A Comparison of Parallel Gaussian Elimination Solvers for the Computation of Electrochemical Battery Models on the Cell Processor

by

James R. Geraci

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2008

Author .....
Department of Electrical Engineering and Computer Science
May 22, 2008

Certified by .....
John L. Wyatt, Jr.
Professor
Thesis Supervisor

Certified by .....
Thomas A. Keim
Principal Research Engineer
Thesis Supervisor

Accepted by .....
Terry P. Orlando
Chairman, Department Committee on Graduate Students

A Comparison of Parallel Gaussian Elimination Solvers for the Computation of Electrochemical Battery Models on the Cell Processor

by

James R. Geraci

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2008, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

The rising cost of fossil fuels, together with a push for more eco-friendly methods of transportation, has increased interest in and demand for electrically powered or assisted vehicles. The majority of these electric or hybrid electric vehicles will be, for the foreseeable future, powered by batteries.

One of the major problems with batteries is their aging. For batteries, aging means that the maximum charge they can store decreases as number of charge/discharge cycles increases. Aging also means that after a certain number of charge/discharge cycles, the battery will fail. In lead-acid batteries, one of the major phenomenon that promotes battery failure is the development of a non-uniform concentration gradient of electrolyte along the electrodes' height. This phenomenon is known as electrolyte stratification.

This thesis develops a simple two-level circuit model that can be used to model electrolyte stratification. The two-level circuit model is justified experimentally using digital Mach-Zehnder interferometry and is explained theoretically by means of two different electrochemical battery models. The experiments show how the usage of the electrode varies along its height while the simulations indicate that the high resistivity of the lead dioxide electrode plays a major role in the development of a stratified electrolyte.

Finally, computational issues associated with the computation of a sophisticated two dimensional electrochemical battery model on the multicore Cell Broadband Engine processor are addressed in detail. In particular, three different banded parallel Gaussian elimination solvers are developed and compared. These three solvers vividly illustrate how performance achieved on the new multicore processors is strongly dependent on the algorithm used.

Thesis Supervisor: John L. Wyatt, Jr.
Title: Professor

Thesis Supervisor: Thomas A. Keim
Title: Principal Research Engineer

Acknowledgments

This is the last thing I am writing. I am tired and want to go to sleep, so if I left you out and you are supposed to be here, please forgive me.

First, I would like to thank my parents. They have been really supportive and have shared in all the ups and downs of this experience with me. It is hard to imagine where I would have ended up if it had not been for their participation in this process. Ever since I was young, they always encouraged me and always believed in me and instilled in me a sense of self confidence that helped me get through this process. They have provided a great example of what it means to be both parents and human beings. I would like to thank my mom for making Christmas an extremely special time for me and the whole family. I am not sure how you are able to always plan such a big event and make it such a success all the time. I would like to thank my dad for talking with me about things that interest me and for being a great dad.

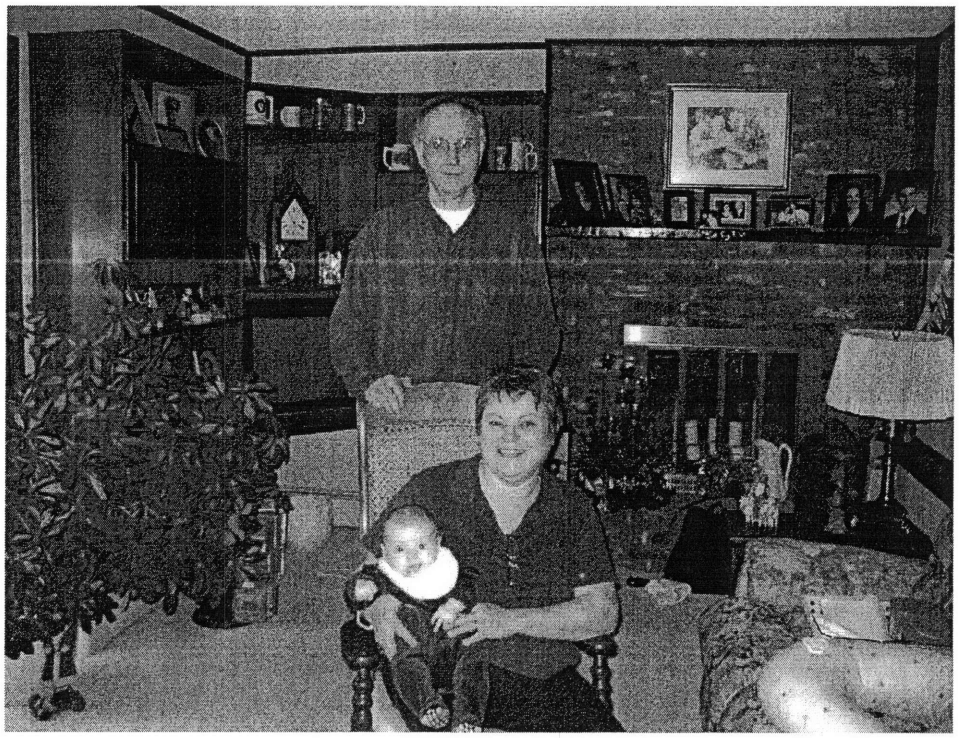


Figure 0-1. My parents Sam and Kathy Geraci with my niece Sophia.

Next, I should thank my committee. Professor Wyatt, Professor Sadoway, and Professor Daniel all contributed in a major way to this thesis. The two-volume model comes directly from conversations with Professor Wyatt. Professor Sadoway was always excited about my work and even when he was teaching classes with over 400 students in them, he somehow found time to think about my project and take the time to share his ideas, thoughts, and insights with me. Finally, Professor Daniel took the time to explain Newton's method to me, and he really thought about all the other numerical issues associated with the solution of the models' numerics in this thesis and helped me make the thesis a better project because of it.

I would also like to thank Professor Jaume Peraire, Le Duc Vinh and Dave Willis for their help with the 2-D model's setup phase. These people gave freely of their time and every time I had a chance to talk with them, they moved my project greatly forward.

Professor Yang-Shao Horn and Professor George Barbastathis allowed me to use their interferometer which really helped the completeness of this thesis. Their student Laura Waller spent countless hours helping me setup and image the physical processes in the lead-acid battery. Without her assistance, I would never have been able to obtain experimental results.

Howard Song of Compact Power also provided a great boost to my research. His support allowed me to have the freedom to work exclusively on my project for a while. Although is support only lasted a semester, it gave me a respite from the TAing that I had been funding myself with and allowed me to move my project forward greatly.

The High Performance Embedded Computing group at MIT Lincoln Labs was also fantastic. They not only provided the financial support needed to do all of the work presented here concerning the Cell Broadband Engine, but they also provided a wonderful atmosphere and place in which to do the work.

Bob Bond, Jeremy Kepner, Sharon Sacco, Han Kim, Nadya Bliss, and Sanjeev have been an excellent group of people with whom I look forward to having a long fruitful relationship.

I would like to thank Professor Sul Seung-ki of Seoul National University for allowing me to visit his lab and making me feel at home. Early on in my graduate career he once told me that he though I would be a very good engineer. These simple positive words had a big positive impact on how I viewed myself and my work.

I would also like to thank all of my siblings for their support. Sam and his wife Caroline, Rose and her husband Jeff, Joe and his wife Sue, and Becca all helped in their own ways.

Next, I would like to thank my nieces and nephews. Seeing them at Christmas has been incredibly fun. They are all very cute. I am looking forward to see how their lives develop. Among them, I have gotten to know my nephew Sam III and my niece Sophia the best. I would like to thank Sam III for his interest in and excitement about my work. It was very encouraging. I would like to thank my niece Sophia for being truly excited about having me over to her house and for sharing her time with me. A place like MIT can cause even the most confident person to doubt themselves at times, but my visits with Sophia, and her parents, and their belief in me really helped me continue. It has been truly great to see her growing up before my eyes.

I would like to thank my friend Sudarshan Raghunathan. Without his help in our January 2007 Playstation 3 programming class, I would not have been able to do the work on the PS3. Ghinwa Choueiter for being a great student, a great friend and having me over for holidays. Andy Copeland for his long time friendship and encouragement. Everest Huang, who was my freshman roommate, for his friendship, advice, support, humor, tolerance, and all round greatness. Obrad Scepanovic for being like a brother to me. Noriko Hara for showing me what Hemingway meant when he said, "Courage is grace under

Figure 0-2. My nephew Sebastian Geraci III.

Figure 0-3. My niece Sophia Geraci.

pressure." Also for sharing her curious and uplifting spirit with me. Brad Bond for listening to more stories about more things than I probably should have shared. Christian Sevilla for being in touch and being a great friend. Alice Chan for being a well read and very interesting person to spend time with who proves that actuaries can be fun. Also for showing me the most impressive thesis defense I saw in my almost 16 years at MIT. Ali Shoeb for being a great student and a really fun friend. Someday I hope to be able to run as well as he does. Laura Jane Finn for showing me what citizenship means. Ali Motamedi for discussing every topic under the sun with me. Bill Richoux for arranging the trip to the world Curling Championship and the Chick-fil-A for my defense. Both are something I will remember my whole life. Dina Katabi for being a captivating TA and for being a kind spirit to me. Agha Mirza for being a great TA. I still remember most of his recitations even 15 years after the fact. Borjan Gagoski for sending me a wonderful email at the end of my being his Stochastic Processes TA. Xuemin Chi for talking with me about life, the universe, and everything. Chris Barnes for being my West Coast twin. Bill Evans for his encouragement and checking up on me. For treating me like family and for keeping me up to date with the news.

Finally, I would like to return to Professor Wyatt and my parents. Both of them provided me with examples of how to teach and how to help a young person develop as a person. Because my parents took great interest in me when I was young I was able to succeed as a young adult. In the same way because Professor Wyatt took an interest in me as a person during my graduate student career, I believe that I am a better person and that I will succeed on a human level because I now have not only my parents excellent example but also his example to follow.

Contents

Chapter 1

Purpose and Contributions

1.1 Introduction

Toyota, GM, Ford, Honda, and Mercedes-Benz all sell some type of hybrid electric vehicle. Toyota, by 2010, plans to have at least 14 hybrid electric vehicles in its lineup and expects a total annual sale of at least 1 million hybrid cars [1]. For its part, General Motors plans to introduce Lithium ion battery powered hybrid cars by 2010 [2]. These trends show not just an increased interest but an actual concrete increased investment in battery technology on the part of the automobile industry. Therefore, it is without a doubt that the number of and importance of batteries will continue to grow.

With the increased interest in hybrid and electric cars, there is also an increased interest in answering not only the question of what is the state-of-charge of a battery but also, what is the state-of-health of the battery?

The state-of-charge of a battery simply describes the amount of charge left in the battery relative to some maximum amount of charge. For example, if a battery is fully charged, one would say that it had a 100% state-of-charge. If the battery were fully drained, one would say it had a 0% state-of-charge.

Over time, and with use, this maximum amount of charge decreases. Most laptop users have noticed this effect. With a new laptop, they seem to experience long battery life. However, as time goes on, and they use their laptop more and more, the battery life seems to get shorter and shorter even though they may leave the laptop on the charger overnight or even for a few days. This shortening of battery life is due to aging effects. It is this change in maximum charge that the state-of-health measurement tries to quantify.

The state-of-health of a battery gives some idea of the how different the battery at the present time is from the battery when it was new. This difference between the battery at the present time $t = T$ and some past initial time $t = 0$ depends greatly on how the battery was used. For example, a typical flooded lead acid battery can be discharged from 100% of full charge down to 30% of full charge and then recharged to 100% of full charge about 1500 times [3]. However, the same battery, if new, is good for about 2200 cycles if it is only discharged from 100% of full charge down to 60% of full charge and back [3]. The number of uses (cycles) that a battery can go through is called the cycle life of the battery.

This usage dependent battery aging is not a phenomenon exclusive to lead acid batteries. For example, in the case of Lithium ion batteries, "volume changes in electrode particles during lithium insertion and extraction create stresses which may induce cracking" [4]. The cracking is also a function of charge and discharge amount and rate. Therefore, although the lead acid battery will be used as the basis for this work, the same sorts of aging issues exist in other battery chemistries, fuel cells, and battery packs.

Not only is cycle life a strong function of how the battery is used, but also "cycle life is dependent on a number of construction factors" [5] These construction factors include, but are not limited to: the thickness of the plates, the active mass density of the active material, the concentration of the acid, and the geometry of the electrodes [5, 6].

The numerous empirical models that allow estimates of the state-of-charge of a battery [7–12] look at the terminal characteristics of the battery; however, they do not incorporate the physical geometry of the battery. The goal of this thesis is to make progress toward a simple lead acid battery model that takes into account the aging effects caused by geometry.

In order to develop an effective model for the state-of-health of a battery, our model must not only take into account the usage of the battery as seen at the terminals, but also must somehow include battery geometry information. One geometry related effect that has been noted in various places in the literature is that the upper portion of the electrodes participate in the reaction much more than the lower portion of the electrodes. In [13, p.296] "sampling tubes placed in various vertical positions" between the electrodes were used to observe that the, "specific gravity scarcely changed at the bottom (of the electrodes) during discharge while at the upper portion (of the electrodes), it decreased to 1.13 $\frac{g}{cm^3}$ from 1.27 $\frac{g}{cm^3}$." While in [14, p.244] holographic laser interferometry was used to observe concentration differences between the electrolyte at the top half of the battery and the bottom half of the battery leading to the observation that the "upper part of the positive electrode (lead dioxide electrode) is more favoured to undergo reaction than the lower part. This is due to the lower ohmic resistance in the upper region." These electrolyte concentration differences between the upper half and lower half of the electrode are significant because it has been found that decreased cycle life is directly related to increased level of difference between the concentration in the upper and lower regions of the battery [15, p.196].

This idea of the upper half of the electrodes reacting more than the lower half of the electrodes is of profound importance for this thesis, so as to better understand it, let us take some time to understand and better visualize what an electrode is and what reactions are going on at the electrodes of a lead acid battery. In the rest of this chapter, we will refer repeatedly to the quote from [14].

1.2 Lead Acid Battery Structure

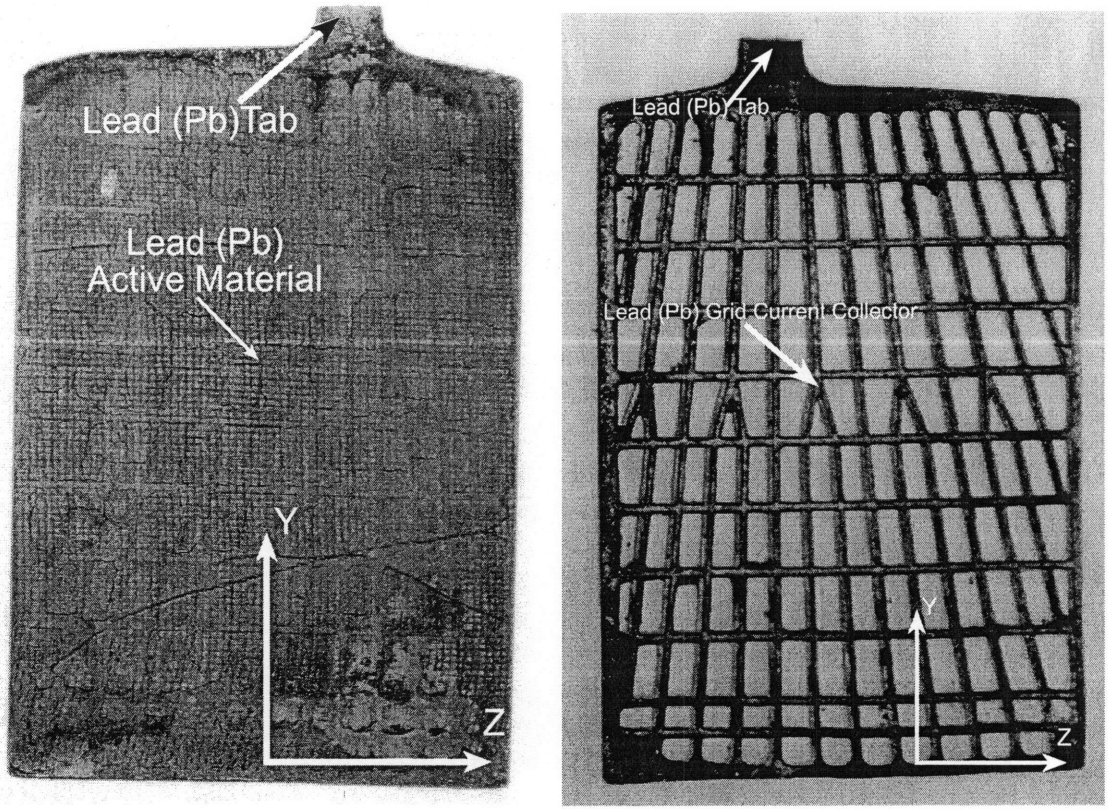
First let us take a look at the structure of the electrode so as to better understand what an electrode looks like and why there might be more ohmic resistance at the bottom of the electrode than the top of the electrode.

A cell of a flooded lead-acid battery consists of a lead (Pb) electrode (fig. 1-1) and a lead dioxide ($Pb(IV)O_2$) electrode (fig. 1-2), both of which sit in a sulfuric acid (H_2SO_4) bath. Each electrode consists of two major parts: an active material region where the primary chemical reaction takes place and a lead (Pb) grid backbone, referred to as the "lead (Pb) current collector grid" [16]. The backbone grid carries electrons between the active material region of the electrode and the lead (Pb) tab, which connects the battery to the external circuit. The lead (Pb) grid current collector runs the entire length of the electrode and also serves as a support framework to which the active material regions are attached.

Figure 1-1 (a) shows a face view photo of a lead (Pb) electrode. This photo clearly shows the lead (Pb) active material region and the lead (Pb) grid current collector's lead (Pb) tab. However, the active material region obscures the major portion of the lead (Pb) grid current collector. Therefore, for fig. 1-1 (b), the active material region of the electrode has been removed to reveal the lead (Pb) grid current collector.

Figure 1-2 (a) shows a face view photo of a lead dioxide electrode. In this photo both the lead dioxide active material region and the outline of the lead (Pb) grid current collector are visible. Figure 1-2 (b) shows the profile view of a lead dioxide electrode.

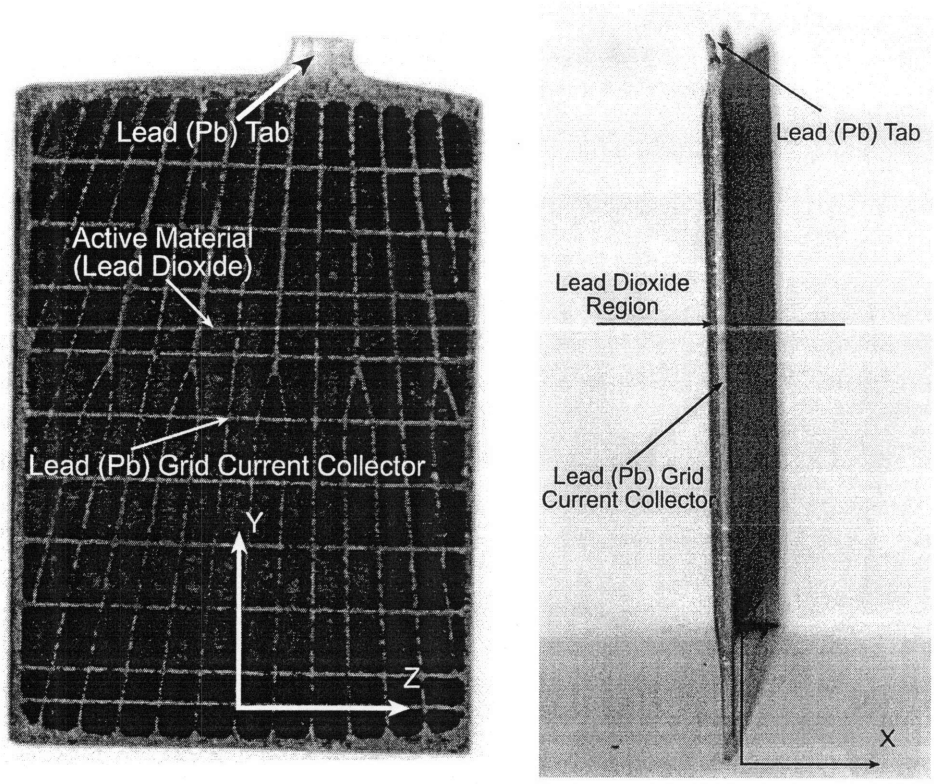
From figs. 1-1 and 1-2, it can clearly be seen that both electrodes are much larger in the y-direction than in the x-direction. Furthermore, current is only delivered to or taken from the top lead (Pb) tab of each electrode. This is significant because it is this tall height that gives rise to the previously mentioned



(a) Photo of lead (Pb) plate with the lead (Pb) active material region still attached. Here the active material obscures the lead (Pb) grid current collector.

(b) Photo of lead (Pb) plate with active material region removed so that the lead (Pb) grid current collect can clearly be seen.

Figure 1-1. A lead (Pb) plate taken from a Yuasa YTX20HL-BS-PW flooded lead-acid battery.



(a) Photo of lead dioxide plate

(b) Photo of the profile of a lead dioxide plate.

Figure 1-2. A lead dioxide plate taken from a Yuasa YTX20HL-BS-PW flooded lead-acid battery.

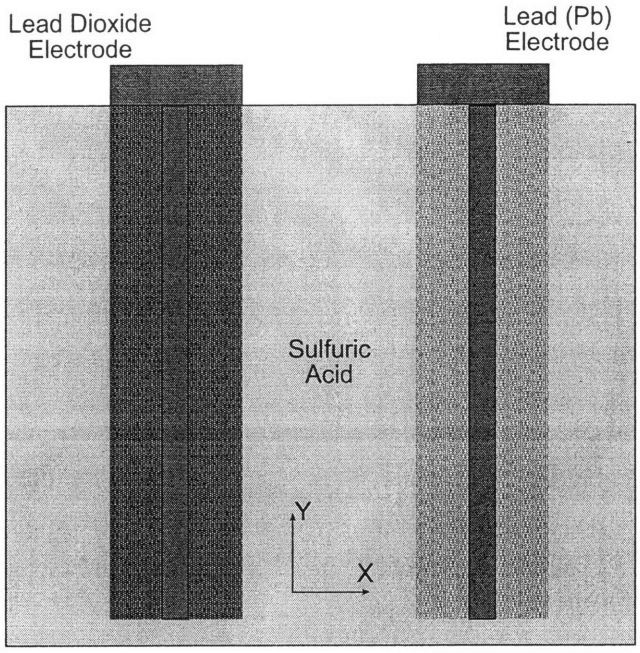


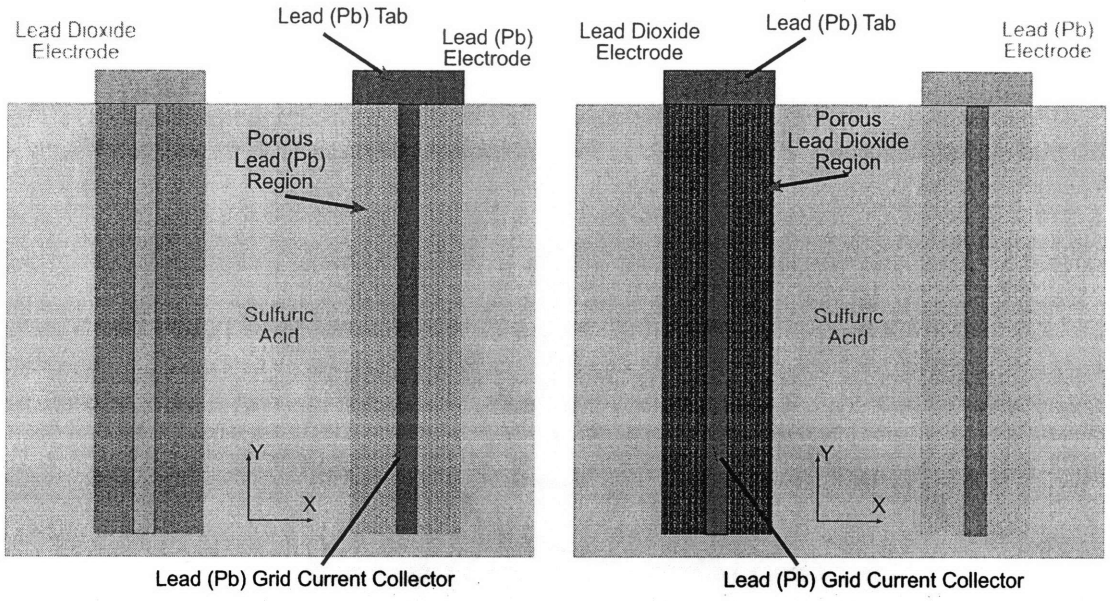
Figure 1-3. Illustration of a lead-acid battery cell.

ohmic drop. Essentially, electrons at the bottom of either electrode have to go further than electrons at the top of the electrode to participate in the reaction.

Continuing with a little more background information, fig. 1-3 shows an illustration of how the two electrodes might look sitting in a sulfuric acid bath. The view is a profile of each electrode like that seen in fig. 1-2 (b).

The labels in fig. 1-4 (a) and fig. 1-4 (b) point out the major parts of each electrode. These parts correspond to the major electrode parts labeled in fig. 1-1 and fig. 1-2. In fig. 1-4 (a) and fig. 1-4 (b), the tall wide textured regions of each electrode represent the active material regions. While the tall skinny solid shaded regions represent the lead (Pb) grid current collector and tab. For the $Pb(IV)O_2$ plate the active material is lead dioxide and for the Pb plate it is lead (Pb).

The active material region on each electrode is not one solid piece of material. Instead, it has a consistency similar to that of packed sand. Figure 1-5 shows a drawing of a magnified cross section



(a) The Pb Electrode

(b) The $Pb(IV)O_2$ Electrode

Figure 1-4. Parts of the lead-acid battery cell.

from the active material region. This cross sectional area contains both solid material and liquid. Any other randomly selected cross section of either electrode would look similar. While not shown explicitly in the drawing, the solid material is a "persistent solid phase, called the solid matrix" [17]. The non-solid phase is a void space filled with electrolyte. This kind of packed sand like material arrangement is called a porous medium. This porous structure has the effect of greatly increasing the surface area at which the charge transfer reaction can occur. Furthermore, while the diagram shows the lead dioxide pieces as being separate and not connected, they are actually connected through material that exists in the z-direction.

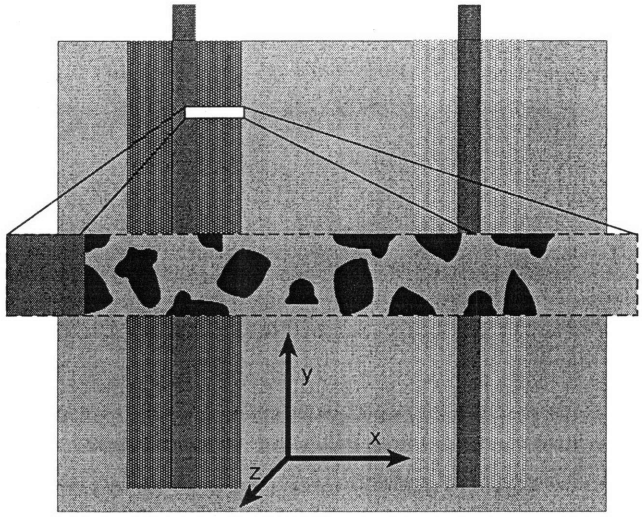


Figure 1-5. Illustration of how a cross section of the active material region of a lead-acid battery cell might look. Notice how it is porous in nature instead of a single completely filled in solid piece.

1.3 Primary Chemical Reactions of Lead Acid Batteries

Now let us take a look at what reaction is being talked about in our quote of interest. When the lead-acid battery cell is either discharged or charged, each plate of the cell undergoes its own primary reaction. For the lead dioxide electrode, this reaction is:

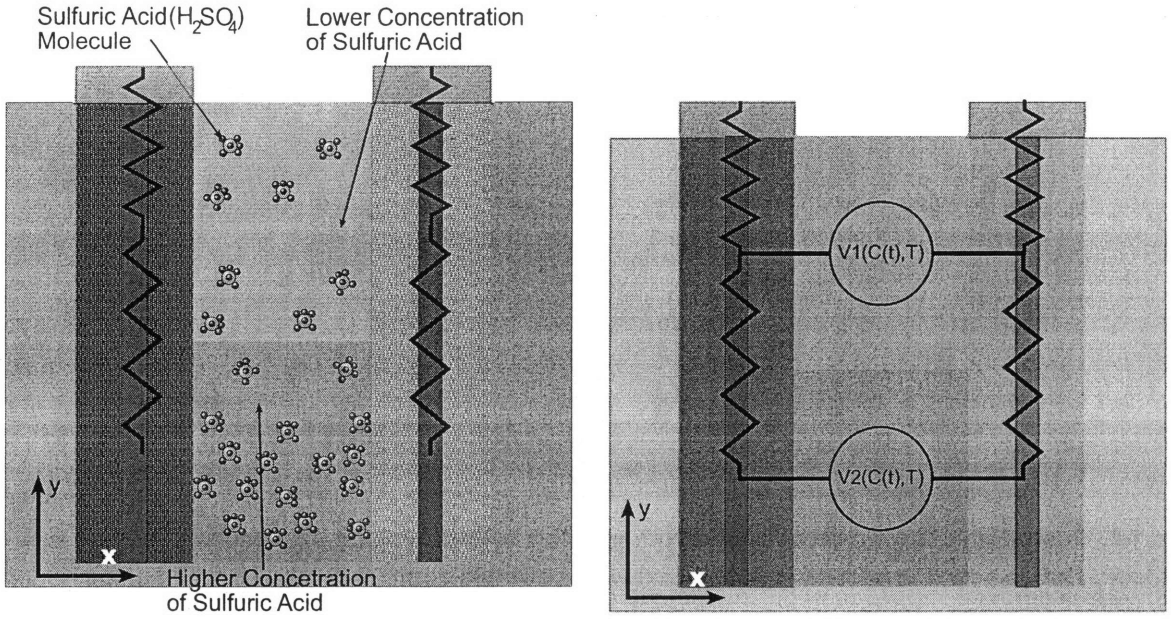
$$\mathrm{Pb(IV)O_2}\,(s) + \mathrm{HSO_4^-}\,(aq) + 3\mathrm{H^+}\,(aq) + 2\mathrm{e^-} \rightleftarrows \mathrm{Pb(II)SO_4}\,(s) + 2\mathrm{H_2O}\,(l) \tag{1.1}$$

For the lead (Pb) electrode the reaction is:

$$\mathrm{Pb}\,(s) + \mathrm{HSO_4^-}\,(aq) + \mathrm{H^+}\,(aq) \rightleftarrows \mathrm{Pb(II)SO_4}\,(s) + 2\mathrm{H^+}\,(aq) + 2\mathrm{e^-} \tag{1.2}$$

Adding these two equations together results in the overall system reaction shown in eq. (1.3).

$$\mathrm{Pb}\,(s) + \mathrm{Pb(IV)O_2}\,(s) + 2\mathrm{H_2SO_4}\,(aq) \rightleftarrows 2\mathrm{Pb(II)SO_4}\,(s) + 2\mathrm{H_2O}\,(l) \tag{1.3}$$



(a) The concentration of sulfuric acid is less in the top half of the electrode than the lower half.

(b) Because of the difference in concentration between the upper half and lower half of the electrode, there is also a difference of potential between the two halves.

Figure 1-6. The two level battery model.

During discharge, the lead dioxide electrode acts as a cathode and undergoes reduction thereby acting as an electron sink. The lead (Pb) electrode acts as an anode and undergoes oxidation thereby acting as an electron source.

1.4 Two Level Model

The reaction our quote is talking about is eq. (1.1). In this reaction, lead dioxide is combined with sulfuric acid to produce lead sulfate and water. So, when the quote talks about the upper part of the electrode being "more favoured to undergo reaction than the lower part", it means that more lead dioxide and sulfuric acid will be consumed at the upper part of the electrode than at the lower part of the electrode. This implies that, at any given point in time, there will be less sulfuric acid at the top of the

electrode than at the bottom of the electrodes. This non-uniform sulfuric acid concentration is shown in fig. 1-6(a).

Fig. 1-6(a) shows a lead acid battery cell. The cell consists of a pair of porous electrodes sitting in a liquid sulfuric acid electrolyte bath. Drawn on top of each electrode is a resistor symbolizing the ohmic drop which gives rise to the geometrically variable reaction rate. The sulfuric acid electrolyte bath between the two electrodes has been divided into two levels, a top level and a bottom level. The bottom level has more moles of sulfuric acid in it than the top half. This is because the top half of the electrodes participate in the reaction more than the bottom half of the electrodes.

So that we can start translating fig. 1-6(a) into a practical implementable model, we next use Nernst's equation eq. (1.4) to derive a simple circuit model for our two level approximate model. Nernst's equation says that there is a relationship between the concentration of a battery's electrolyte, the temperature, and the battery cell's measured external potential.

$$E_{cell} = E^{\circ}_{cell} - \left(\frac{\mathrm{RT}}{\mathrm{nF}}\right) \ln Q \tag{1.4}$$

which relates cell potential to cell temperature and the electrolyte concentration of the cell (1.4) where $Q = \frac{1}{[H_2SO_4]^2}$. [1]

Using Nernst's relationship and the observation about the ohmic drop along the height of the electrodes from [14] we can create the simple circuit shown in fig. 1-6(b). The circuit consists of two voltage sources connected in parallel by two resistors and two more resistors leading to the terminals of the battery. From eq. (1.4), the voltage sources are dependent on electrolyte concentration C and temperature T. For this thesis we will assume constant temperature T, but C is allowed to vary with

[1] Q is called the reaction quotient [18, p.543].

usage and therefore time. The trick now is to determine how $C(t)$ actually varies with time.

For notation purposes, throughout this thesis, physical battery state parameters will be highlighted in color. Here C has been highlighted in orange.

Continuing, if [14] is to be believed, then when the battery is either charged or discharged, V_1 is charged or discharged more than V_2 because there is more resistance between V_2 and the output terminals than between V_1 and the output terminals. This means that V_1 participates in the reaction more than V_2, and therefore, the electrode surfaces near V_1 get cycled more and to a deeper depth of discharge than the electrode surfaces near V_2. Cycling more and to a deeper depth of discharge means, according to [3] that the top half of the battery should fail before the bottom half of the battery. It is this uneven usage of the battery at a constant temperature that we would like to be able to model and incorporate into a real time state-of-health indicator.

1.5 Purpose

This thesis seeks to make progress toward a lead acid battery state-of-health monitor that can be used in an embedded environment. Toward this goal, it will:

1. Experimentally justify the two level model of fig. 1-6(b) as a minimum order aging model.

2. Investigate and develop an understanding of the physical processes that are believed to give rise to the two level model of fig. 1-6(b)

3. Demonstrate, via an appropriately sophisticated numerical model, that the investigated physical processes do in fact show an unequal usage of the upper half and lower half of the battery electrodes.

4. Investigate implementation issues that might be encountered when trying to implement fig. 1-6(b) in a future embedded system.

1.6 Contributions

In doing these four things, a number of contributions will be made:

From 1 Experimental verification the two level model using a digital Mach-Zehnder interferometer instead of the holographic interferometers that have been used to date. This will be the first known use of digital interferometry for this system.

From 2 A unique visualization of the underlying molecular action that occurs during lead acid battery discharge and a two volume model that can be used as a tool to explore the physical parameters that influence the output of a lead acid cell.

From 3 A 2-D electrochemical battery model and an analysis of its abilities.

From 4 Three different Gaussian Elimination based direct solvers based on two different algorithms for use on the Sony, Toshiba, IBM manufactured multicore Cell Broadband Engine. These solvers highlight different ways in which performance can be gotten out of the new multicore processor technologies. Furthermore, one of the solvers will be extended by making it erasure fault tolerant.

Chapter 2

Experimental Justification of Two Level Model

In this section, interferometry will be used to validate our two level model from section 1.4. These results will also later be used to affirm that the 2d model of section 5.3 captures the physical phenomena that leads to an uneven usage of the electrode along its height. The experiment here will produce a much clearer image of the phenomena than can be found in [14]. Furthermore, it will observe the phenomena using digital Mach-Zehnder interferometry instead of the holographic laser interferometry in the paper, thereby using a second method to get the same result.

Interferometry is an appropriate tool for our job because interferometry can be used to provide measurements of the change in index of refraction for a liquid [19, p.254]. The refractive index of sulfuric acid, fig. 2-1, is almost linearly related to the concentration of the acid [20]. Therefore, interferometry has been extensively used as a tool for observing stratification and convection in lead acid batteries [14, 21–23].

The experimental results presented here will not only corroborate the findings of [14], but also justify the two level model presented in section 1.4 and give credence to the idea that the 2d model of section 5.3

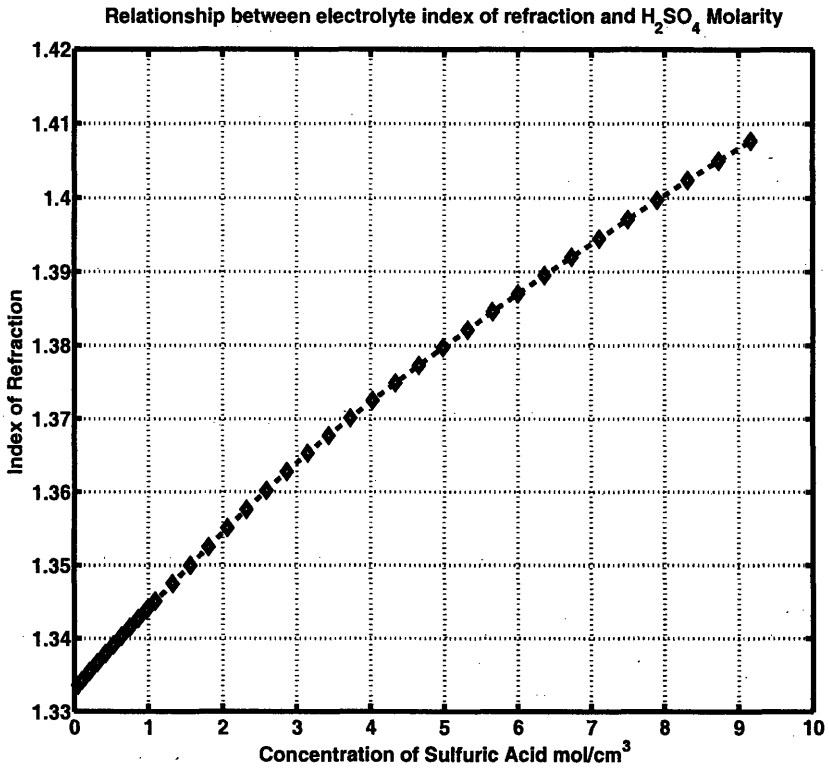


Figure 2-1. The index of refraction of sulfuric acid varies significantly with molarity of acid concentration. Acid concentration varies with electrode usage. This relationship between refractive index, concentration, and usage is why interferometry can be used to provide useful information about electrode utilization.

correctly captures the major physical phenomena that cause the difference in usage between the upper half and lower half of the electrodes.

2.1 Experimental Setup

The Mach-Zehnder interferometer is regarded as "probably the most versatile interferometer for the observation of refractive index fields." [24, p.309]. The operation of the Mach-Zehnder interferometer is well documented and can be found in almost any book on interferometry [25]. The basic idea of the interferometer is that it works by splitting the beam into an object beam and a reference beam. The two

beam paths will be of different length, so when they are rejoined, they will be slightly out of phase and combine destructively. This destructive interference creates the fringe patterns seen at the CMOS sensor.

The experiments described here were run on a Mach-Zehnder interferometer co-owned by Professor Barbastathis and Professor Shao-Horn of MIT's Department of Mechanical Engineering. The interferometer was constructed and operated by Laura Waller of MIT's Department of Electrical Engineering and Computer Science. A labeled photo of the Mach-Zehnder interferometer with an outline of the test cell drawn in is shown in fig. 2-2. A fully labeled drawing of the interferometer is shown in fig. 2-3.

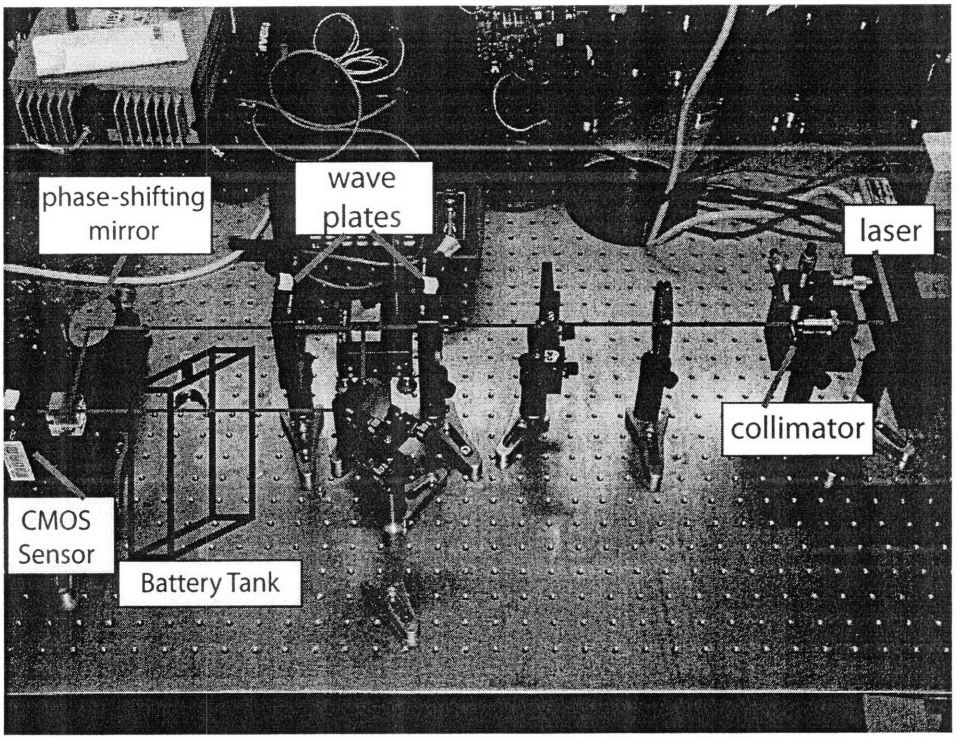


Figure 2-2. The experimental setup was a standard digital Mach-Zehnder interferometer with phase-shifting shown. Drawn onto the picture is the path the laser light would take.

Of note, the CMOS sensor is a Basler 504k. It has 12μm x 12μm square pixels in a 1280H x 1024V grid. The sensor region is 15.26mm horizontal by 12.29mm vertical. It is a black and white sensor that is capable of 500fps shooting. Although the laser was expanded to 2.54cm across, only the center

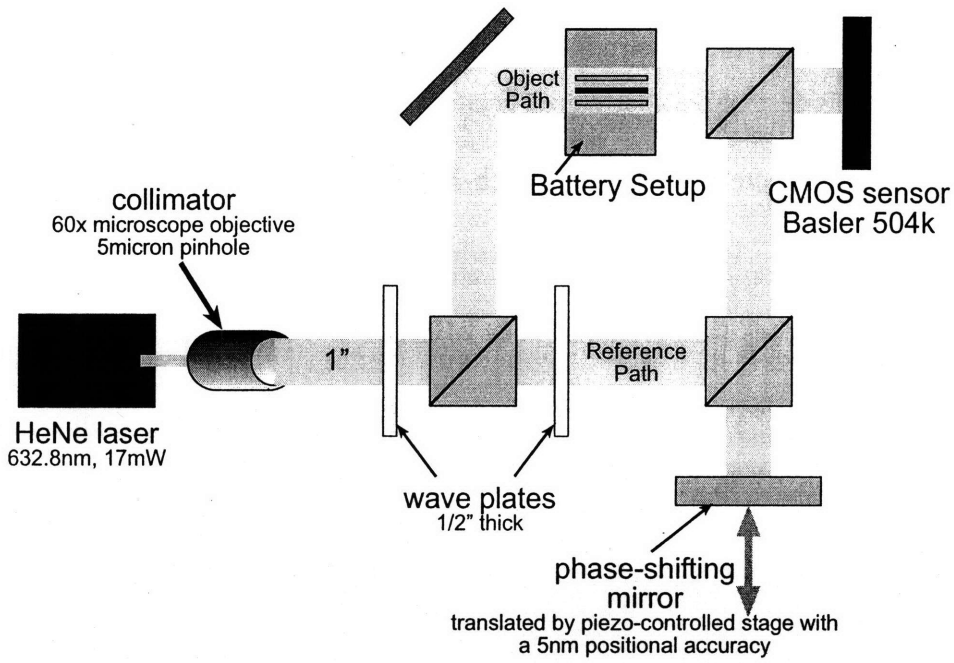


Figure 2-3. The experimental setup was a standard digital Mach-Zehnder interferometer with phase-shifting.

15.26mm horizontal by 12.29mm vertical of the beam was observed and recorded by the CMOS sensor.

The test cell was made up of three electrodes: two lead (Pb) electrodes and one lead dioxide electrode. The lead dioxide electrode was positioned between the two lead (Pb) electrodes as was done in [14]. The interelectrode spacing was 3.51mm between the left lead (Pb) electrode and the lead dioxide electrode and 2.19mm between the lead dioxide electrode to the right lead (Pb) electrode.

The two lead (Pb) electrodes were 2.4mm thick and 15.20mm deep and over 102mm tall. The lead dioxide electrode was 1.25mm thick and 15.20mm deep and greater than 102mm tall.

These electrodes were placed in the quartz glass cell described in fig. 2-4. The glass cell with the electrodes positioned in it, was filled with 5 molar sulfuric acid and allowed to soak overnight. After soaking, the glass cell was placed in the interferometer in the position shown in fig. 2-3 so that the laser light could pass between the electrodes at the middle of the upper half of the cell, recombine with the

reference wave and then strike the CMOS sensor.

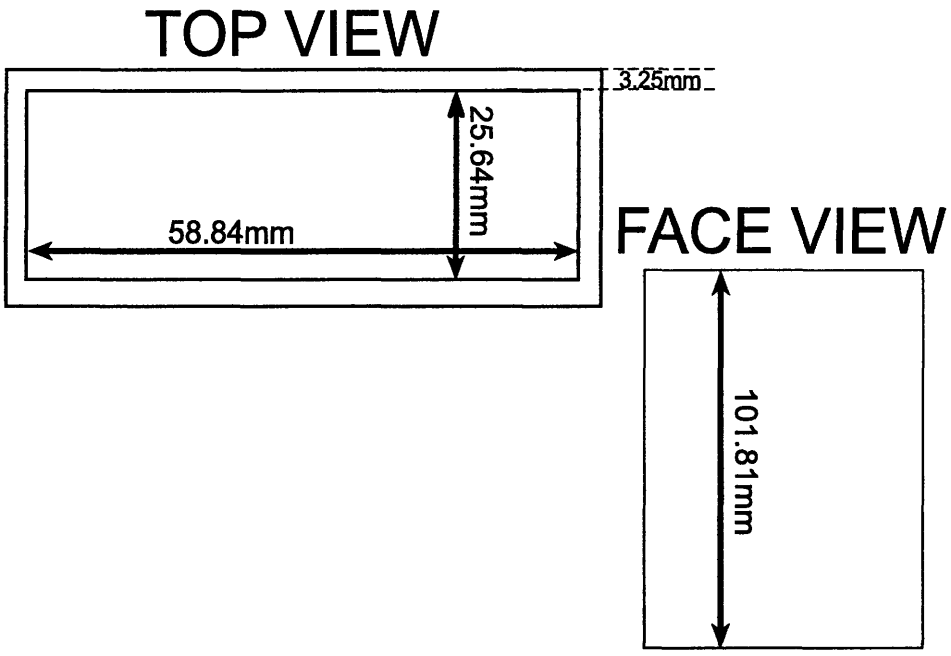


Figure 2-4. The battery cell tank was made out of quartz glass ordered from McMaster Carr.

2.2 Experimental Data

The first experiment conducted was to take an interferogram of the cell at rest. This is seen in fig. 2-5. The cells have been at rest for 10 hours at this point in time.

This interferogram clearly shows the three electrodes. The left-most electrode is a lead (Pb) electrode. The middle electrode is the lead dioxide electrode and the right-most electrode, that is just on the right-most edge of the diagram, is the other lead (Pb) electrode.

In each region, to the left of the left-most lead (Pb) electrode, between the left-most lead (Pb) electrode and the lead dioxide electrode, and between the lead dioxide electrode and the lead (Pb) electrode there are dark evenly spaced horizontal lines called fringes. These fringes are created as a result of the action of the interference of the reference beam with the object beam. Each fringe represents the superposition of

the observation path waveform with that of a waveform coming from the reference path that is $\pi + 2k\pi$ out of phase with it, where k is an integer. These parallel evenly spaced lines means that there is a uniform concentration of electrolyte in the observed area.

Also seen on the image is an off angle oval set of concentric fringes. These fringes are concentric around a point just above the middle of the right edge of the left-most lead (Pb) electrode. These fringes are the result of a protective plastic mask that was poorly glued by the factory to the surface of the CMOS sensor. These extra fringes will have no effect on the data taken as they will be of very small amplitude in the data collection region and orthogonal to the data taken. Therefore, they will not constructively add to the data and will have minimal effect on the results.

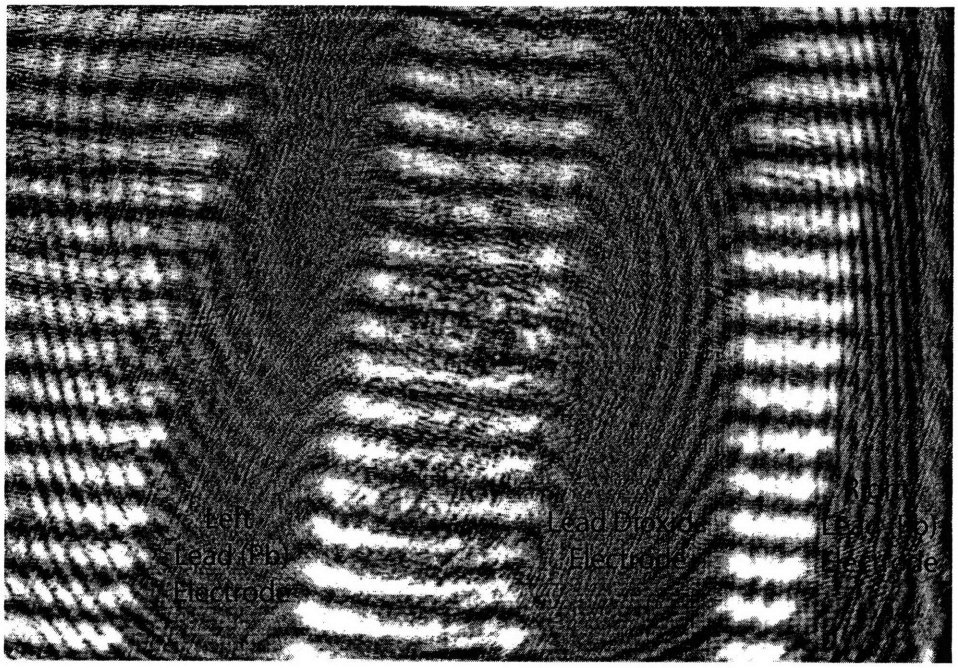


Figure 2-5. The parallel evenly spaced horizontal dark fringes show that the concentration of electrolyte is uniform under rest.

In the second experiment the cell was discharged at a rate of 50mA. Fig.2-6 shows the interferogram after 29 minutes of discharge. It clearly shows significant activity around the center-positioned

lead dioxide electrode. This activity takes the form of non-uniform spacing of the dark fringes. These non-parallel unevenly spaced fringes are representative of a change in concentration at the surface of the electrode. Halfway between the lead dioxide electrode and each of the lead (Pb) electrodes, the fringes become almost uniform. At the surface of the lead electrodes, there is some bending of the dark fringe patterns, but not nearly as much as seen at the lead dioxide electrode. This indicates that the concentration variation at the surface of the lead (Pb) electrode is not as great as that at the surface of the lead dioxide electrode. The concentration at the surface of the lead dioxide electrode dipping more than the concentration at the surface of the lead (Pb) electrode makes sense if one remembers the discharge reactions that take place at the lead dioxide electrode:

$$\mathrm{Pb(IV)O_2}\,(s) + \mathrm{HSO_4^-}\,(aq) + 3\mathrm{H^+}\,(aq) + 2\mathrm{e^-} \rightarrow \mathrm{Pb(II)SO_4}\,(s) + 2\mathrm{H_2O}\,(l) \tag{2.1}$$

and the lead (Pb) electrode:

$$\mathrm{Pb}\,(s) + \mathrm{HSO_4^-}\,(aq) + \mathrm{H^+}\,(aq) \rightarrow \mathrm{Pb(II)SO_4}\,(s) + 2\mathrm{H^+}\,(aq) + 2\mathrm{e^-} \tag{2.2}$$

$\mathrm{H_2O}\,(l)$ is one of the products of the discharge reaction at the lead dioxide electrode, so not only is the reaction *consuming* sulfuric acid, which dilutes the solution, but it is also dumping water into the electrolyte which further dilutes the electrolyte. In contrast to this, the reaction at the lead (Pb) electrode only consumes sulfuric acid and does not produce water. Therefore, it is reasonable to expect a greater change in concentration at the lead dioxide electrode than at the lead (Pb) electrode.

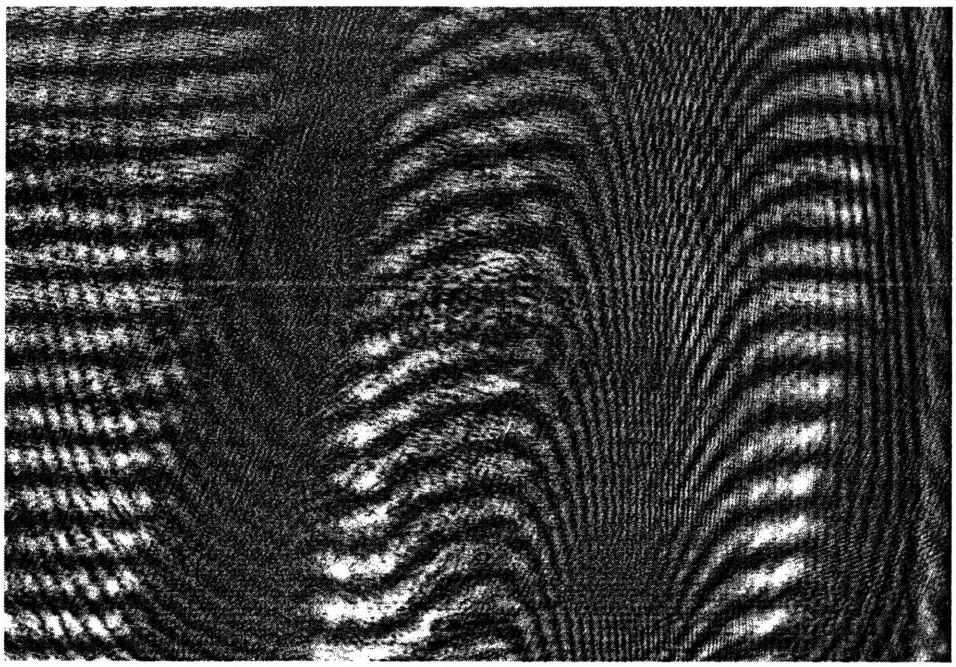


Figure 2-6. The curved non-parallel unevenly spaced dark fringes show that discharge activity is taking place between the battery electrodes. The parallel evenly spaced dark fringes to the left of the left lead (Pb) electrode show that there is no reaction taking place outside the battery.

2.3 Data Analysis

"Since fringe displacement is a measure of optical path change due to the object, originally straight, equidistant fringes, deformed by a one-dimensional concentration field, can be interpreted as plots of concentration versus distance [24, 321]." Therefore, we should be able to get a two dimensional plot of concentration from the images.

Measurements were taken from within the black box in fig. 2-7. There are several horizontal lines drawn from the bulk electrolyte region into the region to the right of the lead dioxide electrode. These lines are parallel and tangent to some of the fringes in the bulk electrolyte. Each time a horizontal line is crossed by a fringe from above, 2π in phase has been lost. The first fringe crossing on a horizontal line represents the loss of $1 \times 2\pi$ the second represents the loss of $2 \times 2\pi$ and so on so that the z^{th} fringe crossing represents a loss of $z \times 2\pi$ in phase. According to equation 5 of [24, p.283], a loss of 2π represents a change in index of refraction, Δn, of:

$$\Delta n = \frac{d}{\lambda_o} = \frac{0.02564\text{m}}{632.8 \times 10^{-9}\text{m}} = 2.468 \times 10^{-5} \tag{2.3}$$

where d is the optical path length, in this case the inner width of the observation tank, 25.64mm was used. λ_o is the wavelength of the HeNe laser which is 632.8nm.

A red circle has been placed at a few sample crossings in fig. 2-7. The (x, y) position of each of the fringe crossings was recorded. The x, y, z data of each fringe crossing was plotted in Matlab and is shown in fig. 2-8. Data was collected for all horizontal fringes in the observation region.

As can be seen in fig. 2-8, the top portion of the electrode within the data collection region has an observably lower acid concentration than the lower portion of the electrode. Since the potential of a

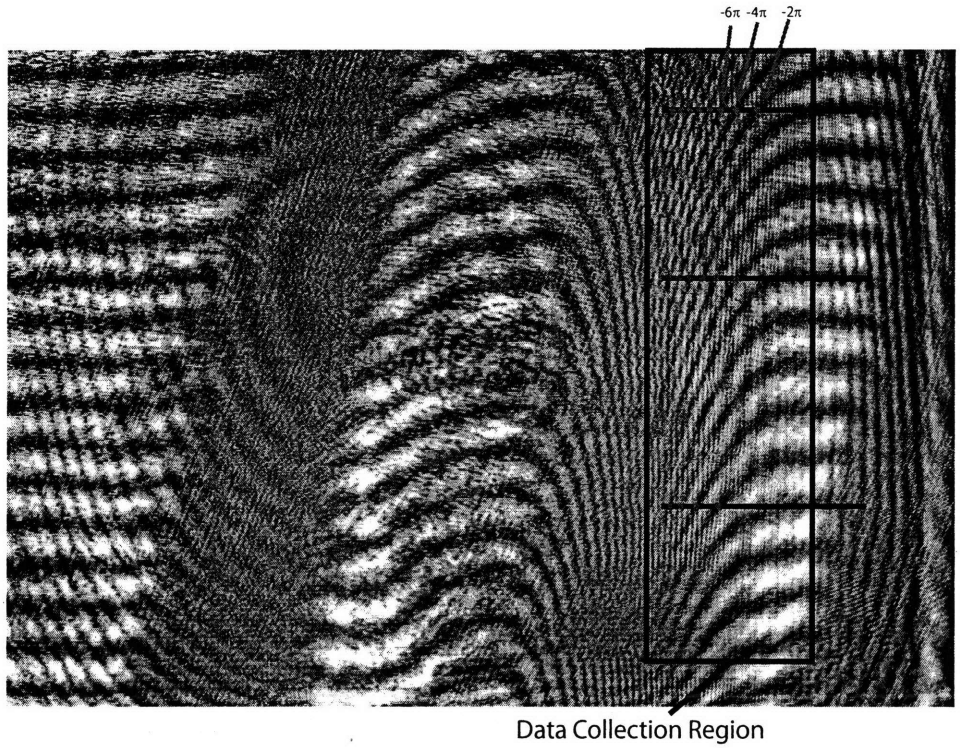


Figure 2-7. The region just to the right of the lead dioxide electrode was used as the area from which data was extracted.

battery is connected to the concentration via Nernst's equation eq. (1.4), the top half of the cell has a lower, thus different potential than the bottom half, thereby justifying the two level model of fig.1-6(b).

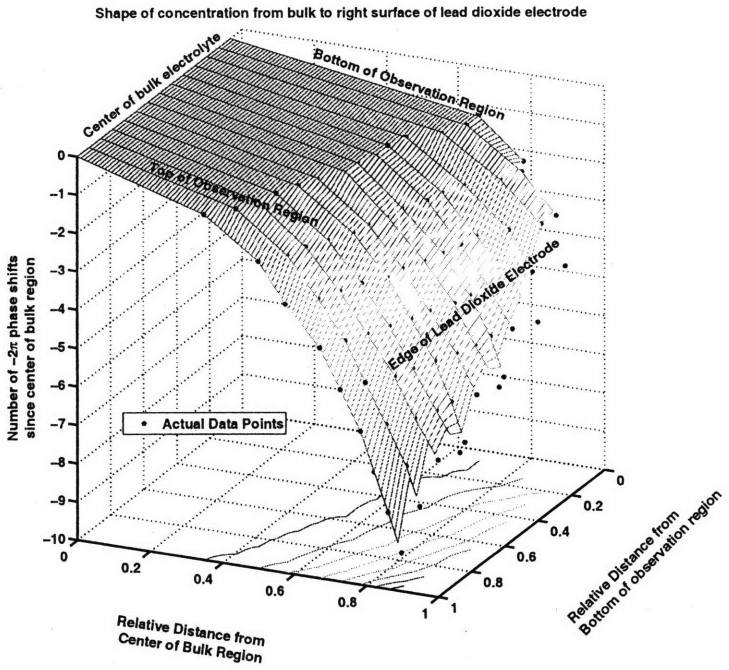


Figure 2-8. The results from the interferogram are translated here into a Matlab mesh plot which clearly shows that the concentration of electrolyte dips closely at the surface of the lead dioxide electrode. Furthermore, the size of the dip is greater at the top of the observation region than it is at the bottom of the observation region.

Chapter 3

Physical Processes of the Lead-Acid Battery

Chapter 2 showed that the top half of the electrode indeed participates in the reaction more than the bottom half of the electrode as was stated in [14]. This chapter will focus on beginning to understand the physical processes that cause $C(t)$ to change with time.

We start our quest for understanding how the concentration in a lead-acid cell changes with time by taking an intuitive look at the lead-acid battery[1]. We have already examined the physical structure of the lead-acid cell and got to know the tangible parts of the cell in section 1.2. Next, we learn about the primary chemical reactions occurring within a lead-acid cell. Finally, we will examine a set of illustrations that visualize one possible way the chemical species involved in the primary reaction at each of the electrodes interact when the electrodes undergo discharge, reaction eq. (1.1). The insight developed by walking through these illustrations will be used in chapter 4 to create a simple two volume model of the physical and chemical reactions that occur in a lead-acid battery.

[1] One of the most often cited works on lead-acid batteries is Bode [26].

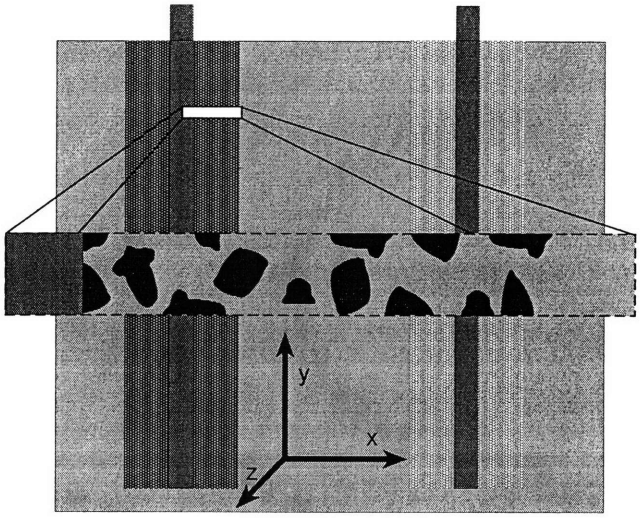


Figure 3-1. An enlarged cross section of a part of the lead dioxide electrode. The section has been divided up into finite volumes.

3.1 Illustration of possible ionic action during discharge at the lead dioxide electrode.

An intuitive feel for what is physically going on during discharge can be developed by studying the progression of illustrations in figs. 3-2–3-6. These figures show one possible way of visualizing the processes that occur during discharge in a volume element randomly selected from the porous active material region of the lead dioxide electrode. Such a volume element might come from dividing up a section of electrode in a way like that shown in fig. 3-1.

For the time being we will simply be passive observers; we won't concern ourselves as to *why* the discharge reaction happens, only *what* happens during discharge. The discharge reaction that takes place within this volume element is eq. (3.1).

$$\mathrm{Pb(IV)O_2}\,(s) + \mathrm{HSO_4^-}\,(aq) + 3\mathrm{H^+}\,(aq) + 2\mathrm{e^-} \rightarrow \mathrm{Pb(II)SO_4}\,(s) + 2\mathrm{H_2O}\,(l) \qquad (3.1)$$

The discharge reaction can occur because a valence shift on Pb^{4+} allows it to become Pb^{2+}, enter solution and interact with the sulfuric acid.

$$Pb^{4+} + 2e^- \rightarrow Pb^{2+} \tag{3.2}$$

3.1.1 Initial Setup

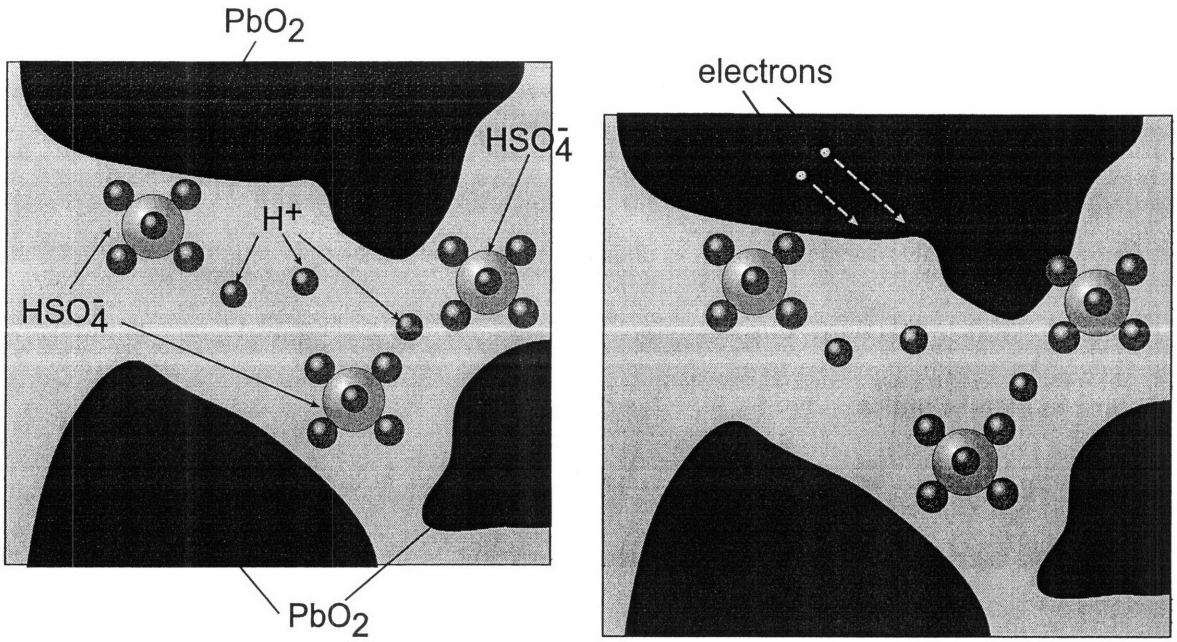
Chemical equations such as eq. (3.1) and eq. (1.2) are written in terms of molar quantities. The volume element being looked at in figs. 3-2–3-6 is too small to contain a mole of anything. The present section, however, is an effort to understand the physical changes that occur because of reactions eq. (3.1) and eq. (1.2), so in an effort to maintain consistency between figs. 3-2–3-6 and eq. (3.1), all quantities will be talked about in units of moles.

Starting with fig. 3-2 (a), we note that the volume element has both a solid phase and a liquid phase. The volume element's liquid phase contains one solvent, water (H_2O), and one acid, sulfuric acid (H_2SO_4). Furthermore, each mole of sulfuric acid is assumed to have dissociated into an equal number of moles of H^+ and HSO_4^- ions[2]. In fig. 3-2 (a), no moles of water are shown, but three moles of dissociated sulfuric acid have been drawn to stand out. Three moles of sulfuric acid have been drawn because the reaction eq. (3.1) requires three H^+ ions.

3.1.2 Electro-neutrality

The system, as drawn in fig. 3-2 (a), is electrically neutral. This means that the total number of positive charges in the volume element equals the total number of negative charges. Electro-neutrality

[2]More accurately, in water, sulfuric acid would dissociate according to: $H_2SO_4 + H_2O \rightarrow H_3O^+ + HSO_4^-$ and sometimes further by $HSO_4^{2-} + H_2O \rightarrow H_3O^+ + SO_4^{2-}$. These dissociations are neglected in the diagrams in order to keep the diagram as simple as possible.



(a) A volume element from the lead dioxide porous region with three moles of dissociated sulfuric acid drawn to stand out against the rest of the electrolyte.

(b) Two moles of electrons enter the volume element's solid phase and perturb its electroneutrality.

Figure 3-2. The start of the reaction in a volume element from a lead dioxide electrode of a lead-acid battery.

can be stated in mathematical terms as:

$$\sum_{<i>} z_i [i] = 0 \tag{3.3}$$

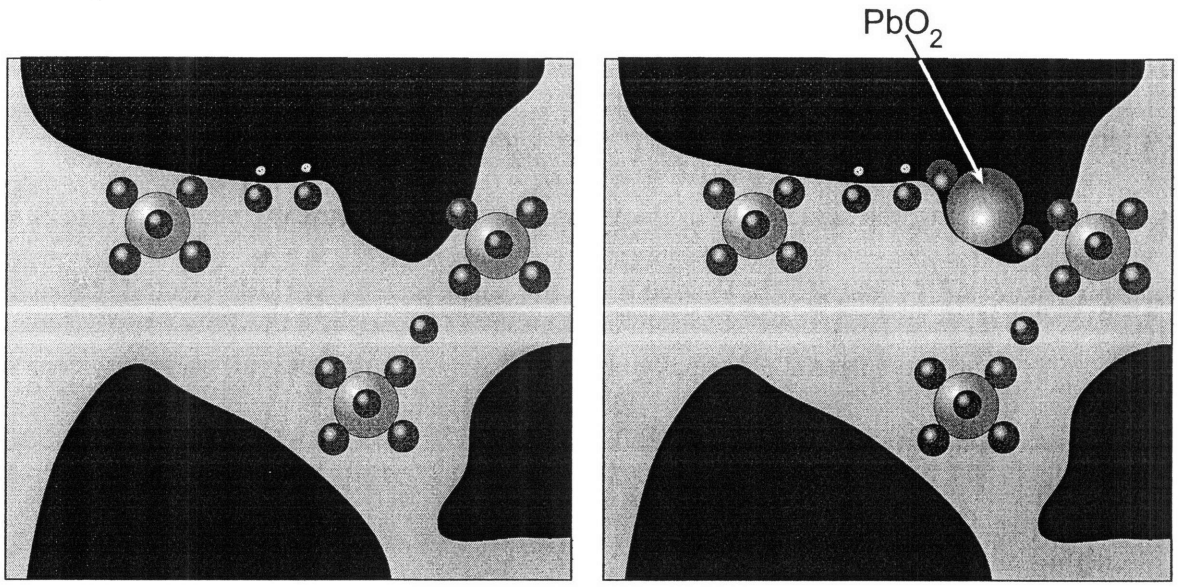
where z_i is the charge on species i and $[i]$ is the concentration of species i.

Electro-neutrality means that the net charge of the entire system, and any subdivision of it, is zero. If the system were slightly perturbed from electro-neutrality, an electric field would exist within the system. Charge would move in order to neutralize that field and maintain electro-neutrality.

Before the reaction starts, the solid phase of the volume element is electrically neutral. There is an equal number of moles of H^+ and HSO_4^- ions, so the liquid phase is also electrically neutral. Taken all together, the whole volume element is electrically neutral. The system will take all necessary steps to maintain this electro-neutrality.

3.1.3 Charging the Double Layer

Fig. 3-2 (b) shows two moles of electrons entering the volume element's solid phase. These electrons come from the lead electrode through an external circuit. Now, the solid region has 2 moles of negative charge in it (-2 for short), so the solid phase's electro-neutrality has been perturbed. Because of the two electrons, the solid phase appears negative to the mobile ions in the liquid phase. So, the positive ions in the liquid phase try to get close to the negative solid phase, fig. 3-3 (a). The result is that a double layer capacitor now exists along the solid/liquid interface [27, p.171]. If the solid phase were made of an inert material such as platinum, the story would end here with a double layer capacitor. However, the electrode isn't made of something chemically inert such as platinum, it is made of something reactive like Pb(IV)O_2 which participates in this reaction.



(a) The positively charged H^+ ions go to the solid/liquid interface (aka electrode/electrolyte interface) to get close to the negatively charged electrons. In doing so, they create a double layer capacitor.

(b) The electrode is made of $Pb(IV)O_2$. At the solid/liquid interface, one mole of $Pb(IV)O_2$ is drawn to stand out.

Figure 3-3. The system starts by charging the double layer capacitor at the solid/liquid interface before the faradaic reaction occurs.

3.1.4 Faradaic Reaction

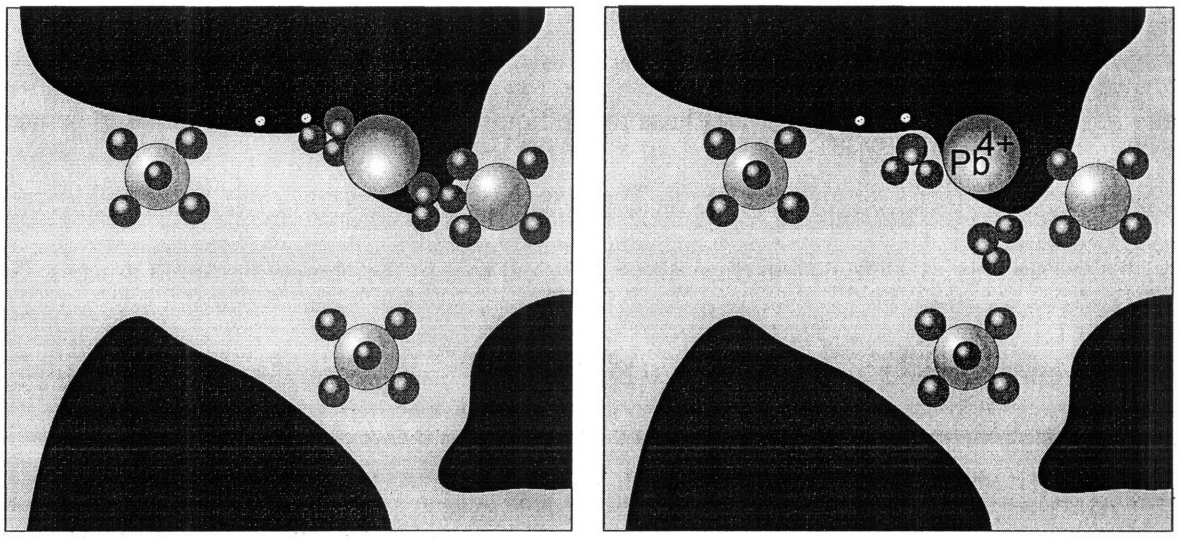
In order to transfer charge from the solid phase to the liquid phase, both phases must be in contact because electrons cannot pass through the liquid. The two phases are in contact only at the solid/liquid interface. Therefore, the solid/liquid interface plays a critical role in the charge/discharge process. So as to help use better understand this charge transfer process, a mole of Pb(IV)O_2 has been drawn at the solid/liquid interface in fig. 3-3 (b).

Fig. 3-4 (a) shows the hydrogen ions bonding with the lead sulfate's oxygen atoms at the solid/liquid interface. In fig. 3-4(b), the reaction has now produced the two moles of water product of reaction as required by eq. (3.1). The reaction proceeds with Pb^{4+} attracting the electrons and the SO_4^{2-} ion, fig. 3-5(a). The Pb(II)SO_4 product needed for reaction eq. (3.1) is now created. Furthermore, the solid phase is now electrically neutral, and the two moles of negative charge have been transfered from the solid phase to the liquid phase.

3.1.5 Transport Regime of Operation

As seen in fig. 3-4 (b), the solid phase is electrically neutral; however, the liquid phase still has a -2 charge. An ionic current is needed in the liquid phase to conduct this -2 charge out of the volume element. There are a number of conceivable compositions of this ionic current. One possible composition might be to bring two moles of H^+ ions into the volume element. Another might be to send two moles of HSO_4^- ions out of the volume element. A third way would be to bring in some H^+ ions and to send out some HSO_4^- ions such that the net charge moved into the volume element would be +2.

The third method is in fact what the system will do. Some positive ions will be transported into the area from neighboring volume elements and some negative ions will be transported out of the volume

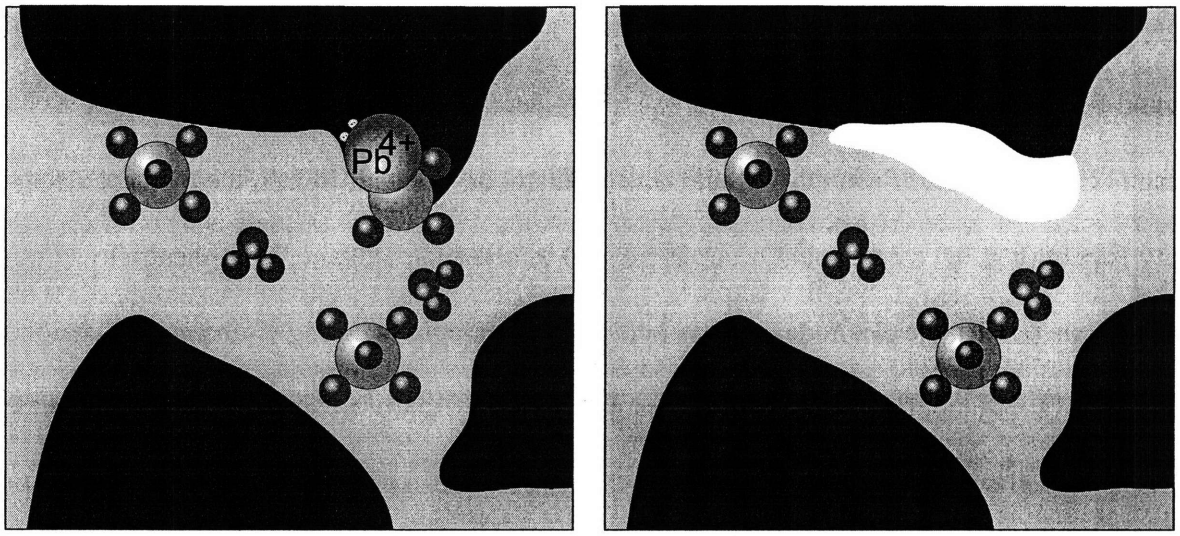


(a) The oxygens of the Pb(IV)O_2 join with the H^+ ions at the solid/liquid interface.

(b) Water is produced.

Figure 3-4. Water is formed as required by eq. (3.1). The solid phase has a plus 2 charge and the liquid phase has a negative 4 charge.

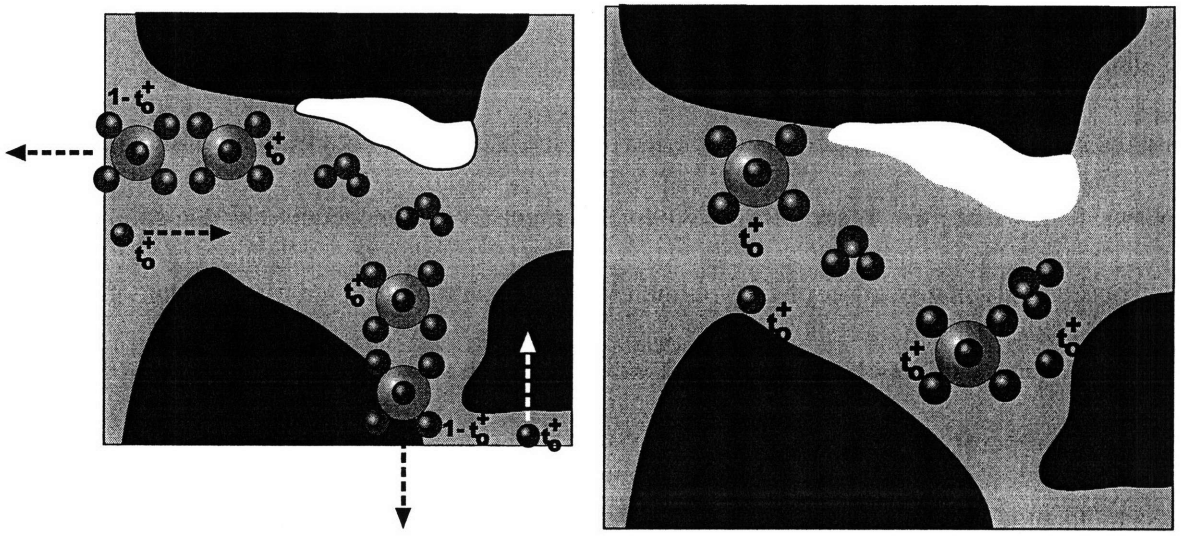
element. Due to the fact that HSO_4^- ions and H^+ ions have different mobilities, the number of H^+ ions that enter the area will not equal the number of HSO_4^- ions that exit the area. The *transference number*, t_i°, is the fraction of the ionic current carried by species i [27,28, p.13,p.65]. In our electrolyte, there are only two charge carriers H^+ and HSO_4^-. H^+ has a transference number of $t_{H^+}^\circ = t_+^\circ$, so HSO_4^- must have a transference number of $t_{HSO_4^-}^\circ = 1 - t_+^\circ$ fig. 3-5 (a) shows $2t_+^\circ$, of H^+ ions moving into the area and $2\left(1 - t_+^\circ\right)$ moles of HSO_4^- ions moving out of the area. In fig. 3-5 (b), all ion transport has stopped and the liquid phase within this area is once again electrically neutral. The -2 charge is well on its way to the lead electrode where it will neutralize the $+2$ charge that exists in the liquid phase near the lead electrode. At this point, the reaction at the Pb(IV)O_2 electrode stops. Concurrently, the reaction at the Pb electrode also stops.



(a) The Pb^{4+} at the solid/liquid interface attracts the electrons and the SO_4^{2-}.

(b) The Pb^{4+}, electrons, and SO_4^{2-} ion bond together to become Pb(II)SO_4 .

Figure 3-5. Two moles of negative charge have now been transfered to the electrolyte from the solid phase.



(a) Positive and negative ions enter and leave the volume element in an attempt to restore electroneutrality.

(b) $2\left(1-t_+^\circ\right)$ moles of negative charge have left the volume element and t_+° moles of positive charge have entered the volume element.

Figure 3-6. Electro-neutrality is maintained.

3.1.6 Discussion

It should be noted that in an actual system, the above described steps would not occur in a such a serial manner and that electro-neutrality would never actually be violated. In fact, the moment electro-neutrality is perturbed, all steps would occur in parallel in order to maintain electro-neutrality. From the diagrams, we see that the electrode operates in three distinct regimes during discharge: the charging of the double layer regime, faradaic transfer of charge across the electrode/electrolyte interface regime, and a transport regime. For the purposes of this thesis, only the faradaic transfer and the transport regimes will be important. This is because the applications looked at for this thesis all involve steady state charge/discharge, so the small time delay introduced by the charging of the double layer will not be consequential.

Next, we see that electrons stay in the solid phase. Protons stay in the liquid phase, and oxygen and SO_4^{2-} molecules cross the solid/liquid interface. This will be important in the next chapter to develop our first elementary cell model.

During the kinetic regime of operation, a faradaic reaction transfers charge from the solid phase to the liquid phase. This causes the fraction of the volume element's volume occupied by the solid phase to change. Consequently, the fraction of the volume element's volume occupied by the liquid phase also changes, but it changes in the opposite direction with that of the solid phase. Our models will have to take this change of porosity into consideration.

As seen in the illustrations, part of the discharge process involves a faradaic or charge transfer reaction. Our equations will have to model this reaction. Because the reaction takes place at the solid/liquid interface, our model's kinetics equation will have to take into account the solid/liquid interface area. The interface area, however, changes with time as the reaction progresses; $Pb(IV)O_2$ is converted into

$Pb(II)SO_4$, which covers the electrode. Anywhere there is $Pb(II)SO_4$, an insulator, the discharge reaction cannot take place, so as time progresses, there is less and less electrode participating in the discharge reaction.

At the beginning of the reaction, we see two moles of negatively charged electrons enter the volume element and at the end of the reaction, we seen two moles of negative charge leave the volume element via an ionic current. During the reaction, charge was neither created nor destroyed, so batteries obey the law of conservation of charge.

At the beginning of the reaction, there were three moles of sulfuric acid. At the end of the reaction, there were only $2t_+^\circ$ moles remaining. The difference, $3-2t_+^\circ$ moles, was either consumed in the reaction or transported out of the volume element, so batteries obey the law of conservation of matter.

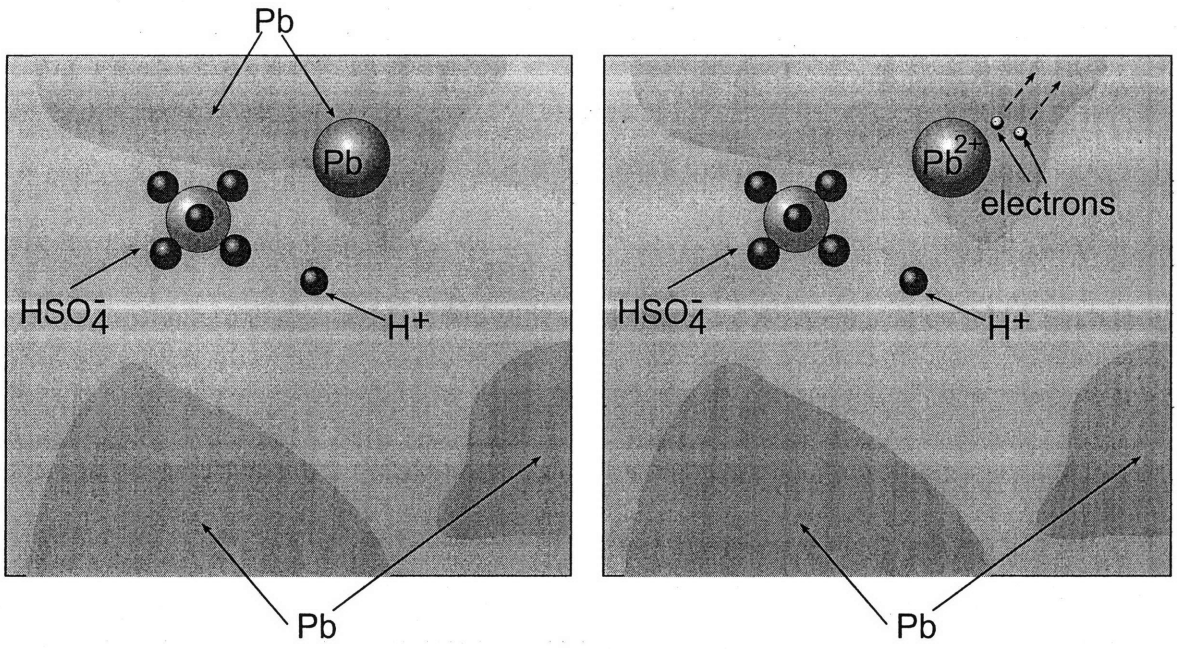
From our diagrams and the discussion, we see that our model will have to have a change of porosity relationship, a kinetics relationship, a conservation of charge relationship and a conservation of matter relationship. We will develop these equations in section 4.1

3.1.7 Illustrations for Lead Electrode

A similar procedure can be used to illustrate the processes going on at the lead electrode, and figs. 3-7–3-10 illustrate one possible path that could happen under cell discharge.

The reaction on the lead (Pb) electrode occurs because of a valence shift on the lead electrode allows the Pb^{2+} to interact with the sulfuric acid in solution.

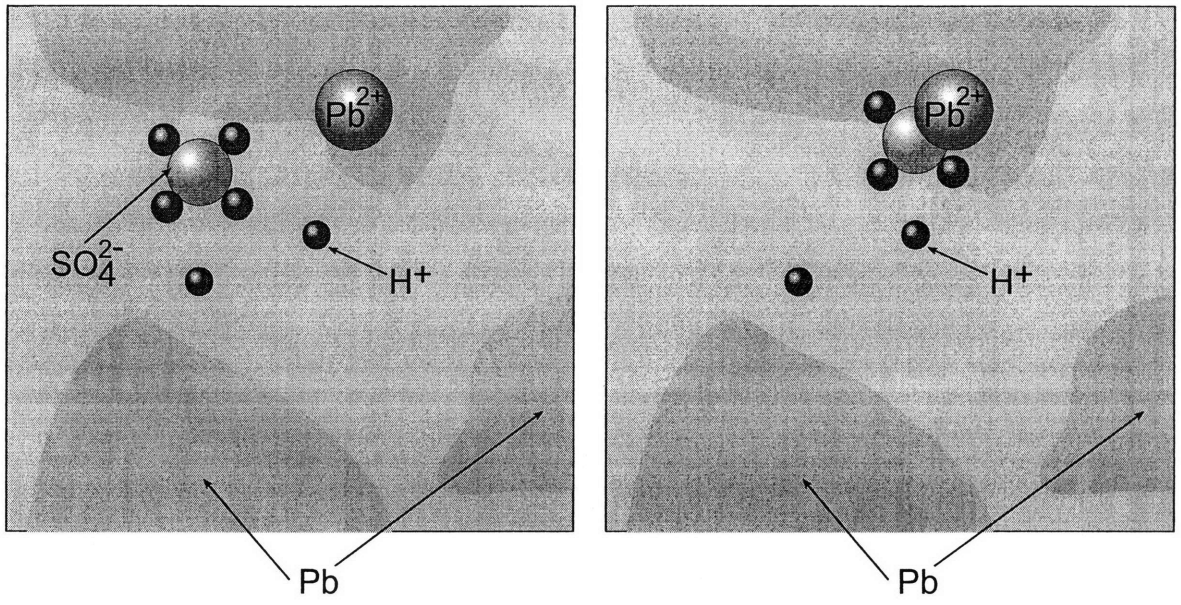
$$Pb \rightarrow Pb^{2+} + 2e^- \tag{3.4}$$



(a) A volume element from the lead (Pb) electrode's porous region before the start of the reaction

(b) To moles of electrons break off from a lead (Pb) atom and leave the volume element.

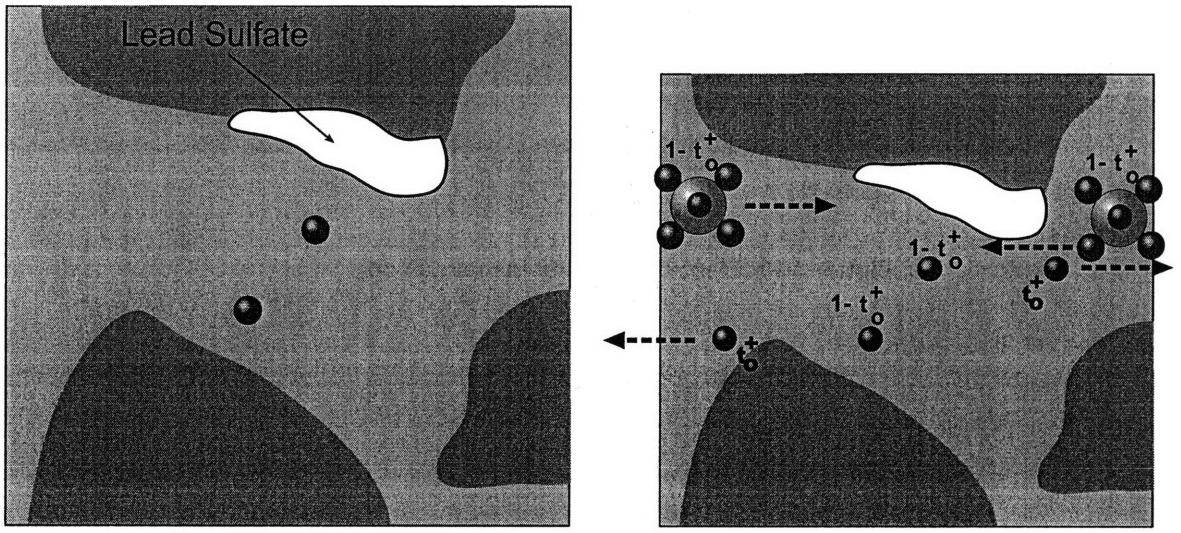
Figure 3-7. The start of the reaction in a small volume element from a lead (Pb) electrode of a lead-acid battery .



(a) A H^+ ion is liberated from a HSO_4^- ion leaving behind a SO_4^{2-} ion.

(b) The SO_4^{2-} ion moves to the solid/liquid interface where it will join the Pb^{2+} in fig.3-9.

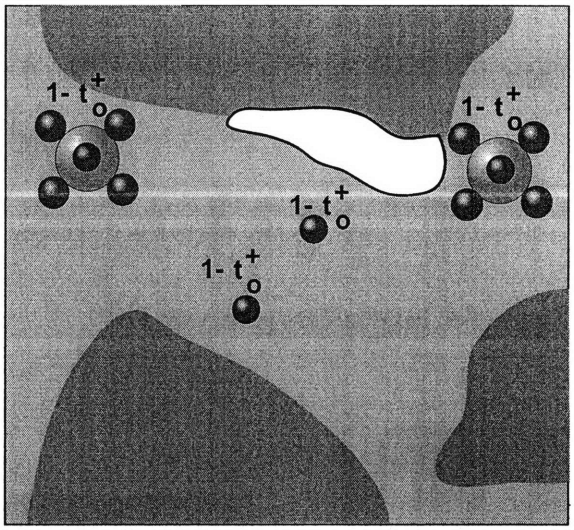
Figure 3-8. Lead sulfate joins with Pb^{2+}.



(a) The SO_4^{2-} and the Pb^2 ion join to form Pb(II)SO_4 . Two H^+ ions remain in the liquid phase.

(b) $2\left(1-t_+^\circ\right)$ mole of HSO_4^- ions enter the liquid phase as $2t_+^\circ$ H^+ ions leave.

Figure 3-9. The porosity changes.



(a) A total of 2 moles of positive charge have left the electrolyte.

Figure 3-10. Electro-neutrality is maintained.

Chapter 4

Two-Volume Model of the Lead Acid Battery

In this chapter, the intuition about the movement of ions and electrons during discharge in a lead acid battery developed in chapter 3 will be translated into a set of equations. These equations will be used to implement a simple two-volume model of the lead acid cell. Although this model will not capture the two-level effects of fig. 1-6(b), it will still be useful in forwarding our understanding of the lead acid cell's physics. A result of great importance, found in section 4.4.3, is that our simple model can be used to argue that, in the absence of very detailed information about how the active area of the electrodes evolves during reaction, (information which is unlikely to be practically obtainable),electrochemical battery models should be used only to understand trends and not to try to predict the performance of a specific battery cell. In section 5.3, the equations developed in this chapter will be extended to develop a 2-dimensional model that captures the two-level effect of fig. 1-6(b).

4.1 A simplified model of a finite volume element taken from the porous region of a lead dioxide electrode.

From section 3.1 we saw that the geometry within any volume element is quite complex. There are many curved arbitrarily located solid particles within each volume element. Modeling these curved interfaces would be quite difficult; however, the important kinetics of the reaction occur at the solid/liquid interface, so it is important to have a model that incorporates a solid/liquid interface.

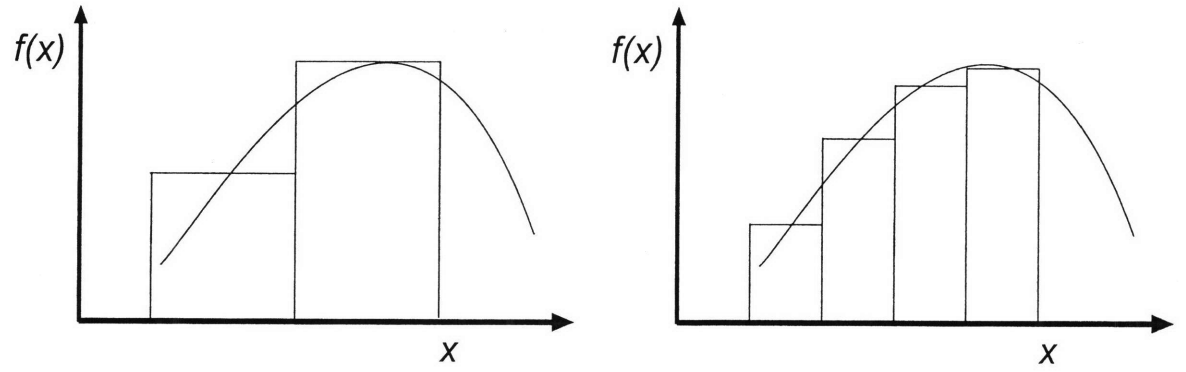
Therefore, we would like to develop a model that does not incorporate the complex geometry but still captures the salient kinetics and transport phenomena. We can achieve this by first remembering some calculus.

In calculus, we learn that if a curve is continuous and differentiable, then it can be reasonably approximated by a sequence of steps, each step consisting of a flat line as seen in fig. 4-1 (a). As the number of steps increases, the approximation improves as seen in fig. 4-1 (b). In much the same way, by looking at part of a pore and focusing in on a very narrow region of the curved solid/liquid interface, it is reasonable to approximate the curved solid/liquid interface within that narrow region by a smooth, flat interface, as seen in fig. 4-2[1].

Fig. 4-3 redraws the foreground volume element of fig. 4-2. The model in fig. 4-3 consists of a solid region and a liquid electrolyte region that share a simple straight and flat interface. Gone is the complicated geometry seen in section 3.1. This simple volume element will serve as the basis from which the equations that govern the kinetics and transport phenomena of reactions eq. (1.1) and eq. (1.2) will be derived.

Specializing the model in fig. 4-3 to the lead dioxide electrode gives fig. 4-4. In the diagram, one can

[1]This model comes as a result of conversations with Professor Wyatt.



(a) A curve being approximated by two steps.

(b) The same curve as in (a) being approximated by four steps.

Figure 4-1. In calculus, differentiable curves can be approximated by steps. The larger number of steps, the better the approximation.

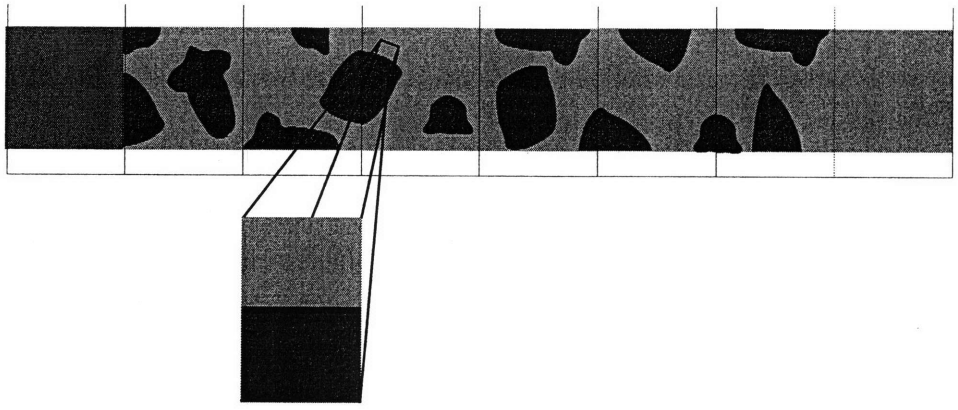


Figure 4-2. The foreground shows the simplified volume element. Note how the curved surface of the porous material (background) is being approximated by a straight plane. For a narrow enough region, this is a reasonable approximation.

