Nearly Isostatic Networks and Allosteric Effects

Ian McKenzie

April 2020

I certify that this project report has been written by me, is a record of work carried out by me, and is essentially different from work undertaken for any other purpose or assessment.

Abstract

There has been a lot of recent research interest in tuning the properties of spring networks. This project specifically focuses on the ability to tune the strain response of one part of the network in response to a strain applied to a remote part of the network. In this project we explore the fascinating mathematics of rigidity theory and implement a computational model for working with spring networks in Python. We then recreate the results of a recent paper, *Designing allostery-inspired response in mechanical networks* [1], with some slight tweaks. We find that the linear response is almost always tunable, but the non-linear response does not necessarily correlate well with it.

Contents

1 Introduction

A framework in the mathematical sense is effectively a drawing of a graph with physics. That is, we consider a drawing of a graph embedded in a d-dimensional space and suppose that its elements can move in some way. In this project we will consider only $\mathbb{R}^2$ and "bar-and-joint" frameworks (hereafter simply frameworks). This means that the vertices can move freely in the space under the constraint that the distance between any pair of vertices connected by an edge stays constant (i.e. the edges have fixed length). We say that a framework is *rigid* if the distance between all pairs of vertices stays the same through any allowed movement, not just between those connected by an edge.

Frameworks are not only theoretically interesting objects of study, but are also useful in analysing many physical situations. Classic examples include scaffolding and architecture, where a rigid construction is vital for safety and longevity. Other interesting areas that are related include the structures of proteins, which are essentially molecular frameworks.

It is this link to biology that provides the inspiration for this project. Enzymes are proteins that catalyse reactions. They have an *active site* where the substrate upon which the enzyme acts binds to it. Some also have a regulatory or *allosteric site*, where

a different molecule can bind to the enzyme. This changes the shape of the active site – which need not be close to the allosteric site – inhibiting its catalysis. The aim of this project is to understand this "action at a distance" as it occurs in almost-rigid frameworks. In doing so we closely follow a 2017 paper, *Designing allostery-inspired response in mechanical networks* by Rocks et al.[1], hereafter referred to as Rocks et al.

The reader is assumed to have a general undergraduate level of background mathematical knowledge, especially in graph theory and linear algebra. We start by going through the theory of rigid frameworks. We then look at and elaborate on the linear algebra used in Rocks et al. This is followed by some computational rigidity experiments to test our computational model. After this we describe how the network tuning algorithm works – the main aim of the project. We give three different implementations and then examine the results of them. We conclude with a summary of the work done and some speculation as to what could be done next to better understand the dynamics at play.

All of the code used can be found at `https://github.com/naimenz/rigidity_masters` and the animations produced are available either as files on github or at `https://irmckenzie.co.uk/animations`.

2 Background

In this section we look at the mathematics of rigid frameworks (including a definition for framework) and outline the linear algebra-based formalism we will be using.

2.1 Rigidity theory

We start with definitions for frameworks and motions of frameworks before stating a few preliminary theorems. Much of this theory is drawn from Jack E. Graver's book *Counting on Frameworks* [2].

2.1.1 Frameworks and Rigidity

Definition 2.1. *A* framework $\mathcal{F} = (G, \mathbf{p})$ *is a graph* $G = (V, E)$ *with an embedding* $\mathbf{p}$ *in a* d*-dimensional Euclidean space, here taken to be* $\mathbb{R}^2$ *unless otherwise specified.* G *is called the* structure graph *of* $\mathcal{F}$*. We call the* N *elements of* $V = \{a_0, a_1, ..., a_{N-1}\}$ *vertices and the* N_b *elements of* $E = \{b_0, b_1, ..., b_{N_b-1}\}$ *edges, bars, or bonds. Then the embedding* $\mathbf{p} \in \mathbb{R}^{dN}$ *is a vector composed of* N $d \times 1$ *vectors stacked, with the* i*th sub-vector* $\mathbf{p}_i$ *corresponding to the position of node* a_i*.*

In keeping with the conception of frameworks as being constructed of inflexible bars and perfect joints, any movement of the framework (a "motion") will preserve the length of the "bars" (edges).

Definition 2.2. *A* motion *of a framework $\mathcal{F} = (G, \mathbf{p})$ in $\mathbb{R}^n$ is a family of embeddings $\mathbf{p}(t)$, $t \in [0, 1]$, such that*

(i) $\mathbf{p}(0) = \mathbf{p}$ *and* $\mathbf{p}(t)$ *is continuous* $\forall t \in [0, 1]$, *and*

(ii) the Euclidean distance $d : \mathbb{R}^n \to \mathbb{R}$ between the endpoints of any edge is constant. That is, $\forall b = (a_i, a_j) \in E$, $\forall t \in [0, 1]$, $d(\mathbf{p}_i(t),\ \mathbf{p}_j(t)) = d(\mathbf{p}_i, \mathbf{p}_j)$.

A rigid motion is then a motion where the relative positions of the vertices is preserved, i.e. the distance between all vertices is preserved, not just those joined by an edge. This is contrasted with a deformation, where the distance between some pair of vertices is different and so the shape is changed. A rigid framework is then one that admits no deformations.

Definition 2.3. *A* rigid motion *of a framework $\mathcal{F} = (G, \mathbf{p})$ in $\mathbb{R}^n$ is a motion where additionally $\forall a_i, a_j \in V$, $\forall t \in [0, 1]$, $d(\mathbf{p}_i(t),\ \mathbf{p}_j(t)) = d(\mathbf{p}_i, \mathbf{p}_j)$. A motion is a* deformation *if this property does not hold. $\mathcal{F}$ is a* rigid framework *if all its allowed motions are rigid.*

The notion of a *rigid component* is useful in understanding how a framework behaves when it is not fully rigid.

Definition 2.4. *A* rigid component *of a framework is a maximal rigid sub-framework. A framework is rigid if it has just one rigid component (the whole framework).*

2.1.2 Infinitesimal Rigidity

There is an almost equivalent concept that turns out to be a lot more useful computationally: infinitesimal rigidity. The idea is to look at the effect that infinitesimal displacements have on the lengths of the edges. Instead of defining motions as trajectories of the vertices, here we look at "velocities" applied to the vertices. We want these infinitesimal velocities to "cancel out" so that the overall effect is that the bars stay the same length. In keeping with the previous pattern of rigidity, infinitesimal rigid motions will maintain this cancelling out for all pairs of vertices, and infinitesimal deformations will change this distance.

Informally, infinitesimal rigidity implies rigidity as if nothing "looks like" it could be the velocity of a motion then there isn't one. The other direction does not follow, however, as there could be what looks like a valid set of velocities that cannot give rise to a proper motion.

Definition 2.5. *An* infinitesimal motion *of a framework $\mathcal{F} = (G = (V, E), \mathbf{p})$ is a function $\mathbf{v} : V \to \mathbb{R}^n$ such that $\forall (a_i, a_j) \in E, (\mathbf{p}_i - \mathbf{p}_j) \cdot (\mathbf{v}_i - \mathbf{v}_j) = 0$, where $\mathbf{v}_i = \mathbf{v}(a_i)$ is the velocity of the ith node. An infinitesimal motion is an* infinitesimal rigid motion *if additionally $\forall (a_i, a_j) \in V \times V, (\mathbf{p}_i - \mathbf{p}_j) \cdot (\mathbf{v}_i - \mathbf{v}_j) = 0$. An infinitesimal motion is an* infinitesimal deformation *if $\exists (a_i, a_j) \in V \times V$ (necessarily not a bond of E) such that $(\mathbf{p}_i - \mathbf{p}_j)(\mathbf{v}_i - \mathbf{v}_j) \neq 0$. A framework $\mathcal{F}$ is* infinitesimally rigid *if it only admits infinitesimal rigid motions.*

These definitions are useful as they allow us to consider linear equations rather than quadratic. The following theorem justifies using infinitesimal rigidity instead.

Theorem 2.6. *Let $\mathcal{F}$ be a framework. If $\mathcal{F}$ is infinitesimally rigid then $\mathcal{F}$ is rigid.*

Proof. This is a non-trivial result and a full proof can be found in [3] for example. □

The following definition of a rigidity matrix is very useful for working with frameworks computationally. We will define it for d dimensions, but will work in two dimensions.

