A negative result for hearing the shape of a triangle

A computer-assisted proof

Gerard Orriols Giménez

Joint Bachelor's thesis for the degrees of
Mathematics and Engineering Physics

Supervisor (Princeton University): Javier Gómez-Serrano
Supervisor (UPC): Xavier Cabré
Universitat Politècnica de Catalunya
May 2019

Abstract

We prove that there exist two distinct triangles for which the first, second and fourth eigenvalues of the Laplace operator with null Dirichlet boundary conditions coincide. This solves a conjecture raised by Antunes and Freitas and suggested by their informal numerical evidence. We use a novel technique for a computer-assisted proof about the spectrum of an operator, which combines a Finite Element Method, to locate roughly the first eigenvalues keeping track of their position in the spectrum, and the Method of Fundamental Solutions, to get a much more precise bound of these eigenvalues. Due to the time constraints, some of the computations still remain to be finished.

Keywords— computer-assisted proof, Laplace eigenvalues, spectral geometry, Finite Element Method, Method of Fundamental Solutions.

Contents

1 Introduction

In this thesis, we present a computer-assisted proof of the following original theorem.

Theorem 1. *The first, second and fourth eigenvalues of the Laplace operator on an Euclidean triangle with null Dirichlet boundary conditions are not enough to determine it up to isometry.*

This is a conjecture proposed by Antunes and Freitas in [AF11], suggested by numerical evidence, but a rigorous proof was required. The Dirichlet eigenvalues of the Laplace operator for a triangle Ω are real numbers λ such that there is a nonzero smooth function u defined on Ω and continuous on $\overline{\Omega}$ such that

$$\begin{cases} -\Delta u = \lambda u & \text{in } \Omega \\ u = 0 & \text{on } \partial\Omega. \end{cases}$$

It is a well known fact that the set of such λ forms an increasing sequence $0 < \lambda_1 < \lambda_2 \leq \lambda_3 \leq \cdots$ whose only limit point is ∞, an that the corresponding eigenfunctions u_j form an orthonormal basis of $L^2(\Omega)$. The eigenvalues of a domain are closely related with its geometric properties, constituting an active area of research called *spectral geometry*. A classical example of this relationship is Weyl's law, which relates the asymptotics of the eigenvalues to the volume of the domain, and a later result by McKean and Singer states that the perimeter is also determined by the eigenvalues [MS67]. More results of this kind can be found in [BG90] and [vdBS88]. Other results about how the geometry of a domain determines its spectrum can be found in Henrot's book [Hen06].

The question of the determination of a domain given the set of its Laplace eigenvalues was posed by Mark Kac in his famous paper "Can one hear the shape of a drum?" [Kac66]. Since then, the answer has been found to be negative in general; in particular, for euclidean polygons, the first example of a pair of non-isometric polygons with the same spectrum is due to Gordon, Webb and Wolpert [GWW92]. However, there are positive results when we restrict the determination to a class of domains, the most successful of which was found by Zelditch [Zel09], who proved spectral determination for analytic domains with two classes of symmetries.

Less is known about domains with less regular boundaries, the simplest of which are polygons. In the case of triangles, it has been proven that the whole spectrum of the Laplace operator determines the shape of a triangle ([Dur88], with a recent simple proof by [GM13]), and later Chang and De Turck proved that only a finite amount of eigenvalues, which depends on λ_1 and λ_2, is enough [CD89]. It is natural to try to improve the result to only a finite and fixed amount of eigenvalues, answering the question "Can a *human* hear the shape of a *triangular* drum?".

Since the space of triangles up to isometries is has dimension 3, we would expect that 3 eigenvalues should be enough, but a priori it is not clear which ones. Antunes and Freitas [AF11] conjectured that indeed the three first eigenvalues λ_1, λ_2 and λ_3 do determine the shape of a triangle. Instead, numerical evidence by themselves seems to indicate that this is not the case for λ_1, λ_2 and λ_4, and in this paper we will prove this fact (Theorem 1). This will give an example of an obstruction to determining the shape of a triangle from a finite portion of its spectrum.

2 Structure of the proof of Theorem 1

By the scaling of the problem, we reduce our search to the set of triangles with a fixed base length (together with additional conditions that ensure that we only consider one triangle for each similarity class); instead of looking for all three eigenvalues $\lambda_1, \lambda_2, \lambda_4$ to be equal, we just require the quotients $\xi_{21} = \lambda_2/\lambda_1$ and $\xi_{41} = \lambda_4/\lambda_1$ to take the same value. Since the eigenvalues scale by λr^{-2} when the lengths of a triangle are scaled by r, the two conditions are equivalent.

Fixing the first two vertices of the triangle to be $(0,0)$ and $(1,0)$, we use the coordinates (c_x, c_y) of the third vertex to parametrize the search space. Our approach consists in using a topological argument to show that in each of two disjoint regions in this parameter space there is a triangle in which ξ_{21} and ξ_{41} take the same prescribed value. More precisely, we claim that there are two distinct triangles for which $\xi_{21} = \bar{\xi}_{21} := 1.67675$ and $\xi_{41} = \bar{\xi}_{41} := 2.99372$.

Since rigorous calculations with the computer are done using interval arithmetic, we need a topological technique to transform the closed condition into an open condition which can be compatible with the intervals. For this we will use the Poincaré–Miranda theorem (see [Mir40]):

Theorem 2. *Given two continuous functions $f, g : [-1,1]^2 \to \mathbb{R}$ such that $f(x,y)$ has the same sign as x when $x = \pm 1$ and $g(x,y)$ has the same sign as y when $y = \pm 1$, there exists a point $(x,y) \in [-1,1]^2$ such that $f(x,y) = g(x,y) = 0$.*

The regions that we will consider are two parallelograms around the points $A = (0.63500, 0.27500)$ and $B = (0.84906, 0.31995)$, designed such that ξ_{21} and ξ_{41} have approximately a constant value each in a pair of opposite edges. Using the computer we will verify that the functions $\xi_{21} - \bar{\xi}_{21}$ and $\xi_{41} - \bar{\xi}_{41}$ each have a constant and opposite sign in opposite edges of the parallelogram, and hence by the theorem, together with the well known continuity of eigenvalues, we will conclude that such two distinct triangles exist.

The vectors defining the parallelogram are of course obtained from the inverse of an approximation of the differential of the $\mathbb{R}^2$-valued funcion (ξ_{21}, ξ_{41}) at the points A and B, reescaled so that the increment of ξ_{21} and ξ_{41} along each vector is approximately 0.01. Explicitly, they are given by:

- $v_{21,A} = (0.04269082311, 0.01148489707)$
- $v_{41,A} = (-0.02082984105, -0.00255790585)$
- $v_{21,B} = (0.01717048015, 0.01266844996)$
- $v_{41,B} = (0.03590363291, 0.01065148385)$

Moreover, for the point A in the negative $v_{21,A}$ direction, the vector is shrinked by a factor 0.6, because after that the behavior deviates from linear and the bound becomes worse after this displacement. Hence the vertices of the parallelograms will be

- $A + v_{21,A} + v_{41,A}$, $A + v_{21,A} - v_{41,A}$, $A - 0.6v_{21,A} - v_{41,A}$, $A - 0.6v_{21,A} + v_{41,A}$.
- $B + v_{21,B} + v_{41,B}$, $B + v_{21,B} - v_{41,B}$, $B - v_{21,B} - v_{41,B}$, $B - v_{21,B} + v_{41,B}$.

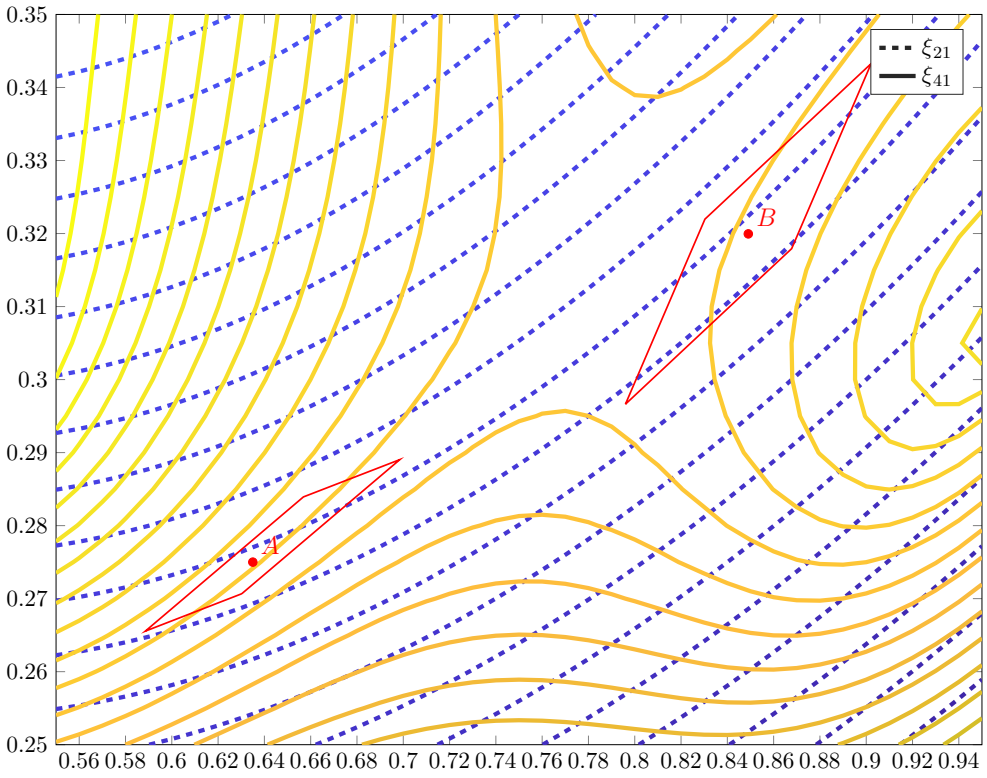

Figure 1: Numerical approximate plot of the quotients ξ_{21} (discontinuous lines) and ξ_{41} (continuous lines) around the region of interest. The points A and B and the validated parallelograms are shown in red.

This setup is displayed in Figure 1 together with a plot of non rigorous contour lines of the eigenvalue quotients.

The pointwise verification of the values ξ_{21} and ξ_{41}, which depends on an accurate calculation of λ_i for $i = 1, 2, 4$, consists of two steps. The first one, treated in Section 3, is about showing that the computed eigenvalues actually correspond to the ordered ones λ_1, λ_2 and λ_4; in order to do that, we will prove a lower bound for λ_5 combining techniques from the Finite Element Method with rigorous bounds linking the finite dimensional problem to the infinite dimensional one. The second step consists of finding accurate values of four eigenvalues that lie below the threshold obtained in the first part, which implies that they will indeed have to be the four lowest ones. This is done using the Method of Fundamental Solutions and recent rigorous bounds based on the L^2 norm of the boundary error, explained in Section 4.

We emphasize the difficulty of finding the order of an eigenvalue, which is a global problem, compared to the local easier task of refining its value. To the best knowledge of the author, this is the first computer-assisted proof in which these two distinct, local and global methods are used to verify eigenvalues of an operator.

In order to extend the computer-assisted pointwise verifications of the eigenvalues λ_1, λ_2 and λ_4 to the continuous region in which they need to be validated, we will use explicit continuity results of the spectrum of an elliptic operator with the domain to extend the bounds to a neighborhood of the verified points. The details of this part are explained in Section 5. Section 6 contains the details of the implementation and execution of the proof. Finally, the Appendix contains the codes used to verify an upper bound for the 4th eigenvalue at a point and to validate a segment of triangles.

We end this section by introducing the reader to how a computer-assisted proof works. In the recent years, the application of calculations done by computers to mathematical proofs have become more popular due to the increment of computational resources, but in order to make sure that their results are rigorous, we need to control the errors that floating point arithmetic can accumulate. This is usually done by means of interval arithmetic, in which the data that a computer stores for a real number is an interval (two endpoints, or a midpoint and a radius) of real numbers, stored by two floating point numbers, instead of just one.

