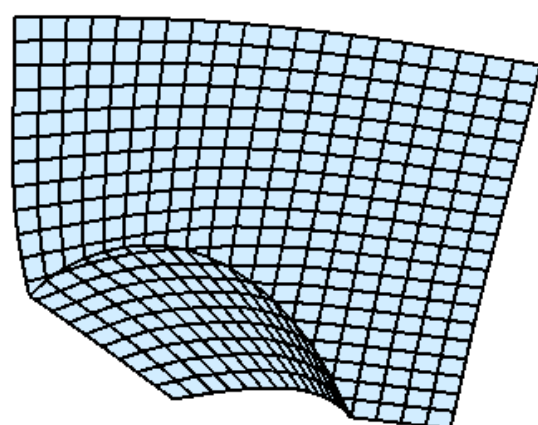


# **Scalable DOGs**

Linus Wigger

March 9, 2020







# **1 Abstract**



## 2 Introduction

Developable surfaces, defined as smooth surfaces with zero Gaussian curvature, are locally isometric to a plane. They are of interest in areas like architecture or manufacturing because they can be obtained by transforming a plane via bending (but not stretching). As such, simulating developable surfaces is a TODO topic in computer graphics.

This project uses the work from [Rabinovich et al. 2018a], [Rabinovich et al. 2018b], and [Rabinovich et al. 2018c] as a starting point. These papers introduce the idea of using Discrete Orthogonal Geodesic nets (abbreviated as DOGs) to simulate deformations of developable surfaces. This approach has certain advantages compared to previous methods, mainly in terms of flexibility. They do not depend on predefined rulings, making it possible to explore the full shape space.

[Rabinovich et al. 2018a] made an editor where the user can interactively deform a DOG. Said editor supports creased folds ([Rabinovich et al. 2019]) and various types of 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 with many constraints. The mesh is initially constructed from a crease pattern - a two-dimensional set of curves defining the shape of the surface and its folds. Consequently, the set of crease curves can't be altered later (beyond changing the positions of the curve points).

Figure 3.1 shows an example of a crease pattern and the resulting mesh. The areas and vertices coloured in green are 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).

#### 3.1 Objective function

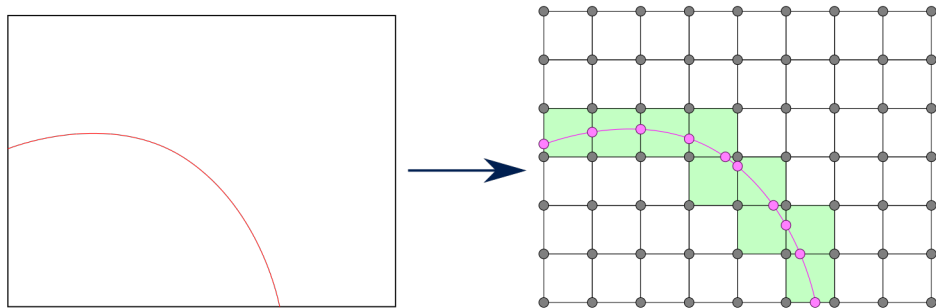
The goal of the optimization is to minimize the bending energy of the DOG while satisfying all constraints. Most of the constraints are soft, meaning they are part of the objective function. The only constraints that are strictly enforced are the DOG constraints.

$$f_{obj}(\mathbf{x}) = w_{bend} \cdot E_{bending}(\mathbf{x}) + \sum_{i \in \text{soft constraints}} w_i \cdot E_i(\mathbf{x}) \quad (3.1)$$

$\mathbf{x}$  in this case is a vector of size  $3|V|$ , containing the coordinates of all mesh vertices.

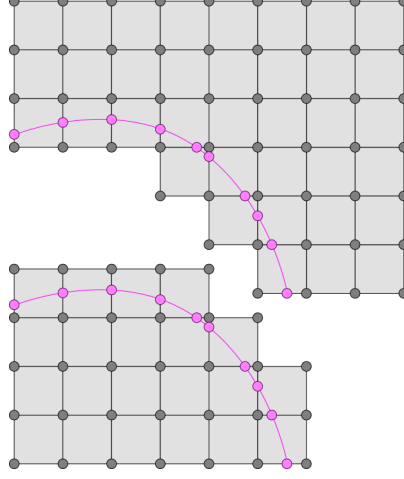
The bending energy is computed by comparing the Laplacian  $L$  of the current configuration to the Laplacian of the flat state

$$E_{bending}(\mathbf{x}) = |L(\mathbf{x}_0) - L(\mathbf{x})|^2 \quad (3.2)$$



**Figure 3.1:** Left: A crease pattern with one fold; Right: DOG obtained from the crease pattern with resolution 9,7

### 3 Problem setting



**Figure 3.2:** Patches when separated from each other

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 correct squared length  $l_{0i}^2$  is added to the objective. 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.

