Structures 6 (2016) 170–181

Contents lists available at ScienceDirect

Structures

journal homepage: www.elsevier.com/locate/structures

A general characterization of the Hardy Cross method as sequential and multiprocess algorithms

John Baugh *, Shu Liu

Department of Civil, Construction, and Environmental Engineering, North Carolina State University, Raleigh, NC, USA

ARTICLE INFO

Article history:
Received 18 March 2016
Accepted 22 March 2016
Available online 9 April 2016

Keywords:
Hardy Cross method
Moment distribution
Convergence
Algorithms
Concurrency

ABSTRACT

The Hardy Cross method of moment distribution admits, for any problem, an entire family of distribution sequences. Intuitively, the method involves clamping the joints of beams and frames against rotation and balancing moments iteratively, whether consecutively, simultaneously, or in some combination of the two. We present common versions of the moment distribution algorithm and generalizations of them as both sequential and multiprocess algorithms, with the latter exhibiting the full range of asynchronous behavior allowed by the method. We prove, in the limit, that processes so defined converge to the same unique solution regardless of the distribution sequence or interleaving of steps. In defining the algorithms, we avoid overspecifying the order of computation initially using a sequential, nondeterministic process, and then more generally using concurrent processes.

© 2016 The Institution of Structural Engineers. Published by Elsevier Ltd. All rights reserved.

1. Introduction

Moment distribution is a well-known iterative technique for analyzing statically indeterminate beams and frames [1,2]. The method works by “clamping” joints, applying external loads, and then successively releasing them, allowing them to rotate, and reclamping them. Each time, the internal moments at the joints are distributed based on the relative stiffnesses of the adjoining members. The method converges under a variety of distribution sequences, e.g., varying the order in which joints are unclamped. In addition, there is inherent concurrency in the method—and hence internal nondeterminism—since moments can be distributed simultaneously and summed.

The method was first published in 1930 by Professor Hardy Cross, years after having taught it to his students at the University of Illinois [12]. The calculations can easily be performed by hand, and the rapid convergence of the method in practice made it possible for engineers to estimate end moments in just a few iterations. Although the method has largely been superseded by the convenience and availability of more general computational approaches, for decades it was the primary tool used to analyze reinforced concrete structures [5].

Conceived before the advent of computers, the Hardy Cross method nevertheless displays features that are interesting from a computational point of view. Its conventional, tabular layout suggests inherent parallelism in the method, and hence internal nondeterminacy, since moments can be distributed in different orders or even simultaneously. In this paper we offer a general characterization of the method as algorithms that are externally deterministic—in the limit the same input produces the same result—but internally the operations can be performed in any number of different ways. The manner by which algorithms can be so expressed to avoid overspecifying behavior, and yet also sufficiently constrained to ensure correctness, motivates the presentation and results that follow. Other aspects like data representation and the expression of concurrency share features with more complex domain decomposition approaches and element-by-element solvers used in finite element analysis, making the Hardy Cross method an attractive vehicle for their exploration in the classroom.

In addition to its computational aspects, it should be noted that only relatively recently have convergence proofs been published for the method, and only for its two most common, and fixed, distribution sequences. As the name implies, the simultaneous joint balancing approach balances all non-fixed joints at the same time, and then records carry-over moments simultaneously. In 2002, Volokh [18] characterized the Hardy Cross method as an incremental form of the Jacobi iterative method [7], in which calculations of the current iteration use only those from the prior iteration, and none from the current one. The equations on which his procedure operates are derived from the displacement method, and convergence guaranteed, he argues, since the coefficient matrix corresponds to stiffness and is therefore diagonally dominant.

Not addressed by Volokh is the consecutive joint balancing approach, in the terminology of Gere [6], which corresponds more directly to the intuitive idea of physically releasing and clamping joints in turn. Based on this version, Guo [8] characterized the Hardy Cross method in 1987 as an incremental form of the Gauss–Seidel method [7], which uses the most recently updated estimates, including those in the current iteration. Like Volokh, and apparently unknown to him, Guo starts with the classical displacement method of structural analysis

* Corresponding author.
E-mail address: jwb@ncsu.edu (J. Baugh).

http://dx.doi.org/10.1016/j.istruc.2016.03.004
2352-0124/© 2016 The Institution of Structural Engineers. Published by Elsevier Ltd. All rights reserved.

to derive his system of equations. He then argues that this form converges due to the positive definiteness of the coefficient matrix.

In contrast with the work of both Volokh and Guo, our results show that the Hardy Cross method is neither purely a Jacobi-like nor a Gauss–Seidel-like iteration. Instead, it can be presented in a general form that encompasses those as well as other joint balancing approaches. That Cross himself viewed the method as being flexible in its application is clear from his 1932 publication [2], where he describes variations that allow for "abbreviated computations" using the method, that accommodate structures with several conditions of loading, and so on. Another contrast with Volokh and Guo is that, instead of beginning with the displacement method, we work directly with the Hardy Cross method itself to construct corresponding iteration matrices. Proofs are then performed by focusing on the mathematical properties of those matrices without resorting to physical or structural analogies.

In the sections that follow, we introduce an approach for representing continuous beams and frames, and use it to define algorithms for both consecutive and simultaneous joint balancing. We then present more general characterizations of the Hardy Cross method as a) a sequential, nondeterministic algorithm, and b) a multiprocess algorithm, and formalize them by showing their relationships to matrix forms. The series of iteration matrices produced by the algorithms are then shown to be equivalent.

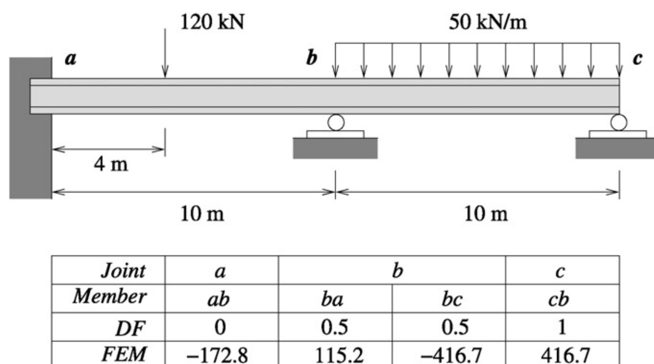
2. Problem representation and basic algorithms

"The idea behind digital computers may be explained by saying that these machines are intended to carry out any operations which could be done by a human computer." –Alan Turing [17]

Professor Cross defined moment distribution as a hand technique well before Turing's seminal work and the modern notion of an algorithm. Nevertheless, it is worth looking at the method as such and in a modern context. Here, we formalize what that means and define the conditions under which the method converges.

First, though, a note is in order about syntax of the pseudocode that follows. We include the elements of conventional imperative programming languages, along with common mathematical structures like sets for convenience. In the presentation, a hash symbol (#) comments to the end of line. For structuring data, compound items are represented as objects with attributes. We employ the convention adopted by many object-oriented languages, that of a dot followed by an attribute name. For instance, attribute a of object x can be referred to as $x.a$, and it could be assigned a value of 5 as so: $x.a=5$. For corresponding implementations of the algorithms in the Python programming language [14], please see Appendix A.

To see how problems can be represented for processing, we start with the continuous beam shown in Fig. 1, which we refer to as our model problem. Each end has a distribution factor (*DF*) and a fixed-end moment from the applied load (*FEM*), a carry-over factor of 0.5, and EI is constant. Recalling that a graph can be defined as $G=(V,E)$, with vertex set $G.V$ and edge set $G.E$, we define the directed graph in Fig. 2 to maintain the relevant beam properties.



Joint	a	b		c
Member	*ab*	*ba*	*bc*	*cb*
DF	0	0.5	0.5	1
FEM	−172.8	115.2	−416.7	416.7

Fig. 1. Continuous beam with loading conditions and corresponding fixed-end moments.

