The Bempp Handbook

Timo Betcke and Matthew W. Scroggs

August 28, 2020

Welcome to the Bempp Handbook, the main documentation for bempp-cl. Installation instructions and example problems can be found at bempp.com/documentation, as well as documentation for the legacy version of Bempp (3.3.4).

Part I

Introduction

The boundary element method (BEM) is a numerical method for approximating the solution of partial differential equations (PDEs). The method finds the approximation by discretising a boundary integral equation that can be derived from the PDE.

Bempp is an open-source boundary element method library that can be used to assemble all the standard integral kernels for Laplace, Helmholtz, modified Helmholtz, and Maxwell problems. The library has a user-friendly Python interface that allows the user to use BEM to solve a variety of problems, including problems in electrostatics, acoustics and electromagnetics.

Bempp began life as BEM++, and was a Python library with a fast C++ computational core. The ++ slowly changed to a pp as functionality gradually moved from C++ to Python. The latest version, `bempp-cl`, is a complete rewrite of the library, with the C++ core replaced by just-in-time compiled OpenCL kernels, and has many improvements over past versions of the library.

Bempp is divided into two parts: `bempp.api` and `bempp.core`. The user interface of the library is contained in `bempp.api`. The core assembly routines of the library are contained in `bempp.core`. The majority of users of Bempp are unlikely to need to directly interact with the functionality in `bempp.core`.

In this handbook, we introduce and demonstrate the functionality of Bempp. The handbook is split into three parts. First, we present a guide to using Bempp. This part of the handbook takes the reader through all the major functionality contained in the `bempp.api` user interface.

In the second part of the handbook, we take a journey into the Bempp core where we look in more detail at the internal behaviour of the Bempp assemblers

For the final part of the handbook, we present a Bempp user's introduction to boundary element methods. This part of the handbook aims the introduce the mathematics underlying BEM and highlight important aspects of the theory that should be taken into account when deciding how to approach BEM problems.

This handbook is hosted on GitHub and we welcome suggestions for improving it in the issues and pull requests.

Part II

A Guide to Using Bempp

This section of the Bempp Handbook gives details of the functionality of Bempp.

The Bempp library is divided into two parts: `bempp.api` and `bempp.core`. The user-friendly functionality of the library is contained in `bempp.api`, while the fast computation routines are contained in `bempp.core`.

In this section we focus on the functionality in `bempp.api`.

Chapter 1

Grids

In order to solve a problem using the boundary element method, we must first define the grid (or mesh) on which the problem will be discretised. In Bempp, the grid will be a triangulation of a 2D surface in 3D space. Bempp currently only supports grids consisting of flat surface triangles.

This section looks at how grids can be created and used in Bempp.

1.1 Creating a Grid

A Bempp grid can be created using a built-in grid, by importing a gmsh file, or by providing the grid data.

We first import Bempp and NumPy.

```
import bempp.api
import numpy as np
```

1.1.1 Built-in grids

Various shapes are included in the `bempp.api.shapes` module, and discretisations with different numbers of elements can be created using these. In order for the these built-in grids to work, Gmsh must be installed and the command `gmsh` must be available in the path.

The command `regular_sphere` creates a sphere by refining a base octahedron. The number of elements in the sphere is given by 8×4^n, where n is the refinement level. The following command creates a triangulation of a sphere with refinement level 3:

```
grid = bempp.api.shapes.regular_sphere(3)
```

The command `sphere` creates a sphere with a chosen element size. The following command creates a sphere with element diameter $h = 0.1$:

```
grid = bempp.api.shapes.sphere(h=0.1)
```

The command `cube` creates a cube with a chosen element size. The following command creates a cube with element diameter $h = 0.3$:

```
grid = bempp.api.shapes.cube(h=0.3)
```

Full automatically-generated documentation of Bempp's available built-in grids can be found here.

1.1.2 Importing a grid

Grids can be importedusing Bempp's `import_grid` command. For example, the following command will load a grid from the Gmsh file `my_grid.msh`.

```
grid = bempp.api.import_grid('my_grid.msh')
```

Bempp uses the file ending to recognise a number of grid formats. Importing grids is handled by the meshio library.

This works through the external `meshio` library. A list of all files types that can be imported can be found in the meshio documentation. Frequently used formats with Bempp are `.msh` (Gmsh), `.vtk` (Legacy VTK), and `.vtu` (VTK Xml Format).

1.1.3 Creating a grid from element data

Bempp grids can be generated from arrays containing vertex coordinates and connectivity information. For example, to create a grid consisting of two triangles with vertices $\{(0,0,0),(1,0,0),(0,1,0)\}$ and $\{(1,0,0),(1,1,0),(0,1,0)\}$ we use the following commands:

```
vertices = np.array(
    [[0, 1, 0, 1], [0, 0, 1, 1], [0, 0, 0, 0]],
    dtype=np.float64)
elements = np.array(
    [[0, 1], [1, 3], [2, 2]], dtype=np.uint32)
grid = bempp.api.Grid(vertices, elements)
```

Note that the three arrays in the `vertices` array are the x-coordinates, then the y-coordinates, then the z-coordinates. Similarly, the three arrays in the `elements` array are the first points of each triangle, then the second points, then the third points. In general, the `vertices` and `elements` arrays should have shapes $3 \times M$ and $3 \times N$ (respectively) for a grid with M vertices and N elements.

The array `vertices` contains the 3D coordinates of all vertices. The array `elements` contains the connectivity information. In this case the first triangle consists of the vertices 0, 1, 2, and the second triangle consists of the vertices 1, 3, 2.

Optionally, we can specify a list `domain_indices` that gives different groups of elements the same id. This can be used for example to generate different types of boundary data on different parts of the grid, or to specify function spaces only on parts of the grid. In this example, both triangles automatically have the identifier 0 since nothing else was specified. This is equivalent to running:

```
grid = bempp.api.Grid(vertices, elements, domain_indices=[0, 0])
```

1.2 Working with Grids

Once you have created a Bempp grid, you may want to use information about your grid. This page show how commonly used information can be obtained from a Bempp grid.

1.2.1 Querying grid information

The number of elements, edges and vertices in a grid are given by:

```
grid.number_of_elements
grid.number_of_edges
grid.number_of_vertices
```

To query the maximum and minimum element diameter use the following attributes:

```
grid.maximum_element_diameter
grid.minimum_element_diameter
```

The vertex indices of the element 5 can be obtained using:

```
grid.elements[:, 5]
```

Note that the numbering of element starts at 0, so element 5 is the grid's sixth element.

The vertex coordinates of element 5 can be found using:

```
grid.vertices[:, grid.elements[:, 5]]
```

The area of the element 5 is:

```
grid.volumes[5]
```

The edge indices associated with element 5 are:

```
grid.element_edges[5]
```

The vertex indices of the edges of element 5 can be obtained using:

```
grid.edges[:, grid.element_edges[:, 5]]
```

This returns a 2×3 array of the vertex coordinates associated with the three edges of the element. Edges in Bempp are ordered in the following way:

Edge	First vertex	Second vertex
0	0	1
1	0	2
2	1	2

Full automatically-generated documentation of the Bempp `Grid` class can be found here.

