Verified Correctness, Accuracy, and Convergence of a Stationary Iterative Linear Solver: Jacobi Method

Mohit Tekriwal[1(✉)], Andrew W. Appel[2], Ariel E. Kellison[3], David Bindel[3], and Jean-Baptiste Jeannin[1]

[1] University of Michigan, Ann Arbor, USA
{tmohit,jeannin}@umich.edu
[2] Princeton University, Princeton, USA
appel@princeton.edu
[3] Cornell University, Ithaca, USA
{ak2485,bindel}@cornell.edu

Abstract. Solving a sparse linear system of the form $Ax = b$ is a common engineering task, e.g., as a step in approximating solutions of differential equations. Inverting a large matrix A is often too expensive, and instead engineers rely on iterative methods, which progressively approximate the solution x of the linear system in several iterations, where each iteration is a much less expensive (sparse) matrix-vector multiplication.

We present a formal proof in the Coq proof assistant of the correctness, accuracy and convergence of one prominent iterative method, the Jacobi iteration. The accuracy and convergence properties of Jacobi iteration are well-studied, but most past analyses were performed in real arithmetic; instead, we study those properties, and prove our results, in floating-point arithmetic. We then show that our results are properly reflected in a concrete implementation in the C language. Finally, we show that the iteration will not overflow, under assumptions that we make explicit. Notably, our proofs are faithful to the details of the implementation, including C program semantics and floating-point arithmetic.

Keywords: Formal Verification · Numerical Methods · Jacobi Method

1 Introduction

Many scientific and engineering computations require the solution x of large sparse linear systems $Ax = b$ given an $n \times n$ matrix A and a vector b. There are many algorithms for doing this; Gaussian elimination is rare when n is large, since it takes $\mathcal{O}(n^3)$ time. The widely used *stationary iterative methods* have an average time complexity of $\mathcal{O}(nsk)$, where sparseness s is the number of nonzeros per row (often $s \ll n$) and k is the number of iterations (often small). Even where iterative methods are not the principal algorithms for solving $Ax = b$, they are often used in transformations of the problem (preconditioning) before using other workhorses such as Krylov subspace methods [27].

When using a stationary iterative method, one starts with an initial vector x_0 and uses A and b to derive successive vectors $x_1, x_2, \ldots$ that—one hopes—will

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
C. Dubois and M. Kerber (Eds.): CICM 2023, LNAI 14101, pp. 206–221, 2023.
https://doi.org/10.1007/978-3-031-42753-4_14

converge to a value x_k such that the *residual* $Ax_k - b$ is small and x_k is close to the true solution. Because these methods are often used as subroutines deep within larger computational libraries and solvers, it is quite inconvenient to the end user if some such subroutine reports that it failed to converge—often, the user has no idea what is the subproblem A, b that has failed. Thus it is useful to be able to prove theorems of the following form: "Given inputs A and b with certain properties, the algorithm will converge to tolerance τ within k iterations."

Since these methods are so important, analyses of their convergence properties have been studied in detail. However, most of these analyses assume real number arithmetic operations [27], whereas their implementations use floating-point; or the analysis uses a simplified floating-point model that omits subnormal numbers [14]; or the analysis is for a *model* of an algorithm [19] but not the actual software. And when one reaches correctness and accuracy proofs of actual software, it's useful to have machine-checked proofs that connect in a machine-checkable way to the actual program that is executed, for programs can be complex and as programs evolve one must ensure that their correctness theorems evolve with them.

We focus on Jacobi iteration applied to *strictly diagonally dominant matrices*, i.e., in which in each row the magnitude of the diagonal element exceeds the sum of the magnitudes of the off-diagonals. Strict diagonal dominance is a simple test for invertibility and guarantees convergence of Jacobi iteration in exact arithmetic [27]. Strictly diagonally dominant matrices arise in cubic spline interpolation [1], analysis of Katz centrality in social networks [18], Markov chains associated with PageRank and related network analysis methods [12], and market equilibria in economic theory [24], among other domains.

We present both a Coq *functional model* of floating-point Jacobi iteration (at any desired floating-point precision) and a C program (in double-precision floating-point), with Coq proofs that:

- the C program (which uses efficient sparse-matrix algorithms) correctly implements the functional model (which uses a simpler matrix representation);
- for any inputs A, b and desired accuracy τ that satisfy the *Jacobi preconditions* for a given natural number k, the functional model (and the C program) will converge within k iterations to a vector x_k such that $||Ax_k - b||_2 < \tau$;
- this computation will not overflow into floating-point "infinity" values;
- and the *Jacobi preconditions* are natural properties of A, b, τ, k that (1) are easily tested and (2) for many natural engineering problems of interest (mentioned above), are guaranteed to be satisfied.

Software packages not written in C can still be related to our functional model and make use of our floating-point convergence and accuracy theorems. And even for inputs that do not satisfy the Jacobi preconditions, we have proved that our C program correctly and robustly detects overflow.

Together, these theorems guarantee that a Jacobi solver deep within some larger library will not be the cause of a mysterious "failed to converge" message; and that when it does believe it has converged, it will have a correct answer.

Contributions. First convergence proof of Jacobi that takes into account floating-point underflow or overflow; first machine-checked proof of a stationary iterative method; first machine-checked connection to a real program. Our Coq formalization is available at:

https://github.com/VeriNum/iterative_methods/tree/v0.1.0

2 Overview of Iterative Methods and Our Proof Structure

Stationary iterative methods [27] are among the oldest and simplest methods for solving linear systems of the form $Ax = b$, for $A \in \mathbb{R}^{n\times n},\ b \in \mathbb{R}^n$. The nonsingular matrix A and vector b in such systems typically appear, for example, in the solution of a partial differential equation. In stationary methods, matrix A is decomposed into $A = M + N$ where M is chosen such that it is easily invertible; for Jacobi it is simply the diagonal of A and we will often call it D. Rather than solving the system $Ax = b$ exactly, one can approximate the solution vector x using stationary iterations of the form

$$Mx_m + Nx_{m-1} = b, \tag{1}$$

where the vector x_m is an approximation to the solution vector x obtained after m iterations; we typically start with $x_0 = \mathbf{0}$. The unknown x_m is therefore

$$x_m = M^{-1}(b - Nx_{m-1}) \qquad \text{that is for Jacobi,} \qquad x_m = D^{-1}(b - Nx_{m-1}) \tag{2}$$

This iterates until x_k satisfies $\|Ax_k - b\|_2 < \tau$, or until the program detects *failure:* overflow in computing x_k, or maximum number of iterations exceeded. Throughout this paper, we let $\|\cdot\|$ denote the infinity vector norm and its induced matrix norm, and we let $\|\cdot\|_2$ denote the ℓ^2 norm on vectors.