We denote by $e=(u,v)$ and $\overline{e}=(v,u)$ a pair of antiparallel directed edges. Each member has two member ends, with edge (u,v) defined to be the member end associated with joint u. We define for each edge $e \in G.E$ attributes (d,c,m), where $e.d$ is the distribution factor, $e.c$ is the carry-over factor, and $e.m$ is the moment (initially defined to be the fixed-end moment). For the model problem, for instance, we have

$$(a,b).m=-172.8$$

Since the carry-over factors are all 0.5, for all member ends e we have $e.c=0.5$.

We define the notion of a member end set associated with each joint, i.e., the set of member ends incident from joint u:

$$\text{ends}(G,u) \triangleq \{(w,v) | (w,v) \in G.E \wedge w=u\}$$

so for the model problem we have

$$\text{ends}(G,b)=\{(b,a),(b,c)\}$$

For any joint u we can find its unbalanced moment by summing the moments of its member ends:

$$\text{moment}(G,u) \triangleq \sum \{e.m | e \in \text{ends}(G,u)\}$$

A non-fixed joint is eligible for unclamping if it has member ends with positive distribution factors. We define a function active of joints that are eligible:

$$\text{active}(G) \triangleq \{u | u \in G.V \wedge \exists e.(e \in \text{ends}(G,u) \wedge e.d>0)\}$$

For instance, for joint a we have only a single member end with $(a,b).d=0$, so the only active joints are b and c:

$$\text{active}(G)=\{b,c\}$$

A basic step in the moment distribution method is that of unclamping, or releasing a joint, and distributing its unbalanced moment and the associated carry-over amounts. The algorithm Release is shown in Fig. 3, where $-=$ is the infix decrement-by operator and $\times$ is scalar multiplication. It operates on a joint u and distributes an unbalanced moment x by updating its own moments on edges (u,v) and, for carry-over amounts, edges (v,u).

We are now able to find a solution by repeatedly releasing eligible joints until a desirable level of convergence has been obtained:

$$\text{Release}(G,b,\text{moment}(G,b))$$

$$\text{Release}(G,c,\text{moment}(G,c))$$

$$\text{Release}(G,b,\text{moment}(G,b))$$

$$\text{Release}(G,c,\text{moment}(G,c))$$

…

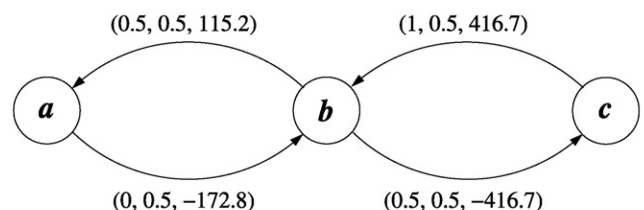


Fig. 2. Structure represented as a directed graph.

```
1  RELEASE(G, u, x)
2      for each edge (u, v) ∈ G.E
3          (u, v).m −= (u, v).d × x
4          (v, u).m −= (u, v).c × (u, v).d × x
```

Fig. 3. Algorithm for distributing at joint u an unbalanced moment x.

Such a process corresponds to a consecutive joint balancing scheme, because the moments distributed in each call to RELEASE draw on the most recently updated member end moments. When converged, the solution is available in the edge attribute $(u,v).m$ for all $(u,v) \in G.E$.

2.1. Consecutive joint balancing

Putting consecutive releases into an iterative algorithm that tests for convergence, as above, is straightforward. A consecutive joint balancing algorithm, SOLVE-CON, is shown in Fig. 4. It is defined so that in one cycle, each active joint is released and therefore balanced once, and the corresponding carry-over amount is recorded immediately afterward, yielding a Gauss–Seidel-like iteration. A tolerance *tol* is used to determine convergence. Once convergence is reached, as before, the solution is available in the edge attribute $(u,v).m$ for all $(u,v) \in G.E$. Note that, because the order in which joints are released is left undefined, the algorithm displays a minor degree of internal nondeterminism, though this leaves convergence unaffected [6].

2.2. Simultaneous joint balancing

Cross's initial approach using simultaneous balancing requires that moments be "read" and cached, like the Jacobi iterative method, which performs a cycle of updates exclusively from values obtained in the prior cycle. The simultaneous joint balancing algorithm SOLVE-SIM, shown in Fig. 5, is thus defined so that member end moments calculated in one cycle are used to determine moments in the next cycle. To do so, we make use of an attribute (m) of vertex objects to store unbalanced moments for each joint (lines 5 and 6), allowing updates to be simultaneously performed en masse (line 10). Again, a tolerance *tol* is used to determine convergence, and on completion the solution is available in the edge attribute $(u,v).m$ for all $(u,v) \in G.E$.

A characteristic of both algorithms, above, is that they complete one cycle before moving to the next. Joints are released either simultaneously or one by one until all have been released, completing a cycle and allowing the next round of releases to begin, though this is not strictly required for convergence [6]. As we show in the following sections, further abstraction of the consecutive and simultaneous joint balancing algorithms is possible, resulting in a more general characterization of the Hardy Cross method.

```
1  SOLVE-CON(G, tol)
2      converged = FALSE
3      while not converged
4          converged = TRUE
5          for each vertex v ∈ active(G)
6              if |moment(G, v)| > tol
7                  converged = FALSE
8                  RELEASE(G, v, moment(G, v))
```

Fig. 4. Moment distribution with consecutive joint balancing.

```
1  SOLVE-SIM(G, tol)
2      converged = FALSE
3      while not converged
4          converged = TRUE
5          for each vertex v ∈ active(G)
6              v.m = moment(G, v)
7          for each vertex v ∈ active(G)
8              if |v.m| > tol
9                  converged = FALSE
10                 RELEASE(G, v, v.m)
```

Fig. 5. Moment distribution with simultaneous joint balancing.

3. Abstraction of the basic algorithms

"Nondeterminism plays an important role in the specification of systems, since it enables underspecification, providing some flexibility to the implementor and enabling some decisions to be deferred until the appropriate time." –Steve Schneider [16]

Abstraction is the process of ignoring details that are of no immediate concern: it is a many-to-one mapping that allows one to treat different things as though they are the same. Here, those details are the differences in an algorithm that might be viewed as incidental, and we can use nondeterminism to abstract from them.

Moving in the other direction is refinement, the opposite of abstraction. If algorithm P refines S, then we view S as being less prescriptive than P, and we write $S \sqsubseteq P$. The algorithms SOLVE-CON and SOLVE-SIM are refinements of the sequential and multiprocess algorithms presented in this section.

3.1. A sequential algorithm

First, we present a sequential abstraction of SOLVE-CON and SOLVE-SIM. The algorithm SOLVE-SEQ, shown in Fig. 6, makes use of a nondeterministic subset operator that, as the name implies, yields an arbitrary subset of another set. It is implicitly defined as:

$$\forall x.(x \in subset(B) \Rightarrow x \in B)$$

and with the mild fairness constraint that, in the limit, an element of the set is produced infinitely often.

The algorithm is defined so that in one iteration of the while loop, a subset of active joints is released simultaneously. Since those being released are chosen arbitrarily at each iteration, some joints may be balanced more frequently than others. In other words, it is not necessary

```
1  SOLVE-SEQ(G, tol)
2      converged = FALSE
3      while not converged
4          converged = TRUE
5          s = subset(active(G))
6          for each vertex v ∈ active(G)
7              v.m = moment(G, v)
8          for each vertex v ∈ active(G)
9              if |v.m| > tol
10                 converged = FALSE
11                 if v ∈ s
12                     RELEASE(G, v, v.m)
```

Fig. 6. Generalized moment distribution as a sequential algorithm.

to complete a full cycle before moving on to the next cycle. In the limit, joints are balanced infinitely often and none is left unbalanced. We prove the algorithm is correct in Section 5 by showing it is functionally equivalent to SOLVE-CON and SOLVE-SIM.

