Efficient point location with ploc

Scientific computing in Rust

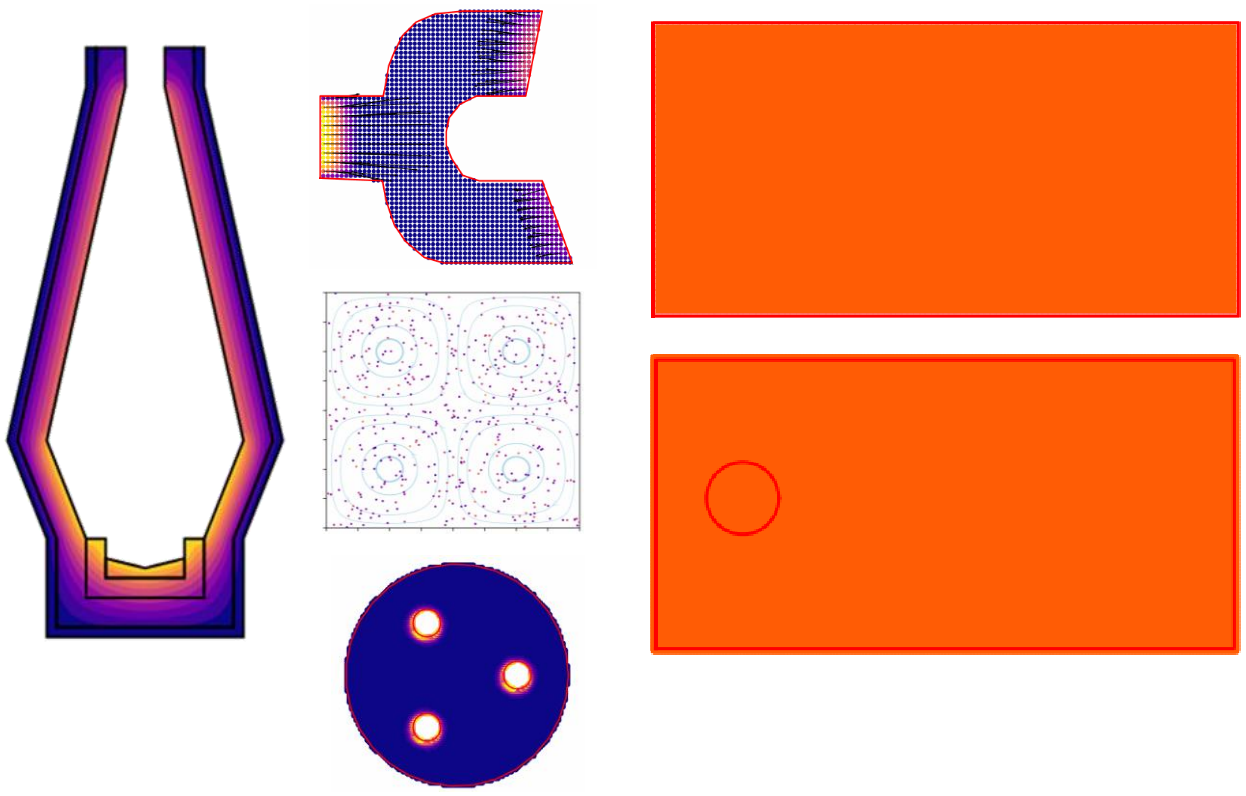
06/06/2025

Who am I?

- Background in solid mechanics
- Worked as a Cast3M developer for a while (FEM code developed at the French Atomic Energy Commission)
- Co-founded DotBlocks in 2023

  We want to make numerical simulation *more accessible*
- Prototypes in Python, then Rust is used for performance and robustness

Lattice Boltzmann (LBM)

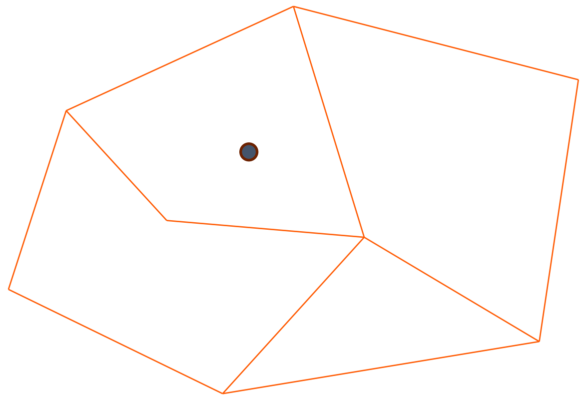


Boundary Element Method (BEM)

…and more!

