Special Issue in honor of John W. Neuberger
Electronic Journal of Differential Equations, Special Issue 02 (2023), pp. 67–80.
ISSN: 1072-6691. URL: https://ejde.math.txstate.edu or https://ejde.math.unt.edu

NUMERIC ESTIMATES OF THE PRINCIPAL EIGENVALUE OF THE p-LAPLACIAN USING INTERVAL ARITHMETIC

JIŘÍ BENEDIKT, JAN PŮLPÁN

Abstract. We present a numerical algorithm for computing rigorous upper and lower estimates of the principal eigenvalue of the p-Laplacian. To control all possible errors including the rounding errors of the computer arithmetic, we use the interval arithmetic. We implement our algorithm in the Julia programming language using IntervalArithmetic.jl package [12].

1. Introduction

We consider the eigenvalue problem

$$\begin{gathered} -(|u'|^{p-2}u')' = \lambda |u|^{p-2}u \quad \text{in } (0,1), \\ u(0) = u(1) = 0 \end{gathered} \tag{1.1}$$

where $p > 1$ and $\lambda \in \mathbb{R}$ is the spectral parameter. Though our algorithm is general enough to treat more general problems, for instance problems with non-constant coefficients or partial differential equations with the p-Laplacian on various domains, we present our algorithm applied to the problem (1.1) for the sake of clarity. Moreover, the dependence of the principal eigenvalue $\lambda_{1,p}$ of (1.1) on p is expressed explicitly by

$$\lambda_{1,p} = (p-1)\Big(\frac{2\pi}{p\sin\frac{\pi}{p}}\Big)^p \tag{1.2}$$

which allows us to track the errors of our numerical estimates easily. Historical remarks on the practical use of p-Laplace equations can be found in [4] and the references therein.

Similarly to [2, 3], we estimate the principal eigenvalue $\lambda_{1,p}$ of (1.1) from above by using the variational characterization

$$\lambda_{1,p} = \min \frac{\int_0^1 |v'|^p \mathrm{d}t}{\int_0^1 |v|^p \mathrm{d}t} \tag{1.3}$$

where the minimum is taken over all $v \in W_0^{1,p}(0,1)$, $v \not\equiv 0$, and from below by using the following consequence of [1, Theorem 2.1].

2020 *Mathematics Subject Classification.* 34B09, 34B15, 65L15, 65G30.
Key words and phrases. Quasilinear equation; p-Laplacian; principal eigenvalue; interval arithmetic.

Published March 27, 2023.

Theorem 1.1. *Assume $v \in C^1(0,1)$, $v > 0$ in $(0,1)$ and $|v'|^{p-2}v' \in C^1(0,1)$. Then*

$$\lambda_{1,p} \geq \inf_{t\in(0,1)} \frac{-\left(|v'|^{p-2}v'\right)'}{v^{p-1}}. \tag{1.4}$$

Obviously, if we choose an arbitrary (but admissible) test function v and compute the Rayleigh quotient in (1.3), we obtain an upper estimate and from (1.4) we obtain a lower estimate. In both cases the optimal choice of the test function v is the principal eigenfunction $\varphi_{1,p}$, corresponding to the principal eigenvalue. The more similar the test function v to $\varphi_{1,p}$ is, the better estimate we obtain. But from the practical point of view, we have a very limited choice of test functions if we want to compute useful explicit estimates even in the case when the domain is a ball (cf. [2, 3]), and the computation becomes practically impossible on more complex domains.

Hence, our approach is to construct the test function from a numerical approximation of the principal eigenfunction. In one dimension we use the shooting method, in higher dimensions there are numerous effective algorithms for various domains – see, e.g., [5, 6, 7, 8, 9, 10, 11, 15]. But, we have to be careful when inserting the numerical solution into (1.3) and (1.4). The numerical solution is typically obtained as an interpolation function through some control points. So first we need to make sure that the interpolation function is an admissible function. Second, the integrals in (1.3) cannot be computed symbolically due to the general power p. Consequently, we integrate numerically. Third, all the operations in computer arithmetic (double precision, IEEE 754) necessarily produce rounding errors. We use the interval arithmetic to control both the error of the numerical integration and the rounding errors to get guaranteed bounds for $\lambda_{1,p}$.

This article is organized as follows. We start with the easier part, which is the upper estimate, in Section 2. Then we deal with the lower estimate in Section 3. Finally, we present numerical results for various values of p in Section 4.