That SOLVE-SEQ is an abstraction of the basic algorithms should be clear. For instance,

$$\text{SOLVE}-\text{SEQ} \sqsubseteq \text{SOLVE}-\text{CON}.$$

since we can imagine *subset*(*active*(G)) on line 5 always returning a single, different element of *active*(G) until the set is exhausted, and then repeating, yielding the behavior of the joint balancing approach of SOLVE-CON. Also,

$$\text{SOLVE}-\text{SEQ} \sqsubseteq \text{SOLVE}-\text{SIM}$$

since we can imagine *subset*(*active*(G)) always returning the full set *active*(G) at each iteration, yielding the behavior of SOLVE-SIM.

3.2. A multiprocess algorithm

Sometimes it is more natural to specify the behavior of a system in terms of multiple, concurrent processes. A multiprocess algorithm is a nondeterministic composition of program components, or processes, that work independently but share state and coordinate updates as necessary. Such an approach can be used to define a fully asynchronous version of the Hardy Cross method that allows releases to be performed simultaneously, consecutively, and in combination, as before, but also interleaved arbitrarily, down to the statement level.

The multiprocess algorithm SOLVE-MUL, shown in Fig. 7, defines processes that evolve concurrently for each active joint, i.e., for each vertex $i \in \text{active}(G)$. The processes share access to G, the structure being analyzed. Thus, the process associated with joint u has access to its own moments $(u,v).m$ for any v, as well as a moment of joint v, i.e., $(v,u).m$, when carry-over moments are written. Each joint process then responds when its moment is unbalanced, which can be induced by moments being carried over from one or more adjacent joints.

Several aspects of the algorithm warrant comment. Generally speaking, because sharing state between processes can result in conflicts, we may specify some steps to be performed atomically, such as assignment statements, tests in a loop or a conditional, and decrement-by operations. A statement is atomic if it appears to other processes to occur instantaneously, so that no intermediate states are visible. Atomicity, in our algorithm, must be guaranteed by the decrement-by operation within RELEASE to sidestep the so-called lost update problem.[1]

Guarding each joint release is an await statement, which is successful only when the joint's unbalanced moment exceeds a specified tolerance. That is, the process cannot advance beyond the statement when its moments balance: it blocks until the condition becomes true. Numerically there is no harm in balancing joints that are already balanced, but there is the potential danger that, in real implementations, the process will otherwise get away from us and take over the computer's CPU.[2]

```
1  SOLVE-MUL(G, tol)
2      for each vertex v ∈ active(G)
3          run PROCESS(G, v, tol)
4
5  PROCESS(G, v, tol)
6      while TRUE
7          await |moment(G, v)| > tol
8          RELEASE(G, v, moment(G, v))
```

Fig. 7. Generalized moment distribution as a multiprocess algorithm.

When converged, all joint processes block on their await statements, since all joints are balanced, and the solution is available in the edge attribute $(u,v).m$ for all $(u,v) \in G.E$, as before. In contrast with sequential programs, however, a multiprocess implementation would ideally detect this state of termination using a separate, lower-priority process, quiescence detection [19], or more general techniques like Dijkstra's ring-based termination detection algorithm [3]. See Liu's thesis [10] for additional details.

It should be clear that SOLVE-MUL is an abstraction of the sequential algorithm, so we have

$$\text{SOLVE}-\text{MUL} \sqsubseteq \text{SOLVE}-\text{SEQ}$$

Since this is so, SOLVE-MUL is also an abstraction of SOLVE-CON and SOLVE-SIM. For instance, a multiprocess scheduler could choose to execute the joint processes sequentially, one after another, to yield the behavior of SOLVE-CON. Alternatively, it might schedule them concurrently so that line 8 of each process causes the unbalanced moments, *moment*(G,i), to be read concurrently, after which releases are simultaneously performed, to yield the behavior of SOLVE-SIM.

4. Iteration matrices for consecutive and simultaneous approaches

"The inherent simplicity of the moment-distribution method can be combined with the matrix methods to give a practical method of analysis." –Ralph Mozingo [13]

In 1968, Mozingo published a matrix formulation for simultaneous joint balancing that results in a geometric series whose sum can be expressed in closed form. Terms in the series are the sums and products of matrices that encode distribution and carry-over factors. We make use of this basic formulation, elaborating on matrix properties needed for the proofs that follow, and extending the approach so that it applies to generalized distribution sequences.

Edge indices. To allow assembly into a system of equations, we define an edge index function, a bijection that maps from M edges, or member ends, to indices:

$$f : G.E \rightarrow \{1, \ldots, M\}$$

Antiparallel mates. Each member has two member ends, so we denote by $e = (u,v)$ and $\bar{e} = (v,u)$ a pair of antiparallel directed edges, and define a function p that maps an edge index to the index of its antiparallel mate:

$$p(f(e)) = f(\bar{e})$$

for all e in $G.E$. Since an antiparallel edge is its own mate's mate, we have $i = p(p(i))$ for all $i \in 1..M$.

Member end sets. Recall that we have a member end set *ends*(G,u) associated with each joint u, i.e., the set of member ends incident from joint u. An analogous notion can be defined for the indices associated with those member ends.

[1] In our multiprocess algorithm, when one joint carries over a moment to an adjacent joint, the adjacent joint may also be distributing a moment to the same member end, creating a potential conflict. If the decrement-by operation is non-atomic, one of the updates could be lost. Consider, for instance, the following scenario. A variable x with a value of 10 is shared by two processes, and one of them executes the statement $x = x - 7$ while the other executes $x = x - 2$. If the statements are performed atomically, executing them in either order results in x having a final value of 1. If they are not performed atomically, however, and both statements simultaneously "read" the initial value of x (accessing its value on the their right-hand sides), and then perform their updates, the final value of x will be either 3 or 8. In other words, one of the updates is lost.

[2] There is no counterpart to await in sequential languages. Instead, it corresponds to the well-known wait/notify pattern of process communication, with condition variables and locks, in multithreaded languages like Python and Java. Here, the **await** statement is modeled after the construct of the same name in the algorithm language PlusCal [9].

Let set $S_i, i = 1, 2, \ldots, M$, be a set containing all member ends, including i itself, incident from a common joint. In other words,

$$S_i \triangleq \{f(u,v) | (u,v) \in G.E \wedge \exists w.(f(u,w) = i)\}$$

For instance, our model problem has ends (a,b), (b,a), (b,c), and (c,b), which we index from 1 through 4. Then $S_2 = \{2,3\}$ since $f(b,a) = 2$, edge (b,a) is incident from joint b, and the set of edges incident from b is $\{(b,a),(b,c)\}$

Since a member end can only be incident from a single joint,

$$\begin{array}{l} S_i \cap S_j = \varnothing \text{ if and only if ends } i \text{ and } j \text{ are } \textit{not} \text{ incident from the same joint} \\ S_i = S_j \text{ otherwise.} \end{array} \tag{1}$$

At times we use the condition $S_i = S_j$ idiomatically to establish that ends i and j are incident from the same joint, i.e., are in the same member end set.

4.1. Distribution matrix

Let K_i be the stiffness of the member containing end i, recognizing that, in general, K_i is not necessarily equivalent to $K_{p(i)}$. Then let matrix D be an $M \times M$ distribution matrix, where D_i is the ith column of D, and where d_{ij} is the ratio between the increment of the moment at end i due to the moment at end j during the distribution process. In other words:

$$d_{ij} = \begin{cases} 0 & \text{if } S_i \neq S_j \text{ or end } i \text{ is fixed} \\ -\dfrac{K_i}{\sum\limits_{k \in S_i} K_k} & \text{otherwise} \end{cases}, \tag{2}$$

which gives D some special properties:

1. D has multiple identical columns. In particular, if $S_j = S_k$, then D_j is equivalent to D_k, so

$$d_{ij} = d_{ik} \text{ if } S_j = S_k. \tag{3}$$

Thus there are at least $|S_j|$ columns identical to column D_j, including itself. Also, because of Eq. (1), there is at most one distinct negative value in each row.