1.2.2 Plotting and exporting grids

To export a grid, we can use the `export` command:

```
bempp.api.export('grid.msh', grid=grid)
```

This commands export the object `grid` as Gmsh file with the name `grid.msh`.

In order to plot a grid, we can simply use the command:

```
grid.plot()
```

By default, this will plot using Gmsh (or plotly if you are inside a Jupyter notebook). The following command can be used to change the plotting backend.

```
bempp.api.PLOT_BACKEND = "gmsh"
bempp.api.PLOT_BACKEND = "paraview"
```

This requires Gmsh or Paraview to be available in the system path.

Chapter 2

Function Spaces

Once we have created a grid, we can define finite-dimensional function spaces on the grid. These spaces will then be used to discretise the boundary integral formulations of our problem.

2.1 Defining a Function Space

The function `function_space` is used to initialise spaces. To define a space of piecewise constant functions we use the command:

```
space = bempp.api.function_space(grid, "DP", 0)
```

The parameter `DP` is short for Discontinuous Polynomial. The number 0 is the degree of the polynomial space. To define a space of continuous, piecewise linear functions use:

```
space = bempp.api.function_space(grid, "P", 1)
```

Bempp-cl only supports function spaces up to degree 1. This is an important difference to earlier versions that also supported higher order spaces.

The number of degrees of freedom (DOFs) in a space can be found using:

```
space.global_dof_count
```

For solving Laplace or Helmholtz problems, scalar function spaces should be used. For solving Maxwell's equations, vector function spaces should be used.

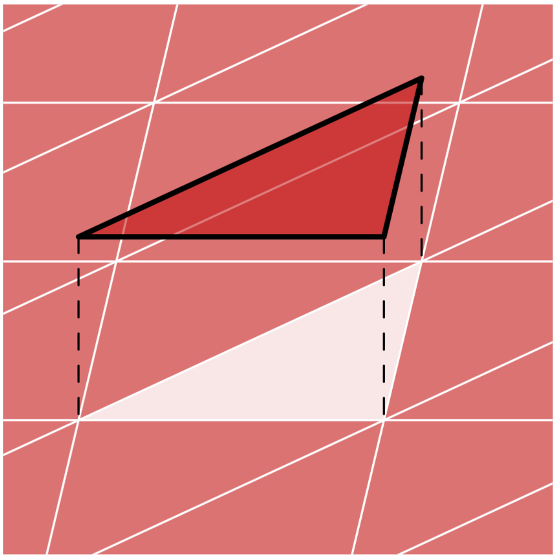
2.2 Scalar Function Spaces

The following scalar-valued spaces are supported in Bempp:

Space Type	Order(s)	Description
`"DP"`	0 or 1	Discontinuous polynomials
`"P"`	1	Continuous polynomials
`"DUAL"`	0 or 1	Dual spaces on the barycentrically refined grid

2.2.1 Discontinuous polynomial spaces

DP spaces are polynomial inside each element and discontinuous between elements. An example basis function of an order 0 DP space is shown below.



Discontinuous polynomial order 0 basis function

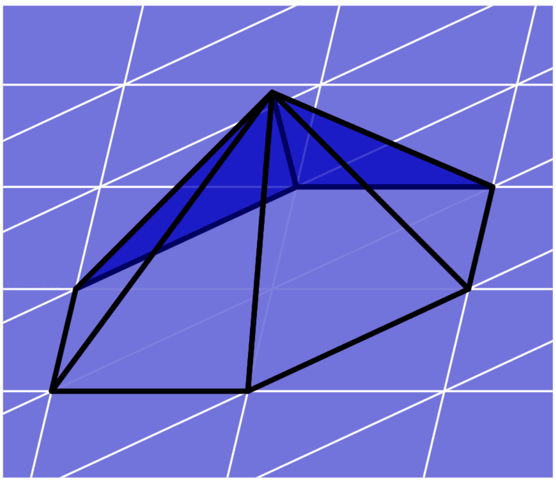
These spaces can be created in Bempp with:

```
space = bempp.api.function_space(grid, "DP", 0)
space = bempp.api.function_space(grid, "DP", 1)
```

The DOFs of an order 0 DP space are at the midpoints of each cell. The DOFs of an order 1 DP space are at the three vertices of each cell.

2.2.2 Continuous polynomial spaces

P spaces are polynomial inside each element and continuous between elements. An example basis function of an order 1 P space is shown below.



Continuous polynomial order 1 basis function

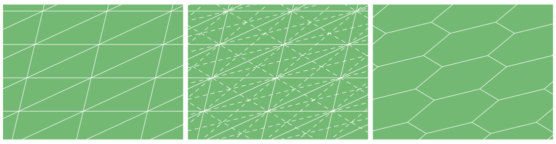
This space can be created in Bempp with:

```
space = bempp.api.function_space(grid, "P", 1)
```

The DOFs of an order 1 P space are at the three vertices of each cell.

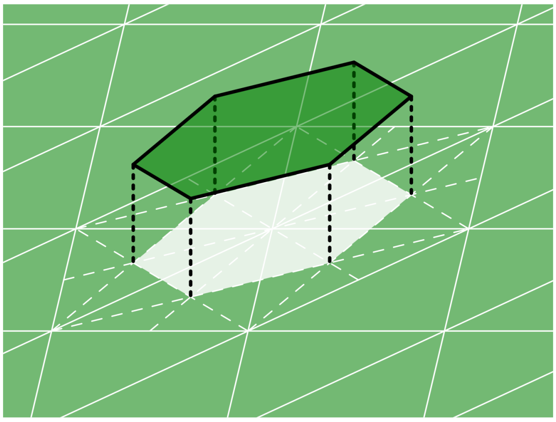
2.2.3 Barycentric dual spaces

To define the barycentric dual space, we first create the barycentrically refined mesh by joining each vertex of every triangle with the centre of the opposite side, as shown below.



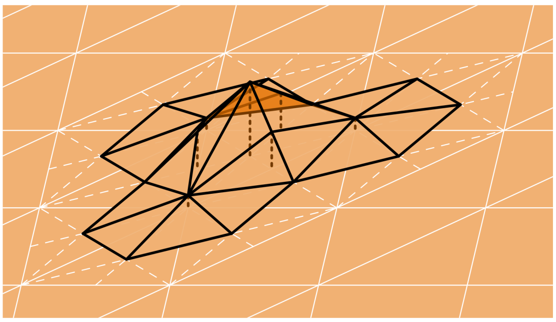
Barycentrically refining a grid

The order 0 dual spaces are piecewise constant functions on the dual cells. An example basis function of an order 0 DUAL space is shown below. Order 0 DUAL spaces form a stable dual pairing with order 1 P spaces.