2. Upper estimate

Firstly we compute a numerical approximation of the principal eigenvalue $\lambda_{1,p}$ and the corresponding eigenfunction $\varphi_{1,p}$ by the shooting method. Because of the uniqueness of the solution of the associated initial value problem, it must be $\varphi_{1,p}'(0) \neq 0$. Since problem (1.1) is homogeneous, we can normalize $\varphi_{1,p}$ assuming $\varphi_{1,p}'(0) = 1$. Clearly, $\lambda \in \mathbb{R}$ is an eigenvalue of (1.1) if and only if the solution (u_1, u_2) of the initial value problem for a system of two first-order equations

$$\begin{gathered} u_1'(t) = |u_2(t)|^{\frac{1}{p-1}} \cdot \operatorname{sign}(u_2(t)) \text{ in } (0,1), \quad u_1(0) = 0, \\ u_2'(t) = -\lambda |u_1(t)|^{p-1} \cdot \operatorname{sign}(u_1(t)) \text{ in } (0,1), \quad u_2(0) = 1, \end{gathered} \tag{2.1}$$

satisfies $u_1(1) = 0$. By the bisection method we find λ such that $u_1(1) = 0$ whereas (λ, u_1) becomes the principal eigenpair. For this purpose we need to start the bisection with an initial interval which contains $\lambda_{1,p}$ and no other eigenvalue. In our simple case we can use (1.2). In general we can use analytical estimates, for example [2] or [3] on a ball in $\mathbb{R}^N$. The Julia code follows.

```
"""
    plaplace_solve(λ_init, p, n; u₂0=1.0, dom=(0.0, 1.0))

Numericaly solves p-Laplace equation using shooting method.
```

```
Arguments:
λ_init ...  initial interval for λ_1
p ...  p of p-Laplacian
n ...  number of solution interpolation points
u_2 0 ...  initial condition for u_2
dom ...  domain

Return:
t, t^I ...  division points where the solution is interpolated,
           double precision value and its interval representation
U_1, U_1^I ...  numerical approximation of u_1 at t, t^I
U_2, U_2^I ...  numerical approximation of u_2 at t, t^I
Λ_1 ...  numerical approximation of the first eigenvalue λ_1
"""
function plaplace_solve(λ_init, p, n; u_20=1.0, dom=(0.0, 1.0))

    function sl(du, u, P, t)
        λ, p = P
        du[1] = abs(u[2])^(1/(p-1)) * sign(u[2])
        du[2] = -λ * abs(u[1])^(p-1) * sign(u[1])
    end

    tl, tr = dom

    u0 = [0.0; u_20;] # initial condition
    a, b = λ_init
    Λ_1 = (a + b)/2
    Δt = (tr-tl)/(n-1)
    e = 1e-12 # stop condition

    while (b-a) >= e
        prob = ODEProblem(sl, u0, dom, (Λ_1, p))
        sol = solve(prob, saveat=Δt, abstol=1e-8, reltol=1e-8)
        if sol(tr)[1] == 0
            break
        else
        probA = ODEProblem(sl, u0, dom, (a, p))
        solA = solve(probA, saveat=Δt, abstol=1e-8, reltol=1e-8)
        probS = ODEProblem(sl, u0, dom, (Λ_1, p))
        solS = solve(probS, saveat=Δt, abstol=1e-8, reltol=1e-8)
        if solA(tr)[1] * solS(tr)[1] < 0
            b = Λ_1
        else
            a = Λ_1
        end
            Λ_1 = (a+b)/2
        end
    end

    prob = ODEProblem(sl, u0, dom, (Λ₁, p))
    sol = solve(prob, saveat=Δt, abstol=1e-8, reltol=1e-8)

    t = LinRange(0, 1, n-1)
    tᴵ = [@interval(i) for i in t]
    U₁ = [u[1] for u in sol(t).u]
    U₁ᴵ = [@interval(u[1]) for u in sol(t).u]
    U₂ = [u[2] for u in sol(t).u]
    U₂ᴵ = [@interval(u[2]) for u in sol(t).u]

    return t, tᴵ, U₁, U₁ᴵ, U₂, U₂ᴵ, Λ₁
end
```

The function plaplace_solve returns vectors U$_1$ and U$_2$ of numerical approximations of the values of $u_1 = \varphi_{1,p}$ and u_2 at n (a parameter) equidistant division points t of the interval $[0, 1]$. As the test function v for which we compute the Rayleigh quotient in (1.3) we choose the cubic spline interpolation function through the points U$_1$ with zero second derivatives at the endpoints. These boundary conditions seem to be natural since for $p = 2$ we clearly have $\varphi''_{1,2}(0) = \varphi''_{1,2}(1) = 0$. Since we set the first and the last element of U$_1$ to exact zero and the cubic spline interpolation function is of class C^2, we have $v \in W_0^{1,p}(0, 1)$ and so our choice of the test function in (1.3) is admissible.

