Deterministic Random Walk: A New Preconditioner for Power Grid Analysis

Jia Wang, Member, IEEE, Xuanxing Xiong, Member, IEEE, and Xingwu Zheng

Abstract—Iterative linear equation solvers rely on high-quality preconditioners to achieve fast convergence. For sparse symmetric systems arising from large power grid analysis problems, however, preconditioners generated by traditional incomplete Cholesky factorization are usually of low quality, resulting in slow convergence. On the other hand, preconditioners generated by random walks are quite effective to reduce the number of iterations, though requiring considerable amount of time to compute in a stochastic manner. We propose in this paper a new preconditioning technique for power grid analysis, named deterministic random walk that combines the advantages of the above two approaches. Our proposed algorithm computes the preconditioners in a deterministic manner to reduce computation time, while achieving similar quality as stochastic random walk preconditioning by modifying fill-ins to compensate dropped entries. We have proved that for such compensation scheme, our algorithm will always succeed, which otherwise cannot be guaranteed by traditional incomplete factorizations. We demonstrate that by incorporating our proposed preconditioner, a conjugate gradient solver is able to outperform a state-of-the-art algebraic multigrid preconditioned solver for dc analysis, and is very efficient for transient simulation on public IBM power grid benchmarks.

Index Terms—Power grid analysis, preconditioned conjugate gradient, preconditioner, random walks.

I. INTRODUCTION

POWER supply noise is a critical issue that must be addressed in modern chip designs since it may affect important design metrics including timing and power consumption [5], [10], [18], [35]. Modeling devices as current sources and power distribution networks with parasitics as RLC grids, power grid analysis techniques [10], [28] compute power supply noises under given current excitations by solving one or more systems of linear equations [16] depending on whether the steady-state dc analysis or the time-domain transient simulation should be performed. The noises are verified against certain bounds that help to decompose power grid verification from timing verification [33], and the designers could take further actions if the outcomes are not satisfactory.

The key problem of power grid analysis can be formulated as the following sparse system of linear equations, where the noise vector \( \mathbf{x} \) should be obtained given the system matrix \( A \) and the excitation vector \( \mathbf{b} \):

\[
A \mathbf{x} = \mathbf{b}.
\]

In this paper, we consider dc analysis and transient simulation when there is no mutual inductance, and thus assume \( A \) to be a sparse nonsingular symmetric diagonally dominant matrix with nonpositive off-diagonal elements [6]. Note that these also implies that \( A \) is symmetric positive definite (s.p.d.). Despite extensive studies, it remains extremely challenging [29] to solve the above key problem to match the increasing complexity of power grids: a single system may contain more than one million unknown variables in recent public benchmarks [24], [25], [29], and capacitive couplings between VDD and GND grids and inductances in wires further complicate transient simulation. At such large problem sizes, direct solvers like Cholesky factorization [7], [14] are inefficient, and are prohibitive in memory usage without introducing a hierarchy [45]. The state-of-the-art research on power grid analysis can be separated in two groups. The efforts of the first group lead to efficient algorithms that can obtain approximated solutions by exploiting structures of the powergrid, e.g., hierarchical analysis [45], multigrid [11], [20], [28], [36], [49], [50], random walks [23], [31], Poisson solver [34], frequency domain methods [17], [46], and model order reduction techniques [21], [22], [39]. The second group [6], [8], [12], [32], [37], [41], [42], [47], [48] utilizes iterative solvers, especially preconditioned conjugate gradient (PCG) as most systems are s.p.d., to achieve better solution accuracy.

It is well known that the quality of the preconditioners has a huge impact on the performance of the iterative solvers. However, for large problems arise from power grid analysis, obtaining a good preconditioner remains very challenging. Among the traditional preconditioning techniques, including incomplete Cholesky (IC) [27], support graph [3], approximated inverse [2], and algebraic multigrid (AMG) methods [30], recent published results for the aforementioned benchmarks [25], [29] show that only an AMG–PCG solver [42] has shown significant improvement, while other preconditioning techniques [8], [41], [47] are less effective, requiring a large number of iterations toward convergence. On the other hand, the stochastic preconditioning technique [32] based on random walks is very effective to reduce the number of PCG iterations. Qian and Sapatnekar [32] pointed out, the superior quality comes from the fact that the columns of the factor are computed independently so that errors will not accumulate as in traditional incomplete Cholesky (IC)/incomplete LU (ILU) preconditioners.
However, since to compute the columns independently requires to perform Monte Carlo simulations on the power grid, even when the walks are extensively reused [32], it takes much more time than other techniques [19], [27], [47] that compute the factors in a rather deterministic manner. Stochastic random walk preconditioning is, therefore, not suitable when the system is only solved for a few times, e.g., for dc analysis. Nevertheless, since the preconditioner itself is an approximated root-free Cholesky (LDL$^T$) factorization of the system matrix, each PCG iteration is simpler than AMG preconditioning [42], making it desirable to study new factorization-based preconditioners.

In this paper, we propose a new preconditioner, named deterministic random walk (DRW) that overcomes the above difficulty, while achieving similar quality compared to stochastic random walk preconditioning. Our key observation is that if the columns of the factor are computed sequentially, a random walk can be decomposed into a few random walks whose probabilities are either easily known or already computed. This enables us to avoid most, if not all, Monte Carlo simulations, resulting in a deterministic algorithm that computes the approximated factor efficiently. As our algorithm will usually locate more entries in the factor than stochastic random walk, we propose to keep only the largest few in absolute value per column to maintain its sparsity, and to compensate the dropped ones by decreasing the remaining ones. Such compensation scheme differs our preconditioner from various traditional IC/ILU preconditioners [15], [26], [27], where the off-diagonal elements of the factor are only allowed to be increased or dropped (increased to 0) to guarantee the correctness of the algorithm for the system matrix being a symmetric M-matrix. As a comparison, by leveraging random walks, we are able to prove the correctness of our algorithm when more flexible dropping and compensation schemes are adopted. In addition, our proposed preconditioner differs from hierarchical random walks [23], [31] in the sense that they explicitly approximate the possibly dense Schur complement after partial factorization by Monte Carlo simulations, while we implicitly factor the Schur complement without actually generating it first.

We demonstrate that a PCG solver using our proposed DRW preconditioner can efficiently explore the tradeoffs between the preconditioner size, the time to compute the preconditioner, and the time to solve a problem iteratively. Using a preconditioner size similar to that of the system matrix $A$, we reduce the time to compute the preconditioner without degrading its quality dramatically and obtain an efficient solver for dc analysis that is able to outperform the state-of-the-art AMG–PCG solver PowerRush [42]. Moreover, in comparison to stochastic random walk preconditioning, using a preconditioner of similar size, we reduce the time to compute it by 6.8× and the solver is 2.4× faster for IBM public power grid transient simulation benchmarks [24]. Finally, when more fill-ins are allowed for transient simulation, our iterative solver is able to complete all the six benchmarks within an hour, while a direct solver would need at least 20 min but requiring 5× fill-ins.

The rest of this paper is organized as follows. Incomplete factorization and stochastic random walk preconditioning are reviewed in Section II. Our proposed DRW preconditioner is presented in Section III. Implementation details are discussed in Section IV. After experimental results are reported in Section V, Section VI concludes this paper.

