INCREMENTAL INTERVAL ASSIGNMENT BY INTEGER LINEAR ALGEBRA

Scott A. Mitchell

Center for Computing Research, Sandia National Laboratories, samitch@sandia.gov

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, yet many approaches perform floating-point optimization and convert a floating-point solution into an integer solution. We avoid such steps: *we start integer, stay integer.* Incremental Interval Assignment (IIA) uses integer linear algebra (Hermite normal form) to find an initial solution to the matrix equation $Ax = b$ satisfying the meshing constraints. 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. Compared to floating-point optimization approaches, IIA is faster and always produces an integer solution. The potential drawback is that there is no theoretical guarantee that the solution is optimal, but in practice we achieve solutions close to the user goals. 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. For example, midpoint subdivision imposes a form of triangle inequality. Equality constraints arise from structured meshing schemes, such as mapping with a rectangular grid of quads, and from requiring volume sweep paths to have positive and consistent lengths. 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.

Interval assignment is important for automation and meshing independence, and also for mesh quality. Conforming meshes of assemblies, or even just single parts, 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 an integer number of intervals to each curve so that the assigned value is close to the user-desired goal, and all constraints are satisfied. Once IA is solved, each surface and volume can be meshed independently and compatibly.

Incremental Interval Assignment (IIA) is a discrete algorithm over integers, based on integer linear algebra. To our knowledge, it is the first IA method to take this approach. It is a departure from floating point numerical optimization approaches. 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 $x > 0$, 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 in serial, on all realistic inputs we tested.

1.1 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 and volumes 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. 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.

1.1.1 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. [2]

In the extreme, for smooth closed surfaces used for some computer graphics models [3], IA methods have the freedom to define the structured patches themselves. Frame fields combine IA with partitioning the domain into structured patches. Frame fields are found by solving a PDE (Partial Differential Equation), then the numeric solution is "integerized" by matching adjacent frames, forming streamlines. These define the location of discrete singularities (non-4 valent surface mesh vertices) and patches. 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 [4] has way to automatically select which meshing algorithm ("scheme") to use on each surface and volume [5]. 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 [6]. In a similar way, IA is used to adjust the corners [7] of surfaces, and edge types between surfaces, to set up the structure of swept volumes [5, 8, 9].

1.1.2 Numerical Optimization

Tam and Armstrong in 1993 [10] described IA as an optimization problem with linear constraints $Ax \leq b$. The 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 formulation 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 global structure of an assembly must not conspire to link constraints such that the system reduces to one where some variables have relatively-prime coefficients. An example of this difficulty is the "radish" in section 8.2: a mapped surface has opposite sides with different numbers of curves, and all curves on a single side must have the same number of intervals.

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 [11]. Cecil Armstrong said he does not observe drastic deviations and concentrations in practice.

BBIA. In “High Fidelity Interval Assignment” (BBIA for Branch & Bound IA) [12] I designed an objective function 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 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 non-restrictive 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 resolve 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 [13].

The general outline of IIA has some similarities to BBIA. IIA’s objective is also lex min-max, but of a non-linear 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) [14] sought to improve the speed and robustness of BBIA. I switched 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 [10], 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 global structure such as the “radish” in figs. 5 and 6 and by many curves with equal sizes and goals. Such “degeneracies” are common in CAD models, with e.g., many holes and bolts of the same diameter and plates of uniform thickness. The method was deployed in MeshKit [15] but has yet to be fully productionized and extended to all available meshing algorithms.

Frame Fields. Bommes et al. [16, 17, 18] 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. A key efficiency is using the solver open-box. Connections are made and variables are integerized by successive rounding. The prior solution is updated and the solver can continue from it. The recent observation that cross field design is related to the Ginzburg–Landau problem provides additional tools for boundary alignment. [19]

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. [20, 21]

1.1.3 Greedy Approaches

Guru–protege. Beatty and Mukherjee [22] present an IA “guru–protege” method, which identifies the next curve whose interval assignment is most important, the guru, and fixes its intervals next. Protege 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.

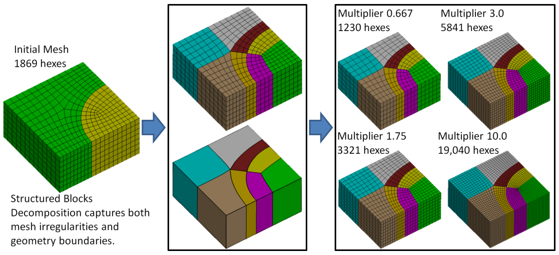


Figure 1: Mesh Scaling, courtesy Staten et al. [24].

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.

1.1.4 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 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" [23] 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 in the case that this helps runtime. The runtime is often several seconds for a few thousand elements, and sometimes minutes, and is unpredictable.

IIA for Mesh Scaling. I previously developed a simple form of Incremental Interval Assignment (IIA) for the restricted context of "Mesh Scaling" [25, 24]. 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. 1.

