQ
              ysetup0(ey0,fy0,gy0)
              SEQ
                SEQ k=1 FOR onsteps
                  SEQ
                    bcon(u)
                    solvev12()
                    interface(v,rate,ratm,me)
                    halo(v)
                    bcon(v)
                    solveforu01()
                    halo(u)
                --currinterp()
                westout ! array;u
                singid:=nr/xnodes
                SEQ i=1 FOR singid-(xid-1)
                  SEQ
                    eastin ? CASE array;u
                    westout ! array;u
          TRUE
            SEQ
              ysetup(ey0,fy0,gy0)
              SEQ
                SEQ k=1 FOR onsteps
                  SEQ
                    bcon(u)
                    solvevgt1layer(exe,fxe,gxe,exm,
                                   fxm,gxm,ze1,ze3,ze4,re3,
                    zm1,zm3,zm4,rm3,rate,ratm,me-yposn)
                    interface(v,rate,ratm,me)
                    halo(v)
                    bcon(v)
                    solveforu0gt1()
                    halo(u)
  singularity = 1    -- sub mesh above singularity
    SEQ
      xsetup(exe,fxe,gxe,re1,re2,re3)
      xsetup(exm,fxm,gxm,rm1,rm2,rm3)
      IF
        yid=1
          SEQ
            xsetup0(ex0,fx0,gx0)
```

```
      ysetup0(ey0,fy0,gy0)
      ysetup(ey,fy,gy)
      SEQ
        singcoeff()
        clock ? now
        SEQ k=1 FOR onsteps
          SEQ
            bcon(u)
            solvev12()
            interface(v,rate,ratm,me)
            halo(v)
            bcon(v)
            solveforu11()
            halo(u)
            singcorr(u)
      --calcc()
      --currsing()
      westout ! array;u
TRUE
  SEQ
    ysetup(ey0,fy0,gy0)
    SEQ
      ysetup(ey0,fy0,gy0)
      ysetup(ey,fy,gy)
      SEQ
        SEQ k=1 FOR onsteps
          SEQ
            bcon(u)
            solvevgt12layer(exe,fxe,gxe,exm,
                            fxm,gxm,ze1,ze3,ze4,re3,
            zm1,zm3,zm4,rm3,rate,ratm,me-yposn)
            interface(v,rate,ratm,me)
            halo(v)
            bcon(v)
            solveforu1gt1()
            halo(u)
singularity = 2     -- sub mesh above boundary
  SEQ
    xsetup(exe,fxe,gxe,re1,re2,re3)
    xsetup(exm,fxm,gxm,rm1,rm2,rm3)
    IF
      yid=1
        SEQ
          xsetup(ex0,fx0,gx0,re1,re2,re3)
          ysetup(ey,fy,gy)
          SEQ
            SEQ k=1 FOR onsteps
              SEQ
                bcon(u)
                solvev12()
                interface(v,rate,ratm,me)
                halo(v)
                bcon(v)
                solveforu21()
                halo(u)
TRUE
  SEQ
    ysetup(ey,fy,gy)
    SEQ
      SEQ k=1 FOR onsteps
        SEQ
          bcon(u)
          solvevgt2layer(exe,fxe,gxe,exm,fxm,
                         gxm,ze1,ze3,ze4,re3,
          zm1,zm3,zm4,rm3,rate,ratm,me-yposn)
          interface(v,rate,ratm,me)
          halo(v)
          bcon(v)
          solveforu2gt1()
          halo(u)
    TRUE
      SKIP
membrane = 3
  --solve in electrolyte/membrane/sample
  IF
    singularity = 0     -- sub mesh above electrode
      SEQ
        xsetup(exe,fxe,gxe,re1,re2,re3)
        xsetup(exm,fxm,gxm,rm1,rm2,rm3)
        xsetup(exs,fxs,gxs,rs1,rs2,rs3)
        IF
          yid=1
            SEQ
              ysetup0(ey0,fy0,gy0)
              SEQ
                SEQ k=1 FOR onsteps
                  SEQ
                    bcon(u)
                    solvev13()
                    interface(v,rate,ratm,me)
```

```
            interface(v,ratm,rats,(me+mm))
            halo(v)
            bcon(v)
            solveforu01()
            halo(u)
        --currinterp()
        singid:=nr/xnodes
        westout ! array;u
        SEQ i=1 FOR singid-(xid-1)
          SEQ
            eastin ? CASE array;u
            westout ! array;u
TRUE
  SEQ
    ysetup(ey0,fy0,gy0)
    SEQ
      SEQ k=1 FOR onsteps
        SEQ
          bcon(u)
          solvevgt13layer()
          interface(v,rate,ratm,me)
          interface(v,ratm,rats,(me+mm))
          halo(v)
          bcon(v)
          solveforu0gt1()
          halo(u)
singularity = 1     -- sub mesh above singularity
  SEQ
    xsetup(exe,fxe,gxe,re1,re2,re3)
    xsetup(exm,fxm,gxm,rm1,rm2,rm3)
    xsetup(exs,fxs,gxs,rs1,rs2,rs3)
    IF
      yid=1
        SEQ
          xsetup0(ex0,fx0,gx0)
          ysetup0(ey0,fy0,gy0)
          ysetup(ey,fy,gy)
          SEQ
            singcoeff()
            clock ? now
            SEQ k=1 FOR onsteps
              SEQ
                bcon(u)
                solvev13()
            interface(v,rate,ratm,me)
            interface(v,ratm,rats,(me+mm))
            halo(v)
            bcon(v)
            solveforu11()
            halo(u)
            singcorr(u)
        --calcc()
        --currsing()
        westout ! array;u
TRUE
  SEQ
    ysetup(ey0,fy0,gy0)
    ysetup(ey,fy,gy)
    SEQ
      SEQ k=1 FOR onsteps
        SEQ
          bcon(u)
          solvevgt13layer()
          interface(v,rate,ratm,me)
          interface(v,ratm,rats,(me+mm))
          halo(v)
          bcon(v)
          solveforu1gt1()
          halo(u)
singularity = 2     -- sub mesh above boundary
  SEQ
    xsetup(exe,fxe,gxe,re1,re2,re3)
    xsetup(exm,fxm,gxm,rm1,rm2,rm3)
    xsetup(exs,fxs,gxs,rs1,rs2,rs3)
    IF
      yid=1
        SEQ
          xsetup(ex0,fx0,gx0,re1,re2,re3)
          ysetup(ey,fy,gy)
          SEQ
            SEQ k=1 FOR onsteps
              SEQ
                bcon(u)
                solvev13()
                interface(v,rate,ratm,me)
                interface(v,ratm,rats,(me+mm))
                halo(v)
                bcon(v)
```

```
                    solveforu21()
                    halo(u)
            TRUE
              SEQ
                ysetup(ey,fy,gy)
                SEQ
                  SEQ k=1 FOR onsteps
                    SEQ
                      bcon(u)
                      solvevgt13layer()
                      interface(v,rate,ratm,me)
                      interface(v,ratm,rats,(me+mm))
                      halo(v)
                      bcon(v)
                      solveforu2gt1()
                      halo(u)
      TRUE
        SKIP
  membrane = 4
    --solve in membrane
    IF
      singularity = 0     -- sub mesh above electrode
        SEQ
          xsetup(exm,fxm,gxm,rm1,rm2,rm3)
          ysetup(ey0,fy0,gy0)
          SEQ
            SEQ k=1 FOR onsteps
              SEQ
                bcon(u)
                solvevgt11layer(exm,fxm,gxm,zm1,zm3,zm4,rm3)
                halo(v)
                bcon(v)
                solveforu0gt1()
                halo(u)
      singularity = 1     -- sub mesh above singularity
        SEQ
          xsetup(exm,fxm,gxm,rm1,rm2,rm3)
          SEQ
            ysetup(ey0,fy0,gy0)
            ysetup(ey,fy,gy)
            SEQ
              SEQ k=1 FOR onsteps
                SEQ
                  bcon(u)
                  solvevgt11layer(exm,fxm,gxm,zm1,zm3,zm4,rm3)
                  halo(v)
                  bcon(v)
                  solveforu1gt1()
                  halo(u)
        singularity = 2     -- sub mesh above boundary
          SEQ
            xsetup(exm,fxm,gxm,rm1,rm2,rm3)
            SEQ
              ysetup(ey,fy,gy)
              SEQ
                SEQ k=1 FOR onsteps
                  SEQ
                    bcon(u)
                    solvevgt11layer(exm,fxm,gxm,zm1,zm3,zm4,rm3)
                    halo(v)
                    bcon(v)
                    solveforu2gt1()
                    halo(u)
        TRUE
          SKIP
  membrane = 5
    --solve in membrane/sample
    IF
      singularity = 0     -- sub mesh above electrode
        SEQ
          xsetup(exm,fxm,gxm,rm1,rm2,rm3)
          xsetup(exs,fxs,gxs,rs1,rs2,rs3)
          ysetup(ey0,fy0,gy0)
          SEQ
            SEQ k=1 FOR onsteps
              SEQ
                bcon(u)
                solvevgt12layer(exm,fxm,gxm,exs,fxs,gxs,zm1,zm3,zm4,rm3,
                zs1,zs3,zs4,rs3,ratm,rats,(me+mm)-yposn)
                interface(v,ratm,rats,(me+mm))
                halo(v)
                bcon(v)
                solveforu0gt1()
                halo(u)
      singularity = 1     -- sub mesh above singularity
        SEQ
          xsetup(exm,fxm,gxm,rm1,rm2,rm3)
          xsetup(exs,fxs,gxs,rs1,rs2,rs3)
```