For our model problem, the steps are as follows.

1. Write a C program that implements (2) by Jacobi iterations (and also implements an appropriate stopping condition).
2. Write a *floating-point functional model* in Coq (a recursive functional program that operates on floating-point values) that models Jacobi iterations of the form (2). This model must perform almost exactly (see Sect. 7.1) the same floating-point operations as the C program. (As we will explain, we have two statements of this model and we prove the two *models equivalent.*)
3. Prove that the program written in Step 1 implements the floating-point functional model of Step 2, using a program logic for C.
4. Write a *real functional model* in Coq that performs Jacobi iteration $x_m = D^{-1}(b - Nx_{m-1})$ in the exact reals. Of course, it is impractical to compute with this model, but it is useful for proofs.
5. Prove a relation between x_k (the k-th iteration of the floating-point model) and the real solution x of the real functional model: the *Jacobi forward error bound.* If one could run the Jacobi method purely in the reals, this is obviously contractive: $\|x_{k+1} - x\| < \rho\|x_k - x\|$, where $\rho < 1$ is the spectral radius of $D^{-1}N$. But in the floats, there is an extra term caused by roundoff error.

6. Prove *floating-point convergence:* under certain conditions (*Jacobi preconditions*), this extra term does not blow up, and within a specified k iterations the residual $\|Ax_k - b\|_2$ is less than tolerance τ.
7. Compose these to prove the main theorem: the C program converges to an accurate result.

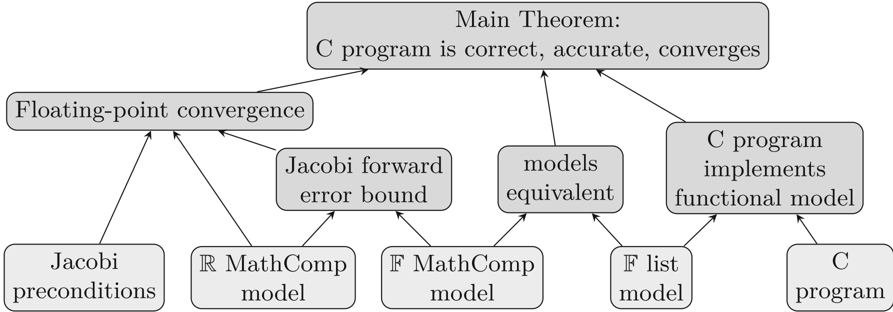

Fig. 1. Theorem dependency. Bottom row: models and definitions; middle row: theorems relating models.

Figure 1 shows our correctness and accuracy theorem as a modular composition of reusable models and lemmas. We have two float-valued models: for proving the relation of the float-valued model to the real solution we use the MathComp library (in Coq). For proving the C program implements the float-valued model, we use the Verified Software Toolchain (in Coq). But MathComp [22] and VST [10] prefer different notations and constructions for specifying data structures such as matrices and vectors; so we write the same functional model in each of the two styles, and prove their equivalence. We used Coq for our formalization because of Coq's expressive general-purpose logic, with powerful libraries such as MathComp, Flocq [8], VCFloat [3], LAProof [21], Coquelicot [7], VST, support both high-level mathematical reasoning and low-level program verification.

The float-valued model will experience round-off errors compared to the real-valued model; we prove bounds on these errors in a *forward error bound* lemma (Sect. 4). Under certain "Jacobi preconditions" the float-valued model is guaranteed to converge to a result of specified accuracy without ever overflowing; this is the *Jacobi iteration bound* lemma (Sect. 5).

3 Parametric Models and Proofs; Important Constants

We have proved accuracy bounds for any floating-point precision. That is, our floating-point functional models, and the proofs about them, are parameterized by a *floating-point type*, expressed in Coq as `type:Type`, with operations: [3]

```
fprec: type → ℤ          (* number of mantissa bits *)
femax: type → ℤ          (* maximum binary exponent *)
ftype: type → Type       (* floating–point numbers *)
```

So for t:type, we have x:ftype(t) meaning that x is a floating-point number in format t. We will write p for fprec(t) and $e_{\max}$ for femax(t). The maximum representable finite value is $F_{\max} = 2^{e_{\max}}(1 - 2^{-p})$. If $|r| \leq F_{\max}$ then rounding r to the nearest float yields a number f such that $f = r(1 + \delta_f) + \epsilon_f$, where $|\delta_f| \leq \delta = \frac{1}{2}2^{1-p}$ and $|\epsilon_f| \leq \epsilon = \frac{1}{2}2^{3-e_{\max}-p}$.

