From low to high level languages and back

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

Joint with S. Kailasa (UCL)

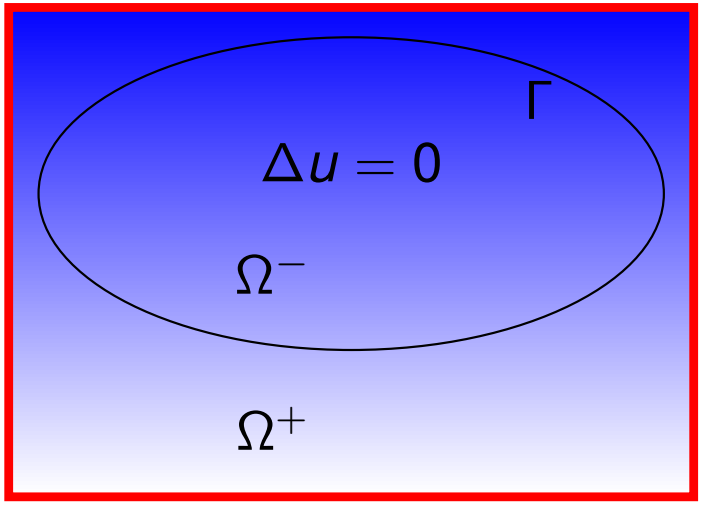
About me

- Studied Computational engineering at Hamburg University of Technology (1998 - 2002)
- DPhil at Oxford Numerical Analysis Group, (2002 - 2005, graduated 2006)
- Postdocs in Braunschweig, Manchester, Reading (2002 - 2009)
- EPSRC Career Acceleration Fellow, Reading (2009 - 20011)
- Since 20011 at UCL Department of Mathematics
- Since 2018 Professor of Computational Mathematics
- Since 2022 ARC Associate Director

What you don’t find in my CV

- First PC: 25MHz 386DX with 2MB Ram, just enough to play Civilisation
- I owned the original Nintendo Entertainment System and played Super Mario on a black and white TV
- Favourite Youtube Channels: PBS Spacetime and Digital Foundry
- Long term Vim user, now overcome by the dark side and moved to VS Code
- I beat Elden Ring (my son still tells me I am bad at games since I never won Fall Guys)
- Best song of all times: Mad World by Tears for Fears (or as my son says, old and cringy)

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.

As boundary condition on Γ we use $u(\mathbf{x}) = f(\mathbf{x})$.

Green's fct.: The function $g(\mathbf{x}, \mathbf{y}) = \frac{1}{4\pi|\mathbf{x}-\mathbf{y}|}$ satisfies $\Delta_{\mathbf{x}} g(\mathbf{x}, \mathbf{y}) = 0$ for every $\mathbf{x} \neq \mathbf{y}$. This function is special. It is called the Green's fct. (or fundamental solution) of the PDE.

Representing the solution

We want to find a function ϕ defined on Γ such that

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

We plugin the known boundary data and arrive at the **boundary integral equation**

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

We call V the single-layer boundary integral operator defined by

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

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.

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

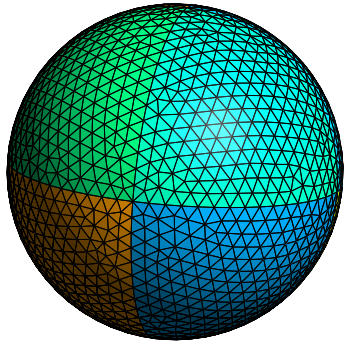
We approximate $\phi = \sum_{j=1}^{N} c_j \phi_j$. Multiplying with ϕ_i and integrating gives

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

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

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

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

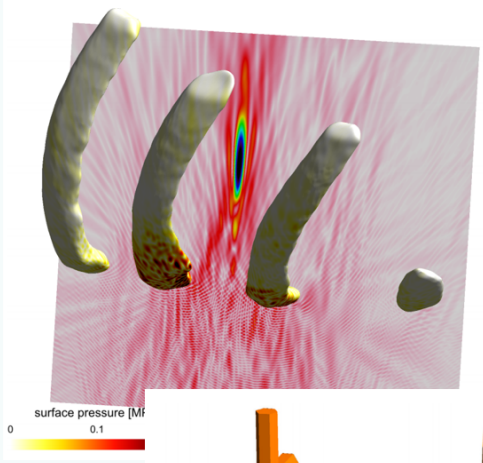


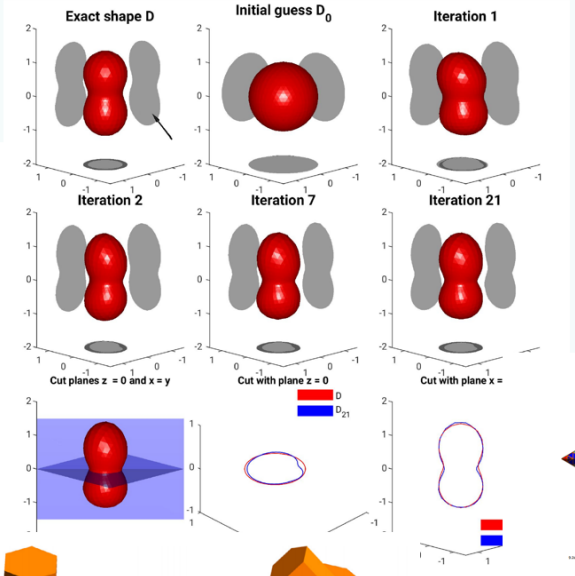
Sounds easy - or not?