The point location problem

Given a *mesh* and some query points, we want to know which cells/elements of the mesh contain each query point



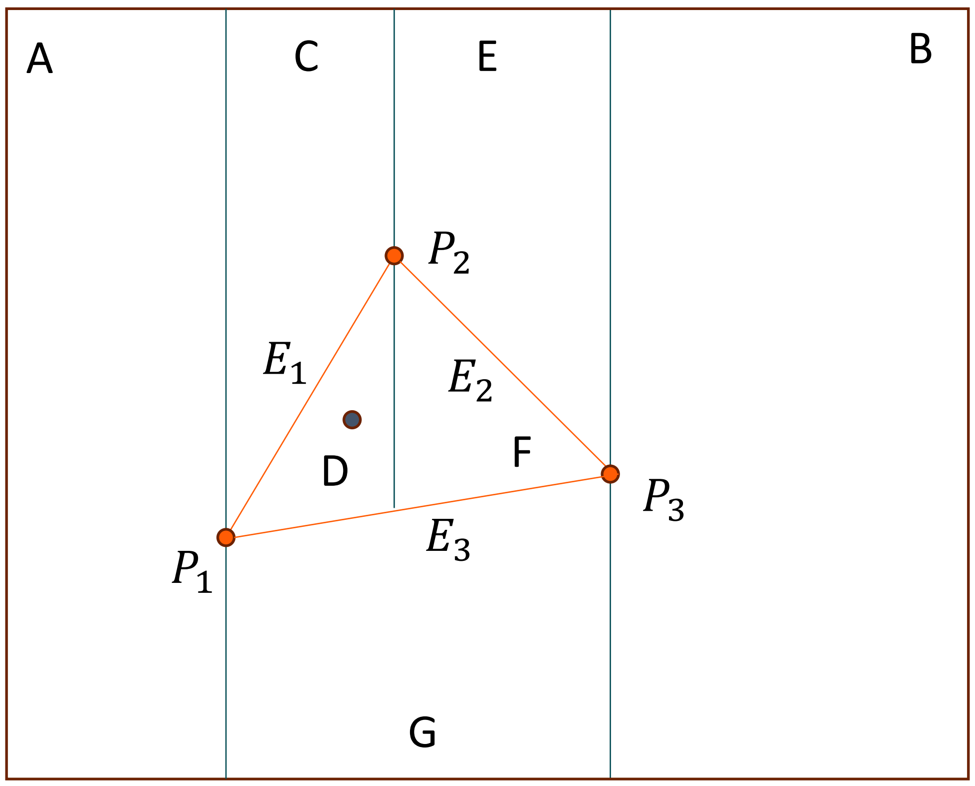
Why is that an interesting problem?

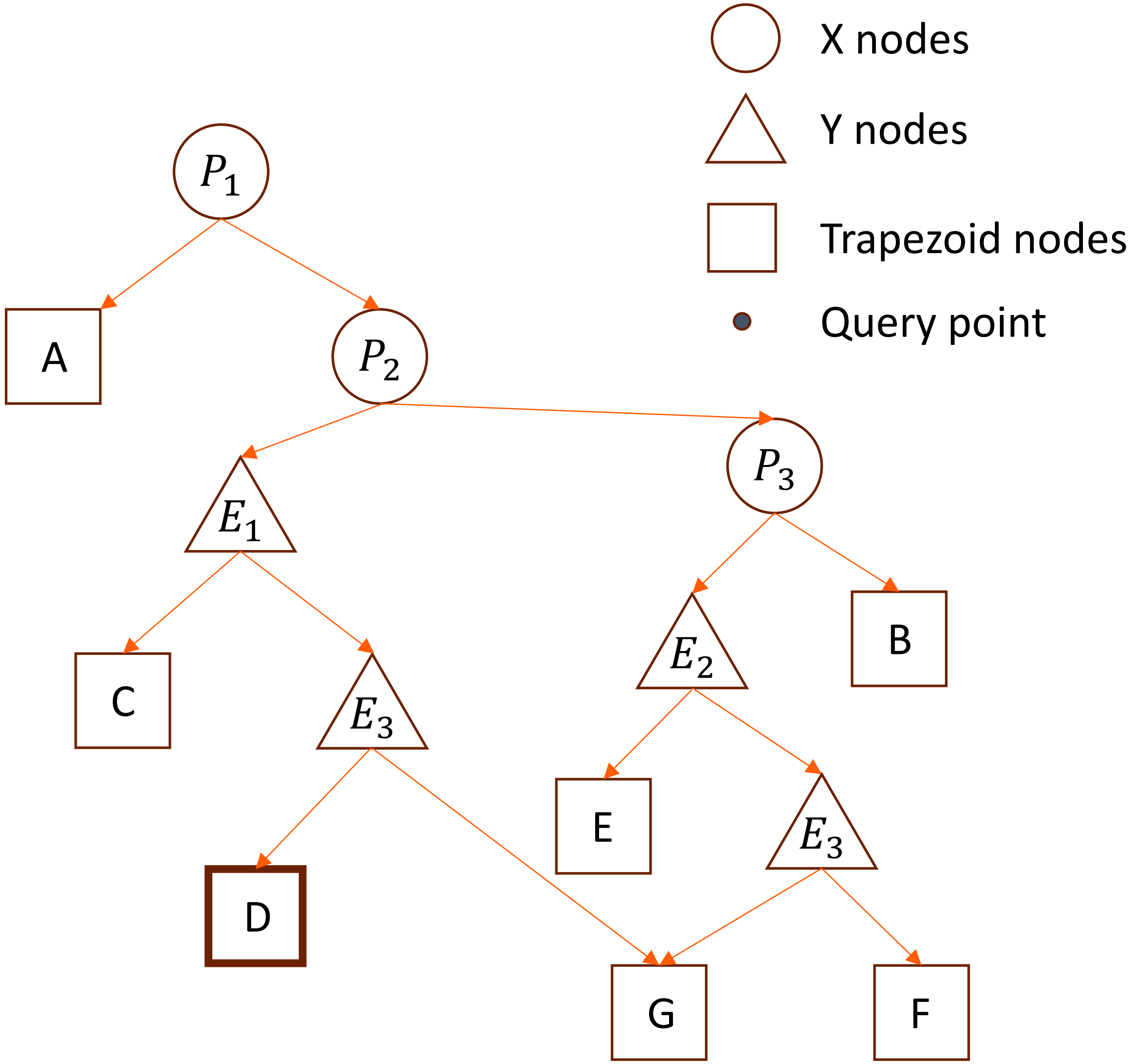
- Comes up in:
  - Interpolation of discrete fields
  - Computing statistical quantities in Lagrangian particle tracking
  - In many other areas (computer graphics, CAD, etc…)
- Leads to some very interesting data structures
- Brute force approach is $O(n)$ for each query point… Can we do better?