Given a floating-point format t and matrix-dimension n, the following functions will be useful in reasoning:

$$g_\delta(n) = (1 + \delta)^n - 1 \qquad\qquad g_\epsilon(n) = n\epsilon(1 + g_\delta(n - 1)) \tag{3}$$

For example, suppose t is double-precision floating-point ($p = 53,\ e_{\max} = 1024$), $n = 10^6$ (Jacobi iteration on a million-by-million matrix), $s = 5$ (the million-element *sparse* matrix has 5 nonzeros per row). Then some relevant quantities are,

$$\begin{array}{lll} \delta = 2^{-54} = 5.6 \cdot 10^{-17} & g_\delta(n) = 5.6 \cdot 10^{-11} & g_\epsilon(n) = 2.5 \cdot 10^{-318} \\ \epsilon = 2^{-1075} = 2.5 \cdot 10^{-324} & g_\delta(5) = 2.8 \cdot 10^{-16} & g_\epsilon(5) = 1.2 \cdot 10^{-323}. \end{array}$$

Our error analyses will often feature formulas with $\delta, \epsilon, g_\delta, g_\epsilon$; remember that in double-precision these are *small* quantities (in single- or half-precision, not so small). Henceforth we will write g_δ, g_ϵ for $g_\delta(n), g_\epsilon(n)$.

4 Forward Error Bound for Dot Product

In separate work, Kellison *et al.* [21] prove (in Coq) the correctness and accuracy of floating-point dot-product and sparse matrix-vector multiply, as Coq functional models and as C programs.

Define dot-product $\langle u, v\rangle$ between two real vectors u and v as $\sum_{0\leq i<n} u_i v_i$. A matrix-vector multiplication Av can be seen as the dot-product of each row of A with vector v. Forward error bounds for a matrix-vector multiplication are therefore based on forward error bounds for dot-product.

Our implementation and functional model of the dot-product use *fused multiply-add* (FMA), which computes a floating-point multiplication and addition (i.e., $a \otimes b \oplus c$) with a single rounding error rather than two.

```
Definition dotprod {t: type} (u v: list (ftype t)) : ftype t :=
  fold_left (fun z a ⇒ FMA (fst a) (snd a) z) (List.combine u v) (Zconst t 0).
```

The parameters to the dotprod functional model are the floating-point format t and two lists of floating-point numbers. The algorithm zips the two lists into a list of pairs (using List.combine) and then adds them from left to right, starting with a floating-point 0 in format t.

We denote floating-point summation by $\bigoplus$, so the floating-point dot product is $\bigoplus_{0\leq i<n} u_i v_i$; real-valued summation is denoted as $\sum_{0\leq i<n} u_i v_i$. The notation finite(z) signifies that the floating-point number z is within the range of the floating-point format (not an infinity or NaN).

Theorem 1 (forward error + no overflow). *Let u, v be lists of length n of floats in format $t = (p, e_{\max})$, in which every element is $\leq v_{\max}$, and no more than s elements of u are nonzero. The absolute forward error of the resulting dot product is*

$$\left| \bigoplus_{0 \leq i < n} u_i v_i - \sum_{0 \leq i < n} u_i v_i \right| \;\leq\; g_\delta(s) \sum_{0 \leq i < n} |u_i v_i| \;+\; g_\epsilon(s). \tag{4}$$

Proof. See Kellison *et al.* [21].

Subnormal Numbers. When some of the vector elements are order-of-magnitude 1, the term $g_\epsilon(s)$ is negligible. But if the u and v vectors are composed of subnormal numbers, then neglecting the underflow-error term would be unsound. Most previous proofs about dot-product error (and about Jacobi iteration), and all previous machine-checked proofs to our knowledge, omit reasoning about underflow.

5 Jacobi Forward Error

We prove an explicit bound on the distance between the approximate solution x_k at step k and the exact solution x of the problem. Such bounds are commonly studied in computational science, but rarely take into account the details of floating-point arithmetic (including underflow and overflow). They also usually have paper proofs while we provide a machine-checked proof.

Theorem 2 (jacobi_forward_error_bound). *After k Jacobi iterations, if no iteration resulted in an overflow, then the distance between the approximate (floating-point) solution x_k and the exact (real) solution x is bounded:*

$$\|x - x_k\| \leq \rho^k \|x - x_0\| + \frac{1 - \rho^k}{1 - \rho} d_{\text{mag}}$$

where ρ is a bound on the spectral radius (largest eigenvalue) of $D^{-1}N$, adjusted for floating-point roundoff errors that arise in one iteration of the Jacobi method. The (small) value d_{mag} is the floating-point roundoff error in computing the residual $\|Ax_k - b\|$. In Coq,

```
Theorem jacobi_forward_error_bound {ty} {n:nat}
 (A: 'M[ftype ty]_n.+1) (b: 'cV[ftype ty]_n.+1): ∀x0: 'cV[ftype ty]_n.+1,
  forward_error_cond A x0 b →
  ∀k i, finite (x_{k, i})) ∧
  (f_error k ≤ rho^k ∗ (f_error 0) + ((1 − rho^k) / (1−rho)) ∗ d_mag.
```

where f_error(k) is $\|x - x_k\|$, and the conditions on $A \in \mathbb{F}^{(n+1)\times(n+1)}, x_0 \in \mathbb{F}^{(n+1)\times 1}, b \in \mathbb{F}^{(n+1)\times 1}$ for $n \geq 0$, are characterized as follows:

```
Definition forward_error_cond {ty} {n:nat}
 (A: 'M[ftype ty]_n.+1) (x0 b: 'cV[ftype ty]_n.+1) :=
  (∀ i, finite (A _{i,i} ) ∧ (rho < 1) ∧ invertible A ∧ (∀ i, finite (1 / A_{i,i} )) ∧
  (∀ i, finite (x_{o,i} ) ∧ (∀ i j, finite(N_{i,j} )) ∧ (∀ i, finite (b_i) ∧
  size_constraint n ∧ input_bound A x0 b.
```

`size_constraint n` is a constraint on the dimension n of the matrix A (in double-precision, about $6 \cdot 10^9$). The predicate `input_bound` provides conditions on the bounds for the inputs A, b, x_0; it is implied by the *Jacobi preconditions* (Definition 1) defined in the next section.