Definition 2.7. *Let $\mathcal{F} = (V, E, \mathbf{p})$ be a d-dimensional framework. The* rigidity matrix *R of $\mathcal{F}$ is an $N_b \times dN$ matrix. R is the matrix representation of the system of equations $\langle \mathbf{p}_j - \mathbf{p}_i, \mathbf{v_j} - \mathbf{v_i} \rangle = 0$, where the $\mathbf{v}_i$s are taken as the variables. That is (recalling that each $\mathbf{v}_i$ has d entries),*

$$R \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \vdots \\ \mathbf{v}_N \end{bmatrix} = \mathbf{0}\,.$$

The ith row corresponds to the ith bond, b_i. In two dimensions, suppose that b_i has nodes a_j and a_k. Then $R_{i,2j-1} = (x_j - x_k)$, $R_{i,2j} = (y_j - y_k)$, $R_{i,2k-1} = (x_k - x_j)$, $R_{i,2k} = (y_k - y_j)$, with zeros elsewhere in the row.

The reason the rigidity matrix is useful is that it encodes information about all the bonds of the network. Two nodes share a bond if and only if a row of the rigidity matrix has non-zero entries in the columns corresponding to their positions. The degree of a node is the number of rows in which the column corresponding to its x- or y-coordinate is non-zero. Row i, corresponding to $b_i = (a_j, a_k)$, can be thought of as consisting of two row vectors, $(\mathbf{p}_j - \mathbf{p}_k)^\top$ and $(\mathbf{p}_k - \mathbf{p}_j)^\top$, padded by zeros. These row vectors "point" from a_k to a_j and from a_j to a_k respectively.

2.2 Linear algebra formalism

Rocks et al. uses a substantial amount of linear algebra that is lightly explained, and we provide more details here. We will use this same formalism throughout. This formalism is specifically designed for working with a framework where each bond is a perfect Hookean spring. We shall often call such objects "spring networks".

2.2.1 Background concepts

There are some results from linear algebra that will come in handy later on. The first of these is the singular value decomposition.

Theorem 2.8 (Singular Value Decomposition (SVD))**.** *Let A be an $m \times n$ matrix with entries from a field F, either $\mathbb{C}$ or $\mathbb{R}$. Then A can be written as*

$$A = U\Sigma V^*$$

where Σ is an $m \times n$ rectangular diagonal matrix (that is, it only has non-zero entries on the main diagonal) with non-negative real numbers on the diagonal, and U and V are $m \times m$ and $n \times n$ unitary matrices, respectively.

The elements of Σ are called the singular values *of A, the columns of U are called the* left singular vectors*, and the columns of V are called the* right singular vectors*. Denoting the ith left and right singular values as $\mathbf{u}_i$ and $\mathbf{v}_i$ respectively, and the ith singular value as σ_i, we have that $A\mathbf{u}_i = \sigma_i \mathbf{v_i}$ and $A^*\mathbf{v_i} = \sigma_i \mathbf{u_i}$.*

Further, if $F = \mathbb{R}$ then $V^* = V^\top$, and U and V are orthogonal matrices.

Proof. See for example [4]. □

The second useful concept is the Moore-Penrose inverse, or pseudoinverse [5]:

Definition 2.9 (Moore-Penrose Generalised Inverse)**.** *Let A be an $m \times n$ matrix with entries from a field F. Then $A^\dagger$ is a pseudoinverse for A if:*

1. $AA^\dagger A = A$
2. $A^\dagger AA^\dagger = A^\dagger$
3. $(AA^\dagger)^* = AA^\dagger$
4. $(A^\dagger A)^* = A^\dagger A$

In other words, $AA^\dagger$ is Hermitian and acts like the identity on the column space of A, and $A^\dagger A$ is also Hermitian and acts like the identity on the column space of $A^\dagger$.

Note that if A is an invertible matrix, its inverse A^{-1} satisfies all of these conditions (as $AA^{-1} = A^{-1}A = I$).

Informally, $A^\dagger$ acts like a real inverse on the image of A sending everything else to zero, and A does the same for $A^\dagger$.

The SVD gives us a convenient way to calculate the pseudoinverse of a matrix.

Theorem 2.10. *Let A be an $m \times n$ matrix with entries from a field F. Then*

$$A^\dagger = V\Sigma^\dagger U^* \,,$$

where $A = U\Sigma V^*$ is the SVD of A. To take the pseudoinverse of Σ, a rectangular diagonal matrix, replace each non-zero element with its reciprocal and transpose.

Proof. See for example [6]. □

We will also need a result about the column-space of matrices multiplied by their transposes for calculating the pseudoinverse as described in Section 4.3.2. In order to prove the required result, we first need some preliminary facts.

Lemma 2.11. *Let A be an $m \times k$ matrix and B be a $k \times n$ matrix, both with entries from a field F, either $\mathbb{R}$ or $\mathbb{C}$. Let $row(A)$ and $col(A)$ denote the row- and column-spaces of A respectively.*

Then $col(AB) = col(A) \iff ker(A) \cap im(B) = \mathbf{0}$.

Proof. Since the columns of AB are linear combinations of the columns of A, we have that $col(AB) \subseteq col(A)$, as any linear combination of the columns of AB will also be a combination of the columns of A.

If one subspace contains another, they are equal if and only if they have the same dimension. Now consider $Z = ker(AB) \subseteq F^n$.

Z consists of vectors $x \in F^n$ such that $Bx = k \in ker(A)$ (as then $ABx = Ak = \mathbf{0}$). Thus $Z = B^{-1}[ker(A)]$, where B^{-1} denotes the preimage rather than the inverse.

Since $r \in ker(B) \implies ABr = A(Br) = A\mathbf{0} = \mathbf{0}$, we have that $ker(B) \subseteq Z$.

By restricting the domain of B to Z and noticing that

$$ker(B) \subseteq Z \implies null(B|Z) = null(B)\,,$$

we get (by the Rank-Nullity Theorem):

$$dim(Z) = rank(B|Z) + null(B|Z) = rank(B|Z) + null(B)\,.$$

Note that $BZ = B(B^{-1}[ker(A)]) = ker(A)$ and so $rank(B|Z) = dim(ker(A) \cap im(B))$. Then

$$\begin{aligned} rank(AB) &= n - null(AB) \\ &= n - dim(Z) \\ &= n - (null(B) + rank(B|Z)) \\ &= n - null(B) - dim(ker(A) \cap im(B)) \\ &= rank(B) - dim(ker(A) \cap im(B)) \,. \end{aligned}$$

Thus if $ker(A) \cap im(B) = \mathbf{0}$, its dimension is 0, and $rank(AB) = rank(B)$. Since $dim(col(A)) = rank(A) = rank(AB) = dim(col(AB))$, we have $col(A) = col(B)$. □

Corollary 2.12. *Let A be an $m \times n$ matrix with entries from $\mathbb{R}$ or $\mathbb{C}$, and let $row(A)$ and $col(A)$ denote the row- column-spaces of A respectively.*

Then $col(A) = col(AA^\top)$.

Proof. By Lemma 2.11, it suffices to show that $ker(A) \cap im(A^\top) = \mathbf{0}$.

Because $A^\top$ is the dual map of A, $im(A^\top)$ is the annihilator of $ker(A)$. Therefore $ker(A) \cap im(A^\top) = \mathbf{0}$ and we are done. □

Finally we are ready to prove the main result.

Theorem 2.13. *Let A be an $m \times n$ matrix with entries from $\mathbb{C}$ and let D be an $n \times n$ diagonal matrix with diagonal entries $d_1, d_2, \ldots, d_n \in \mathbb{C}$. If $d_i \neq 0$ then column i of A is in the column-space of $ADA^\top$.*

Proof. Because we are working over $\mathbb{C}$ and D is diagonal, there is a matrix $D^{\frac{1}{2}}$ such that $D^{\frac{1}{2}} D^{\frac{1}{2}} = D$. $D^{\frac{1}{2}}$ is also diagonal with diagonal entries $\sqrt{d_1}, \sqrt{d_2}, \ldots, \sqrt{d_n} \in \mathbb{C}$. Then we can write $ADA^\top = A(D^{\frac{1}{2}} D^{\frac{1}{2}})A^\top = (AD^{\frac{1}{2}})(D^{\frac{1}{2}} A^\top)$. Note that $(D^{\frac{1}{2}} A^\top) = (AD^{\frac{1}{2}})^\top$ so by Corollary 2.12 we get that $col(ADA^\top) = col(AD^{\frac{1}{2}})$.

When $d_i \neq 0$, the ith column of $AD^{\frac{1}{2}}$ is simply a scaled version of column i in A, and so column i of A is in $col(AD^{\frac{1}{2}}) = col(ADA^\top)$. □

2.2.2 Mechanical formalism

We start by defining how some mechanical terms are used. The units for each quantity are given, but in practice since everything is linearly proportional they are unimportant.