$$E_{isometry}(\mathbf{x}) = \sum_{i=1}^{n_{edges}} (l_i^2(\mathbf{x}) - l_{0i}^2)^2 \quad (3.3)$$

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:

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

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:

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

$$\mathbf{e}_i = t_i \mathbf{v}_{ai} + (1 - t_i) \mathbf{v}_{bi} \quad (3.6)$$

Where  $R$  is the number of edge point constraints and the  $\mathbf{r}_i$  are the target positions. 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.

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

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  $\mathbf{c}_{ai}$  and  $\mathbf{c}_{bi}$ , 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 angle between the edges involved is not actually the same as the folding angle (which uses the surface tangents). This means that the correct angle  $\alpha_i$  between edges from different patches needs to be computed when the constraint is created. 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:

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

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 (TODO src) solver. Each iteration starts by updating all constraints. Following that, PARDISO computes a better solution for the current constraints.



## 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 [TODO citation]. This method can be applied to a problem of the form

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

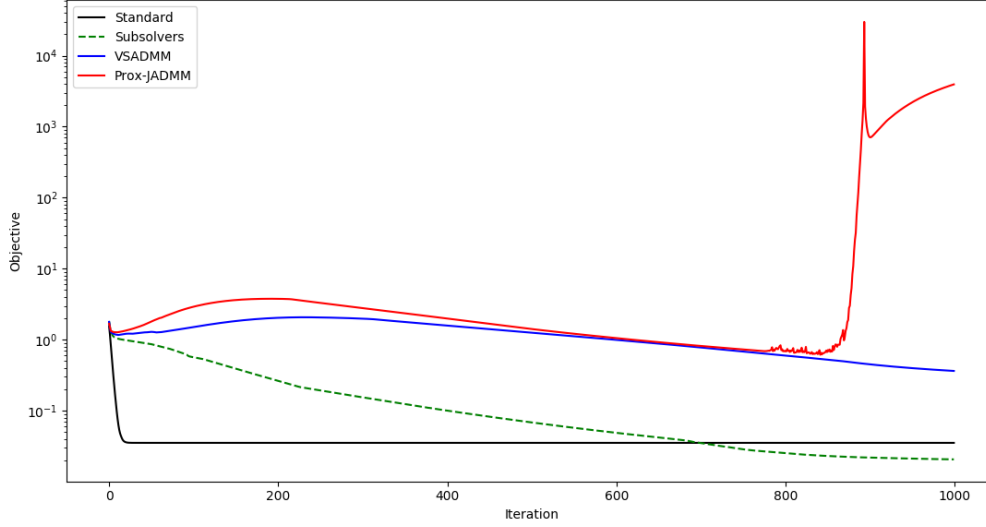
In this case, the  $\mathbf{x}_i$  are the coordinate vectors for each patch and the stitching constraints match the second line.

### 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 (TODO see algorithm 1 in src) works similar to Subsolvers but adds a few extra elements: The local objective functions are extended by an additional term  $\frac{\rho}{2} |A_i x_i - z_i^k - \frac{\lambda_i^k}{\rho}|^2$ .  $\lambda_i$  and  $z_i$  are other extra variables that are updated in each iteration. 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  $\lambda_i$  get updated to  $\lambda_i - \rho(A_i x_i - z_i + \frac{\lambda_i}{\rho})$



**Figure 4.1:** Objective function values for flattening a simple DOG

### 4.3 Variant: Proximal Jacobian ADMM

Proximal Jacobian ADMM (TODO see algorithm 4 in src) adds another extra term to VSADMM and changes the way  $z_i$  and  $\lambda$  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  $\lambda$  (which is constant across patches) becomes  $\lambda - \rho\gamma \sum_{i=1}^N A_i x_i$  ( $\gamma$  being a damping parameter  $> 0$ ).

### 4.4 ADMM results

The ADMM methods are not good for this problem. All of them are extremely slow at moving away from the initial state. Figure 4.1 plots the objective function for a 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 other 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 op-

timization, this is a very bad assumption. Additionally, ADMM as described in TODOsource assumes that the problem is convex, which it is not (due to all the constraints).