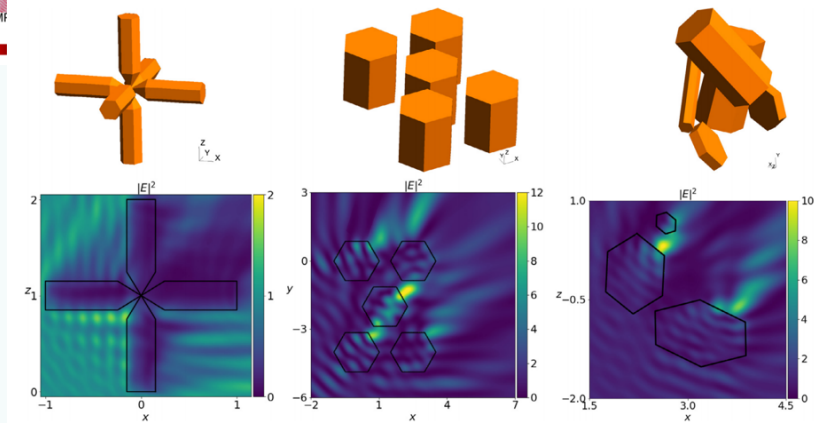
Two difficulties

- The Green's function g is singular for $\mathbf{x} = \mathbf{y}$. Each integral of a triangle with itself or and adjacent triangle requires special singularity treatment.
- The matrix $\mathbf{V}$ is dense with N^2 elements if there are N triangles. Huge assembly cost, and for large problems need linear scaling in complexity. (Can solve this by using Fast Multipole Methods for large problems, but complex and won't go into detail here)

The Bempp boundary element software







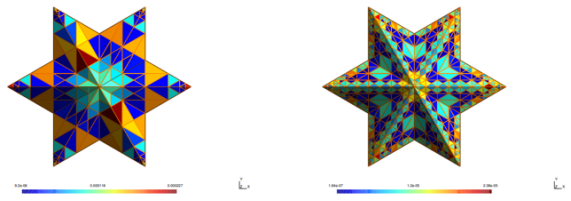


FIGURE 17. Example 3.3: Mesh $\mathcal{T}_\ell$ and distribution of the error estimator η_ℓ^2 for $\ell = 10$ (left) with 360 elements and $\ell = 15$ (right) with 1524 elements.

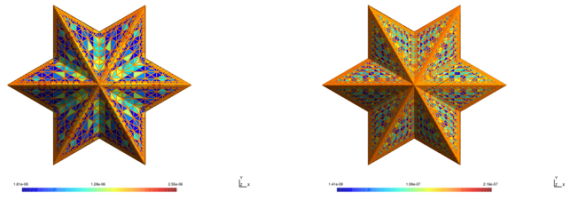


FIGURE 18. Example 3.3: Mesh $\mathcal{T}_\ell$ and distribution of the error estimator η_ℓ^2 for $\ell = 20$ (left) with 5754 elements and $\ell = 25$ (right) with 20304 elements.

Basic Objects

- Grid objects. Managing grids and geometric information.
- Grid functions. Discrete functions defined on grids.
- Boundary Operators. Definition of the various boundary operators
- Discrete boundary operators. Boundary operators after discretisation

The early C++ years of Bempp

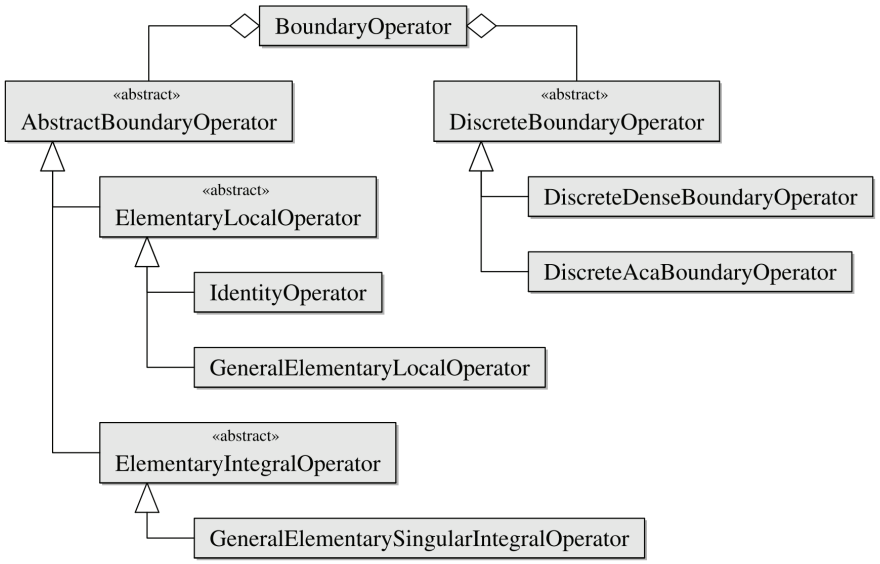


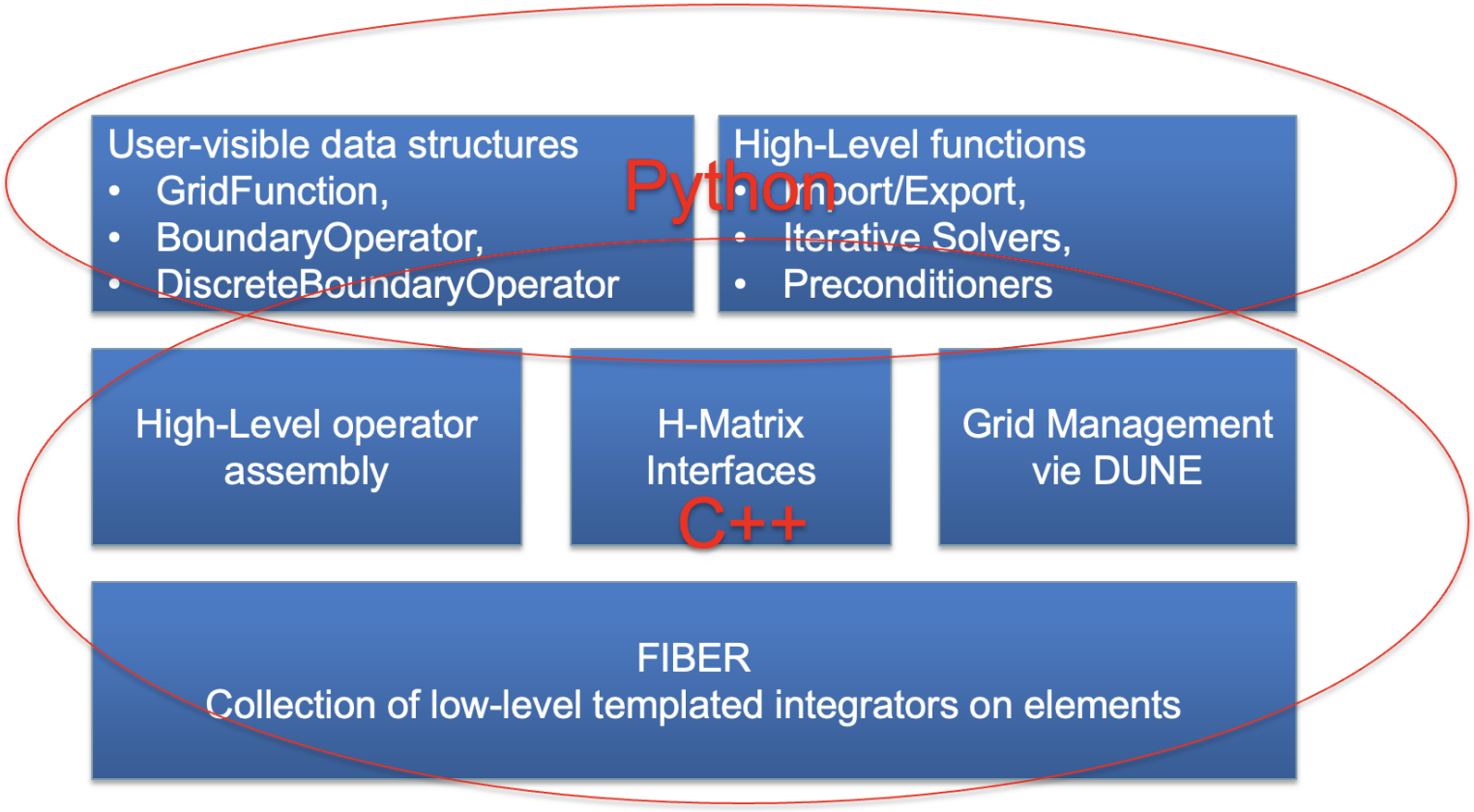
Fig. 3. Relationships between the main classes representing boundary operators.

