Low Energy Band-Limited Surfaces

Stephen Jasina
Advised by Ian Tobasco

July 18, 2019

Abstract

We investigate surfaces with low energies as defined by elasticity theory. For some simplification, we restrict our attention to the height functions $w : \mathbb{R}^2 \to \mathbb{R}$ of band-limited double Fourier series with fundamental domain $\mathbb{T}^2 = [0, 1] \times [0, 1]$. We minimize the quantity

$$E_{\text{stretch}} + \left| \frac{1}{2} \int_{\mathbb{T}^2} \nabla w \otimes \nabla w - M \right|^2 + h^2 E_{\text{bend}} + \frac{1}{h^p} \int_{\mathbb{T}^2} w^2 \tag{1}$$

(where M is some constant matrix) to apply a compressive constraint to our surface. A computer program is made to find approximate local minimizers of this energy functional, and some plots are presented.

1 Motivation

We begin by presenting a theorem and an outline of its proof due to Stoker [1]:

Theorem 1. *Any developable surface S parameterized by $X(u, v)$ with continuous second derivatives is necessarily a cylinder provided that any straight line in the surface contains no boundary points of the surface.*

Outline of Proof. By hypothesis, S has a constant curvature of 0. A point in S is called *planar* if every normal section of S passing through the point has zero curvature. Otherwise, the point is said to be *parabolic*.

Suppose p_0 is a parabolic point. Locally, lines of curvature can act as parameter curves. Assume that the level curves of u are the lines of curvature for which $k_2 = 0$ and that the level curves of v are the lines of curvature for which $k_1 \neq 0$. Assume also that u and v are arc-length parameterized. Denote the normal to the surface as X_3. Some calculations show that X_{vv} is perpendicular to each of X_u, X_v, and X_3. In particular, we must have $X_{vv} = 0$. From here we have that the asymptotic lines of S are straight, and thus we can write $X(u, v) = Y(u) + vZ(u)$ for suitable functions Y and Z in a neighborhood of p_0.

The next step is to prove a lemma. Some terminology is first required. An *essentially parabolic point* of S is a limit point of parabolic points. A *generator* is a straight line segment in S on which the normal to S is constant. A line segment here is allowed extend infinitely in one or both directions.

Lemma 1. *Suppose p_0 is an essentially parabolic point of S.*

1. *p_0 is contained in a unique generator $\ell = \ell(p_0)$ whose endpoints (if they exist) lie on the boundary of S.*
2. *All points on $\ell(p_0)$ are essentially parabolic.*
3. *The points of $\ell(p_0)$ are either all parabolic or all planar.*

A full proof of this lemma is omitted, but can be found in Stoker [1]. That being said, the general strategy for a parabolic point p_0 is to construct a generator by starting with a line segment passing through p_0 (from above) and then extending it repeatedly, either ad infinitum or until the extension reaches the boundary of S. In the case that p_0 is planar, then, since p_0 is essentially parabolic, we can consider the limit of parabolic

points p_n. The limit of the lines $\ell(p_n)$ yields $\ell(p_0)$. The second and third parts of the lemma follow from the intermediate work.

The final step is to show that, under the assumptions of the theorem, through every point of S, there exists a unique generator that extends infinitely far in both directions (unless, of course, S is a plane, in which case it is clearly a cylinder). The strategy here is to show that every generator of S is contained in a neighborhood of generators that are all parallel to each other. From here, we use continuous induction. In the full proof, a distinction must be made here between *flat points* (points contained in a neighborhood of completely planar points) and other points.

This final fact yields the theorem. □

For this REU, we draw inspiration from a procedure conducted by Chen and Hutchinson [2]. A thin gold foil is deposited uniformly onto a flat silicone substrate. The system is then cooled, which causes equi-biaxial compression, thus crumpling the foil. This crumpling intuitively should not stretch the foil much. From Theorem 1, one consequently might expect the crumpling to form approximate cylinders. Chen and Hutchinson found, instead, that the foil forms a herringbone pattern. The motivation for this investigation thus comes from the disparity between our expectations and the physical results of the experiment.

So as not to require as much time and as many resources as Chen and Hutchinson, we use a computer model based on the von Kármán approximations of the elastic energy of the foil. In particular, we assume that a doubly periodic Fourier series can approximate the surface of the foil. We further assume that this Fourier series is band-limited. The first assumption is justified by assuming the block is very large, which would effectively allow us to ignore the foil near the edges. The second assumption will be explored more in Section 4.

2 Preliminary Definitions and Notation

A Fourier series

$$\begin{aligned} w : \mathbb{R}^2 &\to \mathbb{R} \\ x &\mapsto \sum_{k \in \mathbb{Z}^2} a_k e^{2\pi i k \cdot x} \end{aligned}$$

is said to be supported in the band

$$A_{\rho_1,\rho_2} := \left\{ k \in \mathbb{Z}^2 : \rho_1 \leq |k| \leq \rho_2 \right\}$$

if $a_k = 0$ when $k \notin A_{\rho_1,\rho_2}$.

We define

$$M := \begin{pmatrix} M_{11} & M_{12} \\ M_{12} & M_{22} \end{pmatrix}$$

to be a constant symmetric matrix.

The stretching energy and bending energies of w are defined as

$$E_{\text{stretch}} = \int_{\mathbb{T}^2} \left| |\nabla|^{-2} \det \nabla\nabla w \right|^2$$

and

$$E_{\text{bend}} = \int_{\mathbb{T}^2} |\nabla\nabla w|^2 .$$

We also define

$$E_M = \left| \frac{1}{2} \int_{\mathbb{T}^2} \nabla w \otimes \nabla w - M \right|^2$$

and

$$E_{\text{height}} = \int_{\mathbb{T}^2} w^2.$$

Combining these all together, we define

$$E_{M,h,p} = E_{\text{stretch}} + E_M + h^2 E_{\text{bend}} + \frac{1}{h^p} E_{\text{height}}.$$

Of particular interest is the case $p = 1$ and $M = I$, which correspond to the physical experiment described in Section 1.

Finally, for a truth value B, we use the notation $[\![B]\!]$ to mean the value 1 when B is true and 0 when B is false.

3 Generating Random Low Energy Band-Limited Surfaces

3.1 Representing Fourier Series in Code

In this section, we will write

$$\begin{aligned}
\begin{pmatrix} k_+^+ \end{pmatrix} &= \begin{pmatrix} k_1 \\ k_2 \end{pmatrix}, \\
\begin{pmatrix} k_-^+ \end{pmatrix} &= \begin{pmatrix} k_1 \\ -k_2 \end{pmatrix}, \\
\begin{pmatrix} k_+^- \end{pmatrix} &= \begin{pmatrix} -k_1 \\ k_2 \end{pmatrix},
\end{aligned}$$

and

$$\begin{pmatrix} k_-^- \end{pmatrix} = \begin{pmatrix} -k_1 \\ -k_2 \end{pmatrix}$$

for a vector

$$k = \begin{pmatrix} k_1 \\ k_2 \end{pmatrix} \in \mathbb{Z}^2.$$

We represent a Fourier series as a mapping from pairs of integers $k = (k_1, k_2)^T$ to complex numbers a_k. Since we are only interested in band-limited series, the sum

$$w(x) = \sum_{k \in \mathbb{Z}^2} a_k e^{2\pi i k \cdot x} = \sum_{k \in A_{\rho_1, \rho_2}} a_k e^{2\pi i k \cdot x}$$

is finite, so we only need to store a finite number of a_k's in our program. For ease of storage, we choose to use an `unordered_map` in C++.

Note that we want w to be a real-valued function, so we must impose some constraints on what the a_k's can be. Well,

$$\begin{aligned}
w(x) &= \overline{w(x)} \\
\sum_{k \in \mathbb{Z}^2} a_k e^{2\pi i k \cdot x} &= \overline{\sum_{k \in \mathbb{Z}^2} a_k e^{2\pi i k \cdot x}} \\
&= \sum_{k \in \mathbb{Z}^2} \overline{a_k} e^{-2\pi i k \cdot x} \\
&= \sum_{k \in \mathbb{Z}^2} \overline{a_{-k}} e^{2\pi i k \cdot x} \\
a_k &= \overline{a_{-k}}.
\end{aligned}$$

Consequently, for every $k \in \mathbb{Z}^2$, there are choices of $r_k, s_k \in \mathbb{R}$ such that

$$a_k = \frac{1}{2} r_k + \frac{1}{2} i s_k$$

and

$$a_{-k} = \frac{1}{2} r_k - \frac{1}{2} i s_k.$$

In terms of code, we can generate a random real-valued band-limited w by picking, at random, real values for r_k and s_k. We only do this process for $k \in A_{\rho_1,\rho_2} \cap \Big((\mathbb{Z} \times \mathbb{Z}_{>0}) \cup \big(\mathbb{Z}_{\geq 0} \times \{0\}\big) \Big)$ so as to avoid double counting. We also assert that $s_0 = 0$.