Operations between intervals are implemented to return intervals which are guaranteed to contain every possible result when the operands belong to the input intervals. For example, if $[x] = [\underline{x}, \overline{x}]$ and $[y] = [\underline{y}, \overline{y}]$ are two intervals, their sum will can be given by the interval $[x] + [y] = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]$ and their product by $[x] \cdot [y] = [\min\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}, \max\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}]$. The same rule applies to function implementations: a function f evaluated on $[x]$ should return an interval containing every $f(x)$ for $x \in [x]$. We refer to the book [Tuc11] for an introduction to validated numerics, in which most of the techniques used here are explained, and to [GS18] for a more specific treatment of computer-assisted proofs in PDE.

3 Separation of the first four eigenvalues

In order to find a rigorous lower bound for the fifth eigenvalue of a triangle we will use a recent bound found by Liu [Liu15], which is similar to the one in [CG14] but simplifies the hypotheses and improves the constant. Both use the non-conforming Finite Element Method of Crouzeix–Raviart; other rigorous bounds with conforming finite elements were explored, like [LO13], but the bound is worse and the method is harder

to implement with validated numerics because its mass matrix is not diagonal.

The Crouzeix–Raviart finite-element method uses a triangulation of the domain Ω, which in our case we will take to be the trivial triangulation given by N^2 triangles with sides equal to $1/N$ of the original one and similar to it. The basis functions are indexed by interior edges of the triangulation: if E is a common edge of triangles T_1, T_2, the basis function ψ_E is the unique function supported on $T_1 \cup T_2$ such that restricted to each triangle is affine, takes the value 1 in the midpoint of E and the value 0 in the midpoints of the other edges of T_1 and T_2.

We define the coefficients of the stiffness and mass matrices $\boldsymbol{A} = (a_{EF}), \boldsymbol{B} = (b_{EF})$ by the bilinear forms

$$a_{EF} = \int_\Omega \nabla\psi_E \cdot \nabla\psi_F \qquad b_{EF} = \int_\Omega \psi_E\psi_F$$

For our choice of triangulation, $\boldsymbol{B}$ is simply a multiple of the identity $2|\Omega|\boldsymbol{I}/\left(3N^2\right)$, whereas $\boldsymbol{A}$ is a sparse matrix. This will allow us to work with a matrix eigenvalue problem instead of a generalized one. The main result that we will use is the following [Liu15, Theorem 2.1 and Remark 2.2]:

Theorem 3. *Consider a polygonal domain Ω with a triangulation so that each triangle has diameter at most h. Let λ_k be the k-th eigenvalue of Ω and $\lambda_{k,h}$ the k-th eigenvalue of the Crouzeix–Raviart discretized problem for Ω. Then*

$$\frac{\lambda_{h,k}}{1 + C_h^2\lambda_{h,k}} \leq \lambda_k, \tag{1}$$

where $C_h \leq 0.1893h$ is a constant.

In order to be able to deal with approximate eigenvalues we will need in addition the following lemma from [Par80, Theorem 15.9.1].

Lemma 4. *Let $(\tilde{\lambda}_h, \tilde{\boldsymbol{u}}_h)$ be an approximated algebraic eigenpair such that $\tilde{\lambda}_h$ is closer to some λ_h than to any other discrete eigenvalue. Suppose that the coefficient vector $\tilde{\boldsymbol{u}}_h$ is normalised with respect to $\boldsymbol{B}$, $\|\boldsymbol{B}\tilde{\boldsymbol{u}}_h\|_{\boldsymbol{B}^{-1}} = \|\tilde{\boldsymbol{u}}_h\|_{\boldsymbol{B}} = 1$. Then the algebraic residual $\boldsymbol{r} := \boldsymbol{A}\tilde{\boldsymbol{u}}_h - \tilde{\lambda}_h\boldsymbol{B}\tilde{\boldsymbol{u}}_h$ satisfies*

$$|\lambda_h - \tilde{\lambda}_h| \leq \|\boldsymbol{r}\|_{\boldsymbol{B}^{-1}}.$$

Remark 5. *We can combine Theorem 3 with Lemma 4 using the monotonicity of* (1)*, using $\lambda_{h,k} - \|\boldsymbol{r}\|_{\boldsymbol{B}^{-1}}$ as a lower bound of $\lambda_{h,k}$ instead.*

It is easy to obtain estimations $\tilde{\lambda}_h$ with a very small residual. The hardest part here before applying the theorem is to check that they have indeed the correct index, i.e., that they are closer to the appropriate λ_h than to any other discrete eigenvalue. This is why we need to control the whole spectrum of the discrete problem, and we will do that by applying Gershgorin's disks theorem after performing some Givens rotations to the matrix to get useful bounds.

More precisely, in order to get a lower bound for λ_5 we need to separate the first 5 eigenvalues from the rest so that we can control them, and in order to do that we will perform Givens rotations until the intervals provided by Gershgorin's theorem can be separated in two disjoint components, one containing the 5 smallest eigenvalues and the other containing the rest. If this holds, then the strong version of Gershgorin's

theorem will guarantee an upper bound for $\lambda_{h,5}$. Therefore, provided that the residuals are all very small and that all approximate eigenvalues are different (which happens in our setting), Lemma 4 will guarantee that there are 5 distinct discrete eigenvalues below the upper bound and therefore they will be forced to have the correct indices.

This allows us to verify $\lambda_{h,5}$ with an error only depending on its residual, and using Remark 5, get a lower bound of λ_5. Therefore, the first four eigenvalues can be separated just by checking that they are distinct and smaller than this lower bound.

4 Rigorous eigenvalue bounds for individual triangles

Our approach to find tight bounds for the eigenvalues of triangles uses the Method of Fundamental Solutions (MFS), introduced by Fox, Henrici and Moler in [FHM67] and more recently revived by Betcke and Trefethen [BT05]. In this method, a function u is written as a linear combination of functions ϕ_i $(1 \leq i \leq N)$ that satisfy pointwise the equation $(\Delta + \lambda)\phi_i = 0$ for a fixed λ. The coefficients are chosen to optimize the proximity of the function to the eigenspace for the actual eigenvalue λ_j, in a sense made precise in [BT05], and this is measured by the least singular value of a certain matrix that involves the values of u at discrete points of the boundary $\partial\Omega$. This parameter is minimized with respect to λ by using a golden ratio search. This provides a candidate $\lambda \in \mathbb{R}$ and coefficients c_i for which $u(x) = \sum_{i=1}^{N} c_i\phi_i(x)$ can be computed with arbitrary precision.

The functions that we will use for the MFS consist of two types: the first ones are of the form $\phi(x) = Y_0(\sqrt{\lambda}|x - x_0|)$, for x_0 a point outside Ω. The second type of functions are parametrized by a vertex of the triangle and a positive integer j, and take the form $\psi_j(r, \theta) = J_{j\alpha}(\sqrt{\lambda}r)\sin(j\alpha\theta)$, where (r, θ) are the polar coordinates of the point with respect to a vertex in the triangle whose total angle is π/α, and θ is measured from an adjacent side. The first kind of functions allow us to approximate the function in the interior of the triangle and near the sides, while the second kind gives the correct asymptotic behavior of the solution near the vertices of the triangle.

The main tool that we will use to find rigorous bounds for eigenvalues is the L^2 bound given by Barnett and Hassell [BH11]. However, their method is optimized for high eigenvalues, so we will have to adapt some of the steps to our case of small eigenvalues. We summarize the main results that we will use. Let Ω be a triangle and $u \in C^2(\Omega)$ be nonzero such that $(\Delta + \lambda)u = 0$. Consider the tension

$$t[u] = \frac{\|u\|_{L^2(\partial\Omega)}}{\|u\|_{L^2(\Omega)}}.$$

Let λ_j, u_j be the sequence of eigenvalues and eigenfunctions of Ω, satisfying $(\Delta + \lambda_j)u_j = 0$ with Dirichlet null boundary conditions. Let v_j be the normal derivative of u_j, defined on $\partial\Omega$. We define the operator

$$A(\lambda) = \sum_{\lambda_j} \frac{v_j\langle v_j, \cdot\rangle}{(\lambda - \lambda_j)^2},$$

and its decomposition as a sum of three:

$$A_{\text{near}}(\lambda) = \sum_{|\lambda-\lambda_j|\leq\sqrt{\lambda}} \frac{v_j\langle v_j, \cdot\rangle}{(\lambda - \lambda_j)^2},$$

$$A_{\text{far}}(\lambda) = \sum_{\lambda/2 \le \lambda_j \le 2\lambda, |\lambda - \lambda_j| > \sqrt{\lambda}} \frac{v_j \langle v_j, \cdot \rangle}{(\lambda - \lambda_j)^2},$$

$$A_{\text{tail}}(\lambda) = \sum_{\lambda_j < \lambda/2 \text{ or } \lambda_j > 2\lambda} \frac{v_j \langle v_j, \cdot \rangle}{(\lambda - \lambda_j)^2}$$

where $\langle \cdot, \cdot \rangle$ is the standard inner product on $L^2(\partial\Omega)$.

This operator is useful because its norm is controlled by the tension (see [BH11, Section 3]):

$$t[u]^{-2} \le \|A(\lambda)\|. \tag{2}$$

Moreover we have the following explicit bounds from [BH11, Lemma 4.1] and [BH11, Lemma 4.2]:

$$\|A_{\text{far}}(\lambda)\| \le C_1, \tag{3}$$

$$\|A_{\text{tail}}(\lambda)\| \le C_2 \lambda^{-1/2}, \tag{4}$$

with constants C_1, C_2 given below. For the near term, since we are working with very low eigenvalues, $\sqrt{\lambda}$ is actually small enough that only the summand with $\lambda_j = \lambda$ appears:

$$\|A_{\text{near}}(\lambda)\| = \frac{\|v_j\|^2_{L^2(\partial\Omega)}}{(\lambda - \lambda_j)^2}.$$

For convex domains, like in our case, Section 6 of [BH11] offers explicit bounds for the constants. In particular, they are based on another constant C_Ω coming from a quasi-orthogonality inequality (see [BH11, Theorem 1.3]) that we will not use directly. We will simply use the bounds $C_1 \le 7C_\Omega$ and $C_2 \le 7C_\Omega$ quoted from the end of page 1058 of the paper.

Finally, the constant C_Ω can be obtained from $\sup_{\partial\Omega}(\boldsymbol{x} \cdot \boldsymbol{n})$ and $\inf_{\partial\Omega}(\boldsymbol{x} \cdot \boldsymbol{n})$, as explained in pages 1058-1059 of the paper. Of course, since the problem is coordinate-invariant, we are free to chose an origin, and the optimal choice is the incenter of the triangle. In this case, both the supremum and the infimum are equal to the inradius ρ, so we can take $C_\Omega = 4(1 + \rho)/\rho$.

Putting this together we have

$$t[u]^{-2} \le \frac{\|v_j\|^2_{L^2(\partial\Omega)}}{(\lambda - \lambda_j)^2} + 7C_\Omega(1 + \lambda^{-1/2}). \tag{5}$$

Finally, recall Rellich's formula [Rel40]:

$$\int_{\partial\Omega} (\partial_n u_j)^2 \boldsymbol{x} \cdot \boldsymbol{n} \mathrm{d}s = 2\lambda_j.$$

For our choice of origin of coordinates, this just gives us $\|v_j\|^2_{L^2(\partial\Omega)} = 2\lambda_j/\rho$. Inserting this into (5) we have proved:

Proposition 6. *The distance d from λ to the spectrum of the Laplacian on Ω, can be bounded above by*

$$t[u]^{-2} \le \frac{2\tilde{\lambda_j}}{\rho d^2} + 7C_\Omega(1 + \lambda^{-1/2}).$$

where $\tilde{\lambda_j}$ is an upper bound for λ_j.