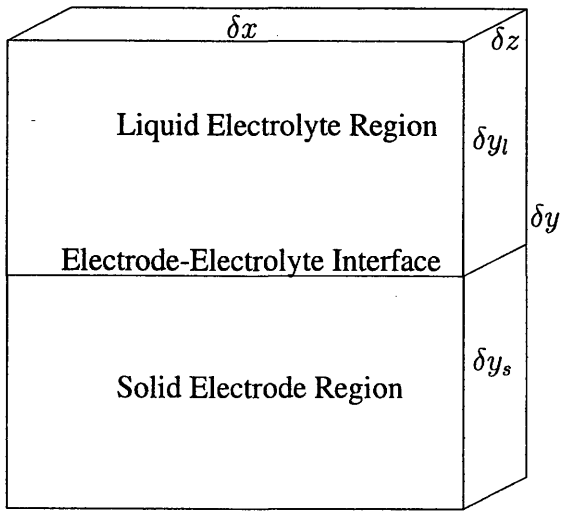


Figure 4-3. A volume element from an electrode's porous region. The lower half consists entirely of solid material. The upper half consists of liquid. If this element were from the lead dioxide electrode, the solid region would consist of $Pb(IV)O_2$ and $Pb(II)SO_4$. If it were from the lead electrode's porous region, the solid region would consist of Pb and $Pb(II)SO_4$. Note that, $\delta y = \delta y_s + \delta y_l$. This simple model still maintains all the elements needed to develop a solid understanding of the battery's electro-chemistry, but it has the added benefit of preventing the complicated geometry seen in fig. 4-2 (b) from obscuring the processes going on during reaction.

see the flux densities of ions into and out of the liquid region, the solid region, and between the liquid and solid regions.

In fig. 4-4, positive fluxes flow in the direction of the arrows. Here j_i is the flux density of species i in $\frac{moles}{cm^2 \cdot sec}$.[2] $[\gamma]$ is the concentration of species γ in $\frac{moles}{cm^3}$. The incremental distances δx, δy, δy_s, δy_l, δz all have units of centimeters and $\delta y = \delta y_s + \delta y_l$. Note that all z and y direction fluxes have been ignored. This is done for the sole purpose of simplifying the derivation of the equations. This model can be readily expanded to include fluxes which flow in two or three dimensions if one feels like doing so.

The fluxes shown in fig. 4-4 where selected because of their participation in the discharge of the

[2]A flux N_i is defined as $N_i = j_i \text{Area}$

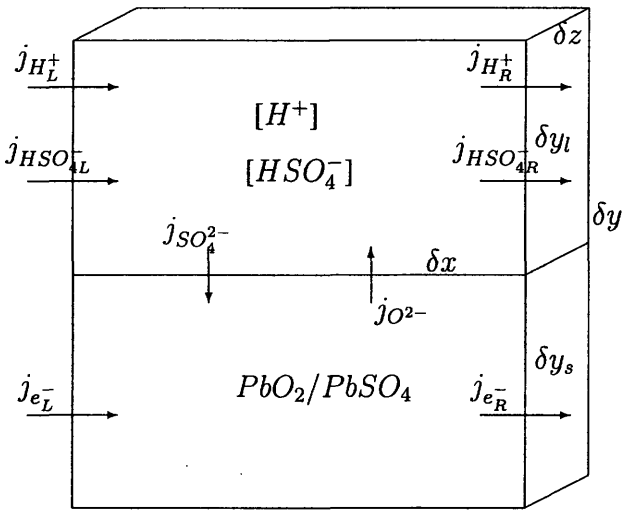


Figure 4-4. A specialization of the model in fig. 4-3 to the lead dioxide area. Here only x-direction flux densities are shown.

Pb(IV)O_2 electrode as was seen in section 3.1. Fig. 3-2 (b) shows that there must be a flow of electrons in the solid phase. In fig. 4-4, the flux densities of electrons into and out of our volume element are labeled $j_{e_L^-}$ and $j_{e_R^-}$. O^{2-} ions cross the solid/liquid interface in fig. 3-4 (a), so this flux density $j_{O^{2-}}$ is also included in our molecular model. Fig. 3-5 (a) shows an SO_4^{2-} ion traversing the phase boundary, so this flux $j_{SO_4^{2-}}$ is also shown in fig. 4-4. Finally, there are two pairs of flux densities, $j_{H_L^+}$ with $j_{HSO_{4L}^-}$ and $j_{H_R^+}$ with $j_{HSO_{4R}^-}$ that are entering and leaving the left and right sides of our model. These fluxes come from the fact that fig. 3-6 shows flows of both H^+ and HSO_4^- ions.

The mathematical equations that describe the reaction at the Pb(IV)O_2 plate will be derived from the flows seen in fig. 4-4, the chemical reaction described in eq. (1.1) and the assumption of electro-neutrality eq. (3.3).

4.2 Derivation of fundamental equations for the reaction at the lead dioxide electrode

In this section, five equations will be derived. These are the Change of Porosity equation in section 4.2.1, the Conservation of Charge equation in section 4.2.2, Ohm's Law in Solution in section 4.2.3, Conservation of Matter equation in section 4.2.4 and the Electrode Kinetics equation in section 4.2.5. Together these equations will describe the behavior of the four state variables ε, C, ϕ_s, and ϕ_l. As the equations in this thesis can become quite complex, color, where available, is used to draw the reader's attention to these variables.

4.2.1 Change of Porosity

The porosity of a volume element denoted by ε and defined to be the ratio of the void volume[3] within a volume element to the total volume of the volume element [29]. As seen in the illustrations of section 3.1, the amount of space within a volume element required by the solid phase changes due to the fact that Pb(IV)O_2 molecules and Pb(II)SO_4 molecules are not the same size. During discharge Pb(IV)O_2 molecules are converted into Pb(II)SO_4 molecules, and during charging, Pb(II)SO_4 molecules are converted into Pb(IV)O_2 molecules. This conversion of one type of molecule into the other changes the amount of space occupied by the solid phase within the volume element. Consequently, the amount of space occupied by the volume element's liquid phase also changes. Hence, because the total volume of a volume element is assumed to be fixed, the porosity of the volume element must also change.

In the simple cell, because δx, δy, and δz are all known fixed quantities, the solid region's volume can only change by changing δy_s which also changes δy_l and therefore affects the volume of the liquid

[3]For this model, the void volume is assumed to be filled with electrolyte.

region. Starting with,

$$\frac{\Delta \text{Volume Solid Phase}}{\Delta t} = \frac{\Delta \text{Volume Occupied by } \text{Pb(II)SO}_4}{\Delta t} + \frac{\Delta \text{Volume Occupied by } \text{Pb(IV)O}_2}{\Delta t}$$

$$= \left(\frac{volume}{mole}\right)_{\text{Pb(II)SO}_4} \left(\frac{\Delta \#\text{moles Pb(II)SO}_4}{\Delta t}\right) + \left(\frac{volume}{mole}\right)_{\text{Pb(IV)O}_2} \left(\frac{\Delta \#\text{moles Pb(IV)O}_2}{\Delta t}\right)$$

$$= \left(\frac{MW_{\text{Pb(II)SO}_4}}{\rho_{\text{Pb(II)SO}_4}}\right) \left(\frac{\Delta \#\text{moles Pb(II)SO}_4}{\Delta t}\right) + \left(\frac{MW_{\text{Pb(IV)O}_2}}{\rho_{\text{Pb(IV)O}_2}}\right) \left(\frac{\Delta \#\text{moles Pb(IV)O}_2}{\Delta t}\right) \tag{4.1}$$

where MW_i is the molecular weight of species i and ρ_i is the density of species i. The time rate of change of the number of moles of Pb(IV)O_2 and Pb(II)SO_4 can be related to each other and to the divergence of the electron flow by using eq. (1.1) to give us

$$-\frac{\Delta \#\text{moles Pb(II)SO}_4}{\Delta t} = \frac{\Delta \#\text{moles Pb(IV)O}_2}{\Delta t} = \left(\frac{j_{e_R^-} - j_{e_L^-}}{2}\right) \delta z \delta y_s \tag{4.2}$$

Substituting the results of eq. (4.2) into eq. (4.1) allows the following derivation:

$$
\begin{aligned}
\frac{\Delta\text{Volume Solid Area}}{\Delta t} &= -\left[\left(\frac{MW_{\text{Pb(II)SO}_4}}{\rho_{\text{Pb(II)SO}_4}}\right) - \left(\frac{MW_{\text{Pb(IV)O}_2}}{\rho_{\text{Pb(IV)O}_2}}\right)\right]\left(\frac{j_{e_R^-} - j_{e_L^-}}{2}\right)\delta z \delta y_s \\
\frac{\delta x \delta z \Delta\left(\delta y_s\right)}{\Delta t} &= -\left[\left(\frac{MW_{\text{Pb(II)SO}_4}}{\rho_{\text{Pb(II)SO}_4}}\right) - \left(\frac{MW_{\text{Pb(IV)O}_2}}{\rho_{\text{Pb(IV)O}_2}}\right)\right]\left(\frac{j_{e_R^-} - j_{e_L^-}}{2}\right)\delta z \delta y_s \\
\frac{\delta x \delta z \delta y \Delta\left(\frac{\delta y - \delta y_l}{\delta y}\right)}{\Delta t} &= -\left[\left(\frac{MW_{\text{Pb(II)SO}_4}}{\rho_{\text{Pb(II)SO}_4}}\right) - \left(\frac{MW_{\text{Pb(IV)O}_2}}{\rho_{\text{Pb(IV)O}_2}}\right)\right]\left(\frac{j_{e_R^-} - j_{e_L^-}}{2}\right)\delta z \delta y \left(\frac{\delta y - \delta y_l}{\delta y}\right) \\
\frac{\Delta\left(\frac{-\delta y_l}{\delta y}\right)}{\Delta t} &= -\left[\left(\frac{MW_{\text{Pb(II)SO}_4}}{\rho_{\text{Pb(II)SO}_4}}\right) - \left(\frac{MW_{\text{Pb(IV)O}_2}}{\rho_{\text{Pb(IV)O}_2}}\right)\right]\left(\frac{j_{e_R^-} - j_{e_L^-}}{2\delta x}\right)\left(\frac{\delta y - \delta y_l}{\delta y}\right) \\
\frac{\Delta\left(-\varepsilon\right)}{\Delta t} &= -\left[\left(\frac{MW_{\text{Pb(II)SO}_4}}{\rho_{\text{Pb(II)SO}_4}}\right) - \left(\frac{MW_{\text{Pb(IV)O}_2}}{\rho_{\text{Pb(IV)O}_2}}\right)\right]\left(\frac{j_{e_R^-} - j_{e_L^-}}{2\delta x}\right)(1-\varepsilon)
\end{aligned}
\tag{4.3}
$$

Dividing the last of eqns. (4.3) by -1 gives

$$
\boxed{\frac{\Delta\varepsilon}{\Delta t} = \left[\left(\frac{MW_{\text{Pb(II)SO}_4}}{\rho_{\text{Pb(II)SO}_4}}\right) - \left(\frac{MW_{\text{Pb(IV)O}_2}}{\rho_{\text{Pb(IV)O}_2}}\right)\right]\left(\frac{j_{e_R^-} - j_{e_L^-}}{2\delta x}\right)(1-\varepsilon)} \tag{4.4}
$$

4.2.2 Conservation of Charge

As noted in section 3.1.6, batteries obey the law of conservation of charge, so we need an equation to represent this law. Because of conservation of charge, there must also be a relationship between the number of electrons produced or consumed within a volume element and the number of H^+ and HSO_4^- ions produce or consumed within the same volume element. Simply put, any charge that leaves the solid phase within a volume element must enter the liquid phase of the same volume element. The converse also holds true.

The relationship between the number of electrons produced or consumed within a volume element and the total number of SO_4^{2-} ions that cross the solid-liquid interface can be written as:

$$\left(\frac{j_{e_R^-} - j_{e_L^-}}{2}\right)\delta z\delta y_s = -j_{SO_4^{2-}}\delta z\delta x \tag{4.5}$$

This relationship can be seen by first looking at fig. 3-2 where two moles of electrons enter the volume and then seeing the one mole of SO_4^{2-} in fig. 3-5(a) cross the solid/liquid interface to become part of a Pb(II)SO_4 molecule.

Expanding on eq. (4.5) to include the flux of O^{2-} across the solid-liquid interface gives

$$\left(\frac{j_{e_R^-} - j_{e_L^-}}{2}\right)\delta z\delta y_s = -j_{SO_4^{2-}}\delta z\delta x = -\frac{j_{O^{2-}}}{2}\delta z\delta x \tag{4.6}$$

Next, again using eq. (1.1) as our guide, we write down how the total number of H^+ and HSO_4^- ions changes with time.

$$\frac{\Delta\left([H^+]\delta x\delta y_l\delta z\right)}{\Delta t} = -\left(j_{H_R^+} - j_{H_L^+}\right)\delta z\delta y_l - 2j_{O^{2-}}\delta z\delta x + j_{SO_4^{2-}}\delta z\delta x \tag{4.7}$$

$$\frac{\Delta\left([HSO_4^-]\delta x\delta y_l\delta z\right)}{\Delta t} = -\left(j_{HSO_{4R}^-} - j_{HSO_{4L}^-}\right)\delta z\delta y_l - j_{SO_4^{2-}}\delta z\delta x \tag{4.8}$$

Substituting from eq. (4.6) into eq. (4.7) and eq. (4.8) and then simplifying gives

$$\frac{\Delta\left([H^+]\delta x\delta y_l\delta z\right)}{\Delta t} = -\left(j_{H_R^+} - j_{H_L^+}\right)\delta z\delta y_l + \frac{3}{2}\left(j_{e_R^-} - j_{e_L^-}\right)\delta y_s\delta z \tag{4.9}$$

$$\frac{\Delta\left([HSO_4^-]\delta x\delta y_l\delta z\right)}{\Delta t} = -\left(j_{HSO_{4R}^-} - j_{HSO_{4L}^-}\right)\delta z\delta y_l + \left(j_{e_R^-} - j_{e_L^-}\right)\delta y_s\delta z \tag{4.10}$$

writing both eq. (4.9) and eq. (4.10) in terms of porosity gives, i.e. using $\varepsilon = \frac{\delta y_l}{\delta y}$

$$\frac{\Delta\left([H^+]\varepsilon\right)}{\Delta t}\delta x\delta y\delta z = -\left(j_{H_R^+} - j_{H_L^+}\right)\varepsilon\delta y\delta z + \frac{3}{2}\left(j_{e_R^-} - j_{e_L^-}\right)(1-\varepsilon)\,\delta y\delta z \tag{4.11}$$

$$\frac{\Delta\left([HSO_4^-]\varepsilon\right)}{\Delta t}\delta x\delta y\delta z = -\left(j_{HSO_{4R}^-} - j_{HSO_{4L}^-}\right)\varepsilon\delta y\delta z + \frac{1}{2}\left(j_{e_R^-} - j_{e_L^-}\right)(1-\varepsilon)\,\delta y\delta z \tag{4.12}$$

In order for the liquid phase to maintain electro-neutrality, $\frac{\Delta\left([H^+]\varepsilon\right)}{\Delta t}\delta x\delta y\delta z = \frac{\Delta\left([HSO_4^-]\varepsilon\right)}{\Delta t}\delta x\delta y\delta z$ must be true, so we can set the right sides of eq. (4.11) equal to eq. (4.12) to get

$$\begin{aligned}-\left(j_{H_R^+} - j_{H_L^+}\right)\varepsilon\delta y\delta z + \frac{3}{2}\left(j_{e_R^-} - j_{e_L^-}\right)(1-\varepsilon)\,\delta y\delta z& \\ = -\left(j_{HSO_{4R}^-} - j_{HSO_{4L}^-}\right)\varepsilon\delta y\delta z + \frac{1}{2}\left(j_{e_R^-} - j_{e_L^-}\right)(1-\varepsilon)\,\delta y\delta z&\end{aligned} \tag{4.13}$$

dividing both sides of the equation by $\delta x\delta y\delta z$ and combining like terms

$$\left[-\left(\frac{j_{H_R^+} - j_{H_L^+}}{\delta x}\right) + \left(\frac{j_{HSO_{4R}^-} - j_{HSO_{4L}^-}}{\delta x}\right)\right]\varepsilon = -\left(\frac{j_{e_R^-} - j_{e_L^-}}{\delta x}\right)(1-\varepsilon) \tag{4.14}$$

The right side of eq. (4.14) is simply the divergence of the electron fluxes. The left side is the divergence of the ionic fluxes. Grouping liquid region fluxes not by type of ion, but by the side of the volume element into which the ion flows gives, and moving the electronic flux terms to the left side of the equation gives,

$$\left[\left(\frac{j_{HSO_{4R}^-} - j_{H_R^+}}{\delta x}\right) - \left(\frac{j_{HSO_{4L}^-} - j_{H_L^+}}{\delta x}\right)\right] \varepsilon + \left(\frac{j_{e_R^-} - j_{e_L^-}}{\delta x}\right)(1-\varepsilon) = 0 \tag{4.15}$$

Now we will define net negative charge flowing in the liquid phase as

$$j_{l_i} = j_{HSO_{4i}^-} - j_{H_i^+} \tag{4.16}$$

where i stands for the side of the volume element from which the charge is entering. Rewriting eq. (4.15) in terms of eq. (4.16) gives

$$\boxed{\left(\frac{j_{l_R} - j_{l_L}}{\delta x}\right)\varepsilon + \left(\frac{j_{e_R^-} - j_{e_L^-}}{\delta x}\right)(1-\varepsilon) = 0} \tag{4.17}$$

Equation eq. (4.17) shows that charge is conserved in the volume element. This means that the total number of negative charges produced in the liquid phase is equal to the negative of the amount of negative charge produced in the solid phase and vice versa.

4.2.3 Ohm's Law in Solution

In fig. 3-6, we saw ions leaving and entering our volume element. There are three ways in which ions move: migration, diffusion, and convection. Our present model neglects convection, which is not significant in the pores of the battery electrodes.

Ohm's Law in solution is very similar to the well known Ohm's Law from circuit theory in which charge is transported by electrons. Ohm's law is expressed as

$$\begin{aligned} J &= \sigma \overrightarrow{\mathbf{E}} \\ &= \sigma \left(-\frac{\partial \phi}{\partial x} \right) \end{aligned} \tag{4.18}$$

Where J is current density in terms of $\frac{Coulombs}{cm^2 \cdot s}$, σ is the conductivity $\left(\frac{Coulombs}{Volts \cdot cm \cdot s}\right)$ through which the electrons are flowing, and $\overrightarrow{\mathbf{E}}$ $\left(\frac{Volts}{cm}\right)$ is the electric field in which the electrons are moving.

For the purpose of discussion, all derivatives will be made in the x direction and positive fluxes will be flowing in the positive x direction.

Ohm's Law in solution is very similar to eq. (4.18) with a few modifications that will now be explored. In the electrochemical cell's electrolyte, charge is not transported by electrons but instead by ions, and depending on the solution, there may be multiple types of ions with different charges and signs. For this discussion, H_2SO_4 is assumed to dissociate into only H^+ and HSO_4^- ions.

The total number of ions of a certain species passing through an area per unit time is

$$N_i = [i]\, v_i A \tag{4.19}$$

Where N_i has units of $\frac{moles}{s}$,$[i]$ is $\frac{moles}{cm^3}$,v_i is in $\frac{cm}{s}$, and A is in cm. The flux per unit area j_i, which has units of $\frac{moles}{cm^2 \cdot s}$, is

$$\begin{aligned} j_i &= \frac{N_i}{A} \\ &= [i]\, v_i \end{aligned} \tag{4.20}$$

the current passing through a unit area due to the movement of species i will be

$$z_i \mathbf{F} j_i = z_i \mathbf{F} [i] v_i \tag{4.21}$$

Where $\mathbf{F}$ is Faraday's constant $\approx 96485 \frac{Coulomb}{mole}$. The velocity of species i, v_i, is representative not of each individual ion, but of how ions of species type i on average behave in the medium. It is well known that the velocity of a charged species in the presence of a constant field is proportional to the field strength. In the lead acid cell's electrolyte, the field is the negative of the gradient of the electrochemical potential, so

$$v_i = u_i \left(-\frac{\partial \overline{\mu}_i}{\partial x} \right) \tag{4.22}$$

where u_i is the mobility of species i. The mobility reflects the ease with which an ion of species i can move through the medium in which it exists. This gives the current due to species i as

$$z_i \mathbf{F} j_i = z_i \mathbf{F} [i] u_i \left(-\frac{\partial \overline{\mu}_i}{\partial x} \right) \tag{4.23}$$

The current passing through a unit area within the electrolyte will be the sum of the current due to the movement of H^+ ions and the current due to the movement of HSO_4^- ion as seen below,

$$\left(z_{\mathrm{H^+}} \mathbf{F} j_{\mathrm{H^+}} + z_{\mathrm{HSO_4^-}} \mathbf{F} j_{\mathrm{HSO_4^-}} \right) = z_{\mathrm{H^+}} \mathbf{F} \left[\mathrm{H^+} \right] u_{\mathrm{H^+}} \left(-\frac{\partial \overline{\mu}_{\mathrm{H^+}}}{\partial x} \right) + z_{\mathrm{HSO_4^-}} \mathbf{F} \left[\mathrm{HSO_4^-} \right] u_{\mathrm{HSO_4^-}} \left(-\frac{\partial \overline{\mu}_{\mathrm{HSO_4^-}}}{\partial x} \right) \tag{4.24}$$

The electrochemical potential of a species i, $\overline{\mu}_i$ can be represented by an electric potential ϕ plus a chemical potential μ [28]

$$\overline{\mu}_i = z_i \mathbf{F}\phi + \mu_i \tag{4.25}$$

where ϕ is electric potential, $\mathbf{F}$ is Faraday's constant, z_i is the charge carried by an ion of species i and μ_i is the chemical potential of species i. Substitution of eq. (4.25) into eq. (4.24) yields,

$$\begin{aligned}\left(z_{\mathrm{H^+}}\mathbf{F}j_{\mathrm{H^+}} + z_{\mathrm{HSO_4^-}}\mathbf{F}j_{\mathrm{HSO_4^-}}\right) = {} & z_{\mathrm{H^+}}\mathbf{F}\left[\mathrm{H^+}\right]u_{\mathrm{H^+}}\left(-\left(z_{\mathrm{H^+}}\mathbf{F}\frac{\partial\phi}{\partial x} + \frac{\partial\mu_{\mathrm{H^+}}}{\partial x}\right)\right) \\ & + z_{\mathrm{HSO_4^-}}\mathbf{F}\left[\mathrm{HSO_4^-}\right]u_{\mathrm{HSO_4^-}}\left(-\left(z_{\mathrm{HSO_4^-}}\mathbf{F}\frac{\partial\phi}{\partial x} + \frac{\partial\mu_{\mathrm{HSO_4^-}}}{\partial x}\right)\right)\end{aligned} \tag{4.26}$$

rearranging gives

$$\begin{aligned}\left(z_{\mathrm{H^+}}\mathbf{F}j_{\mathrm{H^+}} + z_{\mathrm{HSO_4^-}}\mathbf{F}j_{\mathrm{HSO_4^-}}\right) = {} & z_{\mathrm{H^+}}^2\mathbf{F}^2\left[\mathrm{H^+}\right]u_{\mathrm{H^+}}\left(-\frac{\partial\phi}{\partial x}\right) + z_{\mathrm{HSO_4^-}}^2\mathbf{F}^2\left[\mathrm{HSO_4^-}\right]u_{\mathrm{HSO_4^-}}\left(-\frac{\partial\phi}{\partial x}\right) \\ & + z_{\mathrm{H^+}}\mathbf{F}\left[\mathrm{H^+}\right]u_{\mathrm{H^+}}\left(-\frac{\partial\mu_{\mathrm{H^+}}}{\partial x}\right) + z_{\mathrm{HSO_4^-}}\mathbf{F}\left[\mathrm{HSO_4^-}\right]u_{\mathrm{HSO_4^-}}\left(-\frac{\partial\mu_{\mathrm{HSO_4^-}}}{\partial x}\right) \\ = {} & \left(z_{\mathrm{H^+}}^2\mathbf{F}^2\left[\mathrm{H^+}\right]u_{\mathrm{H^+}} + z_{\mathrm{HSO_4^-}}^2\mathbf{F}^2\left[\mathrm{HSO_4^-}\right]u_{\mathrm{HSO_4^-}}\right)\left(-\frac{\partial\phi}{\partial x}\right) \\ & + z_{\mathrm{H^+}}\mathbf{F}\left[\mathrm{H^+}\right]u_{\mathrm{H^+}}\left(-\frac{\partial\mu_{\mathrm{H^+}}}{\partial x}\right) + z_{\mathrm{HSO_4^-}}\mathbf{F}\left[\mathrm{HSO_4^-}\right]u_{\mathrm{HSO_4^-}}\left(-\frac{\partial\mu_{\mathrm{HSO_4^-}}}{\partial x}\right)\end{aligned} \tag{4.27}$$

The chemical potential μ_i can be written

$$\mu_i = \mathrm{RT}ln\left(a_i\right) \tag{4.28}$$

where R is the universal gas constant, T is absolute temperature and a_i is the activity of species i.

Activity is a function of the concentration of species i and can be written as $a = \gamma [i]$ where γ is the activity coefficient and $[i]$ is the concentration of species i giving

$$\mu_i = \mathrm{RT} ln (\gamma_i [i]) \tag{4.29}$$

If there is no change in the concentration of chemical species i then $\frac{\partial \mu_i}{\partial x} = 0$ and eq. (4.27) reduces to

$$\left(z_{\mathrm{H^+}} \mathbf{F} j_{\mathrm{H^+}} + z_{\mathrm{HSO_4^-}} \mathbf{F} j_{\mathrm{HSO_4^-}}\right) = \left(z_{\mathrm{H^+}}^2 \mathbf{F}^2 \left[\mathrm{H^+}\right] u_{\mathrm{H^+}} + z_{\mathrm{HSO_4^-}}^2 \mathbf{F}^2 \left[\mathrm{HSO_4^-}\right] u_{\mathrm{HSO_4^-}}\right) \left(-\frac{\partial \phi}{\partial x}\right) \tag{4.30}$$

which is a function of the electric field alone, and thus simply the familiar ohm's law given in eq. (4.18). Pattern matching with eq. (4.18) allows us to write the electrolyte conductivity κ as

$$\kappa = \left(z_{\mathrm{H^+}}^2 \mathbf{F}^2 \left[\mathrm{H^+}\right] u_{\mathrm{H^+}} + z_{\mathrm{HSO_4^-}}^2 \mathbf{F}^2 \left[\mathrm{HSO_4^-}\right] u_{\mathrm{HSO_4^-}}\right) \tag{4.31}$$

which, fortunately, is a specialization of the definition of conductivity given in [27] to our system. Substituting the definition of conductivity into eq. (4.27) gives

$$\begin{aligned}\left(z_{\mathrm{H^+}} \mathbf{F} j_{\mathrm{H^+}} + z_{\mathrm{HSO_4^-}} \mathbf{F} j_{\mathrm{HSO_4^-}}\right) = {} & \kappa \left(-\frac{\partial \phi}{\partial x}\right) \\ & + \kappa \frac{z_{\mathrm{H^+}} \mathbf{F} \left[\mathrm{H^+}\right] u_{\mathrm{H^+}}}{\left(z_{\mathrm{H^+}}^2 \mathbf{F}^2 \left[\mathrm{H^+}\right] u_{\mathrm{H^+}} + z_{\mathrm{HSO_4^-}}^2 \mathbf{F}^2 \left[\mathrm{HSO_4^-}\right] u_{\mathrm{HSO_4^-}}\right)} \left(-\frac{\partial \mu_{\mathrm{H^+}}}{\partial x}\right) \\ & + \kappa \frac{z_{\mathrm{HSO_4^-}} \mathbf{F} \left[\mathrm{HSO_4^-}\right] u_{\mathrm{HSO_4^-}}}{\left(z_{\mathrm{H^+}}^2 \mathbf{F}^2 \left[\mathrm{H^+}\right] u_{\mathrm{H^+}} + z_{\mathrm{HSO_4^-}}^2 \mathbf{F}^2 \left[\mathrm{HSO_4^-}\right] u_{\mathrm{HSO_4^-}}\right)} \left(-\frac{\partial \mu_{\mathrm{HSO_4^-}}}{\partial x}\right)\end{aligned} \tag{4.32}$$

In section 3.1.5, transference number of a species was defined to be the fraction of current carried by that species. This is because physically, the t_+° the contribution to κ of a species [27, p.275] means that t_+° is

$$t_+^\circ = \frac{z_{H^+}^2 \mathbf{F}^2 \left[H^+\right] u_{H^+}}{\left(z_{H^+}^2 \mathbf{F}^2 \left[H^+\right] u_{H^+} + z_{HSO_4^-}^2 \mathbf{F}^2 \left[HSO_4^-\right] u_{HSO_4^-}\right)} \tag{4.33}$$

substituting eq. (4.33) into eq. (4.32) gives

$$\begin{aligned}
\left(z_{H^+}\mathbf{F}j_{H^+} + z_{HSO_4^-}\mathbf{F}j_{HSO_4^-}\right) &= \kappa\left(-\frac{\partial\phi}{\partial x} + \frac{t_+^\circ}{z_{H^+}\mathbf{F}}\left(-\frac{\partial\mu_{H^+}}{\partial x}\right) + \frac{1-t_+^\circ}{z_{HSO_4^-}\mathbf{F}}\left(-\frac{\partial\mu_{HSO_4^-}}{\partial x}\right)\right) \\
\left(\mathbf{F}j_{H^+} - \mathbf{F}j_{HSO_4^-}\right) &= -\kappa\left(\frac{\partial\phi}{\partial x} + \frac{t_+^\circ}{\mathbf{F}}\left(\frac{\partial\mu_{H^+}}{\partial x}\right) + \frac{1-t_+^\circ}{-\mathbf{F}}\left(\frac{\partial\mu_{HSO_4^-}}{\partial x}\right)\right) \\
\left(\mathbf{F}j_{HSO_4^-} - \mathbf{F}j_{H^+}\right) &= \kappa\left(\frac{\partial\phi}{\partial x} + \frac{t_+^\circ}{\mathbf{F}}\left(\frac{\partial\mu_{H^+}}{\partial x}\right) - \frac{\left(1-t_+^\circ\right)}{\mathbf{F}}\left(\frac{\partial\mu_{HSO_4^-}}{\partial x}\right)\right)
\end{aligned} \tag{4.34}$$

substituting eq. (4.29) into eq. (4.34) gives

$$\begin{aligned}
\left(j_{HSO_4^-} - j_{H^+}\right)\mathbf{F} &= \kappa\left(\frac{\partial\phi}{\partial x} + t_+^\circ\frac{\mathrm{RT}}{\mathbf{F}}\frac{\partial ln\left(a_{H^+}\right)}{\partial x} - \left(1-t_+^\circ\right)\frac{\mathrm{RT}}{\mathbf{F}}\frac{\partial ln\left(a_{HSO_4^-}\right)}{\partial x}\right) \\
&= \kappa\left(\frac{\partial\phi}{\partial x} + t_+^\circ\frac{\mathrm{RT}}{\mathbf{F}}\frac{\partial ln\left(\gamma_{H^+}\left[H^+\right]\right)}{\partial x} - \left(1-t_+^\circ\right)\frac{\mathrm{RT}}{\mathbf{F}}\frac{\partial ln\left(\gamma_{HSO_4^-}\left[HSO_4^-\right]\right)}{\partial x}\right)
\end{aligned} \tag{4.35}$$

As is commonly done in the literature, we will assume unit activity but recognize that this assumption is actually only valid for dilute solutions, whereas the sulfuric acid found in a lead acid battery is not a dilute solution. Therefore, this approximation may lead to some error. Next, realizing that, because of electro-neutrality, the concentrations of H^+ and HSO_4^- are the same and equivalent to the electrolyte

concentration C, gives

$$\left(j_{HSO_4^-} - j_{H^+}\right)\mathbf{F} = \kappa\left(\frac{\partial\phi}{\partial x} + \left(2t_+^\circ - 1\right)\frac{\mathrm{RT}}{\mathbf{F}}\frac{\partial ln\left(C\right)}{\partial x}\right) \tag{4.36}$$

dividing both sides by κ and substituting $j_l = \left(j_{HSO_4^-} - j_{H^+}\right)$ gives the final form of Ohm's Law in solution

$$\boxed{\frac{j_l\mathbf{F}}{\kappa} = \frac{\partial\phi}{\partial x} - \left(1 - 2t_+^\circ\right)\frac{\mathrm{RT}}{\mathbf{F}}\frac{\partial ln\left(C\right)}{\partial x}} \tag{4.37}$$

4.2.4 Conservation of Matter

The law of conservation of matter is used to keep track of how the concentration of the electrolyte within the volume changes with time. We start with keeping track of how the total number of moles of H^+ and HSO_4^- ions change with time. The time rate of change of the total number of H^+ ions within our volume element is given by eq. (4.11). Eq. 4.12 gives the time rate of change of the number of moles of HSO_4^- ions. Both equations are reproduced here in eq. (4.38) and eq. (4.39) for ease of reference.

$$\frac{\Delta\left([H^+]\delta x\delta y_l\delta z\right)}{\Delta t} = -\left(j_{H_R^+} - j_{H_L^+}\right)\delta z\delta y_l + \frac{3}{2}\left(j_{e_R^-} - j_{e_L^-}\right)\delta y_s\delta z \tag{4.38}$$

$$\frac{\Delta\left([HSO_4^-]\delta x\delta y_l\delta z\right)}{\Delta t} = -\left(j_{HSO_{4R}^-} - j_{HSO_{4L}^-}\right)\delta z\delta y_l + \left(j_{e_R^-} - j_{e_L^-}\right)\delta y_s\delta z \tag{4.39}$$

The first term on the right of both eq. (4.38) and eq. (4.39) describes the change in the total number of a species due to the combined action of diffusion and migration. The second term on the right describes the amount of ions consumed by the reaction. Initially, expressions for the influence of diffusion and

migration will be developed.

The H^+ ionic fluxes in eq. (4.38) in the term just to the right of the equal sign are due to the transport phenomena of diffusion and migration. Using a first difference approximation for the derivative, the H^+ fluxes can be written in the forms found in eq. (4.40) and eq. (4.41). Here $[H^+]$ with the $(n-1)$ subscript means the concentration of H^+ ions in the small volume element to the left of the present volume element. The (n) subscript is the index of the volume element presently under consideration. $(n+1)$ is the index of the volume element to the right of volume element n. All volume elements are in the same horizontal slice from the electrode like those seen in fig. 3-1.

$$j_{H^+_{L(n)}} \delta z \delta y_l = -D_{H^+} \left(\frac{[H^+]_{(n)} - [H^+]_{(n-1)}}{\delta x} \right) \delta z \delta y_l - \xi t^\circ_+ \left(j_{e^-_R} - j_{e^-_L} \right) \delta z \delta y_s \qquad (4.40)$$

$$j_{H^+_{R(n)}} \delta z \delta y_l = -D_{H^+} \left(\frac{[H^+]_{(n+1)} - [H^+]_{(n)}}{\delta x} \right) \delta z \delta y_l + (1-\xi)\, t^\circ_+ \left(j_{e^-_R} - j_{e^-_L} \right) \delta z \delta y_s \qquad (4.41)$$

The first term on the right in each of eq. (4.40) and eq. (4.41) represents the ionic flux that would flow solely because of a concentration gradient of that ionic species. Here D_{H^+} represents the diffusivity of the H^+ in the electrolyte.

The second term on the right of these equations comes from the fact that when the reaction in eq. (1.1) occurs within a volume element , there will be $\left(j_{e^-_R} - j_{e^-_L} \right) \delta y_s \delta z$ moles of negative charge transported into the liquid phase from the solid phase. These negative charges need to be neutralized so that electro-neutrality can be maintained. Moving some positive charges into the liquid phase of the volume element and moving some negative charges out of the liquid phase of that volume element neutralizes these

negative charges and creates an ionic current within the electrolyte.

The fraction of the ionic current that is made up of positive H^+ ions is t_+°. For the system depicted in fig. 4-3, a fraction ξ of these H^+ ions will come in from the left and a fraction $1-\xi$ will come in from the right. Moving ions like this will, of course, perturb the electro-neutrality of the neighboring elements. This is fine because the reaction at the lead plate will produce two moles of positive charge in the lead plate's liquid phase. This positive charge will attract the negative charge from the lead dioxide plate and the system will maintain electro-neutrality by forcing an ionic current to flow between the two plates that will neutralize the positive 2 charge at the lead plate and the negative 2 charge at the lead dioxide plate.

By substituting eq. (4.40) and eq. (4.41) into eq. (4.38) we get eq. (4.42).

$$\frac{\left(\Delta[H^+]_{(n)}\delta y_l\right)}{\Delta t}\delta x\delta z = \overbrace{D_{H^+}\left(\frac{[H^+]_{(n-1)}-2[H^+]_{(n)}+[H^+]_{(n+1)}}{\delta x}\right)\delta y_l\delta z}^{Diffusion} \underbrace{-t_+^\circ\left(j_{e_R^-}-j_{e_L^-}\right)\delta y_s\delta z}_{migration} + \underbrace{\left(\frac{3}{2}\right)\left(j_{e_R^-}-j_{e_L^-}\right)\delta y_s\delta z}_{reaction} \tag{4.42}$$

The right side of eq. (4.42) clearly shows the change in the total number of H^+ ions due to diffusion, migration, and reaction. Grouping the migration and reactions terms results in

$$\frac{\left(\Delta[H^+]_{(n)}\delta y_l\right)}{\Delta t}\delta x\delta z = D_{H^+}\left(\frac{[H^+]_{(n-1)}-2[H^+]_{(n)}+[H^+]_{(n+1)}}{\delta x}\right)\delta y_l\delta z + \left(\frac{3-2t_+^\circ}{2}\right)\left(\frac{j_{e_R^-}-j_{e_L^-}}{\delta x}\right)\delta y_s\delta z \tag{4.43}$$

An expression for the material balance of HSO_4^- ions can be found in a similar manner resulting in eq. (4.44).

$$\frac{\Delta\left([HSO_4^-]_{(n)}\delta y_l\right)}{\Delta t}\delta x\delta z = D_{HSO_4^-}\left(\frac{[HSO_4^-]_{(n-1)} - 2[HSO_4^-]_{(n)} + [HSO_4^-]_{(n+1)}}{\delta x}\right)\delta y_l\delta z + (1-t_+^\circ)\left(\frac{j_{e_R^-} - j_{e_L^-}}{\delta x}\right)\delta y_s\delta z + \left(\frac{1}{2}\right)\left(\frac{j_{e_R^-} - j_{e_L^-}}{\delta x}\right)\delta y_s\delta z \quad (4.44)$$

Again grouping migration and reaction effects

$$\frac{\Delta\left([HSO_4^-]_{(n)}\delta y_l\right)}{\Delta t}\delta x\delta z = D_{HSO_4^-}\left(\frac{[HSO_4^-]_{(n-1)} - 2[HSO_4^-]_{(n)} + [HSO_4^-]_{(n+1)}}{\delta x}\right)\delta y_l\delta z + \left(\frac{3-2t_+^\circ}{2}\right)\left(\frac{j_{e_R^-} - j_{e_L^-}}{\delta x}\right)\delta y_s\delta z \quad (4.45)$$

In eq. (4.43) and eq. (4.45) there are two seemingly different coefficients D_{H^+} and $D_{HSO_4^-}$. However, due to electro-neutrality, $[H^+] = [HSO_4^-] = [H_2SO_4]$. Therefore, from this point forward we will assume that we are talking about sulfuric acid and call $[H_2SO_4]$ as C. This allows us to write:

$$\frac{\Delta\left(C_{(n)}\delta y_l\right)}{\Delta t}\delta x\delta z = D_C\left(\frac{C_{(n-1)} - 2C_{(n)} + C_{(n+1)}}{\delta x}\right)\delta y_l\delta z + \left(\frac{3-2t_+^\circ}{2}\right)\left(j_{e_R^-} - j_{e_L^-}\right)\delta y_s\delta z \quad (4.46)$$

Remembering that $\varepsilon = \frac{\delta y_l}{\delta y}$ allows us to rewrite eq. (4.46) in terms of porosity giving us

$$\frac{\Delta\left(C_{(n)}\varepsilon\right)}{\Delta t}\delta x\delta y\delta z = D_C\left(\frac{C_{(n-1)} - 2C_{(n)} + C_{(n+1)}}{\delta x}\right)\varepsilon\delta y\delta z + \left(\frac{3-2t_+^\circ}{2}\right)\left(j_{e_R^-} - j_{e_L^-}\right)(1-\varepsilon)\,\delta y\delta z \quad (4.47)$$

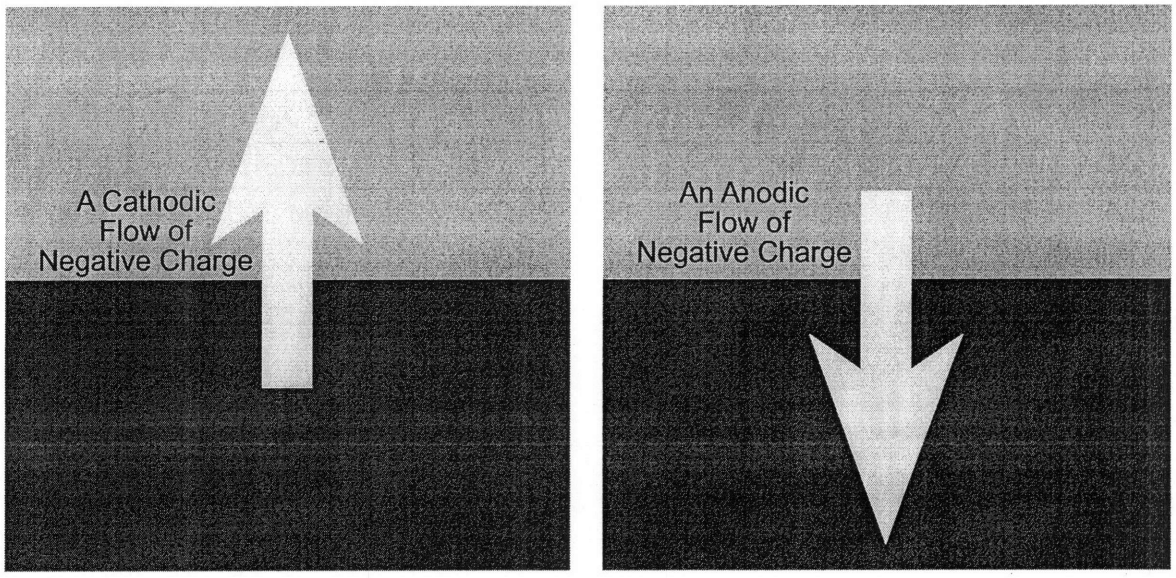
Dividing both sides by δx, δy, and δz gives

$$\boxed{\frac{\Delta\left(C_{(n)}\varepsilon\right)}{\Delta t} = D_C\left(\frac{C_{(n-1)} - 2C_{(n)} + C_{(n+1)}}{\delta x^2}\right)\varepsilon + \left(\frac{3-2t_+^\circ}{2}\right)\left(\frac{j_{e_R^-} - j_{e_L^-}}{\delta x}\right)(1-\varepsilon)} \tag{4.48}$$

4.2.5 Electrode Kinetics

The electrode kinetics equation describes the rate at which charge will be exchanged between the element's two phases due to a perturbation of the element's conditions from equilibrium. When the Pb(IV)O_2 plate is connected to the Pb plate through an external circuit, the potential difference across the solid/liquid junction in each volume element of each plate will change from its equilibrium potential. The difference between the equilibrium potential and the potential difference across the solid/liquid interface is called the *over-potential*, η. When there is a nonzero over-potential in a volume element, charge will be transported across the solid/liquid interface within that volume element. This will cause electrons to flow in the external circuit from the Pb plate to the Pb(IV)O_2 plate or vice versa. Every tiny volume element of the porous region of each electrode will contribute to this flow of electrons by either producing or consuming electrons. The over-potential in a particular volume element will determine how many electrons are either produced or consumed in a unit of time.

To find out how rapidly electrons are produced or consumed in a volume element due to a change in the over-potential, we start by reviewing the concept of chemical equilibrium. In chemistry, the term "equilibrium" doesn't mean the lack of chemical activity. Instead, it means that there is a net activity of zero. Equation 1.1 shows the bi-directional chemical reaction that occurs at the lead dioxide electrode. There are really two reactions going on at all times. These two reactions are called the *forward reaction*



(a) When a lead acid cell discharges, at the lead dioxide electrode, negative charge is transported from the solid phase to the liquid phase. This type of current is referred to as a *cathodic current*.

(b) When a lead acid cell charges, at the lead dioxide electrode, negative charge is transported from the liquid phase to the solid phase. This type of current is referred to as an *anodic current*.

Figure 4-5. The cell reaction causes the exchange of charge between the two phases.

and *back reaction*. The forward reaction is the discharge reaction

$$\mathrm{Pb(IV)O_2}\,(s) + \mathrm{HSO_4^-}\,(aq) + 3\mathrm{H^+}\,(aq) + 2\mathrm{e^-} \rightarrow \mathrm{Pb(II)SO_4}\,(s) + 2\mathrm{H_2O}\,(l) \tag{4.49}$$

and the back reaction is the charging reaction

$$\mathrm{Pb(II)SO_4}\,(s) + 2\mathrm{H_2O}\,(l) \rightarrow \mathrm{Pb(IV)O_2}\,(s) + \mathrm{HSO_4^-}\,(aq) + 3\mathrm{H^+}\,(aq) + 2\mathrm{e^-} \tag{4.50}$$

When the lead dioxide electrode under goes discharge as in reaction eq. (4.49), the net effect is to exchange negative charge from the solid phase into the liquid phase (i.e. in the cathodic direction) as seen in fig. 4-5 (a). When the electrode is charged, negative charge is transported from the liquid phase to

the solid phase (i.e. in the anodic direction). At equilibrium, the rate of charge transport in the cathodic direction equals that in the anodic direction, so the net amount of charge exchanged between the two phases is zero; however, there is *always* charge being exchanged between the two phases.

The exchange of charge between the two phases is done so by the movement of SO_4^{2-} and O^{2-} ions as seen in figs. 3-4 and 3-5. The flows are drawn in fig. 4-6 (a). As depicted, these fluxes have units of $\frac{moles}{sec}$. At equilibrium, since the net amount of charge exchanged between the two phases is zero, the net charge carried by the flow of these two ionic species across the solid/liquid interface is zero.

The fluxes of SO_4^{2-} and O^{2-} ions can be broken down into smaller fluxes as seen in fig. 4-7, where the notation N_i^{sl} represents the flux of species i going from the solid to the liquid and N_i^{ls} to symbolize the flux of species i going from the liquid to the solid. While discharging or charging, the sub-fluxes change in magnitude as seen in fig. 4-7 (a) for the discharging case and fig. 4-7 (b) for the case of charging.

For the trans-interface flow of SO_4^{2-} ions, the total flux, $N_{SO_4^{2-}}$, will be the difference $N_{SO_4^{2-}}^{sl} - N_{SO_4^{2-}}^{ls}$, so

$$N_{SO_4^{2-}} = N_{SO_4^{2-}}^{ls} - N_{SO_4^{2-}}^{sl} \tag{4.51}$$

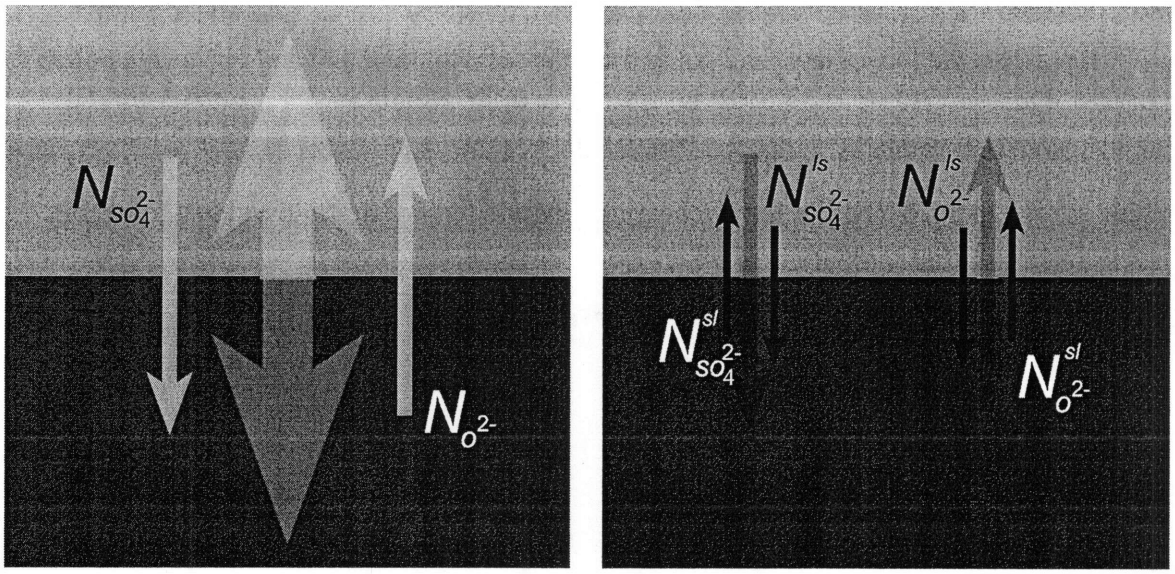
A similar expression can be found for the flux of O^{2-} ions.

$$N_{O^{2-}} = N_{O^{2-}}^{sl} - N_{O^{2-}}^{ls} \tag{4.52}$$

These fluxes can also be seen in fig. 4-6 (a) and (b).

The total trans-interface flow of negative charge in terms of moles of negative charge per second can be written as

$$
\begin{aligned}
N_{NegCharge} &= |z_{O^{2-}}|\, N_{O^{2-}} - \left|z_{SO_4^{2-}}\right| N_{SO_4^{2-}} \\
&= |z_{O^{2-}}|\, j_{O^{2-}} \delta x \delta z - \left|z_{SO_4^{2-}}\right| j_{SO_4^{2-}} \delta x \delta z \\
&= -2\left(j_{e_R^-} - j_{e_L^-}\right) \delta z \delta y_s + 2\frac{\left(j_{e_R^-} - j_{e_L^-}\right)}{2} \delta z \delta y_s \\
&= -\left(j_{e_R^-} - j_{e_L^-}\right) \delta z \delta y_s
\end{aligned} \tag{4.53}
$$

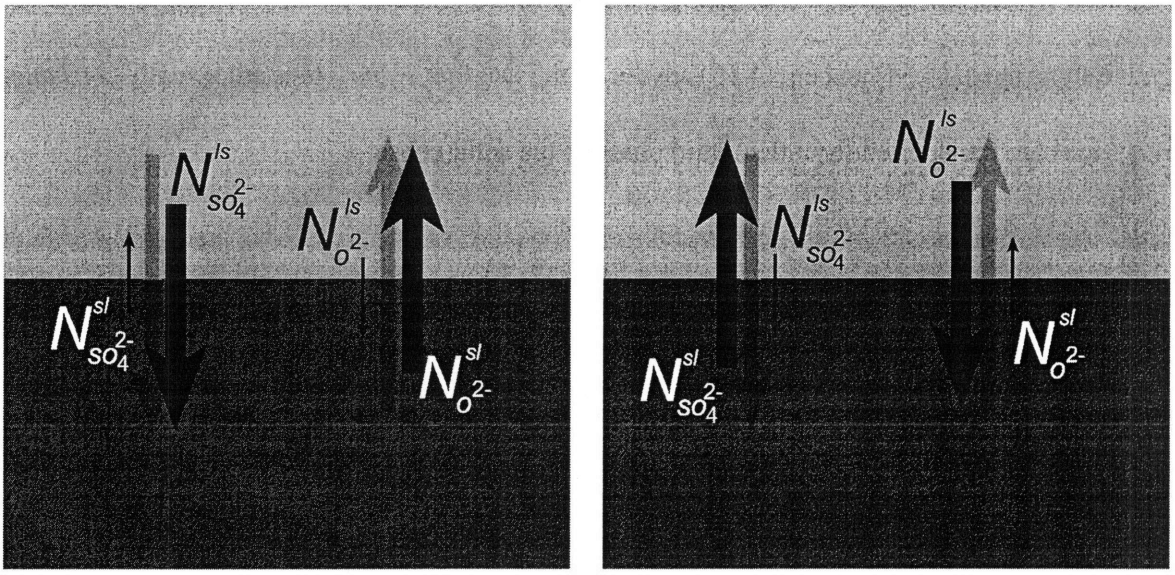


(a) The charge that is exchanged between the two phases is carried by SO_4^{2-} and O^{2-} ions.

(b) The trans-solid/liquid interface fluxes decomposed into their constituent sub-fluxes. At equilibrium, the two fluxes for each of the trans-solid/liquid interface fluxes are equal in magnitude but opposite in direction for a net flux of zero.

Figure 4-6. The trans-solid/liquid interface fluxes and their sub-fluxes

Thus, we need to find an expression for $\left(j_{e_R^-} - j_{e_L^-}\right) \delta z \delta y_s$. Starting with $j_{SO_4^{2-}}$, equation eq. (4.49) says that for every mole of HSO_4^- consumed in the reaction, one mole of SO_4^{2-} has to cross the solid/liquid junction and become part of a lead sulfate molecule. Mathematically, this means that the time rate of



(a) The constituent fluxes during *discharge*.

(b) The constituent fluxes during *charge*.

Figure 4-7. The sub-fluxes change in magnitude when the system is perturbed from equilibrium.

change of the number of moles of HSO_4^- is proportional to the flux of SO_4^{2-} that crosses the solid/liquid interface as seen in eq. (4.54).

$$\frac{\Delta\left([HSO_4^-]\right)}{\Delta t}^{Reaction} \delta x \delta y_l \delta z = -N_{SO_4^{2-}} \tag{4.54}$$

Here δy_l was deliberately written outside the time difference because the electrode kinetics equation focuses on ionic change due solely to reaction and not to volumetric changes or other effects that are secondary to the reaction.

Now, as stated earlier, the flux of SO_4^{2-} ions can be broken into two parts, one part that flows from the liquid phase to the solid phase and another part that flows from the solid phase to the liquid phase. The flow from the liquid phase to the solid phase will be called $N^{ls}_{SO_4^{2-}} = j^{ls}_{SO_4^{2-}} \delta x \delta z$, and, as seen in eq. (4.55), it is equal to $-\frac{\Delta\left([HSO_4^-]^{ls}\right)}{\Delta t}\delta x \delta y_l \delta z$. Here $\frac{\Delta\left([HSO_4^-]^{ls}\right)}{\Delta t}\delta x \delta y_l \delta z$ is the amount of HSO_4^- that is

being *produced* by the flux of SO_4^{2-} ions going from the liquid phase to the solid phase in a time Δt. The negative signs in eq. (4.54) and eq. (4.55) are due to the fact that HSO_4^- ions are actually *consumed* when SO_4^{2-} ions are transported from the liquid phase to the solid phase.

$$j^{ls}_{SO_4^{2-}}\delta x\delta z = -\frac{\Delta\left(\left[HSO_4^-\right]^{ls}\right)}{\Delta t}\delta x\delta y_l\delta z \tag{4.55}$$

By considering the amount of HSO_4^- *produced* by the flux of SO_4^{2-} that flows from the solid phase to the liquid phase in a time Δt, eq. (4.56) can be written.

$$j^{sl}_{SO_4^{2-}}\delta x\delta z = \frac{\Delta\left(\left[HSO_4^-\right]^{sl}\right)}{\Delta t}\delta x\delta y_l\delta z \tag{4.56}$$

Because $N_{SO_4^{2-}}$ is defined positive in the liquid to solid direction, so is $j_{SO_4^{2-}}$. Thus eq. (4.56) is subtracted from eq. (4.55) to get an expression for $N_{SO_4^{2-}}$.

$$j_{SO_4^{2-}}\delta x\delta z = j^{ls}_{SO_4^{2-}}\delta x\delta z - j^{sl}_{SO_4^{2-}}\delta x\delta z = -\left(\frac{\Delta\left(\left[HSO_4^-\right]^{ls}\right)}{\Delta t} + \frac{\Delta\left(\left[HSO_4^-\right]^{sl}\right)}{\Delta t}\right)\delta x\delta y_l\delta z \tag{4.57}$$

The flux $N^{ls}_{SO_4^{2-}}$ is dependent on there being an ample supply of HSO_4^- at the electrode/electrolyte interface. At this point we choose to make the standard assumption that the forward reaction eq. (4.49) is first order with respect to the concentration of HSO_4^- [18, p.502], which allows us to write:

$$\frac{\Delta\left[HSO_4^-\right]^{ls}}{\Delta t} = k_{ls}\left[HSO_4^-\right] \tag{4.58}$$

It should be noted that the assumption of first order for the reaction could be wrong. The flux $N^{sl}_{SO_4^{2-}}$,

on the other hand, depends on the abundance of H^+ at the electrode surface giving

$$\frac{\Delta\left[HSO_4^-\right]^{sl}}{\Delta t} = k_{sl}\left[H^+\right] \tag{4.59}$$

The reaction coefficients k_{ls} and k_{sl} have units of $\frac{1}{s}$ and are short hand for

$$k_i = \beta_i e^{\frac{-E_{a_i}}{RT}} \tag{4.60}$$

Equation eq. (4.60) is referred to as the Arrhenius equation [18, 28, p.88, p.511]. Here E_{a_i}, the activation energy, is the barrier energy that impedes progress toward the new lower energy state following perturbation. R is the gas constant and T is absolute temperature. The product RT is the thermal energy available for reaction. β_i, the *frequency factor*, represents the total number of attempts by SO_4^{2-} ions to be transported across the solid/liquid interface in the i direction. β_i has units of $\frac{1}{s}$.

β_i can be thought of as being made up of two parts. The first factor that goes into making up β_i, SA_{int}, is the per unit volume surface area of the solid/liquid interface. It has units of $\frac{cm^2}{cm^3}$. The second component, the *standard rate constant* k_0, has units of $\frac{cm}{s}$.

$$\beta_i = SA_{int_i} k_{0_i} \tag{4.61}$$

Modeling the reaction as first order with respect to the concentration of HSO_4^- and H^+, allows us to write

$$\frac{\Delta\left[HSO_4^-\right]^{ls}}{\Delta t} = -\beta_{ls}\left[HSO_4^-\right] e^{\frac{\alpha_{ls} nF\left(\phi_l - \phi_s + \phi_{eq}\right)}{RT}} \tag{4.62}$$

and

$$\frac{\Delta\left[HSO_4^-\right]^{sl}}{\Delta t} = \beta_{sl}\left[H^+\right] e^{\frac{\alpha_{sl} n\mathbf{F}(\phi_s - \phi_l - \phi_{eq})}{RT}} \tag{4.63}$$

Here ϕ_s is the electrical potential in the solid phase, ϕ_l is the potential in the liquid phase, and ϕ_{eq} is the equilibrium potential of the electrode. The over-potential $\eta = \phi_s - \phi_l - \phi_{eq}$. **F** is Faraday's constant which allows the conversion between the number of moles of an ion and the number of Coulombs of that ion. The transfer coefficients α_{sl} and α_{ls} reflect the asymmetry of the activiation barrier [28]. α_{sl} and α_{ls} must sum to unity [28]. The n is the number of moles of electrons involved in the reaction. For a lead acid battery $n = 2$.

Substituting the right hand sides of eq. (4.63) and eq. (4.62) into the right hand side of eq. (4.57) and using the left most side of that same equation, we get eq. (4.64).

$$j_{SO_4^{2-}}\delta x\delta z = -\left(-SA_{int}^{ls}k_0^{ls}\left[HSO_4^-\right] e^{\frac{\alpha_{ls} n\mathbf{F}(-\eta)}{RT}} + SA_{int}^{sl}k_0^{sl}\left[H^+\right] e^{\frac{\alpha_{sl} n\mathbf{F}(\eta)}{RT}}\right)\delta x\delta y_l\delta z \tag{4.64}$$

The standard rate constant, however, is not what is used in the literature. Instead, the *exchange current density* is reported. The exchange current density is usually denoted by i_0 and has units of $\frac{Coulomb}{cm^2 \cdot s}$. The exchange current density represents the amount of charge per unit area that is transported across the solid/liquid interface in each direction at equilibrium.

At equilibrium, the amount of charge transported across the interface in the liquid to solid direction by SO_4^{2-} ions, terms of $\frac{Coulombs}{sec}$ is

$$i_0^{ls} SA_{int}^{ls} \delta x \delta y \delta z = \left| z_{SO_4^{2-}} \right| \mathbf{F} SA_{int}^{ls} k_0^{ls} \left[\mathrm{HSO_4^-} \right]^* \delta x \delta y_l \delta z$$
$$= \left| z_{SO_4^{2-}} \right| \mathbf{F} SA_{int}^{ls} k_0^{ls} \left[\mathrm{HSO_4^-} \right]^* \varepsilon \delta x \delta y \delta z \tag{4.65}$$

A similar expression can be written for i_0^{sl}. k_0^i, then, can be written as,

$$k_0^i = \frac{i_0^i}{\left| z_{SO_4^{2-}} \right| \mathbf{F} \left[\mathrm{HSO_4^-} \right]^* \varepsilon} \tag{4.66}$$

Substituting expressions for k_0^{ls} and k_0^{sl} into eq. (4.64) gives,

$$j_{SO_4^{2-}} \delta x \delta z = - \left(-SA_{int}^{ls} \frac{i_0^{ls}}{\left| z_{SO_4^{2-}} \right| \mathbf{F} \varepsilon} \frac{[HSO_4^-]}{\left[\mathrm{HSO_4^-} \right]^*} e^{\frac{\alpha_{ls} nF(-\eta)}{RT}} + SA_{int}^{sl} \frac{i_0^{sl}}{\left| z_{SO_4^{2-}} \right| \mathbf{F} \varepsilon} \frac{[H^+]}{\left[\mathrm{H^+} \right]^*} e^{\frac{\alpha_{sl} nF(\eta)}{RT}} \right) \varepsilon \delta x \delta y \delta z \tag{4.67}$$

$$j_{SO_4^{2-}} \delta x \delta z = - \left(-SA_{int}^{ls} \frac{i_0}{2\mathbf{F}} \frac{[HSO_4^-]}{\left[\mathrm{HSO_4^-} \right]^*} e^{\frac{\alpha_{ls} nF(-\eta)}{RT}} + SA_{int}^{sl} \frac{i_0}{2\mathbf{F}} \frac{[H^+]}{\left[\mathrm{H^+} \right]^*} e^{\frac{\alpha_{sl} nF(\eta)}{RT}} \right) \delta x \delta y \delta z \tag{4.68}$$

Since electro-neutrality is a fundamental assumption of this model, $[HSO_4^-] = [H^+] = C$ and $\left[\mathrm{HSO_4^-} \right]^* = \left[\mathrm{H^+} \right]^* = C^*$. C is the concentration of the electrolyte and C^* is the bulk concentration of the electrolyte. Also, by again looking at eq. (1.1) and noting that for every two moles of electrons that are consumed, one mole of SO_4^{2-} ions is transported from the liquid to the solid phase (i.e. $\left(\frac{j_{e_R^-} - j_{e_L^-}}{2} \right) \delta z \delta y_s = -j_{SO_4^{2-}} \delta x \delta z$), we get eq. (4.69).

$$\left(\frac{j_{e_R^-} - j_{e_L^-}}{2}\right)\delta z\delta y_s = SA_{int}\frac{i_0}{\mathbf{F}}\frac{C}{C^*}\left(e^{\frac{\alpha_{sl}n\mathbf{F}(\eta)}{RT}} - e^{\frac{\alpha_{ls}n\mathbf{F}(-\eta)}{RT}}\right)\delta x\delta y\delta z \tag{4.69}$$

Assigning $SA_{int} = \frac{\delta x\delta z}{\delta x\delta y\delta z}$ and rearranging eq. (4.69) we get eq. (4.70),

$$\left(\frac{j_{e_R^-} - j_{e_L^-}}{\delta x}\right)(1-\varepsilon) = \frac{\delta x\delta z}{\delta x\delta y\delta z}\frac{i_0}{\mathbf{F}}\frac{C}{C^*}\left(e^{\frac{\alpha_{sl}n\mathbf{F}(\eta)}{RT}} - e^{\frac{\alpha_{ls}n\mathbf{F}(-\eta)}{RT}}\right) \tag{4.70}$$

The notation α^{ls} and α^{sl} is particular to this derivation. In order to bring our notation more in line with that of the literature, we introduce the idea of a *cathodic current* and an *anodic current*. The trans-solid/liquid interface current is said to behave like a cathodic current when negative charge is being transported from the electrode into the solution which is tantamount to e^- consumption. Ironically enough, negative charge is transported into the electrolyte when SO_4^{2-} ions flow from the liquid phase into the solid phase. The electrode interface current is said to be an anodic current when negative charge is being transfered from the solution to the electrode which is tantamount to e^- emission [28]. Therefore, we can write α_{ls} as α_c and α_{sl} as α_a where the c stands for *cathodic* and the a stands for *anodic*. This gives us

$$\boxed{\left(\frac{j_{e_R^-} - j_{e_L^-}}{\delta x}\right)(1-\varepsilon) = \frac{1}{\delta y}\frac{i_0}{\mathbf{F}}\frac{C}{C^*}\left(e^{\frac{\alpha_a n\mathbf{F}(\eta)}{RT}} - e^{\frac{\alpha_c n\mathbf{F}(-\eta)}{RT}}\right)} \tag{4.71}$$

It should be stressed that a cathodic **AND** an anodic reaction occurs at each electrode. However, at the cathode, the cathodic transport of charge in the cathodic direction is much greater than the transport of charge in the anodic direction. At the anode, the opposite is true.

Summary of equations for single cell lead dioxide model

1. Change of Porosity

$$\frac{\Delta\varepsilon}{\Delta t} = \left[\left(\frac{MW_{\mathrm{Pb(II)SO_4}}}{\rho_{\mathrm{Pb(II)SO_4}}}\right) - \left(\frac{MW_{\mathrm{Pb(IV)O_2}}}{\rho_{\mathrm{Pb(IV)O_2}}}\right)\right]\left(\frac{j_{e_R^-} - j_{e_L^-}}{2\delta x}\right)(1-\varepsilon) \tag{4.72}$$

2. Conservation of Matter

$$\frac{\Delta\varepsilon C_{(n)}}{\Delta t} = \varepsilon D_C\left(\frac{C_{(n-1)} - 2C_{(n)} + C_{(n+1)}}{\delta x^2}\right) + \left(\frac{3-2t_+^\circ}{2}\right)\left(\frac{j_{e_R^-} - j_{e_L^-}}{\delta x}\right)(1-\varepsilon) \tag{4.73}$$

3. Electrode Kinetics

$$0 = \left(\frac{j_{e_R^-} - j_{e_L^-}}{\delta x}\right)(1-\varepsilon) - \frac{1}{\delta y}\frac{i_0}{\mathbf{F}}\frac{C}{C^*}\left(e^{\frac{\alpha_a nF(\eta)}{RT}} - e^{\frac{\alpha_c nF(-\eta)}{RT}}\right) \tag{4.74}$$

4. Conservation of Charge

$$0 = \left(\frac{j_{l_R} - j_{l_L}}{\delta x}\right)\varepsilon + \left(\frac{j_{e_R^-} - j_{e_L^-}}{\delta x}\right)(1-\varepsilon) \tag{4.75}$$

5. Ohm's Law in Solution

$$0 = \frac{j_l \mathbf{F}}{\kappa} - \frac{\partial\phi_l}{\partial x} + \left(1-2t_+^\circ\right)\frac{\mathbf{RT}}{\mathbf{F}}\frac{\partial ln\,(C)}{\partial x} \tag{4.76}$$

4.2.6 Summary of equations for single volume element lead model

The techniques just discussed for a lead dioxide volume element can readily be applied to a lead volume element too. The reaction for the lead volume element would be eq. (1.2), and the finite volume would look like fig. 4-8

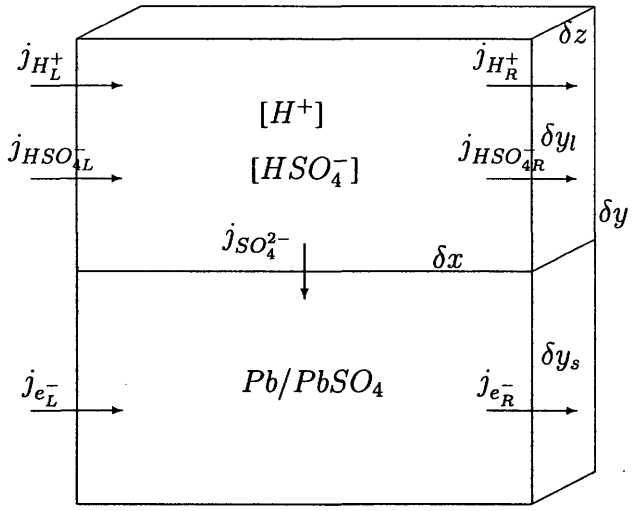


Figure 4-8. A simplified model of a section of an Pb electrode's porous region. The lower half consists entirely of lead and lead sulfate. The upper half consists of electrolyte. Furthermore, $\delta y = \delta y_s + \delta y_l$

1. Change of Porosity

$$\frac{\Delta \varepsilon}{\Delta t} = -\left[\left(\frac{MW_{PbSO_4}}{\rho_{PbSO_4}}\right) - \left(\frac{MW_{Pb}}{\rho_{Pb}}\right)\right]\left(\frac{j_{e_R^-} - j_{e_L^-}}{2\delta x}\right)(1-\varepsilon) \tag{4.77}$$

2. Conservation of Matter

$$\frac{\Delta \varepsilon C_{(n)}}{\Delta t} = \varepsilon D_C\left(\frac{C_{(n-1)} - 2C_{(n)} + C_{(n+1)}}{\delta x^2}\right) + \left(\frac{1-2t_+^\circ}{2}\right)\left(\frac{j_{e_R^-} - j_{e_L^-}}{\delta x}\right)(1-\varepsilon) \tag{4.78}$$

3. Electrode Kinetics

$$0 = \left(\frac{j_{e_R^-} - j_{e_L^-}}{\delta x}\right)(1-\varepsilon) - \frac{1}{\delta y}\frac{i_0}{\mathbf{F}}\frac{C}{C^*}\left(e^{\frac{\alpha_a n\mathbf{F}(\eta)}{RT}} - e^{\frac{\alpha_c n\mathbf{F}(-\eta)}{RT}}\right) \tag{4.79}$$

4. Conservation of Charge

$$0 = \left(\frac{j_{l_R} - j_{l_L}}{\delta x}\right)\varepsilon + \left(\frac{j_{e_R^-} - j_{e_L^-}}{\delta x}\right)(1-\varepsilon) \tag{4.80}$$

5. Ohm's Law in Solution

$$0 = \frac{j_l \mathbf{F}}{\kappa} - \frac{\partial \phi_l}{\partial x} + \left(1 - 2t_+^\circ\right)\frac{\mathrm{RT}}{\mathbf{F}}\frac{\partial ln\left(C\right)}{\partial x} \tag{4.81}$$

4.3 Two Volume Model Implementation

In section 4.1, a set of equations that describe the phenomena observed in section 3.1 were derived. Those equations are now used to develop a simple lead (Pb) acid battery cell model. Fig. 4-9 shows the simplified cell model that was implemented. Moving from left to right, first there is the lead tab that acts as an interface to the external circuit. Next, there is a simplified Pb(IV)O_2 electrode that consists of one of the volume elements developed in section 4.1. Then there is the electrolyte basin. Then a simplified Pb electrode, and finally another lead (Pb) tab. Fig. 4-10 shows the cell with all important dimensions labeled.

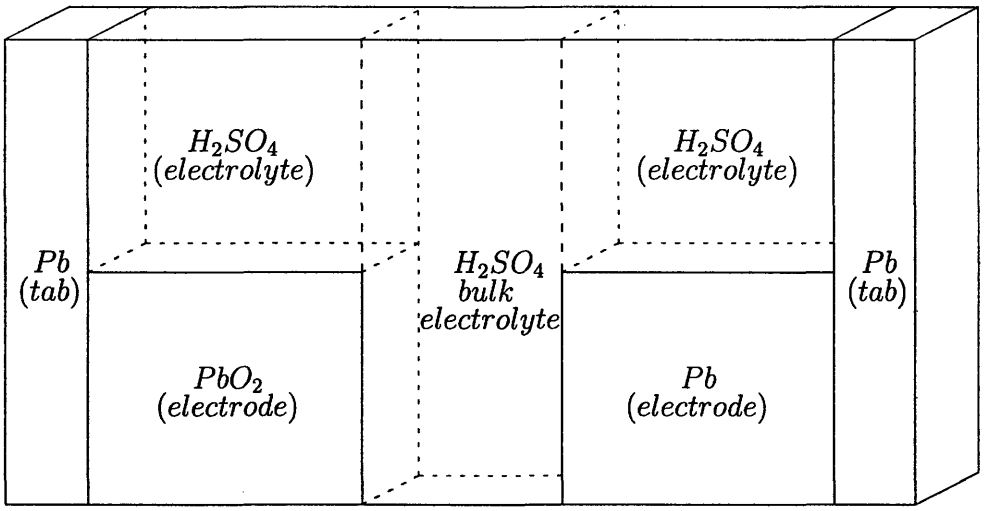


Figure 4-9. Simplified single cell model of a lead (Pb) acid battery.

At first glance, this model may seem too simple to be of use in the lab, in the field, or in the classroom. However, if one were to think of each electrode as a collection of small elementary volumes, and if an elementary volume were selected from each electrode and the bulk electrolyte region, as seen in fig. 4-11, together these three elements should act as a battery. A battery cell, then, is a collection of these smaller batteries. Therefore, the simplified model of fig. 4-10 should exhibit battery-like behavior.

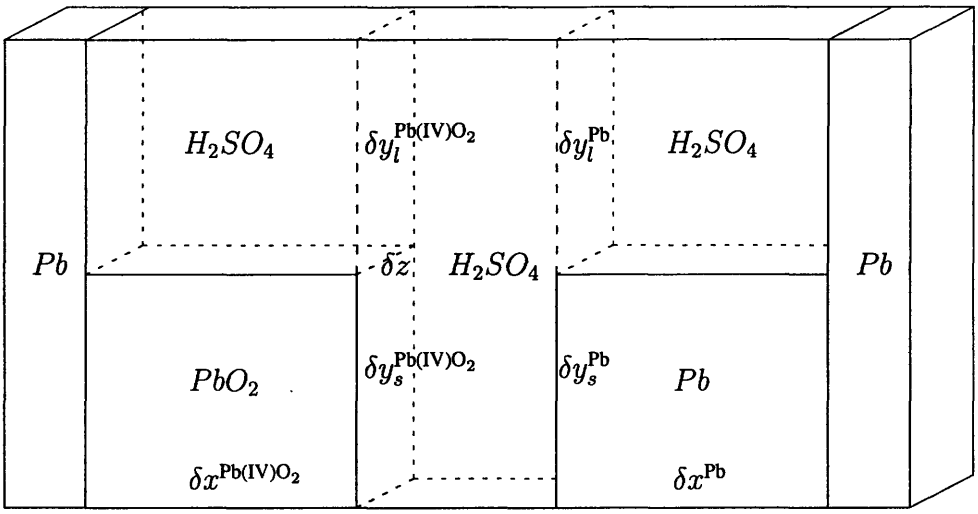


Figure 4-10. Simplified Single Cell Model with Dimensions Labeled

While the exact value of the output potential will most certainly not be representative of an actual battery, the general shape of the time/potential and potential/current curves produced by our simple cell should look very similar to those of a real battery. This simple model will give us the opportunity to build up our intuition about how the model parameters affect the performance of the battery cell. This intuition will be useful when implementing and debugging the larger and more complex model to be developed in chapter 5.

Aside from the simplification done to the structure of each electrode, the simulations make four more noteworthy approximations. First, the lead (Pb) tabs have infinite conductivity. Next, the electrolyte basin has infinite capacity. That is to say the concentration of electrolyte in the bulk region between the two electrodes is independent of time and usage. Third, there are no thermal effects, and finally, acid is always assumed to be able to reach the active material. In other words, the $Pb(II)SO_4$ that forms at the electrode/electrolyte interface does not make a barrier that impedes transport of electrolyte to the electrode surface. Consequently, the electrode/electrolyte interface area is constant.

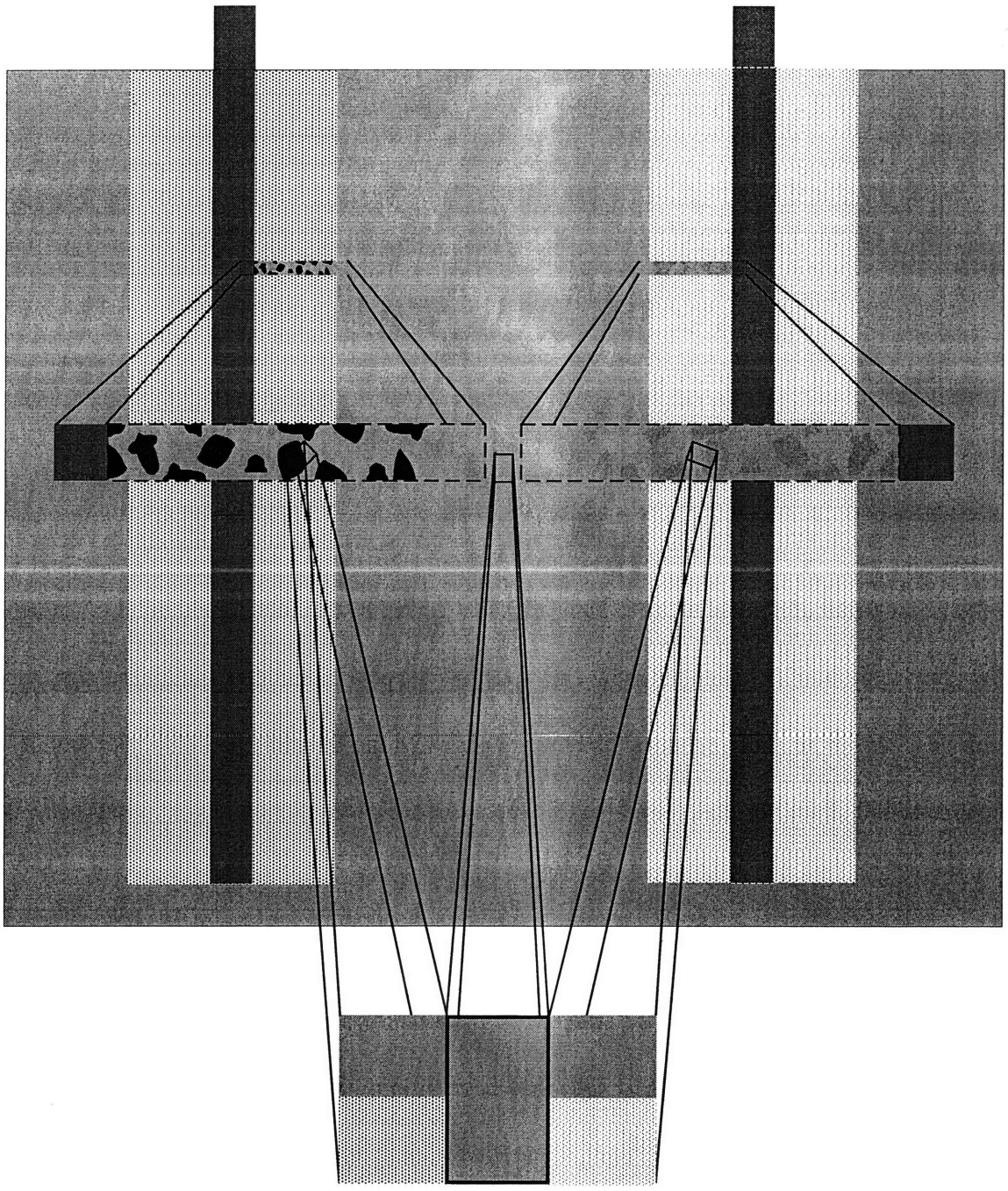


Figure 4-11. Each electrode can be thought of being made up of thousands of little volumes like the one described in section 4.1. If one were to take an elementary volume from each of the electrodes and an elementary volume from the electrolyte, these three volumes should give battery like behavior.

4.3.1 Boundary Conditions

The boundaries of interest in this model are the two electrode region/lead (Pb) tab interfaces and the two electrode region/bulk electrolyte interfaces. How these boundaries effect the relevant fluxes is important. In fig. 4-12, only the relevant x-direction fluxes have been drawn. It is true that the division between the $Pb(IV)O_2$ and H_2SO_4 of the $Pb(IV)O_2$ electrode and the division between the Pb and H_2SO_4 of the Pb electrode also represent boundaries; however, these boundaries do not cause changes in the equations, so they have not been drawn in fig. 4-12.

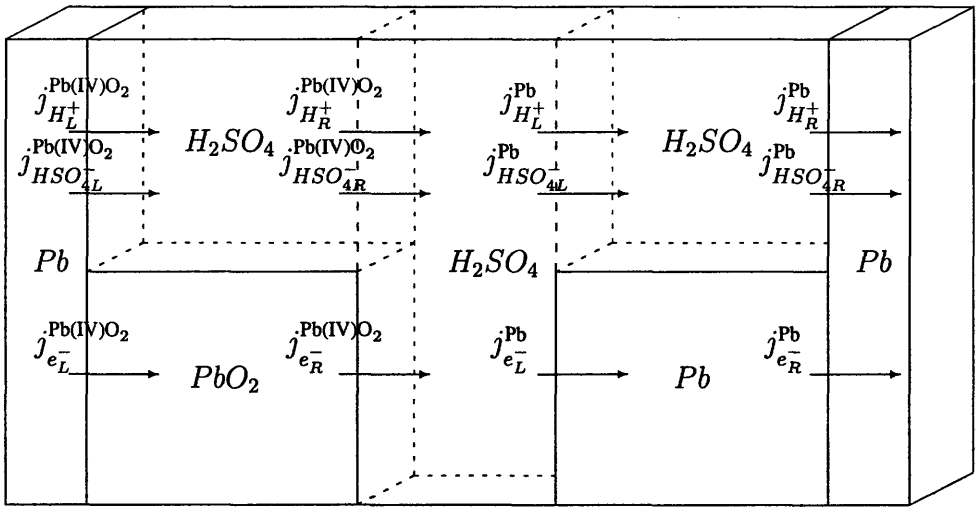


Figure 4-12. Simplified single cell model with x-direction intra-phase fluxes

For the given labeled fluxes, there will be no flow of ions between a solid region and a liquid region. Therefore, flux densities $j^{Pb(IV)O_2}_{H^+_L}$, $j^{Pb(IV)O_2}_{HSO^-_{4L}}$, $j^{Pb}_{H^+_R}$, and $j^{Pb}_{HSO^-_{4R}}$ must all be zero. Also, electrons cannot flow from a solid region into a liquid region. Therefore, flux densities $j^{Pb(IV)O_2}_{e^-_R}$ and $j^{Pb}_{e^-_L}$ must also be zero. Applying these boundary conditions results in fig. 4-13.

Fig. 4-14 further simplifies the model by combining the H^+ and HSO_4^- fluxes of each electrode into one electrolyte flux j_l, where $j_l = j_{HSO_4^-} - j_{H^+}$. It is from the model in fig. 4-14 that eqs. (4.82)–(4.86)

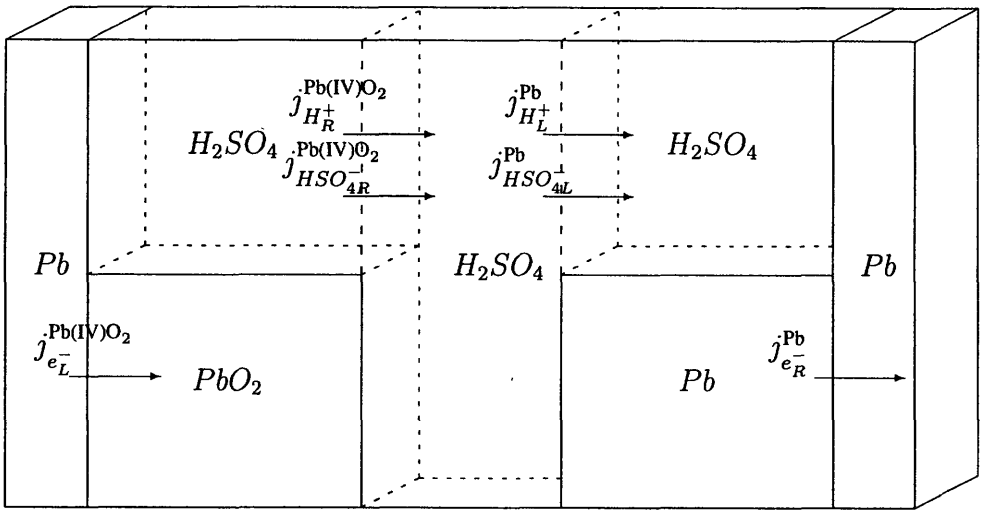


Figure 4-13. Simplified Single Cell Model with Boundary Conditions

for the lead dioxide electrode and eqs. (4.87)–(4.91) for the lead electrode were derived.

Lead Dioxide Electrode Reaction Equations w/ Boundary Conditions

1. Change of Porosity

$$\varepsilon^{k+1} = \varepsilon^k - \Delta t \left[\left(\frac{MW_{PbSO_4}}{\rho_{PbSO_4}} \right) - \left(\frac{MW_{PbO_2}}{\rho_{PbO_2}} \right) \right] \left(\frac{I}{2\mathbf{F}\delta x \delta y \delta z} \right) \tag{4.82}$$

2. Conservation of Matter

$$C^{k+1} = C^k - \Delta t \frac{C^k}{\varepsilon^{k+1}} \left[\left(\frac{MW_{PbSO_4}}{\rho_{PbSO_4}} \right) - \left(\frac{MW_{PbO_2}}{\rho_{PbO_2}} \right) \right] \left(\frac{I}{2\mathbf{F}\delta x \delta y \delta z} \right) + \Delta t \varepsilon^{k+1} D_C \left(\frac{C^k_{(res)} - C^k}{\delta x^2} \right) - \Delta t \left(\frac{3 - 2t^\circ_+}{2} \right) \left(\frac{I}{\mathbf{F}\delta x \delta y \delta z} \right) \tag{4.83}$$

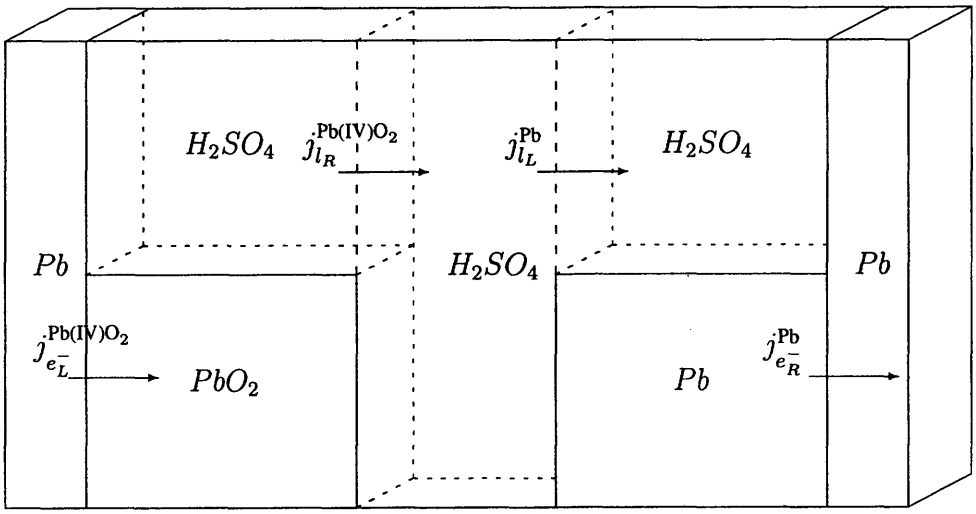


Figure 4-14. Simplified Single Cell Model with Boundary Conditions. Here the liquid fluxes j_{H^+} and $j_{HSO_4^-}$ have been combined into $j_l = j_{HSO_4^-} - j_{H^+}$

3. Conservation of Charge

$$j_{l_R}^{k+1} = \frac{I}{\mathbf{F}\varepsilon^{k+1}\delta y \delta z} \tag{4.84}$$

4. Ohm's Law in Solution

$$\phi_l{}^{k+1} = -\frac{j_{l_R}^{k+1}\mathbf{F}\delta x}{\kappa} + \phi_{l(res)} + \left(1 - 2t_+^\circ\right)\frac{RT}{\mathbf{F}}\left(log\left(C_{(res)}\right) - log\left(C^{k+1}\right)\right) \tag{4.85}$$

5. Electrode Kinetics

$$\frac{I}{\mathbf{F}\delta x \delta y \delta z} + \frac{1}{\delta y}\frac{i_0}{\mathbf{F}}\frac{C^{k+1}}{C^*}\left(e^{\frac{\alpha_a n\mathbf{F}(\phi_s{}^{k+1} - \phi_l{}^{k+1} - \phi_{eq})}{RT}} - e^{\frac{\alpha_c n\mathbf{F}(\phi_l{}^{k+1} - \phi_s{}^{k+1} + \phi_{eq})}{RT}}\right) = 0 \tag{4.86}$$

Lead Electrode Reaction Equations w/ Boundary Conditions

1. Change of Porosity

$$\varepsilon^{k+1} = \varepsilon^k - \Delta t \left[\left(\frac{MW_{PbSO_4}}{\rho_{PbSO_4}} \right) - \left(\frac{MW_{Pb}}{\rho_{Pb}} \right) \right] \left(\frac{I}{2\mathbf{F}\delta x \delta y \delta z} \right) \tag{4.87}$$

2. Conservation of Matter

$$C^{k+1} = C^k + \Delta t \frac{C^k}{\varepsilon^{k+1}} \left[\left(\frac{MW_{PbSO_4}}{\rho_{PbSO_4}} \right) - \left(\frac{MW_{Pb}}{\rho_{Pb}} \right) \right] \left(\frac{I}{2\mathbf{F}\delta x \delta y \delta z} \right) + \Delta t \varepsilon^{k+1} D_C \left(\frac{C^k_{(res)} - C^k}{\delta x^2} \right) + \Delta t \left(\frac{1 - 2t^\circ_+}{2} \right) \left(\frac{I}{\mathbf{F}\delta x \delta y \delta z} \right) \tag{4.88}$$

3. Conservation of Charge

$$j_{l_L}^{k+1} = \frac{I}{\mathbf{F}\varepsilon^{k+1}\delta y \delta z} \tag{4.89}$$

4. Ohm's Law in Solution

$$\phi_l{}^{k+1} = \frac{j_{l_L}^{k+1}\mathbf{F}\delta x}{\kappa} + \phi_{l(res)} + \left(1 - 2t^\circ_+\right) \frac{RT}{\mathbf{F}} \left(log\left(C^{k+1}\right) - log\left(C_{(res)}\right) \right) = 0 \tag{4.90}$$

5. Electrode Kinetics

$$\frac{I}{\mathbf{F}\delta x \delta y \delta z} - \frac{1}{\delta y} \frac{i_0^{\mathrm{Pb}}}{\mathbf{F}} \frac{C^{k+1}}{C^*} \left(e^{\frac{\alpha_a n \mathbf{F}(\phi_s{}^{k+1} - \phi_l{}^{k+1} - \phi_{eq})}{RT}} - e^{\frac{\alpha_c n \mathbf{F}(\phi_l{}^{k+1} - \phi_s{}^{k+1} + \phi_{eq})}{RT}} \right) = 0 \tag{4.91}$$

Model Variables peculiar to the Lead Dioxide Electrode			
Symbol	Meaning	Initial Value	Units
C	Electrolyte Concentration	0.0049	$\frac{mol}{cm^3}$
ϕ_l	Electrolyte Potential	0	$\frac{Joules}{Coulomb}$
ϕ_s	Solid Potential	1.685	$\frac{Joules}{Coulomb}$
ϵ	Porosity	0.5	unitless
$j_{e_L^-}^{\text{Pb(IV)O}_2}$	Flux of density of electrons entering the metal phase of the electrode's porous region	$\frac{I}{F\delta x\delta dy_s}$	$\frac{mol}{cm^2 \cdot sec}$
$MW_{\text{Pb(IV)O}_2}$	molecular weight of Pb(IV)O_2	239.1988	$\frac{g}{mol}$
$\rho_{\text{Pb(IV)O}_2}$	density of Pb(IV)O_2	9.79	$\frac{g}{cm^3}$
$i_0^{\text{Pb(IV)O}_2}$	Exchange Current Density	$1e-2$	$\frac{Coulombs}{cm^2 \cdot sec}$
δy_s	Height of Solid Region	5.0	cm
δx	width of porous region	0.03	cm
δy	height of porous region	10.0	cm
δz	depth of porous region	7.5	cm
δy_l	height of liquid region	5.0	cm
$A_{int}^{\text{Pb(IV)O}_2}$	solid/liquid interface area	$\frac{1}{\delta y}$	$\frac{cm^2}{cm^3}$
α_a	Transfer Coefficient Anodic	0.575	unitless
α_c	Transfer Coefficient Cathodic	0.425	unitless
ϕ_{eq}	electrode equilibrium potential	1.685	$\frac{Joules}{Coulomb}$

Table 4.1. Variables for Simple Model Lead Dioxide Electrode

4.3.2 Initial Conditions

Some initial conditions for our simulation come from [30,31]. Table 4.1 gives the physical parameters related to the lead dioxide electrode. Table 4.2 for the physical parameters for the lead electrode, and Table 4.3 for parameters common to both electrodes.

Model Variables peculiar to the Lead Electrode			
Symbol	Meaning	Initial Value	Units
C	Electrolyte Concentration	0.0049	$\frac{mol}{cm^3}$
ϕ_l	Electrolyte Potential	0	$\frac{Joules}{Coulomb}$
ϕ_s	Solid Potential	-0.356	$\frac{Joules}{Coulomb}$
ε	Porosity	0.5	unitless
$j^{\mathrm{Pb}}_{e^-_R}$	Flux density of electrons leaving the metal phase of the electrode's porous region	$\frac{I}{\mathrm{F}\delta x \delta dy_s}$	$\frac{mol}{cm^2 \cdot sec}$
MW_{Pb}	molecular weight of Pb	207.2	$\frac{g}{mol}$
ρ_{Pb}	density of Pb	11.34	$\frac{g}{cm^3}$
i_0^{Pb}	Exchange current density	$1e-2$	$\frac{Coulombs}{cm^2 \cdot sec}$
δy_s	Height of Solid Region	5.0	cm
δx	width of porous region	0.03	cm
δy	height of porous region	10.0	cm
δz	depth of porous region	7.5	cm
δy_l	height of liquid region	5.0	cm
A^{Pb}_{int}	solid/liquid interface area	$\frac{1}{\delta y}$	$\frac{cm^2}{cm^3}$
α_a	Transfer Coefficient Anodic	0.775	unitless
α_c	Transfer Coefficient Cathodic	0.225	unitless
ϕ_{eq}	electrode equilibrium potential	-0.356	$\frac{Joules}{Coulomb}$

Table 4.2. Variables for Simple Model Lead Electrode

Model Variables Shared by both Electrodes			
Symbol	Meaning	Initial Value	Units
I	Current being drawn from cell		$\frac{Coulombs}{sec}$
$C_{(res)}$	Electrolyte Concentration	0.0049	$\frac{mol}{cm^3}$
$\phi_{l(res)}$	Electrolyte Potential	0	$\frac{Joules}{Coulomb}$
MW_{PbSO_4}	molecular weight of $PbSO_4$	303.2636	$\frac{g}{mol}$
ρ_{PbSO_4}	density of $PbSO_4$	6.39	$\frac{g}{cm^3}$
F	Faraday's constant	96,487	$\frac{Coulomb}{mol \cdot electrons}$
R	universal gas constant	8.3143	$\frac{J}{mol \cdot K}$
t	time	0	seconds
Δt	time step size	0.1	seconds
t_+^o	fraction of ionic current carried by H^+ in absence of diffusion	0.72	unitless
T	absolute temperature	298	Kelvin
n	number of electrons participating in reaction	2	unitless
κ	conductivity of the solution	0.79	$\frac{1}{\Omega cm}$
D_c	Diffusion coefficient of electrolyte	$3.02e-5$	$\frac{cm^2}{sec}$

Table 4.3. Variables for Simple Model Shared by both Electrodes

4.4 Simulation Results

The purpose of developing the simplified model was to create a tool that would help the user develop insight into the workings of a lead-acid cell. The two-volume model developed does this by giving the user access to all the parameters necessary to control the behavior of the lead-acid cell's primary reactions. The benefit of this model over more complex models is that the major behaviors can be observed without being clouded by the complexity of the implementation.

The equations that describe the behavior of the cell model seen in fig. 4-14 were implemented in Matlab chapter A. To show how this model can be used to develop insight into the behavior of a lead-acid battery, several different simulations were run in which none, one, or two model parameters were modified from the values assigned to them in section 4.3.2. The parameters that were varied are, D_C, SA_{int}, α_c, α_a, and i_0. The behavior of the model, and the effects that modifying the model parameters had on the model's behavior are discussed below.

The discussion will make most sense if one keeps the Nernst equation in mind.

$$E_{cell} = E^{\circ}_{cell} - \left(\frac{\mathrm{RT}}{\mathrm{nF}}\right) \ln Q \tag{4.92}$$

4.4.1 Initial Two-Volume Model Results

We start by looking at a constant current discharge vs time plot. The cell was discharged at a rate of 50mA for 1000 seconds. The initial conditions for the simulation are given in section 4.3.2. The time/voltage discharge curve is shown in fig. 4-15.

Fig. 4-15 exhibits behavior like that of a discharging battery in that when a battery discharges, the cell potential collapses. In fig. 4-15, the cell potential starts at 1.91988V, below the cell equilibrium potential

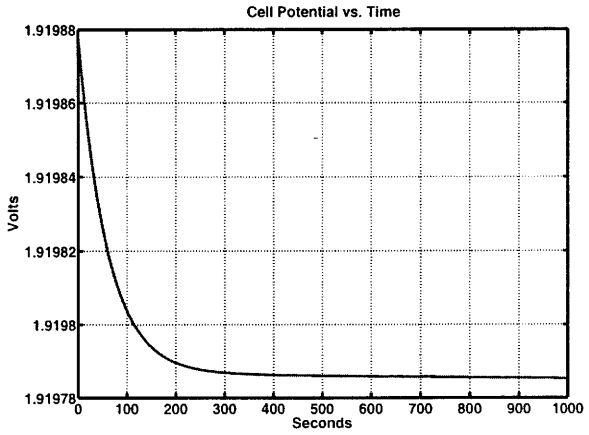
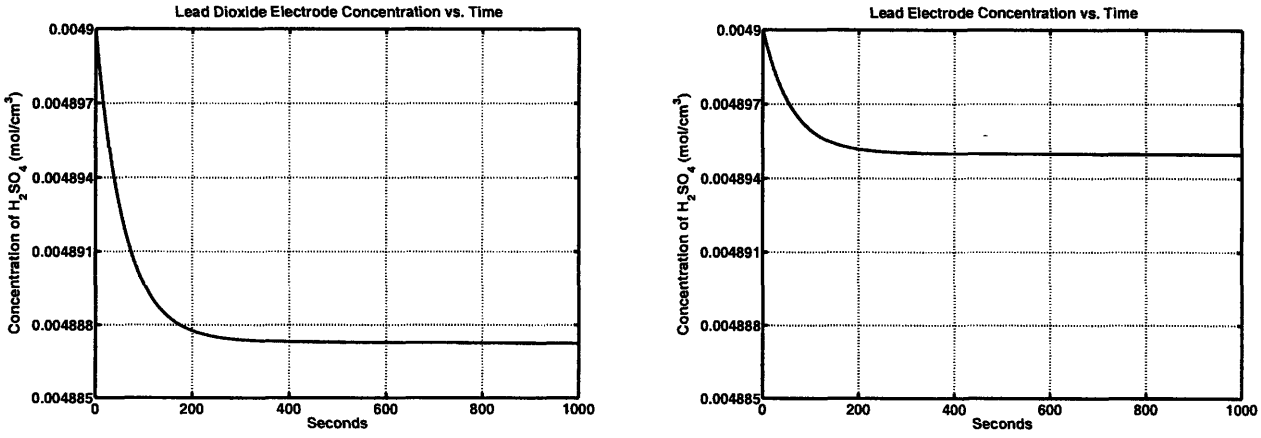


Figure 4-15. Cell potential vs. time curve of the simple cell of section 4.3 which was discharged at a rate of 50mA for 1000 seconds.

of 2.041 Volts, and decreases from there. The potential seems to flatten off after about 300 seconds. This is due to the fact that the rate at which electrolyte is diffusing from the bulk into each electrode is now equal to the rate at which electrolyte is being consumed at each of the electrodes. Furthermore, because the bulk electrolyte is assumed to be infinite capacity, the concentration of electrolyte in the bulk never drops. The result is that the cell reaches an equilibrium and passes current without any further decrease in potential. Therefore, after about 300 seconds, diffusion holds the concentration of the electrolyte in each volume constant as can be seen in fig. 4-16.

