Scalable DOGs

Linus Wigger

March 16, 2020

1 Abstract

1 Abstract

2 Introduction

The simulation of developable surfaces as been a topic of interest in computer graphics for years. A surface is called developable if it is locally isometric to a plane. In other words, it can be obtained by bending a flat shape without stretching it. In areas like architecture, developable surfaces are useful because they are easy to manufacture. The problem is that developable surfaces are hard to design because many constraints have to be satisfied for a surface in 3D to be developable.
The base of this project is formed by the work from [Rabinovich et al. 2018a], where the idea of using Discrete Orthogonal Geodesic nets (abbreviated as DOGs) to simulate deformations of developable surfaces is used. This was followed up by [Rabinovich et al. 2018b], which goes into more detail about the possible deformations, and [Rabinovich et al. 2019], where the focus lay on folding with curved creases.
[Rabinovich et al. 2018a]'s approach to modeling developable surfaces has advantages to other, previous methods in terms of flexibility. They do not depend on predefined rulings, making it possible to explore the full shape space of possible deformations without having to change the mesh topology.
[Rabinovich et al. 2018a] made an editor where the user can interactively deform a DOG. Said editor supports curved folds (starting in [Rabinovich et al. 2018b], extended by [Rabinovich et al. 2019]) and various types of user-defined constraints. The goal of this project was to work on the solver algorithm, mostly trying to make it suited for parallelization later on.

3 Problem setting

The DOG is a quad mesh consisting of sets of vertices V, edges E and faces F, joined by many constraints of various types. The mesh is initially constructed from a crease pattern - a two-dimensional set of curves defining the flat shape of the surface and its folds. The creases are fixed at construction, meaning the mesh topology remains constant afterwards and the user can't add or remove creases.

Figure 3.1 shows an example of a crease pattern and the resulting mesh. The given resolution 9×7 is the number of vertices in x and y direction, leading to 8×6 base quads. The parts around the crease are coloured green to signify that they are actually duplicated. All of the patches separated by the creases are actually disconnected, as seen in figure 3.2. They are held together by constraints on the curve points (which are shown in pink). For those constraints to work, all patches involved need to contain the relevant mesh elements, hence the duplication. Crease patterns with more creases can end up with more than two instances of one vertex.

A constrained optimization problem is then formulated to allow for specific deformations of the mesh. Bending, rigid transformations, and folding along creases are allowed. Stretching should not be possible. At the same time, certain other constraints must be satisfied.

The most important constraints are the "DOG constraints". As described [Rabinovich et al. 2018a], all angles between edges around a single vertex mus be equal to guarantee that the surface is developable. The DOG constraints are slightly different on the boundary: Corner vertices have only two adjacent edges, and the angle between them must be equal to their angle in the original flat state.

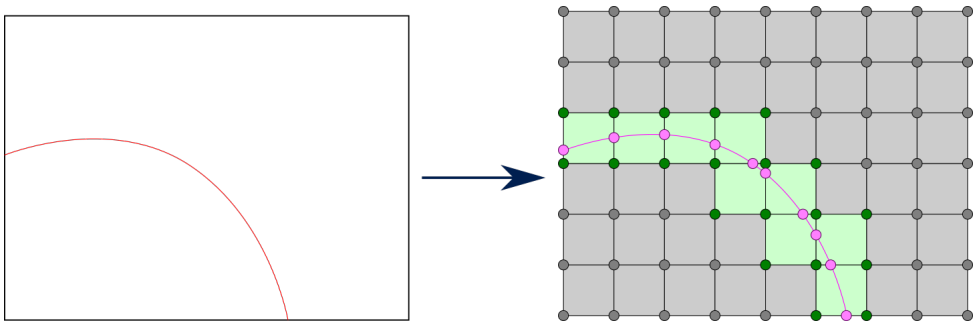

Figure 3.1: *Left: A crease pattern with one fold; Right: DOG obtained from the crease pattern with resolution* 9×7*; Green mesh elements have duplicates*

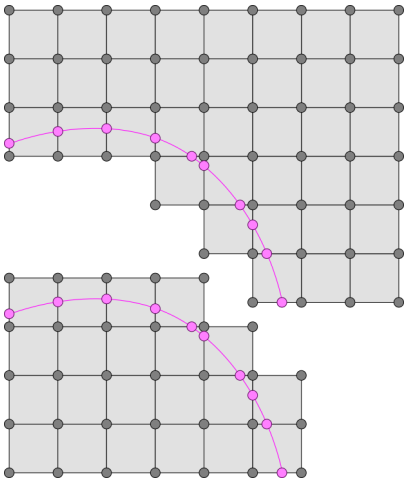

Figure 3.2: *Patches when separated from each other*

3.1 Objective function

The goal of the optimization is to minimize the bending energy of the DOG while satisfying all constraints. Most of the constraints are soft, meaning they are part of the objective function. The only constraints that always get enforced strictly are the DOG constraints. The objective function ends up as a weighted sum of various energy terms.

$$f_{obj}(\mathbf{x}) = w_{bend} \cdot E_{bending}(\mathbf{x}) + w_{iso} \cdot E_{isometry}(\mathbf{x}) + w_{soft} \cdot E_{posconst}(\mathbf{x}) + ... \tag{3.1}$$

$\mathbf{x}$ in this case is a vector of size $3|V|$, containing the coordinates of all mesh vertices.
The bending energy is computed by comparing the Laplacian L of the current configuration to the Laplacian of the flat state.

$$E_{bending}(\mathbf{x}) = |L(\mathbf{x_0}) - L(\mathbf{x})|^2 \tag{3.2}$$

The DOG Laplacian was developed in [Rabinovich et al. 2018b], chapter 6 to be specific.
Among the soft constraints, the following are the most important:
The isometry constraints ensure that all edges in the mesh keep their original length. For each edge e_i, the squared difference between the current squared length l_i^2 and the target squared length l_{0i}^2 is added to the objective, where l_{0i} is the length that the edge e_i has in the initial state. The weight associated with the isometry constraints should be rather high to ensure that almost no stretching happens. This is important because the allowed transformations for developable surfaces do not include stretching.

$$E_{isometry}(\mathbf{x}) = \sum_{i=1}^{|E|} (l_i^2(\mathbf{x}) - l_{0i}^2)^2 \tag{3.3}$$

Position constraints are the most common user-defined constraints. The user can select certain vertices to constrain, then drag them around so they have the desired positions. Each position constraint adds a simple spring energy to the objective. Assuming P position constraints, each constraint having an associated vertex position $\mathbf{v}_i$ and target position $\mathbf{p}_i$, the energy from the

position constraints can be written as:

$$E_{posconst}(\mathbf{x}) = \sum_{i=1}^{P} (\mathbf{v}_i(\mathbf{x}) - \mathbf{p}_i) \cdot (\mathbf{v}_i(\mathbf{x}) - \mathbf{p}_i) \tag{3.4}$$

Similar to the position constraints are the edge point constraints. Their only difference is that they are defined on an edge point $\mathbf{e}$ instead of a mesh vertex $\mathbf{v}$. Edge points are defined by an edge $e \in E$ and a relative distance $t \in]0, 1[$. As such, the edge point constraints add the following term to the objective:

$$E_{edgepoints}(\mathbf{x}) = \sum_{i=1}^{R} (\mathbf{e}_i(\mathbf{x}) - \mathbf{r}_i) \cdot (\mathbf{e}_i(\mathbf{x}) - \mathbf{r}_i) \tag{3.5}$$

$$\mathbf{e_i}(\mathbf{x}) = t_i \mathbf{v}_{ai}(\mathbf{x}) + (1 - t_i)\mathbf{v}_{bi}(\mathbf{x}) \tag{3.6}$$

Where R is the number of edge point constraints, the r_i are the target positions, and $\mathbf{v}_{ai}$ and $\mathbf{v}_{bi}$ stand for the two vertices connected by the edge. It should be noted that the users can't define edge point constraints directly, but they are still used sometimes.
The stitching constraints have a very important role. They ensure that the different patches of the DOG are actually connected to each other. The way they are implemented resembles the edge point constraints. Instead of moving an edge point to a specific location, they hold two different edge points together.

$$E_{stitching}(\mathbf{x}) = \sum_{i=1}^{C} (\mathbf{c}_{ai}(\mathbf{x}) - \mathbf{c}_{bi}(\mathbf{x})) \cdot (\mathbf{c}_{ai}(\mathbf{x}) - \mathbf{c}_{bi}(\mathbf{x})) \tag{3.7}$$

The stitching constraints are generated automatically and can't be changed by the user (except for their weight). The stitching constraints are applied on all C crease points, which are the points where the crease curves intersect the mesh edges. They come in pairs $\mathbf{c}_{ai}$ nad $\mathbf{c}_{bi}$, corresponding to duplicate edge points in two different patches.
Angle constraints can also be defined on crease points. The solver will then attempt to make the folding angle at that location equal to the desired angle. The objective works with the cosine of the angles, making angles with absolute value greater than $\frac{\pi}{2}$ problematic. The energy for A angle constraints between edges e_{ai} and e_{bi} from different patches is computed as:

$$E_{angleconst}(\mathbf{x}) = \sum_{i=1}^{A} (\cos(\angle(e_{ai}, e_{bi})(\mathbf{x})) - \cos(\alpha_i))^2 \tag{3.8}$$

The angle α_i between the edges involved is not actually the same as the folding angle, which is defined by the surface tangents. This means that the correct angle α_i between edges from different patches needs to be computed when the constraint is created. α_i gets computed automatically using the nearby curve points. The full explanation can be found in section 5.1 of [Rabinovich et al. 2019]: Given a target folding angle θ, and the angle β (called α in [Rabinovich et al. 2019]) between the surface tangent on one patch and the crease curve tangent, α must satisfy

$$\cos\alpha = \cos^2\beta + \sin^2\beta\cos\theta \tag{3.9}$$

There are still more parts of the objective function. Some are variants of the elements above (e.g. paired vertices), others are rarely relevant and have more complicated formulas. They will not be described in more detail.