We now need a way to calculate various useful quantities in terms of these coefficients a_k alone (that is, we do not want to do numerical integration, as it it slow and can produce larger error). First, we calculate the Hessian:

$$\begin{aligned}
\nabla\nabla w(x) &= \nabla\nabla \sum_{k\in\mathbb{Z}^2} a_k e^{2\pi i k\cdot x} \\
&= \begin{pmatrix} \partial_1^2 \sum_{k\in\mathbb{Z}^2} a_k e^{2\pi i k\cdot x} & \partial_1\partial_2 \sum_{k\in\mathbb{Z}^2} a_k e^{2\pi i k\cdot x} \\ \partial_1\partial_2 \sum_{k\in\mathbb{Z}^2} a_k e^{2\pi i k\cdot x} & \partial_2^2 \sum_{k\in\mathbb{Z}^2} a_k e^{2\pi i k\cdot x} \end{pmatrix} \\
&= \begin{pmatrix} -4\pi^2 \sum_{k\in\mathbb{Z}^2} k_1^2 a_k e^{2\pi i k\cdot x} & -4\pi^2 \sum_{k\in\mathbb{Z}^2} k_1 k_2 a_k e^{2\pi i k\cdot x} \\ -4\pi^2 \sum_{k\in\mathbb{Z}^2} k_1 k_2 a_k e^{2\pi i k\cdot x} & -4\pi^2 \sum_{k\in\mathbb{Z}^2} k_2^2 a_k e^{2\pi i k\cdot x} \end{pmatrix}.
\end{aligned}$$

From here, we have

$$\begin{aligned}
\det \nabla\nabla w(x) &= 16\pi^4 \left(\left(\sum_{k\in\mathbb{Z}^2} k_1^2 a_k e^{2\pi i k\cdot x} \right) \left(\sum_{k\in\mathbb{Z}^2} k_2^2 a_k e^{2\pi i k\cdot x} \right) - \left(\sum_{k\in\mathbb{Z}^2} k_1 k_2 a_k e^{2\pi i k\cdot x} \right)^2 \right) \\
&= 16\pi^4 \sum_{k\in\mathbb{Z}^2} \sum_{j\in\mathbb{Z}^2} \Big((k_1 - j_1)^2 j_2^2 - j_1 j_2 (k_1 - j_1)(k_2 - j_2) \Big) a_j a_{k-j} e^{2\pi i k\cdot x}.
\end{aligned}$$

The second equality here comes from convolving the summations. We now know that

$$\widehat{\det \nabla\nabla w}(k) = 16\pi^4 \sum_{j\in\mathbb{Z}^2} \Big((k_1 - j_1)^2 j_2^2 - j_1 j_2 (k_1 - j_1)(k_2 - j_2) \Big) a_j a_{k-j}.$$

Note that, by the same reasoning for w, we know that

$$\widehat{\det \nabla\nabla w}(k) = \overline{\widehat{\det \nabla\nabla w}(-k)},$$

for $\det \nabla\nabla w$ is real-valued.

With these quantities, we are in a position to derive formulæ for E_{stretch} and E_{bend}. We start with

E_{stretch}:

$$\begin{aligned}
E_{\text{stretch}} &= \int_{\mathbb{T}^2} \left| |\nabla|^{-2} \det \nabla\nabla w(x) \right|^2 \mathrm{d}x \\
&= \int_{\mathbb{T}^2} \left| \sum_{k \in \mathbb{Z}^2 \setminus \{0\}} \left| \frac{1}{\left(2\pi|k|\right)^2} \widehat{\det \nabla\nabla w}(k) \right| e^{2\pi i k \cdot x} \right|^2 \mathrm{d}x \\
&= \sum_{k \in \mathbb{Z}^2 \setminus \{0\}} \left| \frac{1}{\left(2\pi|k|\right)^2} \widehat{\det \nabla\nabla w}(k) \right|^2 \\
&= \frac{1}{16\pi^4} \sum_{k \in \mathbb{Z}^2 \setminus \{0\}} \frac{1}{|k|^4} \left| \widehat{\det \nabla\nabla w}(k) \right|^2 \\
&= 16\pi^4 \sum_{k \in \mathbb{Z}^2 \setminus \{0\}} \frac{1}{|k|^4} \left| \sum_{j \in \mathbb{Z}^2} \left((k_1 - j_1)^2 j_2^2 - j_1 j_2 \left(k_1 - j_1\right)\left(k_2 - j_2\right) \right) a_j a_{k-j} \right|^2 .
\end{aligned}$$

In terms of code, we first calculate every value of

$$\left((k_1 - j_1)^2 j_2^2 - j_1 j_2 \left(k_1 - j_1\right)\left(k_2 - j_2\right) \right) a_j a_{k-j}$$

for $j, k-j \in A_{\rho_1,\rho_2}$ with $k \neq 0$ (every other value is necessarily 0). This is the same as computing every value of

$$\left(k_1^2 j_2^2 - k_1 k_2 j_1 j_2 \right) a_j a_k$$

for $k, j \in A_{\rho_1,\rho_2}$ with $k \neq j$. We then compute the sums.

For E_{bend}, we have

$$\begin{aligned}
E_{\text{bend}} &= \int_{\mathbb{T}^2} \left| \nabla\nabla w(x) \right|^2 \mathrm{d}x \\
&= \int_{\mathbb{T}^2} \left| \begin{pmatrix} -4\pi^2 \sum_{k \in \mathbb{Z}^2} k_1^2 a_k e^{2\pi i k \cdot x} & -4\pi^2 \sum_{k \in \mathbb{Z}^2} k_1 k_2 a_k e^{2\pi i k \cdot x} \\ -4\pi^2 \sum_{k \in \mathbb{Z}^2} k_1 k_2 a_k e^{2\pi i k \cdot x} & -4\pi^2 \sum_{k \in \mathbb{Z}^2} k_2^2 a_k e^{2\pi i k \cdot x} \end{pmatrix} \right|^2 \mathrm{d}x \\
&= 16\pi^4 \int_{\mathbb{T}^2} \left(\left| \sum_{k \in \mathbb{Z}^2} k_1^2 a_k e^{2\pi i k \cdot x} \right|^2 + 2 \left| \sum_{k \in \mathbb{Z}^2} k_1 k_2 a_k e^{2\pi i k \cdot x} \right|^2 + \left| \sum_{k \in \mathbb{Z}^2} k_2^2 a_k e^{2\pi i k \cdot x} \right|^2 \right) \mathrm{d}x \\
&= 16\pi^4 \left(\sum_{k \in \mathbb{Z}^2} \left| k_1^2 a_k \right|^2 + 2 \sum_{k \in \mathbb{Z}^2} |k_1 k_2 a_k|^2 + \sum_{k \in \mathbb{Z}^2} \left| k_2^2 a_k \right|^2 \right) \\
&= 16\pi^4 \sum_{k \in \mathbb{Z}^2} \left(k_1^2 + k_2^2 \right)^2 |a_k|^2 \\
&= 16\pi^4 \sum_{k \in \mathbb{Z}^2} |k|^4 |a_k|^2 .
\end{aligned}$$

We also derive a formula for the normalization terms.

First, we find E_M. We begin by calculating

$$\begin{aligned}\int_{\mathbb{T}^2} \nabla w(x) \otimes \nabla w(x)\,\mathrm{d}x &= \int_{\mathbb{T}^2} \begin{pmatrix} 2\pi i \sum_{k\in\mathbb{Z}^2} k_1 a_k e^{2\pi i k\cdot x} \\ 2\pi i \sum_{k\in\mathbb{Z}^2} k_2 a_k e^{2\pi i k\cdot x} \end{pmatrix} \otimes \begin{pmatrix} 2\pi i \sum_{k\in\mathbb{Z}^2} k_1 a_k e^{2\pi i k\cdot x} \\ 2\pi i \sum_{k\in\mathbb{Z}^2} k_2 a_k e^{2\pi i k\cdot x} \end{pmatrix} \mathrm{d}x \\ &= \begin{pmatrix} 4\pi^2 \sum_{k\in\mathbb{Z}^2} k_1^2 |a_k|^2 & 4\pi^2 \sum_{k\in\mathbb{Z}^2} k_1 k_2 |a_k|^2 \\ 4\pi^2 \sum_{k\in\mathbb{Z}^2} k_1 k_2 |a_k|^2 & 4\pi^2 \sum_{k\in\mathbb{Z}^2} k_2^2 |a_k|^2 \end{pmatrix}.\end{aligned}$$

From here we can directly calculate the squared Frobenius norm:

$$\begin{aligned}E_M &= \left| \frac{1}{2} \int_{\mathbb{T}^2} \nabla w \otimes \nabla w - M \right|^2 \\ &= \left(2\pi^2 \sum_{k\in\mathbb{Z}^2} k_1^2 |a_k|^2 - M_{11} \right)^2 + 2 \left(2\pi^2 \sum_{k\in\mathbb{Z}^2} k_1 k_2 |a_k|^2 - M_{12} \right)^2 + \left(2\pi^2 \sum_{k\in\mathbb{Z}^2} k_2^2 |a_k|^2 - M_{22} \right)^2.\end{aligned}$$

Finally,

$$\begin{aligned}E_{\text{height}} &= \int_{\mathbb{T}^2} w^2 \\ &= \sum_{k\in\mathbb{Z}^2} |a_k|^2,\end{aligned}$$

since w is real-valued.

Implementations of these calculations in C++ can be found in Section 6.1 and Section 6.2.

3.2 Gradient Descent

The goal of this section is to create surfaces that have low energies as defined by Equation 1. Our strategy is to generate a random surface and use gradient descent (GD) to locally minimize $E_{M,h,p}$. Following the minimum total potential energy principle, we expect the functions output by our program to approximate a possible final configuration of the foil after cooling.

In the following sections, the ∇ symbol should be taken to mean the vector of derivatives with respect to r_z and s_z from Section 3.1, except the cases ∇w and $\nabla\nabla w$ (here, we mean to differentiate with respect to the two input variables for $w : \mathbb{R}^2 \to \mathbb{R}$). We differentiate with these variables since it is easiest to apply GD on a function of real variables.

3.2.1 Gradient of E_{stretch}

To implement GD, our first objective is to calculate

$$\nabla E_{\text{stretch}} = \nabla 16\pi^4 \sum_{k\in\mathbb{Z}^2\setminus\{0\}} \frac{1}{|k|^4} \left| \sum_{j\in\mathbb{Z}^2} \left((k_1 - j_1)^2 j_2^2 - j_1 j_2 (k_1 - j_1)(k_2 - j_2) \right) a_j a_{k-j} \right|^2.$$

For convenience of notation, define

$$b_{k,j} := (k_1 - j_1)^2 j_2^2 - j_1 j_2 (k_1 - j_1)(k_2 - j_2).$$

Note that

$$\begin{aligned}b_{k,j} + b_{k,k-j} &= (k_1 - j_1)^2 j_2^2 - j_1 j_2 (k_1 - j_1)(k_2 - j_2) + j_1^2 (k_2 - j_2)^2 - j_1 j_2 (k_1 - j_1)(k_2 - j_2) \\ &= \big((k_1 - j_1) j_2 - j_1 (k_2 - j_2) \big)^2 \\ &= (k_1 j_2 - k_2 j_1)^2.\end{aligned}$$

This fact will be used later.

