Electrochemical Battery Model for the Cell Broadband Engine

James Geraci Sudarshan Raghunathan John Chu

Massachusetts Institute of Technology
jrgeraci@mit.edu

Abstract

This paper discusses the implementation of a two-dimensional electrochemical battery model of a lead acid battery cell on the Playstation 3. The battery model equations are a set of coupled highly non-linear partial differential equations that evolve with time. At each time step, these equations are solved using Newton's Method, and a direct skyline LU solver without partial pivoting is used to solve for the battery cell's state at each Newton step.

The entire model was implemented on a Sony Playstation 3 and the direct solver at each Newton iteration step exploits the parallelism in the Cell Broadband Engine to improve performance. The parallelized direct solver outperforms state-of-the-art dense and sparse solvers running on an AMD Opteron 246-based workstation.

1 Outline

This article first describes the electrochemical model used and the types of output one can get from the model. Next, the numerical algorithm used for simulating the model is described along with a description of the implementation on the Cell Broadband Engine. Finally, some performance and scaling results are reported and compared with existing dense and sparse solvers on a contemporary desktop workstation.

2 Two-Dimensional Battery Model

The electrochemical battery model used in this study is based on the model given in [1] and derived from first principles in [2].

A typical lead acid battery, consisting of a lead dioxide electrode, a lead electrode, and a liquid sulfuric acid electrolyte is illustrated in Figure 1. The model tries to simulate the primary electrochemical reactions that occur during charge and discharge in the area of an electrochemical cell between the lead and lead dioxide plates (indicated by the dotted region in Figure 1).

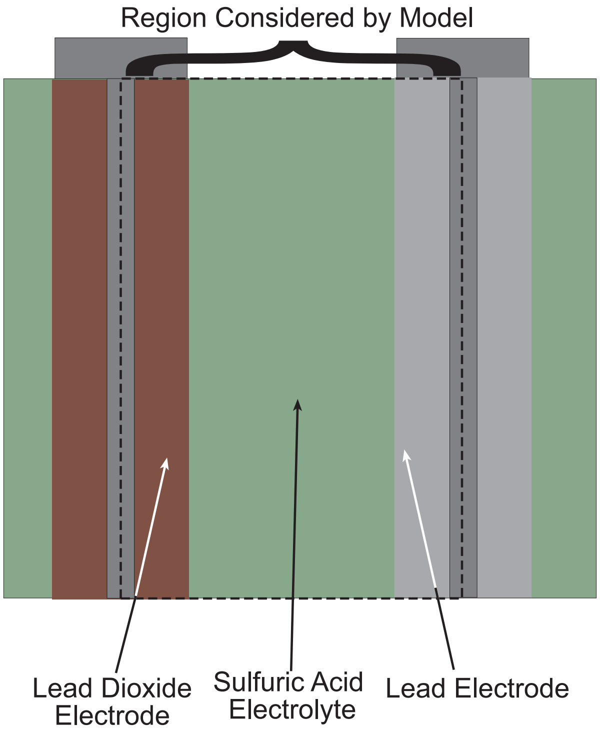

Figure 1. A complete lead acid battery cell.

The model has four state variables that evolve with time: the porosity of a region ε, the concentration of the electrolyte C, the liquid phase electrical potential ϕ_l, and the solid phase electrical potential ϕ_s.

The spatial region of interest is discretized into two different staggered two-dimensional grids of volumes as seen in Figure 2. This technique is known as a *staggered grid*. Three of the state variables, Concentration C, Liquid phase electrical potential ϕ_l, and Solid phase electrical potential ϕ_s are centered on one grid (called the PV grid for potential values), while Porosity ε is centered on the other grid (called the FV grid for flux values). In contrast to the model described in [1], the staggered grid approach for the potential and flux values avoids having to enforce continuity conditions at the electrode/electrolyte boundaries.