5 Extension of the bounds to a region of triangles

Our goal is to propagate the rigorous bounds of an eigenvalue of the Laplacian of a triangle with Dirichlet boundary conditions to a neighborhood of triangles. The main tool that we will use is an abstract theorem about continuity of eigenvalues with respect to the operator norm, which can be found at [Hen06, Theorem 2.3.1]:

Proposition 7. *Given two compact, self-adjoint and positive operators T, T' acting on a separable Hilbert space, the difference of their eigenvalues μ_n, μ'_n can be bounded as*

$$|\mu_n - \mu'_n| \leq \|T - T'\|,$$

where $\|\cdot\|$ is the L^2 operator norm.

To apply this to our setting, we suppose that the triangle Ω for which we have bounds has vertices $(0,0)$, $(1,0)$ and (c_x, c_y) and consider a small vector (d_x, d_y). The one-parameter family of affine transformations that keeps the first two vertices fixed and moves the third along the vector is given by

$$(x,y) \mapsto (x^t, y^t) = \left(x + \frac{ty}{c_y} d_x, y + \frac{ty}{c_y} d_y\right),$$

with $t = 0$ corresponding to the identity map. Let us denote by Ω^t the image of Ω by the transformation with parameter t and λ_n^t its Laplace eigenvalues. Our first lemma gives a bound for λ_n^t based on the eigenvalues of an explicit matrix.

Lemma 8. *Consider the matrix*

$$A^t = \begin{bmatrix} \frac{t^2 d_x^2}{(c_y + t d_y)^2} + 1 & -\frac{t b d_x}{(c_y + t d_y)^2} \\ -\frac{t b d_x}{(c_y + t d_y)^2} & \frac{c_y^2}{(c_y + t d_y)^2} \end{bmatrix}$$

and suppose that its eigenvalues lie in $[1-\delta, 1+\delta]$. Then

$$\frac{|\lambda_n - \lambda_n^t|}{\lambda_n \lambda_n^t} \leq \frac{\delta}{(1-\delta)(2\pi)^2}.$$

Proof. We have the change of coordinates

$$\begin{cases} \frac{\partial}{\partial x} &= \frac{\partial}{\partial x^t} \\ \frac{\partial}{\partial y} &= \frac{t d_x}{c_y} \frac{\partial}{\partial x^t} + \left(1 + \frac{t d_y}{c_y}\right) \frac{\partial}{\partial y^t} \end{cases}$$

which gives place to

$$\begin{cases} \frac{\partial}{\partial x^t} &= \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y^t} &= \frac{1}{1 + \frac{t d_y}{c_y}} \left(-\frac{t d_x}{c_y} \frac{\partial}{\partial x} + \frac{\partial}{\partial y}\right) \end{cases}.$$

Hence, given $u : \Omega \to \mathbb{R}$, we can introduce $u^t : \Omega^t \to \mathbb{R}$, $u^t(x^t, y^t) = u(x(x^t, y^t), y(x^t, y^t))$. We have that

$$\begin{aligned} \Delta u^t(x^t, y^t) &= \left(\frac{\partial}{\partial x^t}\right)^2 u^t + \left(\frac{\partial}{\partial y^t}\right)^2 u^t \\ &= \left(\frac{t^2 d_x^2}{(c_y + t d_y)^2} + 1\right) \frac{\partial^2 u}{\partial x^2} - 2 \frac{t b d_x}{(c_y + t d_y)^2} \frac{\partial^2 u}{\partial x \partial y} + \frac{c_y^2}{(c_y + t d_y)^2} \frac{\partial^2 u}{\partial y^2} \\ &=: \Delta^t u. \end{aligned}$$

This is an elliptic operator with constant coefficients, and can be written as $\Delta^t u = \nabla \cdot (A^t \nabla u)$ for the matrix A^t defined above. We will apply Proposition 7 to the operators $T = \Delta^{-1}$ and $T' = (\Delta^t)^{-1}$, so given a function $f \in L^2(\Omega)$ we need to bound $\|u - u^t\|_{L^2}$, where $\Delta u = \Delta^t u^t = f$ and $u, u^t = 0$ on $\partial\Omega$. Let Γ be the rectangle containing Ω with sides of lengths 1 and c_y parallel to the X and Y axes, and write

$$u^t(x) = \sum_k c_k e^{ik\cdot x},$$

where the sum is over (k_x, k_y) with $k_x, k_y c_y \in 2\pi\mathbb{Z}$. Then

$$f = \Delta^t u^t = -\sum_{k\neq 0} c_k (k^T A^t k) e^{ik\cdot x}$$

and

$$(\Delta^t - \Delta)\left(\Delta^t\right)^{-1} f = (\Delta^t - \Delta)u^t = \sum_{k\neq 0} c_k (k^2 - k^T A^t k) e^{ik\cdot x}$$

which implies

$$(\Delta^{-1} - (\Delta^t)^{-1})f = \Delta^{-1}(\Delta^t - \Delta)\left(\Delta^t\right)^{-1} f = \sum_{k\neq 0} c_k \left(1 - \frac{k^T A^t k}{k^2}\right) e^{ik\cdot x}.$$

By the condition on the eigenvalues of A^t, we have that $\frac{k^T A^t k}{k^2} \in [1-\delta, 1+\delta]$ too and

$$\|(\Delta^{-1} - (\Delta^t)^{-1})f\|_{L^2}^2 = \sum_{k\neq 0} |c_k|^2 \left(1 - \frac{k^T A^t k}{k^2}\right)^2 \leq \delta^2 \sum_{k\neq 0} |c_k|^2.$$

Now remember that, since we take the $k \neq 0$ in a lattice, $k^2 \geq (2\pi)^2$, and moreover $k^T A^t k \geq (1-\delta)k^2$. Putting this together,

$$\|(\Delta^{-1} - (\Delta^t)^{-1})f\|_{L^2}^2 \leq \delta^2 \sum_{k\neq 0} |c_k|^2 \leq \frac{\delta^2}{(1-\delta)^2(2\pi)^4} \sum_{k\neq 0} |c_k|^2 (k^T A^t k)^2 = \frac{\delta^2}{(1-\delta)^2(2\pi)^4} \|f\|_{L^2}^2.$$

Hence we get the bound

$$\|(\Delta^{-1} - (\Delta^t)^{-1})\| \leq \frac{\delta}{(1-\delta)(2\pi)^2}$$

which implies the desired inequality

$$\frac{|\lambda_n - \lambda_n^t|}{\lambda_n \lambda_n^t} \leq \frac{\delta}{(1-\delta)(2\pi)^2}.$$

□

The following lemma will give us an explicit upper bound for the δ above.

Lemma 9. *Suppose that $c_y > \max\{3|d_x|, 3|d_y|\}$. Then the δ from Lemma 8 can be taken as*

$$c_y \frac{d_x + 2|d_y|}{(c_y - d_y)^2}.$$

Proof. We will use Gershgorin's theorem for the matrix A^t. In particular, if we want a uniform delta for all $t \in [-1, 1]$, we want to take δ as the maximum of both of the following expressions for all $-1 \leq t \leq 1$:

$$\frac{t^2 d_x^2 + |tbd_x|}{(c_y + td_y)^2}, \quad \frac{|td_y(2b + td_y)| + |tbd_x|}{(c_y + td_y)^2}$$

Whenever d_x, d_y are small enough, these functions are monotonous on $[-1, 0]$ and $[0, 1]$, having a minimum at zero, so the maximum is at $t = \pm 1$. Without loss of generality we will assume that $d_x > 0$. For $t > 0$, the only zero of the derivative of the first function is at $t = c_y/(d_y - 2d_x)$ (if it is positive), so it increases on $[0, 1]$. Analogously, for $t < 0$ the only zero of the derivative is at $-t = -c_y/(d_y + 2d_x)$, so the same argument applies. The second function can be rewritten as

$$\frac{|t|c_y d_x + |t||d_y|(2c_y + td_y)}{(c_y + td_y)^2}.$$

For positive t it is monotonic until $t = c_y(d_x + 2|d_y|)/(d_x d_y)$ and for negative t it is until $-t = -c_y(d_x + 2|d_y|)/(d_x d_y)$, which are both greater than 1 in absolute value, so the maximum is also attained at $t = \pm 1$. In both cases, which extreme gives a larger value depends on the sign of d_y, giving the optimal δ as

$$\max\left\{\frac{d_x^2 + c_y d_x}{(c_y - |d_y|)^2}, \frac{c_y d_x + 2c_y|d_y| - |d_y|^2}{(c_y - |d_y|)^2}\right\}.$$

Finally, a simpler upper bound of this expression, which we will use for convenience, is

$$\delta = c_y \frac{d_x + 2|d_y|}{(c_y - |d_y|)^2}.$$

□

Finally we need to propagate the error to the quotient ξ_{n1}^t of λ_n^t and λ_1^t. For this, observe that the error obtained in Lemma 8 does not depend on the index of the eigenvalue, and call it ϵ.

Lemma 10. *Suppose that we have the bounds*

$$\frac{|\lambda_1 - \lambda_1^t|}{\lambda_1 \lambda_1^t} \leq \epsilon, \qquad \frac{|\lambda_n - \lambda_n^t|}{\lambda_n \lambda_n^t} \leq \epsilon.$$

Then,

$$|\xi_{n1} - \xi_{n1}^t| \leq \lambda_n \epsilon (1 + \xi_{n1}^t).$$

Proof. This is just a simple computation.

$$|\xi_{n1} - \xi_{n1}^t| = \frac{|\lambda_n^t \lambda_1 - \lambda_n \lambda_1^t|}{\lambda_1^t \lambda_1} \leq \frac{\lambda_1 |\lambda_n^t - \lambda_n|}{\lambda_1^t \lambda_1} + \frac{\lambda_n |\lambda_1^t - \lambda_1|}{\lambda_1^t \lambda_1} = \epsilon \frac{\lambda_n \lambda_n^t}{\lambda_1^t} + \epsilon \lambda_n = \lambda_n \epsilon (1 + \xi_{n1}^t)$$

□

In our approach we will proceed in the reverse way: we will obtain verified ξ_{n1} that will be above (or below) $\bar{\xi}_{n1}$, and will consider the longest vector $(d_x, d_y) = \ell(v_x, v_y)$ that we can move in a fixed directon (v_x, v_y) preserving the condition of being above (or below) $\bar{\xi}_{n1}$. The steps that we follow are the following,

for example, for an upper bound: first, we compute the margin M, which is a lower bound for $\xi_{n1} - \bar{\xi}_{n1}$. Then we compute the ϵ that will guarantee that ξ_{n1} fits in the margin, from Lemma 10:

$$\epsilon = \frac{M}{\lambda_n(1 + \xi_{n1} + M)}.$$

After this, we set δ so that the hypotheses of Lemma 10 hold:

$$\delta = \frac{(2\pi)^2\epsilon}{1 + (2\pi)^2\epsilon}.$$

Finally, we choose the step length ℓ that will guarantee δ to have the at most imposed value resulting from Lemma 9:

$$\delta = c_y \ell \frac{|v_x| + 2|v_y|}{(c_y - \ell|v_y|)^2}.$$

This is done by explicitly solving a quadratic equation.

6 Implementation

The validated computations are performed using the rigorous arithmetic library Arb, developed by Fredrik Johansson [Joh17], which can be found at `http://arblib.org`. Other non-rigorous computations are made using common libraries such as ALGLIB or Boost.

The key rigorous verifications are described in the following three subsections. At the time of the publishing of this thesis not all computations were finished because of time constraints. In particular, the verification of ξ_{41} in the two segments (one for A and the other one for B) in which it is greater than $\bar{\xi}_{41}$ remains to be completed. The computation of the lower bound for λ_5 has been done in many points around the perimeter of the parallelograms, but it remains to be checked that the bound applies to the whole perimeter.