```
      ysetup(ey0,fy0,gy0)
      ysetup(ey,fy,gy)
      SEQ
        SEQ k=1 FOR onsteps
          SEQ
            bcon(u)
            solvevgt12layer(exm,fxm,gxm,exs,fxs,gxs,zm1,zm3,zm4,rm3,
            zs1,zs3,zs4,rs3,ratm,rats,(me+mm)-yposn)
            interface(v,ratm,rats,(me+mm))
            halo(v)
            bcon(v)
            solveforu1gt1()
            halo(u)
singularity = 2     -- sub mesh above boundary
  SEQ
    xsetup(exm,fxm,gxm,rm1,rm2,rm3)
    xsetup(exs,fxs,gxs,rs1,rs2,rs3)
    ysetup(ey,fy,gy)
    SEQ
      SEQ k=1 FOR onsteps
        SEQ
          bcon(u)
          SEQ
            bcon(u)
            solvevgt12layer(exm,fxm,gxm,exs,fxs,gxs,zm1,zm3,zm4,rm3,
            zs1,zs3,zs4,rs3,ratm,rats,(me+mm)-yposn)
            interface(v,ratm,rats,(me+mm))
            halo(v)
            bcon(v)
            solveforu2gt1()
            halo(u)
    TRUE
      SKIP
membrane = 6
  --solve in sample
  IF
    singularity = 0     -- sub mesh above electrode
      SEQ
        xsetup(exs,fxs,gxs,rs1,rs2,rs3)
        ysetup(ey0,fy0,gy0)
        SEQ
          SEQ k=1 FOR onsteps
            SEQ
              bcon(u)
              solvevgt11layer(exs,fxs,gxs,zs1,zs3,zs4,rs3)
              halo(v)
              bcon(v)
              solveforu1gt1()
              halo(u)
singularity = 1     -- sub mesh above singularity
  SEQ
    xsetup(exs,fxs,gxs,rs1,rs2,rs3)
    SEQ
      ysetup(ey0,fy0,gy0)
      ysetup(ey,fy,gy)
      SEQ
        SEQ k=1 FOR onsteps
          SEQ
            bcon(u)
            solvevgt11layer(exs,fxs,gxs,zs1,zs3,zs4,rs3)
            halo(v)
            bcon(v)
            solveforu1gt1()
            halo(u)
singularity = 2     -- sub mesh above boundary
  SEQ
    xsetup(exs,fxs,gxs,rs1,rs2,rs3)
    SEQ
      xsetup(exs,fxs,gxs,rs1,rs2,rs3)
      SEQ
        ysetup(ey,fy,gy)
        SEQ
          SEQ k=1 FOR onsteps
            SEQ
              bcon(u)
              solvevgt11layer(exs,fxs,gxs,zs1,zs3,zs4,rs3)
              halo(v)
              bcon(v)
              solveforu2gt1()
              halo(u)
      TRUE
        SKIP
  TRUE
    SKIP
clock ? finish
time:=finish-start
IF
  xid = 1
    IF
      yid = 1
        westout ! cs ; c
      TRUE
```

```
        SKIP
    TRUE
      SKIP
  IF
    xid = 1
      IF
        yid = 1
          westout ! xvector1 ; box
        TRUE
          SKIP
    TRUE
      SKIP
  IF
    xid = 1
      IF
        yid = 1
          westout ! real ; current
        TRUE
          SKIP
    TRUE
      SKIP
  IF
    SKIP
  IF
    printflag = 1
      SEQ
        SEQ i=0 FOR xnodes+2
          box[i]:=u[i][2]
        westout ! xvector1 ; box
        IF
          xid = 1
            IF
              yid = 1
                westout ! array; u
              TRUE
                SKIP
          TRUE
            SKIP
        IF
          yid=1
            SEQ
              IF
                xid=1
                  SEQ
                    northout ! array ;  u
                    SEQ i=1 FOR (yprocs-1)
                      SEQ
                        southin  ? CASE  array ;  u
                        northout ! array ;  u
                    SEQ i=1 FOR (yprocs*(xprocs-1))
                      SEQ
                        eastin   ? CASE  array ;  u
                        northout ! array ;  u
              TRUE
                SEQ
                  westout ! array ;  u
                  SEQ i=1 FOR (yprocs-1)
                    SEQ
                      southin ? CASE  array ;  u
                      westout ! array ;  u
                  IF
                    xid=xprocs
                      SKIP
                    TRUE
                      SEQ i=1 FOR ((xprocs-xid)*yprocs)
                        SEQ
                          eastin ? CASE  array ;  u
                          westout ! array ;  u
                      TRUE
                        SEQ
                          northout ! array ;  u
                          SEQ i=1 FOR (yprocs-yid)
                            SEQ
                              southin ? CASE  array ;  u
                              northout ! array ;  u
          TRUE
            SKIP
        IF
          xid = 1
            IF
              yid = 1
                SEQ
                  corrtime:=now-start
                  westout ! int ; corrtime
                  westout ! int ; time
              TRUE
                SKIP
          TRUE
            SKIP
:
```

Appendix C

The Fortran Code to Solve the Clark Electrode Problem

```
c
c       Program to solve diffusion for a membrane electrode using ADI
c
      parameter(Mpe=100,Mpm=200,Mps=200,Mp=500,
     +Np=500,Npr=200,ipoff=20000,ipon=10000)
      parameter(npt=4,fpt=30,lpt=20,kpt=20)
c
c     Declare common blocks
c
      common/block a/dr,dz,dt,ion,ioff,l,k,is
      common/block r/rc
      common/block d/ c
      common/block b/nure,nurm,nurs,nuze,m,n,me,mm,ms
     +,nuzm,nuzs,pe,pm,ps,dze,dzm,dzs,iflag,ton,toff,nr
      common/decomp/e0on,f0on,g0on,e0off,f0off,g0off,ee,fe,ge,
     +em,fm,gm,es,fs,gs,ezon,fzon,gzon,ezoff,fzoff,gzoff
      common/consts/cft1,cft2,cm,cne,cnm,cns,
     +rnue2,rnum2,rnus2,rate,ratm,rats,ratem,ratms
      common/diff/de,dm,ds
c
c     Declare Variables
c
      real u(-1:Np,-1:Mp),v(-1:np,-1:mp),
     +nure,nurm,nurs,nuze,nuzm,nuzs,current(ipon),
     +cft1(0:np),cft2(0:np),cm,cne,cnm,cns
      real e0on(0:np),f0on(0:np),g0on(0:np),
     + e0off(0:np),f0off(0:np),g0off(0:np),ee(0:np),fe(0:np)
     + ,ge(0:np),em(0:np),fm(0:np),gm(0:np),es(0:np),
     + fs(0:np),gs(0:np),ezon(0:np),fzon(0:np),gzon(0:np),
     + ezoff(0:np),fzoff(0:np),gzoff(0:np)
c
c     Declare vbles for sing correction
c
      real coeff(-lpt:lpt,0:kpt,fpt),
     +c(npt),pt(fpt),t1(0:np),test(0:10000)
      real*8 finv(npt,fpt)
c
c     Open data files
c
      open(1,file='adires.dat',status='new')
      open(2,file='diff.dat',status='old')
      open(3,file='graph.dat',status='new')
c
c     Read in parameter values
c
      read(2,*) de,dm,ds,pe,pm,ps,ze,zm,zs,m,n,nr,me,mm,ms,ton,toff,
     +iflag1,rc,tf,dt,nprint,l,k,is
      close(2)
c
c     Calculate number of timesteps and mesh spacings
c
      nt=int(tf/dt)
      ion=int(ton/dt)
      ioff=int(toff/dt)
      dr=rc/Nr
      dze=ze/me
      dzm=zm/mm
      dzs=zs/ms
c
c     Set up and decompose tridiag systems
c
      call setup
c
c     Set up singularity correction
c
      dz=dze
      mpt=nr-1
      call motz(finv,coeff)
c
c     Set initial values of u and v. u is soln at n, v at n+1.
c
      do 10 i=-1,N,1
      do 20 j=-1,M,1
       u(i,j)=1.
       v(i,j)=1.
  20  continue
  10  continue
c
c     Start time loop with electrode on (iflag=2)
c
      icount=0
      iflag=2
c
c     Time loop
c
      do 50 it=1,nt
        icount=icount+1
        if (iflag.eq.1) call bdataoff(u)
```