The coefficients of the cubic spline are standardly obtained from the vector of the second derivatives at the division points which is computed as the solution of a system of linear algebraic equations with a tri-diagonal matrix and a right-hand side constructed from the interpolation values and the division points. As it was already mentioned, these computations cannot be carried out in the double precision arithmetic without rounding errors. Hence, instead of t and U$_1$ we use vectors t$^{\mathrm{I}}$ and U$_1{}^{\mathrm{I}}$ of the corresponding intervals which are also returned by the Julia function plaplace_solve. This makes all the computations to be carried out in the interval arithmetic and we obtain the coefficients of the cubic spline represented as intervals instead of double precision numbers. The properties of the interval arithmetic guarantee that these intervals contain the exact values of the coefficients of the test function v. The following function cubic_natural_spline returns the interval coefficients csc_V.

```
"""
    cubic_natural_spline(t, tᴵ, U, Uᴵ, Uₗd2, Uᵣd2; ns=10)

IA interpolation of given points by natural cubic spline.
Returns spline coefficients ‘csc_V‘ as well as interval values ‘V‘
of the spline function.

Arguments:
t, tᴵ ...  division points
U, Uᴵ ...  values to be interpolated
Uₗd2, Uᵣd2 ...  left and right boundary values of second derivative
ns ...  number of division points for each single piece of spline
```

```
"""
function cubic_natural_spline(t, t^I, U, U^I, U_ld2, U_rd2; ns=10)
    # A matrix
    n = length(U^I)
    dv = [4..4 for i in 1:n-2]
    ev = [1..1 for i in 1:n-3]
    A = Array(SymTridiagonal(dv, ev))
    A^{-1} = inv(A)

    # right-hand side
    h = 1.0/(n-1)
    rhs = []
    for i in 3:length(U^I)
        append!(rhs, 6/h^2 * (U^I[i] - 2 * U^I[i-1] + U^I[i-2]))
    end

    rhs[1] = rhs[1]-U_ld2
    rhs[end] = rhs[end]-U_rd2

    # second derivatives vector
    Ud2 = []
    append!(Ud2, @interval(U_ld2))
    append!(Ud2, A^{-1}*rhs)
    append!(Ud2, @interval(U_rd2))

    # spline coefficients
    csc_V = Vector[]
    for i in 1:length(U^I)-1
        a = b = c = d = 0
        a = (Ud2[i+1]-Ud2[i])/(6*h)
        b = Ud2[i]/2
        c = (U^I[i+1]-U^I[i])/h - h*(2*Ud2[i]+Ud2[i+1])/6
        d = U^I[i]
        append!(csc_V, [Interval{Float64}[a,b,c,d]])
    end

    V = Interval{Float64}[]
    for i in 1:length(U^I)-1
        x_dom = t[i]..t[i+1]
        x_int = mince(x_dom, ns)
        f(x) = csc_V[i][4] + (x-t[i])*(csc_V[i][3] +
            (x-t[i])*(csc_V[i][2] + csc_V[i][1]*(x-t[i])))
        append!(V, f.(x_int))
    end

    return csc_V, V
end
```

Now we compute the integrals in (1.3). Though the cubic spline is a piecewise polynomial, we integrate numerically because of the general power p. Since the division at `n` points may be too coarse, we divide each single subinterval by `ns` subdivision points. On each of the subsubintervals we compute an interval which must contain the range of all possible values of any cubic function with coefficient from the respective interval in `csc_V`. The vector V of these intervals is also returned by the above function `cubic_natural_spline`. To compute the numerator in (1.3) we need a vector of intervals which represent the values of v'. The interval coefficients of the piecewise quadratic function v' are easily obtained from the coefficients of v. A vector of intervals representing the values of v' on each subsubinterval is returned by the following function `der_cubic_spline`.

```
"""
    der_cubic_spline(csc, t, tᴵ, ns)

Computes the first derivative ‘V‘ of a given interval cubic spline.

Arguments:
csc ...  cubic spline coefficients interval representation
t, tᴵ ...  division points
ns ...  number of division points for each single piece of spline
"""
function der_cubic_spline(csc, t, tᴵ, ns)

    V_tmp = Interval[]
    csc_Vder = [ [@interval(3) * c[1], @interval(2) * c[2],
        c[3]] for c in csc ]
    for i in 1:length(t)-1
        x_dom = t[i]..t[i+1]
        x_int = mince(x_dom, ns)
        f(x) = csc_Vder[i][3] + (x-t[i])*(csc_Vder[i][2] +
            (x-t[i])*csc_Vder[i][1])
        append!(V_tmp, f.(x_int))
    end

    V = Interval[]
    for i in 1:length(U_tmp)-1
        append!(V, V_tmp[i] ∪ V_tmp[i+1])
    end
    append!(V, V_tmp[end])

    return V
end
```