Rewriting the above, we want to compute

$$\nabla 16\pi^4 \sum_{k\in\mathbb{Z}^2\setminus\{0\}} \frac{1}{|k|^4}\left|\sum_{j\in\mathbb{Z}^2} b_{k,j}a_j a_{k-j}\right|^2 = \nabla 16\pi^4 \sum_{k\in\mathbb{Z}^2\setminus\{0\}} \frac{1}{|k|^4}\left[\left(\sum_{j\in\mathbb{Z}^2} b_{k,j}a_j a_{k-j}\right)\overline{\left(\sum_{j\in\mathbb{Z}^2} b_{k,j}a_j a_{k-j}\right)}\right]$$
$$= \nabla 16\pi^4 \sum_{k\in\mathbb{Z}^2\setminus\{0\}} \frac{1}{|k|^4}\left[\left(\sum_{j\in\mathbb{Z}^2} b_{k,j}a_j a_{k-j}\right)\left(\sum_{j\in\mathbb{Z}^2} b_{k,j}\overline{a_j a_{k-j}}\right)\right].$$

We begin by finding the derivative (ignoring the leading constant temporarily) with respect to r_z, where $z \in \mathbb{Z}^2_{\geq 0}$. Since

$$\frac{\partial}{\partial r_z} a_j a_{k-j} = \frac{1}{2}a_{k-j}\left[\!\left[j \in \{z,-z\}\right]\!\right] + \frac{1}{2}a_j\left[\!\left[k-j \in \{z,-z\}\right]\!\right]$$

and

$$\frac{\partial}{\partial r_z} \overline{a_j a_{k-j}} = \frac{1}{2}\overline{a_{k-j}}\left[\!\left[j \in \{z,-z\}\right]\!\right] + \frac{1}{2}\overline{a_j}\left[\!\left[k-j \in \{z,-z\}\right]\!\right],$$

we must have

$$\begin{aligned}
\frac{\partial}{\partial r_z}\sum_{j\in\mathbb{Z}^2} b_{k,j}a_j a_{k-j} &= \sum_{j\in\mathbb{Z}^2} b_{k,j}\left(\frac{1}{2}a_{k-j}\left[\!\left[j \in \{z,-z\}\right]\!\right] + \frac{1}{2}a_j\left[\!\left[k-j \in \{z,-z\}\right]\!\right]\right)\\
&= \frac{1}{2}\sum_{j\in\mathbb{Z}^2} b_{k,j}a_{k-j}\left[\!\left[j \in \{z,-z\}\right]\!\right] + \frac{1}{2}\sum_{j\in\mathbb{Z}^2} b_{k,j}a_j\left[\!\left[k-j \in \{z,-z\}\right]\!\right]\\
&= \frac{1}{2}\sum_{j\in\mathbb{Z}^2} b_{k,j}a_{k-j}\left[\!\left[j \in \{z,-z\}\right]\!\right] + \frac{1}{2}\sum_{j\in\mathbb{Z}^2} b_{k,k-j}a_{k-j}\left[\!\left[j \in \{z,-z\}\right]\!\right]\\
&= \frac{1}{2}\sum_{j\in\mathbb{Z}^2} \left(k_1 j_2 - k_2 j_1\right)^2 a_{k-j}\left[\!\left[j \in \{z,-z\}\right]\!\right]\\
&= \frac{1}{2}\left(k_1 z_2 - k_2 z_1\right)^2\left(a_{k-z} + a_{k+z}\right).
\end{aligned}$$

Some care must be taken for the last equality when $z = 0$. In this case, both quantities are 0, so the equality holds. Similarly,

$$\frac{\partial}{\partial r_z}\sum_{j\in\mathbb{Z}^2} b_{k,j}\overline{a_j a_{k-j}} = \frac{1}{2}\left(k_1 z_2 - k_2 z_1\right)^2\left(\overline{a_{k-z}} + \overline{a_{k+z}}\right)$$

Now,

$$\begin{aligned}
\frac{\partial}{\partial r_z} E_{\text{stretch}} &= \frac{\partial}{\partial r_z} 16\pi^4 \sum_{k\in\mathbb{Z}^2\setminus\{0\}} \frac{1}{|k|^4} \left| \sum_{j\in\mathbb{Z}^2} b_{k,j} a_j a_{k-j} \right|^2 \\
&= \frac{\partial}{\partial r_z} 16\pi^4 \sum_{k\in\mathbb{Z}^2\setminus\{0\}} \frac{1}{|k|^4} \left[\left(\sum_{j\in\mathbb{Z}^2} b_{k,j} a_j a_{k-j} \right) \left(\sum_{j\in\mathbb{Z}^2} b_{k,j} \overline{a_j a_{k-j}} \right) \right] \\
&= 16\pi^4 \sum_{k\in\mathbb{Z}^2\setminus\{0\}} \frac{1}{|k|^4} \left(\sum_{j\in\mathbb{Z}^2} b_{k,j} a_j a_{k-j} \right) \cdot \frac{1}{2} \left(k_1 z_2 - k_2 z_1\right)^2 \left(\overline{a_{k-z}} + \overline{a_{k+z}}\right) \\
&\quad + 16\pi^4 \sum_{k\in\mathbb{Z}^2\setminus\{0\}} \frac{1}{|k|^4} \left(\sum_{j\in\mathbb{Z}^2} b_{k,j} \overline{a_j a_{k-j}} \right) \cdot \frac{1}{2} \left(k_1 z_2 - k_2 z_1\right)^2 \left(a_{k-z} + a_{k+z}\right) \\
&= \sum_{k\in\mathbb{Z}^2\setminus\{0\}} \frac{1}{|k|^4} \left(k_1 z_2 - k_2 z_1\right)^2 \Re\Big(\widehat{\det \nabla\nabla w}(k) \left(\overline{a_{k-z}} + \overline{a_{k+z}}\right) \Big) \\
&= \sum_{k\in\mathbb{Z}^2\setminus\{0\}} \frac{1}{|k|^4} \left(k_1 z_2 - k_2 z_1\right)^2 \Re\Big(\widehat{\det \nabla\nabla w}(k) \overline{a_{k-z}} \Big) \\
&\quad + \sum_{k\in\mathbb{Z}^2\setminus\{0\}} \frac{1}{|k|^4} \left(k_1 z_2 - k_2 z_1\right)^2 \Re\Big(\widehat{\det \nabla\nabla w}(k) \overline{a_{k+z}} \Big) \\
&= \sum_{k\in\mathbb{Z}^2\setminus\{0\}} \frac{1}{|k|^4} \left(k_1 z_2 - k_2 z_1\right)^2 \Re\Big(\widehat{\det \nabla\nabla w}(k) \overline{a_{k-z}} \Big) \\
&\quad + \sum_{k\in\mathbb{Z}^2\setminus\{0\}} \frac{1}{|k|^4} \left(k_1 z_2 - k_2 z_1\right)^2 \Re\Big(\widehat{\det \nabla\nabla w}(-k) \overline{a_{-k+z}} \Big) \\
&= \sum_{k\in\mathbb{Z}^2\setminus\{0\}} \frac{1}{|k|^4} \left(k_1 z_2 - k_2 z_1\right)^2 \Re\Big(\widehat{\det \nabla\nabla w}(k) \overline{a_{k-z}} \Big) \\
&\quad + \sum_{k\in\mathbb{Z}^2\setminus\{0\}} \frac{1}{|k|^4} \left(k_1 z_2 - k_2 z_1\right)^2 \Re\Big(\widehat{\det \nabla\nabla w}(k) \overline{a_{k-z}} \Big) \\
&= \sum_{k\in\mathbb{Z}^2\setminus\{0\}} \frac{2}{|k|^4} \left(k_1 z_2 - k_2 z_1\right)^2 \Re\Big(\widehat{\det \nabla\nabla w}(k) \overline{a_{k-z}} \Big) \\
&= \sum_{k\in\mathbb{Z}^2\setminus\{0\}} \frac{2}{|k|^4} \left(k_1 z_2 - k_2 z_1\right)^2 \Re\Big(\widehat{\det \nabla\nabla w}(k) a_{z-k} \Big). \qquad (2)
\end{aligned}$$

Next, we compute the derivative with respect to s_z. Following the same steps as above, we see

$$\frac{\partial}{\partial s_z} a_j a_{k-j} = \frac{1}{2} i a_{k-j} \left([\![j = z]\!] - [\![j = -z]\!] \right) + \frac{1}{2} i a_j \left([\![k-j = z]\!] - [\![k-j = -z]\!] \right)$$

and

$$\begin{aligned}
\frac{\partial}{\partial s_z} \overline{a_j a_{k-j}} &= \frac{1}{2} i \overline{a_{k-j}} \left([\![j = -z]\!] - [\![j = z]\!] \right) + \frac{1}{2} i \overline{a_j} \left([\![k-j = -z]\!] - [\![k-j = z]\!] \right) \\
&= \frac{1}{2} \overline{i a_{k-j}} \left([\![j = z]\!] - [\![j = -z]\!] \right) + \frac{1}{2} \overline{i a_j} \left([\![k-j = z]\!] - [\![k-j = -z]\!] \right).
\end{aligned}$$

Continuing, the above yields

$$\begin{aligned}\frac{\partial}{\partial s_z}\sum_{j\in\mathbb{Z}^2} b_{k,j}a_j a_{k-j} &= \frac{1}{2}\sum_{j\in\mathbb{Z}^2}\left(k_1j_2-k_2j_1\right)^2 ia_{k-j}\left(\llbracket j=z\rrbracket-\llbracket j=-z\rrbracket\right)\\ &= \frac{1}{2}\left(k_1z_2-k_2z_1\right)^2\left(ia_{k-z}-ia_{k+z}\right)\end{aligned}$$

and

$$\begin{aligned}\frac{\partial}{\partial s_z}\sum_{j\in\mathbb{Z}^2} b_{k,j}\overline{a_j a_{k-j}} &= \frac{1}{2}\sum_{j\in\mathbb{Z}^2}\left(k_1j_2-k_2j_1\right)^2 \overline{ia_{k-j}}\left(\llbracket j=z\rrbracket-\llbracket j=-z\rrbracket\right)\\ &= \frac{1}{2}\left(k_1z_2-k_2z_1\right)^2\left(\overline{ia_{k-z}}-\overline{ia_{k+z}}\right).\end{aligned}$$

Finally,

$$\frac{\partial}{\partial s_z}E_{\text{stretch}} = \sum_{k\in\mathbb{Z}^2\setminus\{0\}}\frac{2}{|k|^4}\left(k_1z_2-k_2z_1\right)^2\Im\Big(\widehat{\det\nabla\nabla w}(k)a_{z-k}\Big). \tag{3}$$

Some steps have been omitted here, as they are much the same as the $\frac{\partial}{\partial r_z}$ case above.