3.2 Optimization

The optimization is formulated as a Sequential Quadratic Programming problem and solved using the PARDISO solver ([De Coninck et al. 2016], [Verbosio et al. 2017], [Kourounis et al. 2018]). Each iteration starts by updating all constraints. Following that, PARDISO computes a better solution for the current constraints. As mentioned, the DOG constraints are enforced directly in PARDISO while other constraints like the user-defined ones are part of the objective function, but that is not all:
There are also folding constraints which make sure that the patches are folded in the same direction on all crease points in a single curve. One of the extra terms of the objective function mentioned at the end of the previous section is used for this. It wasn't described explicitly there because the computations involved are not human-readable and the value of the end result is usually close to zero once the mesh is no longer flat.
That said, the folding constraints are enforced more strictly than others: If the mesh isn't folded when using the result from PARDISO, then the vertex positions get rolled back and the iteration is repeated with a higher weight ($\times 10$) for the folding constraints. This can happen multiple times in "one" iteration, but it rarely happens more than once.
There is a checkbox in the editor where the user can disable "folding mode". For simple crease patterns, this has only minor effects and the result looks almost the same. More problematic are larger meshes with multiple folds, especially when said folds result in many patches connected in series. In those cases, disabling folding mode makes it likely that adjacent patches are not folded, meaning there is no C^0 discontinuity on the patch border. This is undesirable because it defeats the point of having the creases.
For more details on the folding, see [Rabinovich et al. 2019], particularly section 6.3. The methods introduced later on in this work tend to not care too much about the folding being correct, especially because they split the problem across patches and the folding constraints are difficult to enforce locally.

4 ADMM

As can be seen in the previous chapter, most parts of the objective function are separable in the sense that they can be split into contributions from different patches. This is the case for the bending energy, isometry constraints, and position constraints. The stitching and angle constraints connect vertices across patch boundaries and are therefore not separable.
The idea is to use the alternating direction method of multipliers (ADMM) as described in [Deng et al. 2017]. This method can be applied to a problem of the form

$$\begin{aligned} \text{min} \quad & \Sigma_{i=1}^{N} f_i(\mathbf{x}_i) \\ \text{s.t.} \quad & \Sigma_{i=1}^{N} A_i \mathbf{x}_i = c \end{aligned} \tag{4.1}$$

If N is the number of patches, f_i would be the objective function on the submesh corresponding to the patch i. The second part corresponds to patch-crossing constraints like the stitching constraints. The advantage of ADMM is that the local subproblems can be solved in parallel.

4.1 Variant: Subsolvers

The most basic ADMM-like method optimizes each patch separately. The stitching constraints on the global mesh are replaced by edge point constraints on the patches. After each iteration, these edge point constraints get updated with the positions obtained in the adjacent patches. This method requires no additional objective terms.

4.2 Variant: Variable Splitting ADMM

Variable Splitting ADMM (see algorithm 1 in [Deng et al. 2017]) works similar to Subsolvers but adds a few extra elements: The local objective functionsare extended by an additional term $\frac{\rho}{2}|A_i - z_i^{k+1} - \frac{\lambda_i^k}{\rho}|^2$. λ and z_i are extra variables that are updated in each iteration while ρ is a new parameter that was usually simply set to 1. The z_i are set to $A_i x_i - \frac{1}{N}\sum_{j=1}^{N}(A_j x_j + \frac{\lambda_j}{\rho})$. After obtaining a new x_i from the solver, the λ_i get updated to $\lambda_i - \rho(A_i x_i - z_i + \frac{\lambda_i}{\rho})$

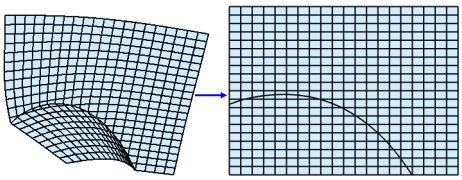

Figure 4.1: *Setup for ADMM testing: Left is the initial state, right is how the final result should look like*

4.3 Variant: Proximal Jacobian ADMM

Proximal Jacobian ADMM (see algorithm 4 in [Deng et al. 2017]) adds another extra term to VSADMM and changes the way z_i and λ are computed. A proximal term $\frac{\rho}{2} dx_i^T P dx_i$ is added, where dx_i is the difference $x_i - x_{0i}$ (x_{0i} being the resulting x_i from the previous iteration) and P is a diagonal matrix. z_i in this variant is computed as $A_i x_i - \sum_{j=1}^{N} + \frac{\lambda}{\rho}$ and λ (which is constant across patches) becomes $\lambda - \rho\gamma \sum_{i=1}^{N} A_i x_i$ (γ being a damping parameter > 0).

4.4 ADMM results

The ADMM methods are not good for this problem. This can be demonstrated by a simple test: Starting from a folded configuration as seen in figure 4.1, the goal is to minimize the bending energy by flattening the mesh. There are only minimal constraints: Beyond the fixed constraints (DOG, stitching, etc.) there is only a single positional constraint.
Figure 4.2 plots the objective function for this simple problem over 1000 iterations. Even ignoring the exact numbers (because all methods have slightly different objective functions) the existing "standard" solver reaches a solution very quickly (less than 50 iterations after which the solution stays constant). The ADMM-based methods are slow at best and at worst there are stability issues: Proximal Jacobian ADMM can spontaneously break everything, even long after it looks like the DOG has stabilized.
Angle constraints are difficult to enforce in the ADMM solver because, like the stitching constraints, they connect vertices from different patches to each other. Updating the coordinates of points outside the local patch after each iteration causes heavy jittering in the results (due to overshooting from both sides). Angle constraints could in theory be reformulated to work with a crease curve of the DOG instead of arbitrary edges. This would allow them to be split into two half-constraints that use only local information on each patch.
The main problem with the ADMM variants is that they assume the crease curves to be constant. Given that finding the shape of the curves in 3D is one of the more important aspects of the optimization, this is a very bad assumption. Additionally, ADMM as described in [Deng et al. 2017] assumes that the problem is convex, which it is not (due to all the constraints).

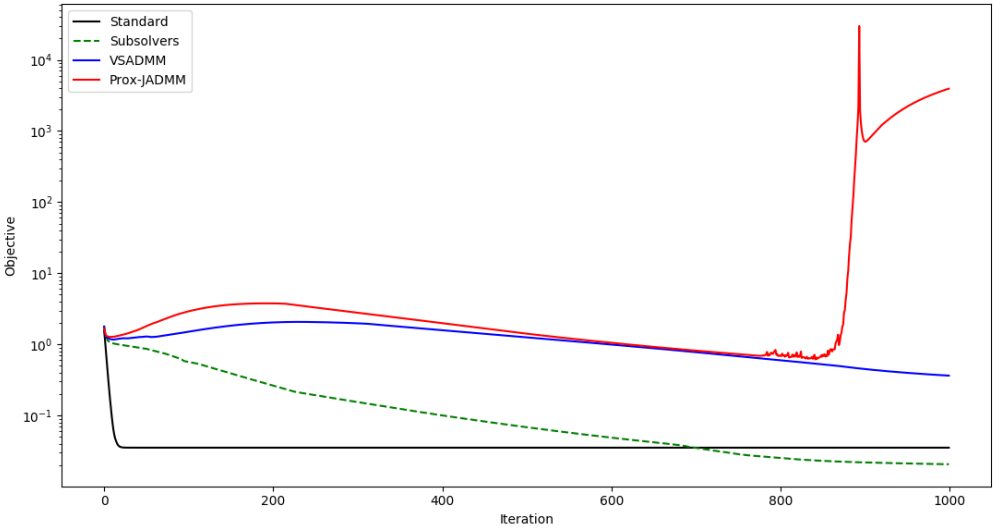

Figure 4.2: *Objective function values for flattening a simple DOG*

5 Coarse method

It's very important to know where in 3D the crease curves end up but the ADMM methods are unsuited for that. An approach that can handle deformations of the curves was needed.
The idea behind the following method is to quickly compute a guess for the crease curves in 3D and use these to optimize all patches locally. To compute the guess, a coarse version of the global DOG will be used.
In the spirit of multigrid methods, a single iteration with the coarse mesh works in four main steps, pictured in figure 5.1:

- Coarse solve
- Coarse-to-fine update
- Fine solve
- Fine-to-coarse update

The coarse solve is a standard global solver iteration. It is fast because it doesn't use the actual DOG but a coarse version of it. The fine solve consists of solving each patch separately (like in ADMM). The coarse-to-fine and fine-to-coarse updates transfer information between the two resolutions.
How exactly the four steps work varies slightly between most of the tried variants. The coarse and fine solves work the same for the most part, plus or minus a few constraints. The coarse-to-fine and especially fine-to-coarse updates can differ very much. It is like that because for DOGs there is no straightforward way to coarsen/subdivide a mesh while fulfilling all necessary constraints.