Finally, we compute the p-th power of the intervals representing v and v', multiply by the length of the subsubintervals, sum up over all subsubintervals and divide the numerator by the denominator. Everything in the interval arithmetic, of course. We get an interval which is guaranteed to contain the exact value of the Rayleigh quotient for the test function v. Consequently, the right end-point of the interval is the desired rigorous upper estimate of $\lambda_{1,p}$.

```
"""
    function upper_estimate(V₁, V₁_der, p)

Returns guaranteed upper estimate of the first eigenvalue `λ₁ᵘᵖ`.

Arguments:
p ...  p of p-Laplacian
V₁ ...  interval values of v₁
V₁_der ...  interval values of v₁'
"""
function upper_estimate(V₁, V₁_der, p)

    f(x) = abs(x)^(p)
    ni = mince(0..1, length(V₁))

    numerator = 0..0
    for i in 1:length(V₁_der)
        numerator = numerator + f(V₁_der[i]) * diam(ni[i])
    end

    denominator = 0..0
    for i in 1:length(V₁)
        denominator = denominator + f(V₁[i]) * diam(ni[i])
    end

    λ₁ᵘᵖ = sup(numerator/denominator)

    return λ₁ᵘᵖ
end
```

The parameters n and ns are to be experimented with to get a sharp enough estimate.

3. Lower estimate

There are two reasons why such a straightforward approach which we used in Section 2 does not work for the lower estimate. First, both the numerator and the denominator in (1.4) go to 0 when $t \to 0+$ or $t \to 1$-. If we represent them again by a vector of intervals, the first and the last interval would have to contain 0 and probably also negative numbers. Consequently, the ratio in (1.4) would become $(-\infty, +\infty)$ and the lower estimate would be $-\infty$ which is true but useless. Second, if $v \in C^2(0,1)$, then for $p < 2$ the condition $|v'|^{p-2}v' \in C^1(0,1)$ is not satisfied at a point t_m where $v'(t_m) = 0$ and $v''(t_m) \neq 0$. On the other hand, for $p > 2$ the condition is satisfied but $\big(|v'|^{p-2}v'\big)' = 0$ at $t = t_m$. Hence, the lower estimate would not be even a positive number.

Our strategy to avoid the first problem is that we choose the test function v as a numerical approximation of the restriction to $(0,1)$ of the principal eigenfunction $\tilde{\varphi}$ on $(-r, 1+r)$ for a small $r > 0$ instead of on $(0,1)$. If we insert $\tilde{\varphi}$ into (1.4), we obtain the principal eigenvalue $\tilde{\lambda}$ on $(-r, 1+r)$ which is smaller but close to $\lambda_{1,p}$ if $r > 0$ is small. Since we normalize by $\tilde{\varphi}'(-r) = 1$, the value $\tilde{\varphi}(0)$ is asymptotically equal to r for $r \to 0+$. Consequently, both the numerator and the denominator in (1.4) now go to a positive constant when $t \to 0+$ or $t \to 1-$. A similar approach is applicable even in higher dimensions thanks to the Hopf Maximum Principle (see [13, Prop. 3.2.1 and 3.2.2, p. 801], [14, Theorem 5, p. 200]). To compute the numerical approximation of $\tilde{\varphi}$ we use the function `plaplace_solve` again. We may even decrease the test function by a small positive constant (the test function must not become negative) which does not change the numerator, decreases the denominator, and improves the lower estimate.