II. PRELIMINARIES

A. Matrices and Graphs

For an $n \times 1$ vector $\mathbf{x}$, we denote its $i$th element by $x_i$ for $1 \leq i \leq n$, and a vector obtained from its $i$th to $j$th elements by $\mathbf{x}_{i:j}$. For an $n \times n$ matrix $X$, we denote its element on the row $i$ and the column $j$ by $x_{i,j}$ for $1 \leq i \leq n$ and $1 \leq j \leq n$. The submatrix of $X$ spanning the rows $i_1$ to $i_2$ and the columns $j_1$ to $j_2$ is denoted by $X_{i_1:j_2,j_1:j_2}$. We opt to omit $j_2$ or $j_1$ if $i_1 = i_2$ or $j_1 = j_2$, respectively.

We define $\mathcal{D}(X)$ to be the diagonal of $X$, i.e., $\mathcal{D}(X) = \{x_{i,i} \mid 1 \leq i \leq n\}$. We say that $X$ is column-wise diagonally dominant if $\forall k, x_{k,k} \geq \sum_{l \neq k} |x_{l,k}|$, and that $X$ is unit lower triangular if $\forall k, x_{k,k} = 1$, and $X_{1:k-1,k} = 0$.

We define the extended matrix graph of $X$ to be a weighted directed graph $G^*_{X} = (V^*_{X}, E^*_{X}, w^*_{X})$. The vertex set $V^*_{X}$ consists of the vertices 1, 2, ..., $n$ corresponding to the rows and columns of $X^1$ and an additional vertex $n+1$. For any $x_{i,j} \neq 0$ where $i \neq j$, an edge $(i, j)$ is introduced to $E^*_{X}$ with the weight $w^*_{X}(i, j) = -x_{i,j}$. For any $k$ satisfying $x_{k,k} \neq 0$, an edge $(k, n+1)$ is introduced with the weight $w^*_{X}(k, n+1) = x_{k,k}$. Similarly, an edge $(n+1, k)$ is introduced with the weight $w^*_{X}(n+1, k) = x_{k,k} - \sum_{l \neq k} x_{l,k}$ if the weight is not zero. An example system matrix $A$ and its extended matrix graph are shown in Fig. 1(a).

B. Incomplete LDL$^T$ Factorization

We review incomplete Cholesky factorization in its root-free forms, i.e., incomplete LDL$^T$ factorization, because of its close relationship to random walk-based preconditioning techniques studied in [32] and in this paper.

Let $n$ be the dimension of the system matrix $A$. Since $A$ is s.p.d., it is well known [14] that there exists a unit lower triangular matrix $L$ and a diagonal matrix $D$ with all diagonal elements being positive such that $A = LDL^T$. Since $A$ is also an M-matrix, its incomplete LDL$^T$ factorization does

1Our definition of $G^*_{X}$ implicitly depends on the matrix ordering of $X$.
2An M-matrix is a matrix where the elements in its inverse are all nonnegative.
C. Stochastic Random Walk Preconditioning

Consider the matrix $T = AD(A)^{-1}$, i.e., the matrix obtained from $A$ by dividing its column $k$ by $a_{kk}$ for every $k$. A random walk game can be defined in the extended matrix graph $G_T^*$, treating the edge weights as the transition probabilities. For $1 \leq k < i \leq n+1$, let $k \xrightarrow{d_{k,i}} i$ be the set of the random walks from $k$ to $i$ whose internal vertices, if there are any, are all no more than $k$, and denote their probability by $Pr[k \xrightarrow{d_{k,i}} i]$. Consider all such random walks from a particular $k$, i.e., $\bigcup_{i=k+1}^{n+1} k \xrightarrow{d_{k,i}} i$. Since $A$ is symmetric nonsingular, $n+1$ is reachable from $k$. It implies that $\sum_{i=k+1}^{n+1} Pr[k \xrightarrow{d_{k,i}} i] = 1$ and that the expectation $E_k$ of the number of the times that one such random walk passes $k$ is finite.

The above definition of random walks is adopted from [32] with the modification that the vertices are numbered reversely—the vertices 1, 2, ..., $n$ as defined above are the same set of vertices defined as $n$, ..., 2, 1 in [32]. While the conclusion of [32] states that the outcomes of their random walks are the entries in the LDL$^T$ factorization of the matrix obtained by reversing $A$’s columns and rows, this modification allows us to rewrite the conclusion as that the outcomes of the above random walks are the entries in the LDL$^T$ factorization of $A$. To be more specific, we have

$$
\hat{d}_{k,i} = \frac{d_{k,i}}{E_k} \quad \forall 1 \leq k \leq n
$$

$$(5)$$

$$
\hat{l}_{i,k} = -Pr[k \xrightarrow{d_{k,i}} i] \quad \forall 1 \leq k < i \leq n.
$$

$$(6)$$

The factorization and the probabilities and the expectations of the system shown in Fig. 1(a) are listed in Fig. 1(b).

Note that (6) can be extended to all $1 \leq k < i \leq n+1$ if we define $\hat{l}_{n+1,k} = -\sum_{i=1}^{n} \hat{l}_{i,k}$.

Equations (5) and (6) motivate one to obtain an approximated LDL$^T$ factorization of $A$ via Monte Carlo simulations. When $L$ and $D$ are compared with the outcome of incomplete LDL$^T$ factorization as described in (4), an immediate advantage is that there is no systematic difference since estimations of the probabilities and expectations could be either larger or smaller than their exact values, and there is no error accumulation.

However, the fundamental difficulty of stochastic random walk preconditioning is that the length of a random walk cannot be bounded, and thus the algorithm only terminates in a probabilistic sense even when one bounds the number of Monte Carlo simulations. For example, for the system shown in Fig. 1(a), a random walk starting from 4 could pass the edges $1 \rightarrow 2$ and $2 \rightarrow 1$ for unbounded number of times before finally reaching 5. In the extreme case when there are traps in the system [31], e.g., when the weights of the edges $1 \rightarrow 2$ and $2 \rightarrow 1$ are much larger compared with others, it will take excessive time to compute the preconditioner. To overcome such difficulty, one may partition the system [45] and use hierarchical random walks [23], [31] to generate a macromodel of the global grid before applying stochastic random walk preconditioning. However, since the macromodel essentially is an approximation of the Schur complement after partial factorization, to compute and to store it explicitly could be very costly. As a comparison, our proposed preconditioner utilizes the key observation that if the columns of the factor are computed sequentially, a random walk can be decomposed into a few random walks whose probabilities are either easily known or already computed, and thus circumvents the difficulty to walk the trap directly to approximate the probability or to generate a macromodel that could be dense. Details follow.

III. PRECONDITIONING WITH DETERMINISTIC RANDOM WALKS

According to (4)–(6), both incomplete LDL$^T$ factorization and stochastic random walks preserve the following structures of the exact $D$ and $\hat{L}$ in their preconditioners $D$ and $L$.

$P1$: The off-diagonal elements of $L$ are nonpositive and $L$ is column-wise diagonally dominant.

$P2$: An off-diagonal element of $L$ is zero if the corresponding element of $\hat{L}$ is zero.

$P3$: The diagonal elements of $D$ are positive.

As we are interested in designing a new preconditioning technique to exploit the advantages of both approaches, it is desirable for our preconditioner to meet all the above conditions from $P1$ to $P3$.

