SAND2023-00957J

Incremental Interval Assignment by Integer Linear Algebra with Improvements

Scott A. Mitchell

Center for Computing Research, Sandia National Laboratories

Abstract

Interval Assignment (IA) is the problem of selecting the number of mesh edges (intervals) for each curve for conforming quad and hex meshing. The intervals x is fundamentally integer-valued. Many other approaches perform numerical optimization then convert a floating-point solution into an integer solution, which is slow and error prone. We avoid such steps: *we start integer, and stay integer.* Incremental Interval Assignment (IIA) uses integer linear algebra (Hermite normal form) to find an initial solution to the meshing constraints, satisfying the integer matrix equation $Ax = b$. Solving for reduced row echelon form provides integer vectors spanning the nullspace of A. We add vectors from the nullspace to improve the initial solution, maintaining $Ax = b$. Heuristics find good integer linear combinations of nullspace vectors that provide strict improvement towards variable bounds or goals. IIA always produces an integer solution if one exists. In practice we usually achieve solutions close to the user goals, but there is no guarantee that the solution is optimal, nor even satisfies variable bounds, e.g. has positive intervals. We describe several algorithmic changes since first publication that tend to improve the final solution. The software is freely available.

Keywords: mesh generation, intervals, integer, optimization, linear algebra

1. Introduction

Intervals is the number of mesh edges on a curve. Interval Assignment (IA) means deciding the intervals on curves so the adjoining surfaces and volumes can be meshed compatibly. This is a non-issue for simplicial meshing, because any number can be chosen for each curve, and there will be some conformal mesh of each surface and volume. However, quad element topology places fundamental constraints on the number of boundary edges [1]. All manifold quad meshes are bounded by an even number of intervals. Certain meshing algorithms impose additional constraints; see figs. 1 and 2 for some examples. Equality constraints arise from structured meshing schemes, such as mapping with a rectangular grid of quads [2], and from requiring volume sweep paths to have positive and consistent lengths [3]. Midpoint subdivision [4] (related to the discrete structure of Catmull-Clark subdivision [5]) imposes a form of triangle inequality. The constraints also depend on algorithm parameters, e.g. when mapping a surface, one may choose which curves comprise each of the four sides. See section 2.1 for the formulas for the three most common types of constraints.

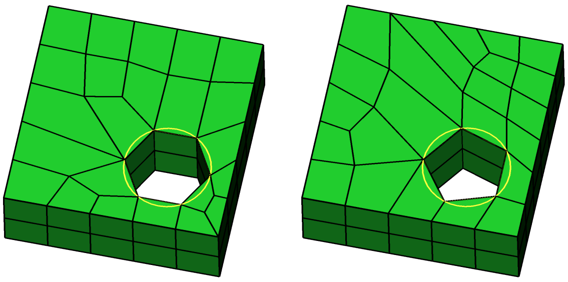


Figure 1: Swept hex meshes. The side surfaces are mapped: opposite sides have equal intervals. For volume sweeping constraints, the intervals through the hole must be the same as on the outside; here two. The top surface is meshed with paving. Any quad mesh must have an even number of intervals. CUBIT's [6] paving algorithm [7] is more robust if the constraints are more restrictive: each boundary component must be even, not just all of them in total. CUBIT will create the mesh on the left, with 6 intervals around the hole, not the right with 5 intervals, unless forced.

Interval assignment is important for automation and meshing independence, and also for mesh quality. Conforming meshes of assemblies, or even just single parts,

Email address: samitch@sandia.gov (Scott A. Mitchell)
URL: {www.sandia.gov/~samitch} (Scott A. Mitchell)

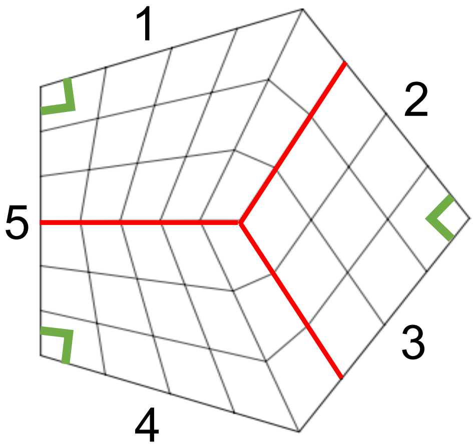


Figure 2: Midpoint-subdivision of a pentagon using a three-sided primitive. The three sides are between the right-angle marks: $A = \{1, 2\}$, $B = \{3, 4\}$, and $C = \{5\}$.

must agree on how many edges to place on each shared curve. The meshing constraints form a globally-coupled system of linear equations over integer variables; half an edge is nonsense. The problem is to assign a natural number of intervals to each curve satisfying all constraints and bounds, and ideally close to user-desired goals. Once IA is solved, each surface and volume can be meshed independently and compatibly.

1.1. Contribution

Incremental Interval Assignment (IIA) is a discrete algorithm over integers, the first IA method using integer linear algebra and nullspaces. IIA uses two passes: a "mapping" or equality phase, and a "sum-even" phase. The first pass relaxes the problem by ignoring any even-interval constraints; the second pass re-introduces them and uses the first-pass solution as a warm-start. Independent subproblems are solved independently. In each pass, IIA uses variants of Gaussian elimination to find an initial integer solution that satisfies $Ax = b$, and integer nullspace vectors spanning $Ax = 0$. Combinations of nullspace vectors are added to the current solution, first to satisfy variable bounds such as $x \geq 1$, then to find a solution close to the user's goals.

IIA scales well, is more reliable, and produces better quality output, than the prior approach of numerical optimization followed by Branch & Bound (BB) for integerization. IIA runs at interactive speeds, less than one second for reasonably-sized inputs.

The author presented an earlier version of this algorithm at the 2021 International Meshing Roundtable (IMR) [8].

1.2. Prior Approaches

The interval assignment problem is deceptively simple. After all, we are only considering edges, and for most surface and volume meshing algorithms the constraints are straightforward. Some constraints, such as requiring an even number of intervals on a paved (unstructured quad) surface seem so mild and there are so many solutions that finding one of them should be easy. Indeed, it *is* easy for humans to look at one surface and pick some intervals by inspection. The difficulty arises when the model is large and the global system of constraints conspires against us. It is tempting to assign intervals to surfaces one by one, but this can fail by "painting yourself into a corner," e.g., leaving a remaining surface unmeshable because it has an odd number of intervals on its boundary. For a manifold topology composed of many surfaces, the parameters (e.g. mapping sides) can couple the constraints for two distant surfaces in a non-obvious way. The combinatorial difficulty increases for 3D non-manifold models. A single curve can be in many surfaces, and couples the chains of constraint fanning out from all of them. Meshing the adjoining volumes brings in yet more constraints. A global problem must be solved. For this problem the constraints are standard and necessary, but the objective is a matter of mesh quality and there is some flexibility in how to define it.

Interval assignment methods fall into several categories. Numerical optimization is a common approach, e.g., floating-point linear or nonlinear programming followed by integerization with branch and bound, branch and cut, or some other technique. The key challenge for floating point methods is obtaining an integer solution. Greedy algorithms select the worst constraint violation, then adjust the intervals to move closer to feasibility. Once feasible, the worst quality can be improved while maintaining feasibility.

1.2.1. Mesh Structure Interdependence

In many cases, such as our IIA, methods assume that the mesh structure is given, and the only remaining degrees of freedom are the intervals. Other methods combine IA with selecting the mesh structure. This may be as limited as deciding where to put the four corners in a five-sided surface. Network flows combine IA with selecting the meshing templates within rectangular surfaces [9].

In the extreme, IA methods have the freedom to define the structured patches themselves. This approach is common for smooth closed surfaces for some computer graphics models [10]. Frame field methods combine

IA with partitioning the domain into structured quad patches [11, 12, 13]. A *frame field* is a mathematical field that defines a frame at each point of the domain, where a frame is two orthogonal vectors in the local tangent plane. A frame can be viewed as the dual of a quadrilateral, so the field indicates the ideal quad orientation. Frame fields are found by solving a PDE (Partial Differential Equation). Then the numeric solution is "integerized" by selecting discrete points representing the centers (duals) of patches of quads, and matching the frames of nearby points to indicate a shared quad-patch edge. The matching introduces "singularities" (non-4 valent surface mesh vertices) wherever a patch vertex is not surrounded by 4 quads patches, e.g. 3, 5, or 6. As with the network flow approach and CFD (Computational Fluid Dynamics) meshing, a patch is not limited to being mapped, and can be meshed with a variety of templates. The choice of template is intertwined with IA, in that the choice changes the IA problem and may even change its feasibility.

CUBIT [6] has a way to automatically select which meshing algorithm ("scheme") to use on each surface and volume [14]. It uses IA as part of that process. All CUBIT IA solvers assume that the mesh structure and scheme are fixed. However, candidate schemes are fed to IA, and the feasibility and quality of the IA solution determines which candidate to ultimately use. IA is run on each surface individually for each available meshing scheme, starting from the most restrictive. E.g., if the mapping IA is infeasible, then a less structured surface meshing scheme should be tried, say submapping; if the submapping IA solution quality is poor, then we should select an unstructured scheme like Paving [7]. In a similar way, IA is used to adjust the corners [15] of surfaces, and edge types between surfaces, to set up the structure of swept volumes [14, 3, 16].

1.2.2. Numerical Optimization

Tam and Armstrong in 1993 [2] described IA as an optimization problem with linear constraints $Ax \leq b$. Their choice of objective is also linear, a weighted sum of differences between the goals and assigned intervals: $\min_x w^T(x-g)$ for constant vectors w and g. The weights are inversely proportional to the goals. Intervals are bounded below by the goals, and unbounded above.

The potential upside to Tam and Armstrong's objective is that the simplex method's solution is integer "for free," without recourse to expensive integerization techniques, in many situations. It helps if the weights are unique and the goals are integer. However, the variables must have relatively-prime coefficients in A. This is broken by sum-even constraints. It is also broken when the global structure of an assembly conspires to link constraints, such as the "radish" in fig. 3. Section 8.2 compares simplex plus Branch & Bound (BB) to IIA over these problems.

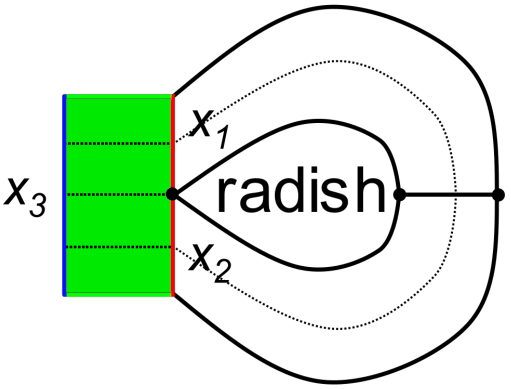


Figure 3: The "2-1 radish" assembly model circa 1997 from Mitchell [17]. A *radish* is a mapped surface where opposite sides have different numbers of curves, and all curves on a single side must have the same number of intervals. In this example it arises from the global structure of the assembly. The curves on the red side are linked by mapping constraints to all have the same intervals: $x_1 = x_2$. Thus $x_3 = x_1 + x_2$ reduces to $2x_1 = 1x_3$. Other combinations are possible.

The potential downside is all of the deviations from goals may be concentrated into a few curves. This is because the objective is linear in the deviations, and it is common for L_1 minimization solutions to be sparse [18]. However, the authors of [2] do not observe drastic deviations and concentrations in practice, per personal communication.

BBIA. "High Fidelity Interval Assignment" (BBIA for Branch & Bound IA) [17] introduces an objective function designed to distribute any potential concentration: lexicographically minimize the maximum weighted deviations. By lexicographic, we mean minimize the maximum weighted deviation, remove that variable from the problem by fixing it at its current value, and recurse. Such optimization objectives are known as *lex min-max.* In principle, one can simply define the BBIA problem and call a Linear Program (LP) solver with a Branch & Bound (BB) postprocess as a black box. In practice, the runtime of the integerization step is prohibitive. As is typical of large optimization problems, exploiting the problem structure was key.

In the first pass, we ignore the *sum-even* constraints that the number of intervals bounding a surface must be an even number (see eq. (1) in section 2.1). We do this because these are relatively nonrestrictive and removing them often allows the global problem to be broken up into many smaller problems. The LP finds a floating point solution. We identify the variables stuck at the maximum deviation, and use BB to force those to integer values. These are removed from the problem and the process is repeated.

In the second pass, all constraints must be satisfied. The integer solution from the first pass guides the LP re-solve and subsequent BB. We define upper and lower bounds on the integer variables containing the first-pass solution. If an integer solution cannot be found quickly enough, the bounds are widened and we try again.

It may be possible to update the BBIA method to use modern solvers. Many current multi-objective optimization methods are based on the same ideas of solving a series of optimization problems. There are specialized lex min-max solvers, but these problems are still generally expensive [19].

The general outline of IIA has some similarities to BBIA. IIA's objective is also lex min-max, but of a nonlinear function of each deviation. IIA uses two passes, the first one ignoring the sum-even constraints. Within a pass, IIA successively concentrates on the worst-valued variables.

NLIA. (NonLinear IA) [20] sought to improve the speed and robustness of BBIA by switching the lex min-max objective to a sum-of-cubes objective. This sped up runtime, at the price of the optimal floating point solution being farther from the goals. However, once it is found, we switch the objective to a piecewise linear function in a local neighborhood around it. The idea is to exploit the same L_1 minimization integers-for-free advantage as Tam and Armstrong [2], but keep it local to avoid large deviations. This resembles the branch and cut method for integerization, but in NLIA we apply it to the objective rather than add it as a constraint. This approach usually found a nearby integer solution very quickly, but was challenged by the same types of problems that challenged BBIA, e.g. global structure such as the "radish" in figs. 3 and 9, and by many curves with equal sizes and goals. Such situations are called "degeneracies" in optimization, but are common in CAD models, e.g., many holes and bolts of the same diameter and plates of uniform thickness. The method was deployed in MeshKit [21] but has yet to be fully productionized and extended to all available meshing algorithms.