2. D is singular.
3. The elements of D are in the range

$$-1 \le d_{ij} \le 0. \tag{4}$$

Moreover, an element of D is strictly less than zero ($-1 \le d_{ij} < 0$) if and only if $S_i = S_j$ and the corresponding joint is not fixed. This implies

$$d_{ii} \begin{cases} = 0 & \text{if end } i \text{ is fixed} \\ < 0 & \text{otherwise} \end{cases} \tag{5}$$

4. The sum of any column D_j is 0 if end j is fixed, or -1 otherwise:

$$\sum_{i=1}^{M} d_{ij} = \sum_{\substack{i=1 \\ S_i = S_j}}^{M} d_{ij} = \begin{cases} 0 & \text{if end } j \text{ is fixed} \\ -1 & \text{otherwise} \end{cases} \tag{6}$$

The added condition on the second summation, $S_i = S_j$, follows from above, i.e., that an element of D is strictly less than zero if and only if $S_i = S_j$ and the corresponding joint is not fixed.

5. Due to Eqs. (1) and (2),

$$d_{ij} < 0 \Rightarrow d_{p(i),j} = 0, \tag{7}$$

meaning that a particular end is unaffected by the other end of the same member during the redistribution process.

4.2. Carry-over matrix

Let matrix C be the matrix of carry-over factors. Because the carry-over moments are only transferred between the two ends of a member,

$$\begin{cases} 0 < c_{ij} < 1 & \text{if } j = p(i) \\ c_{ij} = 0 & \text{otherwise} \end{cases}. \tag{8}$$

Matrix product CD, then, represents the ratio of the carry-over moment on one end and the calculated unbalanced moment on the other end of the same member. Matrix CD essentially swaps any two rows in D that correspond to the two ends of a member, and then multiplies each row by the corresponding carry-over factor. Also due to Eq. (7), D and CD cannot both have negative elements at the same position, i.e., $d_{ij}(CD)_{ij} = 0$.

4.3. E matrix

For the iteration matrices that follow, it is helpful to define a matrix E as follows

$$E = D + CD \tag{9}$$

which also has special properties similar to, but in some cases distinct from, D:

1. Like D, matrix E has multiple identical columns, and if $S_j = S_k$, then E_j is equivalent to E_k, so

$$e_{ij} = e_{ik} \text{ if } S_j = S_k. \tag{10}$$

Thus there are at least $|S_j|$ columns identical to E_j, including itself. Also, because of Eq. (1), there are at most two distinct negative values in each row.

2. Also, like D, matrix E is singular.
3. The elements of E are again in the range

$$-1 \le e_{ij} \le 0. \tag{11}$$

An element of E is strictly less than zero ($-1 \le e_{ij} < 0$) if and only if $S_i = S_j$ and the corresponding joint is not fixed, or, in the case of E, any end j associated with the joint at the opposite end of end i. This again implies

$$e_{ii} = d_{ii} \begin{cases} = 0 & \text{if end } i \text{ is fixed} \\ < 0 & \text{otherwise} \end{cases} \tag{12}$$

and if $S_i = S_j$, then

$$e_{p(i),j} = c_{p(i),j} e_{ij}. \tag{13}$$

4. When limited to elements i,j satisfying $S_i = S_j$, the sums of columns D_j and E_j are equivalent:

$$\sum_{\substack{i=1 \\ S_i = S_j}}^{M} e_{ij} = \sum_{\substack{i=1 \\ S_i = S_j}}^{M} d_{ij} = \begin{cases} 0 & \text{if end } j \text{ is fixed} \\ -1 & \text{otherwise} \end{cases} \tag{14}$$

4.4. Iteration matrix for simultaneous joint balancing

The simultaneous joint balancing algorithm in Section 2.2, SOLVE-SIM, is akin to an incremental form of the Jacobi iterative method, in which calculations of the current iteration use only those from the prior iteration, so the order in which the equations are solved is irrelevant.

The corresponding iteration matrix, B_{sim}, can be defined as

$$B_{sim} = I + D + CD = I + E \tag{15}$$

and the iteration can be written as

$$\mathcal{M}^{(k+1)} = B_{sim}\mathcal{M}^{(k)} \tag{16}$$

where $\mathcal{M}^{(k)}$ is a vector representing moments at each member end at step k. The limit of Eq. (16) for simultaneous joint balancing[3] is

$$\mathcal{M}^{(\infty)} = B_{sim}^{\infty}\mathcal{M}^{(0)}. \tag{17}$$

4.5. Iteration matrix for consecutive joint balancing

The consecutive joint balancing algorithm in Section 2.1, Solve-con, corresponds to an incremental form of the Gauss–Seidel method, which uses the most recently updated estimates in each iteration. Consequently, the equations at each iteration cannot be evaluated independently, and the order in which they are taken affects the intermediate results.

Like the simultaneous approach, however, the consecutive joint balancing algorithm can also be represented as a simple iteration but with a different iteration matrix, B_{con}. We begin by decomposing matrix E based on joint contributions as follows

$$E = \sum_{l=1}^{N} \hat{E}_l \tag{18}$$

where N is the number of joints, and $\hat{E}_l$ is a square matrix that contains the columns of E corresponding to all ends connected to joint l, while the other columns of $\hat{E}_l$ are zero. The matrix representing the process of releasing a joint l once, then, can be expressed as

$$\hat{B}_l = I + \hat{E}_l \tag{19}$$

and the iteration matrix for consecutive joint balancing, B_{con}, can be expressed as

$$B_{\text{con}} = \prod_{l=1}^{N} \hat{B}_l = \prod_{l=1}^{N}\left(I + \hat{E}_l\right) \tag{20}$$

The corresponding iteration can be written as

$$\mathcal{M}^{(k+1)} = B_{con}\mathcal{M}^{(k)} \tag{21}$$

and, in the limit, for consecutive joint balancing we have

$$\mathcal{M}^{(\infty)} = B_{con}^{\infty}\mathcal{M}^{(0)}. \tag{22}$$

Theorem 1. Let $\hat{B}_l$ be the matrix representing the process of releasing a joint l once, then $\hat{B}_l^2 = \hat{B}_l$. Proof. Due to Eqs. (2), (3), (13), and (14),

$$\left(\hat{E}_l^2\right)_{ij} = \sum_{k=1}^{M}\left(\hat{E}_l\right)_{ik}\left(\hat{E}_l\right)_{kj} = \left(\hat{E}_l\right)_{ij}\sum_{k=1}^{M} D_{kj} = -\left(\hat{E}_l\right)_{ij}, \tag{23}$$

implying that

$$\hat{E}_l^2 + \hat{E}_l = 0, \tag{24}$$

which holds even even when members are not symmetric, i.e., when $K_i \neq K_{p(i)}$. Then

$$\begin{aligned} 2\hat{E}_l + \hat{E}_l^2 &= \hat{E}_l \\ \Rightarrow I + 2\hat{E}_l + \hat{E}_l^2 &= I + \hat{E}_l \\ \Rightarrow \left(I + \hat{E}_l\right)^2 &= I + \hat{E}_l \\ \Rightarrow \hat{B}_l^2 &= \hat{B}_l \end{aligned} \tag{25}$$

The physical interpretation of Theorem 1 is that releasing a particular joint twice consecutively is the same as doing so only once. This makes sense because, as we saw with algorithm Solve-mul, there will be a zero unbalanced moment left after any release.