Fig. 4-16 shows that the concentration of electrolyte at the $Pb(IV)O_2$ electrode drops more than the concentration of electrolyte at the Pb electrode. This makes physical sense in that the reaction at the $Pb(IV)O_2$ electrode not only consumes sulfuric acid, but it also produces water which further dilutes the local electrolyte concentration.

This effect is also clearly visible in fig. 3 and fig. 4 of [30]. Fig. 3 of [30] is reprinted here as fig. 4-17. In fig. 4-17, it can be seen that during discharge, the concentration of the electrolyte at the lead dioxide electrode decreases more than that at the lead (Pb) electrode. This result was also be verified by



(a) Lead dioxide electrode

(b) Lead (Pb) electrode

Figure 4-16. The concentration of the electrolyte at each electrode during discharge.

the digital Mach-Zehnder laser interferometry experiments of chapter 2 and the more sophisticated 2-D model of section 5.3 in section 5.6.

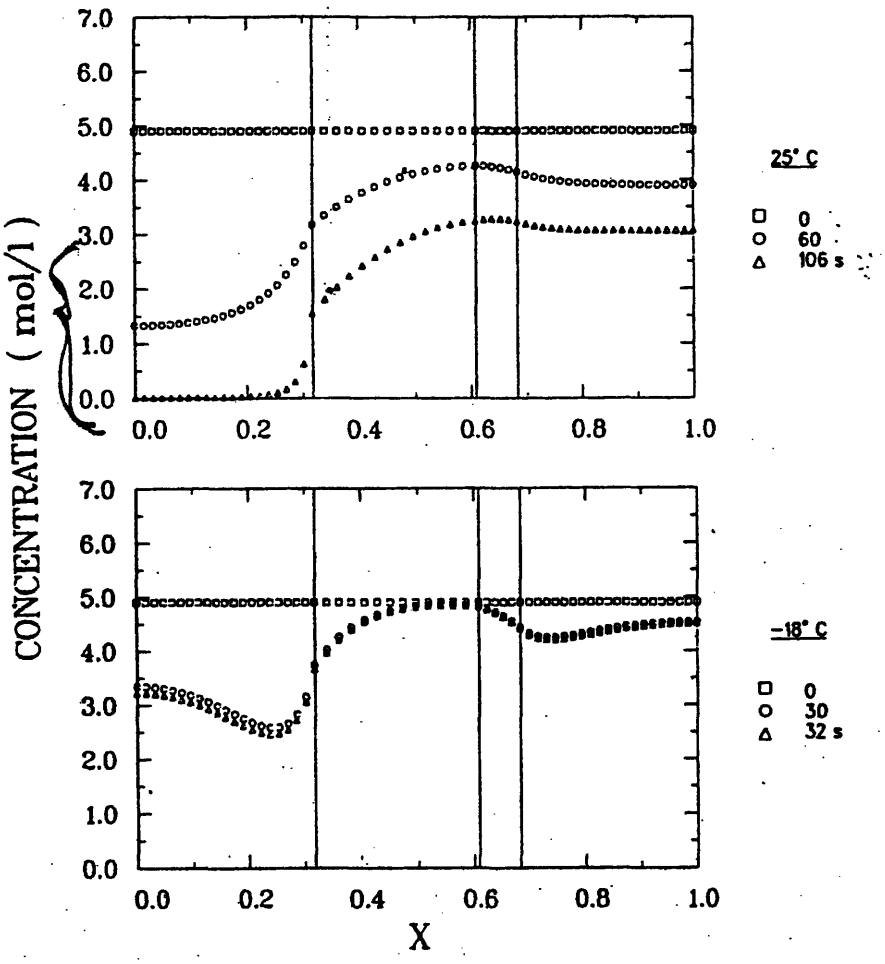
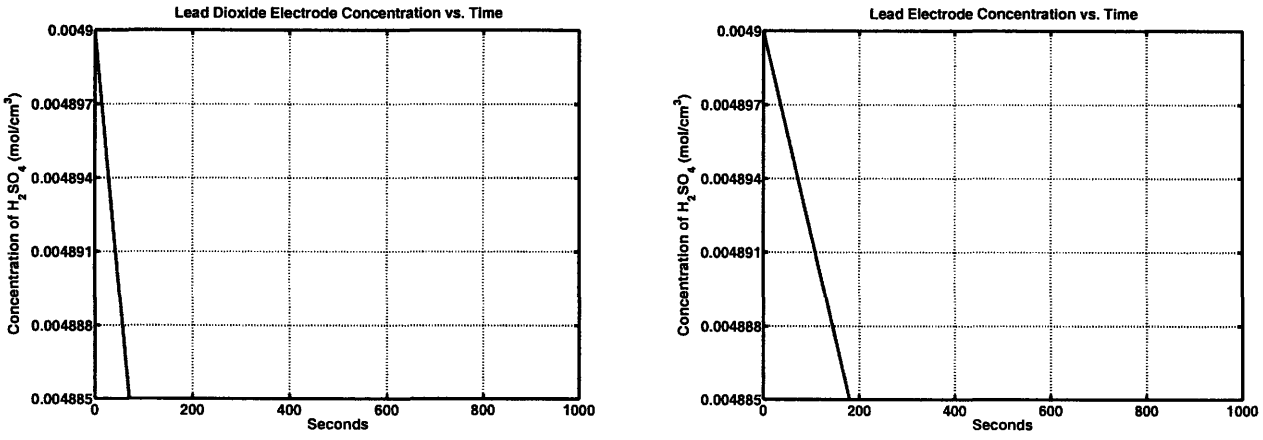


Fig. 3. Profiles of acid concentration during a discharge at 340 mA/cm². The regions from left to right are: positive electrode, reservoir, separator, and negative electrode. $X = 1.0$ is for $l = 0.19$ cm.

Figure 4-17.



(a) Lead dioxide electrode

(b) Lead (Pb) electrode

Figure 4-18. The concentration of the electrolyte at each electrode during discharge. The electrolyte diffusion coefficient has now been set to zero.

4.4.2 The diffusion coefficient, D_C

According to Nernst's equation eq. (4.92), the potential of an electrochemical system is a strong function of the concentration at the electrode/electrolyte interface. Therefore, if the diffusion coefficient were to go to zero, i.e. there is no diffusion from the bulk electrolyte into the electrodes, each electrode's concentration would should experience the leveling off seen in fig. 4-16. This is vividly illustrated in fig. 4-18. As a result, the cell potential would not level off like it did in fig. 4-15 but would instead continue to collapse as seen in fig. 4-19.

On the other hand, if diffusion were incredibly fast, each electrode's concentration would only change very little, fig. 4-20, so the cell potential should also experience minimal change, fig. 4-21.

In figs. 4-15, 4-16 and 4-18–4-21 the model produces believable diffusion like behavior in that ions diffuse from areas of high concentration to low concentration. Furthermore, the model properly exhibits a strong dependence of cell potential on electrolyte concentration as predicted by the Nernst Equation,

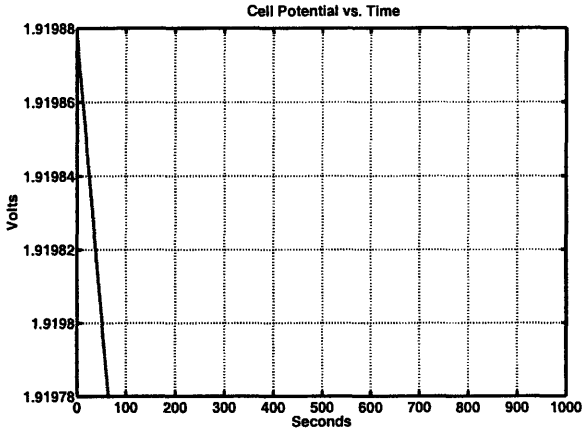
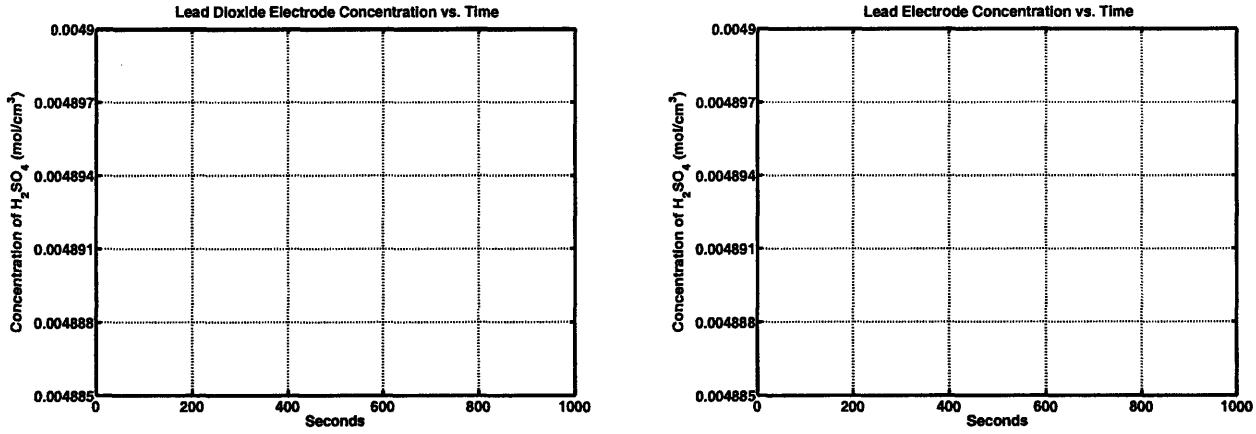


Figure 4-19. Cell potential vs. time curve when there is no diffusion of electrolyte from the bulk electrolyte into each electrode's porous region. The cell potential drops rapidly because the electrolyte is being rapidly consumed as seen in fig. 4-18. Without electrolyte, the battery cannot do work, and therefore, there is no EMF.



(a) Lead dioxide electrode

(b) Lead (Pb) electrode

Figure 4-20. The concentration of the electrolyte at each electrode during discharge. The diffusion coefficient in now effectively infinite, so the concentration does not change from its original value of $0.0049 mol/cm^3$

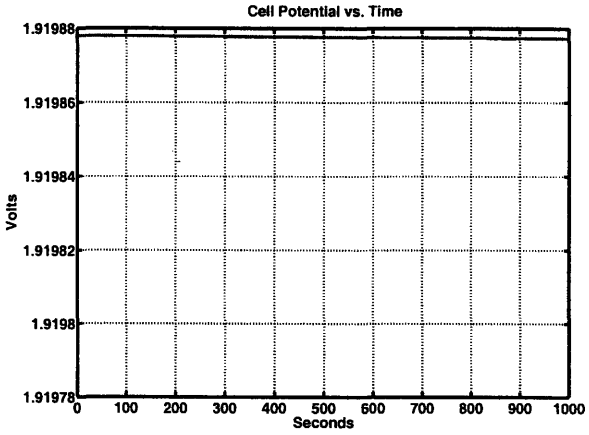


Figure 4-21. Cell potential vs. time curve when there is a fast diffusion rate of $10^{-2}\frac{A}{cm^2}$. The cell's potential doesn't change because there is no change in concentration of electrolyte.

eqs. (1.4) and (4.92).

4.4.3 The Interface Surface Area, SA_{int}

We now take a look at the cell potential vs time curve for our cell under charging conditions with initial conditions from section 4.3.2, seen in fig. 4-22.

Fig. 4-22, does not look like what one might expect from a charging battery. Even though the charging cell potential starts above the equilibrium cell potential of 2.041V, the cell potential decreases while charging. This is not what is expected to happen. When a battery charges, one would expect the cell potential to start above the equilibrium potential and be on an upward trajectory.

The reason the model does not behave realistically here lies in the fact that in a real cell, as the cell charges or discharges, the electrode/electrolyte interface area changes. This effect has not yet been taken into account in our model. Instead, a constant per unit volume electrode/electrolyte interface area of $\frac{\delta x \delta z}{\delta x \delta y \delta z} = \frac{1}{\delta y}$ has been assumed. Furthermore, the electrolyte is assumed to always be in constant contact with the entire electrode surface. This constant electrode/electrolyte interface area can be seen in the

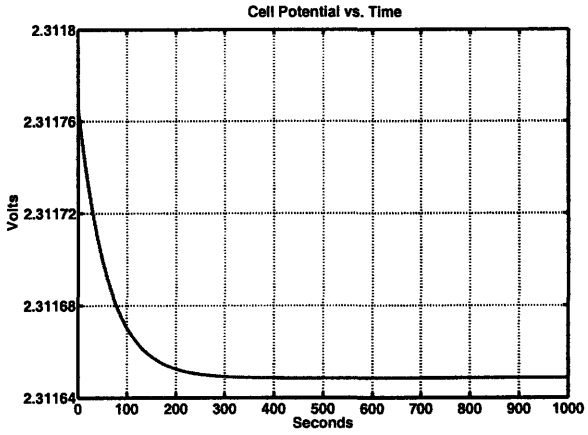


Figure 4-22. Cell potential vs. time curve as the simple cell of section 4.3 is charged at a rate of 50mA for 1000 seconds.

electrode kinetics equation eq. (4.86) which has been rewritten here in eq. (4.93).

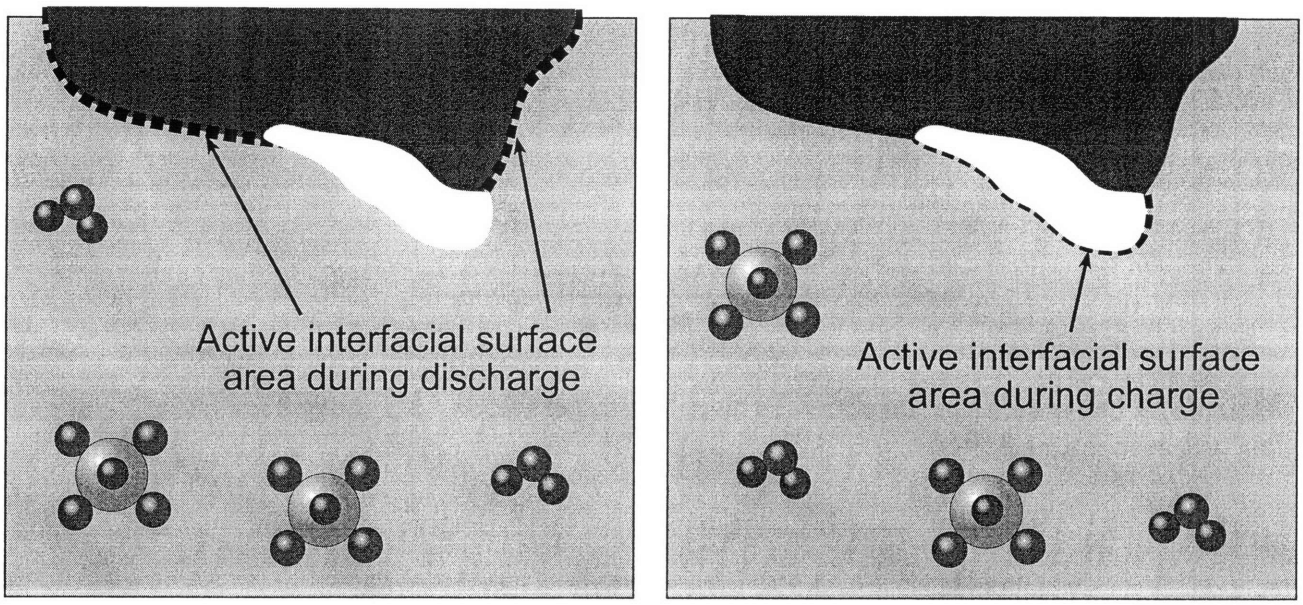
$$\frac{I}{\mathbf{F}\delta x\delta y\delta z}+\underbrace{\frac{1}{\delta y}}_{SA_{int}}\frac{i_0}{\mathbf{F}}\frac{C^{k+1}}{C^*}\left(e^{\frac{\alpha_a n\mathbf{F}({\phi_s}^{k+1}-{\phi_l}^{k+1}-\phi_{eq})}{RT}}-e^{\frac{\alpha_c n\mathbf{F}({\phi_l}^{k+1}-{\phi_s}^{k+1}+\phi_{eq})}{RT}}\right)=0 \qquad (4.93)$$

In order to have the model exhibit more proper charging behavior, we must implement a model of the electrode/electrolyte interface that incorporates the changes of the interface area due to reaction. When developing a model for the interfacial surface area, it is important to understand that the active material of interest is different in the case of discharging and charging. Under discharge

$$\mathrm{Pb(IV)O_2}\,(s)+\mathrm{HSO_4^-}\,(aq)+3\mathrm{H^+}\,(aq)+2\mathrm{e^-}\rightarrow\mathrm{Pb(II)SO_4}\,(s)+2\mathrm{H_2O}\,(l) \qquad (4.94)$$

$\mathrm{Pb(IV)O_2}$ is consumed, so the active material of interest is $\mathrm{Pb(IV)O_2}$, but when charging,

$$\mathrm{Pb(II)SO_4}\,(s)+2\mathrm{H_2O}\,(l)\rightarrow\mathrm{Pb(IV)O_2}\,(s)+\mathrm{HSO_4^-}\,(aq)+3\mathrm{H^+}\,(aq)+2\mathrm{e^-} \qquad (4.95)$$



(a) The surface area of the usable active material during discharge is indicated by a dashed line. Active material is usable if it is in contact with the sulfuric acid. Here the active material is lead dioxide. The white are is lead sulfate, which does not participatc in thc discharge reaction of the lead dioxide electrode eq. (4.94). Therefore, areas covered by lead sulfate cannot be included in the active interface area for the discharge reaction.

(b) When the battery charges, lead sulfate is necessary for the reaction to proceed. The lead sulfate is shown in white. The amount of lead sulfate in contact with the sulfuric acid determines the electrode/electrolyte interface's surface area. Here, as in fig. 4-23 (a), the interface is delineated by a dashed black line.

Figure 4-23. Discharge and Charge Regime Interfacial Surface Areas.

$Pb(II)SO_4$ is consumed, so the active material of interest is $Pb(II)SO_4$. Therefore, in a real battery, there is a different SA_{int} during charging and discharging. This difference of active interfacial area can be seen in fig. 4-23. Interestingly enough, under both discharging and charging regimes of operation, the amount of active material, and therefore active interfacial surface area always decreases independent of the direction of current flow.

There have been numerous attempts to model the change in the electrode/electrolyte interface area. Reference [32] provides a good summary of the more popular techniques. In general the electrode/elec-

trolyte interfacial surface area is modeled as:

$$\text{“}A = A_{max}\theta^{\beta 1} \rightarrow \text{discharge} \tag{4.96}$$

$$A = A_{max}\left(1 - \theta^{\beta 1}\right) \rightarrow \text{charge} \tag{4.97}$$

where A_{max} is the maximum active surface area $\left(\frac{cm^2}{cm^3}\right)$, and $\beta 1$ is the tortuosity exponent and θ is the state of charge of the battery [32, p44].” It would be virtually impossible to know A_{max} and there seems to be little agreement on the value for the tortuosity exponent. Some want to use 1.5 [31,33] while other prefer to use 0.5 [34,35]. Finally, state of charge θ is defined as:

$$\frac{\partial \theta}{\partial t} = \alpha_{Ah} \mathcal{Q}_{max}^{-1} \frac{\partial i_1}{\partial x} \tag{4.98}$$

where $\mathcal{Q}$ is the theoretical capacity $\left(\frac{C}{cm^3}\right)$ and α_{Ah} is the charge efficiency [32, p.45][4]. The charge efficiency supposedly improves model prediction during charging. Without it, “the accumulation rate of inaccuracy is 5-15% on every step if this parameter is ignored. This parameter was not used in the cell models published in literature [32, p.45].” It is unclear as to how the authors came up with these numbers. Furthermore, it should be pointed out that charge efficiency is probably a fairly difficult term to determine with any great degree of accuracy. However, the point remains that the models in the literature can be expected to provide little more than general trends in battery operation. General trends are still extremely useful as they can help us develop insight, but one should be careful and not try to use these

[4]While eq. (4.98), as reported in [32], uses $\frac{\partial i_1}{\partial x}$ in the denominator on the right hand side, it seems like this is an error and which should instead be $\frac{\partial i_1}{\partial t}$

models in situations where they should not be used.

Another, more successful, interface area model used while charging is:

$$A = A_{max}\rho\frac{exp(\gamma\theta) - exp(-\gamma)}{1 - exp(-\gamma)} \rightarrow \text{charge} \tag{4.99}$$

where $\rho = 3.5$ and $\gamma = 3.7$. But it is admitted that, "Exact morphology of porous electrode is not fully understood. [32, p.45]" Basically, it seems that while all due diligence has been given to the models for electrode/electrolyte interface area, they are just models to give general trends and not very accurate beyond that. Since the interface area is critical to electrochemistry, this inability to precisely model the morphology of the porous electrode limits these models to giving only trends and prevents them from being used as exact representations of real batteries.

For our simple model, we choose a third and different model of the interface area which takes advantage of the porosity as an indicator of the available electrode/electrolyte interface area [16, 30]. The model in [16, 30], has an interfacial surface area under discharge of the form,

$$SA_{int}^{\text{discharge}} = A_{max}\left(\frac{\varepsilon - \varepsilon_{\text{min}}}{\varepsilon_{\text{max}} - \varepsilon_{\text{min}}}\right)^{\varsigma} \tag{4.100}$$

and under charging the model in [30][5] proposes

$$SA_{int}^{\text{charge}} = A_{max}\left(\frac{\varepsilon - \varepsilon_{\text{min}}}{\varepsilon_{\text{max}} - \varepsilon_{\text{min}}}\right)^{\varsigma}\left(\frac{\varepsilon_{\text{max}} - \varepsilon}{\varepsilon_{\text{max}} - \varepsilon_{\text{min}}}\right) \tag{4.101}$$

ε_{max} and ε_{min} are the maximum and minimum possible porosities one might find in a volume. ς is a fudge factor that is often set to 1 [16, 30] which is what is done for our model. Substituting eq. (4.100)

[5]The model in [16] is only for discharge

into eq. (4.86) gives

$$\frac{I}{\mathbf{F}\delta x\delta y\delta z}+\underbrace{\frac{1}{\delta y}\left(\frac{\varepsilon-\varepsilon_{\min}}{\varepsilon_{\max}-\varepsilon_{\min}}\right)^{\varsigma}}_{\text{Discharge } SA_{int} \text{ model}}\frac{i_0}{\mathbf{F}}\frac{C^{k+1}}{C^*}\left(e^{\frac{\alpha_a n\mathbf{F}({\phi_s}^{k+1}-{\phi_l}^{k+1}-\phi_{eq})}{RT}}-e^{\frac{\alpha_c n\mathbf{F}({\phi_l}^{k+1}-{\phi_s}^{k+1}+\phi_{eq})}{RT}}\right)=0 \quad (4.102)$$

which is the new electrode kinetics equation under discharge. The new electrode kinetics equation under charging conditions is:

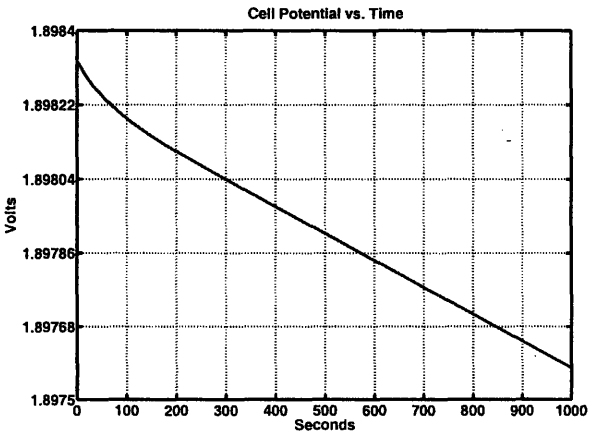
$$\frac{I}{\mathbf{F}\delta x\delta y\delta z}+\underbrace{\frac{1}{\delta y}\left(\frac{\varepsilon-\varepsilon_{\min}}{\varepsilon_{\max}-\varepsilon_{\min}}\right)^{\varsigma}\left(\frac{\varepsilon_{\max}-\varepsilon}{\varepsilon_{\max}-\varepsilon_{\min}}\right)}_{\text{Charge } SA_{int} \text{ model}}\frac{i_0}{\mathbf{F}}\frac{C^{k+1}}{C^*}\left(e^{\frac{\alpha_a n\mathbf{F}({\phi_s}^{k+1}-{\phi_l}^{k+1}-\phi_{eq})}{RT}}-e^{\frac{\alpha_c n\mathbf{F}({\phi_l}^{k+1}-{\phi_s}^{k+1}+\phi_{eq})}{RT}}\right)=0 \quad (4.103)$$

Using these two electrode kinetics equations and the initial conditions from section 4.3.2 in the simulations gives fig. 4-24.

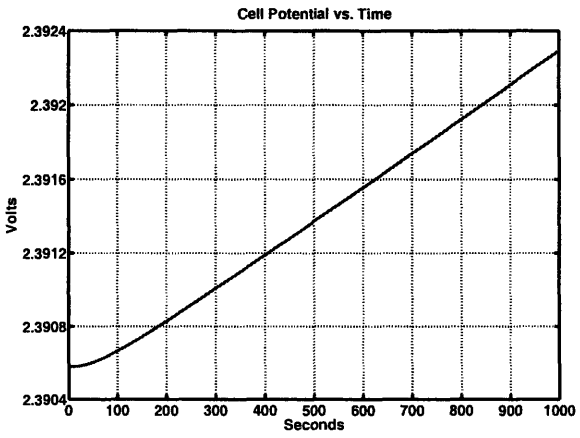
The curves in fig. 4-24 are more believable than those in fig. 4-15 for a couple of reasons. First, both curves head in the correct direction. The cell potential under discharge decreases while the cell potential under charging increases. Also, neither curve plateaus as they did in the previous section. This shows that, in spite of diffusion, because the interfacial surface area is decreasing, the output potentials can change.

4.4.4 The transfer coefficients, α_a and α_c

Fig. 4-25 shows the constant current which must be applied to the model for 750 seconds, to produce at the end of that time, the overpotential indicated. Charging currents are negative and discharging



(a) Discharge 50mA with eq. (4.100) as the model for interfacial surface area.



(b) Charging 50mA with eq. (4.101) as the model for interfacial surface area.

Figure 4-24. Discharge and Charge Curves using a ξ dependent time varying electrode/electrolyte interfacial surface area.

currents are positive. Therefore, the curve to the left of $\eta_{\text{cell}} = 0$ shows the charging behavior of the battery while the portion of the curve to the right of $\eta_{\text{cell}} = 0$ shows the discharging behavior of the cell.

Fig. 4-25 has many noteworthy characteristics. First, the larger the discharging current, the more the cell's potential collapses, and the larger the charging current, the more the cell's potential increases. Said in another way, the further the cell electrodes are perturbed from their respective equilibrium potentials, the faster the reaction proceeds at that electrode. This is very reasonable behavior for a lead-acid cell.

The IV curve in fig. 4-25 also clearly shows that the cell exhibits an asymmetry in its behavior between charging and discharging. Given a certain magnitude of current flow through the battery, it takes a significantly larger deviation from cell equilibrium potential to have the battery charge than to have it discharge at the same rate. This asymmetry is largely a function of the transfer coefficients α_a and α_c. In fig. 4-25, α_a does not equal α_c. Setting the two transfer coefficients equal to each other largely, but not completely, eliminates the asymmetry in the IV characteristic as seen in fig. 4-26. The remaining

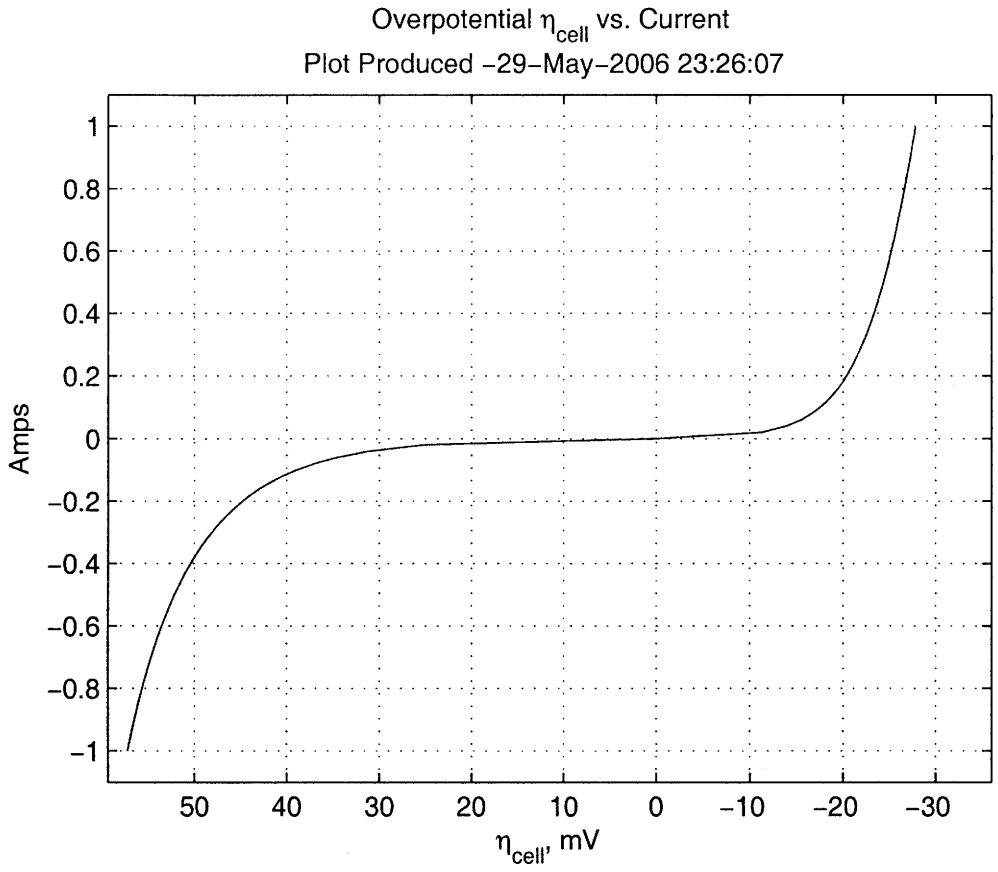


Figure 4-25. Single Cell IV Characteristics after 750 seconds of charge or discharge. The cell was always started from the same initial conditions for each current used.

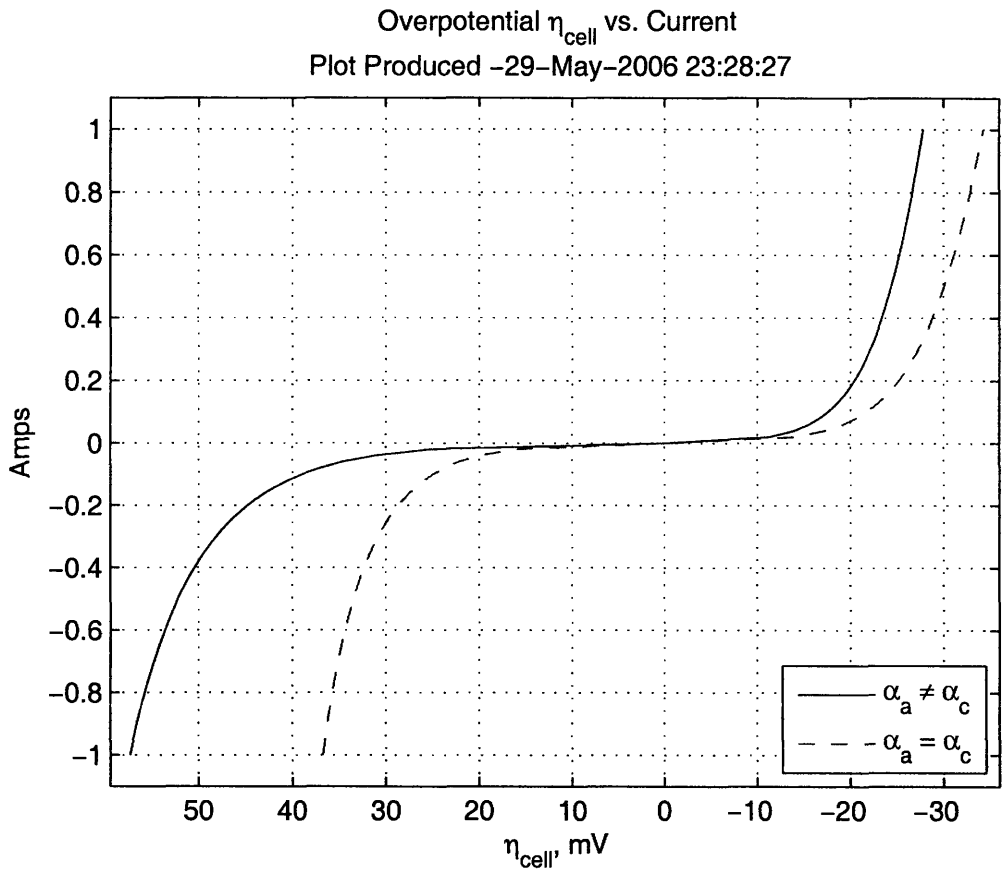


Figure 4-26. This figure illustrates the dramatic effect that the transfer coefficients α_a and α_c have on the behavior of the cell. The solid curve is the same curve as that in fig. 4-25 with values of α_c and α_a coming from Table 4.1 and Table 4.2. The dashed curve comes from a simulation of the same cell as that used to draw the solid curve with the exception that the two transfer coefficients are now equal. Notice how the asymmetry of the solid curve has largely been eliminated in the dashed curve.

asymmetry is due to the interfacial surface areas not being equal between charge and discharge. Fig. 3.4.3 of [28, p.101] shows similar results.

4.4.5 The exchange current density, i_0

Finally, the shape of the IV curve is greatly influenced by the magnitude of the exchange current densities, $i_0^{Pb(IV)O_2}$ and i_0^{Pb}. The larger the exchange current density, the less the cell needs to be perturbed from equilibrium potential in order to get a certain amount of current to pass through the cell. The effect

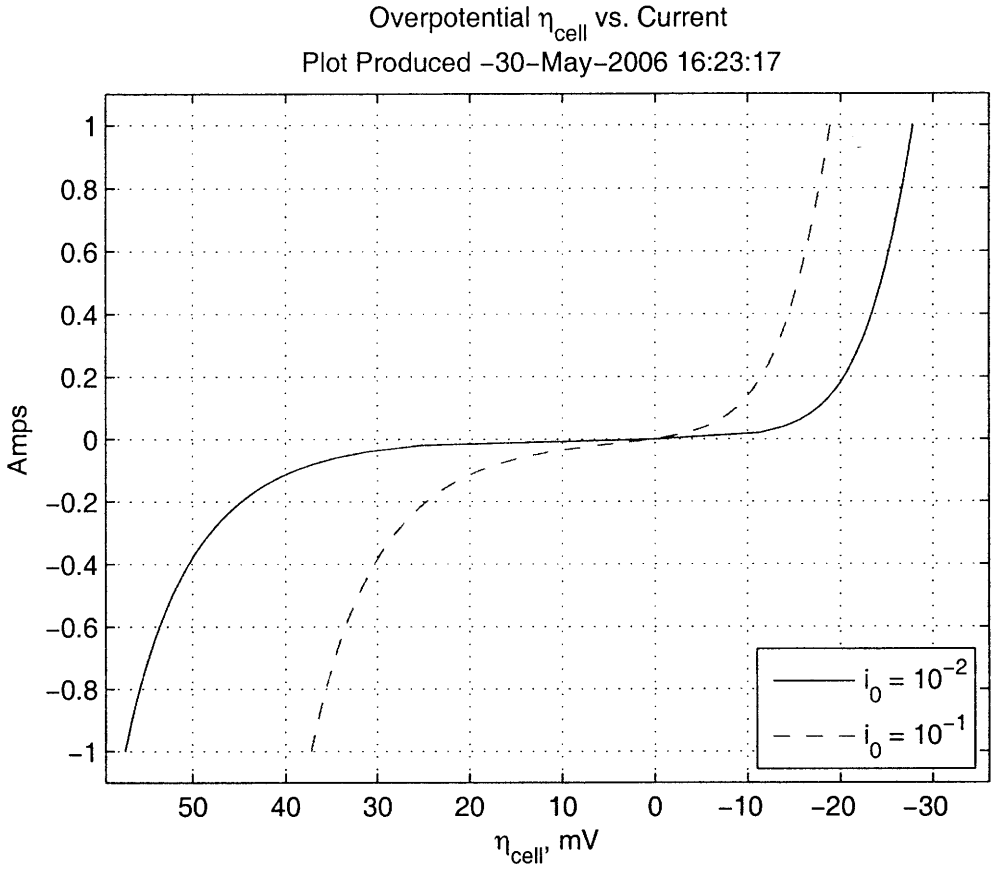


Figure 4-27. Single Cell IV Characteristics after 750 seconds of charge or discharge. The cell was always started from the same initial conditions for each current used. The IV characteristics of the cell show a strong dependence on the exchange current density found in the electrode kinetics equation. For both curves, $i_0^{Pb(IV)O_2} = i_0^{Pb}$.

of increasing the exchange current density can be seen in fig. 4-27. In this figure, the solid blue line is the same as the solid blue line in fig. 4-25. The exchange current densities $i_0^{Pb(IV)O_2}$ and i_0^{Pb} are both 10^{-2} for the solid blue line and 10^{-1} for the dashed red line. Notice now how the same current output as the blue line can be achieved by the red line with a much smaller perturbation from equilibrium. Fig. 3.4.2 of [28, p.101] shows similar results.

4.4.6 Simple Model Summary

The purpose of creating the simplified model was to develop a tool that would allow us to learn about the internal dynamics of a lead-acid cell and the relationship between the cell's internal state and its external responses. The model's simple geometry gives rise is a set of easily coded, computed, and modified equations. The cell exhibits battery like IV behavior and has been demonstrated to correctly show how several different internal physical parameters effect the cell behavior. Furthermore, the discussion of section 4.4.3 about interfacial surface area exposes a fundamental weakness in electrochemical models for lead-acid batteries. This weakness cannot be readily corrected and thereby limits the usability of these sorts of fundamental models. However, the model can still be used to gain insight into the processes that occur within a lead-acid battery. Using the insight gained from this model, we can now develop a multidimensional model that is not constrained to a particular geometry or volume discretization scheme like the one from fig. 4-14 that was used to derive our model's equations. This new 2-D model will be able to capture the two-level effect that we are concerned about.

Chapter 5

Two-dimensional model

In the last chapter, a simplified model of a lead-acid cell was developed. While we started deriving the equations for the model from the fixed geometry of fig. 4-3, the final equations eqs. (4.82)–(4.91), are written in terms of porosity ε and are not dependent on the initial geometry. However, the final equations are dependent on the single volume per electrode discretization scheme used. This chapter will modify the equations used from the previous model so that they are multidimensional, independent of discretization scheme, and account for a factor known as tortuosity. Furthermore, it will present model outputs that show that it captures the two-level discharge behavior of fig. 1-6(b).

Essentially, we will modify our equations so as to implement a macroscopic model very similar to the one developed by Bernardi and Gu in [31]. Since both our fixed geometry model and a macroscopic model are attempts at dealing with the complex porous nature of the electrodes, we begin our transition from fixed geometry model to macroscopic model by taking a closer look at how porosity affects the physical parameters of the system.

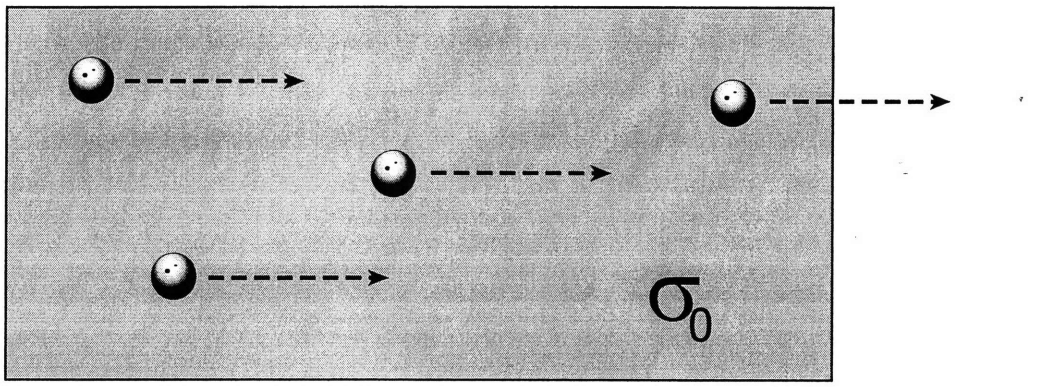


Figure 5-1. A bar of conductivity σ_0. Electrons can move through it without running into obstructions or having to change course.

5.1 Model Modifications

5.1.1 Porosity

As a reminder, in section 4.2.1, porosity was defined to be the ratio of the void volume within a region to the total volume of the region. In our model, all void space is assumed to be filled with electrolyte, so in our case, porosity is the ratio of the liquid phase volume to the total volume of a region. To understand how the porosity can affect physical parameters of a battery, let us start with a solid conducting bar of conductivity σ_0, as seen in fig. 5-1. Here there are no obstructions in the path of the electrons, so they are able to move as freely as possible across the bar. Their movement is only limited by the value of the bar's natural conductivity σ_0.

If the bar in fig. 5-1 were to have voids drilled into it at random points along its length as seen in fig. 5-2, the voids would make it more difficult for electrons to traverse the bar. In other words, the voids reduce the mobility of the electrons within the bar.

In section 4.2.3, we saw that conductivity is a strong function of charge carrier mobility. Now, since the mobility of the electrons has decreased, the effective conductivity of the bar has also decreased. One

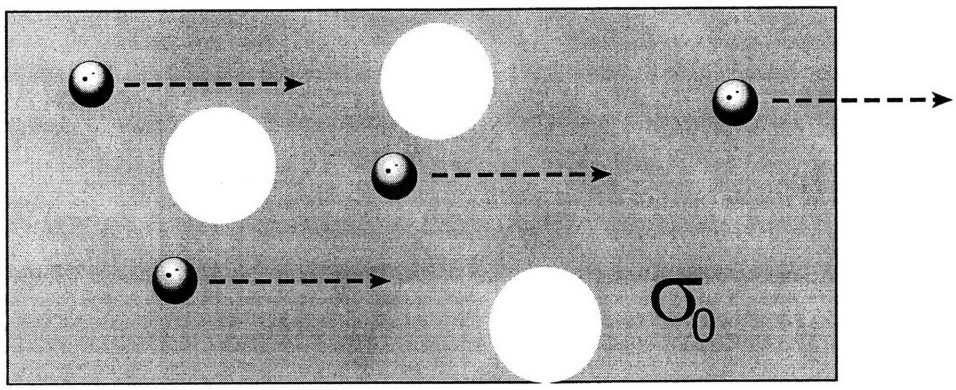


Figure 5-2. A bar of conductivity σ_0. The voids now obstruct the path of some of the electrons, thereby reducing the mobility of the electrons. The lower mobility of the electrons means that the bar has a lower conductivity.

could, in fact, replace the bar with voids, by a solid bar that has a lower conductivity but no voids. To do this, one would multiply the conductivity of the bar with voids by the total solid volume of that bar to get the conductance of the bar with voids. One would then divide that conductance by the volume of the bar that has no voids. By this method, if σ_0 is the conductivity of a solid bar without the presence of pores, then

$$\sigma^{\text{eff}} \delta x \delta y \delta z = (1 - \varepsilon)\, \sigma_0 \delta x \delta y \delta z$$

$$\sigma^{\text{eff}} = (1 - \varepsilon)\, \sigma_0 \tag{5.1}$$

might be an acceptable model for the conductivity of a solid bar of the same outer dimensions as the porous bar.

Model eq. (5.1) fails to capture an effect know as *tortuosity*. Tortuosity describes how the location of the voids in a material affect the properties of the material. In our case, tortuosity describes how the conductivity of the bar is affected by the location of the voids. For example, the pore pattern in fig. 5-2

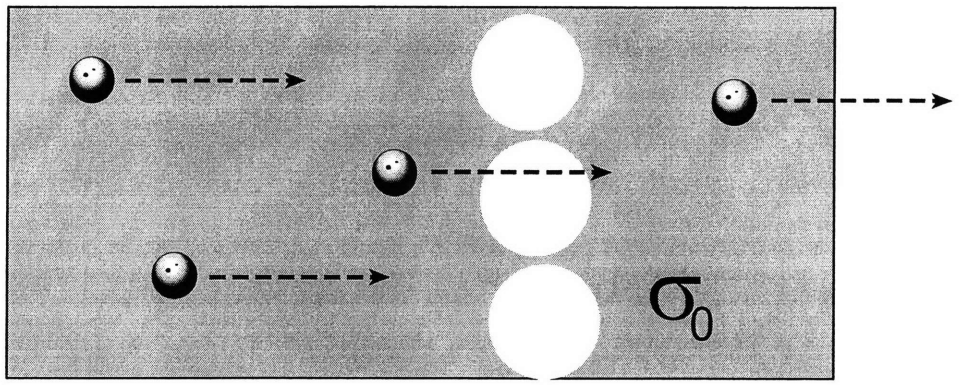


Figure 5-3. This bar has same porosity as the bar in fig. 5-2; however, because of the way in which the voids are arranged, it has a much lower conductivity.

should have a quantifiably smaller effect on the movement of electrons than the pore pattern of fig. 5-3. Therefore, one would expect the effective conductivity of the bar in fig. 5-3 to be much lower than that of the bar in fig. 5-2. Our model eq. (5.1), however, would produce the same effective conductivity for both bars.

To take tortuosity into effect, we might modify our model as follows,

$$\sigma^{\text{eff}} = (1-\varepsilon)^{\beta}\,\sigma_0 \tag{5.2}$$

where β is a number that is chosen so as $(1-\varepsilon)^{\beta}$ realistically captures the tortuosity effect.

A similar development can be used for the liquid phase. This will result in an identical model in terms of the liquid conductivity κ.

$$\kappa^{\text{eff}} = \varepsilon^{\beta}\kappa_0 \tag{5.3}$$

The literature does not carry both the $(1-\varepsilon)$ term and the ε terms around. Instead, only the ε term is used and a different tortuosity scale factor is used for the solid phase and the liquid phase. For the solid

phase, instead of using the tortuosity factor β, the tortuosity factor is represented by exm. In the liquid phase, instead of β, the tortuosity factor is written as ex which is *different* than the solid phase's exm factor. These two different parameters allow us to do away with the $(1-\varepsilon)$ terms and simply write

$$\sigma^{\text{eff}} = \varepsilon^{exm}\sigma_0$$
$$\kappa^{\text{eff}} = \varepsilon^{ex}\kappa_0 \tag{5.4}$$

as our models for the solid and liquid conductivity of a solid bar that covers the same volume as the bar with voids. Equations like those of eq. (5.4) are known as Bruggeman equations [27].

These models for tortousity allow us to replace a section of porous material with a section of continuous material. Both the solid phase and liquid phases can now be thought of as continuous phases that occupy the same space at the same time as seen in fig. 5-4. The movement between the two phases will be described by our electrode kinetics equation.

5.2 Equations with geometry independent porosity

Applying the modifications discussed in the previous section to the equations developed in section 4.3 gives equations eq. (5.5)-eq. (5.18), where we are now using the flow of positive charge as our definition of current. These equations are almost the same as those found in [31] with the exception of the diffusion term in the Conservation of Matter equation, eqs. (5.6), (5.11) and (5.16). Their diffusion term has been simplified for linear diffusion where as ours keeps all non-linearities.

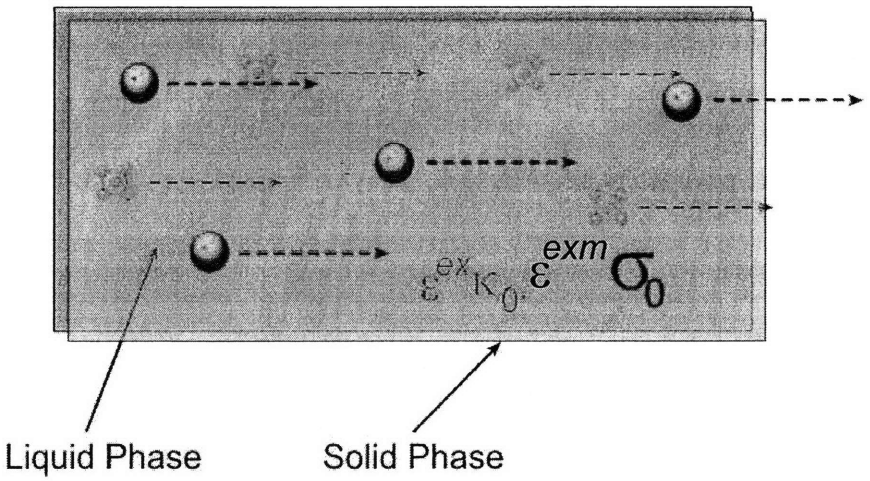


Figure 5-4. The effective conductivity idea allows us to think of the solid phase as a continuum and the liquid phase as a continuum. Both continua are thought of as occupying the same space at the same time. Movement of charge between the two continua is dictated by the electrode kinetics equation.

5.2.1 Equations for lead dioxide electrode

1. Change of Porosity

$$\frac{\partial \varepsilon}{\partial t} - \frac{1}{2\mathbf{F}}\left[\frac{MW_{PbSO_4}}{\rho_{PbSO_4}} - \frac{MW_{PbO_2}}{\rho_{PbO_2}}\right]\nabla \cdot i_l = 0 \tag{5.5}$$

2. Conservation of Matter

$$\left(\frac{\partial \varepsilon C}{\partial t}\right) + \nabla \cdot (-\varepsilon^{ex} D \nabla C) + \left(\frac{2t_+^\circ - 3}{2\mathbf{F}}\right)\nabla \cdot i_l = 0 \tag{5.6}$$

3. Electrode Kinetics

$$\nabla \cdot i_l - SA_{int} i_0^{\mathrm{Pb(IV)O_2}} \frac{C}{C^*}\left(e^{\frac{\alpha_a n(\phi_s - \phi_l - \phi^{eq}_{\mathrm{Pb(IV)O_2}})\mathbf{F}}{RT}} - e^{\frac{\alpha_c n(\phi_l - \phi_s + \phi^{eq}_{\mathrm{Pb(IV)O_2}})\mathbf{F}}{RT}}\right) = 0 \tag{5.7}$$

4. Conservation of Charge

$$\nabla \cdot i_l + \nabla \cdot i_s = 0$$

$$\nabla \cdot i_l + \nabla \cdot \left(-\varepsilon^{exm} \sigma_s \nabla \phi_s\right) = 0 \tag{5.8}$$

5. Ohm's Law in Solution

$$i_l - \left(-\varepsilon^{ex} \kappa \nabla \phi_l\right) + \left(-\varepsilon^{ex} \kappa \left(1 - 2t_+^\circ\right) \frac{\mathrm{RT}}{\mathbf{F}} \nabla ln\left(C\right)\right) = 0 \tag{5.9}$$

5.2.2 Equations for the lead electrode

1. Change of Porosity

$$\frac{\partial \varepsilon}{\partial t} - \frac{1}{2\mathbf{F}} \left[\frac{MW_{Pb}}{\rho_{Pb}} - \frac{MW_{PbSO_4}}{\rho_{PbSO_4}}\right] \nabla \cdot i_l = 0 \tag{5.10}$$

2. Conservation of Matter

$$\left(\frac{\partial \varepsilon C}{\partial t}\right) + \nabla \cdot \left(-\varepsilon^{ex} D \nabla C\right) + \left(\frac{2t_+^\circ - 1}{2\mathbf{F}}\right) \nabla \cdot i_l \tag{5.11}$$

3. Electrode Kinetics

$$\nabla \cdot i_l + SA_{int} \frac{i_0^{\mathrm{Pb}}}{\mathbf{F}} \frac{C}{C^*} \left(e^{\frac{\alpha_a n(\phi_s - \phi_l - \phi_{\mathrm{Pb}}^{eq})\mathbf{F}}{RT}} - e^{\frac{\alpha_c n(\phi_l - \phi_s + \phi_{\mathrm{Pb}}^{eq})\mathbf{F}}{RT}}\right) = 0 \tag{5.12}$$

4. Conservation of Charge

$$\nabla \cdot i_l + \nabla \cdot i_s = 0$$

$$\nabla \cdot i_l + \nabla \cdot (-\varepsilon^{exm} \sigma_s \nabla \phi_s) = 0 \tag{5.13}$$

5. Ohm's Law in Solution

$$i_l - (-\varepsilon^{ex} \kappa \nabla \phi_l) + \left(-\varepsilon^{ex} \kappa \left(1 - 2t_+^\circ\right) \frac{\mathrm{RT}}{\mathbf{F}} \nabla ln\left(C\right) \right) = 0 \tag{5.14}$$

5.2.3 Equations for the bulk electrolyte

1. Change of Porosity

$$\frac{\partial \varepsilon}{\partial t} = 0 \tag{5.15}$$

2. Conservation of Matter

$$\left(\frac{\partial C}{\partial t} \right) + \nabla \cdot (-D \nabla C) = 0 \tag{5.16}$$

3. Conservation of Charge

$$\nabla \cdot i_l = 0 \tag{5.17}$$

4. Ohm's Law in Solution

$$i_l - (-\kappa \nabla \phi_l) + \left(-\kappa \left(1 - 2t_+^\circ\right) \frac{\mathrm{RT}}{\mathbf{F}} \nabla ln\left(C\right) \right) = 0 \tag{5.18}$$

5.3 Sophisticated Numerical Model

There is a striking similarity between the equations for each region, so they can be replaced by one set of universally applicable equations, eqs. (5.19)–(5.22).

5.3.1 Equations for all regions

1. Change of Porosity

$$\frac{\partial}{\partial t}(\varepsilon) + SF_{cop}\left\{\nabla\cdot\left(-\kappa^{\text{eff}}\nabla\phi_l\right) + \nabla\cdot\left(-\kappa_{ec}^{\text{eff}}\nabla ln\left(C\right)\right)\right\} = 0 \tag{5.19}$$

2. Conservation of Matter

$$\frac{\partial}{\partial t}(\varepsilon C) + \nabla\cdot\left(-D^{\text{eff}}\nabla C\right) + SF_{MB}\left\{\nabla\cdot\left(-\kappa^{\text{eff}}\nabla\phi_l\right) + \nabla\cdot\left(-\kappa_{ec}^{\text{eff}}\nabla ln\left(C\right)\right)\right\} = 0 \tag{5.20}$$

3. Electrode Kinetics

$$\nabla\cdot\left(-\kappa^{\text{eff}}\nabla\phi_l\right)+\nabla\cdot\left(-\kappa_{ec}^{\text{eff}}\nabla ln\left(C\right)\right)-SA_{int}i_0\frac{C}{C^*}\left(e^{\frac{\alpha_a n(\phi_s-\phi_l-\phi^{eq})\mathbf{F}}{RT}} - e^{\frac{\alpha_c n(\phi_l-\phi_s+\phi^{eq})\mathbf{F}}{RT}}\right) = 0 \tag{5.21}$$

4. Conservation of Charge

$$\nabla\cdot\left(-\kappa^{\text{eff}}\nabla\phi_l\right) + \nabla\cdot\left(-\kappa_{ec}^{\text{eff}}\nabla ln\left(C\right)\right) + \nabla\cdot\left(-\sigma^{\text{eff}}\nabla\phi_s\right) = 0 \tag{5.22}$$

Where κ^{eff} and κ_{ec}^{eff} come from Ohm's Law in Solution,

$$i_l = -\underbrace{\varepsilon^{ex}\kappa}_{\kappa^{\text{eff}}}\nabla\phi_l + -\underbrace{\varepsilon^{ex}\kappa\left(2t_+^\circ - 1\right)\frac{\text{RT}}{\text{F}}}_{\kappa_{ec}^{\text{eff}}}\nabla ln\left(C\right)$$

$$= -\kappa^{\text{eff}}\nabla\phi_l + -\kappa_{ec}^{\text{eff}}\nabla ln\left(C\right) \tag{5.23}$$

and SF_{cop}, SF_{mb}, SA_{int}, ϕ^{eq}, and σ are region dependent and are given by,

$$SF_{cop} = \begin{cases} -\frac{1}{2\text{F}}\left[\frac{MW_{PbSO_4}}{\rho_{PbSO_4}} - \frac{MW_{PbO_2}}{\rho_{PbO_2}}\right] & \text{lead dioxide electrode} \\ 0 & \text{electrolyte} \\ -\frac{1}{2\text{F}}\left[\frac{MW_{Pb}}{\rho_{Pb}} - \frac{MW_{PbSO_4}}{\rho_{PbSO_4}}\right] & \text{lead (Pb) electrode} \end{cases} \tag{5.24}$$

$$SF_{mb} = \begin{cases} \left(\frac{2t_+^\circ - 3}{2\text{F}}\right) & \text{lead dioxide electrode} \\ 0 & \text{electrolyte} \\ \left(\frac{2t_+^\circ - 1}{2\text{F}}\right) & \text{lead (Pb) electrode} \end{cases} \tag{5.25}$$

$$SA_{int} = \begin{cases} SA_{int}^{\text{Pb(IV)O}_2} & \text{lead dioxide electrode} \\ 0 & \text{electrolyte} \\ SA_{int}^{\text{Pb}} & \text{lead (Pb) electrode} \end{cases} \tag{5.26}$$

$$\phi^{eq} = \begin{cases} \phi_{\text{Pb(IV)O}_2}^{eq} & \text{lead dioxide electrode} \\ 0 & \text{electrolyte} \\ \phi_{\text{Pb}}^{eq} & \text{lead (Pb) electrode} \end{cases} \tag{5.27}$$

$$\sigma^{\text{eff}} = \begin{cases} \varepsilon^{exm}\sigma^{\text{Pb(IV)O}_2} & \text{lead dioxide electrode} \\ 0 & \text{electrolyte} \\ \varepsilon^{exm}\sigma^{\text{Pb}} & \text{lead (Pb) electrode} \end{cases} \tag{5.28}$$

5.4 Layout of Volumes

The 2-D model tries to emulate the behavior of the lead-acid cell that is made up of a single lead dioxide electrode, a single lead electrode and a sulfuric acid bath. Such a lead-acid cell can be seen in fig. 1-3. The region to be simulated was divided up using a *staggered* grid like that first suggested

in [36] and detailed in [37]. When using a staggered grid, there are two or more different grids on which properties are centered. In the case of the present model, one grid is called the FV grid. This stands for Flux Volume grid. Fluxes and the porosity ε are centered at the center of these grids. The other type of grid is the PV grid. This stands for Potential Volume grid. The properties C, ϕ_l, and ϕ_s are centered at the center of volumes on this grid. Why the porosity is not centered on this grid will become apparent during the following discussion.

Fig. 5-5 shows a 1-dimensional cell model with a staggered grid layout in which only the fluxes are centered on a grid that is offset from the properties ε, C, ϕ_s, and ϕ_l.

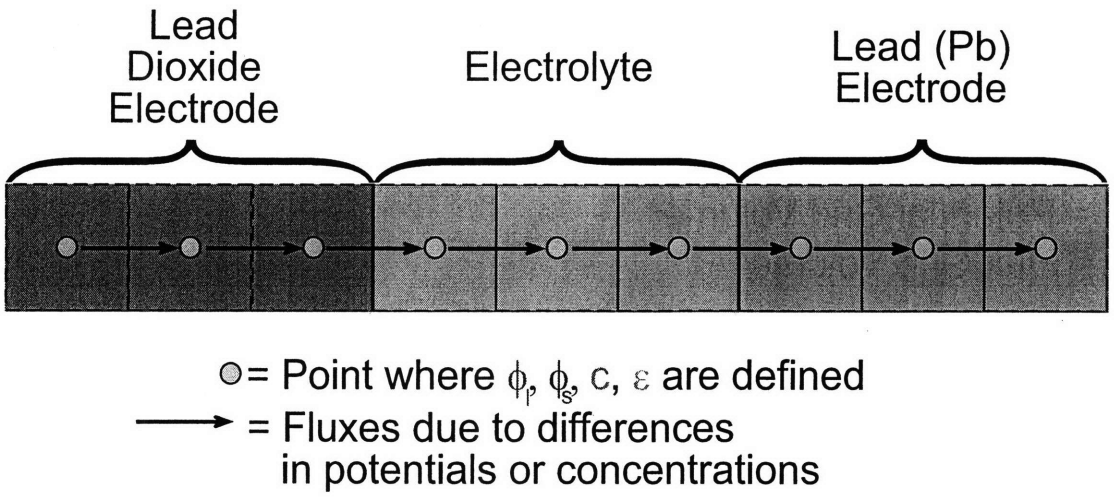


Figure 5-5. A one dimensional model of a lead (Pb) acid battery that has been divided up into finite volumes. The potentials ϕ_l, ϕ_s, the concentration C, and the physical property ε are all defined in the center of the volumes.

Defining all the property values at the center of the volumes creates the problem that the flux densities between the properties in adjacent volumes will now be discontinuous when the porosities ε of the two volumes are not equal. Unequal porosities means a discontinuity in properties like conductivity and diffusion coefficient on the boundaries between volumes. This, in turn, means that fluxes will be discontinuous on the boundaries. This is a problem for the finite volume method which uses the

divergence theorem to convert volume integrals into surface integrals. The flux densities will be ill defined on the boundaries, so it will be impossible to take surface integrals on the boundaries. To avoid this problem, the properties ϕ_l, ϕ_s, and C are all staggered off the grid that ε is defined on, fig. 5-6.

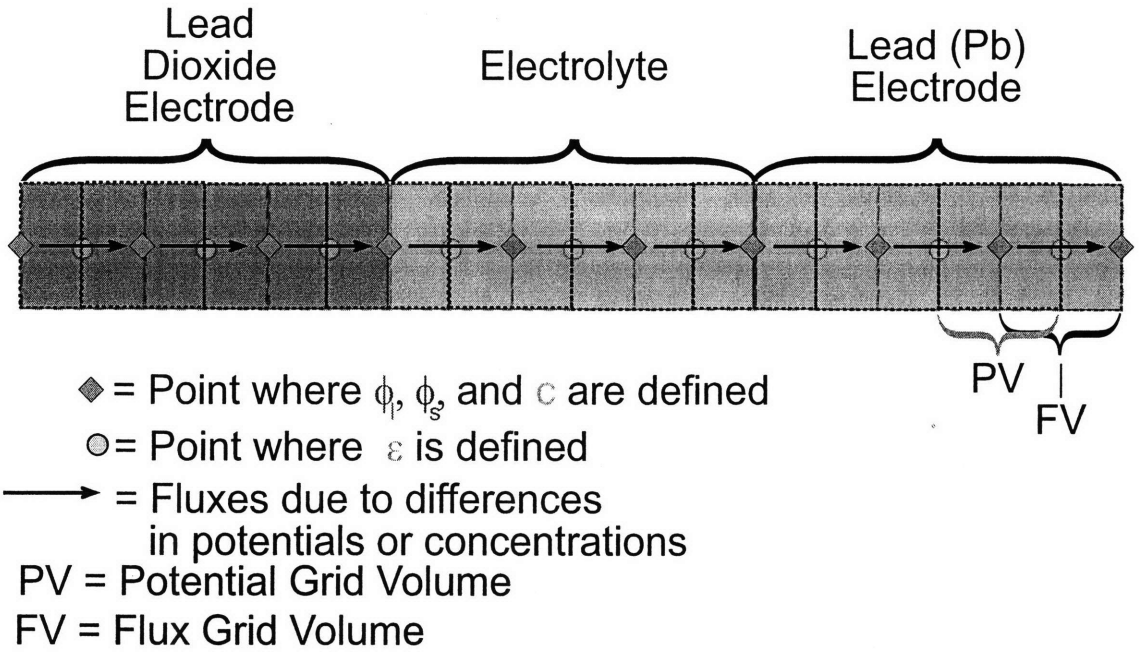


Figure 5-6. A one dimensional model of a lead (Pb) acid battery that has been divided up into two staggered grids of finite volumes.

The fluxes, which are defined to be between the values C, ϕ_s, and ϕ_l, are now all in regions of constant material property and are well defined on the borders of the PV volumes which pass through the centers of the FV volumes. Furthermore, all the derivatives used to determine the fluxes are now second order centered differences as opposed to the right sided/left sided derivatives that would be needed when only one grid was being used. They are centered differences because the derivatives that create the fluxes are now defined at the center of the FV grid instead of at the center of the PV grid.

The PV volumes now overlap two different FV volumes, but this is ok as the divergence theorem uses surface integrals, so the contribution of to the divergence of a PV volume from the left side of the PV

volume and the right side of the PV volume can be computed independently and simply added together to get the total divergence for that PV volume.

Finally, the PV volumes on each end are not the same size as the other PV volumes. Accounting for this sort of difference is the type of small problem that makes programming tricky. The problem was dealt with by saying that each PV is made up of two Side Volumes (SV). Then each of these Side Volumes can either exist or not exist. If they exist, then they can contribute to the overall divergence of the volume. If they do not exist, then they do not contribute to the divergence. By testing to see if an SV exists before trying to compute the equations for that SV, we can now use the same set of equations for all PVs.

5.4.1 Two Dimensions

Moving to two dimensions is a little more tricky. The physical property ϵ is still defined at the center points of the FV volumes and the potentials ϕ_l, ϕ_s, and the concentration C are still all defined in the center of the *staggered* PV volumes; however, the PV volumes now are not only offset in the x direction but also in the y direction as seen in fig. 5-7.

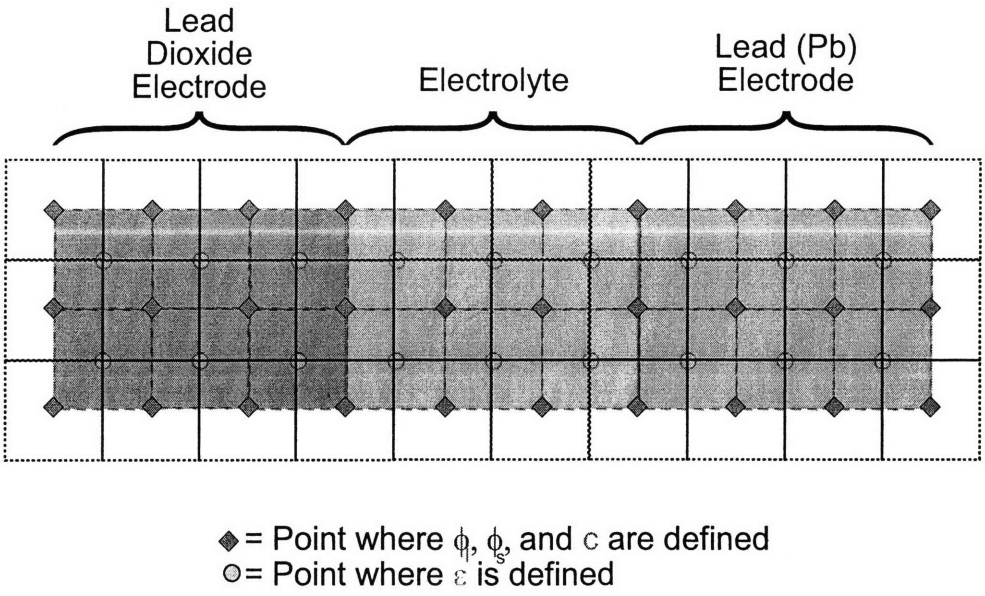


Figure 5-7. A two dimensional model of a lead (Pb) acid battery that has been divided up into two staggered grids of finite volumes.

Now, the questions becomes, where should the fluxes be defined? One cannot simply draw a flux from one PV to its nearest neighbor because this would run along a boundary where at least two FVs meet. The porosity, conductivity and other physical properties would not be defined at these boundaries.

This problem is tackled by noticing how each of the PVs can be subdivided into 4 different sections in much the same way that each of the PVs in the one dimensional model were subdivided into two sections, fig. 5-8. These are the 4 corners of the PVs, so the areas are called Corner Volumes or CVs. Each CV is associated with a different FV.

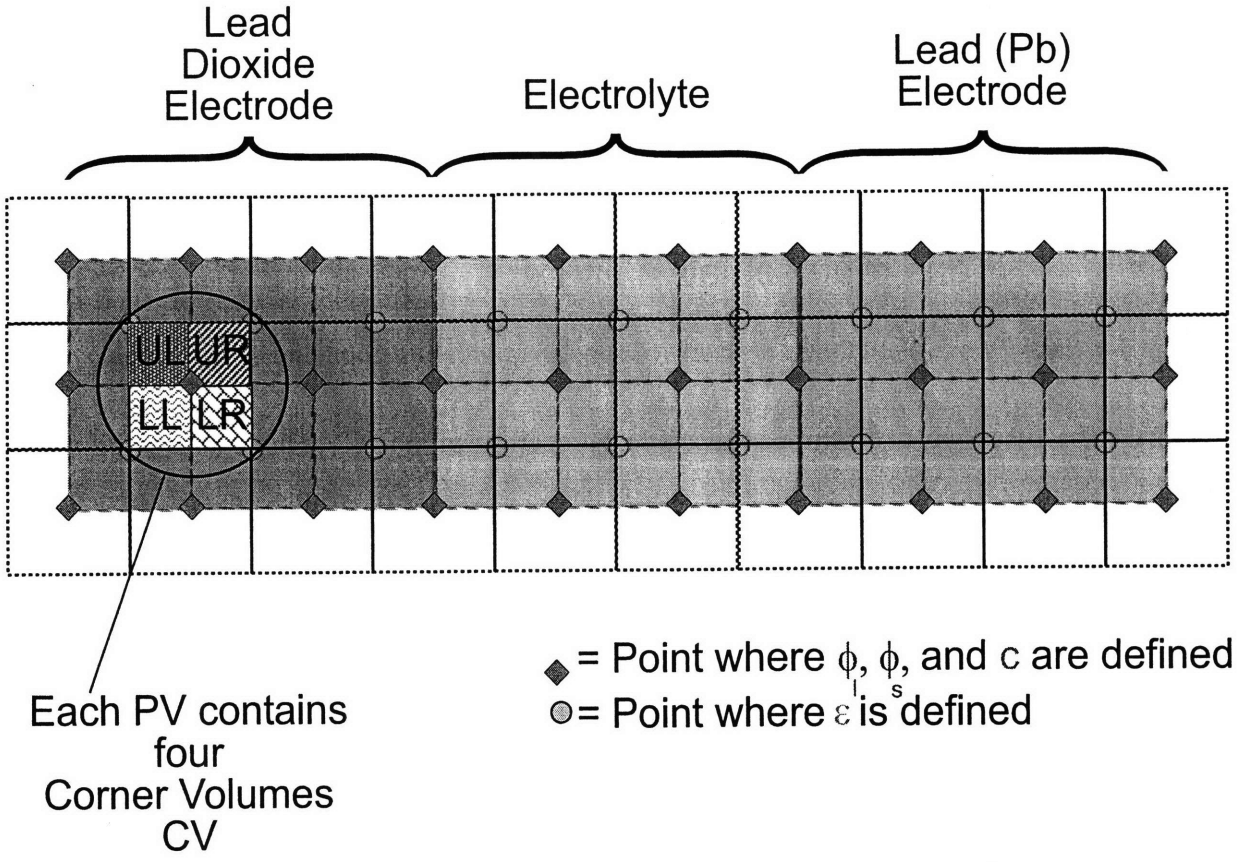


Figure 5-8. Each PV contains four CVs.

Now any CV and its neighbors to the norm of its surfaces will all be within the same FV and therefore flux densities between these CVs will be continuous as shown in fig. 5-9.

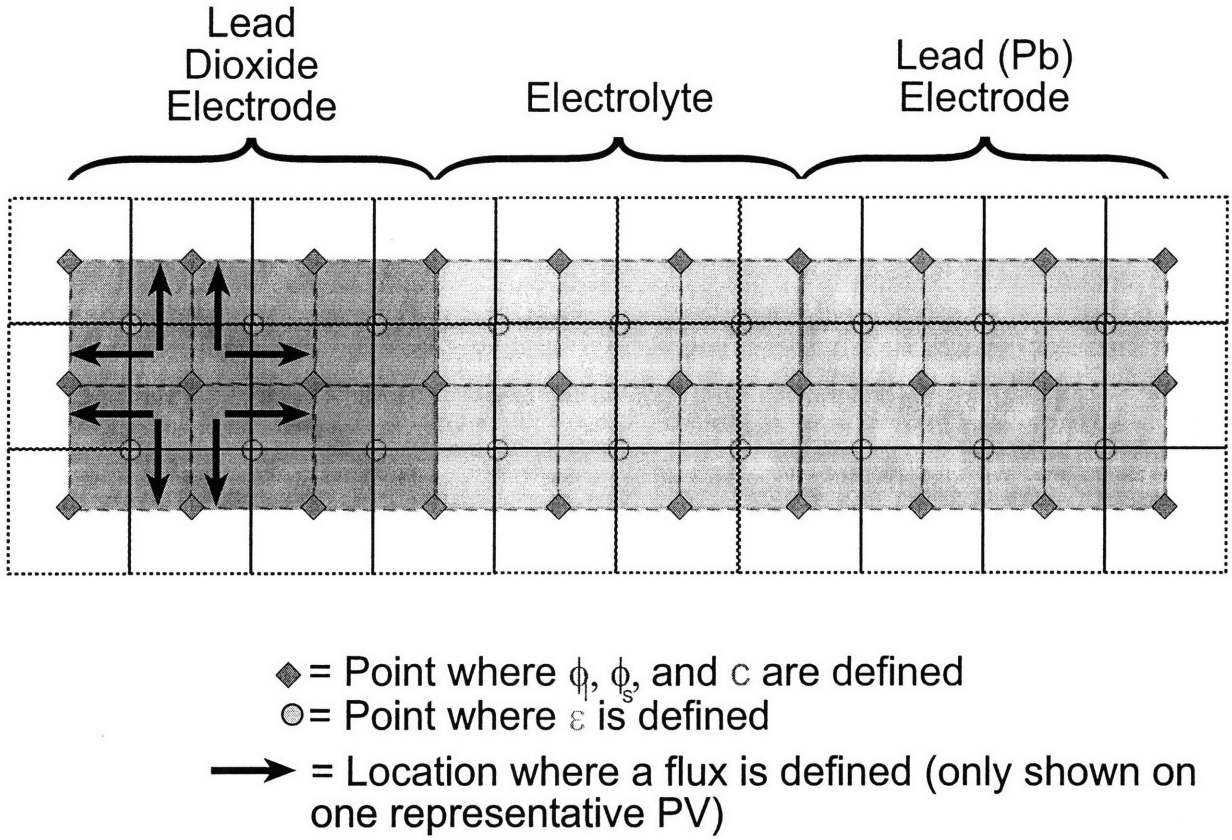


Figure 5-9. Each PV has eight fluxes defined at Gaussian quadrature points on its four faces.

The trick then becomes how to determine the potential properties within a CV. This is done by bilinear interpolation. Bilinear interpolation takes a weighted average of the potential properties of the four PVs nearest the CV in question, as illustrated in fig. 5-10. The potential value of a CV will be found for the Gaussian quadrature point within the CV.

Continuing, like the end PVs of the one dimensional model, the corner and edge PVs of the two dimensional model will have some CVs, that don't exist. Therefore, like the SVs, the CVs also contain the property 'exist'. For example, the PV in the upper left corner contains 4 CVs, but UL, UR, and LL have their 'exist' property set to FALSE as seen in fig. 5-11.

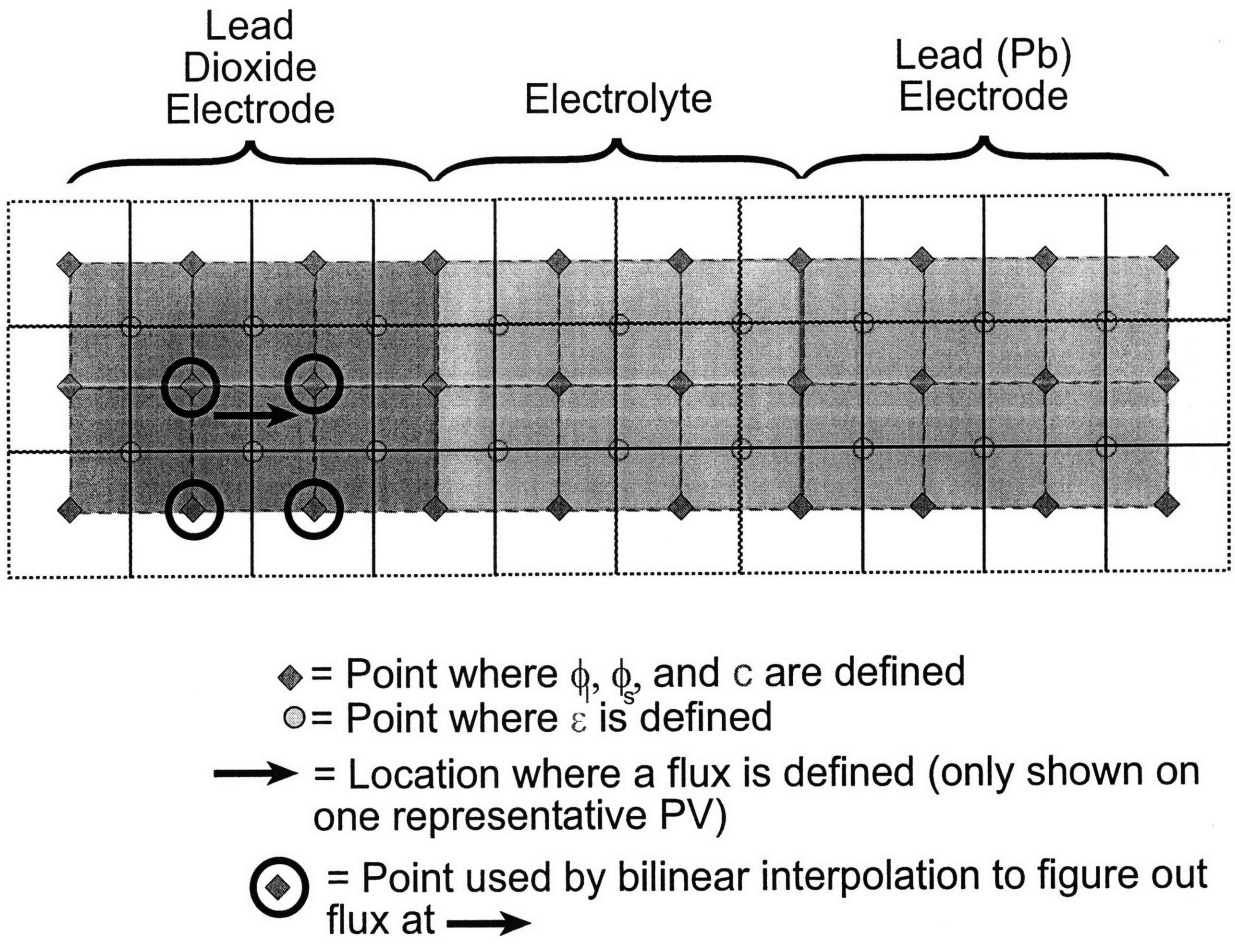


Figure 5-10. The eight fluxes on the four faces or the PV volumes are computed using bilinear interpolation.

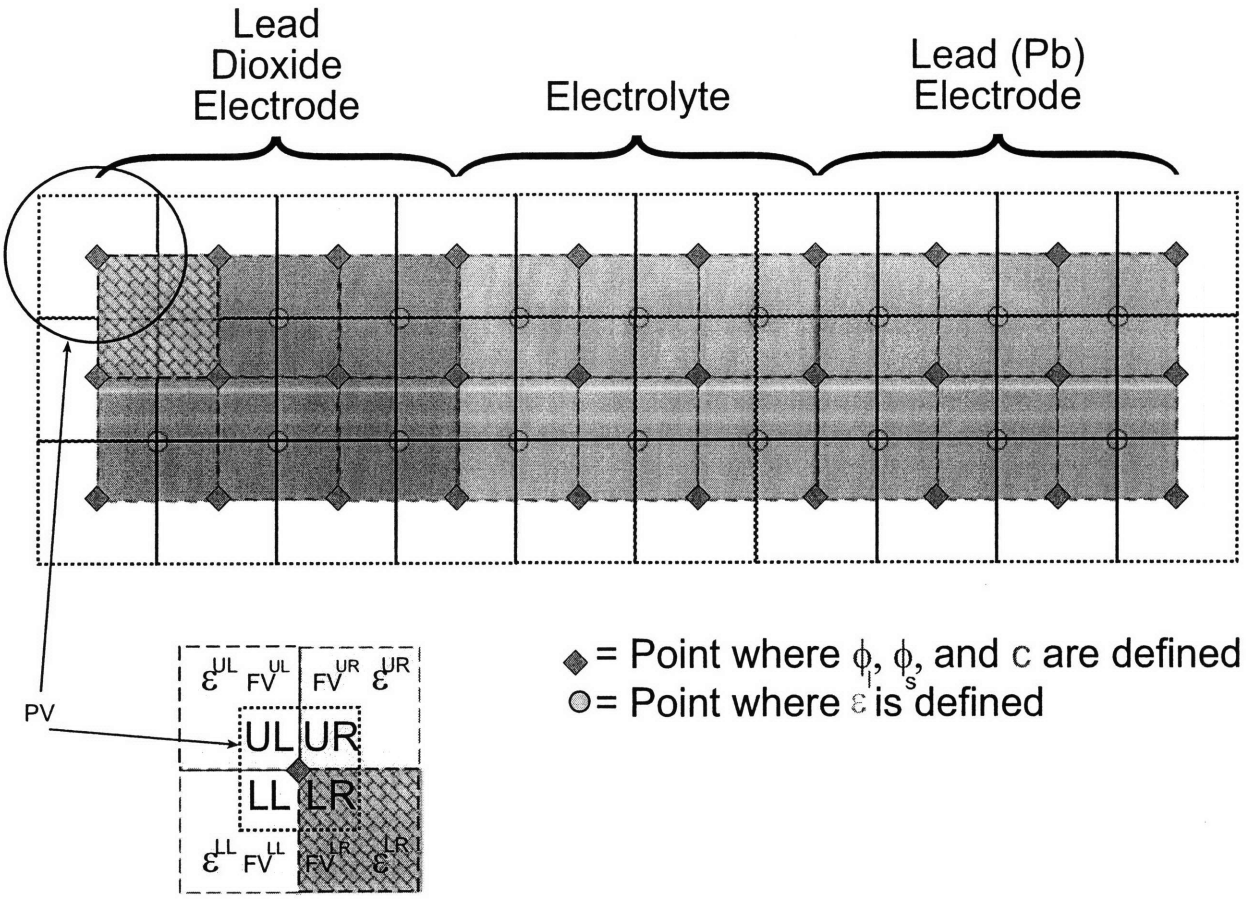


Figure 5-11. Each of the PVs are made up of 4 CVs; however, at the edges of the battery cell, some of those CVs might not exist.

5.4.2 Special Considerations for Porosity

The staggered grid solves the discontinuity of fluxes at physical boundaries problem, but in this case it creates the possibility of non-physical behavior of ε as it time steps. This non-physical behavior, however, is not a result of taking too large of a time step, but comes from the fact that both ε and the fluxes needed to figure out the divergence of current from an FV volume are defined at the center of the FV volume as seen in fig. 5-12. To see how non-physical behavior might arise from having ε and the fluxes defined at the same point, let us follow a line of reasoning similar to that in [38, p.137].

Fig. 5-12 shows our staggered FV volumes and PV volumes. At the edge of a few of the PV volumes are labeled fluxes. These fluxes' magnitudes are labeled in a 50, 100, 50, 100, 50 alternating pattern in the x-direction.

Now, our model for the change in porosity, eq. (5.19), requires computing $\nabla \cdot i_l$ for the FV volume of interest, so let us try to compute $\frac{\partial i_l}{\partial x}$ from fig. 5-12 and ignore the y-direction for the time being.

$$\begin{aligned}\frac{\partial i_l}{\partial x} &= \frac{i_l(\beta) - i_l(\alpha)}{\delta x} = \frac{\left(\frac{i_l(c)+i_l(b)}{2}\right) - \left(\frac{i_l(b)+i_l(a)}{2}\right)}{\delta x} \\ &= \frac{i_l(c) - i_l(a)}{\delta x}\end{aligned} \tag{5.29}$$

However, $i_l(c) = i_l(a)$, so we would think that $\frac{\partial i_l}{\partial x} = 0$ when in fact it is not. This kind of situation can lead to more severe numerical stabilities and needs to be avoided.

Therefore, three different models for porosity, therefore, were explored in for this thesis. The first model simply computed the divergence of the current around each FV volume as described above. This resulted in terrible numerical instabilities in the ε solution. However, this did not translate into major

numerical instabilities in the other variables as the size of the perturbations were small over the simulation time of interest. Next, a constant ε model was tried. This resulted in believable stable numerical output of C, ϕ_s, ϕ_l. Finally, $\nabla \cdot i_l$ for an was approximated by taking one fourth of $\nabla \cdot i_l$ for each of the PV volumes that intersect the FV volume and summing them. This produced numerically stable and physically believable ε behavior, but it did not result in a noteworthy difference in C, ϕ_s, ϕ_l from the C, ϕ_s, ϕ_l produced by the constant ε model. The simulations presented here all employ the last model of $\nabla \cdot i_l$.

It is interesting that the only other electrochemical battery model known by this author to employ a staggered grid, [34], also does not include the change of porosity equation among its main set of equations that it solves implicitly. After talking about the equations used to solve for C, ϕ_s, ϕ_l, pressure and fluid velocity in the x and y directions, [34, p.2056] merely suggests that the model, "can be used to determine the electrode porosity evolution." Then it goes on to exclude the equation that would be used to determine the evolution of porosity from its list of model equations and their numerical implementation, Table 1 of [34]. It, therefore, uses a constant porosity approximation. This issue of what to do with the porosity equation should be addressed in future work.

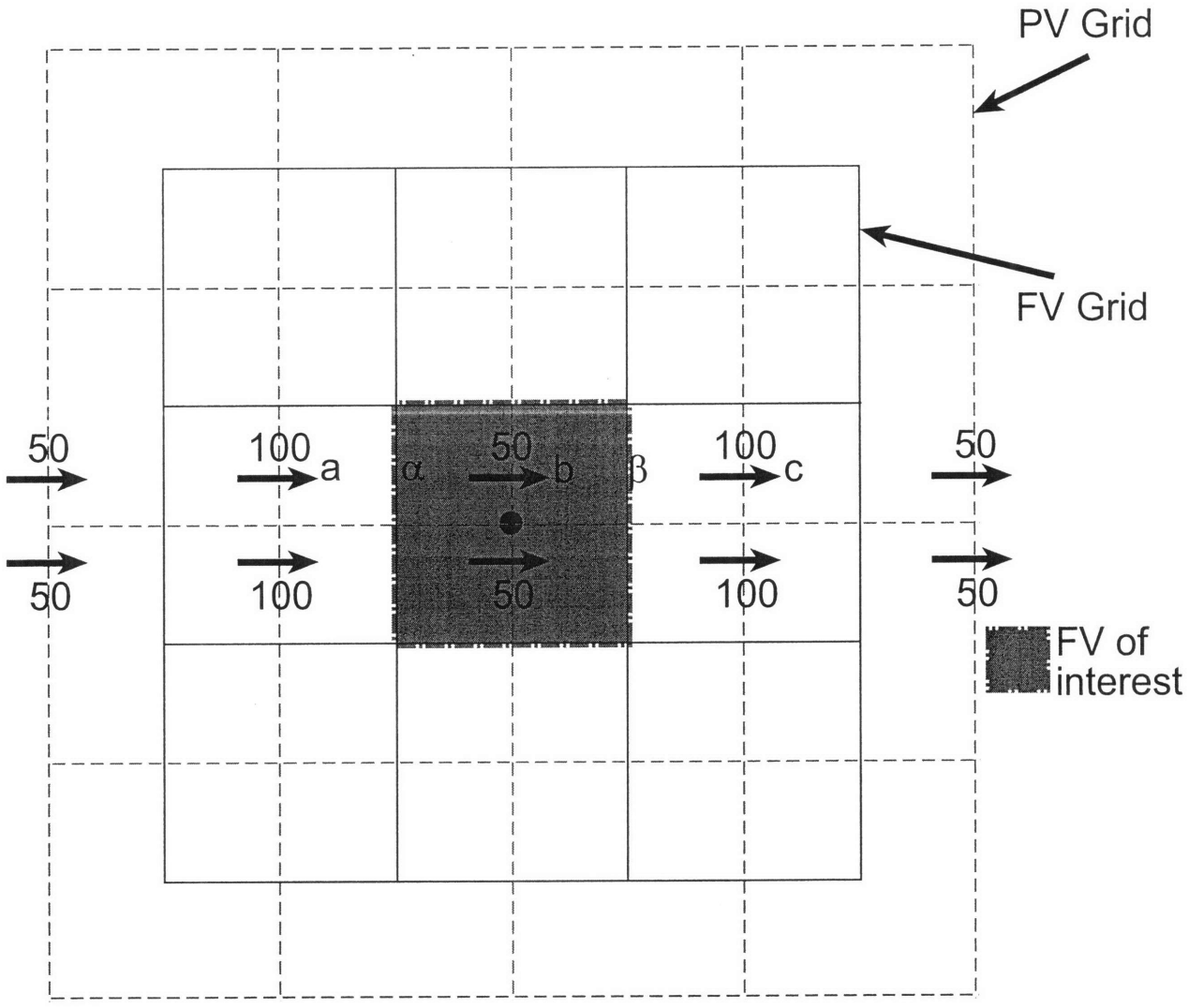


Figure 5-12. The fluxes needed to compute the divergence of the current from an FV volume are defined at the center of FV volumes.

5.5 Numerics

The non-linear coupled partial differential equations section 5.3.1 were spatially discretized on the grid described in section 5.4 using the finite volume method. The code that performs this is in chapter B. Backward Euler was used to approximate the time derivative for eq. (5.20). The set of non-linear equations, eqs. (5.20)–(5.22), was solved using Newton-Raphson [39, p.373].

When used, the porosity equation, eq. (5.19), was incremented separately from the other equations. At the beginning of each time step, $\varepsilon(t+1)$ was computed explicitly using $\varepsilon(t)$, $C(t)$ and $\phi_l(t)$. $\varepsilon(t+1)$ was then substituted into eqs. (5.20)–(5.22). This was done because ε is not defined on the PV grid but on the FV grid. Therefore, it does not fit into the 9-point stencil used on the PV grid.

One has to be careful, however, when using Forward Euler as using too large of a time step can lead to numerical instabilities under certain conditions [40]. The reason for the instabilities is different from that just presented in section 5.4.2 where the physical layout of the volumes could lead to problems. I this case it is the time stepping that leads to problems. To see this let us look at a summary of the example in [40]. In the situation

$$\frac{\partial \varepsilon}{\partial t} = -\alpha\varepsilon,\ \alpha > 0 \tag{5.30}$$

when discretized as

$$\frac{\partial \varepsilon}{\partial t} = \frac{\varepsilon^{t+1} - \varepsilon^{t}}{\Delta t} = -\alpha\varepsilon^{t} \tag{5.31}$$

to get

$$\varepsilon^{t+1} = \varepsilon^{t} - \Delta t\alpha\varepsilon = (1 - \Delta t\alpha)\,\varepsilon^{t} \tag{5.32}$$

we see that

$$\varepsilon^{t+1} = (1 - \Delta t\alpha)^{N+1} \varepsilon^{t-N} \tag{5.33}$$

Therefore, in order for eq. (5.33) to be numerically stable, $|(1 - \Delta t\alpha)| < 1$ must be true, so $\Delta t < \frac{2}{\alpha}$.

Therefore, the system was tested over for orders of magnitude of dt from 0.01s to 100s. No oscillations in ε, C, ϕ_s, and ϕ_l were found. This is most likely due to the fact that oscillations occur when one has a system of $\frac{d\varepsilon}{dt} = -\alpha\varepsilon$ where as our system is of the form $\frac{d\varepsilon}{dt} = f(t, \varepsilon(t))$ where the right hand side of the last equation is a function of time and only a weak function of ε effectively making the system $\frac{d\varepsilon}{dt} = f(t)$ where $f(t)$ is independent of $\varepsilon(t)$. In such a system, no oscillatory behavior should be observed due to time step size.

5.5.1 Newton-Raphson

Newton-Raphson is an iterative procedure for finding the solution vector S of a system of coupled non-linear equations $G(\mathbf{S}) = 0$. It is used in this thesis to compute solutions to C, ϕ_s, and ϕ_l in every PV volume. It starts with an initial approximation to the solution vector S which we shall call $\mathbf{s}^0$, and it tries to refine that approximation to come up with a better approximation to the solution vector. This new approximation to S is called $\mathbf{s}^1$. It then takes the guess $\mathbf{s}^1$ and uses it to try to get a better estimate of S called $\mathbf{s}^2$. The process continues until $\mathbf{s}^k$ is sufficiently close to S. In general, the relationship between the k^{th} estimate and the $k+1$ estimate can be found by expanding $G(\mathbf{S}) = 0$ via the Taylor series and keeping the linear terms as seen below:

$$G\left(\mathbf{s}^{k+1}\right) \approx G\left(\mathbf{s}^k\right) + \mathbf{J}\left(\mathbf{s}^k\right)\left(\mathbf{s}^{k+1} - \mathbf{s}^k\right) \tag{5.34}$$

Setting $G\left(\mathbf{s}^{k+1}\right) = 0$ allows us to rewrite eq. (5.34) as

$$\mathbf{s}^{k+1} = \mathbf{s}^{k} - \mathbf{J}^{-1}\left(\mathbf{s}^{k}\right) G\left(\mathbf{s}^{k}\right) \tag{5.35}$$

Now the right hand term of eq. (5.35) is the inverse of a matrix times a vector. This results in a vector, i.e. $\mathbf{x} = \mathbf{J}^{-1}\left(\mathbf{s}^{k}\right) G\left(\mathbf{s}^{k}\right)$. Because computing an inverse is computationally extremely expensive, one would never want to compute it. Therefore, instead of finding $\mathbf{x}$ via the inverse, $\mathbf{x}$ is the solution to the system of equations in eq. (5.36),

$$\mathbf{J}\left(\mathbf{s}^{k}\right)\mathbf{x} = G\left(\mathbf{s}^{k}\right) \tag{5.36}$$

and is found using a method other than inverting the Jacobian matrix. In this thesis, $\mathbf{x}$ will be found using banded Gaussian elimination which will be explained in section 6.3.1. Eq. 5.37 explicitly shows where solving the system of linear equations is involved in the Newton step.

$$\mathbf{s}^{k+1} = \mathbf{s}^{k} - \underbrace{\mathbf{J}^{-1}\left(\mathbf{s}^{k}\right) G\left(\mathbf{s}^{k}\right)}_{\mathbf{J}\left(\mathbf{s}^{k}\right)\mathbf{x}=G\left(\mathbf{s}^{k}\right)} \tag{5.37}$$

From here on out, eq. (5.36) will be written as

$$\mathbf{J}\mathbf{x} = \mathbf{r} \tag{5.38}$$

Where $\mathbf{J} = \mathbf{J}\left(\mathbf{s}^{k}\right)$ and $\mathbf{r} = G\left(\mathbf{s}^{k}\right)$. It should be clear now that Newton-Raphson involves two steps. The first step involves determining the residual vector $\mathbf{r}$ and the Jacobian $\mathbf{J}$. The second step involves solution of eq. (5.38).

Newton-Raphson, forms the basis for our time stepping algorithm which can be found in Algorithm 1. Each iteration through 1.4 is a Newton step. Each iteration through 1.4 in which the time counter t is incremented at 1.15 is called a time step. Each time step, therefore, consists of one or more Newton steps.

Input: A system of non-linear equations $G(\mathbf{S}(t=0))=0$ at time $t=0$
Result: A time sequence of system states $\mathbf{S}(t), \forall t < t_{max}$
// set time step to zero
1.1 $t \longleftarrow 0$;
// set Newton step to zero
1.2 $k \longleftarrow 0$;
// Initial guess of $\mathbf{S}(1)$
1.3 $\mathbf{s}^0 \longleftarrow \mathbf{S}(t=0)$;
1.4 **while** $t < t_{max}$ **do**
// Update Porosity
1.5 $\phi(t+1) \longleftarrow$ update_porosity $(C(t), \phi_l(t), \phi(t))$;
// Set up Residual vector $\mathbf{r}$
1.6 $\mathbf{r}^k \longleftarrow$ compute_residual_vector$(\mathbf{S}(t), \mathbf{s}^k, \mathbf{r})$;
// Set up Jacobian matrix $\mathbf{J}$
1.7 $\mathbf{J}^k \longleftarrow$ setup_Jacobian$(\mathbf{S}(t), \mathbf{s}^k, \mathbf{J})$;
1.8 $\mathbf{J}^k \longleftarrow$ apply_BoundaryConditions$(\mathbf{J}^k)$;
1.9 $\mathbf{J}^k \longleftarrow$ apply_preconditioner$(\mathbf{J}^k)$;
// compute $\mathbf{x}$ vector
1.10 $\mathbf{x}^k \longleftarrow$ Gaussian_Solver $\left(\mathbf{J}^k, \mathbf{r}^k\right)$;
// Compute updated estimate of $\mathbf{S}(t+1)$
1.11 $\mathbf{s}^{k+1} \longleftarrow \mathbf{s}^k - \mathbf{x}^k$;
1.12 **if** $\|\mathbf{r}\|_\infty < \delta_\mathbf{r}$ ***and*** $\|\mathbf{x}^k\|_\infty < \delta_\mathbf{x}$ **then**
// save $\mathbf{S}$
1.13 $\mathbf{S}(t+1) \longleftarrow \mathbf{s}^{k+1}$;
// Reset the Newton step counter to zero
1.14 $k \longleftarrow 0$;
// Increment the time index
1.15 $t \longleftarrow t+1$;
// Initial guess of next $\mathbf{S}(t+1)$
1.16 $\mathbf{s}^0 \longleftarrow \mathbf{S}(t)$;
1.17 **end**
// Increment the Newton step counter k
1.18 $k \longleftarrow k+1$;
1.19 **end**

Algorithm 1: The Newton-Raphson procedure for the system.