Definition 2.14.

Extension *The change in length of a bond due to some applied strain. We typically use e to denote extension, and it has units of metres.*

(Engineering) Strain *Change in length (extension) divided by the equilibrium length of the bond. The strain on bond b_i is $\epsilon_i = \frac{e_i}{l_i}$ where e_i is the extension and l_i is the equilibrium length of bond b_i. As a ratio of lengths, strain is unitless.*

Tension *The force along a bond due to an applied strain. In the 1-dimensional case, this is simply a scalar. Positive means the bond is being stretched, negative means compressed. We write the tension on bond b_i as t_i. Tension has units of newtons.*

(Net) Force *This is the force experienced by the nodes. It is the restoring force resulting from the extensions of the spring-like bonds and the displacement of the nodes this causes. We write f_i for the force on node a_i. Net force is also measured in newtons.*

Displacement *How much the nodes have been moved from their original positions by some applied strain. We write u_i for displacement of node a_i. Displacement has units of metres.*

The goal is to be able to tune the response of a target pair of nodes by manipulating a pair of source nodes. This is formulated as the ratio of the strain on a "ghost bond" of zero stiffness between the target nodes to the strain on a similar bond between the source nodes. We write this ratio as $\eta = \frac{\epsilon_T}{\epsilon_S}$, where ϵ_T is the target strain and ϵ_S is the source strain.

Calculating the full strain would be far more complicated and require a lot more computational power, and so we are only working to linear order. This means we consider the bond lengths and node positions as fixed, while still working with extensions and displacements. Therefore the strains we are working with are only first-order approximations of the true strains. Later, we evaluate the results using the full response.

We define a cost function on this response η based on the ratio of the actual response to the desired response, η^*, with a special case where the desired response is zero:

$$\Delta^2 = \sum_{j=1}^{n} \begin{cases} (\frac{\eta_j}{\eta_j^*} - 1)^2 & \eta_j^* \neq 0 \\ \eta_j^2 & \eta_j^* = 0 \end{cases},$$

where j indexes the source/target pairs. To minimise this cost, we need to calculate how η varies as we remove each bond from the network.

To follow the convention of the paper (and for readability), we will often use bra-ket notation:

Definition 2.15. *Suppose we are in a vector space V, and $a, b \in V$ are vectors. Then $|b\rangle$ is the column vector b, $\langle a|$ is the conjugate transpose of a, and so $\langle a|b\rangle$ is the standard inner product of a and b and $|a\rangle\langle b|$ is the outer product.*

We define the vectors of bond extensions $|e\rangle$ and of bond tensions $|t\rangle$ in response to the *externally applied strain*. These vectors have length N_b and are defined with respect to the complete orthonormal bond basis, where we identify bond i with the ith standard basis vector, $\mathbf{e}_i$. In bra-ket notation, we write $|i\rangle = \mathbf{e}_i$. In other words, each $|i\rangle$ corresponds to one bond in the network, has length N_b and consists of a 1 in the ith position and 0 everywhere else. We can use these to get the extensions and tensions on any bond:

$$e_i = \langle i|e\rangle$$
$$t_i = \langle i|t\rangle$$

The strain on bond i is $\epsilon_i = \frac{e_i}{l_i}$, where l_i is the equilibrium length of the bond. We model all of the bonds as linear springs, and so they obey Hooke's law (force is proportional to displacement). This is represented by a *flexibility matrix* F, defined by:

$$\langle i|\, F\, |j\rangle = \frac{\delta_{ij}}{k_i}\,,$$

where $k_i = \frac{\lambda_i}{l_i}$ (with λ_i the stiffness of the bond), and δ_{ij} is the Kronecker delta. This is a diagonal matrix with $\frac{1}{k_i}$ along the diagonal.

This matrix relates the tensions and extensions by

$$|e\rangle = F\,|t\rangle\ ,$$

which makes sense as the flexibility matrix encodes how easily each bond is stretched (creating an extension) by a force (in this case tension along the bond), according to the linear relationship described by Hooke's law.

For the tuning algorithm as described later, it is convenient to introduce so-called "ghost bonds". These allow us to examine the extension and strain between two unlinked nodes by having an edge in the graph that does not affect the mechanics of the situation. This is implemented by expanding the square flexibility matrix F, adding a row (and hence column) whose entries are all zero.

We would like to be able to invert F, but the addition of a zero row means it is now singular. We use instead the Moore-Penrose (MP) pseudoinverse $F^\dagger$, defined in section 2.2.1.

We can then relate extensions back to tensions by

$$|t\rangle = F^\dagger |e\rangle \ .$$

This still makes physical sense as a bond of zero stiffness experiences no tension as it is extended.

Another useful pair of vectors to define are those of node displacements $|u\rangle$ and net forces on the nodes $|f\rangle$. These are related to the bond quantities by the equilibrium matrix Q. Q is the transpose of the rigidity matrix R we defined earlier, with each row of R divided by the length of the bond to which it corresponds.

Theorem 2.16. *For a framework with equilibrium matrix Q, then the following equations involving tensions, forces, displacements, and extensions hold:*

1. $Q\,|t\rangle = |f\rangle$

2. $Q^\top |u\rangle = |e\rangle$

Proof.

1. The columns of Q can be thought to each correspond to a bond of our framework. Each pair (in the 2d case) of rows corresponds to the position of a node, the first of the pair for the x coordinate, the second for y.

   Consider a node a_j, and bond $b_i = (a_j, a_k)$. Then row $2j$ contains, in the ith position, the x-coordinate of $\mathbf{p}_j - \mathbf{p}_k$ (call it $x_j - x_k$). If the tension in bond b_i is t_i, then bond b_i has a contribution of $(x_j - x_k)t_i$ to the force in the x direction on node j.

2. The rows of $Q^\top$ (which is a scaled version of R) can be thought of as consisting of a series of row vectors of shape $1 \times d$, where most are $\mathbf{0}$, except for two in each row. For row i, representing bond $b_i = (a_j, a_k)$, the jth vector is $(\mathbf{p}_j - \mathbf{p}_k)/l_i$, and similarly the kth vector is $\mathbf{p}_k - \mathbf{p}_j/l_i$, where $l_i = ||\mathbf{p}_k - \mathbf{p}_j$ is the equilibrium length of bond b_i.

   The displacement vector $|u\rangle$ can also be thought of as consisting of vectors, this time a stack of column vectors of shape $d \times 1$ where the ith vector is

the displacement on node i, $\mathbf{u}_i$. Thus in the product $Q^\top |u\rangle$, the ith element (corresponding to the ith bond) is $(\mathbf{p}_j - \mathbf{p}_k)\mathbf{u}_j + (\mathbf{p}_k - \mathbf{p}_j)\mathbf{u}_k$.

The first term can be thought of as the projection of $\mathbf{u}_i$ onto the vector pointing from $\mathbf{p}_k$ to $\mathbf{p}_j$. Therefore this gives, to linear order, the change in distance between the nodes (and hence the extension of the bond) due to the displacement of node j.

The second term is symmetrically the extension due to the displacement of node k, together giving the extension of bond i, as required.

□

Because we are considering a system of ideal springs, all the energy in the system is due to the stretching of the bonds. The energy in a Hookean spring is $E = \frac{1}{2}kx^2$, where k is its stiffness and x is the extension of the edge. Using the relations described above, we can write this purely in terms of displacement:

$$\begin{aligned}
E &= \sum_{i=1}^{N_b} \tfrac{1}{2} k_i {e_i}^2 \\
&= \tfrac{1}{2} \langle e| F^\dagger |e\rangle && \text{(using } F^\dagger_{i,i} = k_i\text{)} \\
&= \tfrac{1}{2} (Q^\top |u\rangle)^\top F^\dagger (Q^\top |u\rangle) && \text{(using } |e\rangle = Q^\top |u\rangle \text{ and } \langle e| = |e\rangle^\top\text{)} \\
&= \tfrac{1}{2} \langle u| QF^\dagger Q^\top |u\rangle && \text{(as } (AB)^\top = B^\top A^\top\text{)} \\
&= \langle u| H |u\rangle
\end{aligned}$$

where we define the Hessian matrix $H = QF^\dagger Q^\top$. By Hessian, we mean that H is a matrix of the second derivatives of the energy [7].