- 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++

The mixed C++/Python years



Only performance critical logic remained in C++. Python/C++ communication via Cython.

Why was I not happy with it?

- Fewer dependencies but still complicated build process and dependency hell.
- Difficult for PhD students to do low-level code work on the C++ side.
- Very tight object hierarchy on C++ code. Code not well written for modern vector registers (lots of arrays of structs instead of structs of arrays)
- Lots of attempts to shoehorn accelerator support into the code. But it was a mess.
- ...I am never happy with my codes.

> What I wanted: All code in Python, high-performance, SIMD and Accelerator Support, shallow object structure, no dependency hell.

Can we do it in Python?

```
def compute_operator_matrix():
    """Python pseudo-code for BEM evaluation."""
    A = numpy.zeros((n, n), dtype='float64')
    for test_element in test_elements:
        for trial_element in trial_elements:
            if is_adjacent(test_element, trial_element):
                local_interactions = \
                    singular_integration(test_element, trial_element)
            else:
                local_interaction = \
                    regular_integration(test_element, trial_element)
            sum_into_global_matrix(
                A, local_interaction)
    return A
```

- How should we parallelise?
- What about data structures and cache locality?
- Is the interpreter not too slow to handle these loops?

Acceleration technologies

UCL

Numba

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

OpenCL

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

Other technologies

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

Numba acceleration

```python
@_numba.njit
def _project_function_vectorized(
    function_data,
    grid_data,
    support_elements,
    local2global,
    local_multipliers,
    normal_multipliers,
    evaluate_on_element,
    shapeset_evaluate,
    points,
    weights,
    codomain_dimension,
    projections,
    function_parameters,
):
    """Project a Numba callable onto a grid."""

    npoints = points.shape[1]

    for index, element in enumerate(support_elements):

        element_vals = evaluate_on_element(
            element,
            shapeset_evaluate,
            points,
            grid_data,
            local_multipliers,
            normal_multipliers,
        )

        for local_fun_index in range(element_vals.shape[1]):
            projections[local2global[element, local_fun_index]] += (
                _np.sum(
                    _np.sum(
                        element_vals[:, local_fun_index, :]
                        * function_data[:, index * npoints : (1 + index) * npoints]
                        * weights,
                        axis=0,
                    )
                )
                * grid_data.integration_elements[element]
            )
```

- Numba just-in-time compiles Python functions to machine code (using LLVM).
- Executable code independent of the Python interpreter.
- Allows simple loop-based threading.
- Most Python features and many Numpy features supported.

Evaluating the Green's fct. in Numba

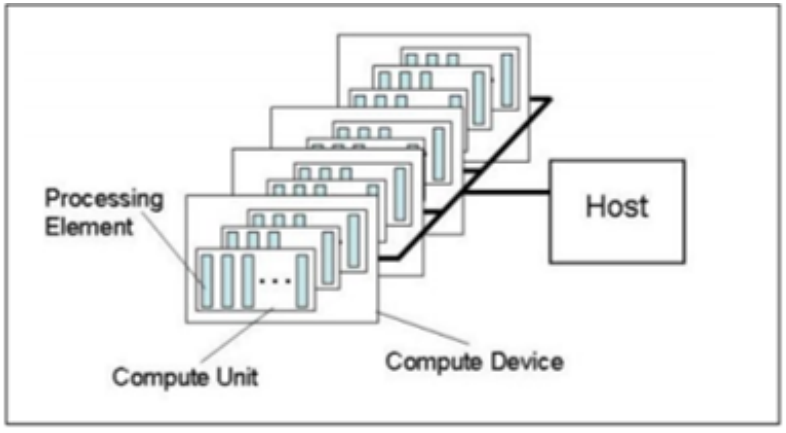
```python
@_numba.jit(
    nopython=True, parallel=False, error_model="numpy", fastmath=True, boundscheck=False
)
def laplace_single_layer_regular(
    test_point, trial_points, test_normal, trial_normals, kernel_parameters
):
    """Laplace single layer for regular kernels."""
    npoints = trial_points.shape[1]
    dtype = trial_points.dtype
    output = _np.zeros(npoints, dtype=dtype)
    m_inv_4pi = dtype.type(M_INV_4PI)
    for i in range(3):
        for j in range(npoints):
            output[j] += (trial_points[i, j] - test_point[i]) ** 2
    for j in range(npoints):
        output[j] = m_inv_4pi / _np.sqrt(output[j])
    return output
```

- Evaluate $g(x, y) = \frac{1}{4\pi|x-y|}$ for one x-value and many values for y.
- Loops written so that the LLVM autovectorizer can produce AVX code.
- Called on the element assembly for each test-quadrature point and evaluated for all trial quadrature points.
- Kernels implemented for all the standard Green's functions and their derivatives for Laplace and Helmholtz.

