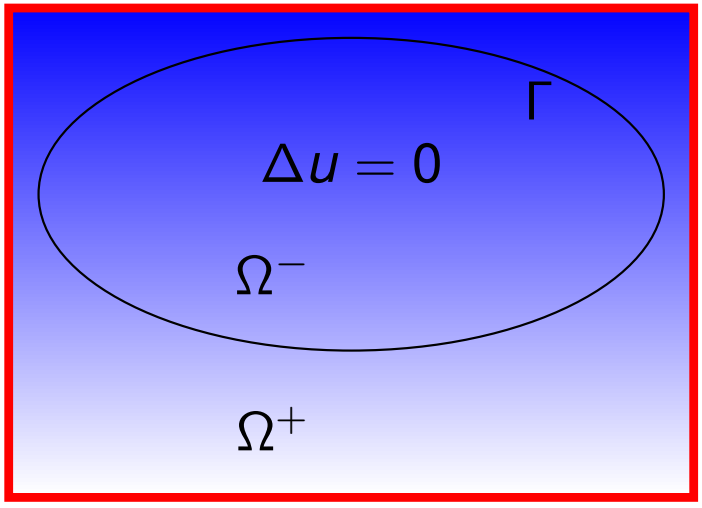
A rusty roadmap to integral equations at exascale

Timo Betcke
t.betcke@ucl.ac.uk
University College London

Joint with S. Kailasa, M. Scroggs and many other collaborators

Supported by EPSRC Excalibur grants EP/W026260/1 and EP/W007460/1

Some Maths to set the stage



$$\Delta u(\mathbf{x}) := \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}$$

Application areas: Heat diffusion, electrostatics, etc.

Boundary condition:
$u(\mathbf{x}) = f(\mathbf{x})$.

Represent solution as

$$u(\mathbf{x}) = \int_\Gamma g(\mathbf{x}, \mathbf{y}) \phi(\mathbf{y}) ds(\mathbf{y}), \quad \mathbf{x} \in \Omega$$

Need to solve $V\phi = f$ with

$$[V\phi](\mathbf{x}) = \int_\Gamma g(\mathbf{x}, \mathbf{y}) \phi(\mathbf{y}) ds(\mathbf{y}), \quad \mathbf{x} \in \Gamma$$

How to discretise the integral equation?

We want to solve $V\phi = f$. Introduce a triangulation $\mathcal{T}$ of the domain Ω into triangles τ_j. Define basis fct.

$$\phi_j(\mathbf{x}) = \begin{cases} 1 & \mathbf{x} \in \tau_j \\ 0, & \text{otherwise} \end{cases}$$

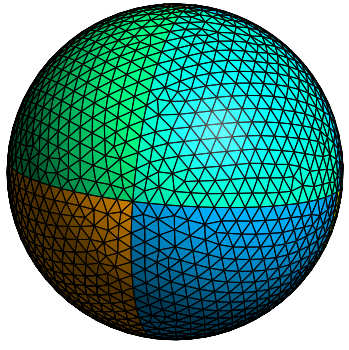
We approximate $\phi = \sum_{j=1}^{N} c_j \phi_j$. Multiplying with ϕ_i and integrating gives

$$\int_\Gamma \phi_i(\mathbf{x}) \left[V\phi\right](\mathbf{x}) ds(\mathbf{x}) = \int_\Gamma f(\mathbf{x})\phi_i(\mathbf{x}) ds(\mathbf{x}), i = 1, \ldots, N$$