A. Structure of Random Walks

Our first step toward a better preconditioner than stochastic random walks is to circumvent the difficulty in bounding the lengths of the random walks. We introduce a new group of random walks defined as follows. For $1 \leq k < i \leq n+1$, let $k \xrightarrow{\hat{d}_{k,i}} i$ be the set of the random walks from $k$ to $i$ whose internal vertices, if there is any, are all less than $k$, and let
$k \xrightarrow{r} k$ be the set of the random walks from $k$ to $k$ whose internal vertices are all less than $k$. As we will show later, the probabilities and the expectations in (5) and (6) can be easily calculated from the probabilities of these group of random walks. Moreover, if we assume the probabilities are computed iteratively for $k = 1, 2, \ldots, n$, then any random walk in $k \xrightarrow{r} k$ can be decomposed into at most $k$ random walks whose probabilities are already known, totally eliminating any computation that may need unbounded time.

To compose a longer random walk from shorter ones, let $W$ and $U$ be two sets of random walks and assume that all walks in $W$ stop at and all walks in $U$ start from the same vertex. We define $W + U$ to be the set of the random walks obtained by concatenating every pair of the random walks, one from $W$ and the other from $U$.

For a random walk $r \in k \xrightarrow{r} i$ for $1 \leq k < i \leq n+1$. If $r$ passes $k$ for more than once, i.e., $r \notin k \xrightarrow{r} i$, then we can uniquely decompose $r$ into two walks $r_1$ and $r_2$ such that $r_1 \in k \xrightarrow{r} k$ and $r_2 \in k \xrightarrow{r} i$ by identifying the first time that $r$ returns to $k$. For example, the random walk $r = 3 \to 1 \to 3 \to 5$ in Fig. 1(a) is decomposed as $r_1 = 3 \to 1 \to 3$ and $r_2 = 3 \to 5$. Therefore

$$k \xrightarrow{r} i = k \xrightarrow{r} i \cup (k \xrightarrow{r} k + k \xrightarrow{r} i) \quad \forall 1 \leq k < i \leq n+1. \quad (7)$$

Such decomposition leads to the following two lemmas.

Lemma 1: $\forall 1 \leq k < i \leq n+1$, $\Pr[k \xrightarrow{r} i] = \Pr[k \xrightarrow{r} i] / [1 - \Pr[k \xrightarrow{r} k]$. Proof: Taking the probabilities of the random walks at both sides of (7), we have

$$\Pr[k \xrightarrow{r} i] = \Pr[k \xrightarrow{r} i] + \Pr[k \xrightarrow{r} k] \Pr[k \xrightarrow{r} i]$$

which can be rearranged as

$$\Pr[k \xrightarrow{r} i] / [1 - \Pr[k \xrightarrow{r} k]) = \Pr[k \xrightarrow{r} i].$$

Therefore, it is sufficient to prove $1 - \Pr[k \xrightarrow{r} k] > 0$.

Since $A$ is symmetric nonsingular, $n+1$ is reachable from $k$. This implies that, first, any random walk from $k$ either returns to $k$ or reaches a vertex $i > k$ eventually, i.e., $\sum_{i=k+1}^{n+1} \Pr[k \xrightarrow{r} i]$ and second, $\Pr[k \xrightarrow{r} n+1] > 0$. As all the probabilities should be nonnegative, we have

$$1 - \Pr[k \xrightarrow{r} k] = \sum_{i=k+1}^{n+1} \Pr[k \xrightarrow{r} i] \geq \Pr[k \xrightarrow{r} n+1] > 0$$

and thus the lemma holds. $\blacksquare$

Lemma 2: $\forall 1 \leq k \leq n$, $E_k = 1 / [1 - \Pr[k \xrightarrow{r} k]$. Proof: Taking a union at both sides of (7) for $i = k + 1$, $k + 2, \ldots, n+1$, we have

$$\bigcup_{i=k+1}^{n+1} k \xrightarrow{r} i = \left( \bigcup_{i=k+1}^{n+1} k \xrightarrow{r} i \right) \cup \left( k \xrightarrow{r} k + k \xrightarrow{r} i \right). \quad (8)$$

Recall that $E_k$ is the expectation of the number of times that a random walk belonging to the set at the LHS of (8) passes $k$.

Thus, we can calculate $E_k$ based on the decomposition at the RHS of (8).

For the set $\bigcup_{i=k+1}^{n+1} k \xrightarrow{r} i$, each random walk will pass $k$ exactly once, i.e., the conditional expectation is $1$. On the other hand, for the set $k \xrightarrow{r} k + \bigcup_{i=k+1}^{n+1} k \xrightarrow{r} i$, the conditional expectation is $1 + E_k$. Therefore, (8) implies that

$$E_k = 1 + \sum_{i=k+1}^{n+1} \Pr[k \xrightarrow{r} i] + (1 + E_k) \Pr[k \xrightarrow{r} k].$$

The lemma thus holds since $\sum_{i=k+1}^{n+1} \Pr[k \xrightarrow{r} i] = 1 - \Pr[k \xrightarrow{r} k] > 0$, as shown in the proof of Lemma 1. $\blacksquare$

Lemma 1 and 2, together with (5) and (6), show that one can factor $A$ by computing $\Pr[k \xrightarrow{r} i]$ for all $1 \leq k \leq i \leq n$, though further decompositions are still necessary to eliminate Monte Carlo simulations.

For a random walk in $\bigcup_{i=k+1}^{n+1} k \xrightarrow{r} i$ and a positive integer $\omega$ that is no more than the number of the edges in the random walk, we define its $\omega$-step prefix to be the path consisting its first $\omega$ edges. It is clear by such a definition a random walk could be its own prefix but cannot be a prefix of other random walks. While many random walks can share the same prefix, the probability for a random walk to have a particular prefix equals to the product of the transition probabilities along the edges of the prefix. Consider a set of prefixes $N_k$ such that any random walk in $\bigcup_{i=k+1}^{n+1} k \xrightarrow{r} i$ has exactly one prefix belonging to $N_k$. Obviously, such set $N_k$ exists—for example, $N_1$ could be the set of all the one-step prefixes, i.e., all the one-step walks starting from $k$. Let $p$ be an $(n+1) \times 1$ vector where $p_j$ is the probability for the random walk to have a prefix in $N_k$ ending at the vertex $j$ for $1 \leq j \leq n+1$. For example, for the above choice of $N_3$ as all the one-step prefixes (3 \to 1, 3 \to 5) in Fig. 1(a), we have $p = (1/2, 0, 0, 0, 1/2)$.

For $k \leq i \leq n+1$, a random walk $r$ in $k \xrightarrow{r} i$ that is not itself a prefix in $N_k$ can be uniquely decomposed into a prefix $r_1 \in N_k$ and a random walk $r_2$, where $r_1$ ends at and $r_2$ starts from some vertex $j < k \leq i$. For example, for the aforementioned choice of $N_3$, the random walk $r = 3 \to 1 \to 2 \to 1 \to 2 \to 1 \to 4$ in Fig. 1(a) is decomposed into $r_1 = 3 \to 1$ and $r_2 = 1 \to 2 \to 1 \to 2 \to 1 \to 4$. While the probability for the random walks to have a prefix in $N_k$ ending at $j$ can be obtained from $p$ as $p_j$, the difficulty here is to compute the probability of all such $r_2$’s. Our idea is to further decompose $r_2$ into the random walks whose probabilities are known from previous iterations.