Dual order 0 basis function

The order 1 dual basis functions are linear combinations of piecewise linear functions on the barycentric cells, and are defined in *A dual finite element complex on the barycentric refinement* (2007) by A. Buffa and S. Christiansen. An example basis function of an order 1 DUAL space is shown below. Order 1 DUAL spaces form a stable dual pairing with order 0 DP spaces.



Dual order 1 basis function

These spaces can be created in Bempp with:

```
space = bempp.api.function_space(grid, "DUAL", 0)
space = bempp.api.function_space(grid, "DUAL", 1)
```

The DOFs of an order 0 DUAL space are at the three vertices of each cell (ie the midpoints of each barycentric dual cell). The DOFs of an order 1 DUAL space are at the mispoints of each cell (ie the vertices of each barycentric dual cell).

2.3 Vector Function Spaces

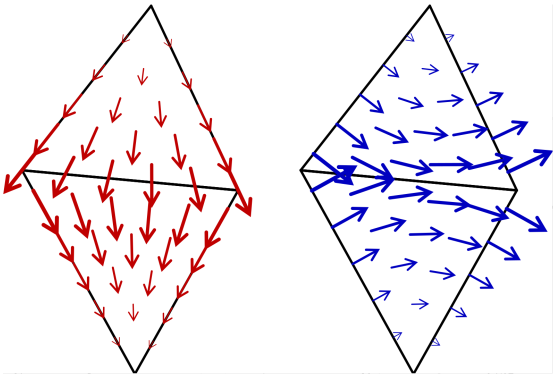
The following vector-valued spaces are supported in Bempp:

Space Type	Order	Description
"RWG"	0	Rao–Wilson–Glisson Hdiv functions
"SNC"	0	Scaled Nédélec Hcurl functions
"BC"	0	Buffa–Christiansen Hdiv functions
"RBC"	0	Rotated Buffa–Christiansen Hcurl functions

When solving Maxwell's equations, the correct combination of Hdiv and Hcurl spaces must be used.

2.3.1 RWG and SNC spaces

RWG and SNC spaces are vector-valued spaces, whose values are tangential to the surface triangles. Inside each cell, these spaces are linear combinations of the vectors $\begin{pmatrix}1\\0\end{pmatrix}$, $\begin{pmatrix}0\\1\end{pmatrix}$, and $\begin{pmatrix}-y\\x\end{pmatrix}$. Between cells, RWG functions are continuous normal to the triangle's edges, while SNC spaces are continuous tangential to the triangle's edges. Example RWG (left) and SNC (right) basis functions are shown below.



An RWG and a SNC basis function

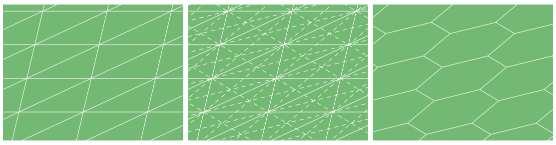
These spaces can be created in Bempp with:

```
rwg_space = bempp.api.function_space(grid, "RWG", 0)
snc_space = bempp.api.function_space(grid, "SNC", 0)
```

The DOFs of RWG and SNC spaces are at the midpoints of the edges of each cell.

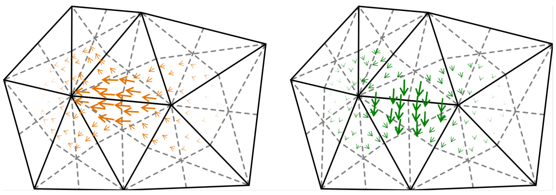
2.3.2 Barycentric dual spaces

Like the scalar DUAL spaces, BC and RBC spaces are defined on the barycentrically refined grid. This grid is formed by joining each vertex of every triangle with the centre of the opposite side, as shown below.



Barycentrically refining a grid

BC and RBC spaces are combinations of RWG and SNC (repectively) spaces on the barycentric grid, and are defined in *A dual finite element complex on the barycentric refinement* (2007) by A. Buffa and S. Christiansen. Example BC (left) and RBC (right) basis functions are shown below.



Dual order 0 basis function

These spaces can be created in Bempp with:

```
bc_space = bempp.api.function_space(grid, "BC", 0)
rbc_space = bempp.api.function_space(grid, "RBC", 0)
```

The DOFs of BC and RBC spaces are at the midpoints of the edges of each cell (or equivalently at a point on the edges of the barycentric dual cells).

2.4 Function Spaces on Segments

Bempp can create spaces on segments of a grid as well as on the entire grid. In order to do this, domain indices must be provided when creating the grid. Some built-in grids have domain indices: for example, the cube has six segments for the faces (numbered 1 to 6).

To create a space on a segment, first create the grid:

```
grid = bempp.api.shapes.cube()
```

A space on segments 1 and 2 of the grid can then be created using:

```
space = bempp.api.function_space(grid, "DP", 0, segments=[1, 2])
```

2.4.1 Controlling segment spaces

There are two options that can be used to control the behaviour of a space on a segment: `include_boundary_dofs` and `truncate_at_segment_edge`.

If `include_boundary_dofs` is set to `True`, DOF points on vertices and edges on the boundary. By default, this is set to `False`. Setting this option to `True` is likely to make the function space extend outside its segments, as part of the basis functions with DOFs on the boundary will be outside the domain.

The option `truncate_at_segment_edge` can be used to truncate basis functions at the edge of the segment, whenever a basis function will extend outside the segments. By default, this is `False`. Setting this option to `True` will cause the function space to be discontinuous at the edge of the segment.

Chapter 3

Grid functions

In Bempp, data on a given grid is represented as a grid function object. A grid function consists of a set of basis function coefficients and a corresponding function space.

3.1 Initialising with a Python callable

Grid functions can be created from Python callables.

```
@bempp.api.complex_callable
def fun(x, normal, domain_index, result):
    result[0] = np.exp(1j * x[0])
```

The first argument `x` is the coordinates of an evaluation point. The second argument `normal` is the normal direction at the evaluation point. The third one is the `domain_index`: this corresponds to the physical id in Gmsh and can be used to assign different boundary data to different parts of the grid. The last argument `result` is the variable that stores the value of the callable. It is a Numpy array with as many components as the basis functions of the underlying space have.

A Python callable that you want to use to build a grid function should always have these four inputs. An optional fifth input may be used to pass parameters into the function.

In order for Bempp to assemble a grid function with coefficients of the correct data type, these callables must be decorated with either `@bempp.api.real_callable` or `@bempp.api.real_callable`.