## 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:

- 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.

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 5.1 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

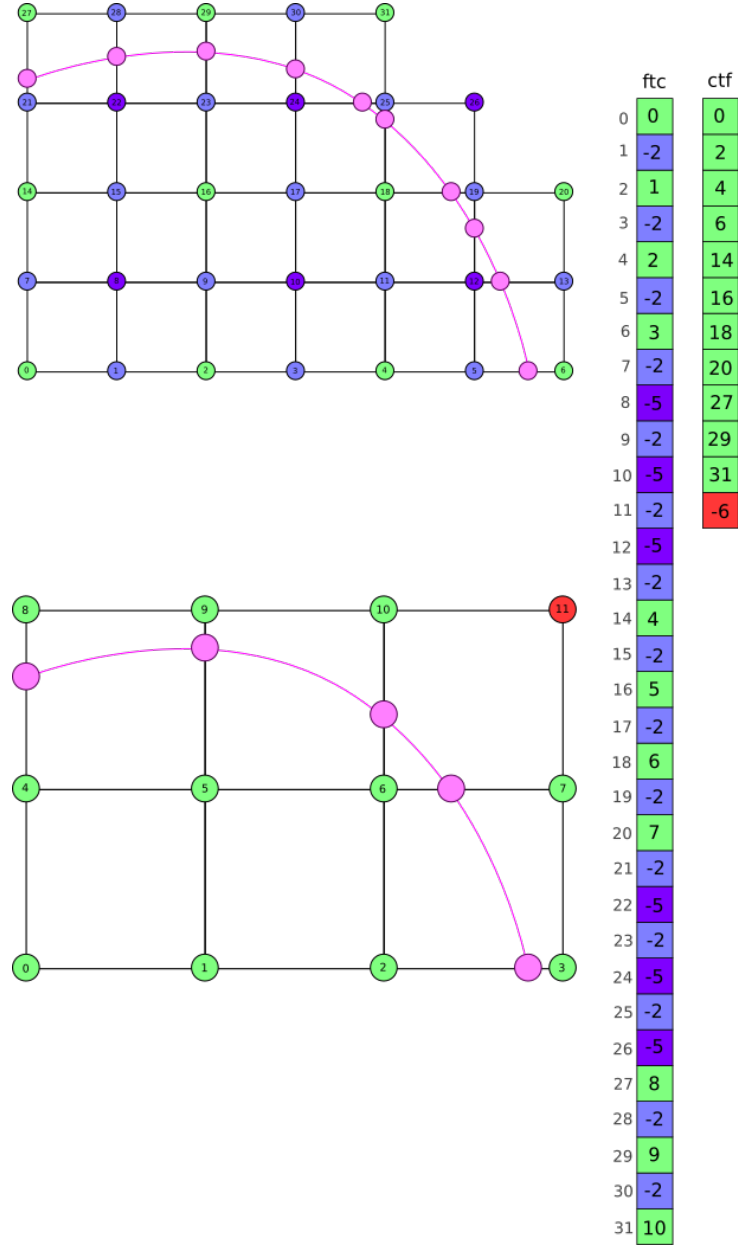
This section provides a detailed description of the data structure linking the fine and coarse meshes. It is initially constructed after the meshes were created from the crease pattern and exploits the fact that all vertices lie in the  $xy$ -plane at that moment.

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.