According to the principle of least action, when tension is applied to the system, the resulting configuration will be that with the lowest energy. With a set of externally applied tensions $|t^*\rangle$, the minimum energy configuration satisfies (to linear order)

$$H |u\rangle = Q |t^*\rangle \ . \tag{1}$$

Expanding $H |u\rangle$ gives

$$H |u\rangle = QF^\dagger Q^\top |u\rangle = QF^\dagger |e\rangle = Q |t\rangle = |f\rangle \ ,$$

so Eq. 1 can be thought of as looking for the node displacements that balance the applied forces.

We have no guarantees that H is invertible, and in general it is not, so we must use the pseudoinverse, $H^\dagger$, if we want to solve for $|u\rangle$. This is justified as we are

multiplying both sides of Eq. 1 by $H^\dagger$, and as $H\,|u\rangle$ is by definition in the image of H, we have $H^\dagger H\,|u\rangle = |u\rangle$. This can be rearranged to give:

$$|u\rangle = H^\dagger Q\,|t^*\rangle$$

and recalling Theorem 2.16, we get the extensions:

$$|e\rangle = Q^\top\,|u\rangle = Q^\top H^\dagger Q\,|t^*\rangle \tag{2}$$

3 Computational model

In order to get an understanding of how the rigidity of a graph changes as you remove edges, we develop a computational model for rigidity in Python to demonstrate applied tensions and resultant stresses, and implement a pebble game [8] to efficiently detect rigid components in graphs.

3.1 Rigid component experiment

The original pebble game algorithm was modified to give a "constructive" pebble game. In the original pebble game, the graph is built up by adding edges one at a time until the whole graph is reconstructed. Instead of repeatedly applying this process to the full graph, then for the graph with one edge removed, and so on, the graph is built up edge by edge with the rigid components amended after each edge is added.

In Fig. 1 and Fig. 2 below, rigid components that consist of more than a single edge are shown in bold black, and single-edge components are in grey. Vertices in more than one component are shown in green, and the edge about to be removed is in blue. The removal of the blue edge has a large effect, breaking even some fairly distant rigid components into pieces.

Fig. 3 shows how the number of rigid components changes as number of edges changes. Read left to right, it shows rigid components forming and joining together as edges are added. Read right to left, it shows the break-up of the rigid framework into rigid components and then the removal of the remaining individual edges. The edge removal in the two earlier figures corresponds to the sharp jump in 3 just before the 200 mark.

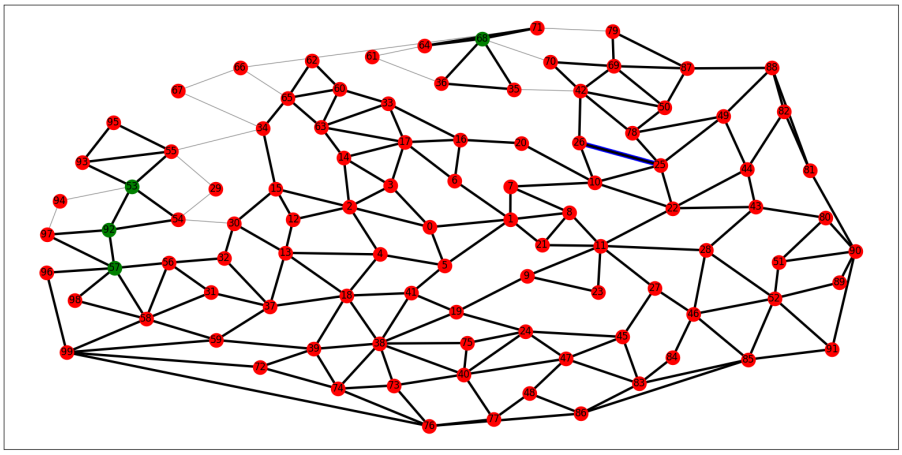


Figure 1: The framework before the removal of the blue edge.

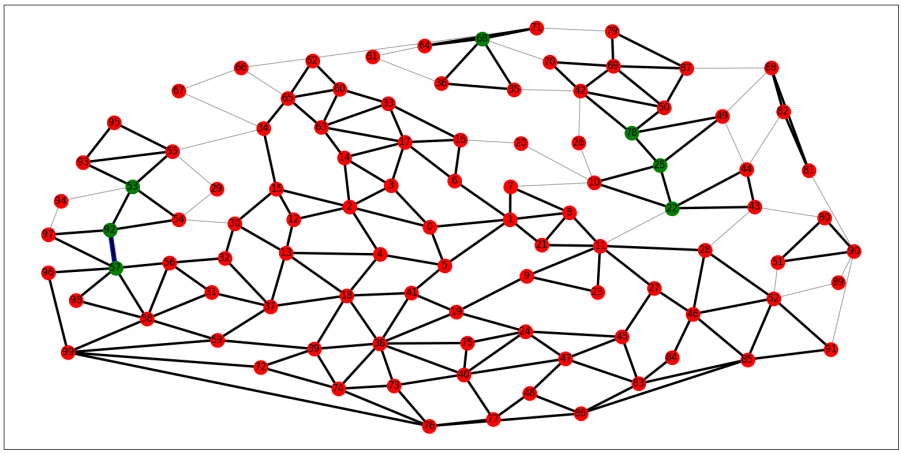


Figure 2: The framework after removal. Note the number of edges that turned grey on the top right.

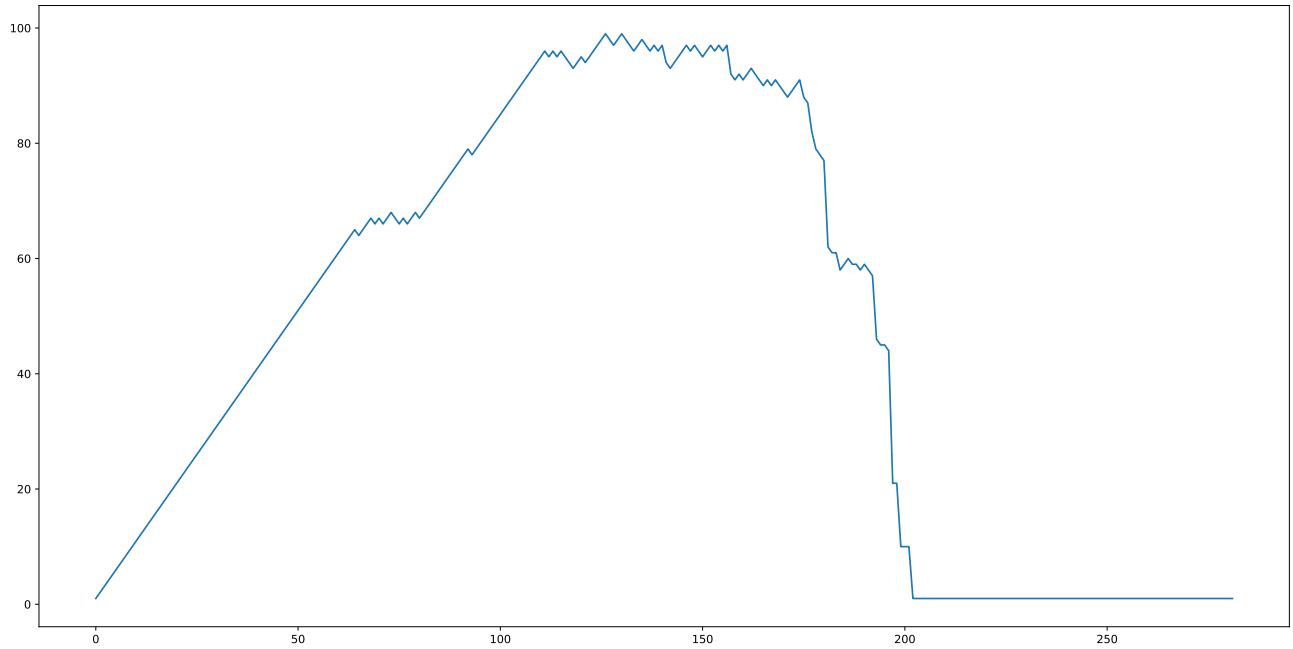


Figure 3: The number of rigid components in the framework from Fig. 1 against the number of edges at that stage.

3.2 Applying forces and tensions

One of the main ideas in this project was applying a force or tension to one part of the network, and having it propagate through the whole system. We develop Python code that allows one to apply forces or tensions to a set of nodes or bonds, respectively, and visualise the stresses that arose in the bonds as a result. In practice we exclusively work with applied tensions, as this allows us to stress a particular bond. Fig. 4 shows an example of the plots produced.