Frame Fields. Bommes et al. [11, 12, 13] partitions smooth graphics surfaces into structured quad patches and assign intervals. The algorithm uses a series of mixed-integer optimizations with linear constraints and quadratic objectives, Mixed-Integer Quadratic Programs (MIQPs). The first MIQP fixes the number and position of irregular vertices, the corners of the patches. The second MIQP sets the mesh structure of the patches, connecting the dual loops globally, and assigns intervals. The cross field defines a background that determines the objectives of the MIQP, by considering the orientation of the dual loops with respect to the surface curvature and any sharp features. The irregular vertices correspond to singularities in the cross field. Connections are made and variables are integerized by successive rounding. A key efficiency is using the solver as a white-box instead of a black-box after each rounding: the prior solution and internal state is updated and the solver can continue from it, rather than restarting from scratch. The observation that cross field design is related to the Ginzburg–Landau problem provides additional tools for boundary alignment [22].

A variation is to select irregular vertices and patches without assigning intervals, leaving that to a later step. But, in both variations, extending frame fields from 2D to 3D is challenging because 3D solutions do not always correspond to hex meshes the same way that 2D solutions correspond to quad meshes [23, 24].

1.2.3. Greedy Approaches

Guru–Protégé. Beatty and Mukherjee [25] present an IA "guru–protégé" method, which identifies the next curve whose interval assignment is most important, the guru, and fixes its intervals next. Protégé edges follow those assigned intervals. Each fixed edge reduces the remaining degrees of freedom. (From the IIA viewpoint, each fixed edge reduces the dimension of the remaining nullspace). When a remaining subsystem of equations has only one solution then it is applied. This can be viewed as a greedy approach with similar goals to lex min-max. The overall method first assigns corners, which partitions the model into mapped regions with T-junctions, and determines the IA constraints.

The BBIA framework was not used in Beatty and Mukherjee's context, automotive body panels, because the runtime of LP and BB was prohibitive. Another issue is that when an LP or BB solver reports that the problem is infeasible, not enough feedback is available for the user to know how to change the model to make it feasible.

1.2.4. IA and Mesh Refinement

The problem of locally refining an existing quad mesh is related to interval assignment: select the mesh edges to split (increase intervals) subject to the constraints of the available refinement templates (meshing

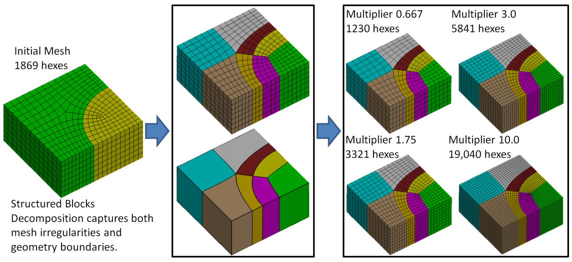


Figure 4: Mesh Scaling, courtesy Staten et al. [27].

schemes), with the goal of refining the mesh where the user wants, and leaving the mesh unchanged elsewhere (interval goals).

Binary Optimization. "Cost Minimizing Local Anisotropic Quad Mesh Refinement" [26] considers the refinement problem, and adds the goal of introducing few irregular vertices. They pose and solve this as a binary optimization problem. They state that an outstanding problem is to develop a specialized solver that would solve the problem more quickly. They would be satisfied with suboptimal solutions if it helps runtime. The runtime is often several seconds for a few thousand elements, and sometimes minutes, and is unpredictable.

IIA for Mesh Scaling. A simple form of Incremental Interval Assignment (IIA) was previously developed for the restricted context of "Mesh Scaling" [28, 27]. The problem is to refine an existing mesh for verification studies, but without simply splitting every hex, e.g., into 8, as that would produce too many elements. Instead, the irregular vertices and block structure of the mesh are identified, then we may remesh these blocks with slightly increased intervals on their sides; see fig. 4.

IA for mesh-scaling is simpler than the general IA problem for two reasons. First, the input mesh already provides a feasible interval assignment, $Ax = b$, so we only need to maintain feasibility as we adjust the solution closer to the goals. Second, we only have structured blocks meeting face to face. It is unambiguous how a change of intervals propagates throughout the mesh, so there are few degrees of freedom and the choices are simple. Unlike general IIA, there is no nullspace to compute, and we do not have to consider combinations of nullspace vectors to make progress. (A variant with more degrees of freedom and choices allows re-paving surfaces and re-sweeping volumes.)

The IIA Mesh Scaling (IIAMS) solution method follows. A priority queue selects the least-refined curve in the mesh, and that curve's intervals are incremented by one. The selection criteria "least-refined" considers how refined a curve is, how refined the neighboring area of the mesh is, and how much the element count would increase. A series of queues prioritizes these differently, with some passes increasing intervals and others decreasing them, to hone in on a good assignment.

IIAMS was a dramatic improvement in both speed and output quality compared to using BBIA for mesh scaling. BBIA was failing after running overnight on some problems with about a thousand curves. In contrast, IIAMS achieved success in less than a second on all test problems. IIAMS's element count is also closer to the user request. IIAMS's success was the inspiration for researching a general IIA method.

2. Formal Definitions

2.1. Interval Constraints

The constraints typically have three forms: equality, inequality, and sum-even.

Equality Constraints. For mapping surfaces we have constraints that curves on opposite sides contain exactly the same number of edges. Equality constraints also arise from sub-mapping and some other templates, and from ensuring that volume sweep path lengths are consistent; see fig. 1.

$$\sum_A x = \sum_B x$$

Inequality Constraints. For midpoint-subdivision and similar primitives, we have triangle-inequality type constraints. A three-sided triangle primitive with sides A, B, and C requires

$$\begin{aligned} \sum_A x + \sum_B x - \sum_C x &\geq 2, \\ \sum_A x - \sum_B x + \sum_C x &\geq 2, \\ -\sum_A x + \sum_B x + \sum_C x &\geq 2. \end{aligned}$$

For example, the constraints in fig. 2 are

$$\begin{aligned} x_1 + x_2 + x_3 + x_4 - x_5 &\geq 2, \\ x_1 + x_2 - x_3 - x_4 + x_5 &\geq 2, \\ -x_1 - x_2 + x_3 + x_4 + x_5 &\geq 2. \end{aligned}$$

Inequalities with $b \neq 0$ can also be used to ensure that a submapped mesh matches the geometry's topology; see fig. 5.

Sum-Even Constraints. For an unstructured scheme, such as paving, we have the constraint that the sum of intervals on a surface's boundaries must be an even number; see fig. 1. It takes some manipulation to express this as a linear constraint:

$$\sum_A x - 2y = 0, \tag{1}$$

where we introduce y as an integer slack variable. If the sum must be at least 4, we can bound y to the range $[2, \infty)$. For certain small shapes, a larger lower bound can improve mesh quality; see fig. 8. For circular holes in plates, often the user desires $y \in [3, \infty)$.

We also use slack variables to convert all inequalities to equalities. These constraints are distinguished by the slack variables having coefficients of 1, whereas in sum-even constraints their coefficients are 2.

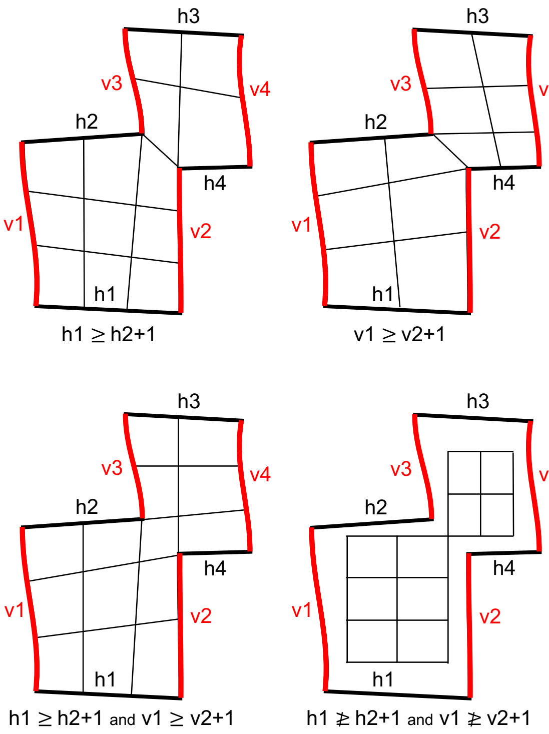


Figure 5: Submapping of a simple surface. Prior to interval assignment, corners are picked, assigning each curve to be either horizontal h or vertical v. The sum of the horizontal intervals must be zero; ditto vertical. Beyond that, for this shape we required that $h1 \geq h2 + 1$ **or** $v1 \geq v2 + 1$. If we require both to be satisfied, then the top row solutions are excluded. If we require neither, then the mesh might not match the topology of the geometry because it has a pinch point (lower right) or worse, an overlap. A single surface may require many non-overlap "or" constraints to exclude all bad-topology.

"Or" Constraints. These require only one of a set of linear constraints to be satisfied. See fig. 5 for an example of how these arise in submapping. "Or" constraints are not linear, not convex, and can not be represented by $Ax = b$ x directly. Most approaches avoid these by picking one of the linear constraints to enforce a priori. This reduces the degrees of freedom and may even cause the problem to be infeasible depending on the global structure. For submapping, CUBIT instead solves IA without enforcing either constraint, then checks the solution. If neither constraint is satisfied, we pick the one that is closer to being satisfied, with tie breakers, add it to A, and re-solve.

2.2. *Goals*

We have an idea of the number of intervals we would like for each curve, the *goals*. These may come from a sizing function. E.g. if the user wants edges about length 4, then a curve of length 10 has a goal of 2.5 intervals. Or the user may specify the goal directly. As long as slack variables are above their lower bounds, we are indifferent to their values; these have no goals.

There may be no feasible solution exactly matching all of the goals, so we measure the deviation of the achieved interval x_i from its goal g_i. In general, we have some *objective function* $f(x, g)$ of the deviations, where $f(g, g)$ is a minimum, preferably a unique minimum.

2.3. *Problem Definition*

IA in standard matrix-vector notation is

$$\begin{aligned} \min f(x, g) : & \\ & Ax = b \\ & x_i \in \mathbb{Z} \\ & l \leq x \leq u. \end{aligned} \tag{2}$$

This arises because an equality constraint has the following form, where we allow the possibility of arbitrary integer coefficients a, and some intervals prescribed to be specific numbers that add up to b,

$$a^T x = b.$$

For inequalities, we have

$$a^T x \geq b,$$

but these are converted to equalities by introducing a slack variable x' and constraining $x' \geq 0$:

$$\begin{bmatrix} a \\ 1 \end{bmatrix}^T \begin{bmatrix} x \\ x' \end{bmatrix} = b.$$

Here, most A coefficients are ±1, except the sum-even slack variables have coefficient 2. Thus, A is not totally unimodular, but it is close, and this special structure is helpful when performing Gaussian elimination and other steps.

For IIA, we choose the objective $f(x, g)$ to be the lexicographic max vector $R(x, g)$, where R is the ratio between the assigned intervals and the goals. Slack variables have no goal and do not contribute to the objective. Recall *lex min-max* means we minimize the maximum ratio amongst all edges. Then, we conceptually ignore this worst edge and minimize the maximum ratio amongst all remaining edges, etc. We define

$$R = \begin{cases} x/g \geq 1 & \text{if } x \geq g \\ g/x \geq 1 & \text{if } g \geq x > 0 \\ 1000(2 - x)R(1, g) & \text{if } x \leq 0 \end{cases} \tag{3}$$

Note $R(x, g) \geq 1$ with $R(g, g) = 1$ for integer g. If the bounds in eq. (2) are not satisfied, we may measure their signed distance away from being satisfied by

$$B = \begin{cases} l - x > 0 & \text{if } x < l \\ 0 & \text{if } l \leq x \leq u \\ u - x < 0 & \text{if } x > u \end{cases} \tag{4}$$

The sign of B indicates whether incrementing or decrementing improves the solution. We use $|B|$ as an objective when trying to improve variables to make them in bounds, and R afterwards.

Discussion. R in eq. (3) has a unique minimum at $\lfloor g \rfloor$ or $\lceil g \rceil$, except for $g = \sqrt{(k+1)k}$ for integer k, in which case $R(k) = R(k+1) = \sqrt{(k+1)/k}$. If a variable has a goal we require it to be positive, and suggest $10^6 > g > 10^{-6}$ for numerical stability. If the user requests a mesh size larger than the length of a curve, the floating point goal for that curve is less than 1. Our implementation represents "no goal" internally with $g = 0$. For variables without goals, we define $R = 0$, depending on context. The last case in eq. (3) ensures that $R(0, g) > R(1, g)$ and $R(x - 1, g) > R(x, g)$ for integer $x \leq 0$. Defining R for $x \leq 0$ is a numerical safety measure for unexpected problem instances, since in the context of CUBIT all x with a goal also have a lower bound of 1, and until bounds are satisfied R is only relevant as a priority tie-breaker. Intermediate solutions to $Ax = b$ may have $x \leq 0$, but the final solution does not.

3. Solution Method

Overall method. See also section 7.

- Define integer *ideal* values for variables.
  - Interval variables: close to goals.
  - Dummy variables: a value that satisfies (or nearly satisfies) their constraint given interval variables at ideal values.
- First-phase, mapping-phase. Ignore sum-even constraints.
  - Separate into independent subproblems.
  - Solve each subproblem using the *core* below.
- Update ideal values for variables.
  - The ideal value of a variable that was in a first-phase problem is its first-phase solution.
  - Sum-even and other new dummy variables are recomputed from these new ideal values.
- Second-phase, even-phase. Solve full problem.
  - Separate and solve as in the first-phase, using the *core* with slight variations.
- Re-solve. If new constraints arise (e.g. submap overlaps need to be eliminated):
  - Augment A with additional constraints, breaking $Ax = b$.
  - Add slack variables y to re-satisfy $Ax = b$.
  - Increment x to satisfy $y = 0$ and improve $f(x, g)$ using the *core*, skipping its first stage.