Global assembly in Numba

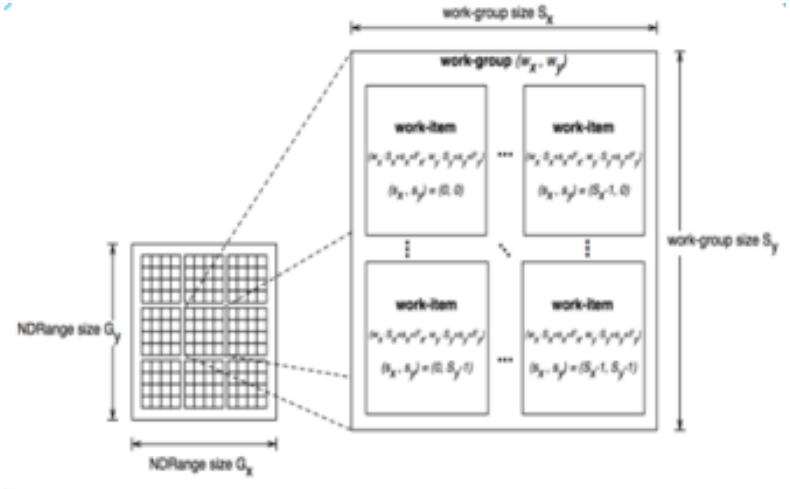
```
def global_assembly():
    """Global assembly of operator"""

    A = empty_array()

    trial_points = get_all_trial_points()
    for test_element in numba.prange(test_elements):
        local_result = zero_array()
        test_points = get_integration_points(test_element)
        for test_point in test_points:
            interactions = evaluate_kernel(test_point, trial_points)
        for trial_element in trial_elements:
            if is_adjacent(test_element, trial_element):
                continue
            sum_into_local_results(
                interactions,
                geometry_data,
                test_fun_values,
                trial_fun_values
            )
        sum_into_global_matrix(
            A, test_element, local_results,
            local2global_data
        )
```

Kernel based programming with OpenCL



- Heterogeneous compute model
- MIMD/SIMD/SIMT support
- C99 based kernel language
- Current version 2.2
- OpenCL drivers by Intel, AMD, Nvidia plus open-source

- Each device consists of compute units (e.g. CPU cores) and processing elements (e.g. SIMD lanes)
- Work items arranged into work groups
- Memory synchronization only inside a workgroup
- Excellent Python support through PyOpenCL by Andreas Kloeckner



Compute kernels for regular integrals

```
__kernel void kernel_function(...){

  size_t testIndex = testIndices[gid[0]];
  size_t trialIndex = trialIndices[gid[1]];

  // Compute geometric data

  // Evaluate all integral contributions for test/trial pair

  // Sum into global matrix if test/trial pair not adjacent

}
```

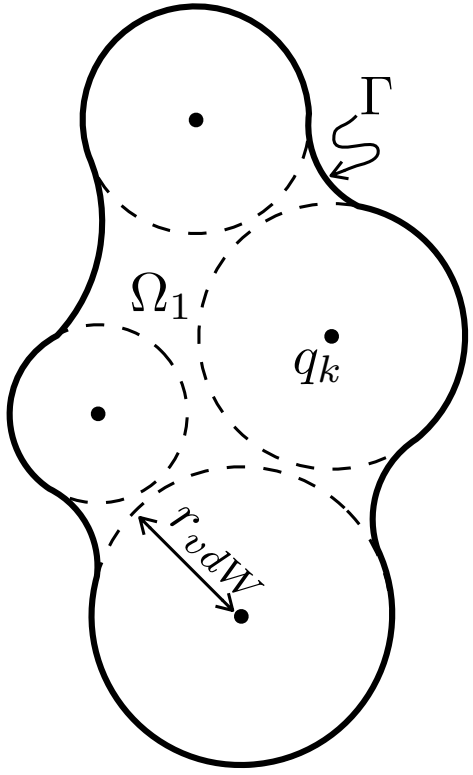
- Compute kernels are JIT-compiled during Python execution by the OpenCL driver
- On CPUs one test triangle coalesced with 4/8/16 trial triangles to efficiently target SIMD lanes.
- A typical computation launches millions of kernels

Poisson-Boltzmann for virus simulations

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.

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

Boundary Integral Formulation

Exterior field formulation[1]

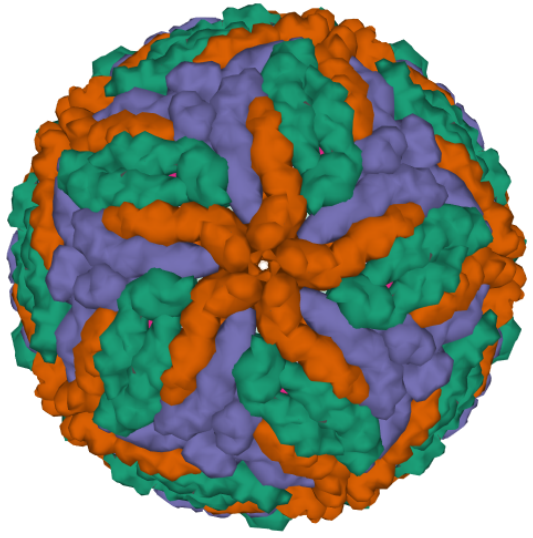
$$\frac{\phi_{2,\Gamma}}{2}\left(\frac{\epsilon_1}{\epsilon_2}+1\right)-\left(K_Y^\Gamma-\frac{\epsilon_1}{\epsilon_2}K_L^\Gamma\right)(\phi_{2,\Gamma})$$
$$+\left(V_Y^\Gamma-V_L^\Gamma\right)\left(\frac{\partial}{\partial\mathbf{n}}\phi_{2,\Gamma}\right)=\sum_{k=0}^{N_q}\frac{q_k}{4\pi\epsilon_2|\mathbf{r}_\Gamma-\mathbf{r}_k|}$$
$$-\frac{\epsilon_1}{\epsilon_2}\left(W_Y^\Gamma-W_L^\Gamma\right)(\phi_{2,\Gamma})+\frac{1}{2}\frac{\phi_{2,\Gamma}}{\partial\mathbf{n}}\left(1+\frac{\epsilon_1}{\epsilon_2}\right)$$
$$+\left(\frac{\epsilon_1}{\epsilon_2}K_Y'^\Gamma-K_L'^\Gamma\right)\left(\frac{\partial}{\partial\mathbf{n}}\phi_{2,\Gamma}\right)=\sum_{k=0}^{N_q}\frac{\partial}{\partial\mathbf{n}_\mathbf{r}}\left(\frac{q_k}{4\pi\epsilon_2|\mathbf{r}_\Gamma-\mathbf{r}_k|}\right)$$

Also possible to derive interior field formulation (due to Juffur) or simple direct coupling from first line of Calderòn projector. Exterior formulation most favourable in terms of conditioning.