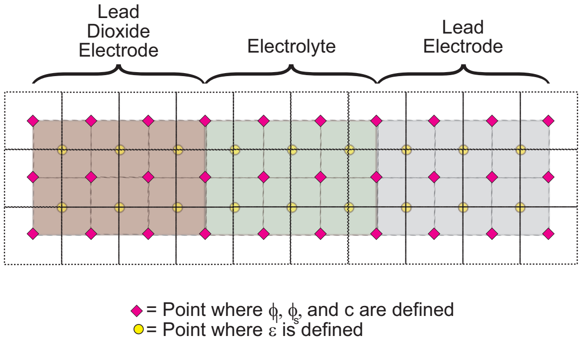
Lead Dioxide Electrode
Electrolyte
Lead Electrode
◆ = Point where ϕ_l, ϕ_s, and c are defined
● = Point where ε is defined

Figure 2. A battery cell with both the PV grid and FV grids shown. Diamonds show the center of the PV grid while circles show the center of the FV grid.

Implicit time stepping is used for the variables on the PV grid. They are treated as a coupled set of non-linear equations and are solved for at each time step using Newton's method. The porosity ε, however, is not included in the Jacobian for each Newton step as it is on the FV grid. Explicit time stepping is used for the porosity equation.

The equations for the model are listed here:

1. Change of Porosity

$$\frac{\partial}{\partial t}(\varepsilon)+SF_{cop}\left\{\nabla\cdot\left(-\kappa^{\text{eff}}\nabla\phi_l\right)+\nabla\cdot\left(-\kappa^{\text{eff}}_{ec}\nabla ln\left(C\right)\right)\right\}=0 \tag{1}$$

2. Material Balance

$$\frac{\partial}{\partial t}(\varepsilon C)+\nabla\cdot\left(-D^{\text{eff}}\nabla C\right) + SF_{MB}\left\{\nabla\cdot\left(-\kappa^{\text{eff}}\nabla\phi_l\right)+\nabla\cdot\left(-\kappa^{\text{eff}}_{ec}\nabla ln\left(C\right)\right)\right\}=0 \tag{2}$$

3. Electrode Kinetics

$$\nabla\cdot\left(-\kappa^{\text{eff}}\nabla\phi_l\right)+\nabla\cdot\left(-\kappa^{\text{eff}}_{ec}\nabla ln\left(C\right)\right) -SA_{int}i_0^{\text{PbO}_2}\frac{C}{C*}\left(e^{\frac{\alpha_a n(\phi_s-\phi_l-\phi^{eq})\mathbf{F}}{RT}}-e^{\frac{\alpha_c n(\phi_l-\phi_s+\phi^{eq})\mathbf{F}}{RT}}\right)=0 \tag{3}$$

4. Divergence of Current

$$\nabla\cdot\left(-\kappa^{\text{eff}}\nabla\phi_l\right)+\nabla\cdot\left(-\kappa^{\text{eff}}_{ec}\nabla ln\left(C\right)\right)+\nabla\cdot\left(-\sigma^{\text{eff}}\nabla\phi_s\right)=0 \tag{4}$$

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

$$\begin{aligned} i_l &= -\underbrace{\varepsilon^{ex}\kappa}_{\kappa^{\text{eff}}}\nabla\phi_l+-\underbrace{\varepsilon^{ex}\kappa\left(2t_+^\circ-1\right)\frac{\text{RT}}{\mathbf{F}}}_{\kappa^{\text{eff}}_{ec}}\nabla ln\left(C\right) \\ &= -\kappa^{\text{eff}}\nabla\phi_l+-\kappa^{\text{eff}}_{ec}\nabla ln\left(C\right) \end{aligned} \tag{5}$$

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

$$SF_{cop}=\begin{cases}-\frac{1}{2\mathbf{F}}\left[\frac{MW_{PbSO_4}}{\rho_{PbSO_4}}-\frac{MW_{PbO_2}}{\rho_{PbO_2}}\right] & \text{lead dioxide electrode}\\ 0 & \text{electrolyte}\\ -\frac{1}{2\mathbf{F}}\left[\frac{MW_{Pb}}{\rho_{Pb}}-\frac{MW_{PbSO_4}}{\rho_{PbSO_4}}\right] & \text{lead electrode}\end{cases} \tag{6}$$

$$SF_{mb}=\begin{cases}\left(\frac{2t_+^\circ-3}{2\mathbf{F}}\right) & \text{lead dioxide electrode}\\ 0 & \text{electrolyte}\\ \left(\frac{2t_+^\circ-1}{2\mathbf{F}}\right) & \text{lead electrode}\end{cases} \tag{7}$$