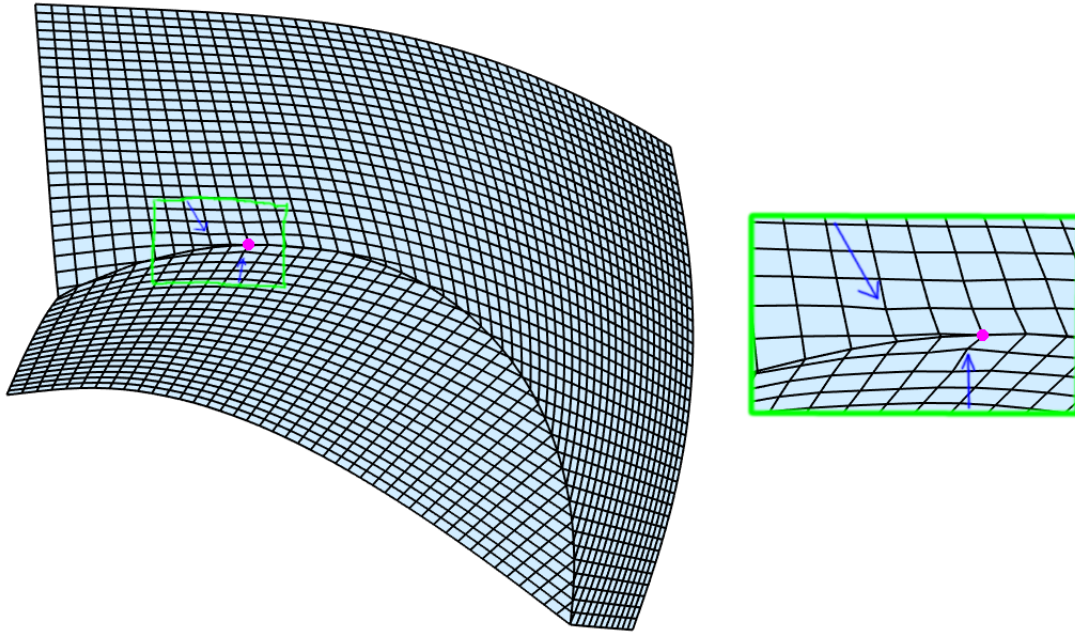
Originally the method to find equivalent vertices in the fine and coarse meshes was a flooding algorithm. It started at one vertex and traversed the meshes in a BFS manner. This was quite error-prone and required quite a bit extra work to find a good origin and cross from one patch to another.

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.



**Figure 5.1:** Top: Fine patch; Bottom: Coarse patch; Right: Fine-to-coarse and coarse-to-fine arrays



**Figure 5.2:** 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.

## 5.3 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. The coarse vertex in question can be found quickly if a mapping from I-vertices to coarse edges is precomputed.

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 5.2, 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.

## 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 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.

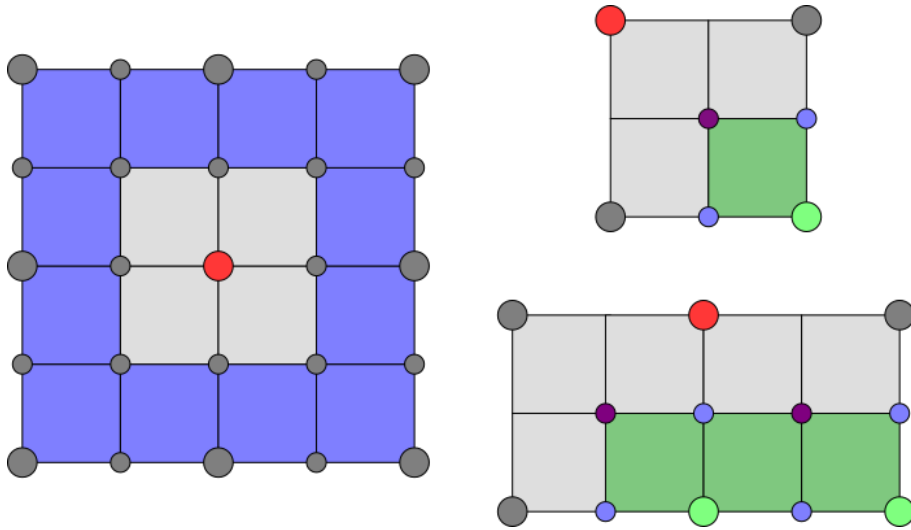
The following two sections (5.6 and 5.7) concern major changes made to remove the coarse-only vertices. The remainder of this section is dedicated to variants that do try to update the coarse-only vertices while keeping the existing meshes.

Finding a good way to update coarse-only vertices requires knowing some facts about them. Coarse-only vertices appear only on the boundary of a patch, near creases. They are always outside the "real" patch, 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 5.3 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 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.



**Figure 5.3:** Left: Potential fine quads near a coarse-only vertex; Top right: Minimum of one nearby quad; Bottom right: Realistic case with three nearby quads

## 5.6 Alternative coarse mesh construction

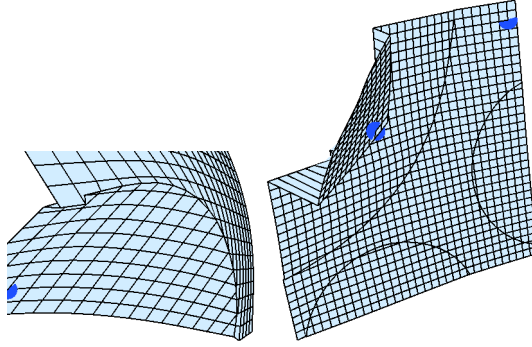
To avoid having to update coarse-only vertices, one variant used a different strategy to obtain the coarse mesh. Instead of building it roughly the same way as the regular DOG, the coarse DOG is constructed by removing vertices from the fine mesh. By removing every second row and column of vertices and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

The resulting 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 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.