Proof. Assuming no overflow, the floating-point iteration satisfies

$$x_{k+1} = \tilde{D}^{-1} \otimes (b \ominus (N \otimes x_k))$$

where the operators $\otimes$, $\ominus$ represent the floating-point multiplication and subtraction operations respectively. $\tilde{D}^{-1}$ is the floating-point (not exact) inverse of the diagonal elements. The forward error at step $k+1$ is defined as

$$f_{k+1} = \|x_{k+1} - x\| = \|(\tilde{D}^{-1} \otimes (b \ominus (N \otimes x_k))) - (D^{-1}(b - Nx))\|$$

Here, we write the true solution as $x = D^{-1}(b - Nx)$ which can be derived from $Ax = b$. To move forward, we will be using the following auxiliary error bounds on matrix-vector operations.

- Matrix-vector multiplication: $\|N \otimes x - Nx\| \leq \|N\|\|x\|g_\delta + g_\epsilon$. This bound is derived from the dot product error bound that we stated in Sect. 4. In the $\|\cdot\|$ norm, the dot product errors directly give the error bound for matrix-vector multiplication.
- Vector subtraction: $||(b \ominus (N \otimes x_k) - b(N \otimes x_k)|| \leq (||b|| + ||N \otimes x_k||)\delta$. Here, we use the fact that $|x \ominus y - (x - y)| \leq (|x| + |y|)\delta$.
- Vector inverse: $||\tilde{D}^{-1} - D^{-1}|| \leq ||D^{-1}||\delta + \epsilon$. Here, we make use of the fact that inverting each element of the vector D satisfies, $|(1 \oslash D_i) - (1/D_i)| \leq |(1/D_i)|\, \delta + \epsilon$.

We use these norm-wise errors to expand the error definition f_{k+1}:

$$\begin{aligned} f_{k+1} \leq (&(||\tilde{D}^{-1}||\ (||b|| + ||N||\ ||x_k||(1 + g_\delta + g_\epsilon))(1+\delta)g_\delta + g_\epsilon) + \\ &||\tilde{D}^{-1}||\ (||b|| + ||N||\ ||x_k||(1 + g_\delta) + g_\epsilon)\delta + \\ &||\tilde{D}^{-1}||\ (||N||\ ||x_k||g_\delta + g_\epsilon) + \\ &(||D^{-1}||\delta + \epsilon)\ (||b|| + ||N||\ ||x_k||) + ||D^{-1}||\ ||N||f_k \end{aligned}$$

We then collect the coefficients of $||x_k||$ and expand using the error relation for f_k as $||x_k|| \leq f_k + ||x||$ to get the error recurrence relation:

$$f_{k+1} \leq \rho f_k + d_{\text{mag}}$$

where d_{mag} is a constant independent of k depending only on $||x||$, δ, ϵ, g_δ, g_ϵ. We can then expand the above error recurrence relation to get

$$f_{k+1} \leq \rho^{k+1} f_o + (1 + \rho + \rho^2 + ... + \rho^k) d_{\text{mag}}$$

This geometric series converges if $\rho < 1$ with closed form

$$f_{k+1} \leq \rho^{k+1} f_o + \frac{1 - \rho^{k+1}}{1 - \rho} d_{\text{mag}}$$

Note that if the above iterative process were done in reals, then we would only require $\rho_r := ||D^{-1}N||$ to be less than 1. Thus, the presence of rounding errors forces us to choose a more conservative convergence radius ρ.

6 Convergence Guarantee: Absence of Overflow

Theorem 2 had the premise, "if no iteration resulted in an overflow." Most previous convergence theorems for Jacobi iteration [27] have been in the real numbers, where overflow is not an issue: multiplying two finite numbers cannot "overflow." Higham and Knight [14] proved convergence (but not absence of overflow), on paper, in a simplified model of floating-point (without underflows). Let us now state a theorem, for an accurate model of floating-point, *not* conditioned on absence of overflow.

Theorem 3 (jacobi_iteration_bound_lowlevel). *If the inputs A, b, desired tolerance τ, and projected number of iterations k satisfy the (efficiently testable) Jacobi preconditions, then the floating-point functional model of Jacobi iteration converges, within $j \leq k$ iterations, to a solution x_j such that $\|Ax_j - b\|_2 < \tau$, without overflowing.*

Proof. The proof for this theorem follows by the following cases:

- If A is diagonal then N is a zero matrix. Therefore, the solution vector at each iteration is given by the constant vector $x_k = \tilde{D}^{-1} \otimes b$. Hence, the solution of Jacobi iteration has already converged after the first step, assuming certain bounds on b and A implied by the *Jacobi preconditions*.
- If A is not diagonal but the vector b is small enough, Jacobi iteration has already converged, without even running the iteration.
- Suppose A is not diagonal and the vector b is not too small. Then
  (a) The residual does not overflow for every iteration $\leq j$. This follows from the *Jacobi preconditions* and Theorem 2.
  (b) We can calculate $k_{\min}$ such that the residual $< \tau$ within $k_{\min}$ iterations.

The *Jacobi preconditions* (Definition 1) are efficiently computable: a straightforward arithmetic computation with computational complexity linear in the number of nonzero matrix elements.

Definition 1 (jacobi_preconditions_Rcompute). *A, b, τ, k satisfy the* Jacobi preconditions *when:*