The validation of one of the sides of a parallelogram can use approximately from 500 to 2000 points, and the total running time can take from 4 to 14 hours in 120 parallel machines, approximately.

6.1 Givens rotations

A key rigorous computation consists of applying the Givens rotations to the Crouzeix–Raviart matrix in order to separate the Gershgorin intervals into two components (see Section 3). The angle of the rotation is chosen in a non rigorous way, but the rotation is done using interval arithmetic. The rotations are performed by steps: in each step, several iterations are performed to reduce the upper bound of the lowest 5 Gershgorin intervals below a fixed threshold and to increase the lower bound of the highest Gershgorin intervals above another threshold. These thresholds are improved at each step progressively (the former is reduced, the latter is increased).

The iterations consist in making a Givens rotation to set to zero each off-diagonal entry whose absolute value exceeds the maximum Gershgorin radius allowed (i.e. the difference between the diagonal value and the current threshold) divided by the number of off-diagonal entries. This heuristic tolerates small absolute values in off-diagonal entries and stops when the Gershgorin radius reached is small enough.

The execution of this algorithm for our data requires a subdivision into N^2 triangles, for N between 18 and 21, and has a running time of between 15 and 45 minutes at a precision of 1024 bits. The results for the execution with $N = 15$ at triangle with vertex A are plotted to illustrate the method: Figure 2 shows the Gershgorin intervals before performing the rotations, Figure 3 shows them after performing them, and Figure 4 zooms into the second plot around the line of separation.

6.2 Upper bound of the MFS boundary norm

Another fundamental verified computation is the upper bound for the L^2 norm of the candidate eigenfunction at the boundary (see Section 4). This is done by dividing the sides of the triangle into many small intervals, in positions given by Chebyshev nodes, and in each of them performing a validated computation using Taylor series: the Taylor polynomial of the function at the center point is evaluated in the whole interval, and to this value the remainder of Taylor's theorem is added. A validated enclosure for this remainder consists of the Taylor polynomial of the function at the whole interval evaluated in the whole interval.

When this computation exceeds a threshold in absolute value (in our case, 10^{-5}), the interval is split in half and the validating function is called recursively for the two halves. In the end, all contributions from all intervals are added up to get the L^2 bound. This calculation takes approximately 10 minutes per point at a precision of 128 bits, using a total of 317 charge basis functions and 15 vertex basis functions.

6.3 Lower bound of the MFS interior norm

The last critical computation consists of getting a lower bound for the L^2 norm of the candidate eigenfunction in the interior (see Section 4). This is done by using a grid of 8×8 small triangles that occupy a smaller triangle of side 0.8 times the original one (hence the area of each triangle is $0.01|\Omega|$). Figure 5 displays the grid for the plot of the first and the fourth eigenfunction of triangle B. In each of the triangles a lower bound of the absolute value of u is obtained using the same method as above (Taylor series bounds and splitting recursively).

Whenever we obtain a validated estimate, say $u \geq a > 0$, on ∂T, where T is one of the small triangles in the grid, we can extend this inequality to the whole T by using the minimum principle. More precisely, it is enough to show that $-\Delta u \geq 0$ in T to get that the minimum of u is in ∂T and hence is at least a. If this does not hold, it means that $\lambda u = -\Delta u < 0$ at some point inside T. This means that the open set $U = \{u < 0\} \cap T \subset T$ has λ as a Dirichlet eigenvalue, and hence by the Faber-Krahn inequality, we get a lower bound for its area:

$$0.01|\Omega| = |T| \geq |U| \geq \frac{\pi j_{0,1}^2}{\lambda} > \frac{18.1684}{\lambda}.$$

This is a contradiction by orders of magnitude for our triangles ($|\Omega| \leq 0.25$ and $\lambda < 1000$).

The calculation of this part also takes approximately 10 minutes per point at a precision of 128 bits, using a total of 317 charge basis functions and 15 vertex basis functions.

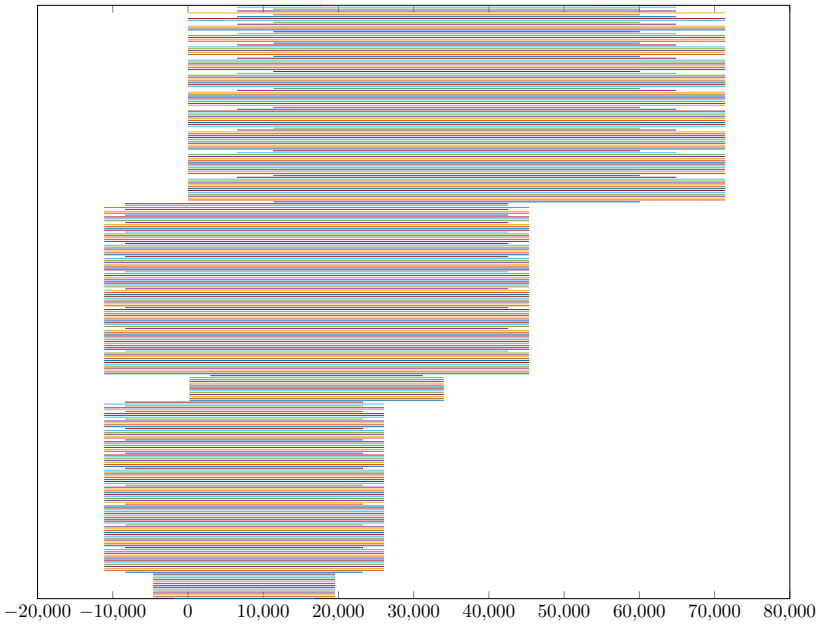

Figure 2: Gershgorin intervals before applying the Givens rotations for triangle A.

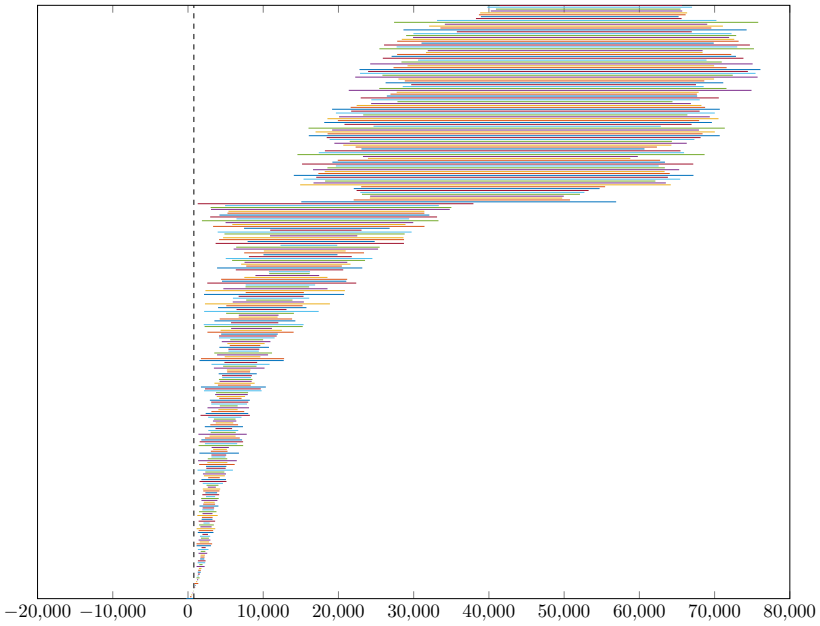

Figure 3: Gershgorin intervals after applying the Givens rotations for triangle A. The dashed vertical line at position 790 separates the first 5 eigenvalues from the rest.

Acknowledgments

First of all, I would like to thank my supervisor Javier Gómez-Serrano for hosting me at Princeton University, suggesting me the problem, providing me references, giving me a lot of guidance and sharing with me many

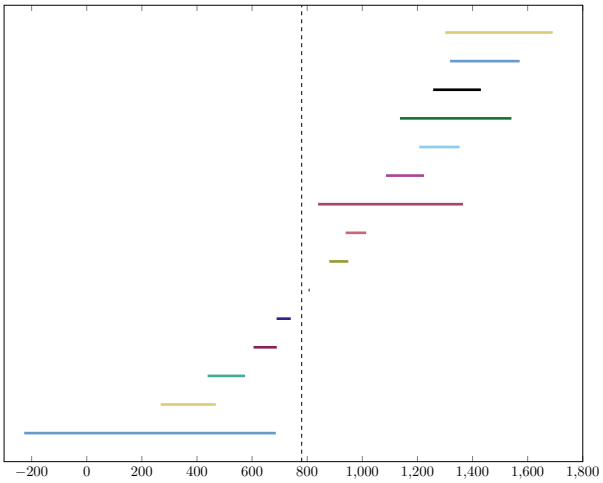

Figure 4: Zoom of the separation of the first 5 eigenvalues after the Givens rotations.

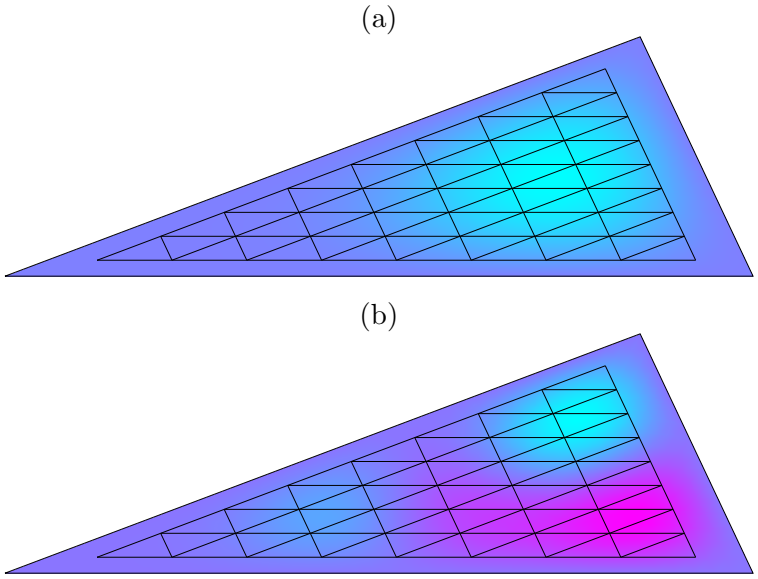

Figure 5: Grid used to validate a lower bound for $\|u\|_{L^2(\Omega)}$ for triangle B, with (a) the first eigenfunction and (b) the second eigenfunction plotted on top.

useful discussions. I am very grateful to the Mathematics Department of Princeton University, in particular the Graduate Program, for funding my university fee and allowing my stay as a visitor researcher. I would also like to thank CFIS and the administrators of the Mobility program for giving me the oportunity to do this research in another university and funding me. I also thank the MOBINT scholarship for providing partial financial support for my stay.