Since $r_2$ starts from $j$ and stops at $i > j$, we may consider the first time that $r_2$ reaches a vertex $j_2 > j$. If $j_2 \neq i$, then it must be an internal vertex on $r_2$. Therefore, $j_2 < k \leq i$ and we may consider the first time that $r_2$ reaches a vertex $j_3 > j_2$. Note that $r_2$ must reach $j_3$ before $j_2$ since any internal vertices on $r_2$ between $j$ and $j_2$ must be less than $j$ and thus be less than $j_2$. Clearly, we may continue such reasoning until $r_2$ reaches its last vertex $i$. Formally, there exists certain $1 \leq m < k$ and $j = j_1 < j_2 < \cdots < j_m < k$, such that we can decompose $r_2$ into $m$ walks, one each from $j_1 \xrightarrow{r_1} j_2$, $j_2 \xrightarrow{r_2} j_3$, $j_{m-1} \xrightarrow{r_{m-1}} j_m$, and $j_m \xrightarrow{r_m} i$. For example, for $k = 3$ and $i = 4$, the random walk $r_2 = 1 \to 2 \to 1 \to 2 \to 1 \to 4$...
in Fig. 1(a) is decomposed into \( m = 2 \) random walks \( 1 \to 2 \) and \( 2 \to 1 \to 2 \to 1 \to 4 \) with \( j_1 = 1 \) and \( j_2 = 2 \). This decomposition leads to the following lemma.

**Lemma 3**: \( \forall 1 \leq k \leq n \) and \( k \leq i \leq n+1 \)

\[
\Pr[k \xrightarrow{c}\ i] = p_i - \hat{L}_{i,1:k-1} \hat{L}^{-1}_{1,k-1,1:k-1} p_{1:k-1}
\]

where we define \( \hat{L}_{n+1,k} = -\sum_{i=1}^{n} \hat{L}_{i,k} \).

**Proof**: For \( 1 \leq j < k \), let \( j \xrightarrow{c} i \) be the set of the random walks from \( j \) to \( i \) with its internal vertices, if there is any, are all less than \( k \). Based on the decomposition of \( r \) into \( r_1 \) and \( r_2 \) for \( r \notin N_k \), we have

\[
\Pr[k \xrightarrow{c}\ i] = p_i + \sum_{j=1}^{k-1} p_j \Pr[j \xrightarrow{c}\ i]. \tag{9}
\]

Based on the decomposition of \( r_2 \) into \( m \) random walks and (6), we have

\[
\sum_{j=1}^{k-1} p_j \Pr[j \xrightarrow{c}\ i] = \sum_{j=1}^{k-1} p_j \sum_{m=1}^{k-j_1} \sum_{j_1 < j_2 < \ldots < j_m < k} \left( \prod_{i=1}^{m-1} \Pr[j \xrightarrow{c}\ i] \right) \Pr[j_m \xrightarrow{c}\ i] \tag{10}
\]

\[
= \sum_{j=1}^{k-1} \sum_{m=1}^{k-j_1} \sum_{j_1 < j_2 < \ldots < j_m < k} \left( \prod_{i=1}^{m-1} (-\hat{L}_{i,j_1+j_2}) \right) \left( \prod_{i=1}^{m} (-\hat{L}_{i,j_1+j_2}) \right) p_{j_1} \tag{11}
\]

\[
= \sum_{m=1}^{k-1} \sum_{j_1 < j_2 < \ldots < j_m < k} \left( \prod_{i=1}^{m} (-\hat{L}_{i,j_1+j_2}) \right) p_{j_1} \tag{12}
\]

\[
= -\hat{L}_{i,1:k-1} \left( \sum_{m=1}^{k-1} \left( \prod_{i=1}^{m} (-\hat{L}_{i,1:k-1}) \right) p_{1:k-1} \right). \tag{13}
\]

Since \( I - \hat{L}_{1:k-1,1:k-1} \) is a \( (k-1) \times (k-1) \) lower triangular matrix whose diagonal elements are all \( 0 \), we have

\[
\sum_{m=1}^{k-1} \left( \prod_{i=1}^{m} (-\hat{L}_{i,1:k-1}) \right) = \hat{L}_{1:k-1,1:k-1}^{-1}
\]

and thus the lemma holds according to (9).

As an example of Lemmas 1–3, consider the aforementioned choice of \( \mathcal{N}_3 \) being the set of all the one-step walks starting from the vertex 3 in Fig. 1(a). Then

\[
\Pr[3 \xrightarrow{c}\ 4]
\]

\[
= p_4 + p_4 \Pr[1 \xrightarrow{c}\ 4] + p_2 \Pr[2 \xrightarrow{c}\ 4]
\]

\[
= p_4 + p_4 \Pr[1 \xrightarrow{c}\ 4] + p_4 \Pr[2 \xrightarrow{c}\ 4] + p_2 \Pr[2 \xrightarrow{c}\ 4]
\]

\[
= p_4 + \left( \frac{1}{3} \right) \left( \frac{1}{2} \right) \left( \frac{1}{3} \right) \left( \frac{1}{2} \right) = \frac{1}{4}.
\]

**Algorithm Deterministic Random Walk**

**Inputs**

\( A \) : the system matrix.

\( \mathcal{N}_k \) : sets of prefixes for \( 1 \leq k \leq n \).

\( f \) : dropping and compensation scheme.

**Outputs**

A diagonal matrix \( D \) and a unit lower triangular matrix \( L \).

1. For \( k = 1 \) to \( n \):
   2. Compute \( p \) from \( A \) and \( \mathcal{N}_k \).
   3. For \( i = k \) to \( n \):
      4. \( q_i \leftarrow p_i - L_{i,1:k-1} L_{1:k-1,1:k-1}^{-1} p_{1:k-1} \).
      5. \( d_{k,k} \leftarrow a_{k,k} (1 - q_k) \).
      6. \( L_{k+1:n,k} \leftarrow -f(q_k, q_{k+1}, \ldots, q_n) \).

Fig. 2. DRW algorithm.

Similarly

\[
\Pr[3 \xrightarrow{c}\ 3] = p_3 + \left( \Pr[1 \xrightarrow{c}\ 3] \Pr[2 \xrightarrow{c}\ 3] \right) \left( \frac{1}{3} \right) \left( \frac{1}{3} \right) \left( \frac{1}{3} \right) = \frac{1}{4}.
\]

We can apply the DRW algorithm to compute \( \Pr[3 \xrightarrow{c}\ 4] = \Pr[3 \xrightarrow{c}\ 4] / 1 - \Pr[3 \xrightarrow{c}\ 4] = 1/3 \) and \( E_3 = 1/3 - \Pr[3 \xrightarrow{c}\ 4] = 4/3 \).

Clearly, if one chooses the \( \mathcal{N}_k \) such that the vector \( p \) can be computed without performing Monte Carlo simulations, then Lemmas 1–3 eliminate the need to simulate not only the random walks of unbounded number of steps but all of them. That motivates us to call our proposed technique deterministic random walk.

**B. Proposed Preconditioner**

Based on (5) and (6), and Lemmas 1–3, we design the DRW algorithm that computes an approximated \( \text{LDL}^T \) factorization of \( A \). As shown in Fig. 2, the DRW algorithm depends on the choice of the prefix set \( \mathcal{N}_k \) and a function \( f \) that computes the column \( k \) from approximations of \( \Pr[k \xrightarrow{c}\ i] \) utilizing some dropping and compensation scheme. Note that we do not explicitly initialize \( L \) and \( D \) in the algorithm since one only need to store the nonzero off-diagonal elements in the lower triangle of \( L \) and the diagonal elements of \( D \), which will all be computed in the algorithm.