The Core.

- Solve integer $Ax = b$ for x.
  - Try Reduce Row Echelon Form (RREF) as a heuristic; see section 3.2.
  - Use Hermite Normal Form (HNF) if needed; see section 3.3.
  - Assign ideal values to independent variables, in both cases.
- Satisfy bounds $l \leq x \leq u$.
  - Find integer vectors N spanning the nullspace of A using RREF.
  - Iteratively increment x by adding integer combinations of nullspace vectors.
- Improve ratios $f(x, g)$.
  - Via the same methods as "satisfy bounds."

Solving $Ax = b$ is described in section 3.5. We try to solve $Ax = b$ using RREF first, because we tend to get a final solution that is closer to the goals.

When we increment x, we select the *worst-quality* x_i as the target for improvement. For satisfying bounds, quality is defined by how far out-of-bounds the variable is: $|B|$ from eq. (4). For improving $f(x, g)$, quality is defined by R, the ratio of x_i to g_i from eq. (3). We track the sorted quality of x using a form of priority queue that supports fast replacement.

We accept only strict improvements: post-increment the quality for all modified x_j must be strictly better than the quality of x_i pre-increment. If incrementing x by some nullspace row makes x_j worse than that, then we say x_j *blocks* improving x_i. The run-time efficiency of incrementing depends heavily on tracking the blocking variables, and identifying when no further improvement is likely; see section 5.1. We use Gaussian elimination to find nullspace vectors without those variables, or at least with variable coefficients with non-blocking signs. The success rate of incrementing depends strongly on the initial nullspace vectors, which are determined by the choice of pivots during the RREF process. Solving constraints $Ax = b$ uses one pivot criteria in the mapping-phase, and a second criteria in the even-phase. Satisfying bounds and improving ratios uses a third pivot criteria; see section 4.

3.1. Improvements Since [8]

To guide readers familiar with the prior version [8], here we summarize the algorithmic improvements since then; and section 8.5 summarizes their effect on solution quality. Other readers may skip to section 3.2.

- **Augmented nullspace.** For the second pass, the nullspace is augmented by short and local vectors. These are the first-pass equality-only nullspace vectors extended to include the sum-even variables. For huge coarse models, these extra search directions can enable finding a solution that is in-bounds, where otherwise the algorithm would fail; see section 8.4.

- **HNF goals.** The Hermite Normal Form (HNF) solution to $Ax = b$ sets the free-variable values to their compound ideal values; see section 3.5.3. This often make the initial solution closer to the optimal solution, which can improve the final solution found.

- **Best improvement.** The best vector is now used, instead of the first one, when satisfying bounds and improving the solution. Amongst all the vectors that improve the primary variable, the “best” is the one that improves a secondary variable that is farthest from its bounds or goals, or degrades a secondary variable closest to its bounds or goals.

- **Goal-based pivots.** We select a variable with the largest goal to pivot on in RREF in the even-phase; see section 4.1. Thus the independent variables tend to have small goals, which is desirable because the solution quality is more sensitive to them, and we can keep them at their mapping-phase solution values. The prior selection rule avoided fill-in.

- **Goal-based rounding.** When RREF has no integer solution, we round fractional values in hopes of finding integers close to a good solution; see section 3.5.2. We round in the direction that tends to be most helpful to the bounds and goals. Previously, such fractions were always rounded up.

- **Improve currently-worst.** The quality of the current solution value, $f(x, g)$, instead of what it would be after incrementing it, $f(x \pm 1, g)$, is used for the priority queue that selects the next variable x_i to improve towards its goal; see section 6.

- **Circumventing blockers in even-phase.** Section 8.4.1 outlines two strategies for combining nullspace rows in the even-phase to handle blocking variables. Instead of doing Gaussian elimination, **flipping signs** of blocking variable is achieved by adding a nullspace row with sum-even variables and a 2-coefficient for the blockers. For future work, we also describe how we might **select pairs** of such rows systematically.

3.2. Reduced Row Echelon Form (RREF)

RREF is a generalization of diagonalization of square matrices to matrices with extra columns and redundant rows [29]. It is what you get when you perform Gaussian elimination on a matrix with more columns than rows. If you are restricted to integer operations, then Gaussian elimination is the right approach to solve $Ax = 0$, because floating point alternatives can lead to errors [30]. For a full explanation of getting the nullspace from RREF, and some easy-to-follow examples, see Mitra [31]. We summarize the operations here, and provide a small concrete example at the beginning of section 3.4.

$$\text{RREF}(A) = M = \begin{bmatrix} D & F \\ 0 & 0 \end{bmatrix} \text{ and } Ax = 0 \Leftrightarrow Mx = 0$$

We allow swapping columns (and the corresponding edge variables) so the upper left of the RREF matrix, D, is diagonal. Note F is a matrix and the 0's are matrices. We discard the zero rows; they do not matter.

We can now "read off" the nullspace vectors from the columns of F, with minimal computation as follows. Let $L = \text{lcm}(D)$ be the least common multiple of the entries of D. Each column of F contributes a nullspace vector. For each column F_j of F, we multiply each entry F_{ji} by $-L/D_{ii}$. We append the elementary unit column vector e_j, where the length of e_j is chosen so that the resultant vector has same length as the number of columns of A. The transpose is a nullspace row vector.

If the matrix is totally unimodular, then we can always have D be the identity. This is not the case for us, e.g., the sum-even dummy variables have coefficient 2. However, *most* of our coefficients are ±1, so heuristics for selecting the pivot entry can often get us something close to the identity. This helps us find nullspace vectors with small coefficients, by keeping L small, and avoids some numerical issues with very large integers.

3.3. Hermite Normal Form (HNF)

We use Hermite Normal Form (HNF) to solve integer $Ax = b$; see Kopparty [32] for a more complete description. We use the column-form of HNF. Readers may be familiar with using Gaussian elimination and RREF to solve *floating point* $Ax = b$; see section 3.4 for an explanation of why this is insufficient when restricted to integer operations. (HNF also arises in integer programming: do floating point computations, then use HNF for integer cuts to attempt to find a nearby integer solution [33].)

An interpretation of solving $Ax = b$ is finding some integer linear combination of the columns of A that add up to b. HNF is basically Gaussian elimination on the *column space* of A, rather than its row space, so that the transformed system of equations is easy to solve. The operations preserve the column space of A, but transform the variables x. Finding HNF(A) means finding H and U such that

$$H = AU$$

where U is unimodular, square invertible with determinant ±1; and H is lower triangular with any zero columns on its right, and its diagonal entries are larger than other entries in the same row, and all entries non-negative.

If we solve $Hc = b$ for c, then $x = Uc$ is a solution to $Ax = b$, because

$$Ax = AUc = Hc = b.$$

Moreover, solving $Hc = b$ is easy by back-substitution because H is triangular, and the other properties ensure c will be integer.

We compute H and U as follows. Since RREF(A)$x = Ax$, we start with RREF(A) instead of A, and discard any zero rows so we are dealing with matrices of full rank. We initialize H = RREF(A) and $U = I$ and then perform matrix operations to achieve the necessary H properties. All the while we preserve $H = AU$: whenever we perform a column operation on H, we perform the corresponding operation on U.

We iterate over the columns j of H and perform the following three steps.

1. For each remaining column $k \geq j$ of H, we ensure the uppermost non-zero coefficient is positive; if not already positive, we multiply the column of H (and U) by -1.
2. For row j, we get one non-zero in columns $k \geq j$ by adding integer multiples of columns together. We find a non-zero as small as possible in a brute force way: find the column with the smallest-magnitude non-zero, then subtract it from (or add it to) all other columns to make them have smaller magnitude. We stop when no more reduction is possible (because they are the same or zero). We then swap the smallest column into j. (Again, all operations are also done on U.)
3. We ensure off-diagonal entries are non-negative and smaller than the diagonal. If any off-diagonal is too big, we add a multiple of the diagonal entry's column to the off-diagonal column to reduce it; see "Algorithm: ReduceOffDiagonal" [32] for details.

At the end of the iterations, H has the necessary properties. The steps that reduce coefficients are done both to ensure c is integer, and for numerical reasons along the way. They keep integer values from blowing up and overflowing the size of the integer representation on the computer. Since our sparse-matrix data-structures are designed to be efficient for row operations, we implement all of the above using row operations on the transpose, instead of column operations.

3.4. Discussion: Why RREF and *HNF?*

RREF Example. We begin with a simple concrete example of RREF. Consider four faces of a cube meeting curve to curve, and we wish to map each face. Intervals on the two curves on opposing sides of a face must be equal. Each face has one constraint that is an independent subproblem, and there is one subproblem that couples the other constraint for each face. In it there are four curve variables, and four equality constraints. Thus

$$A = \begin{bmatrix} 1 & -1 & & \\ 1 & & -1 & \\ & 1 & & -1 \\ & & 1 & -1 \end{bmatrix} \tag{5}$$

which RREF reduces to

$$M = \begin{bmatrix} 1 & & & -1 \\ & 1 & & -1 \\ & & 1 & -1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

RREF reduced the constraints to a minimal independent set. Creating (discovering) rows of zeros at the bottom of the matrix identifies the presence of redundant constraints. There is one, because the problem is the same whether the four cube faces form a cycle, or are cut into a chain.

From M, we have $D = I_{3,3}$ and $F = [-1 - 1 - 1]^T$. Since D has one fewer column than M, there is one independent variable, in the fourth column of M. Let us assume we did not do any column swaps so it is still x_3. Assigning x_3 to its ideal value, back substitution using the identity assigns $x_{0,1,2}$ the same value.

RREF allows the generation of vectors spanning the nullspace of A as follows. From M, we have $D = I_{3,3}$ so $L = \mathrm{lcm}(D) = 1$. Observe $F = [-1 - 1 - 1]^T$ has only one column, so there is only one nullspace vector. The number of rows of F is one less than the number of columns of A, so the elementary column vector we need to append to the column of F has length one: $e_0 = [1]$. Thus the one nullspace vector is $[-F_{0,0}L/D_{0,0}, -F_{0,1}L/D_{1,1}, -F_{0,2}L/D_{2,2}, e_0] = [1111]$. This makes sense, because whatever the initial solution, we can always add or subtract one interval from all of the curves and still map the sides of the cube.

HNF Example. Sometimes RREF is insufficient to solve $Ax = b$. As an example, consider the fictional RREF system

$$\begin{bmatrix} 2 & & & & -1 & 2 & 2 & 1 & 2 \\ & 4 & & & 2 & & & 2 & 3 \\ & & 1 & & & -2 & 1 & 2 & \\ & & & 3 & & 2 & -4 & 2 & \end{bmatrix} x = \begin{bmatrix} 7 \\ 1 \\ 2 \\ 5 \end{bmatrix} \tag{6}$$

There is no integer solution using just the first 4 column variables, because 2 does not divide 7, and 4 is greater than 1, etc. But, there are many degrees of freedom provided by the five variables whose coefficients are not diagonalized. They enrich the integer column space of A, filling in the lattice of possible solutions. Is it possible to chose some combination of values for them so that the system is solvable? This is the question that HNF answers for us. HNF makes the diagonal terms as small as possible, which allows us to visit the necessary integer lattice points to solve $Ax = b$. Our RREF implementation has heuristics which choose small coefficients for the diagonal entries, but they are not guaranteed to be 1. In our fictitious example, for the first row, simply swapping the 1st and 5th columns and negating provides a 1 on the diagonal. For the second row, subtracting the last column from the second provides a 1 on the diagonal. In general, we get the gcd (greatest common divisor) of the coefficients of each row on the diagonal; in this example they are all one.

HNF for eq. (6) is

$$H = \begin{bmatrix} 1 & & & & 0 & \cdots & 0 \\ & 1 & & & 0 & \cdots & 0 \\ & & 1 & & 0 & \cdots & 0 \\ & & & 1 & 0 & \cdots & 0 \end{bmatrix}$$

Now, the first four variables (columns) are in a transformed space. Each is a complicated linear combination of the original variables (columns), as captured by U. When we solve $Hc = b$, then transform back to the original space with $x = Uc$, the original variables x may not be so sparse. What about the extra c variables, columns 5–9? We can set them to any values we like. They represent the degrees of freedom inherent in A being short-and-long rectangular. Going back to our 4-sides-of-a-cube example, eq. (5), there is one extra variable, c_4, which represents the constant we can add to each of the four curves and still be feasible. That is, U contains nullspace vectors which are activated by non-zero values of the extra c variables.

3.5. Solve Integer $Ax = b$

We attempt to find a solution to $Ax = b$ using back-substitution in the RREF we found. If this succeeds, then we have saved the expense of computing HNF, and usually find a better solution than HNF yields. If we are stuck with something resembling eq. (6), then RREF will not succeed and we must do back-substitution on HNF. To compute HNF, we want to first compute RREF anyway, because RREF identifies the rank of A, and allows us to use methods for generating HNF that depend on the matrix being full rank. That is, there is little wasted computation if solving $Ax = b$ using RREF fails.

3.5.1. Ideal Values

To take advantage of the extra degrees of freedom in RREF and HNF, we set the independent variables x

equal to their *ideal* values g'. In the first phase, an ideal value follows from the goal: the integer value that minimizes $R(x, g)$ from eq. (3) subject to satisfying bounds. In the second phase, an ideal value is the first-phase solution value: these satisfy constraints and were chosen to provided low values of R globally.

When doing back substitution with RREF, the dependent variables x are derived from the ideal values of the independent variables, and may be far from their own ideal values or bounds. For their use in HNF, see section 3.5.3.

In detail, the following procedure calculates g'. An edge variable starts with its goal. If a goal is fractional, we round it to the integer that minimizes R. Specifically, let $h = \lceil g \rceil$ and $\ell = \lfloor g \rfloor$. If $g \geq \sqrt{h\ell}$ then $g' = h$, else $g' = \ell$.

