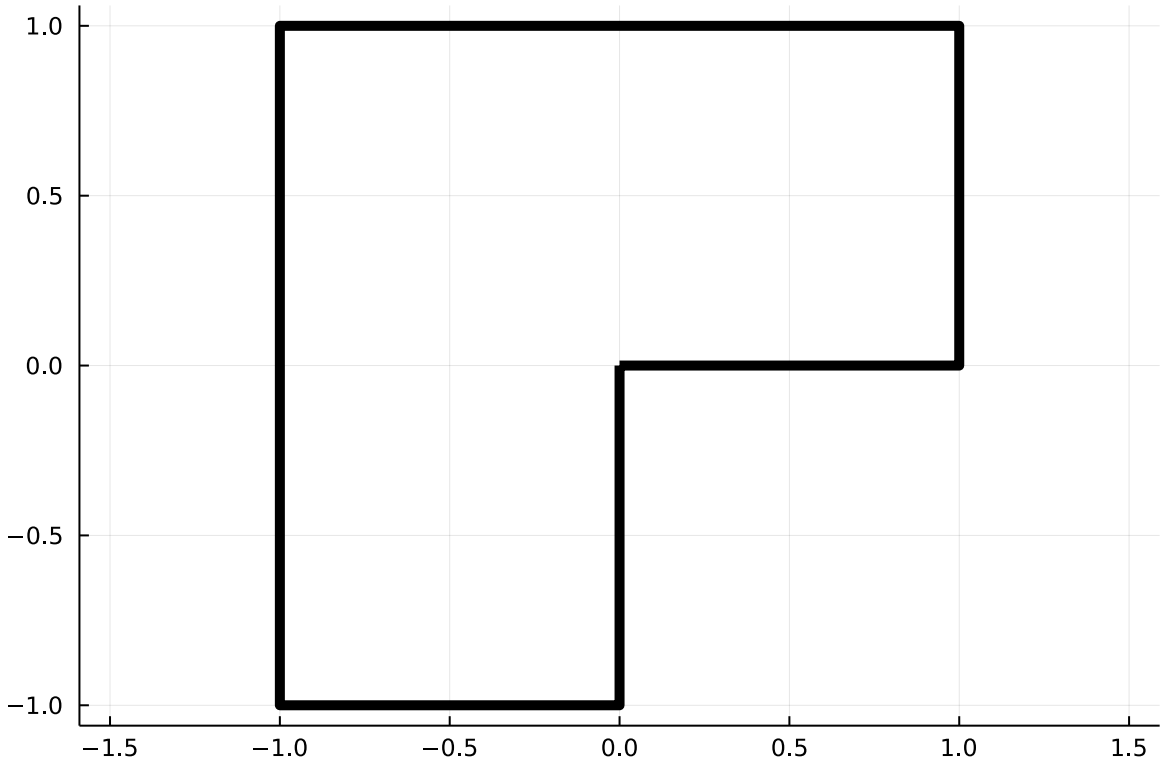
Elliptic PDEs

In this notebook we give an example where we compute an encloser of the first eigenvalue of the so called L-shaped domain. This procedure very much follows that in Dahne, J., & Salvy, B., Computation of tight enclosures for laplacian eigenvalues, SIAM Journal on Scientific Computing, 42(5), 3210–3232 (2020). http://dx.doi.org/10.1137/20m1326520



For this we will make use of `Arblib.jl` since `IntervalArithmetic.jl` doesn't support the Bessel functions, which we will make use of. We will also make use of some of the tools I use for bounding functions and finding zeros, they are in a package called `ArbExtras.jl` but it is not a registered package so we install it from Github.

```
let
    import Pkg
    Pkg.activate(mktempdir())
    Pkg.add(["Arblib", "Plots", "PlutoUI", "SpecialFunctions"])
    Pkg.add(url = "https://github.com/Joel-Dahne/ArbExtras.jl", rev = "besselj")

    using ArbExtras, Arblib, LinearAlgebra, Plots, PlutoUI, SpecialFunctions

    setprecision(Arb, 64)

    nothing
end
```