```
      if (iflag.eq.2) call bdataon(u)
      call solveforv(u,v)
      if (iflag.eq.1) call bdataoff(v)
      if (iflag.eq.2) call bdataon(v)
      call solveforu(u,v)
c
c     Implement sing correction
c
      call crank(u,coeff,nr-1)
      call calcc(u,finv,c)
c
c     Calculate current
c
      call interp(u,nr-1,curr)
      current(it)=curr
c
c     Write results at end of on--pulse and
c     switch to iflag=1 so electrode is off.
c
      if (icount.eq.ion) then
        call reswrite(u)
        iflag=1
      endif
c
c     End of off--pulse.
c     Switch to iflag=2 so that electrode is on
c
      if (icount.eq.ion+ioff) then
          iflag=2
          icount=0
      endif
c
c     End time loop
c
 50   continue
      write(3,*)ion,dt,(current(i),i=1,ion)
      close(1)
      stop
      end


c
c     Subroutine sets up the tridiagonal systems in each
c     layer and in both directions, and decomposes them
c     into LU form by calling decompose.
c
      subroutine setup
      parameter(Mpe=100,Mpm=200,Mps=200,Mp=500,
     +Np=500,Npr=200,ipoff=5000,ipon=2000)
      common/decomp/e0on,f0on,g0on,e0off,f0off,g0off,ee,fe,ge,
     +em,fm,gm,es,fs,gs,ezon,fzon,gzon,ezoff,fzoff,gzoff
      common/block a/dr,dz,dt,ion,ioff,l,k,is
      common/block b/nure,nurm,nurs,nuze,m,n,me,mm,ms
     +,nuzm,nuzs,pe,pm,ps,dze,dzm,dzs,iflag,ton,toff,nr
      common/consts/cft1,cft2,cm,cne,cnm,cns,
     +rnue2,rnum2,rnus2,rate,ratm,rats,ratem,ratms
      common/diff/de,dm,ds
      real u(-1:Np,-1:Mp),v(-1:np,-1:mp),
     +nure,nurm,nurs,nuze,nuzm,nuzs,current(ipon),
     +cft1(0:np),cft2(0:np),cm,cne,cnm,cns
       real a(0:np),b(0:np),c(0:np),e0on(0:np),f0on(0:np)
     + ,g0on(0:np),e0off(0:np),f0off(0:np),g0off(0:np),
     + ee(0:np),fe(0:np),ge(0:np),em(0:np),fm(0:np),
     + gm(0:np),es(0:np),fs(0:np),gs(0:np),ezon(0:np),
     + fzon(0:np),gzon(0:np),ezoff(0:np),fzoff(0:np),gzoff(0:np)
c
c      Calculate dr,dt and nu-values from input parameters
c
      pi=4.0*atan(1.0)
      nuze=de*dt/dze/dze
      nuzm=dm*dt/dzm/dzm
      nuzs=ds*dt/dzs/dzs
      nure=de*dt/dr/dr
      nurm=dm*dt/dr/dr
      nurs=ds*dt/dr/dr
c
c     Calculate f.d. ceoffts in r-dirn
c
      do 14 i=0,n
        cft2(i)=i/(2.*i+1.)
        cft1(i)=(i+1.)/(2.*i+1.)
 14   continue
c
c       electrode off-set up  tridiag mx on j=0 separately
c
        do 21 i=nr,n-1
          a(i)=-nure*cft2(i)
          b(i)=1+nure
```

```
          c(i)=-nure*cft1(i)
 21       continue
c
c         decompose  into LU
c
          call decompose(a,b,c,e0on,f0on,g0on,nr,n)
c
          do 22 i=0,n-1
            a(i)=-nure*cft2(i)
            b(i)=1+nure
            c(i)=-nure*cft1(i)
 22       continue
c
c          implement  l.h. b.c. and  decompose
c
          b(0)=b(0)+a(0)
c
          call decompose(a,b,c,e0off,f0off,g0off,0,n)

c
c       Set up tridiag system in electrolyte
c
          do 20 i=0,n-1
            a(i)=-nure*cft2(i)
            b(i)=1+nure
            c(i)=-nure*cft1(i)
 20       continue
c
c          implement  l.h. b.c. and  decompose
c
          b(0)=b(0)+a(0)
c
          call decompose(a,b,c,ee,fe,ge,0,n)
c
c          cne, cnm, cns are use for r.h. b.c. when calculating r.h.s.
c
          cne=c(n-1)
c
c       Set up tridiag system in membrane
c
          do 50 i=0,n-1
            a(i)=-nurm*cft2(i)
            b(i)=1+nurm
            c(i)=-nurm*cft1(i)
 50       continue

c
c         implement  l.h. b.c. and decompose
c
          b(0)=b(0)+a(0)
          call decompose(a,b,c,em,fm,gm,0,n)
          cnm=c(n-1)
c
c       Set up tridiag system in sample
c
          do 80 i=0,n-1
            a(i)=-nurs*cft2(i)
            b(i)=1+nurs
            c(i)=-nurs*cft1(i)
 80       continue
c
c          implement  l.h. b.c and  decompose
c
          b(0)=b(0)+a(0)
          call decompose(a,b,c,es,fs,gs,0,n)
          cns=c(n-1)
c
c       Calculate f.d. ceoffts in z-dirn
c
          rnue2=nuze/2.
          rnum2=nuzm/2.
          rnus2=nuzs/2.
          rate=pe/dze
          ratm=pm/dzm
          rats=ps/dzs
          ratem=rate+ratm
          ratms=ratm+rats
c
c       electrolyte
c
          do 10 j=0,me-1
            a(j)=-nuze/2.
            b(j)=1.+nuze
            c(j)=-nuze/2.
 10       continue
c
c       membrane
```

```
c
      do 24 j=me+1,me+mm-1
        a(j)=-nuzm/2.
        b(j)=1.+nuzm
        c(j)=-nuzm/2.
 24   continue
c
c     sample
c
      do 30 j=me+mm+1,m-1
        a(j)=-nuzs/2.
        b(j)=1.+nuzs
        c(j)=-nuzs/2.
 30   continue
c
c      Eqtns at interfaces from b.c.'s (x(dt/dze) as conditioning)
c
      a(me)=-pe/dze*(dt/dze)
      b(me)=(pe/dze+pm/dzm)*(dt/dze)
      c(me)=-pm/dzm*(dt/dze)
      a(me+mm)=-pm/dzm*(dt/dze)
      b(me+mm)=(pm/dzm+ps/dzs)*(dt/dze)
      c(me+mm)=-ps/dzs*(dt/dze)
c
c      if electrode on,set up eqtns above electrode (nj=1) and away
c      from electrode (nj=0) to satisfy b.c.'s
c
c     decompose into LU. cm is used to implement bc on r=rmax
c     when calculating r.h.s.
c
        nj=1
        call decompose(a,b,c,ezon,fzon,gzon,nj,m)
        cm=c(m-1)
c
c        b.c. away from electrode
c
        c(0)=c(0)+a(0)
c
c        decompose into LU
c
        nj=0
        call decompose(a,b,c,ezoff,fzoff,gzoff,nj,m)
        return
        end
c
c      Subroutine decompose, decomposes a tridiag matrix
c      into LU form.   1,e give L, f,g give U
c
      subroutine decompose(a,b,c,e,f,g,n1,nn)
      parameter(Mpe=100,Mpm=200,Mps=200,Mp=500,
     +Np=500,Npr=200,ipoff=5000,ipon=2000)
      real a(0:np),b(0:np),c(0:np),e(0:np),f(0:np),g(0:np)
      f(n1)=b(n1)
      do 10 i=n1+1,nn-1
        e(i)=a(i)/f(i-1)
        f(i)=b(i)-e(i)*c(i-1)
        g(i-1)=c(i-1)
  10  continue
      return
      end
c
c      Subroutine bdataon imposes b.c.'s on u or v when the electrode is on
c
      subroutine bdataon(c)
      parameter(Mpe=100,Mpm=200,Mps=200,Mp=500,
     +Np=500,Npr=200,ipoff=5000,ipon=2000)
      common/block b/nure,nurm,nurs,nuze,m,n,me,mm,ms
     +,nuzm,nuzs,pe,pm,ps,dze,dzm,dzs,iflag,ton,toff,nr
      real c(-1:np,-1:mp)
c
c   set no-flow-thro condn away from electrode
c
      do 70 i=Nr,N,1
       c(i,-1)=c(i,1)
  70  continue
c
c      set cathode conc. to zero
c
      do 31 i=0,Nr-1,1
       do 41 j=-1,0,1
        c(i,j)=0.
  41   continue
  31  continue
c
```