The projection of this callable into a Bempp space can be created with:

```
grid_fun = bempp.api.GridFunction(space, fun=fun)
```

3.1.1 Disabling just-in-time compilation

By default, Bempp with use Numba just-in-time compilation when creating grid functions from callables. In some cases, this compilation is not possible and should be disabled. This can be done by passing `jit=False` into the decorator:

```
@bempp.api.complex_callable(jit=False)
def fun(x, normal, domain_index, result):
    result[0] = np.exp(1j * x[0])

grid_fun = bempp.api.GridFunction(space, fun=fun)
```

The construction of this grid function will be slower as it is not sped up by Numba.

3.2 Initialising with coefficients or projections

Instead of a callable, we can initialise a grid function from a vector of coefficients or a vector of projections. This can be done as follows.

```
c = np.array([...])  # These are the coefficients
grid_fun = GridFunction(space, coefficients=c)

p = np.array([...])  # These are the projections
grid_fun = GridFunction(space, projections=p, dual_space=dual)
```

The argument `dual_space` gives the space with which the projection coefficients were computed. The parameter is optional and if it is not given then `space == dual_space` is assumed.

3.3 Coefficients and projections

The functions in a discrete function space are represented as a linear combination of some basis functions. The coefficients of a grid function are the scalars which each basis function is multiplied by in this combination. The coefficients of a grid function can be obtained using:

```
grid_fun.coefficients
```

The projections of a grid function are calculated by applying a discrete mass matrix to the coefficients. The mass matrix will be between the grid function's space and a dual space provided to the `projections` call. These can be obtained using:

```
grid_fun.projections(dual_space)
```

In some situations, for example when the space and the dual are RWG and SNC spaces, the mass matrix for projections may be numerically singular. In these cases, the coefficients of a grid function that has been initialised using projections cannot be accurately calculated. Bempp, however, can still use these grid functions by only querying the projections.

3.4 Plotting and exporting grid functions

To export a grid function, we can use the `export` command:

```
bempp.api.export('grid_function.msh', grid_function=grid_fun)
```

This commands export the object `grid_fun` as Gmsh file with the name `grid_function.msh`.

In order to plot a grid function, we can simply use the command:

```
grid_fun.plot()
```

By default, this will plot using Gmsh (or plotly if you are inside a Jupyter notebook). The following command can be used to change the plotting backend.

```
bempp.api.PLOT_BACKEND = "gmsh"
bempp.api.PLOT_BACKEND = "paraview"
```

This requires Gmsh or Paraview to be available in the system path.

Chapter 4

Boundary Operators

Boundary integral formulations of problems are commonly written using boundary integral operators. In this section of the Bempp Handbook, we look at how these operators can be defined and assembled using Bempp.

4.1 Domains, ranges, and duals

When creating an operator in Bempp, three spaces are provided: the domain, the range, and the dual to the range (given as inputs in that order). The domain and dual spaces are used to calculate the weak form of the operator. The range is used by the operator algebra to correctly assemble product of operators.

4.2 Sparse Boundary Operators

Discretising the identity operator leads to a matrix $M = (m_{ij})$, defined by

$$m_{ij} = \int_\Gamma \phi_j \cdot \overline{\psi_i},$$

where ϕ_j and ψ_i are the basis functions of the domain and dual spaces respectively. As this integral will only be non-zero when the basis functions overlap, the resulting matrix will be sparse.

The identity operator can be created in Bempp using:

```
ident = bempp.api.operators.boundary.sparse(domain, range_, dual)
```

A `SparseDiscreteBoundaryOperator` can be obtained using:

```
mat = ident.weak_form()
```

This matrix is commonly called the mass matrix between the domain and dual spaces.

If desiried, a SciPy CSR matrix can be obtained from this discrete boundary operator with:

```
mat.A
```

4.3 Boundary Operators for Laplace's Equation

For Laplace's equation, there are four boundary operators that are used, as given in the table below.

Operator	Symbol	Matrix entries
Single layer	V	$m_{ij} = \int_\Gamma \int_\Gamma G(\mathbf{x},\mathbf{y})\phi_j(\mathbf{y})\psi_i(\mathbf{x})\,\mathrm{d}\mathbf{y}\,\mathrm{d}\mathbf{x}$
Double layer	K	$m_{ij} = \int_\Gamma \int_\Gamma \frac{\partial G(\mathbf{x},\mathbf{y})}{\partial \nu_{\mathbf{y}}}\phi_j(\mathbf{y})\psi_i(\mathbf{x})\,\mathrm{d}\mathbf{y}\,\mathrm{d}\mathbf{x}$
Adjoint double layer	K'	$m_{ij} = \int_\Gamma \int_\Gamma \frac{\partial G(\mathbf{x},\mathbf{y})}{\partial \nu_{\mathbf{x}}}\phi_j(\mathbf{y})\psi_i(\mathbf{x})\,\mathrm{d}\mathbf{y}\,\mathrm{d}\mathbf{x}$
Hypersingular	W	$m_{ij} = \int_\Gamma \int_\Gamma \frac{\partial^2 G(\mathbf{x},\mathbf{y})}{\partial \nu_{\mathbf{y}}\partial \nu_{\mathbf{x}}}\phi_j(\mathbf{y})\psi_i(\mathbf{x})\,\mathrm{d}\mathbf{y}\,\mathrm{d}\mathbf{x}$

In each case, ϕ_j and ψ_i are the basis functions of the domain and dual spaces (respectively), and $G(\mathbf{x},\mathbf{y})$ is the Green's function for Laplace's equation. The Green's function will have a singularity when $\mathbf{x}=\mathbf{y}$, so internally Bempp will use appropriate singular quadrature rules to handle this.

These operators can be initialised in Bempp using:

```
from bempp.api.operators.boundary import laplace
single = laplace.single_layer(domain, range_, dual)
double = laplace.double_layer(domain, range_, dual)
adjoint_d = laplace.adjoint_double_layer(domain, range_, dual)
hypersingular = laplace.hypersingular(domain, range_, dual)
```

The spaces passed into each operator should be appropriately chosen scalar function spaces.

A keyword argument `assembler` may be passed into each constructor to change the assembler used to assemble the operator. For example, the single layer operator will be discretised using the fast multipole method (FMM) if it is initialised with:

```
single = laplace.single_layer(domain, range_, dual, assembler="fmm"
                              )
```

When using dense assembly, the keyword argument `device_interface` can be used to switch between assembly using OpenCL and Numba:

```
single = laplace.single_layer(
    domain, range_, dual, wavenumber, assembler="dense",
    device_interface="numba"
    )
single = laplace.single_layer(
    domain, range_, dual, wavenumber, assembler="dense",
    device_interface="opencl"
    )
```

The matrix discretisation of an operator can be obtained using, for example:

```
single.weak_form()
```

The strong form discretisation of an operator can be obtained using:

```
single.strong_form()
```

The interpretation of the strong form is discussed in the operator algebra section.

4.4 Boundary Operators for the Helmholtz Equation

For the Helmholtz equation, there are four boundary operators that are used, as given in the table below.