$$SA_{int}=\begin{cases}SA_{int}^{\text{PbO}_2} & \text{lead dioxide electrode}\\ 0 & \text{electrolyte}\\ SA_{int}^{\text{Pb}} & \text{lead electrode}\end{cases} \tag{8}$$

$$\phi^{eq}=\begin{cases}\phi^{eq}_{\text{PbO}_2} & \text{lead dioxide electrode}\\ 0 & \text{electrolyte}\\ \phi^{eq}_{\text{Pb}} & \text{lead electrode}\end{cases} \tag{9}$$

$$\sigma^{\text{eff}}=\begin{cases}\varepsilon^{exm}\sigma^{\text{PbO}_2} & \text{lead dioxide electrode}\\ 0 & \text{electrolyte}\\ \varepsilon^{exm}\sigma^{\text{Pb}} & \text{lead electrode}\end{cases} \tag{10}$$

2.1 Code Interface

2.1.1 PPU Code

The PPU code is located in the BatteryModel directory. Within the BatteryModel directory, three important files are 'FVbatteryModel.cpp', 'Globals.cpp', and 'common.h'.

Bulding the Model To build the program simply type

```
make
```

at the command prompt from within the BatteryModel directory on any system with a properly installed cell-sdk. This should build the battery model and produce a program called FVbatteryModel.

Running the Model The model can be run by typing

```
./FVbatteryModel
```

at the command prompt.

Controlling the Model The physical dimensions of the model (units of cm) are set using the following code from 'FVbatteryModel.cpp'.

```
  double ldxHeight = 10.0;
  double ElectrolyteHeight = 10.0;
  double ldHeight = 10.0;

  double ldxDepth = 7.0;
  double ElectrolyteDepth = 7.0;
  double ldDepth = 7.0;

  double ldxWidth = 0.03;
  double ElectrolyteWidth = 0.03;
  double ldWidth = 0.03;
```

The number of volumes in the FV grid is also set in the file 'FVbatteryModel.cpp' using the following lines:

```
  csData->setldxFVColumns(11);
  csData->setElectrolyteFVColumns(11);
  csData->setldFVColumns(11);
  csData->setnumFVRows(31);
```

The number of columns in the PV grid will be the total number of columns in the FV grid plus 1. The number of rows in the PV grid will be the total number of rows in the FV grid plus 1.

The time step size, the number of time steps to be taken, and the current drawn from the cell are set within the 'myGlobals.cpp' file by adjusting the following code:

```
extern double dt = 4;  // seconds
extern int tSteps = 60000; // number of time steps
extern double I = 1e0;   // current
```

Compute Power The number of SPUs used is set by the

```
NUM_SPU
```

parameter in 'common.h'.

2.1.2 SPU Code

There are four files in the 'BatteryModel/spu' directory. These are 'Makefile', 'spu_prog.cc', 'spu_prog.cc.BANDED' and 'spu_prog.cc.FULL'. 'spu_prog.cc.BANDED' contains a banded direct solver, and 'spu_prog.cc.FULL' contains a dense direct solver. Both solvers work off forward elimination back substitution.

Selecting a solver Either solver can be used by copying it onto the file 'spu_prog.cc', removing the file 'spu_prog.o' if it is present in the directory, and remaking the program. For example,

```
[BatterModel/spu]# cp spu_prog.cc.FULL spu_prog.cc
[BatterModel/spu]# rm spu_prog.o
[BatterModel/spu]# cd ..
[BatterModel]# make
```

will build the model with the dense solver instead of the default banded solver.

2.2 Output

After each time step, the value of each of the three physical parameters C, ϕ_l, and ϕ_s is written to a file named 'mySV_x.dat', where 'x' is the time step starting with 0. These files can be read into Matlab and plotted using the function 'svDataTool(int,int,int)' found in the file 'svDataTool.m' which is also found in the 'BatteryModel' directory.

'svDataTool(numColumns, numRows, TimeStep)' is a function that takes three arguments. The first is the number of PV columns that existed in the model that created the data file. Next is the number of PV rows that existed in the model that created the data file, and last is the time step of the data that you wish to plot. Typing

```
svDataTool(34,32,0)
```

at the Matlab prompt will plot the data from 'mySV_0.dat' if there were 34 PV columns and 32 PV rows in the model that created the file 'mySV_0.dat'.