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.

$$\sum_A x = \sum_B x.$$

Inequality Constraints. For midpoint-subdivision and similar primitives, we have triangle-inequality type constraints:

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

Inequalities can also be used to ensure that submapping primitives have no overlapping curves or negative areas in parameter space.

Sum-Even Constraints. For an unstructured scheme, such as paving, we have constraints that the sum of intervals around the boundary must be an even number. (For CUBIT's paver, each connected component of the boundary must be even.) 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 $[2, \infty)$.

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.

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

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

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

$$R = \begin{cases} x/g & \text{if } x > g \\ g/x & \text{otherwise} \end{cases} \tag{3}$$

Note this is a linear function for $x > g$, but nonlinear otherwise.

We can also transform the bounds from a constraint to an objective. If the bounds in eq. (2) are not satisfied, we may measure how far they are from being satisfied by

$$B = \max(l - x, x - u, 0). \tag{4}$$

3. SOLUTION METHOD

Overall method. See also section 7.

- Solve, but ignore sum-even constraints.
  - Separate into independent subproblems.
  - Solve each subproblem using the *core* below.
- Solve full problem.
  - Start with the solution for x that ignored the sum-even constraints, then solve as before by identifying subproblems and solving them with the core.
- 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.

The Core.

- Solve integer $Ax = b$ for x.
  - Try Reduce Row Echelon Form (RREF) as a heuristic; see section 3.1.
  - Use Hermite Normal Form (HNF) if needed; see section 3.2.
- Satisfy bounds $l \le x \le u$.
  - 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."

We try to solve $Ax = b$ using RREF first, because we tend to get a solution that is closer to the goals. If we must use HNF, then the solution may be arbitrarily far from the bounds and goals; see section 3.3.1.

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: $\max(l-x, x-u, 0)$ 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: the new quality for all modified x_j must be better than the original quality of x_i. 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, and satisfying bounds and improving ratios use a second pivot criteria; see section 4.

3.1 Reduced Row Echelon Form (RREF)

Readers are more familiar with reduced row echelon form so we describe it before Hermite Normal Form (HNF). RREF is a generalization of diagonalization of square matrices to matrices with extra columns and redundant rows [26]. 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 [27]. For a full explanation of getting the nullspace from RREF, see Mitra [28]. We summarize the operations here.

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

where 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.

We can now "read off" the nullspace vectors from the columns of F, with minimal computation; again, see Mitra [28] for an easy-to-follow example. Here is the computation. Let $L = \mathrm{lcm}(D)$ the least common multiple of the entries of D. Each column of F contributes a nullspace vector as follows. For each column F_j of F, we multiply each entry F_{ji} by $-L/D_{ii}$ and append the elementary unit column vector e_j so it has the same length as the rows of A. The transpose of this vector is a row vector in the nullspace.

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.

3.2 Hermite Normal Form (HNF)

We use Hermite Normal Form (HNF) to solve integer $Ax = b$; see Kopparty [29] for a more complete description. Readers may be familiar with using Gaussian elimination and RREF to solve *floating point* $Ax = b$; see section 3.3 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 [30].)

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. We use the column-form of HNF. 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 nonnegative.

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 $\mathrm{RREF}(A)x = Ax$, we start with RREF(A) instead of A, and discard any zero rows so we are dealing with matrices of full rank. We initialize $H = \mathrm{RREF}(A)$ 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 (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 its column to reduce it; see "Algorithm: ReduceOffDiagonal" [29] 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.

3.3 Discussion: Why RREF and HNF?

RREF reduces the constraints to a minimal independent set. Creating (discovering) rows of zeros at the bottom of the matrix identifies the presence of redundant constraints. For example, consider four faces of a cube meeting curve to curve, and we wish to map-mesh each of them. There are four curve variables, and four equality constraints: intervals on the two curves on opposing sides of a face must be equal. Thus RREF reduces

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

to

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

RREF allows the generation of vectors spanning the nullspace of A. However, RREF is also useful before HNF, because it identifies the rank of A, and allows us to use methods for generating HNF that depend on the matrix being full rank. But RREF is not sufficient by itself to solve $Ax = b$ for nontrivial 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.

3.3.1 Solve Integer $Ax = b$

The drawback to using HNF to solve $Ax = b$ is that the solution x is not influenced by the goals or bounds on x, and may be very far from both. It may be possible to choose these c variables such that x are closer to their goals or in bounds; but the only strategies that have come to mind are equivalent to adding nullspace vectors as a post-process, which is precisely what the next step of our overall algorithm does anyway. So, before we resort to HNF, we attempt to find a solution to $Ax = b$ using the RREF we found with our pivot-selection heuristics. If we are stuck with something resembling eq. (6), then we do HNF.

When doing back substitution with RREF, we assign initial values to edge variables equal to their goals, and each dummy variable the value that makes its row constraint satisfied. (If the dummy variable coefficient is not 1, e.g., is 2, then its initial value may have to be rounded up to the next integer and the constraint is not satisfied.) Thus all RREF independent variables are close to their goals, and only the dependent variables may be out of bounds.