Operator	Symbol	Matrix entries
Single layer	V	$m_{ij} = \int_\Gamma \int_\Gamma G_k(\mathbf{x}, \mathbf{y}) \phi_j(\mathbf{y}) \psi_i(\mathbf{x}) \,\mathrm{d}\mathbf{y} \,\mathrm{d}\mathbf{x}$
Double layer	K	$m_{ij} = \int_\Gamma \int_\Gamma \frac{\partial G_k(\mathbf{x}, \mathbf{y})}{\partial \nu_{\mathbf{y}}} \phi_j(\mathbf{y}) \psi_i(\mathbf{x}) \,\mathrm{d}\mathbf{y} \,\mathrm{d}\mathbf{x}$
Adjoint double layer	K′	$m_{ij} = \int_\Gamma \int_\Gamma \frac{\partial G_k(\mathbf{x}, \mathbf{y})}{\partial \nu_{\mathbf{x}}} \phi_j(\mathbf{y}) \psi_i(\mathbf{x}) \,\mathrm{d}\mathbf{y} \,\mathrm{d}\mathbf{x}$
Hypersingular	W	$m_{ij} = \int_\Gamma \int_\Gamma \frac{\partial^2 G_k(\mathbf{x}, \mathbf{y})}{\partial \nu_{\mathbf{y}} \partial \nu_{\mathbf{x}}} \phi_j(\mathbf{y}) \psi_i(\mathbf{x}) \,\mathrm{d}\mathbf{y} \,\mathrm{d}\mathbf{x}$

In each case, ϕ_j and ψ_i are the basis functions of the domain and dual spaces (respectively), and $G_k(\mathbf{x}, \mathbf{y})$ is the Green's function for the Helmholtz equation with wavenumber k. The Green's function will have a singularity when $\mathbf{x} = \mathbf{y}$, so internally Bempp will use appropriate singular quadrature rules to handle this.

These operators can be initialised in Bempp using:

```
from bempp.api.operators.boundary import helmholtz
single = helmholtz.single_layer(domain, range_, dual, wavenumber)
double = helmholtz.double_layer(domain, range_, dual, wavenumber)
adjoint_d = helmholtz.adjoint_double_layer(domain, range_, dual,
                                           wavenumber)
hypersingular = helmholtz.hypersingular(domain, range_, dual,
                                         wavenumber)
```

The spaces passed into each operator should be appropriately chosen scalar function spaces.

A keyword argument `assembler` may be passed into each constructor to change the assembler used to assemble the operator. For example, the single layer operator will be discretised using the fast multipole method (FMM) if it is initialised with:

```
single = helmholtz.single_layer(
    domain, range_, dual, wavenumber, assembler="fmm")
```

When using dense assembly, the keyword argument `device_interface` can be used to switch between assembly using OpenCL and Numba:

```
single = helmholtz.single_layer(
    domain, range_, dual, wavenumber, assembler="dense",
    device_interface="numba"
    )
single = helmholtz.single_layer(
    domain, range_, dual, wavenumber, assembler="dense",
    device_interface="opencl"
    )
```

The matrix discretisation of an operator can be obtained using, for example:

```
single.weak_form()
```

The strong form discretisation of an operator can be obtained using:

```
single.strong_form()
```

The interpretation of the strong form is discussed in the operator algebra section.

4.5 Boundary Operators for the Modified Helmholtz Equation

For the modified Helmholtz equation, there are four boundary operators that are used, as given in the table below.

Operator	Symbol	Matrix entries
Single layer	V	$m_{ij} = \int_\Gamma \int_\Gamma G_\omega(\mathbf{x},\mathbf{y})\phi_j(\mathbf{y})\psi_i(\mathbf{x})\,\mathrm{d}\mathbf{y}\,\mathrm{d}\mathbf{x}$
Double layer	K	$m_{ij} = \int_\Gamma \int_\Gamma \frac{\partial G_\omega(\mathbf{x},\mathbf{y})}{\partial \nu_\mathbf{y}}\phi_j(\mathbf{y})\psi_i(\mathbf{x})\,\mathrm{d}\mathbf{y}\,\mathrm{d}\mathbf{x}$
Adjoint double layer	K'	$m_{ij} = \int_\Gamma \int_\Gamma \frac{\partial G_\omega(\mathbf{x},\mathbf{y})}{\partial \nu_\mathbf{x}}\phi_j(\mathbf{y})\psi_i(\mathbf{x})\,\mathrm{d}\mathbf{y}\,\mathrm{d}\mathbf{x}$
Hypersingular	W	$m_{ij} = \int_\Gamma \int_\Gamma \frac{\partial^2 G_\omega(\mathbf{x},\mathbf{y})}{\partial \nu_\mathbf{y}\partial \nu_\mathbf{x}}\phi_j(\mathbf{y})\psi_i(\mathbf{x})\,\mathrm{d}\mathbf{y}\,\mathrm{d}\mathbf{x}$

In each case, ϕ_j and ψ_i are the basis functions of the domain and dual spaces (respectively), and $G_\omega(\mathbf{x},\mathbf{y})$ is the Green's function for the modified Helmholtz equation with frequency ω. The Green's function will have a singularity when $\mathbf{x} = \mathbf{y}$, so internally Bempp will use appropriate singular quadrature rules to handle this.

These operators can be initialised in Bempp using:

```
from bempp.api.operators.boundary import modified_helmholtz
single = modified_helmholtz.single_layer(domain, range_, dual,
                                          omega)
double = modified_helmholtz.double_layer(domain, range_, dual,
                                          omega)
adjoint_d = modified_helmholtz.adjoint_double_layer(domain, range_,
                                          dual, omega)
hypersingular = modified_helmholtz.hypersingular(domain, range_,
                                          dual, omega)
```

The spaces passed into each operator should be appropriately chosen scalar function spaces.

A keyword argument `assembler` may be passed into each constructor to change the assembler used to assemble the operator. For example, the single layer operator will be discretised using the fast multipole method (FMM) if it is initialised with:

```
single = modified_helmholtz.single_layer(
    domain, range_, dual, omega, assembler="fmm")
```

When using dense assembly, the keyword argument `device_interface` can be used to switch between assembly using OpenCL and Numba:

```
single = modified_helmholtz.single_layer(
    domain, range_, dual, wavenumber, assembler="dense",
    device_interface="numba"
```

```
    )
single = modified_helmholtz.single_layer(
    domain, range_, dual, wavenumber, assembler="dense",
    device_interface="opencl"
    )
```

The matrix discretisation of an operator can be obtained using, for example:

```
single.weak_form()
```

The strong form discretisation of an operator can be obtained using:

```
single.strong_form()
```

The interpretation of the strong form is discussed in the operator algebra section.

4.6 Boundary Operators for Maxwell's Equations

For Maxwell's equations, there are two boundary operators that are used, as given in the table below.