5.5.2 The Jacobian Matrix J

The Jacobian matrix J has three notable features that will be discussed below.

Banded Jacobian Matrix

First, it is banded as seen in fig. 5-13. Fig. 5-13 is a Matlab spy plot of an example Jacobian produced by our system. The non-zero elements of the matrix are shown by the colored points. White space denotes elements with zero values. The solvers in section 6.3 take advantage of this special structure to try to speed up the calculations by only performing elimination between the two outer bands.

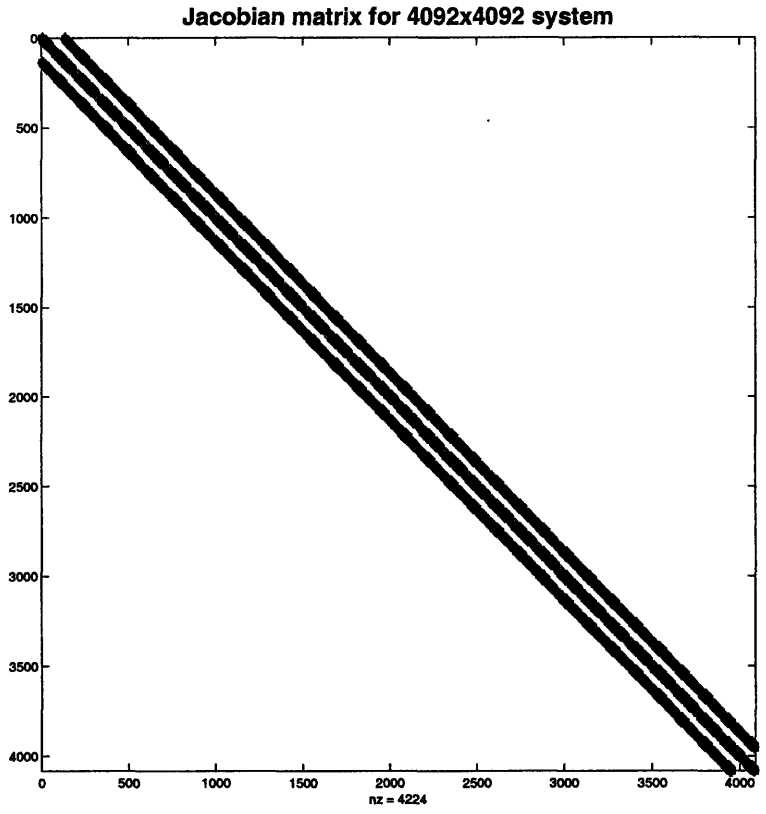


Figure 5-13. Notice the banded structure of the matrix. The size of the system was 4092×4092.

Asymmetric Jacobian Matrix

Next, the matrix is not symmetric. This is due to the fact that a nine point stencil is being used and each volume has a number of different variables in it. A nine point stencil consists of a center volume its eight nearest neighbors: the upper left neighbor, the neighbor directly above the volume, the upper right neighbor, the neighbor to the left of the volume, the neighbor to the right of the volume, the neighbor to the lower left of the volume, the neighbor directly below the volume and the neighbor to the lower right of the volume.

Since each volume has multiple variables associated with it, the distance (in number of variables) from the first variable of the center volume to the last variable of the lower right neighbor is greater than the distance from the first variable of the center volume to the first variable of the upper left neighbor. The pivots of the Jacobian matrix are always one of the variables of the center volume under consideration. Thus, the matrix cannot be symmetric.

Jacobian Matrix Condition Number

Finally, the Jacobian matrix of a particular Newton step at 1.8 is poorly conditioned, with condition number on the order of 1×10^{12}. This number 1×10^{12} means that the largest eigenvalue of the Jacobian is 12 orders of magnitude larger than the smallest eigenvalue. This is important because the condition number gives us a rough estimate of the number of digits of precision that will be lost when solving systems like eq. (5.38). For example, when using double precision, one has about 16 decimal places of precision to work with. If a system has a condition number of 1×10^{12}, then at most only the first 4 digits of the solution could be considered accurate.

There are many different methods for improving the condition number of a matrix before performing

operations with the matrix. Operating on a matrix with the goal of improving the condition number of a matrix before using the matrix is known as preconditioning the matrix. The operations that are performed on the matrix that improve the condition number of that matrix are known as a preconditioner. The preconditioner we focus on is row scaling. We also discuss how different physical properties affect the condition number. This information could be used to develop an improved preconditioner.

Jacobian Matrix Row and Column Scaling

Row scaling, multiplying an entire row of a matrix and its corresponding right hand side by a scale factor, was used to improve the condition number of the Jacobian matrix. In particular, eq. (5.20) was scaled up by a factor of 30,000. Eq. 5.22 was scaled down by a factor of 50, and eq. (5.21) was multiplied by a factor of 10. These row scalings improved the condition number of the Jacobian by about 3 orders of magnitude from around 1×10^{12} to around 5×10^{9}, giving our solutions about 6 to 7 digits of precision instead of the 3 to 4 digits of the non-preconditioned Jacobian matrix. Fig. 5-14 shows the condition numbers of the preconditioned Jacobian matrices from the first 1004 Newton steps. This was for a discharge rate I=1.0, a time step size of 1 second, and a model that employed a 22x31 PV volume grid.

Column scaling, or multiplying each entry in a matrix column by a factor, is the equivalent of a units change (e.g. milligrams to kilograms) for the corresponding variable. Some attempts were made to use column scaling to improve condition number, but no noteworthy results were achieved.

Column pivoting, or re-arranging the order of the variables in the solution vector, was also attempted. The idea was to try to bring larger values onto the diagonal in an attempt to make the matrix as close to diagonally dominant as possible. It resulted in a non-preconditioned Jacobian condition number on the order of 1×10^{11}; however, once our simple preconditioner was applied, the condition number was

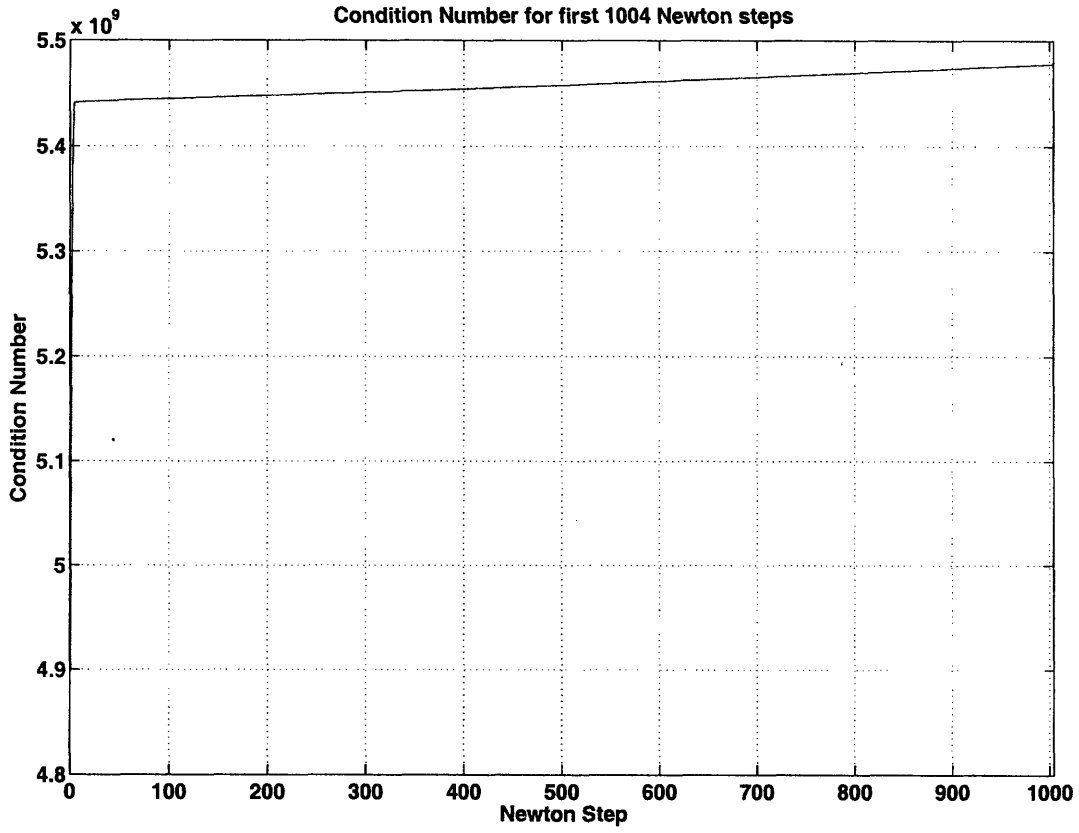


Figure 5-14. Condition number for first 1004 preconditioned Jacobian matrices. The size of the system was 4092×4092. All Jacobian matrices were preconditioned using the same set of row scaling coefficients.

again on the order of 5×10^9.

Even 5×10^9 is still an uncomfortably large condition number. It is possible, and perhaps even probably, that more time spent studying the application of preconditioning to these equations could yield a more robust solution. If this model is applied to other problems and cases, such work may be necessary or desirable in the future.

Matrix Size N	1057	2016	3936
Condition Number	1.1×10^{12}	1.6×10^{12}	2.2×10^{12}
Band Size	212		
lead dioxide conductivity σ	80		
lead (Pb) conductivity σ	48000		

Table 5.1. Matrix Size with band size of 212. **Time step size of 1 second and discharge parameter I of** 0.5.

Band Size	212	284	318	354
Condition Number	2.22×10^{12}	3.97×10^{12}	4.82×10^{12}	5.71×10^{12}
Matrix Size	≈ 4025			
lead dioxide conductivity σ	80			
lead (Pb) conductivity σ	48000			

Table 5.2. Band Size with matrix size of approximately 4025. Time step size of 1 second and discharge parameter I of 0.5.

Effects of Physical Properties on Condition Number

Condition number of the Jacobian matrix J changes very little with matrix size as evidenced by the data in Table 5.1. Changing the band size also changed the condition number very little, Table 5.2.

The condition number of the Jacobian matrix J decreases rapidly with decreasing lead (Pb) conductivity until the lead (Pb) conductivity is less than that of lead dioxide. Then decreasing lead dioxide conductivity becomes more important for decreasing the condition number of the Jacobian matrix J.

Other values, such as the diffusion coefficient, also can change the condition number of the Jacobian matrix as seen in Table 5.4; however, decreasing lead (Pb) conductivity seems to have the most dramatic effect.

	lead dioxide conductivity σ			
lead (Pb) conductivity σ	800	80	8	1
48000	3.95×10^{12}	3.97×10^{12}	3.98×10^{12}	
4800		3.96×10^{11}		
480		3.97×10^{10}		
4.8		3.75×10^{9}	6.67×10^{8}	
1		3.02×10^{9}		$\mathbf{1.62 \times 10^{8}}$

Table 5.3. Matrix Size of 4092×4092 with band size of 284. Time step size of 1 second and discharge parameter I of 0.5. Entries in the grid are condition numbers.

Diffusion Coefficient	3.0×10^{5}	0.3	300
Condition Number	3.97×10^{12}	9.02×10^{10}	5.7×10^{10}

Table 5.4. Matrix Size of 4092×4092 with band size of 284. Time step size of 1 second and discharge parameter I of 0.5. Lead dioxide conductivity of 80 and lead (Pb) conductivity of 48000.

5.6 Simulations using 2-D Model

5.6.1 2-D Model Verification

The 2-D model was verified by showing that it behaves like the results of the interferometry experiments and examples of battery behavior in the literature. Because of the limitations of the model, we will only concern ourselves with battery like behavior and not exact potentials and concentrations.

2-D Model Potential Curves

We start by checking that the potential seen across the battery terminals is as we would expect it. Fig. 5-15 shows that the battery cell potential as seen at the terminals of the battery, under discharge, starts below the open circuit potential of the battery cell and continues to decrease throughout the time of discharge. This behavior is typical of a discharging battery.

Fig. 5-16 shows the terminal potential of the same battery cell discharged at 4 different rates. It has the feature that the faster the discharge rate, the smaller the terminal potential. That is to say, the faster the discharge rate, the further the battery cell has to be perturbed from its equilibrium potential in order to achieve that discharge rate. This effect was also seen in the two-volume model developed in chapter 4 and shown in fig. 4-25. Also, the faster the discharge rate, the more rapidly the battery cell potential decreases while discharging. This can be seen in fig. 5-16 where the battery cell potential decrease with time is only visible for a discharge rate of $I = 2.0$. The potential decreases with time at the slower discharge rates too, but when plotted on the scale of fig. 5-16, the effect is not apparent.

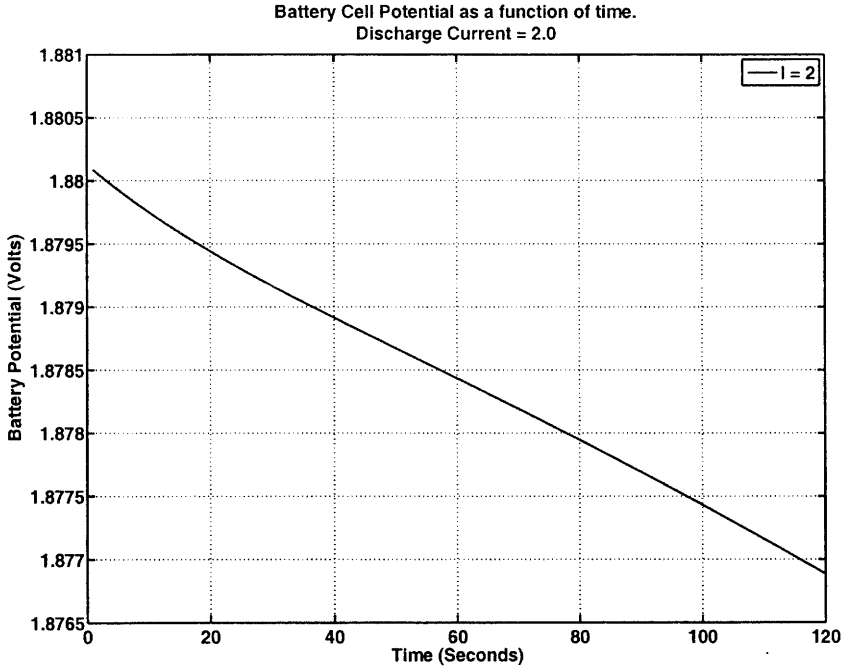


Figure 5-15. The terminal potential under discharge is below the equilibrium potential of the battery and continues to decrease throughout the period of discharge.

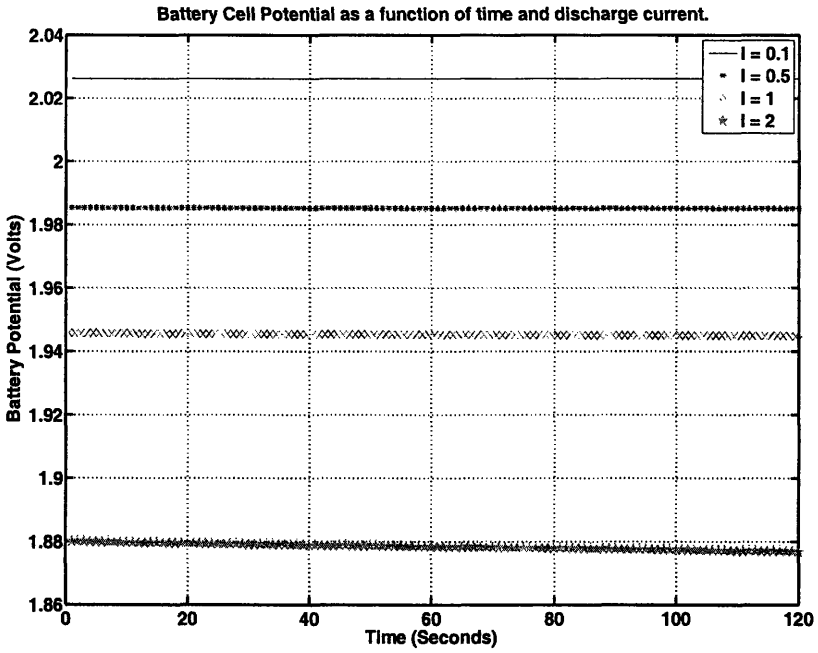


Figure 5-16. The starting potential of the discharge curve is smaller for higher discharge rates. This means that the electrodes are being perturbed more from equilibrium when discharging at a higher rate.

2-D Model Concentration Profile Curves

Next, we look at the concentration profile during discharge and compare it to the results of the interferogram. The interferogram, fig. 2-6, had two striking features. First, it showed more reaction at the top of the observation region than at the bottom of the observation region. Then it showed that the change in concentration during discharge at the lead dioxide electrode was greater than that at the lead (Pb) electrode. The concentration profile shown here in fig. 5-17 exhibits both of these same behaviors.

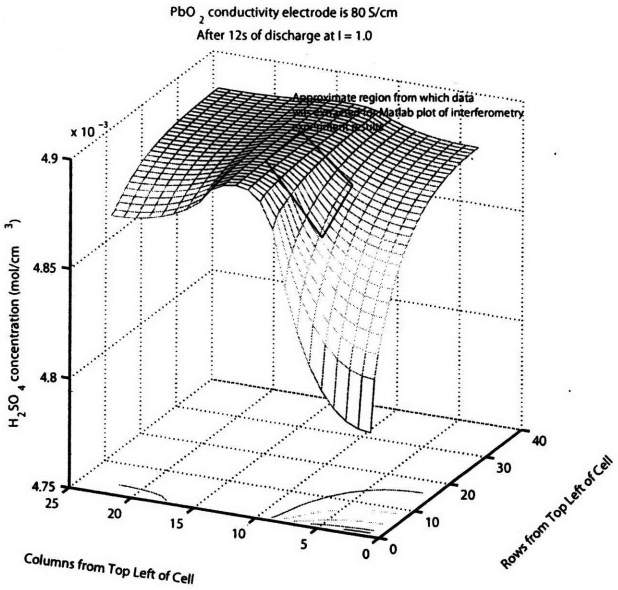


Figure 5-17. Like the experiments, the 2-D model exhibits a greater decrease in the concentration of electrolyte at the top of the electrodes than at the bottom of the electrodes and mostly constant concentration in the bulk electrolyte. Fig. 5-18 shows the region being observed in the simulation.

In fig. 5-17, the axis labeled "Rows from Top Left of Cell" corresponds to the y direction of the cell. The bottom of the cell would be at position 31 on this axis. The axis labeled "Columns from Top Left of Cell" corresponds to the x direction of the cell. The value zero on this axis corresponds to the left-most part of the lead dioxide electrode. The value 22 on this axis corresponds to the right most

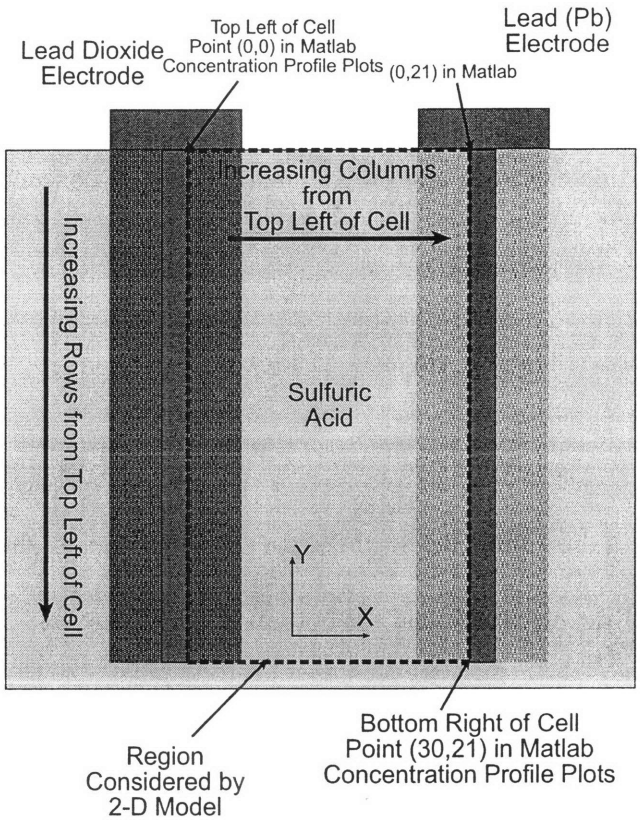


Figure 5-18. Illustration of a lead-acid battery cell. All concentration profile plots

portion of the lead electrode. Each electrode is 8 volumes wide and the electrolyte is 6 volumes wide. Therefore, the lead dioxide electrode occupies the entires region encompassed by positions 0 through 7 on the "Columns" axis and 0 through 30 on the "Rows" axis. The electrolyte takes up 8 through 13 on the "Columns" axis and 0 through 30 on the "Rows" axis. Finally, the lead electrode takes up 14 through 21 on the "Columns" axis and 0 through 30 on the "Rows" axis. The mapping of the data in fig. 5-17 is further explained by fig. 5-18. The z direction on this plot shows the concentration of the electrolyte at a given position within the cell.

For further verification, if one were to look at any cross section of concentration taken at a given row across the "columns" axis, one would see that it looks very similar to the plots in fig. 4-17. Plot fig. 4-17 comes from [30] which was a one dimensional model. Both plots show that the reduction in

concentration of electrolyte during discharge at the lead dioxide electrode is greater than that at the lead (Pb) electrode. This corresponds to what was observed on the interferogram with a notable change in concentration at the lead dioxide electrode and only a small change in concentration at the lead (Pb) electrode.

5.6.2 Two-Level Model Justification via 2-D Model

We now use the 2-D model to justify the two-level model of fig. 1-6. In order to justify the model, we first show that the resistance drop along the height of the electrodes creates the difference in usage of electrolyte between the top of the electrodes and the bottom of the electrodes. Then we will show that there exists a difference in potential difference between the electrodes along the height of the electrodes thus justifying the two different potentials in our two-level model.

Two-Level Model Resistor Justification

The 2-D model was used to simulate two simple experiments to justify the resistors of fig. 1-6. The first experiment was done by varying the conductivity of lead dioxide while holding the conductivity of lead (Pb) constant at 48000 S/cm. The results of this experiment can be seen in fig. 5-19. The second experiment done was the same as the first except now the conductivity of lead dioxide was held constant at 80 S/cm while the conductivity of lead (Pb) was varied. The results of decreasing the conductivity of lead (Pb) can be seen in fig. 5-20.

Fig. 5-19 clearly shows that the reducing the resistivity of lead dioxide by an order of magnitude almost eliminates the non-uniform usage of electrolyte along the height of BOTH electrodes. Furthermore, fig. 5-20 shows that the resistivity of lead (Pb) must *increase* by three orders of magnitude before it has a discernible effect on the concentration profile. Two conclusions can be drawn from these plots.

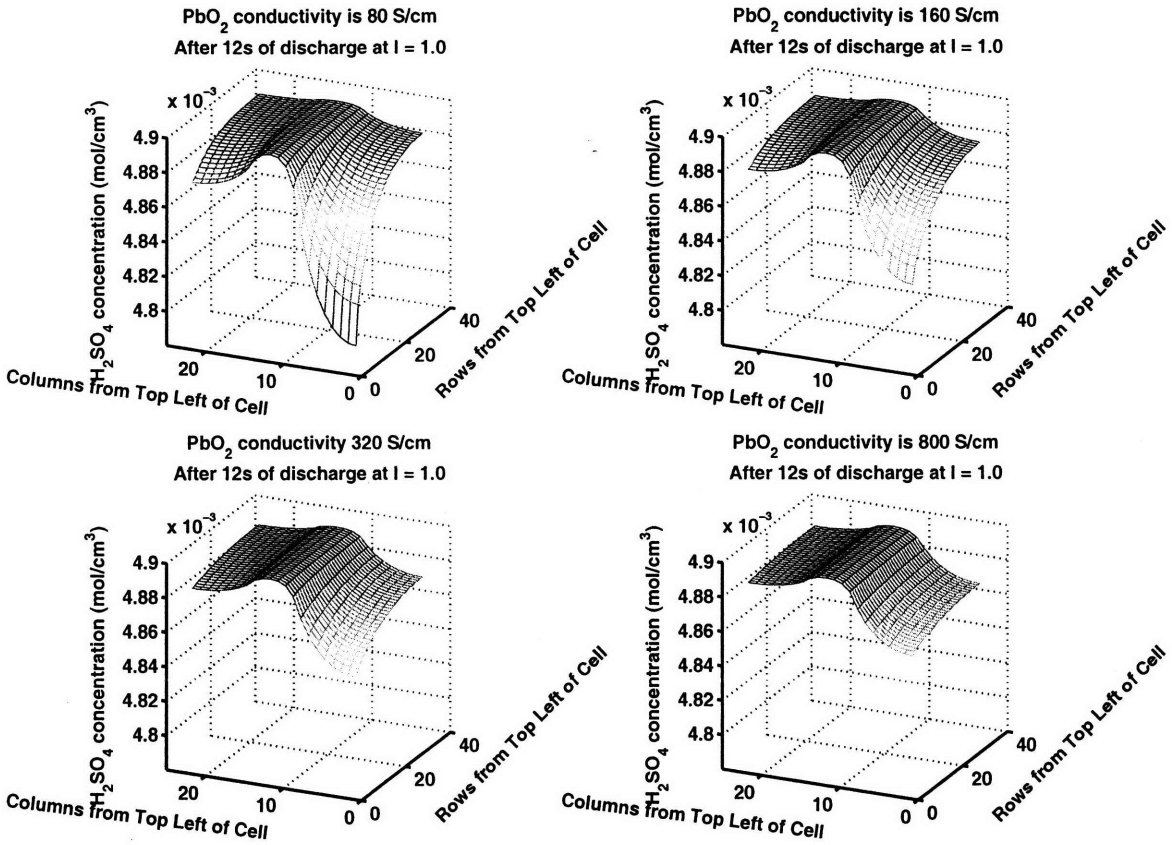


Figure 5-19. Results from four different simulations. Together they show that increasing the conductivity of the lead dioxide electrode eliminates the non-uniform usage of both electrodes.

First, it does appear that the non-uniform usage of the electrodes *is*, at least in-part, a side effect of the resistance drop down the height of the electrodes and therefore the resistor portion of the two-level model is well justified. Second, it appears that the high resistivity of lead dioxide electrode is the main factor in the creation of the non-uniform usage profiles. This could be one clue as to why, "The cycle life of lead/acid batteries is often limited by the positive plates [41]."

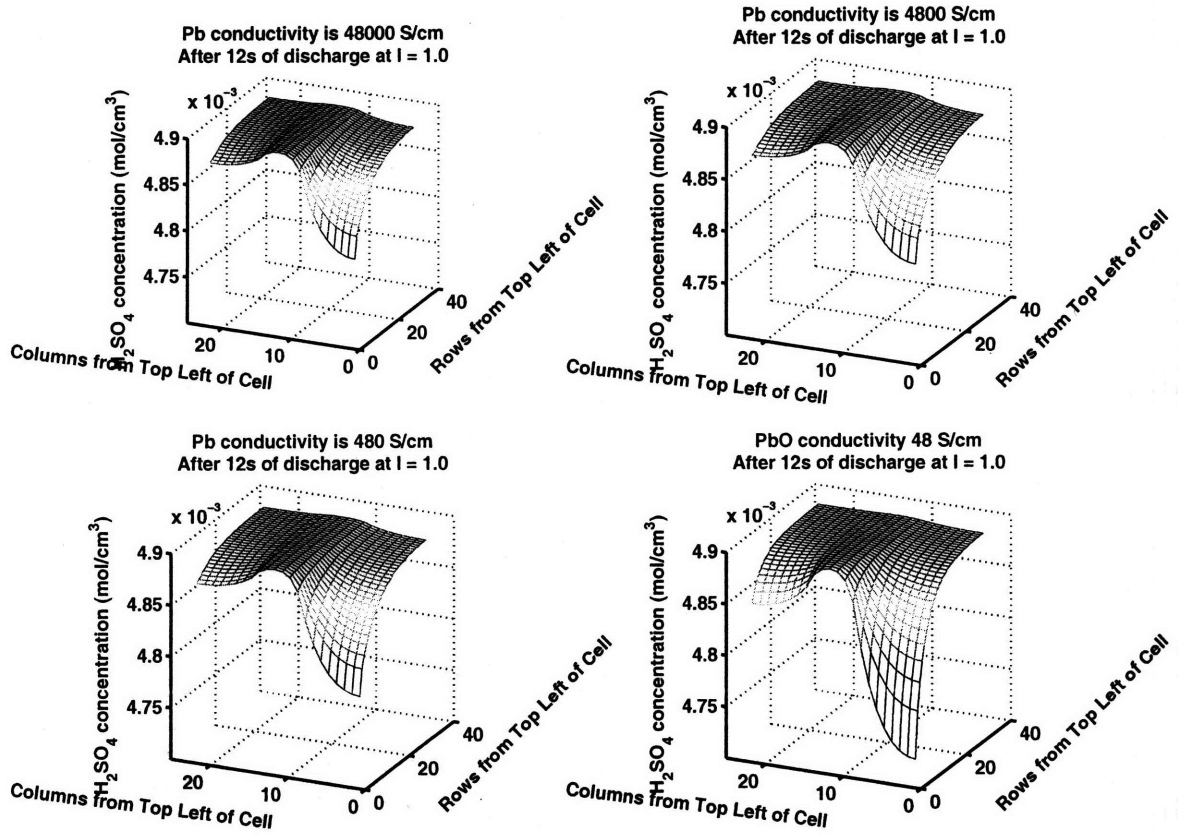


Figure 5-20. Results from four different simulations. Together they show that decreasing the conductivity of the lead (Pb) electrode by almost two orders of magnitude does not have an effect on electrode usage.

Two-Level Model Two Voltage Sources Justification

Next, we show that the potential difference between the electrodes at the top half of the battery cell is different than that between the electrodes at the bottom half of the battery cell thereby justifying the use of two different potentials within our two-level model.

Fig. 5-21 and fig. 5-22 show the potential between the electrodes for each row simulated in the experiments from section 5.6.2. The upper-left plot in fig. 5-21 shows a dramatic rise in potential between the electrodes from the top of the battery cell to the bottom of the battery cell. Certainly one cannot

consider the 1.95V difference between the electrodes found at the top of the battery cell (row 0) to be the same potential as the 2.02V found at the bottom of the battery cell (row 30). The sequence of images also shows that as lead dioxide's conductivity is artificially increased, thereby making electrode use more uniform along the height, the potential difference along the height of the electrode becomes more uniform. Fig. 5-22 shows similar results for lead (Pb) .

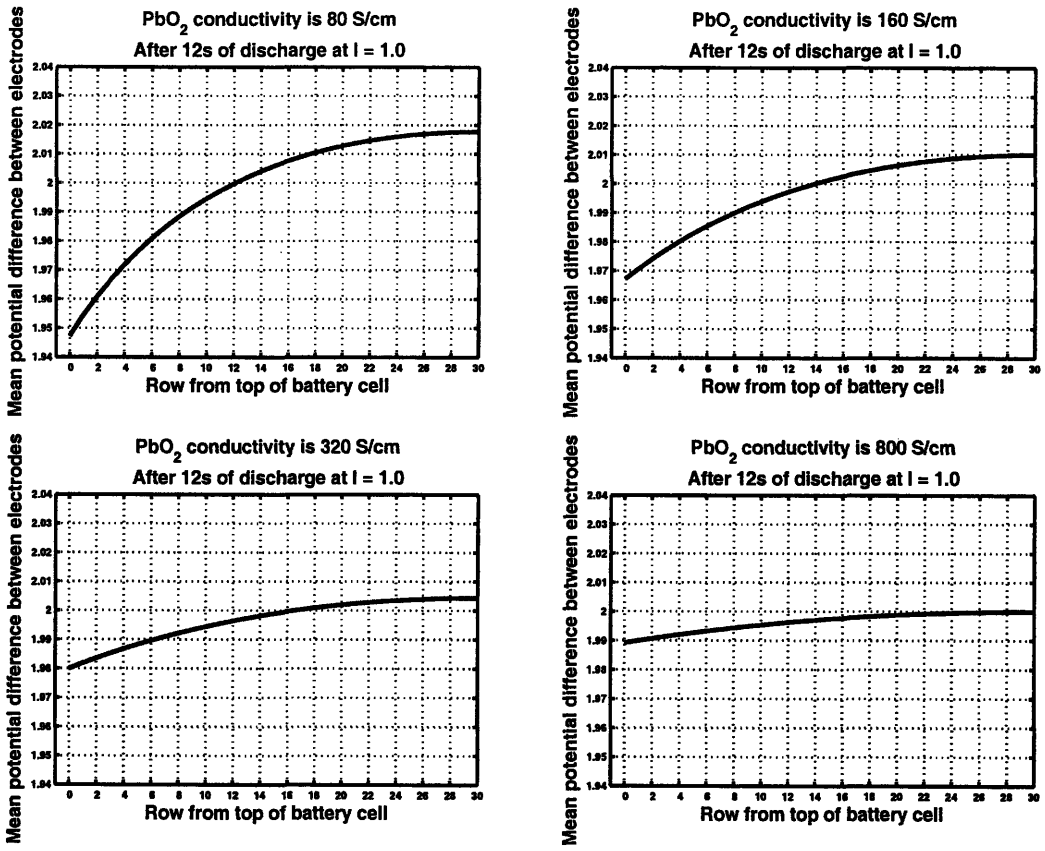


Figure 5-21. Results from four different simulations. Together they show that electrode usage isn't affected by decreasing the conductivity of the lead (Pb) electrode until the conductivity of the lead (Pb) electrode is on the order 4800 S/cm.

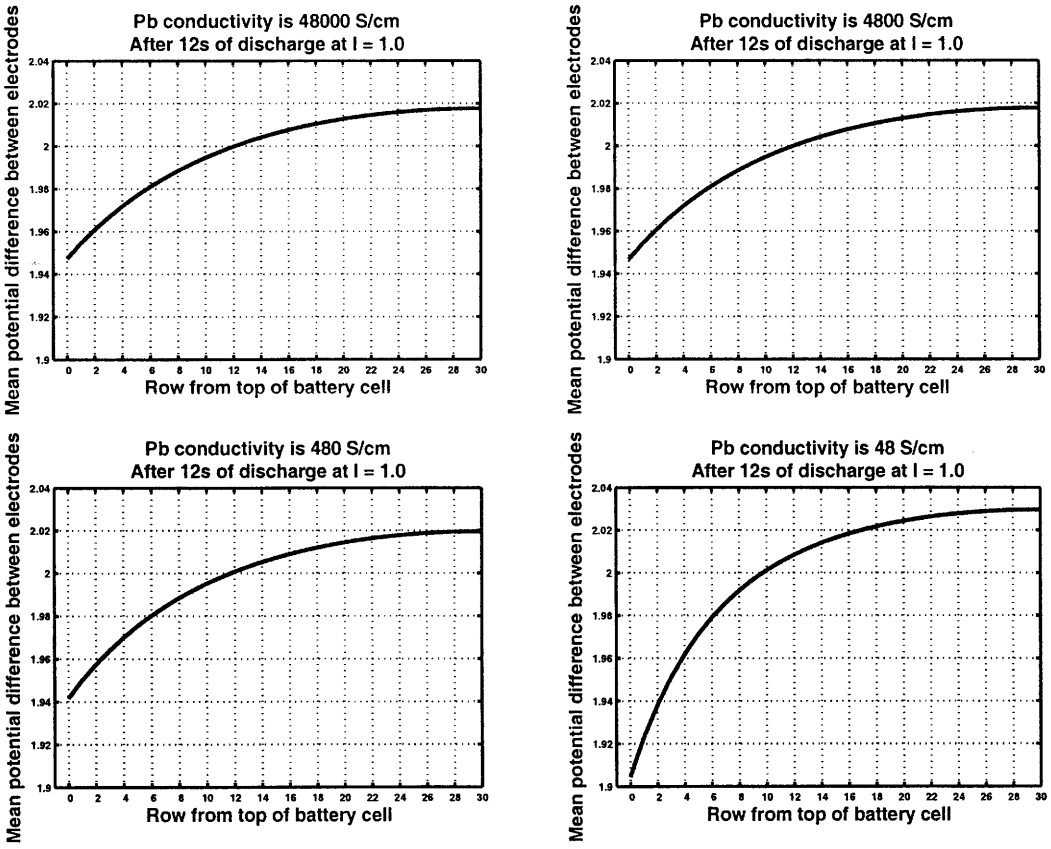


Figure 5-22. Results from four different simulations. Together they show that electrode usage isn't affected by decreasing the conductivity of the lead (Pb) electrode until the conductivity of the lead (Pb) electrode is on the order 4800 S/cm.

5.6.3 Reason Discharge Rate affects Battery Life

It was observed in [41] that, "Increasing the discharge current density decreases the battery life." It is theorized that if the discharge rate increases, more of the delivered charge will come from the top of the battery cell. Thereby, taking the top of the cell to a greater depth of discharge which means that it will fail sooner than the portion of the battery not taken to such a great depth of discharge. Fig. 5-23 shows concentration profiles of the 16×11 cell discharged at four different rates. The plots are battery cell concentration profiles after the same amount of charge has been removed from each cell. This shows that the faster the discharge rate, the larger fraction of charge taken from the top of the electrodes. Therefore, the faster one discharges a battery cell, the deeper the depth of discharge at the top of the electrodes and the sooner those parts of the electrode will fail. Therefore, the *rate of discharge* is also very important to aging.

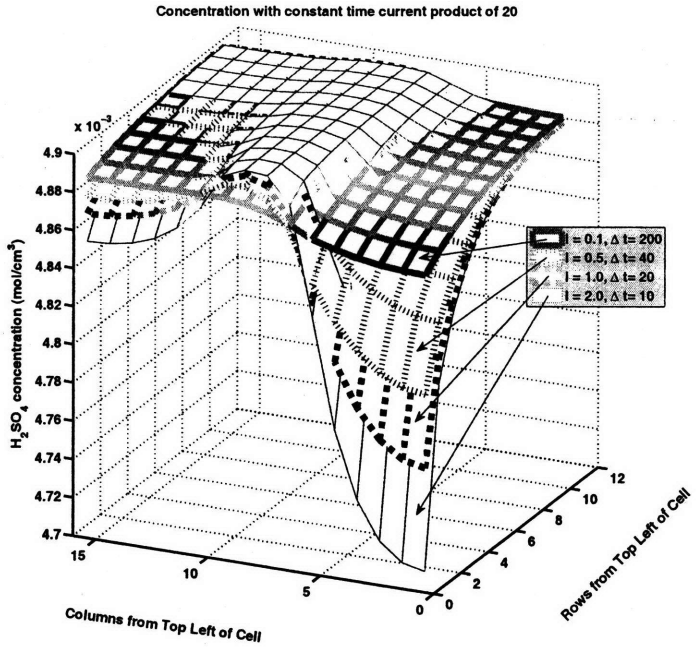


Figure 5-23. The same number of Coulombs of charge were removed from identical cells; however, the charge was removed at different rates. This shows that the *rate* of discharge is an important factor in aging.

Chapter 6

Numerics of Implementation in an Embedded Environment

This chapter introduces the Cell Broadband Engine® as a possible compute platform for an electrochemical battery model in section 6.2. It reviews Gaussian elimination in section 6.3.1. It introduces two elimination algorithms and three implementations of those algorithms in sections 6.3.3 and 6.3.7. The code for the implementations can be found in chapter C. The performance of the three implementations is discussed at length in sections 6.3.5, 6.3.6 and 6.3.9. This discussion vividly illustrates the performance achievable on the new multicore processors is extremely dependent on the algorithm and its implementation. It presents a fault tolerant implementation of one of the Gaussian elimination solvers, section 6.4. Finally it gives recommendations for future work, section 6.6.

6.1 Introduction

The compute platform for the battery condition monitoring system has, to date, been little more than a Voltmeter or Ammeter. These devices are extremely energy efficient, cost effective, reliable, and provide a considerable amount of relevant information about the amount of charge stored in the battery. In order for a platform to be considered as a replacement for the simple meters used today, it not only has to have all of their advantages but also has to provide information that cannot be obtained by simple voltmeter and ammeter measurements. Information about the battery's state-of-health, a measure of its useful age, is such a beneficial piece of information.

As noted in chapter 1, battery geometry plays an important role in battery aging. Ammeter and Voltmeter measurements alone cannot give information about battery geometry or aging. Including battery geometry information into an on-board battery monitoring system requires an extended set of sensors with more compute power.

The compute power for the battery condition monitoring system will likely come from some type of microprocessor. The trend in desktop microprocessor design is toward multicore processors [42]. Moreover, Toshiba's announcement of the Cell-Broadband-Engine-based 4-core SpursEngine® [43], capable of 48 Gflops single precision performance at only around 15W of power on a 65nm process [44], indicates that multicore processors will soon make a strong entrance into the embedded processor space. Consequently, battery models, whether embedded or otherwise, will most likely be implemented on a multicore compute platform. The Voltmeter and Ammeters will remain, but only act as sensors feeding information to the more advanced digital platform.

Unfortunately, programming the new multicore processors is a challenge for which the software industry is ill prepared [45, 46]. The problem of developing algorithms that effectively harness the power

of multicore processors is presently viewed as such a challenge that Intel and Microsoft have together announced more than $20 million dollars in investment in joint ventures with UC Berkeley and UI Urbana-Champaign to investigate this issue [47]. A consortium of companies including IBM, AMD, Sun Microsystems, Nvidia, and Hewlett-Packard have announced a $6 million dollar joint venture with Stanford University to promote research in how to write code for multicore systems [48].

Therefore, in order to advance our understanding of and explore issues associated with writing algorithms for multicore processors, the 2-D battery model of section 5.3 was implemented on the Cell Broadband Engine. As stated in section 5.3, at each Newton step a system of equations like eq. (5.38) was solved using Gaussian elimination. For this thesis three different Gaussian elimination solvers were developed and used with the 2-D battery model. Each solver is more complex than the previous solver. Each solver builds off the knowledge and experienced gained in writing the previous solver. Their development highlights the dependence of achieved performance on algorithm implementation and the difficulty in actually achieving the theoretical performance of multicore processors.

6.2 The Cell Broadband Engine

The Cell Broadband Engine (CBE) is best known as the compute brains of Sony's Playstation 3. It is the first truly heterogeneous multicore microprocessor available to mass markets. It is capable of over 250 Gflops single precision and around 26 Gflops double precision performance at about 45W on a 45nm process at 3.2Ghz. By comparison, Sandia National Laboratories' ASCI Red, the first teraflop computer, "was 104 cabinets housing 10,000 Pentium Pros and spread out over 2500 square feet. It consumed a mere 500kw" [49]. ASCI Red was the world's fastest supercomputer from introduction in June 1997 until June 2000 [50].

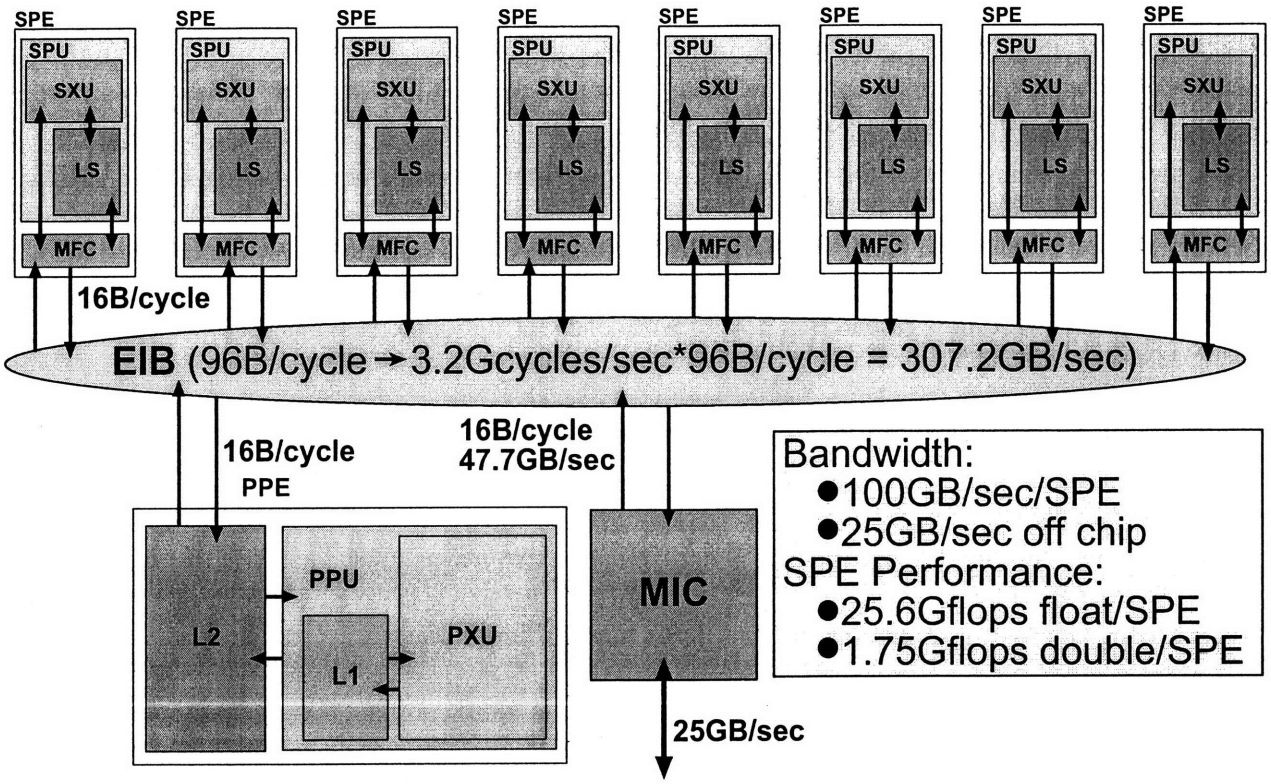


Figure 6-1. The Cell Broadband Engine is a nine core heterogeneous microprocessor. There are eight Synergistic Processing Elements (SPE) and one Power PC Element (PPE). All nine elements are interconnected by a high speed Element Interconnect Bus (EIB). The CBE differs from previous processors in its low Watts/Gflop count, its high on chip and off chip bandwidth, and its ability to quickly send short messages between processor cores.

A drawing of the internals of the CBE is shown in fig. 6-1. The CBE consists of nine cores connected by a high speed data bus. Fig. 6-1 shows eight Synergistic Processor Elements (SPE) and one PowerPC Processor Element (PPE) clustered around a ring data bus labeled the EIB in the figure. The EIB (Element Interconnect Bus) is capable of transmitting 96B/clock cycle for an internal bandwidth of 286GB/second.

6.2.1 The PPE

The PPE is a complete 64-bit Power PC processor. It is designed to be the master processor for the system, running the operating system (OS), launching the SPE threads, and helping deal work out to the SPEs. It has full branch prediction capabilities, 32kB of Level 1 instruction cache and 32kB of Level 1 data cache. It has 512kB of Level 2 cache. It has a complete AltiVec® engine for vector math operations and is capable of fused multiply add operations, something the x86 cannot do. Like all PowerPCs, it is Big Endian as opposed to the Little Endian format used by the x86. In spite of all these advanced features, there has been high dissatisfaction with its performance.

6.2.2 The SPE

The SPEs are complete processors that are highly tuned for high performance vector math at low power consumption. Each SPE consists of two parts: a Synergistic Processor Unit (SPU) and a Memory Flow Controller (MFC). The SPU consists of local memory and a compute engine. It is responsible for the computation. The Memory Flow Controller (MFC) is a Direct Memory Access (DMA) controller that acts as a coprocessor to the SPU. Its only function is to handle all the SPU's off core memory accesses.

While the SPEs do not have any cache, they do have 256kB of local storage (LS) that has to be used for both code storage and data storage. Furthermore, they have 128 128-bit general purpose vector registers for an extra 2kB of on SPE memory. The lack of cache forces the programmer to manually control all memory accesses through the MFC. While this gives the programmer greater control and possibly leads to greater performance, it also makes programming the processor extremely challenging.

Like the PPE, the SPEs are Big Endian machines, and they too can perform fused multiply add opera-

tions and support fully pipelined single precision arithmetic. However, the single precision floating point on the SPE is limited in use in that it does not support all of the rounding modes in the full IEEE-754 specification but only supports truncation. Also, single precision floating point denorms are all rounded to zero. Denorms are numbers that whose magnitude is smaller than the smallest normalized number that can be represented in floating point [51]. Both of these limitations come from the CBE's origin as a graphics processor for the Playstation 3.

Double precision floating point operations on the SPEs do support all of the rounding modes of IEEE-754. However, double precision is not pipelined properly. Double precision operations take 13 clock cycles to complete and both processor pipelines stall during the first seven clock cycles of a double precision operation. Clearly this results in a reduction of system performance, and the SPEs are collectively only capable of about 14Gflops of double precision math.

While each SPE *can* run a complete OS, it is not recommended because the SPEs lack the hardware necessary for effective branch prediction. SPEs also lack any instruction cache or data cache, both of which are helpful when running an OS as these free the programmer from having to explicitly control the flow of data to and from the processors' on-chip memory.

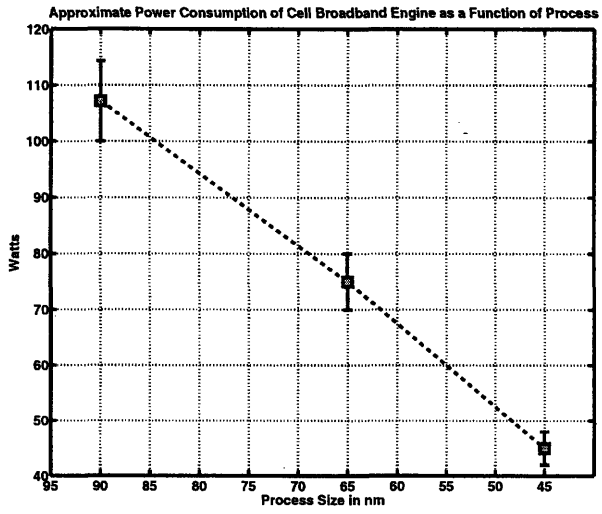
As stated above, all SPE memory accesses are performed by a dedicated DMA controller. This controller frees the SPU from having to deal with the memory access. The result is that the SPU can continue its computing in parallel with the MFC gets or puts of data from or to memory. All memory accesses are performed at 16 byte boundaries, however, ideally all data should be aligned to 128 byte boundaries. Attempting to access memory that is not located on a 16-byte boundary results in a segmentation fault and crashes the processor. All accesses are at least 128 bits (16 bytes) wide. Memory accesses of sizes smaller than 128-bits (16 bytes) are retrieved as part of a 128-bit chunk of data; however, making the

small data part of a 128-bit chunk requires extra setup time as seen in fig. 6-3(b), so small memory accesses are strongly discouraged [52, p.456]. The minimum recommended memory access size is 128 bytes. This is because the SPE cache line width is 128 bytes. A 128 byte access fills the cache line and therefore makes the most effective use of the hardware. The maximum data that can be transfered in one DMA request is 16kB.

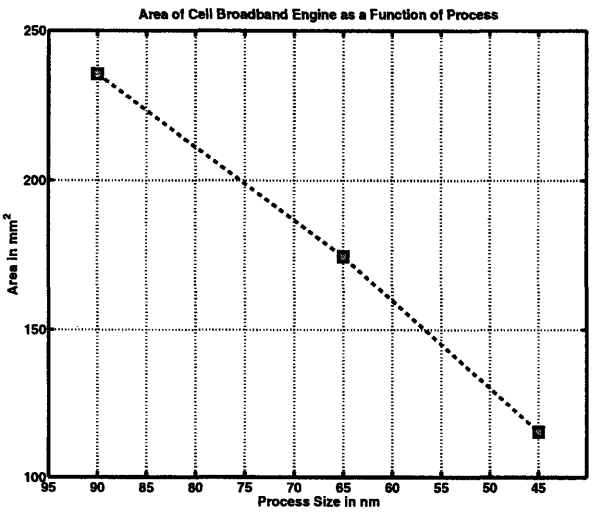
The SPEs perform all data operations on 128-bit vectors. The SPEs perform a Single Instruction for the Multiple Data (SIMD) elements of the 128-bit vector. In the case of single precision floating point data (32-bit), an SPE can operate on up to 4 floats at a time. In the case of double precision floating point data (64-bit), the SPEs can operate on up to two double at a time. When operating on scalar data, that data is stored in the vector register's preferred slot. For single precision floating point and for 32-bit integer data the preferred slot is usually bits 0 to 31 of the 128-bit vector register. In the case of double precision floating point data, it is bits 0 to 63 of the vector register. The rest of the register is left empty, but any operation performed on the scalar data is still performed on the entire 128-bit vector register.

6.2.3 Unique Features of the Cell Broadband Engine

Besides the heterogeneous nature of the CBE and the fact that all memory accesses from the SPEs must be controlled manually, there are four features that set the CBE apart from previous high performance computing systems. First is the amount of power consumed by the CBE. As seen in fig. 6-2(a), the total power used by the CBE has decreased dramatically since its introduction in 2006 from about 105 Watts to about 45 Watts at 3.2Ghz operating frequency [53–55]. The 45nm CBE, therefore, can achieve about 5.5Gflops/Watt as opposed to ASCI Red's 2Mflops/Watt. This is a more than 2000x improvement in performance per Watt in under 10 years. Because of this lower power consumption, and



(a) Two die shrinks have resulted in a 58% reduction in power consumed by the Cell when operating at 3.2Ghz.



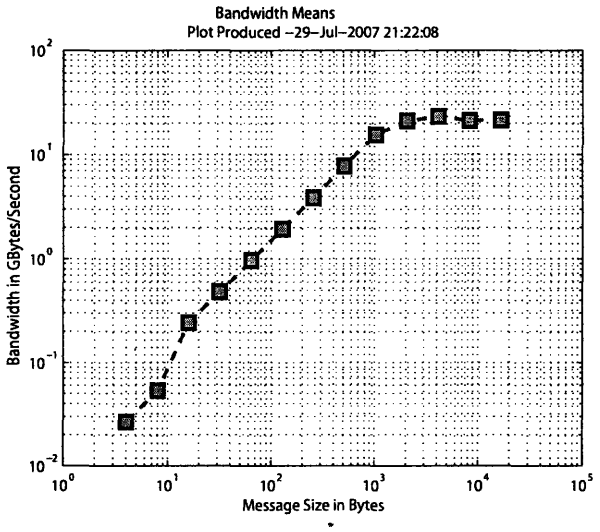
(b) Two die shrinks have resulted in a 51% reduction in die area.

Figure 6-2. Since its introduction, the Cell Broadband Engine has undergone two major die shrinks that have reduced area and power consumption.

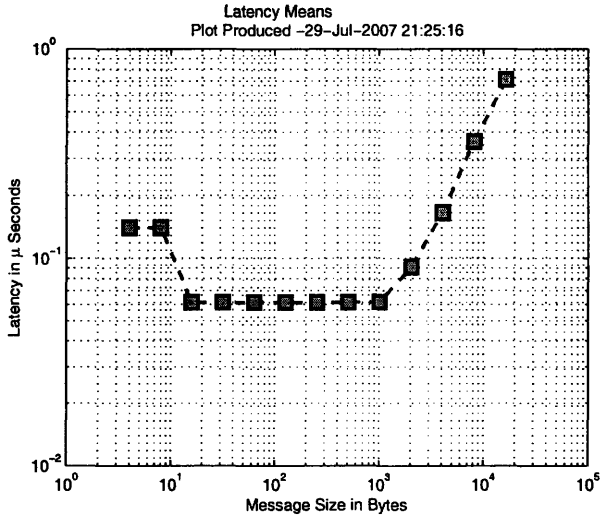
therefore high performance per Watt, the CBE enables one to start thinking about new environments where high performance computing can become feasible.

The second distinguishing feature of the processor is its compact size. When built on a 45nm process, the CBE only requires 115.46mm^2 of die area [54]. This is a reduction of required area of over 2 million times from the 232, 257, 600mm^2 required by ASCI Red [56]. While it is impossible to think of putting ASCI Red into an automobile, or other embedded environment, for the CBE, it is quite possible.

The third unique feature of the CBE is the extremely high bandwidth both on and off the chip, fig. 6-3(a). Latency is a function of bandwith, and it too is an important performance parameter, fig. 6-3(b). Data for these plots was obtained by repeatedly putting a single piece of data from the processor into main memory and then figuring out the average memory put time. Data for both of these curves was taken by the author on a Sony Playstation 3 with 256MB of XDR memory, Fedora Core 6 and IBM Cell



(a) The Cell Broadband Engine can achieve its theoretical bandwidth for larger DMA sizes.



(b) The Cell Broadband Engine can support single DMA transfers of up to 16kb.

Figure 6-3. The Cell Broadband Engine was designed for excellent memory I/O.

Broadband Engine Software Development Kit (SDK) 2.1 with xlc++ compiler version 0.8.2.

The bandwidth curve shows that, for larger data transfer sizes, the Cell Broadband Engine can approximately achieve its theoretical processor to main memory bandwidth of 25GB/sec. The bandwidth limited region of data transfer starts at transfers of 1024 bytes. The dashed line is drawn between the data points in order to help the eye follow the shape of the curve and is not meant to indicate actual achievable performance.

The latency curve is also very revealing. As previously mentioned, there is an extra performance penalty for memory transfers of fewer than 16 bytes. This can clearly be seen in fig. 6-3(b) where a memory transfer of 16 bytes has a latency of $0.06\mu s$, but a transfer of on 8 bytes requires $0.15\mu s$, 2.5 times the required time for 16 bytes. Curiously enough, the latency dominated region of memory transfer operation seems to extend out to 1024 byte transfers. This is greater than the 128 bytes of the SPE cache lines. Therefore, even though IBM recommends a minimum data transfer size of 128 bytes, it can be

said that the CBE doesn't start to perform its best, for single transfers from single SPEs, until the data transfer size is at least 8x the cache line size. A minimum 128 byte transfer size is recommended more likely because there are 8 SPEs. If each of the 8 SPEs individually requests 128 bytes of memory at once, then these 8 requests will together be the same as a 1024 byte off chip request. This will put the CBE into its bandwidth limited region of performance, fig. 6-3(a). All data points on the plot after 1024 bytes are multiples of 128 bytes, so the latency curve represents the best possible performance. As in the case of the bandwidth curve, the dashed line does not indicate actual achievable performance, but is only drawn to help delineate the shape of the curve.

This high bandwidth, combined with low latency, fig. 6-3(b), opens doors to new algorithms because one can more easily move large amounts of data and hide the data transfer costs with less computation than in previous systems.

The last unique feature of the CBE to be mentioned here is its very high intercore communication rate. This rate is more than 20x the communication rate found in previous and very expensive Infiniband systems and over 100x the performance of ethernet base systems, fig. 6-4 [57]. The small amount of time required for round trip intercore communication (fig. 6-5(a)) allows for more frequent intercore synchronizations, fig. 6-5(b). For example, a single SPE-PPE-SPE round trip communication only requires 0.3μsec when using a Playstation 3 running Fedora Core 7 with IBM SDK 3.0 and xlc++ compiler version 0.9. Synchronizing all six usable Playstation 3 SPEs takes only 1.35μsec. The sublinear increase in required synchronization time can be explained by the fact that all the SPEs can talk to the PPE simultaneously, so the SPE-PPE phase of the synchronization happens in parallel while the PPE-SPE phase happens serially. These fast synchronizations mitigate the harmful effects of core synchronization and therefore allow for more flexibility in algorithm design.

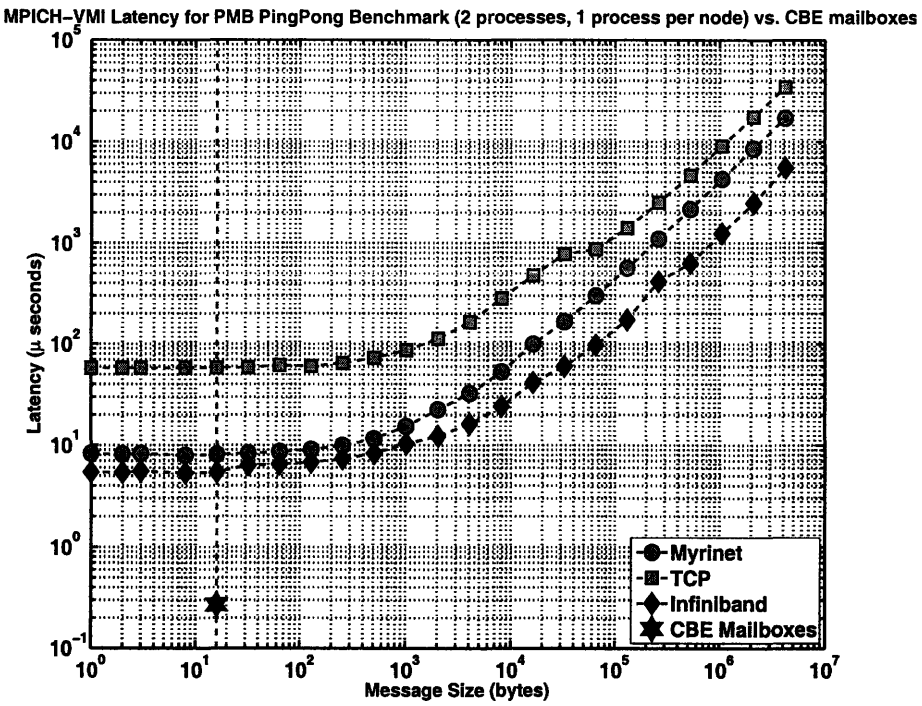
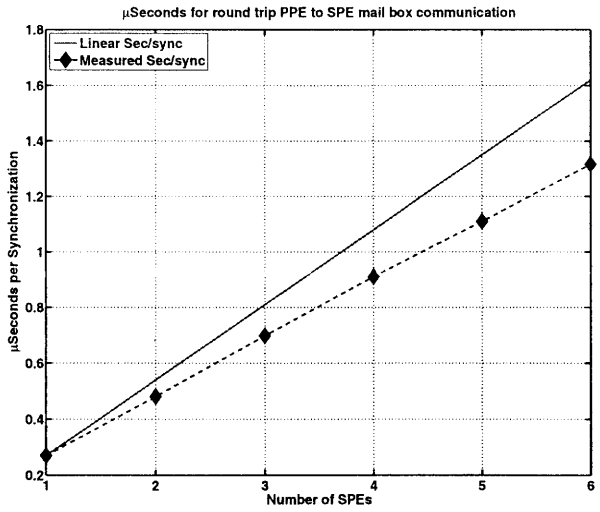
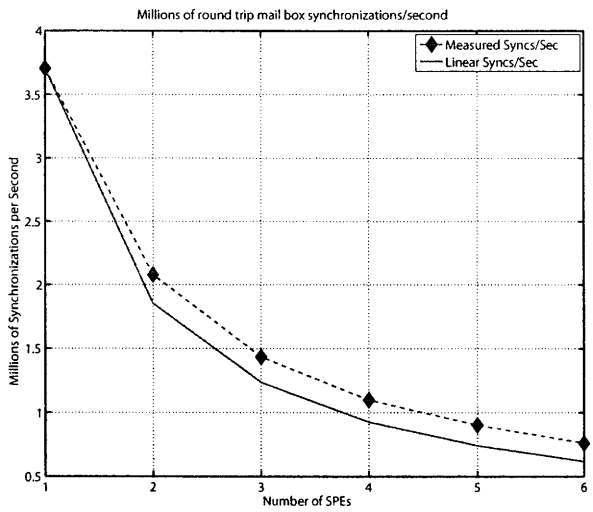


Figure 6-4. The CBE SPEs have the ability to send short messages between each other more than an order of magnitude faster than even the fastest Infiniband connected processors.



(a) For the synchronization scheme used in our LU solver, SPE-PPE-SPE round trip communication (synchronization) time grows sub-linearly with the number of SPEs.



(b) Using pointers to memory mapped mailbox registers allows for millions of SPE-PP-SPE synchronizations per second.

Figure 6-5. Addressing memory mapped SPE system registers via pointers in C is an effective method for fast communication of short messages between CBE SPEs. The data for these plots was taken on a PS3 running Fedora core 7 with IBM SDK 3.0 and xlc++ complier version 0.9.

These four features: lower power, small size, high bandwidth, and fast intercore communication allow us to think of using the high computing capability of the CBE in embedded applications. As of this writing, the only way to get access to a CBE, however, is by buying a Playstation 3 or an IBM BladeCenter QS20 or BladeCenter QS21 system. These systems are all pre-made, so we cannot use them to try to take advantage of the lower power or small size aspect of the CBE. Therefore, we will focus on trying to use the high bandwidth and fast intercore communication to our advantage when the two forward elimination algorithms and the three implementations of those algorithms are discussed in the next section.

6.3 Gaussian Elimination Solvers

The parallel Gaussian elimination solvers developed in this thesis provide a high performance way of solving the system of equations $\mathbf{Jx} = \mathbf{r}$. $\mathbf{r}$ is the residual produced at each Newton step and $\mathbf{x}$ is the correction needed at each Newton step. In our case, $\mathbf{J}$ is the $N \times N$ Jacobian matrix produced by the 2-D battery model of section 5.3. The system $\mathbf{Jx} = \mathbf{r}$ needs to be solved at each Newton step.

Gaussian elimination for sparse matrices is not a new area of research. One of the best known suites of sparse LU solvers (closely related to Gaussian elimination) is the SuperLU suite of solvers. These solvers are: SuperLU [58], SuperLU_MT [59], and SuperLU_DIST [60]. SuperLU is for sequential machines, SuperLU_MT is for shared memory parallel machines with up to 32 processors and SuperLU_DIST is for distributed memory machines. Both SuperLU and SuperLU_MT *can* partial pivot, while SuperLU_DIST uses static pivoting. Static pivoting tries to select the best arrangement of the rows before the LU solve begins. All of the SuperLU solvers use Newton's method help ensure the numerical accuracy of the solutions [61, p.6,p.9].

None of the solvers presented here partial pivot, instead they use the static pivoting with Newton refinement as is done in SuperLU_DIST. Partial pivoting, however, is an important future goal. Because the CBE can be configured to behave as a shared memory machine, a partial-pivoting-parallel-sparse LU (Gaussian elimination) solver is probably an achievable goal; however, implementing it on the CBE will most likely be a more difficult task than implementing it on a standard shared memory parallel machine as not only does one have to control the synchronization of the processors, one also has to explicitly control all memory movement. Implementing a partial-pivoting-parallel-sparse LU (Gaussian elimination) solver will have to be done in stages as the development tools for, the programming techniques for, and the understanding of this new architecture improve.

Although they do not partial pivot, the solvers presented here represent significant contributions in that they are guaranteed to work for a large class of problems that involve either symmetric positive definite or diagonally dominant matrices. Furthermore, when used with Newton refinement, they work for more problems outside that realm as shown by the battery model example of this thesis. Finally, they represent the first implementations of parallel sparse Gaussian elimination tailored for the CBE and thereby contribute to the base of experience that is needed in order to develop even more advanced and robust Gaussian solvers on this new architecture.

These solvers differ from the SuperLU family in a variety of ways including the following ways. SuperLU is a serial program while these Gaussian solvers are all parallel. SuperLU_MT is a shared memory algorithm that relies on the processor to control memory accesses and relies on the processors having data caches. These algorithms all require and implement explicit control of the memory accesses. Finally, SuperLU_DIST requires the use of the Message Passing Interface, MPI, while MPI is too large to work on the small SPE cores of the CBE. Therefore, a different non-MPI system of message passing between processor cores is utilized.

Two different algorithms and three different solvers based on those algorithms were developed. The algorithm names are the "out of core algorithm" and the "inCore algorithm." The three implementations names are called: out of core solver revision 1, out of core solver revision 2, and inCore solver. All of the solvers are implemented in double precision on the Cell Broadband Engine. While they all solve systems of equations by forward elimination and then back substitution, the three achieve very different performance. This section describes the two algorithms, investigates the three solvers and explains why they achieve different performance. Reading through the development of these three solvers will hopefully convey the sense that a mastery of the computer architecture on with the solver is being

implemented is vital for achieving good compute performance. Before diving into the solvers, however, a brief review of Gaussian elimination is presented.

6.3.1 Review of Gaussian elimination

Gaussian elimination can be used to solve the system $\mathbf{Jx} = \mathbf{r}$ by forward elimination followed by back substitution. The goal of the forward elimination sweep is to produce an upper triangular matrix that can be used in the back substitution step of the algorithm.

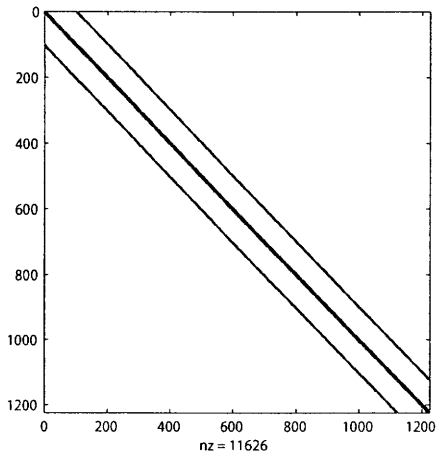
Forward elimination takes a pivot of a matrix and makes all values below that pivot equal to zero. The row that the pivot belongs to is called the *present base row*. Any row that has elimination performed on it is called an *elimination row*. The row that is presently having elimination performed on it is called the *present elimination row*. All rows in the matrix except the last row will act as a base row at some point during forward elimination. Also, with the exception of the top row, all rows in the matrix will have elimination performed on them at some point during the forward elimination sweep.

The result of the elimination operation is that all data elements in the same column as and below the base row pivot will become zero. The other elements of the elimination rows usually also change. If a

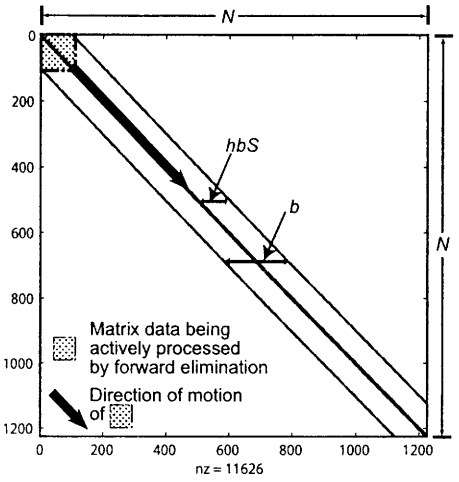
Input: The Jacobian matrix, $\mathbf{J}$ at a Newton iteration and the residual vector, $\mathbf{r}$
Result: Forward elimination is performed on the Jacobian and the residual vector

```
2.1 for i ⟵ 1 to N do
2.2 |   for j ⟵ i to N do
    |   |   // Perform elimination on row j
2.3 |   |   J_{j,i} ⟵ J_{j,i}/J_{i,i}
2.4 |   |   for k ⟵ i to N do
2.5 |   |   |   J_{j,k} ⟵ J_{j,k} − J_{j,i} × J_{i,k}
2.6 |   |   |   r_j ⟵ r_j − J_{j,i} × r_i
2.7 |   |   end
2.8 |   end
2.9 end
```

Algorithm 2: Basic Dense Gaussian Forward Elimination algorithm.



(a) This Jacobian is produced by a 5-point stencil. Notice the internal band and the two outer bands.



(b) The half band size hbS and the band size b are two important parameters.

Figure 6-6. The matrices of interest are banded in nature.

base row element is zero, then elimination will have no effect on any of the elements in the same column as the base row element. This fact will be exploited to greatly speed up the computation.

Since these solvers are being developed in conjunction with the 2-D battery model of section 5.3, let us take a look at the kind of system they will be used to solve. First, the Jacobian is being stored in memory in row major order. This means that all elements of a row are stored in contiguous memory addresses. Next, the 2-D model's Jacobian J has a special banded structure as seen in fig. 6-6(a). The banded structure of the Jacobian comes from the fact that the battery model used to tests these solvers used a 5-point nearest neighbor stencil.

The Jacobian matrix produced by the battery model is extremely sparse, fig. 6-7. The algorithms will try to take advantage of this sparsity and banded structure to help speed up computation. Because anything outside the right most or left most band edge is zero, and elimination has no effect if the base row element is zero, elimination need only be performed on data elements that lie between and include

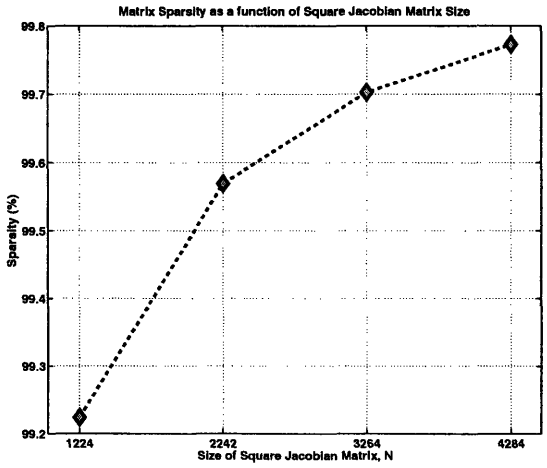


Figure 6-7. The Jacobian is extremely sparse.

the two outermost bands. Because useful computation can only occur between the bands of the Jacobian matrix, Algorithm 2 can be modified to become Algorithm 3. In Algorithm 3 hbS is the number of columns of the matrix from the pivot to the right most element of the outer band of a particular row as seen in fig. 6-8. b is the total width of the band. Because bandwidth is frequently used to mean the amount of data that can be transfered between the processor to the memory, the term *band size* will be used when referring to b. b does not have to equal, and in fact does not equal, for the Jacobian matrices of interest, $2 \times hbS$ as explained in 5.5.

Input: The Jacobian matrix, $\mathbf{J}$ at a Newton iteration and the residual vector, $\mathbf{r}$
Result: Forward elimination is performed on the Jacobian and the residual vector

```
3.1 for i ⟵ 1 to hbS do
3.2 |   for j ⟵ i to i + hbS do
    |   |   // Perform elimination on row j
3.3 |   |   J_{j,i} ⟵ J_{j,i}/J_{i,i}
3.4 |   |   for k ⟵ i to hbS do
3.5 |   |   |   J_{j,k} ⟵ J_{j,k} − J_{j,i} × J_{i,k}
3.6 |   |   |   r_j ⟵ r_j − J_{j,i} × r_i
3.7 |   |   end
3.8 |   end
3.9 end
```

Algorithm 3: Basic Banded Gaussian Forward Elimination algorithm.

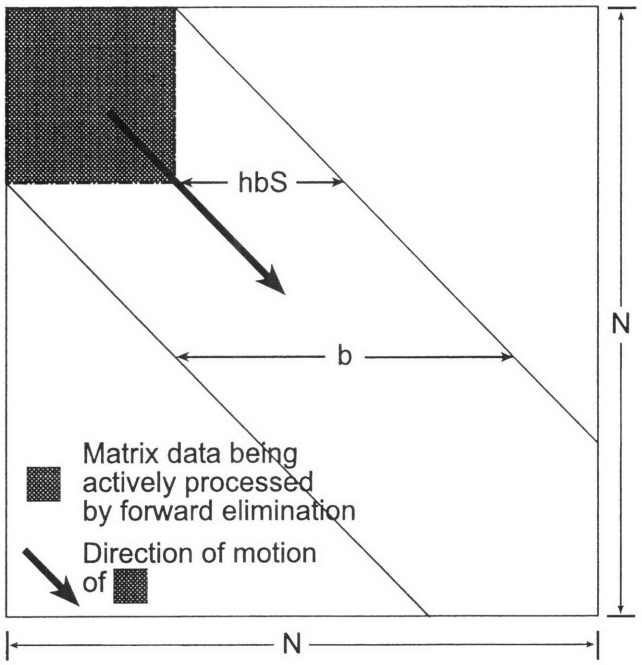


Figure 6-8. In banded Gaussian elimination, only a small amount of information is operated on for a given base row.

Banded forward elimination operates only on a small $hbS \times hbS$ sized square of the matrix at any given time. An example of this square of data is highlighted in fig. 6-6(b). This block of actively being operated upon data will be called a "work block." As previously stated, the data that is being operated on always lies between the two extreme bands of the matrix, so any values outside the outer bands will always be zero. Data between the bands that is initially zero, however, can become non-zero during the process of elimination. The zero values that become non-zero values are called fillins. The area of present operation of data moves from the upper left of the matrix to the lower right as elimination progresses, fig. 6-8. More specifically, the "work block" slides from the upper left to the lower right one row and one column at a time.

6.3.2 Two Forward Elimination Algorithms

Two forward elimination algorithms were developed. From these two algorithms, three distinct Gaussian elimination solvers were written. Each builds off either the code base or the knowledge obtained from the previous solvers. Therefore, the order of development in time is important. The order of development is as follows: out of core solver revision 1, out of core solver revision 2, inCore solver.

Out of core solver rev 1 and rev 2 share a common code source. The major difference between the two implementations is the way data is stored in memory. Also, rev 1 was implemented with IBM Software Development Kit (SDK) 1.1 whereas rev 2 utilized SDK 2.1. "Out of Core" means that all data is stored off the Cell Processor and only moved onto the processor when it is needed for computation. Immediately after computation, the data is moved off the processor back into main memory. The idea behind this algorithm is the streaming data model of computation [62]. The rows were thought of as a continuous stream of data that flows into and out of the processor. The second algorithm implemented as a solver was the author named "inCore" algorithm. It is embodied in the inCore solver. It does not share a code base with the out of core solvers, however, it does try to make use of many of the things learned by implementing those solvers. It is called an "in core" algorithm because data is moved onto the chip and persists there until the algorithm can no longer possibly ever use the data again. Thus, the data remains "in core."

6.3.3 Out of Core LU Algorithm and Solvers

The out of core LU solvers were written first. There are two slightly different incarnations of this solver. Their differences in implementation and performance will be explored below, but first their common algorithm, Algorithm 4, is explained. The basic idea of the algorithm is that an SPE brings a row of data from main memory into the SPE local store, performs elimination on it, and then sends the row back to main memory.

Using this basic idea as our guide, the number of compute operations and memory accesses performed by the algorithm can be tabulated. The compute operations and memory accesses are summarized in table 6.1. The compute/memory ratio in the table indicates that the algorithm will improve in performance as the width of the band increases. Also, it seems that performance is independent of the matrix height and width, N.

Even though the basic idea of the algorithm is sound, it would be a terrible waste of time if every time the processor wanted to perform elimination on a row, it stopped to get the row from memory, performed elimination on it, and then stopped again to put the results of the computations back into memory. Instead, the processor fetches its *next* elimination row, Algorithm 4.9, but instead of waiting for that row to arrive, the processor preforms elimination on the elimination row it presently has in its local memory, Algorithm 4.13. Then the processor puts the row it just performed elimination on back into memory, Algorithm 4.15, and waits for both the row fetch and row put to complete, Algorithm 4.16. Then the loop repeats with the next elimination row.

Implementing the next elimination row pre-fetch requires the SPE to have two buffers for elimination rows. While the SPE is performing elimination on the row in one buffer, the other buffer is receiving the next row from memory. Once elimination is completed on the first row, the SPE switches to the

Input: The Jacobian matrix, $\mathbf{J}$ at a Newton iteration with half-band size hbS and the residual vector, $\mathbf{r}$
Result: Forward elimination is performed on the Jacobian and the residual vector

4.1 **for** $i \longleftarrow 1$ **to** $N-1$ **do**
    // Fetch base row i from main memory
4.2 $\mathbf{J}_{\text{local},i} = \texttt{post\_recv}(\mathbf{J}[\mathbf{P}[i], i : i + hbS])$
    // Fetch first elimination row for this SPU, $i+$spuid from main memory
4.3 **if** $i + spuid \leq N$ **then**
4.4 $\mathbf{J}_{\text{local},i+\text{spuid}} = \texttt{post\_recv}(\mathbf{J}[\mathbf{P}[i + spuid], i + spuid : i + spuid + hbS])$
4.5 **end**
    // Wait for base row and first elimination row to arrive
4.6 $\texttt{wait\_for\_completion}(\mathbf{J}_{local,i}, \mathbf{J}_{local,i+spuid})$
4.7 **for** $j \longleftarrow i + spuid$ **to** $i + hbS$ **do**
        // Pre-fetch next elimination row for this SPU, $j+$spuid from main memory
4.8 **if** $j + spuid \leq N$ **then**
4.9 $\mathbf{J}_{\text{local},j+\text{spuid}} = \texttt{post\_recv}(\mathbf{J}[\mathbf{P}[j + spuid], j + spuid : j + spuid + hbS])$
4.10 **end**
        // Perform elimination on row j
4.11 $\mathbf{J}_{\text{local},j}[i] \longleftarrow \mathbf{J}_{\text{local},j}[i] / \mathbf{J}_{\text{local},i}[i]$
4.12 **for** $k \longleftarrow i+1$ **to** $i + hbS$ **do**
4.13 $\mathbf{J}_{\text{local},j}[k] \longleftarrow \mathbf{J}_{\text{local},j}[j] \times \mathbf{J}_{\text{local},i}[k]$
4.14 **end**
        // Post the updated row back to main memory
4.15 $\texttt{post\_send}(\mathbf{J}_{local,j})$
        // Wait for all pending posts
4.16 $\texttt{wait\_for\_completion}(\mathbf{J}_{local,j} \& \mathbf{J}_{local,j})$
4.17 **end**
    // Wait before starting next base row
4.18 notify_PPE
4.19 **end**

Algorithm 4: Out of Core algorithm for parallel forward elmination.

Out of Core Algorithm Operation Count	
Compute Operations	
Multiplies	$N\,(hbS^2)$
Subtracts	$N\,(hbS^2)$
Total Compute	$2N\,(hbS^2)$
Memory Access Operations	
Memory Reads	$N\,(hbS)$
Memory Writes	$N\,(hbS)$
Total Memory Access	$2N\,(hbS)$
Compute to Memory Access Ratio	
Compute/Memory	hbS

Table 6.1. Performance of the out of core algorithms.

buffer that has just received a row and starts elimination on the data in that buffer. In the mean time, the original compute buffer is now receiving the next elimination row. This process of using two buffers of alternating functionality, one for elimination and one for memory accessing, is known as double buffering. Double buffering helps ensure that, the processor never stops to get data from memory. Unfortunately, it still stops when writing data back to memory, Algorithm 4.15.

The out of core solver is SIMDized [63]. SIMD is an acronym that stands for Single Instruction Multiple Data. Elimination fits perfectly into this classification of computation. There is only one operation that needs to be performed on every element in an elimination row. Ideally the computer could perform the elimination operation on every elimination row element in a single pass. No machine can do this right now, but it is common to find machines that can do the same operation on two double precision numbers at once. The SPEs have vector registers that allow operating on two double precision numbers at one time. This allows the SPEs to perform elimination on two columns at a time. All solver implementations take advantage of this feature.

SPE level parallelization is achieved by assigning rows to SPEs in a round robin fashion. Initially

rows are assigned to SPEs by their SPE number also known as their rank aka spuid. SPE 0 would get row 0, SPE 1 gets row 1 and so on and so forth. Each SPE knows how many SPEs there are. For the purpose of illustration, assume there are Z of them. If an SPE is performing elimination on row k, at the same time it is fetching row $k + Z$.

6.3.4 The Out of Core Algorithm's Two Implementations

As mentioned earlier, there are two incarnations of the out of core algorithm. The first written incarnation is called out of core solver revision 1. The second later written incarnation is called out of core solver revision 2. They differ in both the software used to develop the solvers and the way they store data in memory.

Out of core solver revision 1 was implemented using the IBM Cell Broadband Engine SDK 1.1. This incarnation stores its data in dense form in memory. This means that all matrix entries are stored in row major order in memory. The memory storage pattern for a single row of data for the out of core solver revision 1 can be seen in fig. 6-9. The entire row is stored in memory and retrieved from memory when needed. The position of the pivot within the memory storage block changes depending upon the row.

Out of core solver revision 2 was implemented using IBM Cell Broadband Engine SDK 2.1. It stores its data in a custom sparse form in main memory. The sparse storage format utilized by both the out of core solver and the inCore solver is shown in fig. 6-10. The position of the pivot is at a fixed offset within the memory block for all rows. In the case of storing things in a dense format, when a row was brought in for elimination, all the data members were already properly aligned. However, with the new sparse format, there needs to be an extra step of aligning the data before elimination can be performed. This alignment step will be shown to have a dramatic effect on solver performance.

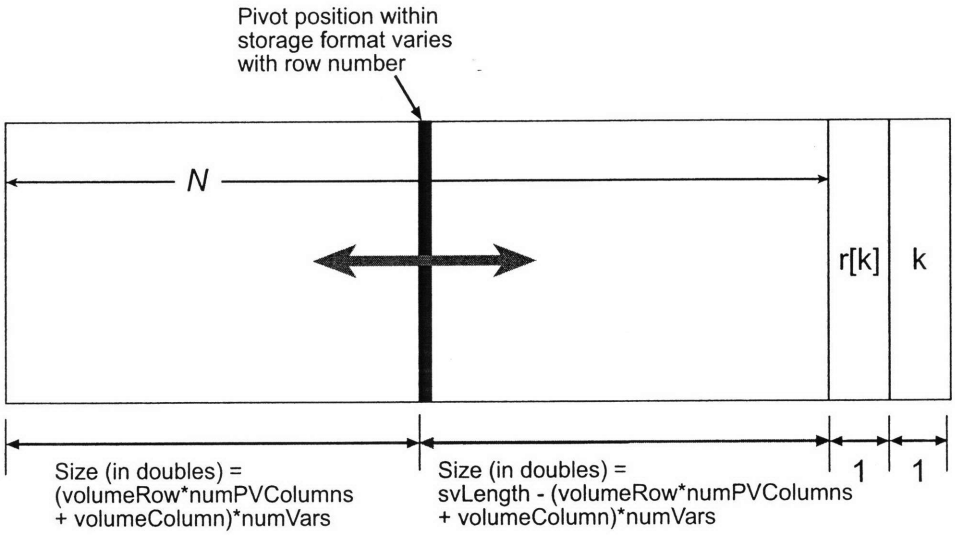


Figure 6-9. The memory storage pattern of each row of the Jacobian matrix for out of core solver Revision 1.

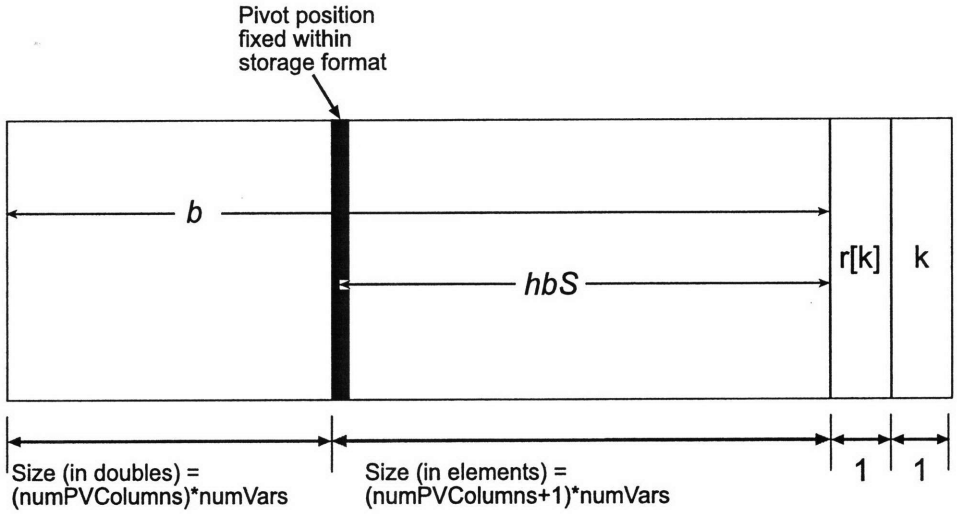


Figure 6-10. The memory storage pattern of each row of the Jacobian matrix for both out of core solver Revision 2 and the inCore solver.

Both SDK 1.1 and SDK 2.1 were alpha level products; however, SDK 2.1 was significantly more reliable than SDK 1.1. Since SDK 1.1 was a very alpha level product, IBM felt it had the right to change the API when it introduced SDK 2.1. Therefore, programs written for SDK 1.1 cannot work on a later SDKs. Programs written on SDK 2.1 can be quickly migrated to SDK 3.0. Both products, SDK 1.1 and SDK 2.1 are now defunct and have been replaced by the IBM SDK for Multicore Acceleration, Version 3, aka SDK 3.0. Neither SDK 1.1 nor SDK 2.1 is obtainable through regular channels as of the writing of this document.

6.3.5 Out of Core Solver Revision 1 Performance

Four different matrices were used to test the performance of out of core solver revision 1. These matrices were generated by the 2d battery model and are of sizes: 1224x1224, 2142x2142, 3264x3264, 4284x4284. All four tested matrix sizes had the same half band size hbS of 210 doubles. The tests were conducted on a Playstation 3 (PS3). The maximum matrix size was limited because the PS3 only has 256MB of XDR system memory and the matrices were stored in dense format in the system memory.

Fig. 6-11 shows the performance of the solver for all four matrix sizes. Gigaflops (Gflops) were computed using:

$$\text{Gflops} = \frac{hbS^2 * N}{\Delta t \, (1024)^3} \tag{6.1}$$

Where hbS is the half band size of the matrix as defined in fig. 6-8 and N is the length and width of the matrix also shown in fig. 6-8. The number of SPEs used during the computation are listed on the x axis. The y axis displays the Gflops attained by the solver. It is a loglog plot.

Included on the graph are several straight dotted lines that diverge rather quickly from the dashed data lines. These straight dotted lines represent the linear speed up lines. The linear speed up lines delineate the theoretical best performance (as measured in Gflops) that could be achieved as more processors are added. From this plot, it is clear that the solver does not come anywhere close to achieving a linear speed up as more SPEs are added.

Failure to achieve linear speed up, however, does not mean that the absolute performance of the solver is bad. In fig. 6-12, the performance of out of core solver revision 1 is compared to Matlab solving the same system of equations on a 2Ghz AMD Opteron 246 processor. The theoretical maximum

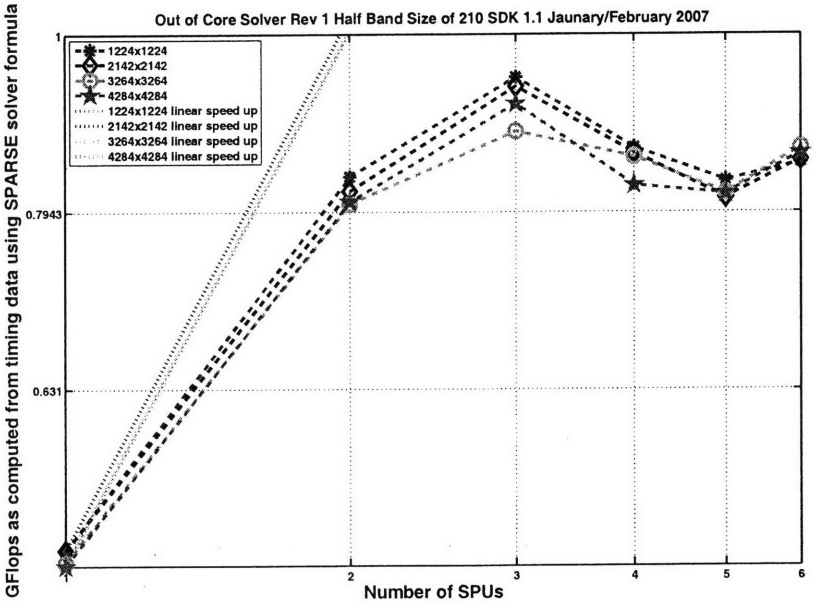


Figure 6-11. Out of core solver revision 1 has modest performance, but does not achieve a linear speedup.

performance of the Opteron 246 is 4Gflops double precision.

This plot shows that the performance of the solver is fairly independent of matrix size. This is because on the matrix parameter N was varied while the band size was held constant. Varying only N simply creates a linear increase in the total number of compute operations needed by the forward elimination sweep. These new compute operations can be computed in a linear increase in time. Therefore, there should be no change in Gflops performance due to an increase in N. For that reason, all future performance tests will have hbS varied while N will be held approximately constant. This is logical from a theoretical perspective; however, as will soon be shown with out of core solver revision 2, this is not necessarily the case.

For larger matrix sizes, out of core solver rev 1 seems to perform better than UMFPACK as called by Matlab. The out of core solver rev 1, however, does not come close to the performance of LAPACK

as called by Matlab. LAPACK's performance is fairly independent of matrix size, but UMFPACK's varies greatly. The decrease in UMFPACK performance probably has to do with the size of the matrix compared to the size of the 1MB of L2 data cache on the Opteron 246 processor. Once the matrix exceeds a certain size, it no longer fits entirely into the L2 cache, so UMFPACK then has to start going back to main memory which slows things down. This is our first example of where matrix size can effect performance.

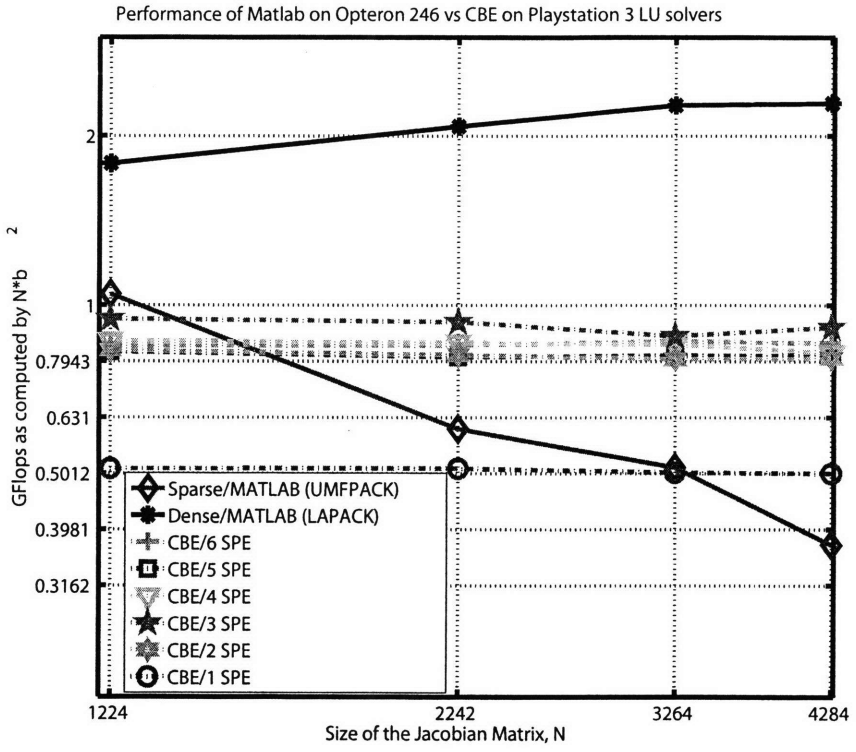


Figure 6-12. Out of core solver revision 1 beats UMFPACK as called by Matlab for larger matrix sizes, but it does not come close to the performance of LAPACK.

6.3.6 Out of Core Solver Revision 2 Performance

Fig. 6-13 shows the performance of the second revision of the out of core algorithm vs the number of SPEs used to perform forward elimination. The width N of the matrix was held approximately constant at $N \approx 33500$ while the half band width hbS was varied from 66 all the way to 282.

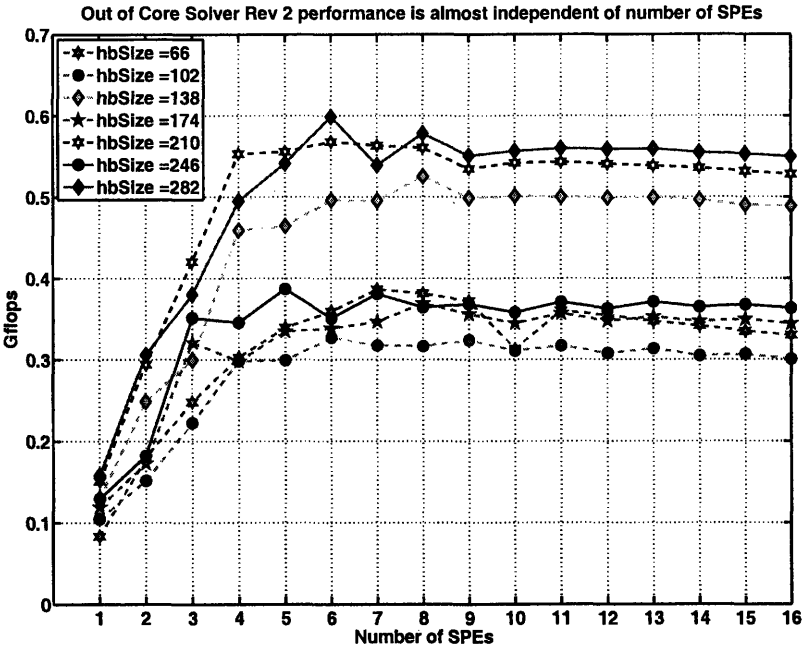


Figure 6-13. The performance of the out of core solver revision 2 is almost independent of number of SPEs employed. Furthermore, there seem to be two distinct performance regions.

There are a number of points of interest in fig. 6-13. First, there are now 16 SPEs because, instead of using a Playstation 3, this data was taken on an IBM BladeCenter QS20. Time on the QS20 system was donated by Dr. Rodric Rabbah of IBM at Thomas J. Watson. The QS20 consists of two CBEs linked together by a high speed bus on the same motherboard. They have access to 1GB of system memory (512GB each).

The extra SPEs provide a much clearer view as to the attainable scalability of the solver. While the

extra memory, combined with the new sparse representation, allows the LU solver to tackle much larger problems.

Returning to fig. 6-13, there seems to be some performance gain up to 4 SPEs instead of three SPEs as in the first revision of the solver in fig. 6-11. This speed up comes from a speed up in the synchronization of the SPEs afforded by the move from IBM SDK 1.1 to IBM SDK 2.1. IBM SDK 2.1 allows programmers to map certain SPE system registers to global memory. This allows other SPEs and the PPE to write to these registers via pointers to those registers. IBM SDK 1.1, did not have this functionality. Instead users called IBM-provided libraries that allowed access to these registers. The internal operation of these libraries was unknown, however, the library calls are more than 20 times slower than using pointers. Unfortunately, the mapping of the system registers did not work completely properly, so a hybrid combination of using IBM's library calls and pointers was adopted. This lead to the improved performance, but not to the level that one might hope. The mapping of system registers problem has been fixed in the new IBM SDK 3.0.

Second, performance appears to only gradually decrease with added SPEs beyond 4 SPEs. This seems to indicate that performance is not being limited by the computation or by the synchronization of the SPEs. Instead, at this point, the performance is probably being dominated by waiting for the put to memory that occurs at the end of each row, Algorithm 4.15.

Third, there seems to be two different distinct levels of performance based on the half band size of the matrix. The higher level of performance contains matrices with half band size of 282, 210 and 138. The lower level of performance contains matrices of half band size 246, 174, 102, and 66. The cause of this anomaly can be understood by looking at the data from a different angle. Instead of plotting curves where the half band size if fixed and the number of SPEs varies, in fig. 6-14 we plot curves where the

number of SPEs is fixed and the half band size is allowed to vary.