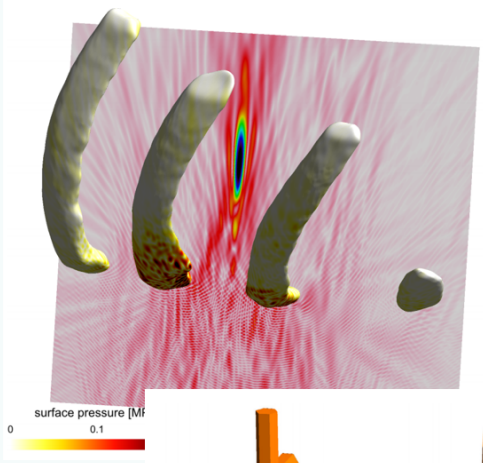
Solve $\mathbf{Vc} = \mathbf{b}$

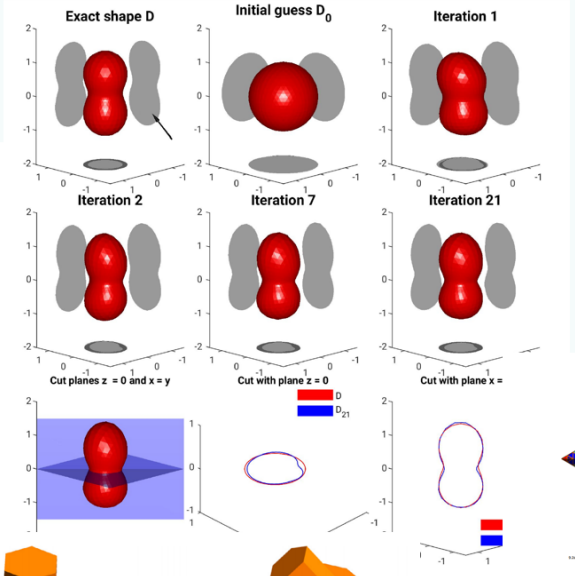
$$\mathbf{V}_{ij} = \int_{\tau_i} \int_{\tau_j} g(\mathbf{x}, \mathbf{y}) ds(\mathbf{y}) ds(\mathbf{x})$$

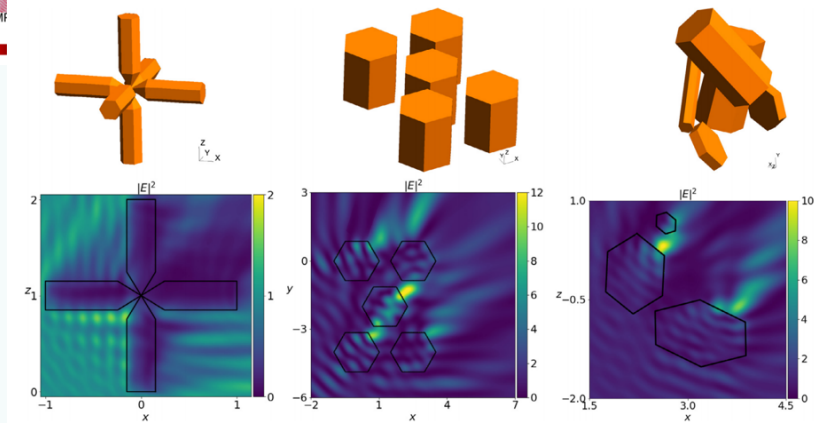
$$\mathbf{b}_i = \int_\Gamma f(\mathbf{x})\phi_i(\mathbf{x}) ds(\mathbf{x})$$



The Bempp boundary element software







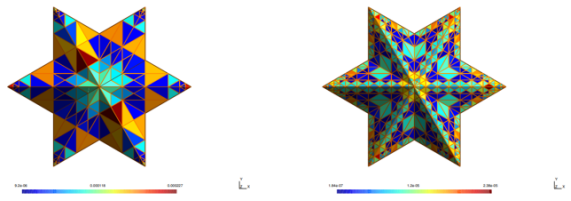


FIGURE 17. Example 3.3: Mesh $\mathcal{T}_\ell$ and distribution of the error estimator η_ℓ^2 for $\ell = 10$ (left) with 360 elements and $\ell = 15$ (right) with 1524 elements.

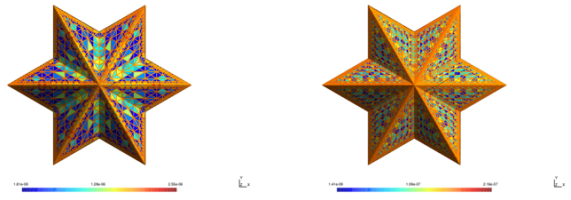


FIGURE 18. Example 3.3: Mesh $\mathcal{T}_\ell$ and distribution of the error estimator η_ℓ^2 for $\ell = 20$ (left) with 5754 elements and $\ell = 25$ (right) with 20304 elements.

The early C++ years of Bempp[1]

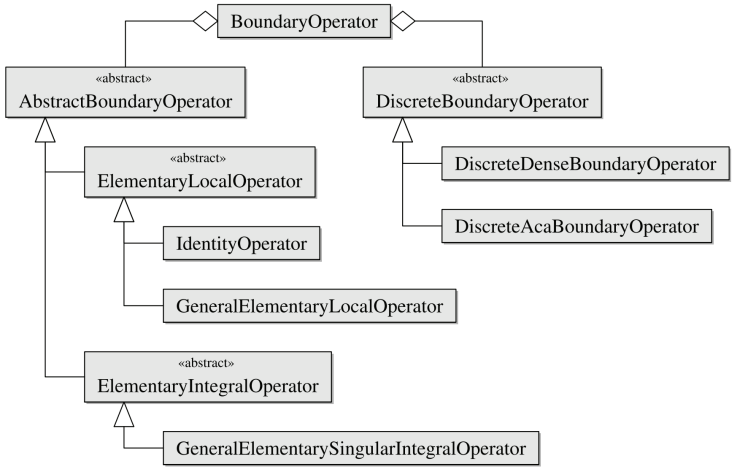


Fig. 3. Relationships between the main classes representing boundary operators.

