Cellogram: On the Fly Traction Force Microscopy

Tobias Lendenmann[1,2,ç], Teseo Schneider[2,ç], Jérémie Dumas[2,3], Marco Tarini[4], Costanza Giampietro[7], Apratim Bajpai[5], Weiqiang Chen[5], Julia Gerber[1], Dimos Poulikakos[1], Aldo Ferrari[1,6,7], Daniele Panozzo[2]**

1 ETH Zurich, Laboratory of Thermodynamics in Emerging Technologies, 8092 Zurich, Switzerland

2 New York University, Courant Institute of Mathematical Sciences, New York 10003, USA

3 nTopology, New York 10013, USA

4 Università degli Studi di Milano, Department of Computer Science, 20133 Milano, Italy

5 New York University, Department of Mechanical and Aerospace Engineering, New York 11201, USA

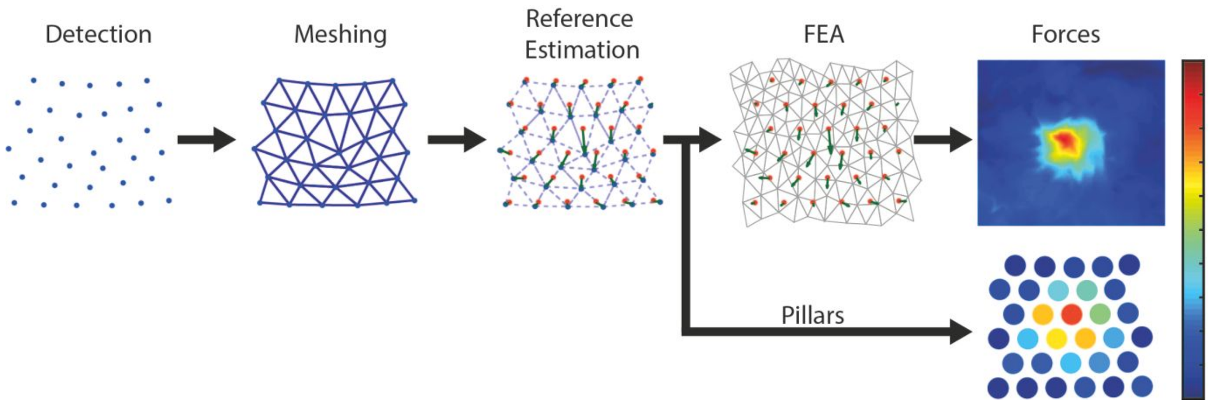
6 ETH Zurich, Institute for Mechanical Systems, 8092 Zürich, Switzerland

7 EMPA, Swiss Federal Laboratories for Materials Science and Technology, 8600 Dübendorf, Switzerland

ç – Equal contribution

* **-** Corresponding authors

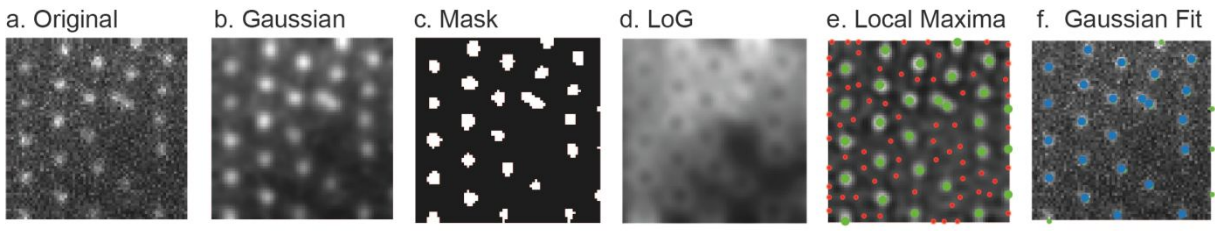
This document provides a detailed description of the algorithms used in the image processing pipeline, which is composed of 4 steps: (1) markers detection, (2) inference of reference graph topology, (3) computation of displacements, and (4) traction force reconstruction. The 4 steps are illustrated in Supplementary Figure 1 and can be seen in supplementary videos 1 and 2.



Supplementary Figure 1. The 4 pipeline stages, which automatically convert images of reference-free TFM markers to traction forces.

1. Markers Detection

The algorithm input is a gray scale image acquired by fluorescence microscopy (Supplementary Figure 2 (a)). It contains either quantum dots[1] or micropillars[2] as markers. The first step of the algorithm analyzes the image to extract the 2D position of each marker. First, a Gaussian filter with a user-defined sigma is applied to the image to reduce noise, i.e. the image is convoluted with a 2D-Gaussian (Supplementary Figure 2 (b)). This image is then binarized at the intensity level determined algorithmically[3] to generate a mask with candidate regions where the markers could potentially be located (Supplementary Figure 2 (c)). In parallel a Laplacian of Gaussian (LoG) filter is applied to the original image. Here, the image is convoluted with a LoG kernel. The result is an image with low values in locations where originally there was no gradient of intensity, i.e. the center of markers, and high values where intensity gradients were strong (Supplementary Figure 2 (d)). The LoG image then is subtracted from the Gaussian image to generate a new frame with sharper peaks at the position of the markers. In this resulting image the local maxima are determined. A pixel is considered a local maxima if it displays the highest intensity value among its eight neighboring pixels (Supplementary Figure 2 (e)). If a pixel is a local maxima and is in the candidate region of the mask, then the pixel's position is used to least square fit a 2D Gaussian in its vicinity (Supplementary Figure 2 (f)[4]) in the original unfiltered image. The fitted Gaussian has a single user-controlled parameter: sigma. Duplicate detections that can arise from two maxima located close together, for which the Gaussians merge, are eliminated after the fitting.



Supplementary Figure 2. Detection (a) Original image, (b) Gaussian filter (c) Mask generated from thresholded image in (b) using Otsu-Threshold, (d) Laplacian of Gaussian (LoG), (e) Detected local maxima in resulting image from LoG subtracted from Gaussian image. The maxima detected in the mask and outside of the mask are in green and red, respectively. (f) Least square fit of Gaussians at the detected local maxima

2. Inferring Graph Topology

The previous step generates a set of 2D position of the markers: the regular lattice connectivity between them shall now be reconstructed. The challenge lies in the fact that the grid is deformed by the traction forces, which are unknown. The only prior is that, before the displacement induced by the cells, the markers were positioned over an unknown subset of a regular triangular lattice of unknown size and orientation. Crucially, for every marker we only use its position and do not require any additional information.

Differently from previous methods, our reconstruction shall be tolerant to occasional errors in the detection of markers, correcting false negatives (markers missed by the detection procedure) as well as false positives (markers "hallucinated" by the detection procedure). For example, false negatives and positives may be induced by noise, or arise by the misinterpreting two adjacent markers as a single one.

2.1. Formal Problem Definition

The input of this step is a set of *detected* 2D points *D*, each point represented by a pair of coordinates. The output is a set $P \subset D$ of *false positives*, a set *N* of *false negatives* (new 2D points which are inferred to exist although they were never directly observed), and a permutation α of 2D points $(D \;/\; P \cup N)$ into the position of a regular triangular grid. Once the permutation is known, the relaxed positions can be inferred by smoothing the grid (Section 3) and the displacements are then defined by the difference.