[1] Lu et. al., Proc. Natl. Acad. Sci. USA 103 (51) (2006) 19314-19319

Zika virus

- Around 1.6m atoms
- Generated mesh has around 10m surface elements



Use Galerkin BEM code Bempp-cl accelerated with fast particle interactions through Exafmm

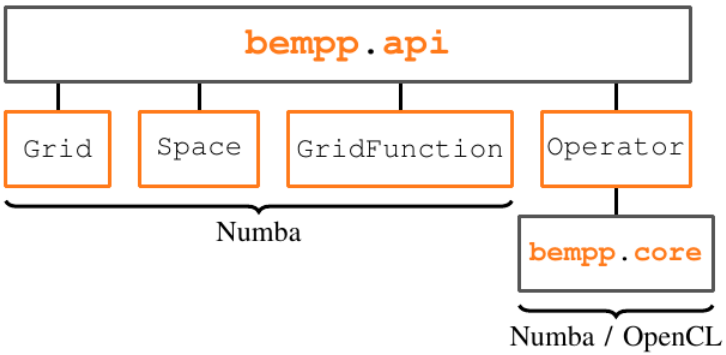
Outline of rest of talk:

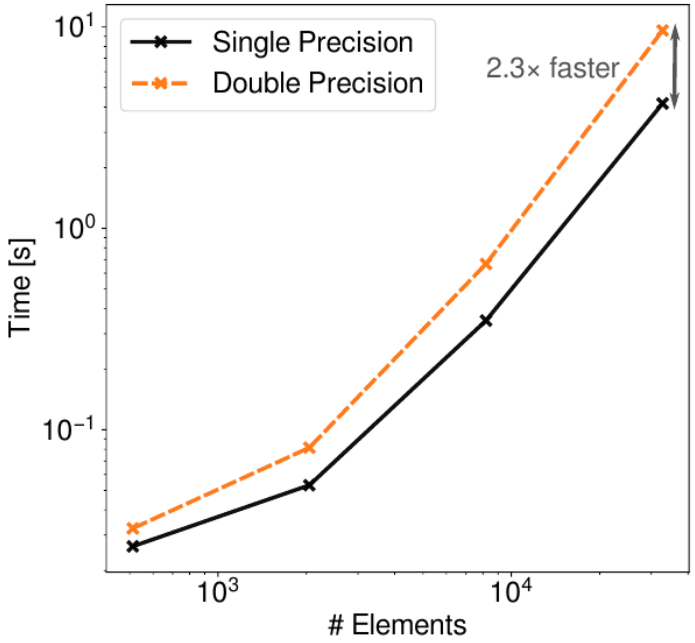
- A short introduction to Bempp-cl and Exafmm
- Black-box coupling strategies for Galerkin BEM
- Practical issues and ongoing work

Bempp-cl

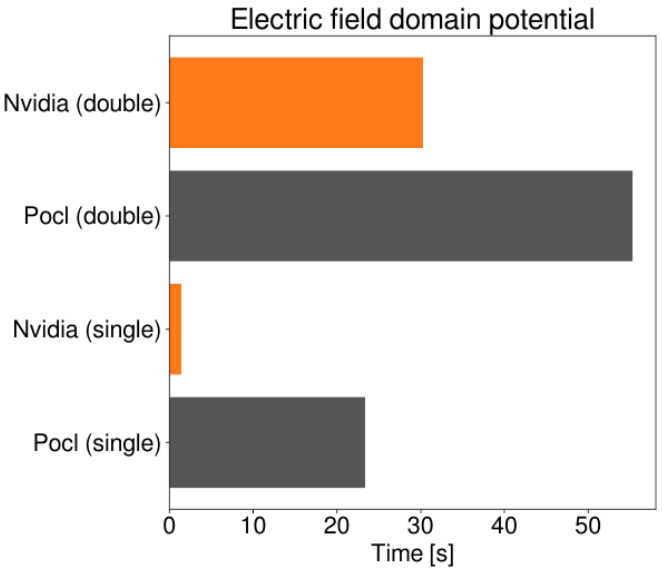
- Successor of the Bempp boundary element library.
- Fully developed using Python with fast OpenCL kernels for computational routines.
- Galerkin discretisation of operators for Laplace, Helmholtz, and Maxwell problems.
- Compute kernels can be run on CPU or offloaded to GPU.
- Core routines optimised for fast dense assembly of opertors.
- Library provides coupling interfaces to Exafmm, can be easily adapted to other fast particle summation libraries.

Characteristics of Bempp-cl



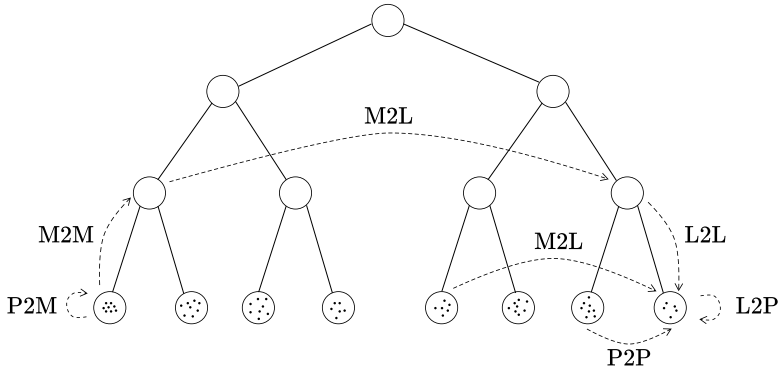


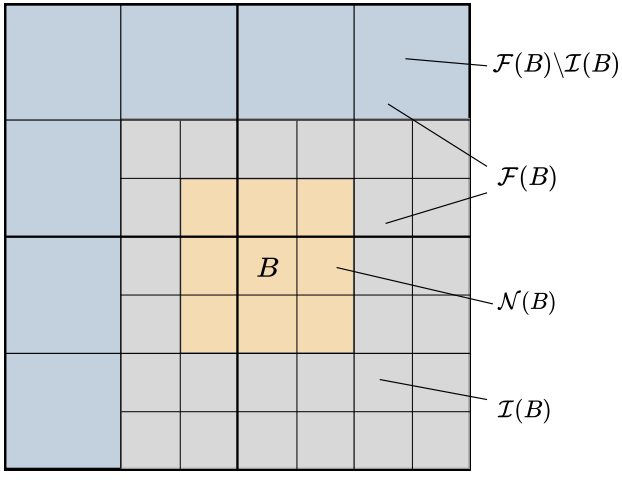
- Top: Structure of Bempp-cl
- Top-Right: AVX acceleration in Bempp-cl
- Bottom-Right: Offloading of a domain potential operator