To deal with the second problem, we still use the cubic spline interpolation function but instead of $\tilde{\varphi}$ we approximate $|\tilde{\varphi}'|^{p-1} \cdot \operatorname{sign}(\tilde{\varphi}')$, i.e., the function u_2 from the associated initial value problem. Let us denote the cubic spline interpolation function through the corresponding numerical values of u_2 by v_2. The natural boundary conditions for the spline are $v_2'(0) = v_2'(1) = -\tilde{\lambda}\tilde{\varphi}^{p-1}(0)$. Instead of exact $\tilde{\lambda}$ and $\tilde{\varphi}$ we use the corresponding numerical approximations. Therefore, we need the following code to compute the intervals for the coefficient of the spline using the conditions on the end slopes.

```
"""
    cubic_end_slope_spline(t, tᴵ, U, Uᴵ, Uₗd1, Uᵣd1; ns=10)

IA interpolation of given points by end slope cubic spline.
Returns spline coefficients ‘csc_V‘ as well as interval values ‘V‘
of the spline function.

Arguments:
t, tᴵ ...  division points
U, Uᴵ ...  values to be interpolated
Uₗd1, Uᵣd1 ...  left and right boundary values of first derivative
ns ...  number of division points for each single piece of spline
"""

function cubic_end_slope_spline(t, tᴵ, U, Uᴵ, Uₗd1, Uᵣd1; ns=10)
    # A matrix
    n = length(Uᴵ)
    dv = [4..4 for i in 1:n-2]
    ev = [1..1 for i in 1:n-3]
    A = Array(SymTridiagonal(dv, ev))
    A[1,1] = 3.5..3.5
    A[end,end] = 3.5..3.5
    A⁻¹ = inv(A)

    # right-hand side
    h = 1.0/(n-1)
    rhs = []
    for i in 3:length(Uᴵ)
        append!(rhs, 6/h^2 * (Uᴵ[i] - 2 * Uᴵ[i-1] + Uᴵ[i-2]))
    end

    rhs[1] = rhs[1] - 3/h * ( (Uᴵ[2]-Uᴵ[1])/h - Uₗd1)
    rhs[end] = rhs[end] - 3/h * (Urd1 - (UI[end]-UI[end-1])/h)

    # second derivatives vector
    sol = A^-1*rhs
    Ud2 = []

    σ0 = 3/h * ( (UI[2]-UI[1])/h - U1d1) - sol[1]/2
    σ1 = 3/h * (Urd1 - (UI[end]-UI[end-1])/h) - sol[end]/2
    append!(Ud2, @interval(σ0))
    append!(Ud2, sol)
    append!(Ud2, @interval(σ1))

    # spline coefficients
    csc_V = Vector[]
    for i in 1:length(UI)-1
        a=b=c=d=0
        a = (Ud2[i+1]-Ud2[i])/(6*h)
        b = Ud2[i]/2
        c = (UI[i+1] - UI[i])/h - h*(2*Ud2[i]+Ud2[i+1])/6
        d = UI[i]
        append!(csc_V, [Interval{Float64}[a,b,c,d]])
    end

    V = Interval{Float64}[]
    for i in 1:length(UI)-1
        x_dom = t[i]..t[i+1]
        x_int = mince(x_dom,ns)
        f(x) = csc_V[i][4] + (x-t[i])*(csc_V[i][3] +
            (x-t[i])*(csc_V[i][2] + csc_V[i][1]*(x-t[i])))
        append!(V, f.(x_int))
    end

    return csc_V, V
end
```

Since the numerator in (1.4) is $-v_2'$, we compute the interval representation of the numerator by the function `der_cubic_spline`. We point out that in the denominator we cannot use a cubic spline interpolation function v_1 through the numerical values of $\tilde{\varphi}$ (i.e., u_1) since we would fail to guarantee $|v_1'|^{p-1}\cdot\operatorname{sign}(v_1') = v_2$. Instead, we reconstruct v_1 from v_2 as a primitive function to $|v_2|^{\frac{1}{p-1}} \cdot \operatorname{sign}(v_2)$. The additive constant is chosen as the smallest one such that v_1 does not become negative. The following Julia function `get_v1` returns the interval representation of v_1.

```
"""
    get_v1(p, V2, t)

Rebuilds interval expression of ‘V1‘ by integrating ‘V2‘.

Arguments:
```

```
p ...  $p$ of $p$-Laplacian
V$_2$ ...  interval values of $v_2$
t ...  division points
"""
function get_v1(p, V$_2$, t)

    f(x) = abs(x)^(1/(p-1))*sign(x)
    ni = mince(0..1, length(V$_2$))

    V$_1$_tmp = Interval[0..0]
    for i in 1:length(V$_2$)
        append!(V$_1$_tmp, V$_1$_tmp[end] + f(V$_2$[i]) * diam(ni[i]))
    end

    V$_1$ = Interval[]
    for i in 1:length(V$_1$_tmp)-1
        append!(V$_1$, V$_1$_tmp[i] ∪ V$_1$_tmp[i+1])
    end

    V$_1$ = V$_1$ .- inf(minimum(V$_1$))

    return V$_1$
end
```