5. Correctness of the sequential algorithm

The sequential algorithm defined in Section 3.1 can be shown to be functionally equivalent to the consecutive and simultaneous joint balancing algorithms presented in Sections 2.1 and 2.2. The computations performed are analogous to Eq. (16) but with a generalized iteration matrix, $B_{seq}^{(k)}$, that depends on k, the index of iteration, and where $k = 0$ corresponds to the initial state $B_{seq}^{(0)} = I$. It can be written as

$$B_{\text{seq}}^{(k)} = I + \sum_{l \in V^{(k)}} \hat{E}_l, \quad k = 1, 2, 3, \ldots \tag{26}$$

where $V^{(k)}$ is a schedule set containing all joints to be released simultaneously at the kth iteration. The iteration can be written as

$$\mathcal{M}^{(k+1)} = B_{seq}^{(k)}\mathcal{M}^{(k)}. \tag{27}$$

The limit of Eq. (27) for generalized sequential iteration is

$$M^* = \prod_{k=0}^{\infty} B_{\text{seq}}^{(k)} M^{(0)}. \tag{28}$$

As one would expect, both simultaneous and consecutive iterations are special cases that can be accommodated by choosing an appropriate schedule set $V^{(k)}$. For simultaneous joint balancing, each $V^{(k)}$ may be chosen to include all joints, so we have

$$B_{\text{seq}}^{(k)} = I + \sum_{l=1}^{N} \hat{E}_l = I + E = B_{\text{sim}}. \tag{29}$$

For consecutive joint balancing, each $V^{(k)}$ may be chosen to be a single joint $(k\text{-}1 mod N) + 1$, so every N iterations we build up a sequence of products

$$\prod_{l=1}^{N} B_{\text{seq}}^{(l)} = \prod_{l=1}^{N}\left(I + \hat{E}_l\right) = B_{\text{con}}. \tag{30}$$

Example. Consider a structure with three joints, so we have $\hat{E}_1$, $\hat{E}_2$, and $\hat{E}_3$, and schedule sets whose first few values are arbitrarily chosen as follows:

$$V^{(1)} = \hat{E}_1, V^{(2)} = \hat{E}_2 + \hat{E}_3, V^{(3)} = \hat{E}_1, V^{(4)} = \hat{E}_1 + \hat{E}_3, V^{(5)} = \ \ldots$$

Using the schedule sets, we show that the product of the first k iteration matrices in the sequence, defined as $G^{(k)}$, can be expanded as a sum of matrix products

$$G^{(k)} = \prod_{i=0}^{k} B_{\text{seq}}^{(i)} \tag{31}$$

so that $G^{(k)}M^{(0)} = \prod_{i=0}^{k} B_{\text{seq}}^{(i)} M^{(0)}$ is the moment vector after the kth iteration. During the expansion, we perform simplifications such as gathering and canceling terms, yielding the following:

$$G^{(1)} = \prod_{i=0}^{1} B_{\text{seq}}^{(i)} = I + \hat{E}_1$$

$$\begin{aligned} G^{(2)} = \prod_{i=0}^{2} B_{\text{seq}}^{(i)} &= \left(I + \hat{E}_1\right)\left(I + \hat{E}_2 + \hat{E}_3\right) \\ &= I + \hat{E}_1 + \hat{E}_2 + \hat{E}_3 + \hat{E}_1\hat{E}_2 + \hat{E}_1\hat{E}_3 \end{aligned}$$

[3] The equivalent closed-form solution given by Mozingo [13], though not used here, is $\mathcal{M}^{(\infty)} = (I + D)(I - CD)^{-1}\mathcal{M}^{(0)}$.

$$G^{(3)} = \prod_{i=0}^{3} B_{seq}^{(i)} = \left(I+\hat{E}_1\right)\left(I+\hat{E}_2+\hat{E}_3\right)\left(I+\hat{E}_1\right) = I+\hat{E}_1+\hat{E}_2+\hat{E}_3+\hat{E}_1\hat{E}_2+\hat{E}_1\hat{E}_3+\hat{E}_2\hat{E}_1+\hat{E}_3\hat{E}_1+\hat{E}_1\hat{E}_2\hat{E}_1+\hat{E}_1\hat{E}_3\hat{E}_1$$

$$G^{(4)} = \prod_{i=0}^{4} B_{seq}^{(i)} = \left(I+\hat{E}_1\right)\left(I+\hat{E}_2+\hat{E}_3\right)\left(I+\hat{E}_1\right)\left(I+\hat{E}_1+\hat{E}_3\right) = I+\hat{E}_1+\hat{E}_2+\hat{E}_3+\hat{E}_1\hat{E}_2+\hat{E}_1\hat{E}_3+\hat{E}_2\hat{E}_1+\hat{E}_3\hat{E}_1+\hat{E}_1\hat{E}_2\hat{E}_1+\hat{E}_1\hat{E}_3\hat{E}_1 + \hat{E}_2\hat{E}_3+\hat{E}_1\hat{E}_2\hat{E}_3+\hat{E}_2\hat{E}_1\hat{E}_3+\hat{E}_3\hat{E}_1\hat{E}_3+\hat{E}_1\hat{E}_2\hat{E}_1\hat{E}_3+\hat{E}_1\hat{E}_3\hat{E}_1\hat{E}_3$$

And so on. Terms in the simplified series, e.g., matrix products like $\hat{E}_1\hat{E}_3\hat{E}_1$, are unique: even when subsequent iterations produce a duplicate term, they also simultaneously produce a corresponding canceling term, e.g., $\hat{E}_1\hat{E}_3\hat{E}_1+\hat{E}_1\hat{E}_3\hat{E}_1^2$, whose sum is zero due to Eq. (24), since $\hat{E}_l^2+\hat{E}_l=0$. We show this below.

Theorem 2. For an arbitrary series of schedule sets $V^{(k)}, k=1,2,3,\ldots$, if each joint is balanced infinitely often, then generalized sequential iteration is equivalent to the following sum:

$$\prod_{k=0}^{\infty} B_{\text{seq}}^{(k)} = \sum_{m=0}^{\infty} U_m, \tag{32}$$

where

$$U_0 = I \tag{33}$$

and

$$U_m = \sum_{l_1=1}^{N} \sum_{\substack{l_2=1 \\ l_2\neq l_1}}^{N} \sum_{\substack{l_3=1 \\ l_3\neq l_2}}^{N} \cdots \sum_{\substack{l_m=1 \\ l_m\neq l_{(m-1)}}}^{N} \hat{E}_{l_1}\hat{E}_{l_2}\hat{E}_{l_3}\ldots\hat{E}_{l_m} \tag{34}$$

Proof. The left-hand side of Eq. (32) accommodates an arbitrary series of schedule sets and is equivalent to the constant right-hand side expression, which depends only on the properties and layout of a structure.

Recalling that $G^{(k)}$ is the product of the first k iteration matrices in the sequence, let

$$\Delta G^{(k)} = G^{(k)} - G^{(k-1)}, k = 1,2,3,\ldots \tag{35}$$

and

$$\Delta G^{(0)} = G^{(0)} = I. \tag{36}$$

Since $G^{(k)}\mathcal{M}^{(0)}$ and $G^{(k-1)}\mathcal{M}^{(0)}$ are the moment vectors after the kth and $(k\text{-}1)^{th}$ iterations, respectively, the physical interpretation of $\Delta G^{(k)}\mathcal{M}^{(0)} = G^{(k)}\mathcal{M}^{(0)} - G^{(k-1)}\mathcal{M}^{(0)}$ is the increment of the moment vector in the kth iteration.

This implies when $k\geq 1$ that

$$\begin{aligned}\Delta G^{(k)} &= B_{\text{seq}}^{(k)} G^{(k-1)} - G^{(k-1)} \\ &= \left(I + \sum_{l\in V^{(k)}} \hat{E}_l\right) G^{(k-1)} - G^{(k-1)} \\ &= \sum_{l\in V^{(k)}} \hat{E}_l G^{(k-1)},\end{aligned} \tag{37}$$

which is central to the proof that follows.

The left-hand side products of Eq. (32) can now be rewritten as the summation

$$\prod_{k=0}^{\infty} B_{\text{seq}}^{(k)} = \sum_{k=0}^{\infty} \Delta G^{(k)}. \tag{38}$$

Then it can be proved by induction on m that

$$\sum_{k=0}^{\infty} \Delta G^{(k)} = \sum_{m=0}^{\infty} U_m \tag{39}$$

by first showing that the left-hand side contains U_0 and U_1 exactly once, and then showing that if it contains U_m exactly once, it likewise contains U_{m+1} exactly once. It can likewise be shown that there are no duplicate terms on the left-hand side after simplification. In addition, due to the nature of $B_{seq}^{(k)}$, i.e., that it consists only of I and E_l terms, no other forms can be present.