Among all potential choices of *N, P,* α (for a given observation *D*), the one minimizing its *unlikelihood* is defined as

$$k_1|N| + k_2|P| + E(\alpha(D/P \cup N)), \tag{1}$$

where $|X|$ denotes the cardinality of the set *X*, k_1 and k_2 are fixed scalar parameters weighting the penalty associated to different factors (determining how much false positives and negatives impact the likelihood of a solution), and *E(X)* is an approximation of the potential elastic energy which would be required to deform a regular pattern into the given pattern *X* (on the ground that the least energetic configuration is the most plausible).

To define *E*, the elastic substrate is approximated as a network of Hookean linear springs connecting each pair of direct neighbors on the lattice, resulting in:

$$E(X) = \sum_{i=0}^{|X|} \sum_{x_j \in N_i} \left(|x_i - x_j|_2 - L\right)^2, \tag{2}$$

where L is the step length of the regular lattice, and N_i is the set of neighbors of x_i.

2.2. Problem Analysis

This formulation makes the complexity of the problem apparent: it is a combinatorial optimization over a large space, with a strict set of constraints that must be enforced to obtain valid solutions. As observed in[1], an exhaustive search of the optimal solution via a branch and bound algorithm would be practical only for tiny problem instances (up to around 20 vertices).

We propose an algorithm to efficiently find a low energy solution. While the approach is not guaranteed to find the optimal solution, in the large majority of cases, it indeed finds the same solution obtained by an exhaustive search, while being orders of magnitudes faster and scaling to tens of thousands of markers.

The proposed approach is incremental: an initial guess for permutation is initially proposed, and then iteratively improved until convergence.

The key novelty of the algorithm is the use of two complementary representation for an intermediate solution for α: a *Mesh*-based and a *Lattice*-based representation. The two representations are equally capable of expressing any consistent permutations α into a lattice, but crucially, each can also represent certain inconsistencies of different nature. Switching from one to the other allows reducing these inconsistencies very efficiently.

Overview. The initial guess of αis expressed as a Mesh-based representation. This solution is then converted back and forth between Lattice and Mesh representation, applying an optimization step after every conversion. The optimization step improves the solution, greedy applying a sequence of local operations (i.e. operations affecting only a small, constant portion of the representation).

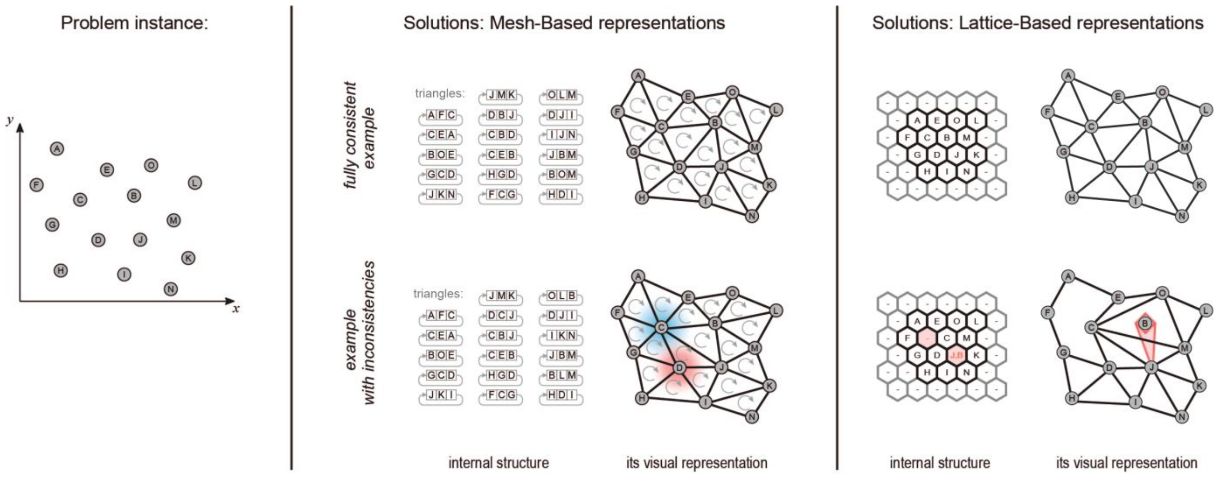
In both representations, a local operation lowers the number of inconsistencies and the energy term for *E* in equation (2).

At the end of the process, residual inconsistencies in the final α are interpreted as false positive and negatives, creating sets *P* and *N* respectively.

In summary, the algorithm can be written as:

```
•  Variables: Mesh, Lattice
1. Mesh ←initial_guess ( D );                                   Section 2.3
2. loop:
      a. local_operations( Mesh )                               Section 2.4
      b. convert: Mesh → Lattice                                Section 2.5
      c. local_operations( Lattice )                            Section 2.6
      d. if converged then exit loop;                           Section 2.8
      e. convert: Lattice → Mesh                                Section 2.7
3. ( P,N ) ←fix_residual_inconsistencies( Lattice );            Section 2.9
```

First, the two representations are described, and then the other parts of the algorithm are described in more detail.


Problem instance:
Solutions: Mesh-Based representations
Solutions: Lattice-Based representations
fully consistent example
example with inconsistencies
triangles:
AFC CEA BOE GCD JKN JMK DBJ CBD CEB HGD FCG OLM DJI IJN JBM BOM HDI
triangles:
AFC CEA BOE GCD JKI JMK DCJ CBJ CEB HGD FCG OLB DJI IKN JBM BLM HDI
internal structure
its visual representation
internal structure
its visual representation

Supplementary Figure 3. An instance of the graph-topology problem consists of a set of 2D points, like in this toy example (left most). In our system, a solution is represented in either of two alternative representations: mesh-based, and lattice-based. Mesh-based representation consists of a two-manifold, well-oriented and simple triangular mesh (middle column), whose vertices are the given set of points. A lattice-based solution consists of a regular honeycomb grid of cylindrical cells which can be assigned to one (or multiple) vertices (rightmost column). Either representation is capable of expressing any permutation into a lattice, like the one depicted on top (which is, in this case, the energy minimizer, and therefore the optimum). Each representation is also subject to include inconsistencies of different types. In the mesh representation, for example, internal vertices can have a valency different from 6 (in the example on bottom: a valency-5 vertex is highlighted in red, and valency-7 vertex in blue). In the lattice-based representation, there can be internal "holes" in the regular grid, or multiple nodes assigned to the same grid cell, (bottom-right).

Mesh Representation. In this representation, the permutation into a lattice is represented as a two-manifold, triangular, mesh whose vertices are *D*. In other words, a Mesh consists of all points in *D* connected by a set of triangles; the sides of the triangles are termed edges. This structure is the ubiquitous way to represent piecewise linear surfaces, and has been deeply studied for example in Geometry Processing (e.g. see[5] for an overview). The meshes are open and simple, meaning that they have a unique loop of boundary edges and vertices (i.e. there are no internal "holes"). They are two-manifold, meaning that that every edge of each triangle is either a boundary edge or is shared by exactly another triangle. The valency of a vertex is defined as the number of triangles sharing that vertex. A mesh is called regular when each boundary vertex has valency *<6*, and each other vertex has valency 6.