Finally, we find the smallest left end-point of all intervals representing $-v_2'/v_1^{p-1}$ to get the guaranteed lower estimate.

```
"""
    lower_estimate(V$_2$_der, V$_1$, p)

Returns guaranteed lower estimate of first eigenvalue `$\lambda_1$$^{\text{low}}$` and
interval values of $-u_2'/u_1^{p-1}$ in `F$^{\text{low}}$`.

Arguments:
V$_2$_der ...  interval values of $v_2'$
V$_1$ ...  interval values of $v_1$

"""
function lower_estimate(V$_2$_der, V$_1$, p)
    f(x,y) = -x / y^(p-1)
    $\lambda_1$_tmp = f.(V$_2$_der, V$_1$)

    F$^{\text{low}}$ = Interval[]
    for i in 1:length($\lambda_1$_tmp)-1
        append!(F$^{\text{low}}$, $\lambda_1$_tmp[i] ∪ $\lambda_1$_tmp[i+1])
    end
    append!(F$^{\text{low}}$, $\lambda_1$_tmp[end])

    $\lambda_1$$^{\text{low}}$ = inf(minimum(F$^{\text{low}}$))
    return λ₁ˡᵒʷ, Fˡᵒʷ
end
```

4. Results

We present results for $p \in \{1.5, 1.6, \dots, 3.0\}$. The parameters which are subject of experimentation are `n`, `ns` and r. Our experiments show that $r = 10/(\mathtt{n} * \mathtt{ns})$ is a good choice. The first result is for 20 subintervals of $[0,1]$ and 10 subsubintervals of each subinterval.

```
    n = 21 # number of division points
    ns = 11 # number of subdivision points of each subinterval

    λ₁ᵉˣᵃᶜᵗ(P) = (P-1)*(2*(π/P)/(sin(π/P)))^P

    λ₁ˡᵒʷs = Float64[]
    λ₁ˡᵒʷs_err = Float64[]

    λ₁ᵘᵖs = Float64[]
    λ₁ᵘᵖs_err = Float64[]

    si = mince(0..1, (n-1)*ns)

    ps = 1.5:0.1:3.0
    for p in ps
        λ₁ = λ₁ᵉˣᵃᶜᵗ(p)
        λ_init = (3.,1.5*λ₁)
        r = 10/(n*ns)
        dom = (-r, r+1)

        ### lower estimate
        t, tᴵ, U₁, U₁ᴵ, U₂, U₂ᴵ, Λ₁ = plaplace_solve(λ_init, p, n,
            dom=dom);
        ud1 = -Λ₁ * U₁[end]^(p-1)
        csc_V₂, V₂ = cubic_end_slope_spline(t, tᴵ, U₂, U₂ᴵ,
            ud1, ud1, ns=ns);
        V₁ = get_v1(p, V₂, t);
        V₂_der = der_cubic_spline(csc_V₂, t, tᴵ, ns);
        λ₁ˡᵒʷ, Fˡᵒʷ = lower_estimate(V₂_der, V₁, p)
        append!(λ₁ˡᵒʷs, λ₁ˡᵒʷ)
        append!(λ₁ˡᵒʷs_err, λ₁ˡᵒʷ-λ₁)

        ### upper estimate
        t, tᴵ, U₁, U₁ᴵ, U₂, U₂ᴵ, Λ₁ = plaplace_solve(λ_init, p, n);
        U₁[end] = 0
        U₁ᴵ[end] = 0..0
        csc_V₁, V₁ = cubic_natural_spline(t, tᴵ, U₁, U₁ᴵ, 0., 0.,
            ns=ns);
        V₁_der = der_cubic_spline(csc_V₁, t, tᴵ, ns);
        λ₁ᵘᵖ = upper_estimate(V₁, V₁_der, p);
        append!(λ₁ᵘᵖs, λ₁ᵘᵖ)
        append!(λ₁ᵘᵖs_err, λ₁ᵘᵖ-λ₁)
    end
```

The computation time was 18 seconds (in total) on one core of Mac Mini, 3.2GHz 6-core Intel i7, 32GB DDR4 RAM. The results are shown in Figures 1 and 2. The results for `n` $= 201$ and `n` $= 101$ are shown in Figures 3 and 4, the computation time was 81 seconds on the same machine. Since the graphs in Figure 3 are too close to each other, we give the numerical values in the following table. The lower estimate is rounded down and the upper estimate is rounded up so the estimates in the table are still guaranteed.