The For loop at line 3 can be reorganized to simplify both the presentation and the implementation, and to facilitate sparse matrix operations. Define \( L_{m+1,1:k-1} = -I^T \).

\( L_{1:k-1,1:k-1} \). Let \( b \) be an \((n+1) \times 1 \) vector with \( b_{1:k-1} = -I^T \).

\( q_{k+1} \) being computed as line 4, and \( q_{n+1} = p_{n+1} - L_{n+1,1:k-1} q_{1:k-1} \). We have

\[
q = \left( \begin{array}{c}
L_{1:k-1,1:k-1} & 0 \\
L_{k+1:n,k} & I
\end{array} \right) \left( \begin{array}{c}
p \end{array} \right).
\]

In other words, the For loop at line 3 can be replaced by a partial forward substitution on \( L \) up to the first \( k - 1 \) columns.
Since it is reasonable to assume \( p \) to be sparse and \( L \) is also sparse, such computation can be done very efficiently without visiting all the \( k-1 \) columns leveraging the algorithm proposed in [13]. To be more specific, one should first perform a depth-first search in the extended matrix graph of \( L_{1:k-1,1:k-1} \), starting from the vertices corresponding to the nonzero elements in \( p_{1:k-1} \) and stopping at \( k \), to obtain a topological ordering of the reachable vertices, and then follow such ordering to use the columns of \( L \).

On the other hand, the choice of \( f \) is essential for the preconditioner \( D \) and \( L \) to meet the conditions P1–P3. Before we present the requirements of \( f \), we establish the following three sets of invariants for the For loop at line 1.

**Lemma 4:** At line 3, if the first \( k-1 \) columns of \( L \) satisfy P1, then \( \sum_{i=k}^{n} q_i = 1 \), and \( q_i \geq p_i \geq 0 \), \( \forall 1 \leq i \leq n+1 \).

**Proof:** Consider the partial forward substitution in (10) that computes \( q \) from \( p \) using the first \( k-1 \) columns of \( L \). We may implement this computation as follows: first, \( q \) is initialized as \( p \); then in the \( j \)th iteration for \( j = 1, 2, \ldots, k-1 \), we increase \( q_i \) for \( i = j + 1, j + 2, \ldots, n+1 \) by \(-q_i h_j \).

Since \( L_{m+1,1:k-1} = -D^{-1}L_{1:m,1:k-1} \) by definition, it can be proved by induction that at the end of the \( j \)th iteration, we have \( \sum_{i=j+1}^{n+1} q_i = \sum_{i=1}^{n+1} p_i = 1 \). Therefore, at the end of the \((k-1)\)th iteration, we have \( \sum_{i=k}^{n+1} q_i = 1 \).

On the other hand, the condition P1 ensures that \( l_{i,j} \)s are nonpositive for all \( j < i \leq n+1 \). Therefore, since all the elements of \( p \) are nonnegative (as they are probabilities), by induction, all the elements of \( q \) will not decrease during the computation. Thus \( q_i \geq p_i \geq 0 \) for all \( 1 \leq i \leq n+1 \).

Intuitively, Lemma 4 indicates that the conditions P1 can be satisfied by requiring \( f \) to return a vector with any nonnegative elements that add up to no more than 1.

**Lemma 5:** At line 3, in addition to P1, if the first \( k-1 \) columns of \( L \) further satisfy P2, then \( q_i \neq 0 \) only if \( \hat{l}_{i,k} \neq 0 \) for \( k < i \leq n \).

**Proof:** Consider \( i \) and \( j \) satisfying \( 1 \leq j < k-1 \) and \( j < i \leq n \) such that \( l_{i,j} \neq 0 \). The condition P2 implies that \( \hat{l}_{i,j} \neq 0 \). According to (6), we must have that the set \( j \leq i \) is not empty.

On the other hand, based on the definition of \( N_k \), \( p_i \neq 0 \) implies that the set \( k \leq i \) is not empty for \( 1 \leq i \leq n \).

Therefore, similar to the proof of Lemma 4, it can be proved by induction that at the end of the \( j \)th iteration of the partial forward substitution for \( j = 1, 2, \ldots, k-1 \), \( q_i \neq 0 \) implies that the set \( k \leq i \) is not empty for \( j < i \leq n \). Thus at the end of the computation, \( q_i \neq 0 \) implies the set \( k \leq i \) is not empty for \( k < i \leq n \), and thus \( \hat{l}_{i,k} \neq 0 \) according to (6).

Note that it is possible to have \( q_{n+1} \neq 0 \) with an empty set \( k \leq n+1 \) for certain \( k \) due to the choice of \( f \) so that the above lemma will not necessarily hold for \( i = n+1 \). On the other hand, Lemma 5 indicates that the condition P2 can be satisfied by requiring \( f \) not to introduce additional nonzero elements.

To establish the last set of invariants, we define the height \( h_j \) of a vertex \( j \) to be the shortest distance from \( j \) to \( n+1 \) in \( G^+_f \), with each edge contributing a distance of 1. This definition is clearly consistent since \( n+1 \) is reachable from any other vertex. Line 5 implies that the condition P3 is equivalent to \( q_k < 1 \). According to Lemma 4, it is sufficient to require \( q_i > 0 \) for some \( k+1 \leq s \leq n+1 \).

**Lemma 6:** Assume at line 3, the first \( k-1 \) columns of \( L \) satisfy P1. If for any \( 1 \leq j < k \), there exists \( j < i \leq n+1 \) such that \( l_{i,j} \neq 0 \) and \( h_j > h_i \), then there exists \( k+1 \leq s \leq n+1 \) such that \( q_s > 0 \) and \( h_k > h_s \).

**Proof:** According to the definition of the height, we have \( h_k > 0 \). Moreover, there exists a shortest path from \( k \) to \( n+1 \) since \( n+1 \) is reachable from \( k \). The definition of \( N_k \) thus implies that there exists some \( j \) along the shortest path satisfying that \( p_{j} > 0 \) and \( h_k > h_j \). If \( j > k \), we have proved the lemma with \( s = j \) since \( q_j \geq p_j \) according to Lemma 4 and thus \( q_j > 0 \). Hence, we only consider the case \( j_1 < k \) as follows.

Since \( j_1 < k \), our assumptions in the lemma imply that there exists some \( j_2 \) satisfying that \( l_{j_2,j_1} \neq 0 \) and \( h_{j_2} > h_{j_1} \). Similar to the cases for \( j_1 \), we can either have \( j_2 > k \) or \( j_2 < k \). If \( j_2 < k \), we can rely on the same reasoning to obtain \( j_2 \) and so on, with the heights of the newly found vertices decreasing monotonically. As the height of a vertex must be an integer and nonnegative, this process must terminate eventually with a sequence of vertices, denoted by \( j_1, j_2, \ldots, j_m \) for certain \( m \), such that first, \( h_{j_1} > h_{j_2} > \cdots > h_{j_m} \); second, \( \prod_{i=1}^{m-1}(l_{j_{i+1},j_i}) \) are positive; and third, \( j_m = k \).