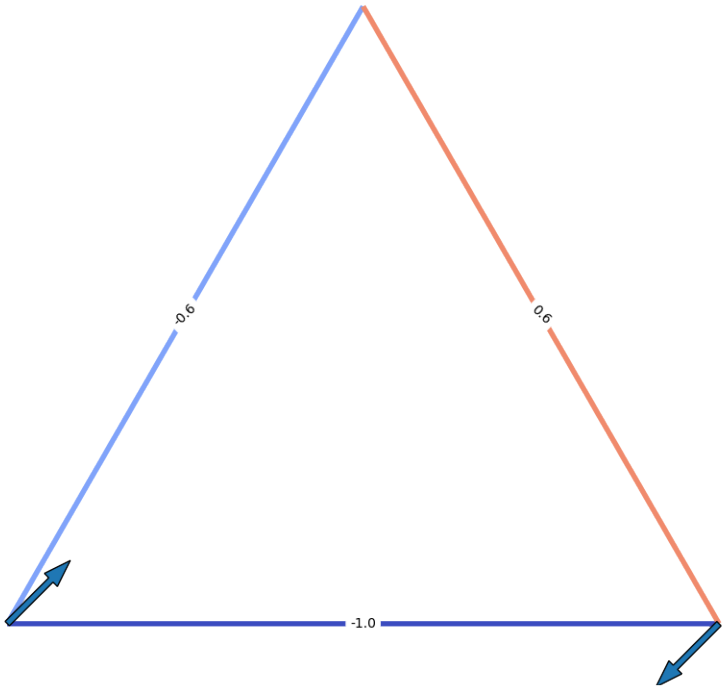


Figure 4: Stresses on a triangular framework. The arrows indicate the direction and relative magnitude of applied force. Negative numbers (blue) correspond to compression of the edge and positive numbers (red) to stretching.

3.3 Animation

To visualise the full non-linear response of the network to an applied strain, we minimise the configurational (potential) energy as described in the *Supporting Information* for Rocks et al. This minimisation is performed subject to the constraint that the strain on the source bond is a specified value, and we vary this value to build up frames of the animation.

However, there appears to be a typographical error in their Eq. S24. Because we are working with a network of springs, the potential energy is proportional to the square of the change in length of the spring. Their equation omits the square, and so should read:

$$E = \sum_{\langle i,j \rangle} k_{ij}(X_{ij} - l_{ij})^2$$

We minimise this corrected energy using a built-in SciPy [9] algorithm. This re-

turns a set of node displacements that minimises the energy. We can use these to calculate the non-linear strain on the target bond and hence the non-linear strain ratio. We also use these to draw the deformation of the network under each given strain and compile them into an animation demonstrating the response of the network. Examples of such animations can be found at `irmckenzie.co.uk/masters/animations`. The source bond is indicated in green, and the target in red.

4 Developing a tuning algorithm

4.1 Network generation

In generating their starting random networks, Rocks et al. used the contact graph of jammed packings of soft spheres. Such packings consist of overlapping soft spheres that have been brought to a local energy minimum. They chose this set-up because such networks have material properties that are well-understood. The fact that these networks start at a local energy minimum also justifies taking the minimum energy configuration to be the response of the network to applied tension.

We instead make our networks using a three step process:

(i) We implement a Poisson disc sampling algorithm [10] to generate N node positions.

(ii) We add all the edges from a Delaunay triangulation of the node positions. This was done using an in-built SciPy package.

(iii) We prune edges until we have $2N$ remaining, which is just above $2N - 3$, the minimum number for a rigid graph.

First, long edges are removed, defined as those with lengths greater than twice the node spacing.

Then we remove edges at random, as long as the graph continues to satisfy two conditions described in [11]:

(a) For each node, we do not remove any edges if this would mean all its edges were on the same "side" (i.e. the node must stay in the convex hull of its neighbours). Note that nodes on the periphery always have all their edges on the same side, so we remove no edges from them.

(b) No edge should be removed that would create a node of degree less than 3.

These Delaunay networks were chosen because they are in some ways similar to jammed packings [11] [12] but are much easier to generate and more rigorously defined. However, our networks do not share all the nice properties of sphere packings, and we wanted to see if this would have any effect on the success of the algorithm.

4.2 General approach

The general approach taken to tuning a network for a given strain ratio is as follows:

(i) Remove each edge in turn from the network.

(ii) Compute the extensions, and thus the strains, resulting from removing that edge from the network.

(iii) Calculate the cost Δ^2 associated with these strains.

(iv) Remove the bond that resulted in the lowest cost.

(v) Repeat steps (i) - (iv) until the cost is below some threshold (taken here to be $\sqrt{\Delta^2} < 0.01$).

This is a greedy algorithm - at each step we look only one step ahead. Therefore we have no guarantees that it will find the best solution (i.e. fewest edge removals), or even that it will terminate at all. There are three methods by which the algorithm could fail: a) after removing all possible edges the cost is still above the threshold, b) no bond can be removed without creating a zero mode, or c) the relative numerical error exceeds 0.01%. This error is calculated by recomputing the cost on a network where this edge has been removed (effectively comparing to the brute force approach defined below in Section 4.3.1). The remarkable result of the Rocks et al. paper is that this simple approach did work reliably, and with very few edge removals.

The most expensive step, and consequently the one to which most attention is given, is step (ii). Here we have to solve Eq. 2, which involves finding the pseudoinverse of a matrix. Finding the pseudoinverse is an expensive operation: it is of the order $\mathcal{O}((dN)^2)$, where n in our case is dN. This is done numerically using the SVD.

We have no guarantees that H is invertible, and in general it is not, so we must use the pseudoinverse, $H^\dagger$. This is justified as we are multiplying both sides of Eq. 1 by $H^\dagger$, and as $H\,|u\rangle$ is by definition in the image of H, we have $H^\dagger H\,|u\rangle = |u\rangle$.

4.3 Implementation

There are various ways to approach step (ii). The three methods we used are outlined in the following sections.

4.3.1 Brute force (BF)

The most straightforward approach is to simply recalculate $H^\dagger$ after the trial removal of each edge. This comes with the benefit of being (relatively) easy to implement, and so was the first approach tried. It does, however, have the downside of a time complexity of $\mathcal{O}((dN)^2)$ per step, which gets expensive quickly as the size of the graph increases.

This can be reduced by noticing that

$$H^\top = (QF^\dagger Q^\top)^\top = (Q^\top)^\top (F^\dagger)^\top Q^\top = QF^\dagger Q^\top = H\,.$$

Therefore H is symmetric and – because we are working only with reals – equal to its own conjugate transpose, and hence Hermitian. This allows us to use a more efficient method for calculating the pseudoinverse built in to SciPy [9].

4.3.2 Sherman-Morrison updating (SM)

Rather than recalculating the entire $H^\dagger$ matrix for each edge removal, we can instead update it incrementally using a modified version of the Sherman-Morrison (SM) formula [13]. The SM formula provides a method for computing the inverse of the sum of an invertible matrix A and the outer product of two vectors, $\mathbf{u}\mathbf{v}^* = |u\rangle \langle v|$. We are only working with $\mathbb{R}$, so $\mathbf{v}^* = \mathbf{v}^\top$. A rank 1 update takes place when one replaces a matrix A with $\widetilde{A} = A + B$, where $rank(B) = 1$. Note that the outer product of two vectors is a matrix with rank 1, and that any rank 1 matrix can be written as an outer product of two vectors, so we can say that A has undergone a rank 1 update. We can frame our problem in terms of a rank 1 update to H.
Suppose we have $H = QF^\dagger Q^\top$ for a framework $\mathcal{F} = (G = (V, E), \mathbf{p})$, and suppose we set the stiffness for a bond b_i to zero (effectively removing it, in our formalism). We wish to calculate the Hessian matrix $\widetilde{H}$ for this new system. Recall that $\langle i| F |j\rangle = \frac{\delta_{ij}}{k_i}$ and so the flexibility matrix for the new system is

$$\widetilde{F} = F - \frac{1}{k_i} |i\rangle \langle i|\ .$$

Taking the pseudoinverse of F and $\widetilde{F}$ gives

$$\widetilde{F}^\dagger = F^\dagger - k_i |i\rangle \langle i|\ .$$

The equilibrium matrix Q is unchanged, so

$$\begin{aligned}\widetilde{H} &= Q\widetilde{F}^{\dagger}Q^{\top} \\ &= Q(F^{\dagger} - k_i \left|i\right\rangle \left\langle i\right|)Q^{\top} \\ &= QF^{\dagger}Q^{\top} - k_i Q(\left|i\right\rangle \left\langle i\right|)Q^{\top} \\ &= H + (-k_i Q \left|i\right\rangle \left\langle i\right| Q^{\top}) .\end{aligned}$$

We can then take $\boldsymbol{u} = -k_i Q \left|i\right\rangle$ and $\boldsymbol{v}^{\top} = \left\langle i\right| Q^{\top}$. Note that $\boldsymbol{u}$ is a constant times the ith column of Q, and $\boldsymbol{v}^{\top}$ is the ith row of $Q^{\top}$, which is again the ith column of Q.