For tied variables, section 7.2, an x is associated with multiple goals $\{g_i\}$, so finding the value of x that minimizes the maximum R is slightly more involved. We let $g^h = \max\{g_i\}$ and $g_\ell = \min\{g_i\}$. Then $m = \sqrt{g^h g_\ell}$ and $h = \lceil m \rceil$ and $\ell = \lfloor m \rfloor$. If $m \geq \sqrt{h\ell}$ then $g' = h$, else $g' = \ell$.

If g' is below x's lower bound, we adjust g' to its lower bound. If it is above its upper bound, we set it to its upper bound. For tied variables, the limits are the maximum lower bound and minimum upper bound.

For a dummy variable, its *ideal value* g' is the one that satisfies its row constraint when the edge variables are at their ideal values. If the dummy variable coefficient is not 1 then sometimes there is no integer value that satisfies the row constraint: e.g., if the coefficient is 2 and the row sum is odd. In this case its ideal value is rounded to an integer.

3.5.2. Rounding Direction

In an earlier version of this work [8], these fractional values were always rounded up to the next larger integer. This was usually a good choice, but for coarse meshes (goals near 1) this sometimes led to worse solutions. Our new heuristic rule considers bounds and goals. We round towards getting the dummy variable in bounds. If it stays in bounds regardless, then we consider the bounds and goals of the other variables in its row.

The motivation is to consider how rounding the dummy variable requires the edge variables to change, whether it would move them in aggregate closer to their goals or farther away. For example, for a sum-even variable, if the edge interval sum is odd, then rounding the dummy variable up will cause one of the edge variables to increment. Rounding down causes a decrement.

This is the minimum change required. We do not know if other constraint rows will produce a more complicated change, such as two variables incrementing and one decrementing. Moreover, we know which variables are dependent and will change immediately, but perhaps these changes can successfully be sent to more favorable variables during the step that adds nullspace vectors.

Hence we consider whether incrementing or decrementing is more favorable on average over all the row variables. For each variable we compute the signed deviation from its bounds, or goals if it is in bounds. Depending on the signs of the coefficients, the variable is correlated or anti-correlated with the sum-even one; for anti-correlated variables we flip the sign of the deviation. We sum these signed deviations. This is a measure of how far the edge variables are in aggregate from the *floating point* values that minimize R. If the sum is greater than 0, then the sum of the variables is below the sum of their goals, and a good guess is to round in the direction that forces an increment. Empirically, rounding up more often than this works best: specifically, rounding up when the sum is greater than -3.

3.5.3. Ideal Values Used in HNF

The ideal values were used for RREF but not HNF in an earlier version of this work [8]. Specifically, when solving $Ax = b$ via HNF, each extra c variable was set to "1" before back substitution. Thus the solution was not influenced by the goals or bounds on x, and could be very far from both. The fix for this follows.

For HNF, some values of c are constrained by $Hc = b$, but others may be independent. We would like to set the independent variables so that x are equal to g'. Since $x = Uc$, we assign independent c variables to $c = U^{-1}g'$. The additional cost is that we must compute U^{-1}, since otherwise HNF has no need for an explicit U^{-1}. We accumulate U^{-1} by applying the corresponding inverse operation when accumulating U. (Recall in the implementation we actually accumulate U^T since row operations are more efficient than column operations.) We initialize $U = U^{-1} = I$. The inverse of negating a column of U is negating a row of U^{-1}. The inverse of swapping columns of U is swapping the corresponding rows of U^{-1}. The inverse of subtracting a multiple of a column, is swapping indices and adding that multiple of a row. I.e., using Matlab notation $U[:, c]$ to denote column c of U, the inverse of $U[:, c] = U[:, c] - mU[:, d]$ is $U^{-1}[d, :] = U^{-1}[d, :] + mU^{-1}[c, :]$.

4. Pivot Selection Heuristics

When forming the RREF we have the freedom to choose which variable to *pivot on* (a.k.a. *reduce*, eliminate from all other rows) at each Gaussian elimination step. We have two different selection criteria, one for the constraint phase, and one for the bounds and goals phases of the *core*. For satisfying constraints, we want an RREF that allows back-substitution. For satisfying bounds and goals, the choice of pivot affects the nullspace matrix and the success rate of the subsequent increment attempts. Ideally, we would like nullspace vectors that allow us to increment the out-of-bound variables while keeping other variables in bounds, and generally prefer sparse vectors.

For each, we select RREF pivots using a hierarchy of criteria. By "hierarchy," we mean we pick the variable with the best primary criteria value. We use the secondary criteria to break ties. We iteratively pick the "best" remaining variable to pivot on at each step. Note that pivoting on a variable changes the matrix and may change the desirability of other variables in the pivot's rows, so their priorities must be updated. If no desirable variables are left, we pivot on an arbitrary variable, using the row in which its coefficient is smallest.

4.1. Pivots For Satisfying Constraints

We desire a RREF system that can be solved by back-substitution, where HNF is not needed.

- We select variables with a coefficient of 1 in some row.

If any such variable is in only one row, it is already in RREF form and pivoting on it requires no work. For variables in more than one row, we choose amongst them in two different ways.

In the mapping phase, we use the following criteria:

1. The primary criteria is the number of rows it appears it; fewer is better.
2. The secondary criteria is we find the *set* of variables in all of its rows and prefer smaller sets. (A variable in more than one row just counts once.)
3. The tertiary criteria is we prefer variables with no goal, followed by variables with a larger goal. Note sum-even variables have an initial coefficient of 2 and are thus rarely selected even though they have no goal. But slack variables can be selected here.

In the sum-even phase, we use the following criteria:

1. The primary criteria is we prefer variables with no goal followed by variables with a larger goal.
2. The secondary criteria is we prefer variables with larger $f(x,g)$, where x is the mapping-phase solution, its ideal value g'.

- If the variables with a 1 coefficient are exhausted, we select variables based on the gcd of its coefficients. The motivation is that if the gcd is 1, then it is possible to combine rows to get a leading coefficient of 1.

Selecting 1-coefficient variables helps us find a RREF system that yields an integer solution more often. Recall that if we do not find an integer solution, then we resort to HNF, with the RREF as the starting matrix.

In the mapping phase, choosing a variable in the fewest rows is desirable because it reduces fill-in, and leads to sparser nullspaces. In the sum-even phase, choosing a variable with a large goal is desirable because chosen variables are dependent, and when the goal is larger the solution quality is less sensitive to the number of intervals. This leaves the small-goal variables as the independent ones, who can retain their ideal mapping-phase solution values. Both tend to lead to a better-quality final solution.

4.2. Pivots For Bounds and Goals

We seek to deter fill-in and keep rows short, i.e. few non-zeros. We dislike rows with many sum-even dummy variables, since these couple multiple paving surfaces. We use the following pivot selection criteria.

1. The primary criteria is the number of rows a variable appears in; fewer is better. We pick all variables that are only in 1 row, including inequality slack variables. When all remaining variables are in multiple rows, then if a row has a sum-even dummy variable we penalize it as if it were three rows.
2. As a secondary criteria, we prefer variables with a small coefficient magnitude, ideally 1.
3. As a tertiary criteria, we prefer variables with no goals, followed by variables with larger goals.

4.2.1. Small Subspaces

If we are unable to get a variable in bounds, then we have no useful solution to give the user. So, if the optimization gets stuck in this situation, it is worth the computational effort to try harder. We attempt to find a small submatrix containing that variable that gives us sparse and local nullspace vectors. We augment the nullspace with these vectors to increase the chances of being able to improve the solution.

To ensure the submatrix's nullspace is contained in the matrix's nullspace, we must select complete columns from the matrix, but have the freedom to not select full rows. We initialize with the column of the out-of-bounds variable. All other variables in the rows of the selected columns are candidates. If some such row has only one selected variable, then that variable is not in a nullspace vector. So we first select columns to ensure every such row has at least two variables. The primary criteria is to prefer columns that add fewer new rows. The secondary criteria is just the total number of rows. Once every row has two variables, we continue adding more columns according to the same primary and secondary criteria. We stop when we have more columns than rows and are able to find a non-trivial nullspace containing the stuck variable.

5. Solution in Bounds

Once we have solved $Ax = b$, and have found vectors spanning the nullspace N using the criteria for the RREF pivots in section 4, we are ready to increment x with (combinations of) nullspace vectors in an attempt to obtain $l \leq x \leq u$. We have a priority queue with replacement for selecting which variable x_i to improve. The primary criteria is how far a variable is below its lower bound (or above its upper bound): eq. (4). The secondary criteria is its goal; we improve variables with larger goals first.

Once we have selected x_i for improvement, we check whether any existing nullspace row gives strict improvement with respect to its variables bounds. I.e., the new values of all changed x_j must be closer to in-bounds than how far the old value of x_i was out-of-bounds. Requiring strict improvement prevents infinite loops, and often prevents stuck cases that arise from shifting the limiting variable from x_i to some other variable that is difficult to improve.

If possible, we pick a row which changes variables in unbounded directions. E.g., all non-zeros row coefficients are positive and correspond to x_j with no upper bound. Otherwise, we accept a row that gives strict improvement with respect to how far variables are out of bounds. In the current version we pick the best row in terms of bounds, and if several rows tie, we pick the best one in terms of their goals. In a prior version we picked the first acceptable row we encountered.

Once we have a nullspace vector n that provides strict improvement, we continue to increment x by n as long as it provides a strict improvement over the prior increment's values. This is stronger than requiring it to be a strict improvement over the original x. Without this stronger requirement we tend to go from one x_i far from its bounds, to many x_j just outside their bounds, which require many subsequent iterations to fix.

If no nullspace vector provides strict improvement, it is because they are *blocked* by variables that would get worse. We search for some combination of nullspace vectors that provide improvement and are not blocked; see section 5.1. If we find a new vector that provides strict improvement, we save it by appending it to the nullspace, so we can check it in future iterations for other x_i. If no vector provides improvement, the variable is stuck, and we attempt to improve the remaining variables.

At the end this process, if some variable is stuck out of bounds, we attempt to find some small nullspace vectors containing it and try to get it in bounds as before; see section 4.2.1.

5.1. Blocking Variables

When checking incrementing by existing nullspace vectors, whenever a changed variable x_k would be as bad or worse than x_i, we save it in the set of *blocking variables* K.

We copy the original nullspace once, at the very beginning of the satisfy-bounds stage. We perform Gaussian elimination (partial RREF) on this copy and pivot on the blocking variables, so they each appear in only one row and are removed from all other rows. Among these other rows, the ones containing x_i are our candidates for improving x_i. We continue by checking these candidates to see if they provide strict improvement. If not, then we accumulate more blocking variables and eliminate them, until we either find an improvement vector or no further elimination is possible.

In future iterations, when attempting to improve x_i farther, or improving some other $x_{j \neq i}$, we continue to work with the same copy of the nullspace that we have already eliminated some blocking variables from; this is essential for efficiency. When we successfully increment x we mark any improved variable as no longer being a blocker, and unmark its row and column as a pivot so future Gaussian elimination steps may undiagonalize it; this is essential for robustness.

Further, we save the sign of the blocking variable increment. E.g., if a blocking variable is below its lower bound, then we cannot decrease it, but increasing it is acceptable, indeed desirable. Thus when performing Gaussian elimination we do not need to eliminate a blocking variable k with coefficient n_k if it has a favorable sign relative to the sign of n_j, and again may undiagonalize x_k. This also improves robustness.

6. Optimization Towards Goals

The procedure to improve variables towards their goals is essentially the same as the procedure to improve them so they lie within bounds. The differences are the following.

We define strict improvement in terms of the value of $f(x, g) = R(x, g)$ from eq. (3), and all variables must continue to stay in bounds, i.e. $B = 0$ in eq. (4). We do not check for unbounded improvement directions, since goals are finite.

We use a priority queue for selecting the next variable x_i to improve, choosing the one with the largest $f(x, g)$ value, except if it is already optimal. It is optimal if it is equal to its goal, or nearly equal to its goal and incrementing or decrementing it makes f worse. We say that such variables "self block." Sometimes, we are unable to increment a variable that is farther from its goal because all of its non-zero coefficients are large (not 1) in the nullspace vectors. Reducing the coefficient may be sufficient to enable improvement. We do not search for small subspaces; because it is expected that the global constraints may prevent some variables from achieving their optimal values and searching small subspaces for these would waste time.

"Incremental interval assignment for mesh scaling" [28] gets better quality solutions by selecting the next variable x_i to improve based on what the quality of the solution would be *after* it was incremented, $f(x \pm 1, g)$, rather than its current quality, $f(x, g)$. Because of that, initially IIA also used $f(x \pm 1, g)$, but we have since determined empirically that $f(x, g)$ gives higher quality solutions.

7. Other Efficiencies

The following efficiencies reduce the overall runtime because runtime is superlinear in problem size. We divide the problem into smaller ones, and remove redundant variables when possible. We solve the problem in two passes, first ignoring sum-even constraints, and second including all constraints. The first pass allows us to start the second pass closer to optimality. This improves runtime because the first pass contains more-but-smaller problems, and the second fewer-but-larger problems.

7.1. Independent Subproblems

It is straightforward to partition the matrix into independent rows and columns. We treat variables as graph nodes, and the non-zero entries in a row as edges between the nodes. Then a simple (depth first) search over the graph will identify connected components. In the first pass, it is key to ignore rows that contain a sum-even dummy variable and not include the corresponding graph edges. This approach is essentially the same as in BBIA [17].