4. PIVOT SELECTION HEURISTICS

When forming the RREF (Reduced Row Echelon Form) to generate nullspace vectors, 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. The choice of pivot affects the reliability of the increment step. Ideally, for satisfying constraints, we would like initial nullspace vectors that allow us to increment the out-of-bound variables while keeping other variables in bounds. We have similar desires for improving variables towards their goals.

With that motivation, we select RREF pivots with the following 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 if multiple variables have the same best primary criteria value. If no desirable variables are left, we pivot on an arbitrary variable, using the row in which its coefficient is smallest. 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.

4.1 Pivots For Satisfying Constraints

- 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.
  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 slack or dummy variables, followed by variables with a larger goal.
- If the variables with a 1 coefficient are exhausted, we select variables based on the gcd of its coefficients. The thinking is that if the gcd is 1, then it is possible to combine rows to get a leading coefficient of 1.

These choices help 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.

4.2 Pivots For Bounds and Goals

We use these criteria to help keep rows short, i.e. few non-zeros. We also seek to deter generating rows with many sum-even dummy variables, since these couple multiple paving surfaces. We never pick sum-even dummy variables with a coefficient of 2, and never pick equality-constraint slack variables. We select amongst the remaining variables as follows.

1. The primary criteria is the number of rows it appears in; fewer is better. Thus we first pick 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. We first pick any row n that does not make *any* variable closer to its bounds. E.g., all nonzeros of n are positive and correspond to x_j with no upper bound. Otherwise, we use the first row we find that gives strict improvement, defined just in terms of the bounds, and ignoring the secondary criteria of the goals. I.e., the new values of all changed x_j must be further from their respective bounds than the old value of x_i was from its bound. 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.

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 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. The interval assignment for mesh scaling approach [25] got 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, i.e., $f(x \pm 1, g)$, rather than its current priority, i.e., $f(x, g)$. So, we use $f(x \pm 1, g)$ in our context as well.

It is common for some variables to "self block", meaning they are (nearly) equal to their goal and incrementing them in either direction makes the solution worse. Sometimes, however, this is just because their coefficients are large (not 1) in the nullspace vectors, so reducing the coefficient is sufficient to enable improvement. We do not search for small subspaces; it is expected that some variables will not be at their optimal values due to the global constraints.

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 [12].

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

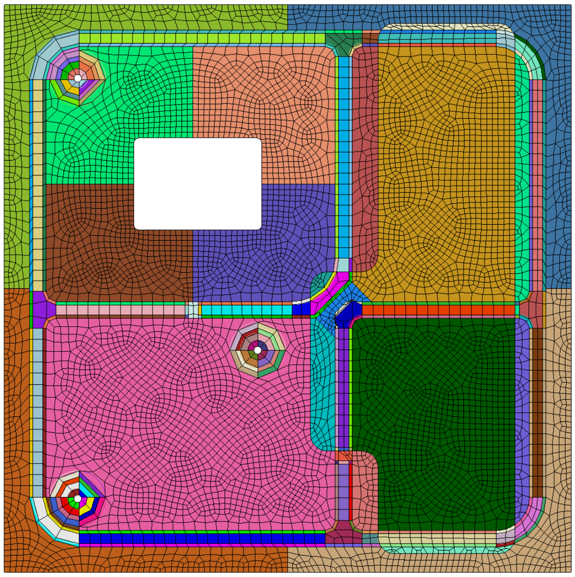


Figure 2: IIA with quad mesh paving 181 surfaces with the "skeleton" sizing from the open CUBIT regression tests.

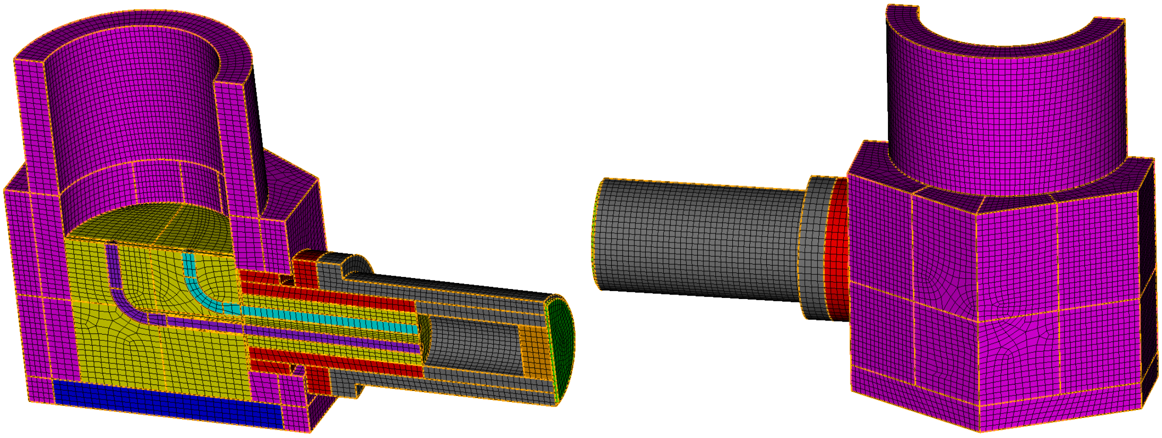


Figure 3: IIA with hex mesh sweeping 56 interlocking volumes, exercising auto scheme selection, sweeping constraints and verification. The problem is made more constrained for IA because some curves' intervals are user-prescribed and cannot change. Front and back view. From the open CUBIT regression tests.