Construct the approximate eigenfunction

Finding an approximate eigenfunction can be a hard task in itself. In this tutorial we skip this part and use an already computed approximation of the eigenvalue and the eigenfunction, we will only care about the rigorus verification of this approximation. The approximation was computed using the Method of Particular Solutions.

```
vertices =
  [[0, 0], [1.0000000000000000000, 0], [1.0000000000000000000, 1.0000000000000000000], [-1.00
```

```
vertices = Vector{Arb}[[0.0, 0.0], [1.0, 0.0], [1.0, 1.0], [-1.0, 1.0], [-1.0,
-1.0], [0.0, -1.0]]
```

```
λ = [9.639664913984496764 +/- 4.68e-19]
```

```
λ = Arb(9.639664913984497)
```

```
coefficients =
  [[-0.20655200000000000134 +/- 3.12e-20], [4.971820000000000113e-6 +/- 3.09e-27], [-0.052
```

```
coefficients = Arb[-0.206552, 4.97182e-6, -0.0529564, 0.164401, 0.000266473,
0.422812, 0.159835, 0.0546326]
```

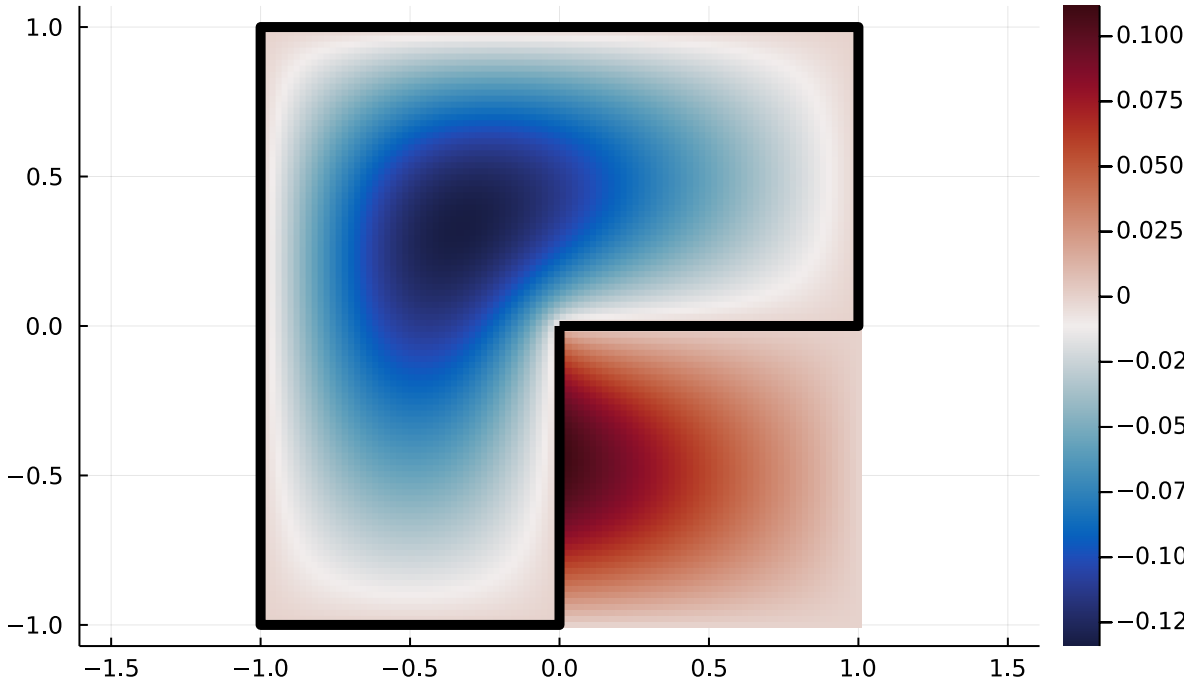
```
u (generic function with 1 method)
```

```
function u(xy, λ)
    # Convert from cartesian to polar coordinates
    r = norm(xy)
    if xy[1] > 0 || xy[2] > 0
        θ = atan(xy[2], xy[1])
    else
        θ = π + atan(-xy[2], -xy[1])
    end

    # Add together the functions in the approximation
    res = zero(r)
    for i in eachindex(coefficients)
        ν = oftype(res, 2(1 + (i - 1) * 2) // 3)
        res += coefficients[i] * besselj(ν, r * sqrt(λ)) * sin(ν * θ)
    end

    return res
end
```

```
u (generic function with 2 methods)
```

```
function u(xy::AbstractVector{ArbSeries}, λ)
    # Convert from cartesian to polar coordinates
    r = sqrt(xy[1]^2 + xy[2]^2)
    if xy[1][0] > 0 || xy[2][0] > 0
        θ = atan(xy[2], xy[1])
    else
        θ = π + atan(-xy[2], -xy[1])
    end

    # Add together the functions in the approximation
    res = zero(r)
    for i in eachindex(coefficients)
        ν = Arb(21 + (i - 1) * 2 // 3)
        res += coefficients[i] * besselj(ν, r * sqrt(λ)) * sin(ν * θ)
    end

    return res
end
```

```
function Base.atan(y::ArbSeries, x::ArbSeries)
    xdy = x * Arblib.derivative(y)
    ydx = y * Arblib.derivative(x)
    x2 = x^2
    y2 = y^2

    z = Arblib.integral((xdy - ydx)/(x^2 + y^2))
    z[0] = atan(y[0], x[0])

    return z
end
```