7.2. Tied Variables

We search for rows of the form $x_i - x_j = 0$ and then mark x_i and x_j as *tied* because they must have the same value. Chains of the form $x_1 - x_2 = 0$, $x_2 - x_3 = 0$, $x_3 - x_4 = 0, \ldots$ form sets of tied variables. We replace each set with just one variable, x_t, in the matrix. The lower bound of x_t is the maximum lower bound of its constituents, and its upper bound the minimum upper bound. We save the maximum g_h and minimum g_l goal of the constituents, and use $f(x_t, g) = \max(f(x_t, g_h), f(x_t, g_l))$ when optimizing towards the goals.

It would be possible to reduce the number of variables farther by considering other types of constraints, such as when the b coefficient is not zero. However, in our context, searching for just this simple equality provided a large runtime benefit and it is unclear whether it would be worth the additional complexity to search for other types of constraints.

8. Applications and Experiments

IIA is in production use in CUBIT. We demonstrate that IIA succeeds on an academic challenge problem called a "radish." We study the runtime scaling of IIA, and highlight some runtime and robustness challenges with extreme-scale models.

8.1. CUBIT Production Use

BBIA was implemented in CUBIT in 1996–1997, and was run for every CUBIT quad and hex mesh, including autoscheme selection, from that time forward. IIA replaced BBIA as the default method in CUBIT at the beginning of year 2020. IIA is in production use by thousands of CUBIT users. IIA succeeds on every problem within CUBIT's extensive regression test suite, with hundreds of models and thousands of meshing problems. Users provided many of the models and meshing scripts when they encountered problems with earlier versions of CUBIT. On these realistic models, IIA performs well.

IIA often has slightly different solutions than BBIA, because of its slightly different and nonlinear objective, and the method often succeeds in coming closer to the optimal solution. In some cases in the test suite, intervals or sizes were manually adjusted to get good quality meshes. This arose from two reasons. First, the geometry of the meshing problem is not explicitly represented within the IA abstraction, so some geometric requirements are not captured by the IA constraints and goals. Second, certain research methods such as multi-sweep are fragile, and their success is unpredictable depending on the exact numbers of intervals in surface meshes and how the projection of one quad mesh overlays another.

For sweeping models, usually there are a handful of paving surfaces forming the source surfaces, bounded by submap surfaces forming the sides of the sweep. There are many such models in the CUBIT test suite, including assemblies of interlocking swept volumes; see fig. 7 for an example. The test suite also contains many surface-meshing problems; see fig. 6 for an example.

An open problem is modifying eq. (2) to capture general mesh quality criteria, such as element stretch or skew. Concurrent with IIA development, we developed some geometric reasoning algorithms in CUBIT to add interval lower bounds for small surfaces with curved curves or sharp angles. Without these, paving sometimes created poor meshes with flat or reflex angles. These new constraints were easy to pass to IIA, but updating the legacy BBIA solver to support them was prohibitive; see fig. 8.

8.2. Radishes

We demonstrate that IIA has superior robustness and solution quality for a family of challenge problems called "radishes." They are challenging for floating point methods because the space of integer solutions is sparse.

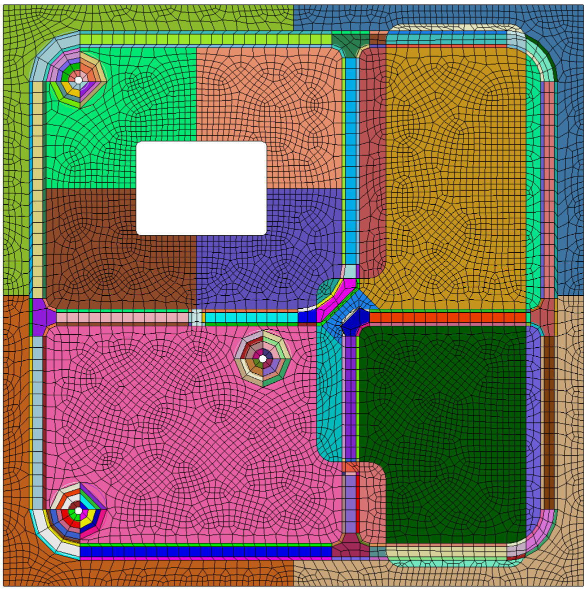


Figure 6: IIA with quad mesh paving 181 surfaces with the "skeleton" sizing from the open CUBIT regression tests.

By *radish* we mean a mapped surface where opposite sides have different numbers of curves, and all curves on a single side must have the same number of intervals. The term "radish" is a nickname for a particular assembly where the global geometry and meshing schemes give rise to this type of constraint; see fig. 3. While it is easy to see how the "2-1 radish" of fig. 3 occurs in real-world assemblies, we can extend this concept to create a series of academic problems that are increasingly challenging, albeit increasingly unlikely to be encountered in the real world. The "3-2 radish" in fig. 9 has one side with three curves, and its opposite side has two, and again all the curves on a given side have to have the same number of intervals. The only solutions are when the number of intervals for each side is an integer multiple of 6. This is because the first side's intervals must be divisible by 2, and the opposite side's intervals must be divisible by 3, and the least common multiple (lcm) of 3 and 2 is 6. Feasible solutions are pairs $\{3, 2\}k$ for integer k, i.e. $\{3, 2\}, \{6, 4\}, \{9, 6\}, \{12, 8\}, \ldots$

It is easy for IIA to find a feasible solution for any radish, because HNF finds an integer solution directly, and the nullspace contains vector $\{r, s\}$ for an r-s radish, which is an unbounded direction for making the solution positive. Radishes may be challenging for floating point methods, because the feasible integer solutions are a sparse subset of the integer lattice, and may be far from the relaxed solution.

3-2 Radish. See figs. 9 and 10. If the mesh size is selected so that the goal for each curve is $g = 8.5$, the ideal intervals for the side with three curves is 25.5, and the ideal intervals for the side with two curves is 17.0. So, for floating point methods, some compromise between 25.5 and 17 will be the relaxed solution for each side. Using $\min\max R$ as our objective, the optimal intervals for each side are $g\sqrt{6} \approx 20.82$. Hence two curves will have $x_i \approx 10.41$ and the opposite three will have $x_i \approx 6.94$. Good integer solutions are $\{9, 6\}$ and $\{12, 8\}$. Both are farther than distance 1 away from the relaxed solution $\{10.41, 6.94\}$, but still close enough for branch and bound methods to work well. For the 30-20 radish, pairs $\{3, 2\}k$ are also feasible solutions, so this is also easy for BBIA. From fig. 10, we see that both BBIA and IIA produce reasonable solutions, with BBIA being sub-optimal and coarser for some borderline sizes.

7-5 Radish. For the 7-5 radish, the solutions are $\{7, 5\}k$; see fig. 9c. For very coarse sizes, when the initial float-

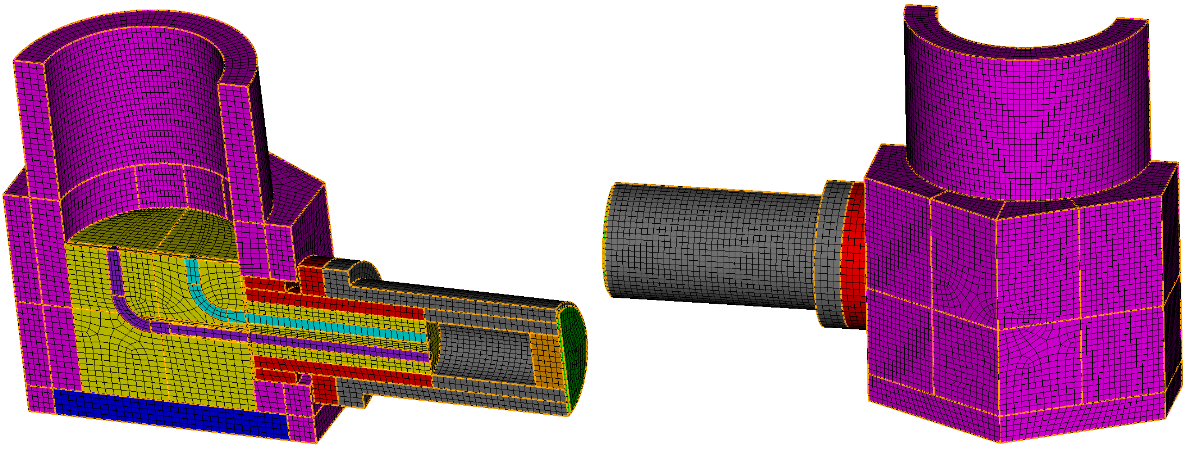


Figure 7: IIA with hex mesh sweeping 56 interlocking volumes, exercising auto scheme selection, sweeping constraints and verification. The problem is made more constrained for IA because some curves' intervals are user-prescribed and cannot change. Front and back view. From the open CUBIT regression tests.

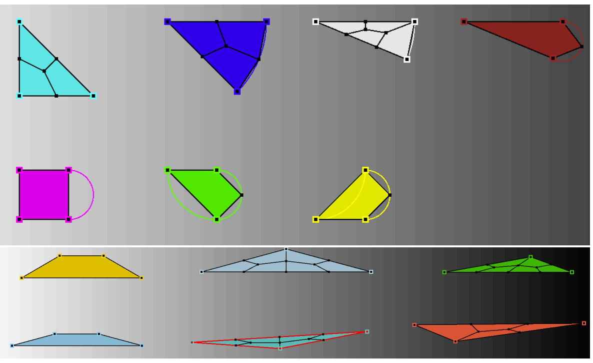


Figure 8: IIA supports interval lower bounds based on geometric reasoning. Mesh quads are linear and solid color; nonlinear CAD curves appear outside the elements. CAD vertices are surrounded by a small colored box. From the open CUBIT regression tests.

ing point solution is $< \{7, 5\}0.5$, BBIA fails because the relaxed solution is too far from the nearest integer solution; see fig. 11. BBIA has a search factor cutoff of 2 in one of its steps to avoid large runtimes for other problems, especially infeasible ones.

79-74 Radish. Here 79 and 74 are relatively prime, so the only solutions are $\{79, 74\}k$. This model has the same problems with coarse solutions as the 7-5 radish: BBIA fails when the relaxed solution is $< \{79, 74\}0.5$. Further, BBIA also fails for some intermediate sizes. We speculate that failure is due to the sparsity of the integer solutions and the heuristic bounds on BBIA's search distance, runtime, or both. The relaxed solution is $g\{\sqrt{79/74}, \sqrt{74/79}\}$. For example, for $g = 100.1$, we have $x_{\text{relaxed}} \approx \{103.4, 96.9\}$, and BBIA finds neither $\{79, 74\}$ nor $\{158, 148\}$, and returns "no solution" after three seconds of runtime. For other goals where BBIA does succeed, it takes at least 2 seconds. In contrast, IIA takes microseconds. See fig. 12.

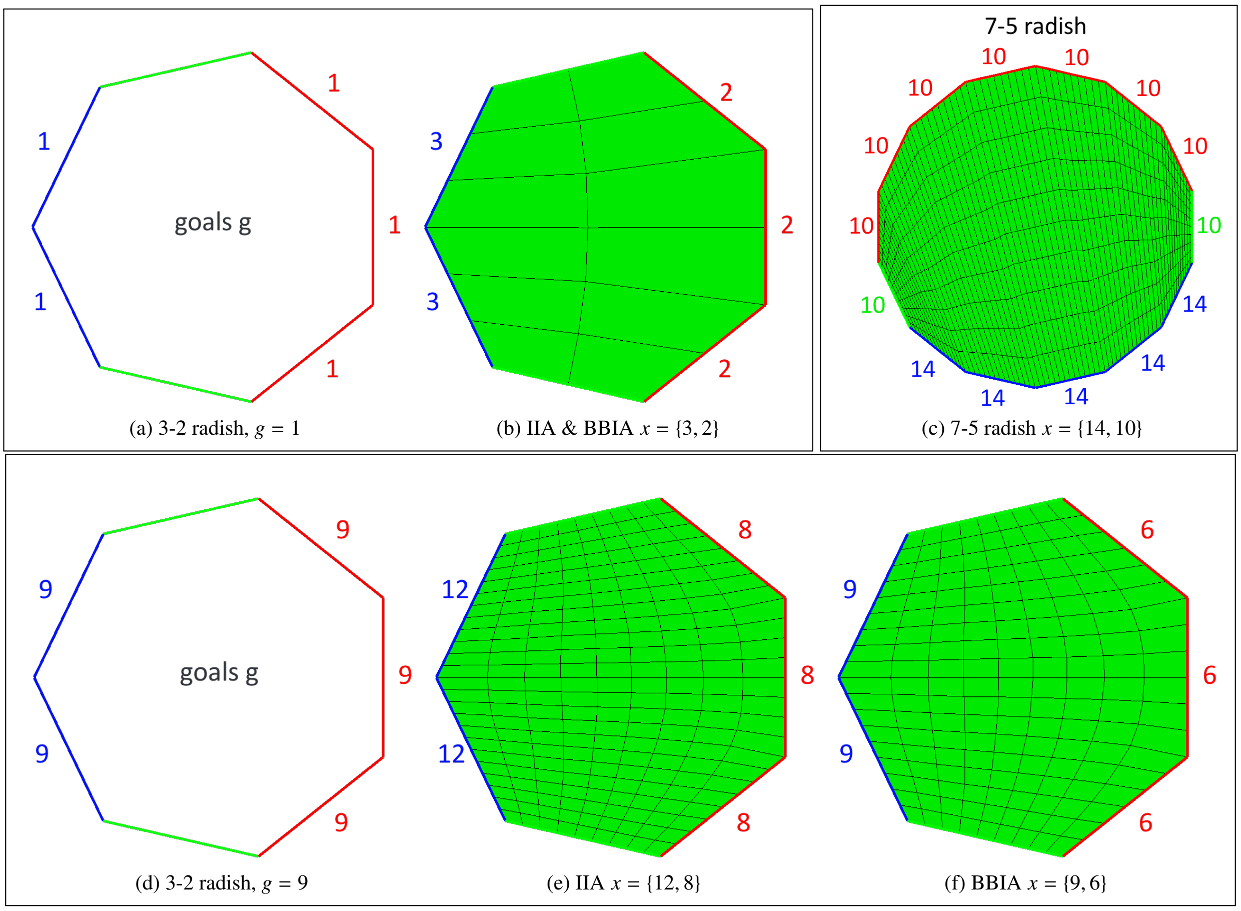
8.3. Runtime

We discuss runtime on "typical" problems, and include a scaling study to show the range of models for which the method is practical. All problems were run on a modest laptop, a MacBook Air, Early 2015, 2.2 GHz Intel Core i7, and 8 Gb memory.