A regular mesh can be interpreted as a permutation αof its vertices *D* into a lattice, since it encodes a regular lattice. A mesh which is not regular, conversely, does not correspond to any such permutation. The inconsistencies of a Mesh representation are, therefore, vertices breaking the above requirement on the valency (termed "irregular" vertices).

Lattice Representation. Our lattice, or matrix, is a 2D regular grid of hexagonal cells (honeycomb tiling). Each element of *D* is hosted in one cell. A cell can be empty, or host one or more elements. In the lattice, non-empty cells always form a contiguous subset of the grid, and all cells on the boundary of the grid are empty.

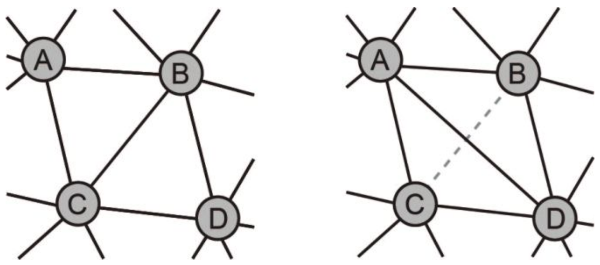
Ideally, each non-empty cell hosts only a single element of *D*. Also, empty cells are all found in one continuous set around the lattice (i.e. there is no island of empty cells completely surrounded by non-empty cells). A lattice with these properties can be trivially converted to a valid permutation α. The inconsistencies of a lattice representation are, therefore, cells hosting more than one element of *D* (these elements are called “colliding”), or empty cells that are not connected to the boundary of the grid by a sequence of empty cells (these cells are called “holes”).

2.3. Step 1: Mesh Initialization

The algorithm is initialized by computing the 2D Delaunay triangulation[6] of the set of vertices *D*. This process is fast and guarantees to produce a two-manifold, simple mesh. In undeformed areas, this mesh reproduces the connectivity and the shape of a regular grid and it is thus a correct solution. However, in distorted regions, irregular vertices are introduced. The next steps address these inconsistencies.

2.4. Step 2: Mesh Local Operations

An edge-flip[6] is a standard local operation commonly used in the context of mesh optimization and simplification. In the represented context, edge-flips are used to improve the quality of the mesh, striving to obtain a regular connectivity on the entire mesh and a decreased energy.



Supplementary Figure 4. An example of an edge-flip operation: in this mesh, flipping the edge B-C decreases the valency of vertices B and C by one, and increases the valency of vertices A and B by one.

Specifically, the effect of an edge-flip is scored by two numbers (e_a, e_b). e_a is the induced increase in the number of regular vertices (or decrease if negative): as an edge-flip increases the valency of two vertices by one, and increases the valency of other two vertices by one (Supplementary Figure 4) it can either increase, decrease or leave unaffected the total number of irregular vertices on the mesh. e_b is the induced decrease is on energy *E*: only the edge being flipped changes its length and thus its contribution to the energy *E* (Equation (2)). An edge-flip is beneficial either when $e_a > 0$, or when $e_a = 0$ and $e_b > 0$ All potential edge-flips (there is exactly one per edge) are tested, and the ones that are beneficial are performed. In a second pass, the algorithm tests all the possible pairs of consecutive edge flips affecting a common vertex that have a *combined* (summed) beneficial effect.

Edge-flip operations which would reduce an internal vertex valency below 3, are always disallowed as it would compromise the two-manifoldness of the mesh. While this step is effective at solving many locally inconsistent configurations, it might fail at identifying the long sequences of flips that may be necessary in images with large displacements. The mesh is therefore converted to a lattice representation, to continue the optimization.

2.5. Step 3: Mesh to Lattice Conversion

Starting from an empty lattice, the mesh is explored, copying the indices of the encountered vertices into the lattice cells, one by one. The mesh is visited with a “flood-fill” approach, i.e. from a “seed” triangle and iteratively proceeding by expanding the visit to neighboring triangles, until the entire mesh is visited.

Crucially, the parts of the mesh which are more regular are explored first; in this way, the less ambiguous parts of the mesh, which can be interpreted with higher confidence, serve as a guidance to settle the more ambiguous parts.

In practice the problematic parts of the mesh are surrounded by the visit (thus isolated) and then conquered from the exterior inwards.

A precise description of the employed algorithm is given below. In the following, we define the “equilateral factor” of a triangle with sides lengths a,b,c, with $a > b > c$, as the real number $((c + b)/a - 1)$. A perfectly equilateral triangle has factor 1, a completely degenerate triangle has factor 0, and any intermediate case has values in between.

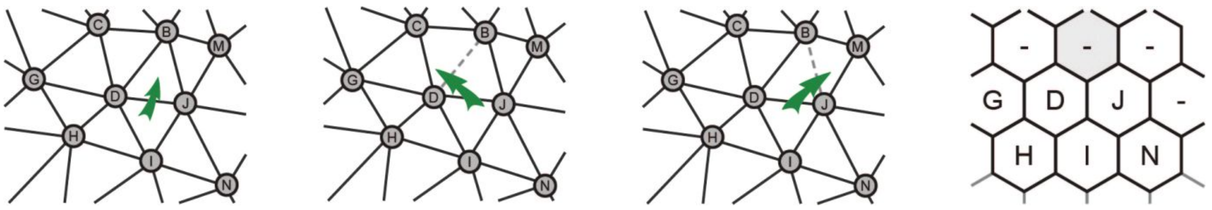
Identifying the Seed Triangle. Every triangle of the mesh is labelled as either reliable or not reliable. A reliable triangle fulfills three conditions: (i) it is not on the mesh boundary, (ii) its three vertices are regular (valency 6), and (iii) its shape is sufficiently equilateral (equilateral factor larger than 0.85). The seed is selected as the furthest reliable triangle (under hop distance) from any unreliable triangle; in other words, the triangle which is surrounded by the maximal number of reliable triangles.

The three vertex indices of the seed triangle are copied into a group of three reciprocally adjacent grid cells.

Enumerating and Prioritizing the Expansion Moves. An expansion move potentially enlarges the set of visited triangles by one element, traversing one mesh edge adjacent to an already visited triangle.

There is one expansion move for each internal mesh edge. At any given time, an expansion move is available if at least one of the triangles shared by the corresponding edge has been visited (note that this includes edges with visited triangles on both sides). During the fill, all available moves are kept in a set, and prioritized, from highest to lowest confidence, according to summed equilateral factors of the two triangles sharing the edge. The set of available expansion moves is kept in a priority queue and is initialized with the three edges of the seed triangle.

Iteratively, the element with the highest priority is extracted from the set of potential moves, removed from the set, and the move executed. After the move, up to two new available expansion moves are added to the set to reflect the expansion of the set of visited triangles (and evaluate their priority). Traversed edges are flagged as such and never added to the set of available moves a second time. The procedure terminates when the set is empty (i.e. when each internal mesh edges has been traversed exactly once).