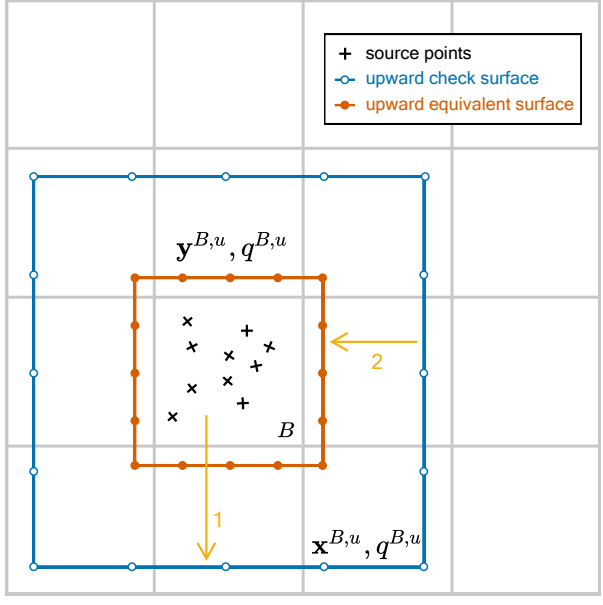


Exafmm-t: High-Performance KIFMM

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







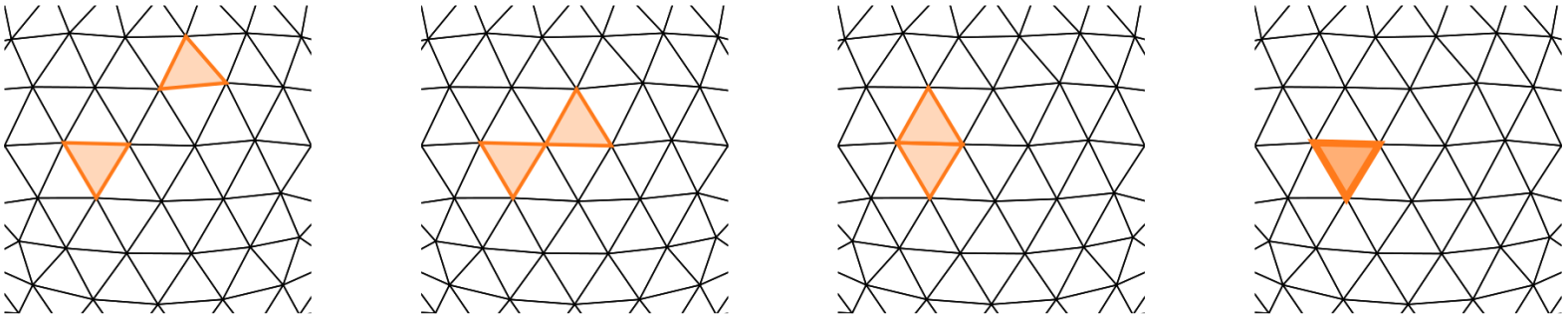
Coupling Bempp-cl and Exafmm-t

Let A be the matrix representation of a Galerkin discretised boundary integral operator. Let x be any vector. Reformulate Ax as

$$Ax = P_{test}^T (G - C) P_{dom} x + Sx$$

- P_{test} and P_{dom} highly sparse matrices that map function space coefficients to particle weights at quadrature points.
- G is black-box operator evaluating particle sums over weights across all quadrature points across all elements.
- C is sparse matrix that contains the particle interactions between neighboring triangles.
- S is sparse matrix containing all singular Galerkin integrals in adjacent elements.

The correction matrix



Problem: Any FMM code 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.

Number of FMM passes per Operator

Operator	V	K	K'	W_Y, W_L
FMM Passes	1	3	1	6, 3

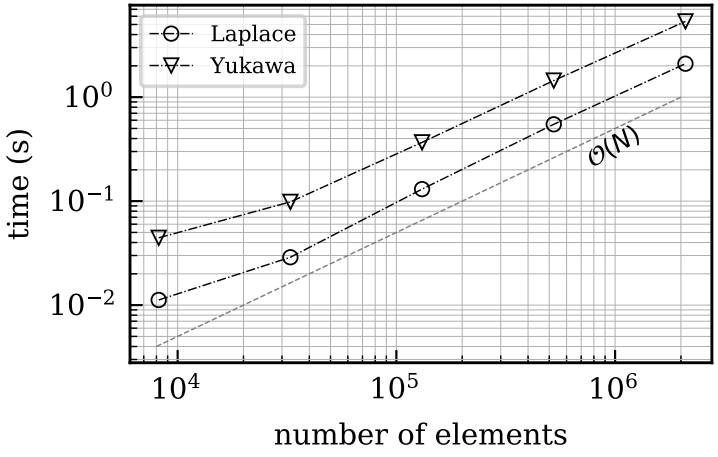
Example: Double-Layer Potential Operator

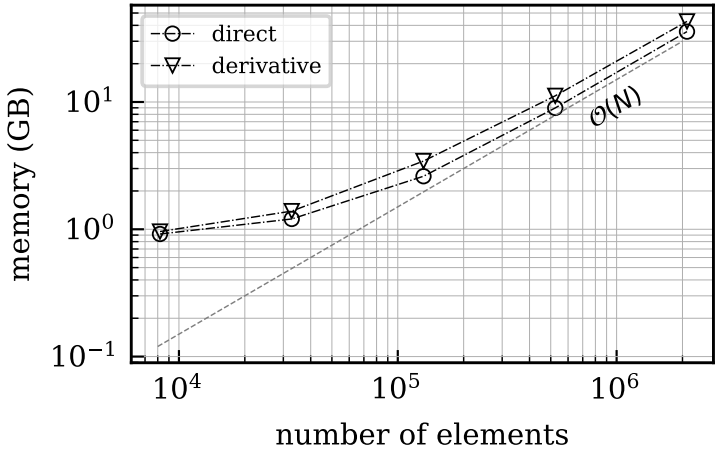
$$[K\phi](\mathbf{x}) = \int_\Gamma \partial_{\mathbf{n}(\mathbf{y})} g(\mathbf{x},\mathbf{y})\phi(\mathbf{y}) ds(\mathbf{y})$$
$$= -\sum_{j=1}^{3} [\nabla_{\mathbf{x}} g(\mathbf{x},\mathbf{y})]_j \, \mathbf{n}_j(\mathbf{y})\phi(\mathbf{y}) ds(\mathbf{y})$$

Require three FMM passes for the three different densities $\tilde{\phi}_j(\mathbf{y}) := \mathbf{n}_j(\mathbf{y})\phi(\mathbf{y})$, $j = 1, \ldots, 3$.