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 non-linear 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. 3 for an example. The test suite also contains many surface-meshing problems; see fig. 2 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. 4.

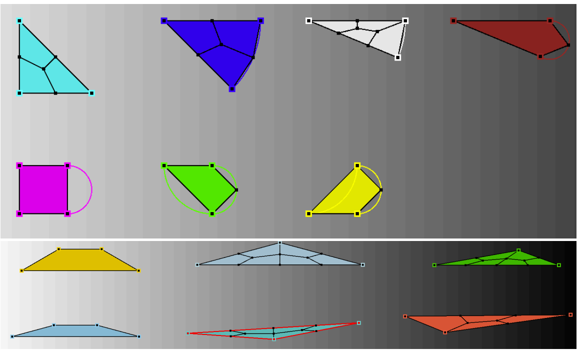


Figure 4: IIA supports interval lower bounds based on geometric reasoning. Quads are linear and solid color; non-linear CAD curves appear outside the elements. CAD vertices are surrounded by a small colored box. From the open CUBIT regression tests.

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.

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 mesh-

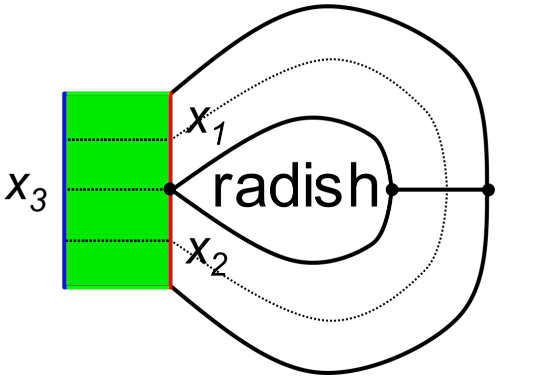


Figure 5: The "2-1 radish" assembly model circa 1997 from Mitchell [12]. The global structure of the assembly constrains the curves on the red side to all have the same intervals: $x_1 = x_2$. Thus $x_3 = x_1 + x_2$ reduces to $2x_1 = 1x_3$.

ing schemes give rise to this type of constraint; see fig. 5. While it is easy to see how the "2-1 radish" of fig. 5 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. 6 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. 6 and 7. 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. 7, 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. 6c. For very coarse sizes, when the initial floating point solution is $< \{7, 5\}0.5$, BBIA fails because the relaxed solution is too far from the nearest integer solution; see fig. 8. BBIA has a search factor cutoff of 2 in one of its steps to avoid large runtimes for other problems, especially infeasible ones.