```
c   set l.h. symm condition
c
      do 60 j=-1,M
       c(-1,j)=c(0,j)
 60   continue
      return
      end

c
c     Subroutine bdataoff imposes b.c.'s on u or v when the electrode is off
c
      subroutine bdataoff(c)
     parameter(Mpe=100,Mpm=200,Mps=200,Mp=500,
     +Np=500,Npr=200,ipoff=5000,ipon=2000)
     common/block b/nure,nurm,nurs,nuze,m,n,me,mm,ms
     +,nuzm,nuzs,pe,pm,ps,dze,dzm,dzs,iflag,ton,toff,nr
      real c(-1:np,-1:mp)
c
c      Set no-flow-thro' condn. along electrode bdy. when electrode off
c
      do 71 i=0,N-1,1
       do 72 j=-1,0,1
        c(i,j)=c(i,1)
 72    continue
 71   continue
c
c   set l.h. symm condition
c
      do 61 j=-1,M,1
       c(-1,j)=c(0,j)
 61   continue
      return
      end

c
c     Subroutine solveforv sets up a tridiagonal system for v in the r-dirn.
c     for each j.The system is of the form
c
c              a(i).v(i-1,j)+b(i).v(i,j)+c(i).v(i+1,j)=d(i) , j=0..n-1
c
c     with b(i)>0,a(i),c(i)<or=0,and b(i)>or=abs(a(i)+c(i)) for all i.

c
```

```
      subroutine solveforv(u,v)
      parameter(Mpe=100,Mpm=200,Mps=200,Mp=500,
     +Np=500,Npr=200,ipoff=5000,ipon=2000)
      common/block a/dr,dz,dt,ion,ioff,l,k,is
      common/block b/nure,nurm,nurs,nuze,m,n,me,mm,ms
     +,nuzm,nuzs,pe,pm,ps,dze,dzm,dzs,iflag,ton,toff,nr
       common/decomp/e0on,f0on,g0on,e0off,f0off,g0off,ee,fe,ge,
     +  em,fm,gm,es,fs,gs,ezon,fzon,gzon,ezoff,fzoff,gzoff
      common/consts/cft1,cft2,cm,cne,cnm,cns,
     +rnue2,rnum2,rnus2,rate,ratm,rats,ratem,ratms
      real e0on(0:np),f0on(0:np),g0on(0:np),
     + e0off(0:np),f0off(0:np),g0off(0:np),ee(0:np),fe(0:np)
     + ,ge(0:np),em(0:np),fm(0:np),gm(0:np),es(0:np),
     + fs(0:np),gs(0:np),ezon(0:np),fzon(0:np),gzon(0:np),
     + ezoff(0:np),fzoff(0:np),gzoff(0:np)
      real a(0:np),b(0:np),c(0:np),d(0:np),v(-1:np,-1:mp)
     +,u(-1:np,-1:mp),soln(0:np),nure,nurm,nurs,nuze
     +,nuzm,nuzs,
     +cft1(0:np),cft2(0:np),cm,cne,cnm,cns
c
       if (iflag.eq.1) goto 111
c
c       electrode on-set up eqtns on j=0 separately
c
      j=0
       do 21 i=nr,n-1
         d(i)=u(i,j)+rnue2*(u(i,j+1)-2*u(i,j)+u(i,j-1))
 21    continue
c
c        implement r.h. b.c. and call tridiag solver
c
       d(n-1)=d(n-1)-cne*v(n,0)
       call ludec(d,e0on,f0on,g0on,soln,nr,n)
c
c        Overwrite v(i,j) at n-1/2 with value at n+1/2
c
       do 31 i=nr,n
         v(i,j)=soln(i)
 31    continue
       goto 112
c
c        electrode off-set up eqtns on j=0 separately
c
111    j=0
```

```
      do 22 i=0,n-1
        d(i)=u(i,j)+rnue2*(u(i,j+1)-2*u(i,j)+u(i,j-1))
 22   continue
c
c      implement r.h. and l.h. b.c.'s and call tridiag solver
c
      d(n-1)=d(n-1)-cne*v(n,0)
      call ludec(d,e0off,f0off,g0off,soln,0,n)
c
c      Overwrite v(i,j) at n-1/2 with value at n+1/2
c
      do 32 i=0,n-1
        v(i,j)=soln(i)
 32   continue
c
c     Set up tridiag system in electrolyte
c
112   do 10 j=1,me-1
        do 20 i=0,n-1
          d(i)=u(i,j)+rnue2*(u(i,j+1)-2.*u(i,j)+u(i,j-1))
 20     continue
c
c        implement r.h. and l.h. b.c.'s and call tridiag solver
c
        d(n-1)=d(n-1)-cne*v(n,j)
        call ludec(d,ee,fe,ge,soln,0,n)
c
c        Overwrite v(i,j) at n-1/2 with value at n+1/2
c
        do 30 i=0,n-1
          v(i,j)=soln(i)
 30     continue
 10   continue
c
c     Set up tridiag system in membrane
c
      do 40 j=me+1,me+mm-1
        do 50 i=0,n-1
          d(i)=u(i,j)+rnum2*(u(i,j+1)-2*u(i,j)+u(i,j-1))
 50     continue

c
c        implement r.h. and l.h. b.c.'s and call tridiag solver
c
        d(n-1)=d(n-1)-cnm*v(n,j)
        call ludec(d,em,fm,gm,soln,0,n)
c
c        Overwrite v(i,j) at n-1/2 with value at n+1/2
c
        do 60 i=0,n-1
          v(i,j)=soln(i)
 60     continue
 40   continue
c
c      Set up tridiag system in sample
c
      do 70 j=me+mm+1,m-1
        do 80 i=0,n-1
          d(i)=u(i,j)+rnus2*(u(i,j+1)-2*u(i,j)+u(i,j-1))
 80     continue
c
c        implement r.h. and l.h. b.c.'s and call tridiag solver
c
        d(n-1)=d(n-1)-cns*v(n,j)
        call ludec(d,es,fs,gs,soln,0,n)
c
c        Overwrite v(i,j) at n-1/2 with value at n+1/2
c
        do 90 i=0,n-1
          v(i,j)=soln(i)
 90     continue
 70   continue
c
c      impose b.c. at electrolyte/membrane interface
c
      do 100 i=0,n-1
        v(i,me)=(ratm*v(i,me+1)+rate*
     +  v(i,me-1))/ratem
100   continue
c
c      impose b.c. at membrane/sample interface
c
      do 110 i=0,n-1
        v(i,me+mm)=(ratm*v(i,me+mm-1)+rats
     +  *v(i,me+mm+1))/ratms
110   continue
      return
      end
```

```
c
c      Subroutine solveforu sets up a tridiagonal system for u in the z-dirn.
c      for each i.The system is of the form
c
c              a(j).v(i,j-1)+b(j).v(i,j)+c(j).v(i,j)=d(j) , j=0..n-1
c
c      with b(j)>0,a(j),c(j)<or=0,and b(j)>or=abs(a(j)+c(j)) for all j.