IA runs at interactive speeds for today's models and runtime is insignificant. IIA runs in a fraction of a second for test-suite models. Serial runtime is fast enough that it is simply a non-issue. It takes CUBIT about 2–3× longer to decide how to define the IA problem than it takes to run the IIA solver. Other steps such as loading the CAD model, performing geometric Booleans, actually generating the mesh, or even just displaying the mesh graphics, take significantly longer.

IIA's runtime is often linear in the output mesh size, but unfortunately Gaussian elimination (for RREF and HNF) runtime is cubic, so IIA runtime can be cubic in the input assembly size. For typical model sizes, the linear factor dominates.

Before we judge IIA too harshly for cubic asymptotic complexity, let us recall BBIA's runtime is often observably cubic in the input assembly size, and sometimes exponential, e.g. when the BB step has many alternatives to consider. And, lest we shift our derision to BBIA, let us recall that we are performing integer optimization, and for many integer optimization problems



(a) 3-2 radish, $g = 1$

(b) IIA & BBIA $x = \{3, 2\}$

(c) 7-5 radish $x = \{14, 10\}$

(d) 3-2 radish, $g = 9$

(e) IIA $x = \{12, 8\}$

(f) BBIA $x = \{9, 6\}$

Figure 9: Radish meshes for different goals and algorithms. **Subfig. (a–b):** The 3-2 radish in (a) has $g = 1.0$. Both IIA and BBIA produce the solution $x = \{3, 2\}$ in (b). **Subfig. (d–f):** The 3-2 radish in (d) has $g = 9.0$. IIA produces $x = \{12, 8\}$ in (e) and BBIA produces $x = \{14, 10\}$ in (f). BBIA is worse. **Subfig. (c):** The 7-5 radish has one side with 7 curves and opposite side with 5, and feasible solutions $\{7, 5\}k$ for any natural number k.

sub-exponential complexity bounds are difficult to obtain.

Our scaling challenge is the heat-sink mock-up in fig. 13, where we purposely do not take shortcuts to exploit the obvious symmetry. Many surfaces are submapped, with many curves on each side. If two long curves are constrained to have a fixed number of intervals, say equal to the number of opposite curves times 1.5, this forces the solver to decide which half of the small opposite curves to give 2 intervals and which half to give 1. All such solutions are symmetric and equally desirable from an algebraic viewpoint. Hence, BBIA can take a long time, 20 minutes. CUBIT with IIA solves it 6000× faster, in a fraction of a second.

For the realistic models we considered, the IIA runtime was not the bottleneck. As we shall see in the next section, for the heat sink the majority of the interval assignment time was actually spent in the overhead of setting up and applying the solution, not in the IIA solver itself! It is possible to construct extreme models where the cubic runtime of Gaussian elimination dominates the IIA runtime. The next section explores these limits.

8.3.1. *Scaling to Extreme Models*

We scale the problem by doubling the heat sink model size. In the "Fin" scaling, we copy the model along one axis to create more fins, but each individual fin surface remains the same. That is, we increase the number of surfaces but not their complexity (except the one surface on the underside). In the "Cren" scaling, we copy the model on the other axis to create more crenellations per fin surface: we increase the complexity of the crenellated surfaces, but not how many there are. (The number of small trivial mapping surfaces on the top of the crenellations does increase, but these are all removed

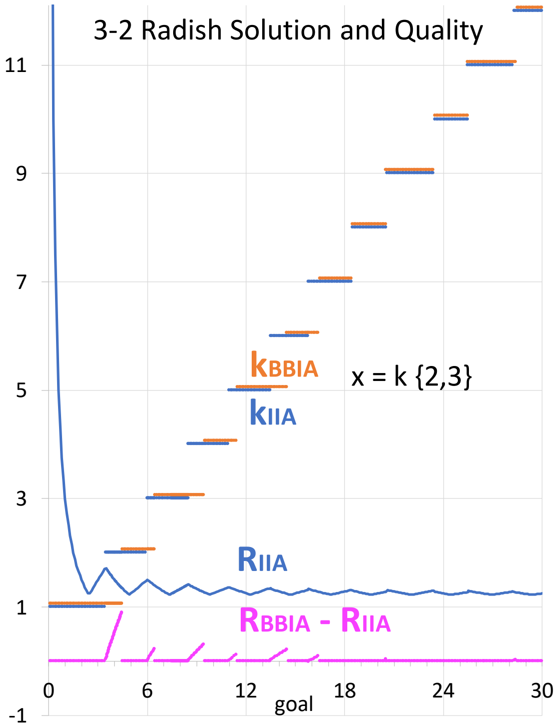


Figure 10: 3-2 radish solution and quality for different goals. The solution is $k\{2,3\}$, e.g., the solution for $g = 7.5$ is $k = 3$ and $x = \{6,9\}$ for both IIA and BBIA. For each constant-x IIA-solution interval, the quality R is best toward the middle of the interval, and worst at the ends, where the selected solution and the next k value are nearly equally desirable. For many ranges of goals, the BBIA solution is smaller than the IIA solution, and has worse quality. In exceptional ranges the BBIA solution is larger, e.g, near $g = 28.4$, and also has worse quality. The bottom "R_{BBIA}-R_{IIA}" curve shows how much worse the BBIA solution is than the IIA solution. The exception is a very small, 0.05-neighborhood around $g = 15.80$ where the IIA solution is slightly suboptimal and the BBIA solution is better.

by the "tied variables" step from section 7.2.) We also study scaling the model in both ways at the same time: "Both".

In the first study we mesh the surfaces and volume with submapping. The heat sink has 1266 curves and 424 surfaces. This leads to 2532 non-zeros in the constraint matrix A. Doubling the problem size in either direction about doubles the number of non-zeros, although in the Fin case we are adding equal rows and columns, and in the Cren case we are adding more columns than rows.

"CUBIT" is the time it takes CUBIT to set up the IA problem and pass it to IIA, and, after it is solved, check for submap parameter space overlaps and to apply the solution to the model. "RREF" is the time it takes to create all the RREFs during the course of the solve:

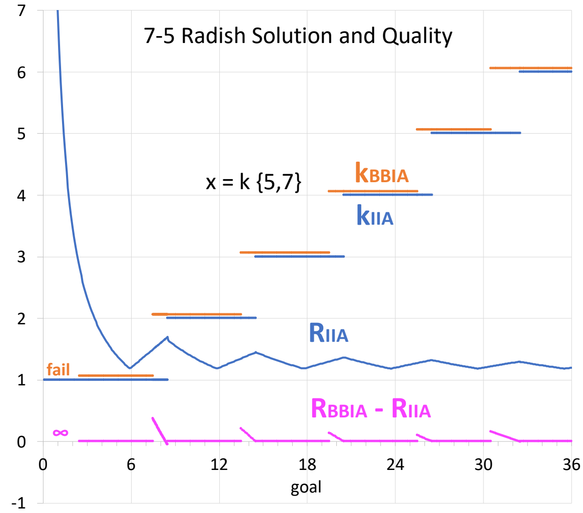


Figure 11: 7-5 radish solution and quality for different goals. BBIA fails for $g < 2.5$. When the BBIA and IIA solutions differ, the BBIA solution is almost always of poorer quality. Here the exception is a very small, 0.06-neighborhood around $g = 8.44$. But, in contrast to the 3-2 radish, the BBIA solution is often *larger* than the IIA solution.

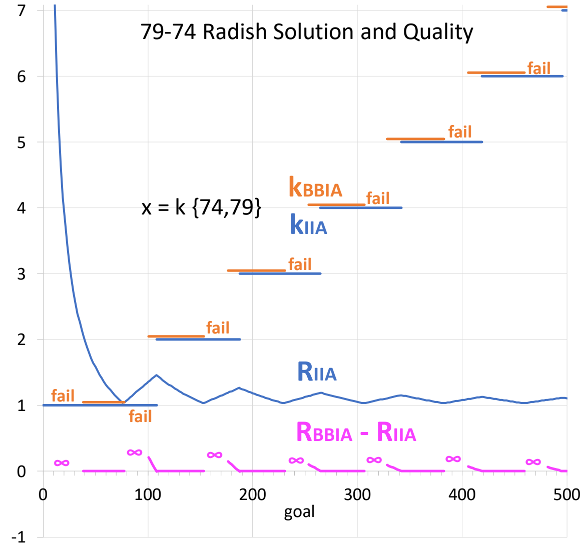


Figure 12: 79-74 radish solution and quality for different goals. BBIA fails for $g < 38.5$, and for some intervals around where the optimal solution transitions from k to $k + 1$. The BBIA solution is usually at least as large as the IIA solution. The IIA solution is better than the BBIA solution, except for a small neighborhood around $g = 187.49$.

for this problem, there are three, one for each of the submap axes. There are no sum-even variables so no RREFs are needed for that phase. In these examples HNF was not needed, but its runtime would scale the same as RREF's, just with the trending in the number of

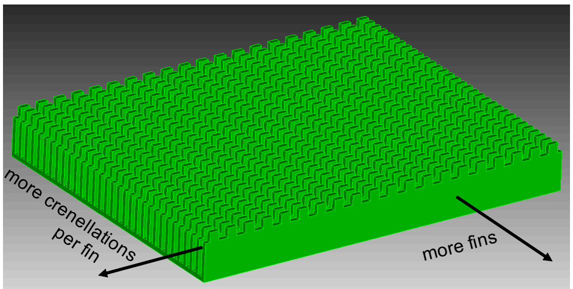


Figure 13: Heat sink mock-up. With the long curves constrained to fixed intervals, BBIA takes 20 minutes to solve this problem, but CUBIT with IIA can solve it in 0.2 seconds. This is a 6000× speedup.

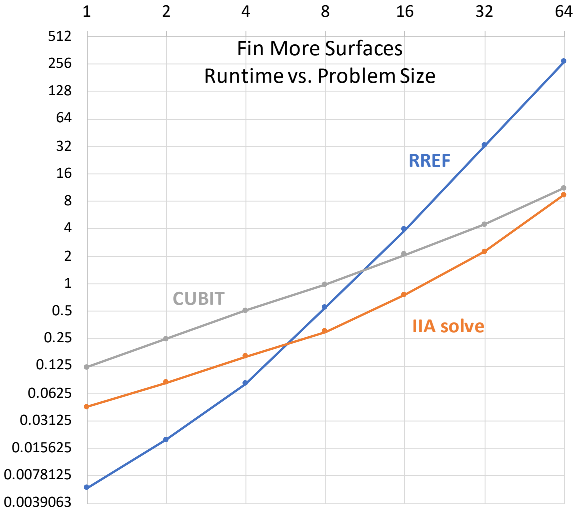


Figure 14: Runtime scaling (seconds) for multiplying the heat sink by creating more fins.

rows and columns swapped. "IIA solve" is the time that IIA takes excluding "RREF."

See Figures 14 to 16. The vertical axes are time in seconds. The horizontal axes are the problem size in multiples of the heat sink. Note the log-log scale. A straight line indicates a constant polynomial scaling, with the slope indicating the exponent of the polynomial complexity. For example, the "CUBIT" time in fig. 14 is roughly linear in problem size up until the largest models. "RREF" is cubic for fig. 14 and quadratic for fig. 15. For each of Fin and Cren "IIA solve" is close to linear up to about size 8 (20k non-zeros) and close to quadratic above it.

"Both" Scaling. "RREF" and "IA solve" performed better when scaling "Both" compared to scaling either one. It appears that for the same number of non-zeros, performance is better if the rows and columns are balanced and A is square, compared to tall and skinny or

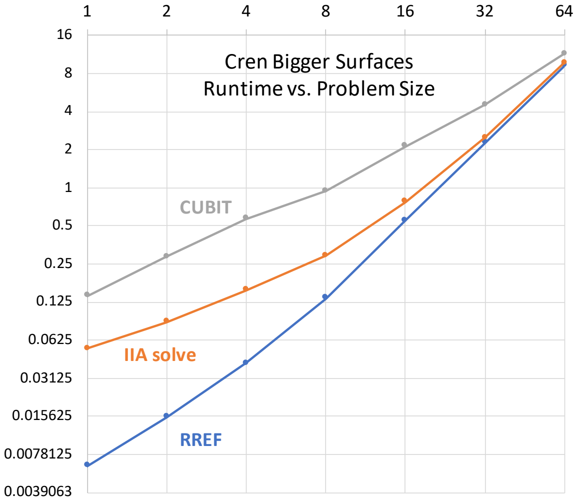


Figure 15: Runtime scaling (seconds) for multiplying the heat sink by making the fins longer, with more crenellations.

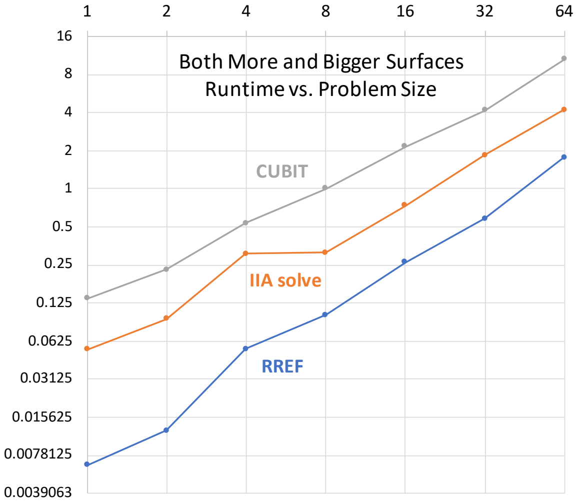


Figure 16: Runtime scaling (seconds) for multiplying the heat sink by both creating more fins and making them longer.

short and wide.