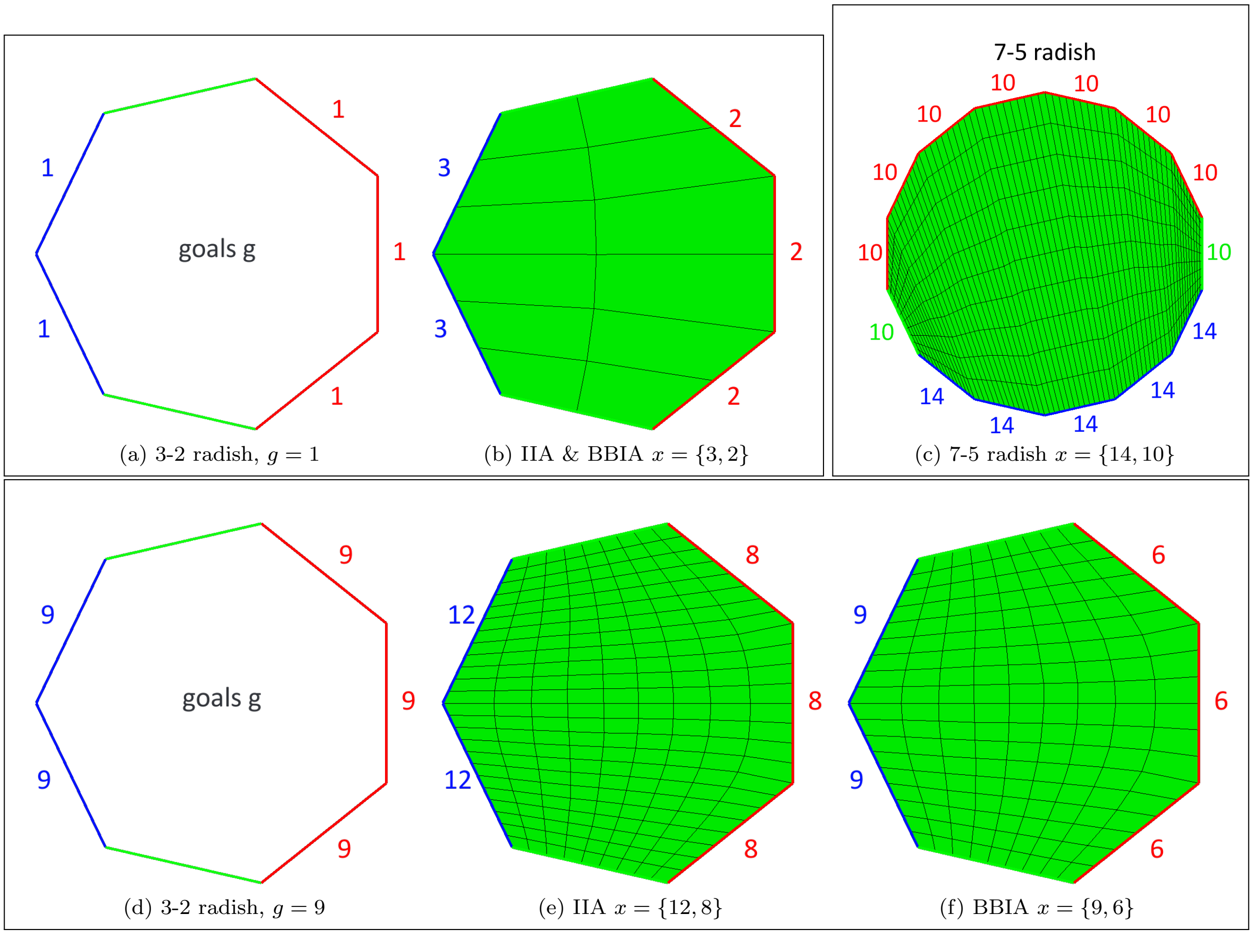
79-74 Radish. Here 79 and 74 are relatively prime, so the only solutions are $\{79, 74\}k$. This model has the same problems with coarse solutions as the 7-5 radish: BBIA fails when the relaxed solution is $< \{79, 74\}0.5$. Further, BBIA also fails for some intermediate sizes. We speculate that failure 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. 9.

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.



(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 6: 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 some natural number k.

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 sub-exponential complexity bounds are difficult to obtain.

Our scaling challenge is the heat-sink mock-up in fig. 10, 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

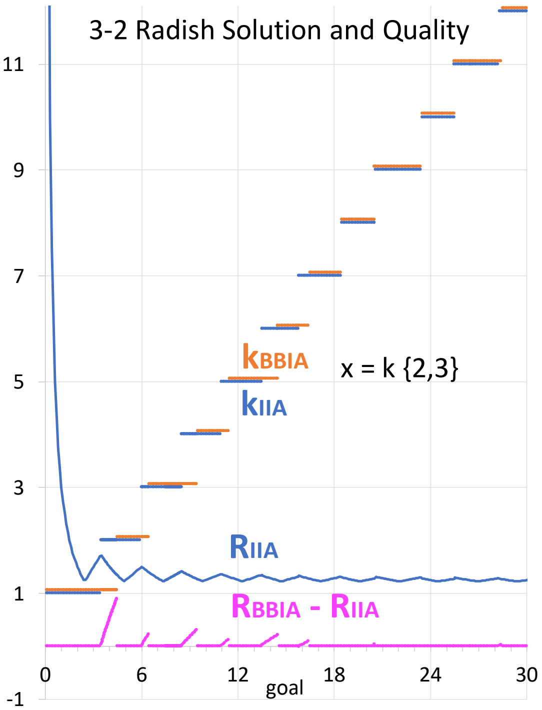


Figure 7: 3-2 radish solution and quality for different goals. The solution is $k\{2, 3\}$, e.g., the solution for $g = 7.5$ is $k = 3$ and $x = \{6, 9\}$ for both IIA and BBIA. For each constant-x IIA-solution interval, the quality R is best toward the middle of the interval, and worst at the ends, 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 “Rbbia-Riia” 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.

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

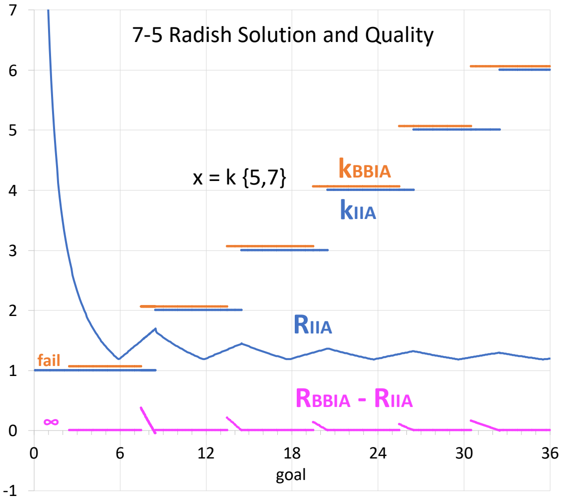


Figure 8: 7-5 radish solution and quality for different goals. BBIA fails for $g < 2.5$. When the BBIA and IIA solutions differ, the BBIA solution is almost always of poorer quality. Here the exception is a very small, 0.06-neighborhood around $g = 8.44$. But, in contrast to the 3-2 radish, the BBIA solution is often *larger* than the IIA solution.