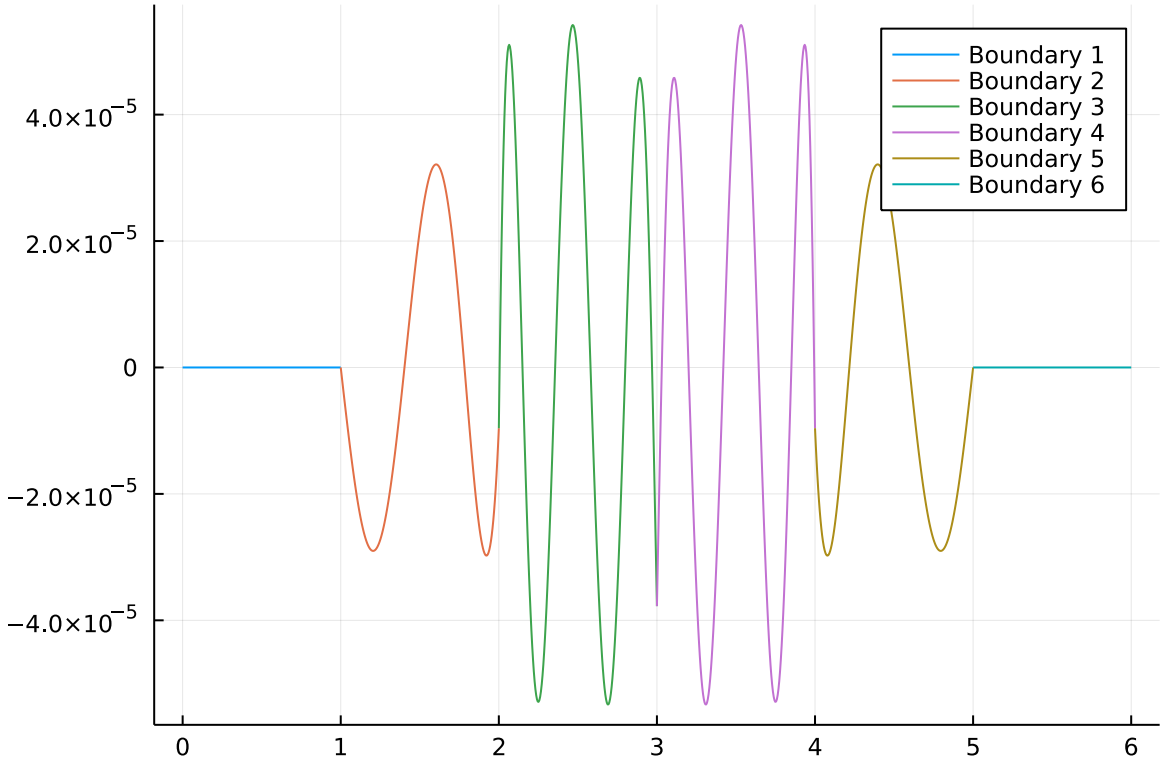
Estimate bounds

To get an enclosure of the eigenvalue we need to upper bound the defect of our approximation and lower bound its norm. Lets start with a non-rigorous approach to see how it looks like.