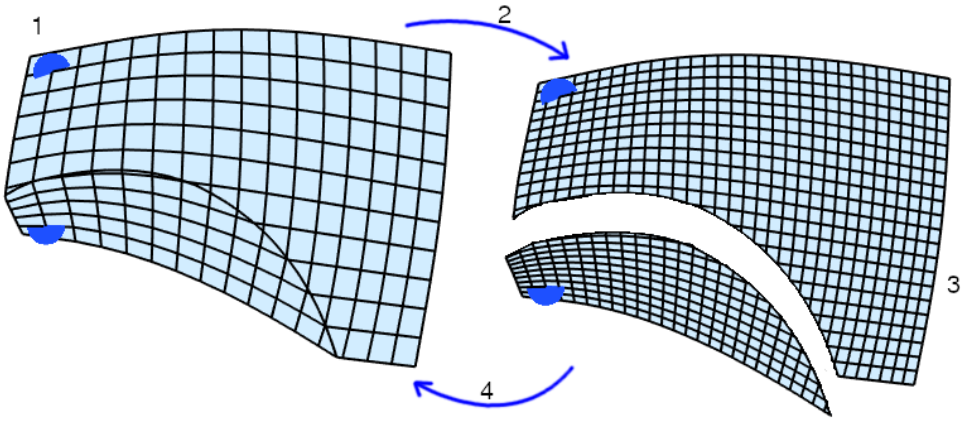

Figure 5.1: *The four steps of the coarse method: 1) Solve on coarse mesh 2) Coarse-to-fine update 3) Solve on fine patch submeshes 4) Fine-to-coarse update*

The coarse-to-fine update always involves the crease curves. The crease points from the coarse mesh get used as constraints in the appropriate fine patches. For some points on the fine curve this involves interpolating between coarse curve points

5.1 Coarse mesh

The coarse mesh gets constructed using the same method as the fine one, the only difference being its resolution, which is half the regular resolution in both x and y, rounded up. As the resolution corresponds to the number of vertices in each direction, this ensures that each coarse edge consists of two fine edges and each coarse quad covers four fine quads (when looking at a flat DOG).
Given the fine and coarse meshes, an auxiliary data structure that links the two gets constructed. This data structure holds all information necessary to switch between resolutions. Important parts include the fine-to-coarse and coarse-to-fine arrays. For each vertex in the fine and coarse mesh, they store the index of the vertex that has the same position in the other resolution. Given that the fine mesh is roughly four times the size of the coarse mesh, there are of course many vertices with no direct counterpart. For those, the arrays store a negative number which can further identify the type of the vertex in question.
Figure 5.2 shows an example of a small patch, its coarse version, and the associated helper arrays. The type of a vertex determines its colour:

- **Link** vertices are those that are part of both meshes. They are coloured green in the picture.
- **Fine-only** vertices appear only in the fine mesh. They can further be differentiated into
  - **I**-vertices (shown in light blue) which are adjacent to at least one link vertex
  - **X**-vertices (shown in purple) whose only direct neighbours are other fine-only vertices
- **Coarse-only** vertices (red in the picture) can only appear near the creases. As their name implies, they exist only on the coarse mesh.

5.2 FineCoarseConversion

This section provides a detailed description of the data structure linking the fine and coarse meshes. It is initially constructed after the meshes were created from the crease pattern and exploits the fact that all vertices lie in the xy-plane at that moment.
At the beginning, finding the link vertices is the most important step. The most basic way to identify the link vertices is to compare the coordinates of vertices in the fine mesh with those in the coarse mesh, but this is a brute force approach that is very expensive for large meshes.
Originally the method to find equivalent vertices in the fine and coarse meshes was a flooding algorithm. It started at one vertex and traversed the meshes in a BFS manner. This was quite

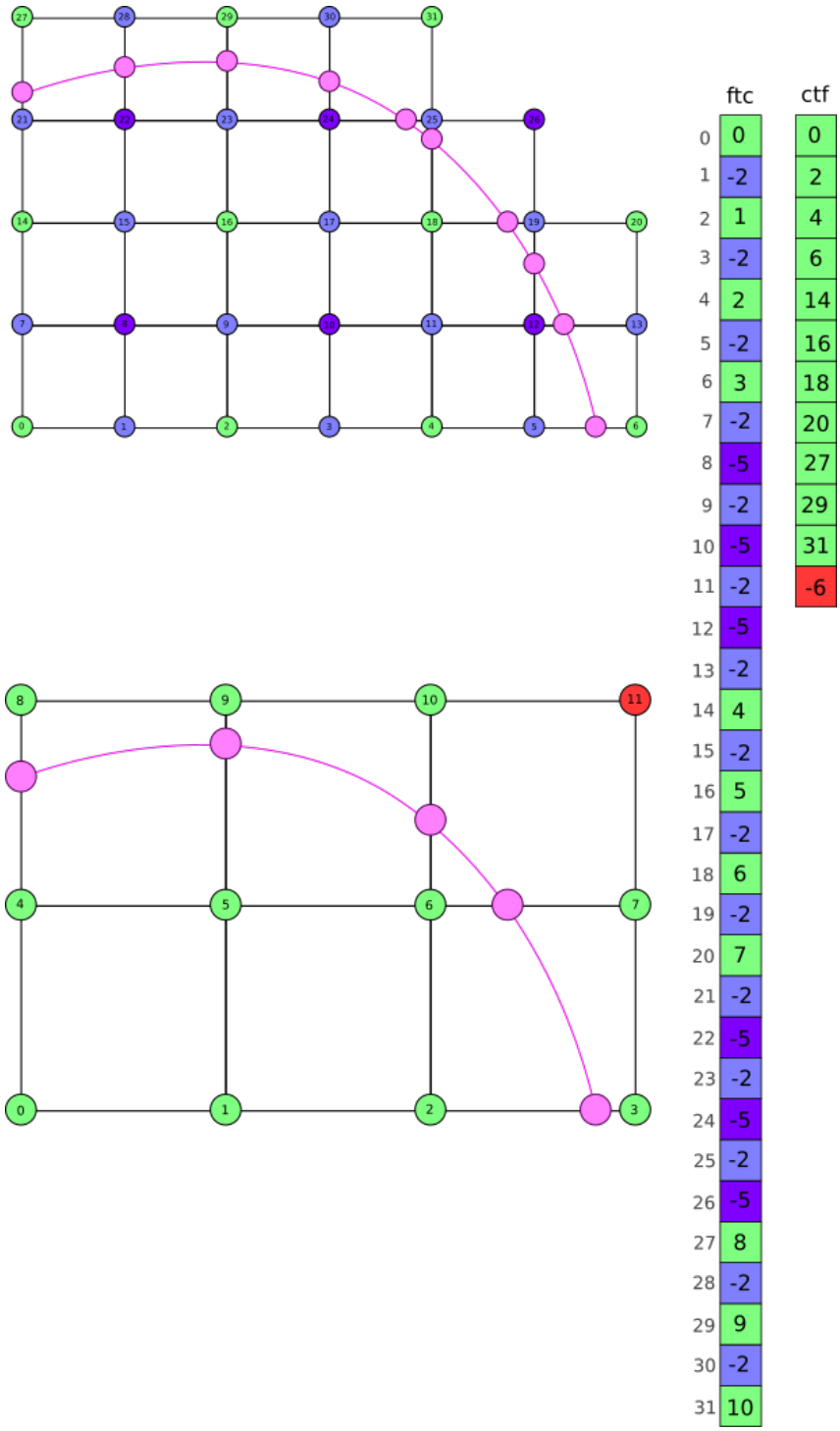

Figure 5.2: *Top: Fine patch; Bottom: Coarse patch; Right: Fine-to-coarse and coarse-to-fine arrays*

error-prone and required quite a bit extra work to find a good origin and cross from one patch to another.
The flooding method was replaced by another algorithm which treats each patch separately. Using the bounding box of all vertices within each patch and the distance between adjacent vertices it creates two rectangular grids per patch (one for the fine patch and one for the coarse patch). Each mesh vertex is assigned to one position in a grid. The advantage here is that it is very easy to go from fine to coarse and vice versa. As an example, the vertex at coarse grid coordinates $(1, 2)$ corresponds to the vertex at fine grid coordinates $(2, 4)$, the vertex at coarse coordinates $(1, 3)$ is linked to the vertex at fine coordinates $(2, 6)$, and the vertex at fine coordinates $(2, 5)$ is a fine-only I-vertex that lies between them.
In the end, this data structure holds all information that might be needed when changing between the fine and the coarse DOG. Most often it is used to convert indices of link vertices from one resolution to the other. It can also be used to determine the type of a vertex, map fine-only vertices to coarse edges or quads, and assign crease points in the two resolutions to each other. Mapping crease points to their coarse or fine equivalents works the same as for mesh vertices: For each curve there are two arrays, one storing the coarse-to-fine correspondences and one for fine-to-coarse. Initializing these does not require any grid: Traversing the coarse and fine curves while comparing coordinates is enough.

5.3 Constraints on the coarse mesh

The coarse mesh should provide good approximation of the fine solution. This means subjecting it to the same constraints as the fine mesh. For some constraints this is easier than for others.
The main DOG constraints of equal angles between edges around one vertex still apply, just using the topology of the coarse mesh now. Similarly, isometry and stitching constraints work like in the fine mesh but using other vertices.
The user-defined position and angle constraints can cause issues. If a constraint is defined on a fine-only vertex, then the coarse solver would need to approximate that constraint. This wasn't implemented, so all user-defined position constraints need to use link vertices.

As the angle constraints lie on edge points (i.e. between two mesh vertices) it is impossible to have all relevant vertices be link vertices. The constraint is thus modified a bit: Assuming one of the vertices involved is a link vertex, then the other is a fine-only I-vertex. In the coarse mesh, the I-vertex gets replaced by the next coarse vertex in that direction. Figure 5.3 shows an example of how the vertices used change in the coarse mesh. Finding the coarse vertices in the location of the dashed circles is easy if a mapping from I-vertices to coarse edges is precomputed and stored in the FineCoarseConversion.