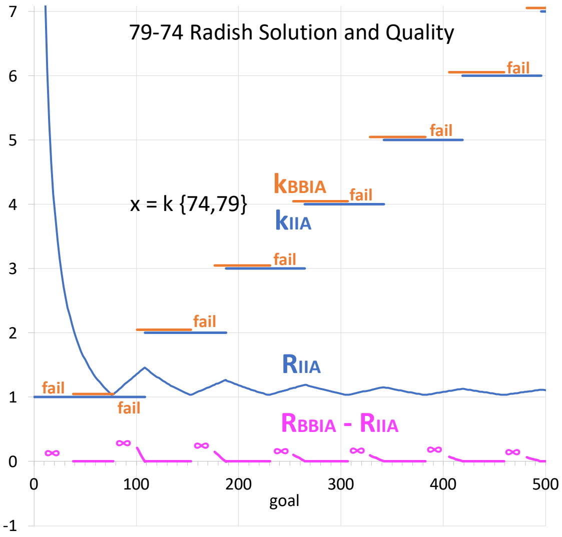


Figure 9: 79-74 radish solution and quality for different goals. BBIA fails for $g < 38.5$, and for some intervals around where the optimal solution transitions from k to $k+1$. The BBIA solution is usually at least as large as the IIA solution. The IIA solution is better than the BBIA solution, except for a small neighborhood around $g = 187.49$.

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, al-

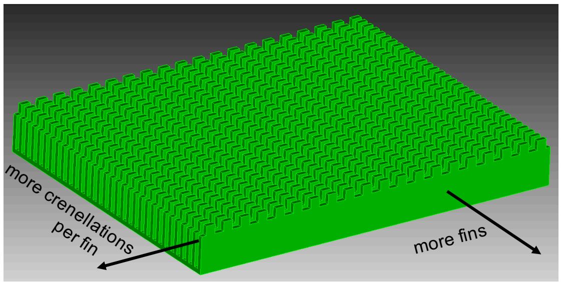


Figure 10: Heat sink mock-up. With the long curves constrained to fixed intervals, BBIA takes 20 minutes to solve this problem, but CUBIT with IIA can solve it in 0.2 seconds. This is a 6000× speedup.

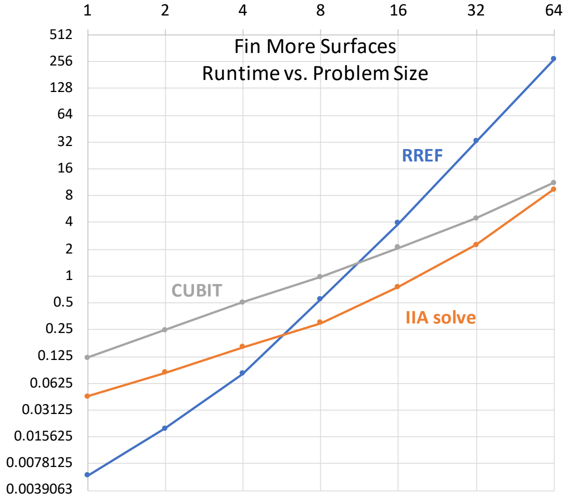


Figure 11: Runtime scaling (seconds) for multiplying the heat sink by creating more fins.

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

See Figures 11 to 13. 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,

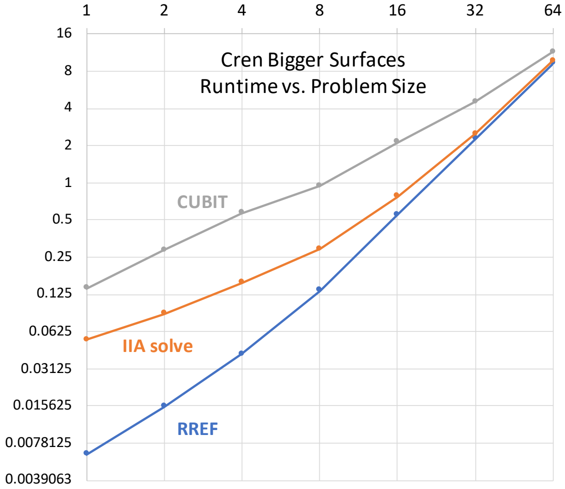


Figure 12: Runtime scaling (seconds) for multiplying the heat sink by making the fins longer, with more crenellations.