We cannot directly employ the SM formula as it is specifically for invertible matrices. Meyer [14] introduces six generalisations of the SM formula that deal with all possible cases for updating the pseudoinverse.

Meyer sets out three important criteria, which we express in terms of our current example: whether $\mathbf{u}$ is in the column-space of H, whether $\mathbf{v}^{\top}$ is in the column-space of $H^{\top}$ (which is the same as whether $\mathbf{v}$ is in the column-space of H), and whether $\beta = 1 + \mathbf{v}^{\top}H^{\dagger}\mathbf{u} = 0$.

As previously identified, $\boldsymbol{u} \in col(Q)$ and $\boldsymbol{v} \in col(Q)$. By taking $A = Q$ and $D = F^{\dagger}$, Theorem 2.13 gives that $\boldsymbol{u}$ and $\boldsymbol{v}$ are in the column-space of H as long as the corresponding entry in $F^{\dagger}$ is non-zero. This is the case for all non-ghost bonds in the network, which are the only ones we will be testing for removal anyway. Therefore we will in practice only be using Theorem 3 or Theorem 6 from [14]. This theoretical result is important because $\boldsymbol{u}$ and $\boldsymbol{v}$ are often slightly outside the column-space due to numerical errors. We obtain better results by assuming they are always inside, which is justified by Theorem 2.13.

4.3.3 Green's function method (GF)

The method used by Rocks et al. involves calculating a discrete Green's function [15] to map tensions to extensions. They start by taking the SVD of Q and looking at its right singular vectors. These form a basis for the column-space of Q. Recall that $Q = R^{\top}$, so the column-space of Q is the row-space of R. Each row of R corresponds to a bond in the network, so this basis still accounts for all of the bonds. Each basis vector is essentially a linear combination of the bonds in the network. This basis can be split into two subbases, one of singular vectors with a singular value of 0, and the other of singular vectors with a non-zero singular value.

The vectors for this first subbasis are called the *states of self stress (SSS)*, each denoted $\left|s_{\beta}\right\rangle$, where β indexes the vectors. This is because tension on the bonds according to the coefficients of an SSS vector (or a linear combination of them) results in no net force on the nodes. Therefore the network is still in equilibrium (all tensions

balanced) without an external applied force. What this means is that if one were to consider a tension whose magnitude on each bond of the network was simply the corresponding entry of an SSS vector, there would be no resulting net displacement of the nodes.

The second subbasis is called the *states of compatible stress (SCS)* with vectors $|c_\alpha\rangle$, where α indexes the vectors. Tensions applied according to these vectors *would* result in net forces on the nodes.

We denote the size of the SSS and SCS bases N_s and N_c respectively, and note that $N_b = N_c + N_s$.

Rocks et al. go through a long derivation in their *Supplementary Information* to find the change in extensions when removing a bond which we largely omit, but there are some points of interest:

Identical bond stiffnesses First, they define an equivalent network where the bonds are rescaled so that the stiffness $k_i = \frac{\lambda_i}{l_i}$ is identical for all non-ghost bonds. By doing so they simplify the algebra and avoid having to take an expensive matrix inverse. This is done by scaling Q to get $\bar{Q} = QF'^{-\frac{1}{2}}$, where we have defined the diagonal $N_b \times N_b$ matrix F' such that

$$F'_{i,i} = \begin{cases} F_{i,i} & F_{i,i} \neq 0 \\ 1 & F_{i,i} = 0 \end{cases},$$

because we must treat ghost bonds separately. Leaving the 0 entries in F' kills all information about ghost bonds in $\bar{Q}$. Since we use the same λ throughout, we end up with all non-ghost bonds having stiffness $k = \lambda$. After rescaling, we can use a modified flexibility matrix $\bar{F}$ whose diagonal entries are k for regular bonds and 0 for ghost bonds. They write the energy for this scaled system as

$$E = \frac{1}{2} \langle u | \bar{Q} \bar{F}^\dagger \bar{Q}^\top | u \rangle ,$$

and say that this is the same as the unscaled energy. It is unclear as to whether they mean that this energy gives literally the same values for each input displacement, or that it is proportional and so has minima in the same places. In practice we find that the latter is the case. The extensions and tensions are related to their scaled versions by $|\bar{e}\rangle = F'^{-\frac{1}{2}} |e\rangle$ and $|\bar{t}\rangle = F'^{\frac{1}{2}} |t\rangle$ respectively.

Unique SCS vector They define a "unique SCS vector" for each bond, defined as $|C_i\rangle = kG|i\rangle$, where

$$G = \frac{1}{k} \sum_\alpha |c_\alpha\rangle$$

is the discrete Green's function. This is the sum of the projection of $|i\rangle$ onto each SCS basis vector. By rotating the basis such that one of the vectors lines up with $|C_i\rangle / C_i^2$, we end up with an orthonormal basis where only one vector has a non-zero $|i\rangle$ projection.

Modifying multiple bonds When testing the change in extensions from removing an edge, we typically have three edges whose stiffnesses aren't k: the source and target ghost bonds, and the edge in question. This requires treating this set of edges separately, which we call $\mathcal{B}$. This is done by finding the unique SCS vector for each edge in $\mathcal{B}$ and orthonormalising these using the Gram-Schmidt process, giving a set $\mathcal{V}$. The rest of the SCS basis with respect to the orthonormal set $\mathcal{V}$ to give an orthonormal basis $|\widetilde{c}_\alpha\rangle$ for the SCS where only the vectors in $\mathcal{V}$ have non-zero projection onto the bonds in $\mathcal{B}$. In practice the change in extensions only relies on the set $\mathcal{V}$, so the full rotated basis is only used when calculating the starting extensions.

Zero modes Rocks et al. add the condition that they do not wish to introduce any zero-energy modes. Informally, these are portions of the network that can take on many different configurations without affecting the energy. For example, if bond $b_i = (a_j, a_k)$ is a pendant edge in the graph, then its pendant node a_k can take on any position along a circle of distance l_i from a_k without affecting the energy. By Maxwell-Calladine counting [16], $N_0 - N_s = dN - N_b$ where N_0 is the number of zero modes. Since removing an edge decreases N_b by one and dN stays constant, either N_0 increases by one or N_s decreases by one. As long as the bond that is removed has non-zero projection onto a vector of the SSS basis, an SSS vector is removed when that bond is removed.

We therefore check before removing bond i that $S_i^2 = \langle S_i | S_i \rangle > 0$, where

$$S_i = \sum_\beta |s_\beta\rangle \langle s_\beta | i\rangle \ .$$

We do this in all three implementations, not just the Green's function version, for ease of comparison.

Extension formula Their derivation culminates in Eq. **S20** which we reproduce here:

$$|\Delta e\rangle = \sum_{\alpha,\alpha' \in \mathcal{V}} |\widetilde{c}_\alpha\rangle \, \Delta(\widetilde{K}^{-1}_{\alpha\alpha'}) \, \langle \widetilde{c}_{\alpha'} | t^* \rangle \ .$$

It is this equation that we use to calculate the cost associated with removing each edge. We first calculate the starting extension with all edges present. We then calculate the change in the extensions for each edge in succession, which (after transforming the

extensions back to their unscaled versions) allows us to calculate the linear strains and hence costs.

5 Results of tuning algorithms

Our results are an interesting mix of successes and unexplained behaviour. In this section we describe these successes and the things that did not go quite as expected.

5.1 Overall performance

Table 1: Running time on the graph described in Section 5.2 below. Each algorithm removed six edges.

Algorithm	Time taken (s)
Sherman-Morrison	31.0330
Green's Function	76.4417
Brute Force	90.2633

When working properly, all three methods performed almost identically, always removing the same edges in sequence unless two edges were associated with nearly exactly the same cost in which case numerical differences could cause variation. GF was most prone to failure, being especially sensitive to scaling of the input positions. The reason for this behaviour is unclear, but meant that in practice we most often used SM.