The coarse angle constraints definitely change the results of the computation, but arguably it is an improvement. Angle constraints often result in shapes like in figure 5.4, where the angle constraint is technically satisfied at the crease point, but the mesh doesn't look smooth in that region. Taking a coarse angle constraint lessens this local irregularity.

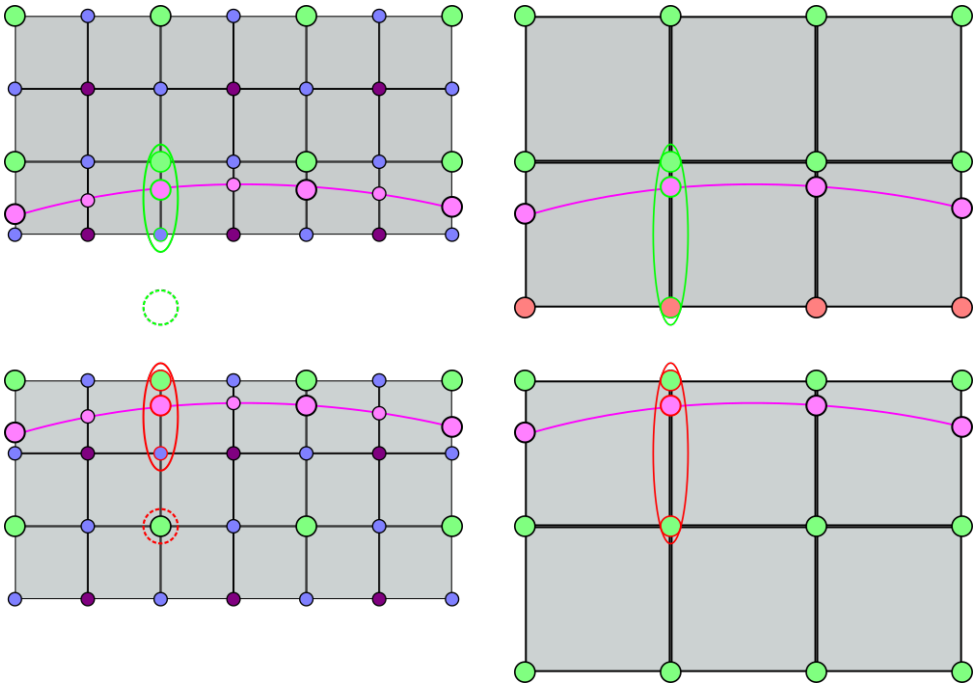

Figure 5.3: *Angle constraint between two patches: The mesh vertices involved in the angle constraint are circled in red/green (depending on the patch). The original resolution is on the left side, coarse is on the right side. Notice how the coarse version of the constraint uses different vertices*

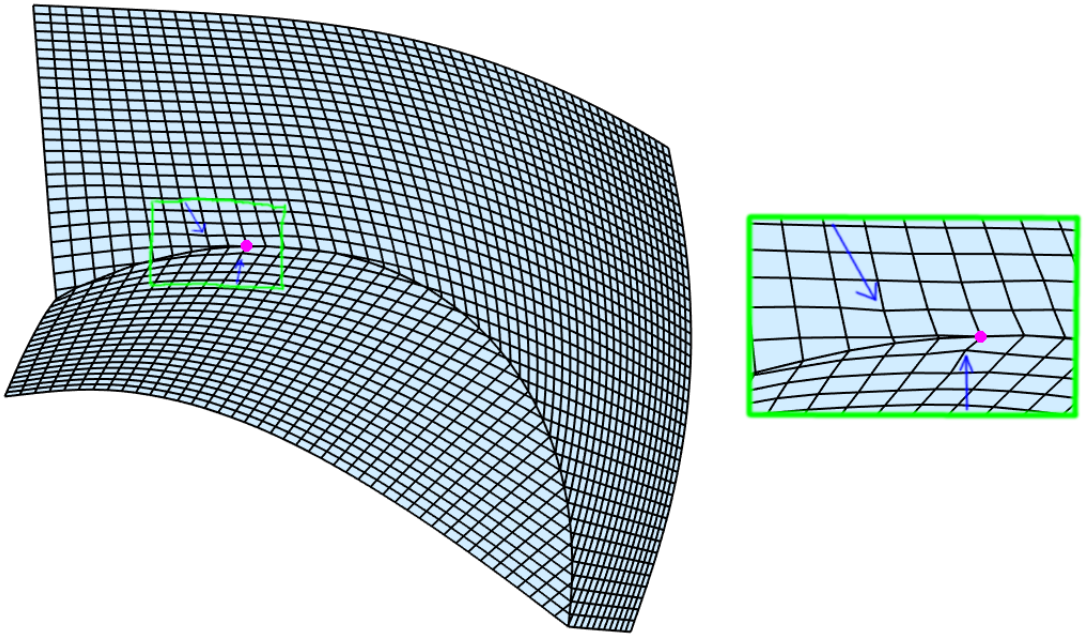

Figure 5.4: *Angle constraint at the pink marker. On the right is zoomed-in view of the green rectangle. Blue arrows point to vertices that lie outside the local plane.*

5.4 First coarse solver

The first serious variant of the coarse method starts with a global solver iteration on the DOG described in the previous section.
In the coarse-to-fine update, only the crease points are relevant. Interpolating between the coarse curve points results in coordinates for the fine curve points. These are used as edge point constraints and replace the stitching constraints on the patches.
The fine-to-coarse update is almost nonexistent in this variant. Assuming that any position constraints lie on link vertices, their target positions are equivalent in both coarse and fine.

5.5 Coarse-to-fine update

What the patches need from the coarse mesh are constraints for the stitching. Instead of stitching constraints (linking two edge points from different patches), the patches use edge point contraints (constraining edge points to fixed positions). The constrained edge points on the patches are the crease points and the weights are equal to half the stitching weight. This way, the edge point constraints on the patches correspond to the stitching contraints on the global DOG.
The positions for the local edge point constraints are taken from the crease curves of the coarse mesh. Not all fine crease points appear in the coarse DOG (compare the number of pink crease points in figure 5.2), so some of them have to be interpolated somehow. The interpolation should be smooth, so simple linear interpolation is not the way to go.
What's done is that the coarse curve gets stored in a piecewise manner. A list of discrete lengths l, curvatures k and torsions t are stored. The length is defined as the distance between two crease points, the curvature is computed using three adjacent points, and the torsion needs four points. Using length, curvature, torsion and a starting point (with orientation), it's possible to reconstruct a curve in 3D. For the interpolation, FineCoarseConversion stores an offset for each fine crease point, describing where it lies between the closest coarse crease points as a single value in $[0, 1]$. It is then possible to approximate positions for the fine crease points, using the new coarse crease points and the precomputed offsets. Given n fine crease points $\mathbf{c_0}, \mathbf{c_1}, ..., \mathbf{c_{n-1}}$, where the first and last points $\mathbf{c_0}$ and $\mathbf{c_{n-1}}$ are also coarse crease points, the offsets o_k for the points in between are computed as:

$$o_k = o_{k-1} + \frac{|\mathbf{c_k} - \mathbf{c_{k-1}}|}{\sum_{j=1}^{n-1} |\mathbf{c_j} - \mathbf{c_{j-1}}|} \tag{5.1}$$

With o_0 initialized as 0 (and o_{n-1} is always 1). This is not the only way of obtaining these offsets. One could also use a dot product:

$$o_k = \frac{(\mathbf{c_k} - \mathbf{c_0}) \cdot (\mathbf{c_{n-1}} - \mathbf{c_0})}{|\mathbf{c_{n-1}} - \mathbf{c_0}|} \tag{5.2}$$

The offset formula above might end up with values outside the interval $[0, 1]$, which is a reason not to use it. The best way to compute the offsets would probably involve optimizing the

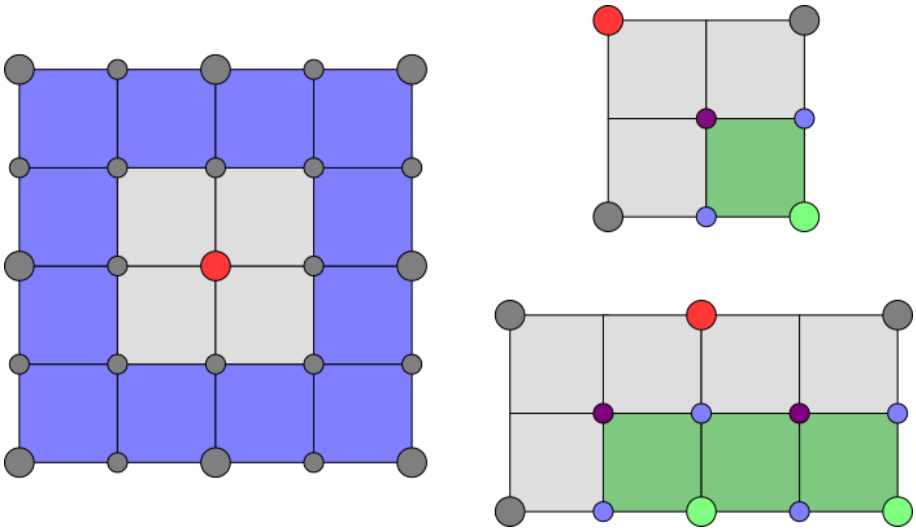

Figure 5.5: *Left: Potential fine quads near a coarse-only vertex; Top right: Minimum of one nearby quad; Bottom right: Realistic case with three nearby quads*

expected error in the approximation and could use more information than just the closest two coarse curve points. In practice the first variant (with the sum of fine lengths) worked well, so there was not much point in trying a different formula.