- Complex object hierarchy
- Highly templated code
- Dependencies on complex packages (Boost, BLAS, Armadillo, Trilinos, DUNE, Intel TBB)

Python Interface

- Auto-generated with Swig
- Thin layer on C++ code
- Almost no logic in the Python layer, all handled by C++

Over time moved all high-level functions into Python, only kept low-level performance critical data structures in C++.

[1] W. Śmigaj et. al., *Solving boundary integral problems with BEM++*, ACM Transactions on Mathematical Software 41 (2015), pp. 1 - 40

Fast JIT compiled Galerkin BEM in Python

UCL

Numba for grid iterations

- Grid topology computations
- Python callables
- Routines on grid functions, (L^2-norm, etc.)
- Fallback for operators

OpenCL for matrix assembly/evaluation

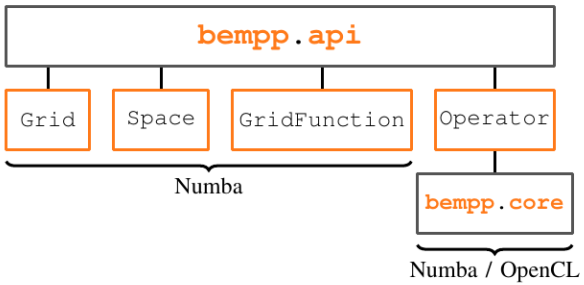
- All potential evaluations
- Mass matrices
- FMM Near-Field

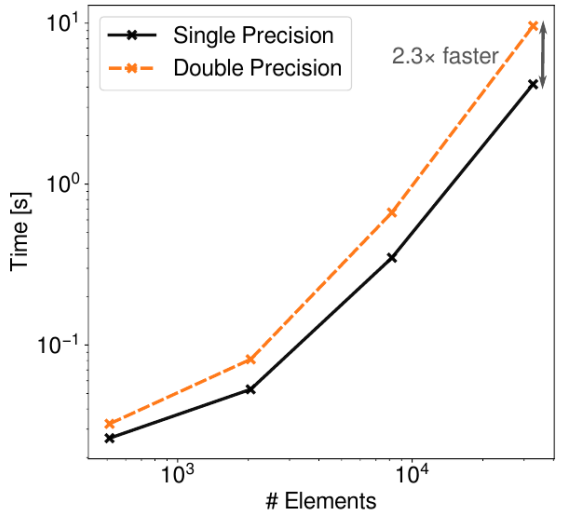
Other technologies

- Scipy for spare matrix operations and iterative solvers
- Numpy dense solvers

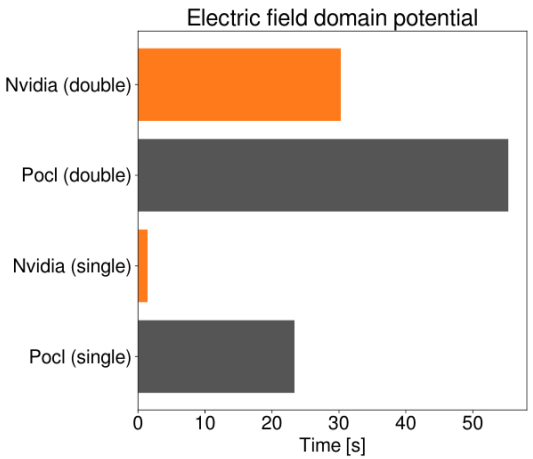
- All logic controlled from Python (easy to hack on).
- Very fast SIMD optimized dense matrix assembly.
- But need lots of OpenCL C99 kernel code.