A Code for the separation of the smallest eigenvalues

This program reads the parameter N and the coordinates of the third vertex of the triangle and returns a lower bound for λ_5.

```
#include "arb.h"
#include "arb_mat.h"
#include "linalg.h"
#include "arbcc.h"

#include <cassert>
#include <cmath>
#include <iostream>
#include <map>
#include <vector>

// Compile with -DPREC=1024

using namespace std;
using namespace alglib;

int di[6] = {0, 0, 1, -1, 1, -1};
int dj[6] = {1, -1, 0, 0, -1, 1};

int side(int i, int j) {
  if (i % 2 == 1) return 2;
  if (j % 2 == 0) return 0;
  return 1;
}

double sq(double t) { return t * t; }

/* struct ArfReal definition omitted for brevity */

int n, m;

void givens(vector<vector<ArbReal> >& A, double lowbar, double upbar,
            int nsmall) {
  vector<pair<ArbReal, int> > srt(m);
  for (int i = 0; i < m; ++i) {
    srt[i] = make_pair(A[i][i], i);
```

```
}
sort(srt.begin(), srt.end());
vector<bool> small(m, false);
for (int i = 0; i < nsmall; ++i) {
  small[srt[i].second] = true;
}
bool done = false;
int it = 0;
while (!done and it < 10) {
  ++it;
  done = true;
  for (int p = 0; p < m; ++p) {
    ArbReal rad = 0.;
    for (int q = 0; q < m; ++q) {
      if (q != p) {
        rad = rad + A[q][p].abs();
      }
    }
    ArbReal margin;
    if (small[p]) {
      margin = ArbReal(upbar) - A[p][p];
    } else {
      margin = A[p][p] - ArbReal(lowbar);
    }
    if (rad < margin) {
      continue;
    }
    done = false;
    printf("it %d: changing column %d with margin ", it, p);
    margin.print();
    for (int q = 0; q < m; ++q) {
      if (q != p && margin / (m - 1) < A[p][q].abs()) {
        ArfReal tt = (ArfReal(A[q][q]) - ArfReal(A[p][p])) /
                     (ArfReal(A[p][q]) * ArfReal(2));
        ArfReal t = (tt).sign() / (tt.abs() + (tt.sq() + 1).sqrt());
        ArbReal t_arb(t.x);
        ArbReal c = 1_a / (t_arb.sq() + 1_a).sqrt();
        ArbReal s = c * t_arb;
        for (int i = 0; i < m; ++i) {
          ArbReal u = A[p][i];
```

```
          ArbReal v = A[q][i];
          A[p][i] = c * u - s * v;
          A[q][i] = c * v + s * u;
        }
        for (int i = 0; i < m; ++i) {
          ArbReal u = A[i][p];
          ArbReal v = A[i][q];
          A[i][p] = c * u - s * v;
          A[i][q] = c * v + s * u;
        }
      }
    }
  }
}

vector<pair<ArbReal, ArbReal> > gershgorin(vector<vector<ArbReal> >& A) {
  vector<pair<ArbReal, ArbReal> > ret(m);
  for (int i = 0; i < m; ++i) {
    ret[i].first = A[i][i];
    ret[i].second = 0.;
    for (int j = 0; j < m; ++j) {
      if (j != i) {
        ret[i].second = ret[i].second + A[j][i].abs();
      }
    }
  }
  sort(ret.begin(), ret.end());
  return ret;
}

int main() {
  ios::sync_with_stdio(false);
  cout.setf(ios::fixed);
  cout.precision(10);

  cin >> n;
  double x_s, y_s;
  cin >> x_s >> y_s;
  ArbReal x(x_s), y(y_s);
```

```
ArbReal area = y / 2_a;
ArbReal l_a = (y * y + (x - 1_a) * (x - 1_a)).sqrt();
ArbReal l_b = (y * y + x * x).sqrt();
ArbReal l_c = 1;
ArbReal len[3];
len[0] = l_a;
len[1] = l_b;
len[2] = l_c;
ArbReal h_a = y / l_a;
ArbReal h_b = y / l_b;
ArbReal h_c = y;
ArbReal height[3];
height[0] = h_a;
height[1] = h_b;
height[2] = h_c;

ArbReal cos_a = x / l_b;
ArbReal cos_b = (1_a - x) / l_a;
ArbReal cos_c = (x * (x - 1_a) + y * y) / (l_a * l_b);
ArbReal cosine[3];
cosine[0] = cos_a;
cosine[1] = cos_b;
cosine[2] = cos_c;

map<pair<int, int>, int> encode;
for (int i = 0; i < 2 * n - 2; ++i) {
  for (int j = 0; i + j < 2 * n - 2; ++j) {
    if (i % 2 == 0 or j % 2 == 0) {
      encode[make_pair(i, j)] = m++;
    }
  }
}

vector<vector<ArbReal> > mat(m, vector<ArbReal>(m));

ArbReal mass_diag = (area * 2) / (3 * n * n);

for (int i = 0; i < 2 * n - 2; ++i) {
  for (int j = 0; i + j < 2 * n - 2; ++j) {
```

```
    if (i % 2 == 1 and j % 2 == 1) continue;

    int u = encode[make_pair(i, j)];
    int side_u = side(i, j);
    mat[u][u] = (area * 8) / height[side_u].sq();
    for (int d = 0; d < 6; ++d) {
      int i2 = i + di[d];
      int j2 = j + dj[d];
      if (i2 >= 0 and j2 >= 0 and i2 + j2 < 2 * n - 2 and
          (i2 % 2 == 0 or j2 % 2 == 0)) {
        int v = encode[make_pair(i2, j2)];
        int side_v = side(i2, j2);
        ArbReal tmp = -(cosine[3 - side_u - side_v] * area * 4) /
                      (height[side_u] * height[side_v]);
        mat[u][v] = tmp;
        mat[v][u] = tmp;
      }
    }
  }
}

for (int i = 0; i < m; ++i) {
  for (int j = 0; j < m; ++j) {
    mat[i][j] = mat[i][j] / mass_diag;
  }
}

real_2d_array mat_d;
mat_d.setlength(m, m);

for (int i = 0; i < m; ++i) {
  for (int j = 0; j < m; ++j) {
    mat_d(i, j) = mat[i][j].get_approx_double();
  }
}

ae_int_t num_eigs_found;
real_1d_array eigs_d;
real_2d_array eigenvectors;
```

```
assert(smatrixevdr(mat_d, m, 1, 1, 0.0, 1000.0, num_eigs_found, eigs_d,
                   eigenvectors));
cerr << "Found " << num_eigs_found << " below 1000\n";

vector<ArbReal> discrete_eigs_lb(num_eigs_found);
for (int i = 0; i < num_eigs_found; ++i) {
  cout << eigs_d(i) << endl;
  ArbReal lambda(eigs_d(i));
  vector<ArbReal> vec(m);
  for (int j = 0; j < m; ++j) {
    vec[j] = ArbReal(eigenvectors(j, i));
  }
  ArbReal num = 0., den = 0.;
  for (int j = 0; j < m; ++j) {
    ArbReal residue = -lambda * vec[j];
    for (int k = 0; k < m; ++k) {
      residue = residue + ArbReal(mat[j][k]) * vec[k];
    }
    num = num + residue.sq();
    den = den + vec[j].sq();
  }
  ArbReal error = (num / den).sqrt();
  printf("Error: ");
  error.print();
  discrete_eigs_lb[i] = lambda - error;
}

assert(num_eigs_found >= 6);
double barrier = (eigs_d[4] + eigs_d[5]) / 2;

// Successively apply Givens rotations to reduce the margin.
// It does not work if one tries to separate them with the
// barrier from the beginning.
givens(mat, -100000, 1000000, 10);
givens(mat, -80000, 1000000, 10);
givens(mat, -60000, 1000000, 10);
givens(mat, -50000, 1000000, 10);
givens(mat, -20000, 1000000, 10);
givens(mat, -10000, 1000000, 10);
givens(mat, -1000, 100000, 10);
```

```
  givens(mat, 0, 100000, 10);
  givens(mat, 1000, 10000, 10);
  givens(mat, 1000, 2000, 10);
  givens(mat, barrier, 2000, 5);
  givens(mat, barrier, barrier, 5);

  printf("Discrete lambda_5 successfully separated\n");

  vector<pair<ArbReal, ArbReal> > ger = gershgorin(mat);

  ArbReal liu_constant = 0.1893;
  ArbReal lambda5lb =
      discrete_eigs_lb[4] /
      (1_a + discrete_eigs_lb[4] * liu_constant.sq() / ArbReal(n * n));

  printf("Verified lower bound for lambda_5: ");
  lambda5lb.print();
}
```

B Code for validating an interval of triangles

This program reads the name of the third vertex (A or B), the indexs of the eigenvalue quotient to validate ("21" or "41"), a letter standing for the sign ('p' for plus, 'm' for minus) of $\xi - \bar{\xi}$ that we have to validate, the current coordinate c, and the total number of coordinates n_c in which this side is divided ($c \in \{1, \ldots, n_c\}$). This will validate a segment around the position given by $2(c-0.5)/n_c - 1$ in the corresponding side of the parallelogram (where the position is normalized so that vertices are at positions -1 and 1), and the radius ℓ of this segment will be the value returned by the program.

For example, if the input is "B 21 p 312 1000" and the program returns a value of 0.0013, this means that for all the points in the segment between $B + v_{21,B} - 0.1898 v_{41,B}$ and $B + v_{21,B} - 0.1872 v_{41,B}$ the corresponding ξ_{21} has been verified to be greater than $\bar{\xi}_{21}$.