We define a parameterization of the boundary of our domain and use this to plot the eigenfunction on the boundary.

```
boundary_parameterization (generic function with 1 method)
```

```
function boundary_parameterization(i::Integer)
    v = vertices[i]
    w = vertices[mod1(i + 1, length(vertices))]
    p(t) = v + t * (w - v)
    p(t::ArbSeries) = begin
        res = v + Ref(t) .* (w - v)
        res[1].arb_poly.length = t.arb_poly.length
        res[2].arb_poly.length = t.arb_poly.length
        return res
    end
    return p
end
```




```
let pl = plot()
    ts = range(0, 1, length = 200)
    for i in 1:6
        p = boundary_parameterization(i)
        plot!(pl, (i - 1) .+ ts, u.(p.(ts), λ), label = "Boundary $i")
    end
    pl
end
```

Next we do a **very** simple approximation of the norm by just sampling random points in the interior of the domain.

```
[0.10816269035257844 +/- 7.75e-18]
```

```
let n = 1000
    norms = zeros(Arb, n)
    sum = zero(Arb)
    area = 3
    for i in 1:n
        # Generate random point
        xy = (2rand(Arb) - 1, 2rand(Arb) - 1)
        while xy[1] > 0 && xy[2] < 0
            xy = (2rand(Arb) - 1, 2rand(Arb) - 1)
        end

        # Evaluate u at this point
        sum += abs2(u(xy, λ))
        norms[i] = sqrt(area * sum / i)
    end

    plot(norms, ylims = [0, NaN], label = "")

    norms[end]
end
```

Upper bounding the defect

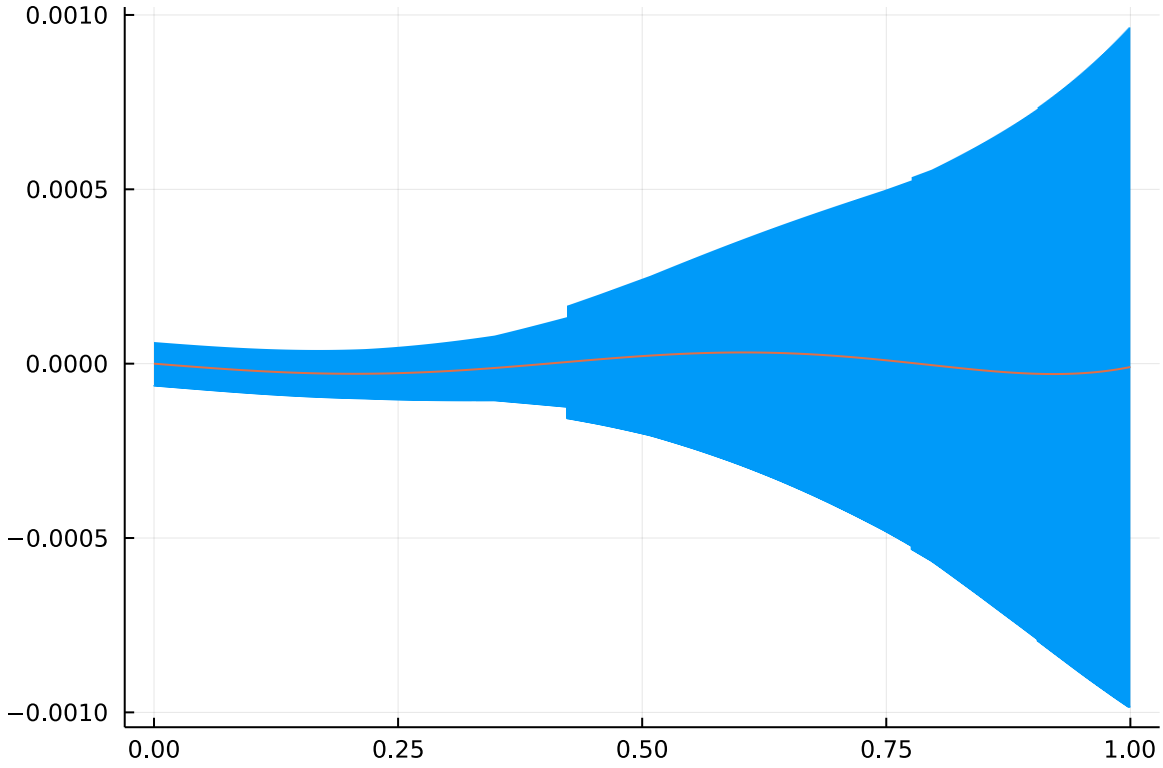
Now that we have seen that the approximations look good we try to rigorusly bound the values instead. Starting with the defect.

