PyLS: a Validated Python Lighting Stokes Solver.

Python in Scientific Computing.

Candidate Number: 1070966

Abstract

Series solvers for Laplace and Stokes flow problems are an alternative to discretisation methods that can quickly produce accurate results for simple domains. Recently Brubeck and Trefethen introduced "Lighting Stokes" [2], a rational series method to solve Stokes flow on polygonal domains that runs in seconds of Laptop time. PyLS is an open source implementation of the Lighting Stokes algorithm that facilitates easy creation of Polygonal domains, boundary condition setting and automatic placement of poles. The solver runs in similar time and is validated against examples computed from the MATLAB code. Analytical comparison is made to Pousielle, Couette flow and Lid Driven Cavity Flow.

1 Introduction

Many physical and biological problems can be modelled by 2d stokes flow. Standard discretisation methods often struggle to resolve flow around sharp corners and features that cross multiple scales like Moffat Eddies. For

2 The Lightning Stokes Algorithm

The Lighting Stokes algorithm is a rational series approximation method. In brief, it turns the classical formation of Stokes Flow in 2d into an equivalent problem for the complex streamfunction ψ, which is known as the biharmonic equation. This has a known solution which can be expressed as

$$\psi = \mathrm{Im}\left(\bar{z}f + g\right) \tag{2.1}$$

for arbitrary analytic functions f and g, which is known as the Goursat representation of the streamfunction. The aim now is to efficiently find analytic functions f and g that satisfy the boundary data for a given problem. Lighting Stokes provides a method to construct a well conditioned rational basis, characterised by it's exponentially clustered "lightning poles" at sharp corners. This basis can then be used to create a tall-skinny linear system $Ac = b$ representing the boundary data. The linear system can be solved to find accurate approximations of f and g, and from this many relevant physical quantities can be computed at arbitrary points in the domain. In the rest of this a section we discuss some of the technical details of the method, but for a complete description see [2].

2.1 Goursat Representation of the Streamfunction

Stokes flow is a reduction of the Navier Stokes equations in for low Reynolds number flows which take the form

$$\mu\nabla^2\mathbf{u} = \boldsymbol{\nabla} p, \tag{2.2}$$

$$\boldsymbol{\nabla} \cdot u = 0 \tag{2.3}$$

where μ is the dynamic viscosity, $\mathbf{u}$ is the fluid velocity, and p is the pressure. Recall the stream function is defined as a function ψ with

$$\frac{\partial \psi}{\partial y} = u \qquad\qquad \frac{\partial \psi}{\partial x} = -v \tag{2.4}$$

where u and v are the horizontal components of $\mathbf{u}$. We can guarantee the stream function exists since the flow is incompressible as given by equation (2.3). Lightning Stokes uses the fact that (2.2) can be rewritten as an equivalent problem for the streamfunction, which we briefly derive below.
Recall that the vorticity $\boldsymbol{\omega}$ and vorticity magniutde ω in 2d are defined as

$$\boldsymbol{\omega} := \boldsymbol{\nabla} \times \mathbf{u} = \omega\hat{\mathbf{k}}, \tag{2.5}$$

$$\omega := \left(\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}\right). \tag{2.6}$$

Applying the curl to both sides in (2.2) and using (2.5) we obtain

$$\mu\nabla^2\boldsymbol{\omega} = \boldsymbol{\nabla} \times \boldsymbol{\nabla} p = 0,$$

as $\boldsymbol{\nabla} \times \boldsymbol{\nabla}\mathbf{f} = 0$ for all $\mathbf{f}$. Hence using (2.6) and (2.4) we have

$$\nabla^2\omega = -\nabla^2\nabla^2\psi = 0 \tag{2.7}$$

that is to say ψ satisfies the biharmonic equation

$$\nabla^4\psi = 0. \tag{2.8}$$

Let $z = x + iy$ where $i = \sqrt{-1}$. Using an abuse of notation consider $\psi : \mathbb{C} \to \mathbb{R}$ with $\psi(z, \bar{z}) = \psi(x, y)$. Using Wirtinger derivatives we can rewrite equation (2.8) as

$$16\frac{\partial^4\psi}{\partial z^2 \partial \bar{z}^2} = 0 \tag{2.9}$$

As shown in [2] this has solution

$$\psi(z, \bar{z}) = \mathrm{Im}\left(\bar{z}f(z) + g(z)\right) \tag{2.10}$$

where f and g are arbitrary analytic complex functions, which is known as the *Goursat representation* of the streamfunction. The lightning Stokes algorithm works by constructing a rational basis to efficiently approximate f and g, known as the *Goursat functions*. Note that we can calculate physical quantities directly from the Goursat functions [2], with the fluid velocity given by