5.6 Fine-to-coarse update

Finding a good way to update the coarse mesh after the fine patches are done is quite difficult. The most straightforward approach would be to take the coordinates from the fine mesh and assign them to the appropriate vertices in the coarse mesh. This does not work for coarse-only vertices, so those need to be handled in another way.
The first coarse solver lacked a fine-to-coarse update and was not too bad, so it's not unthinkable that the fine-to-coarse update could just ignore the coarse-only vertices. That said, it is not a good idea to only update some parts of the coarse mesh while leaving others as they were.
The following two sections (5.7 and 5.8) concern major changes made to remove the coarse-only vertices. The remainder of this section is dedicated to variants that do try to update the coarse-only vertices while keeping the existing meshes.
To find a good update method for coarse-only vertices, some of their special qualities should be known and exploited. Coarse-only vertices appear only on the boundary of a patch, near creases. They are always outside the "real" patch (in the sense that they are not visible in the rendering), on the other side of a curve compared to the majority of all vertices in the patch. The idea is to use information from assorted close-by fine vertices to compute coordinates for each coarse-only vertex.
Each coarse-only vertex can potentially have any of the blue quads shown in figure 5.5 nearby, but only one of them is guaranteed to exist. The four quads in the middle are never present (because if any of them were, the central vertex wouldn't be coarse-only). Realistically, there will never be a full ring around a coarse-only vertex and testing showed that the number of fine

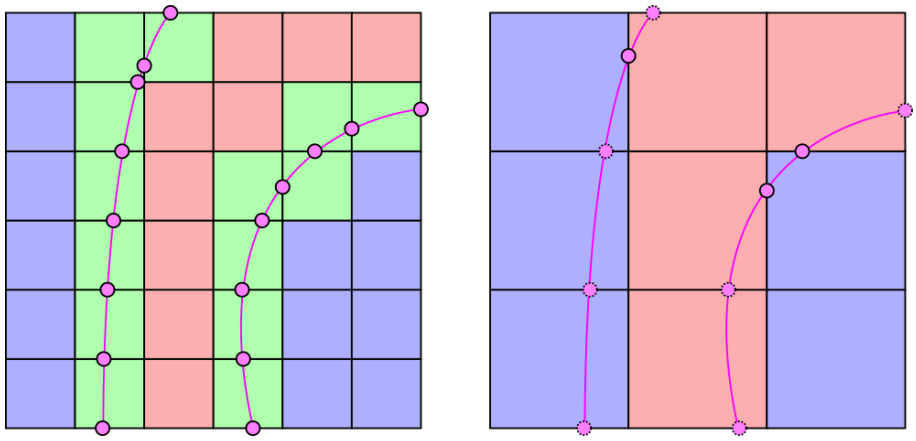

Figure 5.6: *Alternative coarse coarse construction: Right side is the coarse version of the left side. Red and blue quads correspond to different patches, green quads are duplicated.*

quads near a coarse-only vertex is usually between 1 and 6.
Multiple methods using the nearby fine quads to extrapolate coordinates for the coarse-only vertex were implemented and tested, but none of them were particularly good and most had serious stability problems. As it happens, the whole plan was fundamentally flawed: The fine result may satisfy all necessary constraints on the fine DOG, but directly inserting some of those coordinates into the coarse mesh will generally not satisfy the coarse DOG constraints.
Instead of directly setting the coordinates to be equivalent, adding another set of soft constraints turned out to be a solution of sorts. Each link vertex in both resolutions gets an extra position constraint moving it to its equivalent in the other resolution. These link constraints have a much lower weight than actual position constraints, so they can keep the fine and coarse meshes in similar shapes, but not to the detriment of any important constraints.

5.7 Alternative coarse mesh construction

To avoid having to update coarse-only vertices, one variant used a different strategy to obtain the coarse mesh. Instead of building it roughly the same way as the regular DOG, the coarse DOG is constructed by removing vertices from the fine mesh. By removing every second row and column of vertices and connecting the remaining vertices correctly, we get a coarser mesh where every vertex is a link vertex.

Figure 5.6 shows a small DOG and its coarse version as obtained with the alternative construction. Note how there are no duplicated quads in the coarse mesh. The new coarse mesh no longer covers all fine patches. The most problematic part is what happens to the crease curves in the coarse mesh. Only a limited number of crease points are still in the coarse mesh (in figure 5.6 they are marked by a solid black outline) and, crucially, the endpoints of the curve (i.e. where the crease curve intersects the mesh boundary) are not among them. This makes it impossible to interpolate coordinates for the crease points near the boundary.

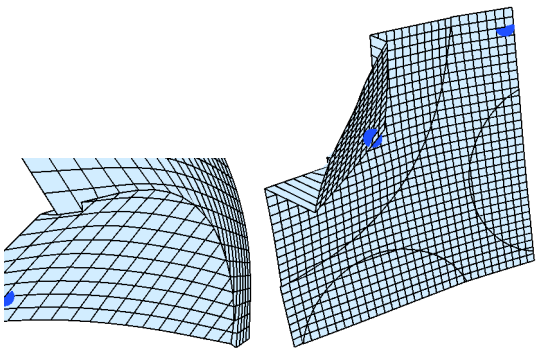

Figure 5.7: *Result when using the alternative coarse construction without any further constraints. The region near the curve end is problematic, especially if said region is larger like in the right mesh.*

5.7.1 Results

With the alternative coarse mesh construction, the endpoints of all crease curves and all crease points between the endpoints and the closest coarse crease point can't be interpolated from the coarse mesh. If nothing is done with those points, the results look like in figure 5.7. There is a clearly visible disconnect between the two patches at the boundary.

It's possible to reconnect the boundary parts of the crease using edge point constraints. This leads back to the same problem that ADMM had: The crease points near the boundary assume that the curve shape is constant even though it isn't, and so those parts only change very slowly. By altering the associated weights at runtime, it's possible to get fast changes initially and connected patches later. It's still usually not worth it because one of the two problems almost always remains and the reweighting process would have to be automated for general user inputs.

5.8 Curve submeshes

The idea behind this variant came upon realizing something: Most problems arise near the crease curves. Anything related to stitching, angle constrints, coarse-only vertices will only happen around a curve and never on the interior of a patch. The following idea came up: Create a new type of mesh that specifically handles these areas, where various problems had been showing up.

As an addition to the entire coarse mesh and the fine patch submeshes, we introduce curve submeshes. These are fine meshes containing only the areas near a crease curve. This means that they contain vertices from multiple patches while still being very small (and thus fast to compute solutions). The curve submeshes function as interfaces between the coarse mesh and the fine patches. To that purpose, the curve submeshes contain fine versions of all coarse quads through which a curve passes. This includes fine vertices corresponding to the previously coarse-only-vertices. An example of the full set of meshes is displayed in figure 5.8. Note that all elements near the curve in both the coarse mesh and the curve submeshes are duplicated. They are also duplicated in the fine mesh, but the patches are already separated in the picture and so all dupli-

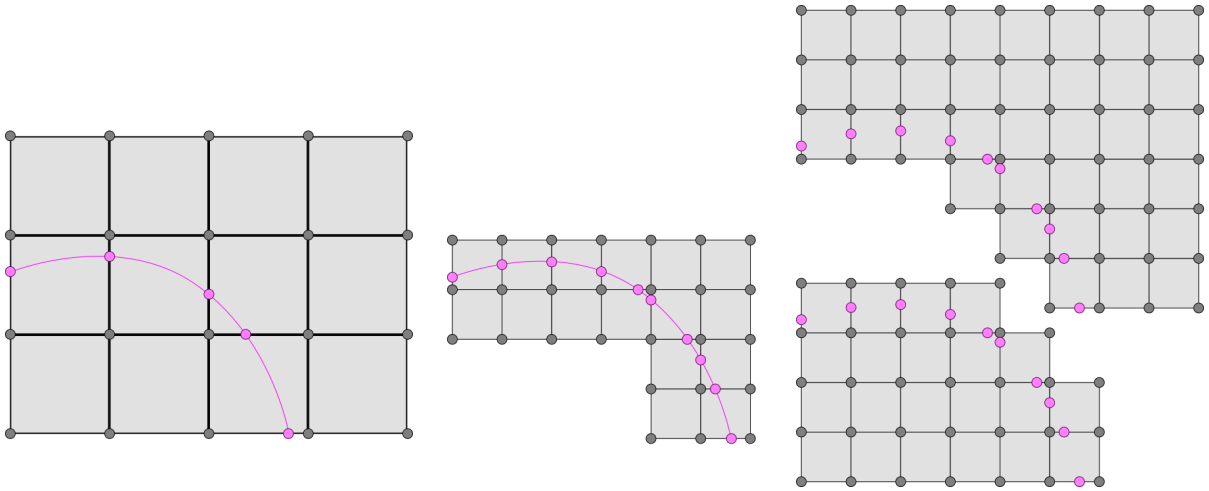

Figure 5.8: *From left to right: Coarse mesh, curve submesh, patch submeshes*

cate elements are visible there.
A single iteration with curve submeshes works as follows: Initially, the coarse solver produces guesses for certain points of the curves. The curve submeshes use those as constraints and compute the position of all crease points, while also ensuring that angle constraints are satisfied (remember, those are among the constraints that work across patch borders). The patches then gather the shape of their borders from the curve submeshes and work on minimizing their local objective functions. The curve submeshes also contain the necessary parts to propagate data back from fine to coarse.