Note that the term $\widehat{\det\nabla\nabla w}(k)$ appears in both final forms of the derivatives and is independent of z, so, to save on computation time, we can compute this value once per iteration of GD and then reuse it. Also, we only need to sum over $k\in A_{\rho_1,\rho_2}+\Big(A_{\rho_1,\rho_2}\cap\Big((\mathbb{Z}\times\mathbb{Z}_{>0})\cup(\mathbb{Z}_{\geq 0}\times\{0\})\Big)\Big)$, as those are the only k's for which the summand could possibly be non-zero.

3.2.2 Gradient of E_{mat}

Our next step is to find formulæ to compute

$$\begin{aligned}\nabla\left|\frac{1}{2}\int_{\mathbb{T}^2}\nabla w\otimes\nabla w - M\right|^2 = \nabla\Bigg[&\left(2\pi^2\sum_{k\in\mathbb{Z}^2}k_1^2|a_k|^2-M_{11}\right)^2+2\left(2\pi^2\sum_{k\in\mathbb{Z}^2}k_1k_2|a_k|^2-M_{12}\right)^2\\ &+\left(2\pi^2\sum_{k\in\mathbb{Z}^2}k_2^2|a_k|^2-M_{22}\right)^2\Bigg].\end{aligned}$$

We start by finding

$$\frac{\partial}{\partial r_z}|a_j|^2 = \Re(a_j)\,\llbracket j\in\{z,-z\}\rrbracket$$

and

$$\frac{\partial}{\partial s_z}|a_j|^2 = \Im(a_j)\left(\llbracket j=z\rrbracket-\llbracket j=-z\rrbracket\right).$$

By the chain rule, we get

$$\begin{aligned}
\frac{\partial}{\partial r_z} E_M &= 4\pi^2 \Bigg[\left(2\pi^2 \sum_{k\in\mathbb{Z}^2} k_1^2 |a_k|^2 - M_{11} \right) \sum_{j\in\mathbb{Z}^2} j_1^2 \Re\big(a_j\big) \, [\![j \in \{z, -z\}]\!] \\
&\qquad + 2 \left(2\pi^2 \sum_{k\in\mathbb{Z}^2} k_1 k_2 |a_k|^2 - M_{12} \right) \sum_{j\in\mathbb{Z}^2} j_1 j_2 \Re\big(a_j\big) \, [\![j \in \{z, -z\}]\!] \\
&\qquad + \left(2\pi^2 \sum_{k\in\mathbb{Z}^2} k_2^2 |a_k|^2 - M_{22} \right) \sum_{j\in\mathbb{Z}^2} j_2^2 \Re\big(a_j\big) \, [\![j \in \{z, -z\}]\!] \Bigg] \\
&= 4\pi^2 \Bigg[\left(2\pi^2 \sum_{k\in\mathbb{Z}^2} k_1^2 |a_k|^2 - M_{11} \right) \cdot 2 z_1^2 \Re(a_z) + 2 \left(2\pi^2 \sum_{k\in\mathbb{Z}^2} k_1 k_2 |a_k|^2 - M_{12} \right) \cdot 2 z_1 z_2 \Re(a_z) \\
&\qquad + \left(2\pi^2 \sum_{k\in\mathbb{Z}^2} k_2^2 |a_k|^2 - M_{22} \right) \cdot 2 z_2^2 \Re(a_z) \Bigg] \\
&= 8\pi^2 \Bigg[\left(2\pi^2 \sum_{k\in\mathbb{Z}^2} k_1^2 |a_k|^2 - M_{11} \right) z_1^2 + 2 \left(2\pi^2 \sum_{k\in\mathbb{Z}^2} k_1 k_2 |a_k|^2 - M_{12} \right) z_1 z_2 \\
&\qquad + \left(2\pi^2 \sum_{k\in\mathbb{Z}^2} k_2^2 |a_k|^2 - M_{22} \right) z_2^2 \Bigg] \Re(a_z) \qquad (4)
\end{aligned}$$

and

$$\begin{aligned}
\frac{\partial}{\partial s_z} E_M &= 4\pi^2 \Bigg[\left(2\pi^2 \sum_{k\in\mathbb{Z}^2} k_1^2 |a_k|^2 - M_{11} \right) \sum_{j\in\mathbb{Z}^2} j_1^2 \Im\big(a_j\big) \left([\![j = z]\!] - [\![j = -z]\!] \right) \\
&\qquad + 2 \left(2\pi^2 \sum_{k\in\mathbb{Z}^2} k_1 k_2 |a_k|^2 - M_{12} \right) \sum_{j\in\mathbb{Z}^2} j_1 j_2 \Im\big(a_j\big) \left([\![j = z]\!] - [\![j = -z]\!] \right) \\
&\qquad + \left(2\pi^2 \sum_{k\in\mathbb{Z}^2} k_2^2 |a_k|^2 - M_{22} \right) \sum_{j\in\mathbb{Z}^2} j_2^2 \Im\big(a_j\big) \left([\![j = z]\!] - [\![j = -z]\!] \right) \Bigg] \\
&= 8\pi^2 \Bigg[\left(2\pi^2 \sum_{k\in\mathbb{Z}^2} k_1^2 |a_k|^2 - M_{11} \right) z_1^2 + 2 \left(2\pi^2 \sum_{k\in\mathbb{Z}^2} k_1 k_2 |a_k|^2 - M_{12} \right) z_1 z_2 \\
&\qquad + \left(2\pi^2 \sum_{k\in\mathbb{Z}^2} k_2^2 |a_k|^2 - M_{22} \right) z_2^2 \Bigg] \Im(a_z). \qquad (5)
\end{aligned}$$

To be slightly more efficient, we calculate the quantities

$$\sum_{k\in\mathbb{Z}^2} k_1^2 |a_k|^2 ,$$
$$\sum_{k\in\mathbb{Z}^2} k_1 k_2 |a_k|^2 ,$$

and

$$\sum_{k\in\mathbb{Z}^2} k_2^2 |a_k|^2$$

just once.

3.2.3 Gradient of E_{bend}

Using the results from above,

$$\begin{aligned}\frac{\partial}{\partial r_z} E_{\text{bend}} &= 16\pi^4 \sum_{j\in\mathbb{Z}^2} \left(j_1^2 + j_2^2\right)^2 \Re(a_j) \left[\!\left[j \in \{z, -z\}\right]\!\right] \\ &= 32\pi^4 \left(z_1^2 + z_2^2\right)^2 \Re(a_z) \end{aligned} \tag{6}$$

and

$$\begin{aligned}\frac{\partial}{\partial s_z} E_{\text{bend}} &= 16\pi^4 \sum_{j\in\mathbb{Z}^2} \left(j_1^2 + j_2^2\right)^2 \Im(a_j) \left(\left[\!\left[j = z\right]\!\right] - \left[\!\left[j = -z\right]\!\right]\right) \\ &= 32\pi^4 \left(z_1^2 + z_2^2\right)^2 \Im(a_z). \end{aligned} \tag{7}$$

Note again that for the first equation, some attention must be given to the case $z = 0$.

3.2.4 Gradient of E_{height}

Again using the previous results,

$$\frac{\partial}{\partial r_z} E_{\text{height}} = \begin{cases} 2\Re(a_z) & z \neq 0 \\ \Re(a_z) & z = 0 \end{cases} \tag{8}$$

and

$$\frac{\partial}{\partial s_z} E_{\text{height}} = 2\Im(a_z). \tag{9}$$

3.2.5 The Algorithm Proper

Using Equations 2 through 9, we can find

$$\frac{\partial}{\partial r_z} E_{M,h,p} = \frac{\partial}{\partial r_z} E_{\text{stretch}} + \frac{\partial}{\partial r_z} E_M + h^2 \frac{\partial}{\partial r_z} E_{\text{bend}} + \frac{1}{h^p} \frac{\partial}{\partial r_z} E_{\text{height}}$$

and

$$\frac{\partial}{\partial s_z} E_{M,h,p} = \frac{\partial}{\partial s_z} E_{\text{stretch}} + \frac{\partial}{\partial s_z} E_M + h^2 \frac{\partial}{\partial s_z} E_{\text{bend}} + \frac{1}{h^p} \frac{\partial}{\partial s_z} E_{\text{height}}.$$