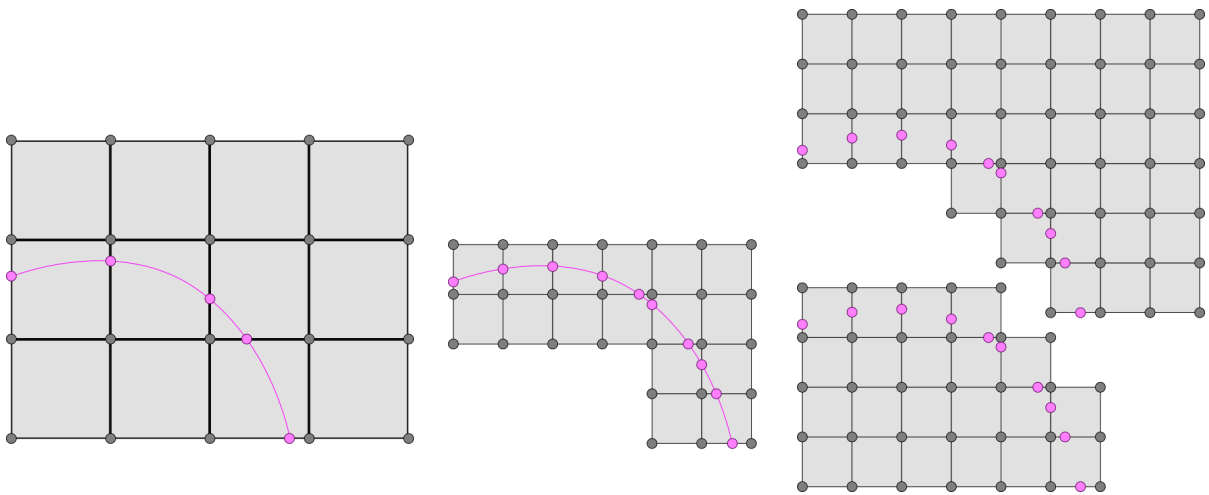
### 5.6.1 Results

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 5.4. 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.



**Figure 5.4:** 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.



**Figure 5.5:** From left to right: Coarse mesh, curve submesh, patch submeshes

## 5.7 Curve submeshes

This variant was "invented" after realizing something: Most problems arise near the crease curves. Anything related to stitching, angle constraints, coarse-only vertices will only happen around a curve. The following idea came up: Create a new type of mesh that specifically handles these problematic areas.

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 5.5. 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 duplicate elements are visible there.

A single iteration with curve submeshes works as follows: Initially, the coarse solver produces

## 5 Coarse method

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.

### 5.7.1 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 5.5, 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 indistinguishable 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 failure) if they don't actually improve the result, the curve submeshes were scrapped.



# 6 Results

## 6.1 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 6.1. "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 6.1. 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.