– *All elements of* A*,* b*, and* $\tilde{D}^{-1}$ *are finite (representable in floating-point);*
– A *is strictly diagonally row-dominant, that is,* $\forall i.\ D_{ii} > \sum_j |N_{ij}|$*;*
– $\tilde{\tau}^2$ *is finite;*
– $\tilde{\tau}^2 > g_\epsilon + n(1+g_\delta)(g_\epsilon) + 2(1+g_\delta)(1+\delta)\|D\|(\hat{d}_{\text{mag}}/(1-\hat{\rho}))^2$*;* $\tilde{\tau}^2 > n(1+g_\delta)(\|D\|(\|\tilde{D}^{-1}\|(\|A\|\hat{d}_{\text{mag}}/(1-\hat{\rho}) + g_\epsilon)(1+\delta)(1+g_\delta) + g_\epsilon)(1+\delta)(1+g_\delta) + g_\epsilon)^2 + g_\epsilon$*;*
– $k_{\min} \leq k$*;*
– $n < ((F_{\max} - \epsilon)/(1+\delta) - g_\epsilon - 1)/(g(n-1)+1)$*;* $n < F_{\max}/((1+g(n+1))\delta) - 1$
– $\forall i.\ |A_{ii}|(1+\hat{\rho})x_{\text{bound}} + 2\hat{d}_{\text{mag}}/(1-\hat{\rho}) + 2x_{\text{bound}} < v_{\max} - \epsilon)/(1+\delta)^2$*;*
– $\forall i, j.\ |N_{ij}| < v_{\max}$*;*
– $\forall i.\ |b_i| + (1+g_\delta)((2x_{\text{bound}} + \hat{d}_{\text{mag}}/(1-\hat{\rho}))\sum_j |N_{ij}|) + g_\epsilon < F_{\max}/(1+\delta)$*;*
– $\forall i.\ |\tilde{D}_{ii}^{-1}|(|b_i| + (1+g_\delta)(2x_{\text{bound}} + \hat{d}_{\text{mag}}/(1-\hat{\rho}))\sum_j |N_{ij}|) + g_\epsilon < F_{\max}/(1+\delta)$*;*
– $(1+\hat{\rho})x_{\text{bound}} + 2\hat{d}_{\text{mag}}/(1-\hat{\rho}) + 2x_{\text{bound}} < F_{\max}/(1+\delta)$*;*
– $\forall i.\ |b_i| < (F_{\max} - \epsilon)/(1+\delta)$*;*
– $\forall i.\ |\tilde{D}_{ii}^{-1}||b_i| < (F_{\max} - \epsilon)/(1+\delta)$*;*
– $\forall i.\ |\tilde{D}_{ii}^{-1}||b_i|(1+\delta) + \epsilon < (F_{\max} - \epsilon)/(1+\delta)$*;*
– $\forall i.\ |A_{ii}|(|\tilde{D}_{ii}^{-1}||b_i|(1+\delta) + \epsilon) < (v_{\max} - \epsilon)/(1+\delta)$*.*

where $\hat{d} = (\|\tilde{D}^{-1}\| + \epsilon)/(1-\delta)$ is a bound on $\|D^{-1}\|$. Defining $R = \hat{d}\,\|N\|$, we define an upper bound on the norm of the solution x to $Ax = b$ as $x_{\text{bound}} = \hat{d}\|b\|/(1-R)$. $\hat{\rho}$ is the adjusted spectral radius ($\rho_r = \|D^{-1}N\|$) of the iteration matrix, obtained by accounting for the floating-point errors in its computation. For the iteration process to converge in presence of rounding, we want $\hat{\rho} < 1$. $\hat{d}_{\text{mag}}$ is a bound on the additive error in computing the residual $\|Ax_k - b\|$, the difference between computing the residual in the reals versus in floating-point. $\tilde{\tau}^2 = \tau \otimes \tau$ is the floating-point square of τ. The minimum k for which we guarantee convergence is calculated as

$$k_{\min} = 1 + \left\lceil \frac{\ln\left(\frac{x_{\text{bound}}(1+\delta)}{((\sqrt{(\tilde{\tau}^2 - g_\epsilon)/(n(1+g_\delta))} - g_\epsilon)/((1+g_\delta)+\|D\|(1+\delta))) - 2\hat{d}_{\text{mag}}/(1-\hat{\rho})}\right)}{\ln(1/\hat{\rho})} \right\rceil$$

Indeed these conditions are quite tedious – one might have difficulty trusting them without a machine-checked proof. But they are all easy to compute in linear time. And, although we state them here (mostly) in terms of operations on the reals, they are all straightforwardly boundable by floating-point operations.

Remark 1 (not proved in Coq). The Jacobi preconditions can be computed in time linear in the number of nonzero entries of A.

Proof. Let S be the number of nonzeros. Then $n < S$ since the diagonal elements are nonzero. The inverse diagonal $\tilde{D}^{-1}$ is computed in linear time. The infinity norm ($\|N\|, \|D\|, \hat{d}, \|b\|$) is simply the largest absolute value of any row-sum (for matrix) or element (for vector), which can be found in $O(S)$ time. Then the values $x_{\text{bound}}, \hat{\rho}, \hat{d}_{\text{mag}}, v_{\max}, k_{\min}$ can all be computed in constant time. Then each of the tests in Definition 1 can be done in $O(S)$ time.

7 An Efficient and Correct C Program

Our C program uses standard numerical methods to achieve high performance and accuracy: sparse matrix methods, fused multiply-add, efficient testing for overflow, and so on. What's not so standard is that we have proved it correct, with a foundational machine-checked proof that composes in Coq with the numerical accuracy (and convergence) proof of our functional model.

7.1 Sparse Matrix-Vector Multiply

Many implementations of stationary iterative methods, including Jacobi, are on large sparse matrices. A naive dense matrix-vector multiply would take $O(n^2)$ time per iteration, while sparse representations permit $O(sn)$ time per iteration, there are s nonzeros per row. Our program uses Compressed Row Storage (CRS), a standard sparse representation [4, §4.3.1].

Kellison *et al.* [21] describe the Coq floating-point functional model, an implementation in C, and the Coq/VST proof that the C dot-product program correctly implements the model. VST (Verified Software Toolchain) [10] is a tool embedded in Coq for proving C programs correct. From that, here we prove that the Jacobi program implements *its* model.

For sweep-form Jacobi iteration it is useful to have a function that computes *just one row* of a sparse matrix-vector multiply, which in CRS form is implemented as,

```
double crs_row_vector_multiply(struct crs_matrix *m, double *v, unsigned i);
 /* compute dot-product of row i of matrix m with vector v */
```

Separation of Concerns. The floating-point *accuracy* proof should be kept completely separate from the sparse-matrix data-structure-and-algorithm proof. This function is proved to calculate (almost) exactly the same *floating-point* computation as the naive dense matrix multiply algorithm.