c
      subroutine solveforu(u,v)
      parameter(Mpe=100,Mpm=200,Mps=200,Mp=500,
     +Np=500,Npr=200,ipoff=5000,ipon=2000)
      common/block a/dr,dz,dt,ion,ioff,l,k,is
      common/block b/nure,nurm,nurs,nuze,m,n,me,mm,ms
     +,nuzm,nuzs,pe,pm,ps,dze,dzm,dzs,iflag,ton,toff,nr
        common/decomp/e0on,f0on,g0on,e0off,f0off,g0off,ee,fe,ge,
     +  em,fm,gm,es,fs,gs,ezon,fzon,gzon,ezoff,fzoff,gzoff
      common/consts/cft1,cft2,cm,cne,cnm,cns,
     +rnue2,rnum2,rnus2,rate,ratm,rats,ratem,ratms
      real e0on(0:np),f0on(0:np),g0on(0:np),
     + e0off(0:np),f0off(0:np),g0off(0:np),ee(0:np),fe(0:np)
     + ,ge(0:np),em(0:np),fm(0:np),gm(0:np),es(0:np),
     + fs(0:np),gs(0:np),ezon(0:np),fzon(0:np),gzon(0:np),
     + ezoff(0:np),fzoff(0:np),gzoff(0:np)
      real a(0:np),b(0:np),c(0:np),d(0:np),v(-1:np,-1:mp)
     +,u(-1:np,-1:mp),soln(0:np),nure,nurm,nurs,nuze
     +,nuzm,nuzs,
     +cft1(0:np),cft2(0:np),cm,cne,cnm,cns
      if (iflag.eq.1) goto 111
c
c      if electrode on,set up eqtns above electrode (nj=1) and away
c      from electrode (nj=0) to satisfy b.c.'s
c
c        b.c. away from electrode
c
c
c        set up rhs of tridiag system in all layers
c
       do 40 i=0,nr-1
        do 50 j=1,me-1
          d(j)=v(i,j)+nure*(cft1(i)*v(i+1,j)-v(i,j)+
     +          cft2(i)*v(i-1,j))
 50     continue
        do 60 j=me+1,me+mm-1
          d(j)=v(i,j)+nurm*(cft1(i)*v(i+1,j)-v(i,j)+
     +          cft2(i)*v(i-1,j))
 60     continue
        do 70 j=me+mm+1,m-1
          d(j)=v(i,j)+nurs*(cft1(i)*v(i+1,j)-v(i,j)+
     +          cft2(i)*v(i-1,j))
 70     continue
c
c        impose b.c. 'at infinity' and at interfaces, call tridiag solver
c
        d(m-1)=d(m-1)-cm*u(i,m)
        d(me)=0.
        d(me+mm)=0.
       call ludec(d,ezon,fzon,gzon,soln,1,m)
        do 80 j=1,m-1
          u(i,j)=soln(j)
 80     continue
 40    continue
       do 42 i=nr,n-1
c
c        b.c. away from electrode
c
c
c        set up rhs of tridiag system in all layers
c
        do 52 j=0,me-1
          d(j)=v(i,j)+nure*(cft1(i)*v(i+1,j)-v(i,j)+
     +          cft2(i)*v(i-1,j))
 52     continue
        do 62 j=me+1,me+mm-1
          d(j)=v(i,j)+nurm*(cft1(i)*v(i+1,j)-v(i,j)+
     +          cft2(i)*v(i-1,j))
 62     continue
        do 72 j=me+mm+1,m-1
          d(j)=v(i,j)+nurs*(cft1(i)*v(i+1,j)-v(i,j)+
     +          cft2(i)*v(i-1,j))
 72     continue
c
c        impose b.c. 'at infinity' and at interfaces, call tridiag solver
c
        d(m-1)=d(m-1)-cm*u(i,m)
        d(me)=0.
```

```
       d(me+mm)=0.
      call ludec(d,ezoff,fzoff,gzoff,soln,0,m)
       do 82 j=0,m-1
         u(i,j)=soln(j)
 82    continue
 42   continue
      goto 112
c
c      electrode off, impose no-flow-thro' along z=0 boundary
c
c
c      set up rhs of tridiag system in all layers
c
111   do 41 i=0,n-1
        do 51 j=0,me-1
          d(j)=v(i,j)+nure*(cft1(i)*v(i+1,j)-v(i,j)+
     +          cft2(i)*v(i-1,j))
 51     continue
        do 61 j=me+1,me+mm-1
          d(j)=v(i,j)+nurm*(cft1(i)*v(i+1,j)-v(i,j)+
     +          cft2(i)*v(i-1,j))
 61     continue
        do 71 j=me+mm+1,m-1
          d(j)=v(i,j)+nurs*(cft1(i)*v(i+1,j)-v(i,j)+
     +          cft2(i)*v(i-1,j))
 71     continue
c
c       impose b.c. 'at infinity' and at interfaces, call tridiag solver
c
        d(m-1)=d(m-1)-cm*u(i,m)
        d(me)=0.
        d(me+mm)=0.
        call ludec(d,ezoff,fzoff,gzoff,soln,0,m)
c
c       Overwrite u(i,j) at n with value at n+1
c
        do 81 j=0,m-1
          u(i,j)=soln(j)
 81     continue
 41   continue
112   return
      end

c
```

```
c      subroutine ludec solves a tridiag system in LU form
c      by forwards and backwards substitution
c

      subroutine ludec(d,e,f,g,soln,n1,nn)
      parameter(Mpe=100,Mpm=200,Mps=200,Mp=500,
     +Np=500,Npr=200,ipoff=5000,ipon=2000)
      real a(0:np),b(0:np),c(0:np),d(0:np),v(-1:np,-1:mp)
     +,u(-1:np,-1:mp),soln(0:np),e(0:np),f(0:np),g(0:np),z(0:np),y(0:np)
c
c      forward subst. using L to solve Uz=d for z
c
      soln(nn)=1.
      z(n1)=d(n1)
      do 20 i=n1+1,nn-1
        z(i)=d(i)-e(i)*z(i-1)
 20   continue
c
c      backward subst. using U to solve L.soln=z for soln
c
      soln(nn-1)=z(nn-1)/f(nn-1)
      do 30 i=1,nn-1-n1
        soln(nn-1-i)=(z(nn-1-i)-g(nn-1-i)*soln(nn-i))/f(nn-1-i)
 30   continue
      return
      end


c
c     Subroutine Motz finds the pseudo--inverse of a matrix
c     using NAg routine F01 BLF, and then sets up the
c     appropriate vectors for the singularity correction
c
      subroutine motz(finv,coeff)
      parameter(npt=4,fpt=30,kpt=20,lpt=20)
      real*8 px(fpt),tx(fpt),f(fpt,npt),finv(npt,fpt),
     +wkspce(npt),t
      real coeff(-lpt:lpt,0:kpt,fpt)
      integer inc(fpt)
      real*8 fijmax(npt),d(fpt),u(npt,npt),du(npt),x02aaf
      common/block a/dr,dz,dt,ion,ioff,l,k,is
      common/block r/rc
      external f01blf,ux
      n=npt
      m=2*k+2*l+3
```

```
      if=fpt
      iu=npt
      ifail=0
      pi=4.*atan(1.0)
c
c     Calculate rho, theta (px,tx) coords of each far point
c
      do 20 i=1,k
        px(i)=sqrt(((l+0.5)*dr)**2+((i)*dz)**2)
        tx(i)=pi-atan((i)*dz/((l+0.5)*dr))
20    continue
      do 21 i=k+1,k+2*l+2
        px(i)=sqrt(((i-(k+l+1.5))*dr)**2+((k+1)*dz)**2)
        if (i.le.k+l+1) then
          tx(i)=pi-atan((k+1)*dz/(abs(i-(k+l+1.5))*dr))
        else
          tx(i)=atan((k+1)*dz/((i-(k+l+1.5))*dr))
        endif
21    continue
      do 22 i=k+2*l+3,2*k+2*l+3
        px(i)=sqrt(((l+0.5)*dr)**2+((2*k+2*l+3-i)*dz)**2)
        tx(i)=atan((2*k+2*l+3-i)*dz/((l+0.5)*dr))
22    continue
      do 15 ix=1,npt
        do 25 jx=1,2*l+2*k+3
         f(jx,ix)=ux(ix,px(jx),tx(jx))
 25     continue
 15   continue
      eps=x02aaf(xxxx)
      t=10.*eps
      irank=npt
c
c     Find pseudo--inverse
c
      call f01blf(m,n,t,f,if,fijmax,irank,
     +inc,d,u,iu,du,ifail)
      do 30 i=1,npt
        do 30 j=1,2*l+2*k+3
           finv(i,j)=f(j,i)
 30   continue
c
c     Calculate multipliers of far points to calculate values of
c     near points at each tstep, and store in coeff
c
      do 101 i=-l+1,l,1
        do 102 j=0,k
          p=sqrt(((i-0.5)*dr)**2+(j*dz)**2)
          if (i.le.0) then
            t=pi-atan(-j*dz/((i-0.5)*dr))
          else
            t=atan(j*dz/((i-0.5)*dr))
          endif
          do 103 k1=1,2*l+2*k+3
            coeff(i,j,k1)=0.0
            do 104 k2=1,npt
              coeff(i,j,k1)=coeff(i,j,k1)+ux(k2,p,t)*finv(k2,k1)
104         continue
103       continue
c          print*,(coeff(i,j,ii),ii=1,2*l+2*k+3)
102     continue
101   continue
      return
      end

c
c     Subroutine Crank implements the singularity
c     correction at each timestep
c
      subroutine crank(v,coeff,m)
      parameter(np=500,mp=500)
      parameter(npt=4,fpt=30,lpt=20,kpt=20)
      real v(-1:np,-1:mp),coeff(-lpt:lpt,0:kpt,fpt),
     +pt(fpt)
      common/block a/dr,dz,dt,ion,ioff,l,k,is
      common/block r/rc
c
c     Assign appropriate far point values into pt
c
      do 200 i=1,k
        pt(i)=v(m-l,i)
200   continue
      do 210 i=k+1,k+2*l+2
        pt(i)=v(m-l-(k+1-i),k+1)
210   continue
      do 220 i=k+2*l+3,2*k+2*l+3
        pt(i)=v(m+l+1,2*k+2*l+3-i)
220   continue
c
```