Fin Scaling. For reasonable-size problems the runtime of RREF is trivial. The crossover for Fin is about 30,000 non-zeros. Below this, performing the "optimization" steps take longer: selecting linear combinations of nullspace vectors for downhill improvement. Above this, generating the initial nullspace via RREF takes longer. While the runtime of RREF in the final Fin case is large compared to the rest of IIA, it is still dwarfed by the runtime of other meshing steps. For example, doubling the geometry from the prior size takes

1.3× as long, and actually creating the mesh takes 11.5× as long. Simply displaying the surface quads for the first time takes 2.5× as long. This problem has 80k curves, 27k surfaces, and A has 161k non-zeros. We generate a coarse mesh with 58k hexes.

8.3.2. Runtime Scaling Future Work

For IIA to scale well beyond 100,000 non-zeros, generating the RREF on the entire matrix must be avoided. Alternatively, generating the entire RREF using implicit numerical methods may also be possible, but comes with the challenge of obtaining integer nullspace vectors. Possible heuristics include manually dividing the model into independent pieces, or ordering pieces so that one piece of the model can be meshed before the next piece. Perhaps one could construct only some of the vectors in the nullspace, and these could be sufficient for the optimization step.

The typical approach to deal with polynomial scaling issues in linear algebra is to switch from an explicit discrete solver to an implicit floating point solver which scales much better. The challenge in our setting is that the whole approach is predicated on having only integers in the nullspace. It may be possible to use the floating point nullspace vectors to find nearby integer nullspace vectors. Recall that we do not need a nullspace basis; redundant vectors are useful and we just need sufficient vectors to make progress.

One typical approach to deal with scaling issues in optimization is to break the problem into subproblems and solve each one. The subproblems are designed so that their solutions are expected to be nearby to the solution of the global problem. These nearby solutions are used as a warm start to solving the original global problem, perhaps with heuristics to find a feasible solution and not an optimal one. This is exactly IIA's approach to solving the mapping constraints first before the sum-even variables are considered. It may be possible to use this idea in another way.

A second typical approach in optimization to keep problem size small is to only add some of the constraints and solve the problem. Then check all of the constraints, and if any are violated add them into the problem definition. Then re-solve the updated problem, using the prior solution as a warm start. This is what IIA already does for submap overlap constraints. We could consider extending this approach to other constraints, such as the sum-even ones. In our context we would have the added step of updating the nullspace based on the violated constraints.

8.4. Improvements for Sum-even Constraints

Performing the heat-sink scaling study with pave-and-sweep, and also with all-paving surfaces, uncovered a robustness issue. For large models or very coarse mesh sizes, sometimes IA failed to find a solution in bounds. All of the sum-even constraints are satisfied by the HNF step, but not all of the intervals are positive and not all of the sum-even variables are at least two. There were two root causes. First, using a poor initial value for the HNF c variables caused the initial solution to $Ax = b$ to be out of bounds: not all of the intervals were positive and not all of the sum-even variables were at least two. Second, the RREF nullspace computation creates vectors with multiple sum-even variables with coefficients of opposite sign. This causes an accumulation of blocking variables and the satisfy-bounds step gave up. We fixed the first root cause and mitigated the second; either was sufficient in our heat-sink tests.

These improve the satisfy-bounds step so that it no longer gets stuck for large heat-sinks. One can observe that for a curve shared by two paving surfaces, the vector $[y_1\ y_2\ 2x]$ is in the nullspace, where y_i is a sum-even variable and x is the shared-curve variable. If the two paving surfaces are connected by a chain of mapping or submapping surfaces, vectors like this still exist, with "$2x$" replaced by a nullspace vector we found during the first pass when sum-even constraints were ignored. RREF nullspace vectors such as these are now identified and gathered in a second matrix N'. (For efficiency, any vector that is in both N' and N is removed from N'.)

Searching N' and N. When checking existing nullspace vectors, we search through the concatenation of N' and N. This occurs when satisfying bounds and improving the solution, checking for existing improvement directions. If no existing direction works, then we do Gaussian elimination on a copy of N for the blocking variables. We do not do Gaussian elimination on N', as this would be largely redundant.

Searching through just N' is insufficient. In limited tests, we have observed that every vector of N' is spanned by integer combinations of vectors of N. However, the converse is not always true. Our pivot heuristics for RREF on N produces nullspace vectors with small coefficients, for example, some vectors contain a sum-even variable but have edge variables with a 1-coefficient, not a 2. In contrast, N' vectors for a sum-even variable always have a 2 coefficient for the edge variables. Thus these small-coefficient N vectors are not spanned by integer combinations of N'. Specifically, they are spanned by fractional $1/2$ multiples of N' vectors. Alternatively, 2 times such an N vector is spanned by an integer combination of N'.

We experimented on a coarsely paved heat sink with just 10 crenellations for each of its 32 fins. We paved all surfaces with a requested mesh size four times larger than the shortest edges. IIA now succeeds and takes less time. Previously IIA failed to find a solution in bounds after 81 seconds. Now, either the HNF or nullspace improvement enables IIA to find an in-bound solution successfully. The total solution time when enabling just the HNF improvement is 17 seconds. When augmenting the nullspace it drops to 12 seconds.

8.4.1. Alternatives to Eliminating Blocking Variables

The blocking variable strategy, totally eliminating variables whose increment makes the solution quality worse, is not as effective in the even-phase as the mapping-phase. Specifically, in the mapping-phase, variables are correlated (opposite sides of a mapping face) or anti-correlated (same sides of a mapping face). Hence if they block in one direction, the only recourse is to eliminate them. However, in the even-phase, if an edge in a sum-even surface is incremented, some other edge can be *either incremented or decremented* and the sum-even constraint will still be satisfied. Thus, Gaussian elimination of a variable is overkill, as it is possible to add other rows to change the sign of a blocking variables coefficient. Requiring elimination causes an excess accumulation of eliminated variables and the algorithm stops making improvements even when it could. See the following "Example" for a concrete illustration of this issue and alternative strategies.

Flipping Signs Using N' Vectors. Nullspace N' contains short vectors with coefficient 2 for many edge variables, and coefficient 1 for sum-even variables. We can use these to flip the sign of blocking variables. Specifically, when attempting to improve x_i by adding a vector from $n \in N$ to the solution, n may have a variable x_j that blocks with coefficient 1. If x_j is not blocked in the negative direction, then we may subtract some vector $n' \in N'$ from n to obtain a coefficient of -1 for x_j. A basic capability following this strategy has been implemented and improves the output quality of some tests in the CUBIT test suite. For future work, we would like a more robust strategy for selecting which nullspace vectors to add, and to handle coefficient values other than ± 1.

Pairs of N' Vectors. For future work, when attempting to improve an edge variable x_i, we would like to systematically consider pairs of nullspace vectors n from N'. The first vector $\pm n_1$ of the pair is any one that contains x_i, with the sign selected so that it improves x_i. For the second vector n_2, we may use any of them that contain the same sum-even variable. We may predetermine how adding or subtracting n_2 would affect its variables quality, e.g. assuming a positive coefficient, a variable might be above its goal so adding is not allowed, but subtracting n_2 would actually improve quality. We may then compute $n_1 \pm n_2$, divide by two, and thus have a vector that has a 1-coefficient for x_i and any blocking variables have the opposite sign in which they are blocked. This algorithm is expected to produce better solutions compared to eliminating blocking variables, and be faster.

Example. We illustrate the problem of excessive blocking and how the future work could overcome it. Consider paving a single surface bounded by three curves. Thus $Ax = b$ is $-x_0 - x_1 - x_2 + 2x_3 = 0$. Let goals $g_{0,1,2} = 3$, and $x_3 \in [2, \infty)$.

The RREF M of

$$A = \begin{bmatrix} -1 & -1 & -1 & 2 \end{bmatrix}$$

is A itself, with x_0 as the reduced (dependent) variable. This RREF can be solved via back-substitution, with no HNF needed. Since $x_{0,1,2}$ are free, they get assigned their ideal values: 3. The ideal value of x_3 is 5, since $x_1 + x_2 + x_3 = 9$ and the algorithm wisely chooses to round 9/2 up to 5. Thus, using the ideal values for $x_{1,2,3}$, we assign $x_0 = -x_1 - x_2 + 2x_3 = -3 - 3 + 10 = 4$. This solution $[4, 3, 3, 5]$ is already optimal, but our purpose is to illustrate how the algorithm could get stuck, and how the nullspace could be explored more thoroughly, in a more complex problem. So let us suppose that we somehow started the even-phase with the suboptimal solution $[4, 4, 4, 6]$.

For N, we read off the dual vectors from the RREF. Each contains the dependent variable x_0.

$$N = \begin{bmatrix} 1 & -1 & & \\ 1 & & -1 & \\ 2 & & & 1 \end{bmatrix}$$

Note this representation implies that x_0 is anti-correlated with x_1 and x_2, and misses that they can also be correlated.

For N', there are no mapping constraints, hence no explicit mapping nullspaces. However, each free variables that was in no mapping constraint constitutes an elementary nullspace vector in the mapping subproblems.

$$N_{\mathrm{map}} = \begin{bmatrix} 1 & & \\ & 1 & \\ & & 1 \end{bmatrix} \tag{7}$$

Each of these contain a variable that is in a sum-even row of M, hence we get

$$N' = \left[\begin{array}{lcccc} \texttt{0 row:} & 2 & & & 1 \\ \texttt{1 row:} & & 2 & & 1 \\ \texttt{2 row:} & & & 2 & 1 \end{array}\right]$$

If this were a more complex problem with submapping surfaces, then in the first row instead of "$2x_0$" we would have "$2n$" where n is some more complex map-phase nullspace vector containing x_0. Indeed, we get one row for each mapping-phase nullspace vector n containing x_0. If this were a more complex problem with multiple paving surfaces, each nullspace vector would contain the sum-even variables of the paving surfaces containing each of the edge-variables in n. In this simple problem, N' illustrates that each of $x_{0,1,2}$ could be incremented (decremented) by 2 independently. It explicitly represents the symmetry of x_0 with x_1 that is not explicit in N.

Other useful information not explicitly represented in vectors of N and N' includes

- how to increment x_1 and x_2 by 1, if possible;
- freedom to choose any pair of $x_{0,1,2}$ and either increment them both or increment one and decrement the other.

These are all explicitly represented if we take any pair of rows from N', then add them and divide by 2, and also subtract them and divide by two. This expansion is

$$\left[\begin{array}{lcccc} \texttt{(0+1 row)/2:} & 1 & +1 & & 1 \\ \texttt{(0-1 row)/2:} & 1 & -1 & & \\ \texttt{(0+2 row)/2:} & 1 & & +1 & 1 \\ \texttt{(0-2 row)/2:} & 1 & & -1 & \\ \texttt{(1+2 row)/2:} & & 1 & +1 & 1 \\ \texttt{(1-2 row)/2:} & & 1 & -1 & \end{array}\right] \tag{8}$$

This expansion has quadratic size: $2\binom{r}{2} = \mathrm{O}(r^2)$, where r is the number of rows of N'. Hence explicitly expanding and searching all these vectors could be prohibitively expensive, especially if we consider that the current algorithm would do this for each of $x_{0,1,2}$, and have cubic complexity.

Our proposed "Pairs of N' Vectors" subroutine may be more efficient. We may test each vector of N_{map} in eq. (7) to determine whether incrementing or decrementing it improves quality. If we find two such vectors, we may apply them in tandem provided any lower bound on x_3 is not violated. In our example, the solution $[4, 4, 4, 6]$ can be improved by $-$`(0+1 row)/2:` $= -x_0 - x_1$, giving the optimal solution $[3, 3, 4, 5]$.

If this were a more complex problem with multiple paving surfaces, then the expansion in eq. (8) might contain other sum-even variables with coefficient $1/2$. Thus we could not apply an expansion vector immediately, but would also have to find a vector in the expansion for that other surface, and apply it as well, in order to maintain integer coefficients. Conceptually, this involves finding a chain of map-phase nullspace vectors that form a closed loop, or terminate at the domain boundary curves (if any). A form of breadth-first-search that limits branching by solution quality may be effective.

We now illustrate how the current algorithm of eliminating blocking variables regardless of sign can get stuck at a non-optimal solution. Given the solution $[4, 4, 4, 6]$, the priority queue selects x_2 with $R = 4/3$ to improve. Of the existing N' vectors, $-2x_2 - x_3$ would make $x_2 = 3$, with worse R: $3/2$ instead of $4/3$. Hence it self-blocks with coefficient 2; this is correct and causes no problems. Of the existing N vectors, $+1x_0 - 1x_2$ would make x_0 worse than x_2 already is, with $R = 5/3$. Hence "x_0 blocks x_2" with coefficient $+1$. We have exhausted all the existing rows of N' and N with x_2 in them, so we do Gaussian elimination on N to remove the blocking variable x_0 from x_2's rows. This yields a new nullspace vector, $x_1 - x_2$ which unfortunately has the same issues as $x_0 - x_2$. Hence x_1 also blocks with coefficient $+1$. Gaussian elimination only produces x_2 vectors without x_0 or x_1, and none of these can improve the solution. Gaussian elimination lacks the mechanisms for reintroducing x_0 or x_1 with opposite sign, and will not find the nullspace vectors $-x_0 - x_2$ or $-x_1 - x_2$ which would improve the solution. The next variable in the queue to improve is x_1, which fails in the same way, as does x_0.

"Flipping Signs Using N' Vectors" instead of elimination will get us out of this state. Again, the priority queue selects x_2 to improve (decrement). Adding N's middle row, $n = x_0 - x_2$, would improve x_2 but is blocked by x_0 with coefficient 1. Selecting $n' = 2x_0 + x_3$, then $n - n' = -x_0 - x_2 - x_3$ can be added to the current solution $[4, 4, 4, 6]$ to obtain an optimal solution $[3, 4, 3, 5]$.