Figure 3 shows how 'svDataTool' might plot the concentration of electrolyte, C, throughout the cell after 800 seconds of discharge at a rate of $I = 1^{-3}$.

3 Sparse Algorithm and Implementation

Implicit time stepping was used for the variables on the PV grid, and the values of the state of the variables on the PV grid were solved for at each time step using Newton iteration. Each Newton iteration requires the solution of a system of the form $Jx = r$, where J is the Jacobian matrix produced by the system, see Figure 4, and r is the residual vector.

The Jacobian for this system has a condition number of approximately 10^8. This is in part due to the weak coupling between the volumes in the y-direction due to the large height of the battery cell (10.0 cm) when compared to the small width of the battery cell (0.09 cm). Due to the large spacing in the y direction, the reference potential, chosen to be ϕ_l in the lower right corner of the lead electrode, has

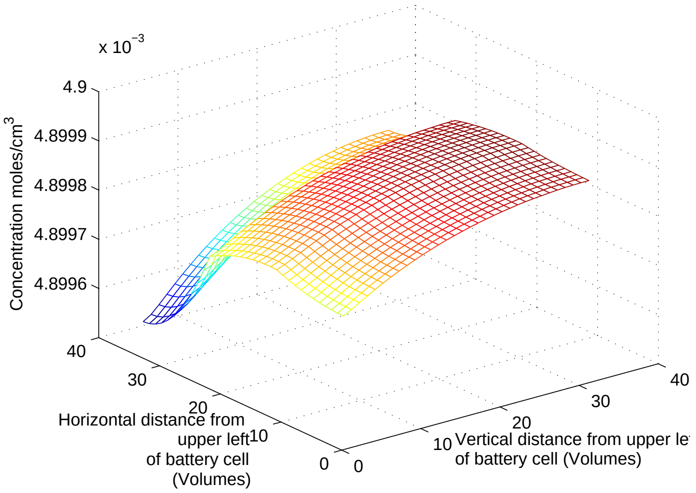

Figure 3. Concentration of electrolyte C within each PV of the battery cell after the battery cell has been discharged at a rate of $I = 1^{-3}$ for 800 seconds. The point located at coordinates $(0, 0)$ corresponds to the upper left corner of the region described by the model as seen in Figure 1

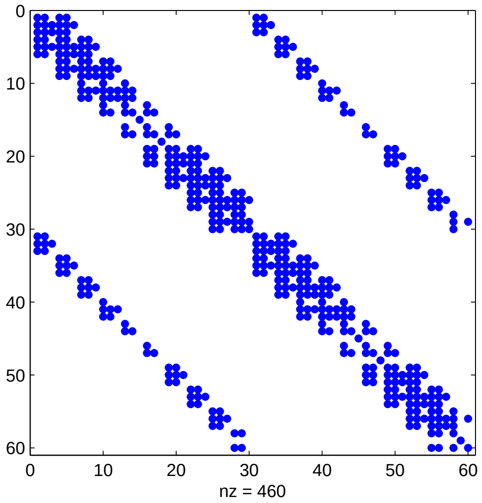

Figure 4. Location of the non-zero entries in the Jacobian. In this case, the Jacobian is a 60 × 60 with matrix with 460 non-zero entries. The skyline form of the matrix can be clearly seen.

only a weak influence on the values at other points in the system in the y-direction.

Since the Jacobian has a condition number of approximately 10^8, there are effectively only 8 digits of useful precision in any answer, consequently, the model is said to converge when the largest r value is less than 10^{-8}. Furthermore, such a poor condition number does not often work well with an iterative solver, so a direct solver is used to compute x at each Newton iteration.

The direct solver has two major computational phases: forward elimination followed by back substitution. The forward elimination part is more computationally intensive, $\mathcal{O}(b^2N)$ (where N is the size of the Jacobian and b is the half-bandwidth) and is therefore performed in parallel on the Synergistic Processing Units (SPUs) of the cell processor with the Power Processing Unit (PPU) being used for synchronization. The back substitution phase is computationally less expensive ($\mathcal{O}(bN)$) and is therefore done completely on the PPU.

Pseudocode for the forward elimination step running on the SPUs is given in Algorithm 1. For conciseness, Algorithm 1 only shows the operations on the Jacobian; a similar set of operations is used to update the corresponding component of the residual.