Characteristics of Bempp-cl[2]





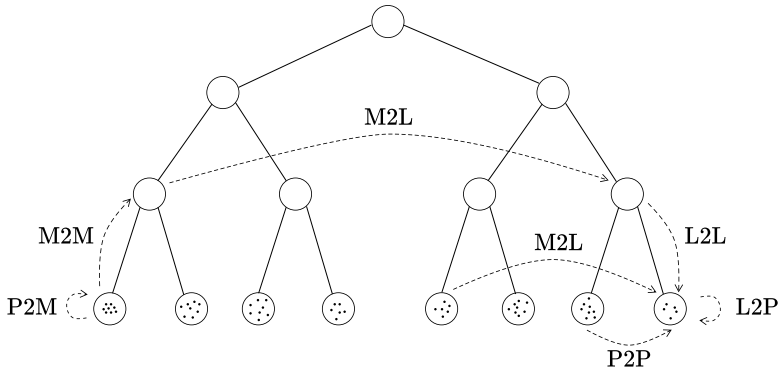
- Top: Structure of Bempp-cl
- Top-Right: AVX acceleration in Bempp-cl
- Bottom-Right: Offloading of a domain potential operator

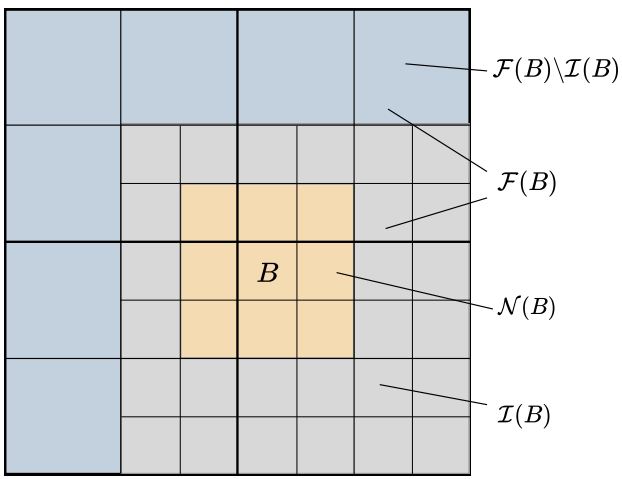


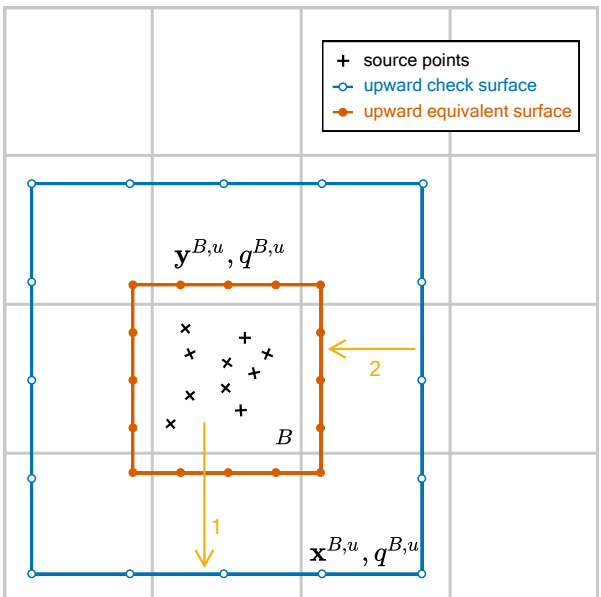
[2] T. Betcke, M. Scroggs, *Designing a high-performance boundary element library with OpenCL and Numba*, Computing in Science and Engineering, 23 (2021), pp. 18 - 28

Exafmm-t: High-Performance KIFMM[3]

- Kernel-Independent FMM library.
- Written in C++, Provides Python Interface.
- Highly efficient at higher accuracies.







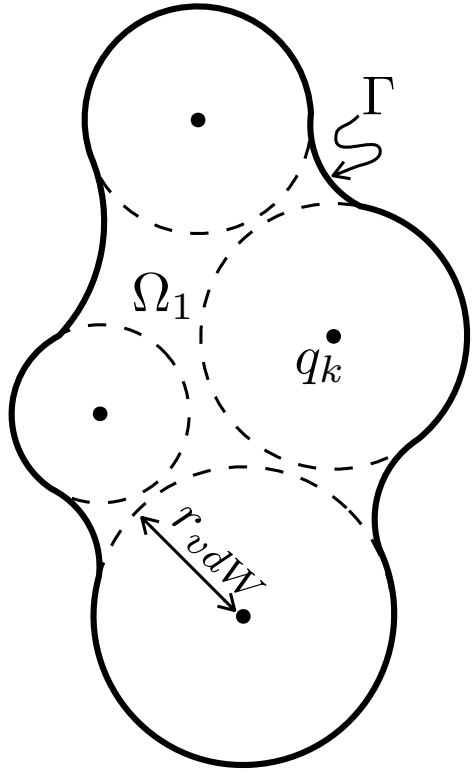
[3] T. Wang, R. Yokota, L. Barba, *ExaFMM: a high-performance fast multipole method library with C++ and Python interfaces*, Journal of Open Source Software 6(61), 3145

Why large-scale BEM?