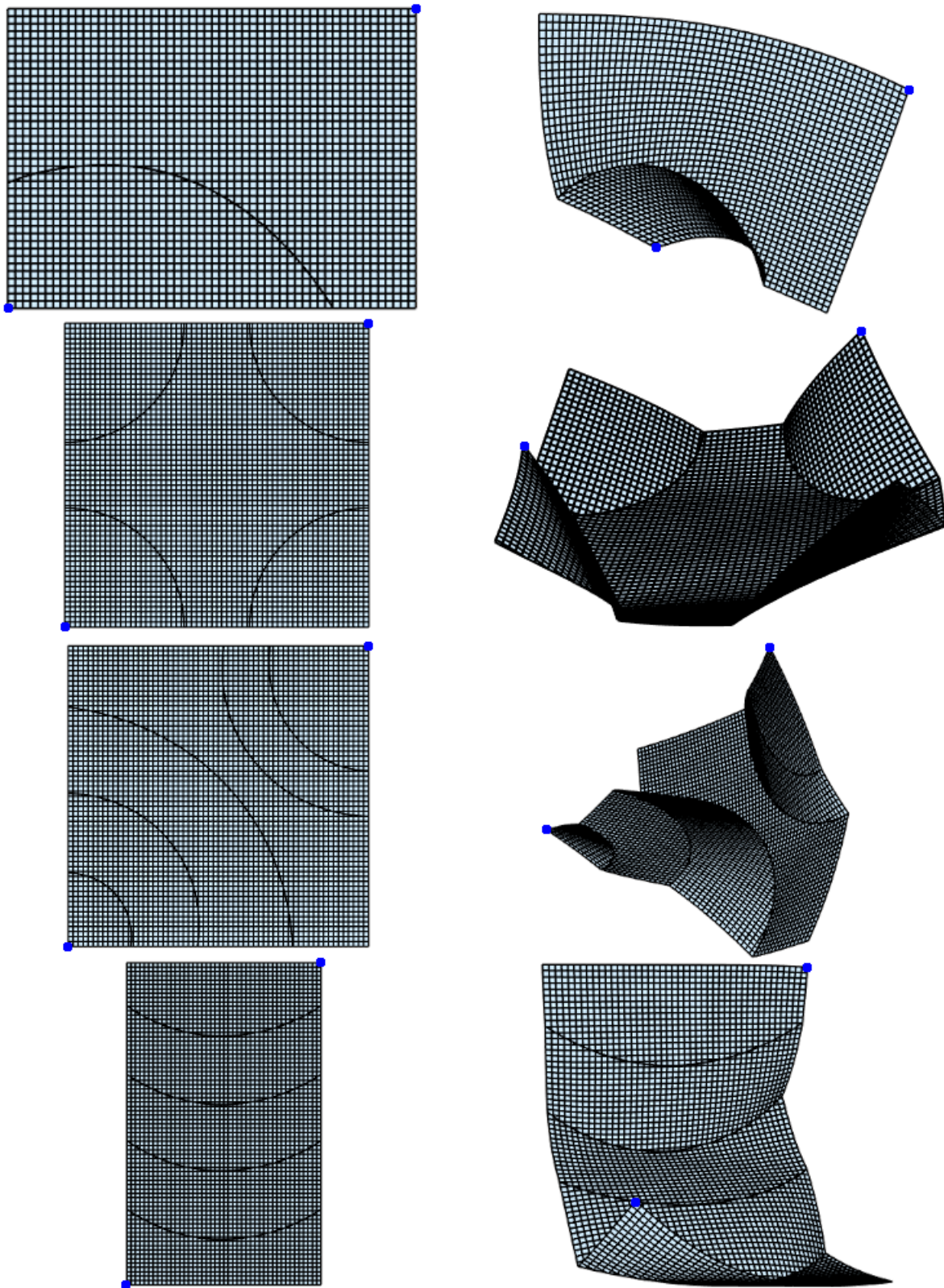
The following solver modes are used: Standard (optimization on the global regular mesh), Coarse with no fine-to-coarse update (as described in chapter 5), Coarse with fine-to-coarse update (with varying weights for the soft constraints on the link vertices). Other methods (like ADMM) were deemed unlikely to be an improvement over the standard solver and will not be considered here.

Once the DOG is ready, we add positional constraints at two corners (marked by blue dots in figure 6.1). 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.