In the actual implementation, the Jacobian is assembled in the main memory of the PS3 as a dense matrix with elements stored in row-major order. This scheme ensures unit stride access to the non-zero elements in each row by the SPUs during elimination. The actual code that runs on the SPUs is double buffered and employs SIMD instructions to maximize computation efficiency.

4 Performance results

Figure 5 illustrates the performance of the direct banded solver for Jacobians of different sizes with increasing number of SPUs. It is observed that while the algorithm does scale with increasing number of SPUs, the speedup is sublinear beyond two SPUs. This can be attributed to the increase in time it takes the PPU to synchronize the increasing numbers of SPUs as seen in Figure 6.

Figure 7 shows the performance of the parallel direct solver relative to existing sparse and dense solvers in UMFPACK [3] and LAPACK [4] respectively for a Jacobian matrix of size 3264×3264.

Even though the technique used in each of the solvers is different, for consistency the performance in GFlops is measured using the operation count for solving a dense linear linear system of order N:

$$\text{Gflops} = \frac{\frac{2}{3}N^3 - \frac{1}{2}N^2}{\Delta t \times 1024^3} \tag{11}$$

```
Input: The Jacobian matrix, $\mathbf{J}$ at a Newton iteration
       with half-bandwidth $b$ and the residual vector, $\mathbf{r}$
Result: Forward elimination is performed on the
        Jacobian and the residual vector
for $i \longleftarrow 1$ to $N-1$ do
    // Fetch base row $i$ from main memory
    $\mathbf{J}_{\text{local},i} =$ post_recv($\mathbf{J}[\mathbf{P}[i], i : i+b]$)
    // Fetch first elimination row for this SPU, $i+$spuid from main memory
    if $i + spuid \le N$ then
        $\mathbf{J}_{\text{local},i+\text{spuid}} =$ post_recv($\mathbf{J}[\mathbf{P}[i+spuid], i+spuid : i+spuid+b]$)
    end
    // Wait for base row and first elimination row to arrive
    wait_for_completion($\mathbf{J}_{local,i}$, $\mathbf{J}_{local,i+spuid}$)
    for $j \longleftarrow i+spuid$ to $i+b$ do
        // Pre-fetch next elimination row for this SPU, $j+$spuid from main memory
        if $j + spuid \le N$ then
            $\mathbf{J}_{\text{local},j+\text{spuid}} =$ post_recv($\mathbf{J}[\mathbf{P}[j+spuid], j+spuid : j+spuid+b]$)
        end
        // Perform elimination on row $j$
        $\mathbf{J}_{\text{local},j}[i] \longleftarrow \mathbf{J}_{\text{local},j}[i] / \mathbf{J}_{\text{local},i}[i]$
        for $k \longleftarrow i+1$ to $i+b$ do
            $\mathbf{J}_{\text{local},j}[k] \longleftarrow \mathbf{J}_{\text{local},j}[j] \times \mathbf{J}_{\text{local},i}[k]$
        end
        // Post the updated row back to main memory
        post_send($\mathbf{J}_{local,j}$)
        // Wait for all pending posts
        wait_for_completion($\mathbf{J}_{local,j}$)
        // Wait for next elimination row to arrive
        if $j + spuid \le N$ then
            wait_for_completion($\mathbf{J}_{local,j+spuid}$)
        end
    end
    // Wait before starting next base row
    wait_for_notification
end
```

Algorithm 1: Algorithm for parallel forward elmination step of the banded direct solver.

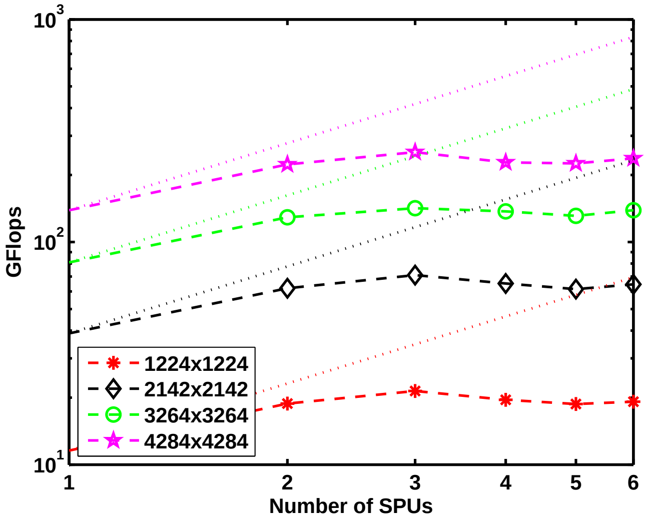

Figure 5. Performance in Gflops achieved by the CBE based LU algorithm. To maintain consistency between this figure and Figure 7 Gflops has been computed using (11)**, the equation for a dense solver even though the PS3 solver is a banded solver.**