A single iteration of GD looks like

Algorithm 1 Gradient Descent Iteration

for each $z \in A_{\rho_1,\rho_2} \cap \Big((\mathbb{Z} \times \mathbb{Z}_{>0}) \cup (\mathbb{Z}_{\geq 0} \times \{0\}) \Big)$ **do**
  $\widetilde{r_z} \leftarrow r_z - \eta \left(\frac{\partial}{\partial r_z} E_{\text{stretch}} + \frac{\partial}{\partial r_z} E_M + h^2 \frac{\partial}{\partial r_z} E_{\text{bend}} + \frac{1}{h^p} \frac{\partial}{\partial r_z} E_{\text{height}} \right)$
  $\widetilde{s_z} \leftarrow s_z - \eta \left(\frac{\partial}{\partial s_z} E_{\text{stretch}} + \frac{\partial}{\partial s_z} E_M + h^2 \frac{\partial}{\partial s_z} E_{\text{bend}} + \frac{1}{h^p} \frac{\partial}{\partial s_z} E_{\text{height}} \right)$
for each $z \in A_{\rho_1,\rho_2} \cap \Big((\mathbb{Z} \times \mathbb{Z}_{>0}) \cup (\mathbb{Z}_{\geq 0} \times \{0\}) \Big)$ **do**
  $a_z \leftarrow \frac{1}{2}\widetilde{r_z} + \frac{1}{2} i \widetilde{s_z}$
  $a_{-z} \leftarrow \frac{1}{2}\widetilde{r_z} - \frac{1}{2} i \widetilde{s_z}$

Here, the learning rate $\eta \in \mathbb{R}_{>0}$ is chosen every iteration. A rough outline of our strategy is to increase η when an iteration decreases the total energy and to increase η otherwise. This yields the algorithm

Algorithm 2 Gradient Descent

generate a random surface w
while $E_{M,h,p} >$ target value **do**
  run a Gradient Descent Iteration
  if $E_{M,h,p}$ has decreased for the past N iterations **then**
    increase η
  else
    decrease η
    revert w to what it was in the previous iteration

The number N here is fixed and should be chosen beforehand. Choosing to revert w back in the case $E_{M,h,p}$ were to increase is a result of some testing.

An implementation of these algorithms in C++ can be found in Section 6.3.

4 Results and Observations

Some examples of outputs for $\rho_1 = 1$, $p = 1$, and $M = I$ were generated using the code from Section 6.4 and are reproduced in Figure 1, Figure 2, and Figure 3. The plots of the surfaces described by each w depict herringbone-like patterns that are semi-consistent with the physical results found by Chen and Hutchinson [2].

As mentioned in Section 1, we have made a rather strong assumption that the gold foil can be well approximated by a Fourier series that is *band-limited*. Some empirical results justify this assumption. Keeping $h = 0.005$ and $\rho_1 = 1$ constant and varying ρ_2, we observe that the Fourier coefficients that are the largest in magnitude are supported roughly in the band $A_{8,19}$ for sufficiently large ρ_2 (see Figure 4). Similar results hold for other fixed values of h. This observation suggests that minimizing our energy functional over the space of *all* doubly periodic surfaces can be approximately simplified to minimizing the energy functional over the space of doubly periodic Fourier series that have coefficients supported in an appropriate band.

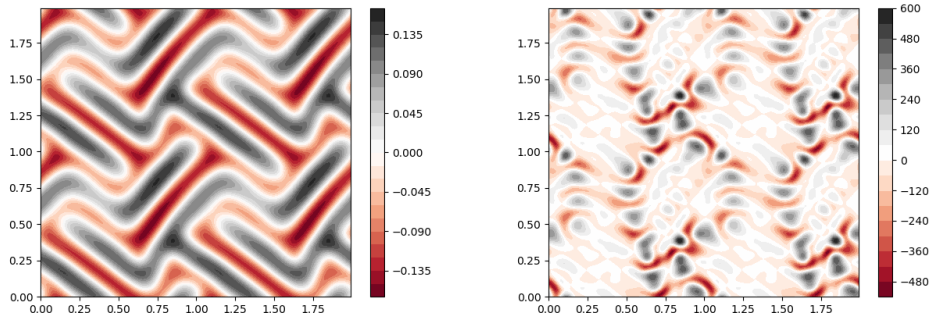

Figure 1: Using $h = 0.02$ and $\rho_2 = 10$. Plot of w is on the left; $\det \nabla\nabla w$ is on the right

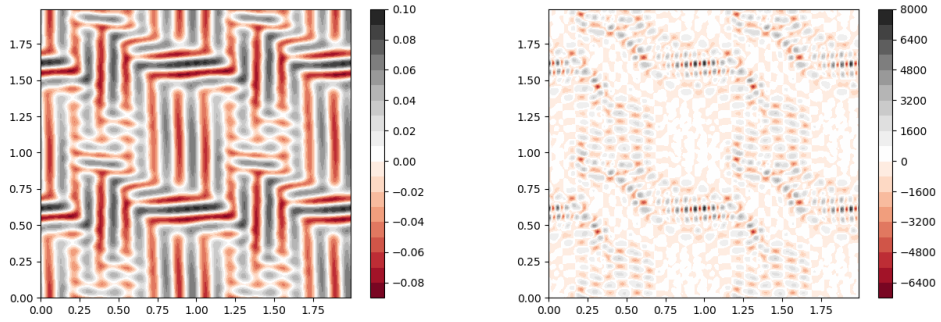

Figure 2: Using $h = 0.01$ and $\rho_2 = 20$. Plot of w is on the left; $\det \nabla\nabla w$ is on the right

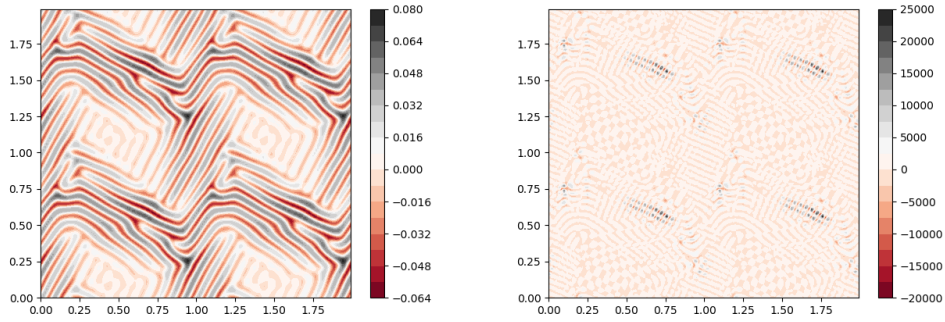

Figure 3: Using $h = 0.005$ and $\rho_2 = 30$. Plot of w is on the left; $\det \nabla\nabla w$ is on the right

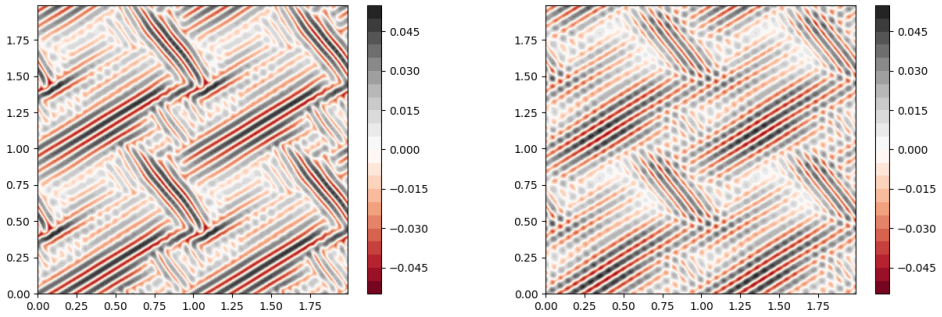

Figure 4: Using $h = 0.005$, $\rho_1 = 1$, $\rho_2 = 35$. The plot on the left is the function returned after gradient descent. The plot on the right is the same function but after manually setting every "small" coefficient (i.e. magnitude at most 0.001) to 0. Note that these two plots are extremely similar, even though the coefficients of the second series are supported in $A_{8,19}$.

5 Reflection and Ideas for Future Work

While no theorem has yet come of this work, the programs created allow for relatively quick generation of random low energy band-limited Fourier series provided h is not too small.

Future work could include looking into the following:

- As $h \to 0$, how much band-limiting occurs?
- The current program becomes quite slow when many coefficients are allowed to vary. Possible remedies could include a better choice of random coefficients at the beginning of the algorithm and a better choice of learning rate η. Additionally, determining the band-limiting effect would reduce the search space significantly.
- The model currently does not take into account the in-plane displacements of the foil. If these sorts of displacements are considered, maybe a more clear Herringbone pattern could emerge, more consistent with physical experiments.
- For a low energy surface determined by w, it appears that $\det \nabla\nabla w$ is close to 0 except at very few places. Is this phenomenon necessary?

6 Appendix[1]

6.1 `fourier.h`

```
#ifndef FOURIER_H
#define FOURIER_H

#include <complex>
#include <functional>
#include <unordered_map>
#include <utility>
#include <vector>

// Represents a real-valued band-limited Fourier series w on the domain [0, 1] x
// [0, 1] that can be written as the sum of terms like a_k * exp(2 * pi * i *
// <k, x>). Here, k takes on all values in Z^2. All functions herein assume that
// sum is real-valued for all x in R^2.
class FourierSeries {
 public:
  // Structure that can hash a pair of ints. Used to create and use the
  // coefficients variable.
  struct pair_hash {
    std::size_t operator()(const std::pair<int, int> &) const;
  };

  // Holds the coefficients of this series. The value a_k can be accessed by
  // coefficients[k], where k here is represented as a pair of ints.
  std::unordered_map<std::pair<int, int>, std::complex<long double>,
                     FourierSeries::pair_hash>
      coefficients;

  // Constructor. Takes in an unordered_map that will represent the coefficients
  // of the Fourier series as described above the coefficients variable.
  FourierSeries(const std::unordered_map<
                std::pair<int, int>, std::complex<long double>, pair_hash> &);

  // Prints the coefficient list into the ostream that is passed in. The list is
  // formatted as a Python dictionary for easy plotting.
  friend std::ostream &operator<<(std::ostream &, const FourierSeries &);

  // Evaluates the Fourier series at a point. Since the series is assumed to be
  // real-valued, the series is evaluated and then the real part is returned.
  long double at(const std::pair<long double, long double> &) const;

  // Returns a copy of the unordered_map coefficients.
  std::unordered_map<std::pair<int, int>, std::complex<long double>, pair_hash>
  get_coefficients() const;

  // Returns the k values necessary to compute E_stretch.
  std::vector<std::pair<int, int> > sum_keys() const;

  // Returns the keys of the unordered_map coefficients that lie within the
  // first quadrant, inclusive of the axes. Since the series is assumed to be
  // real-valued, this set of keys could be used to generate the entire set of
  // keys by reflecting across the x and y axes.
  std::vector<std::pair<int, int> > half_keys() const;

  // Returns the integral of the outer product of the gradient of the series
  // with itself over the domain [0, 1] x [0, 1].
  std::vector<std::vector<long double> > outer_product_integral() const;

  // Returns the square of the Frobenius norm of the difference between the
  // identity matrix and half of the outer product integral.
  long double e_mat(const long double [2][2]) const;
```