Basis. For $m=0$, we know from Eq. (37) that I will not appear as a term in $\Delta G^{(k)}$ when $k\geq 1$. Thus the limit contains $\Delta G^{(0)}=U_0=I$ exactly once. For $m=1$, if $B_{seq}^{(k)}$ is the first B_{seq} term that contains $\hat{E}_l$, then because of Eq. (37) and the fact that $G^{(k-1)} = \sum_{k=0}^{k-1} \Delta G^{(k)}$ has the term $\Delta G^{(0)}=I$, $\Delta G^{(k)}$ is also the first ΔG term that contains $\hat{E}_l$. Because all joints get updated infinitely often, for each $\hat{E}_l, l=1,2,\ldots,N$, there exists a k such that $B_{seq}^{(k)}$ contains that $\hat{E}_l$. Thus the limit contains all $\hat{E}_1,\hat{E}_2,\ldots,\hat{E}_N$. In addition, $\Delta G^{(k)}$ is the only ΔG term that contains $\hat{E}_l$ after simplification. The reason is that if $\Delta G^{(j)}$ for $j>k$ also contains $\hat{E}_l$, then $B_{seq}^{(j)}$ must contain $\hat{E}_l$. Since $G^{(j-1)}$ contains both I and $\hat{E}_l$ exactly once, from Eq. (37) we know $\Delta G^{(j)}$ contains both $\hat{E}_l$ and $\hat{E}_l^2$ exactly once, which cancel each other due to Eq. (24). Thus, all terms in U_1, i.e., $\hat{E}_1,\hat{E}_2,\ldots,\hat{E}_N$, occur exactly once.

Hypothesis**.** Assume that the left-hand side contains U_m exactly once.

Inductive step**.** Show that the left-hand side contains U_{m+1} exactly once.

If $G^{(k)}$ is the first G term that contains a particular term u of U_m after simplification, say $u=\hat{E}_l w$ that leads with $\hat{E}_l$, then $B_{seq}^{(k)}$ must contain $\hat{E}_l$.

1. Let $B_{seq}^{(j)}, j>k$ be the next B_{seq} term that contains $\hat{E}_i$, $i\neq l$, then $\hat{E}_i u$ first appears in $\Delta G^{(j)}$ due to Eq. (37). Moreover, $\Delta G^{(j)}$ is the only ΔG term that contains $\hat{E}_i u$ after simplification. The reason is that, if $\Delta G^{(q)}, q>j$ also contains $\hat{E}_i u$, then $B_{seq}^{(q)}$ must contain $\hat{E}_i$. Since $G^{(q-1)}$ contains both u and $\hat{E}_i u$ exactly once, $\Delta G^{(q)}$ contains both $\hat{E}_i u$ and $\hat{E}_i^2 u$ exactly once, which cancel each other due to Eq. (24). Thus, together with the fact that all joints get updated infinitely often, the above statements prove that the limit contains U_{m+1} exactly once.
2. Let $B_{seq}^{(j)}, j>k$ be the next B_{seq} term that contains $\hat{E}_l$. Since $G^{(j-1)}$ contains both w and $\hat{E}_l w$ exactly once, from Eq. (37) $\Delta G^{(j)}$ contains both $\hat{E}_l w$ and $\hat{E}_l^2 w$ exactly once, which cancel each other due to Eq. (24) Hence, the limit cannot have terms with the same index in any two consecutive E_l terms after simplification.

As a result, the left-hand side contains U_{m+1} exactly once, and there are no extraneous terms.

Hence Theorem 2 is true, proving that the sequential algorithm is functionally equivalent to its consecutive and simultaneous joint balancing refinements.

6. Correctness of the multiprocess algorithm

Further generalizing the sequential algorithm is the multiprocess abstraction defined in Section 3.2, which can be shown to be equivalent, while offering the full range of asynchronous behavior allowed by the Hardy Cross method. To accommodate the arbitrary interleaving of releases, we further decompose matrix E and again prove equivalence by induction.

6.1. Further decomposition of E matrix

The matrix E can be further decomposed in two dimensions. In other words, instead of Eq. (18), E can be defined as

$$E = \sum_{i=1}^{N} \sum_{j=1}^{N} [i,j] \tag{40}$$

where N is the number of joints. Recalling that M is the number of member ends, we adopt the unconventional but serviceable notation $[i,j]$ to denote an $M\times M$ square matrix such that

$$[i,j]_{kl} = \begin{cases} E_{kl} \le 0 & \text{if ends } k \text{ and } l \text{ are connected to joints } i \text{ and } j, \text{ respectively} \\ 0 & \text{otherwise} \end{cases}. \tag{41}$$

Matrix $[i,j]$ contains some elements of E whose rows correspond to all ends connected to joint i, and whose columns correspond to all ends connected to joint j. The other elements of matrix $[i,j]$ are zero. Due to the properties of matrix D, we see that

$$[i,i]_{kl} = \begin{cases} D_{kl} \le 0 & \text{if end } l \text{ is connected to join } i \\ 0 & \text{otherwise} \end{cases} \tag{42}$$

and the column sum of matrix $[i,i]$ is

$$\sum_k [i,i]_{kl} = \begin{cases} -1 & \text{if end } l \text{ is connected to joint } i \\ 0 & \text{otherwise} \end{cases}. \tag{43}$$

The physical interpretation of $[i,j]$ is how the moments at joint i are affected by joint j. Thus, when $i=j$, $[i,j]\mathcal{M}^{(k)}$ is the redistribution step for the unbalanced moment at joint i. When $i\neq j$, $[i,j]\mathcal{M}^{(k)}$ is the action of joint j carrying over a moment to a particular end of joint i. Although $[i,j]$ is zero if joints i and j are not connected, or if joint j is fixed, this case need not be addressed separately for purposes of the proof.

There are two important properties of matrix $[i,j]$ that are required for proofs later in this section. Due to Eqs. (41), (42) and (43),

$$[*,j][j,j] = -[*,j] \tag{44}$$

and

$$[*,i][k,j] = 0, i\neq k. \tag{45}$$

where the following wildcard notation is adopted:

1. Let $[*,j]$ stand for any matrix in the set $\{[i,k] | i\in 1..N \wedge j=k\}$.
2. Let $[?,j]$ stand for any matrix in the set $\{[i,k] | i\in 1..N \wedge j=k \wedge i\neq j\}$.
3. Let $\sum [*,j]$ be the sum of the set $\{[i,k] | i\in 1..N \wedge j=k\}$, so clearly

$$\sum [*,j] = \hat{E}_j. \tag{46}$$

4. Let $\sum [?,j]$ be the sum of the set $\{[i,k] | i\in 1..N \wedge j=k \wedge i\neq j\}$.

6.2. Elementary operations

The multiprocess algorithm can be represented as a series of elementary operations, i.e., as a series of $[i,j]$ matrices. Here, an elementary operation is defined to be atomic, i.e., it is not interfered with by other elementary operations. The action of releasing a joint j, then, can be described as follows:

1. Start by taking a “snapshot,” say $\mathcal{M}'$, of the current member end moments.
2. Calculate the increments of moments based on $\mathcal{M}'$ without making any updates. The increments need not be immediately added to $\mathcal{M}$, and their application order is arbitrary, so we can imagine their being kept in a pending operation set, O.
3. Pick one of the increments from operation set O and add it to $\mathcal{M}$.
4. Ensure that increments within joint j, $[j,j]\mathcal{M}'$, are added to $\mathcal{M}$ before releasing joint j again.

Commentary. Since any matrix $[?,j]$ has at most one row containing non-zero elements, it only updates one value of the moment vector. Although $[j,j]$ could update multiple values of the moment vector because it could have multiple rows that contain non-zero elements, $[j,j]$ can still be considered an elementary operation due to item 4 above. Thus, from a computational point of view, if updates are performed atomically, then the multiprocess algorithm can be viewed as interleaving elementary operations arbitrarily. An available elementary operation then is either 1) a joint taking a snapshot and calculating increments of moments if no balancing moment for that joint is pending in operation set O, or 2) moving a matrix multiplication result from O to $\mathcal{M}$.