Almost exactly – because where $A_{ij} = 0$, the dense algorithm computes $A_{ij} \cdot x_i + s$ where the sparse algorithm just uses s. In floating-point it is not the case that $\forall y.\ 0 \cdot y + s = s$, for example when y is ∞ or NaN. Even when y and s are finite, it is not always true that $y \cdot 0 + s$ is the same floating-point value as s: it could be that one is $+0$ and the other is -0. And finally, even when matrix A and vector x are all finite, we cannot assume that intermediate results s are finite—there may be overflow.

So we reason modulo equivalence relations (using Coq's Parametric Morphism system for rewriting with partial equivalence relations). We define feq x y to mean that either both x and y are finite and equal (with $+0 = -0$), or neither is finite. Our function will have a precondition that A and x are all finite, and postcondition that the computed result is feq to the result that a dense matrix multiply algorithm would compute.

7.2 Jacobi Iteration

Listing 1.1. C program for a single iteration of Jacobi iteration

```
double jacobi2_oneiter(double *A1, struct crs_matrix *A2, double *b, double *x,
                                                              double *y) {
  unsigned i, n=crs_matrix_rows(A2); double s = 0.0;
  for (i=0; i<n; i++) {
    double u = b[i] - crs_row_vector_multiply(A2,x,i);
    double a1 = A1[i], new = (1/a1)*u, r = a1*(new - x[i]);
    s = fma(r,r,s);
    y[i] = new;
    }
  return s;
}
```

The C program in the Lisiting 1.1 loops over rows i of the matrix, which is also elements i of the vectors b and x. For each i it computes a new element y_i of the result vector, as well as an element r_i of residual vector. It returns s, the sum of the squares of the r_i. By carefully computing r_i from y_i, and not vice versa, we can prove (in Coq, of course) that all overflows are detected: if s is finite, then all the y_i must be finite.

The program in the Listing 1.2 runs until convergence ($s < \tau^2$), giving up early if there's overflow (tested by s*0=0.0, since if s overflows then s*0 is NaN) or if maxiter iterations is reached.

Listing 1.2. C program for Jacobi iteration until convergence

```
double jacobi2(double *A1, struct crs_matrix *A2, double *b, double *x, double τ²,
                                                         unsigned maxiter) {
  unsigned i, n=crs_matrix_rows(A2);
  double s, *t, *z=x, *y = (double *)surely_malloc(n*sizeof(double));
  do { s = jacobi2_oneiter(A1,A2,b,z,y);
       t=z; z=y; y=t;
       maxiter--;
  } while (s*0==0.0 && s ≥ τ² && maxiter);
  if (y==x) y=z; else { for (i=0; i<n; i++) x[i]=y[i]; }
  free(y);
  return s;
}
```

This program starts with $x^{(0)}$ in x, computes $x^{(1)}$ into y, then $x^{(2)}$ back into x, and so on. It mallocs y for that purpose and frees it at the end. Depending on whether the number of iterations is odd or even, it may need to copy from y to x at the end.

This program is conventional and straightforward. Our proof tools allow the numerical analyst to use standard methods and idioms—and nontrivial data structures—and still get an end-to-end correctness proof. For each of these functions we prove in VST that the function exactly implements the functional model,

modulo equivalence relations on floating-point numbers. At this level there are no accuracy proofs, the programs *exactly* implement the functional models, except that one might have -0 where the other has $+0$, and one might have different NaNs than the other, if any arise.

Correctness Theorem. The jacobi2 function is specified and proved with a VST function-spec that we will not show here, but in English it says,

Theorem 4 (body_jacobi2). *Let A be a matrix, let b and $x^{(0)}$ be vectors, let* A1p *be the address of a 1-dimensional array holding the diagonal of A, let* A2p *be the address of a CRS sparse matrix representation of A without its diagonal, let* bp *and* xp *be the addresses of arrays holding b and $x^{(0)}$, let τ be desired residual accuracy, and let* maxiter *be an integer. Suppose these* ***preconditions*** *hold: the dimension of A is $n \times n$, b and $x^{(0)}$ have length n, $0 < n < 2^{32}$, $0 <$* maxiter *$< 2^{32}$, all the elements of A, b, x, acc^2 (as well as the* inverses *of A's diagonal) are finite double-precision floating-point numbers; the data structures* A1p,A2p,bp *have read permission and* xp *has read/write permission. Suppose one calls* jacobi2(A1p,A2p,bp,xp,acc,maxiter)*; then afterward it will satisfy this* ***postcondition:*** *the function will return some s and the array at* xp *will contain some $x^{(k)}$, such that $(s, x^{(k)}) \simeq$* jacobi $A\, b\, x\, \tau^2$, maxiter*, where $\simeq$ is the floating-point equivalence relation and* jacobi *is our functional model in Coq of Jacobi iteration; and the data structures at* A1p,A2p,bp *will still contain their original values.*

8 The Main Theorems, Residuals, and Stopping Conditions

The C program jacobi2() (and its functional model jacobi) satisfies either of two different specifications:

Theorem 4 (above): if A, b, x satisfy the *basic preconditions*[1] then perhaps Jacobi iteration will return after maxiter iterations—having failed to converge—or might overflow to floating-point infinities and stop early. But even so, the result (s, y) will be such that the (squared) residual $s = |Ay - b|_2^2$ accurately characterizes the result-vector y: if y contains an ∞ then $s = \infty$, but if $\sqrt{s} < \tau$ then y is indeed a "correct" answer. That's because the *functional model* preserves infinities in this way, and the C program correctly implements the model.

Theorem 5: if $A, b, x,$maxiter satisfy the *Jacobi preconditions* then the result (s, y) will be such that $s = |Ay - b|_2^2$, and $\sqrt{s} < \tau$ and indeed y is a "correct" finite answer. In fact this is our main result:

Theorem 5 (main_jacobi). *If the inputs satisfy the Jacobi preconditions, then the C program will converge within k iterations to an accurate result.*

Proof. Using Theorems 3 and 4, with some additional reasoning about the stopping condition in the functional model of the C program.