where Δt is the total amount of time to perform forward elimination and back substitution.

The results for UMFPACK and the PS3 based solver can be interpreted as being the performance of a dense solver that would be needed in order to achieve the Δt achieved by UMFPACK or the PS3 based solver.

Figure 7 shows that the PS3 based solver is 1.56x faster than UMFPACK when 2 SPUs are used. However, due to the extra synchronization overhead incurred when using more SPUs, there is no performance to be gained when using more than 2 SPUs.

References

[1] D. M. Bernardi and H. Gu. Two-dimensional mathematical model of a lead-acid cell. *Journal of Electrochemical Society*, 140(8):2250–2258, 1993.

[2] J. Geraci. *Electrochemical Battery Models*. PhD thesis, Massachusetts Institute of Technology, 2007.

[3] T.A. Davis. Algorithm 832: UMFPACK v4.3—an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30(2):196–199, 2004.

[4] *LAPACK UsersǴuide*. SIAM, 1999.

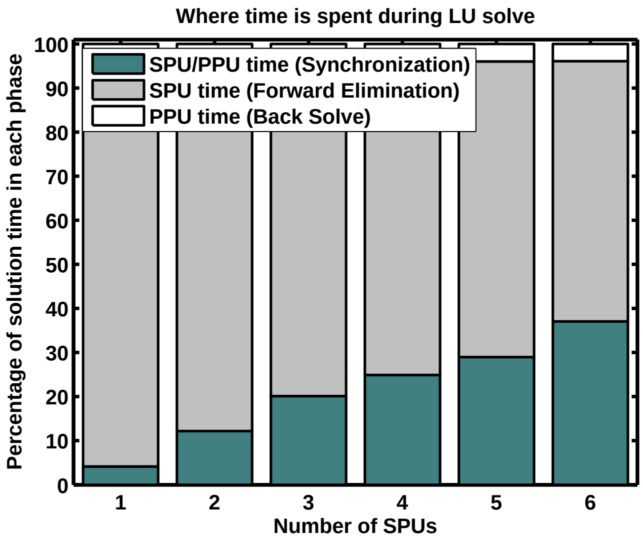

Figure 6. Three major time consuming steps for the sparse LU solver on the PS3. These are forward elimination, back solve and SPU synchronization. As the number SPUs is increased, the percent of time spent synchronizing the SPUs increases dramatically.

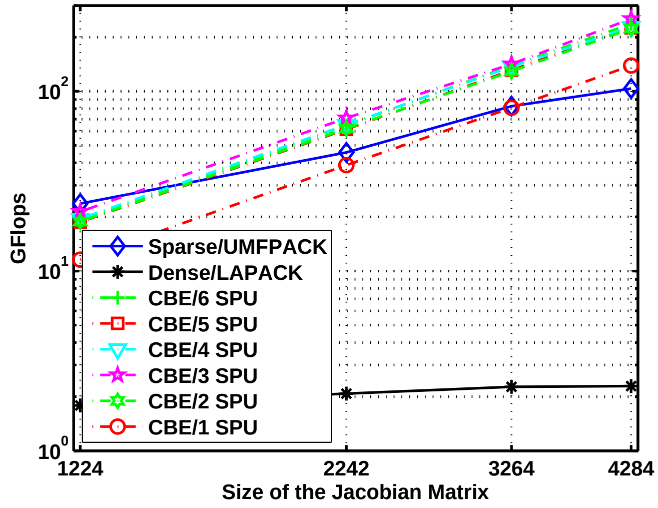

Figure 7. A comparison of the performance of LAPACK, UMFPACK, and the PS3 based solver. All performance numbers were computed using (11)**.**