Supplementary Figure 5. An example of the execution of an expansion-move. Executing the move crossing the edge DJ and visiting triangle DJB (left-most diagram), would have the effect of filling the grayed square of the lattice with the vertex labelled as B (right-most diagram). If that square is already filled by a vertex other than B, or if vertex B is already allocated anywhere else on the lattice, then this move would cause an inconsistency. In this case, an edge-flip of one of the other two edges of DJB is attempted first: flipping the BD edge causes the grayed square to be filled with vertex C instead of B (mid-left diagram); flipping the JB edge causes it to be filled with vertex M instead of B (mid-right diagram). If either edge-flip avoids the conflict, then it is performed before the move executed.

Executing an Expansion Move. An expansion move consists of expanding the visit from the visited triangle t_A, over edge e_0, into the (potentially not yet visited) triangle t_B. Let v_i be the vertex of t_B opposite to edge e_0. Executing the move consists of copying v_i into a given cell of the grid with index c_i (see Supplementary Figure 5). The cell index c_i is fully determined by the cell position on the grid, of the vertices of t_A. Note that, due to previous expansion moves, it is possible that cell c_i is already occupied by a vertex; likewise, and independently, it is possible that vertex v_i is already allocated in a grid cell.

Therefore, three cases can arise:

Expand, when vertex v_i is not currently allocated anywhere in the grid, *and* cell c_i is currently vacant;

Confirm, when vertex v_i is already present in cell c_i (i.e. cell c_i already contains vertex v_i);

Contradiction, when vertex v_i is already allocated in some other cell position different from c_i, or cell $c_i$is already occupied by some vertex different from v_i, or both.

In the Expand case, cell c_i is filled with vertex v_i. In the Confirm case, nothing needs to be done; when that case arises, it means that the current expansion moves is consistent with the lattice layout as inferred from the already visited triangles. This can happen, for example, because vertex v_i was previously reached from a different direction, possibly along a completely different triangle paths from the seed.

Vice-versa, the third case happens because the starting mesh is not fully regular. Executing the move would create an inconsistency. At this point, it is checked whether the conflict can be avoided by means of edge flips performed on the mesh. There are two potential edge flips, corresponding to 2 edges of t_B that are not e_0. If either edge flip is viable, it would result in a different vertex indices v_i' and v_i'' in place of v_i (see Supplementary Figure 5), and therefore in a different case. The edge-flip is performed if it is viable and results in the removal of the contradiction (if the extremely rare case when both edge-flips qualify, the one resulting in the highest summed equilateral factors of the two affected triangles is selected). Note that the edge flips are performed, in this phase, regardless of their local effect on the valency of the vertices or the energy.

After the expansion move, assuming no contradiction arose or that it could be resolved, the two external edges of t_B are added to the set of potential moves, unless these edges have been already processed.

Rationale. Importantly, triangles can be assigned to the grid by independently assigning their three vertices to grid cells, before that triangle is explicitly visited. This happens for example when the boundary of the visit meets with itself after that the flood-fill encircled a problematic region (e.g. a region containing irregular vertices) from the two different sides. Eventually, these triangles will be explicitly visited also. When that happens, the visit can only either confirm or contradict the previously found grid layout. In case of contradiction, the grid values are not overwritten, because earlier moves are, by design, considered more reliable than later moves. Instead, the existing grid values are used to correct the mesh connectivity (by means of flips). In conclusion, the flip operations which are performed in this phase are driven by the global grid structure of the mesh, rather than by its local configurations, differently from the local optimization on the mesh (Section 2.4). In this sense, this flip identification strategy is drastically more "long sighted" and capable of avoiding local minima.

2.6. Step 4: Lattice Greedy Optimization

A Lattice admits three local operations: (1) permutation of a small subset of the vertex indexes stored in the cells, (2) hole filling and (3) conflicts resolution. A set of local operations is tested, and all the ones with a positive effect on the global energy are performed (Equation (1))

Operation 1: Greedy Permutations. Given a set of *n* vertices $v_0 \dots v_{n-1}$ assigned to cells c_0 … c_{n-1}, a cycle permutation is the reassignment of each vertexv_i to $c_{(i+1)\%n}$ ($\%$ being the modulo operator). A cycle permutation is beneficial if it results in an overall decrease of the energy. A brute force approach, where each set of 2, 3, and 4 adjacent cells are tested for all potential cycles is used. Cycles of size 2 (that is, swaps between pairs of cells) are also tested between any pairs of cells separated by a single cell. In total, 48 cycles are tentatively tried around each non empty grid cells, in a fixed pattern. The resulting algorithm is linear with the number of cells and fast, because testing for the effect of a cycle requires to sum up only a limited number of addendum to the energy in Equation (2).

Operation 2: Hole Filling. A Lattice "hole" is defined as an empty grid cell that is not connected to the lattice boundary by a path of empty cells.

Each disconnected empty cell is a hole and is evaluated for removal. In order to remove a hole, first, the vertex from a neighboring non-empty cell must move to fill its position, thus shifting the hole to that cell; then, the process has to be repeated until the hole ends up neighboring one boundary empty cell, or the moving vertex is a conflicting vertex (i.e. was one of the two vertices allocated in the same cell). In the latter case, that "conflict" inconsistency is also removed. Each movement of a vertex into the empty position comes with an associated increase (rarely, a decrease) of the energy *E*. In other terms, in order to fix a hole, a path from that position to a either a connected empty cell, or to a conflicting vertex needs to be found.

The problem can be cast as a minimum cost path, which is solved using the Dijkstra algorithm (seeded at the cell presenting the hole, and targeted at any eligible destination of the path). The cost of every step is defined as the increase (rarely, the decrease) of the energy for the corresponding swap. Additionally, the cost of the final step is further decreased (possibly down to a negative number) by the value k_0 (Equation (1)), to reflect the decrease of the number of holes. In case that the final destination of the graph is a conflicting vertex, the cost is also decreased by the additional value k_1, to reflect the decrease of the number of conflicts.

When the minimum cost path is identified, it is applied it if and only if its total cost sum up to a negative number. Otherwise, the total likelihood of the found solution would decrease (in other words, a more likely justification for the hole is to assume that an existing point was undetected). To optimize, we abort the Dijkstra search over paths which results in a total cost larger than $(k_0 + k_1)$.

Operation 3: Conflict Resolution. For conflicts inconsistencies, the situation is conceptually similar. A conflict is a situation where two vertex indices are located in the same lattice cell. It can be solved by, first, moving either vertex index to a neighboring cell, thus shifting the position of the conflict to that cell, and repeating this step until an empty cell is reached. If the final empty cell is also labelled as disconnected from the boundary (i.e. it is a hole), this has the additional side effect of simultaneously fixing that hole. Again, the problem is cast as a minimal cost path search, solved via the Dijkstra algorithm. This time, the path starts from the conflicted cell and terminates into any empty cell, either connected or disconnected to the boundary. The cost associated to the move terminating the path is decreased by k_1 or by $k_0 + k_1$, to reflect the fixing of either one or two inconsistencies. Similarly, to the previous case, the resulting fix is considered profitable if the found path has a negative total cost (otherwise, it is concluded that the inconsistency would be more parsimoniously explained by assuming the conflicting vertex to result from a false positive in the point detection).