The program uses the `ArbTaylor` class, which is not included here for brevity, but computes simultaneously the Taylor polynomial and its residue, and uses them to obtain an enclosure of the function in an interval, as described in [Tuc11].

```
#include "arb.h"
#include "arb_hypgeom.h"
#include "arbcc.h"
#include "arbseries.h"
#include "arbtaylor.h"
```

```
#include <algorithm>
#include <cassert>
#include <fstream>
#include <iostream>
#include <map>
#include <vector>

// Compile with -DPREC=128

using namespace std;

// Declared here and implemented in a separate file.
void fill_in_vectors(vector<double>& r_coefs, double& r_lambda,
                     vector<vector<double>>& r_sources, double cx0, double cy0,
                     double lmin, double lmax);

namespace {

// Coordinates of the third vertex. Set as global because they are used
// everywhere.
ArbReal cx, cy;

/* GENERAL FUNCTIONS */

// Fundamental solution around an external charge point `(xs, ys)`.
ArbTaylor fund_sol_charge(const ArbTaylor& x, const ArbTaylor& y,
                          const ArbReal& xs, const ArbReal& ys,
                          const ArbReal& lambda, int order) {
  ArbTaylor arg = (x - ArbTaylor::constant(xs, order)).sq() +
                  (y - ArbTaylor::constant(ys, order)).sq();
  arg = arg.sqrt();
  arg *= lambda.sqrt();
  return arg.bessely0();
}

// Angle between two vectors.
ArbReal ccw_angle(const ArbReal& x1, const ArbReal& y1, const ArbReal& x2,
                  const ArbReal& y2) {
  ArbReal r = (x1.sq() + y1.sq()).sqrt();
  ArbReal c = x1 / r;
```

```
  ArbReal s = y1 / r;
  ArbReal xx = c * x2 + s * y2;
  ArbReal yy = -s * x2 + c * y2;
  ArbReal ang;
  arb_atan2(ang, yy, xx, PREC);
  return ang;
}

// Angle between two vectors, ArbTaylor version.
ArbTaylor ccw_angle_taylor(const ArbReal& x1, const ArbReal& y1,
                           const ArbTaylor& x2, const ArbTaylor& y2,
                           int order) {
  ArbReal r = (x1.sq() + y1.sq()).sqrt();
  ArbReal c = x1 / r;
  ArbReal s = y1 / r;

  ArbTaylor xx = x2 * c + y2 * s;
  ArbTaylor yy = x2 * (-s) + y2 * c;
  ArbTaylor ang = ArbTaylor::atan2(yy, xx);
  return ang;
}

// Fundamental solution around a vertex of the triangle.
ArbTaylor fund_sol_vertex(ArbTaylor x, ArbTaylor y, int v, int j,
                          const ArbReal& lambda, int order) {
  ArbReal xvert[3] = {0_a, 1_a, cx};
  ArbReal yvert[3] = {0_a, 0_a, cy};
  int nxt = (v + 1) % 3;
  int prv = (v + 2) % 3;
  ArbReal xv = xvert[v];
  ArbReal yv = yvert[v];
  ArbReal vnxt[2] = {xvert[nxt] - xv, yvert[nxt] - yv};
  ArbReal vprv[2] = {xvert[prv] - xv, yvert[prv] - yv};
  ArbReal thmx = ccw_angle(vnxt[0], vnxt[1], vprv[0], vprv[1]);
  ArbReal alpha = ArbReal::pi() * ArbReal(j + 1) / thmx;

  x -= ArbTaylor::constant(xv, order);
  y -= ArbTaylor::constant(yv, order);

  ArbTaylor th = ccw_angle_taylor(vnxt[0], vnxt[1], x, y, order);
```

```
  th *= alpha;

  ArbTaylor arg = x.sq() + y.sq();
  arg = arg.sqrt();
  arg *= lambda.sqrt();

  return arg.besselj(alpha) * th.sin();
}

// Evaluate the solution at a point.
ArbTaylor solution(const vector<ArbReal>& coef, const ArbReal& lambda,
                   const vector<vector<ArbReal>>& sources, const ArbTaylor& x,
                   const ArbTaylor& y, int v, int order) {
  int nst = sources.size();
  int n = (coef.size() - nst) / 3;

  ArbTaylor sum(order);

  if (v != -1) {
    // Point on side (v)--(v+1).
    int vv = (v + 2) % 3;
    for (int j = 0; j < n; ++j) {
      ArbTaylor incr = fund_sol_vertex(x, y, vv, j, lambda, order);
      incr *= coef[vv * n + j];
      sum += incr;
    }
  } else {
    for (int vv = 0; vv < 3; ++vv) {
      for (int j = 0; j < n; ++j) {
        ArbTaylor incr = fund_sol_vertex(x, y, vv, j, lambda, order);
        incr *= coef[vv * n + j];
        sum += incr;
      }
    }
  }
  for (int j = 0; j < nst; ++j) {
    ArbTaylor incr =
        fund_sol_charge(x, y, sources[j][0], sources[j][1], lambda, order);
    incr *= coef[3 * n + j];
    sum += incr;
```

```
  }

  return sum;
}

/* FUNCTIONS FOR THE L2 UPPER BOUND ON THE BOUNDARY: */

// Recursively compute an upper bound for the squared L2 norm of the solution at
// the boundary of the triangle. This returns the bound between positions
// num/den and (num+1)/den of side `v`.
ArbReal dfs(const vector<ArbReal>& coef, const ArbReal& lambda,
            const vector<vector<ArbReal>>& sources, long long num,
            long long den, arf_t goal, int v) {
  ArbReal xvert[3] = {0_a, 1_a, cx};
  ArbReal yvert[3] = {0_a, 0_a, cy};
  int nxt = (v + 1) % 3;
  // Chebyshev nodes.
  ArbReal ratio0 =
      0.5_a * (1_a - (ArbReal::pi() * ArbReal(num) / ArbReal(den)).cos());
  ArbReal x0 = (1_a - ratio0) * xvert[v] + ratio0 * xvert[nxt];
  ArbReal y0 = (1_a - ratio0) * yvert[v] + ratio0 * yvert[nxt];
  ArbReal ratio1 =
      0.5_a * (1_a - (ArbReal::pi() * ArbReal(num + 1) / ArbReal(den)).cos());
  ArbReal x1 = (1_a - ratio1) * xvert[v] + ratio1 * xvert[nxt];
  ArbReal y1 = (1_a - ratio1) * yvert[v] + ratio1 * yvert[nxt];
  int order = 8;

  ArbTaylor t(0_a, 1_a, order);

  ArbTaylor x =
      t * ((x1 - x0) / 2_a) + ArbTaylor::constant((x0 + x1) / 2_a, order);
  ArbTaylor y =
      t * ((y1 - y0) / 2_a) + ArbTaylor::constant((y0 + y1) / 2_a, order);

  ArbTaylor sol = solution(coef, lambda, sources, x, y, v, order);
  ArbReal val = sol.evaluate();
  arf_t abs_val;
  arf_init(abs_val);
  arb_get_abs_ubound_arf(abs_val, val, PREC);
```

```
  ArbReal ret;
  ArbReal len = ((x0 - x1).sq() + (y0 - y1).sq()).sqrt();

  if (arb_is_finite(val) && (arf_cmp(abs_val, goal) <= 0 || den >= 1024)) {
    ret = ArbReal(abs_val).sq() * len;
  } else {
    // Either denominator limit not reached, or reached but still have no finite
    // bound.
    ArbReal inc1 = dfs(coef, lambda, sources, 2 * num, 2 * den, goal, v);
    ArbReal inc2 = dfs(coef, lambda, sources, 2 * num + 1, 2 * den, goal, v);
    ret = inc1 + inc2;
  }
  arf_clear(abs_val);
  return ret;
}

// Upper bound of the L2 norm of the solution at the sides of the triangle.
// Computed recursively.
ArbReal side_l2_rec(const vector<ArbReal>& coef, const ArbReal& lambda,
                    const vector<vector<ArbReal>>& sources, arf_t goal) {
  const int den = 128;
  ArbReal sumsq = 0_a;
  for (int v = 0; v < 3; ++v) {
    for (int num = 0; num < den; ++num) {
      ArbReal incr = dfs(coef, lambda, sources, num, den, goal, v);
      sumsq += incr;
    }
  }
  return sumsq.sqrt();
}

/* FUNCTIONS FOR THE L2 LOWER BOUND IN THE INTERIOR: */

// Gives an interval between positions num/den and (num+1)/den in the path given
// by (xx[i], yy[i]).
// Requires: den is multiple of xx.size()-1=yy.size()-1. Add xx[last] = xx[0].
pair<ArbTaylor, ArbTaylor> parametrize(const vector<ArbReal>& xx,
                                       const vector<ArbReal>& yy, long long num,
                                       long long den, int order) {
  int nint = xx.size() - 1;
```

```
  int div = den / nint;
  int itv = num / div;
  int idx = num % div;
  ArbReal ratio0 =
      0.5_a * (1_a - (ArbReal::pi() * ArbReal(idx) / ArbReal(div)).cos());
  ArbReal x0 = (1_a - ratio0) * xx[itv] + ratio0 * xx[itv + 1];
  ArbReal y0 = (1_a - ratio0) * yy[itv] + ratio0 * yy[itv + 1];
  ArbReal ratio1 =
      0.5_a * (1_a - (ArbReal::pi() * ArbReal(idx + 1) / ArbReal(div)).cos());
  ArbReal x1 = (1_a - ratio1) * xx[itv] + ratio1 * xx[itv + 1];
  ArbReal y1 = (1_a - ratio1) * yy[itv] + ratio1 * yy[itv + 1];

  ArbTaylor t(0_a, 1_a, order);
  ArbTaylor x =
      t * ((x1 - x0) / 2_a) + ArbTaylor::constant((x0 + x1) / 2_a, order);
  ArbTaylor y =
      t * ((y1 - y0) / 2_a) + ArbTaylor::constant((y0 + y1) / 2_a, order);

  return {x, y};
}

pair<ArbReal, ArbReal> point_from_coord(int i, int j, int n,
                                        const ArbReal& fr) {
  ArbReal u = ArbReal(i) / ArbReal(n);
  ArbReal v = ArbReal(j) / ArbReal(n);
  // Gives the following point, where A, B, C are the vertices and G the
  // barycenter.
  // u * (B + fr * (G-B)) + v * (C + fr * (G-C)) + (1-u-v) * (A + fr * (G-A))
  return make_pair(
      (fr * u + (1_a - fr) / 3_a) + (fr * v + (1_a - fr) / 3_a) * cx,
      (fr * v + (1_a - fr) / 3_a) * cy);
}

pair<vector<ArbReal>, vector<ArbReal>> get_triangle(int count, int n,
                                                    const ArbReal& fr) {
  int i = count / n;
  int j = count % n;
  if (i + j < n) {
    pair<ArbReal, ArbReal> a = point_from_coord(i, j, n, fr),
                           b = point_from_coord(i + 1, j, n, fr),
```

```
                        c = point_from_coord(i, j + 1, n, fr);
    return {{a.first, b.first, c.first, a.first},
            {a.second, b.second, c.second, a.second}};
  } else {
    i = n - 1 - i;
    j = n - j;
    pair<ArbReal, ArbReal> a = point_from_coord(i, j, n, fr),