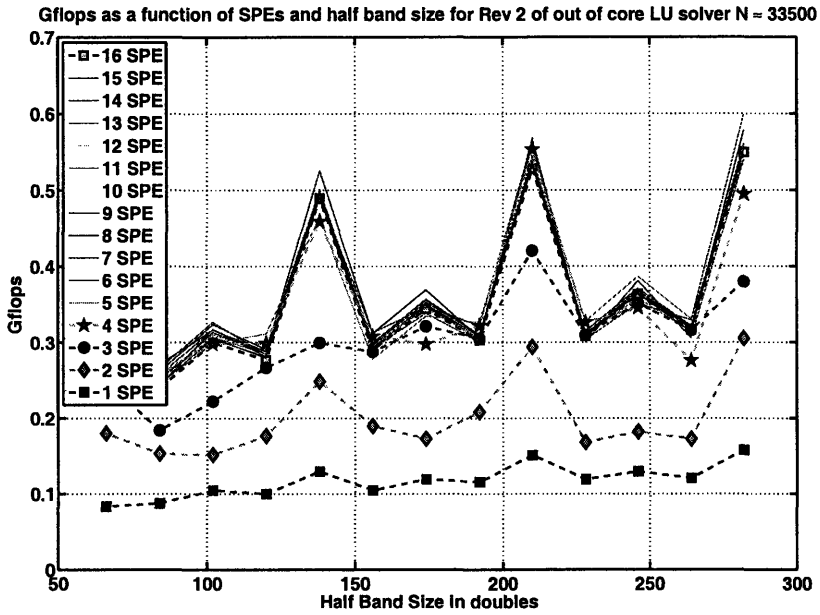


Figure 6-14. Plotting the same data as in fig. 6-13 against hbS shows distinct spikes in performance for $hbS = 138, 210,$ and 282.

As was seen in fig. 6-13, fig. 6-14 also shows some speed up for the first 4 SPEs, but there is none for adding more than four SPEs. Also, one can now see that there is, in general, a slight increase in performance for an increase in the band size of the matrix.

However, the most prominent feature of the plot are the huge spikes of performance at half band sizes 138, 210, and 282. Since waiting for the put to memory, Algorithm 4.15, is theorized to be the biggest time constraint, we will try to find a reason as to why the writes to memory of those sizes would be any faster than those of the other sizes. One possibility is outlined at [52, p.456] where it reads, "Transfers of less than one cache line (128 bytes) should be used sparingly; excessive use of short transfers wastes bus and memory bandwidth."

Now, the total bytes transfered is not the half band size times the size of a double. Instead, it is the full

band size plus the two extra datum for the residual and the line number. The formula for the full number of bytes transfered is: bytes transfered per row = (2*hbS - 4)*sizeof(double).

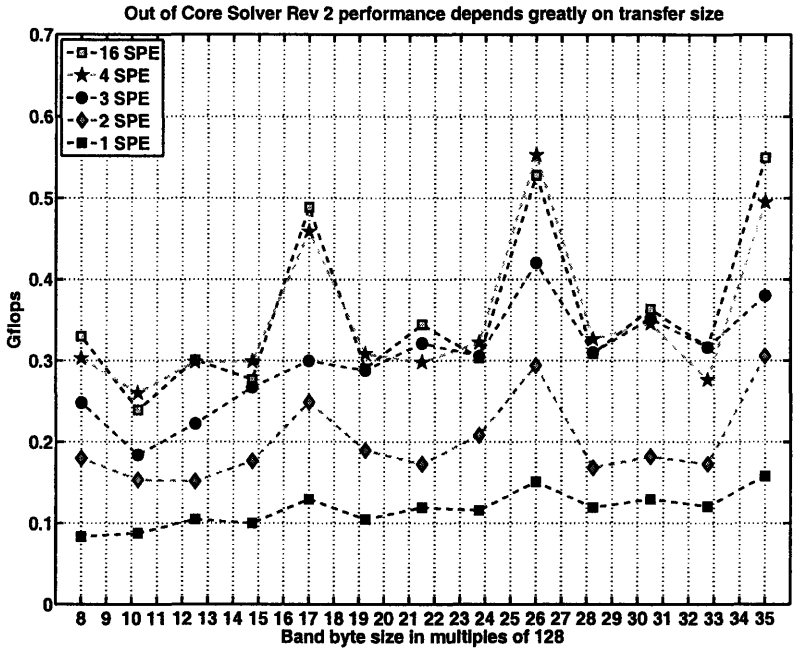


Figure 6-15. The spikes in performance occur at multiples of the cache line size of 128 bytes.

Fig. 6-15 shows the same data as fig. 6-14 except for the x axis has been changed to reflect the data transfered in multiples of 128 bytes. The performance peaks occur at exactly a multiple of the cache line size of 128 bytes. There is even a slight upturn in performance at 8x128 bytes. This is 1024 bytes and that corresponds to the measured start of the bandwidth limited region of performance in fig. 6-3. This seems to strengthen the idea that the put and not SPE synchronization time is what is delaying the performance.

If this were the case, though, why isn't there an obvious spike in performance for the out of core solver rev 1 on the 3264 matrix as each of its rows is 204x the size of the cache line? The reason is probably that the performance of the out of core solver revision 1 is not limited by the memory put. Instead, it

is limited by the extremely slow SPE synchronization step. This is indicated by the dramatic drop in performance seen when going from 3 SPEs to 4 and 5 SPEs.

Finally, even at its best performance level, the second revision does not come close to the performance of the first revision. This is most likely due to the fact that in rev 1, the data was brought into an SPE and there was no adjusting necessary to get it to align properly. However, rev 2 requires quite a bit of adjustment. This adjustment involves some divides and some modulos, so the operation is slow, thus slowing the computation down.

6.3.7 inCore LU Solver

The out of core LU solvers showed that performance can be very dependent on memory access latency. Fig. 6-14 seemed to indicate that memory access latency could have a greater effect on performance than the addition of more processors. Therefore, the inCore algorithm tries to completely hide memory accesses with computation.

Designing an algorithm that successfully overlaps memory access with computation requires a good understanding of the data flow.

It is known that the working blocks are small in size, so start by assuming that there is a processor that can store an entire working block in its on-chip high-speed memory. The rest of the matrix, however, must reside in some off chip memory.

Next, look at how the working block evolves with time. Starting with Fig. 6-16, it can be seen that the difference between the first working block and the second working block only happens around the edge of the working block. The top row and the bottom row are different. Also, the left column and right column are different, but most of the data remains the same. Therefore, there is no reason to send that data out to memory and then bring it back onto the core as was required by the out of core algorithm.

Because the top row is the base row, it does not have elimination performed on it. A copy of it must stay on chip while elimination is being performed from it, but a copy of it can also be transmitted to main memory while the computation proceeds. Moving the top row of the present working block out to main memory and moving the bottom row of the next working block in from main memory can happen in parallel with the data computation as neither gets operated on during elimination under the first working block.

Nothing needs to be done with the left column as that column is guaranteed to be all zeros, so the trick

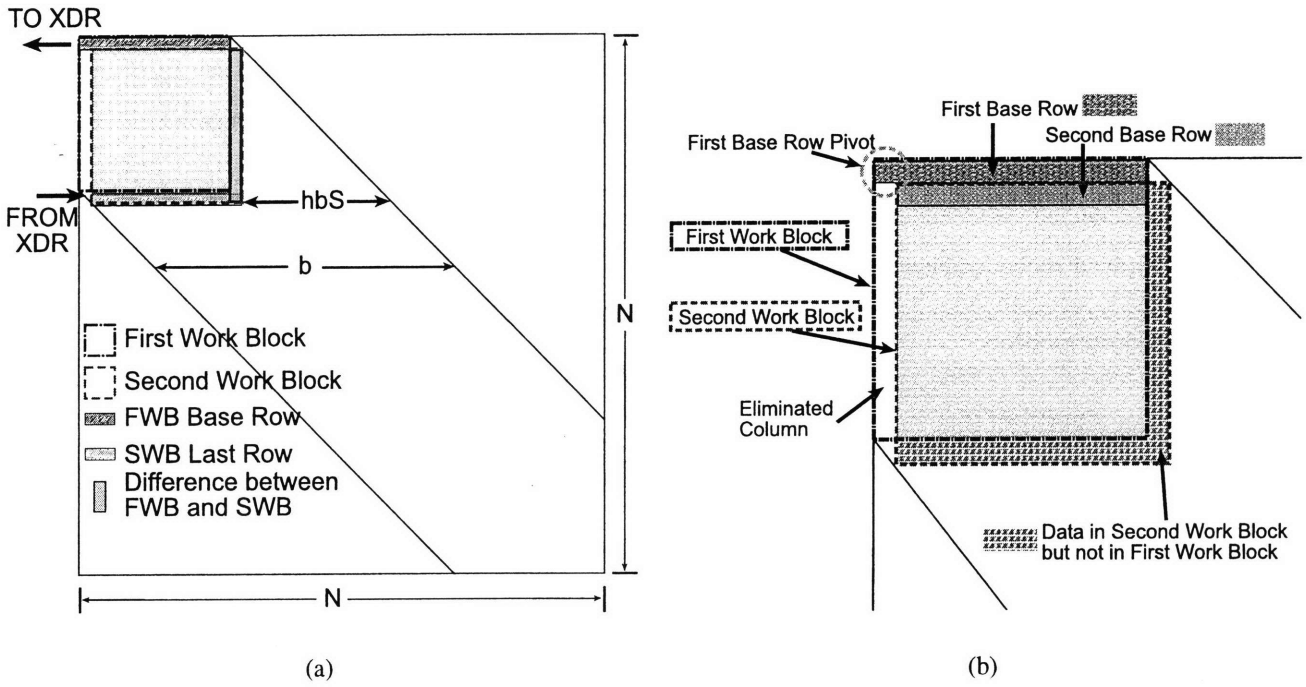


Figure 6-16. The basic idea of the inCore algorithm.

is how to get the right column onto the processor. Remember, the data is represented as contiguous rows in memory. Therefore, data elements of columns are strided across memory. According to latency curve fig.6-3(b), getting one double (eight bytes) would be prohibitively time consuming, so there has to be a better way to get the column data into SPE local store. The answer is instead of just storing the half-band size of data needed for immediate computation, the whole band is stored in on processor memory. This brings the column on chip for free since there is plenty of bandwidth for bringing on the full band size of data instead of just the half-band of data that has been shown for purposes of explaining the algorithm. Therefore, the actual situation is more like that shown in fig. 6-17. The tradeoff is that we cannot store as large a working area as we would if we only stored the half row.

The total number of computes and memory accesses can now be tabulated as was done for the out of core algorithm. The tabulation is summarized in table 6.2. In the table we see that the compute/memory

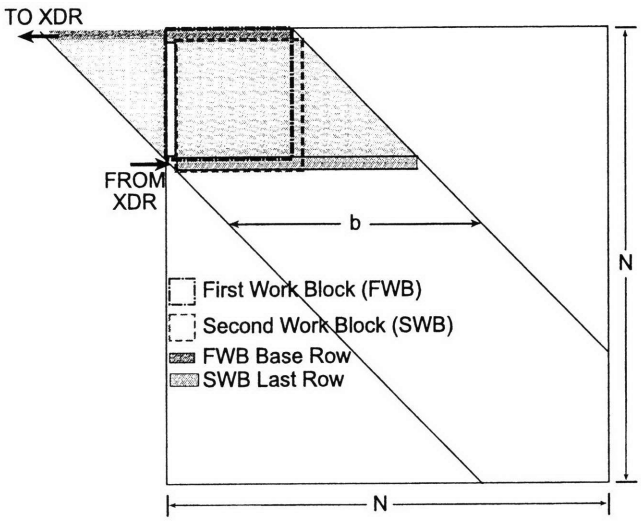


Figure 6-17. The inCore solver actually stored the complete band of data in local memory.

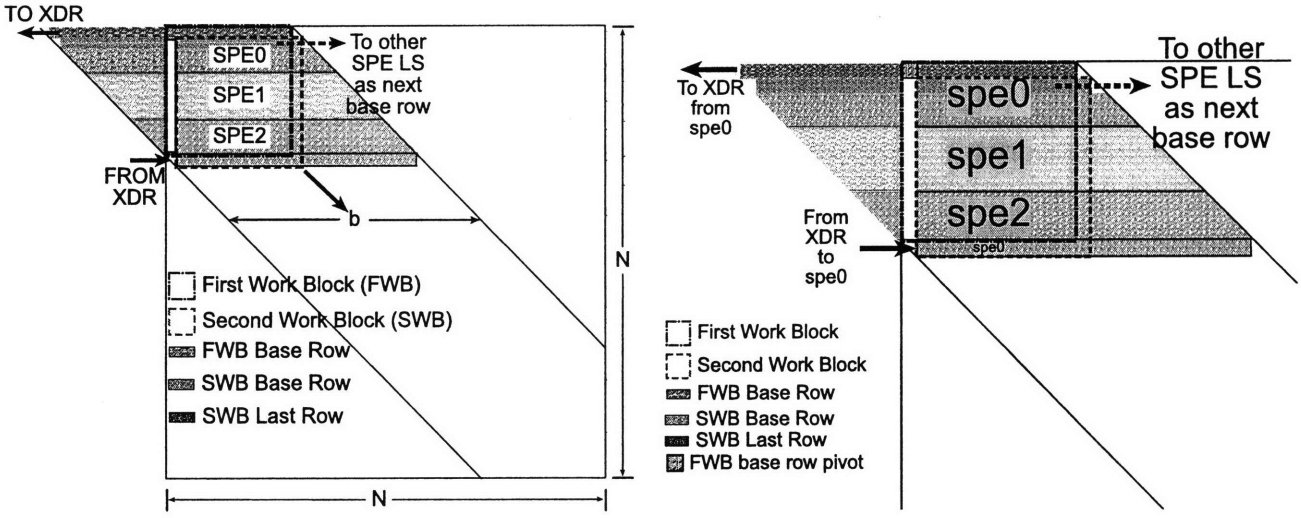
access ratio for the inCore algorithm is hbS^2 as opposed to the out of core algorithm's hbS. This indicates that, for a large enough problem, the perturbations in memory performance should be negligible when compared to overall performance.

Furthermore, there are only $2N$ memory accesses for the inCore algorithm as opposed to the $2N$ (hbS) needed by the out of core algorithm. The hbS difference between the two total memory accesses can be a huge difference. Putting numbers to it for effect, for a 33500x33500 matrix with half-band size of 210 and a band size of 416, almost 43.6GB of data would be moved by the out of core algorithm. In the case of the inCore algorithm, a 33500x33500 matrix with half-band size of 210 and a band size of 416 would only require 0.207GB of data would have to be moved.

SPE level parallelization is achieved in the inCore algorithm by assigning blocks of rows to each of the SPEs, fig.6-18(a). Parallelization requires the additional memory access of transferring of the next base row to all of the other SPEs, as shown in fig. 6-18.

inCore Algorithm Operation Count	
Compute Operations	
Multiplies	$N\,(hbS^2)$
Subtracts	$N\,(hbS^2)$
Total Compute	$2N\,(hbS^2)$
Memory Access Operations	
Memory Reads	N
Memory Writes	N
Total Memory Access	$2N$
Total Bytes Moved	$2N \times b \times$ sizeof(double)
Compute to Memory Access Ratio	
Compute/Memory	hbS^2

Table 6.2. Performance of the inCore algorithm.



(a) Rows are assigned by blocks to the SPEs.

(b) The SPE that sends the base row to memory receives the new bottom row.

Figure 6-18. The basic idea of the inCore algorithm.

```
// spu_id: numerical id of SPE
// jnhpc: number of rows of band below and including the diagonal in
   a given column
// hbS: number of columns to the right of the diagonal in a given
   column
// nSR: number of rows stored in an SPE's Local Store
// lastRowOwner: spu_id of the SPE that will perform elimination on
   the last row of the matrix
// pbr: present base row
// per: present elimination row
// brPtr: pointer to pbr
// nxtbrPtr: pointer to the next base row
// myRows: Array in SPE LS of rows presently being operated on
// brArray: Array containing pbr and nxtbrPtr
// myMinRow: minimum row number of rows in myRows
// myMaxRow: maximum row number of rows in myRows
// pbrOwner: spu_id of SPE that is in charge of sending out pbr after
   and elimination set
// myRowsElimStartIndex: index in myRows of the first elimination row
   for the present base row
// myRowsElimRowIndex: index in myRows of the present elimination row
// J: Jacobian stored in main memory
// pbrPvt: value of the pivot element in the pbr
// perPvt: index of the pivot element in the per
// inv_pbr_factor: 1/pbrPvt
// adjFact: number used to multiply pbr value in elimination step
// numSPE: number of SPE presently running algorithm
// lastBaseRow: row number of last base row
// numVars: number of physical variables in each volume
// numRows: number of rows in J
```

Algorithm 5: Definitions of variables used in Algorithms 6 and 7

```
Result: Forward elimination is performed on the Jacobian and the residual vector
6.1  pbr ⟵ 0;
6.2  nSR ⟵ compute_numStoredRows();
     // Fetch initial block of nSR rows from main memory
6.3  myRows ⟵ inDMA(J[spu_id*nSR:(spu_id + 1)*nSR- 1, :]);
6.4  brArray[0] ⟵ inDMA(J[0]);
     // Begin Computation
6.5  for pbr ⟵ 0 to N − 1 do
         // Assign brPtr and nxtbrPtr pointers depending on pbr
6.6      brPtr ⟵ brArray[pbr%2];
6.7      nxtbrPtr ⟵ brArray[(pbr + 1)%2];
         // Assign myMinRow and myMaxRow values
6.8      myMinRow ⟵ myRows[0];
6.9      myMaxRow ⟵ myRows[(nSR − 1)];
         // Assign pbrOwner, put pbr into main memory, get row hbS rows
            ahead
6.10     pbrOwner ⟵ 0
6.11     if pbr ≥ myMinRow && pbr ≤ myMaxRow then
6.12         pbrOwner ⟵ 1
6.13         J[pbr] ⟵ outDMA(brPtr);
6.14         if pbr < lastBaseRow then
6.15             myRows[(pbr − myMinRow)%nSR] ⟵ inDMA(J[pbr + hbS])
6.16         end
6.17     end
         // Perform elimination on myRows
6.18     EliminationOnRows(brPtr,myRows,pbr,nxtbrPtr)
6.19     wait_for_DMAs();
         // Synchronize SPEs via the PPE
6.20     notify_PPE();
6.21     wait_for_PPE();
6.22 end
```

Algorithm 6: In Core algorithm forward elimination procedure.

```
   Input: brPtr,myRows,pbr,nxtbrPtr
   Result: Forward elimination is performed on the rows in myRows
   // Compute inverse of pbrPvt
7.1 inv_pbr_factor ⟵ 1\pbrPvt;
   // Compute index into myRows of first row to do elimination on
7.2 if pbr − myMinRow ≥ 0 then
7.3 |  myRowsElimStartIndex ⟵ pbr − myMinRow + 1;
7.4 else
7.5 |  myRowsElimStartIndex ⟵ 0;
7.6 end
   // Perform elimination on rows stored in myRows
7.7 for i ⟵ 0 to nSR − pbrOwner do
7.8 |  b ⟵ (myRowsElimStartIndex + i)%nSR;
7.9 |  per ⟵ JacobianRowNumber(myRows[b])
7.10 |  if per = numRows − 1 or per = pbr + jnhpc − pbr%numVars − 1 then
7.11 |  |  i ⟵ nSR − pbrOwner + 1
7.12 |  end
7.13 |  if per < pbr + jnhpc − pbr%numVars then
     |  |  // Find per column below pbrPvt
7.14 |  |  pec ⟵ perPvt − (per − pbr)
7.15 |  |  adjFact ⟵ myRows[b, pec] * inv_pbr_factor
     |  |  // Perform Elimination on the Row
7.16 |  |  for j ⟵ 0 to hbS − pbr%numVars do
7.17 |  |  |  myRows[b, pec + j] ⟵ myRows[b, pec + j] − adjFact * brPtr[pbrPvt + j]
7.18 |  |  end
     |  |  // Broadcast nxtbrPtr to all SPEs
7.19 |  |  if per = pbr + 1 then
7.20 |  |  |  for k ⟵ 0 to numSPE- 1 do
7.21 |  |  |  |  spu_id[brArray((pbr + 1)%2)] ⟵ put(myRows[b])
7.22 |  |  |  end
7.23 |  |  end
7.24 |  end
7.25 end
```

Algorithm 7: The Elimination Procedure for the In Core Algorithm.

6.3.8 inCore Accuracy

The solution vectors x computed on the CBE were compared to those computed on Matlab for the first 1506 Newton steps. Matlab computed $\mathbf{x}_{Matlab}$ by:

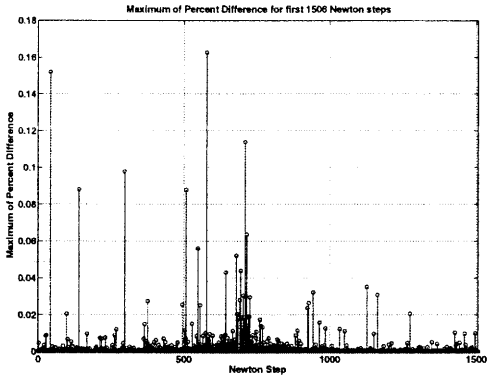
$$\mathbf{x}_{Matlab} = \mathbf{J}_{CBE} \backslash \mathbf{r}_{CBE} \tag{6.2}$$

Where $\mathbf{J}_{CBE}$ and $\mathbf{r}_{CBE}$ were the Jacobian and residuals produced by our 2-D battery simulation on the CBE.

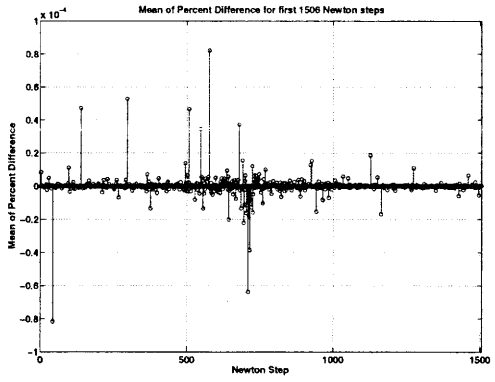
$$\%_{diff_{\mathbf{x}}} = \frac{(\mathbf{x}_{CBE} - \mathbf{x}_{Matlab})}{\mathbf{x}_{CBE}} \tag{6.3}$$

The max percent differences and average percent differences are plotted in fig. 6-19(a) and 6-19(b). Fig. 6-19(a) shows a number of large spikes, however, most solutions are similar between those of Matlab and the inCore solver. Since Matlab partial pivots and our solver does not partial pivot, this seems to indicate that partial pivoting is not really required for this system. The matrix was tested for diagonal dominance, but it was found to not be diagonally dominant.

Furthermore, recalling that our solvers do not partial pivot, the infrequency of large percent differences indicates that the overhead of partial pivoting is not necessary for this system. Instead of doing all the overhead required by partial pivoting, an occasional extra Newton iteration is accepted.



(a) Max percent difference between Matlab computed x and that computed on the Cell Broadband Engine.



(b) Mean percent difference between Matlab computed x and that computed on the Cell Broadband Engine.

Figure 6-19.

6.3.9 inCore Performance

Fig.6-20 shows the performance of the inCore solver vs the number of SPEs used to perform forward elimination. The matrices used to test the performance of the inCore solver are the same matrices used to test the performance of out of core solver revision 2 with the addition of a few larger matrices. The larger matrices were not tested on out of core solver rev 2 simply because the tests were being conducted on donated computer time. For review, the width N of the matrix was held approximately constant at $N \approx 33500$ while the half band size hbS was varied from 66 all the way to 462.

It is immediately obvious from fig. 6-20 that the inCore solver represents a tremendous improvement in performance over both of the out of core solvers. The largest matrix tested on out of core solver rev 2 had a half band size of 282 doubles. Out of core solver rev 2 achieved 0.56Gflops on this matrix. By comparison, the inCore solver was able to achieve 2.6Gflops. This is a factor of about 4.6x improvement in performance! By increasing the half band size to 462 doubles, a performance of 3.2Gflops was obtained. This is 5.7x faster than possible with the out of core solvers!

The inCore solver performance is a strong function of problem size and number of SPEs used. Holding the problem size steady and adding SPEs decreases the compute time/SPE synchronization ratio. At some point, the compute time becomes negligible, and the SPEs spend all their time waiting to synchronize and waiting for memory accesses.

While the performance of the solver seems to be directly tied to the band size, the inCore algorithm does not appear to have the same band size dependent performance spiking as was demonstrated by out of core solver rev 2 in fig. 6-15. This is because most of the memory accesses are well hidden by the inCore solver.

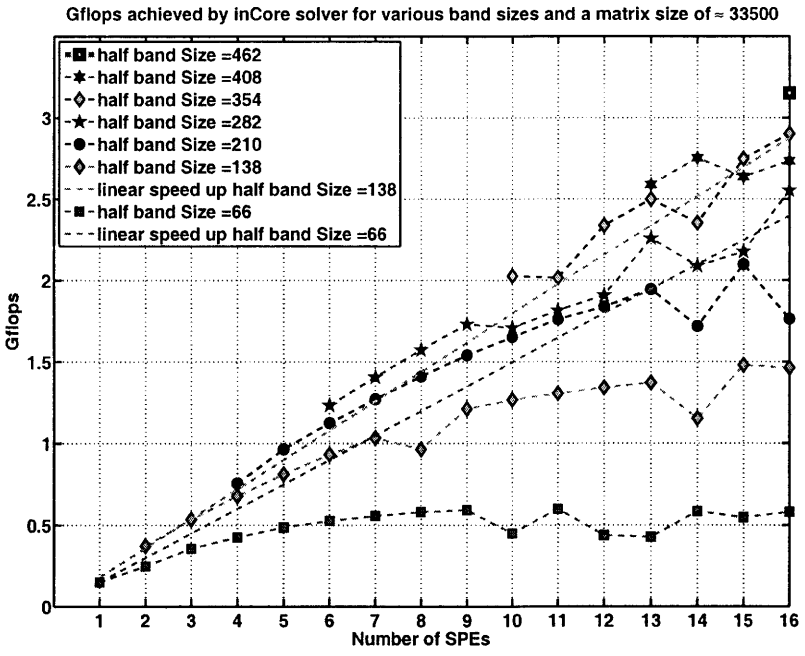


Figure 6-20. The inCore algorithm achieves approximately 11% of theoretical 16 SPE processor performance of 28Gflops.

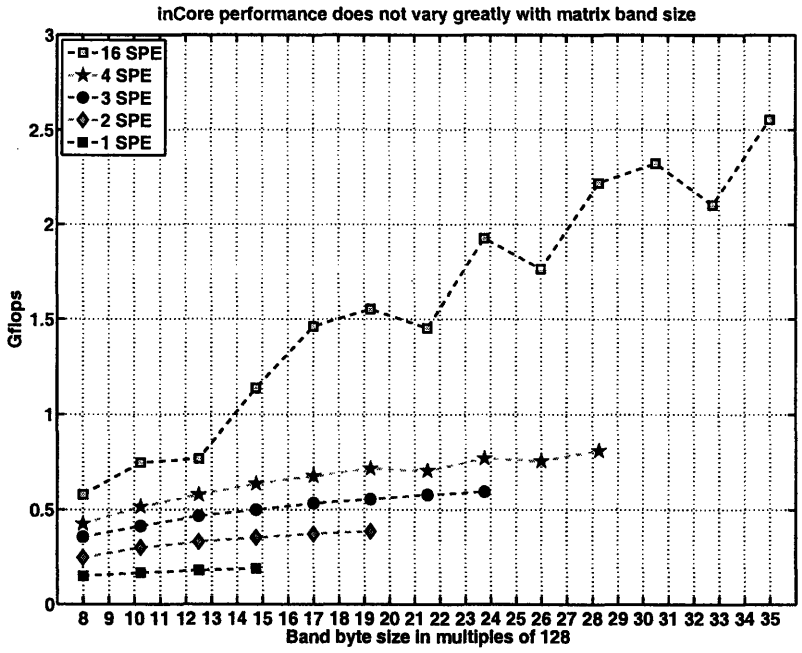


Figure 6-21. The inCore algorithm is much less susceptible to performance fluctuation due to matrix band size than the out of core algorithm.

6.4 Fault Tolerant Parallel Banded LU Algorithm

With Toshiba's announcement of a four SPE variant of the Cell Broadband Engine that it intends to embed into its next generation television systems [64], one can see how today's multicore processors will become tomorrow's embedded processors. One place where multicore processors will likely be embedded in the future is in automotive applications. There are over six million new passenger cars added to US roads every year [65]. Assuming the number of processor cores stays between four and eight, if all of the cores from these machines were assembled into systems of 100,000+ cores, one could make between 240 and 480 new 100,000+ core machines every year.

The automobile's harsh operating conditions will certainly lead to as many core failures as would be experienced in the nice air conditioned homes in which most supercomputers live. That is to say that if a 100,000+ core installed system can expect to experience a core failure every hour, then 100,000+ different embedded systems with multicore processors can expect to experience at least the same number of core failures in the same amount of time. By our rough estimate, that would be somewhere around 360 cars that would fail in some way or another every hour. Therefore, as applications like drive by wire, brake by wire, and monitoring of electrical systems in electric vehicles and hybrid electric vehicles become increasingly common, fault tolerance in these environments will become increasingly important.

Another place where fault tolerant algorithms might be of interest are in aircraft and space applications. It has been well known for a long time that cosmic rays can "induce Soft Errors in integrated circuits and breakdown of power devices. [66]" As transistors get smaller this effect will become more significant. The issue of cosmic rays has become so important that Intel has just been granted two patents for chip level devices that can detect strikes by cosmic rays [67, 68].

A strike by a cosmic ray is a very localized event and can knock out a single core on a multicore chip.

If the chip is in a spacecraft, it would be impossible to replace; therefore, it would be important for the algorithm running on the chip to be fault tolerant so as to be able to continue operation.

More insidious than a core erasure failure would be if the cosmic ray simply created a soft error. That is that it changed some data somehow. This problem can also be avoided by the use of the fault tolerance method combined with a cosmic ray sensor like those recently patented by Intel. It might work something like this, once a cosmic ray strike is detected, all data from the area of the strike is considered faulty. The system declares the effected cores as faulty, and takes them off line. Then the cores are tested for integrity and once they pass the tests, they are restarted. This process of removing a core and then restarting it is the same as multitasking which is discussed in section 6.5

6.4.1 Fault Tolerant Algorithms

Langou and Dongarra [69] and Plank and Dongarra [70] have been working on fault tolerant algorithms for installed systems for quite some time now. Langou [69] points out that as systems grow into the hundreds of thousands of processor range, one can expect a processor failure almost hourly. His fault tolerant algorithms correctly identify and cleverly implement solutions to these erasure failures for single systems that contain large sets of interconnected cores and processors.

Langou's algorithms catch erasure errors and return the system to its effective running state within a matter of moments. However, his systems require $2n$ extra cores for every n^2 original cores [71]. In a large scale system this makes sense in that as the number of processors goes to infinity the cost of not using the $2n$ for computation goes to zero relative to the original n^2 processors. On smaller systems, however, the cost is tremendous. For example, on a 64 processor machine, 16 processors would have to be dedicated to fault tolerance. This is 25% of the total system performance. Therefore, instead of having

a system where backup processors are waiting to operate when there is a fault, on smaller systems, it might be more practical to have all functional processors working and upon a failure, redistribute the load among the remaining working processors. This would allow the system to always operate at its maximum albeit diminished performance.

6.4.2 Implementation

The synchronization step is the key to the mechanics of the fault tolerant algorithm. In the non-fault tolerant implementation of the LU algorithm, the PPE counts the number of SPEs that checked in. When the number of SPEs that have checked in equals the number of working SPEs, the PPE individually tells each of the SPEs that they may begin working again.

In the fault tolerant implementation of the solver, not only does the PPE count the number of SPEs that check in, but it also keeps track of *which* SPEs have checked in. If one or more of the SPEs fails to check in within a give period of time, the algorithm declares those SPEs as failed, kills their threads, and redistributes their load to the remaining working SPEs which then continue with the forward elimination.

While this sounds straight forward, the difficulty lies in the fact that the working SPEs store all the rows they are presently working on in their local store. Therefore, if an SPE fails, all modifications that have been done through elimination to those rows are lost.

Since the LU algorithm is being done "in place," the previous base rows needed for elimination have been written over, so it is not possible to go back to the original matrix. Therefore, it is not possible to restart the calculation without first recreating the Jacobian matrix.

There are two obvious solutions to this problem. The first solution is to keep a second copy of the original Jacobian matrix and residual vector handy. This may seem like overkill, but if the matrix is

sparse and stored in sparse form, this might be an acceptable solution. However, if the Jacobian matrix is large, it may not be practical to store a second copy in memory or on disk, which means this method won't work.

A second solution is to use a form of checkpointing as suggested in [70] where periodic snapshots of the matrix called checkpoints are stored in memory. This is a perfectly acceptable solution even though it requires rolling back the calculation to the most recent checkpoint. By using the enormous bandwidth of the CBE it is possible to implement continuous in place checkpointing. This eliminates the need to repeat any calculations.

To implement continuous checkpointing, each SPE sends the row that it has just performed elimination on out to main memory. This write to memory can be hidden by the elimination operation on the next row. Furthermore, once elimination has been performed on a row, the row is tagged with the number of the base row that has just been used to performed the elimination. This extra tag is called the *ebr* field which is short of "*e*liminated by *b*ase *r*ow."

An example of this tagging and exporting process can be seen in fig. 6-22. This diagram shows 20 rows that have been divided among four SPEs. The present base row is row number 3. Rows with red dots to their right are the rows that the owning SPE is presently performing elimination on. Rows with arrows to their right are the rows that the SPE is presently writing back to XDR memory.

6.4.3 Anatomy of an SPE Failure

Figure 6-23 shows the anatomy of a single SPE failure. The elimination sweep begins with row zero as the base row. All of the SPEs successfully complete elimination on their stored rows and successfully write those rows back to XDR memory. While working on the next set of rows, spe2 fails and does

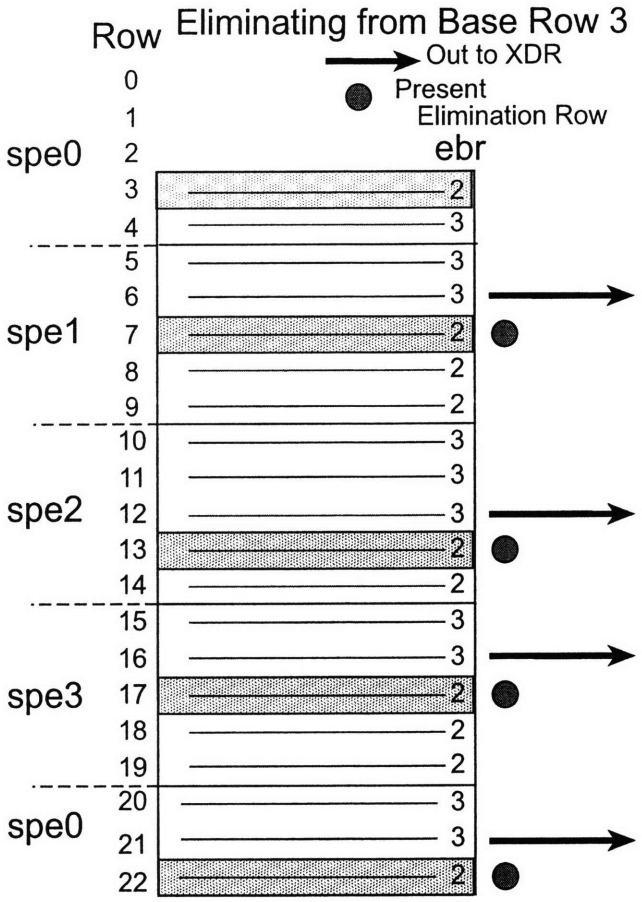


Figure 6-22. In addition to the data movements necessary for the inCore LU algorithm, the fault tolerant algorithm has each SPE send each row that it has completed elimination on to XDR system memory. Further more, once an SPE completes elimination on a row, and just before sending that row to XDR system memory, the SPE updates the row's *ebr* field.

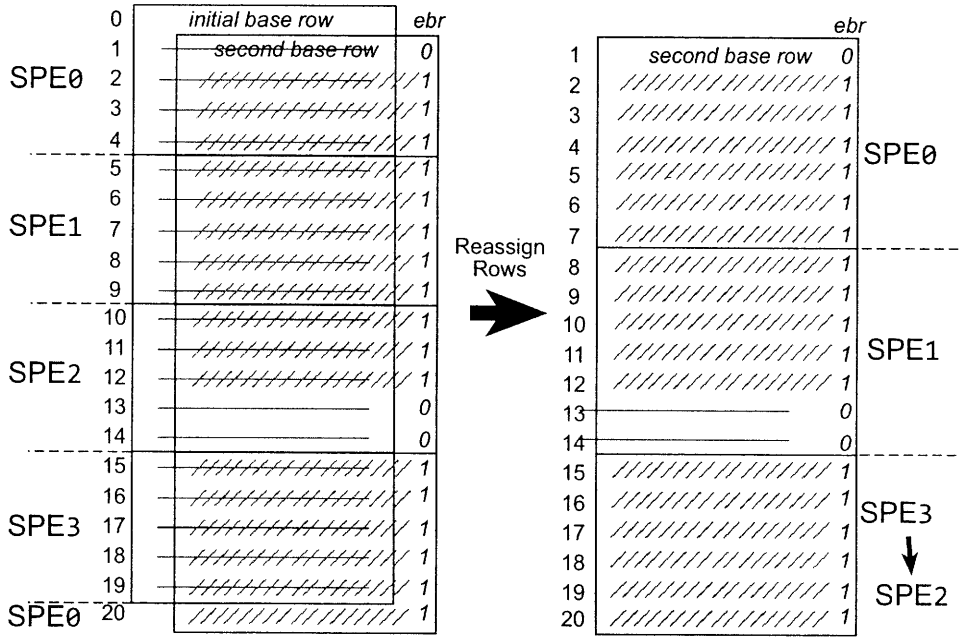


Figure 6-23. This figure shows 20 rows divided among 4 SPEs. All SPEs successfully complete their rows in the first work block. However, on the second work block, spe2 fails, does not complete rows 13 and 14, and does not check in. Therefore, the algorithm redistributes spe2's rows among the remaining 3 SPEs. These SPEs then restart computing from the beginning of the second work block. Before operating on a row, each SPE checks that rows *ebr* to see if it is less than the *pbr*. If the *ebr* is less than the *pbr*, elimination is performed. Otherwise, no operations are performed on that row and the SPE continues on to the next row. Furthermore, the SPEs are renumbered so that they always range from zero to *numWorkingSPEs* where *numWorkingSPEs*

not complete elimination on rows 13 and 14. Therefore, it does not update the *ebr* field for those rows. The synchronization barrier detects this failure and redistributes spe2's rows among the three remaining SPEs. It also reassigns names to the SPEs so that the SPEs are always numbered from zero to 'NUMworkingSPEs'. This is done because there are for loops in the code that look from zero to a max 'NUMworkingSPEs'. This small readjustment helps performance because the PPE only looks for SPEs in the space of working SPEs and not in the space of all SPEs. Other implementations, say with linked lists, might be able to perform the same functions without the renumbering, but these alternative implementations have yet to be explored.

Continuing with the right hand side of Figure 6-23, the computation resumes from the base row at which the fault occurred, which, in this example, is row 1. Each SPE would try to perform elimination on all of its stored rows (except the base row). Since most of the row's *ebr* fields match that of the present base row, no elimination is performed on those rows. Only rows 13 and 14 whose *ebr* field is less than the present base row have elimination performed on them.

The addition of the *ebr* field seems to degrade performance by about 10% as compared to the non-fault tolerant implementation of the algorithm. This is because each SPE always checks the *ebr* field before performing elimination. This testing creates a conditional jump which, as pointed out in section 6.2, is very hard on the SPE's performance. There are certainly ways to code around this problem, but as of this writing, they have not been explored. Furthermore, it may be possible to do away with the *ebr* field in future implementations.

6.4.4 Example, multiple single SPE failures

Figure 6-24 shows the fault tolerant algorithm in action. Here, 6 SPEs start working on a 4096x4096 matrix. Every 8th Newton step, an SPE fails when the LU sweep gets to base row 2118. Each time the system detects a failure, it successfully redistributes the workload and continues the solve. During the intervals of "Normal Computing", the system is collecting timing data and using that timing data to calculate the mean and standard deviation of the total calculation time. An SPE is declared 'failed' when it takes more than 6 standard deviations than the mean check-in time.

It can be noted that the final 2 SPE compute time of 0.81 seconds is not exactly three times that of the original 6 SPE compute time of 0.3 seconds. This comes from the fact that the speedup of adding four processors to the original two is not linear. This is, in part, due to the back solve being computed

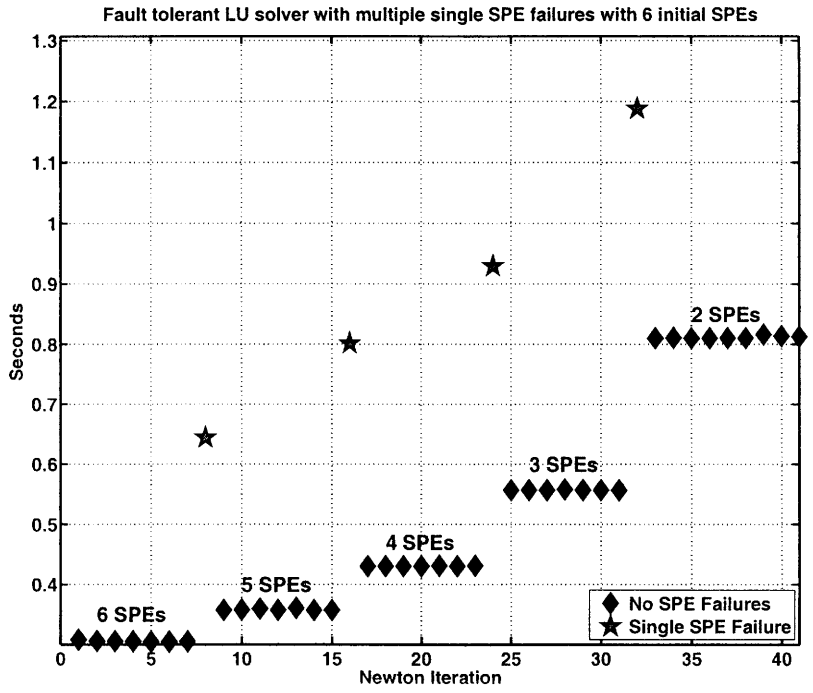


Figure 6-24. This diagram shows that the fault tolerance built into the LU algorithm is capable of handing multiple individual SPE failures.

exclusively on the PPE as it adds a fixed amount of time to the solve independent of the original number of SPEs used.

The fault tolerance does not introduce additional error into the calculations, fig. 6-25. Again, four individual SPE failures occur at different times. The failure times are marked by vertical dashed green lines. The failures occur at time steps: 6,14,32, and 48. The data produced under fault conditions, shown with red stars, exactly matches the data produced when no fault occurs.

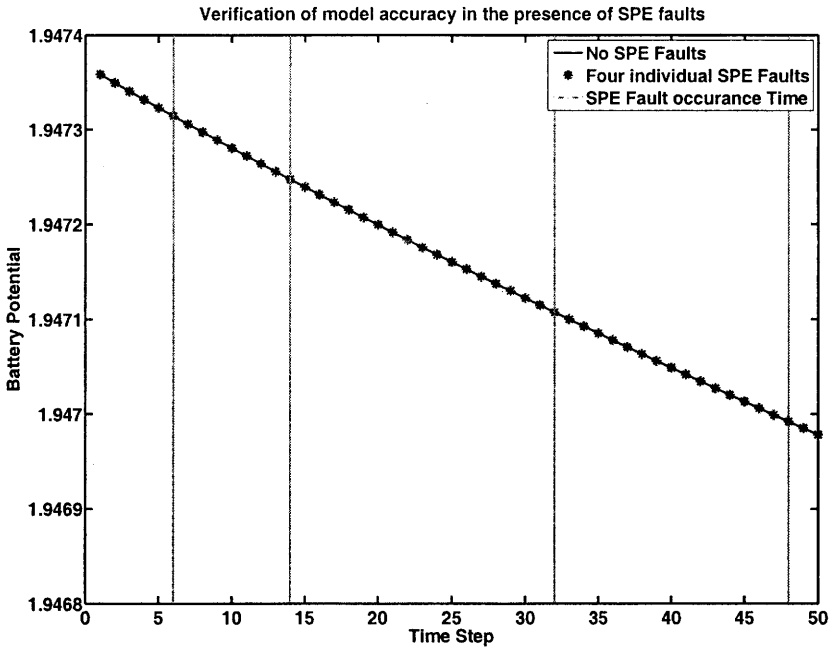


Figure 6-25. This diagram shows that the fault tolerance built into the LU algorithm does not lead to any numerical inaccuracies. SPE erasure failures at time steps 6, 14, 32, and 48.

6.5 Preemptive/Cooperative multitasking environment

6.5.1 Introduction

Langou uses his fault tolerance scheme exclusively for fault tolerance. That leaves the problem of multitasking. It is generally agreed that preemptive multitasking is the most costly form of context switching for these multicore environments. The cost of swapping data in and out of cache many times a second as is done in today's single core machines is too great. Instead, it is recommended that *application-yielding* context switching be used [52, p.351]. But the problem with application yielding context switching, also known as cooperative multitasking, is that sometimes certain applications don't yield. Therefore, a method is needed to force cooperation while not context switching many times each second.

6.5.2 Multitasking via Fault Tolerance

Being able to dynamically redistribute the compute load among SPEs can be used as the basis for a preemptive/cooperative multitasking environment for multicore systems. The multitasking environment is preemptive in that a system scheduler can preempt any process running on any core at any time. The multitasking environment is cooperative in that all processes are fault tolerant in the manner described in the previous section and therefore can dynamically readjust themselves to a new number of cores without disturbing the stability of the entire system, thereby cooperating with the system. The environment is multitasking in that it can now truly run more than one process at any give point in time.

Presently, in large multiprocessor systems, once a process claims a certain number of processors or cores, that process holds onto those cores until the process exits. The problem with this is that there is no way to determine if a program will ever actually exit. This leads to a scheduling problem in that it is very difficult to schedule core usage when it is impossible to determine when cores will be free. To

get around this problem, usually a limited number of cores less than the maximum number of cores are allocated to a particular process. The remaining cores are kept in reserve just in case other users might need the system in the future. Now, if the algorithms were fault tolerant, then the entire system could be allocated to a particular process and cores could be reclaimed by a system scheduler from that process as needed.

The way this works is as follows, an original process is started on all system cores. At some time in the future, another process cames along and requests a certain number of cores. Even though all cores are presently being used by the original process, the system scheduler can get some cores for the new process by first creating erasure failures on some of the original process' cores. This will kick the first process off that set of cores. The scheduler can then begin running the new process on the freed-up cores. Since the original process is fault tolerant, it would detect the erasure faults and re-adjust itself to the new number of cores and continue computing albeit at a slower pace.

Figure 6-26 shows the PPE reclaiming resources from a fault tolerant process. In this example, the LU algorithm starts with all six available SPEs. At Newton step 6, while the system is performing elimination from base row forty, the PPE kills four SPE threads. These four SPEs do not check in on time, so the synchronization barrier redistributes the work between the remaining two SPEs. Those two SPEs continue computation from base row forty and complete the solve.

6.6 Future Work

As stated earlier, the Gaussian elimination solvers presented here do not partial-pivot. It is well know that if a matrix is diagonal dominant or if the matrix is symmetric positive definite, then partial pivoting is never needed. Even if the matrix does not fit into one of those two special categories, sometimes

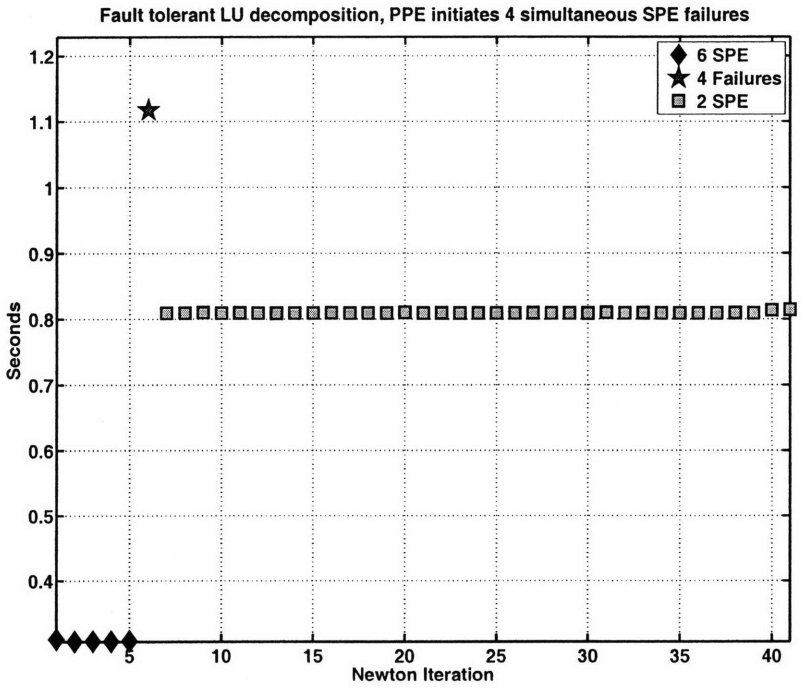


Figure 6-26. This figure shows that the fault tolerant LU algorithm is capable of dealing with multiple simulatneous SPE failures. Furthermore, the failures in this figure were caused by the PPE killing four SPE threads.

partial pivoting is still not required. However, if it may be needed in other situations. Therefore, future work will have to address and incorporate the issue of partial pivoting.

The second half of the multitasking environment, the process of dynamically creating threads and then synchronizing those threads with the original process threads, is a bit more difficult, however, the basic idea remains the same. Further work needs to be done before the results with that half of the multitasking environment are released.

Chapter 7

Conclusions

This thesis has shown that the two level model presented in the introduction is a reasonable model to start with when trying to incorporate a state-of-health metric into an embedded battery condition monitoring system. Furthermore, it has shown that the physical phenomena that cause the uneven usage of the electrodes are well understood and can be reproduced on a computer. It has shown that while the computer simulations produce behavior like that of a battery, because of an inability to systematically model the morphology of the electrode/electrolyte interface surface area and because the models are poorly conditioned, one should not expect more than simple behavior out of the models. This, however, does not mean that the models are without merit or use as they are, for the most part, solidly grounded in the physics of the lead-acid cell. By manipulating battery parameters and seeing how those changes effect the battery behavior, one can get a feel for what physical parameters are important for one's application.

Finally, this thesis explored the computational side of battery modeling. In particular, it explored the new Cell Broadband Engine. It showed that even though the new multicore processors make claims of

multi-Gflop performance, it is the way that the algorithm is implemented that finally determines how much performance can be gotten out of the chip. It also showed that it is vital to try to take advantage of the unique features of the new multicore processors when trying to program them, and it showed how features that can be taken for granted on serial processors can no longer be taken for granted on multicore machines. Instead, new techniques will have to be developed to tackle old problems.

The path for future work is obvious. Having developed a good understanding of the physics that drives the two-level model, and having developed a good understanding of what kind of system we could expect to implement that two-level model on, it would be interesting to actually develop the two-level model. When implemented, however, the model would have to be somewhat different than just a two level version of the 2-D model developed in this thesis as it would have to avoid the interfacial surface area problem if it were to have any chance of being robust enough to be useful.

Appendix A

Two-volume Model Code

```
% James Geraci
% July 21st, 2004
% Single Cell Battery Model
% Complete Model Using Direct Method
% Updated March 28, 2006
% Updated May 24, 2006
% Used for Thesis Data April 30, 2008
tic
t= cputime;
% Cell Parameters
phi_res = 0;
c_res = 0.0049;

tnp = 0.72          % unitless transfer coefficient
N= 2;
%D = 1e-2;%3.02e-5        % Diffusion Coefficient of Electrolyte cm^2/s
D = 3.02e-5          % Diffusion Coefficient of Electrolyte cm^2/s
%D = 0;
Kappa = 0.79        % Siemens/cm
Upb_0 = -.356         % Standard Pb Electrode Potential
Upbx_0 = 1.685        % Standard Pbx Electrode Potential
U_0 = Upbx_0 - Upb_0    % Standard Cell Potential
F = 96487           % Coulomb/eq
T = 298             % Kelvin
R = 8.3143          % J/molK
I = 0.05          % Amps Drawn from the battery
dt = 0.1            % seconds  Time Step size
Pb_c = 0.0049     % mol/cm^3  Initial Concentration in Pb area
```

```
Pbx_c = 0.0049     % mol/cm^3  Initial Concentration in Pbx area
i_Pb = 1e-2        % A/cm^2 exchange current density at Pb electrode
i_Pbx = 1e-2       % A/cm^2 exchange current density at Pb_x electrode
Ve = 45.0          % cm^3/mol  Molar Volume of Electrolyte
Vo = 17.5          % cm^3/mol  Molar Volume of Water

Upbx = Upbx_0;% - (((3-2*tnp)*(R*T))/(N*F))*log(c_res);
Upb = Upb_0;% - (((1-2*tnp)*(R*T))/(N*F))*log(c_res);
U_eq = Upbx-Upb
duration = 10000;
offpoint = 110000;
margin = 1e-14;
% Dimensions of Cell
Pb_dx = 0.03           % cm
Pb_dxsqrd = Pb_dx*Pb_dx;
Pb_dy = 10.0           % cm
Pb_dy_s = 5.0          % cm
Pb_dy_l = Pb_dy-Pb_dy_s    % cm
Pb_dz = 7.5            % cm

Pbx_dx = 0.03          % cm
Pbx_dxsqrd = Pbx_dx*Pbx_dx;
Pbx_dy = 10.0          % cm
Pbx_dy_s = 5.0         % cm
Pbx_dy_l = Pb_dy-Pb_dy_s    % cm
Pbx_dz = 7.5           % cm


Pb_asl = 1.55/2          % unitless
Pb_als = 0.45/2          % unitless
%Pb_asl = 0.5            % unitless
%Pb_als = 0.5            % unitless


Pbx_als = 1.15/2         % unitless
Pbx_asl = 0.85/2         % unitless
%Pbx_als = 0.5           % unitless
%Pbx_asl = 0.5           % unitless


PbSO4_rho = 6.39       % g/cm^3 PbSo4
Pbx_rho = 9.79         % g/cm^3 Pb02
Pb_rho = 11.34         % g/cm^3 Pb

PbSO4_MW = 303.2626    % g PbSo4
Pb_MW = 207.2          % g Pb
Pb_Vf = (PbSO4_MW/PbSO4_rho - Pb_MW/Pb_rho)
```

```
PbSO4_MW = 303.2626    % g PbSo4
Pbx_MW = 239.1988    % g Pb02
Pbx_Vf = (PbSO4_MW/PbSO4_rho - Pbx_MW/Pbx_rho)

Pb_counter = 0;
Pbx_counter = 0;
Pb_scale = (1-2*tnp)/2;
Pbx_scale = (3-2*tnp)/2;
ols_scale = 1-2*tnp;

Pb_phi_s = -.356;
Pb_phi_l = 0;

Pbx_eps = 0.5;
Pb_eps = 0.5;
Pbx_phi_s = 1.685;
Pbx_phi_l = 0;

Pbx_epsmax = 0.65;
Pbx_epsmin = 0.3;
Pb_epsmax = 0.65;
Pb_epsmin = 0.3;
Pb_Amax = 1/Pb_dy;
Pbx_Amax = 1/Pbx_dy;
Pb_Ah = (i_Pb*Pb_Amax)/(F*c_res)   % nnnn/cm^2
Pbx_Ah = (i_Pbx*Pbx_Amax)/(F*c_res)   % nnnn/cm^2


% Start Dynamic part of simluation

Pb_N_l = I/(F*Pb_dz*Pb_dy_l);
Pbx_N_l = I/(F*Pbx_dz*Pbx_dy_l);


Pb_cv = Pb_c;
Pb_gsl = (Pb_asl*N*F)/(R*T);
Pb_gls = (Pb_als*N*F)/(R*T);

      Pb_cv = Pb_c;
      Pb_phi_lv = Pb_phi_l;
      Pb_phi_sv = Pb_phi_s;
      Pb_Nv = Pb_N_l;
      Pb_epsv = Pb_eps;


Pbx_cv = Pbx_c;
```

```
Pbx_gsl = (Pbx_asl*N*F)/(R*T);
Pbx_gls = (Pbx_als*N*F)/(R*T);

     Pbx_cv = Pbx_c;
     Pbx_phi_lv = Pbx_phi_l;
     Pbx_phi_sv = Pbx_phi_s;
     Pbx_Nv = Pbx_N_l;
     Pbx_epsv = Pbx_eps;
     SumV = [];
     SumNf = [];
     SumL = [];
 while Pbx_counter < duration & Pbx_c > 3e-3 & Pbx_c < 8e-3,
     if I > 0
%     Pbx_a  = 0;
%     Pbx_b = 0;

     Pbx_a  = 1;%;0.5;
     Pbx_b = 0;
     else
 %    Pbx_a  = 0;%0.5;
 %    Pbx_b = 0;
     Pbx_a  = 0.1;%0.5;
     Pbx_b = 1;

     end
     Pbx_counter = Pbx_counter + 1;
     Pbx_I = I/(F*Pbx_dx*Pbx_dy*Pbx_dz);
     Pbx_eps_dt = -(Pb_Vf*Pbx_I)/2;
     Pbx_eps = Pbx_eps + dt*Pbx_eps_dt;
     Pbx_c = Pbx_c + dt*(Pbx_c/Pbx_eps)*Pbx_eps_dt + (Pbx_eps*D*dt*(c_res
         - Pbx_c))/Pbx_dxsqrd - dt*Pbx_scale*Pbx_I;

      Pbx_N_l = I/(F*Pbx_eps*Pbx_dy*Pbx_dz);
           A = 1;
    % Pbx_phi_l = -(Pbx_N_l*F*Pbx_dx)/Kappa + phi_res - ols_scale*((R*T)/F)
       *(c_res/Pbx_c-1);
     Pbx_phi_l = -(Pbx_N_l*F*Pbx_dx)/Kappa + phi_res - ols_scale*((R*T)/F)
        *(log(c_res)-log(Pbx_c));
    Pbx_error = 1;
    Pbx_step = 0.1;
    while abs(Pbx_error) >= margin,
        Pbx_step = Pbx_step*.999;
         if Pbx_error  > 0
           Pbx_phi_s = Pbx_phi_s - Pbx_step;
         elseif Pbx_error < 0
            Pbx_phi_s = Pbx_phi_s + Pbx_step;
     end
```

```
        Pbx_Nf =  I/(F*Pbx_dx*Pbx_dy*Pbx_dz);
       A = -exp(Pbx_gls*(Pbx_phi_l-Pbx_phi_s + Upbx));
       B = exp(Pbx_gsl*(Pbx_phi_s-Pbx_phi_l - Upbx));

      Pbx_error = Pbx_Nf + ((Pbx_eps-Pbx_epsmin)/(Pbx_epsmax-Pbx_epsmin))^
         Pbx_a...
          *((Pbx_epsmax-Pbx_eps)/(Pbx_epsmax - Pbx_epsmin))^Pbx_b*Pbx_Ah*
             Pbx_c*(A + B);
    end
       SumL = [SumL; Pbx_Ah*Pbx_c*(A+B)];
       SumNf = [SumNf; Pbx_Nf];
       SumV = [SumV; A+B];
   Pbx_cv = [Pbx_cv Pbx_c];
Pbx_phi_lv = [Pbx_phi_lv Pbx_phi_l];
     Pbx_phi_sv = [Pbx_phi_sv Pbx_phi_s];
     Pbx_Nv = [Pbx_Nv Pbx_N_l];
     Pbx_epsv = [Pbx_epsv Pbx_eps];
 end

  while Pb_counter < duration & Pb_c > 3e-3 & Pb_c < 8e-3,
     if I > 0
      Pb_a  = 1;%0.5;
      Pb_b = 0;
     % Pb_a  = 0;
     % Pb_b = 0;
     else
      Pb_a  = 0.1;
      Pb_b = 1;
     % Pb_a  = 0;
     % Pb_b = 0;
     end
     Pb_counter = Pb_counter + 1;
     Pb_I = I/(F*Pb_dx*Pb_dy*Pb_dz);
     Pb_eps_dt = -(Pb_Vf*Pb_I)/2;
        Pb_eps = Pb_eps  + dt*Pb_eps_dt;
       Pb_c = Pb_c + dt*(Pb_c/Pb_eps)*Pb_eps_dt +(Pb_eps*D*dt*(c_res-Pb_c))
          /Pb_dxsqrd + dt*Pb_scale*Pb_I;

        Pb_N_l = I/(F*Pb_eps*Pb_dy*Pb_dz);

      % Solve for phi_l using OLS
%        Pb_phi_l = (Pb_N_l*F*Pb_dx)/Kappa + phi_res + ols_scale*((R*T)/F)
   *(Pb_c/c_res -1);
        Pb_phi_l = (Pb_N_l*F*Pb_dx)/Kappa + phi_res + ols_scale*((R*T)/F)*(
           log(Pb_c)-log(c_res));

        Pb_error = 1;
```

```
        Pb_step = 0.1;
    while abs(Pb_error) >= margin,
       Pb_step = Pb_step*0.999;
         if Pb_error < 0
           Pb_phi_s = Pb_phi_s - Pb_step;
         elseif Pb_error > 0
            Pb_phi_s = Pb_phi_s + Pb_step;
     end
       Pb_Nf =  I/(F*Pb_dx*Pb_dy*Pb_dz);%(Pbx_N_l)*(Pbx_dy_l)*Pbx_dz;
       A = -exp(Pb_gls*(Pb_phi_l-Pb_phi_s+Upb));
       B = exp(Pb_gsl*(Pb_phi_s-Pb_phi_l-Upb));
       Pb_error = Pb_Nf - ((Pb_eps-Pb_epsmin)/(Pb_epsmax-Pb_epsmin))^Pb_a
           ...
           *((Pb_epsmax-Pb_eps)/(Pb_epsmax-Pb_epsmin))^Pb_b*Pb_Ah*Pb_c*(A
               + B);
    end

   Pb_cv = [Pb_cv Pb_c];
Pb_phi_lv = [Pb_phi_lv Pb_phi_l];
     Pb_phi_sv = [Pb_phi_sv Pb_phi_s];
     Pb_Nv = [Pb_Nv Pb_N_l];
     Pb_epsv = [Pb_epsv Pb_eps];
end



TIME_USED = cputime-t
time1 = [0:dt:duration*dt];
time = [dt:dt:duration*dt];


figure(2)
plot(time,Pb_Nv(2:length(Pb_Nv)))
titleString = 'Lead Electrode Liquid Flux Density vs. Time';
title(titleString);
xlabel('Seconds')
grid on
zoom on
hold on
if I > 0
batteryprinter(strcat('discharging',titleString));
else
batteryprinter(strcat('charging',titleString));
end
%print

figure(3)
```

```
plot(time,Pb_phi_lv(2:length(Pb_phi_lv)))
%plot(Pb_phi_lv)
titleString = 'Lead Electrode Liquid Potential vs. Time';
title(titleString);
xlabel('Seconds')
grid on
zoom on
hold on
if I > 0
batteryprinter(strcat('discharging',titleString));
else
batteryprinter(strcat('charging',titleString));
end
%print

figure(4)
plot(time,Pb_phi_sv(2:length(Pb_phi_sv)))
%plot(Pb_phi_sv)
titleString = 'Lead Electrode Solid Potential vs. Time';
title(titleString);
xlabel('Seconds')
grid on
zoom on
hold on
if I > 0
batteryprinter(strcat('discharging',titleString));
else
batteryprinter(strcat('charging',titleString));
end
%print


figure(5)
Pb_overpotential = Pb_phi_sv(2:length(Pb_phi_sv)) - Pb_phi_lv(2:length(
   Pb_phi_lv));
plot(time,Pb_overpotential)
%plot(Pb_overpotential)
titleString = 'Lead Electrode Overpotential vs. Time';
title(titleString);
xlabel('Seconds')
grid on
zoom on
hold on
if I > 0
batteryprinter(strcat('discharging',titleString));
else
batteryprinter(strcat('charging',titleString));
```

```
end
%print

figure(6)
plot(time , Pb_epsv(2:length(Pb_epsv)))
%plot(Pb_dy_sv)
titleString = 'Lead Electrode Porosity vs. Time';
title(titleString);
xlabel('Seconds')
grid on
zoom on
hold on
if I > 0
batteryprinter(strcat('discharging',titleString));
else
batteryprinter(strcat('charging',titleString));
end
%print
%%
figure(7)
plot(time , Pb_cv(2:length(Pb_cv)),'LineWidth',4)
title(titleString , 'LineWidth', 4, 'FontSize', 22, 'FontWeight', 'b');
set(gca , 'LineWidth', 4, 'FontSize', 22, 'FontWeight', 'b');
%plot(Pb_cv)
titleString = 'Lead Electrode Concentration vs. Time';
title(titleString);
xlabel('Seconds')
ylabel('Concentration of H_2SO_4 (mol/cm^3)')
grid on
zoom on
hold on
if I > 0
ymin = 0.00488;
ymax = 0.0049;
else
    ymin = 0.0049;
    ymax = 0.00492;
end
v = axis();
v(3) = ymin;
v(4) = ymax;
axis(v);
y = [ymin:(ymax-ymin)/5:ymax];
p = 6;
set(gca,'YTick',y,'YTickLabel',{num2str(y(1),p);num2str(y(2),p);num2str(y
   (3),p);num2str(y(4),p);num2str(y(5),p);num2str(y(6),p)})
```

```
if I > 0
batteryprinter(strcat('discharging',titleString));
else
batteryprinter(strcat('charging',titleString));
end
%print

%%
figure(12)
plot(time,Pbx_Nv(2:length(Pbx_Nv)))
titleString = 'Lead Dioxide Electrode Liquid Flux Density vs. Time';
title(titleString);
xlabel('Seconds')
grid on
zoom on
hold on
if I > 0
batteryprinter(strcat('discharging',titleString));
else
batteryprinter(strcat('charging',titleString));
end
%print

figure(13)
plot(time,Pbx_phi_lv(2:length(Pbx_phi_lv)))
%plot(Pbx_phi_lv)
titleString = 'Lead Dioxide Electrode Liquid Potential vs. Time';
title(titleString);
xlabel('Seconds')
grid on
zoom on
hold on
if I > 0
batteryprinter(strcat('discharging',titleString));
else
batteryprinter(strcat('charging',titleString));
end
%print

figure(14)
plot(time,Pbx_phi_sv(2:length(Pbx_phi_sv)))
%plot(Pbx_phi_sv)
titleString = 'Lead Dioxide Electrode Solid Potential vs. Time';
title(titleString);
xlabel('Seconds')
grid on
zoom on
```

```
hold on
if I > 0
batteryprinter(strcat('discharging',titleString));
else
batteryprinter(strcat('charging',titleString));
end
%print

figure(15)
%Pbx_overpotential = Pbx_phi_sv(2:length(Pbx_phi_sv)) - Pbx_phi_lv(2:length
    (Pbx_phi_lv));
Pbx_overpotential = Pbx_phi_sv-Pbx_phi_lv;
length(time)
length(Pbx_overpotential)
plot(time,Pbx_overpotential(2:length(Pbx_overpotential)))
%plot(Pbx_overpotential);
titleString = 'Lead Dioxide Electrode Overpotential vs. Time';
title(titleString);
xlabel('Seconds')
grid on
zoom on
hold on
if I > 0
batteryprinter(strcat('discharging',titleString));
else
batteryprinter(strcat('charging',titleString));
end
%print

figure(16)
plot(time, Pbx_epsv(2:length(Pbx_epsv)))
%plot(Pbx_dy_sv)
titleString = 'Lead Dioxide Electrode Porosity vs. Time'
title(titleString);
xlabel('Seconds')
grid on
zoom on
hold on
if I > 0
batteryprinter(strcat('discharging',titleString));
else
batteryprinter(strcat('charging',titleString));
end
%print
%%
figure(17)
plot(time,Pbx_cv(2:length(Pbx_cv)),'LineWidth',4)
```

```
title(titleString , 'LineWidth' , 4 , 'FontSize' , 22 , 'FontWeight' , 'b');
set(gca , 'LineWidth' , 4 , 'FontSize' , 22 , 'FontWeight' , 'b');
%plot(Pbx_cv)
titleString = 'Lead Dioxide Electrode Concentration vs. Time'
title(titleString);
xlabel('Seconds')
ylabel('Concentration of H_2SO_4 (mol/cm^3)')
grid on
zoom on
hold on
vldx = axis();

if I > 0
ymin = 0.00488;
ymax = 0.0049;
else
    ymin = 0.0049;
    ymax = 0.00492;
end
vldx(3) = ymin;
vldx(4) = ymax;
axis(vldx);
y = [ymin:(ymax-ymin)/5:ymax];
p = 6;
set(gca,'YTick',y,'YTickLabel',{num2str(y(1),p);num2str(y(2),p);num2str(y
   (3),p);num2str(y(4),p);num2str(y(5),p);num2str(y(6),p)})
%%
if I > 0
batteryprinter(strcat('discharging',titleString));
else
batteryprinter(strcat('charging',titleString));
end
%print


cellpotential = Pbx_phi_sv(2:length(Pbx_phi_sv)) -Pb_phi_sv(2:length(
   Pbx_phi_sv));
%%
figure(18)
plot(time,cellpotential,'r', 'LineWidth', 4)
%plot(cellpotential)
title(titleString , 'LineWidth' , 4 , 'FontSize' , 22 , 'FontWeight' , 'b');
set(gca , 'LineWidth' , 4 , 'FontSize' , 22 , 'FontWeight' , 'b');
titleString = 'Cell Potential vs. Time'
title(titleString);
ylabel('Volts')
xlabel('Seconds')
```

```
grid on
zoom on
hold on
if I > 0
batteryprinter(strcat('discharging',titleString));
else
batteryprinter(strcat('charging',titleString));
end
%print
%%
v = axis()

%print
ymin = v(3);
ymax = v(4);
y = [ymin:(ymax-ymin)/5:ymax];
p = 8;
set(gca,'YTick',y,'YTickLabel',{num2str(y(1),p);num2str(y(2),p);num2str(y
    (3),p);num2str(y(4),p);num2str(y(5),p);num2str(y(6),p)})
%%
toc
```

Appendix B

2-D Model Setup Code

Listing B.1. StdAfx.h

```
/* @author James Geraci
 */
// stdafx.h : include file for standard system include files,
// or project specific include files that are used frequently, but
// are changed infrequently
//

#ifndef _StdAfx_H
#define _StdAfx_H


#define WIN32_LEAN_AND_MEAN             // Exclude rarely-used stuff from
   Windows headers
#define UL 0
#define UR 1
#define LL 2
#define LR 3

#define TOP    0
#define RIGHT  1
#define LEFT   2
#define BOTTOM 3

#define qhpc   0
#define qphilc 1
#define qphisc 2
```

```
#define qprc    3
#define qv_yc   4
#define qv_xc   5

#define NUM_CVperPV 4

#include <libspe2.h>
#include <pthread.h>
#include <time.h>
#include <stdio.h>
#include <libmisc.h>
//#include <tchar.h>
//#include <tchar.h>
#include <memory.h>
#include <malloc.h>
#include <math.h>
#include <assert.h>
//#include <emmintrin.h>
//#include "acml.h"
//#include "acml_mv.h"

#include "myGlobals.h"
#include "common.h"
//#include "ctdef.h"
//#include "mat.h"
//#include "matrix.h"
//#include "engine.h"
#include "matrixData.h"
//#include "mySolvers.h"

#include "electrodeChemData.h"
#include "ldxElectrodeChemData.h"
#include "ldElectrodeChemData.h"
#include "electrolyteElectrodeChemData.h"

#include "cellSizeData.h"
#include "volumeSpatialData.h"
#include "electrodeSizeData.h"
#include "volumeChemData.h"
#include "volumeIConditions.h"
#include "PVIConditions.h"
#include "FV.h"
// TODO: reference additional headers your program requires here
#include "ldxElectrode.h"
#include "ldElectrode.h"
#include "Electrolyte.h"
```

```
#include "CV.h"
#include "ULCV.h"
#include "URCV.h"
#include "LLCV.h"
#include "LRCV.h"

#include "PV.h"
//#include "leftPV.h"
//#include "rightPV.h"
#include "OLPV.h"
#include "ORPV.h"
#include "OLULPV.h"
#include "UPV.h"
#include "OLLLPV.h"
#include "ORURPV.h"
#include "ORLRPV.h"


#include "BPV.h"
#endif
```

Listing B.2. StdAfx.cpp

```
// stdafx.cpp : source file that includes just the standard includes
// FVbatterModel.pch will be the pre-compiled header
// stdafx.obj will contain the pre-compiled type information

#include "StdAfx.h"

// TODO: reference any additional headers you need in STDAFX.H
// and not in this file
// left over from Windows world
```

Listing B.3. Electrode.h

```
/* @author James Geraci
 */
#ifndef _ELECTRODE_H
#define _ELECTRODE_H
#include "matrixData.h"
#include "FV.h"
#include "PV.h"
#include "cellSizeData.h"

class Electrode
{
public:
        Electrode(void);
        Electrode(char*,FV** myFV, PV** myPV, matrixData*, electrodeSizeData
           *, electrodeChemData, volumeChemData, volumeIConditions,
           PVIConditions);
public:
        virtual ~Electrode(void);

        FV* returnLeftFVptr(void);
        FV* returnRightFVptr(void);

protected:
        cellSizeData* mycsData;
        int index;

        double myDx;
        double myeps;
        double mydx;
        int numFVColumns;
        int numFVRows;
        int numFVZrows;

        int numPVColumns;
        int numPVRows;
        int numPVZrows;

        double xWidth;
        double yHeight;
        double zDepth;

        double FVvdx;
        double FVvdy;
        double FVvdz;

        double PVvdx;
```

```
        double PVvdy;
        double PVvdz;

        int presFVColumn;
        int presFVRow;

        int presPVColumn;
        int presPVRow;

        int numFVVolumes;
        int numPVVolumes;

        int FVstartColumn;
        int PVstartColumn;

        matrixData * myData;
        volumeSpatialData** vsData;
        volumeSpatialData** PVvsData;
        electrodeSizeData * myeData;
        electrodeChemData myelectrodeChemData;
        volumeChemData myvCData;
        volumeIConditions myiData;
        PVIConditions mypviData;
        FV** myFv;
        PV** myPv;

private:
        void makeLeftElectrode(FV**,PV**);
        void makeRightElectrode(FV**,PV**);
        void makeElectrolyte(FV**,PV**);
};

#endif
```

Listing B.4. Electrode.cpp

```
/*
 *  @author James Geraci
 *  Sets up the battery Cell by creating each electrode and
 *  making each electrode away of the other electrodes.
 */
#include "StdAfx.h"
#include "Electrode.h"
#include <iostream>

Electrode::Electrode(void)
{
}

Electrode::Electrode(char* pos, FV** myFv, PV** myPv, matrixData* myMatrix
    , electrodeSizeData* eData,
                                        electrodeChemData myChem,
                                          volumeChemData myVol,
                                          volumeIConditions iData,
                                          PVIConditions pviData)
{
        myData = myMatrix;
        myeData = eData;
        myelectrodeChemData = myChem;
        myvCData = myVol;
        myiData = iData;
        mypviData = pviData;


        mycsData = eData->mycsData;
        numFVColumns = eData->getnumFVColumns();
        numFVRows = eData->getnumFVRows();
        numFVZrows = eData->getnumFVZrows();

        numFVVolumes = numFVColumns*numFVRows*numFVZrows;

        numPVColumns = eData->getnumPVColumns();
        numPVRows = eData->getnumPVRows();
        numPVZrows = eData->getnumPVZrows();

        numPVVolumes = numPVColumns*numPVRows*numPVZrows;


        yHeight = eData->getyHeight();
        xWidth = eData->getxWidth();
        zDepth = eData->getzDepth();
```

```
FVvdx = (double) xWidth/numFVColumns;
FVvdy = (double) yHeight/numFVRows;
FVvdz = (double) zDepth/numFVZrows;


PVvdx = (double) xWidth/numFVColumns;
PVvdy = (double) yHeight/numFVRows;
PVvdz = (double) zDepth/numFVZrows;

FVstartColumn = myeData->getFVstartColumn();
PVstartColumn = myeData->getPVstartColumn();

try{
        vsData = (volumeSpatialData**) new char [numFVVolumes*
           sizeof(volumeSpatialData*)];
        }
catch(std::bad_alloc){
  std::cout << "Couldn't Allocate vsData array!" << std::endl;
        exit(-1);
        }

for(int counter = 0; counter < numFVVolumes; counter++)
{
        vsData[counter] = NULL;
};

index =0;
presFVColumn = 0;
presFVRow = 0;
while(presFVRow < numFVRows)
{
        while(presFVColumn < numFVColumns)
        {
                try{
                        vsData[index] = new volumeSpatialData(
                           FVvdx,FVvdy,FVvdz,presFVColumn,presFVRow
                           ,0);
                        }
                        catch(std::bad_alloc)
                        {
                        std::cout << "Failed to allocate
                           volumeSpatialData Object number " <<
                           index << std::endl;
                        exit(-1);
                        }
                presFVColumn++;
                index++;
```

```
                }
            presFVColumn = 0;
            presFVRow++;
            }


        if(!strcmp(pos,"ldx"))
        {
                makeLeftElectrode(myFv,myPv);
        }
        else if(!strcmp(pos,"Electrolyte"))
        {
                makeElectrolyte(myFv,myPv);
        }
        else if(!strcmp(pos,"ld"))
        {
                makeRightElectrode(myFv,myPv);
        }
}

Electrode::~Electrode(void)
{
        for(int counter = 0; counter < numFVVolumes; counter++)
        {
          delete vsData[counter];
        }
        for(int counter = 0; counter < numPVVolumes; counter++)
        {
          delete  PVvsData[counter];
        }
        delete [] vsData;
        delete [] PVvsData;
}


void Electrode::makeLeftElectrode(FV** myFv,PV** myPv){
  index = FVstartColumn;
  presFVRow = 0;
  presFVColumn = FVstartColumn;

  while(presFVRow < numFVRows)
    {
      while(presFVColumn < numFVColumns+FVstartColumn)
        {
          try{
            int f = index-(presFVRow*(mycsData->getElectrolyteFVColumns()
                                      + mycsData->getldFVColumns())+
```

```
                                        FVstartColumn);
            myFv[index] =  new FV(vsData[f], myData, myeData, &myvCData, &
                myelectrodeChemData, &myiData);
            f = f;
          }
          catch(std::bad_alloc)
            {
              std::cout << "Cannot allocate FV number " << index << std::
                 endl;
              exit(-1);
            }
          presFVColumn++;
          index++;
        }
      presFVColumn = FVstartColumn;
      presFVRow++;
      index = index + mycsData->getElectrolyteFVColumns() + mycsData->
         getIdFVColumns();
    }

int aa;
int baseIndex;
try{
  PVvsData = (volumeSpatialData**) new char [numPVVolumes*sizeof(
     volumeSpatialData*)];
}
catch(std::bad_alloc){
  std::cout << "Couldn't Allocate vsData array!" << std::endl;
  exit(-1);
}

index = PVstartColumn;
presPVRow = 0;
presPVColumn = PVstartColumn;
while(presPVRow < numPVRows){
  while(presPVColumn < numPVColumns+PVstartColumn){
    try{
      PVvsData[index] =  new volumeSpatialData(PVvdx,PVvdy,PVvdz,
         presPVColumn,presPVRow,0);
    }
    catch(std::bad_alloc)
          {
            std::cout << "Failed to allocate volumeSpatialData Object
               number " << index << std::endl;
            exit(-1);
          }
    presPVColumn++;
```

```
    index++;
  }
  presPVRow++;
      presPVColumn = PVstartColumn;
}

// Now initialize the Property Volumes
// Make the top Row
index = PVstartColumn;
presPVRow = 0;
presPVColumn = PVstartColumn;
baseIndex = numPVColumns+index-1;
try{
  aa = index-(presPVRow*(mycsData->getElectrolytePVColumns()
                         + mycsData->getIdPVColumns())+ PVstartColumn);

  myPv[index] =  new OLULPV(PVvsData[aa],myData,&myvCData,myeData,&
      mypviData);

}
catch(std::bad_alloc)
  {
    std::cout << "Out of Memory!" << std::endl;
    exit(-1);
  }

index++;
// Inner volumes
while(index < baseIndex)
  {
    try{
      aa = index-(presPVRow*(mycsData->getElectrolytePVColumns()
                             + mycsData->getIdPVColumns())+ PVstartColumn
                             );
      myPv[index] =  new UPV(PVvsData[aa],myData,&myvCData,myeData,&
          mypviData);

    }
    catch(std::bad_alloc)
      {
        std::cout << "Out of Memory!" << std::endl;
        exit(-1);
      }
    index++;
  }

// Upper Right Most volume
```

```
  try{
    aa = index-(presPVRow*(mycsData->getElectrolytePVColumns()
                           + mycsData->getIdPVColumns())+ PVstartColumn);
    myPv[index] =  new UPV(PVvsData[aa],myData,&myvCData,myeData,&mypviData
        );
  }
  catch(std::bad_alloc)
    {
      std::cout << "Out of Memory!" << std::endl;
      exit(-1);
    }
        presPVRow++;
        presPVColumn = PVstartColumn;
        index = presPVRow*(mycsData->getIdxPVColumns() + mycsData->
            getElectrolyteFVColumns() + mycsData->getIdFVColumns())
          + PVstartColumn;

        /*************************************************************************
            */
        /************************** Now make Middle Rows of PV Volumes
            ********************************** */
/*************************************************************************************
   */
        while(presPVRow != numPVRows -1){
          baseIndex = numPVColumns+index-1;
          try{
            aa = index-(presPVRow*(mycsData->getElectrolytePVColumns()
                                   + mycsData->getIdPVColumns())+
                                      PVstartColumn);
            myPv[index] =  new OLPV(PVvsData[aa],myData,&myvCData,myeData,&
                mypviData);
        }
          catch(std::bad_alloc)
            {
              std::cout << "Out of Memory!" << std::endl;
              exit(-1);
            }

          index++;

          // Inner volumes
          while(index < baseIndex)
            {
              try{
                aa = index-(presPVRow*(mycsData->getElectrolytePVColumns()
                                       + mycsData->getIdPVColumns())+
                                          PVstartColumn);
```

```
            myPv[index] =  new PV(PVvsData[aa],myData,&myvCData,myeData
                ,&mypviData);
          }
          catch(std::bad_alloc)
            {
              std::cout << "Out of Memory!" << std::endl;
            exit(-1);
            }
          index++;
        }

      //  Right Most volume
      try{
        aa = index-(presPVRow*(mycsData->getElectrolytePVColumns()
                              + mycsData->getIdPVColumns())+
                                PVstartColumn);
        myPv[index] =  new PV(PVvsData[aa],myData,&myvCData,myeData,&
            mypviData);
      }
      catch(std::bad_alloc)
            {
              std::cout << "Out of Memory!" << std::endl;
              exit(-1);
            }
      presPVRow++;
      presPVColumn = PVstartColumn;
      index = presPVRow*(mycsData->getIdxPVColumns() + mycsData->
          getElectrolyteFVColumns() + mycsData->getIdFVColumns())
        + PVstartColumn;
      }


/*************************************************************************
   */
/************************  Now make Bottom Row of PV Volumes
    ********************************** */
/*************************************************************************
   */
      baseIndex = numPVColumns+index-1;
      presPVColumn = PVstartColumn;
      try{
        aa = index-(presPVRow*(mycsData->getElectrolytePVColumns()
                              + mycsData->getIdPVColumns())+
                                PVstartColumn);
        myPv[index] =  new OLLLPV(PVvsData[aa],myData,&myvCData,myeData,&
            mypviData);
      }
```

```
        catch(std::bad_alloc)
          {
                std::cout << "Out of Memory!" << std::endl;
                exit(-1);
          }

        index++;
        // Lower volumes
        while(index < baseIndex)
          {
            try{
                        aa = index-(presPVRow*(mycsData->
                            getElectrolytePVColumns()
                                                + mycsData->getIdPVColumns()
                                                  )+ PVstartColumn);
                        myPv[index] =  new BPV(PVvsData[aa],myData,&
                            myvCData,myeData,&mypviData);
            }
            catch(std::bad_alloc)
                {
                  std::cout << "Out of Memory!" << std::endl;
                  exit(-1);
                }
            index++;
          }

        // Lower Right Most volume
        try{
          aa = index-(presPVRow*(mycsData->getElectrolytePVColumns()
                                  + mycsData->getIdPVColumns())+
                                     PVstartColumn);

          myPv[index] =  new BPV(PVvsData[aa],myData,&myvCData,myeData,&
              mypviData);
        }
        catch(std::bad_alloc)
                {
                  std::cout << "Out of Memory!" << std::endl;
                  exit(-1);
                }
        presPVRow++;
        presPVColumn = PVstartColumn;
        index = presPVRow*(mycsData->getIdxPVColumns() + mycsData->
            getElectrolyteFVColumns() + mycsData->getIdFVColumns())
          + PVstartColumn;
        return;
}
```

```
void Electrode::makeRightElectrode(FV** myFv,PV** myPv){
  index = FVstartColumn;
  presFVRow = 0;
        presFVColumn = FVstartColumn;

        while(presFVRow < numFVRows)
          {
            while(presFVColumn < numFVColumns+FVstartColumn)
              {
                try{
                  int f = index-(presFVRow*(mycsData->
                      getElectrolyteFVColumns()
                                              + mycsData->getIdxFVColumns())
                                                 + FVstartColumn);
                  myFv[index] =  new FV(vsData[f], myData, myeData, &
                      myvCData, &myelectrodeChemData, &myiData);
                  f = f;
                }
                        catch(std::bad_alloc)
                          {
                            std::cout << "Cannot allocate FV number " <<
                               index << std::endl;
                            exit(-1);
                          }
                presFVColumn++;
                index++;
              }
            presFVColumn = FVstartColumn;
            presFVRow++;
            index = index + mycsData->getElectrolyteFVColumns() + mycsData
               ->getIdxFVColumns();
                }

        int aa;
        int baseIndex;
        try{
          PVvsData = (volumeSpatialData**) new char [numPVVolumes*sizeof(
              volumeSpatialData*)];
        }
        catch(std::bad_alloc){
          std::cout << "Couldn't Allocate vsData array!" << std::endl;
          exit(-1);
        }

        index = 0;
        presPVRow = 0;
```

```
presPVColumn = PVstartColumn;
while(presPVRow < numPVRows){
  while(presPVColumn < numPVColumns+PVstartColumn){
    try{
      PVvsData[index] = new volumeSpatialData(PVvdx,PVvdy,PVvdz,
          presPVColumn,presPVRow,0);
    }
    catch(std::bad_alloc)
      {
        std::cout << "Failed to allocate volumeSpatialData Object
            number " << index << std::endl;
        exit(-1);
      }
    presPVColumn++;
    index++;
  }
  presPVRow++;
  presPVColumn = PVstartColumn;
}
// Now initialize the Property Volumes
// Make the top Row
index = PVstartColumn;
presPVRow = 0;
presPVColumn = PVstartColumn;
baseIndex = numPVColumns+index-1;
try{
  aa = index-(presPVRow*(mycsData->getElectrolytePVColumns()
                         + mycsData->getIdxPVColumns())+
                           PVstartColumn);
  myPv[index] = new UPV(PVvsData[aa],myData,&myvCData,myeData,&
      mypviData);
}
catch(std::bad_alloc)
  {
        std::cout << "Out of Memory!" << std::endl;
        exit(-1);
  }

index++;
// Inner volumes
while(index < baseIndex)
  {
    try{
      aa = index-(presPVRow*(mycsData->getElectrolytePVColumns()
                             + mycsData->getIdxPVColumns())+
                               PVstartColumn);
      myPv[index] = new UPV(PVvsData[aa],myData,&myvCData,myeData
```

```
                ,&mypviData);
                }
            catch(std::bad_alloc)
              {
                std::cout << "Out of Memory!" << std::endl;
                exit(-1);
              }
                index++;
          }

        // Upper Right Most volume
        try{
          aa = index-(presPVRow*(mycsData->getElectrolytePVColumns()
                                    + mycsData->getIdxPVColumns())+
                                       PVstartColumn);
          myPv[index] =  new ORURPV(PVvsData[aa],myData,&myvCData,myeData,&
              mypviData);
        }
        catch(std::bad_alloc)
                {
                std::cout << "Out of Memory!" << std::endl;
                exit(-1);
                }
        presPVRow++;
        presPVColumn = PVstartColumn;
        index = presPVRow*(mycsData->getIdxPVColumns() + mycsData->
            getElectrolyteFVColumns() + mycsData->getIdFVColumns())
                        + PVstartColumn;
/*************************************************************************
    */
/************************ Now make Middle Rows of PV Volumes
     ********************************* */
/*************************************************************************
    */
        while(presPVRow != numPVRows -1){
                baseIndex = numPVColumns+index-1;
        try{
                aa = index-(presPVRow*(mycsData->getElectrolytePVColumns()
                                    + mycsData->getIdxPVColumns())+
                                       PVstartColumn);
        myPv[index] =  new PV(PVvsData[aa],myData,&myvCData,myeData,&
            mypviData);
        }
        catch(std::bad_alloc)
        {
                std::cout << "Out of Memory!" << std::endl;
                exit(-1);
```

```
        }

        index++;

        // Inner volumes
        while(index < baseIndex)
        {
                try{
                        aa = index-(presPVRow*(mycsData->
                           getElectrolytePVColumns()
                                        + mycsData->getIdxPVColumns())+
                                           PVstartColumn);
                myPv[index] =  new PV(PVvsData[aa],myData,&myvCData,myeData
                    ,&mypviData);
                }
                catch(std::bad_alloc)
                {
                std::cout << "Out of Memory!" << std::endl;
                exit(-1);
                }
                index++;
        }

        //  Right Most volume
        try{
                aa = index-(presPVRow*(mycsData->getElectrolytePVColumns()
                                        + mycsData->getIdxPVColumns())+
                                           PVstartColumn);
                myPv[index] =  new ORPV(PVvsData[aa],myData,&myvCData,
                    myeData,&mypviData);
                }
        catch(std::bad_alloc)
                {
                std::cout << "Out of Memory!" << std::endl;
                exit(-1);
                }
        presPVRow++;
        presPVColumn = PVstartColumn;
        index = presPVRow*(mycsData->getIdxPVColumns() + mycsData->
            getElectrolyteFVColumns() + mycsData->getIdFVColumns())
                        + PVstartColumn;
        }


/*************************************************************************************
    */
/************************  Now make Bottom Row of PV Volumes
```

```
    ********************************** */
/* ************************************************************************************
   */
baseIndex = numPVColumns+index-1;
        presPVColumn = PVstartColumn;
        try{
                aa = index-(presPVRow*(mycsData->getElectrolytePVColumns()
                                        + mycsData->getIdxPVColumns())+
                                            PVstartColumn);

        myPv[index] =  new BPV(PVvsData[aa],myData,&myvCData,myeData,&
            mypviData);
        }
        catch(std::bad_alloc)
        {
                std::cout << "Out of Memory!" << std::endl;
                exit(-1);
        }

        index++;
        // Lower volumes
        while(index < baseIndex)
        {
                try{
                        aa = index-(presPVRow*(mycsData->
                            getElectrolytePVColumns()
                                        + mycsData->getIdxPVColumns())+
                                            PVstartColumn);
                myPv[index] =  new BPV(PVvsData[aa],myData,&myvCData,
                    myeData,&mypviData);
                }
                catch(std::bad_alloc)
                {
                std::cout << "Out of Memory!" << std::endl;
                exit(-1);
                }
                index++;
        }

        // Lower Right Most volume
        try{
                aa = index-(presPVRow*(mycsData->getElectrolytePVColumns()
                                        + mycsData->getIdxPVColumns())+
                                            PVstartColumn);
                myPv[index] =  new ORLRPV(PVvsData[aa],myData,&myvCData,
                    myeData,&mypviData);
                }
```

```
        catch(std::bad_alloc)
                {
                std::cout << "Out of Memory!" << std::endl;
                exit(-1);
                }
        presPVRow++;
        presPVColumn = PVstartColumn;
        index = presPVRow*(mycsData->getIdxPVColumns() + mycsData->
           getElectrolyteFVColumns() + mycsData->getIdFVColumns())
                        + PVstartColumn;
        return;
}


void Electrode::makeElectrolyte(FV** myFv,PV** myPv){
                index = FVstartColumn;
        presFVRow = 0;
        presFVColumn = FVstartColumn;

                while(presFVRow < numFVRows)
                {
                        while(presFVColumn < numFVColumns+FVstartColumn)
                        {
                        try{
                                int f = index-(presFVRow*(mycsData->
                                   getIdxFVColumns()
                                        + mycsData->getIdFVColumns())+
                                           FVstartColumn);
                        myFv[index] = new FV(vsData[f], myData, myeData, &
                           myvCData, &myelectrodeChemData, &myiData);
                        f = f;
                        }
                        catch(std::bad_alloc)
                        {
                        std::cout << "Cannot allocate FV number " << index
                            << std::endl;
                        exit(-1);
                        }
                        presFVColumn++;
                        index++;
                        }
                        presFVColumn = FVstartColumn;
                        presFVRow++;
                        index = index + mycsData->getIdxFVColumns() +
                           mycsData->getIdFVColumns();
                }
```

```
int aa;
int baseIndex;
try{
        PVvsData = (volumeSpatialData**) new char [numPVVolumes*
            sizeof(volumeSpatialData*)];
        }
catch(std::bad_alloc){
        std::cout << "Couldn't Allocate vsData array!" << std::endl
            ;
        exit(-1);
        }

index = 0;
presPVRow = 0;
presPVColumn = PVstartColumn;
while(presPVRow < numPVRows){
        while(presPVColumn < numPVColumns+PVstartColumn){
        try{
        PVvsData[index] =  new volumeSpatialData(PVvdx,PVvdy,PVvdz,
            presPVColumn,presPVRow,0);
        }
catch(std::bad_alloc)
        {
        std::cout << "Failed to allocate volumeSpatialData Object
            number " << index << std::endl;
        exit(-1);
        }
        presPVColumn++;
        index++;
        }
presPVRow++;
presPVColumn = PVstartColumn;
}
//  Now initialize the Property Volumes
//  Make the top Row
index = PVstartColumn;
presPVRow = 0;
presPVColumn = PVstartColumn;
baseIndex = numPVColumns+index-1;

// Inner volumes
while(index <= baseIndex)
{
        try{
                aa = index-(presPVRow*(mycsData->
                    getElectrolytePVColumns()
                                + mycsData->getIdPVColumns())+
```

```
                                        PVstartColumn);
                        myPv[index] =  new UPV(PVvsData[aa],myData,&
                            myvCData,myeData,&mypviData);
                }
                catch(std::bad_alloc)
                {
                std::cout << "Out of Memory!" << std::endl;
                exit(-1);
                }
                index++;
        }

        presPVRow++;
        presPVColumn = PVstartColumn;
        index = presPVRow*(mycsData->getIdxPVColumns() + mycsData->
            getElectrolyteFVColumns() + mycsData->getIdFVColumns())
                        + PVstartColumn;
/*****************************************************************************
   */
/************************* Now make Middle Rows of PV Volumes
    ********************************* */
/*****************************************************************************
   */
        while(presPVRow != numPVRows -1){
                baseIndex = numPVColumns+index -1;
        // Inner Volumes
        while(index <= baseIndex)
        {
                try{
                        aa = index -(presPVRow*(mycsData->getIdxPVColumns()
                                + mycsData->getIdPVColumns()) +
                                    PVstartColumn);
                        myPv[index] =  new PV(PVvsData[aa],myData,&myvCData
                            ,myeData,&mypviData);
                }
                catch(std::bad_alloc)
                {
                std::cout << "Out of Memory!" << std::endl;
                exit(-1);
                }
                index++;
        }

        presPVRow++;
        presPVColumn = PVstartColumn;
        index = presPVRow*(mycsData->getIdxPVColumns() + mycsData->
            getElectrolyteFVColumns() + mycsData->getIdFVColumns())
```

```
                        + PVstartColumn;
        }


/*****************************************************************************
   */
/************************  Now make Bottom Row of PV Volumes
    ********************************** */
/*****************************************************************************
   */
baseIndex = numPVColumns+index-1;
        presPVColumn = PVstartColumn;
        while(index <= baseIndex)
        {
                try{
                        aa = index-(presPVRow*(mycsData->getIdxPVColumns()
                                + mycsData->getIdPVColumns())+
                                    PVstartColumn);
                        myPv[index] =  new BPV(PVvsData[aa],myData,&
                           myvCData,myeData,&mypviData);
                }
                catch(std::bad_alloc)
                {
                std::cout << "Out of Memory!" << std::endl;
                exit(-1);
                }
                index++;
        }
        presPVRow++;
        presPVColumn = PVstartColumn;
        index = presPVRow*(mycsData->getIdxPVColumns() + mycsData->
           getElectrolyteFVColumns() + mycsData->getIdFVColumns())
                        + PVstartColumn;
        return;
}

FV* Electrode::returnLeftFVptr(void){
        return myFv[0];
}

FV* Electrode::returnRightFVptr(void){
        return myFv[numFVColumns-1];
}
```

Listing B.5. IdxElectrode.h

```
/* @author James Geraci
```

```
 */
#ifndef _LDXELECTRODE_H
#define _LDXELECTRODE_H
#include "Electrode.h"
#include "ldxElectrodeChemData.h"

class ldxElectrode :
        public Electrode
{
public:
        ldxElectrode(void);
        ldxElectrode(FV**,PV**,matrixData*,electrodeSizeData*);
public:
        ~ldxElectrode(void);
private:
        static const ldxElectrodeChemData myChem;
        static const volumeChemData vCD;
        static const volumeIConditions vIC;
        static const PVIConditions pviC;
};

#endif
```

Listing B.6. ldxElectrode.cpp

```
/*
 * @author James Geraci
 * object needed to create a lead dioxide electrode
 */
#include "StdAfx.h"
#include "ldxElectrode.h"

//const ldxElectrodeChemData ldxElectrode::myChem(0.72/*tnp*/,0.0002/*inot
    */,-8.29187e-7/*mbScale*/,23.0260/*myVf*/);
const ldxElectrodeChemData ldxElectrode::myChem(0.72/*tnp*/,0.0002/*inot*/
    ,-8.08399e-6/*mbScale*/,23.0260/*myVf*/);
const volumeChemData ldxElectrode::vCD(298/*T*/,100/*Amax*/,2800/*Qmax*/
    ,0.8/*soc*/,1/*eta*/,1.5/N/*als*/,0.5/N/*asl*/,3.0e-5/*Dx*/,3.0e-5/*Dy*/
    ,
                                        80/*sigx*/,80/*sigy*/,0.79/*Kappax*/
                                            ,0.79/*Kappay*/,1/*k*/,1/*m*/
                                            ,0.5 /*exm*/,1.5 /*ex*/,1.685/*
                                            phieq*/,
                                        4.9e-3/*hnot*/,0.35/*epsnot*/,1/*
                                            mySign*/,10e-2/*mu_x*/,10e-2/*
                                            mu_y*/,35/*beta_x*/,35/*beta_y*/)
                                            ;
const volumeIConditions ldxElectrode::vIC(0.35/*eps*/,0,0);
const PVIConditions ldxElectrode::pviC(4.9e-3/*hp*/,0/*phi_l*/,1.685/*phi_s
    */,atmospheric_pressure/*p*/,0/*v_y*/,0/*v_x*/);

ldxElectrode::ldxElectrode(void)
{
}

ldxElectrode::ldxElectrode(FV** myFV, PV** myPV, matrixData* myMatrix,
    electrodeSizeData* eD):Electrode("ldx",myFV,myPV,myMatrix, eD,myChem,vCD
    ,vIC,pviC)
{
}

ldxElectrode::~ldxElectrode(void)
{
}
```