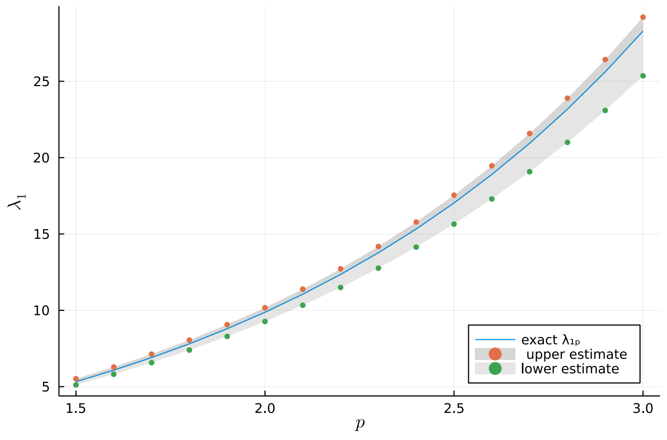

FIGURE 1. Estimates for `n` $= 21$ and `ns` $= 11$.

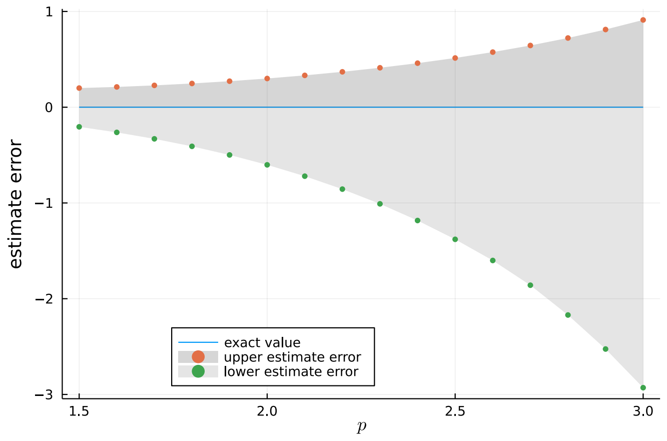

FIGURE 2. Errors for `n` $= 21$ and `ns` $= 11$.

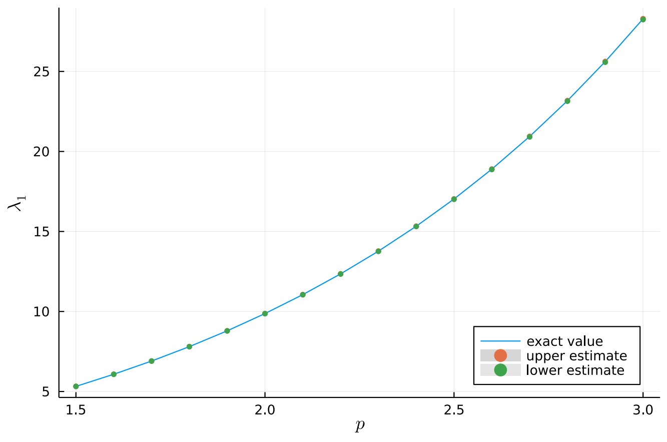

FIGURE 3. Estimates for $\mathtt{n} = 201$ and $\mathtt{ns} = 101$.

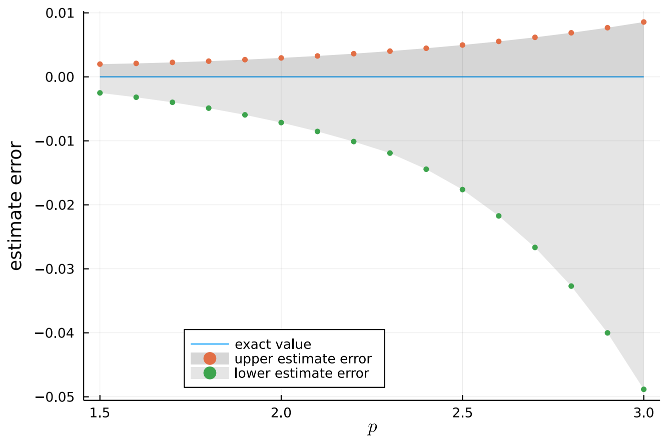

FIGURE 4. Errors for $\mathtt{n} = 201$ and $\mathtt{ns} = 101$.