[1]This code can be found online at `https://github.com/StephenJasina/low-energy-fourier-series`.

```
  long double e_height() const;

  // Returns the Hessian of the series at a point. Since the series is assumed
  // to be real-valued, the Hessian is evaluated and then the real part is
  // returned.
  std::vector<std::vector<long double> > hessian(
      const std::pair<long double, long double> &) const;

  // Returns the determinant of the Hessian at a point.
  long double hessian_determinant(
      const std::pair<long double, long double> &) const;

  // Returns the corresponding coefficient of the Fourier series of the Hessian
  // determinant.
  std::complex<long double> hessian_determinant_coefficient(
      const std::pair<int, int> &) const;

  long double e_stretch() const;

  long double e_bend() const;

  // Generates a random set of coefficients for a real-valued band-limited
  // Fourier series.
  static std::unordered_map<std::pair<int, int>, std::complex<long double>,
                            pair_hash>
  random_coefficients(long double, long double);
};

#endif
```

6.2 fourier.cpp

```
#include "fourier.h"

#include <chrono>
#include <cmath>
#include <complex>
#include <iterator>
#include <random>
#include <unordered_map>
#include <unordered_set>
#include <utility>

using namespace std;

const long double PI = 3.14159265358979323846L;
const complex<long double> I(0, 1);

// Returns the dot product of the vectors that are represented by the pairs a
// and b.
long double dot(const pair<long double, long double> &a,
                const pair<long double, long double> &b) {
  return a.first * b.first + a.second * b.second;
}

size_t FourierSeries::pair_hash::operator()(const pair<int, int> &p) const {
  hash<int> h;

  auto hash1 = h(p.first);
  auto hash2 = h(p.second);

  // The bitwise XOR of the hashes is returned so that the hash is "more
  // independent" of either one of the values in the pairs.
  return hash1 ^ hash2;
}

FourierSeries::FourierSeries(
    const unordered_map<pair<int, int>, complex<long double>, pair_hash>
```

```
        &coefficients) {
  this->coefficients = coefficients;
}

ostream &operator<<(ostream &os, const FourierSeries &series) {
  auto coefficients = series.get_coefficients();
  auto it = coefficients.cbegin();

  os << '{';
  for (; it != coefficients.cend(); ++it) {
    os << '(' << it->first.first << ", " << it->first.second << "): ("
       << it->second.real();

    if (it->second.imag() < 0) {
      os << " - " << -it->second.imag();
    } else {
      os << " + " << it->second.imag();
    }
    os << "j)";

    if (next(it) != coefficients.end()) {
      os << ", ";
    }
  }
  os << '}';

  return os;
}

unordered_map<pair<int, int>, complex<long double>, FourierSeries::pair_hash>
FourierSeries::get_coefficients() const {
  return this->coefficients;
}

vector<pair<int, int> > FourierSeries::sum_keys() const {
  unordered_set<pair<int, int>, pair_hash> unique_keys;
  vector<pair<int, int> > half_keys = this->half_keys(), sum_keys;

  for (auto it1 = half_keys.cbegin(); it1 != half_keys.cend(); ++it1) {
    for (auto it2 = this->coefficients.cbegin();
         it2 != this->coefficients.cend(); ++it2) {
      unique_keys.emplace(pair<int, int>(it1->first + it2->first.first,
                                         it1->second + it2->first.second));
    }
  }

  // Reserve a bit of space to avoid having to reallocate space.
  sum_keys.reserve(unique_keys.size());

  for (auto it = unique_keys.cbegin(); it != unique_keys.cend(); ++it) {
    sum_keys.push_back(*it);
  }

  return sum_keys;
}

vector<pair<int, int> > FourierSeries::half_keys() const {
  vector<pair<int, int> > half_keys;

  // Reserve a bit of space to avoid having to reallocate space too much. A
  // denominator of 3 (as opposed to 4) is chosen to account for the values
  // lying on the axes.
  half_keys.reserve(this->coefficients.size() / 3);

  for (auto it = this->coefficients.cbegin(); it != this->coefficients.cend();
       ++it) {
    if (it->first.second > 0 ||
        (it->first.second == 0 && it->first.first >= 0)) {
```

```
      half_keys.push_back(it->first);
    }
  }

  return half_keys;
}

long double FourierSeries::at(const pair<long double, long double> &x) const {
  complex<long double> result;

  for (auto it = this->coefficients.cbegin(); it != this->coefficients.cend();
       ++it) {
    result += it->second * exp(2 * PI * I * dot(it->first, x));
  }

  return result.real();
}

vector<vector<long double> > FourierSeries::outer_product_integral() const {
  vector<vector<long double> > m(2, vector<long double>(2));

  // First calculate the summations...
  for (auto it = this->coefficients.cbegin(); it != this->coefficients.cend();
       ++it) {
    m[0][0] += it->first.first * it->first.first * norm(it->second);
    m[0][1] += it->first.first * it->first.second * norm(it->second);
    m[1][1] += it->first.second * it->first.second * norm(it->second);
  }

  // ... then multiply by the appropriate leading constants.
  m[0][0] *= 4 * PI * PI;
  m[0][1] *= 4 * PI * PI;
  m[1][1] *= 4 * PI * PI;

  // This matrix is symmetric, so we save a little time by just setting these
  // two values to be equal manually.
  m[1][0] = m[0][1];

  return m;
}

long double FourierSeries::e_mat(const long double M[2][2]) const {
  auto m = this->outer_product_integral();

  // We take slight advantage of the symmetry of M and m here.
  return (M[0][0] - m[0][0] / 2) * (M[0][0] - m[0][0] / 2) +
         2 * (M[0][1] - m[0][1] / 2) * (M[0][1] - m[0][1] / 2) +
         (M[1][1] - m[1][1] / 2) * (M[1][1] - m[1][1] / 2);
}

long double FourierSeries::e_height() const {
  long double e = 0;

  for (auto it = this->coefficients.cbegin(); it != this->coefficients.cend();
       ++it) {
    e += norm(it->second);
  }

  return e;
}

vector<vector<long double> > FourierSeries::hessian(
    const pair<long double, long double> &x) const {
  vector<vector<long double> > h(2, vector<long double>(2));

  // While the Hessian is ultimately real valued, some intermediate complex
  // values will need to be calculated.
  vector<vector<complex<long double> > > h_complex(
```

```
      2, vector<complex<long double> >(2));

  // First calculate the summations...
  for (auto it = this->coefficients.cbegin(); it != this->coefficients.cend();
       ++it) {
    h_complex[0][0] += (long double)(it->first.first * it->first.first) *
                       it->second * exp(2 * PI * I * dot(it->first, x));
    h_complex[0][1] += (long double)(it->first.first) * it->first.second *
                       it->second * exp(2 * PI * I * dot(it->first, x));
    h_complex[1][1] += (long double)(it->first.second * it->first.second) *
                       it->second * exp(2 * PI * I * dot(it->first, x));
  }

  // ... then multiply by the appropriate leading constants, and take the real
  // parts. Note that the imaginary parts are (or at least, should be) 0.
  h[0][0] = -4 * PI * PI * h_complex[0][0].real();
  h[0][1] = -4 * PI * PI * h_complex[0][1].real();
  h[1][1] = -4 * PI * PI * h_complex[1][1].real();

  // This matrix is symmetric, so we save a little time by just setting these
  // two values to be equal manually.
  h[1][0] = h[0][1];

  return h;
}

long double FourierSeries::hessian_determinant(
    const pair<long double, long double> &x) const {
  auto h = this->hessian(x);

  return h[0][0] * h[1][1] - h[0][1] * h[1][0];
}

complex<long double> FourierSeries::hessian_determinant_coefficient(
    const pair<int, int> &k) const {
  complex<long double> coefficient;

  for (auto it = this->coefficients.cbegin(); it != this->coefficients.cend();
       ++it) {
    auto other =
        pair<int, int>(k.first - it->first.first, k.second - it->first.second);
    if (coefficients.count(other)) {
      coefficient +=
          (long double)other.first * it->first.second *
          (other.first * it->first.second - other.second * it->first.first) *
          it->second * coefficients.at(other);
    }
  }

  return 16 * PI * PI * PI * PI * coefficient;
}

long double FourierSeries::e_stretch() const {
  long double e = 0;
  unordered_map<pair<int, int>, complex<long double>, pair_hash> summands;

  // First, calculate the summand of the interior sum for every valid pair of j
  // and k.
  for (auto it1 = this->coefficients.cbegin(); it1 != this->coefficients.cend();
       ++it1) {
    for (auto it2 = this->coefficients.cbegin();
         it2 != this->coefficients.cend(); ++it2) {
      summands[pair<int, int>(it1->first.first + it2->first.first,
                              it1->first.second + it2->first.second)] +=
          (long double)it1->first.first * it2->first.second *
          (it1->first.first * it2->first.second -
           it1->first.second * it2->first.first) *
          it1->second * it2->second;
```

```
    }
  }

  // Next, combine these values to find e.
  for (auto it = summands.cbegin(); it != summands.cend(); ++it) {
    if (it->first.first != 0 || it->first.second != 0) {
      e += norm(it->second) /
           (dot(it->first, it->first) * dot(it->first, it->first));
    }
  }

  return 16 * PI * PI * PI * PI * e;
}

long double FourierSeries::e_bend() const {
  long double e = 0;

  for (auto it = this->coefficients.cbegin(); it != this->coefficients.cend();
       ++it) {
    e += norm(it->second) * dot(it->first, it->first) *
         dot(it->first, it->first);
  }

  return 16 * PI * PI * PI * PI * e;
}

unordered_map<pair<int, int>, complex<long double>, FourierSeries::pair_hash>
FourierSeries::random_coefficients(long double rho1, long double rho2) {
  mt19937 twister(chrono::system_clock::now().time_since_epoch().count());
  normal_distribution<double> normal;
  unordered_map<pair<int, int>, complex<long double>, pair_hash> coefficients;

  // Small tolerances have been put in to ensure that every coefficient one
  // would expect to have a value does indeed have a value (as there is a
  // chance for small rounding error with long doubles).
  const long double EPSILON = 0.0001;

  for (int k1 = -rho2 - EPSILON; k1 <= rho2 + EPSILON; ++k1) {
    for (int k2 = (k1 >= 0 ? 0 : 1); k2 <= rho2 + EPSILON; ++k2) {
      if (rho1 * rho1 - EPSILON <= k1 * k1 + k2 * k2 &&
          k1 * k1 + k2 * k2 <= rho2 * rho2 + EPSILON) {
        double r = normal(twister), s = normal(twister);

        // Ignore extraneous values in the case that we're on an axis.
        if (k1 == 0 && k2 == 0) {
          s = 0;
        }

        // Defining the coefficients like this ensures that the series will be
        // real valued.
        coefficients[pair<int, int>(k1, k2)] =
            complex<long double>(r, s) / (long double)4;
        coefficients[pair<int, int>(-k1, -k2)] =
            complex<long double>(r, -s) / (long double)4;
      }
    }
  }

  return coefficients;
}
```