This arbitrary interleaving can also be shown in mathematical form:

$$\mathcal{M}^{(k+1)} = W^{(k+1)}\mathcal{M}^{(k)} = \begin{cases} \mathcal{M}^{(k)} & \text{if an arbitrary joint takes a snapshot and calculates increments of moments} \\ \mathcal{M}^{(k)} + [i,j]\mathcal{M}^{(l)} & \text{if an arbitrary matrix products is moved from } O \text{ to } \mathcal{M} \end{cases} \tag{47}$$

where $W^{(k+1)}$ is a function that performs an arbitrary available elementary operation on $\mathcal{M}^{(k)}$ at step $k+1$, as long as every joint is updated infinitely often and each joint finishes updating itself before it is released again.

The limit of Eq. (47) for generalized multiprocess iteration is

$$\mathcal{M}^* = \left(W^{(\infty)}\ldots W^{(3)}W^{(2)}W^{(1)}\right)\mathcal{M}^{(0)}. \tag{48}$$

By placing restrictions on W in choosing the order of the elementary operations, one may obtain the more specialized behavior of the sequential algorithm. Specifically, to mimic the sequential refinement, once operations in O begin to be executed, *all* operations in O must be executed before moving on to other joints and taking snapshots.

Theorem 3. For an arbitrary series of elementary operations $W^{(k)}$, $k=1,2,3,\ldots$, if each joint is balanced infinitely often, then generalized multiprocess iteration is equivalent to the following sum:

$$W^{(\infty)}\ldots W^{(3)}W^{(2)}W^{(1)} = \sum_{m=0}^{\infty} Z_m, \tag{49}$$

where

$$Z_0 = I \tag{50}$$

and

$$Z_m = \left(\sum_{j_1}\sum[*,j_1]\right)\left(\sum_{j_2\neq j_1}\sum[*,j_2]\right)\ldots\left(\sum_{j_m\neq j_{m-1}}[*,j_m]\right) \tag{51}$$

Proof. The left-hand side of Eq. (49) accommodates an arbitrary series of elementary operations and is equivalent to the constant right-hand side expression, which, again, depends only on the properties and layout of a structure. And, of course, due to Eq. (46), the right-hand sides of Eqs. (32) and (49) must be identical.

We begin by defining $P^{(k)}$, a partial series of elementary operations, as:

$$\mathcal{M}^{(k)} = \left(W^{(k)}\ldots W^{(3)}W^{(2)}W^{(1)}\right)\mathcal{M}^{(0)} = P^{(k)}\mathcal{M}^{(0)} \tag{52}$$

with $P^{(0)}=I$. Using this notation, the basic steps of the multiprocess algorithm can be described as follows:

1. When releasing a joint j at step k, the snapshot taken is the current $P^{(k)}$. Then, an elementary operation set containing all $[*,j]P^{(k)}$ is added to O.
2. In an arbitrary order, elementary operations are drawn from O and added to P.
3. We require that $[j,j]P^{(k)}$ be executed before releasing joint j again.

4. All joints must be released infinitely often. Thus an elementary operation set containing all $[*,j]P^{(k)}$ is added to O infinitely often.

In the limit, we have

$$\mathcal{M}^* = \left(W^{(\infty)}...W^{(3)}W^{(2)}W^{(1)}\right)\mathcal{M}^{(0)} \tag{53}$$

To assist in carrying out the proof, we add a final piece to the notation:

$$\hat{Z}_m^j = \left(\sum_{j_1 \neq j}\sum[*,j_1]\right)\left(\sum_{j_2 \neq j_1}\sum[*,j_2]\right)...\left(\sum_{j_m \neq j_{m-1}}\sum[*,j_m]\right) \tag{54}$$

with

$$\hat{Z}_0^j = I. \tag{55}$$

The relationship between Z and $\hat{Z}$ is

$$Z_m = \sum_j \sum[*,j]\hat{Z}_{m-1}^j. \tag{56}$$

Then it can be proved by induction on m that

$$W^{(\infty)}...W^{(3)}W^{(2)}W^{(1)} = \sum_{m=0}^{\infty} Z_m \tag{57}$$

by first showing that the left-hand side contains Z_0 and Z_1 exactly once, and then showing that if it contains Z_m exactly once, it likewise contains Z_{m+1} exactly once, and that there are no extraneous terms.

Basis. For $m=0$, we know that I appears exactly once in the left-hand side. For $m=1$, Z_1 appears exactly once when each joint is first released.

Hypothesis. Assume that the left-hand side contains Z_m exactly once.

Inductive step. Show that the left-hand side contains Z_{m+1} exactly once.

1. Since all joints will be released infinitely often, all terms in Z_m will be pre-multiplied by $\sum_j [*,j]$, so Z_{m+1} appears in the left-hand side.
2. Because of Eq. (56), the only possible extra terms that may be created in this process are $[*,j][j,j]\hat{Z}_{m-1}^j$ or $[*,j][?,j]\hat{Z}_{m-1}^j$. The existence of $[*,j]\hat{Z}_{m-1}^j$ implies the existence of $\hat{Z}_{m-1}^j$ that will also get pre-multiplied by $[*,j]$ at the same time, which results in a duplicated $[*,j]\hat{Z}_{m-1}^j$. Since $[*,j][j,j]\hat{Z}_{m-1}^j = -[*,j]\hat{Z}_{m-1}^j$ according to Eq. (44) above, $[*,j][j,j]\hat{Z}_{m-1}^j$ gets canceled by the duplicated $[*,j]\hat{Z}_{m-1}^j$. Meanwhile, $[*,j][?,j]\hat{Z}_{m-1}^j$ is always zero due to Eq. (45) above. As a result, the left-hand side has no extraneous terms.
3. Obviously, adding any term from Z_{m+1}, say $[*,j]\hat{Z}_m^j$, the first time will not create a duplicate term. Because $[j,j]P^{(k)}$ must be executed before releasing joint j again, when $[*,j]\hat{Z}_m^j$ gets created again, $[j,j]\hat{Z}_m^j$ must exist and $[*,j][j,j]\hat{Z}_m^j = -[*,j]\hat{Z}_m^j$ must also be created at the same time. Hence the duplicate $[*,j]\hat{Z}_m^j$ will be canceled.

The duplicated $[*,j]\hat{Z}_m^j$ and its corresponding extra term $[*,j][j,j]\hat{Z}_m^j$ will always be moved from O to $P^{(k)}$ at the same time, due to the fact that $\hat{Z}_m^j$ and $[j,j]\hat{Z}_m^j$ are in the same snapshot.

As a result, the left-hand side contains Z_{m+1} exactly once, and there are no extraneous terms.

Hence Theorem 3 is true, proving that the multiprocess algorithm is functionally equivalent to its sequential refinement. □

We note that, in both theorems, the fact that the column sum of matrix D is -1 for non-fixed joints plays a central role in simplifying and eliminating terms.

7. Conclusion

Despite its age, the Hardy Cross method has been the subject of the occasional publication over the years. Beyond the already cited work of Volokh, Guo, and Mozingo, in 2009 Dowell [4] found closed-form solutions for regular continuous beams and bridge structures with any number of spans. His method is based on the consecutive joint balancing approach, and is derived by taking the limit of infinite geometric series. Presaging his work are the formulas presented by Mawby [11] in 1968 for two- and three-span continuous beams, who likewise recognized the fact that "the process of Moment Distribution is merely an approximate method for finding the sum of several series that converge at infinity." Earlier publications looking into computational aspects of the method, including tables and formulas for fixed-end moments, symmetry and anti-symmetry considerations, and others are cited in Gere's extensive bibliography [6].

In this study, we extend the work of both Volokh and Guo by showing that arbitrary distribution sequences and further generalizations leave the method sound and intact, and we formalize those abstractions as nondeterministic sequential and multiprocess algorithms. To the authors' knowledge, this is the first endeavor at a treatment of this kind.