2.7. Step 5: Lattice to Mesh Conversion

This step is implemented as a variant to Step 4, that is, the mesh is visited again using a flood-fill seeded at an appropriate starting location. The only difference is that this time, the lattice is not initialized as empty, but kept unmodified at its current values. Consequently, the only two possible outcomes for expansion moves are “confirm” or “contradiction” (and never “expand”) and the only sought effect is to perform edge flips in the latter case.

Rationale. The objective is to modify the current mesh configuration to make it more similar to the current lattice configuration, but only by means of valid local mesh operations (the lattice is only used to guide these operations); this ensures that the mesh representation is kept consistent. The aim is not to obtain a mesh configuration perfectly mirroring the one represented by lattice, because inconsistencies which are potentially left in the lattice (“collisions” and “holes”) cannot be represented in the mesh representation.

2.8. Step 6: Convergence Detection

The algorithm stops when an entire iteration (2 conversions and 2 sets of local operations) are not changing the lattice representation. In our experience, this never takes more than 4 iterations.

2.9. Step 7: Removal of Lattice Inconsistencies

For every set of points in *D* allocated to the same cell, one is selected to occupy that cell, and the others are considered false positives and added to *P*. Similarly, every set of isolated empty cells (i.e. separated from the boundary by non-empty cells) are considered false negatives and added to *N*. Their position is computed as the average position of their neighbors (the averaging is repeated until convergence, for islands of two cells of more).

3. Displacement Computation

At this stage, the markers in the input image have been detected, and the connectivity between them computed. The next step is to reconstruct the marker displacements from the rest configuration, to the configuration captured in the image. This is achieved with a relaxation process that warps the markers in the image into a regular grid: the displacements are then computed by taking the difference between the initial and relaxed positions. The relaxation process is modeled using the graph Laplacian

$$\boldsymbol{L}_{ij} = \begin{cases} -\sum_j \boldsymbol{L}_{ij} & i == j \\ 1 & i \text{ is a neighbor of } j \text{ ,} \\ 0 & \text{otherwise} \end{cases}$$

which, for a perfectly regular mesh, satisfies

$$\boldsymbol{Lx} = \boldsymbol{0},$$

where *x* are the coordinates of the vertices in the mesh. The vertices are split into two groups - the inner vertices, which need to be relaxed to their original position, and the boundary vertices which are fixed. The Laplacian L is split accordingly.

$$x = \begin{bmatrix} x_i \\ x_b \end{bmatrix} \qquad L = \begin{bmatrix} L_{ii} & L_{ib} \\ L_{bi} & L_{bb} \end{bmatrix}$$

The system of equations is reduced to

$$\begin{bmatrix} L_{ii} & L_{ib} \end{bmatrix} \begin{bmatrix} x_i \\ x_b \end{bmatrix} = 0$$

and solving

$$x_i = -L_{ii}^{-1} L_{ib} x_b$$

yields the relaxed positions of the inner vertices. With the boundary vertices fixed, this has the effect of simultaneously moving all the vertices to the barycenter of their on-ring neighborhood, thus creating a regular hexagonal grid.

4. Force Reconstruction

The displacements computed in the previous step are already a good proxy for the traction forces. The conversion of the markers' displacements into traction forces depends on the type of image. For pillars, the forces are discretely applied to each pillar and can be reconstructed directly from the displacements. For quantum dots, the forces are applied continuously on the substrate, and thus their reconstruction requires a finite element method.

4.1. Pillars

For pillars, the forces can be directly computed from the displacement field *u* obtained in the previous steps as

$$F = \frac{3EI}{L^3} u,$$

where *E* is the Young's modulus, *I* is the moment of inertia, and *L* is the length of the pillars[2]. Results of this procedure are shown in Figure 3 in the main paper.

4.2. Quantum Dots

For quantum dots, we have to solve a volumetric deformation problem, which given the target displacements of the vertices on the surface, finds the traction forces inducing such displacement.

Volumetric Meshing. To set up our physical simulation problem, it is necessary to first discretize the substrate, decomposing it into a mesh composed of tetrahedra. The tetrahedral mesh is adaptive, with a higher density in the regions corresponding to the higher displacement.

The meshing proceeds in 4 steps.

Step 1. A background 2D mesh is created as the Delaunay triangulation of the displaced markers, plus a few additional points on an extended bounding box whose size is user-controlled. In the following, we will distinguish the inner box (the substrate) from the outer box (substrate + padding).

Step 2. A sizing field is computed in 2D to indicate the target edge length of the mesh to be used in the simulation. First, the “source” of the sizing field is determined as the mesh triangles whose vertices have a displacement larger than a user-given threshold (set as percentage of the max displacement across all dots - *18%* by default in our application). The target size for the source region is then set as a user-defined percentage of the median distance between adjacent quantum dots - *30%* by default in our app. If no marker falls within this threshold, the whole inner-box is set to be the "source", and the target size is set to the median distance between adjacent dots. The sizing field is then propagated from the source region to the rest of the background mesh so that the ratio between adjacent vertices follow a user-given ratio (a gradation of *1.2* is used by default).

Step 3. A dense tetrahedral mesh of the outer box is computed in 3D using TetGen[7]. Let s_{max} be the maximum target size of the 2D sizing field computed above. The 2D sizing field is extended through the dense 3D volume so that it is equal to the original field on the top (the surface), and equal to s_{max} on the bottom.

Step 4. The dense tetrahedral mesh is remeshed with mmg[8] to follow the 3D sizing field.

Finite Element Method. We propose two approaches, one for creating a quick preview of the forces using a linear elastic material model, and a second one for accurate reconstruction using a Neo-Hookean material model. For both modes the material parameters are obtained through material testing (Supplementary Figure 6[1]) and assume no additional external forces, which is a realistic approximation for most experimental setups. We solve for the displacement u

$-div(\sigma(u)) = 0$ and $u = g$ on the boundary,

where $\sigma(u)$ is the stress tensor and g are the boundary conditions. Note that the form of $\sigma(u)$ depends on the material model: for linear elasticity