In terms of time taken, SM was the fastest, followed by GF and then BF (see Table 1). In principle there is no reason for SM to be faster than GF as both skip expensive matrix inversion. It possibly comes down to details of our implementation, although we suspect it also has to do with SM utilising highly optimised linear algebra algorithms built in to NumPy.

All of the methods purported to do suspiciously well on networks generated as in Section 4.1, often bringing the cost down to below the threshold by a large margin after removing only two bonds, regardless of the network size. The tuned networks really did exhibit the desired linear strain ratio, but the full non-linear response was not always as convincing.

The networks sometimes failed to achieve the same magnitude of full response as they did in the linear regime (see Fig. 5), and sometimes the strain ratio even failed

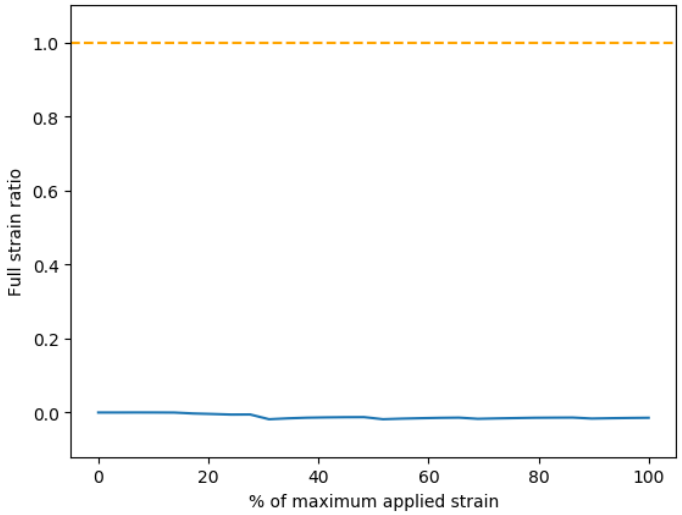


Figure 5: A network generated as in Section 4.1 whose linear response was tuned to 1. The non-linear response is underwhelming.

to exhibit the same sign. This could have been due to the way we implemented the algorithms, or due to the networks and source/target pairs we chose to experiment with. We tried the algorithm on other types of networks in order to find out.

5.2 Recreating a graph from Rocks et al.

To compare our results to those obtained by Rocks et al., we manually transcribed the graph from Fig. 1 of their paper, which is shown in our Fig. 6 below. Removing the six edges that they did when aiming for a strain ratio of 1.0, we get a linear strain ratio whose cost is below the specified cost threshold. This indicates that our method for calculating linear strain is working as intended. When our tuning algorithm was applied to the same network it also removed six edges, the first three of which were among those removed by the algorithm used in the paper. Where our algorithm diverges from theirs, an edge they removed appeared among the ten lowest cost edges, suggesting that it was numerical differences that led to different bond removals. This seems highly plausible as the graphs were recreated by hand and therefore bound to be somewhat imprecise.

Nevertheless, when aiming for a strain ratio of -1, SM removed exactly the same edges as in the paper.

The most striking result from this recreation is that the choice of the source/target pair seems to be quite important. When different pairs of source and target were

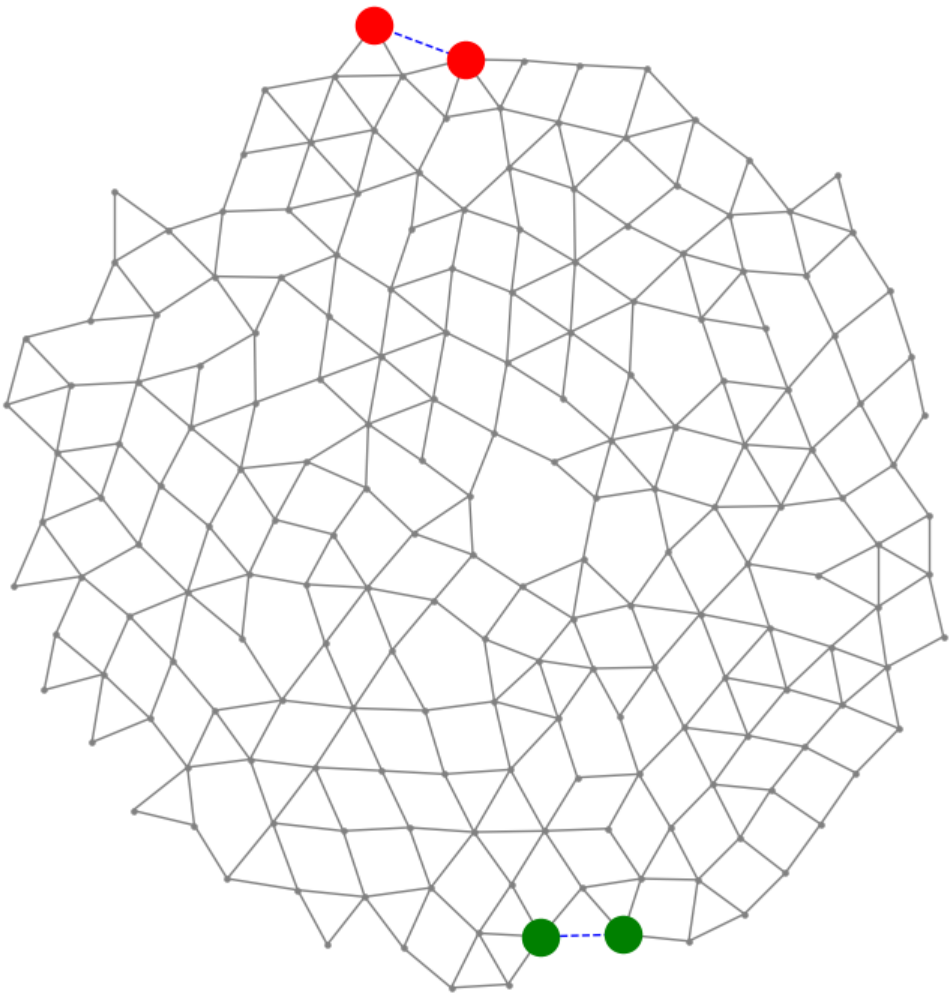
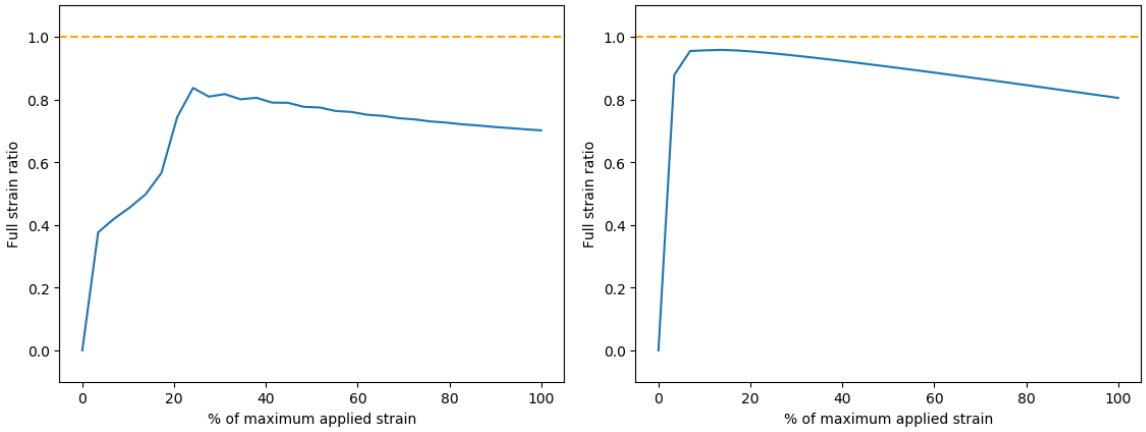


Figure 6: Fig. 1 from Rocks et al. recreated by hand and reproduced here. The source pair (where the strain is applied across) is drawn in green, and the target is in red. The ghost bonds between source nodes and target nodes respectively are both dashed blue.



(a) Our tuned network

(b) Tuned network from the paper

Figure 7: Non-linear responses for Fig. 1 from Rocks et al. When aiming for a strain ratio of 1.0, a) shows how the non-linear strain ratio varies when we tune the network is tuned with SM, and b) shows how it varies when the edges are removed as in the paper.

chosen for the same graph, the algorithm removed eight edges before terminating, and the non-linear response overshot the target, rather unusually (see Fig. 8).