```
c     Overwrite values at each near point using coeff
c     and values at far points.
c
      do 81 i=-l+1,l
        do 82 j=0,k
          if (j.eq.0.and.i.le.0) goto 82
          v(m+i,j)=0.
          do 83 k2=1,2*l+2*k+3
            v(m+i,j)=v(m+i,j)+coeff(i,j,k2)*pt(k2)
 83       continue
 82     continue
 81   continue
      return
      end

c
c     Function ux has the form of the trucated
c     series solution.
c
      function ux(i,p,t)
      common/block a/dr,dz,dt,ion,ioff,l,k,is
      common/block r/rc
      p=p/rc
      if (i.eq.1) ux=p**(.5)*cos(t/2)-p**(1.5)*(cos(t/2)+
     +cos(3*t/2.))/4.+p**(2.5)*
     +(14*cos(t/2.)+9*cos(3*t/2.)+9*cos(5*t/2.))/96.
      if (i.eq.2) ux=(p**(1.5)*cos(3*t/2.)-
     +p**(2.5)*(cos(t/2)+cos(5*t/2.))/4.)
      if (i.eq.3) ux=((p**(2.5))*cos(5*t/2.))
      if (i.eq.4) ux=((p**(2.5))*cos(t/2.))
      p=rc*p
      return
      end

c
c     subroutine calcc calculates the time dependent
c     coeffts, c(t), of the truncated series expansion
c
      subroutine calcc(u,finv,c)
      parameter(Mpe=100,Mpm=200,Mps=200,Mp=500,
     +Np=500,Npr=200,ipoff=20000,ipon=10000)
      parameter(npt=4,fpt=30,lpt=20,kpt=20)
      real u(-1:Np,-1:Mp),c(npt),pt(fpt)
      real*8 finv(npt,fpt)
      do 200 i=1,k
        pt(i)=u(mpt-l,i)
200   continue
      do 210 i=k+1,k+2*l+2
        pt(i)=u(mpt-l-(k+1-i),k+1)
210   continue
      do 220 i=k+2*l+3,2*k+2*l+3
        pt(i)=u(mpt+l+1,2*k+2*l+3-i)
220   continue
      do 130 i=1,npt
        c(i)=0.0
        do 140 j=1,2*l+2*k+3
          c(i)=c(i)+finv(i,j)*pt(j)
140     continue
130   continue
      return
      end

c
c
c     subroutine interp sets up a local interpolant for the partial
c     pressure, including a transformation to polar coords to deal
c     with the singularity at rc. Then uses this interpolant to
c     calculate current.
c
      subroutine interp(u,m,curr)
      parameter(np=500,mp=500)
      real u(-1:np,-1:mp),c(4),phi(0:np)
      common/quad/ al,bl,cl
      common/block d/ c
      common/block a/dr,dz,dt,ion,ioff,l,k,is
      common/block r/rc
c
c     Calculate 2nd order approx to dp/dz
c
      do 5 i=0,m+1
        phi(i)=(4.0*u(i,1)-u(i,2))/(2.0*dz)
 5    continue
      pi=4*atan(1.0)
      nr=m
      sum=0.
      sum1=0.
      sum2=0.
      sum3=0.
      sum1=phi(0)/4.*dr*dr*pi
```

```
      il=0

c
c     Fit a quadratic using divided diffs through dp/dz values.
c     Integrate this to obtain current into interval.
c
      do 10 i=1,int(is/2.)
        temp1=(phi(2*i)-2*phi(2*i-1)+phi(2*i-2))/2./dr/dr
        temp2=(phi(2*i-1)-phi(2*i-2))/dr
        r0=(2*i-1.5)*dr
        r1=(2*i-.5)*dr
        r2=(2*i+.5)*dr
        a1=temp1
        b1=temp2-(r0+r1)*temp1
        c1=r0*r1*temp1-r0*temp2+phi(2*i-2)
        add=(quad(r2)-quad(r0))*2*pi
        sum2=sum2+add
  10  continue
c
c     Use series expn to calc current in vicinity of sing.
c
      do 20 i=is,nr
      add=sqrt(rc)*(q((nr-i+1.5)*dr)-q((nr-i+0.5)*dr))*2*pi
      sum3=sum3+add
  20  continue
      add=sqrt(rc)*(q(0.5*dr)-q(0.0))*2*pi
      sum3=sum3+add
      sum=(sum1+sum2)+sum3
      curr=sum/pi/rc/rc*4.*2.69e-9*96485.0
      return
      end

c
c     Function quad integrates r*(a quadratic in r)
c
      function quad(r)
      common/quad/ a1,b1,c1
      quad=a1/4.0*r**4+b1/3.0*r**3+c1/2.0*r**2
      return
      end

c
c     Function Q calcs contribution to current near sing.
c
      function Q(p)
      real c(4)
      common/block a/dr,dz,dt,ion,ioff,l,k,is
      common/block r/rc
      common/block d/ c
      q=c(1)*sqrt(p)*(1-p/6./rc-((p/rc)**2)/30.)+
     +c(2)/rc*p**1.5*(3*p/10./rc-1.)+(c(3)+c(4)/5.)/rc/rc*p**2.5
      return
      end

      subroutine reswrite(u)
      parameter(Mpe=100,Mpm=200,Mps=200,Mp=500,
     +Np=500,Npr=200,ipoff=20000,ipon=10000)
      parameter(npt=4,fpt=30,lpt=20,kpt=20)
      common/block a/dr,dz,dt,ion,ioff,l,k,is
      common/block r/rc
      common/block d/ c
      common/block b/nure,nurm,nurs,nuze,m,n,me,mm,ms
     +,nuzm,nuzs,pe,pm,ps,dze,dzm,dzs,iflag,ton,toff,nr
      common/decomp/e0on,f0on,g0on,e0off,f0off,g0off,ee,fe,ge,
     +em,fm,gm,es,fs,gs,ezon,fzon,gzon,ezoff,fzoff,gzoff
      common/consts/cft1,cft2,cm,cne,cnm,cns,
     +rnue2,rnum2,rnus2,rate,ratm,rats,ratem,ratms
c
c     Declare Variables
c
      real u(-1:Np,-1:Mp)
      real nure,nurm,nurs,nuze,nuzm,nuzs,current(ipon),
     +cft1(0:np),cft2(0:np),cm,cne,cnm,cns
      real e0on(0:np),f0on(0:np),g0on(0:np),
     + e0off(0:np),f0off(0:np),g0off(0:np),ee(0:np),fe(0:np)
     + ,ge(0:np),em(0:np),fm(0:np),gm(0:np),es(0:np),
     + fs(0:np),gs(0:np),ezon(0:np),fzon(0:np),gzon(0:np),
     + ezoff(0:np),fzoff(0:np),gzoff(0:np)
c
c         Write results
c
          write(1,*) '  Results using time-dependent Crank to'
          write(1,*) '  solve for flux in electrode problem'
          write(1,3)n,m,dt,it*dt,pe*dt*(1/dr/dr
     +    +1/dze/dze),rc
    3     format(/,3x,'Solved on a ',i4,' by ',i4,' mesh ',/,3x,
     +    'Using a timestep of ',f7.4,' at time ',f6.3,/,3x,'tau=',
     +    f6.3,3x,'Electrode radius =',e13.5,//)
```