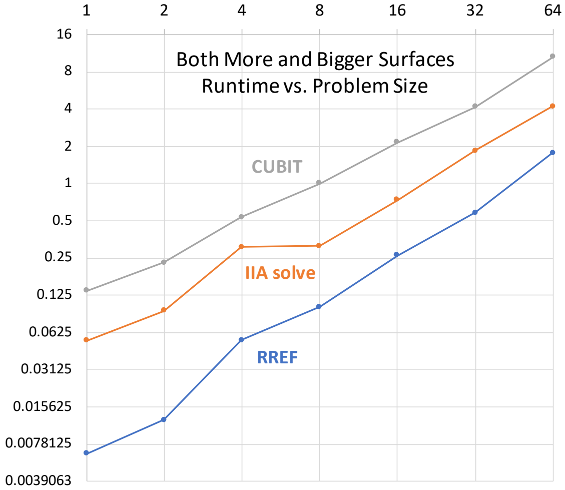


Figure 13: Runtime scaling (seconds) for multiplying the heat sink by both creating more fins and making them longer.

with the slope indicating the exponent of the polynomial complexity. For example, the "CUBIT" time in fig. 11 is roughly linear in problem size up until the largest models. "RREF" is cubic for fig. 11 and quadratic for fig. 12. For each of Fin and Cren "IIA solve" is close to linear up to about size 8 (20k nonzeros) 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 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 resolve 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.

8.3.3 Limitations for Huge Pave-and-Sweep Models

Performing the heat-sink scaling study with pave-and-sweep, and also with all-paving surfaces, uncovered a robustness issue. The heat-sink example works up to about a 4× or 8× size, but sometimes for larger model sizes coupled with very coarse mesh sizes, IA fails 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. The root cause is 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 gives up.

The solution follows. 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. I implemented a method for augmenting the RREF nullspace vectors with vectors such as these, and the satisfy-bounds step no longer gets stuck. It remains to ensure the runtime is not affected too much.

8.4 Carefree Software

IIA is freely available for any use under a BSD-like license. Simply clone IntervalAssignment from github; browse to `https://github.com/samitch/IntervalAssignment`. IIA is C++11 and has *no* compile or link-time dependencies or required flags. Simply 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 slightly less than 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 nonzeros in the constraint matrix. The software is flexible and freely available for any use.

For future work, robustly finding integer combinations of nullspace vectors that point in downhill directions could improve the robustness and solution quality, but this is a longstanding open problem in integer optimization. Heuristics for subdividing the problem, or finding and using a subset of the nullspace vectors, might improve scalability for future-size models.

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. When IA simply reports "infeasible" for a model with hundreds of surfaces and curves, the next command the user issues to the program may be an explicative. For future work, it would be wonderful if IIA could give the user actionable guidance about what to change to make the problem feasible. This might be achieved by testing whether certain subproblems are feasible, or exploring which variables have unachievable bounds.

ACKNOWLEDGEMENTS

The IIA API was inspired by Paul Stalling's SGM (Scalable Geometry Modeler) API. Thank you, Paul, for teaching me that elegant design.

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] Mitchell S.A. "A Characterization of the Quadrilateral Meshes of a Surface which Admit a Compatible Hexahedral Mesh of the Enclosed Volume." *Annual Symposium on Theoretical Aspects of Computer Science*, pp. 465–476. Springer, 1996

[2] Möhring R.H., Müller-Hannemann M., Wiehe K. "Mesh Refinement via Bidirected Flows: Modeling, Complexity, and Computational Results." *J. ACM*, vol. 44, no. 3, 395–426, May 1997. URL http://doi.acm.org/10.1145/258128.258174

[3] Bommes D., Lévy B., Pietroni N., Puppo E., Silva C., Tarini M., Zorin D. "State of the Art in Quad Meshing." *Eurographics STARS*. 2012

[4] Blacker T.D., Bohnhoff W.J., Edwards T.L., Hipp J.R., Lober R.R., Mitchell S.A., Sjaardema G.D., Tautges T.J., Wilson T.J., Oakes W.R., Benzley S., Clements J.C., Lopez-Buriek L., Parker S., Whitely M., White D., Trimble E. "CUBIT Mesh Generation Environment Volume 1: Users Manual." Tech. Rep. SAND94–1100, Sandia National Laboratories, 1994. https://cubit.sandia.gov

[5] White D.R., Tautges T.J. "Automatic Scheme Selection for Toolkit Hex Meshing." *International Journal for Numerical Methods in Engineering*, pp. 49–127. 2000

[6] Blacker T.D., Stephenson M.B. "Paving: A New Approach to Automated Quadrilateral Mesh Generation." *International Journal for Numerical Methods in Engineering*, vol. 32, no. 4, 811–847, 1991

[7] Mitchell S.A. "Choosing Corners of Rectangles for Mapped Meshing." *Symposium on Computational Geometry*, SCG '97, pp. 87–93. ACM, New York, NY, USA, 1997. URL http://doi.acm.org/10.1145/262839.262906