## 6.2 Test results

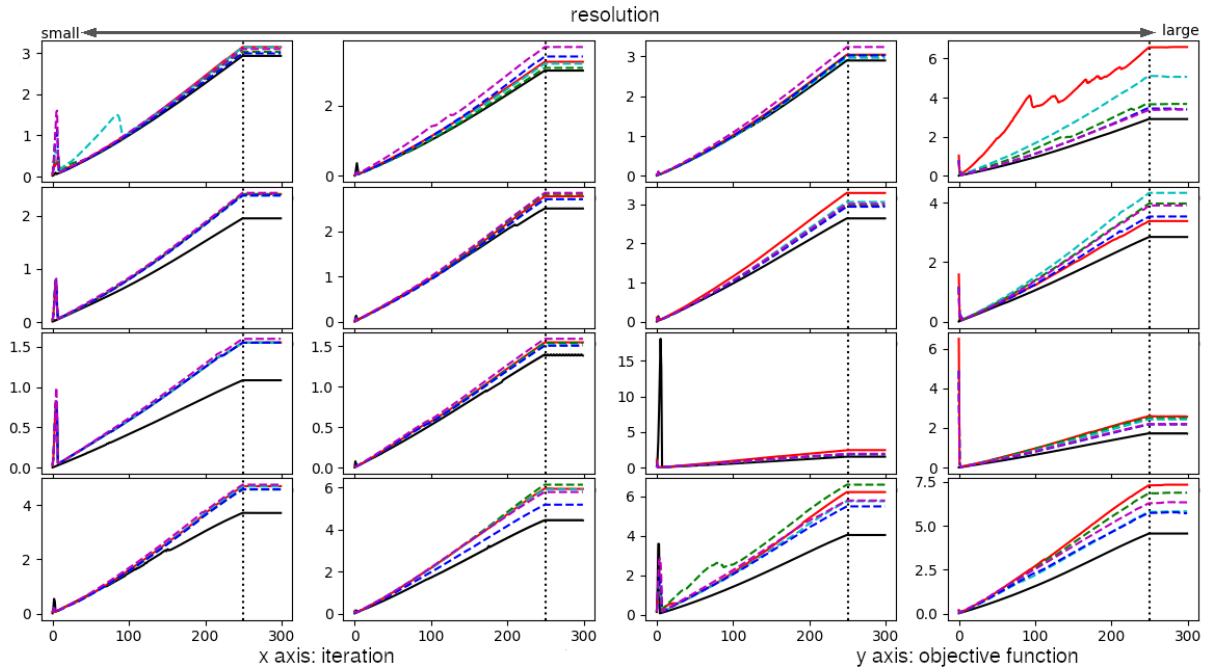
Figure 6.2 shows what happens with the objective functions. As can be seen, the standard and coarse solvers tend to have qualitatively similar results but the objective function always obtains lower values with the standard solver. That is because the standard solver optimizes the objective function 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 solver is used (compare figure 6.4). There are exceptions however: The crease pattern "parallel\_curves\_4" (seen in figure 6.5) 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



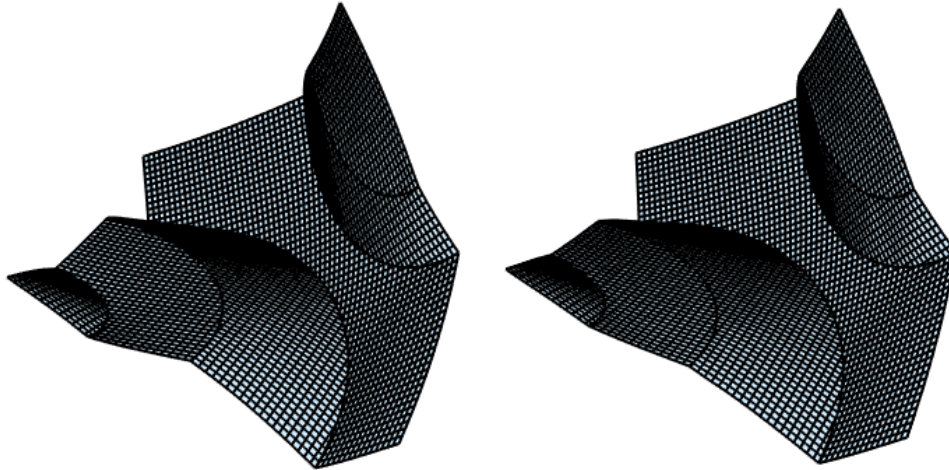
**Figure 6.1:** Crease patterns: Flat initial state on the left, after bending on the right. From top to bottom: 1\_curve, 4\_corners, valley, parallel\_curves\_4

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

**Table 6.1:** Mesh sizes for the test cases in figure 6.2**Figure 6.2:** 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 6.1



**Figure 6.3:** Legend for figure 6.2



**Figure 6.4:** Comparison between final state for crease pattern "valley" using standard solver (left) or coarse solver (right)

have very different orientations.

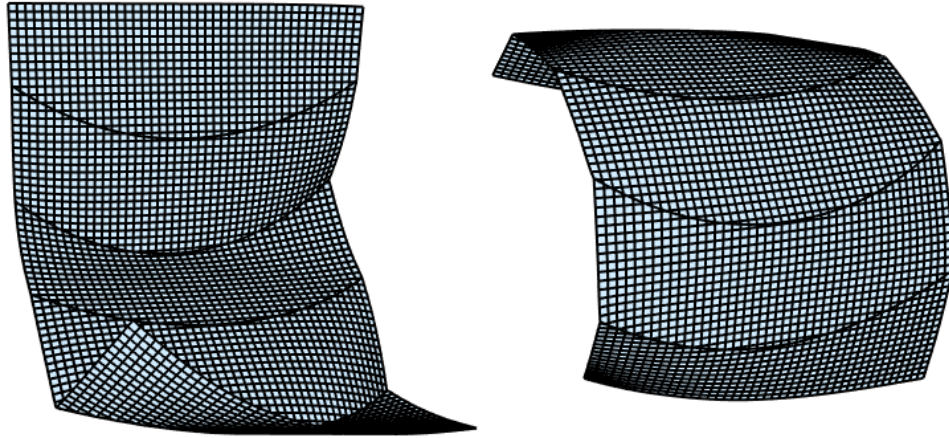
The coarse solver also has visibly different results for small meshes (i.e. the leftmost row). This happens because the coarse mesh still needs a certain size to work properly. In other words, making the resolution too low results in a coarse mesh that is too coarse to be useful.