We begin with taking only one boundary at a time, to easier see how things behave.

```
boundary = 2
```

```
boundary = 2
```

1000




```
let n = num_intervals
    p = boundary_parameterization(boundary)
    pl = plot()

    ts = mince(Arb((0, 1)), n)
    norm.(p.(ts))
    values = u.(p.(ts), Arb(λ))

    xs = [lbound.(ts); ubound(ts[end])]
    lower = [lbound.(values); lbound(values[end])]
    upper = [ubound.(values); ubound(values[end])]
    plot!(pl, xs, lower, fillrange = upper, legend = :none, linetype = :steppre)

    plot!(pl, range(0, 1, length = 1000), t -> u(p(t), λ))
end
```

```
[[+/- 9.85e-4], [+/- 3.63e-3], [+/- 3.63e-3], [+/- 9.85e-4]]
```

```
let n = 1000
    ts = mince(Arb((0, 1)), n)

    res = zeros(Arb, 4)

    for i = 2:5
        p = boundary_parameterization(i)
        res[i - 1] = maximum(abs.(u.(p.(ts), Arb(λ))))
    end

    res
end
```

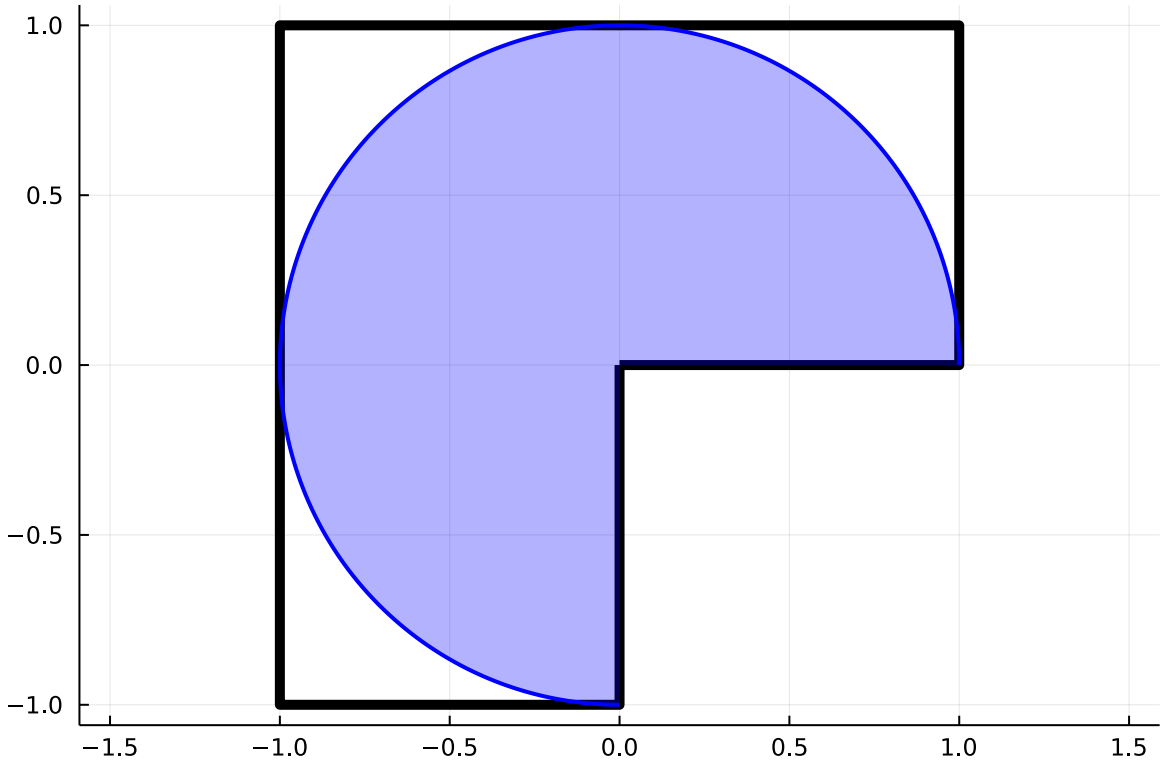
```
δ = [2.309914238903e-5 +/- 3.58e-18]
```

```
δ = let λ = Arb(λ)
    res = zeros(Arb, 4)

    for i = 2:5
        p = boundary_parameterization(i)
        res[i - 1] = ArbExtras.maximum_enclosure(
            t -> u(p(t), λ),
            Arf(0),
            Arf(1),
            abs_value = true,
            degree = 4,
            verbose = true,
        )
    end


    maximum(res)
end
```