                        b = point_from_coord(i + 1, j, n, fr),
                        c = point_from_coord(i + 1, j - 1, n, fr);
    return {{a.first, b.first, c.first, a.first},
            {a.second, b.second, c.second, a.second}};
  }
}

// Gives the lower bound in the piece of the path between num/den and
// (num+1)/den.
ArbReal path_dfs(const vector<ArbReal>& coef, const ArbReal& lambda,
                 const vector<vector<ArbReal>>& sources, long long num,
                 long long den, mag_t err_goal, const vector<ArbReal>& xx,
                 const vector<ArbReal>& yy) {
  if (den > (1ll << 30)) {
    printf("Recursion limit reached, function too irregular. Aborting\n");
    abort();
  }
  int order = 8;
  auto p = parametrize(xx, yy, num, den, order);
  ArbTaylor x = p.first;
  ArbTaylor y = p.second;

  ArbReal val = solution(coef, lambda, sources, x, y, -1, order).evaluate();
  val = val.abs();
  mag_t err_val;
  mag_init(err_val);
  mag_set(err_val, arb_radref((arb_struct*)val));
  ArbReal ans;
  // Do not split if the value is precise enough.
  if (arb_is_finite(val) && mag_cmp(err_val, err_goal) <= 0) {
    arf_t aux;
    arf_init(aux);
    arb_get_abs_lbound_arf(aux, val, PREC);
```

```
    ans = ArbReal(aux);
    arf_clear(aux);
  } else {
    ArbReal val1 =
        path_dfs(coef, lambda, sources, 2 * num, 2 * den, err_goal, xx, yy);
    ArbReal val2 =
        path_dfs(coef, lambda, sources, 2 * num + 1, 2 * den, err_goal, xx, yy);
    arb_min(ans, val1, val2, PREC);
  }
  mag_clear(err_val);
  return ans;
}

// Minimum of u^2 in the path given by (xx[i], yy[i]).
ArbReal path_minsq(const vector<ArbReal>& coef, const ArbReal& lambda,
                   const vector<vector<ArbReal>>& sources, double goal_d,
                   const vector<ArbReal>& xx, const vector<ArbReal>& yy) {
  mag_t err_goal;
  mag_init(err_goal);
  mag_set_d(err_goal, goal_d);

  int den = 18;
  ArbReal lb;
  arb_pos_inf(lb);
  for (int num = 0; num < den; ++num) {
    ArbReal val =
        path_dfs(coef, lambda, sources, num, den, err_goal, xx, yy).sq();
    arb_min(lb, lb, val, PREC);
  }
  mag_clear(err_goal);
  return lb;
}

ArbReal compute_l2meanlb(const vector<ArbReal>& coef, ArbReal lambda,
                         const vector<vector<ArbReal>>& sources) {
  double goal = 5e-2;

  int nt = 8; // Bounding L2 norm using 8*8 triangles,
  ArbReal l2sum = 0_a;
  ArbReal fr = 0.8; // Using 0.64 of the total area to avoid the borders.
```

```
  for (int counter = 0; counter < nt * nt; ++counter) {
    auto path = get_triangle(counter, nt, fr);
    ArbReal pmsq =
        path_minsq(coef, lambda, sources, goal, path.first, path.second);
    l2sum += pmsq * fr.sq() / ArbReal(nt).sq();
  }
  ArbReal l2meanlb = l2sum.sqrt();
  printf("Lower bound of the L2 mean: ");
  l2meanlb.print();
  return l2meanlb;
}

/* COMPUTE THE EIGENVALUE WITH AN ABSOLUTE ERROR */

// Returns the pair {lambda, abs_err}, with lambda in [lbound, ubound]. `index`
// specifies the position of the eigenvalue in the spectrum (1,2,...).
pair<ArbReal, ArbReal> compute_all(int index, double lbound, double ubound) {
  vector<double> coef_d;
  vector<vector<double>> sources_d;
  double lambda_d;

  fill_in_vectors(coef_d, lambda_d, sources_d, cx.get_approx_double(),
                  cy.get_approx_double(), lbound, ubound);

  int nc = coef_d.size();
  int ns = sources_d.size();
  vector<ArbReal> coef(nc);
  vector<vector<ArbReal>> sources(ns, vector<ArbReal>(2));
  ArbReal lambda = lambda_d;
  for (int i = 0; i < nc; ++i) {
    coef[i] = ArbReal(coef_d[i]);
  }
  for (int i = 0; i < ns; ++i) {
    sources[i][0] = ArbReal(sources_d[i][0]);
    sources[i][1] = ArbReal(sources_d[i][1]);
  }

  printf("Candidate lambda: ");
  lambda.print();
```

```
ArbReal meanl2lb = compute_l2meanlb(coef, lambda, sources);

double goal_d = 1e-5;
arf_t goal;
arf_init(goal);
arf_set_d(goal, goal_d);
ArbReal l2bdry = side_l2_rec(coef, lambda, sources, goal);
printf("Validated at goal %e\n", arf_get_d(goal, 10));
printf("L2 norm at the boundary: ");
l2bdry.print();

ArbReal area = cy / 2;
ArbReal totall2lb = meanl2lb * area.sqrt();
ArbReal tension = l2bdry / totall2lb;

ArbReal side_a = (cy.sq() + (1_a - cx).sq()).sqrt();
ArbReal side_b = (cy.sq() + cx.sq()).sqrt();
ArbReal semip = (side_a + side_b + 1_a) / 2_a;
ArbReal inrad =
    ((semip - side_a) * (semip - side_b) * (semip - 1_a) / semip).sqrt();
ArbReal c_omega = 4_a * (1_a + inrad) / inrad;

ArbReal ub_all_lambda = 800_a; // Rough upper bound for all lambda (1 to 4).
ArbReal a_far = 7_a * c_omega;
ArbReal a_tail = 7_a * c_omega / lambda.sqrt();
ArbReal aux = 1_a / tension.sq();

aux -= a_far + a_tail;
ArbReal abs_err = 2_a * ArbReal(ubound) / (aux * inrad);
assert(0_a < abs_err);
abs_err = abs_err.sqrt();

printf(
    "Rigorous value of lambda obtained. Follows lambda, its tension and its "
    "absolute error.\n");
lambda.print();
tension.print();
abs_err.print();
arf_clear(goal);
return {lambda, abs_err};
```

```
}

} // namespace

int main() {
  char ptname, signc;
  string indexs;
  int coord, ncoord;
  cin >> ptname >> indexs >> signc >> coord >> ncoord;
  ArbReal cx0, cy0, v21x, v21y, v41x, v41y, position;

  // Heuristics to find the appropritate eigenvalue.
  // This will be validated later by a separate program.
  vector<double> lmin, lmax;

  if (ptname == 'A') {
    // Coordinates:
    cx0 = 0.635;
    cy0 = 0.275;
    // Vectors of the parallelogram:
    v21x = 4.269082311683548 / 100;
    v21y = 1.148489707350921 / 100;
    v41x = -2.082984105473002 / 100;
    v41y = -0.255790585210996 / 100;
    if (indexs == "21") {
      position = (ArbReal(coord) - 0.5_a) / ArbReal(ncoord) * 2_a - 1_a;
      lmin = {180, 320};
      lmax = {280, 450};
    } else {
      // In this case, the parallelogram is shrinked from below:
      // the position takes values in [-0.6, 1].
      position = (ArbReal(coord) - 0.5_a) / ArbReal(ncoord) * 1.6_a - 0.6_a;
      lmin = {180, 620};
      lmax = {280, 727. - (2 * coord - ncoord) / ncoord * 27};
    }
  } else {
    cx0 = 0.849057346949971;
    cy0 = 0.319950941965592;
    v21x = 1.717048015465781 / 100;
    v21y = 1.266844996461298 / 100;
```

```
  v41x = 3.590363291549854 / 100;
  v41y = 1.065148385561495 / 100;
  position = (ArbReal(coord) - 0.5_a) / ArbReal(ncoord) * 2_a - 1_a;
  if (indexs == "21") {
    if (signc == 'm') {
      lmin = {160, 280};
      lmax = {255, 380};
    } else {
      lmin = {150, 280};
      lmax = {190, 380};
    }
  } else {
    if (signc == 'm') {
      double center_l1 = 214 - coord * 23.0 / ncoord;
      double center_l4 = 637 - coord * 65.0 / ncoord;
      lmin = {center_l1 - 20, center_l4 - 20};
      lmax = {center_l1 + 20, center_l4 + 20};
    } else {
      double center_l1 = 197 - coord * 20.0 / ncoord;
      double center_l4 = 593 - coord * 60.0 / ncoord;
      lmin = {center_l1 - 20, center_l4 - 20};
      lmax = {center_l1 + 20, center_l4 + 20};
    }
  }
}

if (indexs == "21") {
  if (signc == 'm') {
    cx = cx0 - v21x + position * v41x;
    cy = cy0 - v21y + position * v41y;
  } else {
    cx = cx0 + v21x + position * v41x;
    cy = cy0 + v21y + position * v41y;
  }
} else {
  if (signc == 'm') {
    cx = cx0 + position * v21x - v41x;
    cy = cy0 + position * v21y - v41y;
  } else {
    cx = cx0 + position * v21x + v41x;
```

```
    cy = cy0 + position * v21y + v41y;
  }
}

string filename = "output_";
filename += ptname;
filename += indexs;
filename += signc;
filename += to_string(coord);
filename += "of";
filename += to_string(ncoord);
filename += ".txt";
freopen(filename.c_str(), "w", stdout);

auto p1 = compute_all(1, lmin[0], lmax[0]);
auto pk = compute_all((indexs == "21" ? 2 : 4), lmin[1], lmax[1]);

ArbReal xi = pk.first / p1.first;
ArbReal errxi = (pk.second + p1.second * xi) / (p1.first - p1.second);
printf("Xi and error:\n");
xi.print();
errxi.print();

ArbReal goalxi = (indexs == "21" ? 1.67675 : 2.99372);
ArbReal margin = (signc == 'p' ? xi - goalxi - errxi : goalxi - xi - errxi);
ArbReal eps = margin / (pk.first * (1_a + xi + margin));
ArbReal delta = ((2_a * ArbReal::pi()).sq() * eps) /
              (1_a + (2_a * ArbReal::pi()).sq() * eps);
ArbReal cc, kk; // Auxilliary variables to calculate ell.
if (indexs == "21") {
  cc = cy * (v41x.abs() + 2_a * v41y.abs());
  kk = v41y.abs();
} else {
  cc = cy * (v21x.abs() + 2_a * v21y.abs());
  kk = v21y.abs();
}
// Length validated at this point through propagation of the error.
// Given as the coefficient of the vector v21 or v41 that one can move.
ArbReal ell = (2_a * delta * cy * kk + cc -
             (4_a * delta * cy * kk * cc + cc.sq()).sqrt()) /
```

```
              (2_a * delta * kk.sq());
  if (ptname == 'A' && indexs == "41") {
    // Renormalize ell so it represents a length in [-1, 1]:
    ell *= 1.25;
  }
  printf("Value of ell: ");
  ell.print();
}
```