Solve Poisson-Boltzmann equation

$$\Delta\phi_1 = \frac{1}{\epsilon_1}\sum_k q_k\delta(\mathbf{r},\mathbf{r}_k) \text{ in } \Omega_1$$

$$(\Delta - \kappa^2)\phi_2 = 0 \text{ in } \Omega_2$$



Interface conditions on Γ:

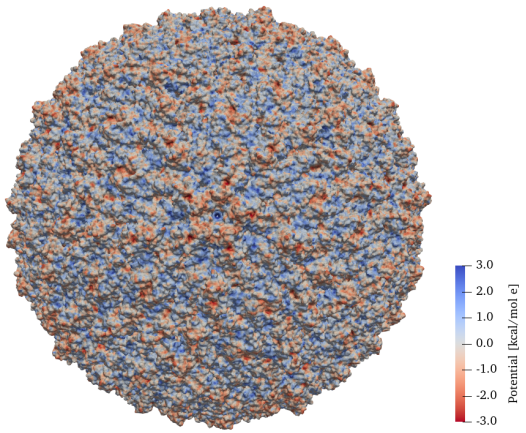
$$\phi_1 = \phi_2,$$

$$\epsilon_1\frac{\partial\phi_1}{\partial\mathbf{n}} = \epsilon_2\frac{\partial\phi_2}{\partial\mathbf{n}}$$

Let $\phi_1 = \phi_{reac} + \phi_{coul}$ (ϕ_{coul} is potential generated from the point charges only). Then want to compute the solvation energy.

$$\Delta G^{polar}_{solv} = \frac{1}{2}\sum_{k=1}^{N_q} q_k\phi_{reac}(\mathbf{r}_k)$$

Exascale FEM/BEM modeling



- Electrostatic simulation of Zika virus (10m elements) on single compute node[4]
- Mass-matrix preconditioned single trace formulation.
- FMM acceleration via ExaFMM.

Goal: Full FEM/BEM modelling of complex protein surfaces.

$$\begin{bmatrix} \epsilon_1 A & -\epsilon_1 M^T \\ \left(\frac{1}{2} I + K\right) & \frac{\epsilon_1}{\epsilon_2} V \end{bmatrix} \begin{bmatrix} \vec{\phi}_1 \\ \vec{\lambda}_2 \end{bmatrix} = \begin{bmatrix} \vec{f} \\ 0 \end{bmatrix}.$$

A: FEM matrix, K: Double-Layer operator, V: Single-Layer operator, M: Mass matrix

Want to scale up to billions surface dofs.

[4] T. Wang, C. D. Cooper, T. Betcke, L. A. Barba, *High-productivity, high-performance workflow for virus-scale electrostatic simulations with Bempp-Exafmm*, submitted (2021)

Deconstructing a BEM operator call

Discrete integral operator evaluation takes the form

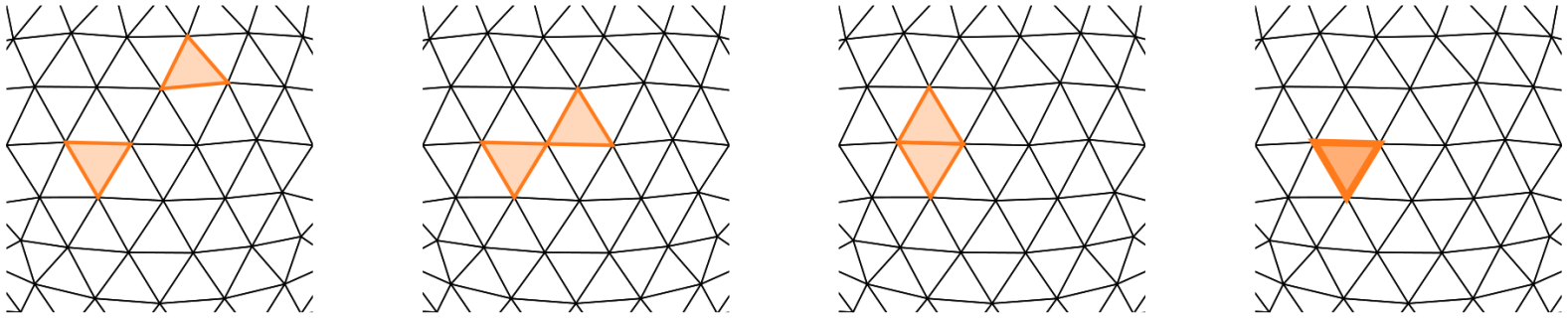
$$Vx = P_{dual}^T (G - C) P_{dom} \overline{x} + S\overline{x}$$

- G dense.
- All other matrices highly sparse.