Now let \( s = j_m \), clearly \( k+1 \leq s \leq n+1 \) and \( h_k > h_s \). According to the proof of Lemma 4, we have \( q_s \geq p_j \prod_{i=1}^{m-1}(l_{j_{i+1},j_i}) \). As both \( p_j \) and \( \prod_{i=1}^{m-1}(l_{j_{i+1},j_i}) \) are positive, we must have \( q_s > 0 \).

In summary, the following theorem states the correctness of the DRW algorithm and defines a set of sufficient requirements for \( f \) that should satisfy to guarantee the conditions P1–P3.

**Theorem 1:** The DRW algorithm will always terminate. The conditions P1–P3 will be satisfied if \( f \) returns a vector satisfying the following requirements.

1. The elements are all nonnegative and their summation is no more than 1.
2. An element is nonzero only if the corresponding \( q_i \) is non-zero.
3. There is at least one nonzero element whose corresponding \( q_s \) satisfying that \( h_s < h_k \).

**Proof:** It is obvious that the DRW algorithm will terminate as long as \( f \) terminates.

The first requirement and Lemma 4 imply P1. The second requirement and Lemma 5 imply P2. Finally, the third requirement and Lemma 4 and 6 imply P3.

Our DRW algorithm shares some similarities with incomplete LDL\(^T\) factorization, e.g., both compute a column of \( L \) using the columns that are already computed. We emphasize the two algorithms are quite different, even if the function \( f \) in the DRW algorithm is chosen to be the same as the dropping scheme in (3). Obviously, both (2) and (3) can use not only \( L_{1:n,1:k-1} \) but also \( D_{1:k-1} \), though our DRW algorithm only depends on \( L_{1:n,1:k-1} \). Such difference make our algorithm not sensitive to the error in the \( D \) matrix. On the other hand, (3) indicates that the column \( k \) of \( L \) is computed from the columns of \( L \) corresponding to the nonzero elements.
in $L_{k,1:k-1}$. Although such columns should be the same as the columns corresponding to the aforementioned set of reachable vertices in our algorithm when the factorization is exact, they are different when some elements are dropped from $L$. We observed that our algorithm will usually locate more columns than incomplete LDL$^T$ factorization, which should appear in an exact factorization, resulting in a better approximation at the cost of more computation time. We believe that such cost is what one has to pay to achieve better quality via a more flexible dropping and compensation scheme.

IV. IMPLEMENTATION DETAILS

There are many ways to implement the various components of the DRW algorithm while still satisfying the requirements in Theorem 1. We present in this section our choices when implementing the DRW algorithm for power grid analysis, together with a system simplification (SS) approach that may reduce the solver running time.

A. Matrix Ordering

Although Theorem 1 does not impose any restriction on how the vertices should be ordered, we have observed that the matrix ordering does impact both the time to compute the preconditioner and its quality (thus the time to solve the problem iteratively). Our implementation may choose either the reverse Cuthill–McKee (RCM) ordering [9] to reduce the envelope of the matrices, or the approximate minimum degree (AMD) ordering [1] to reduce the number of fill-ins.

In general, we prefer AMD over RCM since we found in our experiments that it leads to DRW preconditioners with better quality in all cases. For systems with large bandwidth due to couplings between the vertices, e.g., for transient simulation, the AMD ordering also results in less time to compute the preconditioner, since there will be less computations due to less fill-ins, despite the extra time to obtain the ordering in comparison with RCM. For systems with narrow bandwidth and large number of levels, e.g., for dc analysis, the RCM ordering could be a better choice since it takes less time to compute the preconditioner and the preconditioner quality is comparable with that obtained using AMD.

B. Set $N_k$

We choose $N_k$ to be the set of all the one-step prefixes, i.e., the one-step random walks from $k$. The $p$ on line 2 of the DRW algorithm is computed directly from $A_{1:n,k}$, i.e., $p_j = a_{j,k}/a_{k,k}$ for $j \neq k$ and $p_k = 0$. Note that we store the whole matrix $A$ to make such computation simple instead of only storing half of $A$ exploiting that $A$ is symmetric.

Other choices of $N_k$ may improve the quality of the preconditioner at the cost of more running time to compute or to approximate $p$. In the extreme case, $N_k$ could include all the random walks themselves in $\bigcup_{i=1}^{n+1} k \leadsto i$. In such case, $p_j = 0$ for all $j > k$ and $p_j$ for $j \leq k$ may be approximated by Monte Carlo simulations, leading to a variant of the stochastic random walk preconditioner in [32] utilizing Lemmas 1 and 2. Practically, we have also experimented with the choice of $N_k$ that consists of all the one-step prefixes ending at a vertex larger than $k$, and all the two-step prefixes. The exact $p$ for this case can be computed via a breadth-first search on the extended matrix graph of $A$, though the improvements on the quality of the preconditioner and the overall running time are rather marginal. It remains an open problem that whether there exist certain choices of $N_k$ that could improve the performance of deterministic random walk preconditioning considerably, possibly for specific power grid structures.

C. Dropping and Compensation Scheme

We specify the dropping and compensation scheme in $f$ in two levels—globally and per column.

Globally, three parameters should be provided: a filling factor $\gamma$ specifying the desired ratio of the number of the nonzero off-diagonal elements in $L$ to that in $A$, a keep tolerance $\delta^+$ such that any element of $L$ whose absolute value is larger should always be kept, and a drop tolerance $\delta^-$ such that any element of $L$ whose absolute value is smaller should always be dropped. We do not actually tune these parameters in our experiments. The parameter $\gamma$ is used to constraint the size, and thus the memory usage, of the preconditioner. The parameter $\delta^+$ is always set to 0.05 as a safe guard in case the probabilities of the random walks are distributed evenly. The parameter $\delta^-$ is always set to 0.00005. Our experimental results are not sensitive to the values of $\delta^+$ and $\delta^-$ unless the former is too small or the latter is too large, e.g., 0.001.

Column-wise, when the column $k$ of $L$ should be computed using $f$, we first compute the overall budget of the nonzero off-diagonal elements on the remaining columns of $L$, i.e., $L_{1:n,k,n}$, from the number of the nonzero off-diagonal elements in $A$ and $L_{1:n,1:k-1}$ and the global filling factor $\gamma$. Then, we compute $\Gamma$, the per-column budget of the nonzero off-diagonal elements for the column $k$, as the overall budget divided by $n-k+1$. In other words, we assume that the remaining nonzero off-diagonal elements in $L$ should be distributed evenly across the remaining columns. If the computed $\Gamma$ is less than 2, we will increase it to 2. For the function $f$, we first compute $q_{n+1}$ as $1 - \sum_{i=k+1}^{n} q_i$ and then find the largest $q_i$ from $q_{k+1}, q_{k+2}, \ldots, q_{n+1}$ such that $h_k > h_s$, which is guaranteed by Lemma 6 to be nonzero. We replace all but the largest $\Gamma$ elements that are at least $\delta^-$ in $q_{k+1}, q_{k+2}, \ldots, q_n$ by 0. If any element being replaced is larger than $\delta^+$ or $q_k$ is replaced, we will replace it back. We found in our experiments that when we choose $q_k$ as the aforementioned largest one, in most cases it is one of the largest $\Gamma$ elements and no additional nonzero element is introduced to the returned vector. Let the resulting elements be $q'_{k+1}, q'_{k+2}, \ldots, q'_n$. The function $f$ will return a compensated vector as follows:

$$f(q_k, q_{k+1}, \ldots, q_n) = \frac{\sum_{i=k+1}^{n} q_i}{\sum_{i=k+1}^{n} q'_i} \left( \frac{q'_{k+1}}{1-q_k}, \frac{q'_{k+2}}{1-q_k}, \ldots, \frac{q'_n}{1-q_k} \right)$$

i.e., the dropped elements are added to the remaining ones proportionally. Note that if all of $q_{k+1}, q_{k+2}, \ldots, q_n$ are 0, $f$ will simply return 0.
Overall, our dropping scheme is an extension of [19], while the compensation scheme, though intuitive, cannot be applied directly there.