In some ways, our proofs are unconventional. We take the traditional statement of the moment distribution method—formalized by Solve-sim and Solve-con—as being correct, and prove that Solve-seq and Solve-mul are correct by showing equivalence. We could contrast this approach with the tools and techniques more commonly found in both numerical analysis and software engineering. Regarding the former, for instance, convergence as a property is already established for Solve-sim and Solve-con due to the work of Volokh and Guo. Buttressing their arguments, we have observed that the eigenvalues of the iteration matrix for simultaneous joint balancing lie between -0.5 and 1, inclusive, by making use of the Gershgorin circle theorem. Regarding the latter, in contrast with studies on software engineering techniques, ours does not start with the kind of specification that would enable a straightforward syntactic proof. For sequential programs, however, we now know as a result that the simultaneous release of an arbitrary subset of joints can be used as a loop invariant for partial correctness proofs.

Also on the topic of proofs, we make the observation that the property of a matrix A in Eq. (24), i.e., $A^2+A=0$, plays an important role in proving equivalence and is therefore one of the sufficient and possibly necessary conditions for correctness, and might find a similar role if generalized and applied to other iterative problems in numerical linear algebra.

Finally, about implementations, our rationale for presenting Python code in Appendix A is not because we expect to improve performance by distributing computation across multiple processors, but rather to present the algorithms in executable form and in the context of a popular programming language with multithreading support. Although problems typically addressed by moment distribution are small enough to avoid any concern at all about computation time, we imagine that performance gains for problems of modest size could be obtained through parallelization if one employs red-black ordering, block relaxation schemes, or other techniques for reducing synchronization costs in the iterative solution of sparse linear systems [15].

Acknowledgments

The authors wish to thank Prof. George Turkiyyah at the American University of Beirut for his comments on an earlier draft of this article.

Appendix A. Program Listings

Implementation of the basic and sequential algorithms

```
# A direct translation of the algorithms into Python 3

from random import sample, randint

# Represent member ends and vertices as classes, and graphs as dictionaries

class End(object):

    def __init__(self, d, c, m):
        self.d, self.c, self.m = d, c, m

    def __repr__(self):
        return repr([self.d, self.c, self.m])

class Vertex(object):

    def __init__(self, name):
        self.name = name

    def __repr__(self):
        return repr(self.name)

def model_problem():
    a, b, c = Vertex('a'), Vertex('b'), Vertex('c')
    G = {a: {}, b: {}, c: {}}
    G[a][b] = End(0.0, 0.5, -172.8)
    G[b][a] = End(0.5, 0.5,  115.2)
    G[b][c] = End(0.5, 0.5, -416.7)
    G[c][b] = End(1.0, 0.5,  416.7)
    return G

# Basic functions for working with graphs

def ends(G, u):
    return G[u].values()

def moment(G, u):
    return sum([e.m for e in ends(G, u)])

def active(G):
    return {u: G[u] for u in G if any(e.d > 0 for e in ends(G, u))}

# Release a joint

def release(G, u, x):
    for v in G[u]:
        G[u][v].m -= G[u][v].d * x
        G[v][u].m -= G[u][v].c * G[u][v].d * x


# The moment distribution algorithms

def solve_con(G, tol):
    converged = False
    while not converged:
        converged = True
        for v in active(G):
            if abs(moment(G, v)) > tol:
                converged = False
                release(G, v, moment(G, v))

def solve_sim(G, tol):
    converged = False
    while not converged:
        converged = True
        for v in active(G):             # cache unbalanced moments
            v.m = moment(G, v)          #   of active joints
        for v in active(G):
            if abs(v.m) > tol:
                converged = False
                release(G, v, v.m)
```

```
def subset(G):
    return sample(G.keys(), randint(0, len(G)))

def solve_seq(G, tol):
    converged = False
    while not converged:
        converged = True
        s = subset(active(G))
        for v in active(G):             # cache unbalanced moments
            v.m = moment(G, v)          #   of active joints
        for v in active(G):
            if abs(v.m) > tol:
                converged = False
                if v in s:              # release a subset of joints
                    release(G, v, v.m)

# Use all three algorithms to solve the model problem

def moments(G):
    return {u.name + v.name: G[u][v].m for u in G for v in G[u]}

def main():
    for solve in [solve_sim, solve_con, solve_seq]:
        G = model_problem()
        solve(G, 0.001)
        print(moments(G))

if __name__ == '__main__':
    main()
```

Implementation of the multiprocess algorithm

```
from threading import Condition, Lock, Thread

# Extend classes to incorporate locks and conditions needed for concurrency

class End(object):

    def __init__(self, d, c, m):
        self.d, self.c, self.m = d, c, m
        self.lock = Lock()              # Lock is used only by make decr_moment.

    def decr_moment(self, by):          # In Python, a lock is needed to make
        with self.lock:                 #   this atomic.  Let each end have its
            self.m -= by                #   own to avoid limiting concurrency.

    def __repr__(self):
        return repr([self.d, self.c, self.m])

class Vertex(object):

    def __init__(self, name):           # Let each joint have its own condition
        self.name = name                #   for process communication using
        self.cond = Condition(Lock())   #   wait/notify instead of spinning.

    def __repr__(self):
        return repr(self.name)

# Redefine release so that it performs process communication

def release(G, u, x):
    for v in G[u]:
        G[u][v].decr_moment(G[u][v].d * x)
        G[v][u].decr_moment(G[u][v].c * G[u][v].d * x)
        with v.cond:                    # Notify neighboring joint that it may
            v.cond.notify()             #   now have an unbalanced moment.

# The multiprocess algorithm

def solve_mul(G, tol):
    for v in active(G):
        Thread(target=process, args=(G, v, tol)).start()

def process(G, v, tol):
    while True:
        with v.cond:                    # Block until moment becomes unbalanced.
            while abs(moment(G, v)) <= tol:
                v.cond.wait()
        release(G, v, moment(G, v))
        print(moments(G))
```

References

[1] Cross H. Analysis of continuous frames by distributing fixed-end moments. Proc Am Soc Civ Eng 1930:919–28.
[2] Cross H, Morgan ND. Continuous frames of reinforced concrete. John Wiley & Sons, Inc.; 1932
[3] Dijkstra EW, Feijen WHJ, Van Gasteren AJM. Derivation of a termination detection algorithm for distributed computations. Inf Process Lett 1983;16:217–9.
[4] Dowell RK. Closed-form moment solution for continuous beams and bridge structures. Eng Struct 2009;31(8):1880–7.
[5] Eaton LK. Hardy Cross and the moment distribution method. Nexus Network J 2001; 3(2):15–24.
[6] Gere JM. Moment distribution. Van Nostrand; 1963.
[7] Golub GH, Van Loan CF. Matrix computations. 3rd ed. The Johns Hopkins University Press; 1996.
[8] Guo Y. On a method of structural analysis. Appl Math Mech 1987;8(6):489–95.
[9] Lamport L. The PlusCal algorithm language. Theoretical aspects of computing — ICTAC 2009. Springer; 2009. p. 36–60.
[10] Liu S. On convergence of the Hardy Cross method of moment distribution. [Master''s thesis] North Carolina State University; 2014.
[11] Mawby M. Formulae for continous beams. Proc Inst Civil Eng 1968;41(2):339–50.
[12] McCormac JC. Structural analysis. 3rd ed. Intext Educational Publishers; 1975.
[13] Mozingo RR. Matrix distribution. J Struct Div Proc Am Soc Civ Eng 1968;94(ST4): 1043–52.
[14] Python software foundation. Welcome to Python.org. http://www.python.org (accessed 24.11.15).
[15] Saad Y. Iterative methods for sparse linear systems. 2nd ed. Philadelphia, PA: SIAM; 2003.
[16] Schneider S. The B-method: an introduction. Oxford: Palgrave; 2001.
[17] Turing AM. Computing machinery and intelligence. Mind 1950:433–60.
[18] Volokh KYu. On foundations of the Hardy Cross method. Int J Solids Struct 2002; 39(16):4197–200.
[19] Williamson T, Olsson RA. PySy: a python package for enhanced concurrent programming. Concurrency Comput Pract Exper 2014;26(2):309–35.