- $P_{dom/dual}$: Map from domain/dual space cefficients to values at quadrature points
- G matrix of Green's fct. evaluations at regular Gauss points
- C: Correction matrix containing contribution of G associated with adjacent triangles
- S: Contributions of singular quadrature rule on adjacent triangles

- Parallel fast methods (H-matrices/FMM) for G.
- Distributed sparse linear algebra for everything else.

The correction matrix



Problem: Any fast code for G also evaluates the near-field. Near-field can contain adjacent triangles (singular quadrature rules), and non-adjacent triangles (standard triangle Gauss rule).

The sources and target points in the FMM are arising from the non-singular quadrature rule.

Solution: After FMM evaluation subtract out the regular quadrature rule contribution from adjacent triangles and add in the correct singular quadrature rule contribution.

Software Technologies

Python

- Good for higher level logic.
- Efficient libs for array processing.
- Performance for complex algorithms in Python (e.g. FMM) difficult[5].

Python + Code Gen

- Most flexible solution.
- Complex framework.
- What low-level language to use?

C++

- Complex Makefiles.
- Deployment difficult.
- C++20 support still patchy.

[5] S. Kailas et. al., *PyExaFMM: an exercise in designing high-performance software with Python and Numba*, to be submitted

Along came Rust

- Modern language design.
- Great tooling.
- Compile time safety guarantees.
- Simple deployment.
- Python integration trivial.

Design a set of micro-libraries in Rust that can be easily composed into large parallel FEM/BEM applications.

A set of micro-libraries in Rust

UCL

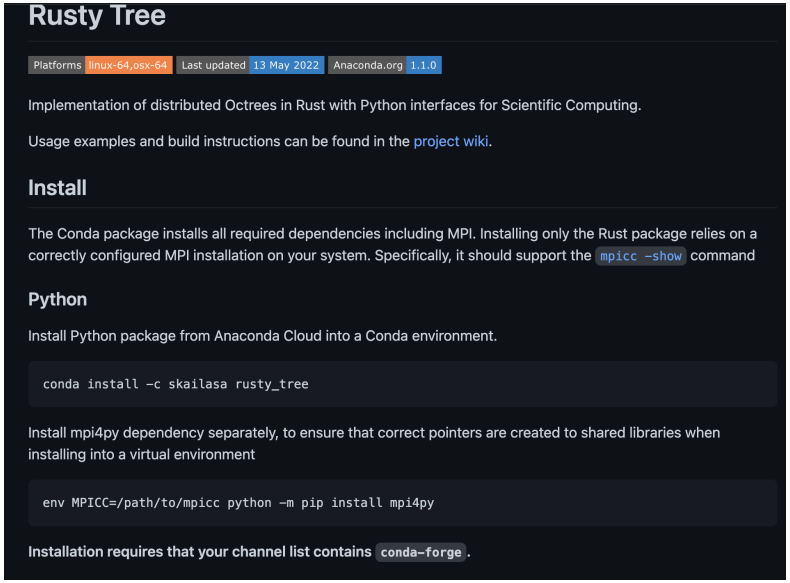
Blue: Finished or in work. Red: In planning

tree	Fast distributed octree management
compression	Linear algebra tools for low-rank compression
field	Field translation operators
green-kernel	Direct evaluation of Green's functions
quadrature	Collection of quadrature rules
element	Finite element definitions
grid	Distributed management of surface grids
fmm	High-Level distributed FMM loops
surface-diff-op	Distributed assembly of surface differential operators
bempp	High-Level distributed BEM interface

- All libraries independent projects
- MPI support where necessary
- C interface + Python bindings for most functionality
- From October team of 1 PhD, 1 Postdoc + Research Software Engineers

rusty-tree

- Modeled on Sundar et. al[6]
- Distributed load-balancing, 2:1 refinement
- Full Python interface
- Can use Rust or Python MPI communicator



Rusty Tree

Platforms linux-64,osx-64 | Last updated 13 May 2022 | Anaconda.org 1.1.0

Implementation of distributed Octrees in Rust with Python interfaces for Scientific Computing.

Usage examples and build instructions can be found in the project wiki.