```
      write(1,4) ze,zm,zs,nr,dr*(n+.5)
 4    format(3x,'Electrolyte thickness = ',e13.5,/,3x,
     +          'Membrane thickness    = ',e13.5,/,3x,
     +          'Sample thickness      = ',e13.5,//,3x,
     +          'Mesh pts covering electrode = ',i4,/,3x,
     +          'Total r-length = ',e13.5)
      write(1,21) l,k,ion*frac*dt,is,nr
21    format(//,3x,'Corrections made on a ',i3,' by',i3,
     +' submesh for times > ',e13.5,/,3x,'and for boxes',
     +i4,' to',i4,' in calculating the flux')
      write (1,22) nprint
22    format(/,3x,'Solution (x 1000) printed every ',i4,' pts',/)
      do 10 i=1,n,10
         write(1,5) (int(u(i,j)*1000),j=1,m,10)
10    continue
 5    format(1x,100i4)
      return
      end
```

Bibliography

[1] M. Aitkin, D. Anderson, B. Francis, and J. Hinde. *Statistical Modelling in GLIM.* Oxford Science Publications, OUP, Oxford, 1989.

[2] K. Aoki and J. Osteryoung. Diffusion controlled current at the stationary finite disk electrode. Experimental. *J. Electroanal. Chem.*, 125:315–320, 1981.

[3] K. Aoki and J. Osteryoung. Diffusion controlled current at the stationary finite disk electrode. Theory. *J. Electroanal. Chem.*, 122:19–35, 1981.

[4] P. W. Atkins. *Physical Chemistry 3rd ed.* OUP, Oxford, 1986.

[5] J. P. Baumberger. Determination of the oxygen dissociation curve of oxyhemoglobin by a new method. *Am. J. Physiol.*, 123:10, 1938.

[6] G. E. Bell and J. Crank. A method of treating boundary singularities in time dependent problems. *J. Inst. Maths Applics*, 12:37–48, 1973.

[7] A. M. Bond, K. B. Oldham, and C. G. Zoski. Theory of electrochemical processes at inlaid disk microelectrodes under steady–state conditions. *J. Electroanal. Chem.*, 245:71–104, 1988.

[8] R. G. Buckles, H. Heitmann, and M. B. Laver. A theoretical and practical analysis of P_{O_2} microelectrode behaviour: The three shell model. In *Proc. of a Workshop on PH and Blood Gas*, pages 203–211, NBS, Galthersburg, Maryland, 1975.

[9] S. D. Conte and C. de Boor. *Elementary Numerical Analysis 3rd. Ed.* McGraw–Hill Kogakusha, Tokyo, 1980.

[10] F. G. Cottrell. Der Reststrom bei galvanischer Polarisation betrachtet als ein Diffusionproblem. *Z. Phys. Chem.*, 42:385–431, 1903.

[11] J. Crank. *The Mathematics of Diffusion.* Clarendon Press, Oxford, 1975.

[12] J. Crank and R. M. Furzeland. The treatment of boundary singularities in axially symmetric problems containing discs. *J. Inst. Maths Applics*, 20:355–370, 1977.

[13] J. Crank and R. M. Furzeland. The numerical solution of elliptic and parabolic partial differential equations with boundary singularities. *J. Comp. Phys*, 26:285–296, 1978.

[14] J. Crank and P. Nicholson. A practical method for numerical evaluation of solutions of parallel differential equations of the heat conduction type. *Proc. Camb. Phil. Soc.*, 43:50–67, 1947.

[15] H. L. Danneel. Über den durch diffundierende Gase hevorgerufenen Reststrom. *Z. Elecktrochem.*, 4:227–242, 1897/98.

[16] P. W. Davies and F. Brink Jr. Microelectrodes for measuring local oxygen tension in tissues. *Rev. Sci. Instrum.*, 13:524–533, 1942.

[17] E. C. Dufort and S. P. Frankel. Stability conditions in the numerical treatment of parabolic differential equations. *Math. Tabl. Aids Comput.*, 7:135–152, 1953.

[18] N. T. S. Evans and A. R. Gourlay. The solution of a two–dimensional time–dependent diffusion problem concerned with oxygen metabolism in tissues. *J. Inst. Maths Applics*, 19:239–251, 1977.

[19] N. T. S. Evans and P. F. D. Naylor. The design and use of electrodes for tissue oxygen measurements. *J. Polarog. Soc.*, 2:40–45, 1960.

[20] N. T. S. Evans and P. F. D. Naylor. The measurement of the partial pressure of oxygen *in vivo*. *J. Polarog. Soc.*, 2:26–32, 1960.

[21] I. Fatt. *Polarographic Oxygen Sensors*. CRC Press, Cleveland, Ohio, 1976.

[22] E. Firouztale, M. J. Skerpon, and J. S. Ultman. A spherical model of the clark electrode in a flowing liquid medium. *J. Electroanal. Chem.*, 134:1–10, 1982.

[23] M. Fleischmann, J. Daschbach, and S. Pons. The behaviour of microdisk and microring electrodes. Mass transport to the disk in the unsteady state. Chronoamperometry. *J. Electroanal. Chem.*, 250:257–267, 1988.

[24] M. Fleischmann, J. Daschbach, and S. Pons. The behaviour of microdisk and microring electrodes. Mass transport to the disk in the unsteady state. Chronopotentiometry. *J. Electroanal. Chem.*, 250:269–276, 1988.

[25] M. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, C-21:948–960, 1972.

[26] L. Fox and R. Sankar. Boundary singularities in time-dependent problems. *J. Inst. Maths Applics*, 5:340–350, 1969.

[27] D. J. Gavaghan, C. S. Gwilliam, and J. S. Rollett. Parallelising efficient sequential codes. In *Proc. International Conference on Applications of Supercomputers in Engineering*, pages 19–36, MIT, Cambridge, Mass., 1991. Elsevier Applied Science.

[28] D. J. Gavaghan and J. S. Rollett. Correction of boundary singularities at unshielded disc electrodes. *J. Electroanal. Chem.*, 295:1–14, 1990.

[29] E. Gnaiger and H. Forstner. *Polarographic Oxygen Sensors.* Springer–Verlag, Berlin, 1983.

[30] P. Gordon. Nonsymmetric difference equations. *J. Soc. Ind. Appl. Math*, 13:667–673, 1965.

[31] A. R. Gourlay. Hopscotch: A fast second-order partial differential equation solver. *J. Inst. Maths Applics*, 6:375–390, 1970.

[32] A. R. Gourlay and G. R. McGuire. General hopscotch algorithm for the numerical solution of partial differential equations. *J. Inst. Maths Applics.*, 7:216–227, 1971.

[33] C. S. Gwilliam. A parallel algorithm for the Navier–Stokes equations in a driven cavity. Master's thesis, University of Oxford, 1990.

[34] W. W. Hager. *Applied Numerical Linear Algebra.* Prentice–Hall Internatinal, London, 1988.

[35] C. E. W. Hahn. Techniques for measuring the partial pressure of gases in the blood. Part I – *in vitro* measurements. *J. Phys. E: Sci. Instrum.*, 13:471–482, 1980.

[36] C. E. W. Hahn. Techniques for measuring the partial pressures of gases in the blood. Part II - *in vivo* measurements. *J. Phys. E: Sci. Instrum.*, 14:783–797, 1981.

[37] C. E. W. Hahn. Blood gas measurement. *Clin. Phys. Physiol. Meas.*, 8:3–38, 1987.

[38] C. E. W. Hahn, A. Davis, and W. J. Albery. Electrochemical improvement of the performance of P_{O_2} electrodes. *Respir. Physiol.*, 25:109–133, 1975.

[39] J. M. Hale and M. L. Hitchman. Some considerations of the steady–state and transient behavior of membrane–covered dissolved oxygen detectors. *J. Electroanal. Chem.*, 107:282–294, 1980.

[40] J. Heinze. Diffusion processes at finite (micro) disk electrodes solved by digital simulation. *J. Electroanal. Chem.*, 124:73–86, 1981.

[41] J. Heyrovsky. Electrolysis with the dropping mercury electrode. *Chemicke Listy*, 16:256–304, 1922.

[42] M. L. Hitchman. *Measurement of Dissolved Oxygen.* John Wiley, New York, 1978.

[43] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall International Series in Computer Science, Prentice Hall International, UK, Ltd., 1985.

[44] J. P. Hoare. *The Electrochemistry of Oxygen.* Interscience, New York, 1968.

[45] R. W. Hockney and C. R. Jesshope. *Parallel Computers 2.* I.O.P. Publishing Limited, 1988.

[46] J. H. Wilkinson *in* D. Jacobs(Ed.). *State of the Art in Numerical Analysis.* Academic Press, 1977.

[47] K. Grasshopf *in* M. Whitfield and D. Jagner (Eds.). *Marine Electrochemistry.* John Wiley, New York, 1981.

[48] O. J. Jensen, T. Jacobsen, and K. Thomsen. Membrane-covered oxygen electrodes. I. Electrode dimensions and electrode sensitivity. *J. Electroanal. Chem.*, 87:203–211, 1978.