[1] A an $n \times n$ matrix; b and x dimension n; $0 < n < 2^{32}$; A, b, x all finite; A, b, x stored in memory in the right places—but nothing else about the *values* of A, b, x.

Jacobi Iteration on Inputs Not Known to Satisfy the Jacobi Preconditions. Theorem 4 is useful on its own, since there are many useful applications of stationary iterative methods where one has not proved in advance the convergence conditions (e.g., Jacobi preconditions)—one just runs the program and tests the residual. For such inputs we must take care to correctly stop on overflow.

The induction hypothesis, for $0 < k \leq \mathsf{maxiter}$ iterations, requires that x_k has not yet overflowed, otherwise our sparse-matrix reasoning cannot be proved (see Sect. 7.1). Therefore the program must check for floating-point overflow in x_k after each iteration. In order to do that efficiently, the program tests $s \otimes 0 = 0$ (which is a portable and efficient way of testing that s is finite); and if so, then x_{k+1} is all finite.

9 Related Work

Convergence of Jacobi Iteration. The standard error analysis for Jacobi iteration in exact arithmetic is well-described in standard books on iterative solvers [27]. A floating-point error analysis of Jacobi and related iterations was carried out by Higham and Knight in the early 90s [14], and is summarized along with references to related work in [13, Ch. 17]. The style of analysis is similar to what we present in this paper. However, earlier analyses implicitly assumed that all intermediates remain in the normalized floating-point range, and did not consider the possibility of underflow or overflow.

Formalization of numerical analysis has been facilitated by advancements in automatic and interactive theorem proving [7,11,23,25]. Some notable works in the formalization of numerical analysis are the formalization of Kantorovich theorem [26], matrix canonical forms by Cano et al. [9], Perron-Frobenius theorem in Isabelle/HOL [30], Lax–equivalence theorem for finite difference schemes [28], consistency, stability and convergence of a second-order centered scheme for the wave equation [5,6], formalized flows, Poincaré map of dynamical systems, and verified rigorous bounds on numerical algorithms in Isabelle/HOL [15–17]. However, these works do not study the problem of iterative convergence formally. Even though the iterative convergence has been formalized in exact arithmetic [29], the effect of rounding error on iterative convergence has not been formalized before.

End-to-End Machine-Checked Proofs. There are few truly end-to-end (C code to high-level accuracy) machine-checked formal proofs in the literature of numerical programs. Our approach has been to prove that a C program exactly implements a functional model (using VST), prove how accurately the functional model approximates a real-valued model, prove the accuracy of the real-valued model; and compose these results together. Something similar has been done for scalar Newton's method [2] and for ordinary differential equations [20], but those works did not leverage the power of the Mathematical Components and MathComp Analysis libraries for the upper-level proofs.

Other previous work [6] verified a C program implementing a second-order finite difference scheme for solving the one-dimensional acoustic wave equation, with a total error theorem in Coq that composes global round-off and discretization error bounds; this was connected (outside of Coq) to a Frama-C correctness proof of the C program. The inexpressiveness of Frama-C's assertion language was a challenge in that verification effort.

10 Conclusion and Future Work

In this paper, we have presented a formal proof in Coq of the correctness, accuracy, and convergence of Jacobi iteration in floating-point arithmetic. The same type of analysis should generalize to many other iterative methods, for both linear and nonlinear problems. Even within the scope of stationary iterations for linear problems, there are several avenues for future work.

We have not fully taken advantage of **sparseness** in our error bound; many of our $g_\delta(n)$ and $g_\epsilon(n)$ could be $g_\delta(s), g_\epsilon(s)$—functions of the number of nonzero elements per row. It would be useful to tighten the bound.

Jacobi iteration is one of the simplest stationary iterative methods. More complicated stationary methods involve splittings $A = M + N$, where M is **not a diagonal matrix.** Solving linear systems with such M is more complicated than diagonal scaling, and requires algorithms like forward and backward substitution that may require their own error analysis. Formalizing the floating-point error analysis of these solvers is an important next step.

Strict diagonal dominance is not a necessary condition for the convergence of Jacobi. We would like to formalize **other sufficient conditions for convergence** of Jacobi and related iterations. For example, irreducible diagonal dominance is sufficient for convergence of Jacobi in general, while positive definiteness is sufficient for convergence of Gauss-Seidel. However, while these conditions guarantee convergence, they do not guarantee an easy-to-compute *rate* of convergence in the same way that strict diagonal dominance does, and hence we expect the analysis to be more subtle.

Finally, we would like to extend our analysis to **mixed precision** methods, in which computations with M are done in one precision, while the residual is computed in a higher precision. These methods are often used to get errors associated with a higher floating-point precision while paying costs associated with a lower precision. However, the use of multiple precisions opens the door for a host of potential problems related to overflow and underflow, and we see formal verification as a particularly useful tool in this setting.

Efforts and Challenges: The formalization effort in this work includes about 1,826 lines of Coq proof script for C program verification and about 14,000 lines of Coq proof script for the convergence and accuracy proofs. A total of about 60 lines of C code were verified, which includes 12 lines for header files, 5 lines for `surely_malloc`, 14 lines of `crs_row_vector_multiply`, and about 31 lines for `jacobi2` and `jacobi2_oneiter`. It took us about 5 person-months for the formalization of accuracy and convergence of the functional model. The most time-consuming

part was the proof of absence of overflow, which involved deriving and formalizing bounds on the input conditions. The proof of correctness of C programs with respect to the functional model, including developing an understanding of what the termination condition should be, determining best ways to compute residuals such that the program properly detects overflow, developing functional models and proving properties about termination of the functional model took us about a couple of weeks.