Install

The Conda package installs all required dependencies including MPI. Installing only the Rust package relies on a correctly configured MPI installation on your system. Specifically, it should support the `mpicc -show` command

Python

Install Python package from Anaconda Cloud into a Conda environment.

```
conda install -c skailasa rusty_tree
```

Install mpi4py dependency separately, to ensure that correct pointers are created to shared libraries when installing into a virtual environment

```
env MPICC=/path/to/mpicc python -m pip install mpi4py
```

Installation requires that your channel list contains `conda-forge`.

---

[6] H. Sunar, R. S. Sampath, G. Biros, *Bottom Up Construction and 2:1 Balance Refinement of Linear Octrees in Parallel*, 30 (2008, pp. 2675 - 2708

rusty-compression

UCL

Low-rank compression

$$A \approx UV^H$$

$A \in \mathbb{C}^{m \times n}$, $U \in \mathbb{C}^{m \times k}$, $V \in \mathbb{C}^{k \times n}$.

- Dense QR, SVD decomposition
- Interpolative decompositions
- Randomized low-rank decompositions

- Required for algebraic compression of translation operators.
- Currently uses Blas/Lapack/ndarray.
- Plan to rebase on new linear algebra core.

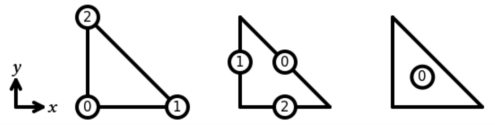
rusty-element

```rust
/// A 0- to 3- dimensional reference cell
7 implementations
pub trait ReferenceCell {
    const DIM: usize;

    /// The dimension of the reference cell (eg a triangle's dimension is 2, tetrahedron's dimens
    fn dim(&self) -> usize {
        Self::DIM
    }

    /// The vertices of the cell
    ///
    /// The first dim components represent the first vertex, the next dim the second vertex, and
    fn vertices(&self) -> &[f64];

    /// The edges of the cell
    ///
    /// The first 2 components are the vertex numbers of the endpoints of the first edge, the nex
    fn edges(&self) -> &[usize];

    /// The faces of the cell
    ///
    /// The first `self.faces_nvertices()[0]` components are the vertex numbers of vertices of th
    fn faces(&self) -> &[usize];

    /// The number of vertices adjacent to each face
    fn faces_nvertices(&self) -> &[usize];
```

Reference Cell Trait

Triangle

In Basix, the sub-entities of the reference triangle are numbered as follows:

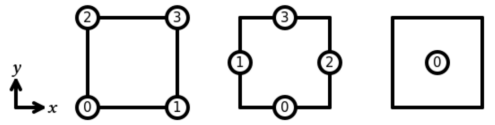


The following elements are supported on a triangle:

- Lagrange
- Nédélec first kind
- Raviart-Thomas
- Nédélec second kind
- Brezzi-Douglas-Marini
- Regge
- Crouzeix-Raviart
- Bubble

Quadrilateral

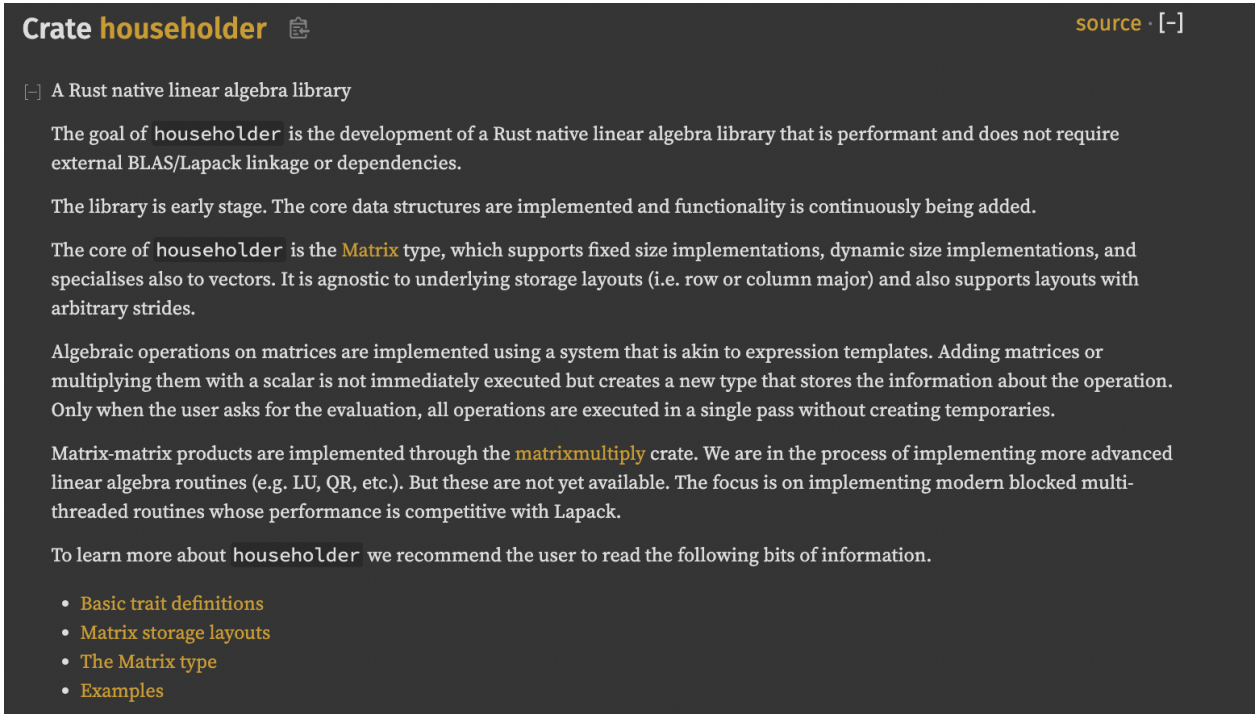
In Basix, the sub-entities of the reference quadrilateral are numbered as follows:



Basix elements

- Modeled on Basix (FEeniCS finite element evaluation library).
- Generic interface and implementation of element definitions.
- Support for higher order triangular and quadrilateral elements.
- Separate from grid library (grid library does data distribution and iteration, rusty-element provides element information).

Householder - A new linear algebra kernel in Rust



Crate householder

source · [-]

[–] A Rust native linear algebra library

The goal of `householder` is the development of a Rust native linear algebra library that is performant and does not require external BLAS/Lapack linkage or dependencies.

The library is early stage. The core data structures are implemented and functionality is continuously being added.

The core of `householder` is the Matrix type, which supports fixed size implementations, dynamic size implementations, and specialises also to vectors. It is agnostic to underlying storage layouts (i.e. row or column major) and also supports layouts with arbitrary strides.

Algebraic operations on matrices are implemented using a system that is akin to expression templates. Adding matrices or multiplying them with a scalar is not immediately executed but creates a new type that stores the information about the operation. Only when the user asks for the evaluation, all operations are executed in a single pass without creating temporaries.

Matrix-matrix products are implemented through the matrixmultiply crate. We are in the process of implementing more advanced linear algebra routines (e.g. LU, QR, etc.). But these are not yet available. The focus is on implementing modern blocked multi-threaded routines whose performance is competitive with Lapack.

To learn more about `householder` we recommend the user to read the following bits of information.

- Basic trait definitions
- Matrix storage layouts
- The Matrix type
- Examples

- New dense/sparse linear algebra kernel in Rust.
- Based on Rust implementation of expression templates (similar to C++ eigen).
- Supported by ARCHER2 eSAFE funding.

Diving into Rust with Householder

UCL

- Matrix traits
- Mutable and immutable access
- Basic data containers (vectors and slices)
- Reference types and lifetimes
- A generic matrix structure
- Documentation and unit tests

The Rust Scientific Computing ecosystem

UCL

Core utilities

- Num - Traits and utility routines for floating point types.
- Cauchy - A complex scalar type library (uses Num).
- approx - Unit test routines for floating point types.
- Serde - Serialization library.

Linear Algebra/Machine Learning

- ndarray - multidimensional array type in Rust.
- nalgebra - Rust linear algebra library with Lapack bindings.
- householder - Early stages linear algebra library.
- Linfa - Native machine learning in Rust.
- tensorflow - Rust bindings to tensorflow.
- petsc-rs - Rust bindings for Petsc.

Parallelization

- rsmpi - MPI Interface for Rust.
- Rayon - Fork/Join threading, parallel loops and iterators.
- async/await (Tokio, etc.) - frameworks for asynchroneous execution.

Language bindings

- PyO3 - Binding complex Rust types to Python.
- maturin - Simple tools to wrap Rust C API bindings with Python cffi.
- Any language that supports C bindings can talk to Rust easily.

Still lots of moving parts in the ecosystem but an enormous amount of development happening.

Some final messages

- Using Python + JIT (e.g. Numba) works well for simple array oriented algorithms. More problematic for complex algorithms.
- We are at the start of a multi-year journey to develop very large-scale integral equation solvers in Rust.
- Rust is ready to be looked at by Scientific Computing Community (though far away from the level of maturity of the C++ ecosystem)
- Rust makes little sense when legacy C++ code exists or there is need for heterogeneous compute with SyCL or Cuda.
- But consider and evaluate Rust if you start a completely new project.