The trapezoidal map

- A *randomized incremental* algorithm which constructs in $O(n \log n)$ a data structure with $O(n)$ size and $O(\log n)$ query time for the point location problem
- Data structure: a *directed acyclic graph* (DAG) with three types of nodes:
  - X nodes
  - Y nodes
  - Trapezoids (leaf nodes)
- Edges of the mesh are *shuffled*, and then added one at a time
- At each iteration, we have a search structure we can use to find where the next edge should be inserted!

Basic example





ploc

- Available on crates.io and GitHub
- Inspired by the matplotlib implementation
  - Generalized to any kind of mesh (multiple cell types and any kind of polygon)
  - Leverages rayon for speed (each query is independent)

Code example

```rust
// The `TrapMap` struct implements the `PointLocator` trait
use ploc::{Mesh, PointLocator, TrapMap};
fn main() {
    // Create a simple mesh (a regular 10x10 grid)
    let (xmin, xmax) = (0., 10.);
    let (ymin, ymax) = (0., 10.);
    let n = 10;
    let mesh = Mesh::grid(xmin, xmax, ymin, ymax, n, n).unwrap();

    // Create the trapezoidal map
    let trap_map = TrapMap::from_mesh(mesh);

    // Locate a single point
    assert_eq!(trap_map.locate_one(&[0.5, 0.5]), Some(0));

    // Locate multiple points
    let query: Vec<_> = (0..10)
        .flat_map(|iy| (0..10).map(move |ix| [0.5 + ix as f64, 0.5 + iy as f64]))
        .collect();
    let expected: Vec<_> = (0..100).map(Some).collect();
    assert_eq!(trap_map.locate_many(&query), expected);
}
```

Rust-specific implementation details

- DAG part not super obvious
  - mpl C++ implementation uses pointers all over the place
  - Possible in Rust but would require a lot of unsafe because parents need references to their children and children to their parents (probably something like the Production-Quality Unsafe Doubly-Linked Deque would work)
  - I instead took inspiration from Arena-Allocated Trees in Rust
    - One Vec of nodes (the node is generic over the data it holds)
    - Each node has a Vec of parent ids and a SmallVec of children ids (always 0 or 2 children)
  - Implemented an "entry API" like that of std::collections::HashMap to link nodes and to mutate the values they hold

Comparison with the matplotlib implementation

- High compatibility with the mpl implementation
  - Property-based testing with hypothesis shows perfect agreement for query points "not too close to mesh vertices"
  - For query points very close to mesh vertices, floating point arithmetic + dependence on the C++ std library used to compile mpl get in the way…
  - Cannot be helped as far as I can tell
- Single-threaded implementation roughly 20% faster than mpl
- Multi-threaded implementation 8x faster than mpl
- But peak memory usage is about 1.2x to 2x higher 😬
  - Due to the DAG, which holds a Vec of enums whose variants have very different sizes…
  - The biggest variant is the least frequent node type!

```
#[derive(Clone, Debug)]
pub(crate) enum Node {
    X(usize),          // 16 bytes
    Y(Edge),           // 56 bytes
    Trap(Trapezoid), // 176 bytes!
}
```

What’s next?

- Release Python bindings (almost ready to publish on PyPI)
- Reducing the memory footprint to get on par with mpl
- Investigate other possible data structures
  - Quadtree implementations seem promising
  - *Much* simpler to implement
  - Very good performance for large meshes
  - Potentially generalizable to 3D
- Provide more information in the output (is the query point on an edge? On a vertex?)

DotBlocks

https://www.dotblocks.com/

contact@dotblocks.fr