$$u - iv = g'(z) + \bar{z}f'(z) - z\overline{f(z)} \tag{2.11}$$

and the pressure and vorticity magnitude by

$$p - i\omega = 4f'(z). \tag{2.12}$$

2.2 The Basis Functions

Consider a simply connected Polygonal domain with K corners w_k with $k \in [1, K]$. We approximate the Goursat functions f and g by a rational series with basis functions ϕ_j, giving

$$f = \sum_{j=1}^{N} f_j \phi_j(z) \qquad\qquad g = \sum_{j=1}^{N} g_j \phi_j(z) \tag{2.13}$$

where $f_j, g_j \in \mathbb{C}$ for $j \in [1, N]$. The basis functions consist of a Polynomial part of degree N_0 and a group of N_k poles at fixed locations for each of the K corners of the polygonal domain, β_{nk}. Hence the set of basis functions is given by

$$\{\phi_j\}_{j=1}^{N} = \bigcup_{k=1}^{K} \left\{ \frac{1}{z - \beta_{kn}} \right\}_{n=1}^{N_k} \bigcup \left\{ z^n \right\}_{n=1}^{N_0}.$$

The "lightning poles" from [3] are exponentially clustered along the exterior angle bisector θ_k at each corner. Their locations are given by the formula

$$\beta_{kn} = w_k + Le^{i\theta_k} e^{-\sigma(\sqrt{N_K} - \sqrt{n})} \tag{2.14}$$

where L is a characteristic length scale of the problem.

2.3 Orthogonalisation of the Basis

To evaluate our rational function at a series of M points $\mathbf{z} = [z_1, z_2, \ldots, z_M]$ we first consider the Vandermonde matrix $\mathbf{V}$ and analogous matrix for the kth pole group $\mathbf{P}_k$

$$\mathbf{V} = \begin{bmatrix} 1 & z_1 & z_1^2 & \dots & z_1^{N_0} \\ 1 & z_2 & z_2^2 & \dots & z_2^{N_0} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & z_M & z_M^2 & \dots & z_M^{N_0} \end{bmatrix} \quad \mathbf{P_k} = \begin{bmatrix} \frac{1}{z_1-\beta_{k1}} & \frac{1}{z_1-\beta_{k2}} & \dots & \frac{1}{z_1-\beta_{kN_k}} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{1}{z_2-\beta_{k1}} & \frac{1}{z_2-\beta_{k2}} & \dots & \frac{1}{z_2-\beta_{kN_k}} \\ \frac{1}{z_M-\beta_{k1}} & \frac{1}{z_M-\beta_{k2}} & \dots & \frac{1}{z_M-\beta_{kN_k}} \end{bmatrix}$$

which form the complete rational basis

$$\mathbf{B} = \begin{bmatrix} \mathbf{V} & \mathbf{P_1} & \mathbf{P_2} & \dots & \mathbf{P_K} \end{bmatrix}.$$

Evaluation of the functions f and g could be achieved by multiplying the basis by the coefficient vector $\mathbf{f} = [f_1, f_2, \dots, f_N]$ so that

$$f(\mathbf{z}) = \mathbf{B}\mathbf{f}. \tag{2.15}$$

However in practice the matrix $\mathbf{B}$ is very poorly conditioned (for the lid driven cavity example $\kappa_{\mathbf{B}} \sim 1 \times 10^{11}$) and it is necessary to create a better conditioned basis using orthogonalisation. Note that the Vandermonde matrix is a Kryolov subspace, with

$$\mathbf{V} = \mathcal{K}(\mathbf{Z}, b) \tag{2.16}$$

where

$$\mathbf{Z} = \mathrm{diag}(z_1, z_2, \dots, z_M) \qquad\qquad b = [1, 1, \dots, 1]^T$$

and we can orthogonalise $\mathbf{V}$ using the Arnoldi process [1]. This can be efficiently implemented since multiplaction by $\mathbf{Z}$ is cheap since it is diagonal.