D. System Simplification via Partial Exact Factorization

For public IBM power grid benchmarks [24], [25], especially those for transient simulation, there are rows and columns with no more than 3 nonzero off-diagonal elements in the system matrix $A$. As $A$ could be reordered such that many of these columns can be factored exactly without introducing additional fill-ins, we may exclude them from the iterative solver to reduce the running time via a preprocessing step that computes a Schur complement in $A$ by partial exact factorization. A postprocessing step will recover the original solution without affecting its accuracy. Overall, our SS approach could be treated as a combination of the direct and iterative solvers without affecting its accuracy. Overall, our SS approach could be treated as a combination of the direct and iterative solvers without affecting its accuracy.

We first validated our implementation using the public IBM benchmarks. For comparison, we implemented in our solver the incomplete LDLT factorization preconditioner as described in Section II-B using the data structure proposed by [19]. The same dropping scheme as our DRW algorithm is applied without compensation. Moreover, we integrated in our solver the binary library for stochastic random walk [32] to generate the preconditioners. Note that the library used some internal matrix ordering we cannot modify and that we did not use their library to actually solve the equations since we found their solver to be always slower than ours. In addition, the AMD package [1] was used to generate the AMD ordering and the CHOLMOD package [7] was employed to provide comparisons between the iterative and direct solvers. All programs were built with GCC version 4.3 and we ran all the experiments on a 64-bit Linux workstation with a 2.4-GHz Intel Q6600 processor and 8-GB memory. Note that only one core was used for all the experiments.

We first validated our implementation using the public IBM power grid benchmarks [24], [25], [29] against their published golden solutions. The results of the largest six circuits for dc analysis are reported in Table I and a comparison to the state-of-the-art AMG–PCG solver PowerRush [42] is provided. Similar to [42], we preprocessed the SPICE netlist to treat resistances less than $10^{-6} \Omega$ as shorts. To reduce the time to compute the DRW preconditioner, the RCM ordering was used and $\gamma$ was set to 1. The PCG iterations were terminated

<table>
<thead>
<tr>
<th>name</th>
<th>DRW+RCM</th>
<th>PowerRush [42]</th>
</tr>
</thead>
<tbody>
<tr>
<td>statistics</td>
<td>name</td>
<td>n</td>
</tr>
<tr>
<td>ibngpg1</td>
<td>283</td>
<td>311</td>
</tr>
<tr>
<td>ibngpg4</td>
<td>953</td>
<td>4.05</td>
</tr>
<tr>
<td>ibngpg5</td>
<td>540</td>
<td>2.69</td>
</tr>
<tr>
<td>ibngpg6</td>
<td>834</td>
<td>4.13</td>
</tr>
<tr>
<td>ibngpgew1</td>
<td>531</td>
<td>2.92</td>
</tr>
<tr>
<td>ibngpgew2</td>
<td>531</td>
<td>2.92</td>
</tr>
<tr>
<td>total</td>
<td>18.21</td>
<td>18.20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>n</th>
<th>ny</th>
<th>x</th>
<th>nyy</th>
<th>z</th>
<th>x</th>
<th>nyy</th>
<th>z</th>
<th>t err (s)</th>
<th>t err (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ibngpg1</td>
<td>311</td>
<td>104</td>
<td>52</td>
<td>103</td>
<td>0.97</td>
<td>192</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ibngpg4</td>
<td>4.05</td>
<td>5/1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ibngpg5</td>
<td>2.69</td>
<td>6/1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ibngpg6</td>
<td>4.13</td>
<td>6/1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ibngpgew1</td>
<td>2.92</td>
<td>9/1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ibngpgew2</td>
<td>2.92</td>
<td>9/1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>18.20</td>
<td>16.97</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To address the second concern, we adopt an iterative greedy heuristic to reorder $A$ to maximize $m$, and to compute $S_m$, $\hat{D}_{1:m,1:m}$, and $\hat{L}_{1:m,1:m}$ simultaneously during the reordering. Initially, we set $m = 0$ and $S_m = A$. For each iteration, we choose an arbitrary column of $S_m$ that has the minimum number of off-diagonal nonzeros. We terminate the process if the number of the off-diagonal nonzeros are larger than 3 or if $m = n$. Otherwise, we reorder $S_m$ (and thus $A$) such that the chosen column will be the next to be eliminated. Note that since to eliminate a column with 3 off-diagonal nonzeros may possibly introduce fill-ins to other columns with 3 off-diagonal nonzeros, this heuristic does not necessarily lead to the maximum $m$ across all the possible reorderings of $A$, though our experimental results show considerable reduction can still be achieved for aforementioned transient simulation benchmarks.

V. Experiments

A. Experimental Setup and Result Validation

We implemented our DRW algorithm and a PCG solver in C++ to solve power grid analysis problems. For comparison, we implemented in our solver the incomplete LDLT factorization preconditioner as described in Section II-B using the data structure proposed by [19]. The same dropping scheme as our DRW algorithm is applied without compensation. Moreover, we integrated in our solver the binary library for stochastic random walk [32] to generate the preconditioners. Note that the library used some internal matrix ordering we cannot modify and that we did not use their library to actually solve the equations since we found their solver to be always slower than ours. In addition, the AMD package [1] was used to generate the AMD ordering and the CHOLMOD package [7] was employed to provide comparisons between the iterative and direct solvers. All programs were built with GCC version 4.3 and we ran all the experiments on a 64-bit Linux workstation with a 2.4-GHz Intel Q6600 processor and 8-GB memory. Note that only one core was used for all the experiments.

We first validated our implementation using the public IBM power grid benchmarks [24], [25], [29] against their published golden solutions. The results of the largest six circuits for dc analysis are reported in Table I and a comparison to the state-of-the-art AMG–PCG solver PowerRush [42] is provided. Similar to [42], we preprocessed the SPICE netlist to treat resistances less than $10^{-6} \Omega$ as shorts. To reduce the time to compute the DRW preconditioner, the RCM ordering was used and $\gamma$ was set to 1. The PCG iterations were terminated