5.8.1 Results

The method using curve submeshes works reasonably well: The patches are connected at the creases. There are no bizarre deformations that break all constraints. Everything is fast enough to edit the constraints interactively. There are still two problems.
As mentioned in section 5.6, updating the coarse vertices with coordinates from fine vertices is not a good idea. The curve submeshes may have fine versions of the coarse-only vertices, but the fine-to-coarse update still needs another method.
Second, the results of the curve submesh solver are almost indistuingishable from those obtained by just using the coarse mesh and the fine patches. As there is no real point to introducing curve submeshes (and therefore more points of failure) if they don't actually improve the result, the curve submeshes were scrapped.

6 Regarding parallelization

The PARDISO solver which is used to actually compute the solution is supposed to be parallelized already. Unfortunately, running the program with a higher number of threads slows down the iterations. The reason for that is not clear: It might be that our systems are too small to profit from the multithreaded version of PARDISO. This is supported by the fact that the slowdown is less extreme for larger meshes. Another possibility that shouldn't be ruled out completely is that there is some semi-hidden parameter setting in the PARDISO solver which isn't being set correctly.
Actually parallelizing the solver is not the goal of this project but finding alternative solver modes that can be parallelized is. The ADMM methods allow every patch to be optimized on its own, which would make it possible to perform almost all computations in parallel. Even the steps where certain variables get updated at the start and end of the iteration could be parallelized to some degree. As such, ADMM methods looked like a great way to make the code more scalable.
The coarse method still includes the local solvers that can be run in parallel, but it also has the coarse solve. The coarse mesh contains slightly more than $\frac{1}{4}$ times the vertices of the global mesh. Assuming the time per iteration scales linearly with the number of vertices, this would mean that the absolute maximum speedup when using a parallelized coarse method is 4, but considering that solving the patches still takes some time, this is not realistic. If the workload of solving the patches can be perfectly split across four cores, then cutting the iteration time down to $50 - 60\%$ of the original time seems reasonable.
Using the methods described above, certain meshes profit more from parallelization than others. There's not much point in using four threads for a crease pattern that has only two patches, one of them being several times the size of the other. The best meshes here are those where the patches can be split between threads in such a way that each thread has a similar amount of work. This works better if there are many patches. The best case would have the number of patches be a multiple of the number of threads, with patches of roughly equal size.

7 Results

7.1 Test setup

The bulk of data presented in the following sections is gathered by performing the same operations on DOGs using different crease patterns, resolutions, and solver modes. The crease patterns are shown in figure 7.1. "1_curve" is a very simple pattern with just one crease. "4_corners" is also rather simple but includes more than two patches. "valley" and "parallel_curves_4" are more complicated and tend to be more problematic.

The resolutions used are tabulated in table 7.1. The number of vertices is not just the product of x-resolution and y-resolution because the areas near creases are duplicated. Note how all resolutions are uneven: This ensures that the coarse DOG can be constructed correctly. Even resolutions would lead to an odd number of edges/quads in one direction, which makes it impossible to link the fine and coarse meshes correctly.

The following solver modes are used: Standard (optimization on the global regular mesh), Coarse with no fine-to-coarse update (as described in chapter 5), Coarse with fine-to-coarse update (with varying weights for the soft constraints on the link vertices). Figure 7.3 shows the line colours used for these methods in the later plots.

Most other methods (like ADMM) were deemed unlikely to be an improvement over the standard solver and will not be considered here. The variant using curve submeshes is the only exception, because it works better than other methods. The reason it doesn't show up here is that the code for the curve submeshes is incompatible with the other solver modes.

Once the DOG is ready, we add positional constraints at two corners (marked by blue dots in figure 7.1). Over 250 iterations the constraints are moved linearly towards a target position. Data is also gathered for 50 iterations after that to see how the solver behaves once the constraints are fixed.

7.2 Test results

Figure 7.2 shows what happens with the objective functions. As can be seen, the standard and coarse solvers tend to have qualitatively similar results but the objective function always obtains lower values with the standard solver. That is because the standard solver optimizes the objective function directly while the coarse solver tries to do so using a different way. Interestingly, the coarse solver sometimes performs better without the fine-to-coarse update. It's still usually better to do the fine-to-coarse update (notice how the no-fine-to-coarse-update curve looks in the largest "1_curve" setting in the upper right corner).

The resulting meshes usually look very similar, whether the standard solver or the coarse

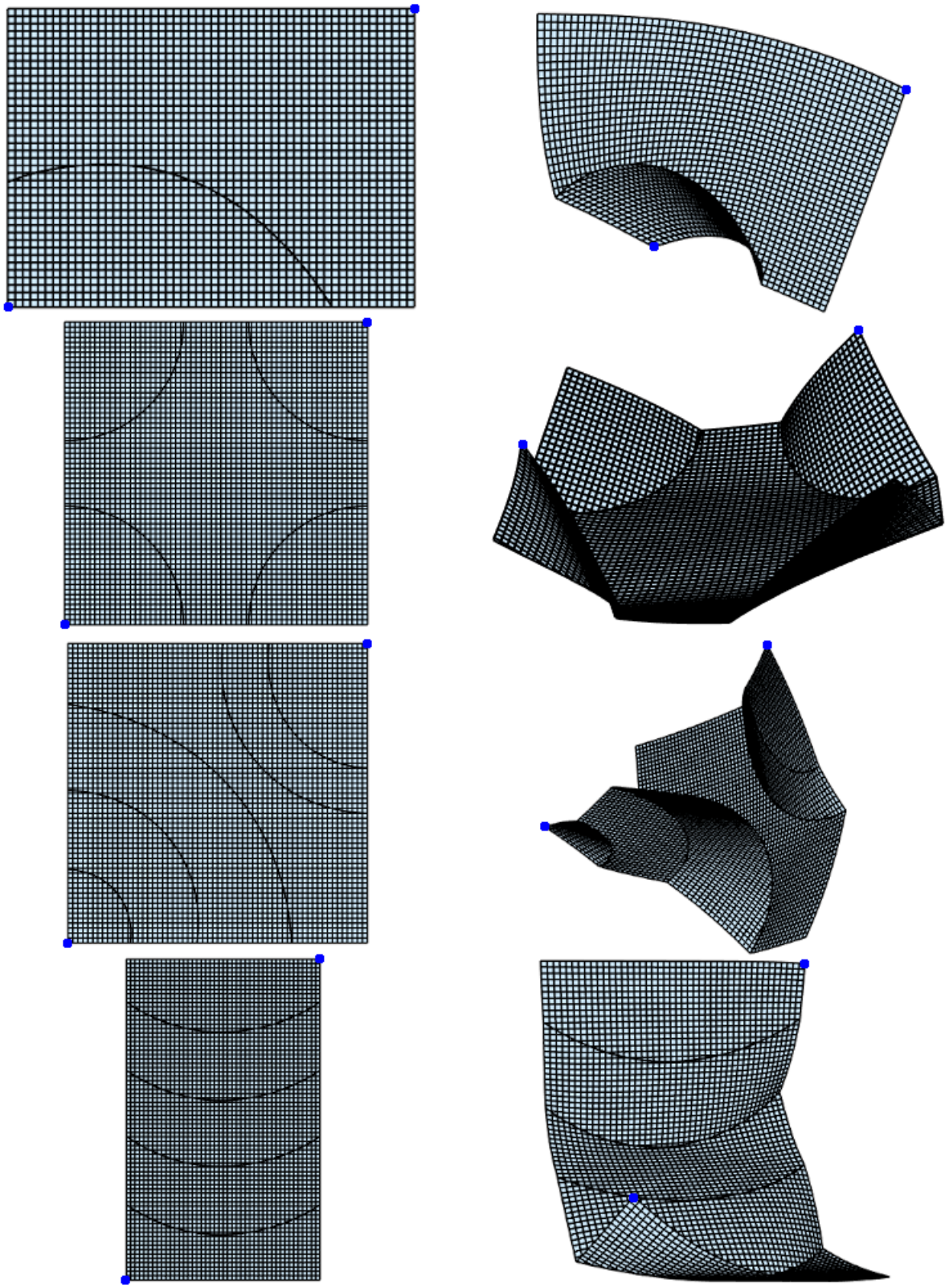

Figure 7.1: *Crease patterns: Flat initial state on the left, after bending on the right. From top to bottom: 1_curve, 4_corners, valley, parallel_curves_4*

crease pattern	resolution 1	resolution 2	resolution 3	resolution 4
	# vertices	# vertices	# vertices	# vertices
1_curve	31×21	43×31	55×41	81×59
	721	1435	2389	4969
4_corners	21×21	41×41	61×61	81×81
	569	1937	4105	7073
valley	21×21	41×41	61×61	81×81
	639	2069	4301	7327
parallel_curves_4	27×45	39×65	45×75	51×85
	1498	2943	3847	4867