Listing B.7. ldElectrode.h

```
/* @author James Geraci
 */
#ifndef _LDELECTRODE_H
#define _LDELECTRODE_H
#include "Electrode.h"
#include "ldElectrodeChemData.h"

class ldElectrode :
        public Electrode
{
public:
        ldElectrode(void);
        ldElectrode(FV**,PV**,matrixData*,electrodeSizeData*);
public:
        ~ldElectrode(void);
        static const ldElectrodeChemData myChem;
        static const volumeChemData vCD;
        static const volumeIConditions vIC;
        static const PVIConditions pviC;
};

#endif
```

Listing B.8. ldElectrode.cpp

```
/*
 * @author James Geraci
 * needed to construct a lead electrode
 */
#include "StdAfx.h"
#include "ldElectrode.h"

const ldElectrodeChemData ldElectrode::myChem(0.72,0.002,2.2801e-6,-29.1873
    /*myVf*/);
const volumeChemData ldElectrode::vCD(298/*T*/,100/*Amax*/,2471/*Qmax*/,0.8
    /*soc*/,1/*eta*/,1.5/N/*als*/,0.5/N/*asl*/,3.0e-5/*Dx*/,3.0e-5/*Dy*/,
                                        48000/*sigx*/,48000/*sigy*/,0.79/*
                                            Kappax*/,0.79/*Kappay*/,1/*k*/,1/*
                                            m*/,0.5 /*exm*/,1.5 /*ex*/,
                                        -0.356/*phieq*/,4.9e-3/*hnot*/,0.35/*
                                            epsnot*/,-1/*mySign*/,10e-2/*mu_x
                                            */,10e-2/*mu_y*/,
                                        35/*beta_x*/,35/*beta_y*/);
const volumeIConditions ldElectrode::vIC(0.35,0,0);
const PVIConditions ldElectrode::pviC(4.9e-3/*hp*/,0/*phi_l*/,-0.356/*phi_s
    */,atmospheric_pressure/*p*/,0/*v_y*/,0/*v_x*/);

ldElectrode::ldElectrode(void)
{
}

ldElectrode::ldElectrode(FV** myFV, PV** myPV, matrixData* myMatrix,
    electrodeSizeData* eD):Electrode("ld",myFV,myPV,myMatrix,eD,myChem,vCD,
    vIC,pviC)
{
}

ldElectrode::~ldElectrode(void)
{
}
```

Listing B.9. PV.h

```
/* @author James Geraci
 */
#ifndef _PV_H
#define _PV_H
#include "FV.h"
#include "CV.h"
#include "ULCV.h"
#include "URCV.h"
#include "LLCV.h"
#include "LRCV.h"

class PV
{
public:
        PV(void);
        PV(volumeSpatialData*,matrixData*,volumeChemData*,electrodeSizeData
           *,PVIConditions*);
public:
        virtual ~PV(void);

        void computeDerivatives(void);
        void computeIValues(void);
        void setupJacobian(void);
        void setupFvPointers(FV** myFv);
        void initializeSize(void);
        void initializembScale(void);
        void computeDiv(void);
        void computeDivTwo(void);
        void computeDensity(void);
        void initializePressure(void);
        void computeAVEpressures(void);
        void computeAVEhps(void);
        //      double midCurrents[4];
        //void resetmidCurrents(void);
protected:
        CV** myCV;
        matrixData* myMatrix;
        double* fluxVptr;
        double* divVptr;
        double* divVtwoptr;
        double* pVptr;
        double* Vptr;
        double** iVptr;
        double** Jacobptr;

        // Functions
```

```
void initializeMatrix(matrixData*);
void initializePosition(volumeSpatialData*, matrixData*,
   electrodeSizeData*);
void initializeOtherPositions(void);
// void initializeSize(volumeSpatialData*);
void setInitialConditions(PVIConditions*);
void initializeVolumeChem(volumeChemData*);
void initializeCVsizes(void);
void setMyVolume(double*);
void tellCVPVvolume(void);
void setupCVPressurePointers(void);
void setupCVConcentrationPointers(void);
void computeAVExPressures(void);
void computeAVEyPressures(void);
void computeAVEhpx(void);
void computeAVEhpy(void);

// 2d model
void setURFV(FV**,CV*);
void setULFV(FV**,CV*);
void setLLFV(FV**,CV*);
void setLRFV(FV**,CV*);

void computeIVmb(void);
virtual void computeIVdoc(void);
void computeIVek(void);
void computeIVconteq(void);
void computeIVcomv_y(void);
void computeIVcomv_x(void);

// Electrode Kinetics Jacobian Equations
void ekJ(void);
void ekJeps(void);
//void ekJhp(void);
void ekJhpz(void);
void ekJphisz(void);
void ekJphilz(void);

// Divergence of Current Jacobian Equations
void docJ(void);
/*
void docJtphil(double);
void docJpphil(double);
void docJphil(double);
void docJnphil(double);
void docJbphil(double);
*/
```

```
void docJthp(double);
void docJphp(double);
void docJhp(double);
void docJnhp(double);
void docJbhp(double);
void docJphisz(void);
void docJphilz(double);
void docJhpz(double);
/*
void docJtphis(void);
void docJpphis(void);
void docJphis(void);
void docJnphis(void);
void docJbphis(void);
*/
// Material Balance Jacobian Equations
void mbJ(void);
void mbJhpz(void);
//      void mbJthp(void);
//      void mbJphp(void);
//      void mbJhp(void);
//      void mbJnhp(void);
//      void mbJbhp(void);
//      void mbJtnhp(void);
//      void mbJtphp(void);
//      void mbJbphp(void);
//void mbJbnhp(void);
// Continuity Equation Jacobian Equations
void conteqJ(void);
void conteqJv_x(void);
void conteqJv_y(void);
void conteqJhp(void);
void conteqJthp(void);
void conteqJbhp(void);
void conteqJnhp(void);
void conteqJphp(void);

// Conservation of Momentum Y Jacobian Equations
void comv_yJ(void);
void comv_yJv_y(void);
void comv_yJhp(void);
void comv_yJp(void);

// Conservation of Momentum X Jacobian Equations
void comv_xJ(void);
void comv_xJv_x(void);
```

```
    void comv_xJhp(void);
    void comv_xJp(void);

    // Time Derivatives
    double dhdt;
    double setdhdt(void);

    // Position information for Cell
    int myColumn;
    int myRow;
    int myZrow;
    int JRow;
    int JRowBuffer;
    int offset;
    // Size Data
    double mydx;
    double mydy;
    double mydz;

// Positioning Data
    int rowLength;
    int maxPvIndex;
    int nextJRow;

    // FV grid Information
    int numFVColumns;
    int numFVRows;

    //int maxPvIndex;
    int numPVColumns;
    int numPVRows;
    int numPVvolumes;
    int dn;
    int nv;

    //  Used in Calculating Addresses
    int prevV;
    int presV;
    int nextV;

    int topV;
    int lowerV;

    // present volume's properties
    int hpc;
    int phisc;
    int philc;
```

```
int pc;
int v_yc;
int v_xc;
// previous volume's properties
int phpc;
int pphisc;
int pphilc;
int ppc;
int pv_yc;
int pv_xc;
// next volume's properties
int nhpc;
int nphisc;
int nphilc;
int npc;
int nv_yc;
int nv_xc;
// top previous volume's properties
int tphpc;
int tpphisc;
int tpphilc;
int tppc;
int tpv_yc;
int tpv_xc;
// top next volume's properties
int tnhpc;
int tnphisc;
int tnphilc;
int tnpc;
int tnv_yc;
int tnv_xc;
// top volume's properties
int thpc;
int tphisc;
int tphilc;
int tpc;
int tv_yc;
int tv_xc;
// bottom previous volume's properties
int bphpc;
int bpphisc;
int bpphilc;
int bppc;
int bpv_yc;
int bpv_xc;
// bottom next volume's properties
int bnhpc;
```

```
        int bnphisc;
        int bnphilc;
        int bnpc;
        int bnv_yc;
        int bnv_xc;
// lower volume's properties
        int bhpc;
        int bphisc;
        int bphilc;
        int bpc;
        int bv_yc;
        int bv_xc;

// present volume's properties for Jacobian Indexing
        int jhpc;
        int jphisc;
        int jphilc;
        int jpc;
        int jv_yc;
        int jv_xc;
// previous volume's properties for Jacobian Indexing
        int jphpc;
        int jpphisc;
        int jpphilc;
        int jppc;
        int jpv_yc;
        int jpv_xc;
// next volume's properties for Jacobian Indexing
        int jnhpc;
        int jnphisc;
        int jnphilc;
        int jnpc;
        int jnv_yc;
        int jnv_xc;
// top previous volume's properties for Jacobian Indexing
        int jtphpc;
        int jtpphisc;
        int jtpphilc;
        int jtppc;
        int jtpv_yc;
        int jtpv_xc;
// top next volume's properties for Jacobian Indexing
        int jtnhpc;
        int jtnphisc;
        int jtnphilc;
        int jtnpc;
        int jtnv_yc;
```

```
        int jtnv_xc;
// top volume's properties for Jacobian Indexing
        int jthpc;
        int jtphisc;
        int jtphilc;
        int jtpc;
        int jtv_yc;
        int jtv_xc;
// lower previous volume's properties for Jacobian Indexing
        int jbphpc;
        int jbpphisc;
        int jbpphilc;
        int jbppc;
        int jbpv_yc;
        int jbpv_xc;
// lower next volume's properties for Jacobian Indexing
        int jbnhpc;
        int jbnphisc;
        int jbnphilc;
        int jbnpc;
        int jbnv_yc;
        int jbnv_xc;
// lower volume's properties for Jacobian Indexing
        int jbhpc;
        int jbphisc;
        int jbphilc;
        int jbpc;
        int jbv_yc;
        int jbv_xc;

        int divc;
        double phi_eq;
        double mySign;
        double hnot;
        double mbScale;
        double myVolume;
        static const int philg = 1;  // 1 = phil before phis
        static const int phisg = 2;  // 1 = phis before phil
};

#endif
```

Listing B.10. PV.cpp

```
/*
 * @author James Geraci
 * creates a PV volume.  These store the states: concentration, phil, phis
    , & pressure
 * and each PV contains 4 CVs
 */
#include "StdAfx.h"
#include "PV.h"
#include <iostream>
using namespace std;
PV::PV(void)
{
}

PV::PV(volumeSpatialData* sData, matrixData* mD,
       volumeChemData* vcData, electrodeSizeData* esData,
       PVIConditions* pviC){
  myCV = (CV**) new char [NUM_CVperPV*sizeof(CV*)];
  myCV[UL] = new ULCV();
  myCV[UR] = new URCV();
  myCV[LL] = new LLCV();
  myCV[LR] = new LRCV();

  for(int index = 0; index < NUM_CVperPV; index++){
    myCV[index]->setPosition(index);
  }
        initializeMatrix(mD);
        initializePosition(sData,mD,esData);
        initializeOtherPositions();
        initializeVolumeChem(vcData);
        setInitialConditions(pviC);
        return;
}

PV::~PV(void)
{
  for(int index = 0; index < NUM_CVperPV; index++){
    delete myCV[index];
  }
  delete [] myCV;
}



void PV::initializeMatrix(matrixData *mData)
{
```

```
  myMatrix = mData;
  fluxVptr = mData->myfluxV();
  divVptr = mData->mydivV();
  divVtwoptr = mData->mydivVtwo();
  pVptr = mData->mypsV();
  Vptr = mData->mysV();
  iVptr = mData->myiV();
  Jacobptr = mData->myJacobian();
  numFVColumns = mData->myFVColumns();
  numFVRows = mData->myFVRows();

  for(int index = 0; index < NUM_CVperPV; index++){
    myCV[index]->fluxVptr = mData->myfluxV();
    myCV[index]->pVptr = mData->mypsV();
    myCV[index]->Vptr = mData->mysV();
    myCV[index]->iVptr = mData->myiV();
    myCV[index]->Jacobptr = mData->myJacobian();
    myCV[index]->numFVColumns = mData->myFVColumns();
    myCV[index]->numFVRows = mData->myFVRows();
    myCV[index]->avePressures_y = mData->myavePressures_y();
    myCV[index]->avePressures_x = mData->myavePressures_x();
    myCV[index]->AVEhp_x = mData->myAVEhp_x();
    myCV[index]->AVEhp_y = mData->myAVEhp_y();
  }
  return;
}

void PV::initializePosition(volumeSpatialData *sData, matrixData* mData,
   electrodeSizeData* eData)
{
  jhpc = mData->myJhpc();
  myColumn = sData->getColumn();
  myRow = sData->getRow();
  myZrow = sData->getZrow();
  rowLength = mData->mysvLength();
  rowLength = mData->mysvLength();
  numPVColumns = mData->myPVColumns();
  numPVvolumes = mData->myPVColumns()*mData->myPVRows()*mData->myPVZrows();
  maxPvIndex = mData->myPVColumns()*mData->myPVRows()*mData->myPVZrows()-1;

  presV = /*eData->getPVstartColumn()+*/myColumn + myRow*mData->myPVColumns
     (); //myColumn;
  prevV = presV-1;
  nextV = presV+1;
  topV = presV - mData->myPVColumns();
  lowerV = presV + mData->myPVColumns();
```

```
  if(mode == 0){
    JRow = presV;
    nextJRow = numPVvolumes;
  }else{
    JRow = myVars*presV;
    nextJRow = 1;
  }

  for(int index = 0; index < NUM_CVperPV; index++){
    // Cell Position Data Initialization
    myCV[index]->myColumn = sData->getColumn();
    myCV[index]->myRow = sData->getRow();
    myCV[index]->myZrow = sData->getZrow();
    myCV[index]->rowLength = mData->mysvLength();
    myCV[index]->maxPvIndex = mData->myPVColumns()*mData->myPVRows()*mData
        ->myPVZrows()-1;

    myCV[index]->presV = myColumn + myRow*mData->myPVColumns();
    myCV[index]->prevV = presV-1;
    myCV[index]->nextV = presV+1;
    myCV[index]->topV = presV - mData->myPVColumns();
    myCV[index]->lowerV = presV + mData->myPVColumns();

    if(mode == 0){
        myCV[index]->JRow = presV;
    }else{
      myCV[index]->JRow = myVars*presV;
    }
  }
  return;
}

void PV::initializeOtherPositions(void)
{
  if(mode == 0){
    dn = 1;
    nv = numPVvolumes;
  }else{
    dn = myVars;
    nv = 1;
  }
  divc = presV;
  /*********************************/
  /******Need to Adjust Index*******/
  /*********************************/
  if(myVars%2){
    offset = presV%2;
```

```
}else{
  offset = 0;
}

// These are for Jacobian Indexing
//jhpc = numPVColumns*myVars+offset;
jphpc = jhpc-dn;
jnhpc = jhpc+dn;
// jthpc = offset;
jthpc = jhpc - numPVColumns*myVars;
jbhpc = jhpc + (numPVColumns)*myVars;
jtnhpc = jthpc + myVars;
jtphpc = jthpc - myVars;
jbnhpc = jbhpc + myVars;
jbphpc = jbhpc - myVars;

jphilc =  jhpc  +  philg*nv;
jpphilc = jphpc +  philg*nv;
jnphilc = jnhpc +  philg*nv;
jtphilc = jthpc +  philg*nv;
jbphilc = jbhpc +  philg*nv;
jtnphilc = jtnhpc + philg*nv;
jtpphilc = jtphpc + philg*nv;
jbnphilc = jbnhpc + philg*nv;
jbpphilc = jbphpc + philg*nv;

jphisc =  jhpc  + phisg*nv;
jpphisc = jphpc + phisg*nv;
jnphisc = jnhpc + phisg*nv;
jtphisc = jthpc + phisg*nv;
jbphisc = jbhpc + phisg*nv;
jtnphisc = jtnhpc + phisg*nv;
jtpphisc = jtphpc + phisg*nv;
jbnphisc = jbnhpc + phisg*nv;
jbpphisc = jbphpc + phisg*nv;

jpc =  jhpc  + 3*nv;
jppc = jphpc + 3*nv;
jnpc = jnhpc + 3*nv;
jtpc = jthpc + 3*nv;
jbpc = jbhpc + 3*nv;
jtnpc = jtnhpc + 3*nv;
jtppc = jtphpc + 3*nv;
jbnpc = jbnhpc + 3*nv;
jbppc = jbphpc + 3*nv;

jv_yc =  jhpc  + 4*nv;
```

```
jpv_yc = jphpc + 4*nv;
jnv_yc = jnhpc + 4*nv;
jtv_yc = jthpc + 4*nv;
jbv_yc = jbhpc + 4*nv;
jtnv_yc = jtnhpc + 4*nv;
jtpv_yc = jtphpc + 4*nv;
jbnv_yc = jbnhpc + 4*nv;
jbpv_yc = jbphpc + 4*nv;

jv_xc =  jhpc  + 5*nv;
jpv_xc = jphpc + 5*nv;
jnv_xc = jnhpc + 5*nv;
jtv_xc = jthpc + 5*nv;
jbv_xc = jbhpc + 5*nv;
jtnv_xc = jtnhpc + 5*nv;
jtpv_xc = jtphpc + 5*nv;
jbnv_xc = jbnhpc + 5*nv;
jbpv_xc = jbphpc + 5*nv;


// These are for iVptr and Vptr Indexing
hpc = dn*presV;
phpc = hpc-dn;
nhpc = hpc+dn;
thpc = dn*topV;
bhpc = dn*lowerV;
tnhpc = thpc + myVars;
tphpc = thpc - myVars;
bnhpc = bhpc + myVars;
bphpc = bhpc - myVars;

philc = dn*presV + philg*nv;
pphilc = philc - dn;
nphilc = philc + dn;
tphilc = dn*topV + philg*nv;
bphilc = dn*lowerV + philg*nv;
tnphilc = tphilc + myVars;
tpphilc = tphilc - myVars;
bnphilc = bphilc + myVars;
bpphilc = bphilc - myVars;

phisc = dn*presV + phisg*nv;
pphisc = phisc - dn;
nphisc = phisc + dn;
tphisc = dn*topV + phisg*nv;
bphisc = dn*lowerV + phisg*nv;
tnphisc = tphisc + myVars;
```

```
 tpphisc = tphisc - myVars;
 bnphisc = bphisc + myVars;
 bpphisc = bphisc - myVars;

 pc = dn*presV + 3*nv;
 ppc = pc - dn;
 npc = pc + dn;
 tpc = dn*topV + 3*nv;
 bpc = dn*lowerV + 3*nv;
 tnpc = tpc + myVars;
 tppc = tpc - myVars;
 bnpc = bpc + myVars;
 bppc = bpc - myVars;

 v_yc = dn*presV + 4*nv;
 pv_yc = v_yc - dn;
 nv_yc = v_yc + dn;
 tv_yc = dn*topV + 4*nv;
 bv_yc = dn*lowerV + 4*nv;
 tnv_yc = tv_yc + myVars;
 tpv_yc = tv_yc - myVars;
 bnv_yc = bv_yc + myVars;
 bpv_yc = bv_yc - myVars;

 v_xc = dn*presV + 5*nv;
 pv_xc = v_xc - dn;
 nv_xc = v_xc + dn;
 tv_xc = dn*topV + 5*nv;
 bv_xc = dn*lowerV + 5*nv;
 tnv_xc = tv_xc + myVars;
 tpv_xc = tv_xc - myVars;
 bnv_xc = bv_xc + myVars;
 bpv_xc = bv_xc - myVars;

 for(int index = 0; index < NUM_CVperPV; index++){
// These are for Jacobian Indexing
 //jhpc = numPVColumns*myVars+offset;
   myCV[index]->jhpc = jhpc;
  myCV[index]->jphpc = jhpc-dn;
  myCV[index]->jnhpc = jhpc+dn;
 // jthpc = offset;
  myCV[index]->jthpc = jhpc - numPVColumns*myVars;
  myCV[index]->jbhpc = jhpc + (numPVColumns)*myVars;
  myCV[index]->jtnhpc = jthpc + myVars;
  myCV[index]->jtphpc = jthpc - myVars;
  myCV[index]->jbnhpc = jbhpc + myVars;
  myCV[index]->jbphpc = jbhpc - myVars;
```

```
myCV[index]->jphilc =  jhpc  +  philg*nv;
myCV[index]->jpphilc = jphpc +   philg*nv;
myCV[index]->jnphilc = jnhpc +   philg*nv;
myCV[index]->jtphilc = jthpc +   philg*nv;
myCV[index]->jbphilc = jbhpc +   philg*nv;
myCV[index]->jtnphilc = jtnhpc + philg*nv;
myCV[index]->jtpphilc = jtphpc + philg*nv;
myCV[index]->jbnphilc = jbnhpc + philg*nv;
myCV[index]->jbpphilc = jbphpc + philg*nv;

myCV[index]->jphisc =  jhpc  + phisg*nv;
myCV[index]->jpphisc = jphpc + phisg*nv;
myCV[index]->jnphisc = jnhpc + phisg*nv;
myCV[index]->jtphisc = jthpc + phisg*nv;
myCV[index]->jbphisc = jbhpc + phisg*nv;
myCV[index]->jtnphisc = jtnhpc + phisg*nv;
myCV[index]->jtpphisc = jtphpc + phisg*nv;
myCV[index]->jbnphisc = jbnhpc + phisg*nv;
myCV[index]->jbpphisc = jbphpc + phisg*nv;

myCV[index]->jpc =  jhpc  + 3*nv;
myCV[index]->jppc = jphpc + 3*nv;
myCV[index]->jnpc = jnhpc + 3*nv;
myCV[index]->jtpc = jthpc + 3*nv;
myCV[index]->jbpc = jbhpc + 3*nv;
myCV[index]->jtnpc = jtnhpc + 3*nv;
myCV[index]->jtppc = jtphpc + 3*nv;
myCV[index]->jbnpc = jbnhpc + 3*nv;
myCV[index]->jbppc = jbphpc + 3*nv;

myCV[index]->jv_yc =  jhpc  + 4*nv;
myCV[index]->jpv_yc = jphpc + 4*nv;
myCV[index]->jnv_yc = jnhpc + 4*nv;
myCV[index]->jtv_yc = jthpc + 4*nv;
myCV[index]->jbv_yc = jbhpc + 4*nv;
myCV[index]->jtnv_yc = jtnhpc + 4*nv;
myCV[index]->jtpv_yc = jtphpc + 4*nv;
myCV[index]->jbnv_yc = jbnhpc + 4*nv;
myCV[index]->jbpv_yc = jbphpc + 4*nv;

myCV[index]->jv_xc =  jhpc  + 5*nv;
myCV[index]->jpv_xc = jphpc + 5*nv;
myCV[index]->jnv_xc = jnhpc + 5*nv;
myCV[index]->jtv_xc = jthpc + 5*nv;
myCV[index]->jbv_xc = jbhpc + 5*nv;
myCV[index]->jtnv_xc = jtnhpc + 5*nv;
```

```
myCV[index]->jtpv_xc = jtphpc + 5*nv;
myCV[index]->jbnv_xc = jbnhpc + 5*nv;
myCV[index]->jbpv_xc = jbphpc + 5*nv;

 myCV[index]->hpc = dn*presV;
 myCV[index]->phpc = hpc-dn;
 myCV[index]->nhpc = hpc+dn;
 myCV[index]->thpc = dn*topV;
 myCV[index]->bhpc = dn*lowerV;
 myCV[index]->tnhpc = thpc + myVars;
 myCV[index]->tphpc = thpc - myVars;
 myCV[index]->bnhpc = bhpc + myVars;
 myCV[index]->bphpc = bhpc - myVars;

 myCV[index]->philc = dn*presV + philg*nv;
 myCV[index]->pphilc = philc - dn;
 myCV[index]->nphilc = philc + dn;
 myCV[index]->tphilc = dn*topV + philg*nv;
 myCV[index]->bphilc = dn*lowerV + philg*nv;
 myCV[index]->tnphilc = tphilc + myVars;
 myCV[index]->tpphilc = tphilc - myVars;
 myCV[index]->bnphilc = bphilc + myVars;
 myCV[index]->bpphilc = bphilc - myVars;

 myCV[index]->phisc = dn*presV + phisg*nv;
 myCV[index]->pphisc = phisc - dn;
 myCV[index]->nphisc = phisc + dn;
 myCV[index]->tphisc = dn*topV + phisg*nv;
 myCV[index]->bphisc = dn*lowerV + phisg*nv;
 myCV[index]->tnphisc = tphisc + myVars;
 myCV[index]->tpphisc = tphisc - myVars;
 myCV[index]->bnphisc = bphisc + myVars;
 myCV[index]->bpphisc = bphisc - myVars;

 myCV[index]->pc = dn*presV + 3*nv;
 myCV[index]->ppc = pc - dn;
 myCV[index]->npc = pc + dn;
 myCV[index]->tpc = dn*topV + 3*nv;
 myCV[index]->bpc = dn*lowerV + 3*nv;
 myCV[index]->tnpc = tpc + myVars;
 myCV[index]->tppc = tpc - myVars;
 myCV[index]->bnpc = bpc + myVars;
 myCV[index]->bppc = bpc - myVars;

 myCV[index]->v_yc = dn*presV + 4*nv;
 myCV[index]->pv_yc = v_yc - dn;
 myCV[index]->nv_yc = v_yc + dn;
```

```
    myCV[index]->tv_yc = dn*topV + 4*nv;
    myCV[index]->bv_yc = dn*lowerV + 4*nv;
    myCV[index]->tnv_yc = tv_yc + myVars;
    myCV[index]->tpv_yc = tv_yc - myVars;
    myCV[index]->bnv_yc = bv_yc + myVars;
    myCV[index]->bpv_yc = bv_yc - myVars;

    myCV[index]->v_xc = dn*presV + 5*nv;
    myCV[index]->pv_xc = v_xc - dn;
    myCV[index]->nv_xc = v_xc + dn;
    myCV[index]->tv_xc = dn*topV + 5*nv;
    myCV[index]->bv_xc = dn*lowerV + 5*nv;
    myCV[index]->tnv_xc = tv_xc + myVars;
    myCV[index]->tpv_xc = tv_xc - myVars;
    myCV[index]->bnv_xc = bv_xc + myVars;
    myCV[index]->bpv_xc = bv_xc - myVars;
  }

  for(int index = 0; index < NUM_CVperPV; index++){
    myCV[index]->myNvalues();
    myCV[index]->setupMyNbrHood();
  }
  return;
};

void PV::initializeVolumeChem(volumeChemData *vData)
{
  phi_eq = vData->getphi_eq();
  hnot = vData->gethnot();
  mySign = vData->getmySign();
  return;
}

/*******************************************************/
/*************** ComputVolume Sizes ***************/
/*******************************************************/
void PV::initializeSize(void)
{
  initializeCVsizes();
  setMyVolume(&myVolume);
  tellCVPVvolume();
  setupCVPressurePointers();
  setupCVConcentrationPointers();
  return;
}

void PV::initializeCVsizes(void){
```

```
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      myCV[index]->computedxSize();
      myCV[index]->computedySize();
      myCV[index]->computedzSize();
      myCV[index]->computemyVolume();
      // myCV[index]->setQscalex();
      //  myCV[index]->setQscaley();
    }


  }
  return;
};

void PV::setMyVolume(double* myV){
  myVolume = 0;
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      *myV += myCV[index]->returnmyVolume();
    }
  }
  return;
}

void PV::tellCVPVvolume(void){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      myCV[index]->setpvVolume(myVolume);
    }
  }
  return;
}

void PV::setupCVPressurePointers(void){
  for(int index=0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      myCV[index]->setupPressurePointers(myVolume);
    }
  }
  return;
}

void PV::setupCVConcentrationPointers(void){
  // printf("Setting up CV Pointers\n");
  for(int index=0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
```

```
      myCV[index]->setupConcentrationPointers(myVolume);
    }
  }
  return;
}

/*****************************************************/
/*************** Initialize mbScale ****************/
/*****************************************************/
void PV::initializembScale(void)
{
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      mbScale = myCV[index]->returnmbScale();
    }
  }
  return;
}

void PV::setInitialConditions(PVIConditions* iData)
{
  Vptr[hpc]    = iData->gethp();
  Vptr[philc]  = iData->getphi_l();
  Vptr[phisc]  = iData->getphi_s();
  Vptr[pc]     = iData->getp();
  Vptr[v_yc]   = iData->getv_y();
  Vptr[v_xc]   = iData->getv_x();
  return;
}

void PV::initializePressure(void){
  if(myRow == 0){
    Vptr[pc] = atmospheric_pressure;
  }
  // if(myVolume > 1 && myVolume < 300){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      myCV[index]->initialPressure();
    }
  }
  // printf("My Volume = %d\n",myColumn+myRow*myVars);
  // }
  return;
}

void PV::computeDerivatives(void){
  for(int index = 0; index < NUM_CVperPV; index++){
```

```
      if(myCV[index]->exist){
        myCV[index]->setdhdt();
        myCV[index]->setdmv_ydt();
        myCV[index]->setdmv_xdt();
        myCV[index]->setddensitydt();
      }
    }
    return;
}

void PV::computeAVEpressures(void){
    computeAVEyPressures();
    computeAVExPressures();
    return;
}

void PV::computeAVEhps(void){
    computeAVEhpx();
    computeAVEhpy();
    return;
}

void PV::computeAVEhpx(void){
    for(int index = 0; index < NUM_CVperPV; index++){
      if(myCV[index]->exist){
        myCV[index]->computeAVEhpx();
      }
    }
    return;
}

void PV::computeAVEhpy(void){
    for(int index = 0; index < NUM_CVperPV; index++){
      if(myCV[index]->exist){
        myCV[index]->computeAVEhpy();
      }
    }
    return;
}

void PV::computeAVEyPressures(void){
    for(int index = 0; index < NUM_CVperPV; index++){
      if(myCV[index]->exist){
        myCV[index]->computeAVEyPressures();
      }
    }
    return;
```

```
}

void PV::computeAVExPressures(void){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      myCV[index]->computeAVExPressures();
    }
  }
  return;
}

/* ************************************************ */
/* ************** Setup FV Pointers ***************** */
/* ************************************************ */
void PV::setupFvPointers(FV** myFv){
  if(myCV[UL]->exist)
    setULFV(myFv,myCV[UL]);
  if(myCV[UR]->exist)
    setURFV(myFv,myCV[UR]);
  if(myCV[LL]->exist)
    setLLFV(myFv,myCV[LL]);
  if(myCV[LR]->exist)
    setLRFV(myFv,myCV[LR]);
  return;
};

void PV::setULFV(FV** FVp, CV* myCV){
  myCV->myFV = FVp[(myRow-1)*numFVColumns+(myColumn-1)];
  return;
};

void PV::setURFV(FV** FVp, CV* myCV){
  myCV->myFV = FVp[(myRow-1)*numFVColumns+(myColumn)];
  return;
};

void PV::setLLFV(FV** FVp, CV* myCV){
  myCV->myFV = FVp[(myRow)*numFVColumns+(myColumn-1)];
  return;
};

void PV::setLRFV(FV** FVp, CV* myCV){
  myCV->myFV = FVp[(myRow)*numFVColumns+(myColumn)];
  return;
};
```

```
void PV::setupJacobian()
{
  JRowBuffer = JRow;
    mbJ();
  //      if(JRow < 1){
  //   unsigned int cmd_status;
  unsigned int ou;
  unsigned int mask = 0;
  int xxx = 0;
  // printf("JRow = %d\n",JRow);
  /*   for(int index = 0; index < 2; index++){
  spe_mfcio_get(myMatrix->speids[0],
                  (unsigned int)myMatrix->spu_buffers[0],
                  (void*)myMatrix->JacobianStorage, myMatrix->alignedRowSize
                     ,0,0,0);
                  }*/
  //xxx = spe_mfcio_tag_status_read(myMatrix->speids[0], 0,
     SPE_TAG_IMMEDIATE, &ou);

  JRow += nextJRow;
   docJ();
  JRow += nextJRow;
   ekJ();
  if(myVars == 6){
    JRow += nextJRow;
        comv_xJ();
    JRow += nextJRow;
    comv_yJ();
    JRow += nextJRow;
    conteqJ();
  }
  JRow = JRowBuffer;
  //xxx=    spe_mfcio_tag_status_read(myMatrix->speids[0],mask,SPE_TAG_ALL,
     NULL);
  //   myMatrix->sendPVJRows();
  //   myMatrix->checkDMAqueueStatus();
  //printf("The tag status is 0x%x with the return value of %d\n",ou,xxx);
  return;
};


/*************************************************************/
/********** Contribution from Each of the 4 X volumes ******/
/********** Material Balance Equation ********************/
/*************************************************************/

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

```
/* ********* Compute Equations ************************ */
/* ************************************************** */

void PV::computeIValues(void){
  computeIVmb();
  computeIVdoc();
  computeIVek();
  if(myVars == 6){
    computeIVcomv_x();
    computeIVcomv_y();
    computeIVconteq();
  }
  return;
};

void PV::computeIVmb(void){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      iVptr[hpc][0] += myCV[index]->setdhdtCV()
        + myCV[index]->setDiffusionCV()
        + myCV[index]->setConvectionCV()
        + mbScale*myCV[index]->setdocCVjl();
    }
  }
  return;
};

void PV::computeIVdoc(void){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      //iVptr[philc][0] += myCV[index]->setdocCV();
      iVptr[JRow+1][0] += myCV[index]->setdocCV();
    }
  }
  return;
};

void PV::computeIVek(void){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      //iVptr[phisc][0] += myCV[index]->setdocCVjl() + myCV[index]->setekCV
          ();
      iVptr[JRow+2][0] += myCV[index]->setdocCVjl() + myCV[index]->setekCV
          ();
    }
  }
  return;
```

```
};



void PV::computeIVcomv_y(void){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      iVptr[v_yc][0] += 0;/*myCV[index]->setdmv_ydtCV();
      + myCV[index]->setdpdyCV()
      + myCV[index]->setGravitySource_yCV();*/
    }
  }
  return;
};

void PV::computeIVcomv_x(void){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      iVptr[pc][0] += 0;/*myCV[index]->setdmv_xdtCV();
      + myCV[index]->setdpdxCV()
      + myCV[index]->setGravitySource_xCV();*/
    }
  }
  return;
};

void PV::computeIVconteq(void){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      iVptr[v_xc][0] += 0;//myCV[index]->setContEqCV();
    }
  }
  return;
};

void PV::computeDivTwo(void){
    if(myCV[UR]->exist){
      divVtwoptr[4*presV+0] += myCV[UR]->setdocCVjlx()/2;
      divVtwoptr[4*presV+2] += myCV[UR]->setdocCVjly()/2;
    }else{
      divVtwoptr[4*presV+0] += 0;
      divVtwoptr[4*presV+2] += 0;
    }
    if(myCV[UL]->exist){
      divVtwoptr[4*presV+0] += myCV[UL]->setdocCVjlx()/2;
      divVtwoptr[4*presV+3] += myCV[UL]->setdocCVjly()/2;
    }else{
```

```
      divVtwoptr[4*presV+0] += 0;
      divVtwoptr[4*presV+3] += 0;
    }
    if(myCV[LR]->exist){
      divVtwoptr[4*presV+1] += myCV[LR]->setdocCVjlx()/2;
      divVtwoptr[4*presV+2] += myCV[LR]->setdocCVjly()/2;
    }else{
      divVtwoptr[4*presV+1] += 0;
      divVtwoptr[4*presV+2] += 0;
    }
    if(myCV[LL]->exist){
      divVtwoptr[4*presV+1] += myCV[LL]->setdocCVjlx()/2;
      divVtwoptr[4*presV+3] += myCV[LL]->setdocCVjly()/2;
    }else{
      divVtwoptr[4*presV+1] += 0;
      divVtwoptr[4*presV+3] += 0;
    }
  return;
}

void PV::computeDiv(void){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      divVptr[divc] += myCV[index]->setdocCVjl();
    }
  }
  return;
};


void PV::computeDensity(void){
 for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      myCV[index]->setDensityCV();
    }
  }
  return;
};

/*****************************************************/
/************ EK Jacobian Equations ***************/
/*****************************************************/
void PV::ekJ(void){
  /*docJtphil(1);
  docJpphil(1);
  docJphil(1);
  docJnphil(1);
```

```
    docJbphil(1);
    */
    docJphilz(1);
    docJhpz(1);
    /*
    docJthp(1);
    docJphp(1);
    docJhp(1);
    docJnhp(1);
    docJbhp(1);
    */
    //ekJeps();
    //   ekJhp();
    ekJhpz();
    ekJphisz();
    ekJphilz();
    //   ekJphis();
    //ekJphil();
    return;
};

void PV::ekJeps(void){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      //          Jacobptr[JRow][epsc] = Jacobptr[JRow][epsc] + myCV[x]->
          ekJeps();
    }
  }
  return;
};

void PV::ekJhpz(void){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      myCV[index]->ekJhp(JRow);
      myCV[index]->ekJxNhp(JRow);
      myCV[index]->ekJyNhp(JRow);
      myCV[index]->ekJxyNhp(JRow);
    }
  }
  return;
};

void PV::ekJphisz(void){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      myCV[index]->ekJphis(JRow);
```

```
      myCV[index]->ekJxNphis(JRow);
      myCV[index]->ekJyNphis(JRow);
      myCV[index]->ekJxyNphis(JRow);
    }
  }
  return;
};

void PV::ekJphilz(void){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      myCV[index]->ekJphil(JRow);
      myCV[index]->ekJxNphil(JRow);
      myCV[index]->ekJyNphil(JRow);
      myCV[index]->ekJxyNphil(JRow);
    }
  }
  return;
};
/*
void PV::ekJphis(void){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      myCV[index]->ekJphis(JRow);
    }
  }
  return;
};

void PV::ekJphil(void){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      myCV[index]->ekJphil(JRow);
    }
  }
  return;
};
*/
/*****************************************************/
/************ DOC Jacobian Equations ***************/
/*****************************************************/
void PV::docJ(void){
  docJphilz(1);
  docJhpz(1);
  /*
  docJthp(1);
  docJphp(1);
```

```
  docJhp(1);
  docJnhp(1);
  docJbhp(1);
  */
  docJphisz();
  return;
};

/*****************************************************************/
/********* DOC Jacobian Phil ***********************************/
/*****************************************************************/
void PV::docJphilz(double scale){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      myCV[index]->myDOCJphil(JRow,scale);
      myCV[index]->myDOCJxNphil(JRow,scale);
      myCV[index]->myDOCJyNphil(JRow,scale);
      myCV[index]->myDOCJxyNphil(JRow,scale);
    }
  }
  return;
};

/*****************************************************************/
/********* DOC Jacobian HP  ************************************/
/*****************************************************************/
void PV::docJhpz(double scale){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      myCV[index]->myDOCJhp(JRow,scale);
      myCV[index]->myDOCJxNhp(JRow,scale);
      myCV[index]->myDOCJyNhp(JRow,scale);
      myCV[index]->myDOCJxyNhp(JRow,scale);
    }
  }
  return;
};

/*****************************************************************/
/********* DOC Jacobian Phis ***********************************/
/*****************************************************************/
void PV::docJphisz(void){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      myCV[index]->myDOCJphis(JRow);
      myCV[index]->myDOCJxNphis(JRow);
      myCV[index]->myDOCJyNphis(JRow);
```

```
        myCV[index]->myDOCJxyNphis(JRow);
      }
    }
    return;
};

/*************************************************************/
/************* doc Jacobian phis terms ******************/
/*************************************************************/
/*
void PV::docJtphis(void){
  if(myCV[UL]->exist){
    Jacobptr[JRow][jtphisc] += myCV[UL]->myJyphis();
  }
  if(myCV[UR]->exist){
    Jacobptr[JRow][jtphisc] += myCV[UR]->myJyphis();
  }
  return;
};

void PV::docJpphis(void){
  if(myCV[UL]->exist){
    Jacobptr[JRow][jpphisc] += myCV[UL]->myJxphis();
  }
  if(myCV[LL]->exist){
    Jacobptr[JRow][jpphisc] += myCV[LL]->myJxphis();
  }
  return;
};

void PV::docJphis(void){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      Jacobptr[JRow][jphisc] += myCV[index]->myJphis();
    }
  }
  return;
};

void PV::docJnphis(void){
  if(myCV[UR]->exist){
    Jacobptr[JRow][jnphisc] += myCV[UR]->myJxphis();
  }
  if(myCV[LR]->exist){
    Jacobptr[JRow][jnphisc] += myCV[LR]->myJxphis();
  }
  return;
```

```
};

void PV::docJbphis(void){
  if(myCV[LL]->exist){
    Jacobptr[JRow][jbphisc] += myCV[LL]->myJyphis();
  }
  if(myCV[LR]->exist){
    Jacobptr[JRow][jbphisc] += myCV[LR]->myJyphis();
  }
  return;
};
*/
/*****************************************************************/
/******** DOC Jacobian Phil ***************************************/
/*****************************************************************/
/*
void PV::docJtphil(double scale){
  if(myCV[UL]->exist){
    Jacobptr[JRow][jtphilc] += scale*myCV[UL]->myJyphil();
  }
  if(myCV[UR]->exist){
    Jacobptr[JRow][jtphilc] += scale*myCV[UR]->myJyphil();
  }
  return;
};

void PV::docJpphil(double scale){
  if(myCV[UL]->exist){
    Jacobptr[JRow][jpphilc] += scale*myCV[UL]->myJxphil();
  }
  if(myCV[LL]->exist){
    Jacobptr[JRow][jpphilc] += scale*myCV[LL]->myJxphil();
  }
  return;
};

void PV::docJphil(double scale){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      Jacobptr[JRow][jphilc] += scale*myCV[index]->myJphil();
    }
  }
  return;
};

void PV::docJnphil(double scale){
  if(myCV[UR]->exist){
```

```
    Jacobptr[JRow][jnphilc] += scale*myCV[UR]->myJxphil();
  }
  if(myCV[LR]->exist){
    Jacobptr[JRow][jnphilc] += scale*myCV[LR]->myJxphil();
  }
  return;
};

void PV::docJbphil(double scale){
  if(myCV[LL]->exist){
    Jacobptr[JRow][jbphilc] += scale*myCV[LL]->myJyphil();
  }
  if(myCV[LR]->exist){
    Jacobptr[JRow][jbphilc] += scale*myCV[LR]->myJyphil();
  }
  return;
};
*/
/* ********************************************************** */
/* ************* doc Jacobian hp terms ********************** */
/* ********************************************************** */
/*
void PV::docJthp(double scale){
  if(myCV[UL]->exist){
    Jacobptr[JRow][jthpc] += (scale*myCV[UL]->myDOCJyhp(1))/Vptr[thpc];
  }
  if(myCV[UR]->exist){
    Jacobptr[JRow][jthpc] += (scale*myCV[UR]->myDOCJyhp(1))/Vptr[thpc];
  }
  return;
};

void PV::docJphp(double scale){
  if(myCV[UL]->exist){
    Jacobptr[JRow][jphpc] += (scale*myCV[UL]->myDOCJxhp(1))/Vptr[phpc];
  }
  if(myCV[LL]->exist){
    Jacobptr[JRow][jphpc] += (scale*myCV[LL]->myDOCJxhp(1))/Vptr[phpc];
  }
  return;
};

void PV::docJhp(double scale){
  for(int x = 0; x < NUM_CVperPV; x++){
    if(myCV[x]->exist){
      Jacobptr[JRow][jhpc] += (scale*myCV[x]->myDOCJhp(1))/Vptr[hpc];
    }
```

```
  }
  return;
};

void PV::docJnhp(double scale){
  if(myCV[UR]->exist){
    Jacobptr[JRow][jnhpc] += (scale*myCV[UR]->myDOCJxhp(1))/Vptr[nhpc];
  }
  if(myCV[LR]->exist){
    Jacobptr[JRow][jnhpc] += (scale*myCV[LR]->myDOCJxhp(1))/Vptr[nhpc];
  }
  return;
};

void PV::docJbhp(double scale){
  if(myCV[LL]->exist){
    Jacobptr[JRow][jbhpc] += (scale*myCV[LL]->myDOCJyhp(1))/Vptr[bhpc];
  }
  if(myCV[LR]->exist){
    Jacobptr[JRow][jbhpc] += (scale*myCV[LR]->myDOCJyhp(1))/Vptr[bhpc];
  }
  return;
};
*/

/*****************************************************/
/********* Material Balance Jacobian Equations *******/
/*****************************************************/
void PV::mbJ(void){
  //   mbJthp();
  // mbJphp();
   //    mbJhp();
  // mbJnhp();
  //mbJbhp();
  //mbJbnhp();
  //mbJbphp();
  //mbJtnhp();
  //mbJtphp();
  mbJhpz();
  /*
  docJtphil(mbScale);
  docJpphil(mbScale);
  docJphil(mbScale);
  docJnphil(mbScale);
  docJbphil(mbScale);
  */
  docJphilz(mbScale);
```

```
  docJhpz(mbScale);
  /*
  docJthp(mbScale);
  docJphp(mbScale);
  docJhp(mbScale);
  docJnhp(mbScale);
  docJbhp(mbScale);
  */
  return;
};

void PV::mbJhpz(void){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      //      printf("jhpc = %d for index = %d\n",jhpc,index);
       myCV[index]->myMBJhp(JRow);
       myCV[index]->myMBJxNhp(JRow);
       myCV[index]->myMBJyNhp(JRow);
       myCV[index]->myMBJxyNhp(JRow);
    }
  }
  return;
};
/*
void PV::mbJtnhp(void){
  if(myCV[UR]->exist){
    Jacobptr[JRow][jtnhpc] += myCV[UR]->myJxyhp();
  }
  return;
};

void PV::mbJtphp(void){
  if(myCV[UL]->exist){
    Jacobptr[JRow][jtphpc] += myCV[UL]->myJxyhp();
  }
  return;
};

void PV::mbJbnhp(void){
  if(myCV[LR]->exist){
    Jacobptr[JRow][jbnhpc] += myCV[LR]->myJxyhp();
  }
  return;
};

void PV::mbJbphp(void){
  if(myCV[LL]->exist){
```

```
    Jacobptr[JRow][jbphpc] += myCV[LL]->myJxyhp();
  }
  return;
};

void PV::mbJthp(void){
  if(myCV[UL]->exist){
    Jacobptr[JRow][jthpc] += myCV[UL]->myJyhp();
  }
  if(myCV[UR]->exist){
    Jacobptr[JRow][jthpc] += myCV[UR]->myJyhp();
  }
  return;
};

void PV::mbJphp(void){
  if(myCV[UL]->exist){
    Jacobptr[JRow][jphpc] += myCV[UL]->myJxhp();
  }
  if(myCV[LL]->exist){
    Jacobptr[JRow][jphpc] += myCV[LL]->myJxhp();
  }
  return;
};

void PV::mbJhp(void){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      Jacobptr[JRow][jhpc] += myCV[index]->myJhp();
    }
  }
  return;
};

void PV::mbJnhp(void){
  if(myCV[UR]->exist){
    Jacobptr[JRow][jnhpc] += myCV[UR]->myJxhp();
  }
  if(myCV[LR]->exist){
    Jacobptr[JRow][jnhpc] += myCV[LR]->myJxhp();
  }
  return;
};

void PV::mbJbhp(void){
  if(myCV[LL]->exist){
    Jacobptr[JRow][jbhpc] += myCV[LL]->myJyhp();
```

```
  }
  if(myCV[LR]->exist){
    Jacobptr[JRow][jbhpc] += myCV[LR]->myJyhp();
  }
  return;
};
*/
/*****************************************************/
/****** Continuity Equation Jacobian Equations ******/
/*****************************************************/
void PV::conteqJ(void){
   conteqJv_x();
  // conteqJv_y();
  // conteqJhp();
  // conteqJthp();
  // conteqJbhp();
  // conteqJnhp();
  // conteqJphp();
  return;
};

void PV::conteqJhp(void){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      Jacobptr[JRow][jhpc] += myCV[index]->contEqJhp();
    }
  }
  return;
};

void PV::conteqJv_x(void){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      if(Jacobptr[JRow][jv_xc] == 0){
        //      printf("Not using computed Jacobptr[%d][%d] = %f\n",JRow,
          jv_xc,Jacobptr[JRow][jv_xc]);
              Jacobptr[JRow][jv_xc] = 1;  /* September 27, 2007 */
      }else{
        //      printf("Using computed Jacobptr[%d][%d] = %.16f\n",JRow,
          jv_xc,Jacobptr[JRow][jv_xc]);
              Jacobptr[JRow][jv_xc] += myCV[index]->contEqJv_x();
        //      Jacobptr[JRow][jv_xc] = 1;
      }
    }
  }
  //Jacobptr[JRow][jv_xc] = 1;
  return;
```

```
};

void PV::conteqJv_y(void){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      Jacobptr[JRow][jv_yc] += myCV[index]->contEqJv_y();
    }
    if(Jacobptr[JRow][jv_yc] == 0){
      // printf("Not using computed Jacobptr[%d][%d] = %f\n",JRow,jv_yc,
         Jacobptr[JRow][jv_yc]);
      Jacobptr[JRow][jv_yc] = 1; /* September 27, 2007 */
      }else{
      // printf("Using computed Jacobptr[%d][%d] = %.16f\n",JRow,jv_yc,
         Jacobptr[JRow][jv_yc]);
    }
  }
  return;
};

void PV::conteqJbhp(void){
  if(myCV[LL]->exist){
    Jacobptr[JRow][jbhpc] += myCV[LL]->contEqJbhp();
  }
  if(myCV[LR]->exist){
    Jacobptr[JRow][jbhpc] += myCV[LR]->contEqJbhp();
  }
  return;
};

void PV::conteqJthp(void){
  if(myCV[UL]->exist){
    Jacobptr[JRow][jthpc] += myCV[UL]->contEqJthp();
  }
  if(myCV[UR]->exist){
    Jacobptr[JRow][jthpc] += myCV[UR]->contEqJthp();
  }
  return;
};

void PV::conteqJnhp(void){
  if(myCV[UR]->exist){
    Jacobptr[JRow][jnhpc] += myCV[UR]->contEqJnhp();
  }
  if(myCV[LR]->exist){
    Jacobptr[JRow][jnhpc] += myCV[LR]->contEqJnhp();
  }
  return;
```

```
};

void PV::conteqJphp(void){
  if(myCV[UL]->exist){
    Jacobptr[JRow][jphpc] += myCV[UL]->contEqJphp();
  }
  if(myCV[LL]->exist){
    Jacobptr[JRow][jphpc] += myCV[LL]->contEqJphp();
  }
  return;
};



/*****************************************************/
/*** Conservation of Momentum Y Jacobian Equations ***/
/*****************************************************/
void PV::comv_yJ(void){
  comv_yJv_y();
  // comv_yJhp();
  //comv_yJp();
  return;
};

void PV::comv_yJv_y(void){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      //  Jacobptr[JRow][jv_yc] += myCV[index]->comv_yJv_y();
        Jacobptr[JRow][jv_yc] = 1.0;
    }
  }
  return;
};

void PV::comv_yJhp(void){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
       Jacobptr[JRow][jhpc] += myCV[index]->comv_yJhp();
       // Jacobptr[JRow][jhpc] = 1->45;
    }
  }
  return;
};

void PV::comv_yJp(void){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
```

```
      //    Jacobptr[JRow][jpc] += 1.45;
    }
  }
  return;
};

/*****************************************************/
/*** Conservation of Momentum X Jacobian Equations ***/
/*****************************************************/
void PV::comv_xJ(void){
  // comv_xJv_x();
  // comv_xJhp();
  comv_xJp();
  return;
};

void PV::comv_xJv_x(void){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      Jacobptr[JRow][jv_xc] += myCV[index]->comv_xJv_x();
      //Jacobptr[JRow][jv_xc] += 1.45;
    }
  }
  return;
};

void PV::comv_xJhp(void){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
      Jacobptr[JRow][jhpc] += myCV[index]->comv_xJhp();
      // Jacobptr[JRow][jhpc] += 1.45;
    }
  }
  return;
};

void PV::comv_xJp(void){
  for(int index = 0; index < NUM_CVperPV; index++){
    if(myCV[index]->exist){
       Jacobptr[JRow][jpc] = 1.0;
    }
  }
  return;
};
```

Listing B.11. ORURPV.h

```
/* @author James Geraci
 */
#ifndef _ORURPV_H
#define _ORURPV_H
#include "PV.h"

class ORURPV :
        public PV
{
public:
        ORURPV(void);
        ORURPV(volumeSpatialData*, matrixData*, volumeChemData*,
            electrodeSizeData*, PVIConditions*);
public:
        ~ORURPV(void);

protected:
void computeIVdoc(void);
};

#endif
```

Listing B.12. ORURPV.cpp

```
/*
 * @author James Geraci
 * creates an Outer Right Upper Right PV
 */
#include "StdAfx.h"
#include "ORURPV.h"

ORURPV::ORURPV(void)
{
}

ORURPV::ORURPV(volumeSpatialData* sData, matrixData* mD, volumeChemData*
    vcD,
                    electrodeSizeData* esData, PVIConditions* pviC):PV(sData
                        , mD, vcD, esData, pviC){
  myCV[UL]->exist = false;
  myCV[UR]->exist = false;
  myCV[LL]->exist = true;
  myCV[LR]->exist = false;
}

ORURPV::~ORURPV(void)
{
}

void ORURPV::computeIVdoc(void){
  for(int x = 0; x < 4; x++){
    if(myCV[x]->exist){
      //        iVptr[philc][0] = iVptr[philc][0] + myCV[x]->setdocCV() +
          mySign*myCV[x]->inCurrentCV();
       iVptr[JRow+1][0] = iVptr[philc][0] + myCV[x]->setdocCV() + mySign*
          myCV[x]->inCurrentCV();
    }
  }
  return;
};
```

Listing B.13. OLPV.h

```
/* @author James Geraci
 */

#ifndef _OLPV_H
#define _OLPV_H
#include "PV.h"

class OLPV :
        public PV
{
public:
        OLPV(void);
        OLPV(volumeSpatialData*,matrixData*,volumeChemData*,
            electrodeSizeData*,PVIConditions*);
public:
        ~OLPV(void);

protected:
};

#endif
```

Listing B.14. OLPV.cpp

```
/*
 * @author James Geraci
 * creates an outer left PV
 */
#include "StdAfx.h"
#include "OLPV.h"

OLPV::OLPV(void)
{
}

OLPV::OLPV(volumeSpatialData* sData, matrixData* mD, volumeChemData* vcD,
                    electrodeSizeData* esData, PVIConditions* pviC):PV(sData
                        , mD, vcD, esData, pviC){
        myCV[UL]->exist = false;
        myCV[UR]->exist = true;
        myCV[LL]->exist = false;
        myCV[LR]->exist = true;
}

OLPV::~OLPV(void)
{
}
```

Listing B.15. OLLLPV.h

```
/* @author James Geraci
 */
#ifndef _OLLLPV_H
#define _OLLLPV_H
#include "PV.h"

class OLLLPV :
        public PV
{
public:
        OLLLPV(void);
        OLLLPV(volumeSpatialData*, matrixData*, volumeChemData*,
            electrodeSizeData*, PVIConditions*);
public:
        ~OLLLPV(void);

protected:
};

#endif
```

Listing B.16. OLLLPV.cpp

```
/*
 * @author James Geraci
 * creates an outer left lower left PV
 */
#include "StdAfx.h"
#include "OLLLPV.h"

OLLLPV::OLLLPV(void)
{
}

OLLLPV::OLLLPV(volumeSpatialData* sData, matrixData* mD, volumeChemData*
    vcD,
                    electrodeSizeData* esData, PVIConditions* pviC):PV(sData
                        , mD, vcD, esData, pviC){
        myCV[UL]->exist = false;
        myCV[UR]->exist = true;
        myCV[LL]->exist = false;
        myCV[LR]->exist = false;
}

OLLLPV::~OLLLPV(void)
{
}
```

Listing B.17. ORLRPV.h

```
/* @author James Geraci
 */
#ifndef _ORLRPV_H
#define _ORLRPV_H
#include "PV.h"

class ORLRPV :
        public PV
{
public:
        ORLRPV(void);
        ORLRPV(volumeSpatialData*, matrixData*, volumeChemData*,
            electrodeSizeData*, PVIConditions*);
public:
        ~ORLRPV(void);

protected:
};

#endif
```

Listing B.18. ORLRPV.cpp

```
/*
 * @author James Geraci
 * creates an Outer Right Lower Right PV
 */
#include "StdAfx.h"
#include "ORLRPV.h"

ORLRPV::ORLRPV(void)
{
}

ORLRPV::ORLRPV(volumeSpatialData* sData, matrixData* mD, volumeChemData*
    vcD,
                    electrodeSizeData* esData, PVIConditions* pviC):PV(sData
                        , mD, vcD, esData, pviC){
        myCV[UL]->exist = true;
        myCV[UR]->exist = false;
        myCV[LL]->exist = false;
        myCV[LR]->exist = false;
}

ORLRPV::~ORLRPV(void)
{
}
```

Listing B.19. UPV.h

```
/* @author James Geraci
 */
#ifndef _UPV_H
#define _UPV_H
#include "PV.h"

class UPV :
        public PV
{
public:
        UPV(void);
        UPV(volumeSpatialData*,matrixData*,volumeChemData*,
            electrodeSizeData*,PVIConditions*);
public:
        ~UPV(void);
protected:
void computeIVdoc(void);
};
#endif
```

Listing B.20. UPV.cpp

```
/*
 * @author James Geraci
 * creates an Upper PV
 */
#include "StdAfx.h"
#include "UPV.h"

UPV::UPV(void)
{
}

UPV::UPV(volumeSpatialData* sData, matrixData* mD, volumeChemData* vcD,
                    electrodeSizeData* esData, PVIConditions* pviC):PV(sData
                        , mD, vcD, esData, pviC){
        myCV[UL]->exist = false;
        myCV[UR]->exist = false;
        myCV[LL]->exist = true;
        myCV[LR]->exist = true;
}

UPV::~UPV(void)
{
}
void UPV::computeIVdoc(void){
  for(int x = 0; x < 4; x++){
    if(myCV[x]->exist){
      //      iVptr[philc][0] = iVptr[philc][0] + myCV[x]->setdocCV() +
          mySign*myCV[x]->inCurrentCV();
      iVptr[JRow+1][0] += myCV[x]->setdocCV() + mySign*myCV[x]->inCurrentCV
          ();
    }
  }
  return;
};
```

Listing B.21. ORPV.h

```
/* @author James Geraci
 */
#ifndef _ORPV_H
#define _ORPV_H
#include "PV.h"

class ORPV :
        public PV
{
public:
        ORPV(void);
        ORPV(volumeSpatialData*,matrixData*,volumeChemData*,
            electrodeSizeData*,PVIConditions*);
public:
        ~ORPV(void);

protected:

};

#endif
```

Listing B.22. ORPV.cpp

```
/*
 * @author James Geraci
 * creates an Outer Right PV
 */
#include "StdAfx.h"
#include "ORPV.h"

ORPV::ORPV(void)
{
}

ORPV::ORPV(volumeSpatialData* sData, matrixData* mD, volumeChemData* vcD,
                    electrodeSizeData* esData, PVIConditions* pviC):PV(sData
                        , mD, vcD, esData, pviC){
        myCV[UL]->exist = true;
        myCV[UR]->exist = false;
        myCV[LL]->exist = true;
        myCV[LR]->exist = false;
}

ORPV::~ORPV(void)
{
}
```

Listing B.23. OLULPV.h

```
/* @author James Geraci
 */
#ifndef _OLULPV_H
#define _OLULPV_H
#include "PV.h"

class OLULPV :
        public PV
{
 public:
  OLULPV(void);
  OLULPV(volumeSpatialData*,matrixData*,volumeChemData*,electrodeSizeData*,
      PVIConditions*);
 public:
  ~OLULPV(void);
 protected:
  void computeIVdoc(void);
};
#endif
```

Listing B.24. OLULPV.cpp

```
/*
 * @author James Geraci
 * creates an Outer Left Upper Left PV
 */
#include "StdAfx.h"
#include "OLULPV.h"

OLULPV::OLULPV(void)
{
}

OLULPV::OLULPV(volumeSpatialData* sData, matrixData* mD, volumeChemData*
    vcD,
                    electrodeSizeData* esData, PVIConditions* pviC):PV(sData
                        , mD, vcD, esData, pviC){
        myCV[UL]->exist = false;
        myCV[UR]->exist = false;
        myCV[LL]->exist = false;
        myCV[LR]->exist = true;
}

OLULPV::~OLULPV(void)
{
}

void OLULPV::computeIVdoc(void){
        for(int index = 0; index < 4; index++){
                if(myCV[index]->exist){
                  //                iVptr[philc][0] += myCV[index]->setdocCV
                      ()
                  iVptr[JRow+1][0] += myCV[index]->setdocCV()
                    + mySign*myCV[index]->inCurrentCV();
                }
        }
        return;
};
```

Listing B.25. BPV.h

```
/**
 * BPV.h  Bottom Potential Volume
 * Generic to any region
 * @author James Geraci
 *
 */

#ifndef _BPV_H
#define _BPV_H
#include "PV.h"

class BPV :
        public PV
{
public:
        BPV(void);
        BPV(volumeSpatialData*,matrixData*,volumeChemData*,
            electrodeSizeData*,PVIConditions*);
public:
        ~BPV(void);

protected:

};
#endif
```

Listing B.26. BPV.cpp

```
/**
 * BPV.cpp  Bottom Row PV
 * Generic to all regions
 * @author James Geraci
 *
 */
#include "StdAfx.h"
#include "BPV.h"

BPV::BPV(void)
{
}

BPV::BPV(volumeSpatialData* sData, matrixData* mD,
                volumeChemData* vcD, electrodeSizeData* esData,
                PVIConditions* pviC):PV(sData, mD, vcD, esData, pviC){
        myCV[UL]->exist = true;
        myCV[UR]->exist = true;
        myCV[LL]->exist = false;
        myCV[LR]->exist = false;
}

BPV::~BPV(void)
{
}
```

Listing B.27. CV.h

```
/*
 *
 * @author James Geraci
 *
 */
#ifndef _CV_H
#define _CV_H
#include "StdAfx.h"
class FV;

class CV
{
public:
        CV(void);
public:
        ~CV(void);

        double returnmbScale(void);
        void setpvVolume(double);
        virtual void setupPressurePointers(double) = 0;
        virtual void setupConcentrationPointers(double) = 0;
        void initialPressure(void);
        virtual void computeAVEyPressures(void) = 0;
        virtual void computeAVExPressures(void) = 0;
        virtual void computeAVEhpy(void) = 0;
        virtual void computeAVEhpx(void) = 0;
        virtual void myNvalues(void) = 0;
        void setupMyNbrHood(void);
        bool exist;

        //      double Qscalex;
        //      double Qscaley;
        double Qvalue(int);
        double pQvalue(int);
        //      void setQscalex(void);
        //      void setQscaley(void);
        FV* myFV;
        double mydx;
        double mydy;
        double mydz;

        double Qdx;
        double Qdy;
        double Qdz;
```

```
        double mydx2;
        double invmydx;
        double invmydx2;
        double mydxsqrd;

        double mydy2;
        double invmydy;
        double invmydy2;
        double mydysqrd;

        double mydz2;
        double invmydz;
        double invmydz2;
        double mydzsqrd;


        void setPosition(int);
        double dhdtCV;
        double dhdt;
        double ddensitydt;
        double dmv_xdt;
        double dmv_xdtCV;
        double dmv_ydt;
        double dmv_ydtCV;
        double dp_yCV;
        double pdensity;
        double density;
        double docCV;

        double ekCV;

        void computedxSize(void);
        void computedySize(void);
        void computedzSize(void);
        void computemyVolume(void);
        double returnmyVolume(void);
        void setDensityCV(void);
//  MB
        double setdhdt(void);
        double setddensitydt(void);
        double setdhdtCV(void);
        double setDiffusionCV(void);
        double setConvectionCV(void);



        //double myJhp(void);
```

```
//double myJxhp(void);
//double myJyhp(void);
//double myJxyhp(void);

void myMBJhp(int);
void myMBJxNhp(int);
void myMBJyNhp(int);
void myMBJxyNhp(int);
double divME(double,double);
double divMEx(double,double,double);
double divMEy(double,double,double);
double divxN(double,double);
double divxNx(double,double,double);
double divxNy(double,double,double);
double divyN(double,double);
double divyNx(double,double,double);
double divyNy(double,double,double);
double divxyN(double,double);
double divxyNx(double,double,double);
double divxyNy(double,double,double);
double myFdx(void);
double myFdy(void);

// DOC
double setdocCV(void);
double setdocCVjl(void);
double setdocCVjlx(void);
double setdocCVjly(void);
double setdocCVse(void);
double myFjlx(void);
double myFsex(void);
double myFjly(void);
double myFsey(void);
double myJphil(void);
//double myDOCJphil(void);
//      double myJyphil(void);
//double myDOCJhp(int);
//double myDOCJxhp(int);
//double myDOCJyhp(int);
void myDOCJhp(int,double);
void myDOCJxNhp(int,double);
void myDOCJyNhp(int,double);
void myDOCJxyNhp(int,double);
void myDOCJphil(int,double);
void myDOCJxNphil(int,double);
void myDOCJyNphil(int,double);
void myDOCJxyNphil(int,double);
```

```
        void myDOCJphis(int);
        void myDOCJxNphis(int);
        void myDOCJyNphis(int);
        void myDOCJxyNphis(int);
        double inCurrentCV(void);    // Used only with the top row
// EK
        double setekCV(void);
        double myArea(void);
        double expTerm(void);
        double ekJeps(void);
        void ekJhp(int);
        void ekJxNhp(int);
        void ekJyNhp(int);
        void ekJxyNhp(int);

        void ekJphis(int);
        void ekJxNphis(int);
        void ekJyNphis(int);
        void ekJxyNphis(int);

        void ekJphil(int);
        void ekJxNphil(int);
        void ekJyNphil(int);
        void ekJxyNphil(int);

        // Conservation of Momentum X
        double setdmv_xdt(void);
        double setdmv_xdtCV(void);
        double comv_XsetAdvection_xFaceCV(void);
        double comv_XsetAdvection_yFaceCV(void);
        double comv_XsetPressure_xFaceCV(void);
        virtual double setdpdxCV(void) = 0;
        double comv_XsetViscosity_xFaceCV(void);
        double comv_XsetViscosity_yFaceCV(void);
        double setGravitySource_xCV(void);
        double comv_xJhp(void);
        virtual double comv_xJp(void) = 0;
        double comv_xJv_x(void);

        // Conservation of Momentum Y
        double setdmv_ydt(void);
        double setdmv_ydtCV(void);
        double comv_YsetAdvection_xFaceCV(void);
        double comv_YsetAdvection_yFaceCV(void);
        double comv_YsetPressure_yFaceCV(void);
        virtual double setdpdyCV(void) = 0;
        double comv_YsetViscosity_xFaceCV(void);
```

```
        double comv_YsetViscosity_yFaceCV(void);
        double setGravitySource_yCV(void);
        double comv_yJhp(void);
        double comv_yJv_y(void);

        // Continuity Equation
        double setContEqCV(void);
        double contEqJv_x(void);
        double contEqJv_y(void);
        double contEqJhp(void);
        double contEqJthp(void);
        double contEqJbhp(void);
        double contEqJphp(void);
        double contEqJnhp(void);

protected:
        double myxNorm;
        double myyNorm;
        double mydxSide;
        double mydySide;
        double nbrdxSide;
        double nbrdySide;

        int myPosition;
        double gls;
        double gsl;
        double overp;
        double myexpTerm;
        double myFVdx;
        double mzFVdy;
        double myVolume;
        double pvVolume;
        double myPVvolumeFraction;
        double Area;

        double* outerXdensityAVE;
        double* outerYdensityAVE;
public:
                // Data Matrix Pointers
        double* fluxVptr;
        double* pVptr;
        double* Vptr;
        double** iVptr;
        double** Jacobptr;
        double* avePressures_y;
        double* avePressures_x;
        double* AVEhp_x;
```

```
    double* AVEhp_y;
    double* Pptr[4];
    double* Cptr[4];

    // Position information for Cell
    int myColumn;
    int myRow;
    int myZrow;
    int JRow;
    int JRowBuffer;

// Positioning Data
    int rowLength;
    int maxPvIndex;

    // FV grid Information
    int numFVColumns;
    int numFVRows;

    //  Used in Calculating Addresses
    int prevV;
    int presV;
    int nextV;

    int topV;
    int lowerV;
    // present volume's properties
    int hpc;
    int phisc;
    int philc;
    int pc;
    int v_yc;
    int v_xc;
    // previous volume's properties
    int phpc;
    int pphisc;
    int pphilc;
    int ppc;
    int pv_yc;
    int pv_xc;
    // next volume's properties
    int nhpc;
    int nphisc;
    int nphilc;
    int npc;
    int nv_yc;
    int nv_xc;
```

```
// top previous volume's properties
        int tphpc;
        int tpphisc;
        int tpphilc;
        int tppc;
        int tpv_yc;
        int tpv_xc;
        // top next volume's properties
        int tnhpc;
        int tnphisc;
        int tnphilc;
        int tnpc;
        int tnv_yc;
        int tnv_xc;
        // top volume's properties
        int thpc;
        int tphisc;
        int tphilc;
        int tpc;
        int tv_yc;
        int tv_xc;
// lower previous volume's properties
        int bphpc;
        int bpphisc;
        int bpphilc;
        int bppc;
        int bpv_yc;
        int bpv_xc;
        // lower next volume's properties
        int bnhpc;
        int bnphisc;
        int bnphilc;
        int bnpc;
        int bnv_yc;
        int bnv_xc;
// lower volume's properties
        int bhpc;
        int bphisc;
        int bphilc;
        int bpc;
        int bv_yc;
        int bv_xc;



// present volume's properties for Jacobian Indexing
        int jhpc;
```

```
        int jphisc;
        int jphilc;
        int jpc;
        int jv_yc;
        int jv_xc;
// previous volume's properties for Jacobian Indexing
        int jphpc;
        int jpphisc;
        int jpphilc;
        int jppc;
        int jpv_yc;
        int jpv_xc;
// next volume's properties for Jacobian Indexing
        int jnhpc;
        int jnphisc;
        int jnphilc;
        int jnpc;
        int jnv_yc;
        int jnv_xc;
// top previous volume's properties for Jacobian Indexing
        int jtphpc;
        int jtpphisc;
        int jtpphilc;
        int jtppc;
        int jtpv_yc;
        int jtpv_xc;
// top next volume's properties for Jacobian Indexing
        int jtnhpc;
        int jtnphisc;
        int jtnphilc;
        int jtnpc;
        int jtnv_yc;
        int jtnv_xc;
// top volume's properties for Jacobian Indexing
        int jthpc;
        int jtphisc;
        int jtphilc;
        int jtpc;
        int jtv_yc;
        int jtv_xc;
// lower previous volume's properties for Jacobian Indexing
        int jbphpc;
        int jbpphisc;
        int jbpphilc;
        int jbppc;
        int jbpv_yc;
        int jbpv_xc;
```

```
// lower next volume's properties for Jacobian Indexing
        int jbnhpc;
        int jbnphisc;
        int jbnphilc;
        int jbnpc;
        int jbnv_yc;
        int jbnv_xc;
// lower volume's properties for Jacobian Indexing
        int jbhpc;
        int jbphisc;
        int jbphilc;
        int jbpc;
        int jbv_yc;
        int jbv_xc;

        /* Important for Quadrature */
        /* xNorm direction values */
        int myxNhpc;
        int myxNphisc;
        int myxNphilc;
        int myxNpc;
        int myxNv_yc;
        int myxNv_xc;
        /* yNorm direction values */
        int myyNhpc;
        int myyNphisc;
        int myyNphilc;
        int myyNpc;
        int myyNv_yc;
        int myyNv_xc;
        /* xyNorm direction values */
        int myxyNhpc;
        int myxyNphisc;
        int myxyNphilc;
        int myxyNpc;
        int myxyNv_yc;
        int myxyNv_xc;

        int myQnbrs[24];

        /* Important for Quadrature */
        /* jxNorm direction values */
        int jxNhpc;
        int jxNphisc;
        int jxNphilc;
        int jxNpc;
        int jxNv_yc;
```

```
        int jxNv_xc;
        /* yNorm direction values */
        int jyNhpc;
        int jyNphisc;
        int jyNphilc;
        int jyNpc;
        int jyNv_yc;
        int jyNv_xc;
        /* xyNorm direction values */
        int jxyNhpc;
        int jxyNphisc;
        int jxyNphilc;
        int jxyNpc;
        int jxyNv_yc;
        int jxyNv_xc;

        double myQhpc;
        double myQphisc;
        double myQphilc;
        double myQpc;
        double myQv_yc;
        double myQv_xc;
};

#endif
```

Listing B.28. CV.cpp

```
/**
 * CV.cpp
 * @author James Geraci
 * Implements the equations COP, MB, EK which control
 * the change of the concentration, phil, and phis with time
 * also sets up the Jacobian
 *
 */
#include "StdAfx.h"
#include "CV.h"
#include <stdio.h>

CV::CV(void)
{
  myFV =(FV*) NULL;
   exist = true;
}

CV::~CV(void)
{
}

void CV::setupMyNbrHood(void){
        /* my values */
        myQnbrs[0] = hpc;
        myQnbrs[1] = philc;
        myQnbrs[2] = phisc;
        myQnbrs[3] = pc;
        myQnbrs[4] = v_yc;
        myQnbrs[5] = v_xc;

        /* xNorm direction values */
        myQnbrs[6] = myxNhpc;
        myQnbrs[7] = myxNphilc;
        myQnbrs[8] = myxNphisc;
        myQnbrs[9] = myxNpc;
        myQnbrs[10] = myxNv_yc;
        myQnbrs[11] = myxNv_xc;
        /* yNorm direction values */
        myQnbrs[12] = myyNhpc;
        myQnbrs[13] = myyNphilc;
        myQnbrs[14] = myyNphisc;
        myQnbrs[15] = myyNpc;
        myQnbrs[16] = myyNv_yc;
        myQnbrs[17] = myyNv_xc;
        /* xyNorm direction values */
```

```
        myQnbrs[18] = myxyNhpc;
        myQnbrs[19] = myxyNphilc;
        myQnbrs[20] = myxyNphisc;
        myQnbrs[21] = myxyNpc;
        myQnbrs[22] = myxyNv_yc;
        myQnbrs[23] = myxyNv_xc;
        return;
};

double CV::returnmbScale(void){
  return myFV->mbScale;
};

void CV::setPosition(int x){
  myPosition = x;
}

void CV::computedxSize(void){
  mydx = (myFV->returndx())/2;
  mydx2 = 2*mydx;
  invmydx = 1/mydx;
  invmydx2 = 1/mydx2;
  mydxsqrd = mydx*mydx;
  return;
}

void CV::computedySize(void){
  mydy = (myFV->returndy())/2;
  mydy2 = 2*mydy;
  invmydy = 1/mydy;
  invmydy2 = 1/mydy2;
  mydysqrd = mydy*mydy;
  return;
}

void CV::computedzSize(void){
  mydz = (myFV->returndz());
  mydz2 = 2*mydz;
  invmydz = 1/mydz;
  invmydz2 = 1/mydz2;
  mydzsqrd = mydz*mydz;
  return;
}

void CV::computemyVolume(void){
  myVolume = mydx*mydy*mydz;
  myFV->setCVvolume(myPosition,myVolume);
```

```
  return;
}

double CV::returnmyVolume(void){
  return myVolume;
}

void CV::setpvVolume(double vol){
  pvVolume = vol;
  myPVvolumeFraction = myVolume/pvVolume;
  myFV->setCVvolumeRatio(myPosition,myPVvolumeFraction);
  return;
}


void CV::initialPressure(void){
  *(Pptr[RIGHT]) = *(Pptr[TOP]);
  *(Pptr[LEFT]) = *(Pptr[TOP]);
  *(Pptr[BOTTOM]) = *(Pptr[TOP])
    - density*g_y*(1+myFV->returnVECy()*(Vptr[hpc]-myFV->hnot))*mydy;
  return;
}

void CV::setDensityCV(void){
  density = slope*(Vptr[hpc])+intercept;
  pdensity = slope*(pVptr[hpc])+intercept;
  return;
};

double CV::inCurrentCV(void){
  return mydx*I;
};

/************************************/
/***** Jacobian Derivate Values *****/
/************************************/
double CV::divME(double MPx,double MPy){
  return  divMEx(MPx,(double)1.0,(double)1.0) + divMEy(MPy,(double)1.0,(
     double)1.0);

}

double CV::divMEx(double MPx, double Qhp, double QxNhp){
  return  - MPx*(mydy*myxNorm*(mydxSide*myQscale/Qhp + nbrdxSide*myQxNscale
     /QxNhp))/myFV->returnQdx();
}
```

```
double CV::divMEy(double MPy, double Qhp, double QyNhp){
  return -MPy*(mydx*myyNorm*(mydySide*myQscale/Qhp + nbrdySide*myQyNscale/
     QyNhp))/myFV->returnQdy();
}
/*****/
/* xN */
/*****/
double CV::divxN(double MPx,double MPy){
  return divxNx(MPx,(double)1.0,(double)1.0) + divxNy(MPy,(double)1.0,(
     double)1.0);
}

double CV::divxNx(double MPx, double Qhp, double QxNhp){
  return -MPx*(mydy*myxNorm*(mydxSide*myQxNscale/Qhp + nbrdxSide*myQscale/
     QxNhp))/myFV->returnQdx();
}

double CV::divxNy(double MPy, double Qhp, double QyNhp){
return -MPy*(mydx*myyNorm*(mydySide*myQxNscale/Qhp + nbrdySide*myQxyNscale
   /QyNhp))/myFV->returnQdy();
}
/*****/
/* yN */
/*****/
double CV::divyN(double MPx, double MPy){
  return divyNx(MPx,(double)1.0,(double)1.0) + divyNy(MPy,(double)1.0,(
     double)1.0);
}

double CV::divyNx(double MPx, double Qhp, double QxNhp){
  return -MPx*(mydy*myxNorm*(mydxSide*myQyNscale/Qhp + nbrdxSide*
     myQxyNscale/QxNhp))/myFV->returnQdx();
}

double CV::divyNy(double MPy, double Qhp, double QyNhp){
  return -MPy*(mydx*myyNorm*(mydySide*myQyNscale/Qhp + nbrdySide*myQscale/
     QyNhp))/myFV->returnQdy();
}
/*****/
/* xyN */
/*****/
double CV::divxyN(double MPx,double MPy){
  return divxyNx(MPx,(double)1.0,(double)1.0) + divxyNy(MPy,(double)1.0,(
     double)1.0);
}

double CV::divxyNx(double MPx, double Qhp, double QxNhp){
```

```
  return - MPx*(mydy*myxNorm*(mydxSide*myQxyNscale/Qhp + nbrdxSide*
     myQyNscale/QxNhp))/myFV->returnQdx();
}

double CV::divxyNy(double MPy, double Qhp, double QyNhp){
  return - MPy*(mydx*myyNorm*(mydySide*myQxyNscale/Qhp + nbrdySide*
     myQxNscale/QyNhp))/myFV->returnQdy();
}
/*****************************************************************/
/********** Compute Quadrature Values ****************************/
/*****************************************************************/
double CV::Qvalue(int Q){
  return myQscale*Vptr[myQnbrs[Q]]
    + myQxNscale*Vptr[myQnbrs[Q+myQxNnbr]]
    + myQyNscale*Vptr[myQnbrs[Q+myQyNnbr]]
    + myQxyNscale*Vptr[myQnbrs[Q+myQxyNnbr]];
}

double CV::pQvalue(int dk){
return myQscale*pVptr[myQnbrs[dk]]
    + myQxNscale*pVptr[myQnbrs[dk+myQxNnbr]]
    + myQyNscale*pVptr[myQnbrs[dk+myQyNnbr]]
    + myQxyNscale*pVptr[myQnbrs[dk+myQxyNnbr]];
}
/*****************************************************************/
/****************** Work for EK **********************************/
/*****************************************************************/
double CV::setekCV(void){
  // overp = (Vptr[phisc] - Vptr[philc] - myFV->phi_eq);
  overp = (Qvalue(qphisc) - Qvalue(qphilc) - myFV->phi_eq);
  gls = ((myFV->als)*N*F)/(R*myFV->T);
  gsl = ((myFV->asl)*N*F)/(R*myFV->T);
  // ekCV = -myArea()*(myFV->io)*(Vptr[hpc]/myFV->hnot)*expTerm()*mydx*
     mydy;
  ekCV = -myArea()*(myFV->io)*(Qvalue(qhpc)/myFV->hnot)*expTerm()*mydx*mydy
     ;
  return ekCV;
};

double CV::myArea(void){
  if(I >= 0)  // discharge
    {
      Area = (myFV->Amax)*(myFV->soceta());
    }
  else{
    Area = (myFV->Amax)*(1-myFV->soceta());
  }
```

```
  return Area;
};

double CV::expTerm(void){
  myexpTerm = exp(gsl*overp) - exp(-gls*overp);
  return myexpTerm;
};

double CV::ekJeps(void){
  return 0;
};

void CV::ekJhp(int JR){
    Jacobptr[JR][jhpc] += -myArea()*((myFV->io*myQscale)/(myFV->hnot))*
       myexpTerm*mydx*mydy;
    return;
};

void CV::ekJxNhp(int JR){
  Jacobptr[JR][jxNhpc] += -myArea()*((myFV->io*myQxNscale)/(myFV->hnot))*
     myexpTerm*mydx*mydy;
  return;
};

void CV::ekJyNhp(int JR){
   Jacobptr[JR][jyNhpc] += -myArea()*((myFV->io*myQyNscale)/(myFV->hnot))*
      myexpTerm*mydx*mydy;
   return;
};

void CV::ekJxyNhp(int JR){
   Jacobptr[JR][jxyNhpc] += -myArea()*((myFV->io*myQxyNscale)/(myFV->hnot))
      *myexpTerm*mydx*mydy;
   return;
};

void CV::ekJphis(int JR){
  Jacobptr[JR][jphisc] += -myArea()*myFV->io*(Qvalue(qhpc)/myFV->hnot)*
     myQscale*(gsl*exp(gsl*overp)+gls*exp(-gls*overp))*mydx*mydy;
  return;
};

void CV::ekJxNphis(int JR){
  Jacobptr[JR][jphisc] += -myArea()*myFV->io*(Qvalue(qhpc)/myFV->hnot)*
     myQxNscale*(gsl*exp(gsl*overp)+gls*exp(-gls*overp))*mydx*mydy;
  return;
```

```
};

void CV::ekJyNphis(int JR){
  Jacobptr[JR][jphisc] += -myArea()*myFV->io*(Qvalue(qhpc)/myFV->hnot)*
     myQyNscale*(gsl*exp(gsl*overp)+gls*exp(-gls*overp))*mydx*mydy;
  return;
};

void CV::ekJxyNphis(int JR){
  Jacobptr[JR][jphisc] += -myArea()*myFV->io*(Qvalue(qhpc)/myFV->hnot)*
     myQxyNscale*(gsl*exp(gsl*overp)+gls*exp(-gls*overp))*mydx*mydy;
  return;
};

void CV::ekJphil(int JR){
  Jacobptr[JR][jphilc] += myArea()*myFV->io*(Qvalue(qhpc)/myFV->hnot)*
     myQscale*(gsl*exp(gsl*overp)+gls*exp(-gls*overp))*mydx*mydy;
  return;
};

void CV::ekJxNphil(int JR){
  Jacobptr[JR][jphilc] += myArea()*myFV->io*(Qvalue(qhpc)/myFV->hnot)*
     myQxNscale*(gsl*exp(gsl*overp)+gls*exp(-gls*overp))*mydx*mydy;
  return;
};

void CV::ekJyNphil(int JR){
  Jacobptr[JR][jphilc] += myArea()*myFV->io*(Qvalue(qhpc)/myFV->hnot)*
     myQyNscale*(gsl*exp(gsl*overp)+gls*exp(-gls*overp))*mydx*mydy;
  return;
};

void CV::ekJxyNphil(int JR){
  Jacobptr[JR][jphilc] += myArea()*myFV->io*(Qvalue(qhpc)/myFV->hnot)*
     myQxyNscale*(gsl*exp(gsl*overp)+gls*exp(-gls*overp))*mydx*mydy;
  return;
};


/*************************************************************/
/*****************  Work for DOC *****************************/
/*************************************************************/

double CV::setdocCV(void){
  docCV = setdocCVjl() + setdocCVse();
  return docCV;
};
```

```
double CV::setdocCVjl(void){
   return setdocCVjlx() + setdocCVjly();
};

double CV::setdocCVjlx(void){
  return myFjlx()*mydy;
};

double CV::setdocCVjly(void){
  return myFjly()*mydx;
};

double CV::setdocCVse(void){
  return myFsex()*mydy + myFsey()*mydx;
};

double CV::myFjly(void){
  return (myyNorm)*myFV->returnFjly(myPosition);
};

double CV::myFjlx(void){
  return myxNorm*myFV->returnFjlx(myPosition);
};

double CV::myFsey(void){
  return myyNorm*myFV->returnFsey(myPosition);
};

double CV::myFsex(void){
  return myxNorm*myFV->returnFsex(myPosition);
};

/*********************/
/*** DOC PHIL J ******/
/*********************/
void CV::myDOCJphil(int JR,double scale){
  Jacobptr[JR][jphilc] += scale*divME(myFV->returnKeffx(),myFV->returnKeffy
     ());
  return;
};

void CV::myDOCJxNphil(int JR, double scale){
  Jacobptr[JR][jxNphilc] += scale*divxN(myFV->returnKeffx(),myFV->
     returnKeffy());
  return;
};
```

```
void CV::myDOCJyNphil(int JR, double scale){
  Jacobptr[JR][jyNphilc] += scale*divyN(myFV->returnKeffx(),myFV->
     returnKeffy());
  return;
};

void CV::myDOCJxyNphil(int JR, double scale){
  Jacobptr[JR][jxyNphilc] += scale*divxyN(myFV->returnKeffx(),myFV->
     returnKeffy());
  return;
};

/**********************/
/***** DOC HP J ******/
/**********************/
void CV::myDOCJhp(int JR,double scale){
Jacobptr[JR][jhpc] += scale*(divMEx(myFV->returnKeffx(),(double)1.0,(double
    )1.0)
                        + divMEy(myFV->returnKeffy(),(double)1.0,(double)
                           1.0));
 return;
};

void CV::myDOCJxNhp(int JR, double scale){
  Jacobptr[JR][jxNhpc] += scale*(divxNx(myFV->returnKeffx(),(double)1.0,(
     double)1.0)
                                  + divxNy(myFV->returnKeffy(),(double)1.0,(
                                     double)1.0));
  return;
};

void CV::myDOCJyNhp(int JR, double scale){
  Jacobptr[JR][jyNhpc] += scale*(divyNx(myFV->returnKeffx(),(double)1.0,(
     double)1.0)
                                  + divyNy(myFV->returnKeffy(),(double)1.0,(
                                     double)1.0));
  return;
};

void CV::myDOCJxyNhp(int JR, double scale){
  Jacobptr[JR][jxyNhpc] += scale*(divxyNx(myFV->returnKeffx(),(double)1.0,(
     double)1.0)
                                   + divxyNy(myFV->returnKeffy(),(double)
                                      1.0,(double)1.0));
  return;
};
```

```
/*
// HP Jacobian Contributions
double CV::myDOCJhp(int JR){
  return mydy*(myFV->returnolsKscalex()/(myFV->returndx()))
    + mydx*myFV->returnolsKscaley()/(myFV->returndy());
};

double CV::myDOCJxhp(int JR){
  return -mydy*myFV->returnolsKscalex()/(myFV->returndx());
};

double CV::myDOCJyhp(int JR){
  return -mydx*myFV->returnolsKscaley()/(myFV->returndy());
};
*/
/**********************/
/*** DOC PHIS J ******/
/**********************/
void CV::myDOCJphis(int JR){
  Jacobptr[JR][jphisc] += divME(myFV->returnsigeffx(),myFV->returnsigeffy()
      );
  return;
};

void CV::myDOCJxNphis(int JR){
  Jacobptr[JR][jxNphisc] += divxN(myFV->returnsigeffx(),myFV->returnsigeffy
      ());
  return;
};

void CV::myDOCJyNphis(int JR){
  Jacobptr[JR][jyNphisc] += divyN(myFV->returnsigeffx(),myFV->returnsigeffy
      ());
  return;
};

void CV::myDOCJxyNphis(int JR){
  Jacobptr[JR][jxyNphisc] += divxyN(myFV->returnsigeffx(),myFV->
      returnsigeffy());
  return;
};

/*************************************************************/
/*****************  Time Derivatives for MB **************/
/*************************************************************/
double CV::setdhdt(void){
```

```
  dhdt = (Qvalue(qhpc) - pQvalue(qhpc))/dt;
  return dhdt;
};

double CV::setddensitydt(void){
  ddensitydt = slope*dhdt;
  return ddensitydt;
};

double CV::setdhdtCV(void){
  dhdtCV = myFV->returneps()*dhdt*mydx*mydy;
  return dhdtCV;
};

double CV::setDiffusionCV(void){
  return  myFdx()*mydy + myFdy()*mydx;
};

double CV::setConvectionCV(void){
  return 0;
};

void CV::myMBJhp(int JR){
  Jacobptr[JR][jhpc] += (mydx*mydy*myFV->returneps()*myQscale)/dt
    + divME(myFV->returnDeffx(),myFV->returnDeffy());
  return;
};

void CV::myMBJxNhp(int JR){
  Jacobptr[JR][jxNhpc] += (mydx*mydy*myFV->returneps()*myQxNscale)/dt
    + divxN(myFV->returnDeffx(),myFV->returnDeffy());
  return;
};

void CV::myMBJyNhp(int JR){
  Jacobptr[JR][jyNhpc] += (mydx*mydy*myFV->returneps()*myQyNscale)/dt
    + divyN(myFV->returnDeffx(),myFV->returnDeffy());
  return;
};

void CV::myMBJxyNhp(int JR){
  Jacobptr[JR][jxyNhpc] += (mydx*mydy*myFV->returneps()*myQxyNscale)/dt
    + divxyN(myFV->returnDeffx(),myFV->returnDeffy());
  return;
};

double CV::myFdx(void){
```

```
  return myxNorm*myFV->returnFdx(myPosition);
};

double CV::myFdy(void){
  return myyNorm*myFV->returnFdy(myPosition);
};

/*************************************************************/
/*************** Conservation of Momentum Y ***************/
/*************************************************************/
double CV::setdmv_ydt(void)
{
  dmv_ydt = (density*Vptr[v_yc] - pdensity*pVptr[v_yc])/dt;
  return dmv_ydt;
};

double CV::setdmv_ydtCV(void){
  dmv_ydtCV = dmv_ydt*mydx*mydy;
  return dmv_ydtCV;
};

double CV::comv_YsetAdvection_xFaceCV(void){
  return *outerXdensityAVE*myxNorm*mydy*Vptr[v_xc]*Vptr[v_yc];
};

double CV::comv_YsetAdvection_yFaceCV(void){
  return *outerYdensityAVE*myyNorm*mydy*Vptr[v_yc]*Vptr[v_yc];
};

double CV::comv_YsetPressure_yFaceCV(void){
  return myFV->returnFp_y(myPosition);
};

double CV::comv_YsetViscosity_xFaceCV(void){
  return myxNorm*mydy*Vptr[v_yc]*myFV->returnFvisc_x(myPosition);
};

double CV::comv_YsetViscosity_yFaceCV(void){
  return myyNorm*mydx*Vptr[v_yc]*myFV->returnFvisc_y(myPosition);
};

double CV::setGravitySource_yCV(void){
  return   -density*g_y*(1+myFV->returnVECy()*(Vptr[hpc] - myFV->hnot))*
     mydx*mydy;
};

double CV::comv_yJv_y(void){
```

```
  return (mydx*mydy*density)/dt;
};

double CV::comv_yJhp(void){
  return (mydx*mydy*Vptr[v_yc]*slope)/dt
    - g_y*mydx*mydy*
    (slope+2*myFV->returnVECy()*slope*Vptr[hpc]+intercept-slope*myFV->hnot)
      ;
};


/*************************************************************/
/*************** Conservation of Momentum X ***************/
/*************************************************************/
double CV::setdmv_xdt(void)
{
  dmv_xdt = (density*Vptr[v_xc] - pdensity*pVptr[v_xc])/dt;
  return dmv_xdt;
};

double CV::setdmv_xdtCV(void){
  dmv_xdtCV = dmv_xdt*mydx*mydy;
  return dmv_xdtCV;
};

double CV::comv_XsetAdvection_xFaceCV(void){
  return *outerXdensityAVE*myxNorm*mydy*Vptr[v_xc]*Vptr[v_xc];
};

double CV::comv_XsetAdvection_yFaceCV(void){
  return *outerYdensityAVE*myyNorm*mydy*Vptr[v_xc]*Vptr[v_yc];
};

double CV::comv_XsetPressure_xFaceCV(void){
  return myFV->returnFp_x(myPosition);
};

double CV::comv_XsetViscosity_xFaceCV(void){
  return myxNorm*mydy*Vptr[v_xc]*myFV->returnFvisc_x(myPosition);
};

double CV::comv_XsetViscosity_yFaceCV(void){
    return myyNorm*mydx*Vptr[v_xc]*myFV->returnFvisc_y(myPosition);
};

double CV::setGravitySource_xCV(void){
  return    density*g_x*(1+myFV->returnVECx()*(Vptr[hpc] - myFV->hnot))*mydx
```

```
      *mydy;
};

double CV::comv_xJv_x(void){
  return (mydx*mydy*density)/dt;
};

double CV::comv_xJhp(void){
  return (mydx*mydy*Vptr[v_xc]*slope)/dt
    - g_x*mydx*mydy*
    (slope + intercept + 2*myFV->returnVECy()*slope*(Vptr[hpc] - myFV->hnot
       ));
};



/*************************************************************/
/***************    Continuity Equation    *****************/
/*************************************************************/
double CV::setContEqCV(void){
  return ddensitydt*mydx*mydy
    + *outerXdensityAVE*mydy*myxNorm*Vptr[v_xc]
    + *outerYdensityAVE*mydx*myyNorm*Vptr[v_yc];
};

double CV::contEqJv_x(void){
  return *outerXdensityAVE*mydy*myxNorm;
};

double CV::contEqJv_y(void){
  return *outerYdensityAVE*mydx*myyNorm;
};

double CV::contEqJhp(void){
  return slope*mydx*mydy
    + (Vptr[v_xc]*myxNorm*mydy
       + Vptr[v_yc]*myyNorm*mydx)/2;
};

double CV::contEqJthp(void){
  return (myyNorm*Vptr[v_yc]*mydx)/2;
};

double CV::contEqJbhp(void){
  return (myyNorm*Vptr[v_yc]*mydx)/2;
};
```

```
double CV::contEqJphp(void){
  return (myxNorm*Vptr[v_xc]*mydy)/2;
};

double CV::contEqJnhp(void){
  return (myxNorm*Vptr[v_xc]*mydy)/2;
};
```

Listing B.29. ULCV.h

```
/*
 *
 * @author James Geraci
 *
 */
#ifndef _ULCV_H
#define _ULCV_H
#include "StdAfx.h"
#include "CV.h"
//class FV;

class ULCV :
        public CV
{
public:
        ULCV(void);
public:
        ~ULCV(void);
        void setupPressurePointers(double);
        void setupConcentrationPointers(double);
        void computeAVEyPressures(void);
        void computeAVExPressures(void);
        void computeAVEhpy(void);
        void computeAVEhpx(void);
        double setdpdxCV(void);
        double setdpdyCV(void);
        double comv_xJp(void);
        void myNvalues(void);
};

#endif
```

Listing B.30. ULCV.cpp

```
/**
 * ULCV.cpp
 * @author James Geraci
 * Implements the equations COP, MB, EK which control
 * the change of the concentration, phil, and phis with time
 * also sets up the Jacobian
 *
 */
#include "StdAfx.h"
#include "ULCV.h"
#include <stdio.h>
#include "CV.h"

ULCV::ULCV(void){
        myxNorm = -1;
        myyNorm =  1;
        mydxSide = 1;
        mydySide = -1;
        nbrdxSide = -1;
        nbrdySide = 1;
}

ULCV::~ULCV(void){
}

void ULCV::myNvalues(void){
        /* xNorm direction values */
        myxNhpc     = phpc;
        myxNphisc   = pphisc;
        myxNphilc   = pphilc;
        myxNpc      = ppc;
        myxNv_yc    = pv_yc;
        myxNv_xc    = pv_xc;
        /* yNorm direction values */
        myyNhpc     = thpc;
        myyNphisc   = tphisc;
        myyNphilc   = tphilc;
        myyNpc      = tpc;
        myyNv_yc    = tv_yc;
        myyNv_xc    = tv_xc;
        /* xyNorm direction values */
        myxyNhpc    = tphpc;
        myxyNphisc  = tpphisc;
        myxyNphilc  = tpphilc;
        myxyNpc     = tppc;
        myxyNv_yc   = tpv_yc;
```

```
        myxyNv_xc   = tpv_xc;

        /* Jacobian Points */
        /* xNorm direction values */
        jxNhpc      = jphpc;
        jxNphisc    = jpphisc;
        jxNphilc    = jpphilc;
        jxNpc       = jppc;
        jxNv_yc     = jpv_yc;
        jxNv_xc     = jpv_xc;
        /* yNorm direction values */
        jyNhpc      = jthpc;
        jyNphisc    = jtphisc;
        jyNphilc    = jtphilc;
        jyNpc       = jtpc;
        jyNv_yc     = jtv_yc;
        jyNv_xc     = jtv_xc;
        /* xyNorm direction values */
        jxyNhpc     = jtphpc;
        jxyNphisc   = jtpphisc;
        jxyNphilc   = jtpphilc;
        jxyNpc      = jtppc;
        jxyNv_yc    = jtpv_yc;
        jxyNv_xc    = jtpv_xc;
  return;
};

void ULCV::setupPressurePointers(double vol){
  //  printf("You are using this version of the function\n");
  Pptr[TOP]    = &(avePressures_y[topV]);
  Pptr[RIGHT]  = &(Vptr[pc]);
  Pptr[LEFT]   = &(avePressures_x[myColumn-1+myRow*numFVColumns]);
  Pptr[BOTTOM] = &(Vptr[pc]);
  return;
}

void ULCV::setupConcentrationPointers(double vol){
  Cptr[TOP]    = &(AVEhp_y[topV]);
  Cptr[RIGHT]  = &(Vptr[hpc]);
  Cptr[LEFT]   = &(AVEhp_x[myColumn-1+myRow*numFVColumns]);
  Cptr[BOTTOM] = &(Vptr[hpc]);
  outerXdensityAVE = Cptr[LEFT];
  outerYdensityAVE = Cptr[TOP];
  return;
}

void ULCV::computeAVEyPressures(void){
```

```
  *(Pptr[TOP]) = (Vptr[tpc] + Vptr[pc])/2;
  return;
}

void ULCV::computeAVExPressures(void){
  *(Pptr[LEFT]) = (Vptr[pc]+Vptr[ppc])/2;
  return;
}

void ULCV::computeAVEhpy(void){
  *(Cptr[TOP]) = (Vptr[thpc] + Vptr[hpc])/2;
  return;
}

void ULCV::computeAVEhpx(void){
*(Cptr[LEFT]) = (Vptr[hpc] + Vptr[phpc])/2;
  return;
}

double ULCV::setdpdxCV(void){
  return mydy*(Vptr[pc] - (Vptr[pc]+Vptr[ppc])/2);
}

double ULCV::setdpdyCV(void){
  return mydx*((Vptr[tpc]+Vptr[pc])/2-Vptr[pc]);
}

double ULCV::comv_xJp(void){
  return -(mydy*Vptr[pc]/2);
}
```

Listing B.31. URCV.h