$$\sigma(u) = 2\,\mu\epsilon(u) + \lambda\,\mathrm{Tr}(\epsilon(u))I,$$

with $\epsilon(u) = \frac{1}{2}(\nabla u^T + \nabla u)$, for Neo-Hookean

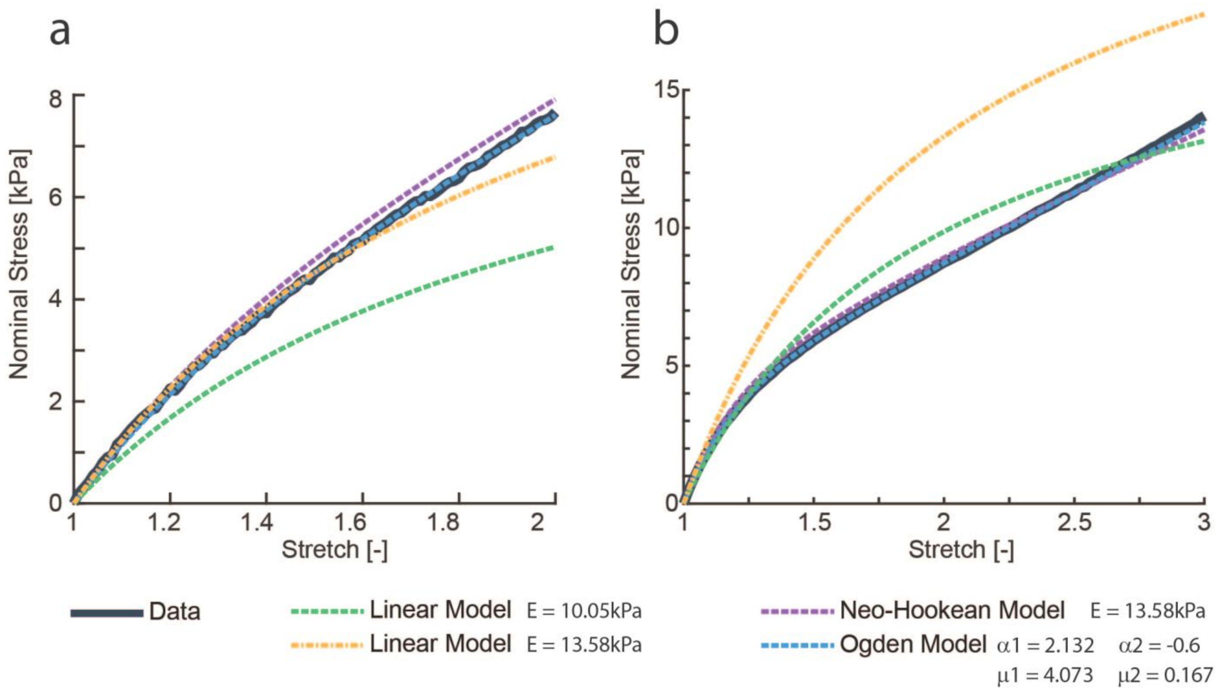
$$\sigma(u) = \mu\left(F - F^{-T}\right) + \lambda \ln{(\det F)} F^{-T},$$

with $F(u) = \nabla u + I$.

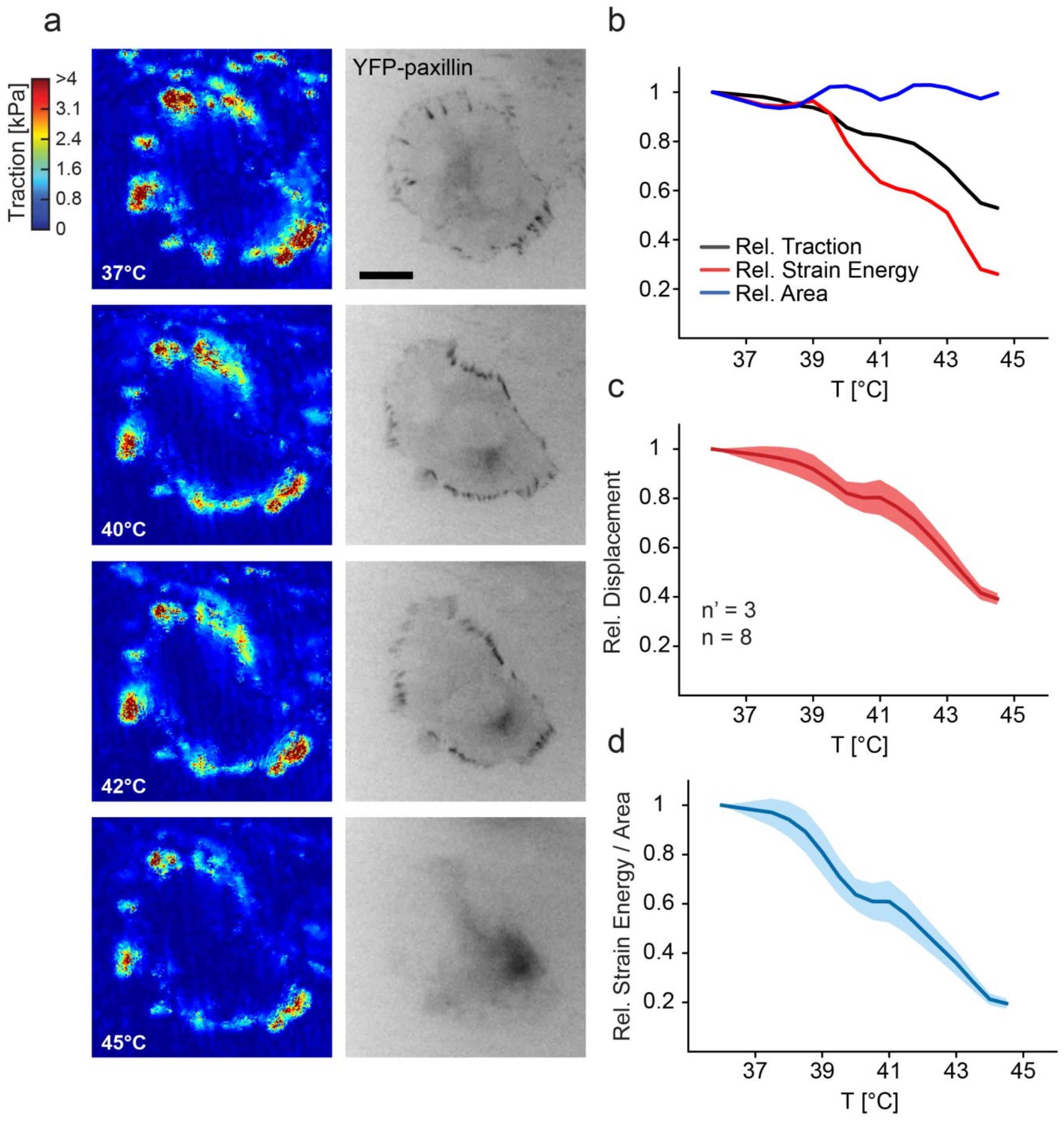
In both cases, the same boundary conditions g are specified. For the bottom side g is zero, while for the top side it corresponds to the displacement field reconstructed in Section 3. Note that, since a planar displacement field is measured in the *xy*-direction, we leave the *z*-direction free to move to account for buckling effects. Since the displacement field is defined only on the vertices of the detected points, radial basis function is used for interpolation with Gaussian kernel[9, 10] to extend it to the whole plane. Finally, to obtain the traction forces from the solution of the partial differential equation, we multiply the stress σ with the face normal. We use isoparametric linear Lagrangian elements in both cases. Results of this procedure are shown in Figure 2 in the main paper.