Table 7.1: *Mesh sizes for the test cases in figure 7.2*

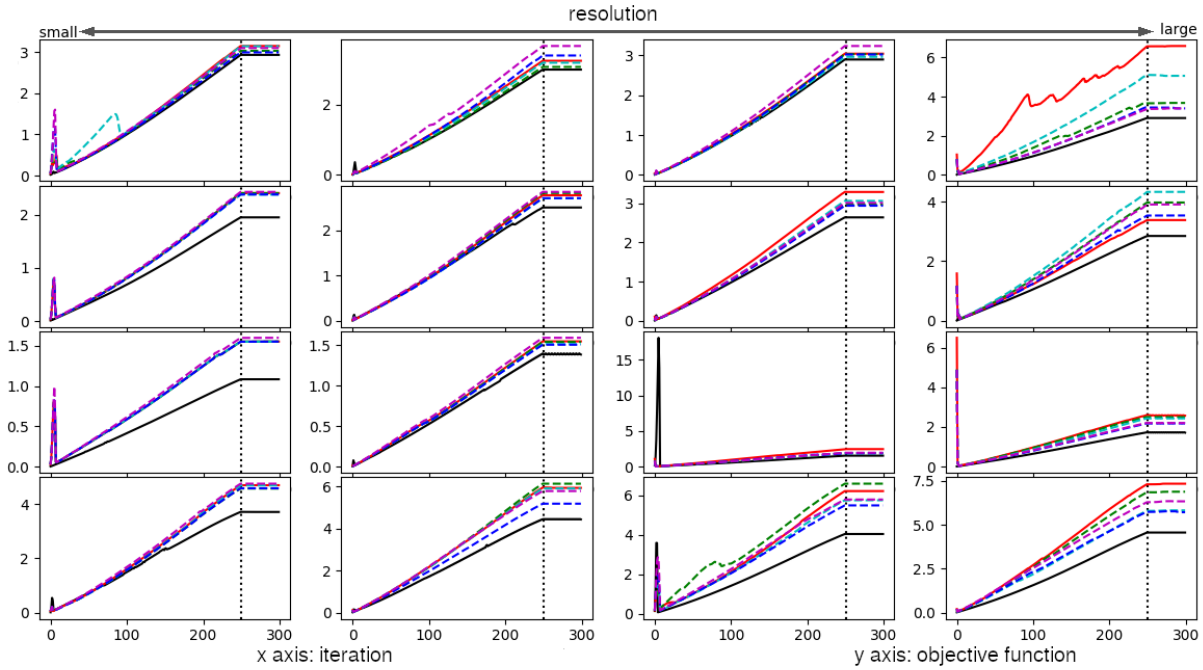

Figure 7.2: *Objective function plotted over 300 iterations for multiple crease patterns, resolutions, and solver modes. Starting from a flat initial state, the constraints reach their final state after 250 iterations (marked by a dotted vertical line). Plots correspond to the crease patterns and resolutions in the same location in table 7.1*

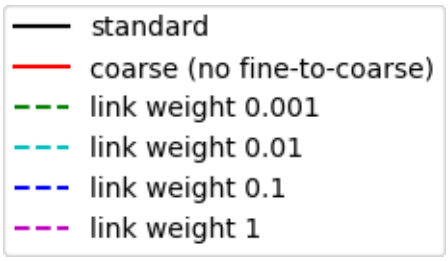

Figure 7.3: *Legend for figure 7.2*

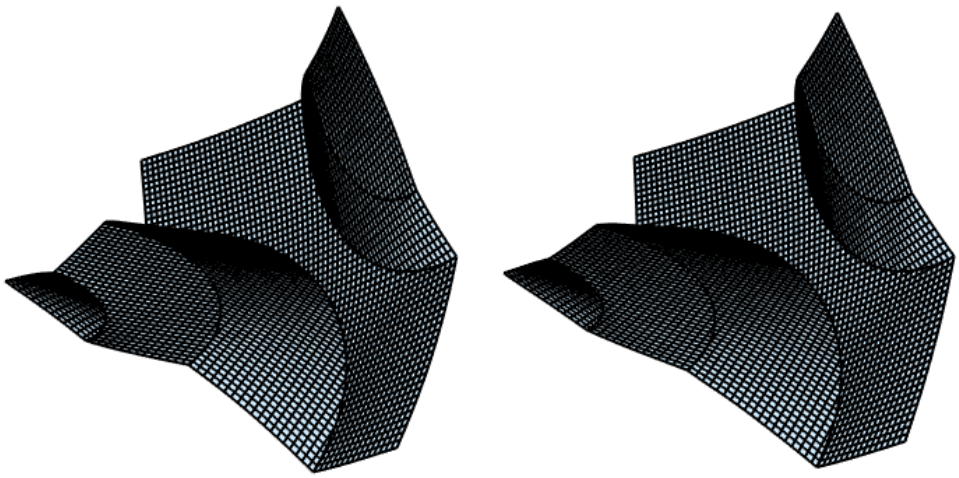

Figure 7.4: *Comparison between final state for crease pattern "valley" using standard solver (left) or coarse solver (right)*

solver is used (compare figure 7.4). There are exceptions however: The crease pattern "parallel_curves_4" (seen in figure 7.5) has quite different results: In the coarse solver the mesh ends up looking like a half-circle from the side, but in the standard solver all adjacent patches have very different orientations.
The coarse solver also has visibly different results for small meshes (i.e. the leftmost row). This happens because the coarse mesh still needs a certain size to work properly. In other words, making the resolution too low results in a coarse mesh that is too coarse to be useful.

7.3 Timing

The time per iteration, plotted over 300 iterations, usually looks like the left example in figure 7.6. The first iteration often takes more time than the later ones, then the time per iteration is roughly the same during the phase where the constraints are moving. Once the constraints arrive at their final positions, the solver usually finds a solution very quickly, after which there is nothing happening and the time per iteration is greatly decreased.
The right side of figure 7.6 shows a more problematic example. About every second iteration in that case takes more time than the one before and after it. This happens because sometimes the solver needs to go back and redo the iteration with different parameters to enforce the folding

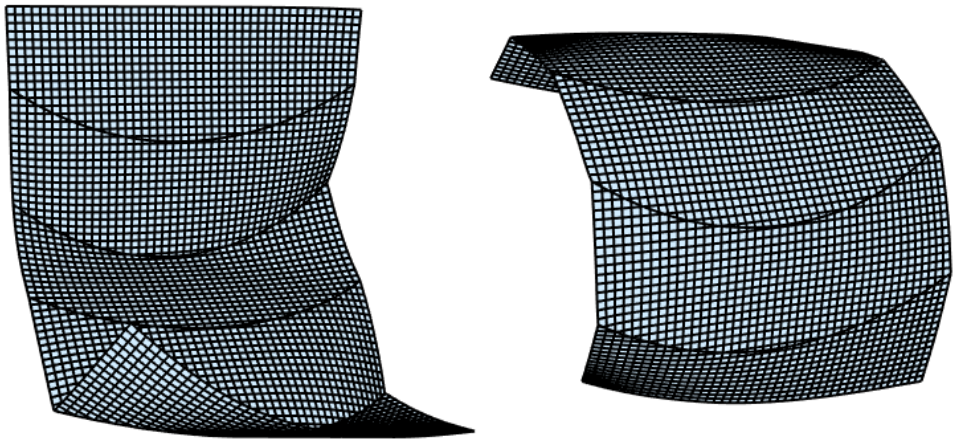

Figure 7.5: *Comparison between final state for crease pattern "parallel_curves_4" using standard solver (left) or coarse solver (right)*

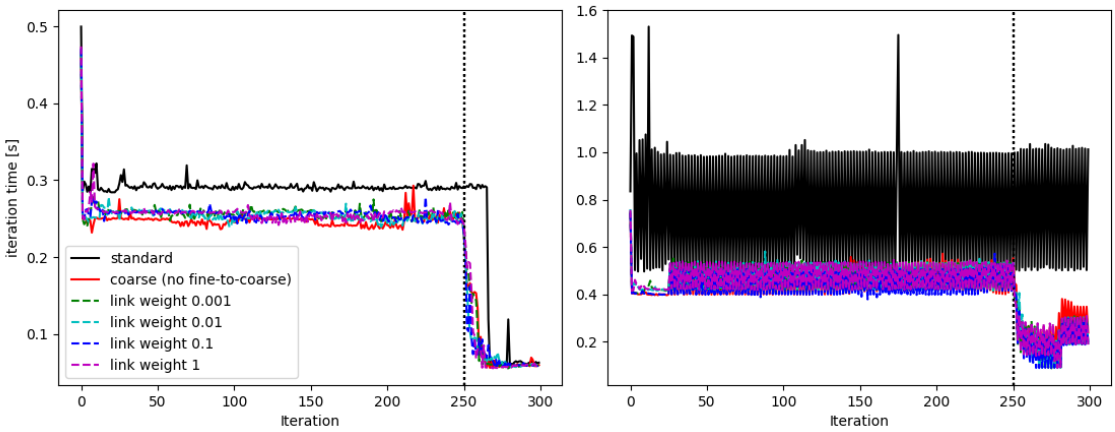

Figure 7.6: *Time needed for iterations* 1 *to* 300*. Crease pattern "4_corners" with resolution* 41×41 *on the left, crease pattern "parallel_curves_4" with resolution* 39×65 *on the right*