Lower bounding the norm

Now that we have an upper bound for the defect we need to compute a lower bound of the norm. In the general case this is quite tedious to implement but we will use a simplification which happens to work in our (simple) case.

We will compute the norm on a subset of the domain, this still gives a lower bound, where the eigenfunction splits nicely. The subset of the domain we will consider is



On this part of the domain `besselj(ν, r * sqrt(λ))` and `sin(ν * θ)` are orthogonal and the integral for the norm therefore splits into integrals of the form

$$\int_0^{3\pi/2}\int_0^1 rB_\nu(r\sqrt{\nu})^2\sin(\nu\theta)^2\,drd\theta = \int_0^{3\pi/2}\sin(\nu\theta)^2\,d\theta\int_0^1 rB_\nu(r\sqrt{\nu})\sin(\nu\theta)\,dr.$$

For our function we up with, after some simplifications,

$$\|u\|^2 \geq \frac{3\pi}{4}\sum_{k=1}^{8} c_k^2\int_0^1 rJ_{2k/3}(r\sqrt{\lambda})^2 dr$$

`norm_lower = [0.12065000238833990 +/- 4.41e-18]`

```
norm_lower = let
    θ_integral = 3 // 4 * Arb(π)

    r_integral = zero(Arb)
    a = Arb(1e-1)
    b = Arb(1)
    for i in eachindex(coefficients)
        integral = Arblib.integrate(
            r -> r * besselj(Acb(2i // 3), sqrt(λ) * r)^2,
            a,
            b,
        )
        r_integral += coefficients[i]^2 * real(integral)
    end

    sqrt(θ_integral * r_integral)
end
```

Enclosure of the eigenvalue

The enclosure of the eigenvalue is now given from

$$\frac{\tilde{\lambda}}{1+\epsilon} \leq \lambda \leq \frac{\tilde{\lambda}}{1-\epsilon}$$

where $\tilde{\lambda}$ is our approximate eigenvalue and

$$\epsilon = \sqrt{A}\frac{\delta}{\|u\|^2}$$

with A being the area of the domain (3 in our case).

`ϵ = [0.0003316111681481 +/- 5.59e-17]`

```
ϵ = sqrt(Arb(3)) * δ / norm_lower
```

`([9.636469353125484 +/- 5.99e-16], [9.642862594913888 +/- 7.65e-16])`

```
λ_lower , λ_upper = (λ / (1 + ϵ)), (λ / (1 - ϵ))
```

`λ_enclosure = [9.64 +/- 3.54e-3]`

```
λ_enclosure = Arb((λ_lower, λ_upper))
```