Algorithm 1: Vandermonde with Arnoldi Process

Input: Vector $\mathbf{Z} \in \mathbb{R}^{M\times 1}$, degree of polynomial N_0

Output: Matrix $H \in \mathbb{R}^{(N_0+1)\times N_0}$, Matrix $Q \in \mathbb{R}^{M\times(N_0+1)}$

Initialize: $\mathrm{q} = \mathbf{ones}(M, 1)$, $Q[:, 1] \leftarrow \mathrm{q}$;

for $k = 1$ **to** N_0 **do**

  $\mathrm{q} \leftarrow \mathbf{z} \odot Q[:, k]$;

  for $j = 1$ **to** k **do**

    $H_{j,k} \leftarrow Q[:, j]^T \cdot \mathrm{q}$;

    $\mathrm{q} \leftarrow \mathrm{q} - H_{j,k} \cdot Q[:, j]$;

  end

  $H_{k+1,k} \leftarrow \frac{\|\mathrm{q}\|}{\sqrt{M}}$;

  $Q[:, k+1] \leftarrow \frac{\mathrm{q}}{H_{k+1,k}}$;

end

return $H \in \mathbb{R}^{(N_0+1)\times N_0}$, $Q \in \mathbb{R}^{M\times(N_0+1)}$;

Here we take $\odot$ to be the elementwise product and $\cdot$ to be the standard dot product. Each row is normalised to have norm $\sqrt{M}$ as in [2] so each element has magnitude $\mathcal{O}(1)$. To orthogonalise each rational part of the basis we use a rational Arnoldi process, as discussed in [4]. To achieve this only one line of Algorithm 1 significantly changes, with $\mathrm{q} \leftarrow \mathbf{z} \odot Q[:, k]$ becoming $\mathrm{q} \leftarrow \frac{1}{\mathbf{z}-\beta_{ik}} \odot Q[:, k]$ for the ith pole group on the kth Arnoldi iteration, and N_0 become N_K throughout [1]. In the python function `pyls.numerics.va_orthogonalise` the basis for $\mathbf{V}$ and each pole group $\mathbf{P_i}$ are orthogonalised individually and concatenated together to give a matrix $\mathbf{Q} = [Q_0, Q_1, \ldots, Q_K]$ which has the same span as $\mathbf{B}$ and is block orthogonal. The resulting hessenbergs that encode this transformation are returned as a list of matrices $[H_0, H_1, \ldots, H_K]$.

The orthogonal basis and basis derivatives need to be evaluated at new sets of points to evaluate the goursat functions and physical quantities f, g, u, v, ψ, p and ω. To achieve this the python function `pyls.numerics.va_evaluate` is used, which takes in the new set of points z_{new}, hessenbergs $[H_0, H_1, \ldots, H_K]$ and poles $\boldsymbol{\beta}$ and return the basis and basis derivatives evaluated at z_{new}.

[1] Index variables k and j here were chosen to match the indexing in the code.

Algorithm 2: Vandermonde with Arnoldi Evaluation

Input: Vector $\mathbf{Z} \in \mathbb{R}^{M\times 1}$, Hessenbergs $[H_0, H_1, \ldots, H_K]$, Poles $\boldsymbol{\beta}$

Output: Matrix $Q \in \mathbb{R}^{M\times N}$, Matrix $D \in \mathbb{R}^{M\times N}$

Initialize: q = **ones**$(M, 1)$, $Q[:, 1] \leftarrow$ q;

for $k = 1$ **to** N_0 **do**

$hkk = H_0[k+1, k]$;

$Q[:, k+1] = (Z \odot Q(:, k) - Q(:, 1:k) \cdot H(1:k, k))/hkk$;

$D[:, k+1] = (Z \odot D(:, k) - D(:, 1:k) \cdot H(1:k, k) + Q(:, k))/hkk$;

end

⋮

;

return $Q \in \mathbb{R}^{M\times N}$, $D \in \mathbb{R}^{M\times N}$;

with a similar evaluation happening for each pole group and being concatenated to the basis Q and basis derivatives D.

Given Q and D evaluated at our boundary points we can now evaluate any of the relevant physical quantities using Equations (2.10), (2.11) and (2.12).

2.4 The Algorithm

Now we have all the pieces to construct the main "Lightning Stokes" algorithm.