5.3 Tests on ordered networks

Rocks et al. also demonstrate tuning on ordered lattices. We implemented similar lattices and found that the tuning algorithm did not jump almost immediately to success here either. Instead, 28 edges were removed, and the non-linear response was closer to the linear response than in typical disordered examples (see Fig. 9). This, combined with the previous example of the importance of the initial network, suggests that the method of network generation has a large bearing on the success of this method to tune the non-linear response.

5.4 Creating animations

The largest bottleneck in running these experiments is not the tuning but the process of creating animations. The tuning algorithm typically takes less than five minutes, even on reasonably large networks with roughly 200 nodes and 400 edges. On the same network, creating 30 frames of animation takes hours. The function to be optimised (the full configurational energy) is of the order $\mathcal{O}(N_b)$ and we are optimising over dN variables (each coordinate of each node position). Each round of optimisation

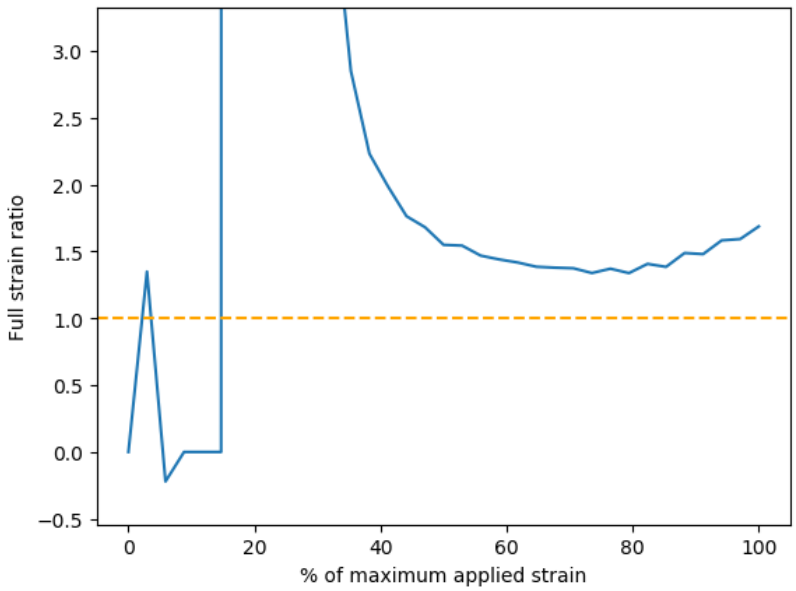


Figure 8: The progression of the non-linear strain ratio for Fig. 1 from Rocks et al. with different source/target. Note that the strange behaviour at the start of the graph – including the huge spike to over 600 – is likely due to issues with the optimisation failing to fully converge. The animation is nowhere near as erratic.

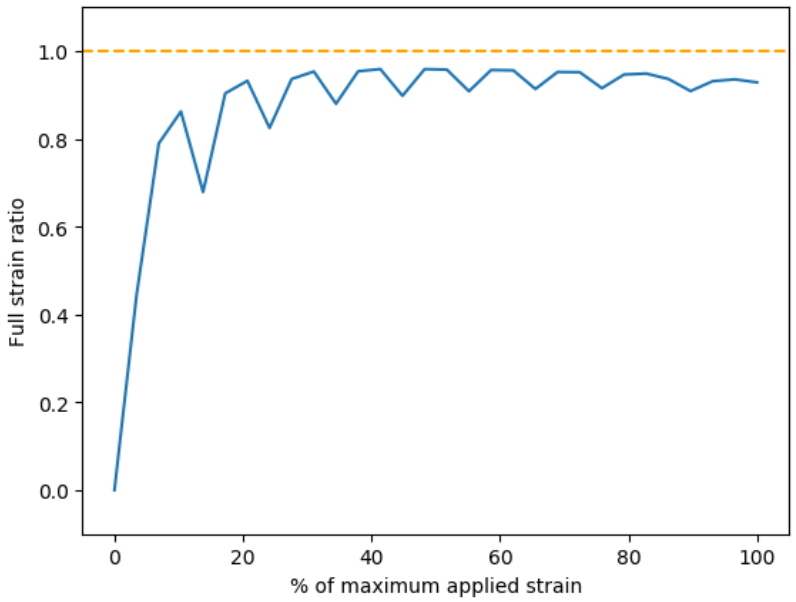


Figure 9: Non-linear strain ratio for the Delaunay triangulation of a square lattice.

is capped at 101 iterations, and often reaches this value, meaning it has not found the true energy minimum. We draw 30 frames, which corresponds to 30 rounds of optimisation.

Unfortunately this is the ultimate test of whether the network exhibits an allosteric response, not only for visualisation but because the optimisation provides the non-linear response. Since Rocks et al. used networks with 200 bonds, in order to compare like with like we must do the same. As animations cannot be included in this document, they have been uploaded to `irmckenzie.co.uk/masters/animations`. Due to the relatively low number of frames generated and the fact that the optimisation often does not successfully complete, the animations can be a bit choppy. They convey the general trend quite well though. The range of motion is kept deliberately small as the network starts to crumple past a certain point when the source nodes are stretched too far apart for the network to handle.

6 Conclusion

In this project we surveyed the theory of frameworks, an interesting topic that is not often seen in undergraduate courses. We established a theoretical and computational framework (no pun intended) on which to build. We then used this to replicate the results of a recent paper, achieving good results in some instances and highlighting what is still not understood about others.

This is an exciting field of research and there is much more work to be done. Given more time, it would have been useful to collect statistics on the behaviour of the algorithms on large numbers of networks. This could help in figuring out how the linear response is related to the non-linear response, and in exactly which circumstances this relationship breaks down. It would also have been useful to have developed a sphere packing algorithm so that we could test directly on networks generated in that way.

Another avenue would be to see how "nice" a network has to be to have its linear response be easily tunable. Are Delaunay triangulations of Poisson disk sampling and jammed sphere packing contact graphs special, or can even more disordered and random networks be tuned? There is much more to be learned about these seemingly simple arrangements of nodes and bonds.

References

[1] Jason W. Rocks, Nidhi Pashine, Irmgard Bischofberger, Carl P. Goodrich, Andrea J. Liu, and Sidney R. Nagel. Designing allostery-inspired response in mechanical networks, Mar 2017.

[2] J. E. Graver. *Counting on Frameworks: Mathematics to Aid the Design of Rigid Structures*. The Mathematical Association of America, 2001.

[3] B. Roth and Walter Whiteley. Tensegrity frameworks. *Transactions of The American Mathematical Society - TRANS AMER MATH SOC*, 265, 02 1981.

[4] G. Strang. *Linear Algebra and Its Applications*. Thomson, Brooks/Cole, 2006. Excerpt available from MIT: http://tiny.cc/svdMIT.

[5] R. Penrose. A generalized inverse for matrices. *Mathematical Proceedings of the Cambridge Philosophical Society*, 51(3):406–413, 1955.

[6] A. Ben-Israel and T.N.E. Greville. *Generalized Inverses: Theory and Applications*. CMS Books in Mathematics. Springer New York, 2006.

[7] R. Connelly and Walter Whiteley. Second-order rigidity and prestress stability for tensegrity frameworks. *SIAM J. Discrete Math.*, 9:453–491, 08 1996.

[8] Audrey Lee and Ileana Streinu. Pebble game algorithms and sparse graphs. *Discrete Mathematics*, 308(8):1425–1437, April 2008.

[9] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[10] Robert Bridson. Fast poisson disk sampling in arbitrary dimensions. In *ACM SIGGRAPH 2007 Sketches*, SIGGRAPH '07, page 22–es, New York, NY, USA, 2007. Association for Computing Machinery.

[11] Varda Hagh and Michael Thorpe. Disordered auxetic networks with no reentrant polygons. *Physical Review B*, 98, 09 2018.

[12] Varda F. Hagh, Eric I. Corwin, Kenneth Stephenson, and M. F. Thorpe. Jamming in perspective, 2018.

[13] Jack Sherman and Winifred J. Morrison. Adjustment of an inverse matrix corresponding to a change in one element of a given matrix. *The Annals of Mathematical Statistics*, 21(1):124–127, 1950.

[14] Carl D. Meyer. Generalized inversion of modified matrices. *SIAM Journal on Applied Mathematics*, 24(3):315–323, 1973.

[15] Fan Chung and S.-T. Yau. Discrete green's functions. *Journal of Combinatorial Theory, Series A*, 91(1):191 – 214, 2000.

[16] C.R. Calladine. Buckminster fuller's "tensegrity" structures and clerk maxwell's rules for the construction of stiff frames. *International Journal of Solids and Structures*, 14(2):161 – 172, 1978.