Proving that it is the first eigenvalue

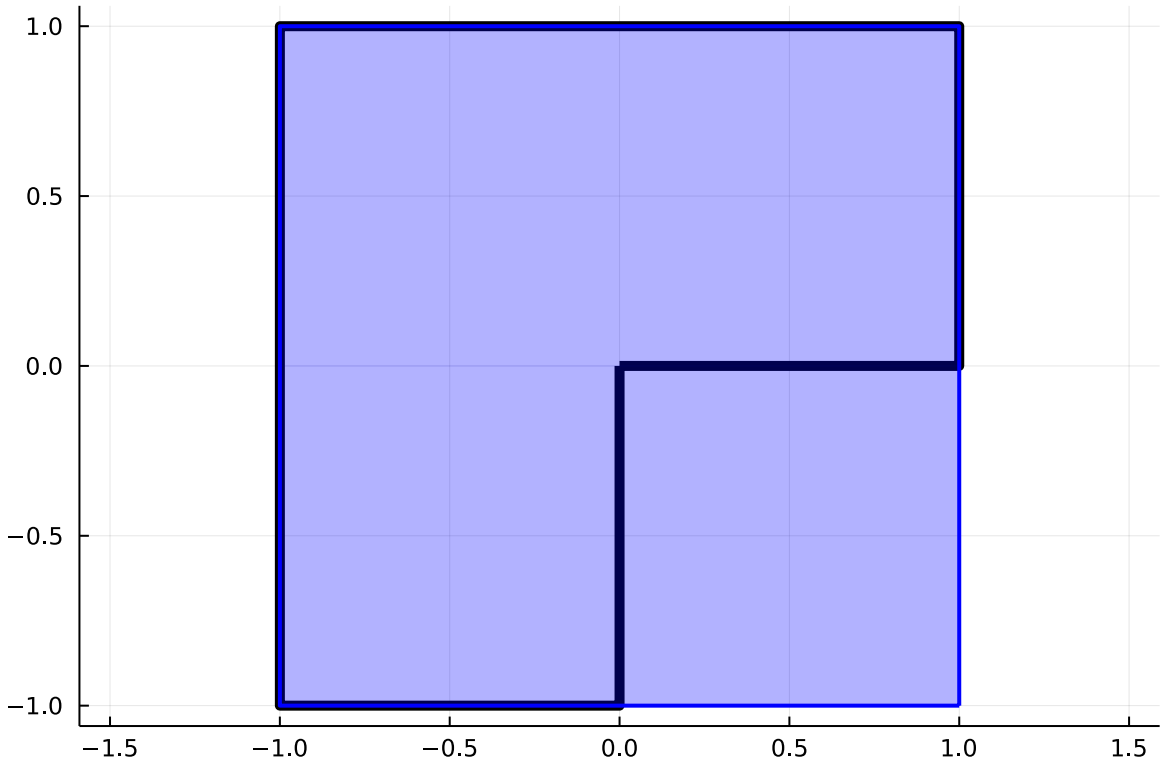
We have now computed an enclosure for the eigenvalue, but at the moment we don't have any information about the index.

Why would this be the first eigenvalue? It is numerically obvious from the construction of the solution, which we skipped here... But it can also be seen by looking at the eigefunction. From the plot of the eigenfunction above we see that it has a constant sign throughout the domain, this would imply that it is the first eigenfunction. This is however **not** something we can rigorously check, in fact our approximation doesn't have a constant sign, it will oscilate around 0 near the boundaries.

How can we then prove that the eigenvalue we have is the first one? We will do that by getting a, rough, lower bound for the second eigenvalue of the domain. If our eigenvalue is lower than the lower bound for the second eigenvalue then clearly our eigenvalue must be the first one.

To get this lower bound we will use the monotonicity property of the eigenvalues and the domain. If we consider a **larger** domain then the eigenvalues are **smaller**.

Consider the square with side lengths 2 centered at the origin, this contains our domain.



A lower bound for the second eigenvalue of this square gives a lower bound for the second eigenvalue of our original domain.

In this case the new domain is a square with side length 2, the eigenvalues are then known explicitly. For a rectangle with side lengths a and b the eigenvalues are given by

$$\lambda_{m,n} = \pi^2\left((m/a)^2 + (n/b)^2\right)$$

In our case $a = b = 2$ and the first eigenvalue is given by

$$\lambda_{1,1} = \pi^2/2$$

The second eigenvalue is double and given by

$$\lambda_{1,2} = \lambda_{2,1} = 5\pi^2/4$$

Comparing this to our eigenvalue we have

```
([4.93480220054467931 +/- 1.51e-18], [9.64 +/- 3.54e-3], [12.33700550136169827 +/- 5.55
```

```
Arb(π)^2 / 2, λ_enclosure, 5Arb(π)^2 / 4
```

```
true
```

```
Arb(π)^2 / 2 < λ_enclosure < 5Arb(π)^2 / 4
```

So the eigenvalue we have is lower than the second eigenvalue of our domain, hence it has to be the first!

Appendix

Some extra methods needed above.

```
Main.workspace2.mince
```