Algorithm 3: Lightning Stokes

Input: Vector $\mathbf{Z} \in \mathbb{R}^{M\times 1}$, Poles $\boldsymbol{\beta}$, Polynomial Degree N_0, BCs

Output: Functions psi, uv, p, omega

$[H_0, \ldots, H_K], Q =$ `va_orthogonalise`(**z**, N_0, $\boldsymbol{\beta}$);

$Q, D =$ `va_evaluate` ($[H_0, \ldots, H_K]$, $\boldsymbol{\beta}$);

U, V, PSI, P = `get_dependents`(Q, D);

A, b = `construct_linear_system`(U, V, PSI, P, BCs);

c = `solve_least_squares`(A, b);

psi, uv, p, omega = `construct_functions`(c, $[H_0, \ldots, H_K]$);

return psi, uv, p, omega;

In the above `get_dependents` uses (2.10), (2.11) and (2.12) to calculate matrices of dependent variables from the orthogonal basis and basis derivatives, `construct_linear_system` uses the user defined boundary conditions to set the indices and values of A and b corresponding to a boundary, physical quantity and value and `construct_functions` uses `va_evaluate` to create functions that return the value of the physical quantity at

an array of points in the domain. In the next section, we discuss how this algorithm was implemented in python.

3 Implementation

The python implementation is composed of one module `pyls.numerics` and three classes, `Domain`, `Solver` and `Analysis`. In this section we will discuss some of their key functions and methods and how they interact.

3.1 The Numerics Module

The large bulk of the numerics module is the two functions to construct and evaluate block orthogonal bases, `va_orthogonalise` and `va_evaluate` which are discussed in Section 2. These are analogous to `VAorthog` and `VAeval` from the MATLAB example code in [2], and validation against these functions is discussed in Section 4. These functions were separated from the `Solver` class so as to allow for optimisations like Just In Time (JIT) compilation or even rewriting in a faster language to be easily implemented. Profiling of the MATLAB example code showed that these two functions took a considerable proportion of the runtime, being the second slowest after least squares. JIT compilation is only effective for a function that will be run multiple times, and due to changes to satisfy Numba the JIT compiled versions ended up being slower that the standard pythonic version. However rewriting these functions in C++ could be investigated in the future if optimisation is required.

Another key function in `numerics` is `make_function`, which creates a callable function for each physical quantity from the hessenbergs and coefficients. This was also seperated to allow for easy JIT compilation if optimisation is required.

Some small helper functions are also included in `numerics` such as `cluster` which generates the exponential lightning pole spacing from (2.14) and `split` which splits the real vector c of coefficients for f and g into the complex valued coefficients f_j and g_j for $j \in [1, \ldots, N]$.

3.2 The Domain Class

The `Domain` class allows for easy creation of simply connected Polygonal domains that can be passed to the Solver class to add boundary conditions and solve. Creating a

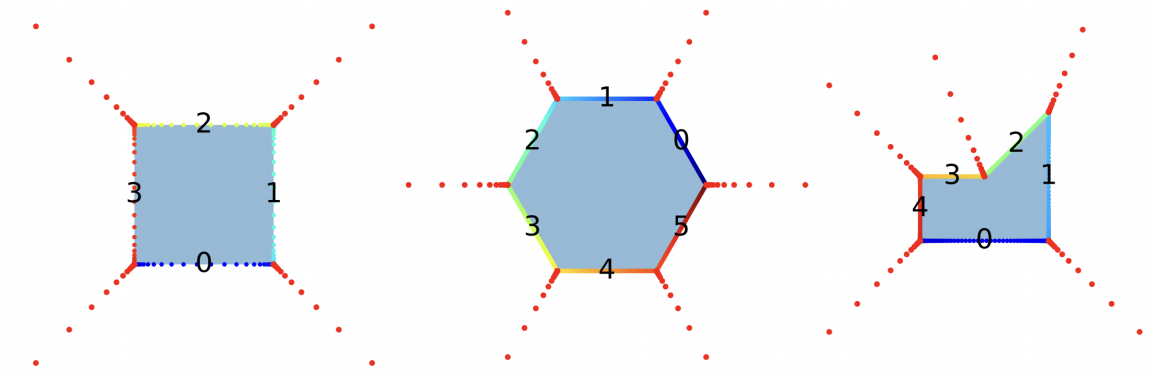

Figure 1: Domains created by the `Domain` class.