5. Material Characterization

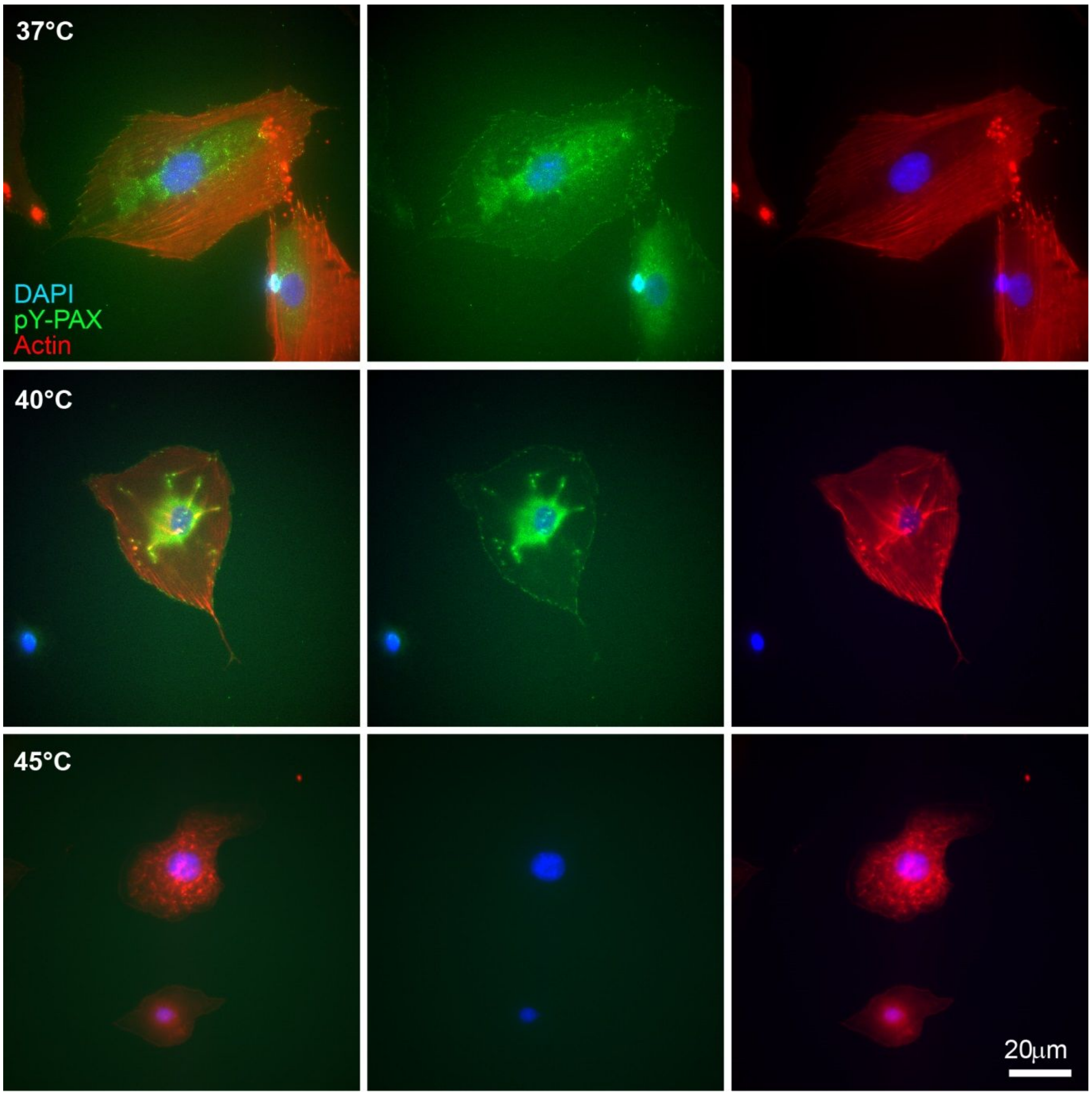
In a previous publication the mechanical properties of the material was thoroughly characterized[1]. Fitting of the hyperelastic Ogden model[11] achieved a very close recreation of the uniaxial and biaxial material tests. Here, for the sake of implementation and computation speed a linear model and Neo-Hookean model were fitted to the test data. The data was least-square fitted for both uniaxial and biaxial simultaneously. Given the incompressible properties of the silicone used, Poisson ratio was fixed at 0.49 and the only free parameter was the Young's modulus E. The Neo-Hookean model shows a very good fit and deviates only slightly from the Ogden model for large stretches (Supplementary Figure 6). The best overall fit for the linear model was achieved at a lower stiffness of $E = 10.05kPa$, in which case the uniaxial did not match well (Supplementary Figure 6 (a)). Using the same stiffness as for the Neo-Hooke, the uniaxial fit was better, but the biaxial fit quite off (Supplementary Figure 6 (b)). Hence, for accuracy the Neo-Hookean model is chosen, whereas for speed the linear model can be used.



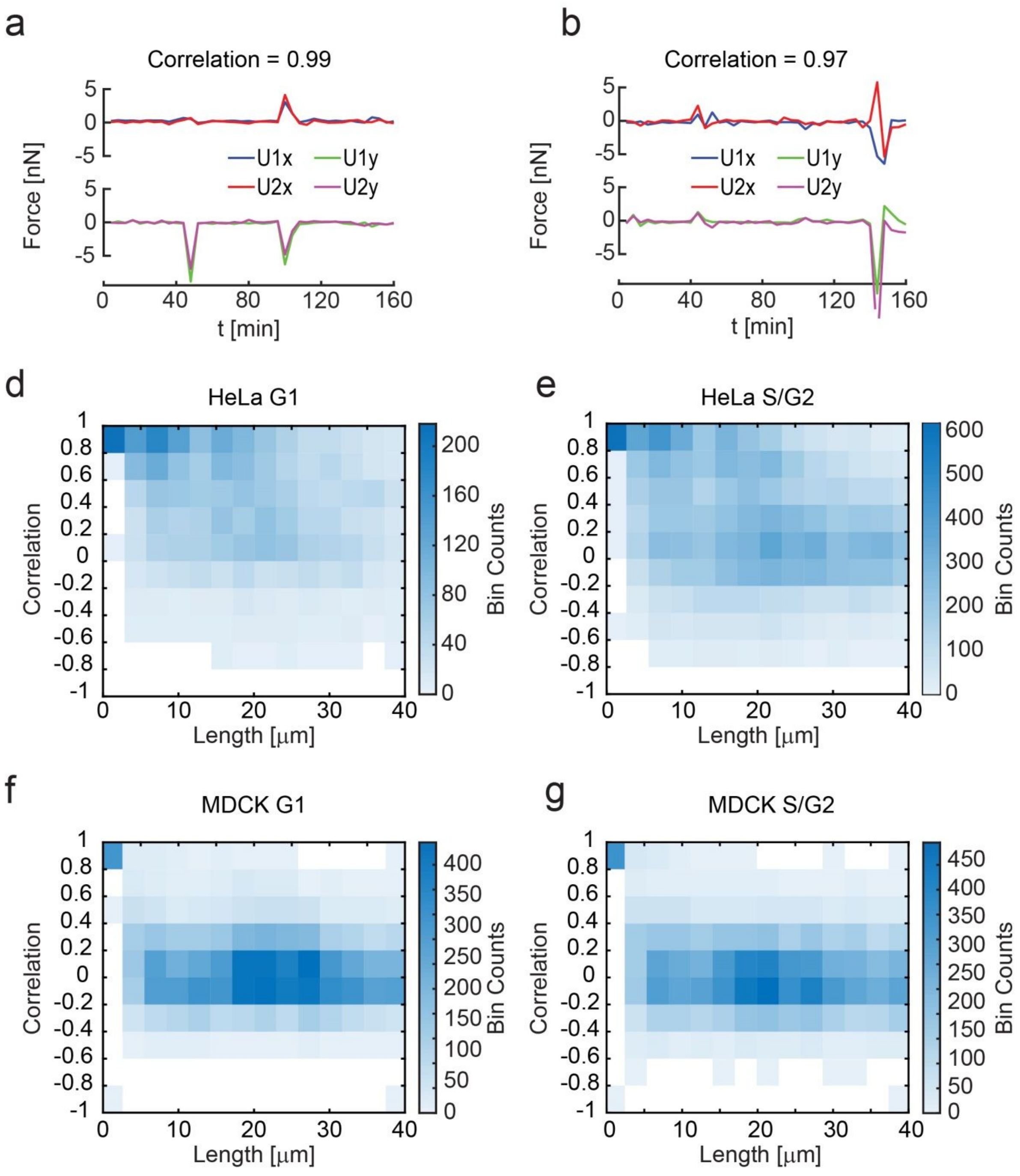
Supplementary Figure 6. Material model fitting. (a) Uniaxial test data and fitted curves. (b) Equibiaxial test data and fitted curves.