8.5. Quantified Improvement Effects.

Figure 17 shows how the algorithm improvements affect the solution quality of the 2700 IA problems that arise during the course of the CUBIT regression test suite. Quality is measured by the lexicographic-max vector of $R(x, g)$ values. The baseline is the released version from 19 March 2020. “Incidental updates (**A**) are a few bug fixes and robustness improvements; the other improvements are defined in section 1.1 and described throughout this paper. The individual results all include (**A**), and (**F**) includes (**E**), by necessity.

A. Incidental updates.
B. Improve currently-worst.
C. Goal-based rounding.
D. Goal-based pivots.
E. Augmented nullspace.
F. Flipping signs.
G. Best improvement direction.
H. HNF goals, $c = U^{-1}g'$.

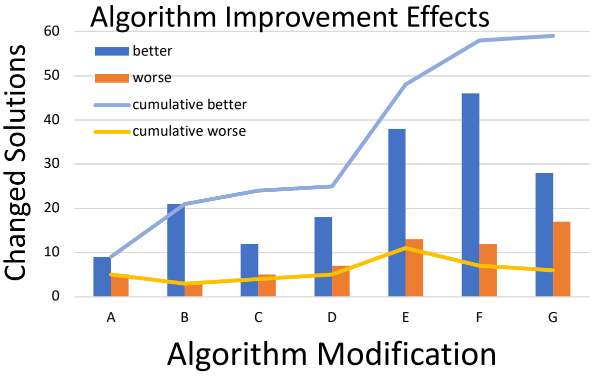


Figure 17: The effect of the algorithm improvements, both individually (bars) and cumulative (lines), in terms of the number of tests with changed outputs. For a single algorithm improvement, the blue bar is the number of tests whose solution quality is better than the baseline, and the orange bar the number with worse quality. The lines represent the net effect of applying all of the improvements up to that letter, e.g., applying A, B, C, and D improves 25 tests and degrades 5.

H, using the goals for the HNF solution to $Ax = b$ via $c = U^{-1}g'$, has a dramatic effect when it is used. However, for the test suite RREF finds an integer solution in almost all cases, and in the remainder the improvement steps recover from the poor initial solution, rendering **H** moot. If we force the algorithm to always use HNF on the test suite, then setting $c = 1$ results in 45 better and 209 worse solutions compared to the baseline. In contrast, setting $c = U^{-1}g'$ results in 21 better and 21 worse solutions; it’s almost as good as RREF. This may help other users of the library besides CUBIT.

The improvements change different steps of the algorithm. **H** and **C** change the initial solution to $Ax = b$ in each phase. **B** selects which variable to improve next. **D**, **E**, **F** and **G** select how the current solution is incremented, which vector is added. The most volatile (i.e. both improving and degrading) change is using the best improvement direction; the others improve many more tests than they degrade. The improvements do not take a significant amount of time to run. They speed up the overall algorithm in the cases where the improvements find better solutions.

8.6. Carefree Software

IIA is freely available for any use under a BSD-like license. Clone IntervalAssignment from github, `https://github.com/samitch/IntervalAssignment/`. IIA is C++11 and has *no* compile or link dependencies or required flags. Just compile it into your code.

The executable driver code test.cpp gives examples of setting up and solving a problem. A trivial “CMakeLists.txt” file is provided.

The interface is pointer-free, template-free and defined by the header files “IA.h” and “IAResult.h”. The interface is about 50 methods. The vast majority are for flexibility in defining the problem and retrieving the solution. To actually solve the problem, simply call “solve(),” or one of its other three argument-free variants, e.g., if you only want to know if the problem is feasible, or if you are resolving but want to discard the prior solution and solve from scratch. The entire code is about 10,000 lines, including comments, braces, and blank lines.

Conclusion

We have shown that Incremental Interval Assignment (IIA) is practical on today’s problems, with insignificant runtime compared to the other steps of the quad/hex meshing process, up to about 100,000 non-zeros in the constraint matrix. The software is flexible and freely available for any use.

Combining Nullspace Vectors. For future work, robustly finding integer combinations of nullspace vectors that point in downhill directions could improve the robustness and solution quality. Doing this for general problems is a longstanding open problem in integer optimization. We do not need to solve the general problem, and may instead exploit sum-even variables being in only one constraint row. Heuristics for subdividing the problem, or finding and using a subset of the nullspace vectors, might improve scalability for future-size models.

Explaining Infeasibility. Interval assignment can easily be infeasible. For example, the corners of mapped surfaces on the sides of sweeps might not be aligned with the sweep direction, or the user might have set a few curves to have some fixed values that are incompatible. If IA simply reports "infeasible" for a model with hundreds of surfaces and curves, the user is reduced to trial and error. If the constraints are feasible, but no solution was found that satisfies the bounds, IIA will report which variables are out of bounds, but not the constraints that are causing this. For future work, it would be wonderful if IIA could give the user actionable guidance about what to change to make the problem solvable. Describing a maximal problem that is feasible and a minimal subproblem that is infeasible has often been helpful in this regard. There are an exponential number of subproblems, so it would be ideal if these maximal and minimal subproblems could be found directly by integer linear algebra, rather than combinatorial exploration.

Acknowledgements

The IIA API was inspired by Paul Stalling's SGM (Scalable Geometry Modeler) API. Thank you, Paul, for teaching me that elegant design. David R. White developed CUBIT's submapping algorithm in the mid 1990's and contributed to CUBIT's interval matching strategy for submapping.

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR), Applied Mathematics Program. This work was partially supported by the Laboratory Directed Research and Development program at Sandia National Laboratories. Sandia National Laboratories is a multi-program laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

References

[1] S. A. Mitchell, A characterization of the quadrilateral meshes of a surface which admit a compatible hexahedral mesh of the enclosed volume, in: Annual Symposium on Theoretical Aspects of Computer Science, Springer, 1996, pp. 465–476.

[2] T. K. H. Tam, C. G. Armstrong, Finite element mesh control by integer programming, International Journal for Numerical Methods in Engineering 36 (1993) 2581–2605.

[3] J. Shepherd, S. Benzley, S. Mitchell, Interval assignment for volumes with holes, International Journal for Numerical Methods in Engineering 49 (1-2) (2000) 277–288.

[4] T. Li, R. McKeag, C. Armstrong, Hexahedral meshing using midpoint subdivision and integer programming, Computer Methods in Applied Mechanics and Engineering 124 (1–2) (1995) 171–193. doi:10.1016/0045-7825(94)00758-F.
URL `http://www.sciencedirect.com/science/article/pii/004578259400758F`

[5] E. Catmull, J. Clark, Recursively generated b-spline surfaces on arbitrary topological meshes, Computer-Aided Design 10 (6) (1978) 350–355. doi:https://doi.org/10.1016/0010-4485(78)90110-0.
URL `https://www.sciencedirect.com/science/article/pii/0010448578901100`

[6] T. D. Blacker, W. J. Bohnhoff, T. L. Edwards, J. R. Hipp, R. R. Lober, S. A. Mitchell, G. D. Sjaardema, T. J. Tautges, T. J. Wilson, W. R. Oakes, S. Benzley, J. C. Clements, L. Lopez-Buriek, S. Parker, M. Whitely, D. White, E. Trimble, Cubit mesh generation environment volume 1: Users manual, Tech. Rep. SAND94–1100, Sandia National Laboratories, `https://cubit.sandia.gov` (1994).

[7] T. D. Blacker, M. B. Stephenson, Paving: A new approach to automated quadrilateral mesh generation, International Journal for Numerical Methods in Engineering 32 (4) (1991) 811–847.

[8] S. A. Mitchell, Incremental interval assignment by integer linear algebra, in: International Meshing Roundtable, Vol. 29, 2021, p. 18.

[9] R. H. Möhring, M. Müller-Hannemann, K. Wiehe, Mesh refinement via bidirected flows: Modeling, complexity, and computational results, J. ACM 44 (3) (1997) 395–426. doi:10.1145/258128.258174.
URL `http://doi.acm.org/10.1145/258128.258174`

[10] D. Bommes, B. Lévy, N. Pietroni, E. Puppo, C. Silva, M. Tarini, D. Zorin, State of the art in quad meshing, in: Eurographics STARS, 2012.

[11] D. Bommes, H. Zimmer, L. Kobbelt, Mixed-integer quadrangulation, ACM Trans. Graph. (TOG) 28 (3) (2009) 77:1–77:10, commercial version in Pixologic, `http://pixologic.com/zbrush/features/QRemesher-retopology/`. doi:10.1145/1531326.1531383.
URL `http://doi.acm.org/10.1145/1531326.1531383`

[12] D. Bommes, H. Zimmer, L. Kobbelt, Practical mixed-integer optimization for geometry processing, Curves and Surfaces (2012) 193–206.

[13] D. Bommes, M. Campen, H.-C. Ebke, P. Alliez, L. Kobbelt, Integer-grid maps for reliable quad meshing, ACM Trans. Graph. 32 (4) (2013) to appear.

[14] D. R. White, T. J. Tautges, Automatic scheme selection for toolkit hex meshing, in: International Journal for Numerical Methods in Engineering, 2000, pp. 49–127.

[15] S. A. Mitchell, Choosing corners of rectangles for mapped meshing, in: Symposium on Computational Geometry, SCG '97, ACM, New York, NY, USA, 1997, pp. 87–93. doi:10.1145/262839.262906.
URL `http://doi.acm.org/10.1145/262839.262906`

[16] J. F. Shepherd, S. A. Mitchell, P. Knupp, D. R. White, Methods for multisweep automation, in: International Meshing Roundtable, 2000, pp. 77–87.

[17] S. A. Mitchell, High fidelity interval assignment, International Journal of Computational Geometry and Applications 10 (4) (2000) 399–415.

[18] E. J. Candès, M. B. Wakin, S. P. Boyd, Enhancing sparsity by reweighted ℓ_1 minimization, Journal of Fourier Analysis and Applications 14 (5) (2007) 877–905, special issue on sparsity.

[19] W. Ogryczak, T. Śliwiński, Lexicographic max-min optimization for efficient and fair bandwidth allocation, in: International Network Optimization Conference (INOC), 2007.

[20] S. A. Mitchell, Simple and fast interval assignment using nonlinear and piecewise linear objectives, in: J. Sarrate, M. Staten (Eds.), International Meshing Roundtable, Vol. 22, Sandia National Laboratories, Springer, 2013, pp. 203–221. doi:10.1007/978-3-319-02335-9.
URL https://imr.sandia.gov/papers/imr22/IMR22_12_Mitchell.pdf

[21] R. Jain, T. J. Tautges, I. Grindeanu, C. Verma, S. Cai, S. A. Mitchell, MeshKit: an open-source library for mesh generation and meshing algorithm research, in: Symposium on Trends in Unstructured Mesh Generation, 12th U.S. National Congress on Computational Mechanics, 2013.

[22] R. Viertel, B. Osting, An approach to quad meshing based on harmonic cross-valued maps and the Ginzburg–Landau theory, SIAM Journal on Scientific Computing 41 (1) (2019) A452–A479. arXiv:https://doi.org/10.1137/17M1142703, doi:10.1137/17M1142703.
URL https://doi.org/10.1137/17M1142703

[23] R. Viertel, M. L. Staten, F. Ledoux, Analysis of non-meshable automatically generated frame fields, Tech. Rep. SAND2016-9447C, Sandia National Laboratories, international Meshing Roundtable, research note, https://imr.sandia.gov/papers/abstracts/Vi831.html (2016).

[24] H. Liu, P. Zhang, E. Chien, J. Solomon, D. Bommes, Singularity-constrained octahedral fields for hexahedral meshing, ACM Trans. Graph. 37 (4) (Jul. 2018). doi:10.1145/3197517.3201344.
URL https://doi.org/10.1145/3197517.3201344

[25] K. Beatty, N. Mukherjee, A transfinite meshing approach for body-in-white analyses, in: S. Shontz (Ed.), Proceedings of the 19th International Meshing Roundtable, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 49–65.

[26] M. Lyon, D. Bommes, L. Kobbelt, Cost minimizing local anisotropic quad mesh refinement, Computer Graphics Forum 39 (5), proceedings of the Symposium on Geometry Processing special issue (2020).

[27] M. Staten, B. Carnes, C. Mcbride, C. Stimpson, J. Cox, Mesh scaling for affordable solution verification, Procedia Engineering 163 (2016) 46 – 58, 25th International Meshing Roundtable. doi:https://doi.org/10.1016/j.proeng.2016.11.015.
URL http://www.sciencedirect.com/science/article/pii/S1877705816333203

[28] S. A. Mitchell, Incremental interval assignment for mesh scaling, Tech. Rep. SAND2019-9334A, Sandia National Laboratories, international Meshing Roundtable, research abstract (2016).

[29] A. Howard, Elementary Linear Algebra with Applications: Applications Version, Wiley, 2000.

[30] J. Burkardt, ROW_ECHELON_INTEGER exact row echelon for integer matrices, https://people.sc.fsu.edu/~jburkardt/c_src/row_echelon_integer/row_echelon_integer.html (2003).

[31] D. Mitra, Finding the basis of a null space, answer to https://math.stackexchange.com/questions/88301/finding-the-basis-of-a-null-space, accessed 1 April 2020 (2012).

[32] S. Kopparty, Lecture 3: Finding integer solutions to systems of linear equations, Algorithmic Number Theory course notes, http://sites.math.rutgers.edu/~sk1233/courses/ANT-F14/lec3.pdf and section 1 of http://sites.math.rutgers.edu/~sk1233/courses/ANT-F14/lec4.pdf, accessed 1 April 2020 (2014).

[33] M. S. Hung, W. O. Rom, An application of the Hermite Normal Form in integer programming, Linear Algebra and its Applications 140 (1990) 163–179.