p	${\lambda_1}^{\text{low}}$	λ_1	${\lambda_1}^{\text{up}}$
1.5	5.316213	5.318718	5.320712
1.6	6.073440	6.076626	6.078731
1.7	6.898054	6.902030	6.904292
1.8	7.798633	7.803521	7.805978
1.9	8.783792	8.789731	8.792420
2.0	9.862458	9.869604	9.872564
2.1	11.044050	11.052580	11.055849
2.2	12.338611	12.348721	12.352344
2.3	13.756919	13.768830	13.772852
2.4	15.310114	15.324548	15.329019
2.5	17.010835	17.028449	17.033426
2.6	18.872410	18.894140	18.899683
2.7	20.909702	20.936359	20.942537
2.8	23.138381	23.171079	23.177967
2.9	25.575610	25.615619	25.623302
3.0	28.239929	28.288762	28.297335

Acknowledgments. J. Benedikt was supported by the Grant Agency of the Czech Republic, Grant No. 22-18261S.

References

[1] Allegretto, W.; Huang, Y. X.; *A Picone's identity for the p-Laplacian and applications.* Nonlinear Anal. 32 (1998), 819–830.

[2] Benedikt, J.; Drábek, P.; *Estimates of the principal eigenvalue of the p-Laplacian.* J. Math. Anal. Appl. 393 (2012), 311–315.

[3] Benedikt, J.; Drábek, P.; *Asymptotics for the principal eigenvalue of the p-Laplacian on the ball as p approaches 1.* Nonlinear Anal. 93 (2013), 23–29.

[4] Benedikt, J.; Girg, P.; Kotrla, L.; Takáč, P.; *Origin of the p-Laplacian and A. Missbach.* Electron. J. Differential Equations (2018), paper no. 16, 17 pp.

[5] Biezuner, R. J.; Brown, J.; Ercole, G.; Martins, E. M.; *Computing the first eigenpair of the p-Laplacian via inverse iteration of sublinear supersolutions*, J. Sci. Comput. 52 (2012) 180–201.

[6] Biezuner, R. J.; Ercole, G.; Martins, E. M.; *Computing the first eigenvalue of the p-Laplacian via the inverse power method.* J. Funct. Anal. 257 (2009), 243–270.

[7] Bognár, G.; Szabó, T.; *Solving nonlinear eigenvalue problems by using p-version of FEM*, Computers and Mathematics with Applications 43 (2003), 57–68.

[8] Bueno, H.; Ercole, G.; Zumpano, A.; *Positive solutions for the p-Laplacian and bounds for its first eigenvalue*, Adv. Nonlinear Stud. 9 (2009) 313–338.

[9] Ercole, G.; Espírito Santo, J. C. D.; Martins, E. M.; *Computing the first eigenpair of the p-Laplacian in annuli.* J. Math. Anal. Appl. 422 (2015), 1277–1307.

[10] Horák, J.; *Numerical investigation of the smallest eigenvalues of the p-Laplace operator on planar domains.* Electron. J. Differential Equations 2011, No. 132, 30 pp.

[11] Lefton, L.; Wei, D.; *Numerical approximation of the first eigenpair of the p-Laplacian using finite elements and the penalty method*, Numer. Funct. Anal. Optim. 18 (1997), 389–399.

[12] Sanders, D. P.; et al.; *JuliaIntervals/IntervalArithmetic.jl: v0.16.7*, `https://zenodo.org/record/3727070#.YV8ouy_RZvI`.

[13] Tolksdorf, P.; *On the Dirichlet problem for quasilinear equations in domains with conical boundary points.* Comm. Partial Differential Equations 7 (1983), 773–817.

[14] Vázquez, J. L.; *A strong maximum principle for some quasilinear elliptic equations.* Appl. Math. Optim. 12(3) (1984), 191–202.

[15] Yao, X.; Zhou, J.; *Numerical methods for computing nonlinear eigenpairs. I. Iso-homogeneous cases.* SIAM J. Sci. Comput. 29 (2007), 1355–1374.

Jiří Benedikt
Department of Mathematics and NTIS, Faculty of Applied Sciences, University of West Bohemia, Univerzitní 8, CZ–301 00 Plzeň, Czech Republic
Email address: `benedikt@kma.zcu.cz`

Jan Půlpán
Department of Mathematics and NTIS, Faculty of Applied Sciences, University of West Bohemia, Univerzitní 8, CZ–301 00 Plzeň, Czech Republic
Email address: `honza@pulpan.net`