<table>
<thead>
<tr>
<th>name</th>
<th>DRW+RCM</th>
<th>PowerRush [42]</th>
</tr>
</thead>
<tbody>
<tr>
<td>statistics</td>
<td>name</td>
<td>n</td>
</tr>
<tr>
<td>ibngpg1</td>
<td>283</td>
<td>311</td>
</tr>
<tr>
<td>ibngpg4</td>
<td>953</td>
<td>4.05</td>
</tr>
<tr>
<td>ibngpg5</td>
<td>540</td>
<td>2.69</td>
</tr>
<tr>
<td>ibngpg6</td>
<td>834</td>
<td>4.13</td>
</tr>
<tr>
<td>ibngpgew1</td>
<td>531</td>
<td>2.92</td>
</tr>
<tr>
<td>ibngpgew2</td>
<td>531</td>
<td>2.92</td>
</tr>
<tr>
<td>total</td>
<td>18.21</td>
<td>18.20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>n</th>
<th>ny</th>
<th>x</th>
<th>nyy</th>
<th>z</th>
<th>x</th>
<th>nyy</th>
<th>z</th>
<th>t err (s)</th>
<th>t err (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ibngpg1</td>
<td>311</td>
<td>104</td>
<td>52</td>
<td>103</td>
<td>0.97</td>
<td>192</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ibngpg4</td>
<td>4.05</td>
<td>5/1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ibngpg5</td>
<td>2.69</td>
<td>6/1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ibngpg6</td>
<td>4.13</td>
<td>6/1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ibngpgew1</td>
<td>2.92</td>
<td>9/1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ibngpgew2</td>
<td>2.92</td>
<td>9/1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>18.20</td>
<td>16.97</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
when the $l_2$ norm of the residue is reduced to $10^{-3}$ of its original value. In Table I, we first show the dimension of $A$ ($n$) and the number of nonzero elements in $A$ ($nnz_A$). Then, for each solver, we show the total solving time in seconds ($t_{sol}$) and the maximum/average errors to the golden solution in $\text{mV}$ (Err.). For our solver, the column $t_{sol}$ includes the time to reorder $A$, to compute the preconditioner, to perform PCG iterations, and to compute the maximum/average errors. For PowerRush, the two columns were obtained from [42, Table 1] and we compensated for the different processors used for the experiments by scaling down the total solving time using a factor of 2.13/2.40. Although different processor microarchitectures may also affect the solving times, we believe that the comparison is reasonably fair so that we can conclude our PCG solver with DRW preconditioning can generate results with better accuracy in less time than PowerRush. For our DRW preconditioner, we further showed the number of nonzero elements in $L$ (nnz$L$) that can be used to estimate the memory usage. Since to store both $L$ and $D$ requires $12\text{nnz}_L$ bytes of memory for 4-bit integer indices and 8-bit floating point values, the maximum memory usage for our DRW preconditioner was less than 50 MB for all the dc analysis benchmarks.

The results of the six circuits for transient simulation are reported in Table II. Each SPICE netlist was first processed to generate two system matrices—one for dc operating point computation and the other for each discretized time step via the trapezoidal rule. We only report the later in the table as it dominated the memory usage and the running time. The SS technique as mentioned in Section IV was applied to simplify both system matrices. Our DRW preconditioners were then computed after the AMD ordering for two settings of $\gamma$. The number of nonzero elements in the partial exact factorization and the preconditioner are collectively reported under the columns nnz$L$. The maximum memory usage for our DRW preconditioner was thus less than 150 MB for all the transient simulation benchmarks. Since the trapezoidal rule was applied for discretization, we used the solution that is two steps back as the initial guess for the current time step and terminated the PCG iterations when the $l_2$ norm of the residue was reduced to $10^{-3}$ of its original value. The columns Err. confirm such choice as the maximum/average errors were close to those obtained by the direct solvers [40], [43]. The total running time ($t_{tot}$) includes everything from reading the SPICE netlist to writing the simulation outputs. It can be seen that our DRW preconditioner is able to explore the tradeoffs between memory usage and solution time—the later can be reduce by 33% with less than 50% memory overheads for the preconditioner.

Overall, Table II showed that our solver is able to complete all the six circuits for transient simulation within 1 h with reasonable amount of memory overhead. It is known that direct solvers are the most efficient for transient simulation when there is enough memory and all the winners [40], [43], [44] of the recent TAU Power Grid Simulation Contest [24] were based on direct solvers. The results shown later in Table IV imply a similar running time and memory overhead as reported in the above publications—on our workstation, all the transient simulations would require a total running time of 20 min using a solver based on CHOLMOD [7], which will be $3 \times$ faster than our solver, but need more than $5 \times$ fill-ins. On the other hand, in comparison with the solvers based on stochastic random walk [32] and incomplete $LDL^T$ factorization, Table IV implies that they will be at least $2 \times$ slower than our solver with the same number of fill-ins.

### B. Detailed Comparison

We then provide a detailed comparison of our DRW preconditioner and the other two preconditioners, i.e., stochastic random walk [32] and incomplete $LDL^T$ factorization, for the two sets of the benchmarks, respectively. For all the benchmarks, the PCG solver solved the same 100 random vectors for all the preconditioners and was terminated when the $l_2$ norm of the residue was reduced to $10^{-6}$ of its original value, or for a maximum of 200 iterations.

The results for the dc analysis benchmarks are shown in Table III. For both of our DRW preconditioner and incomplete $LDL^T$ factorization, we used the same RCM ordering and set $\gamma$ to 1.7 to match the total number of nonzero elements in $L$ ($\text{nnz}_L$) across all the benchmarks. In comparison to our DRW preconditioner, it can be seen that incomplete $LDL^T$ factorization spent very little time to compute the preconditioner ($t_{pre}$) but required about $1.7 \times$ PCG iterations ($\#\text{it.}$) and about $1.7 \times$ time to solve the problems in the PCG solver ($t_{pcg}$). The stochastic random walk was $3.7 \times$ slower to compute the preconditioner but required 20% less PCG iterations. However, it was $2.2 \times$ slower to actually solve the problems, due to its internal ordering that may incur considerable amount of overhead than the RCM ordering for sparse matrix–vector multiplications. Note that when the SS technique, as mentioned in Section IV, and the AMD ordering were applied for the dc analysis benchmarks, the results for our DRW preconditioner were slightly better, though those for the other two were slightly worse.

The results for the transient simulation benchmarks are shown in Table IV. The system matrix was first obtained via the trapezoidal rule. We then applied the SS technique since we found it to be effective for all the three preconditioner. For both of our DRW preconditioner and incomplete $LDL^T$ factorization, we used the same the AMD ordering [1] since we found that it led to less time to compute the preconditioner and better convergence. Moreover, we set $\gamma$ to 1.7 for both
of them to match the total number of nonzero elements in the partial exact factorization and the preconditioner ($nnz_L$). It can be seen that the performance of incomplete LDLᵀ factorization degraded for this set of benchmarks—in comparison to our DRW preconditioner, it used 40% time to compute the preconditioner but required about $3 \times$ PCG iterations and about $3 \times$ time to solve the problems, with certain runs not able to converge in 200 iterations. The stochastic random walk was $6.8 \times$ slower to compute the preconditioner and $2.4 \times$ slower to solve the problems, though requiring almost the same number of PCG iterations.

We further report the results of the direct solver CHOLMOD [7] in both tables. Clearly, although the time to solve the problem by a pair of forward/back substitution ($t_{fbs}$) was much smaller than our iterative solver using the DRW preconditioner, there were $8 \times$ more fill-ins in the exact factors and the time to compute them ($t_{fac}$) was more than that to compute our DRW preconditioners, making our technique a better choice if the exact factors cannot be held in memory or if the system is only solved for a few times.

VI. CONCLUSION

In this paper, we presented the deterministic random walk preconditioning technique for power grid analysis via PCG solvers. Compared to stochastic random walk preconditioning, our proposed algorithm computed the preconditioners in a deterministic manner to reduce computation time. Compared with incomplete LDLᵀ factorization, our proposed algorithm leveraged a compensation scheme to improve preconditioner quality, while maintaining correctness. Solvers built on top of our proposed preconditioner were able to outperform other state-of-the-art iterative solvers for dc analysis and transient simulation.

REFERENCES