```
/*
 *
 * @author James Geraci
 *
 */
#ifndef _URCV_H
#define _URCV_H
#include "StdAfx.h"
#include "CV.h"
//class FV;

class URCV: public CV
{
public:
        URCV(void);
public:
        ~URCV(void);
        void setupPressurePointers(double);
        void setupConcentrationPointers(double);
        void computeAVEyPressures(void);
        void computeAVExPressures(void);
        void computeAVEhpy(void);
        void computeAVEhpx(void);
        double setdpdxCV(void);
        double setdpdyCV(void);
        double comv_xJp(void);
        void myNvalues(void);
};

#endif
```

Listing B.32. URCV.cpp

```
/**
 * URCV.cpp
 * @author James Geraci
 * Implements the equations COP, MB, EK which control
 * the change of the concentration, phil, and phis with time
 * also sets up the Jacobian
 *
 */
#include "StdAfx.h"
#include "URCV.h"
#include <stdio.h>

URCV::URCV(void)
{
        myxNorm = 1;
        myyNorm = 1;
        mydxSide = -1;
        mydySide = -1;
        nbrdxSide = 1;
        nbrdySide = 1;
}

URCV::~URCV(void)
{
}

void URCV::myNvalues(void){
        /* xNorm direction values */
        myxNhpc     = nhpc;
        myxNphisc   = nphisc;
        myxNphilc   = nphilc;
        myxNpc      = npc;
        myxNv_yc    = nv_yc;
        myxNv_xc    = nv_xc;
        /* yNorm direction values */
        myyNhpc     = thpc;
        myyNphisc   = tphisc;
        myyNphilc   = tphilc;
        myyNpc      = tpc;
        myyNv_yc    = tv_yc;
        myyNv_xc    = tv_xc;
        /* xyNorm direction values */
        myxyNhpc    = tnhpc;
        myxyNphisc  = tnphisc;
        myxyNphilc  = tnphilc;
        myxyNpc     = tnpc;
```

```
        myxyNv_yc   = tnv_yc;
        myxyNv_xc   = tnv_xc;

/* xNorm direction values */
        jxNhpc      = jnhpc;
        jxNphisc    = jnphisc;
        jxNphilc    = jnphilc;
        jxNpc       = jnpc;
        jxNv_yc     = jnv_yc;
        jxNv_xc     = jnv_xc;
        /* yNorm direction values */
        jyNhpc      = jthpc;
        jyNphisc    = jtphisc;
        jyNphilc    = jtphilc;
        jyNpc       = jtpc;
        jyNv_yc     = jtv_yc;
        jyNv_xc     = jtv_xc;
        /* xyNorm direction values */
        jxyNhpc     = jtnhpc;
        jxyNphisc   = jtnphisc;
        jxyNphilc   = jtnphilc;
        jxyNpc      = jtnpc;
        jxyNv_yc    = jtnv_yc;
        jxyNv_xc    = jtnv_xc;
  return;
};

void URCV::setupPressurePointers(double vol){
  Pptr[TOP]    = &(avePressures_y[topV]);
  Pptr[RIGHT]  = &(avePressures_x[myColumn+myRow*numFVColumns]);
  Pptr[LEFT]   = &(Vptr[pc]);
  Pptr[BOTTOM] = &(Vptr[pc]);
  return;
}

void URCV::setupConcentrationPointers(double vol){
  Cptr[TOP]    = &(AVEhp_y[topV]);
  Cptr[RIGHT]  = &(AVEhp_x[myColumn+myRow*numFVColumns]);
  Cptr[LEFT]   = &(Vptr[hpc]);
  Cptr[BOTTOM] = &(Vptr[hpc]);
  outerXdensityAVE = Cptr[RIGHT];
  outerYdensityAVE = Cptr[TOP];
  return;
}

void URCV::computeAVEyPressures(void){
    *(Pptr[TOP]) = (Vptr[tpc] + Vptr[pc])/2;
```

```
  return;
}

void URCV::computeAVExPressures(void){
  *(Pptr[RIGHT]) = (Vptr[npc]+Vptr[pc])/2;
  return;
}

void URCV::computeAVEhpy(void){
  *(Cptr[TOP]) = (Vptr[thpc]+Vptr[hpc])/2;

  return;
}

void URCV::computeAVEhpx(void){
  *(Cptr[RIGHT]) = (Vptr[nhpc]+Vptr[hpc])/2;
  //  printf("The value AVEhpx in UR is %f with outerXdensityAVE %f\n", *(
      Cptr[RIGHT]), *outerXdensityAVE);
  return;
}

double URCV::setdpdxCV(void){
  return mydy*((Vptr[npc]+Vptr[pc])/2-Vptr[pc]);
}

double URCV::setdpdyCV(void){
  return mydx*((Vptr[tpc]+Vptr[pc])/2-Vptr[pc]);
}

double URCV::comv_xJp(void){
 return (mydy*Vptr[pc]/2);
}
```

Listing B.33. LLCV.h

```
/*
 *
 * @author James Geraci
 *
 */
#ifndef _LLCV_H
#define _LLCV_H
#include "StdAfx.h"
#include "CV.h"
//class FV;

class LLCV:public CV
{
public:
        LLCV(void);
public:
        ~LLCV(void);
        void setupPressurePointers(double);
        void setupConcentrationPointers(double);
        void computeAVEyPressures(void);
        void computeAVExPressures(void);
        void computeAVEhpy(void);
        void computeAVEhpx(void);
        double setdpdxCV(void);
        double setdpdyCV(void);
        double comv_xJp(void);
        void myNvalues(void);
};

#endif
```

Listing B.34. LLCV.cpp

```
/**
 * LLCV.cpp
 * @author James Geraci
 * Implements the equations COP, MB, EK which control
 * the change of the concentration, phil, and phis with time
 * also sets up the Jacobian
 *
 */
#include "StdAfx.h"
#include "LLCV.h"
#include <stdio.h>

LLCV::LLCV(void)
{
        myxNorm = -1;
        myyNorm = -1;
        mydxSide = 1;
        mydySide = 1;
        nbrdxSide = -1;
        nbrdySide = -1;
}

LLCV::~LLCV(void)
{
}

void LLCV::myNvalues(void){
        /* xNorm direction values */
        myxNhpc     = phpc;
        myxNphisc   = pphisc;
        myxNphilc   = pphilc;
        myxNpc      = ppc;
        myxNv_yc    = pv_yc;
        myxNv_xc    = pv_xc;
        /* yNorm direction values */
        myyNhpc     = bhpc;
        myyNphisc   = bphisc;
        myyNphilc   = bphilc;
        myyNpc      = bpc;
        myyNv_yc    = bv_yc;
        myyNv_xc    = bv_xc;
        /* xyNorm direction values */
        myxyNhpc    = bphpc;
        myxyNphisc  = bpphisc;
        myxyNphilc  = bpphilc;
        myxyNpc     = bppc;
```

```
        myxyNv_yc   = bpv_yc;
        myxyNv_xc   = bpv_xc;

        /* Jacobian Values */
        /* xNorm direction values */
        jxNhpc      = jphpc;
        jxNphisc    = jpphisc;
        jxNphilc    = jpphilc;
        jxNpc       = jppc;
        jxNv_yc     = jpv_yc;
        jxNv_xc     = jpv_xc;
        /* yNorm direction values */
        jyNhpc      = jbhpc;
        jyNphisc    = jbphisc;
        jyNphilc    = jbphilc;
        jyNpc       = jbpc;
        jyNv_yc     = jbv_yc;
        jyNv_xc     = jbv_xc;
        /* xyNorm direction values */
        jxyNhpc     = jbphpc;
        jxyNphisc   = jbpphisc;
        jxyNphilc   = jbpphilc;
        jxyNpc      = jbppc;
        jxyNv_yc    = jbpv_yc;
        jxyNv_xc    = jbpv_xc;
    return;
};

void LLCV::setupPressurePointers(double vol){
    Pptr[TOP]    = &(Vptr[pc]);
    Pptr[RIGHT]  = &(Vptr[pc]);
    Pptr[LEFT]   = &(avePressures_x[myColumn-1+myRow*numFVColumns]);
    Pptr[BOTTOM] = &(avePressures_y[presV]);
    return;
}

void LLCV::setupConcentrationPointers(double vol){
    Cptr[TOP]    = &(Vptr[hpc]);
    Cptr[RIGHT]  = &(Vptr[hpc]);
    Cptr[LEFT]   = &(AVEhp_x[myColumn-1+myRow*numFVColumns]);
    Cptr[BOTTOM] = &(AVEhp_y[presV]);
    outerXdensityAVE = Cptr[LEFT];
    outerYdensityAVE = Cptr[BOTTOM];
    return;
}

void LLCV::computeAVEyPressures(void){
```

```
    *(Pptr[BOTTOM]) = (Vptr[pc] + Vptr[bpc])/2;
  return;
}

void LLCV::computeAVExPressures(void){
  *(Pptr[LEFT]) = (Vptr[pc]+Vptr[ppc])/2;
  return;
}

void LLCV::computeAVEhpy(void){
  *(Cptr[BOTTOM]) = (Vptr[hpc] + Vptr[bhpc])/2;
  return;
}

void LLCV::computeAVEhpx(void){
  *(Cptr[LEFT]) = (Vptr[hpc]+Vptr[phpc])/2;
  return;
}

double LLCV::setdpdxCV(void){
  return mydy*(Vptr[pc] - (Vptr[pc]+Vptr[ppc])/2);
}

double LLCV::setdpdyCV(void){
  return mydx*(Vptr[pc] - (Vptr[pc]+Vptr[bpc])/2);
}

double LLCV::comv_xJp(void){
  return -(mydy*Vptr[pc]/2);
}
```

Listing B.35. LRCV.h

```
/*
 *
 * @author James Geraci
 *
 */
#ifndef _LRCV_H
#define _LRCV_H
#include "StdAfx.h"
#include "CV.h"
//class FV;

class LRCV:public CV
{
public:
        LRCV(void);
public:
        ~LRCV(void);
        void setupPressurePointers(double);
        void setupConcentrationPointers(double);
        void computeAVEyPressures(void);
        void computeAVExPressures(void);
        void computeAVEhpy(void);
        void computeAVEhpx(void);
        double setdpdxCV(void);
        double setdpdyCV(void);
        double comv_xJp(void);
        void myNvalues(void);
};

#endif
```

Listing B.36. LRCV.cpp

```
/**
 * LRCV.cpp
 * @author James Geraci
 * Implements the equations COP, MB, EK which control
 * the change of the concentration, phil, and phis with time
 * also sets up the Jacobian
 *
 */
#include "StdAfx.h"
#include "LRCV.h"
#include <stdio.h>

LRCV::LRCV(void)
{
        myxNorm = 1;
        myyNorm = -1;
        mydxSide = -1;
        mydySide = 1;
        nbrdxSide = 1;
        nbrdySide = -1;
        //      printf("You have made a LRCV\n");
}

LRCV::~LRCV(void)
{
}

void LRCV::myNvalues(void){
        /* xNorm direction values */
        myxNhpc     = nhpc;
        myxNphisc   = nphisc;
        myxNphilc   = nphilc;
        myxNpc      = npc;
        myxNv_yc    = nv_yc;
        myxNv_xc    = nv_xc;
        /* yNorm direction values */
        myyNhpc     = bhpc;
        myyNphisc   = bphisc;
        myyNphilc   = bphilc;
        myyNpc      = bpc;
        myyNv_yc    = bv_yc;
        myyNv_xc    = bv_xc;
        /* xyNorm direction values */
        myxyNhpc    = bnhpc;
        myxyNphisc  = bnphisc;
        myxyNphilc  = bnphilc;
```

```
        myxyNpc     = bnpc;
        myxyNv_yc   = bnv_yc;
        myxyNv_xc   = bnv_xc;


        /* Jacobian Values */
        /* xNorm direction values */
        jxNhpc      = jnhpc;
        jxNphisc    = jnphisc;
        jxNphilc    = jnphilc;
        jxNpc       = jnpc;
        jxNv_yc     = jnv_yc;
        jxNv_xc     = jnv_xc;
        /* yNorm direction values */
        jyNhpc      = jbhpc;
        jyNphisc    = jbphisc;
        jyNphilc    = jbphilc;
        jyNpc       = jbpc;
        jyNv_yc     = jbv_yc;
        jyNv_xc     = jbv_xc;
        /* xyNorm direction values */
        jxyNhpc     = jbnhpc;
        jxyNphisc   = jbnphisc;
        jxyNphilc   = jbnphilc;
        jxyNpc      = jbnpc;
        jxyNv_yc    = jbnv_yc;
        jxyNv_xc    = jbnv_xc;
  return;
};

void LRCV::setupPressurePointers(double vol){
  Pptr[TOP]    = &(Vptr[pc]);
  Pptr[RIGHT]  = &(avePressures_x[myColumn+myRow*numFVColumns]);
  Pptr[LEFT]   = &(Vptr[pc]);
  Pptr[BOTTOM] = &(avePressures_y[presV]);
  return;
}

void LRCV::setupConcentrationPointers(double){
  Cptr[TOP]    = &(Vptr[hpc]);
  Cptr[RIGHT]  = &(AVEhp_x[myColumn+myRow*numFVColumns]);
  Cptr[LEFT]   = &(Vptr[hpc]);
  Cptr[BOTTOM] = &(AVEhp_y[presV]);
  outerXdensityAVE = Cptr[RIGHT];
  outerYdensityAVE = Cptr[BOTTOM];
  return;
}
```

```
void LRCV::computeAVEyPressures(void){
  *(Pptr[BOTTOM]) = (Vptr[pc] + Vptr[bpc])/2;
  return;
}

void LRCV::computeAVExPressures(void){
  *(Pptr[RIGHT]) = (Vptr[npc]+Vptr[pc])/2;
  return;
}

void LRCV::computeAVEhpy(void){
  *(Cptr[BOTTOM]) = (Vptr[hpc] + Vptr[bhpc])/2;
  return;
}

void LRCV::computeAVEhpx(void){
  *(Cptr[RIGHT]) = (Vptr[nhpc]+Vptr[hpc])/2;
  return;
}

double LRCV::setdpdxCV(void){
  return mydy*((Vptr[npc]+Vptr[pc])/2-Vptr[pc]);
}

double LRCV::setdpdyCV(void){
  return mydx*(Vptr[pc] - (Vptr[pc]+Vptr[bpc])/2);
}

double LRCV::comv_xJp(void){
 return (mydy*Vptr[pc]/2);
}
```

Listing B.37. electrodeSizeData.h

```
/* @author James Geraci
 */
#ifndef _ELECTRODESIZEDATA_H
#define _ELECTRODESIZEDATA_H
class electrodeSizeData
{
public:
        electrodeSizeData(void);
        electrodeSizeData(double, double, double, int, int, int, int, int,
           int, int, int, int, cellSizeData*);
public:
        ~electrodeSizeData(void);

public:
        cellSizeData* mycsData;
        void setFVColumns(int);
        void setFVRows(int);
        void setFVZrows(int);
        void setFVStartColumn(int);

        void setxWidth(double);
        void setyHeight(double);
        void setZdepth(double);

        double getyHeight(void);
        double getxWidth(void);
        double getzDepth(void);

        int getnumFVColumns(void);
        int getnumFVRows(void);
        int getnumFVZrows(void);

        int getnumPVColumns(void);
        int getnumPVRows(void);
        int getnumPVZrows(void);

        int getFVstartColumn(void);
        int getFVstartRow(void);

        int getPVstartColumn(void);
        int getPVstartRow(void);

private:


        int numFVColumns;
```

```
        int numFVRows;
        int numFVZrows;


        int numPVColumns;
        int numPVRows;
        int numPVZrows;

        int FVstartColumn;
        int FVstartRow;

        int PVstartColumn;
        int PVstartRow;

        double xWidth;
        double yHeight;
        double zDepth;
};

#endif
```

Listing B.38. electrodeSizeData.cpp

```
/*
 * @author James Geraci
 * Used in the creating of electrodes to set the physical sizes and
 * grid sizes and how they relate to each other
 */
#include "StdAfx.h"
#include "electrodeSizeData.h"
#include "cellSizeData.h"

electrodeSizeData::electrodeSizeData(void)
{
        numFVZrows = 1;
}

electrodeSizeData::~electrodeSizeData(void)
{
}

electrodeSizeData::electrodeSizeData(double w, double h, double d, int FVc
    , int FVr, int FVz,
                                                                    int
                                                                      PVc
                                                                      ,
                                                                      int
                                                                      PVr
                                                                      ,
                                                                      int
                                                                      PVz
                                                                      ,
                                                                      int
                                                                      FVsr
                                                                      ,
                                                                      int
                                                                      FVsc
                                                                      ,
                                                                      int
                                                                      PVsr
                                                                      ,
                                                                      int
```

```
                                                                PVsc
                                                                ,
                                                                cellSizeDa
                                                                *
                                                                csData
                                                                )
                                                                {

        xWidth = w;
        yHeight = h;
        zDepth = d;
        numFVColumns = FVc;
        numFVRows = FVr;
        numFVZrows = FVz;
        numPVColumns = PVc;
        numPVRows = PVr;
        numPVZrows = PVz;
        FVstartRow = FVsr;
        FVstartColumn = FVsc;

        PVstartRow = PVsr;
        PVstartColumn = PVsc;
        mycsData = csData;
}
void electrodeSizeData::setFVColumns(int nColumns)
{
        numFVColumns = nColumns;
        return;
};

void electrodeSizeData::setFVRows(int nRows)
{
        numFVRows = nRows;
        return;
};

void electrodeSizeData::setFVZrows(int nZrows)
{
        numFVZrows = nZrows;
        return;
};

void electrodeSizeData::setFVStartColumn(int sColumn)
{
        FVstartColumn = sColumn;
        return;
};
```

```
void electrodeSizeData::setyHeight(double H)
{
        yHeight = H;
        return;
};

void electrodeSizeData::setxWidth(double W)
{
        xWidth = W;
        return;
};

void electrodeSizeData::setZdepth(double D)
{
        zDepth = D;
        return;
};


int electrodeSizeData::getnumFVColumns(void)
{
        return numFVColumns;
};

int electrodeSizeData::getnumFVRows(void)
{
        return numFVRows;
};

int electrodeSizeData::getnumFVZrows(void)
{
        return numFVZrows;
};


int electrodeSizeData::getnumPVColumns(void)
{
        return numPVColumns;
};

int electrodeSizeData::getnumPVRows(void)
{
        return numPVRows;
};

int electrodeSizeData::getnumPVZrows(void)
```

```
{
        return numPVZrows;
};

double electrodeSizeData::getyHeight(void)
{
        return yHeight;
};

double electrodeSizeData::getxWidth(void)
{
        return xWidth;
};

double electrodeSizeData::getzDepth(void)
{
        return zDepth;
};

int electrodeSizeData::getFVstartColumn(void)
{
        return FVstartColumn;
};

int electrodeSizeData::getFVstartRow(void)
{
        return FVstartRow;
};


int electrodeSizeData::getPVstartColumn(void)
{
        return PVstartColumn;
};

int electrodeSizeData::getPVstartRow(void)
{
        return PVstartRow;
};
```

Listing B.39. volumeChemData.h

```
/* @author James Geraci
 */
#ifndef _VOLUMECHEMDATA_H
#define _VOLUMECHEMDATA_H

class volumeChemData
{
public:
        volumeChemData(void);
        volumeChemData(double, double, double, double, double, double, double,
            double, double, double,
                        double, double, double, double, double, double, double,
                            double, double, double, double,
                        double, double, double, double);
public:
        ~volumeChemData(void);

public:
        double getAmax(void);
        double getT(void);
        double getals(void);
        double getasl(void);
        double getDx(void);
        double getDy(void);
        double getsigx(void);
        double getsigy(void);
        double getKx(void);
        double getKy(void);
        double getMux(void);
        double getMuy(void);
        double getBetax(void);
        double getBetay(void);
        double getk(void);
        double getm(void);
        double getexm(void);
        double getex(void);
        double getphi_eq(void);
        double gethnot(void);
        double getepsnot(void);
        double getmySign(void);
        double getQmax(void);
        double getSoc(void);
        double getEta(void);

private:
        double T;       // Temp in Kelvin
```

```
        double Amax;   // Maximum interfacial area
        double Qmax;   // Maximum Charge/cm^3
        double soc;    // state of charge
        double eta;    // scale factor for SOC
        double als;    // transfer coefficient anodic
        double asl;    // transfer coefficient cathodic
        double Dx;     // diffusion coefficient in x-direction
        double Dy;     // diffusion coefficient in y-direction
        double sigx;    // conductivity of metal in Ohm/cm^2
        double sigy;    // conductivity of metal in Ohm/cm^2
        double Kx;      // conductivity o felectrolyte ohm/cm^2
        double Ky;      // conductivity o felectrolyte ohm/cm^2
        double mu_x;    // Kinematic viscosity x cm^2/s
        double mu_y;    // Kinematic viscosity y cm^2/s
        double beta_x; // Volume expansion coefficient x cm^3/mol
        double beta_y; // Volume expansion coefficient y cm^3/mol
        double k;
        double m;
        double exm;
        double ex;
        double phi_eq;
        double hnot;
        double epsnot;
        double mySign;
};

#endif
```

Listing B.40. volumeChemData.cpp

```
/*
 * @author James Geraci
 * sets initial physical parameters for all volumes
 */
#include "StdAfx.h"
#include "volumeChemData.h"

volumeChemData::volumeChemData(void)
{
}


volumeChemData::
volumeChemData(double T1,double Amax1, double Q, double sc, double eda,
   double als1,double asl1,
               double Dx1,double Dy1,double sig1x,double sig1y,double K1x,
                  double K1y,double k1,
               double m1,double exm1,double ex1,double phi_eq1,double hnot1
                  , double epsnot1,
               double cSign, double mu_x1, double mu_y1, double beta_x1,
                  double beta_y1){
        T = T1;
        Amax = Amax1;
        als = als1;
        asl = asl1;
        Dx = Dx1;
        Dy = Dy1;
        sigx = sig1x;
        Kx = K1x;
        sigy = sig1y;
        Ky = K1y;
        mu_x = mu_x1;
        mu_y = mu_y1;
        beta_x = beta_x1;
        beta_y = beta_y1;
        k = k1;
        m = m1;
        exm = exm1;
        ex = ex1;
        phi_eq = phi_eq1;
        hnot = hnot1;
        epsnot = epsnot1;
        mySign = cSign;
        Qmax = Q;
        soc = sc;
        eta = eda;
```

```
}


volumeChemData::~volumeChemData(void)
{
}

double volumeChemData::getAmax(void)
{
        return Amax;
};

double volumeChemData::getT(void)
{
        return T;
};

double volumeChemData::getals(void)
{
        return als;
};

double volumeChemData::getasl(void)
{
        return asl;
};

double volumeChemData::getDx(void)
{
        return Dx;
};

double volumeChemData::getDy(void)
{
        return Dy;
};

double volumeChemData::getsigx(void)
{
        return sigx;
};

double volumeChemData::getsigy(void)
{
        return sigy;
};
```

```
double volumeChemData::getKx(void)
{
        return Kx;
};

double volumeChemData::getKy(void)
{
        return Ky;
};

double volumeChemData::getMux(void)
{
  return mu_x;
};

double volumeChemData::getMuy(void)
{
  return mu_y;
};

double volumeChemData::getBetax(void)
{
  return beta_x;
};

double volumeChemData::getBetay(void)
{
  return beta_y;
};

double volumeChemData::getk(void)
{
  return k;
};

double volumeChemData::getm(void)
{
  return m;
};

double volumeChemData::getexm(void)
{
        return exm;
};

double volumeChemData::getex(void)
{
```

```
        return ex;
};

double volumeChemData::getphi_eq(void)
{
        return phi_eq;
};

double volumeChemData::gethnot(void)
{
        return hnot;
};

double volumeChemData::getepsnot(void)
{
        return epsnot;
};

double volumeChemData::getmySign(void)
{
        return mySign;
};

double volumeChemData::getQmax(void)
{
        return Qmax;
};

double volumeChemData::getSoc(void)
{
        return soc;
};


double volumeChemData::getEta(void)
{
        return eta;
};
```

Listing B.41. PVIConditions.h

```
/* @author James Geraci
 */
#ifndef _PVIConditions_H
#define _PVIConditions_H

class PVIConditions
{
public:
        PVIConditions(void);
        PVIConditions(double,double,double,double,double,double);
public:
        ~PVIConditions(void);

public:
        double gethp(void);
        double getphi_l(void);
        double getphi_s(void);
        double getp(void);
        double getv_y(void);
        double getv_x(void);
private:
        double hp;
        double phi_l;
        double phi_s;
        double p;
        double v_y;
        double v_x;
};

#endif
```

Listing B.42. PVIConditions.cpp

```
/*
 * @author James Geraci
 * sets initial values for concentration, phil, and phis
 */
#include "StdAfx.h"
#include "PVIConditions.h"
#include <iostream>
using namespace std;
PVIConditions::PVIConditions(void)
{
}

PVIConditions::PVIConditions(double h,double l,double s,double pl, double
   v_yl, double v_xl)
{
        hp = h;
        phi_l = l;
        phi_s = s;
        p = pl;
        v_y = v_yl;
        v_x = v_xl;
}

PVIConditions::~PVIConditions(void)
{
}

double PVIConditions::gethp(void)
{
        return hp;
};

double PVIConditions::getphi_l(void)
{
        return phi_l;
};

double PVIConditions::getphi_s(void)
{
        return phi_s;
};

double PVIConditions::getp(void){
  return p;
};
```

```
double PVIConditions::getv_y(void){
  return v_y;
};

double PVIConditions::getv_x(void){
  return v_x;
};
```

Listing B.43. Electrolyte.h

```
/* @author James Geraci
 */
#ifndef _ELECTROLYTE_H
#define _ELECTROLYTE_H

#include "Electrode.h"

class Electrolyte :
        public Electrode
{
public:
        Electrolyte(void);
        Electrolyte(FV**,PV**,matrixData*,electrodeSizeData*);
public:
        ~Electrolyte(void);

private:
        static const electrolyteElectrodeChemData myChem;
        static const volumeChemData vCD;
        static const volumeIConditions vIC;
        static const PVIConditions pviC;
};

#endif
```

Listing B.44. Electrolyte.cpp

```
/*
 * @author James Geraci
 * Contains the parameters necessary to set up an eletrolyte 'electrode'
 */
#include "StdAfx.h"
#include "Electrolyte.h"


const electrolyteElectrodeChemData Electrolyte::myChem;
const volumeChemData Electrolyte::vCD(298/*T*/,0/*Amax*/,1/*Qmax*/,0/*soc*/
    ,0/*eta*/,0/*als*/,0/*asl*/,0.3e-5/*Dx*/,0.3e-5/*Dy*/,
                                          0/*sigx*/,0/*sigy*/,0.79/*Kappax*/
                                              ,0.79/*Kappay*/,1/*k*/,1/*m*/,0.5
                                              /*exm*/,1.5 /*ex*/,0/*phieq*/,
                                          4.9e-3/*hnot*/,1/*epsnot*/,0/*mySign
                                              */,10e-2/*mu_x*/,10e-2/*mu_y*/,35
                                              /*beta_x*/,35/*beta_y*/);
const volumeIConditions Electrolyte::vIC(1,0,0);
const PVIConditions Electrolyte::pviC(4.9e-3/*hp*/,0/*phi_l*/,0/*phi_s*/,
    atmospheric_pressure/*p*/,0/*v_y*/,0/*v_x*/);

Electrolyte::Electrolyte(void)
{
}

Electrolyte::Electrolyte(FV** myFV, PV** myPV, matrixData* Matrix,
    electrodeSizeData* eDat):Electrode("Electrolyte",myFV,myPV,Matrix,eDat,
    myChem,vCD,vIC,pviC)
{
}

Electrolyte::~Electrolyte(void)
{
}
```

Listing B.45. electrodeChemData.h

```
/* @author James Geraci
 */
#ifndef _ELECTRODECHEMDATA_H
#define _ELECTRODECHEMDATA_H

class electrodeChemData
{
public:
        electrodeChemData(void);
        electrodeChemData(double,double,double,double);
public:
        virtual ~electrodeChemData(void);

        public:
        double getmyVf(void);
        double getmbScale(void);
        double getolsScale(void);
        double getio(void);
        double gettnp(void);

protected:

        static const double PbSO4_MW;
        static const double PbSO4_rho;

        virtual void settnp(double);
        virtual void setmbScale(double);
        virtual void setolsScale(void);
        virtual void setmyio(double);
        virtual void setmyVf(double);
        void ecDataInit(double, double, double, double);
        double myTnp;
        double myVf;
        double mbScale;
        double olsScale;
        double myio;
};
#endif
```

Listing B.46. electrodeChemData.cpp

```
/* @author James Geraci
 */
/**
 * electrodeChemData.cpp
 * Generic to any electrode
 * @author James Geraci
 * Set/Fectch some physical properties for each electrode
 */
#include "StdAfx.h"
#include "electrodeChemData.h"

const double electrodeChemData::PbSO4_MW = 303.2626;  // g
const double electrodeChemData::PbSO4_rho = 6.39; // g/cm^3


electrodeChemData::electrodeChemData(void)
{
}

electrodeChemData::electrodeChemData(double Tnp, double io, double mbscale
    , double vf)
{
        ecDataInit(Tnp,io,mbscale,vf);
}


electrodeChemData::~electrodeChemData(void)
{
}

void electrodeChemData::ecDataInit(double Tnp, double io, double mbS,
   double vf)
{
        settnp(Tnp);
        setmbScale(mbS);
        setolsScale();
        setmyio(io);
        setmyVf(vf);
        return;
}

void electrodeChemData::settnp(double Tnp)
{
        myTnp = Tnp;
        return;
};
```

```
void electrodeChemData::setmbScale(double mb)
{
        mbScale = mb;
        return;
};

void electrodeChemData::setolsScale()
{
        olsScale = -(1-2*myTnp)*(R*298)/F;
        return;
};

void electrodeChemData::setmyio(double io)
{
        myio = io;
        return;
};

void electrodeChemData::setmyVf(double vf)
{
        myVf = vf;
        return;
};

double electrodeChemData::getmbScale(void)
{
        return mbScale;
}

double electrodeChemData::getolsScale(void)
{
        return olsScale;
}

double electrodeChemData::gettnp(void)
{
        return myTnp;
}

double electrodeChemData::getmyVf(void)
{
        return myVf;
}

double electrodeChemData::getio(void)
{
```

```
        return myio;
}
```

Listing B.47. IdxElectrodeChemData.h

```
/* @author James Geraci
 */
#ifndef _IDXELECTRODECHEMDATA_H
#define _IDXELECTRODECHEMDATA_H
#include "electrodeChemData.h"

class IdxElectrodeChemData :
        public electrodeChemData
{
public:
        IdxElectrodeChemData(void);
        IdxElectrodeChemData(double,double,double,double);
public:
        ~IdxElectrodeChemData(void);

protected:
        static const double PbO2_MW;
        static const double PbO2_rho;

        void setmbScale();
        void setmyVf();
};

#endif
```

Listing B.48. ldxElectrodeChemData.cpp

```
/*
 * @author James Geraci
 * electrode Chem Data object specific for a lead dioxide electrode
 */
#include "StdAfx.h"
#include "ldxElectrodeChemData.h"


const double ldxElectrodeChemData::PbO2_MW = 239.1988;  // g
const double ldxElectrodeChemData::PbO2_rho = 9.79; // g/cm^3

ldxElectrodeChemData::ldxElectrodeChemData(void)
{
}

ldxElectrodeChemData::ldxElectrodeChemData(double Tnp,double io,double mbS,
    double vf):electrodeChemData(Tnp,io,mbS,vf)
{
}

ldxElectrodeChemData::~ldxElectrodeChemData(void)
{
}

void ldxElectrodeChemData::setmbScale()
{
        mbScale = -(3-2*myTnp)/(2*F);
        return;
};


void ldxElectrodeChemData::setmyVf()
{
        myVf = PbSO4_MW/PbSO4_rho - PbO2_MW/PbO2_rho;
        return;
};
```

Listing B.49. electrolyteElectrodeChemData.h

```
/* @author James Geraci
 */

#ifndef ELECTROLYTEELECTRODECHEMDATA_H
#define ELECTROLYTEELECTRODECHEMDATA_H
#include "electrodeChemData.h"
```

```
class electrolyteElectrodeChemData :
        public electrodeChemData
{
public:
        electrolyteElectrodeChemData(void);
public:
        ~electrolyteElectrodeChemData(void);
};
#endif
```

Listing B.50. electrolyteElectrodeChemData.cpp

```
/*
 * @author James Geraci
 * an electrolyte electrode specific electrochemData object
 */
#include "StdAfx.h"
#include "electrolyteElectrodeChemData.h"

electrolyteElectrodeChemData::electrolyteElectrodeChemData(void):
    electrodeChemData(0.72,0,0,0)
{
}

electrolyteElectrodeChemData::~electrolyteElectrodeChemData(void)
{
}
```

Listing B.51. ldElectrodeChemData.h

```
/* @author James Geraci
 */
#ifndef _LDELECTRODECHEMDATA_H
#define _LDELECTRODECHEMDATA_H
#include "electrodeChemData.h"

class ldElectrodeChemData :
        public electrodeChemData
{
public:
        ldElectrodeChemData(void);
        ldElectrodeChemData(double, double, double, double);
public:
        ~ldElectrodeChemData(void);


protected:
        static const double Pb_MW;
        static const double Pb_rho;

        void setmbScale();
        void setmyVf();
};

#endif
```

Listing B.52. ldElectrodeChemData.cpp

```
/*
 * @author James Geraci
 * electrode chem data object needed to create a lead electrode
 */
#include "StdAfx.h"
#include "ldElectrodeChemData.h"


const double ldElectrodeChemData::Pb_MW = 207.2;  // g
const double ldElectrodeChemData::Pb_rho = 11.34; // g/cm^3

ldElectrodeChemData::ldElectrodeChemData(void)
{
}

ldElectrodeChemData::ldElectrodeChemData(double Tnp,double io, double mbS,
   double vf):electrodeChemData(Tnp,io,mbS,vf)
{
}

ldElectrodeChemData::~ldElectrodeChemData(void)
{
}

void ldElectrodeChemData::setmbScale()
{
        mbScale = (2*myTnp-1)/(2*F);
        return;
};

void ldElectrodeChemData::setmyVf()
{
        myVf =  Pb_MW/Pb_rho - PbSO4_MW/PbSO4_rho;
        return;
};
```

Listing B.53. volumSpatialData.h

```
/* @author James Geraci
 */
#ifndef VOLUMESPATIALDATA_H
#define VOLUMESPATIALDATA_H

class volumeSpatialData
{
public:
        volumeSpatialData(void);
        volumeSpatialData(double, double, double, int, int, int);
public:
        ~volumeSpatialData(void);


        public:
                double getxWidth(void);
                double getyHeight(void);
                double getzDepth(void);

                int getColumn(void);
                int getRow(void);
                int getZrow(void);

                double mydx;
                double mydy;
                double mydz;

                int myColumn;
                int myRow;
                int myZrow;
};

#endif
```

Listing B.54. volumeSpatialData.cpp

```
/*
 * @author James Geraci
 * provides each volume with knowledge of its own location within the cell
 */
#include "StdAfx.h"
#include "volumeSpatialData.h"

volumeSpatialData::volumeSpatialData(void)
{
}

volumeSpatialData::
                volumeSpatialData(double dx, double dy, double dz, int
                   Column, int Row, int Zrow)
{
        mydx = dx;
        mydy = dy;
        mydz = dz;

        myColumn = Column;
        myRow = Row;
        myZrow = Zrow;
}

volumeSpatialData::~volumeSpatialData(void)
{
}

double volumeSpatialData::getxWidth(void)
{
        return mydx;
}

double volumeSpatialData::getyHeight(void)
{
        return mydy;
}

double volumeSpatialData::getzDepth(void)
{
        return mydz;
}

int volumeSpatialData::getColumn(void)
{
        return myColumn;
```

```
}

int volumeSpatialData::getRow(void)
{
        return myRow;
}

int volumeSpatialData::getZrow(void)
{
        return myZrow;
}
```

Listing B.55. myGlobals.h

```
/* @author James Geraci
 */
#ifndef _MYGLOBALS_H
#define _MYGLOBALS_H

#include "StdAfx.h"
#include <errno.h>
#include <stdio.h>
extern double dt; // seconds
extern int tSteps;
extern double duration; // Total Simulation time
extern double N;  // Number of Moles Participating in Reaction
extern double F; // Coulomb/eq
extern double R; // J/molK
extern double g_x; // gravity in the x direction cm/s^2
extern double g_y; // gravity in the y direction cm/s^2
extern int myVars;
extern int myVols;
extern double ePor; // Electrolyte Porosity
extern double I; // current
//extern int fluxesPerVar;
extern int directionsPerFlux;
extern int fluxesPerVol;
extern double slope;  // slope of concentration density relationship
extern double intercept; // intercept of concentration density
   relationship
extern double atmospheric_pressure; // g/(cm s^2)
extern int mode;
extern "C" void* ppu_pthread_function(void*);
extern int NldxColumns;
extern int NEColumns;
extern int NldColumns;
extern int NRows;
extern int version;
extern double Qpnt;
extern double Qscalex;
extern double Qscaley;
extern double myQscale;
extern double myQxNscale;
extern double myQyNscale;
extern double myQxyNscale;
extern int myQxNnbr;
extern int myQyNnbr;
extern int myQxyNnbr;
extern double shapeDataForMatlab[3];
extern double FVshapeDataForMatlab[3];
```

```
#endif
```

Listing B.56. myGlobals.cpp

```
/*
 * @author James Geraci
 * Contains global variables related to physical properties and
 * variables needed to setup the matrices
 */
#include "myGlobals.h"
#include "StdAfx.h"
#include <stdio.h>
#include <errno.h>

extern double dt = 0.1;  // seconds
extern int tSteps = 120000; // total number of time steps to be taken
extern double I = 1.0;   // current
//extern double I = 0;   // current
// Probably don't want to touch these
extern double duration = tSteps*dt; // Total Simulation time
extern double N = 2;   // Number of Moles Participating in Reaction
extern double F = 96487;  // Coulomb/eq
extern double R = 8.3143;  // J/molK
extern double g_x = 0; // cm/s^2
extern double g_y = -980.6650; // cm/s^2 in -y direction
extern int myVars = 6;
extern int myVols  =7;
extern double ePor = 1;
//extern int fluxesPerVar = 4;
extern int directionsPerFlux = 4;
extern int fluxesPerVol = 7;
extern double slope = 5.5387e-2;
extern double intercept = 1.01;
//extern double atmospheric_pressure = 1013250;  // g/(cm s^2)
extern double atmospheric_pressure = 1;  // g/(cm s^2)
extern int mode = 0;  // Base Mode or Skyline default is Base Mod
extern int NIdxColumns = 0;
extern int NEColumns = 0;
extern int NIdColumns = 0;
extern int NRows = 0;
extern int version = 0;
extern double Qpnt = 1/sqrt(3);
extern double Qscalex = (2-Qpnt)/2;
extern double Qscaley = (2-Qpnt)/2;
extern double myQscale = Qscaley*Qscalex;
extern double myQxNscale = Qscaley*(1-Qscalex);
extern double myQyNscale = (1-Qscaley)*(Qscalex);
extern double myQxyNscale = (1-Qscaley)*(1-Qscalex);
extern int myQxNnbr = 1*myVars;
extern int myQyNnbr = 2*myVars;
```

```
extern int myQxyNnbr= 3*myVars;
extern double shapeDataForMatlab[3] = {0,0,0};
extern double FVshapeDataForMatlab[3] = {0,0,0};
```

Listing B.57. FV.h

```
/* @author James Geraci
 */
#ifndef _FV_H
#define _FV_H
#include "volumeSpatialData.h"
#include "StdAfx.h"

class FV
{
public:
        FV(void);
        FV(volumeSpatialData*,matrixData*,electrodeSizeData*,volumeChemData
            *,
            electrodeChemData*, volumeIConditions*);
public:
        ~FV(void);

        double returnQhp(int);
        double returnQxNhp(int);
        double returnQyNhp(int);
        void computeQvalues(void);
        double returnQvalue(int,int,int,int);
        void computeFluxes(void);
        void computeMaterialProperties(void);
        void computeDiv(void);
        void setupPvPointers(void);

        double returnDeffx(void);
        double returnDeffy(void);

        double returnKeffx(void);
        double returnKeffy(void);

        double returnolsKscalex(void);
        double returnolsKscaley(void);

        double returnsigeffx(void);
        double returnsigeffy(void);

        double returnDVx(void);
        double returnDVy(void);

        double returnVECx(void);
        double returnVECy(void);

        double returndx(void);
```

```
double returndy(void);
double returndz(void);

double returnQdx(void);
double returnQdy(void);
double returnQdz(void);

double returnFdx(int);
double returnFdy(int);

double returnFjlx(int);
double returnFsex(int);

double returnFjly(int);
double returnFsey(int);

double returnFvisc_x(int);
double returnFvisc_y(int);

double returnFp_x(int);
double returnFp_y(int);

void computeNextEPS(void);
void computeNextSOC(void);
double returneps(void);
void setdx(double);
void setCVvolume(int,double);
void setCVvolumeRatio(int,double);

double soceta(void);

// Diffusion Flux in x and y directions
int uFdx;
int bFdx;
int lFdy;
int rFdy;
// Flux Liquid Electrical
int uFlex;
int bFlex;
int lFley;
int rFley;
// Flux Solid Electrical
int uFsex;
int bFsex;
int lFsey;
int rFsey;
// Chemical Flux
```

```
int uFlncx;
int bFlncx;
int lFlncy;
int rFlncy;
// Flux density... huh?
int uFjlx;
int bFjlx;
int lFjly;
int rFjly;
// viscosity flux
int uFvx;
int bFvx;
int lFvy;
int rFvy;
// pressure flux
int uFpx;
int bFpx;
int lFpy;
int rFpy;

double T;
double Amax;  // Interfacial Factor
double als; // liquid to Solid Factor
double asl; // Solid
double gc;
double ga;
double io;

double Dx;
double Dy;
double eps;
double soc;
double eta;
double sigx;
double Kx;
double sigy;
double Ky;
double mu_x;     // dynamic viscosity x g/(cm s)
double mu_y;     // dynamic viscosity y g/(cm s)
double beta_x;   // volume expansion coefficient x cm^3/mol
double beta_y;   // volume expansion coefficient y cm^3/mol
double invKp;    // permeability
double d;        // mean diameter of particle
double k;
double m;
double exm;
double ex;
```

```
        double hnot;
        double epsnot;
        double phi_eq;
        double epsspan;
        bool charging;
        double mySign;
        double mbScale;

        double Qmax;

protected:
        // Data Matrix Pointers
        double* fluxVptr;
        double* divVptr;
        double* divVtwoptr;
        double* epsVptr;
        double* socVptr;
        double* pVptr;
        double* Vptr;
        double** iVptr;
        double** Jacobptr;

        int numPVColumns;
        int numPVRows;
        int numPVZrows;
        int totalPVvolumes;

        int numFVColumns;
        int numFVRows;
        int numFVZrows;
        int totalFVvolumes;

        void setULPV(void);
        void setURPV(void);
        void setLLPV(void);
        void setLRPV(void);

        // Spatial Attributes
        double dx;
        double dy;
        double dz;

        double dx2;
        double invdx;
        double invdx2;
        double dxsqrd;
```

```
        double dy2;
        double invdy;
        double invdy2;
        double dysqrd;

        double dz2;
        double invdz;
        double invdz2;
        double dzsqrd;

        double Qdx;
        double Qdy;
        double Qdz;

        double myVolume;
        double myCVvolume[4];
        double myCVvolumeRatio[4];
        double qVptr[24];
        double myDiv;

        // physical attributes
        double tnp;
//      double mbScale;
        double olsScale;
        double myVf;
        double myVfd2;


        double Deffx;
        double Deffy;
        double Keffx;
        double Keffy;
        double sigeffx;
        double sigeffy;

        double olsKscalex;
        double olsKscaley;

        // functions
        void initializeMatrix(matrixData*);
        void initializePosition(volumeSpatialData*, matrixData*,
           electrodeSizeData*);
        void initializeOtherPositions(void);
        void initializeSize(volumeSpatialData*);
        void initializeElectrodeChem(electrodeChemData*);
        void initializeVolumeChem(volumeChemData*);
```

```
        void setInitialConditions(volumeIConditions*);

        void setMaterialPropertyEffectiveValue(double*,double,double);
        void setMaterialPropertyPermeability(void);
        void computeFlux(double*,double,double,double,double);
        void computeCurrent(double*,double,double);



// Position information for Cell
        int myColumn;
        int myRow;
        int myZrow;
        int JRow;
        int JRowBuffer;

// Positioning Data
        int rowLength;

//  Used in Calculating Addresses
        int pprevV;
        int prevV;
        int presV;
        int nextV;
        int nnextV;

        int topV;
        int lowerV;

        int offSet;
        int numVars;

// present volume's properties
        int epsc;

// Upper Left PV volume's properties
        int ulPVdiv;
        int ulPVhpc;
        int ulPVphilc;
        int ulPVphisc;
        int ulPVpc;
// Upper Right PV volume's properties
        int urPVdiv;
        int urPVhpc;
        int urPVphilc;
        int urPVphisc;
        int urPVpc;
```

```
// Lower Left PV volume's properties
        int llPVdiv;
        int llPVhpc;
        int llPVphilc;
        int llPVphisc;
        int llPVpc;
// Lower Right PV volume's properties
        int lrPVdiv;
        int lrPVhpc;
        int lrPVphilc;
        int lrPVphisc;
        int lrPVpc;

// Upper Left Qvalues volume's properties
        static const int ulQhpc    = 0;
        static const int ulQphilc  = 1;
        static const int ulQphisc  = 2;
        static const int ulQpc     = 3;
// Upper Right Q volume's properties
        static const int urQhpc    = 6;
        static const int urQphilc  = 7;
        static const int urQphisc  = 8;
        static const int urQpc     = 9;
// Lower Left Q volume's properties
        static const int llQhpc    = 12;
        static const int llQphilc  = 13;
        static const int llQphisc  = 14;
        static const int llQpc     = 15;
// Lower Rigth Q volume's properties
        static const int lrQhpc    = 18;
        static const int lrQphilc  = 19;
        static const int lrQphisc  = 20;
        static const int lrQpc     = 21;
// Defines
        static const int myUL = 3;
        static const int myUR = 2;
        static const int myLL = 1;
        static const int myLR = 0;
        static const int philg = 1;  // 1 = phil before phis
        static const int phisg = 2;  // 1 = phis before phil
};

#endif
```

Listing B.58. FV.cpp

```
/*
 * @author James Geraci
 * object that contains the description of the FV volumes
 * these objects control all the fluxes within the model
 * and the evolution of the porosity
 */
#include "FV.h"
#include "StdAfx.h"

FV::FV(void)
{
}

FV::FV(volumeSpatialData* sData, matrixData *mD, electrodeSizeData* esData,
       volumeChemData* vcData, electrodeChemData* ecData, volumeIConditions
          * iData)
{
        initializeMatrix(mD);
        initializePosition(sData,mD,esData);
        initializeOtherPositions();
        initializeSize(sData);
        initializeElectrodeChem(ecData);
        initializeVolumeChem(vcData);
        setInitialConditions(iData);
        return;
}

FV::~FV(void)
{
}

void FV::initializeMatrix(matrixData *mData)
{
        fluxVptr = mData->myfluxV();
        epsVptr = mData->myepsV();
        socVptr = mData->mysocV();
        pVptr = mData->mypsV();
        Vptr = mData->mysV();
        iVptr = mData->myiV();
        divVptr = mData->mydivV();
        divVtwoptr = mData->mydivVtwo();
        Jacobptr = mData->myJacobian();
        numPVColumns = mData->myPVColumns();
        numPVRows = mData->myPVRows();
        numPVZrows = mData->myPVZrows();
        totalPVvolumes = numPVColumns*numPVRows*numPVZrows;
```

```
        numFVColumns = mData->myFVColumns();
        numFVRows = mData->myFVRows();
        numFVZrows = mData->myFVZrows();
        totalFVvolumes = numFVColumns*numFVRows*numFVZrows;

        return;
}

void FV::initializePosition(volumeSpatialData *sData, matrixData* mData,
   electrodeSizeData* eData)
{
                // Cell Position Data Initialization
        myColumn = sData->getColumn()+eData->getFVstartColumn();
        myRow = sData->getRow();
        myZrow = sData->getZrow();
//      rowLength = mData->mysvLength();

        presV = myColumn + myRow*mData->myFVColumns();
//      pprevV = presV-2;
        prevV = presV-1;
        nextV = presV+1;
        //nnextV = presV+2;

        topV = presV - mData->myFVColumns();
        lowerV = presV + mData->myFVColumns();
        return;
}

void FV::initializeOtherPositions(void)
{
        return;
};

void FV::initializeSize(volumeSpatialData *sData)
{
        // Cell Size Data Initialization
        dx = sData->getxWidth();
        dy = sData->getyHeight();
        dz = sData->getzDepth();

        myVolume = dx*dy*dz;

        dx2 = 2*dx;
        invdx = 1/dx;
        invdx2 = 1/dx2;
        dxsqrd = dx*dx;
```

```
        Qdx = dx*(1 - Qpnt);

        dy2 = 2*dy;
        invdy = 1/dy;
        invdy2 = 1/dy2;
        dysqrd = dy*dy;
        Qdy = dy*(1 - Qpnt);

        dz2 = 2*dz;
        invdz = 1/dz;
        invdz2 = 1/dz2;
        dzsqrd = dz*dz;
        Qdz = dz*(1 - Qpnt);
        return;
}

void FV::initializeElectrodeChem(electrodeChemData *cData)
{
        tnp = cData->gettnp();
        mbScale = cData->getmbScale();
        olsScale = cData->getolsScale();
        myVf = cData->getmyVf();
        myVfd2 = myVf/(2*F);
        io = cData->getio();
        return;
};

void FV::initializeVolumeChem(volumeChemData *vData)
{
        T = vData->getT();
        Amax = vData->getAmax();
        Qmax = vData->getQmax();
        soc = vData->getSoc();
        eta = vData->getEta();
        als = vData->getals();
        asl = vData->getasl();
        gc = (als*R*T*N)/(F);
        ga = (asl*R*T*N)/(F);
        Dx = vData->getDx();
        Dy = vData->getDy();
        sigx = vData->getsigx();
        sigy = vData->getsigy();
        Kx = vData->getKx();
        Ky = vData->getKy();
        mu_x = vData->getMux();
        mu_y = vData->getMuy();
        beta_x = vData->getBetax();
```

```
        beta_y = vData->getBetay();
        k = vData->getk();
        m = vData->getm();
        exm = vData->getexm();
        ex = vData->getex();
        hnot = vData->gethnot();
        epsnot = vData->getepsnot();
        mySign = vData->getmySign();
        phi_eq = vData->getphi_eq();
        return;
}


void FV::setupPvPointers(void){
  if(mode == 0){
    offSet = totalPVvolumes;
    numVars = 1;
  }else{
    offSet = 1;
    numVars = myVars;
  }

  setULPV();
  setURPV();
  setLLPV();
  setLRPV();

  uFjlx = (presV)*directionsPerFlux;
  bFjlx = uFjlx+1;
  lFjly = bFjlx+1;
  rFjly = lFjly+1;

  uFdx = (presV+2*totalFVvolumes)*directionsPerFlux;
  bFdx = uFdx+1;
  lFdy = bFdx+1;
  rFdy = lFdy+1;

  uFsex = (presV+1*totalFVvolumes)*directionsPerFlux;
  bFsex = uFsex+1;
  lFsey = bFsex+1;
  rFsey = lFsey+1;

  uFlncx = (presV+3*totalFVvolumes)*directionsPerFlux;
  bFlncx = uFlncx+1;
  lFlncy = bFlncx+1;
  rFlncy = lFlncy+1;
```

```
  uFlex = (presV+4*totalFVvolumes)*directionsPerFlux;
  bFlex = uFlex+1;
  lFley = bFlex+1;
  rFley = lFley+1;

  uFvx = (presV+5*totalFVvolumes)*directionsPerFlux;
  bFvx = uFvx+1;
  lFvy = bFvx+1;
  rFvy = lFvy+1;

  uFpx = (presV+6*totalFVvolumes)*directionsPerFlux;
  bFpx = uFpx+1;
  lFpy = bFpx+1;
  rFpy = lFpy+1;
  return;
};

void FV::setULPV(void){
        ulPVdiv = myRow*(numPVColumns) + myColumn;
        ulPVhpc = numVars*(myRow*(numPVColumns) + myColumn);
        ulPVphilc = ulPVhpc + philg*offSet;
        ulPVphisc = ulPVhpc + phisg*offSet;
        ulPVpc    = ulPVhpc + 3*offSet;
        return;
};

void FV::setURPV(void){
        urPVdiv = myRow*(numPVColumns) + myColumn + 1;
        urPVhpc = numVars*(myRow*(numPVColumns) + myColumn + 1);
        urPVphilc = urPVhpc + philg*offSet;
        urPVphisc = urPVhpc + phisg*offSet;
        urPVpc    = urPVhpc + 3*offSet;
        return;
};

void FV::setLLPV(void){
        llPVdiv = (myRow+1)*(numPVColumns) + myColumn;
        llPVhpc = numVars*((myRow+1)*(numPVColumns) + myColumn);
        llPVphilc = llPVhpc + philg*offSet;
        llPVphisc = llPVhpc + phisg*offSet;
        llPVpc    = llPVhpc + 3*offSet;
        return;
};

void FV::setLRPV(void){
        lrPVdiv = (myRow+1)*(numPVColumns) + myColumn + 1;
        lrPVhpc = numVars*((myRow+1)*(numPVColumns) + myColumn + 1);
```

```
        lrPVphilc = lrPVhpc + philg*offSet;
        lrPVphisc = lrPVhpc + phisg*offSet;
        lrPVpc    = lrPVhpc + 3*offSet;
        return;
};

void FV::setInitialConditions(volumeIConditions* iData)
{
        eps = iData->geteps();
        epsVptr[presV] = eps;
        socVptr[presV] = soc;
//      soc = socVptr[presV];
//      Vptr[hpc] = iData->gethp();
//      Vptr[phi_lc] = iData->getphi_l();
//      Vptr[phi_sc] = iData->getphi_s();
//      Vptr[jlxc] = iData->getjlx();
        return;
}


void FV::setMaterialPropertyEffectiveValue(double *EffectivePropertyValue,
   double PropertyValue, double tortuosityFactor){
        *EffectivePropertyValue = PropertyValue*pow(eps,tortuosityFactor);
return;
};

void FV::setMaterialPropertyPermeability(){
        invKp = (180*pow((1-eps),2))/(pow(eps,3)*pow(d,2));
        return;
};
/*
void FV::computeFlux(double *Flux, double EMPV, double V2, double V1,
   double spacing){
        *Flux = -EMPV*(V2 - V1)/spacing;
        return;
};
*/

void FV::computeFlux(double *Flux, double EMPV, double V2, double V1,
   double spacing){
        *Flux = -EMPV*(V2 - V1)/spacing;
        return;
};

void FV::computeCurrent(double *Fjl, double Fle, double Flc){
        *Fjl = Fle + Flc;
        return;
```

```
};

double FV::returnQvalue(int v, int xNv, int yNv, int xyNv){
  return myQscale*Vptr[v]
    + myQxNscale*Vptr[xNv]
    + myQyNscale*Vptr[yNv]
    + myQxyNscale*Vptr[xyNv];
};

void FV::computeQvalues(void){
  /* Upper Right CV Quadrature values */
  qVptr[urQhpc]    = returnQvalue(urPVhpc,ulPVhpc,lrPVhpc,llPVhpc);
  qVptr[urQphilc] = returnQvalue(urPVphilc,ulPVphilc,lrPVphilc,llPVphilc);
  qVptr[urQphisc] = returnQvalue(urPVphisc,ulPVphisc,lrPVphisc,llPVphisc);
  qVptr[urQpc]     = returnQvalue(urPVpc,ulPVpc,lrPVpc,llPVpc);

  /* Upper Left CV Quadrature values */
  qVptr[ulQhpc]    = returnQvalue(ulPVhpc,urPVhpc,llPVhpc,lrPVhpc);
  qVptr[ulQphilc] = returnQvalue(ulPVphilc,urPVphilc,llPVphilc,lrPVphilc);
  qVptr[ulQphisc] = returnQvalue(ulPVphisc,urPVphisc,llPVphisc,lrPVphisc);
  qVptr[ulQpc]     = returnQvalue(ulPVpc,urPVpc,llPVpc,lrPVpc);

  /* Lower Left CV Quadrature values */
  qVptr[llQhpc]    = returnQvalue(llPVhpc,lrPVhpc,ulPVhpc,urPVhpc);
  qVptr[llQphilc] = returnQvalue(llPVphilc,lrPVphilc,ulPVphilc,urPVphilc);
  qVptr[llQphisc] = returnQvalue(llPVphisc,lrPVphisc,ulPVphisc,urPVphisc);
  qVptr[llQpc]     = returnQvalue(llPVpc,lrPVpc,ulPVpc,urPVpc);
  /* Lower Right CV Quadrature values */
  qVptr[lrQhpc]    = returnQvalue(lrPVhpc,llPVhpc,urPVhpc,ulPVhpc);
  qVptr[lrQphilc] = returnQvalue(lrPVphilc,llPVphilc,urPVphilc,ulPVphilc);
  qVptr[lrQphisc] = returnQvalue(lrPVphisc,llPVphisc,urPVphisc,ulPVphisc);
  qVptr[lrQpc]     = returnQvalue(lrPVpc,llPVpc,urPVpc,ulPVpc);
  return;
};

void FV::computeFluxes(void){
        // Compute Diffusion Fluxes
  /*
        computeFlux(&fluxVptr[uFdx],Deffx,Vptr[urPVhpc],Vptr[ulPVhpc],dx);
        computeFlux(&fluxVptr[bFdx],Deffx,Vptr[lrPVhpc],Vptr[llPVhpc],dx);
        computeFlux(&fluxVptr[lFdy],Deffy,Vptr[ulPVhpc],Vptr[llPVhpc],dy);
        computeFlux(&fluxVptr[rFdy],Deffy,Vptr[urPVhpc],Vptr[lrPVhpc],dy);
  */
  /*
     computeFlux(&fluxVptr[uFdx],Deffx,Vptr[urPVhpc],Vptr[ulPVhpc],Qdx);
     computeFlux(&fluxVptr[bFdx],Deffx,Vptr[lrPVhpc],Vptr[llPVhpc],Qdx);
     computeFlux(&fluxVptr[lFdy],Deffy,Vptr[ulPVhpc],Vptr[llPVhpc],Qdy);
```

```
    computeFlux(&fluxVptr[rFdy],Deffy,Vptr[urPVhpc],Vptr[lrPVhpc],Qdy);
  */
  //   printf("Qdx = %f and Qdy = %f with Qpnt = %f\n",Qdx,Qdy,Qpnt);

  computeFlux(&fluxVptr[uFdx],Deffx,qVptr[urQhpc],qVptr[ulQhpc],Qdx);
  computeFlux(&fluxVptr[bFdx],Deffx,qVptr[lrQhpc],qVptr[llQhpc],Qdx);
  computeFlux(&fluxVptr[lFdy],Deffy,qVptr[ulQhpc],qVptr[llQhpc],Qdy);
  computeFlux(&fluxVptr[rFdy],Deffy,qVptr[urQhpc],qVptr[lrQhpc],Qdy);

  // Compute Viscosity Fluxes
  computeFlux(&fluxVptr[uFvx],mu_x*slope,Vptr[urPVhpc],Vptr[ulPVhpc],dx);
  computeFlux(&fluxVptr[bFvx],mu_x*slope,Vptr[lrPVhpc],Vptr[llPVhpc],dx);
  computeFlux(&fluxVptr[lFvy],mu_y*slope,Vptr[ulPVhpc],Vptr[llPVhpc],dy);
  computeFlux(&fluxVptr[rFvy],mu_y*slope,Vptr[urPVhpc],Vptr[lrPVhpc],dy);
  /*
  // Compute Chemical Potential Fluxes
  computeFlux(&fluxVptr[uFlncx],olsKscalex,log(Vptr[urPVhpc]),log(Vptr[
     ulPVhpc]),dx);
  computeFlux(&fluxVptr[bFlncx],olsKscalex,log(Vptr[lrPVhpc]),log(Vptr[
     llPVhpc]),dx);
  computeFlux(&fluxVptr[lFlncy],olsKscaley,log(Vptr[ulPVhpc]),log(Vptr[
     llPVhpc]),dy);
  computeFlux(&fluxVptr[rFlncy],olsKscaley,log(Vptr[urPVhpc]),log(Vptr[
     lrPVhpc]),dy);
  */
// Compute Chemical Potential Fluxes
  computeFlux(&fluxVptr[uFlncx],olsKscalex,log(qVptr[urQhpc]),log(qVptr[
     ulQhpc]),Qdx);
  computeFlux(&fluxVptr[bFlncx],olsKscalex,log(qVptr[lrQhpc]),log(qVptr[
     llQhpc]),Qdx);
  computeFlux(&fluxVptr[lFlncy],olsKscaley,log(qVptr[ulQhpc]),log(qVptr[
     llQhpc]),Qdy);
  computeFlux(&fluxVptr[rFlncy],olsKscaley,log(qVptr[urQhpc]),log(qVptr[
     lrQhpc]),Qdy);
  // Compute Liquid Current Fluxes
  /*
  computeFlux(&fluxVptr[uFlex],Keffx,Vptr[urPVphilc],Vptr[ulPVphilc],dx);
  computeFlux(&fluxVptr[bFlex],Keffx,Vptr[lrPVphilc],Vptr[llPVphilc],dx);
  computeFlux(&fluxVptr[lFley],Keffy,Vptr[ulPVphilc],Vptr[llPVphilc],dy);
  computeFlux(&fluxVptr[rFley],Keffy,Vptr[urPVphilc],Vptr[lrPVphilc],dy);
  */

  computeFlux(&fluxVptr[uFlex],Keffx,qVptr[urQphilc],qVptr[ulQphilc],Qdx);
  computeFlux(&fluxVptr[bFlex],Keffx,qVptr[lrQphilc],qVptr[llQphilc],Qdx);
  computeFlux(&fluxVptr[lFley],Keffy,qVptr[ulQphilc],qVptr[llQphilc],Qdy);
  computeFlux(&fluxVptr[rFley],Keffy,qVptr[urQphilc],qVptr[lrQphilc],Qdy);
```

```
  /*
    computeFlux(&fluxVptr[uFjlx],Keffx,Vptr[urPVphilc],Vptr[ulPVphilc],dx);
    computeFlux(&fluxVptr[bFjlx],Keffx,Vptr[lrPVphilc],Vptr[llPVphilc],dx);
    computeFlux(&fluxVptr[lFjly],Keffy,Vptr[ulPVphilc],Vptr[llPVphilc],dy);
    computeFlux(&fluxVptr[rFjly],Keffy,Vptr[urPVphilc],Vptr[lrPVphilc],dy);
  */
  // Compute Solid Current Fluxes
  computeFlux(&fluxVptr[uFsex],sigeffx,qVptr[urQphisc],qVptr[ulQphisc],Qdx)
      ;
  computeFlux(&fluxVptr[bFsex],sigeffx,qVptr[lrQphisc],qVptr[llQphisc],Qdx)
      ;
  computeFlux(&fluxVptr[lFsey],sigeffy,qVptr[ulQphisc],qVptr[llQphisc],Qdy)
      ;
  computeFlux(&fluxVptr[rFsey],sigeffy,qVptr[urQphisc],qVptr[lrQphisc],Qdy)
      ;
  /*
 computeFlux(&fluxVptr[uFsex],sigeffx,Vptr[urPVphisc],Vptr[ulPVphisc],dx);
  computeFlux(&fluxVptr[bFsex],sigeffx,Vptr[lrPVphisc],Vptr[llPVphisc],dx);
  computeFlux(&fluxVptr[lFsey],sigeffy,Vptr[ulPVphisc],Vptr[llPVphisc],dy);
  computeFlux(&fluxVptr[rFsey],sigeffy,Vptr[urPVphisc],Vptr[lrPVphisc],dy);
  */
  // Compute the Liquid Phase ElectroChemical Fluxes
  computeCurrent(&fluxVptr[uFjlx],fluxVptr[uFlex],fluxVptr[uFlncx]);
  computeCurrent(&fluxVptr[bFjlx],fluxVptr[bFlex],fluxVptr[bFlncx]);
  computeCurrent(&fluxVptr[lFjly],fluxVptr[lFley],fluxVptr[lFlncy]);
  computeCurrent(&fluxVptr[rFjly],fluxVptr[rFley],fluxVptr[rFlncy]);

  // Compute Pressure Fluxes
  computeFlux(&fluxVptr[uFpx],1,Vptr[urPVpc],Vptr[ulPVpc],dx);
  computeFlux(&fluxVptr[bFpx],1,Vptr[lrPVpc],Vptr[llPVpc],dx);
  computeFlux(&fluxVptr[lFpy],1,Vptr[ulPVpc],Vptr[llPVpc],dy);
  computeFlux(&fluxVptr[rFpy],1,Vptr[urPVpc],Vptr[lrPVpc],dy);
  return;
};

void FV::computeMaterialProperties(void){
        setMaterialPropertyEffectiveValue(&Deffx,Dx,ex);
        setMaterialPropertyEffectiveValue(&Deffy,Dy,ex);
        setMaterialPropertyEffectiveValue(&Keffx,Kx,ex);
        setMaterialPropertyEffectiveValue(&Keffy,Ky,ex);
        setMaterialPropertyEffectiveValue(&sigeffx,sigx,exm);
        setMaterialPropertyEffectiveValue(&sigeffy,sigy,exm);
        setMaterialPropertyPermeability();
        olsKscalex = Keffx*olsScale;
        olsKscaley = Keffy*olsScale;
        return;
};
```

```
void FV::computeDiv(void){
  myDiv = (divVptr[ulPVdiv])*myCVvolumeRatio[myUL]
                  + divVptr[urPVdiv]*myCVvolumeRatio[myUR]
                  + divVptr[llPVdiv]*myCVvolumeRatio[myLL]
                  + divVptr[lrPVdiv]*myCVvolumeRatio[myLR];
        return;
};

/*
void FV::computeDiv(void){
  myDiv = divVtwoptr[ulPVdiv + 1]
    + divVtwoptr[ulPVdiv + 2]
    + divVtwoptr[urPVdiv + 1]
    + divVtwoptr[urPVdiv + 3]
    + divVtwoptr[llPVdiv + 0]
    + divVtwoptr[llPVdiv + 2]
    + divVtwoptr[lrPVdiv + 0]
    + divVtwoptr[lrPVdiv + 3];
  return;
};
*/
/*********************************/
/********* EPS ******************/
/*********************************/
void FV::computeNextEPS(void){
  eps = eps + (dt*myVfd2*myDiv)/myVolume;
        epsVptr[presV] = eps;
        //      printf("Pres %d Updated EPS with %0.16f\n and myDiv %0.16f
           and dt %f and myVfd2 %f\n",presV,eps,myDiv,dt,myVfd2);
        return;
}

/*********************************/
/********* SOC ******************/
/*********************************/
void FV::computeNextSOC(void)
{
        soc = soc + mySign*(dt*myDiv)/Qmax;
        socVptr[presV] = soc;
        return;
}


/*********************************/
/****** Diffusion Fluxes *********/
```

```
/*********************************/
double FV::returnFdx(int p){
        if(p == UL || p == UR)
                return fluxVptr[bFdx];
        else if( p == LL || p == LR)
                return fluxVptr[uFdx];
        else
        return 0;
};

double FV::returnFdy(int p){
        if(p == UL || p == LL)
                return fluxVptr[lFdy];
        else if (p == UR || p == LR)
                return fluxVptr[rFdy];
        else
        return 0;
};


/*********************************/
/*** Quadrature Concentrations ***/
/*********************************/
double FV::returnQhp(int p){
        if(p == UR)
          return qVptr[llQhpc];
        else if(p == UL)
          return qVptr[lrQhpc];
        else if(p == LL)
          return qVptr[urQhpc];
        else if(p == LR)
          return qVptr[ulQhpc];
        else
        return 0;
};

double FV::returnQxNhp(int p){
        if(p == UR)
          return qVptr[lrQhpc];
        else if(p == UL)
          return qVptr[llQhpc];
        else if(p == LL)
          return qVptr[ulQhpc];
        else if(p == LR)
          return qVptr[urQhpc];
        else
        return 0;
```

```
};

double FV::returnQyNhp(int p){
        if(p == UR)
          return qVptr[ulQhpc];
        else if(p == UL)
          return qVptr[urQhpc];
        else if(p == LL)
          return qVptr[lrQhpc];
        else if(p == LR)
          return qVptr[llQhpc];
        else
        return 0;
};

/*********************************/
/* Current in Solution Fluxes *****/
/*********************************/
double FV::returnFjlx(int p){
        if(p == UL || p == UR)
                return fluxVptr[bFjlx];
        else if( p == LL || p == LR)
                return fluxVptr[uFjlx];
        else
        return 0;
};

double FV::returnFjly(int p){
        if(p == UL || p == LL)
                return fluxVptr[lFjly];
        else if (p == UR || p == LR)
                return fluxVptr[rFjly];
        else
        return 0;
};

/*********************************/
/* Current in Solid Fluxes ********/
/*********************************/
double FV::returnFsex(int p){
        if(p == UL || p == UR)
                return fluxVptr[bFsex];
        else if (p == LL || p == LR)
                return fluxVptr[uFsex];
        else
        return 0;
};
```

```
double FV::returnFsey(int p){
        if(p == UL || p == LL)
                return fluxVptr[lFsey];
        else if (p == UR || p == LR)
                return fluxVptr[rFsey];
        else
        return 0;
};

/*********************************/
/* x direction Material Coefficients */
/*********************************/
double FV::returnDeffx(void){
        return Deffx;
};

double FV::returnKeffx(void){
        return Keffx;
};

double FV::returnolsKscalex(void){
        return olsKscalex;
};

double FV::returnsigeffx(void){
        return sigeffx;
};

double FV::returnDVx(void){
        return mu_x;
};

double FV::returnVECx(void){
        return beta_x;
};

double FV::returndx(void){
        return dx;
};

double FV::returnQdx(void){
        return Qdx;
};

/*********************************/
/* y direction Material Coefficients */
```

```
/*********************************/
double FV::returnDeffy(void){
        return Deffy;
};

double FV::returnKeffy(void){
        return Keffy;
};

double FV::returnolsKscaley(void){
        return olsKscaley;
};

double FV::returnsigeffy(void){
        return sigeffy;
};

double FV::returnDVy(void){
        return mu_y;
};

double FV::returnVECy(void){
        return beta_y;
};

double FV::returndy(void){
        return dy;
};

double FV::returndz(void){
        return dz;
};

double FV::returnQdy(void){
        return Qdy;
};

double FV::returnQdz(void){
        return Qdz;
};

void FV::setdx(double d){
        dx = d;
        return;
};

void FV::setCVvolume(int position, double vol){
```

```
        myCVvolume[position] = vol;
        return;
};

void FV::setCVvolumeRatio(int position, double volRatio){
        myCVvolumeRatio[position] = volRatio;
        return;
};

double FV::returneps(void){
        return eps;
};

double FV::soceta(void){
        return pow(soc,eta);
};

double FV::returnFvisc_x(int position){
  return 0;
};

double FV::returnFvisc_y(int position){
  return 0;
};

double FV::returnFp_x(int position){
  return 0;
};

double FV::returnFp_y(int position){
  return 0;
};
```

Listing B.59. cellSizeData.h

```
/**
 * cellSizeData.h
 *
 * @author James Geraci
 *
 */

#ifndef _CELLSIZEDATA_H
#define _CELLSIZEDATA_H

class cellSizeData
{
public:
        cellSizeData(void);
public:
        ~cellSizeData(void);

public:
        void setnumofElectrodes(int);

        void setIdxFVColumns(int);
        void setElectrolyteFVColumns(int);
        void setIdFVColumns(int);

        void setIdxPVColumns(int);
        void setElectrolytePVColumns(int);
        void setIdPVColumns(int);

        void setnumFVRows(int);
        void setnumPVRows(int);


        void setnumFVZrows(int);
        void setnumPVZrows(int);

        void computeTotalFVColumns(void);
        void computeTotalPVColumns(void);


        void computeTotalFVVolumes(void);
        void computeTotalPVVolumes(void);

        int getIdxFVColumns(void);
        int getElectrolyteFVColumns(void);
        int getIdFVColumns(void);
```

```
        int getIdxPVColumns(void);
        int getElectrolytePVColumns(void);
        int getIdPVColumns(void);

        int getnumPVZrows(void);
        int getnumPVRows(void);
        int getnumPVColumns(void);

        int getnumFVZrows(void);
        int getnumFVRows(void);
        int getnumFVColumns(void);

        int gettotalFVVolumes(void);
        int gettotalPVVolumes(void);

        int getnumofElectrodes(void);

private:
        int numofElectrodes;

        int IdxFVColumns;
        int ElectrolyteFVColumns;
        int IdFVColumns;

        int IdxPVColumns;
        int ElectrolytePVColumns;
        int IdPVColumns;

        int numFVRows;
        int numFVZrows;
        int numFVColumns;

        int numPVRows;
        int numPVZrows;
        int numPVColumns;

        int totalFVColumns;
        int totalPVColumns;

        int totalFVVolumes;
        int totalPVVolumes;
};

#endif
```

Listing B.60. cellSizeData.cpp

```
/**
 * cellSizeData.cpp
 * @author James Geraci
 * provides the interface to set/fetch the number of columns and rows
 * in each region of the battery cell
 *
 */
#include "StdAfx.h"
#include "cellSizeData.h"

cellSizeData::cellSizeData(void)
{
}

cellSizeData::~cellSizeData(void)
{
}


void cellSizeData::setnumofElectrodes(int nE){
        numofElectrodes = nE;
        return;
};

void cellSizeData::setIdxFVColumns(int IdxFVc){
        IdxFVColumns = IdxFVc;
        return;
};

void cellSizeData::setElectrolyteFVColumns(int eFVc){
        ElectrolyteFVColumns = eFVc;
        return;
};

void cellSizeData::setIdFVColumns(int IdFVc){
        IdFVColumns = IdFVc;
        return;
};

void cellSizeData::setIdxPVColumns(int IdxPVc){
        IdxPVColumns = IdxPVc;
        return;
};

void cellSizeData::setElectrolytePVColumns(int ePVc){
        ElectrolytePVColumns = ePVc;
```

```
        return;
};

void cellSizeData::setIdPVColumns(int IdPVc){
        IdPVColumns = IdPVc;
        return;
};


void cellSizeData::setnumFVRows(int FVrows){
        numFVRows = FVrows;
        return;
};

void cellSizeData::setnumPVRows(int PVrows){
        numPVRows = PVrows;
        return;
};


void cellSizeData::setnumFVZrows(int FVZrows){
        numFVZrows = FVZrows;
        return;
};

void cellSizeData::setnumPVZrows(int PVZrows){
        numPVZrows = PVZrows;
        return;
};

void cellSizeData::computeTotalFVColumns(void){
        totalFVColumns = IdxFVColumns + ElectrolyteFVColumns + IdFVColumns;
        return;
};

void cellSizeData::computeTotalPVColumns(void){
        totalPVColumns = IdxPVColumns + ElectrolytePVColumns + IdPVColumns;
        return;
};

void cellSizeData::computeTotalFVVolumes(void){
        totalFVVolumes = totalFVColumns*numFVRows*numFVZrows;
        return;
};

void cellSizeData::computeTotalPVVolumes(void){
        totalPVVolumes = totalPVColumns*numPVRows*numPVZrows;
```

```
        return;
};


int cellSizeData::getnumPVZrows(void){
        return numPVZrows;
};

int cellSizeData::getnumPVRows(void){
        return numPVRows;
};

int cellSizeData::getnumPVColumns(void){
        return totalPVColumns;
};


int cellSizeData::getnumFVZrows(void){
        return numFVZrows;
};

int cellSizeData::getnumFVRows(void){
        return numFVRows;
};

int cellSizeData::getnumFVColumns(void){
        return totalFVColumns;
};


int cellSizeData::gettotalFVVolumes(void){
        return totalFVVolumes;
};


int cellSizeData::gettotalPVVolumes(void){
        return totalPVVolumes;
};

int cellSizeData::getnumofElectrodes(void){
        return numofElectrodes;
};

int cellSizeData::getIdxFVColumns(void){
        return IdxFVColumns;
};
```

```
int cellSizeData::getElectrolyteFVColumns(void){
        return  ElectrolyteFVColumns;
};

int cellSizeData::getIdFVColumns(void){
        return IdFVColumns;
};

int cellSizeData::getIdxPVColumns(void){
        return IdxPVColumns;
};

int cellSizeData::getElectrolytePVColumns(void){
        return ElectrolytePVColumns;
};

int cellSizeData::getIdPVColumns(void){
        return IdPVColumns;
};
```

Listing B.61. volumeIConditions.h

```
/* @author James Geraci
 */
#ifndef VOLUMEICONDITIONS_H
#define VOLUMEICONDITIONS_H

class volumeIConditions
{
public:
        volumeIConditions(void);
        volumeIConditions(double,double,double);
public:
        ~volumeIConditions(void);

public:
        double geteps(void);
        double getjlx(void);
        double getjly(void);

private:
        double eps;
        double jlx;
        double jly;
};

#endif
```

Listing B.62. volumeIConditions.cpp

```
/*
 * @author James Geraci
 * sets Initial Conditions for FV volumes
 */
#include "StdAfx.h"
#include "volumeIConditions.h"

volumeIConditions::volumeIConditions(void)
{
}

volumeIConditions::volumeIConditions(double e,double x,double y)
{
        eps = e;
        jlx = x;
        jly = y;
}

volumeIConditions::~volumeIConditions(void)
{
}

double volumeIConditions::geteps(void)
{
        return eps;
};

double volumeIConditions::getjlx(void)
{
        return jlx;
};

double volumeIConditions::getjly(void)
{
        return jly;
};
```

Listing B.63. matrixData.h

```
/* @author James Geraci
 */
#ifndef _MATRIXDATA_H
#define _MATRIXDATA_H
#include "cellSizeData.h"
#include <stdio.h>

#include <libspe2.h>
#include "common.h"
#include "ctdef.h"

class matrixData
{
public:
        matrixData(void);
        matrixData(cellSizeData*,int);
public:
        ~matrixData(void);

public:

        ppu_pthread_data_t* datas;
        spe_program_handle_t* ph;
        //      spe_program_handle_t spu_prog;
        //void* ppu_pthread_function(void*);
        void startSPEs(const char*);
        //CONTROL_BLOCK_T* myCB;
        void restoreiVindex(void);
        //      void sendNUM_SPUS(void);
        void sendSPE_RANKS(void);
        void sendData(int);
        char file_name[1024];
        void openDataFile(const char*);
        void closeDataFile(void);
        void writeDataFiletoDisk(void);
        double collectedDataCounter;
        double collectedDataVars;
        FILE* Vfile;
        FILE* Vfilebefore;
        FILE* Vfileafter;
        void writeVtoDisk(double*,int);
        void writeiVbeforetoDisk(double*,int);
        void writeiVaftertoDisk(double*,int);
        void writeJtoDisk(double*,int);
        double collectedData[5];
        int collectedDataSize;
```

```
//void checkDMAqueueStatus(void);
//       void sendPVJrows(int);
unsigned int cmd_status;
int alignedRowSize;
int neededExtraBytes;
int jhpc;
double *val_CRS, *col_ind, *row_ptr;
int sparseMatrixSize;
int ActualsparseMatrixSize;
unsigned int *nzInRow;
unsigned int* spu_buffers;
double** myJacobian(void);
double* myfluxV(void);
double* mysocV(void);
double* mydivV(void);
double* mydivVtwo(void);
double* myepsV(void);
double** myiV(void);

double* mysV(void);
double* mypiV(void);
double* mybiV(void);
double* mypsV(void);
double* myRowBuffer(void);
double* myExtraBuffer1(void);
double* myavePressures_x(void);
double* myavePressures_y(void);
double* myAVEhp_x(void);
double* myAVEhp_y(void);
//double* rhs_address(void);
int* myintBuffer(void);

int myJacobByteSize(void);
int myJacobSize(void);
int mysvLength(void);
int mySvByteSize(void);

int myPVColumns(void);
int myPVRows(void);
int myPVZrows(void);

int myFVColumns(void);
int myFVRows(void);
int myFVZrows(void);
int myJhpc(void);
int ReturnTotalFVvolumes(void);
```

```
        double* myJacobianStorage(void);
        //      double* JacobianStorageBackup(void);
        void setpiV(void);
        //void resetiV(void);
        void resetdivV(void);
        void resetdivVtwo(void);
        void resetJacobian(void);
        void resetRowBuffer(void);
        void resetFluxes(void);
        void restoresV(void);
        void storesV(void);
        //      void copyiV2rhs(void);
        //      void copyrhs2iV(void);
        /* void cpJacobiantoMatlab(void); */
/*      void cpurJacobiantoMatlab(void); */
/*      void cpIVtoMatlab(void); */
/*      void cpSVtoMatlab(void); */
/*      void cppSVtoMatlab(void); */
        void setInitialSV(void);
/*      void cpfluxVtoMatlab(void); */
/*      void cpdivVtoMatlab(void); */
/*      void cpepsVtoMatlab(void); */
/*      void cpsocVtoMatlab(void);*\/ */
        void applyBoundaryConditions(void);
        void applyPreConditioner(void);
/*      void getSVfromMatlab(void); */
/*      void getIVfromMatlab(void); */
/*      void computnIVinMatlab(void); */
/*      void makeGraphs(void); */
        void computeIV(void);
        //void computeIV(speid_t*, double*,double*);
        void clearColumn(int);
        //void startSPUs(void);
        void barrier(void);
        //      void barrier2(void);
        //      unsigned int SPU_address[6];
        bool* SPU_read;
        //      spe_context_ptr_t* speids;
        //      double** rhs;
        //      unsigned int sum;
        unsigned int sum_delta;
        int present_base_row;
        //double rhsBuffer;
        double* bufaddrf;
        int svLength; // = numColumns*numRows*6;
        int hbWidth;
        int hbWbytes;
```

```
        int bWidth;
        int bWbytes;
        double* JacobianStorage;
        double* JacobianStorageBackup;
        double** Jacobian;
        double* sV;
        double** iV;  // intermediate values vector
        double* iV2;
        //unsigned int* iV2;
        double iVx;
        double iVy;
        double* avePressures_y;
        double* avePressures_x;

private:
        void startNSPEs(int);
        const char* myProg;
        void setupCB(void);
        CONTROL_BLOCK_T* cb CACHE_ALIGNED;
        int* xSPE;
        unsigned int numFailed;
        //      spe_mfc_command_area_t* spu_cmd;
        //      int pvt;
        int edge;
        int NUM_SPE;
        unsigned int workingSPEs;
        double et_start,et_end,et_total;
        double et_start_rowx,et_end_rowx,et_total_rowx;
        double et_start_bs,et_end_bs,et_total_bs;
        double barrier2_time_total, barrier2_time_start, barrier2_time_end;
        int computePivot(int);
        unsigned int r;
        double mean;
        double std;
        double stdTime;
        double AveTime;
        double TotalTime;
        double elapsedTime;
        double tic_time;
        double toc_time;
        double tic;
        double toc;
        double elapsed;
        bool first_call;
        //      Engine *ep;
        //MATFile *pmat;
        //      mxArray *fluxVmat, *divVmat, *epsVmat, *socVmat, *psVmat, *
```

```
    sVmat, *iVmat, *savesVmat, *piVmat, *rowBuffermat, *JacobianMat;
double* fluxV;
double* divV;
double* divVtwo;
double* epsV;
double* socV;
double* psV;

double* savesV;

double* piV; // the Past difference
double* biV; // the Past difference

double* extraBuffer1;
double* AVEhp_y;
double* AVEhp_x;
int* intBuffer;

double max_pivot_value;
unsigned int max_pivot;

int svByteSize;
int FVsvLength;
int FVsvByteSize;
int JacobSize;
int JacobByteSize;
int eyeMaker;

int ldxPVColumns;
int ElectrolytePVColumns;
int ldPVColumns;

int f;
int rc;
int numPVColumns;
int numPVRows;
int numPVZrows;
int totalPVvolumes;

int numFVColumns;
int numFVRows;
int numFVZrows;
int totalFVvolumes;

int reference;
int phisc;
int philc;
```

```
        int refB;
        int ekBoundary;
};

#endif
```

Listing B.64. matrixData.cpp

```
/*
 * @author James Geraci
 * @second author Sudarshan Raghunathan
 * contains all data vectors and jacobian matrix
 *
 */
#include "matrixData.h"
#include "StdAfx.h"
#include <iostream>
#include<sys/time.h>
#include <stdio.h>
#include <errno.h>

extern "C"{
void* ppu_pthread_function(void* arg){
  ppu_pthread_data_t* datap = reinterpret_cast<ppu_pthread_data_t*>(arg);
  int rc;
  rc = pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
  unsigned int entry = SPE_DEFAULT_ENTRY;
  spe_stop_info_t stop_info;
  if ((rc = spe_context_run(datap->speid, &entry, 0, datap->argp, NULL, &
      stop_info)) < 0){
    fprintf(stderr, "Failed spe_context_run(rc=%d, errno=%d, strerror=%s)\n
      ", rc, errno, strerror(errno));
    exit(1);
  }

  // checkResults("pthread_setcanceltype()\n", rc);

  //printf("RC = %d\n",rc);
  pthread_exit(NULL);
  return NULL;
}
typedef void* (*PPF)(void*);
}

PPF px = ppu_pthread_function;

static inline double gettime()
{
 timeval t;
 if (gettimeofday(&t, 0) != 0)
   perror("gettimeofday");

 return (t.tv_usec / 1000000.0) + t.tv_sec;
}
```

```
matrixData::~matrixData(void)
{
  free_align((void*) val_CRS);
  free_align((void*) row_ptr);
  free_align((void*) col_ind);
  free_align((void*) nzInRow);
  // free_align((void*) rhs);
  //  free_align((FILE*) Vfile);
  free_align((void*)divV);
  free_align((void*)fluxV);
  free_align((void*)epsV);
  free_align((void*)socV);
  free_align((void*)psV);
  free_align((void*)sV);
  free_align((void*)savesV);
  free_align((void*)iV);
  free_align((void*)piV);
  free_align((void*)extraBuffer1);
  free_align((void*)intBuffer);
  free_align((void*)JacobianStorage);
  free_align((void*)Jacobian);
  free_align((void*)avePressures_y);
  free_align((void*)avePressures_x);
  free_align((void*)AVEhp_x);
  free_align((void*)AVEhp_y);
}


matrixData::matrixData(cellSizeData* csData, int Nspu)
{
  Vfile = NULL;
  NUM_SPE = Nspu;
  numFailed = 0;
  TotalTime = 0;
  AveTime = 100;
  stdTime = 100;
  mean = 0;
  std = 0;

  first_call = true;
  collectedDataCounter = 0;
  collectedDataVars = 5;
  IdxPVColumns = csData->getIdxPVColumns();
  ElectrolytePVColumns = csData->getElectrolytePVColumns();
  IdPVColumns = csData->getIdPVColumns();
  numPVColumns = csData->getnumPVColumns();
```

```
numPVRows = csData->getnumPVRows();
numPVZrows = csData->getnumPVZrows();
totalPVvolumes = numPVColumns*numPVRows*numPVZrows;
// printf("numPVColumns = %d, numPVRows = %d, myVars = %d\n",numPVColumns,
   numPVRows,myVars);
refB = 1;  // always 1
ekBoundary = 2;  // change to 2 for phil before phis
reference = (totalPVvolumes-1)*myVars + refB;
numFVColumns = csData->getnumFVColumns();
numFVRows = csData->getnumFVRows();
numFVZrows = csData->getnumFVZrows();
totalFVvolumes = numFVColumns*numFVRows*numFVZrows;

svLength = totalPVvolumes*myVars;
svByteSize = svLength*sizeof(double);
std::cout << "Original svByteSize " << svByteSize << std::endl;


FVsvLength = totalFVvolumes*myVars;  // Used to be totalFVvolumes*(myVars
   +2), but I don't know why
FVsvByteSize = FVsvLength*sizeof(double);
// Compute Number of Variables in Sparse Matrix
int ADEGH = 5*totalPVvolumes - 2*(totalPVvolumes/numPVRows) - 2*(
   numPVRows - 1) - 2;
int R = totalPVvolumes - numPVRows*(ElectrolytePVColumns);
int F = ADEGH - 2*numPVRows - 5*ElectrolytePVColumns*numPVRows + 2*(
   ElectrolytePVColumns+1);
int B = ADEGH - 5*(ElectrolytePVColumns + 1)*numPVRows + 2*(
   ElectrolytePVColumns + 1);

int niner = 1;
//  hbWidth = (numPVColumns+1)*myVars;
hbWidth = (numPVColumns+1+niner)*myVars; // modified September 18
hbWidth += hbWidth%2;
hbWbytes = hbWidth*sizeof(double);
//  jhpc = numPVColumns*myVars;
jhpc = (numPVColumns+niner)*myVars; // modified September 18
//jhpc = hbWidth;//numPVColumns*myVars;
bWidth = jhpc + hbWidth;
std::cout << std::endl << "jhpc = " << jhpc << " hbWidth = " << hbWidth
    << std::endl;
bWbytes = bWidth*sizeof(double);


//  neededExtraBytes = ((myVars-1) + (myVars-1)%2)*sizeof(double) +
   bWbytes%QUAD_ALIGNMENT;
neededExtraBytes = myVars*sizeof(double) + bWbytes%QUAD_ALIGNMENT;
```

```
// alignedRowSize = bWbytes+neededExtraBytes + 2*sizeof(double) /* for
   rhs */;
alignedRowSize = bWbytes + 4*sizeof(double);// + neededExtraBytes/* for
   rhs */;
// bWidth = alignedRowSize/sizeof(double);
bWidth += 4; // This adds the space at the end for the iV and pbr count
std::cout << "bWidth = " << bWidth << " bWbytes = " << bWbytes << std::
   endl
        << "hbWidth = " << hbWidth << " hbWbytes = "<< hbWbytes << std
           ::endl
     << " needed ExtraBytes = " << neededExtraBytes << " alignedRowSize
        = " << alignedRowSize << std::endl;

JacobSize = svLength*(bWidth);
std::cout << "Number of Elements in the Jacobian " << JacobSize << std::
   endl;
JacobByteSize = JacobSize*sizeof(double);//svByteSize*alignedRowSize;
// std::cout << "Jacob Byte size = " << JacobByteSize << " jhpc = " <<
   jhpc << std::endl;
ActualsparseMatrixSize = 5*ADEGH + R + F + B;
int maxnzPerRow = 16;
sparseMatrixSize = myVars*maxnzPerRow*totalPVvolumes;
f = totalFVvolumes*fluxesPerVol*directionsPerFlux*sizeof(double);
//printf("f = %d\n",totalFVvolumes*fluxesPerVol*directionsPerFlux*sizeof(
   double));
try{
  val_CRS = (double*) malloc_align(sparseMatrixSize*sizeof(double),7);
  row_ptr = (double*) malloc_align(sparseMatrixSize*sizeof(double),7);
  col_ind = (double*) malloc_align(sparseMatrixSize*sizeof(double),7);
  nzInRow = (unsigned int*) malloc_align(svLength*sizeof(unsigned int),7)
     ;
  spu_buffers = (unsigned int*) malloc_align(NUM_SPE*sizeof(unsigned int)
     ,7);
  //rhs = (double**) malloc_align (svLength * sizeof(double*), 7);
  socV = (double*) malloc_align(totalFVvolumes*sizeof(double),7);
  fluxV = (double*) malloc_align(f,7);
  divV = (double*) malloc_align(totalPVvolumes*sizeof(double),7);
  divVtwo = (double*) malloc_align(4*totalPVvolumes*sizeof(double),7);
  epsV = (double*) malloc_align(totalFVvolumes*sizeof(double),7);
  // can put divV here too.
  psV = (double*) malloc_align(svByteSize,7);
  sV = (double*) malloc_align(svByteSize,7);
  savesV = (double*) malloc_align(svByteSize,7);
  iV = (double**) malloc_align (svLength * sizeof(double*), 7);
  piV = (double*) malloc_align(svByteSize,7);
  biV = (double*) malloc_align(svByteSize,7);
  iV2 = (double*) malloc_align(svByteSize*sizeof(double*),7);
```

```
    extraBuffer1 = (double*) malloc_align(svByteSize,7);
    intBuffer = (int*) malloc_align(svLength*sizeof(int),7);
    //spu_rhs = reinterpret_cast<double*>(malloc_align(svByteSize * 2, 7));
    Jacobian = (double**) malloc_align(svLength*sizeof(double*),7);
    JacobianStorage = (double*) malloc_align(JacobByteSize,7);
    JacobianStorageBackup = (double*) malloc_align(JacobByteSize,7);
    avePressures_y = reinterpret_cast<double*>( malloc_align(numFVRows*
        numPVColumns*sizeof(double),7));
    avePressures_x = reinterpret_cast<double*>( malloc_align(numFVColumns*
        numPVRows*sizeof(double),7));
    AVEhp_y = reinterpret_cast<double*>( malloc_align(numFVRows*
        numPVColumns*sizeof(double),7));
    AVEhp_x = reinterpret_cast<double*>( malloc_align(numFVColumns*
        numPVRows*sizeof(double),7));
    SPU_read = reinterpret_cast<bool*>(malloc_align(NUM_SPE*sizeof(bool),7)
        );
    //printf("The size of a bool is %d and the size of unsigned int is %d\n
        ", sizeof(bool), sizeof(unsigned int));
    //printf("numFVRows*numPVColumns = %d\n",numFVRows*numPVColumns);
    resetJacobian();
    for(int i =0; i < svLength; i++){
      Jacobian[i] = &JacobianStorage[i*bWidth];
    }
    //printf("You are starting to assign rhs pointers\n");
  for(int index = 0; index < svLength; index++){
    //  rhs[index] = &JacobianStorage[(index+1)*bWidth - 2];
    iV[index] = &JacobianStorage[(index+1)*bWidth - 2];
    JacobianStorage[(index+1)*bWidth - 3] = -1;
    //iV2[index] = (unsigned int) &JacobianStorage[(index+1)*bWidth - 2];
  }
 //printf("You have assigned rhs pointers\n");
  }
  catch(std::bad_alloc)
    {
      std::cout << "Aligned Malloc Align has Failed!" << std::endl;
      exit(-1);
    }
  for(int d = 0; d < sparseMatrixSize; d++)
    {
      val_CRS[d] = 0;
      row_ptr[d] = -1;
      col_ind[d] = -1;
    }
  xSPE = reinterpret_cast<int*>(malloc_align(NUM_SPE*sizeof(int),7));
  //  iV2 = &iV[0][0];
  memset(JacobianStorage,0,JacobByteSize);
  memset(JacobianStorageBackup,0,JacobByteSize);
```

```
  memset(fluxV,0,f);
  memset(socV,0,totalFVvolumes*sizeof(double));
  memset(divV,0,totalPVvolumes*sizeof(double));
  memset(epsV,0,totalFVvolumes*sizeof(double));
  memset(piV,0,svByteSize);
  // memset(iV,0,svByteSize);
  memset(sV,0,svByteSize);
  memset(psV,0,svByteSize);
  memset(extraBuffer1,0,svByteSize);
  memset(savesV,0,svByteSize);

  //  memset(spu_buffers,0,2*NUM_SPU*sizeof(unsigned int));
  // memset(rhs,0,2*svByteSize);

  //printf("Matrix construction completed.\n");
  //printf("spu_buffers has been set to %d\n",spu_buffers[0]);
}

void matrixData::restoreiVindex(void){
  for(int index = 0; index < svLength; index++){
      iV[index][1] = (double) index;
  }
  return;
}

void matrixData::startSPEs(const char* myP){
  myProg = myP;
  // NUM_SPE = 1;
  //printf("About to start SPEs\n");
  try{
      cb = reinterpret_cast<CONTROL_BLOCK_T*>(malloc_align(NUM_SPE*sizeof(
         CONTROL_BLOCK_T),7));
    }catch(std::bad_alloc){
    //printf("cb was not properly allocated\n");
    exit(-1);
  }
  for(int p = 0; p < NUM_SPE; p++){
    cb[p].original_rank_modified = 0;
  }
  try{
  datas = reinterpret_cast<ppu_pthread_data_t*>(malloc_align(NUM_SPE*sizeof
     (ppu_pthread_data_t),7));
  }catch(std::bad_alloc){
    //printf("datas not propely allocated\n");
    exit(-1);
  }
  //printf("cb and datas ok\n");
```

```
  int rc = 0 ;
  workingSPEs = 0;
  //printf("The number of physical SPEs is %d\n",spe_cpu_info_get(
     SPE_COUNT_PHYSICAL_CPU_NODES, -1));
  //printf("The number of usable SPEs is %d\n",spe_cpu_info_get(
     SPE_COUNT_USABLE_SPES, -1));
  //  Get the SPE threads running
  startNSPEs(NUM_SPE);
  //  workingSPEs = 0;
  //startNSPEs(NUM_SPE);

  //printf("YOU GOT PAST the THREAD CREATES with %d WORKINGSPES out of %d
     requested SPEs\n",workingSPEs,NUM_SPE);
  setupCB();

  //printf("YOU GOT IN FRONT OF THE BARRIER\n");
  barrier();
  //printf("YOU GOT PAST THE BARRIER\n");
  sendData(NUM_SPE);
  // mData->barrier2();
  barrier();
  sendSPE_RANKS();
  barrier();
  sendSPE_RANKS();
  barrier();
  return;
}


void matrixData::startNSPEs(int Nspe){
  int wspeStart = workingSPEs;
  int pp;
  for (int i = wspeStart; i < Nspe + wspeStart; i++) {
    /* Create context */
    //printf("going to create spe %d\n",i);
    if ((datas[i].speid = spe_context_create (SPE_MAP_PS, NULL)) == NULL){
        fprintf (stderr, "Failed spe_context_create(errno=%d strerror=%s)\n
           ", errno, strerror(errno));
        exit (3+i);
      }
    //  std::cout << "past spe_context_create" << std::endl;
    // std::cin >> pp;
    /* Open Program Image */
    if ((ph = spe_image_open (myProg) ) == NULL){
        fprintf (stderr, "Failed spe_image_open(errno=%d strerror=%s)\n",
           errno, strerror(errno));
        exit (3+i);
```

```
    }
  // std::cout << "past spe_image_open" << std::endl;
  // std::cin >> pp;
  /* Load program */
  if ((rc = spe_program_load (datas[i].speid,ph)) != 0){
      fprintf (stderr, "Failed spe_program_load(errno=%d strerror=%s)\n"
          , errno, strerror(errno));
      exit (3+i);
    }
  //   std::cout << "past spe_program_load" << std::endl;
  // std::cin >> pp;
  spe_image_close(ph);

  //    //printf("About to Initialize Data\n");
  /* Initialize data */
  datas[i].argp = reinterpret_cast<CONTROL_BLOCK_T*>(cb);
  /* Create thread */
  if ((rc = pthread_create (&datas[i].pthread, NULL, px, &datas[i]))
      != 0)
    {
      fprintf (stderr, "Failed pthread_create(errno=%d strerror=%s)\n",
          errno, strerror(errno));
      exit (3+i);
    }else{
    workingSPEs++;
    cb[i].working = 1;     // SPE i is working
  }
  std::cout << "past pthread_create" << std::endl;

  //    std::cin >> pp;
  //spe_context_destroy(datas[i].speid);
  // pthread_cancel(datas[i].pthread);

   //close((int) datas[i].pthread);
  // std::cin >> pp;

  }
}

void matrixData::setupCB(void){
  TotalTime = 0;
  AveTime = 100;
  stdTime = 100;
  mean = 0;
  std = 0;
  int SPE_index = 0;
  //printf("Entering setupCB\n");
```