The main challenge, in an end-to-end verification, is that each layer's theorem must be sufficiently strong to compose with the next layer. The published theorems about Jacobi convergence were insufficient (no treatment of underflow, error bounds relating the wrong quantities, no handling of -0, inadequate treatment of overflow), and new methods were required, which we address in this work.

Acknowledgement. We thank Yves Bertot for feedback on earlier drafts of this paper. This research was supported in part by NSF Grants CCF-2219997 and CCF-2219757, by a US Department of Energy Computational Science Fellowship DE-SC0021110, and by the Chateaubriand fellowship program.

References

1. Ahlberg, J.H., Nilson, E.N.: Convergence properties of the spline fit. J. Soc. Indust. Appl. Math. **11**, 95–104 (1963)
2. Appel, A.W., Bertot, Y.: C-language floating-point proofs layered with VST and Flocq. J. Formalized Reason. **13**(1), 1–16 (2020)
3. Appel, A.W., Kellison, A.E.: VCFloat2: floating-point error analysis in Coq (2022). https://github.com/VeriNum/vcfloat/blob/master/doc/vcfloat2.pdf
4. Barrett, R., et al.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. SIAM (1994)
5. Boldo, S., Clément, F., Filliâtre, J.C., Mayero, M., Melquiond, G., Weis, P.: Wave equation numerical resolution: a comprehensive mechanized proof of a C program. J. Autom. Reason. **50**(4), 423–456 (2013)
6. Boldo, S., Clément, F., Filliâtre, J.C., Mayero, M., Melquiond, G., Weis, P.: Trusting computations: a mechanized proof from partial differential equations to actual program. Comput. Math. Appl. **68**(3), 325–352 (2014)
7. Boldo, S., Lelay, C., Melquiond, G.: Coquelicot: a user-friendly library of real analysis for Coq. Math. Comput. Sci. **9**(1), 41–62 (2015)
8. Boldo, S., Melquiond, G.: Flocq: a unified library for proving floating-point algorithms in Coq. In: 2011 IEEE 20th Symposium on Computer Arithmetic, pp. 243–252. IEEE (2011)
9. Cano, G., Dénès, M.: Matrices à blocs et en forme canonique. In: Pous, D., Tasson, C. (eds.) JFLA - Journées francophones des langages applicatifs. Aussois, France (2013). https://hal.inria.fr/hal-00779376
10. Cao, Q., Beringer, L., Gruetter, S., Dodds, J., Appel, A.W.: VST-Floyd: a separation logic tool to verify correctness of C programs. J. Autom. Reason. **61**(1–4), 367–422 (2018)
11. Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: Packaging mathematical structures. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs

2009. LNCS, vol. 5674, pp. 327–342. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_23
12. Gleich, D.F.: Pagerank beyond the web. SIAM Rev. **57**(3), 321–363 (2015)
13. Higham, N.J.: Accuracy and Stability of Numerical Algorithms. SIAM (2002)
14. Higham, N.J., Knight, P.A.: Componentwise error analysis for stationary iterative methods. In: Meyer, C.D., Plemmons, R.J. (eds.) Linear Algebra, Markov Chains, and Queueing Models, pp. 29–46. Springer, New York (1993). https://doi.org/10.1007/978-1-4613-8351-2_3
15. Immler, F.: A Verified ODE Solver and Smale's 14th Problem. Dissertation, Technische Universität München, München (2018)
16. Immler, F., Hölzl, J.: Numerical analysis of ordinary differential equations in Isabelle/HOL. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 377–392. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32347-8_26
17. Immler, F., Traut, C.: The flow of ODEs: formalization of variational equation and Poincaré map. J. Autom. Reason. **62**(2), 215–236 (2019)
18. Katz, L.: A new status index derived from sociometric analysis. Psychometrika **18**(1), 39–43 (1953)
19. Kellison, A., Tekriwal, M., Jeannin, J.B., Hulette, G.: Towards verified rounding error analysis for stationary iterative methods. In: 2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness), pp. 10–17 (2022). https://doi.org/10.1109/Correctness56720.2022.00007
20. Kellison, A.E., Appel, A.W.: Verified numerical methods for ordinary differential equations. In: 15th International Workshop on Numerical Software Verification (2022)
21. Kellison, A.E., Appel, A.W., Tekriwal, M., Bindel, D.: LAProof: a library of formal accuracy and correctness proofs for sparse linear algebra programs (2023). https://www.cs.princeton.edu/~appel/papers/LAProof.pdf
22. Mahboubi, A., Tassi, E.: Mathematical components. Online book (2021)
23. Martin-Dorel, É., Rideau, L., Théry, L., Mayero, M., Pasca, I.: Certified, efficient and sharp univariate Taylor models in Coq. In: 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, pp. 193–200. IEEE (2013)
24. McKenzie, L.: Matrices with dominant diagonals and economic theory. In: Arroa, K., Karlin, S., Puppes, S. (eds.) Mathematical Methods in the Social Sciences, pp. 47–60. Stanford University Press (1960)
25. O'Connor, R.: Certified exact transcendental real number computation in Coq. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 246–261. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71067-7_21
26. Pasca, I.: Formal verification for numerical methods. Ph.D. thesis, Université Nice Sophia Antipolis (2010)
27. Saad, Y.: Iterative Methods for Sparse Linear Systems. SIAM (2003)
28. Tekriwal, M., Duraisamy, K., Jeannin, J.-B.: A formal proof of the lax equivalence theorem for finite difference schemes. In: Dutle, A., Moscato, M.M., Titolo, L., Muñoz, C.A., Perez, I. (eds.) NFM 2021. LNCS, vol. 12673, pp. 322–339. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-76384-8_20
29. Tekriwal, M., Miller, J., Jeannin, J.B.: Formal verification of iterative convergence of numerical algorithms (2022). https://doi.org/10.48550/arXiv.2202.05587
30. Thiemann, R.: A Perron-Frobenius theorem for deciding matrix growth. J. Log. Algebraic Methods Program. 100699 (2021)