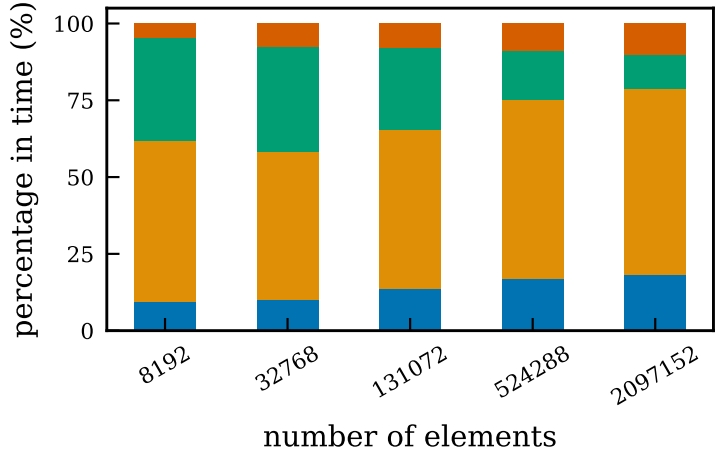
The transmission operator for the exterior Poisson-Boltzmann formulation requires 19 FMM passes.

Poisson-Boltzmann for a sphere

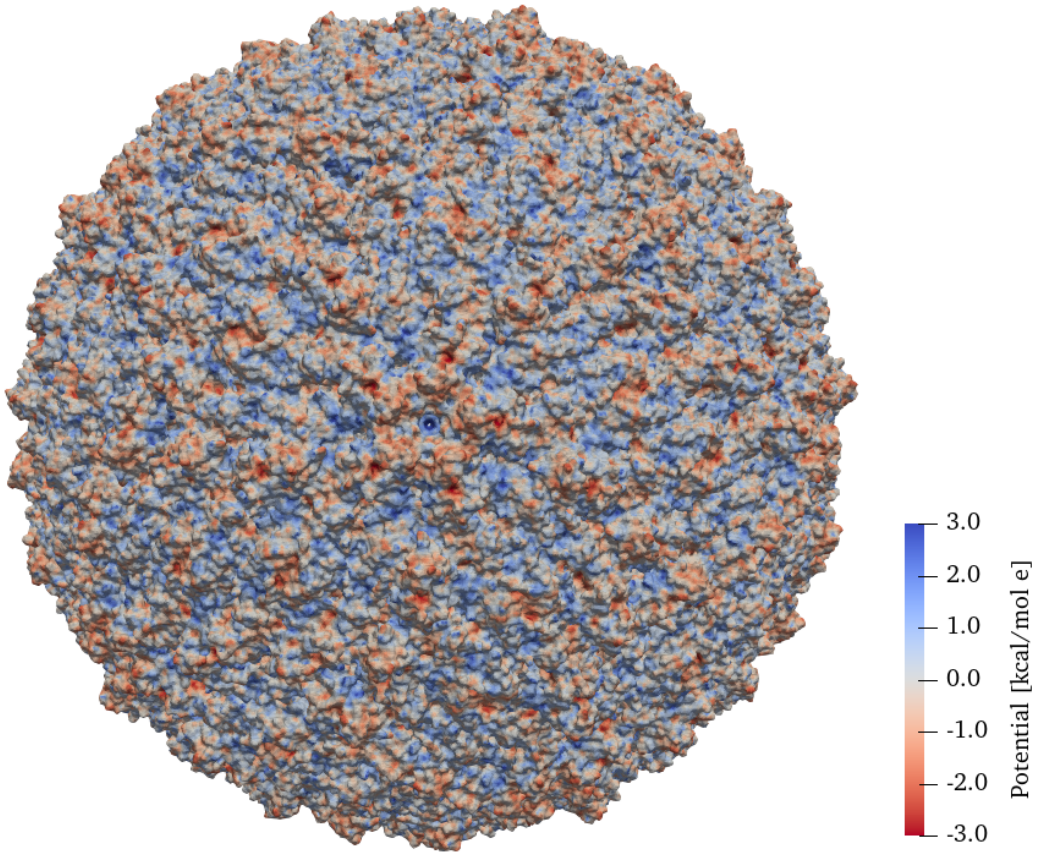




- FMM Expansion Order 5
- 6 regular quadrature points per element
- **Right** Blue: Laplace, Yellow: Yukawa, Green: Singular Correction, Red: Other



Results for Zika Virus



40 Core Compute node. Total time: 139.5 minutes, GMRES time: 80 minutes, 18 GMRES iterations, Max RAM use: 43GB, $\Delta G_{solv} = -116254.9 kcal/mol$. Diagonal preconditioner through inverse of mass matrix with mass lumping.

A FEM/BEM formulation for inhomogeneities

Want to model the interior permittivity as variable. $\rightarrow$ Use FEM for interior/BEM for exterior domain.
(joint work with E. Burman, M. Bosy, C. Cooper)

Find $\phi_1 \in H^1(\Omega_1)$ *and* $\lambda_2 \in H^{-\frac{1}{2}}(\Gamma)$ *such that for all* $v \in H^1(\Omega_1)$ *and* $\zeta \in H^{-\frac{1}{2}}(\Gamma)$