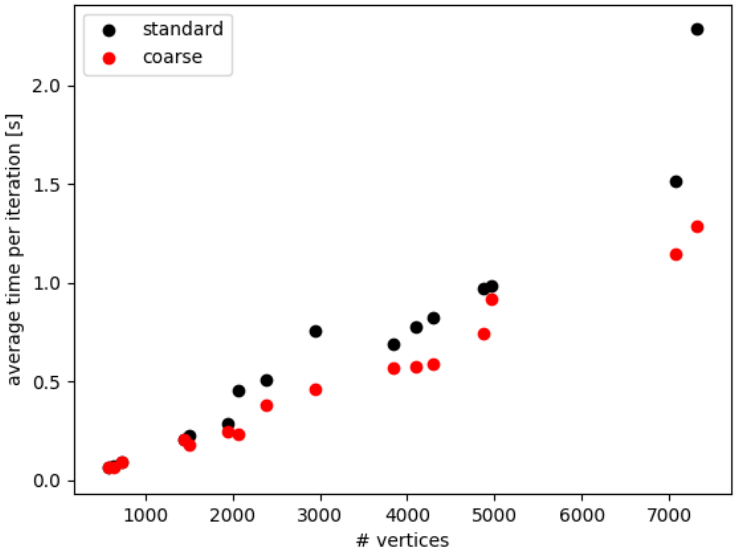

Figure 7.7: *Time per iteration, taken as the averaged in iterations* 10-240

constraints mentioned in section 3.2 (notice how the time the time for a standard iteration is almost exactly 0.5, 1 or 1.5 seconds). It can also happen on the coarse mesh, but the difference in iteration time is less extreme there due to two factors: The coarse mesh is a smaller system than the global mesh, and the solver iterations on the fine patches also make up a major part of the work.
Another thing that can be seen in the plot on the right is that sometimes the iterations don't become faster at the end. In fact, the time for later iterations can even be higher than for early iterations. This seems to happen only with the standard solver and not when using the coarse method.
Following this, an average "time per iteration" is computed using iterations 10 to 240 and compared between solver mode and the number of vertices in figure 7.7. The time per iteration seems to scale linearly with the number of vertices and the coarse solver is consistently faster than standard for the same mesh size.

7.4 Problem cases

As seen in section 7.2, particularly figure 7.5, the crease pattern "parallel_curves_4" gets quite different results in the coarse solver. There are two main reasons to this: First, the folding constraints are enforced less strictly than in the standard solver. They can't be enforced on the local level, but the coarse mesh should provide a reasonable base. The second problem with that particular crease pattern is that the creases are relatively straight. In the coarse mesh their shape is even less distinct. This results in a lot of leeway in terms of possible folding deformations of the coarse mesh, and the final choice may be optimal for the coarse mesh but not the fine one.
The coarse solver assumes that each crease neatly separates two patches. This is not the case

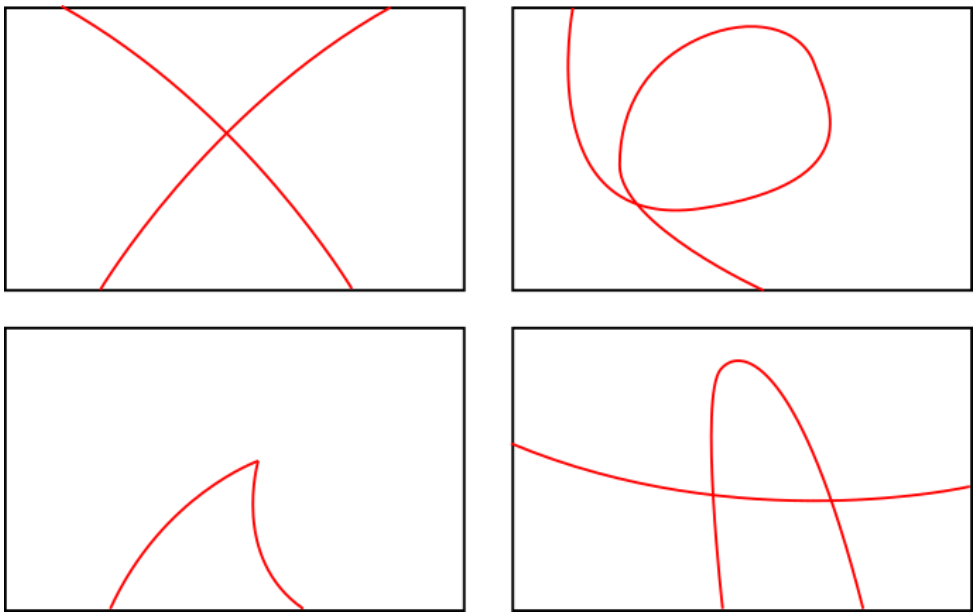

Figure 7.8: *Examples of crease patterns that can currently not be used*

for many crease patterns, like most of those in figure 7.8. Crease curves that intersect each other (upper left and bottom right) or themselves (upper right) violate this assumption. A relatively simple way to make intersecting curves work with the current system would be to replace each curve by one or more curve segments. These curve segments should not intersect and always separate exactly two patches (contrast the upper right pattern in figure 7.8 where a single curve creates three patches).
The lower left crease pattern in figure 7.8 shows another problem case in the shape of a non-smooth crease. This is particularly bad for the coarse method because the coarse-to-fine update involves interpolating points on the curve and the current interpolation method fails around sharp turns.
All examples in figure 7.8 are also problematic for the standard solver. The editor tends crash. It often happens during the mesh construction, in which case the coarse mesh can be at fault. Even if everything seems to work, the results tend to either stay flat or deform horribly and violate all constraints.

8 Conclusion

Of all the methods tried to make the solver faster and more amenable to parallelization, the one using a coarse solution as a guess appears to perform best.
The ADMM-based methods were extremely slow and not stable in addition to that. Working on each patch mostly independently would be great when parallelizing, but the results showed that there needs to be something to compute the shapes of the creases in 3D. Local solutions from all patches are insufficient.
The coarse method, using the curves from a coarse solution as constraints when solving on the patches, has the best results among all tested methods. It's main advantages compared to the standard solver lies in being faster when using large meshes. It still includes local solver iterations on each patch, meaning that part of the algorithm is relatively easy to parallelize.
The coarse method does have certain disadvantages: It requires a certain resolution to work properly, so the standard solver remains useful for small meshes. The results are obiously different from what comes out of the standard solver, but usually they don't seem wrong, just a bit suboptimal.
It is currently not possible to switch between different solver modes at runtime. The standard solver does not update the coarse mesh and getting coarse vertex coordinates from the fine mesh is very difficult (as discussed in section 5.6). Switching from coarse to standard usually fails because the folding constraints are not satisfied. The coarse solver does enforce the folding constraints on the coarse mesh, but the patches break them at the regular resolution.
In terms of methods that could still become useful, the curve submeshes seem most promising. The cycle of coarse $\rightarrow$ curve $\rightarrow$ fine $\rightarrow$ curve $\rightarrow$ coarse would have to be re-examined, as it's possible that certain steps should be replaced, combined, or even removed. The curve-to-fine update is straightforward but for the others it's possible that there are better constraints to keep everything consistent.
Anderson acceleration (as seen in [Peng et al. 2018] or the more ADMM-related [Zhang et al. 2019]) was considered at one point. It promises fast convergence for nonlinear and nonconvex optimization problems using local-global solvers, so it should be applicable here. There was an attempt to include Anderson acceleration here, but there were problems with the stability (which may have been due to an incorrect implementation).
The most obvious follow-up work would be to parallelize the solver. PARDISO is actually supposed to work with multiple threads already, but for some unclear reasons it only increases the time for solving the system in this case, so the program is currently completely serial.
Beyond that, the coarse method can still be extended in different ways. On one hand, it might be a good idea to introduce more levels than just "fine" and "coarse". Going in a slightly different direction, creating a fully adaptive multigrid solver that uses the higher resolutions in difficult regions (e.g. near patch boundaries) and coarse parts for the simpler areas (e.g. in the middle of a large patch) would be great for working with more problematic crease patterns (see section 7.4). Either way, it would probably be a good idea to think about the fine-coarse exchange some

more: Finding a way to refine or coarsen parts of the mesh while satisfying the DOG constraints would open the door to various existing multigrid algorithms.

Bibliography

De Coninck, A., De Baets, B., Kourounis, D., Verbosio, F., Schenk, O., Maenhout, S., and Fostier, J. 2016. Needles: Toward large-scale genomic prediction with marker-by-environment interaction. 543–555.

Deng, W., Lai, M.-J., Peng, Z., and Yin, W. 2017. Parallel multi-block admm with o(1 / k) convergence. *Journal of Scientific Computing 71*, 712–736.

Kourounis, D., Fuchs, A., and Schenk, O. 2018. Towards the next generation of multiperiod optimal power flow solvers. *IEEE Transactions on Power Systems PP*, 99, 1–10.

Peng, Y., Deng, B., Zhang, J., Geng, F., Qin, W., and Liu, L. 2018. Anderson acceleration for geometry optimization and physics simulation. *ACM Transactions on Graphics 37*, 4 (Jul), 1–14.

Rabinovich, M., Hoffmann, T., and Sorkine-Hornung, O. 2018. Discrete geodesic nets for modeling developable surfaces. *ACM Transactions on Graphics 37*, 2.

Rabinovich, M., Hoffmann, T., and Sorkine-Hornung, O. 2018. The shape space of discrete orthogonal geodesic nets. *ACM Transactions on Graphics (proceedings of ACM SIGGRAPH ASIA) 37*, 6.

Rabinovich, M., Hoffmann, T., and Sorkine-Hornung, O. 2019. Modeling curved folding with freeform deformations. *ACM Transactions on Graphics (proceedings of ACM SIGGRAPH ASIA) 38*, 6.

Verbosio, F., Coninck, A. D., Kourounis, D., and Schenk, O. 2017. Enhancing the scalability of selected inversion factorization algorithms in genomic prediction. *Journal of Computational Science 22*, Supplement C, 99 – 108.

Zhang, J., Peng, Y., Ouyang, W., and Deng, B. 2019. Accelerating admm for efficient simulation and optimization. *ACM Transactions on Graphics (TOG) 38*, 6 (Nov), 1âĂŞ21.