domain could be as simple as

```
# Create a square domain
from pyls import Domain
corners = [0, 1, 1 + 1j, 1j]
domain = Domain(corners)
```

which creates the square domain in Figure 1. The number of boundary points on each edge, boundary point spacing, number of poles and pole spacing parameters can also be specified to create more complex domains for different purposes, such as

```
# Create a regular hexagonal domain
corners = [np.exp(2j * np.pi * i / 6) for i in range(6)]
domain = Domain(corners, spacing="linear", L=1, sigma=1)
```

which creates the linearly spaced hexagon from Figure 1 with pole spacing parameters $L = 1$ and $\sigma = 1$ and

```
# Create a non-convex domain
corners = [0, 1, 1 + 1j, 0.5 + 0.5j, 0.5j]
domain = Domain(corners, num_boundary_points=300, num_poles=30)
```

which creates the non-convex pentagonal domain. The corners should be passed in anticlockwise order [2], and will be reoriented to be anticlockwise if they are not. The sides are labelled automatically in the order in which the corners are passed (if positively oriented) and poles are automatically generated to lie with the spacing given by (2.14) with parameters L and σ along the exterior angle bisectors θ_k, where

[2]Here "anticlockwise" means the polygon is positively oriented

the exterior direction is determined using the sign of the cross product of the two edges associated with the current node. The boundary point spacing exponentially clusters points towards the corners by default using

```
tanh(linspace(-10, 10, num_boundary_points))
```

but a linear spacing can be specified.
The method `Domain.show` was added to check the side labels and see that the created domain looks reasonable. A dictionary of the indices of the boundary points corresponding to each side is created and stored to `Domain.indices` which will later be accessed by the `Solver` class for setting boundary conditions. Also to easily check whether an array of points lies in the domain the methods `Domain.__contains__` and `Domain.mask_contains` were added. These are later used by the `Analysis` for determining physical quantities within the domain and plotting.

3.3 The Solver Class

The `Solver` class allows the user to add boundary conditions to the domain and solve the problem. Its constructor takes a domain of type `Domain` and the degree of the polynomial part of the basis. The user can then add boundary conditions using the `Solver.add_boundary_condition` method. For example, creating an instance of the solver and adding boundary conditions for Pouseuille flow on a Square domain which will be solved with polynomial degree 24 can be achieved by the following code.

```
sol = Solver(dom, 24)
# walls on sides 0 and 2, inlet on side 1, outlet on side 3
sol.add_boundary_condition("0", "u(0)", 0)  # no slip
sol.add_boundary_condition("0", "v(0)", 0)  # no penetration
sol.add_boundary_condition("2", "u(2)", 0)
sol.add_boundary_condition("2", "v(2)", 0)
sol.add_boundary_condition("1", "u(1)", "1 - y**2")  # parabolic inlet
sol.add_boundary_condition("1", "v(1)", 0)
sol.add_boundary_condition("3", "p(3)", 0)
sol.add_boundary_condition("3", "v(3)", 0)
```

The `add_boundary_condition` method takes the label of a side, a symbolic expression to be evaluated involving the dependent variables and a value for that ex-

pression to take and adds it to the dictionary of boundary conditions. The value can be a number, `numpy.ndarray` or symbolic expression involving the independent variables (x and y). The syntax of the symbolic expressions is checked using `Solver.validate` and evaluated by using `Solver.evaluate`. `Solver.evaluate` works by taking each token in a symbolic expression and replacing it with it's python counterpart to form a string of python code. This string can then be evaluated using the python function `eval`. For example `"u(0)"` is evaluated by first converting to `"self.U[self.domain.indices[0]]"` and then calling `eval` on that string. This means pythonic syntax can be used to more complicated expressions, such as setting periodic conditions by

```
sol.add_boundary_condition("1", "u(1)-u(3)[::-1]", 0)
```

where `[::-1]` is used to flip the array on the opposite edge to have the same orientation.

3.4 The Analysis Class

3.5 Example Problem: Lid Driven Cavity Flow

4 Validating PyLS

5 Performance and Optimisation

6 Conclusion

References

[1] Pablo D. Brubeck, Yuji Nakatsukasa, and Lloyd N. Trefethen. "Vandermonde with Arnoldi". eng. In: *SIAM review* 63.2 (2021), pp. 405–415. ISSN: 0036-1445.

[2] Pablo D. Brubeck and Lloyd N. Trefethen. "Lightning Stokes Solver". eng. In: *SIAM journal on scientific computing* 44.3 (2022), A1205–A1226. ISSN: 1064-8275.

[3] Abinand Gopal and Lloyd N. Trefethen. "Solving Laplace Problems with Corner Singularities via Rational Functions". eng. In: *SIAM journal on numerical analysis* 57.5 (2019), pp. 2074–2094. ISSN: 0036-1429.

[4] Miroslav S. Pranić et al. “A rational Arnoldi process with applications”. eng. In: *Numerical linear algebra with applications* 23.6 (2016), pp. 1007–1022. ISSN: 1070-5325.