[49] S. L. Johnsson, Y. Saad, and M. H. Schultz. Alternating direction methods on multiprocessors. Research Report YALEU/DCS/RR–382, Yale University, 1985.

[50] J. L. Morris Jr. and L. R. Faulkner. Normal pulse voltammetry in electrochemically poised systems. *Anal. Chem.*, 49:489–494, 1977.

[51] L. C. Clark Jr. Monitor and control of blood and tissue oxygen tensions. *Trans. Am. Soc. Artif. Intern. Organs*, 2:41–48, 1956.

[52] L. Ju and C. S. Ho. Measuring oxygen diffusion coefficients with polarographic oxygen electrodes: I. Electrolyte solutions. *Biotech and Bioeng.*, 27:1495–1499, 1985.

[53] M. Kakihana, H. Ikeuchi, and G. P. Satô. Diffusion current at microdisk electrodes - application to acurate measurement of diffusion coefficient. *J. Electroanal. Chem.*, 117:201–211, 1981.

[54] R. Kok and J. E. Zajic. Dynamic response of a polarographic oxygen probe. *Biotech and Bioeng.*, 17:527–539, 1975.

[55] F. Kreuzer and H. P. Kimmich. Techniques using oxygen electrodes in respiratory physiology. *Resp. Physiol.*, 406:1–29, 1984.

[56] R. W. Leland. *The effectiveness of parallel iterative algorithms for solution of large sparse linear systems.* D. Phil Thesis, University of Oxford, 1990.

[57] M. D. Lilley, J. B. Story, and R. W. Raible. The chronoamperometric determination of dissolved oxygen using membrane electrodes. *J. Electroanal. Chem.*, 23:425–429, 1969.

[58] INMOS Limited. *OCCAM 2 Reference Manual.* Prentice Hall International Series in Computer Science, Prentice Hall International, UK, Ltd., 1988.

[59] INMOS Limited. *Transputer Development System.* Prentice Hall International, UK, Ltd., 1988.

[60] INMOS Limited. *Transputer Instruction Set: a compiler writer's guide.* Prentice Hall International, UK, Ltd., 1988.

[61] INMOS Limited. *Transputer Reference Manual.* Prentice Hall International, UK, Ltd., 1988.

[62] INMOS Limited. *The Transputer Data Book.* INMOS Ltd., 2nd edition, 1989.

[63] V. Linek, V. Vacek, and J. Sinkule. The effects of the accumulation of cathodic reaction products in Clark oxygen probes on probe linearity, signal overshooting, tailing and hysteresis of transient characteristics for step changes in the concentration of oxygen. *J. Electrochem. Soc.*, 187:1–30, 1985.

[64] V. Linek, V. Vacek, J. Sinkule, and P. Benes. *Measurement of Oxygen by Membrane–Covered Probes.* Ellis Horwood, Chichester, 1988.

[65] 3L Ltd. *Parallel Fortran User Guide.* 3L Ltd., Livingston, Scotland, 1988.

[66] NAG Ltd. *The NAG Fortran Library Manual–Mark 13.* The Numerical Algorithms Group Limited, 1988.

[67] Transtech Systems Ltd. *NiCHE parallel solutions. MCP1000 Technical Reference Manual.* Transtech Systems Ltd., High Wycombe, UK, 1989.

[68] K. H. Mancy. *In situ* measurement of dissolved oxygen by pulse and steady state voltammetric membrane electrode systems. *Electrochemm. Soc Extended Abstracts*, 75:741–743, 1975.

[69] K. H. Mancy, D. A. Okun, and C. N. Reilly. A galvanic cell analyzer. *J. Electroanal. Chem.*, 4:65–92, 1962.

[70] B. Molinari. *Introduction to Computer Systems, A User-View.* Cambridge Computer Science Text 21, Press Syndicate of the University of Cambridge, New York, 1985.

[71] K. W. Morton. Numerical solution of partial differential equations. Oxford University Computing Laboratory Numerical Analysis Lecture Notes, 1986.

[72] H. Motz. The treatment of singularities of partial differential equations by relaxation methods. *Q. J. Appl. Math.*, 4:371, 1946.

[73] J. C. Myland and K. B. Oldham. Membrane-covered oxygen sensors: An exact treatment of the switch-on transient. *J. Electrochem. Soc.*, 131:1815–1823, 1984.

[74] P. F. D. Naylor and N. T. S. Evans. An electrode for measuring absolute oxygen tension in tissues. *J. Polarog. Soc.*, 2:46–48, 1960.

[75] P. F. D. Naylor and N. T. S. Evans. The measurement of oxygen tension by rapid voltage sweep polarography and bare platinum electrodes. *J. Polarog. Soc.*, 2:33–39, 1960.

[76] K. B. Oldham. Edge effects in semi-infinite diffusion. *J. Electroanal. Chem.*, 122:1–17, 1981.

[77] K. B. Oldham. Steady-state concentrations and fluxes in the vicinity of a reversible inlaid disc microelectrode. *J. Electroanal. Chem.*, 260:461–467, 1989.

[78] K. B. Oldham. The short–time chronoamperometric behaviour of an electrode of arbitrary shape. *J. Electroanal. Chem.*, 297:317–348, 1991.

[79] D. W. Peaceman and H. H. Rachford. The numerical solution of parabolic and elliptic differential equations. *J. Soc. Indust. Appl. Math.*, 3:28–41, 1955.

[80] R. D. Richtmyer and K. W. Morton. *Difference Methods for Initial Value Problems.* Interscience Publishers, New York, 1967.

[81] Y. Saito. A theoretical study of the diffusion current at the stationary electrodes of circular and narrow band type. *Rev. Polarogr. Jpn*, 15:177–187, 1968.

[82] S. H. Saulson. High speed pulsatile operation of miniature oxygen electrodes in oxygen transport to tissue. In H. I. Bicher and D. F. Bruley, editors, *Advances in Experimental Medical Biology*, pages 29–34. Plenum Press, New York, 1973.

[83] M. Von Schmid and K. H. Mancy. Die elektrochemische Bestimmung des in Wasser gelösten Sauerstoffs mit Hilfe einer modifizierten Membranelecktrodentechnik. *Schueiz. Z. Hydrol.*, 32:328–339, 1970.

[84] R. Schuler and F. Kreuzer. Rapid polarographic *in vivo* oxygen catheter electrodes. *Respiration Physiology*, 3:90–110, 1967.

[85] J. W. Severinghaus and P. J. Astrup. *History of Blood Gas Analysis*, volume 25:4 of *International Anesthesiology Clinics.* Little, Brown and Company, Boston, Mass., 1989.

[86] D. L. Short and G. S. G. Shell. Fundamentals of Clark membrane configuration oxygen sensors: Some confusion clarified. *J. Phys. E: Sci. Instrum.*, 17:1085–1092, 1984.

[87] D. L. Short and G. S. G. Shell. Pulsing amperometric oxygen sensors: Earlier techniques evaluated and a technique implemented to cancel capacitive change. *J. Phys. E. Sci. Instrum.*, 18:79–87, 1985.

[88] D. Shoup and A. Szabo. Chronoamperometric current at finite disk electrodes. *J. Electroanal. Chem.*, 140:237–245, 1982.

[89] D. Shoup and A. Szabo. Hopscotch: An algorithm for the numerical solution of electrochemical problems. *J. Electroanal. Chem.*, 160:1–17, 1984.

[90] D. Shoup and A. Szabo. Explicit, hopscotch and implicit finite difference algorithims for the Cottrell problem: Exact analytical results. *J. Electroanal. Chem.*, 199:437–441, 1986.

[91] G. D. Smith. *Numerical Solution of Parallel Differential Equations: Finite Difference Methods, 3rd Edition.* Oxford Applied Maths and Computing Science Series, OUP, Oxford, 1985.

[92] G. Taylor, H. H. Girault, and J. McAleer. Digital simulation of charge transfer to an ultramicrodisc surface. *J. Electroanal. Chem.*, 293:19–44, 1990.

[93] K. K. Tremper and S. J. Barker. *Advances in Oxygen Monitoring*, volume 25:3 of *International Anesthesiology Clinics.* Little, Brown and Company, Boston, Mass., 1989.

[94] J. S. Ultman, E. Firouztale, and M. J. Skerpon. A spherical model of the Clark electrode. *J. Electroanal. Chem.*, 127:59–66, 1981.

[95] H. H. Wang. A parallel method for tridiagonal equations. *ACM Trans. Math. Soft.*, 7:170–183, 1981.

[96] J. Wang. Evaluation of pulsed-stirring voltammetry. *Anal. Chim. Acta.*, 129:253–257, 1981.

[97] K. D. Wise, R. B. Smart, and K. H. Mancy. A transient current monitoring and electrode characterization system for a pulsed oxygen electrode. *Anal. Chim. Acta.*, 116:297–305, 1980.

[98] G. L. Zick. Determination of oxygen tension by measurement of net charge transport. *IEE Trans. Biomed. Eng.*, 23:472–477, 1976.

[99] G. L. Zick. Design of the pulsed oxygen cathode. *J. Bioeng.*, 1:173–180, 1977.