```
for (int spu = 0; spu < workingSPEs; spu++) {
  //printf("SPE_index = %d and spu = %d\n",SPE_index, spu);

  if(cb[spu].working == 1){
    // RS: First launch all the threads without the control block,
    // RS: but pass in the logical rank of the thread as the argp
    // RS: Then fill-in the control-block parameters
    cb[SPE_index].rank              = SPE_index;

    if(!cb[SPE_index].original_rank_modified){
      cb[SPE_index].original_rank     = SPE_index;
      cb[SPE_index].original_rank_modified = 1;
      //printf(" \n\n\n\n\n\n modifying original rank of \n\n\n");
      //printf("The original rank of %d is %d", cb[SPE_index].rank,cb[
          SPE_index].original_rank);
    }
    cb[SPE_index].old_rank = spu;
    cb[SPE_index].local_store_addr = reinterpret_cast<uint32_t>(
        spe_ls_area_get(datas[spu].speid));
    //printf("Obtained a new SPU id: 0x%x, with local_store_addr = 0x%x\n
        ",
    //      datas[spu].speid, cb[spu].local_store_addr);
    cb[SPE_index].svLength     = svLength;
    cb[SPE_index].matrixData_addr   = reinterpret_cast<uint32_t>(
        JacobianStorage);
    //printf("The address of JacobianStorage is 0x%x\n",JacobianStorage);

    //printf("The address of Row Storage is 0x%x\n",myiV());
    cb[SPE_index].hbWidth           = hbWidth;
    cb[SPE_index].bWidth            = bWidth;
    cb[SPE_index].alignedRowSize    = alignedRowSize;
    cb[SPE_index].jhpc              = jhpc;
    cb[SPE_index].numVars           = myVars;

    cb[SPE_index].control_addr      = reinterpret_cast<uint32_t>(
        spe_ps_area_get(datas[spu].speid,
                                                                  SPE_(
                                                                    )
                                                                    )
                                                                    ;

    //printf("PPE says that SPU %d has a mailbox at 0x%x\n",spu,cb[spu].
        control_addr+SPU_OUT_MBOX);
    cb[SPE_index].signal_1_addr     = reinterpret_cast<uint32_t>(
        spe_ps_area_get(datas[spu].speid,
                                                                  SPE_
                                                                    )
```

```
                                                                    )
                                                                    ;

        cb[SPE_index].signal_2_addr     = reinterpret_cast<uint32_t>(
            spe_ps_area_get(datas[spu].speid,
                                                                SPE_
                                                                    )
                                                                    )
                                                                    ;

        cb[SPE_index].spu_data_addr = reinterpret_cast<uint32_t>(spu_buffers)
            ;
        cb[SPE_index].command_addr  = reinterpret_cast<uint32_t>(
            spe_ps_area_get(datas[spu].speid,
                                                            SPE_MFC_
                                                                    )
                                                                    )
                                                                    ;

        cb[SPE_index].base_row_addr = reinterpret_cast<uint32_t>(&cb[spu]);
        cb[SPE_index].working = (uint32_t)1;
        cb[SPE_index].workingSPEs = workingSPEs;
        datas[SPE_index].speid = datas[spu].speid;
        datas[SPE_index].pthread = datas[spu].pthread;
        SPE_index++;
        //      SPU_RunCtl = 0x1;
      }else{
        pthread_cancel(datas[spu].pthread);
        pthread_join(datas[spu].pthread, NULL);
        //spe_context_destroy(datas[0].speid);
        //printf("SPE %d has just had its context destroyed %d\n",spu,cb[spu
            ].original_rank);
        int x = spe_context_destroy(datas[spu].speid);
      }
    }
    while(SPE_index < workingSPEs){
      cb[SPE_index].working = 0;
      SPE_index++;
    }
    return;
};

void matrixData::barrier(void){
    memset(xSPE,0,NUM_SPE*sizeof(int));
    numFailed = workingSPEs;
    // Wait for all the SPUs to initialize their data
```

```
  tic_time = gettime();
  toc_time = tic_time;
  elapsedTime = toc_time-tic_time;

  while(numFailed != 0 && elapsedTime < AveTime + 5*stdTime){
    toc_time = gettime();
    elapsedTime = toc_time-tic_time;

    for(int i = 0; i < workingSPEs; i++){
        if(*((unsigned int*)(cb[i].control_addr+SPU_MBOX_STAT)) & (unsigned
            int)0xFF){
          r = *((unsigned int *)(cb[i].control_addr + SPU_OUT_MBOX));
          xSPE[r] = 1;
          numFailed--;
      }
    }
  }

  if(numFailed != 0){
    for(int i = 0; i < workingSPEs; i++){
      cb[i].working = xSPE[i];
    }

    present_base_row -= 1;

    setupCB();

    workingSPEs -= numFailed;

    for(int i = 0; i < NUM_SPE; i++){
      cb[i].workingSPEs -= numFailed;
    }
  }
  return;
}

void matrixData::sendData(int dat){
  // uint32_t nspu = NUM_SPU;
  for(unsigned int spu = 0; spu < workingSPEs; spu ++){
      *((unsigned int *)(cb[spu].control_addr + SPU_IN_MBOX)) = dat;
  }
  return;
}


void matrixData::sendSPE_RANKS(void){
for(unsigned int spu = 0; spu < workingSPEs; spu ++)
```

```
      *((unsigned int *)(cb[spu].control_addr + SPU_IN_MBOX)) = spu;
  return;
}

void matrixData::computeIV(void){
  //  memcpy((void*) JacobianStorageBackup, (void*) JacobianStorage,
     JacobByteSize);
  int pvt;
  int jmod;
  int Joffset;
  int testme = 0;
   int nolll = 0;
  spu_buffers;
  tic = gettime();
  et_total = 0;
  et_total_rowx = 0;
  et_total_bs = 0;
  barrier2_time_total = 0;

  first_call = false;
  //sum = 0;
  present_base_row = 0;
  //printf("Going to pack the RHS\n");

  //printf("Going to unstall the %d SPUs\n",workingSPEs);

  sendSPE_RANKS();
  //  sendData(500);
  et_start_rowx = gettime();
  // writeJtoDisk(JacobianStorage,JacobSize);
  while(present_base_row < svLength - 1) {
    //    while(present_base_row < 5) {
    et_start = gettime();

    barrier();

    et_end = gettime();
    et_total += et_end-et_start;

    present_base_row++;

    max_pivot = present_base_row;

    barrier2_time_start = gettime();
   // Have the PPE restart the SPEs
    /*   if(collectedDataCounter == 7 && present_base_row == 20 && nolll
       == 0){
```

```
    //printf("\n\n\n\n\n starting some SPEs\n\n\n\n");
    nolll = 1;
    startNSPEs(1);
    numFailed = -1;
    }*/

  for(unsigned int spu = 0; spu < workingSPEs; spu ++){
    *((unsigned int *)(cb[spu].control_addr + SPU_IN_MBOX)) = numFailed;
  }
  if(numFailed !=0){
    barrier();
    sendSPE_RANKS();
  }
  //  numFailed = 0;
  barrier2_time_end = gettime();
  barrier2_time_total += barrier2_time_end-barrier2_time_start;

  // Have the PPE stop some SPEs
  /*
  if(collectedDataCounter == 5 && present_base_row == 40 && nolll == 0){
     *((unsigned int*)(cb[2].control_addr + 28*sizeof(char))) = 0;
     *((unsigned int*)(cb[3].control_addr + 28*sizeof(char))) = 0;
     *((unsigned int*)(cb[4].control_addr + 28*sizeof(char))) = 0;
     *((unsigned int*)(cb[5].control_addr + 28*sizeof(char))) = 0;

      nolll = 1;
      pthread_cancel(datas[2].pthread);
      pthread_cancel(datas[3].pthread);
      pthread_cancel(datas[4].pthread);
      pthread_cancel(datas[5].pthread);
  }
  */
}

et_end_rowx = gettime();
et_total_rowx += et_end_rowx-et_start_rowx;
// }
//     writeJtoDisk(JacobianStorage, JacobSize);
// BACK SUBSTITUTION BEGIN
et_start_bs = gettime();

for(int k = 0; k < svLength; k++){
  iV2[k]= iV[k][0];
  }
 for (int j = svLength - 1; j >= 0; j--) {
  pvt = computePivot(j);
  jmod = j%myVars;
```

```
    Joffset = j*bWidth + pvt;
    for (int k = 1; k < hbWidth - jmod && k+j < svLength; k++) {
      iV2[j] -= JacobianStorage[Joffset + k] *iV2[j + k];
    }
    iV2[j] = iV2[j]/JacobianStorage[Joffset];
    }

  for(int k = 0; k < svLength; k++){
    iV[k][0] = iV2[k];
    }
  // END

  //   writeJtoDisk(JacobianStorage, JacobSize);
  et_end_bs = gettime();
  et_total_bs = et_end_bs - et_start_bs;
  //printf("Finished Newton iteration.\n"); fflush(stdout);
  //sum = 0;
  present_base_row = 0;
  toc = gettime();
  elapsed = toc-tic;
  collectedDataCounter++;
  //printf("Number of Newton Iterations %f\n", collectedDataCounter);
  TotalTime += elapsed;
  AveTime = TotalTime/collectedDataCounter;
  std += (elapsed - AveTime)*(elapsed - AveTime);
  stdTime = std/collectedDataCounter;
  //printf("Elased Time = %f\n",elapsed);
  //printf("LU Time          =%f\n", et_total_rowx);

  //printf("      Total timeSPUs = %f\n",et_total);
  //printf("      Barrier 2 Time = %f\n", barrier2_time_total);
  //printf("Back Solve Time = %f\n", et_total_bs);
  //printf("Average Time is %1.16f\n",AveTime);
  //printf("Standard Deviation of time is %1.16f\n",stdTime);
  collectedData[0] = elapsed;
  collectedData[1] = et_total_rowx;
  collectedData[2] = et_total;
  collectedData[3] = barrier2_time_total;
  collectedData[4] = et_total_bs;
  //if(collectedDataCounter < 100)
  writeDataFiletoDisk();
  // else
}

int matrixData::myPVColumns()
{
  return numPVColumns;
```

```
};

int matrixData::myPVRows()
{
  return numPVRows;
};

int matrixData::myPVZrows()
{
  return numPVZrows;
};

int matrixData::myFVColumns()
{
  return numFVColumns;
};

int matrixData::myFVRows()
{
  return numFVRows;
};

int matrixData::myFVZrows()
{
  return numFVZrows;
};

int matrixData::ReturnTotalFVvolumes(){
  return totalFVvolumes;
};

double** matrixData::myJacobian()
{
  return Jacobian;
};

double* matrixData::myfluxV(){
  return fluxV;
};

double* matrixData::mydivV()
{
  return divV;
};

double* matrixData::mydivVtwo()
{
```

```
  return divVtwo;
};

double* matrixData::myepsV()
{
  return epsV;
};

double* matrixData::mysocV()
{
  return socV;
};

double** matrixData::myiV()
{
  return iV;
};


double* matrixData::mypiV()
{
  return piV;
};

double* matrixData::mybiV()
{
  return biV;
};

double* matrixData::mysV()
{
  return sV;
};

double* matrixData::mypsV()
{
  return psV;
};

double* matrixData::myavePressures_y(){
  return avePressures_y;
};

double* matrixData::myavePressures_x(){
  return avePressures_x;
};
```

```
double* matrixData::myAVEhp_y(){
  return AVEhp_y;
};

double* matrixData::myAVEhp_x(){
  return AVEhp_x;
};

int matrixData::myJacobByteSize()
{
  return JacobByteSize;
};

int matrixData::myJacobSize()
{
  return JacobSize;
};

int matrixData::mysvLength()
{
  return svLength;
};

void matrixData::setpiV()
{
  for(int counter = 0; counter < svLength; counter++)
    {
      biV[counter] = iV[counter][0];
        piV[counter] = fabs(iV[counter][0]);
    }
  return;
};

int matrixData::myJhpc(void)
{
  return jhpc;
};

/*
void matrixData::resetiV()
{
  memset(iV,0,svByteSize);
  return;
}
*/
void matrixData::resetdivV()
{
```

```
  memset(divV,0,totalPVvolumes*sizeof(double));
  return;
}

void matrixData::resetdivVtwo()
{
  memset(divVtwo,0,4*totalPVvolumes*sizeof(double));
  return;
}

void matrixData::resetFluxes()
{
  memset(fluxV,0,f);
  return;
}

double* matrixData::myJacobianStorage(){
  return JacobianStorage;
}

void matrixData::resetJacobian()
{
  memset(JacobianStorage,0,JacobByteSize);
  // for(int index = 0; index < svLength; index++){
    //   rhs[index] = &JacobianStorage[(index+1)*bWidth - 2];
    //iV[index] = &JacobianStorage[(index+1)*bWidth - 2];
    //JacobianStorage[(index+1)*bWidth - 3] = -1;
    //iV2[index] = (unsigned int) &JacobianStorage[(index+1)*bWidth - 2];
  // }
  return;
};

void matrixData::openDataFile(const char* filename){
  snprintf(file_name, 1024, "testData.dat");
  printf("\n\n\n\nthe file name is %s\n\n\n", file_name);
  if((Vfile = fopen(file_name,"wb")) == NULL)
    printf("File could not be opened.\n");
  return;
}

void matrixData::closeDataFile(void){
   fwrite((void*)&collectedDataCounter, sizeof(double), 1, Vfile);
   fwrite((void*)&collectedDataVars, sizeof(double), 1, Vfile);
   printf("Going to close Data file\n");
   //if(Vfile != NULL){
   fflush(Vfile);
   fclose(Vfile);
```

```
   // }
  return;
}

void matrixData::writeDataFiletoDisk(void){
    fwrite((void*)collectedData, sizeof(double), 5, Vfile);
    return;
}

void matrixData::writeVtoDisk(double* vectorName, int vectorLength){
  if((Vfile = fopen("myVector.dat","w")) == NULL)
    printf("File could not be opened.\n");
  else{
    fwrite((void*)vectorName, sizeof(double), vectorLength,Vfile);
  }
    fclose(Vfile);
    return;
}

void matrixData::writeJtoDisk(double* vectorName, int vectorLength){
  static int count;
  static char file_name[1024];
  printf(file_name, 1024, "myMatrixJacob%dspu_%d.dat", NUM_SPE, count);
  if((Vfile = fopen(file_name,"w")) == NULL)
     printf("File could not be opened.\n");
  else{
    fwrite((void*)vectorName, sizeof(double), vectorLength,Vfile);
  }
  // fflush(Vfile);
    // fclose(Vfile);
  count++;
    return;
}

void matrixData::writeiVbeforetoDisk(double* vectorName, int vectorLength){
  if((Vfilebefore = fopen("iVbefore.dat","w")) == NULL)
    printf("File could not be opened.\n");
  else{
    //printf("The number of bytes to write is %d\n",vectorLength*sizeof(
       double));
    fwrite((void*)vectorName, sizeof(double), vectorLength,Vfilebefore);
  }
  fclose(Vfilebefore);
    return;
}

void matrixData::writeiVaftertoDisk(double* vectorName, int vectorLength){
```

```
  if((Vfileafter = fopen("iVafter.dat","w")) == NULL)
    printf("File could not be opened.\n");
  else{
    fwrite((void*)vectorName, sizeof(double), vectorLength, Vfileafter);
  }
  fclose(Vfileafter);
    return;
}


void matrixData::restoresV()
{
  memcpy((void*)sV,(void *) psV, svByteSize);
  return;
};


void matrixData::storesV()
{
  memcpy((void*)psV,(void *) sV, svByteSize);
  return;
};


int matrixData::mySvByteSize()
{
  return svByteSize;
};


double* matrixData::myExtraBuffer1()
{
  return extraBuffer1;
};

int* matrixData::myintBuffer()
{
  return intBuffer;
};

/*****************************************************/
/********** Boundary Conditions for Row Major ********/
/*****************************************************/
void matrixData::applyBoundaryConditions(void){
  int clearRow;

   // EK Rows
```

```
  for(int j = 0; j < numPVRows; j++){
    for(int i = 0; i < ElectrolytePVColumns; i++){
      clearRow = myVars*ldxPVColumns + myVars*i + ekBoundary + j*myVars*
          numPVColumns;
      iV[clearRow][0] = 0;
      memset((void*) &Jacobian[clearRow][0],0,alignedRowSize);
      Jacobian[clearRow][computePivot(clearRow)] = 1;
    }
    }
  // Doc Row

  clearRow = reference;
  iV[clearRow][0] = 0;
  memset((void*) &Jacobian[reference][0],0,alignedRowSize);

  clearColumn(clearRow);
  Jacobian[clearRow][computePivot(clearRow)] = 1;

  return;
};


void matrixData::clearColumn(int elimColumn){
  int pivot;
  int elimPosition;

  for (int pRow = 0; pRow < svLength; pRow++){
    pivot = computePivot(pRow);
    elimPosition = elimColumn + pivot - pRow;
    if(elimPosition >= 0 && elimPosition < bWidth){
      Jacobian[pRow][elimPosition] = 0;
    }
  }
}

int matrixData::computePivot(int pRow){
  if(myVars%2){
    return jhpc/*numPVColumns*myVars*/ + pRow%myVars + (pRow/myVars)%2;
  }else{
    return jhpc/*numPVColumns*myVars*/ + pRow%myVars;
  }
}

static double  vector_inf_norm(double* x, int length)
{
  double max, max_position;
  max = fabs(x[0]);
```

```
  max_position = 0;
  for(int k = 1; k < length; k++){
    if(fabs(x[k]) > max)
      {
        max = fabs(x[k]);
        max_position = k;
      }
  }
  return max;
}

/*****************************************************/
/********** PreConditioner for Row Major ************/
/*****************************************************/
void matrixData::applyPreConditioner(void){
  int index = 0;
  for(int x = 0; x <= svLength/myVars; x++){
    for(int r = 0; r < bWidth; r++){
      index = x*myVars;
      if(index < svLength){
        Jacobian[index][r] = Jacobian[index][r]*30*1000;
      }
      if(index+1 < svLength){
        Jacobian[index+1][r] = Jacobian[index+1][r]/50;
      }
      if(index+2 < svLength){
        Jacobian[index+2][r] = Jacobian[index+2][r]*10;
      }
    }
  }

  return;
};


// void matrixData::applyPreConditioner(void){
//   double myMax;

//   int end = totalPVvolumes*2;
//   for(int index = totalPVvolumes; index < end; index++){
//     for(int r = 0; r < svLength; r++){
//       extraBuffer1[r] = Jacobian[index][r];
//       //      std::cout << extraBuffer1[r] << std::endl;
//     }

//     myMax = vector_inf_norm(extraBuffer1, svLength);
//     //       std::cout << myMax;
```

```
//      myMax = sqrt(fabs(myMax));
//      for(int p = 0; p < svLength; p++){
//        Jacobian[index][p] = Jacobian[index][p]/myMax;
//      }
//      iV[index][0] = iV[index][0]/myMax;
//    }
//    return;
// };
```

Listing B.65. FVbatteryModel.cpp

```
/* Copyright (c) 2007 Massachusetts Institute of Technology
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
    copy of
 * this software and associated documentation files (the "Software"), to
    deal in
 * the Software without restriction, including without limitation the
    rights to
 * use, copy, modify, merge, publish, distribute, sublicense, and/or sell
    copies of
 * the Software, and to permit persons to whom the Software is furnished to
     do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included
    in all
 * copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
    OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY
    , FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
    AUTHORS OR
 * COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
    WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
 * CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
    SOFTWARE.
 */
/**
 * FVbatterModel.cpp - 2 Dimensional Electrochemical Battery Model main
    program
 * @author James Geraci
 * @second author Sudarshan Raghunathan
 */

// FVbatterModel.cpp : Defines the entry point for the console application.

#include "StdAfx.h"
#include <iostream>
#include "FV.h"
#include "PV.h"
#include <time.h>
#include <signal.h>
#include <assert.h>
```

```
FILE* Vfile;
int biVcount;
int tStep;
int battery_model_simluation(const char *,int);

void exit_routine(matrixData*, FV**, PV**,Electrode**, electrodeSizeData**,
   cellSizeData*);

void exit_routine(matrixData* myMatrix, FV** myFv, PV** myPv,
                  Electrode** myElectrodes, electrodeSizeData**
                     electSizeVector,
                  cellSizeData* csData){

  //printf("Stop All the SPUs\n");
  // unsigned int EXIT = NUM_SPE + 1;
  // for(unsigned int spu = 0; spu < NUM_SPE; spu ++)
  // {
  //    spe_in_mbox_write(datas[spu].speid, &EXIT, 1,
     SPE_MBOX_ANY_NONBLOCKING);
      //       pthread_kill(datas[spu].pthread,0);
  //  }

  int numVolumes;
  int i;
  int rc;

  myMatrix->closeDataFile();
  //printf("Closing Data File\n");
  delete myMatrix;


  for(i = 0; i < csData->gettotalFVVolumes(); i++){
    delete myFv[i];
  }

  for(i = 0; i < csData->gettotalPVVolumes(); i++){
    delete myPv[i];
  }

  for(i = 0; i < csData->getnumofElectrodes(); i++){
    delete myElectrodes[i];
  }

  for (i = 0; i < csData->getnumofElectrodes(); i++){
    delete electSizeVector[i];
  }
```

```
   delete csData;
  delete [] myElectrodes;
  delete [] myFv;
  delete [] myPv;

  //  for(int i = 0; i < NUM_SPE; i++){
    //    rc = pthread_join(datas[i].pthread,0);
    // assert(rc == 0);
    //       rc = spe_context_destroy(datas[i].speid);
    // assert(rc == 0);
  // }

  /* Close Program Image */
  //  if ((rc = spe_image_close (ph) ) == -1)
  // {
  //    f//printf (stderr, "Failed spe_image_close(errno=%d strerror=%s)\n
     ", errno, strerror(errno));
  //    exit (3+i);
  // }

  //  free_align(datas);
  //printf("\n \n Clean UP is FINISHED\n \n");

  return;
}



/*
void writeJtoDisk(double* vectorName, int vectorLength){
  static int count;
  static char file_name[1024];
  sn//printf(file_name, 1024, "myJacob%d_%d.dat", version, count);
  if((Vfile = fopen(file_name,"w")) == NULL)
    //printf("File could not be opened.\n");
  else{
    fwrite((void*)vectorName, sizeof(double), vectorLength, Vfile);
  }
  fflush(Vfile);
  fclose(Vfile);
  count++;
  return;
}
*/
void writeJtoDisk(double* vectorName, int vectorLength){
  static int count;
```

```
  static char file_name[1024];
  snprintf(file_name, 1024, "myJacob%d_%d.dat", tStep, biVcount);
  if((Vfile = fopen(file_name,"w")) == NULL)
    printf("File could not be opened.\n");
  else{
    //printf("Now writing Jacobian\n");
    fwrite((void*)shapeDataForMatlab, sizeof(double), 3, Vfile);
    fwrite((void*)vectorName, sizeof(double), vectorLength, Vfile);
  }
  fflush(Vfile);
  fclose(Vfile);
  count++;
  return;
}

void writeJaftertoDisk(double* vectorName, int vectorLength){
  static int count;
  static char file_name[1024];
  snprintf(file_name, 1024, "myJacob%d_%dafter.dat", version, count);
  if((Vfile = fopen(file_name,"w")) == NULL)
    printf("File could not be opened.\n");
  else{
    fwrite((void*)vectorName, sizeof(double), vectorLength, Vfile);
  }
  fflush(Vfile);
  fclose(Vfile);
  count++;
  return;
}

void writeJfaraftertoDisk(double* vectorName, int vectorLength){
  static int count;
  static char file_name[1024];
  snprintf(file_name, 1024, "myJacob%d_%dfarafter.dat", version,count);
  if((Vfile = fopen(file_name,"w")) == NULL)
    printf("File could not be opened.\n");
  else{
    fwrite((void*)vectorName, sizeof(double), vectorLength, Vfile);
  }
  fflush(Vfile);
  fclose(Vfile);
  count++;
  return;
}


void writeiVtoDisk(double* vectorName, int vectorLength){
```

```
  static int count;
  static char file_name[1024];
  snprintf(file_name, 1024, "myiV%d_%d.dat", tStep, count);
  // snprintf(file_name, 1024, "myiV_%d.dat", tStep);
  if((Vfile = fopen(file_name,"w")) == NULL)
    printf("File could not be opened.\n");
  else{
    fwrite((void*)shapeDataForMatlab, sizeof(double), 3, Vfile);
    fwrite((void*)vectorName, sizeof(double), vectorLength, Vfile);
  }
  fflush(Vfile);
  fclose(Vfile);
  count++;
  return;
}

void writeEPStoDisk(double* vectorName, int vectorLength){
  static int count;
  static char file_name[1024];
  snprintf(file_name, 1024, "myEPS%d_%d.dat", tStep, count);
  if((Vfile = fopen(file_name,"w")) == NULL)
    printf("File could not be opened.\n");
  else{
    fwrite((void*)FVshapeDataForMatlab, sizeof(double), 3, Vfile);
    fwrite((void*)vectorName, sizeof(double), vectorLength, Vfile);
  }
  fflush(Vfile);
  fclose(Vfile);
  count++;
  return;
}


void writebiVtoDisk(double* vectorName, int vectorLength){
  // static int count;
  static char file_name[1024];
  snprintf(file_name, 1024, "mybiV%d_%d.dat", tStep, biVcount);
  // sn//printf(file_name, 1024, "myiV_%d.dat", tStep);
  if((Vfile = fopen(file_name,"w")) == NULL)
    printf("File could not be opened.\n");
  else{
    fwrite((void*)shapeDataForMatlab, sizeof(double), 3, Vfile);
    fwrite((void*)vectorName, sizeof(double), vectorLength, Vfile);
  }
  fflush(Vfile);
  fclose(Vfile);
  biVcount++;
```

```
  return;
}

void writepiVtoDisk(double* vectorName, int vectorLength){
  static int count;
  static char file_name[1024];
  snprintf(file_name, 1024, "mypiV%d_%d.dat", tStep, count);
  // sn//printf(file_name, 1024, "myiV_%d.dat", tStep);
  if((Vfile = fopen(file_name,"w")) == NULL)
    printf("File could not be opened.\n");
  else{
    fwrite((void*)shapeDataForMatlab, sizeof(double), 3, Vfile);
    fwrite((void*)vectorName, sizeof(double), vectorLength, Vfile);
  }
  fflush(Vfile);
  fclose(Vfile);
  count++;
  return;
}
void writeSVtoDisk(double* vectorName, int vectorLength){
  // static int count;
  static char file_name[1024];
  snprintf(file_name, 1024, "mySV%d_%d.dat", tStep, biVcount-1);
  if((Vfile = fopen(file_name,"w")) == NULL)
    printf("File could not be opened.\n");
  else{
    fwrite((void*)shapeDataForMatlab, sizeof(double), 3, Vfile);
    //fflush(Vfile);
    fwrite((void*)vectorName, sizeof(double), vectorLength, Vfile);
  }
  fflush(Vfile);
  fclose(Vfile);
  //  count++;
  return;
}
/*
void writeSVtoDisk(double* vectorName, int vectorLength){
  static int count;
  static char file_name[1024];
  sn//printf(file_name, 1024, "mySV_%d.dat", count);
  if((Vfile = fopen(file_name,"w")) == NULL)
    //printf("File could not be opened.\n");
  else{
    fwrite((void*)vectorName, sizeof(double), vectorLength, Vfile);
  }
  fflush(Vfile);
  fclose(Vfile);
```

```
  count++;
  return;
}
*/
static inline double  IV_vector_inf_norm(double** x, int length)
{
  double max, max_position;
  max = fabs(x[0][0]);
  max_position = 0;
  for(int k = 1; k < length; k++){
    if(fabs(x[k][0]) > max)
      {
        max = fabs(x[k][0]);
        max_position = k;
      }
  }
  return max;
}

static inline double vector_inf_norm(double* x, int length)
{
  double max, max_position;
  max = fabs(x[0]);
  max_position = 0;
  for(int k = 1; k < length; k++){
    if(fabs(x[k]) > max)
      {
        max = fabs(x[k]);
        max_position = k;
      }
  }
  return max;
}

/**
 *  Replaces x with x - y
 */
static inline void IV_inplace_subtract(double* x, double** y, int length)
{
  for(int i = 0;i < length; x[i] -= y[i][0], i ++);
}

int battery_model_simulation(const char *program_to_load, int NSPE){
  matrixData * myMatrix;
  Electrode ** myElectrodes;
  cellSizeData * csData;
  FV** myFv;
```

```
PV** myPv;
electrodeSizeData** electSizeVector;

//mysig_buf = reinterpret_cast<sig_buf_t*>(malloc_align(16*NUM_SPU*sizeof
    (sig_buf_t),7));
mode = 1; // Want a skyline Jacobian
double DeltaTester = 0;
double ResidualTester = 0;
int pass = 0;
tStep = 0;
double DeltaLimit = 7.5e-8;// for gmres 1e-8 for \\ 1e-15
double ResidualLimit = 7.5e-8;// for gmres 1e-8 for \\ 1e-15
double totalTime = 0;

csData = new cellSizeData();
csData->setnumofElectrodes(3);

csData->setIdxFVColumns(NIdxColumns);
csData->setElectrolyteFVColumns(NEColumns);
csData->setIdFVColumns(NIdColumns);
csData->setnumFVRows(NRows);
csData->setnumFVZrows(1);

csData->setIdxPVColumns(csData->getIdxFVColumns()+1);
csData->setElectrolytePVColumns(csData->getElectrolyteFVColumns()-1);
csData->setIdPVColumns(csData->getIdFVColumns()+1);

double IdxHeight = 10.0;
double ElectrolyteHeight = 10.0;
double IdHeight = 10.0;


double IdxDepth = 7.0;
double ElectrolyteDepth = 7.0;
double IdDepth = 7.0;


double IdxWidth = 0.03;
double ElectrolyteWidth = 0.03;
double IdWidth = 0.03;

csData->computeTotalFVColumns();
csData->computeTotalPVColumns();

csData->setnumPVRows(csData->getnumFVRows()+1);
csData->setnumPVZrows(1);
```

```
  csData->computeTotalFVVolumes();
  csData->computeTotalPVVolumes();

  try{
    myMatrix = new matrixData(csData,NSPE);
  }
  catch(std::bad_alloc){
    //std::cout << "Couldn't Allocate myMatrix!" << std::endl;
    exit(-1);
  }

/* Information used by Matlab to format output */
  FVshapeDataForMatlab[0] = (double) myVars;
  FVshapeDataForMatlab[1] = (double) csData->getnumFVColumns();
  printf("The number of FV columns is %f\n",FVshapeDataForMatlab[1]);
  FVshapeDataForMatlab[2] = (double) csData->getnumFVRows();
  printf("The number of FV rows is %f\n",FVshapeDataForMatlab[2]);
  shapeDataForMatlab[0] = (double) myVars;
  shapeDataForMatlab[1] = (double) csData->getnumPVColumns();
  shapeDataForMatlab[2] = (double) csData->getnumPVRows();
  printf("The number of PV rows is %f\n",shapeDataForMatlab[2]);

  myMatrix->openDataFile(program_to_load);

  try{
    myFv = (FV**) new char [(csData->gettotalFVVolumes())*sizeof(FV*)];
  }
  catch(std::bad_alloc){
    //std::cout << "Couldn't Allocate myFv array!" << std::endl;
    exit(-1);
  }

  for(int counter = 0; counter < csData->gettotalFVVolumes(); counter++){
    myFv[counter] = NULL;
  };



  try{
    myPv = (PV**) new char [(csData->gettotalPVVolumes())*sizeof(PV*)];
  }
  catch(std::bad_alloc){
    //std::cout << "Couldn't Allocate myFv array!" << std::endl;
    exit(-1);
  }

  for(int counter = 0; counter < csData->gettotalPVVolumes(); counter++)
```

```
  {
    myPv[counter] = NULL;
  };



try{
  electSizeVector = (electrodeSizeData**) new char [csData->
      getnumofElectrodes()*sizeof(electrodeSizeData*)];
}
catch(std::bad_alloc){
  //std::cout << "Couldn't Allocate electSizeVector!" << std::endl;
  exit(-1);
}

try{
  electSizeVector[0] = new electrodeSizeData(ldxWidth,ldxHeight,ldxDepth,
                                             csData->getldxFVColumns(),
                                                csData->getnumFVRows(),
                                                csData->getnumFVZrows(),
                                             csData->getldxPVColumns(),
                                                csData->getnumPVRows(),
                                                csData->getnumPVZrows(),
                                             0,0,0,0,csData);
}
catch(std::bad_alloc){
  //std::cout << "Couldn't Allocate electSizeVector[0]!" << std::endl;
  exit(-1);
}
try{
  electSizeVector[1] = new electrodeSizeData(ElectrolyteWidth,
      ElectrolyteHeight,ElectrolyteDepth,
                                             csData->
                                                getElectrolyteFVColumns()
                                                ,csData->getnumFVRows(),
                                                csData->getnumFVZrows(),
                                             csData->
                                                getElectrolytePVColumns()
                                                ,csData->getnumPVRows(),
                                                csData->getnumPVZrows(),
                                             0,csData->getldxFVColumns()
                                                ,0,csData->
                                                getldxPVColumns(),csData)
                                                ;
}
catch(std::bad_alloc){
  //std::cout << "Couldn't Allocate electSizeVector[1]!" << std::endl;
```

```
    exit(-1);
  }
  try{
    electSizeVector[2] = new electrodeSizeData(ldWidth,ldHeight,ldDepth,
                                               csData->getldFVColumns(),
                                                  csData->getnumFVRows(),
                                                  csData->getnumFVZrows(),
                                               csData->getldPVColumns(),
                                                  csData->getnumPVRows(),
                                                  csData->getnumPVZrows(),
                                               0,csData->getldxFVColumns()+
                                                  csData->
                                                  getElectrolyteFVColumns()
                                                  ,0,
                                               csData->getldxPVColumns()+
                                                  csData->
                                                  getElectrolytePVColumns()
                                                  ,csData);
  }
  catch(std::bad_alloc){
    //std::cout << "Couldn't Allocate electSizeVector[2]!" << std::endl;
    exit(-1);
  }


  try{
    myElectrodes = (Electrode**) new char [csData->getnumofElectrodes()*
       sizeof(Electrode*)];
  }
  catch(std::bad_alloc){
    //std::cout << "Couldn't Allocate myElectrodes!" << std::endl;
    exit(-1);
  }

  try{
    myElectrodes[0] = new ldxElectrode(myFv, myPv, myMatrix,electSizeVector
       [0]);
  }
  catch(std::bad_alloc){
    //std::cout << "Couldn't Allocate myElectrodes[0] ldxElectrode!" << std
       ::endl;
    exit(-1);
  }

  for(int FvCounter = 0; FvCounter < csData->gettotalFVVolumes(); FvCounter
     ++)
    {
```

```
    myFv[FvCounter] = myFv[FvCounter];
  }

try{
  myElectrodes[1] = new Electrolyte(myFv, myPv, myMatrix,electSizeVector
      [1]);
}
catch(std::bad_alloc){
  //std::cout << "Couldn't Allocate myElectrodes[1] Electrolyte!" << std
      ::endl;
  exit(-1);
}

for(int FvCounter = 0; FvCounter < csData->gettotalFVVolumes(); FvCounter
    ++)
  {
    myFv[FvCounter] = myFv[FvCounter];
  }

try{
  myElectrodes[2] = new ldElectrode(myFv, myPv, myMatrix,electSizeVector
      [2]);
}
catch(std::bad_alloc){
  //std::cout << "Couldn't Allocate myElectrodes[2] ldElectrode!" << std
      ::endl;
  exit(-1);
}

for(int FvCounter = 0; FvCounter < csData->gettotalFVVolumes(); FvCounter
    ++)
  {
    myFv[FvCounter] = myFv[FvCounter];
  }

for(int FvCounter = 0; FvCounter < csData->gettotalFVVolumes(); FvCounter
    ++)
  {
    myFv[FvCounter]->setupPvPointers();
  }

for(int PvCounter = 0; PvCounter < csData->gettotalPVVolumes(); PvCounter
    ++){
  myPv[PvCounter]->setupFvPointers(myFv);
  myPv[PvCounter]->initializeSize();
  myPv[PvCounter]->initializembScale();
}
```

```
for(int PvCounter = 0; PvCounter < csData->gettotalPVVolumes(); PvCounter
    ++){
  myPv[PvCounter]->computeDensity();
  myPv[PvCounter]->initializePressure();
}

 myMatrix->storesV();


 //  //printf("Haven't crashed yet\n");
 // return 0;
pass++;

//  startSPUs(myMatrix, program_to_load);
myMatrix->startSPEs(program_to_load);
        writeSVtoDisk(myMatrix->mysV(),myMatrix->mysvLength());
myMatrix->resetdivVtwo();
while(tStep < tSteps)
  {
    //std::cout << "Time Step is " << tStep << "dt is " << dt << "I is
        " << I <<std::endl;
    //       //std::cout << "Matrix Data svLength " << myMatrix->svLength
        << " with rows " << myMatrix->myPVRows() << std::endl;
    for(int FvolumeCounter = 0; FvolumeCounter < csData->
        gettotalFVVolumes(); FvolumeCounter++)
      {
        myFv[FvolumeCounter]->computeDiv();
      }
    for(int FvolumeCounter = 0; FvolumeCounter < csData->
        gettotalFVVolumes(); FvolumeCounter++)
      {
        if(pass==1){
          myFv[FvolumeCounter]->computeNextEPS();
          myFv[FvolumeCounter]->computeNextSOC();
        }
      }
    //       if(pass == 1){
    //         writeEPStoDisk(myMatrix->myepsV(),myMatrix->
        ReturnTotalFVvolumes());
    // }
    myMatrix->resetdivVtwo();
    // Compute the Material Properties
    for(int FvolumeCounter = 0; FvolumeCounter < csData->
        gettotalFVVolumes(); FvolumeCounter++)
      {
        myFv[FvolumeCounter]->computeMaterialProperties();
```

```
  }

// Compute the Fluxes
for(int FvolumeCounter = 0; FvolumeCounter < csData->
    gettotalFVVolumes(); FvolumeCounter++)
  {
    myFv[FvolumeCounter]->computeQvalues();
    myFv[FvolumeCounter]->computeFluxes();
  }

myMatrix->resetdivV();
for(int PvCounter = 0; PvCounter < csData->gettotalPVVolumes();
    PvCounter++)
  {
    myPv[PvCounter]->computeDiv();
    myPv[PvCounter]->computeDivTwo();
    myPv[PvCounter]->computeDensity();
  }

for(int PvCounter = 0; PvCounter < csData->gettotalPVVolumes();
    PvCounter++)
  {
    myPv[PvCounter]->computeAVEpressures();
    myPv[PvCounter]->computeAVEhps();
  }

// Compute Intermediate value from fluxes
for(int PvCounter = 0; PvCounter < csData->gettotalPVVolumes();
    PvCounter++)
  {
    myPv[PvCounter]->computeDerivatives();
  }

for(int PvCounter = 0; PvCounter < csData->gettotalPVVolumes();
    PvCounter++)
  {
    myPv[PvCounter]->computeIValues();
    myPv[PvCounter]->setupJacobian();

  }
ResidualTester = IV_vector_inf_norm(myMatrix->myiV(),myMatrix->
    mysvLength());

//          writeJtoDisk(myMatrix->myJacobianStorage(),myMatrix->
    myJacobSize());

////std::cout << "iV norm = " << IV_vector_inf_norm(myMatrix->myiV(),
```

```
    myMatrix->mysvLength()) << std::endl;
//  std::cin >> testerA;
//          if(pass == 2){
//cout << "The Jacobian Byte Size is " <<  myMatrix->myJacobSize()
    << endl;

// cout << "Waiting after write, press enter to continue" << endl;
//cin >> myBaby;
//  if(pass == 3){
//          exit_routine(myMatrix,myFv,myPv,myElectrodes,
    electSizeVector,csData);
//          return 0;
// }
//printf("The tStep = %d  Pass = %d and DeltaTester = %.16f and
    ResidualTester = %.16f\n",tStep, pass, DeltaTester,ResidualTester)
    ;

// writeiVtoDisk(myMatrix->myiV(),myMatrix->mySvByteSize());
                                //        }
//myMatrix->copyiV2rhs();
//writeiVtoDisk(myMatrix->myiV(),myMatrix->mySvByteSize());
                                //        }

            myMatrix->applyBoundaryConditions();
            myMatrix->applyPreConditioner();
            //            writeJtoDisk(myMatrix->myJacobianStorage(),
               myMatrix->myJacobSize());
myMatrix->restoreiVindex();



//exit_routine(myMatrix,myFv,myPv,myElectrodes,electSizeVector,csData
    );
// return 0;
////printf("computing iv\n"); fflush(stdout);
myMatrix->computeIV();
// writeJaftertoDisk(myMatrix->myJacobianStorage(),myMatrix->
    myJacobSize());
// exit_routine(myMatrix,myFv,myPv,myElectrodes,electSizeVector,
    csData);
// return 0;
//       double* testData = reinterpret_cast<double*>(malloc_align(
    myMatrix->bWidth*sizeof(double)*myMatrix->svLength,7));
// testData[0] = (double) tStep;
// testData[1] = 6.3;
//     myMatrix->spu_buffers[0] =10;
// //printf("Spu_data_addr = 0x%x\n",myMatrix->spu_buffers[0]);
```

```
//        spe_mfcio_get(datas[0].speid,(unsigned int)testData,(void*)
    cb[0].spu_data_addr,2*sizeof(double),0,0,0);
//if(pass > 1){
// spe_mfcio_get(datas[0].speid,(unsigned int)myMatrix->spu_buffers
    [0],(void*)testData,myMatrix->alignedRowSize,0,0,0);
// spe_mfcio_get(myMatrix->speids[0],(unsigned int)myMatrix->
    spu_buffers[0],
// (void*)myMatrix->JacobianStorage,myMatrix->alignedRowSize,0,0,0);
//spe_mfcio_tag_status_read(myMatrix->speids[0],0,SPE_TAG_ALL,NULL);
//        }
//  writeJtoDisk(myMatrix->myJacobianStorage(),myMatrix->myJacobSize
    ());

//        myMatrix->copyrhs2iV();
////printf("Now writing Jacobian\n");
//        myMatrix->writeCollectedDatatoDisk();

//        myMatrix->copyrhs2iV();
//        writeJtoDisk(myMatrix->myJacobianStorage(),myMatrix->
    myJacobSize());

//        writeSVtoDisk(myMatrix->mysV(),myMatrix->mysvLength());
//        exit_routine(myMatrix,myFv,myPv,myElectrodes,
    electSizeVector,csData);
//        return 0;
//        writeSVtoDisk(myMatrix->mysV(),myMatrix->mysvLength());

//      //std::cout << "Test IV norm = " <<  vector_inf_norm(myMatrix
    ->myiV(),myMatrix->mysvLength()) << std::endl;
IV_inplace_subtract(myMatrix->mysV(), myMatrix->myiV(), myMatrix->
    mysvLength());

myMatrix->setpiV();
if((tStep%1) == 0){
    writebiVtoDisk(myMatrix->mybiV(),myMatrix->mysvLength());
}
DeltaTester = vector_inf_norm(myMatrix->mypiV(),myMatrix->mysvLength
    ());
//tester = 0;
//std::cout << "DeltaTester Value = " << DeltaTester << "
    ResidaulTester Value = " << ResidualTester << std::endl;
      ////printf("Tester Value = %f\n",tester);
//      //std::cout << "sV norm = " << vector_inf_norm(myMatrix->mysV
    (),myMatrix->mysvLength()) << std::endl;
//exit_routine(myMatrix,myFv,myPv,myElectrodes,electSizeVector,csData
    );
//return 0;
```

```
        //                return 0;
      // writeJfaraftertoDisk(myMatrix->myJacobianStorage(),myMatrix->
         myJacobSize());
      if(DeltaTester <= DeltaLimit && ResidualTester <= ResidualLimit)
        {

          myMatrix->storesV();
          if((tStep%1) == 0){
          writeSVtoDisk(myMatrix->mysV(),myMatrix->mysvLength());
          }
          //       totalTime = totalTime+
          tStep++;
          if(tStep >= (int) 5){
            I = 0;
          }
          pass = 0;
        }

      myMatrix->resetJacobian();
      myMatrix->resetFluxes();
      pass++;
    }
  if(tStep == tSteps){
    ////printf("Running Exit Routine\n");
    exit_routine(myMatrix,myFv,myPv,myElectrodes,electSizeVector,csData);
    return 0;
  }
  return 0;
};



int main(int argc, char* argv[])
{
  biVcount = 0;
  NIdxColumns = atoi(argv[2]);
  NEColumns = atoi(argv[3]);
  NIdColumns = atoi(argv[4]);
  NRows = atoi(argv[5]);
  int NUM_SPE = atoi(argv[10]);
  // I = atoi(argv[8]);
  I = 0.25;
  tSteps = atoi(argv[7]);
  //dt = atoi(argv[6]);
  dt = 100.0;
  version = atoi(argv[9]);
  battery_model_simulation(argv[1],NUM_SPE);
```

```
    return 0;
}
```

Listing B.66. common.h

```
/**
 * common.h
 * SPU parameters
 * @author James Geraci
 *
 */
// -*- mode: c++ -*-
#ifndef _COMMON_H
#define _COMMON_H

#include <stdint.h>
#include <math.h>
#include <simdmath.h>

typedef uint32_t uintptr32_t;
#define DMALIST_ALIGNED __attribute__((aligned(8)));
#define QWORD_ALIGNED __attribute__((aligned(16)));
#define CACHE_ALIGNED __attribute__((aligned(128)));




#define QUAD_ALIGNMENT 16

#define MAX_BYTES_ONE_DMA 16384


#define doubleByteSize sizeof(double)

// Branch hint macros
#define LIKELY(exp)    __builtin_expect(exp, true)
#define UNLIKELY(exp) __builtin_expect(exp, false)

#define SIGNAL_OFFSET 12
#define SPU_OUT_MBOX  4
#define SPU_IN_MBOX   12
#define SPU_MBOX_STAT 20
#define MFC_CMDSTATUS 20
#define MFC_QSTATUS   260
#define PRXY_QUERYTYPE 516
#define PRXY_QUERYMASK 540
#define PRXY_TAGSTATUS 556

typedef struct {
  uint32_t brLS_addr[4];
  uint32_t rank;
```

```
  uint32_t local_store_addr;
  uint32_t control_addr;
  uint32_t signal_1_addr;

  uint32_t signal_2_addr;
  uint32_t matrixData_addr;
  uint32_t command_addr;
  uint32_t svLength;
  uint32_t rhs_addr;
  uint32_t sol_addr;
  uint32_t hbWidth;
  uint32_t alignedRowSize;

  uint32_t jhpc;
  uint32_t numVars;
  uint32_t bWidth;
  uint32_t sig_buf;
  uint32_t spu_data_addr;
  uint32_t base_row_addr;
  uint32_t numStoredRows;
  uint32_t working;
  uint32_t workingSPEs;
  uint32_t old_rank;
  uint32_t original_rank;
  uint32_t original_rank_modified;
  uint32_t padding[4];
} CONTROL_BLOCK_T;

typedef struct dma_list_elem{
  union{
    unsigned int all32;
    struct{
      unsigned int stall    : 1;
      unsigned int reserved : 15;
      unsigned int nbytes   : 16;
    }bits;
  }size;
  unsigned int ea_low;
} dma_list_elem_t;
#endif
```

Listing B.67. Makefile

```
#############################################################################
#                              Target
#############################################################################

PROGRAM_ppu       = FVbatteryModel
OBJS= StdAfx.o matrixData.o Electrode.o CV.o electrodeSizeData.o OLLLPV.o
   ORLRPV.o volumeChemData.o UPV.o FVbatteryModel.o
    electrolyteElectrodeChemData.o ORPV.o PVIConditions.o OLULPV.o BPV.o
    Electrolyte.o ldxElectrode.o ldElectrode.o ldElectrodeChemData.o
    volumeSpatialData.o myGlobals.o ORURPV.o electrodeChemData.o PV.o FV.o
   OLPV.o cellSizeData.o volumeIConditions.o ldxElectrodeChemData.o ULCV.o
   URCV.o LLCV.o LRCV.o
#############################################################################
#                              Local Defines
#############################################################################
IMPORTS           = -lspe2 -lmisc -lsimdmath -lpthread


#############################################################################
#                              make.footer
#############################################################################
INCLUDE += -I. -I.. -I/opt/cell/sysroot/opt/cell/sdk/usr/include
LIBS = /opt/cell/sysroot/opt/cell/sdk/usr/lib
CXX=/opt/ibmcmp/xlc/cbe/9.0/bin/ppuxlc++
CXXFLAGS= -qcpluscmt -M -ma $(INCLUDE) -qaltivec -qenablevmx
#OPTFLAGS=-03

RM=rm
RMFLAGS=-f



.cc.s:
        $(CXX) $(CXXFLAGS) $(OPTFLAGS) -S $< -g -o $@

.cc.o:
        $(CXX) $(CXXFLAGS) $(OPTFLAGS) -c $< -g -o $@

.s.o:
        $(ASM) $(INCLUDE) -o $@ $<

$(PROGRAM_ppu) : $(OBJS)
        $(CXX) $(CXXFLAGS) -g -o $@ $^ -L$(LIBS) -Wl,-m,elf32ppc -R$(LIBS)
           $(IMPORTS)

clean:
```

```
$(RM) $(RMFLAGS) *.o
$(RM) $(RMFLAGS) *.d
```

Listing B.68. Run file

```
../FVbatteryModel spu_prog 7 7 7 30 1 50 1 1 6
```

Appendix C

2-D Model Solver Code

Listing C.1. out of core solver rev2

```
/* Copyright (c) 2007 Massachusetts Institute of Technology
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
    copy of
 * this software and associated documentation files (the "Software"), to
    deal in
 * the Software without restriction, including without limitation the
    rights to
 * use, copy, modify, merge, publish, distribute, sublicense, and/or sell
    copies of
 * the Software, and to permit persons to whom the Software is furnished to
    do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included
    in all
 * copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
    OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY
    , FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
    AUTHORS OR
 * COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
    WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
 * CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
    SOFTWARE.
 */
/* @author James Geraci
 */

//  James Geraci
// 6.189 MultiCoreProgramming
// January 23, 2007

#include <spu_mfcio.h>
#include <libmisc.h>
#include <stdio.h>
#include <spu_intrinsics.h>
#include <vec_literal.h>
#include <string.h>
//#include <simdmath/divd2.h>
//#include <sys/time.h>
#include "common.h"
```

```
CONTROL_BLOCK_T* cb CACHE_ALIGNED;

int my_rank;
int NUM_SPUS;
//uint32_t* row_pointers;
int two_double_byteSize;
double base_row_factor;
double adjustFactor;
double inv_base_row_factor;
vector double vFactor;

int present_base_row;
int present_elim_row;
int future_elim_row;
int present_elim_column;
int nbr;

int JacobRowSize;
int hbW;
int vector_hbW;
int vector_bW;
int vector_bW_end;
int hbWbyteSize;
int hbWbufferByteSize;
int jhpc;
int last_row;
int bWbyteSize;

int pbrPivot;
int erPivot;
int rows_to_get;
int num_rows_to_store;
//int rowbool;
unsigned int my_message[4] CACHE_ALIGNED;

static inline int
computePivot(int pRow){
  return (jhpc + pRow%cb[my_rank].numVars - pRow%2)/2;
}

static inline int
computeOffset(int row, int pbr){
    return computePivot(row)-(row-pbr)-pbrPivot%2;
}

static inline void
```

```
get_data(void *row_buf, uint32_t row, uint32_t tag){
  mfc_get((char*) row_buf, cb[my_rank].matrixData_addr + row*JacobRowSize
     /*row_pointers[row]*/, bWbyteSize, tag, 0, 0);
}

static inline void
put_data(void *row_buf, uint32_t row, uint32_t tag){
 mfc_put((char*) row_buf, cb[my_rank].matrixData_addr + row*JacobRowSize /*
    row_pointers[row]*/, bWbyteSize, tag, 0, 0);
}

static inline void
compute_data(vector double* b_row_buf, vector double* e_row_buf, int pbr){
  erPivot = computePivot(present_elim_row);
int lameOffset = (present_elim_row -(pbr-pbr%2))/2;
 // present_elim_column = erPivot - (present_elim_row -(pbr-pbr%2))/2;
 present_elim_column = erPivot - lameOffset;
//       if(present_elim_column < 0){
         //printf("The erPivot is %d and Offset is %d with PBR = %d and PER
             = %d with diff = %d, hbW =%d and PEC = %d\n",
//       erPivot, lameOffset, pbr, present_elim_row, present_elim_row - pbr
   , hbW, present_elim_column);
//}
  adjustFactor = spu_extract(e_row_buf[present_elim_column],
     present_base_row%2)*inv_base_row_factor;
  vFactor = spu_splats(adjustFactor);/*
  if(present_elim_row == pbr+1 && pbr == 0){
      printf("pbrPivot = %d and erPivot = %d and pec = %d with adjustFactor
          = %f\n",
              pbrPivot, erPivot, present_elim_column, adjustFactor);
      printf("The base row is %d with factor is %f and the inv is %f\n",pbr
          , base_row_factor, inv_base_row_factor);
  }*/
  for(int index = 0; index < vector_hbW - (pbr%cb[my_rank].numVars)/2;
     index++){
    e_row_buf[present_elim_column+index]
      = spu_nmsub(vFactor, b_row_buf[pbrPivot+index], e_row_buf[
         present_elim_column+index]);
  }
//     if(pbr == 0){
//  printf("The factor at the end of the b_row is %.16f and %.16f\n",
   spu_extract(b_row_buf[vector_bW_end],0), spu_extract(b_row_buf[
   vector_bW_end],1));
//  printf("vector_bW_end = %d\n", vector_bW_end);
//  }
vFactor = spu_insert((double) 0.0, vFactor, 1);
  e_row_buf[vector_bW_end] = spu_nmsub(vFactor, b_row_buf[vector_bW_end],
```

```
    e_row_buf[vector_bW_end]);
}

int main(uint64_t speid, uint64_t argp, uint64_t envp){
  rows_to_get = 1;
  my_message[3] = 1;
  int ppu_sent_next_base_row = 0;
  nbr = 0;
  adjustFactor = 0;

  unsigned int output = 0x00000001;
  spu_writech(SPU_WrOutMbox, output); //  Tell the PPU that we are here
    do{
    //   printf("The Channel count is now 0x%x\n", spu_readchcnt(
       SPU_WrOutMbox));
  }while(spu_readchcnt(SPU_WrOutMbox));
    NUM_SPUS =  spu_readch(SPU_RdInMbox); // Wait for all the SPU to catch
       up
  spu_writech(SPU_WrOutMbox,1);

  int message = NUM_SPUS;
  my_rank =  spu_readch(SPU_RdInMbox);

  spu_writech(SPU_WrOutMbox,1);

  cb = reinterpret_cast<CONTROL_BLOCK_T*>(malloc_align(NUM_SPUS*sizeof(
     CONTROL_BLOCK_T),7));
  mfc_get(cb, argp, NUM_SPUS*sizeof(CONTROL_BLOCK_T),0,0,0);
  mfc_write_tag_mask(1<<0);
  mfc_read_tag_status_all();


  two_double_byteSize = 2*sizeof(double);

 hbW = cb[my_rank].hbWidth;
 printf("SPU hbW = %d\n",hbW);
 hbW += hbW%2;
 hbWbyteSize = hbW*sizeof(double);
 vector_hbW = hbW/2;
 bWbyteSize = cb[my_rank].bWidth*sizeof(double);
 vector_bW = cb[my_rank].bWidth/2;
 vector_bW_end = vector_bW - 1;
 jhpc = cb[my_rank].jhpc;

 JacobRowSize = bWbyteSize;
 last_row = cb[my_rank].svLength-1;
 printf("JacobRowSize = %d \n",JacobRowSize);
```

```
//  row_pointers = reinterpret_cast<uint32_t*>(malloc_align(sizeof(uint32_t
    )*cb[my_rank].svLength,7));

  uint32_t rowSwapBuffer;
  num_rows_to_store = hbW/NUM_SPUS;
  printf("SPU %d has %d rows to store for a total memory consumption of %d
     bytes\n",
         my_rank, num_rows_to_store, num_rows_to_store*bWbyteSize);
/*
  for(int row = 0; row < cb[my_rank].svLength; row++){
   row_pointers[row] = reinterpret_cast<uint32_t>(cb[my_rank].
      matrixData_addr + row*JacobRowSize);
  }
  */
  //num_rows_to_store = 1;
//  vector double* myRows
  //  = reinterpret_cast<vector double*>(malloc_align
  //       (bWbyteSize*num_rows_to_store,7));
 // memset(reinterpret_cast<void*>(myRows),0, (cb[my_rank].bWidth)*sizeof(
     double)*num_rows_to_store);

  vector double* base_row      = reinterpret_cast<vector double*>(
     malloc_align(bWbyteSize,7));
  vector double* elim_row      = reinterpret_cast<vector double*>(
     malloc_align(bWbyteSize,7));
  vector double* next_elim_row = reinterpret_cast<vector double*>(
     malloc_align(bWbyteSize,7));
  memset(reinterpret_cast<void*>(base_row),      0, bWbyteSize);
  memset(reinterpret_cast<void*>(elim_row),      0, bWbyteSize);
  memset(reinterpret_cast<void*>(next_elim_row), 0, bWbyteSize);

  unsigned int* sig_data = reinterpret_cast<unsigned int*>(malloc_align(
     two_double_byteSize,7));
  sig_data[0] = my_rank+1;
  sig_data[1] = 0;

  int hbPbr;
//  unsigned int* myRows_addr
  //  = reinterpret_cast<unsigned int*>(malloc_align(2*sizeof(unsigned int)
     ,7));
 /// myRows_addr[0] = reinterpret_cast<unsigned int>(myRows);
 // printf("myRows = 0x%x\n and myRows_addr[0] = 0x%x",myRows,myRows_addr
     [0]);
 // mfc_put(reinterpret_cast<char*>(myRows_addr),
//         reinterpret_cast<char*>(cb[my_rank].spu_data_addr),sizeof(
    unsigned int), 0, 0, 0);
```

```
//  mfc_read_tag_status_all();

  vector double* temp_elim_row;
  int ad = 1;
  bool GO = true;

  //  myRows[computePivot(0)] = spu_splats(6.0);
  //  printf("The address of myRows is 0x%x\n with Pivot %d",myRows,
     computePivot(0));
  my_rank =  spu_readch(SPU_RdInMbox);
  spu_writech(SPU_WrOutMbox,1); //  Tell the PPU that we are here
 my_rank =  spu_readch(SPU_RdInMbox);
  //  static int barrierhere = 0;
  //if(!barrierhere){
  //   preload_data(myRows,0,0,hbW);
  //   barrierhere = 1;
  //   }
    while(GO){

      //preload_data(void *row_buf, uint32_t row, uint32_t tag, uint32_t
         num_rows){
      // printf("The first two data points in myRows are %f and %f\n",
      //    spu_extract(myRows[computePivot(0)],0),spu_extract(myRows[
         computePivot(0)],1));

    for(present_base_row = 0; present_base_row < last_row; present_base_row
       ++){

//    for(present_base_row = 0; present_base_row < hbW; present_base_row++)
   {
      pbrPivot = computePivot(present_base_row);
      //       if(present_base_row == 0)
      //                  memcpy((void*)base_row, (void*)myRows,bWbyteSize);
      // else{
      // FETCH BASE ROW
      get_data(base_row,present_base_row,0);
      // END
      // }
      nbr = present_base_row + 1;
      present_elim_row = nbr + my_rank*rows_to_get;
      hbPbr = present_base_row + hbW - present_base_row%cb[my_rank].numVars
         ;

      if (present_elim_row <= last_row){

        //FETCH FIRST
        get_data(elim_row, present_elim_row,0);
```

```
        //FETCH_END
        mfc_read_tag_status_all(); //BARRIER
        // Unpack the base row scale factor
                base_row_factor = spu_extract(base_row[pbrPivot],
                    present_base_row%2);
      inv_base_row_factor = 1/base_row_factor;
      //inv_base_row_factor = 1/spu_extract(base_row[pbrPivot],
        present_base_row%2);
//      if(present_base_row == 0)
        //  printf("The Base Row Factor is %.16f and %.16f\n",spu_extract(
          base_row[pbrPivot],0),spu_extract(base_row[pbrPivot],1));

        for(; present_elim_row+NUM_SPUS <= last_row
              && present_elim_row+NUM_SPUS < hbPbr; present_elim_row +=
                 NUM_SPUS){

          future_elim_row = present_elim_row + NUM_SPUS;
          //FETCH

          get_data(next_elim_row,future_elim_row,0);
          //printf("Fetching Row %d on SPU %d \n",future_elim_row, my_rank)
             ;
          //FETCH_END
          compute_data(base_row,elim_row,present_base_row);

          //SEND

          put_data(elim_row, present_elim_row,0);
          //printf("Sending Row %d on SPU %d \n",present_elim_row, my_rank)
             ;
          //SEND_END

          mfc_read_tag_status_all(); //BARRIER
           temp_elim_row = next_elim_row;
           next_elim_row = elim_row;
           elim_row = temp_elim_row;
        } // END of for-loop elimination

        compute_data(base_row,elim_row,present_base_row);

        //SEND FINAL
        put_data(elim_row,present_elim_row,0);
        //SEND_END
        mfc_read_tag_status_all(); //BARRIER


      } //END of if (present_elim_row <= last_base_row)
```

```
      mfc_put(sig_data,(unsigned int*)cb[my_rank].sig_buf,
         two_double_byteSize, 0, 0, 0);
      spu_readch(SPU_RdInMbox);
      //   printf("%f\n",spu_extract(myRows,0);
    } // ENDS base row loop
    message = spu_read_in_mbox();
    if(message == NUM_SPUS+1)
      GO = false;
      } // Ends Major While loop
  printf("SPU_%d_is_Exiting\n",my_rank);
  return 0;
}
```

Listing C.2. inCore solver

```
/* Copyright (c) 2007 Massachusetts Institute of Technology
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
    copy of
 * this software and associated documentation files (the "Software"), to
    deal in
 * the Software without restriction, including without limitation the
    rights to
 * use, copy, modify, merge, publish, distribute, sublicense, and/or sell
    copies of
 * the Software, and to permit persons to whom the Software is furnished to
    do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included
    in all
 * copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
    OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY
    , FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
    AUTHORS OR
 * COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
    WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
 * CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
    SOFTWARE.
 */
/* @author James Geraci
 */

// James Geraci
// Thesis
// August 15, 2007

#include <spu_mfcio.h>
#include <libmisc.h>
#include <stdio.h>
#include <spu_intrinsics.h>
#include <unistd.h>
#include <string.h>
#include "common.h"
#include <assert.h>
```

```
CONTROL_BLOCK_T* cb CACHE_ALIGNED;
extern int _etext;
extern int _edata;
extern int _end;
static inline void printfData(void);
static inline dma_list_elem_t* createDMA_list(unsigned int*, unsigned int,
   unsigned int);
//void report_memory_info(void){
  // //printf(" &_etext = %p\n", &_etext);
  // //printf(" &_edata = %p\n", &_edata);
  // //printf(" &_end = %p %f\n", &_end, (float) ((int) &_end)/1024);
  // //printf(" sbrk(0) = %p = %d = %f\n",sbrk(0), (int) sbrk(0),(float) ((
     int) sbrk(0))/1024);
  // //printf(" bytes on Heap %d\n", (int) sbrk(0) - (int) &_end);
  // //printf(" Kbytes on Heap %f\n", (float) ((int) sbrk(0) - (int) &_end)
     /1024);
//}

int my_rank;
static int NUM_SPES;
//static uint32_t* row_pointers;
static int two_double_byteSize;
static int hbS;
static int vector_hbS;
static int vector_bS;
static int vector_bS_end;
static int vector_nI;
static int jhpc;
static int last_row;
static int lrOffset;
static uint32_t bSbyteSize;
static int numExtraRows;
static int numStoredRows;
static int numStoredRowsBefore;

double adjustFactor;
double inv_base_row_factor;
unsigned int* base_row_addr;
vector double* myRows;
vector double* base_row;
vector double* nxtBaseRow;
vector double vFactor;
int workingSPEs;
int present_elim_column;
int pbrPivot;
int erPivot;
int Joffset;
```

```
int Joffset2;
int myMinRow;
int myMaxRow;
int pbrOwner;
int extraRows;
double NewtonIter;
double nxtNewtonIter;
unsigned int dma_list_size;
unsigned int hbSbyteOffset;
unsigned int pbrByteOffset;
int numFailed;
int lastRowOwner;
int sRows;
int message;
int matrixAddr;
uint64_t cbLocation;
int sbsRow;
dma_list_elem_t* myDMAlist;

static inline int
computePivot(int pRow){
  return (jhpc + pRow%cb[my_rank].numVars - pRow%2)/2;
}

static inline void
compute_numStoredRows(void){
  numStoredRows = hbS/workingSPEs;
  //printf("\n SPE %d thinks workingSPEs = %d\n",my_rank, workingSPEs);
  // //printf("numStoredRows is %d\n",numStoredRows);
  numExtraRows = hbS%workingSPEs;
  for(int index = 0; index < workingSPEs; index++){
    //  if(cb[index].working == 1){
      cb[index].numStoredRows = numStoredRows;
      if(index < numExtraRows){
        cb[index].numStoredRows += 1;
        // }
    }
  }
  //printf("The Number of Stored Rows for SPE %d is %d\n",my_rank,
     numStoredRows);
  numStoredRows = cb[my_rank].numStoredRows;
  numStoredRowsBefore = 0;

  for(int index = 0; index < my_rank; index++){
    if(cb[index].working){
    numStoredRowsBefore += cb[index].numStoredRows;
    }
```

```
  }
  return;
}

void determineLastRowOwner(int pbOff){
  lastRowOwner = 0;
  extraRows = (cb[my_rank].svLength - pbOff)%hbS;
  sbsRow = cb[my_rank].svLength /*- pbOff*/ - extraRows;
  if(extraRows == 0 && my_rank == workingSPEs - 1){
    lastRowOwner = 1;
    // //printf("SPE %d is the last Row Owner\n",my_rank);
  }else{
    if(sbsRow + numStoredRowsBefore < (cb[my_rank].svLength /*- pbOff*/)
       && sbsRow + numStoredRowsBefore + numStoredRows >= (cb[my_rank].
          svLength /*- pbOff*/))
      {
        lastRowOwner = 1;
        // //printf("SPE %d is the last Row Owner\n",my_rank);
        sRows = (cb[my_rank].svLength /*- pbOff*/) - sbsRow -
           numStoredRowsBefore;
      }
  }
  return;
};

static inline void resetDMAlist(int pbOff){
  free_align((void*) myDMAlist);
  matrixAddr = cb[my_rank].matrixData_addr + numStoredRowsBefore*bSbyteSize
     ;
  //printf("FOR SPE %d numStoredRowsBefore is %d\n",my_rank,
     numStoredRowsBefore);
  myDMAlist = createDMA_list(&dma_list_size,matrixAddr + pbOff*bSbyteSize,
     numStoredRows*bSbyteSize);
}

static inline void
preload_data(uint32_t row_buf, uint32_t br_buf, uint32_t num_rows, uint32_t
    tag, uint32_t offx){
  spu_mfcdma32((volatile*) row_buf, (unsigned int) myDMAlist,
     dma_list_size, 0, MFC_GETL_CMD);
  spu_mfcdma32(reinterpret_cast<void*>(br_buf), cb[my_rank].matrixData_addr
      + offx, bSbyteSize, tag, MFC_GET_CMD);
  // spu_mfcdma32(reinterpret_cast<void*>(row_buf), cb[my_rank].
     matrixData_addr, bSbyteSize, tag, MFC_GET_CMD);
  spu_mfcstat(MFC_TAG_UPDATE_ALL);
  return;
}
```

```
static inline dma_list_elem_t*
createDMA_list(unsigned int* dls, unsigned int ea_low, unsigned int nbytes)
  {
  dma_list_elem_t* mylist;
  unsigned int index = 0;
  unsigned int nbytes2 = nbytes;

  if(!nbytes)
    return 0;

  // first, determine num elements in myDMAlist
  while(nbytes2 > 0){
    unsigned int sz2;

    sz2 = (nbytes2 < MAX_BYTES_ONE_DMA) ? nbytes2 : MAX_BYTES_ONE_DMA;
    nbytes2 -= sz2;
    index++;
  }

  mylist = reinterpret_cast<dma_list_elem_t*>(malloc_align(index*sizeof(
     dma_list_elem),7));

  index = 0;
  while(nbytes > 0){
    unsigned int sz;

    sz = (nbytes < MAX_BYTES_ONE_DMA) ? nbytes : MAX_BYTES_ONE_DMA;

    mylist[index].size.all32 = sz;
    mylist[index].ea_low = ea_low;
    nbytes -= sz;
    ea_low += sz;
    index++;
  }
  *dls = index*sizeof(dma_list_elem_t);
  //  return index*sizeof(dma_list_elem_t);
  return mylist;
}

/************************************/
/********* resetting the LU **********/
/************************************/
static inline void reSetLU(int *pb){
  //printf("SPE %d is adjusting workingSPEs from %d to %d by %d\n",my_rank,
     workingSPEs,workingSPEs-numFailed,numFailed);
  workingSPEs -= numFailed;
```

```
// //printf("Getting the new cb\n");
// Reget the control block with new ranks and all
spu_mfcdma32(cb, cbLocation, NUM_SPES*sizeof(CONTROL_BLOCK_T),0,
   MFC_GET_CMD);
mfc_write_tag_mask(1<<0);
spu_mfcstat(MFC_TAG_UPDATE_ALL);
// //printf("new cb received\n");

// Readjust 'my_rank'
for(int search_ranks = 0; search_ranks < workingSPEs; search_ranks++){
  if(cb[search_ranks].old_rank == my_rank){
    // //printf("making SPE %d into SPE %d\n",cb[search_ranks].old_rank,
       cb[search_ranks].rank);
    my_rank = cb[search_ranks].rank;
    cb[search_ranks].old_rank = my_rank;
  }
}

mfc_put(reinterpret_cast<char*>(base_row_addr),
        reinterpret_cast<char*>(cb[my_rank].base_row_addr),4*sizeof(
           unsigned int), 0, 0, 0);

spu_mfcstat(MFC_TAG_UPDATE_ALL);
spu_writech(SPU_WrOutMbox,1);
spu_readch(SPU_RdInMbox);
//printf("Got The one asdjfkal;sdfajs;ldfjas;djfl;l;sadjfl;asd\n");
mfc_get(cb, cbLocation, NUM_SPES*sizeof(CONTROL_BLOCK_T),0,0,0);
spu_mfcstat(MFC_TAG_UPDATE_ALL);


// Reallocate myRows buffer
free_align((void*) myRows);
// //printf("Number of Stored Rows Old = %d\n",numStoredRows);
compute_numStoredRows();
// //printf("Number of Stored Rows New = %d\n",numStoredRows);
myRows = reinterpret_cast<vector double*>(malloc_align
                                          (bSbyteSize*numStoredRows,7));
memset((void*) myRows, 0, bSbyteSize*numStoredRows);

// Recreate DMAlist and bring matrix back in
resetDMAlist(*pb);
//if(my_rank == 1){
// Use the below to simply restart the LU from pbr = 0;
//  preload_data(reinterpret_cast<uint32_t>(myRows),reinterpret_cast<
   uint32_t>(base_row),
//       numStoredRows,0);
//}
```

```
  // Use the below to restart the LU in mid-computation
  preload_data(reinterpret_cast<uint32_t>(myRows),reinterpret_cast<uint32_t
     >(base_row),
                numStoredRows,0,*pb*bSbyteSize);
  //printf("Below is //printfData while resetting SPE %d\n",my_rank);
  //printfData();
  // return;
  // reset starting condition
  pbrOwner = 0;
  determineLastRowOwner(*pb);
  myMinRow = spu_extract(myRows[vector_bS_end],1);

  if(lastRowOwner && myMinRow >= sbsRow){
    myMaxRow = spu_extract(myRows[(sRows-1)*vector_bS + vector_bS_end],1);
  }else{
    myMaxRow = spu_extract(myRows[(numStoredRows-1)*vector_bS +
       vector_bS_end],1);
  }
  //printf("The present_base_row is being resent by SPE %d\n",my_rank);
  //printf("SPE %d has a min row of %d and a max row of %d for base row of
     %d\n",my_rank,myMinRow, myMaxRow, *pb);
  *pb -= 1;
  numFailed = 0;
  //printf("SPE %d now thinks that numFailed is %d\n",my_rank,numFailed);
  return;
};

static inline void Printfdata(void){
  //printf("cb[%d].rank = %d\n",my_rank,cb[my_rank].rank);
  //printf("cb[%d].old_rank = %d\n",my_rank,cb[my_rank].old_rank);
  //printf("cb[%d].local_store_addr = 0x%x\n",my_rank,cb[my_rank].
     local_store_addr);
  //printf("cb[%d].matrixData_addr = 0x%x\n",my_rank,cb[my_rank].
     matrixData_addr);
  //printf("cb[%d].control_addr = 0x%x\n",my_rank,cb[my_rank].control_addr)
     ;
  //printf("cb[%d].base_row_addr = 0x%x\n",my_rank,cb[my_rank].
     base_row_addr);
  //printf("cb[%d].brLS_addr[0] = 0x%x\n",my_rank, cb[my_rank].brLS_addr
     [0]);
  //printf("cb[%d].working = 0x%x\n",my_rank,cb[my_rank].working);
  //printf("cb[%d].workingSPEs = %d\n",my_rank,cb[my_rank].workingSPEs);
  return;
}

static inline void
compute_data(vector double* b_row_buf, vector double* Jacob, int pbr,
```

```
 vector double* nbr){

pbrPivot = computePivot(pbr);
int pbrModVars = pbr%cb[my_rank].numVars;
int pbrMod2 = pbr%2;
int per;
int Joff;
int perxx;
int addrsa;
int subN = (pbrModVars)/2;

if(pbr-myMinRow >= 0)
  Joff = pbr-myMinRow + 1;
else
  Joff = 0;
inv_base_row_factor = 1/spu_extract(b_row_buf[pbrPivot],pbrMod2);

for(int index = 0; index < numStoredRows - pbrOwner; index++){
  Joffset = ((Joff + index)%numStoredRows)*vector_bS ;

  per = (int) spu_extract(Jacob[Joffset + vector_bS_end],1);
  if(spu_extract(Jacob[Joffset + vector_nI],1) < pbr || pbr == 0){
    //if(my_rank == 0 && pbr == 500)
    ////printf("SPE %d is calling compute in Newton Iter %f\n",my_rank,
       NewtonIter);
    if((per == cb[my_rank].svLength - 1) || per == pbr + hbS - pbrModVars
       - 1){
      index = numStoredRows - pbrOwner + 1000;
    }
    if(per > pbr && per < pbr + hbS - pbrModVars){
      erPivot = computePivot(per);
      int pbbb = (per-(pbr-pbrMod2))/2;
      present_elim_column = erPivot - pbbb;
      perxx = per*bSbyteSize;
      Joffset2 = Joffset + present_elim_column;

      adjustFactor
        = spu_extract(Jacob[Joffset2],pbrMod2)*inv_base_row_factor;
      vFactor = spu_splats(adjustFactor);
      addrsa = cb[my_rank].matrixData_addr + perxx;
      for(int index2 = 0; index2 < vector_hbS - subN; index2++){
        Jacob[Joffset2 + index2]
          = spu_nmsub(vFactor,b_row_buf[pbrPivot + index2],
                      Jacob[Joffset2 + index2]);
      }

      vFactor = spu_insert(0.0,vFactor,1);
```

```
        Jacob[Joffset + vector_bS_end]
          = spu_nmsub(vFactor,b_row_buf[vector_bS_end],
                      Jacob[Joffset + vector_bS_end]);
        // Here is where we write the pbr into the row;
        Jacob[Joffset + vector_nI] = spu_insert((double)pbr,Jacob[Joffset
            + vector_nI],1);


        // Here is where we write the modified row back into main memory
        spu_mfcstat(MFC_TAG_UPDATE_ALL);
        spu_mfcdma32((char*) &Jacob[Joffset], cb[my_rank].matrixData_addr
            + per*bSbyteSize,
                    bSbyteSize, 0, MFC_PUT_CMD);

      }

      if(__builtin_expect(per == pbr+1,0)){
        for(int index3 = 0; index3 < workingSPEs; index3++ ){
          //if(cb[index3].working){
            spu_mfcdma32(&Jacob[Joffset],
                         cb[index3].local_store_addr + cb[index3].brLS_addr
                            [0] + ((pbr + 1)%2)*bSbyteSize,
                         bSbyteSize,0,MFC_PUT_CMD);
            //}
        }
      }
    }
  }
  return;
}

main(uint64_t speid, uint64_t argp, uint64_t envp){
  //  //printf("Returning\n");
  // return 0;
  NewtonIter = (double) 0.0;
  nxtNewtonIter = (double)0.0;
  int Kg = 0;
  //  report_memory_info();
  two_double_byteSize = 2*sizeof(double);
  float dd = (256.0 - (float)((int) &_end)/1024);
  bool GO = true;
  cbLocation = argp;
  // Begin
  // Setup Function
  lastRowOwner = 0;
  sRows = 0;
  spu_writech(SPU_WrOutMbox,1); //  Tell the PPU that we are here
```

```
NUM_SPES =  spu_readch(SPU_RdInMbox); // Wait for all the SPU to catch up
spu_writech(SPU_WrOutMbox,1);

message = NUM_SPES;
my_rank =  spu_readch(SPU_RdInMbox);
// End Setup Function

// Begin
// This doesn't change
cb = reinterpret_cast<CONTROL_BLOCK_T*>(malloc_align(NUM_SPES*sizeof(
   CONTROL_BLOCK_T),7));
memset((void*) cb, 0, NUM_SPES*sizeof(CONTROL_BLOCK_T));

spu_mfcdma32(cb, cbLocation, NUM_SPES*sizeof(CONTROL_BLOCK_T),0,
   MFC_GET_CMD);
mfc_write_tag_mask(1<<0);
spu_mfcstat(MFC_TAG_UPDATE_ALL);

//printfData();
workingSPEs = cb[my_rank].workingSPEs;
hbS = cb[my_rank].hbWidth;

hbS += hbS%2;
vector_hbS = hbS/2;

bSbyteSize = cb[my_rank].bWidth*sizeof(double);
hbSbyteOffset = hbS*bSbyteSize;
vector_bS = cb[my_rank].bWidth/2;
vector_bS_end = vector_bS - 1;
vector_nI = vector_bS - 2;   // The data is READY for the coming Newton
   Iteration
jhpc = cb[my_rank].jhpc;

last_row = cb[my_rank].svLength-1;
lrOffset = last_row*bSbyteSize;

base_row = reinterpret_cast<vector double*>(malloc_align(2*bSbyteSize,7))
   ;
memset((void*) base_row, 0, bSbyteSize*2);
nxtBaseRow = base_row;

base_row_addr = reinterpret_cast<unsigned int*>(malloc_align(4*sizeof(
   unsigned int),7));
memset((void*) base_row_addr, 0, 4*sizeof(unsigned int));
base_row_addr[0] = reinterpret_cast<unsigned int>(base_row);
```

```
extraRows = (cb[my_rank].svLength)%hbS;
sbsRow = cb[my_rank].svLength-extraRows;

int LastBlockStart = sbsRow;
// End This doesn't change

// Begin
// Setup Function 2
compute_numStoredRows();
myRows = reinterpret_cast<vector double*>(malloc_align
                                          (bSbyteSize*numStoredRows,7));
memset((void*) myRows, 0, bSbyteSize*numStoredRows);

mfc_put(reinterpret_cast<char*>(base_row_addr),
        reinterpret_cast<char*>(cb[my_rank].base_row_addr),4*sizeof(
           unsigned int), 0, 0, 0);

spu_mfcstat(MFC_TAG_UPDATE_ALL);
spu_writech(SPU_WrOutMbox,1);
my_rank =  spu_readch(SPU_RdInMbox);

mfc_get(cb, argp, NUM_SPES*sizeof(CONTROL_BLOCK_T),0,0,0);
spu_mfcstat(MFC_TAG_UPDATE_ALL);

// determineLastRowOwner(0);

// END WTF does this do?
matrixAddr = cb[my_rank].matrixData_addr +  numStoredRowsBefore*
   bSbyteSize;
myDMAlist = createDMA_list(&dma_list_size,matrixAddr,numStoredRows*
   bSbyteSize);
//  report_memory_info();
spu_writech(SPU_WrOutMbox,1);
my_rank =  spu_readch(SPU_RdInMbox);
// END Setup Function 2
//printf("FOR SPE %d numStoredRowsBefore is %d\n",my_rank,
   numStoredRowsBefore);
while(GO){
 base_row = reinterpret_cast<vector double*>(cb[my_rank].brLS_addr[0]);
 nxtBaseRow = reinterpret_cast<vector double*>(cb[my_rank].brLS_addr
    [0] + bSbyteSize);
  determineLastRowOwner(0);
  //printf("SPE %d says lastRowOwner == %d\n",my_rank,lastRowOwner);
  resetDMAlist(0);
  LastBlockStart = sbsRow;
  NewtonIter++;
  preload_data(reinterpret_cast<uint32_t>(myRows),reinterpret_cast<
```

```
    uint32_t>(base_row),
              numStoredRows,0,0);

pbrOwner = 0;
myMinRow = -1;
myMaxRow = -1;
// france
for(int present_base_row = 0; present_base_row < last_row;
    present_base_row++){
  //      if(my_rank == 0)
  //         //printf("the present base row is %d\n",present_base_row);

  pbrByteOffset =  present_base_row*bSbyteSize;

  base_row = reinterpret_cast<vector double*>(cb[my_rank].brLS_addr
     [0] + (present_base_row%2)*bSbyteSize);
  nxtBaseRow = reinterpret_cast<vector double*>(cb[my_rank].brLS_addr
     [0] + ((present_base_row+1)%2)*bSbyteSize);
  int compute = 1;
  if(myMinRow >= LastBlockStart && present_base_row > myMaxRow){
    compute = 0;
  }else{
    if(present_base_row > myMaxRow){
      myMinRow = spu_extract(myRows[vector_bS_end],1);

      if(lastRowOwner && myMinRow >= sbsRow){
        myMaxRow = spu_extract(myRows[(sRows-1)*vector_bS +
            vector_bS_end],1);
      }else{
        myMaxRow = spu_extract(myRows[(numStoredRows-1)*vector_bS +
            vector_bS_end],1);
      }
    }
  }
  pbrOwner = 0;
  if(present_base_row >= myMinRow && present_base_row <= myMaxRow){
    pbrOwner = 1;
    if(present_base_row < LastBlockStart){
      spu_mfcdma32((char*) &myRows[((present_base_row-myMinRow)%
          numStoredRows)*vector_bS],
                   cb[my_rank].matrixData_addr + pbrByteOffset +
                      hbSbyteOffset, bSbyteSize, 0, MFC_GET_CMD);
    }
    spu_mfcdma32((char*) base_row,
                 cb[my_rank].matrixData_addr + pbrByteOffset,
                    bSbyteSize, 0, MFC_PUT_CMD);
```

```
  }

  if(compute){
    compute_data(base_row,myRows,present_base_row,nxtBaseRow);
  }

  spu_mfcstat(MFC_TAG_UPDATE_ALL);
  // int ps = my_rank;
    Kg = 0;
    /*
            if(present_base_row == 2118 && ((my_rank == Kg  &&
               workingSPEs == NUM_SPES && NewtonIter == 50.0) ||
                                                    (workingSPEs ==
                                                       NUM_SPES-1 &&
                                                       my_rank == Kg
                                                        && NewtonIter
                                                        == 100.0) ||
                                                    (workingSPEs ==
                                                       NUM_SPES-2 &&
                                                       my_rank == Kg
                                                        && NewtonIter
                                                        == 200.0) ||
                                                    (workingSPEs ==
                                                       NUM_SPES-3 &&
                                                       my_rank == Kg
                                                        && NewtonIter
                                                        == 300.0)
                                                    )){
    */
        if(0){
    //        if(present_base_row == 2118 && (my_rank == Kg || my_rank
        == Kg-1) && workingSPEs == NUM_SPES && NewtonIter == 5.0){
      return 0;
  }else{

    spu_writech(SPU_WrOutMbox,my_rank);
  }
  numFailed = spu_readch(SPU_RdInMbox);
  if(numFailed != 0){
    //printf("The number Failed is %d\n",numFailed);
    reSetLU(&present_base_row);
    //        //printf("SPE %d is now totally done with resetting during
        present base row of %d\n",my_rank, present_base_row);
  }

} // ENDS base row loop
//printf("SPE %d says lastRowOwner == %d\n",my_rank,lastRowOwner);
```

```
    // It needs to be determined at run time who owns the last row to put.
    if(lastRowOwner){
      //  // //printf("SPE %d is sending out the last Row\n",my_rank);
      spu_mfcdma32((char*) &myRows[((last_row-myMinRow)%numStoredRows)*
          vector_bS], cb[my_rank].matrixData_addr + lrOffset,
                    bSbyteSize, 0, MFC_PUT_CMD);
      spu_mfcstat(MFC_TAG_UPDATE_ALL);
    }
    message = spu_read_in_mbox();
    if(message == NUM_SPES+1)
      GO = false;
  } // Ends Major While loop
  return 0;
}
```

Listing C.3. Makefile

```
#############################################################################
#                           Target
#############################################################################

PROGRAM_ppu       = FVbatteryModel
OBJS= StdAfx.o matrixData.o Electrode.o CV.o electrodeSizeData.o OLLLPV.o
   ORLRPV.o volumeChemData.o UPV.o FVbatteryModel.o
    electrolyteElectrodeChemData.o ORPV.o PVIConditions.o OLULPV.o BPV.o
    Electrolyte.o ldxElectrode.o ldElectrode.o ldElectrodeChemData.o
    volumeSpatialData.o myGlobals.o ORURPV.o electrodeChemData.o PV.o FV.o
   OLPV.o cellSizeData.o volumeIConditions.o ldxElectrodeChemData.o ULCV.o
   URCV.o LLCV.o LRCV.o
#############################################################################
#                           Local Defines
#############################################################################
IMPORTS           = -lspe2 -lmisc -lsimdmath -lpthread


#############################################################################
#                           make.footer
#############################################################################
INCLUDE += -I. -I.. -I/opt/cell/sysroot/opt/cell/sdk/usr/include
LIBS = /opt/cell/sysroot/opt/cell/sdk/usr/lib
CXX=/opt/ibmcmp/xlc/cbe/9.0/bin/ppuxlc++
CXXFLAGS= -qcpluscmt -M -ma $(INCLUDE) -qaltivec -qenablevmx
#OPTFLAGS=-03

RM=rm
RMFLAGS=-f



.cc.s:
        $(CXX) $(CXXFLAGS) $(OPTFLAGS) -S $< -g -o $@

.cc.o:
        $(CXX) $(CXXFLAGS) $(OPTFLAGS) -c $< -g -o $@

.s.o:
        $(ASM) $(INCLUDE) -o $@ $<

$(PROGRAM_ppu) : $(OBJS)
        $(CXX) $(CXXFLAGS) -g -o $@ $^ -L$(LIBS) -Wl,-m,elf32ppc -R$(LIBS)
           $(IMPORTS)

clean:
```

```
$(RM) $(RMFLAGS) *.o
$(RM) $(RMFLAGS) *.d
```

Bibliography

[1] Toyota's hybrid plans for 2010. "http://www.soultek.com/blog/2006/06/toyotas-hybrid-plans-for-2010.html%", March 2008.

[2] AP contributed. GM to make lithium-ion hybrids in 2010. "http://money.cnn.com/2008/03/04/autos/gm_lithium_hybrids/index.htm", March 2008.

[3] K. Peters. Review of factors that affect the deep cycling performance of valve-regulated lead/acid batteries. *Journal of Power Sources*, (59):9–13, 1996.

[4] John Christensen and John Newman. Stress generation and fracture in lithium insertion materials. *J Solid State Electrochem*, 10:293–319, March 2006.

[5] P.Ruetschi. Aging mechanisms and service life of lead-acid batteries. *Journal of Power Sources*, (127):33–44, 2004.

[6] D. Berndt. Valve-regulated lead-acid batteries. *Journal of Power Sources*, (100):29–46, 2001.

[7] S. Piller, M. Perrin, and A. Jossen. Methods for state-of-charge determination and their applications. *Journal of Power Sources*, 96:113–120, 2001.

[8] B. Le Pioufle, J.F. Fauvarque, and P. Delalande. A performing lead acid cell state of charge indicator based on data fusion. *Electrochemical Society Proceeding*, 16, 1996.

[9] V.H. Johnson. Battery preformance models in advisor. *Journal of Power Sources*, 4806:1–9, 2002.

[10] Wootaik Lee, Daeho Choi, and Myoungho Sunwoo. Modelling and simulation of wehicle electric power system. *Journal of Power Sources*, 109:58–66, 2002.

[11] A.H. Anbuky and P.E. Pascoe. VRLA battery state-of-charge estimation in telecommunication power systems. *IEEE Transactions on Industrial Electronics*, 47(3):565–573, 2000.

[12] S. Rodrigues, N. Munichandraiah, and A.K. Shukla. A review of state-of-charge indication of batteries by means of a.c. impedance measurements. *Journal of Power Sources*, 87:12–20, 2000.

[13] Y. Morimoto, Y. Ohya, K. Abe, T. Yoshida, and H. Morimoto. Computer simulation of the discharge reaction in lead-acid batteries. *J.Electrochem.Soc.:Electrochemical Science and Technology*, 135(2):293–298, 1988.

[14] C.W. Chao. Continuous monitoring of acid stratification during charge/discharge by holographic laser interferometry. *Journal of Power Sources*, 55:243–246, 1995.

[15] R.J. Ball, R. Kurian, R. Evans, and R. Stevens. Failure mechanisms in valve regulated lead/acid batteries for cyclic applications. *Journal of Power Sources*, 109:189–202, 2002.

[16] E.C. Dimpault-Darcy, T.V. Nguyen, and R.E. White. A two-dimensional mathematical model of a porous lead dioxide electrode in a lead-acid cell. *Journal of Electrochemical Society*, 135(2):278–285, 1988.

[17] Jacob Bear and Yehuda Bachmat. *Introduction to Modeling of Transport Phenomena in Porous Media*. Kluwer Academic Publishers, Boston, Massachusetts, 1990.

[18] Theodore L. Brown, Jr. H. Eugene LeMay, and Bruce E. Bursten. *Chemistry The Central Science, 7th Edition*. Prentice Hall Publishing, Upper Saddle River, New Jersey 07458, 1997.

[19] Charles M. Vest. *Holographic Interferometry*. Wiley, New York, 1979.

[20] ed. David R. Lide. *Concentrative Properties of Aqueous Solutions: Density, Refractive Index, Freezing Point Depression, and Viscosity, in CRC Handbook of Chemistry and Physics, Internet Version 2006*. Taylor and Francis, Boca Raton, FL, 2006.

[21] F. Alavyoon, A. Eklund, F.H. Bark, R. I. Karlsson, and D. Simonsson. Theoretical and experimental studies of free convection and stratification of electrolyte in a lead acid cell during recharge. *Electrochimica Acta*, 36(14):2153–2164, 1991.

[22] A. Eklund and R.I. Karlsson. Free convection and stratification of electrolyte in the lead-acid cell without/with a separator during cycling. *Electrochemica Acta*, 37(4):681–694, 1992.

[23] F.H. Bark and Alavyoon. Convection in electrochemical systems. *Applied Scientific Research*, 53:11–34, 1994.

[24] Rolf H. Muller. *Advances in Electrochemistry and Electrochemical Engineering*, volume 9. Interscience, New York, 1973.

[25] P. Hariharan. *Basics of Interferometry, Second Edition*. Academic Press, Burlington, MA, 2007.

[26] Hans Bode. *Lead Acid Batteries, Translated by Brodd/Kordesch*. Wiley-Interscience, New York, 1977.

[27] John S. Newman. *Electrochemical Systems*. Prentice-Hall, Inc, Englewood Cliffs, N.J., 1991.

[28] Allen Bard and Larry Faulkner. *Electrochemical Methods, Second Edition*. John Wiley and Sons, Inc, New Jersey, 2001.

[29] John Newman and William Tiedemann. Porous-electrode theory with battery applications. *AIChE*, 21(1):25–41, 1975.

[30] Hiram Gu, T.V. Nguyen, and R.E. White. A mathematical model of a lead-acid cell, discharge, rest, and charge. *Journal of Electrochemical Society*, 134(12):2953–2960, 1987.

[31] Dawn M. Bernardi and Hiram Gu. Two-dimensional mathemtical model of a lead-acid cell. *Journal of Electrochemical Society*, 140(8):2250–2258, 1993.

[32] A. Tenno, R. Tenno, and T. Suntio. Charge-discharge behavior of VRLA batteries model calibration and application for state estimation and failure detection. *Journal of Power Sources*, 103:42–53, 2001.

[33] T.V. Nguyen, R.E. White, and H. Gu. The effects of separator design on the discharge performance of a starved lead acid cell. *Journal of Electrochemical Society*, 137(10):2998–3004, 1990.

[34] W.B. Gu, C.Y. Wang, and B.Y. Liaw. Numerical modeling of coupled electrochemical and transport processes in lead-acid batteries. *Journal of Electrochemical Society*, 144(6):2053–2061, 1997.

[35] D. Simonsson P. Ekdunge. The discharge behaviour of the porous lead electrode in lead acid battery.ii. *Journal of Applied Electrochemistry*, 19:136–141, 1989.

[36] J.E. Welch, F.H. Harlow, J.P. Shannon, and B.J. Daly. The MAC method: A computing technique for solving viscous, incompressible, transient fluid flow problems involving free surfaces. *Report LA-3425, Los Alamos Scientific Laboratory*, 1966.

[37] Suhas V. Patankar. *Numerical Heat Transfer and Fluid Flow*. Hemisphere Publishing Corporation, Washington, 1980.

[38] H.K. Versteeg and W. Malalasekera. *An Introduction to Computational Fluid Dynamics, The Finite Volume Method*. Prentice Hall, Essex, England, 1995.

[39] Gilbert Strang. *Introduction to Applied Mathematics*. Wellesley-Cambridge Press, 1986.

[40] R. Sureshkumar. Forward and backward euler methods. "http://web.mit.edu/10.001/Web/Course_Notes/Differential_Equations_Note%s/node3.html", May 2008.

[41] G. Papzov and D. Pavlov. Influence of cycling current and power profiles on the cycle life of lead/acid batteries. *Journal of Power Sources*, 62:193–199, 1996.

[42] Saman Amarasinghe. Multicore programming primer and programming competition introduction. "http://cag.csail.mit.edu/ps3/lectures/6.189-lecture1-intro.pdf", January 2007.

[43] Wikipedia. Spursengine. "http://en.wikipedia.org/wiki/SpursEngine", November 20 2007.

[44] Toshiba starts sample shipping of spursengine se1000 high-performance stream processor. "http://www.toshiba.co.jp/about/press/2008_04/pr0801.htm", April 8 2008.

[45] Kate Greene. The trouble with multi-core computers, adding more cores to a computer makes it faster, but it also makes it tricky to program. how will computer scientists cope? *Technology Review*, November 1 2006.

[46] Ron Wilson. Multicore software problems edging toward center stage. *Electronics Design, Strategy, News*, April 6 2007.

[47] Associated Press. Microsoft and Intel join UC-Berkeley and U. of Illinois to push parallel computing. *The Mercury News*, March 19 2008.

[48] John Markoff. Race is on to advance software for chips. *New York Times*, April 30 2008.

[49] C.Demerjian. Intel 80 core chip revealed in full detail. "http://www.theinquirer.net/default.aspx?article=37572", February 11 2007.

[50] Jack Dongarra. "http://top500.org", April 1 2008.

[51] Wikipedia. Denormal number. "http://en.wikipedia.org/wiki/Denormal_number", May 2 2008.

[52] *Cell Broadband Engine Programming Handbook*. IBM Systems and TEchnology Group, Hopewell Junction, NY, April 2007.

[53] M. Riley, B. Flachs, S. Dhong, G. Gervais, S. Weitzel, M. Wang, D. Boerstler, M. Bolliger, J. Keaty, J. Pille, R. Berry, O. Takahashi, Y. Nishino, and T. Uchino. Implementatin of the 65nm cell broad-

band engine. In *IEEE 2007 Custom Integrated Circuits Conference CICC*. IEEE, September 21-24 2007.

[54] O. Takahashi, C. Adams, E. Behnen, O. Chiang, S. Cottier, P. Coulman, J. Culp, G. Gervais, M. Gray, Y. Itaka, C. Johnson, F. Kono, L.Maurice, K. McCullen, L. Nguyen, Y. Nishino, H. Noro, J. Pille, M. Riley, S. Tokito, T. Wagner, and H. Yoshihara. Migration of cell broadband engine from 65nm soi to 45nm soi. In *IEEE 2008 International Solid-State Circuits Conference ISSCC*. IEEE, February 3-7 2008.

[55] Thomas Lippert. At the limits of scalability? In *13th SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, March 12-14 2008.

[56] James L. Tomkins. The asci red tops supercomputer. "`http://www.sandia.gov/ASCI/Red/`", April 7 2008.

[57] NCSA. Latency results from pallas mpi benchmarks. "`http://vmi.ncsa.uiuc.edu/performance/pmb_lt.php`", March 23 2005.

[58] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Analysis and Applications*, 20(3):720–755, 1999.

[59] James W. Demmel, John R. Gilbert, and Xiaoye S. Li. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM J. Matrix Analysis and Applications*, 20(4):915–952, 1999.

[60] Xiaoye S. Li and James W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Mathematical Software*, 29(2):110–140, June 2003.

[61] James W. Demmel, John R. Gilber, and Xiaoye S. Li. Superlu users' guide. "http://crd.lbl.gov/~xiaoye/SuperLU/superlu_ug.pdf", November 2007.

[62] Bill Thies. Multicore programming primer and programming competition introduction, streamit language. "http://cag.csail.mit.edu/ps3/lectures/6.189-lecture8-streamit.pdf", January 2007.

[63] Phil Sung. Multicore programming primer and programming competition introduction, simd programming on cell. "http://cag.csail.mit.edu/ps3/recitation6/6.189-recitation6.pdf", January 2007.

[64] Toshiba showcases latest advanced technologies at ces 2008. "http://www.tacp.toshiba.com/news/newsarticle.asp?newsid=191", April 8 2008.

[65] Bureau of Transportation Statistics USDOT. Table 1-17: New and used passenger car sales and leases. "http://www.bts.gov/publications/national_transportation_statistics/htm%1/table_01_17.html", February 2 2007.

[66] J.L. Leray. Effects of atmospheric neutrons on devices, at sea level and in avionics embedded systems. *Microelectronics Reliability*, 47:1827–1835, September 2007.

[67] Eric C. Hannah. System with response to cosmic ray detection. United States Patent, January 2007. Patent No. 7,166,847.

[68] Eric C. Hannah. Cosmic ray detectors for integrated circuit chips. United States Patent, December 2007. Patent No. 7,309,866.

[69] J. Langou, Z. Chen, G. Bosilca, and J. Dongarra. Recovery patterns for iterative methods in a parallel unstable environment. *SIAM J. Sci. Comput.*, 30(1):102–116, 2007.

[70] James S. Plank, Youngbae Kim, and Jack J. Dongarra. Fault tolerant matrix operations for networks of workstations using diskless checkpointing. *Journal of Parallel and Distributed Computing*, 43:125–138, 1997.

[71] Remi Delmas, Julien Langou, and Jack Dongarra. Fault-tolerant algorithms for dense linear algebra. In *13th SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, March 12-14 2008.