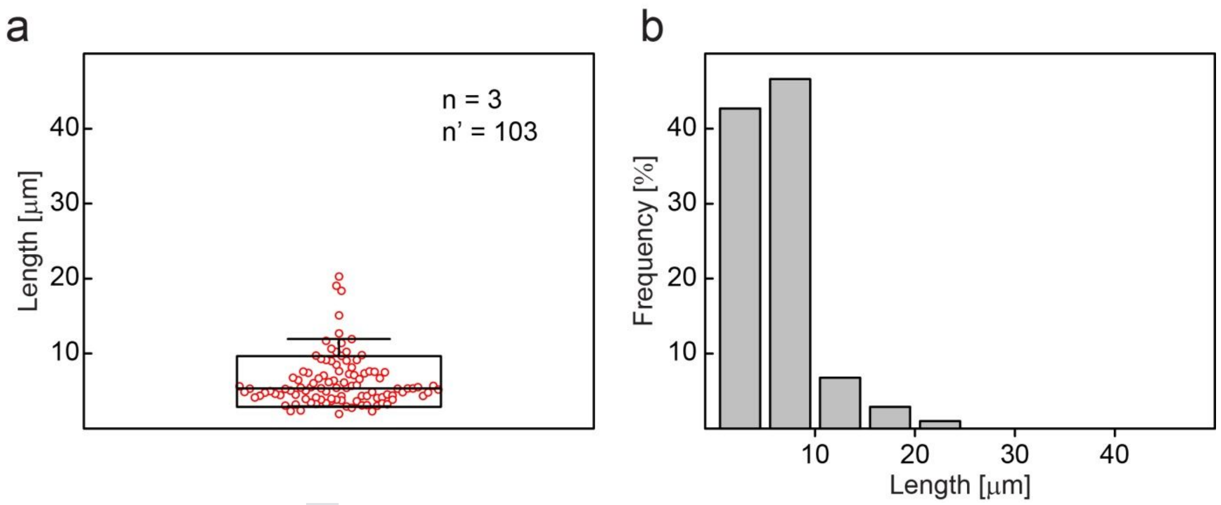
Supplementary Figure 7. Mechanical response of YFP paxillin-expressing REF-52 cells to increasing environmental temperature. (a) Representative example of cellular traction maps (left panels) and corresponding inverted fluorescent images (YFP-paxillin, right panels). Scale bar is 10 µm. (b) Corresponding evolution of traction, strain energy and area relative to the values measured at 37°C. Population analysis reporting the displacement (c) and strain energy/area (d) relative to the values measured at 37°C. A red or blue line defines the average value. The shaded area corresponds to the standard error of the mean. n = number of individual measures, n' = number of independent experiments.



Supplementary Figure 8. Focal adhesion and actin filaments remodelling in YFP paxillin-expressing REF-52 cells exposed to increasing environmental temperature. Representative examples reporting the distribution of actin filaments (red), phosphorylated paxillin (green) and nuclei (blue) in cells exposed to temperatures up to 40°C (middle row) and 45°C (bottom row) as compared to control cells maintained at 37°C (upper row).



Supplementary Figure 9. Correlation of traction forces. (a, b) Two examples of entanglement between highly correlated pillars actuated by HeLa cells. Force components in the x (U1x and U2x) and y (U1y and U2y) are reported over a total time of 160 min. (**d**-**g**) Binned scatter plot of pairwise force correlation between pillars (as function of the length between the pair) at displaced by HeLa cells in the G1 (d) or S/G2 (e) cell cycle phase, respectively. Corresponding binned scatter plots for MDCK cells in the G1 (f) or S/G2 (g) phase.



Supplementary Figure 10. Actin filaments in MDCK cells. (**a**) Boxplot reporting the calibration of actin stress fiber length in fixed samples. The bars extend from the 25th to the 75th percentile. A line in the box represents the mean value while the box height corresponds to its standard deviation. Individual measures are reported as open red circles. (**b**) Corresponding frequency distribution (in percentage) of measured values.

6. Supplementary References

(1) Bergert, M.; Lendenmann, T.; Zündel, M.; Ehret, A. E.; Panozzo, D.; Richner, P.; Kim, D. K.; Kress, S. J.; Norris, D. J.; Sorkine-Hornung, O. J. N. c. **2016,** 7, 12814.

(2) Tan, J. L.; Tien, J.; Pirone, D. M.; Gray, D. S.; Bhadriraju, K.; Chen, C. S. J. P. o. t. N. A. o. S. **2003,** 100, (4), 1484-1489.

(3) Otsu, N. J. I. t. o. s., man,; cybernetics. **1979,** 9, (1), 62-66.

(4) Cheezum, M. K.; Walker, W. F.; Guilford, W. H. J. B. j. **2001,** 81, (4), 2378-2388.

(5) Botsch, M.; Kobbelt, L.; Pauly, M.; Alliez, P.; Lévy, B., *Polygon mesh processing*. AK Peters/CRC Press: 2010.

(6) Shewchuk, J.; Dey, T. K.; Cheng, S.-W., *Delaunay mesh generation*. Chapman and Hall/CRC: 2016.

(7) Si, H. J. A. T. o. M. S. **2015,** 41, (2), 11.

(8) Dapogny, C.; Dobrzynski, C.; Frey, P. J. J. o. c. p. **2014,** 262, 358-378.

(9) Pazouki, M.; Schaback, R. J. J. o. C.; Mathematics, A. **2011,** 236, (4), 575-588.

(10) Fornberg, B.; Larsson, E.; Flyer, N. J. S. J. o. S. C. **2011,** 33, (2), 869-892.

(11) Ogden, R. W. J. P. o. t. R. S. o. L. A. M.; Sciences, P. **1972,** 326, (1567), 565-584.