C Code for optimizing the eigenvalue

This code implements the function `fill_in_vectors` declared in the file above, which finds an approximate eigenvalue and eigenvector in a non-rigorous fashion. It does so by optimizing the smallest nonzero singular value of a matrix that imposes the boundary conditions, as explained above.

```
#include <stdio.h>
#include <sys/time.h>
#include <algorithm>
#include <utility>
#include "linalg.h"
#include "stdafx.h"
#include "optimization.h"
#include "specialfunctions.h"
#include <boost/math/special_functions/bessel.hpp>
#include <boost/math/tools/minima.hpp>

using namespace alglib;
using namespace std;

namespace {

double cx, cy;

inline double sq(double x) { return x * x; }

double norm(const real_1d_array& a) {
  double s = 0;
  for (int i = 0; i < a.length(); ++i) {
    s += sq(a[i]);
  }
```

```
  return sqrt(s);
}

real_1d_array operator*(const real_1d_array& a, double x) {
  real_1d_array c;
  int n = a.length();
  c.setlength(n);
  for (int i = 0; i < n; ++i) {
    c[i] = a[i] * x;
  }
  return c;
}

real_1d_array operator/(const real_1d_array& a, double x) {
  real_1d_array c;
  int n = a.length();
  c.setlength(n);
  for (int i = 0; i < n; ++i) {
    c[i] = a[i] / x;
  }
  return c;
}

real_1d_array operator+(const real_1d_array& a, const real_1d_array& b) {
  real_1d_array c;
  int n = a.length();
  c.setlength(n);
  for (int i = 0; i < n; ++i) {
    c[i] = a[i] + b[i];
  }
  return c;
}

real_1d_array operator-(const real_1d_array& a, const real_1d_array& b) {
  real_1d_array c;
  int n = a.length();
  c.setlength(n);
  for (int i = 0; i < n; ++i) {
    c[i] = a[i] - b[i];
  }
```

```
  return c;
}

double fund_sol_charge_d(double x, double y, double x0, double y0,
                         double lambda) {
  double r = sqrt(sq(x - x0) + sq(y - y0));
  return bessely0(r * sqrt(lambda));
}

double fund_sol_vertex_d(double x, double y, int v, int j, double lambda) {
  double xvert[3] = {0, 1, cx};
  double yvert[3] = {0, 0, cy};
  int nxt = (v + 1) % 3;
  int prv = (v + 2) % 3;
  double xv = xvert[v];
  double yv = yvert[v];
  double vnxt_[2] = {xvert[nxt] - xv, yvert[nxt] - yv};
  double vprv_[2] = {xvert[prv] - xv, yvert[prv] - yv};
  double vcur_[2] = {x - xv, y - yv};
  real_1d_array vnxt, vprv, vcur;
  vnxt.setcontent(2, vnxt_);
  vprv.setcontent(2, vprv_);
  vcur.setcontent(2, vcur_);
  double thmx = atan2(vprv[1], vprv[0]) - atan2(vnxt[1], vnxt[0]);
  if (thmx < 0) {
    thmx += 2 * M_PI;
  }
  if (thmx > 2 * M_PI) {
    thmx -= 2 * M_PI;
  }
  double th = atan2(vcur[1], vcur[0]) - atan2(vnxt[1], vnxt[0]);
  if (th < 0) {
    th += 2 * M_PI;
  }
  if (th > 2 * M_PI) {
    th -= 2 * M_PI;
  }
  double r = norm(vcur);
  double alpha = M_PI * (j + 1) / thmx;
  double sl = sqrt(lambda);
```

```
  return boost::math::cyl_bessel_j(alpha, sl * r) * sin(alpha * th);
}

// Generates source points: `ns` per side and an aditional `nextra`
// situated around the top vertex. `nextra` should be odd.
real_2d_array gen_sources(int ns, int nextra) {
  double beta = 0.01;
  real_2d_array sources;
  sources.setlength(3 * ns + nextra, 2);

  double xvert[3] = {0, 1, cx};
  double yvert[3] = {0, 0, cy};
  for (int v = 0; v < 3; ++v) {
    real_1d_array vcur, vnxt;
    double vcur_[2] = {xvert[v], yvert[v]};
    vcur.setcontent(2, vcur_);
    int nxt = (v + 1) % 3;
    double vnxt_[2] = {xvert[nxt], yvert[nxt]};
    vnxt.setcontent(2, vnxt_);
    for (int i = 0; i < ns; ++i) {
      double rat = 0.5 * (1.0 - cos(double(i + 1) * M_PI / double(ns)));
      real_1d_array pt = vcur * (1.0 - rat) + vnxt * rat;
      real_1d_array vec = vnxt - vcur;
      vec = vec / norm(vec);
      double perp_[2] = {vec[1], -vec[0]};
      real_1d_array perp;
      perp.setcontent(2, perp_);
      for (int c = 0; c < 2; ++c) {
        sources(v * ns + i, c) = pt[c] + beta * perp[c];
      }
    }
  }
  for (int i = 0; i < nextra; ++i) {
    sources(3 * ns + i, 0) = cx + (double(i) - (nextra - 1) / 2) * 0.02;
    sources(3 * ns + i, 1) = cy + 0.1;
  }
  return sources;
}

// Generates `ni` interior points at random.
```

```
real_2d_array gen_intpts(int ni) {
  real_2d_array intpts;
  intpts.setlength(ni, 2);
  for (int i = 0; i < ni; ++i) {
    double c1 = randomreal();
    double c2 = randomreal();
    double c3 = randomreal();
    double s = c1 + c2 + c3;
    c1 /= s;
    c2 /= s;
    c3 /= s;
    intpts(i, 0) = c1 * 0 + c2 * 1 + c3 * cx;
    intpts(i, 1) = c1 * 0 + c2 * 0 + c3 * cy;
  }
  return intpts;
}

real_2d_array fb_matrix(double lambda, int nb, const real_2d_array& intpts,
                        int n, const real_2d_array& sources) {
  double xvert[3] = {0, 1, cx};
  double yvert[3] = {0, 0, cy};
  int ni = intpts.rows();
  int nst = sources.rows();
  real_2d_array A;
  A.setlength(3 * nb + ni, 3 * n + nst);

  for (int v = 0; v < 3; ++v) {
    double vcur_[2] = {xvert[v], yvert[v]};
    int nxt = (v + 1) % 3;
    double vnxt_[2] = {xvert[nxt], yvert[nxt]};
    real_1d_array vcur, vnxt;
    vcur.setcontent(2, vcur_);
    vnxt.setcontent(2, vnxt_);
    for (int i = 0; i < nb; ++i) {
      double rat = 0.5 * (1 - cos(double(i) * M_PI / nb));
      real_1d_array pt = vcur * (1 - rat) + vnxt * rat;
      double x = pt[0], y = pt[1];
      for (int vv = 0; vv < 3; ++vv) {
        for (int j = 0; j < n; ++j) {
          A(v * nb + i, vv * n + j) = fund_sol_vertex_d(x, y, vv, j, lambda);
```

```
      }
    }
  }
}

for (int i = 0; i < ni; ++i) {
  double x = intpts(i, 0);
  double y = intpts(i, 1);
  for (int vv = 0; vv < 3; ++vv) {
    for (int j = 0; j < n; ++j) {
      A(3 * nb + i, vv * n + j) = fund_sol_vertex_d(x, y, vv, j, lambda);
    }
  }
}

for (int v = 0; v < 3; ++v) {
  double vcur_[2] = {xvert[v], yvert[v]};
  int nxt = (v + 1) % 3;
  double vnxt_[2] = {xvert[nxt], yvert[nxt]};
  real_1d_array vcur, vnxt;
  vcur.setcontent(2, vcur_);
  vnxt.setcontent(2, vnxt_);
  for (int i = 0; i < nb; ++i) {
    double rat = 0.5 * (1 - cos(double(i) * M_PI / nb));
    real_1d_array pt = vcur * (1 - rat) + vnxt * rat;
    double x = pt[0], y = pt[1];
    for (int j = 0; j < nst; ++j) {
      A(v * nb + i, 3 * n + j) =
          fund_sol_charge_d(x, y, sources(j, 0), sources(j, 1), lambda);
    }
  }
}

for (int i = 0; i < ni; ++i) {
  double x = intpts(i, 0);
  double y = intpts(i, 1);
  for (int j = 0; j < nst; ++j) {
    A(3 * nb + i, 3 * n + j) =
        fund_sol_charge_d(x, y, sources(j, 0), sources(j, 1), lambda);
  }
```

```
  }

  return A;
}

// Minimum nonzero singular value of the Fourier-Bessel matrix.
double fb_minsv(double lambda, int nb, const real_2d_array& intpts, int n,
                const real_2d_array& sources) {
  real_2d_array mat = fb_matrix(lambda, nb, intpts, n, sources);
  real_1d_array tau;
  int mr = mat.rows(), mc = mat.cols();
  rmatrixqr(mat, mr, mc, tau);
  real_2d_array Q, Qtop;
  rmatrixqrunpackq(mat, mr, mc, tau, mc, Q);
  Qtop.setlength(3 * nb, mc);
  for (int i = 0; i < 3 * nb; ++i) {
    for (int j = 0; j < mc; ++j) {
      Qtop(i, j) = Q(i, j);
    }
  }

  real_1d_array singvals;
  real_2d_array unused;

  rmatrixsvd(Qtop, 3 * nb, mc, 0, 0, 2, singvals, unused, unused);
  return singvals[mc - 1];
}

void compute_eig_and_coefs(int n, int nb, int ni, int ns, double lmin,
                           double lmax, vector<double>& r_coefs,
                           double& r_lambda,
                           vector<vector<double>>& r_sources) {
  real_2d_array intpts = gen_intpts(ni);
  real_2d_array sources =
      gen_sources(ns, /*number of sources at the vertex=*/17);

  pair<double, double> opt =
      boost::math::tools::brent_find_minima([&](double l) {
        double val = fb_minsv(l, nb, intpts, n, sources);
        return val;
```

```
    }, lmin, lmax, /*bits of precision=*/25);
double lambda = opt.first;
double sv = opt.second;
printf("Lambda = %12.8f with singular value = %12.8f\n", lambda, sv);

real_2d_array mat = fb_matrix(lambda, nb, intpts, n, sources);
real_1d_array tau;
int mr = mat.rows(), mc = mat.cols();
rmatrixqr(mat, mr, mc, tau);
real_2d_array Q, R, RR, Qtop;
rmatrixqrunpackq(mat, mr, mc, tau, mc, Q);
rmatrixqrunpackr(mat, mr, mc, RR);
R.setlength(mc, mc);
for (int i = 0; i < mc; ++i) {
  for (int j = 0; j < mc; ++j) {
    R(i, j) = RR(i, j);
  }
}
Qtop.setlength(3 * nb, mc);
for (int i = 0; i < 3 * nb; ++i) {
  for (int j = 0; j < mc; ++j) {
    Qtop(i, j) = Q(i, j);
  }
}

real_1d_array singvals;
real_2d_array unused, vt;

rmatrixsvd(Qtop, 3 * nb, mc, 0, 1, 2, singvals, unused, vt);
real_2d_array v;
v.setlength(mc, 1);
for (int i = 0; i < mc; ++i) {
  v(i, 0) = vt(mc - 1, i);
}
rmatrixlefttrsm(mc, 1, R, 0, 0, 1, 0, 0, v, 0, 0);
real_1d_array coef;
coef.setlength(mc);
for (int i = 0; i < mc; ++i) {
  coef[i] = v(i, 0);
}
```

```
  r_lambda = lambda;
  r_coefs = vector<double>(mc);
  for (int i = 0; i < mc; ++i) {
    r_coefs[i] = coef[i];
  }
  r_sources = vector<vector<double>>(sources.rows(), vector<double>(2));
  for (int i = 0; i < sources.rows(); ++i) {
    r_sources[i][0] = sources(i, 0);
    r_sources[i][1] = sources(i, 1);
  }
}

} // namespace

void fill_in_vectors(vector<double>& r_coefs, double& r_lambda,
                     vector<vector<double>>& r_sources, double cx0, double cy0,
                     double lmin, double lmax) {
  printf("cx = %.10f, cy = %.10f\n", cx0, cy0);

  // Parameters for the basis:
  int n, nb, ni, ns;
  n = 5;
  nb = 300;
  ni = 40;
  ns = 100;

  cx = cx0;
  cy = cy0;

  compute_eig_and_coefs(n, nb, ni, ns, lmin, lmax, r_coefs, r_lambda,
                        r_sources);
}
```

References

[AF11] P. R. S. Antunes and P. Freitas. On the inverse spectral problem for Euclidean triangles. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 467(2130):1546–1562, 2011.

[BG90] T. P. Branson and P. B. Gilkey. The asymptotics of the Laplacian on a manifold with boundary. *Comm. Partial Differential Equations*, 15(2):245–272, 1990.

[BH11] A. Barnett and A. Hassell. Boundary quasi-orthogonality and sharp inclusion bounds for large Dirichlet eigenvalues. *SIAM Journal on Numerical Analysis*, 49(3):1046–1063, 2011.

[BT05] T. Betcke and L. N. Trefethen. Reviving the Method of Particular Solutions. *SIAM Review*, 47(3):469–491, 2005.

[CD89] P. Chang and D. DeTurck. On hearing the shape of a triangle. *Proceedings of the American Mathematical Society*, 105(4):1033–1038, 1989.

[CG14] C. Carstensen and J. Gedicke. Guaranteed lower bounds for eigenvalues. *Mathematics of Computation*, 83(290):2605–2629, 2014.

[Dur88] C. Durso. *On the inverse spectral problem for polygonal domains*. PhD thesis, Massachusetts Institute of Technology, 1988.

[FHM67] L. Fox, P. Henrici, and C. Moler. Approximations and bounds for eigenvalues of elliptic operators. *SIAM Journal on Numerical Analysis*, 4(1):89–102, 1967.

[GM13] D. Grieser and S. Maronna. Hearing the shape of a triangle. *Notices of the AMS*, 60(11):1440–1447, 2013.

[GS18] J. Gómez-Serrano. Computer-assisted proofs in PDE: a survey. *SeMA Journal*, pages 1–26, 2018.

[GWW92] C. Gordon, D. L. Webb, and S. Wolpert. One cannot hear the shape of a drum. *Bulletin of the American Mathematical Society*, 27(1):134–138, 1992.

[Hen06] A. Henrot. *Extremum problems for eigenvalues of elliptic operators*. Springer Science & Business Media, 2006.

[Joh17] F. Johansson. Arb: efficient arbitrary-precision midpoint-radius interval arithmetic. *IEEE Transactions on Computers*, 66:1281–1292, 2017.

[Kac66] M. Kac. Can one hear the shape of a drum? *The American Mathematical Monthly*, 73(4):1–23, 1966.

[Liu15] X. Liu. A framework of verified eigenvalue bounds for self-adjoint differential operators. *Applied Mathematics and Computation*, 267:341–355, 2015.

[LO13] X. Liu and S. Oishi. Verified eigenvalue evaluation for the laplacian over polygonal domains of arbitrary shape. *SIAM Journal on Numerical Analysis*, 51(3):1634–1654, 2013.

[Mir40] C. Miranda. Un'osservazione su un teorema di Brouwer. *Boll. Unione Mat. Ital., II. Ser.*, 3:5–7, 1940.

[MS67] H. P. McKean, Jr. and I. M. Singer. Curvature and the eigenvalues of the Laplacian. *J. Differential Geometry*, 1(1):43–69, 1967.

[Par80] B. N. Parlett. *The Symmetric Eigenvalue Problem*. Classics in Applied Mathematics. Society for Industrial and Applied Mathematics, 1980.

[Rel40] F. Rellich. Darstellung der Eigenwerte von $\Delta u + \lambda u = 0$ durch ein Randintegral. *Math. Z.*, 46:635–636, 1940.

[Tuc11] W. Tucker. *Validated numerics: a short introduction to rigorous computations*. Princeton University Press, 2011.

[vdBS88] M. van den Berg and S. Srisatkunarajah. Heat equation for a region in $\mathbf{R}^2$ with a polygonal boundary. *J. London Math. Soc. (2)*, 37(1):119–127, 1988.

[Zel09] S. Zelditch. Inverse spectral problem for analytic domains. II. $\mathbb{Z}_2$-symmetric domains. *Ann. of Math. (2)*, 170(1):205–269, 2009.