6.3 gd.cpp

```
#include <algorithm>
#include <complex>
#include <fstream>
#include <iostream>
#include <unordered_map>
```

```cpp
#include <utility>

#include "fourier.h"

using namespace std;

const long double PI = 3.141592653589793238462L;
const long double ID[2][2] = {{1, 0}, {0, 1}};

void d_e_stretch(const FourierSeries &series,
                 const vector<pair<int, int> > &sum_keys,
                 const vector<pair<int, int> > &half_keys,
                 unordered_map<pair<int, int>, long double[2],
                               FourierSeries::pair_hash> &derivatives) {
  for (const auto &k : sum_keys) {
    if (k.first == 0 && k.second == 0) {
      continue;
    }

    complex<long double> scaled_hess_det_coef =
        (long double)(2) /
        ((k.first * k.first + k.second * k.second) *
         (k.first * k.first + k.second * k.second)) *
        series.hessian_determinant_coefficient(k);

    // This loop calculates the derivatives of our objective function with
    // respect to r_z and s_z for every valid choice of z.
    for (const auto &z : half_keys) {
      // Auxillary variable to make notation better.
      complex<long double> at_z_minus_k =
          series.coefficients.count(
              pair<int, int>(z.first - k.first, z.second - k.second))
              ? series.coefficients.at(
                    pair<int, int>(z.first - k.first, z.second - k.second))
              : 0;

      complex<long double> stretch_multiplier =
          (long double)(k.first * z.second - k.second * z.first) *
          (k.first * z.second - k.second * z.first) * scaled_hess_det_coef *
          at_z_minus_k;

      // Calculate the E_stretch derivatives.
      derivatives[z][0] += stretch_multiplier.real();
      derivatives[z][1] += stretch_multiplier.imag();
    }
  }
}

void d_e_mat(const FourierSeries &series,
             const vector<pair<int, int> > &half_keys,
             unordered_map<pair<int, int>, long double[2],
                           FourierSeries::pair_hash> &derivatives,
             const long double M[2][2]) {
  // Each mat_multiplier# is to contain the value of a summation that appears in
  // every derivative of the outer product norm. Essentially, they are
  // calculated here to avoid having to recalculate the same value for every z
  // and for every derivative.
  long double mat_multiplier1 = 0, mat_multiplier2 = 0, mat_multiplier3 = 0;
  for (auto it = series.coefficients.cbegin(); it != series.coefficients.end();
       ++it) {
    mat_multiplier1 += it->first.first * it->first.first * norm(it->second);
    mat_multiplier2 += it->first.first * it->first.second * norm(it->second);
    mat_multiplier3 += it->first.second * it->first.second * norm(it->second);
  }
  mat_multiplier1 = 8 * PI * PI * (2 * PI * PI * mat_multiplier1 - M[0][0]);
  mat_multiplier2 = 16 * PI * PI * (2 * PI * PI * mat_multiplier2 - M[0][1]);
  mat_multiplier3 = 8 * PI * PI * (2 * PI * PI * mat_multiplier3 - M[1][1]);
```

```
  for (const auto &z : half_keys) {
    complex<long double> mat_multiplier =
        (mat_multiplier1 * z.first * z.first +
         mat_multiplier2 * z.first * z.second +
         mat_multiplier3 * z.second * z.second) *
        series.coefficients.at(z);

    derivatives[z][0] += mat_multiplier.real();
    derivatives[z][1] += mat_multiplier.imag();
  }
}

void d_e_bend(const FourierSeries &series,
              const vector<pair<int, int> > &half_keys, const long double &h,
              unordered_map<pair<int, int>, long double[2],
                            FourierSeries::pair_hash> &derivatives) {
  for (const auto &z : half_keys) {
    complex<long double> bend_multiplier =
        32 * PI * PI * PI * PI * h * h *
        (z.first * z.first + z.second * z.second) *
        (z.first * z.first + z.second * z.second) * series.coefficients.at(z);

    derivatives[z][0] += bend_multiplier.real();
    derivatives[z][1] += bend_multiplier.imag();
  }
}

void d_e_height(const FourierSeries &series,
                const vector<pair<int, int> > &half_keys, const long double &h,
                const long double &p,
                unordered_map<pair<int, int>, long double[2],
                              FourierSeries::pair_hash> &derivatives) {
  for (const auto &z : half_keys) {
    complex<long double> height_multiplier =
        2 * pow(h, -p) * series.coefficients.at(z);

    // The ternary operator here is guarding against the case where we have a
    // constant term in the Fourier series.
    derivatives[z][0] +=
        (z.first != 0 || z.second != 0 ? 1 : 0.5) * height_multiplier.real();
    derivatives[z][1] += height_multiplier.imag();
  }
}

void update_coefficients(FourierSeries &series, long double &h,
                         unordered_map<pair<int, int>, long double[2],
                                       FourierSeries::pair_hash> &derivatives,
                         long double eta_divisor = 1) {
  long double eta, max_derivative = 0;

  // Find the largest magnitude of the partial derivatives.
  for (auto it = derivatives.cbegin(); it != derivatives.cend(); ++it) {
    for (size_t i = 0; i != 2; ++i) {
      if (abs(it->second[i]) > max_derivative) {
        max_derivative = abs(it->second[i]);
      }
    }
  }

  eta = 1 / max_derivative / eta_divisor;

  // Make the updates to the necessary coefficients.
  for (auto it = derivatives.cbegin(); it != derivatives.cend(); ++it) {
    // Conventient notation.
    auto z = it->first;
    pair<int, int> neg_z = pair<int, int>(-z.first, -z.second);
    long double &ddr = derivatives[z][0], &dds = derivatives[z][1];
```

```
    series.coefficients[z] -=
        eta * complex<long double>(ddr, dds) / (long double)(2);

    if (z.first != 0 || z.second != 0) {
      series.coefficients[neg_z] -=
          eta * complex<long double>(ddr, -dds) / (long double)(2);
    }
  }
}

void update_h(FourierSeries &series, long double &h, const long double &p) {
  h = pow((p * series.e_height()) / (2 * series.e_bend()), 1 / (p + 2));
}

// Do one iteration of stochastic gradient descent. Note that the
// unordered_map coefficients is changed. eta_divisor controls how large of
// a step to take (a larger value means a smaller step).
void sgd_step(FourierSeries &series, vector<pair<int, int> > &sum_keys,
              const vector<pair<int, int> > &half_keys,
              const long double M[2][2], long double &h, const long double &p,
              long double eta_divisor = 1) {
  random_shuffle(sum_keys.begin(), sum_keys.end());

  for (const auto &k : sum_keys) {
    if (k.first == 0 && k.second == 0) {
      continue;
    }

    complex<long double> scaled_hess_det_coef =
        (long double)(sum_keys.size()) /
        ((k.first * k.first + k.second * k.second) *
         (k.first * k.first + k.second * k.second)) *
        series.hessian_determinant_coefficient(k);

    // All of the derivatives must be stored at once to make the best gradient
    // approximation
    unordered_map<pair<int, int>, long double[2], FourierSeries::pair_hash>
        derivatives;

    // This loop calculates the derivatives of our objective function with
    // respect to r_z and s_z for every valid choice of z.
    for (const auto &z : half_keys) {
      // Auxillary variables to make notation better.
      complex<long double>
          at_k_minus_z = series.coefficients.count(pair<int, int>(
                             k.first - z.first, k.second - z.second))
                             ? series.coefficients[pair<int, int>(
                                   k.first - z.first, k.second - z.second)]
                             : 0,
          at_k_plus_z = series.coefficients.count(pair<int, int>(
                            k.first + z.first, k.second + z.second))
                            ? series.coefficients[pair<int, int>(
                                  k.first + z.first, k.second + z.second)]
                            : 0;

      complex<long double> stretch_multiplier =
          (long double)(k.first * z.second - k.second * z.first) *
          (k.first * z.second - k.second * z.first) * scaled_hess_det_coef;

      // Calculate the E_stretch derivatives.
      derivatives[z][0] +=
          (stretch_multiplier * conj(at_k_minus_z + at_k_plus_z)).real();
      derivatives[z][1] +=
          (stretch_multiplier * conj(at_k_minus_z - at_k_plus_z)).imag();
    }

    d_e_mat(series, half_keys, derivatives, M);
    d_e_bend(series, half_keys, h, derivatives);
```

```
    d_e_height(series, half_keys, h, p, derivatives);

    update_coefficients(series, h, derivatives, eta_divisor);
  }
}

// Do one iteration of  gradient descent. Note that the unordered_map
// coefficients is changed. eta_divisor controls how large of a step to take (a
// larger value means a smaller step).
void gd_step(FourierSeries &series, vector<pair<int, int> > &sum_keys,
             const vector<pair<int, int> > &half_keys,
             const long double M[2][2], long double &h, const long double &p,
             long double eta_divisor = 10) {
  unordered_map<pair<int, int>, long double[2], FourierSeries::pair_hash>
      derivatives;

  d_e_stretch(series, sum_keys, half_keys, derivatives);
  d_e_mat(series, half_keys, derivatives, M);
  d_e_bend(series, half_keys, h, derivatives);
  d_e_height(series, half_keys, h, p, derivatives);

  update_coefficients(series, h, derivatives, eta_divisor);
  // update_h(series, h, p);
}

// Do many iterations of  gradient descent. Repeat until either the objective
// function is less than max_e or until our learning rate is very small. The
// variable initial is the beginning value of eta_divisor. The variable initial
// is how much to multiply (divide) eta_divisor by when the current value is
// doing poorly (well).
void gd(FourierSeries &series, long double &h, const long double &p,
        bool verbose = false, long double max_e = 2, long double initial = 10,
        long double multiplier = 1.1, const long double M[2][2] = ID) {
  auto best_coefficients = series.coefficients;

  // These key lists are made here so that we don't have to recreate them every
  // time we run an iteration.
  auto sum_keys = series.sum_keys();
  auto half_keys = series.half_keys();

  long double eta_divisor = initial,
              e = series.e_stretch() + series.e_mat(ID) +
                  h * h * series.e_bend() + pow(h, -p) * series.e_height(),
              best_e = e, best_h = h;

  unsigned consecutive_correct = 0;

  if (verbose) {
    cout << "Initial e: " << e << '\n' << endl;
  }

  // Get a "good" start
  sgd_step(series, sum_keys, half_keys, M, h, p, eta_divisor);

  best_coefficients = series.coefficients;
  e = series.e_stretch() + series.e_mat(M) + h * h * series.e_bend() +
      pow(h, -p) * series.e_height();

  while (e > max_e && eta_divisor < 10000) {
    if (verbose) {
      cout << "eta_divisor = " << eta_divisor << '\n';
    }

    gd_step(series, sum_keys, half_keys, M, h, p, eta_divisor);

    long double e_stretch = series.e_stretch(), e_mat = series.e_mat(M),
                e_bend = series.e_bend(), e_height = series.e_height();
    e = e_stretch + e_mat + h * h * e_bend + pow(h, -p) * e_height;
```

```
    if (verbose) {
      cout << "\tE_stretch = " << e_stretch << "\n\tE_mat = " << e_mat
           << "\n\tE_bend = " << e_bend << "\n\tE_height = " << e_height
           << "\n\tsum = " << e << endl;
    }

    if (best_e > e * 1.0001) {
      best_coefficients = series.coefficients;
      best_e = e;
      best_h = h;
      ++consecutive_correct;

      // If we get many correct steps in a row, our step size might be too
      // large, meaning we should decrease eta_divisor.
      if (consecutive_correct == 4) {
        eta_divisor /= multiplier;
        consecutive_correct = 0;
      }
    } else {
      eta_divisor *= multiplier;
      consecutive_correct = 0;
    }

    series.coefficients = best_coefficients;
    h = best_h;
  }
}

int main(int argc, char **argv) {
  unordered_map<pair<int, int>, complex<long double>, FourierSeries::pair_hash>
      coefficients;
  long double max_e, rho1, rho2, h, p;
  auto M = ID;

  if (argc < 5 || argc > 6) {
    cout << "Usage: " << argv[0] << " rho1 rho2 h p [max_e]" << endl;
    exit(1);
  }

  if (argc == 6) {
    rho1 = stold(argv[1]);
    rho2 = stold(argv[2]);
    h = stold(argv[3]);
    p = stold(argv[4]);
    max_e = stold(argv[5]);
  } else {
    rho1 = stold(argv[1]);
    rho2 = stold(argv[2]);
    h = stold(argv[3]);
    p = stold(argv[4]);
    max_e = 1;
  }

  coefficients = FourierSeries::random_coefficients(rho1, rho2);
  auto series = FourierSeries(coefficients);
  ofstream ofs;
  ofs.precision(15);

  gd(series, h, p, true, max_e, 10, 1.1, M);

  for (size_t i = 0; i != 80; ++i) cout << '=';
  cout << endl;

  cout.precision(15);
  long double e_stretch = series.e_stretch(), e_mat = series.e_mat(ID),
              e_bend = series.e_bend(), e_height = series.e_height(),
              e = e_stretch + e_mat + h * h * e_bend + pow(h, -p) * e_height;
```

```
  cout << "Final results:" << endl;
  cout << "\tE_stretch = " << e_stretch << "\n\tE_mat = " << e_mat
       << "\n\tE_bend = " << e_bend << "\n\tE_height = " << e_height
       << "\n\tsum = " << e << endl
       << endl;

  // ofs.open("coefficients_" + to_string(rho1) + "_" + to_string(rho2) +
  // ".txt");
  ofs.open("coefficients.txt");

  ofs << "# ";
  for (int i = 0; i != argc; ++i) {
    ofs << argv[i] << ' ';
  }
  ofs << "{{" << M[0][0] << ", " << M[0][1] << "}, {" << M[1][0] << ", "
      << M[1][1] << "}}" << endl;

  ofs << "# Final results:" << endl
      << "#\t  E_stretch = " << e_stretch << "\n#\t  E_mat = " << e_mat
      << "\n#\t  E_bend = " << e_bend << "\n#\t  E_height = " << e_height
      << "\n#\t  h = " << h << "\n#\t  sum = " << e << endl
      << endl;

  ofs << series << endl;
  ofs.close();

  return 0;
}
```