Operator	Symbol	Matrix entries
Electric field	E	$m_{ij} = -\mathrm{i}k \int_\Gamma \int_\Gamma G_k(\mathbf{x},\mathbf{y})\phi_j(\mathbf{y}) \cdot \psi_i(\mathbf{x}) \,\mathrm{d}\mathbf{y}\,\mathrm{d}\mathbf{x} - \frac{1}{\mathrm{i}k} \int_\Gamma \int_\Gamma G_k(\mathrm{x},\mathrm{y})\nabla_\Gamma\phi_j(\mathbf{y})\nabla_\Gamma\psi_i(\mathbf{x}) \,\mathrm{d}\mathbf{y}\,\mathrm{d}$
Magnetic field	H	$m_{ij} = -\int_\Gamma \int_\Gamma \nabla_\mathbf{x} G_k(\mathbf{x},\mathbf{y}) \cdot (\psi_j(\mathbf{y}) \times \psi_i(\mathbf{x})) \,\mathrm{d}\mathbf{y}\,\mathrm{d}\mathbf{x})$

In each case, ϕ_j and ψ_i are the basis functions of the domain and dual spaces (respectively), and $G_k(\mathbf{x},\mathbf{y})$ is the Green's function for the Helmholtz equation with wavenumber k. The Green's function will have a singularity when $\mathbf{x} = \mathbf{y}$, so internally Bempp will use appropriate singular quadrature rules to handle this.

These operators can be initialised in Bempp using:

```
from bempp.api.operators.boundary import maxwell
electric = maxwell.electric_field(domain, range_, dual, wavenumber)
magnetic = maxwell.magnetic_field(domain, range_, dual, wavenumber)
```

The spaces passed into each operator should be appropriately chosen vector function spaces: the domain and range spaces should both be Hdiv spaces, while the dual space should be a Hcurl space.

A keyword argument `assembler` may be passed into each constructor to change the assembler used to assemble the operator. For example, the electric field operator will be discretised using the fast multipole method (FMM) if it is initialised with:

```
electric = maxwell.electric_field(
    domain, range_, dual, wavenumber, assembler="fmm")
```

When using dense assembly, the keyword argument `device_interface` can be used to switch between assembly using OpenCL and Numba:

```
electric = maxwell.electric_field(
    domain, range_, dual, wavenumber, assembler="dense",
    device_interface="numba"
```

```
    )
electric = maxwell.electric_field(
    domain, range_, dual, wavenumber, assembler="dense",
    device_interface="opencl"
    )
```

The matrix discretisation of an operator can be obtained using, for example:

```
electric.weak_form()
```

The strong form discretisation of an operator can be obtained using:

```
electric.strong_form()
```

The interpretation of the strong form is discussed in the operator algebra section. For Maxwell's equations, care must be taken to use space for the range and dual spaces that form a stable dual pairing (see the vector function spaces section) in order to be able to correctly obtain the strong form of an operator.

4.7 Operator Algebra

In many boundary element method applications, a discretisation of the product of two operators is required.

Let A and B be two operators with discretisations A_h and B_h. A discretisation of the product AB is given by $A_h M^{-1} B_h$, where M is the mass matrix between the range and dual of the operator B.

In Bempp, the discrete product of two operators can be formed using:

```
op1 = bempp.api.operators.boundary...
op2 = bempp.api.operators.boundary...
product = op1 * op2
```

If `product.weak_form()` is called, Bempp will internally use its knowledge of the range space of `op2` to correctly from the discretisation of this product.

Calling the `strong_form` of an operator B will return the product $M^{-1}B_h$. Calling `product.weak_form()` is equivalent to calculating `op1.weak_form()* op2.strong_form()`. Using the strong form of operators can in general be useful, as the discretisation obtained corresponds to a mass matrix preconditioned version of the relevant formulation.

Chapter 5

Linear Solvers

Once you have assembled the relevant operators, and have created a grid function containing the relevant right-hand-side data, you will need to solve your linear system.

5.1 Direct Solvers

Direct solvers compute the solution of a linear system by (usually indirectly) computing the inverse of the matrix.

SciPy's direct LU solver is wrapped in the function `bempp.api.linalg.lu`. This can be used with:

```
solution = bempp.api.linalg.lu(operator, grid_fun)
```

Direct solvers should only be used if the operator has been assembled in dense mode.

5.2 Iterative Solvers

Iterative solvers solve a linear system iteratively: steps are repeated to achieve better approximations of the solution. For well-condtioned matrices, iterative solvers can achieve fast convergence, so very good approximations of the solution can be achieved in just a few iterations.

SciPy's CG and GMRes iterative solvers are wrapped in the `bempp.api.linalg` submodule. These can be used with:

```
solution, info = bempp.api.linalg.cg(operator, grid_fun)
solution, info = bempp.api.linalg.gmres(opreator, grid_fun)
```

These solvers take a number of optional arguments:

Argument	Description
`tol`	The tolerance the solver should aim for
`maxiter`	The maximum number of iterations
`use_strong_form`	If `True`, the strong form of the operator will be used. If `False`, the weak form is us
`return_residuals`	If `True` the residuals will be returned as well as the solution and info
`return_iteration_count`	If `True` the iteration count will be returned as well as the solution and info

By default, Bempp will use the weak form discretisation of the operator and the coefficients of the grid function when using an iterative solver. If `use_strong_form` is set to `True`, Bempp will use the strong form discretisation of the operator and the projections of the grid function onto the range of the operator. This is equivalent to applying a mass matrix preconditioner to the problem and often leads to a lower iteration count.

Chapter 6

Potential Operators

Once the solution of a boundary integral formulation has been approximated, potential operators can be used to compute point evaluations of the solution inside the domain.

6.1 Potential Operators for Laplace's Equation

For Laplace's equation, there are two potential operators that are used, as given in the table below.

Operator	Definition
Single layer	$(\mathcal{V}\mu)(\mathbf{x}) := \int_\Gamma G(\mathbf{x},\mathbf{y})\mu(\mathbf{y})\,\mathrm{d}\mathbf{y}$
Double layer	$(\mathcal{K}v)(\mathbf{x}) := \int_\Gamma \frac{\partial G(\mathbf{x},\mathbf{y})}{\partial \nu_\mathbf{y}} v(\mathbf{y})\,\mathrm{d}\mathbf{y}$

In each case, $G(\mathbf{x},\mathbf{y})$ is the Green's function for Laplace's equation.

To assemble potential operators in Bempp, the desired evaluation points must first be defined. For example, the following snippet creates a grid of 2500 points in the xy-plane with x and y between -3 and 3.

```
plot_grid = np.mgrid[-3:3:50j, -3:3:50j]
points = np.vstack((plot_grid[0].ravel(),
                    plot_grid[1].ravel(),
                    np.zeros(plot_grid[0].size)))
```

Potential operators can thenbe initialised in Bempp using:

```
from bempp.api.operators.potential import laplace
single = laplace.single_layer(domain, points)
double = laplace.double_layer(domain, points)
```

These can be applied to a grid function with:

```
single.evaluate(solution)
double.evaluate(solution)
```

As with boundary operators, `assembler` and `device_interface` keyword arguments can be used to control the assembly type used for potential operators.