[8] Shepherd J., Benzley S., Mitchell S. "Interval Assignment for Volumes with Holes." *International Journal for Numerical Methods in Engineering*, vol. 49, no. 1-2, 277–288, 2000

[9] Shepherd J.F., Mitchell S.A., Knupp P., White D.R. "Methods for Multisweep Automation." *International Meshing Roundtable*, pp. 77–87. October 2000

[10] Tam T.K.H., Armstrong C.G. "Finite Element Mesh Control by Integer Programming." *International Journal for Numerical Methods in Engineering*, vol. 36, 2581–2605, 1993

[11] Candès E.J., Wakin M.B., Boyd S.P. "Enhancing Sparsity by Reweighted ℓ_1 Minimization." *Journal of Fourier Analysis and Applications*, vol. 14, no. 5, 877–905, 2007. Special issue on sparsity

[12] Mitchell S.A. "High Fidelity Interval Assignment." *International Journal of Computational Geometry and Applications*, vol. 10, no. 4, 399–415, 2000

[13] Ogryczak W., Śliwiński T. "Lexicographic Max-Min Optimization for Efficient and Fair Bandwidth Allocation." *International Network Optimization Conference (INOC)*. 2007

[14] Mitchell S.A. "Simple and Fast Interval Assignment Using Nonlinear and Piecewise Linear Objectives." J. Sarrate, M. Staten, editors, *International Meshing Roundtable*, vol. 22, pp. 203–221. Sandia National Laboratories, Springer, 2013. URL `https://imr.sandia.gov/papers/imr22/IMR22_12_Mitchell.pdf`

[15] Jain R., Tautges T.J., Grindeanu I., Verma C., Cai S., Mitchell S.A. "MeshKit: An Open-source Library for Mesh Generation and Meshing Algorithm Research." *Symposium on Trends in Unstructured Mesh Generation, 12th U.S. National Congress on Computational Mechanics*. 2013

[16] Bommes D., Zimmer H., Kobbelt L. "Mixed-integer Quadrangulation." *ACM Trans. Graph. (TOG)*, vol. 28, no. 3, 77:1–77:10, Jul. 2009. URL `http://doi.acm.org/10.1145/1531326.1531383`. Commercial version in Pixologic, `http://pixologic.com/zbrush/features/QRemesher-retopology/`

[17] Bommes D., Zimmer H., Kobbelt L. "Practical Mixed-Integer Optimization for Geometry Processing." *Curves and Surfaces*, pp. 193–206, 2012

[18] Bommes D., Campen M., Ebke H.C., Alliez P., Kobbelt L. "Integer-Grid Maps for Reliable Quad Meshing." *ACM Trans. Graph.*, vol. 32, no. 4, to appear, 2013

[19] Viertel R., Osting B. "An Approach to Quad Meshing Based on Harmonic Cross-Valued Maps and the Ginzburg–Landau Theory." *SIAM Journal on Scientific Computing*, vol. 41, no. 1, A452–A479, 2019. URL `https://doi.org/10.1137/17M1142703`

[20] Viertel R., Staten M.L., Ledoux F. "Analysis of Non-Meshable Automatically Generated Frame Fields." Tech. Rep. SAND2016-9447C, Sandia National Laboratories, 2016. International Meshing Roundtable, research note, `https://imr.sandia.gov/papers/abstracts/Vi831.html`

[21] Liu H., Zhang P., Chien E., Solomon J., Bommes D. "Singularity-Constrained Octahedral Fields for Hexahedral Meshing." *ACM Trans. Graph.*, vol. 37, no. 4, Jul. 2018. URL `https://doi.org/10.1145/3197517.3201344`

[22] Beatty K., Mukherjee N. "A Transfinite Meshing Approach for Body-In-White Analyses." S. Shontz, editor, *Proceedings of the 19th International Meshing Roundtable*, pp. 49–65. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010

[23] Lyon M., Bommes D., Kobbelt L. "Cost Minimizing Local Anisotropic Quad Mesh Refinement." *Computer Graphics Forum*, vol. 39, no. 5, 2020. Proceedings of the Symposium on Geometry Processing special issue

[24] Staten M., Carnes B., Mcbride C., Stimpson C., Cox J. "Mesh Scaling for Affordable Solution Verification." *Procedia Engineering*, vol. 163, 46 – 58, 2016. URL `http://www.sciencedirect.com/science/article/pii/S1877705816333203`. 25th International Meshing Roundtable

[25] Mitchell S.A. "Incremental Interval Assignment for Mesh Scaling." Tech. Rep. SAND2019-9334A, Sandia National Laboratories, 2016. International Meshing Roundtable, research abstract

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

[27] Burkardt J. "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

[28] Mitra D. "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

[29] Kopparty S. "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

[30] Hung M.S., Rom W.O. "An Application of the Hermite Normal Form in Integer Programming." *Linear Algebra and its Applications*, vol. 140, 163–179, 1990