## 6.3 Timing

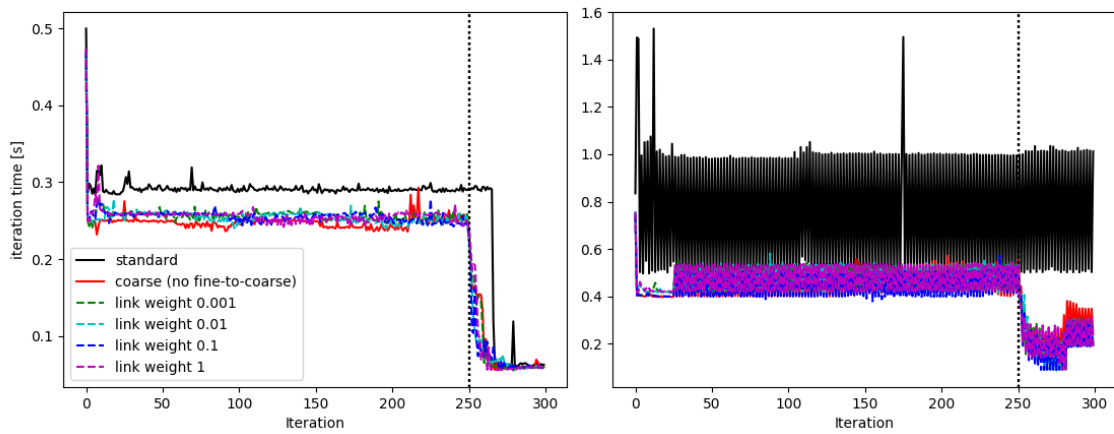
The time per iteration, plotted over 300 iterations, usually looks like the left example in figure 6.6. The first iteration often takes more time than the later ones, then the time per iteration is roughly the same during the phase where the constraints are moving. Once the constraints arrive at their 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 6.6 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 certain constraints (notice how the time for a standard iteration is either 0.5 or 1 second). 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

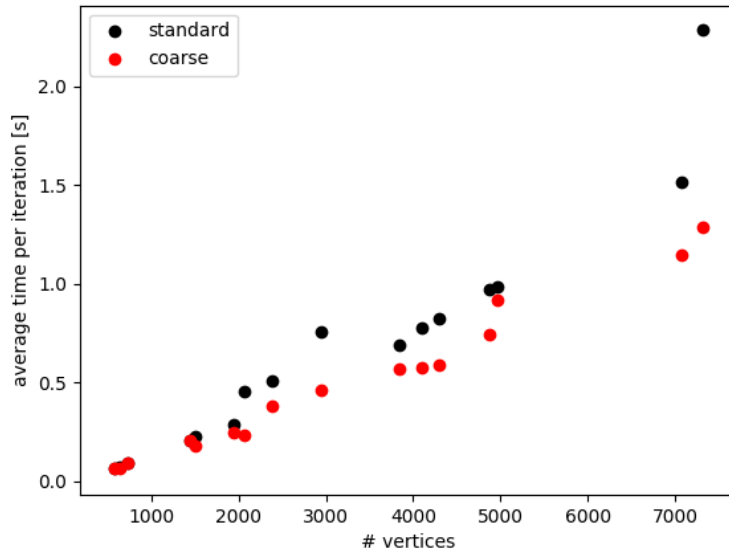




**Figure 6.5:** Comparison between final state for crease pattern "parallel\_curves\_4" using standard solver (left) or coarse solver (right)



**Figure 6.6:** Time needed for iterations 1 to 300. Crease pattern "4\_corners" with resolution  $41 \times 41$  on the left, crease pattern "parallel\_curves\_4" with resolution  $39 \times 65$  on the right



**Figure 6.7:** Time per iteration, taken as the averaged in iterations 10-240

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 6.7. 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 Conclusion

worth it? meh [?][?]

## Bibliography

- 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.