6.2 Potential Operators for the Helmholtz Equation

For the Helmholtz equation, there are two potential operators that are used, as given in the table below.

Operator	Definition
Single layer	$(\mathcal{V}\mu)(\mathbf{x}) := \int_\Gamma G_k(\mathbf{x},\mathbf{y})\mu(\mathbf{y})\,\mathrm{d}\mathbf{y}$
Double layer	$(\mathcal{K}v)(\mathbf{x}) := \int_\Gamma \frac{\partial G_k(\mathbf{x},\mathbf{y})}{\partial \nu_\mathbf{y}} v(\mathbf{y})\,\mathrm{d}\mathbf{y}$

In each case, $G_k(\mathbf{x},\mathbf{y})$ is the Green's function for the Helmholtz equation with wavenumber k.

To assemble potential operators in Bempp, the desired evaluation points must first be defined. For example, the following snippet creates a grid of 2500 points in the xy-plane with x and y between -3 and 3.

```
plot_grid = np.mgrid[-3:3:50j, -3:3:50j]
points = np.vstack((plot_grid[0].ravel(),
                    plot_grid[1].ravel(),
                    np.zeros(plot_grid[0].size)))
```

Potential operators can thenbe initialised in Bempp using:

```
from bempp.api.operators.potential import helmholtz
single = helmholtz.single_layer(domain, points, wavenumber)
double = helmholtz.double_layer(domain, points, wavenumber)
```

These can be applied to a grid function with:

```
single.evaluate(solution)
double.evaluate(solution)
```

As with boundary operators, `assembler` and `device_interface` keyword arguments can be used to control the assembly type used for potential operators.

6.3 Potential Operators for the modified Helmholtz Equation

For the modified Helmholtz equation, there are two potential operators that are used, as given in the table below.

Operator	Definition
Single layer	$(\mathcal{V}\mu)(\mathbf{x}) := \int_\Gamma G_\omega(\mathbf{x},\mathbf{y})\mu(\mathbf{y})\,\mathrm{d}\mathbf{y}$
Double layer	$(\mathcal{K}v)(\mathbf{x}) := \int_\Gamma \frac{\partial G_\omega(\mathbf{x},\mathbf{y})}{\partial \nu_\mathbf{y}} v(\mathbf{y})\,\mathrm{d}\mathbf{y}$

In each case, $G_\omega(\mathbf{x},\mathbf{y})$ is the Green's function for the modified Helmholtz equation with frequency ω.

To assemble potential operators in Bempp, the desired evaluation points must first be defined. For example, the following snippet creates a grid of 2500 points in the xy-plane with x and y between -3 and 3.

```
plot_grid = np.mgrid[-3:3:50j, -3:3:50j]
points = np.vstack((plot_grid[0].ravel(),
                    plot_grid[1].ravel(),
                    np.zeros(plot_grid[0].size)))
```

Potential operators can thenbe initialised in Bempp using:

```
from bempp.api.operators.potential import modified_helmholtz
single = modified_helmholtz.single_layer(domain, points, omega)
double = modified_helmholtz.double_layer(domain, points, omega)
```

These can be applied to a grid function with:

```
single.evaluate(solution)
double.evaluate(solution)
```

As with boundary operators, `assembler` and `device_interface` keyword arguments can be used to control the assembly type used for potential operators.

6.4 Potential Operators for Maxwell's Equations

For Maxwell's equations, there are two potential operators that are used, as given in the table below.

Operator	Definition
Electric field	$(\mathcal{E}(\mathbf{p}))(\mathbf{x}) = \mathrm{i}k \int_\Gamma \mathbf{p}(\mathbf{y}) G_k(\mathbf{x},\mathbf{y})\,\mathrm{d}\mathbf{y} - \frac{1}{\mathrm{i}k}\nabla_\mathbf{x} \int_\Gamma \nabla_\Gamma \cdot \mathbf{p}(\mathbf{y}) G_k(\mathbf{x},\mathbf{y})\,\mathrm{d}\mathbf{y}$
Magnetic field	$(\mathcal{H}(\mathbf{p})(\mathbf{x}) = \nabla_\mathbf{x} \times \int_\Gamma \mathbf{p}(\mathbf{y}) G(\mathbf{x},\mathbf{y})\,\mathrm{d}\mathbf{y}$

In each case, $G_k(\mathbf{x},\mathbf{y})$ is the Green's function for the Helmholtz equation with wavenumber k.

To assemble potential operators in Bempp, the desired evaluation points must first be defined. For example, the following snippet creates a grid of 2500 points in the xy-plane with x and y between -3 and 3.

```
plot_grid = np.mgrid[-3:3:50j, -3:3:50j]
points = np.vstack((plot_grid[0].ravel(),
                    plot_grid[1].ravel(),
                    np.zeros(plot_grid[0].size)))
```

Potential operators can thenbe initialised in Bempp using:

```
from bempp.api.operators.potential import maxwell
electric = maxwell.electric_feild(domain, points, wavenumber)
magnetic = maxwell.magnetic_feild(domain, points, wavenumber)
```

These can be applied to a grid function with:

```
electric.evaluate(solution)
magnetic.evaluate(solution)
```

As with boundary operators, `assembler` and `device_interface` keyword arguments can be used to control the assembly type used for potential operators.

Part III

A Journey into the Bempp Core

Bempp is split into two parts: `bempp.api`, which contains all the user-facing functionality of the library; and `bempp.core`, which contains the library's fast assembly routines.

In the first section of this handbook, we explored the functionality in `bempp.api`. In this section, we take a look at the fast core of the library and look at how operator assembly is carried out.

Chapter 7

Assembling Operators

The functionality in `bempp.core` is almost exclusively for operator assembly and the multiplication of discrete operators and vectors, as these is the most computationally-heavy components of BEM.

Bempp uses just-in-time compiled OpenCL or Numba kernels to quickly assemble the dense matrices that arise from discretising BEM operators.

For larger problems, however, the use of dense matrices is expensive, both in terms of computation time and storage space. For such problems, Bempp can use the fast multipole method to speed up matrix assembly and the computation of matrix-vector products. This is done via interfaces to the external ExaFMM library.

7.1 Assembling Operators using OpenCL

OpenCL is a C-based compute language designed to allow a single script to be parallelised on a wide range of CPU and GPU devices. Bempp uses [PyOpenCL]() to just-in-time compile its OpenCL kernels when they are needed.

Bempp's OpenCL kernels are stored in the folder [`bempp/core/sources/kernels`]().

7.2 Assembling Operators using Numba

On some systems (for example recents versions of MacOS), OpenCL is not available or has some features unavailable. If this is the case, Bempp can use [Numba]() to just-in-time compile operator assembly routines.