6.4 plot.py

```
import copy
import matplotlib.pyplot as plt
import numpy as np
import sys

class FourierSeries:
    """Class representing a band-limited double Fourier series.

    Attributes:
        coefficients: Dictionary of coefficients. The value
            corresponding to the key (k1, k2) is the coefficient on
            the term exp(2 * pi * i * (k1 * x2 + k2 * x2)).
    """

    def __init__(self, coefficients):
        self._coefficients = copy.deepcopy(coefficients)

    def get_coefficients(self):
        return copy.deepcopy(self._coefficients)

    def plot(self, x1, x2, figure = None):
        """
        Plots the real part of the series evaluated on the rectangle
        spanned by ``x1`` and ``x2``. Both ranges are assumed to be
        equally spaced and increasing.

        Args:
            x1 (np.array): Range for the first coordinate.
            x2 (np.array): Range for the second coordinate.
        """

        x1, x2 = np.meshgrid(x1, x2, indexing = 'ij')

        x3 = np.zeros(x1.shape)
        for k, ak in self._coefficients.items():
            x3 += (ak * np.exp(2 * np.pi * 1j * (k[0] * x1 + k[1] * x2))).real
```

```
        plt.contourf(x1, x2, x3, 20, cmap='RdGy')
        plt.colorbar()

    def plot_hessian_determinant(self, x1, x2, figure = None):
        """
        Plots the real part of the Hessian determinant of the series
        evaluated on the rectangle spanned by ``x1`` and ``x2``. Both
        ranges are assumed to be equally spaced and increasing.

        Args:
            x1 (np.array): Range for the first coordinate.
            x2 (np.array): Range for the second coordinate.
        """

        x1, x2 = np.meshgrid(x1, x2, indexing = 'ij')

        h00 = np.zeros(x1.shape)
        h01 = np.zeros(x1.shape)
        h11 = np.zeros(x1.shape)

        for k, ak in self._coefficients.items():
            h00 += (-(2 * np.pi * k[0])**2 * ak
                    * np.exp(2 * np.pi * 1j * (k[0] * x1 + k[1] * x2))).real
            h01 += (-(2 * np.pi)**2 * k[0] * k[1] * ak
                    * np.exp(2 * np.pi * 1j * (k[0] * x1 + k[1] * x2))).real
            h11 += (-(2 * np.pi * k[1])**2 * ak
                    * np.exp(2 * np.pi * 1j * (k[0] * x1 + k[1] * x2))).real

        x3 = h00 * h11 - h01 * h01

        plt.contourf(x1, x2, x3, 20, cmap='RdGy')
        plt.colorbar()


if len(sys.argv) < 2:
    print("Must supply file")
    sys.exit()

x1 = np.arange(0, 2, .01)
x2 = np.arange(0, 2, .01)

with open(sys.argv[1], 'r') as file:
    coefficients = eval(file.read())

    for k, ak in sorted(coefficients.items(), key = lambda i : -abs(i[1])):
        if abs(ak) > 1e-3:
            print('{0:<24}{1}'.format((k[0]**2 + k[1]**2)**0.5, abs(ak)))
        # Uncomment the following two lines to ignore coefficients that are too
        # small
        """else:
            del(coefficients[k])"""

    series = FourierSeries(coefficients)

    plt.figure(1)
    series.plot(x1, x2)
    plt.figure(2)
    series.plot_hessian_determinant(x1, x2)
    plt.show()
```

References

[1] J J. Stoker. Developable surfaces in the large. *Communications on Pure and Applied Mathematics*, 14:627 – 635, 08 1961.

[2] Xi Chen and John W. Hutchinson. A family of herringbone patterns in thin films. 2004.