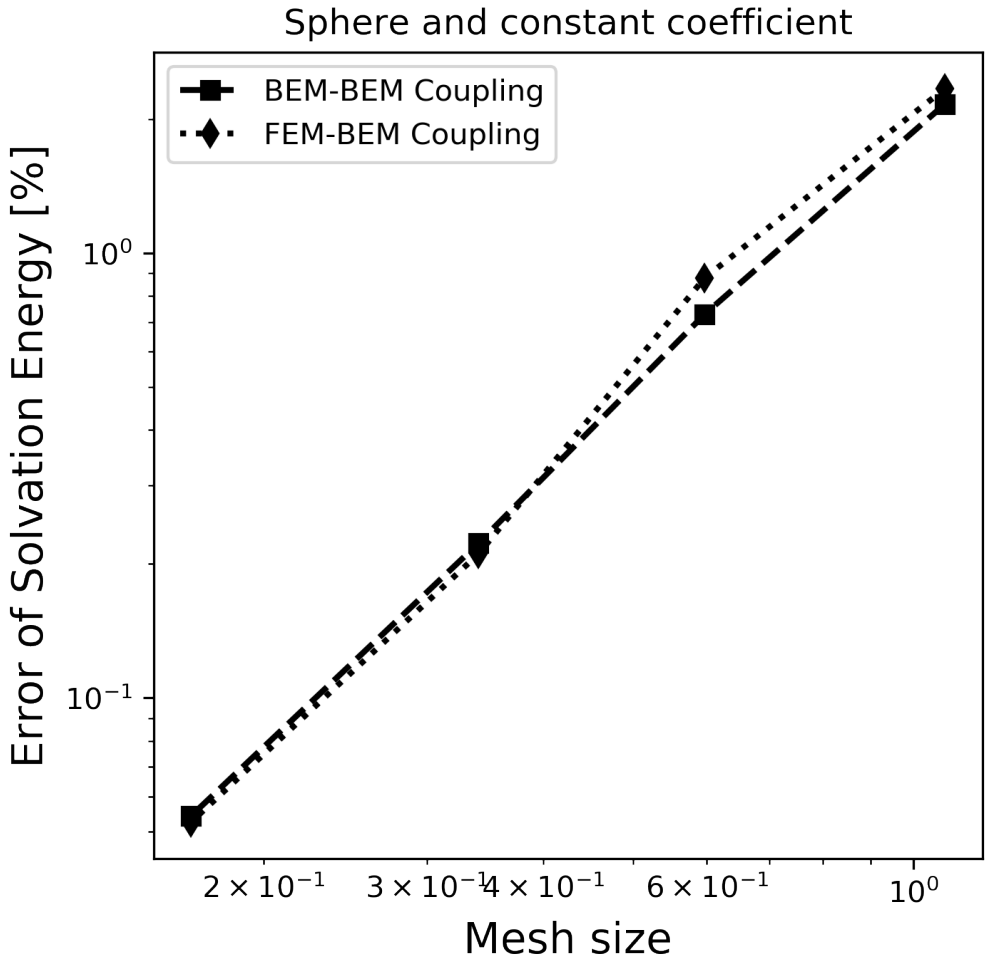
$$\begin{cases} (\epsilon_1 \nabla \phi_1, \nabla v)_{\Omega_1} - \langle \epsilon_1 \lambda_2, v \rangle_\Gamma &= \left(\sum_{k=1}^{N_q} q_k \delta(\mathbf{x}, \mathbf{x}_k), v \right)_{\Omega_1} \\ \left\langle \left(\frac{1}{2} I - K_Y^\Gamma \right) \phi_1, \zeta \right\rangle_\Gamma + \frac{\epsilon_1}{\epsilon_2} \left\langle V_Y^\Gamma \lambda_2, \zeta \right\rangle_\Gamma &= 0. \end{cases}$$

Matrix form:

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

Use FEniCS for interior problem, Bempp-CL for exterior problem.

FEM/BEM vs BEM/BEM for constant permittivity



Error for the Kirkwood sphere.

Comparison with APBS - varying permittivity

	Mesh size Å	ΔG_{solv} kcal/mol
APBS	0.52×0.52×0.52	-32.4652
	0.39×0.39×0.39	-32.4042
	0.26×0.26×0.26	-32.3375
	0.17×0.17×0.17	-32.3413
	Mesh dens. vert/Å^2	
FEM-BEM coupling	2	-35.688
	4	-33.304
	8	-32.634
	16	-31.868

Solvation energy of arginine with a Gaussian-like permittivity, computed using the FEM-BEM approach and APBS. The mesh density for FEM-BEM corresponds to the vertex density of the surface mesh used to generate the volumetric mesh

A Rust based scalable fast solver framework

UCL

Goal: Create a fast solver framework for fast parallel forward evaluation and approximate inversion of integral operators (`https://github.com/rusty-fast-solvers/roadmap`)

All component libraries developed in Rust with Rayon (multithreading) and MPI (across nodes) for parallelization.

Development Status:

- rusty-green-kernel AVX accelerated direct evaluation of Laplace, Helmholtz Green's functions. released.
- rusty-tree Serial and parallel Octree implementations. Serial tree implemented. Parallel tree early stages. close to release.
- rusty-compression Fast randomized compression routines for approximate low-rank matrices. released.
- rusty-translation Implementation of M2M, M2L, L2L, P2M, L2P operators based on numerical field compressions (KIFMM, etc.) early development
- rusty-fmm Serial and parallel FMM loops In planning
- rusty-inverse Evaluation of approximate inverses Not yet started

Summary

- Generic, practically usuable and efficient black-box coupling of Galerkin BEM and Exafmm.
- All computational results shown today steered through interactive Jupyter notebooks.
- Python glue language between the codes. High productivity through Jupyter interface.
- For links to paper, reproducible data and codes see `https://github.com/barbagroup/bempp_exafmm_paper`
- Ongoing development of exascale ready software for EPSRC's new ARCHER2 system (AMD EPYC CPU Cluster)
- Goal: Large-scale coupled Finite element/Boundary element models

Papers

- T. Wang, C. Cooper, T. Betcke, L. Barba, *High-productivity, high-performance workflow for virus-scale electrostatic simulations with Bempp-Exafmm*, arXiv preprint arXiv:2103.01048 (2021), `https://github.com/barbagroup/bempp_exafmm_paper`
- T. Betcke, M. W. Scroggs, *Bempp-cl: A fast Python based just-in-time compiling boundary element library*, Journal of Open Source Software, 6(59), 2879 (2021), `https://doi.org/10.21105/joss.02879`
- T. Betcke, M. W. Scroggs, *Designing a high-performance boundary element library with OpenCL and Numba*, Computing in Science & Engineering 23, pp. 18–28 (2021) `https://doi.org/10.1109/MCSE.2021.3085420`
- 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 (2021), `https://doi.org/10.21105/joss.03145`