Bempp's Numba kernels are defined in the file [`bempp/core/numba_kernels.py`]().

7.3 Assembling Operators using FMM

For larger problems, dense assembly using OpenCL or Numba become very expensive, both in terms of computation time and memory consumption. In such cases, Bempp can use the fast multipole method (FMM) to speed up its calculations and reduce memory usage.

Internally, Bempp uses the [ExaFMM]() library to carry out its FMM computations.

Part IV

A Bempp User's Introduction to Boundary Element Methods

The Bempp data structures closely follow the underlying mathematics. This allows the user to solve problems with Python code that closely resembled thier BEM formulation. In many cases, knowledge of some details of the underlying mathematics is required to decide how best to formulate and solve a problem.

In this section of the Bempp Handbook, we look at some highlights of the mathematical theory behind boundary element methods.

This section of the Bempp Handbook is still being written and some sections are still incomplete.

Chapter 8

Function Spaces

To use the boundary element method, we start with a variational boundary integral equation, for example: Find $u \in H^{1/2}(\Gamma)$ such that for all $v \in H^{1/2}(\Gamma)$,

$$\langle \mathsf{V}u, v \rangle = \langle f, v \rangle .$$

An approximation of the solution of this problem is then found by discretising the problem and searching for a solution in a subspace $\mathcal{V}_h \subset H^{1/2}(\Gamma)$.

In this section of the Bempp Handbook, we look at the definitions of continuous function spaces such as $H^{1/2}(\Gamma)$ that are used in the boundary integral equations, and we look at how discrete subspaces of these are usually defined.

8.1 Sobolev Spaces

This section of the Bempp Handbook introduces the Sobolev function spaces in which the solutions to the variational boundary integral equation are sought.

Let $\Omega^- \subset \mathbb{R}^3$ be a bounded domain and let Γ be the boundary of Ω^-. Let $\Omega^+ = \mathbb{R}^3 \setminus \Omega^-$ be the region exterior to Ω^-.

8.1.1 Scalar function spaces

For Laplace and Helmholtz problems, we use spaces containing scalar functions $f : \Omega^- \to \mathbb{C}$. We begin by defining the space of square integrable functions:

$$H^0(\Omega^-) := L^2(\Omega^-) := \left\{ v : \Omega^- \to \mathbb{C} \middle| \int_{\Omega^-} |v|^2 < \infty \right\}$$

We then define the Sobolev space $H^1(\Omega^-)$ to be the space of square integrable functions whose first derivatives are also square integrable.

$$H^1(\Omega^-) := \left\{ v \in L^2(\Omega^-) \middle| \frac{\partial v}{\partial x}, \frac{\partial v}{\partial y}, \frac{\partial v}{\partial z} \in L^2(\Omega) \right\}$$

In general, for each positive integer k, we define the space $H^k(\Omega^-)$ to be the space of square integrable functions whose derivatives of order up to and including k are also square integrable.

Traces

Next, we define the Dirichlet and Neumann traces of a function on the boundary by

$$(\gamma_{\mathrm{D}}^{-} v)(\mathbf{x}) := \lim_{\Omega^{-} \ni \mathbf{x}' \to \mathbf{x} \in \Gamma} v(\mathbf{x}'),$$

$$(\gamma_{\mathrm{N}}^{-} v)(\mathbf{x}) := \gamma_{\mathrm{D}} \nabla v(\mathbf{x}') \cdot \mathbf{n}_{\mathbf{x}}.$$

We define the space $H^{1/2}(\Gamma)$ to be the Dirichlet trace of the space $H^1(\Omega^-)$:

$$H^{1/2}(\Gamma) := \gamma_{\mathrm{D}} H^1(\Omega^-) = \left\{ \gamma_{\mathrm{D}} v \middle| v \in H^1(\Omega^-) \right\}$$

In general, for each positive integer k, we define the space $H^{k-1/2}(\Gamma)$ to be the Dirchlet trace of the space $H^k(\Omega^-)$.

The space $H^{-1/2}(\Gamma)$ is defined as the dual space of $H^{1/2}(\Gamma)$:

$$H^{-1/2}(\Gamma) = \left\{ f : H^{1/2}(\Gamma) \to \mathbb{C} \right\}.$$

8.1.2 Vector function spaces

For Maxwell problems, we use spaces containing vector functions $\mathbf{f} : \Omega^- \to \mathbb{C}^3$. We begin by defining the space of square integrable functions:

$$\mathbf{H}^0(\Omega^-) := \mathbf{L}^2(\Omega^-) := \left\{ \mathbf{v} : \Omega^- \to \mathbb{C}^3 \middle| \int_{\Omega^-} |\mathbf{v}|^2 < \infty \right\}$$

We then define the Sobolev space $\mathbf{H}^1(\Omega^-)$ to be the space of square integrable functions whose first derivatives are also square integrable.

$$\mathbf{H}^1(\Omega^-) := \left\{ \mathbf{v} \in \mathbf{L}^2(\Omega^-) \middle| \frac{\partial \mathbf{v}}{\partial x}, \frac{\partial \mathbf{v}}{\partial y}, \frac{\partial \mathbf{v}}{\partial z} \in \mathbf{L}^2(\Omega) \right\}$$

In general, for each positive integer k, we define the space $\mathbf{H}^k(\Omega^-)$ to be the space of square integrable functions whose derivatives of order up to and including k are also square integrable.

Traces

On Γ, we define the space of square integrable tangential vector fields:

$$\mathbf{L}^2_{\mathbf{t}}(\Gamma) := \left\{ \mathbf{v} \in \mathbf{L}^2(\Gamma) \middle| \mathbf{v} \cdot \mathbf{n} = 0 \right\}$$

Next, we define the tangential and Neumann traces of a function on the boundary by

$$(\gamma_{\mathbf{t}}^{-} \mathbf{v})(\mathbf{x}) := \lim_{\Omega^{-} \ni \mathbf{x}' \to \mathbf{x} \in \Gamma} \mathbf{v}(\mathbf{x}') \times \mathbf{n}_{\mathbf{x}},$$

$$(\gamma_{\mathbf{N},\mathbf{k}}^{-} \mathbf{v})(\mathbf{x}) := \frac{1}{\mathrm{i}k} \gamma_{\mathbf{t}} \nabla \times \mathbf{v}(\mathbf{x}').$$

We define the space $\mathbf{H}^{1/2}_{\times}(\Gamma)$ to be the tangential trace of the space $\mathbf{H}^1(\Omega^-)$:

$$\mathbf{H}^{1/2}_{\times}(\Gamma) := \gamma_{\mathbf{t}} \mathbf{H}^1(\Omega^-) = \left\{ \gamma_{\mathbf{t}} \mathbf{v} \middle| \mathbf{v} \in \mathbf{H}^1(\Omega^-) \right\}$$

We define the spaces of div- and curl-conforming functions by:

$$\mathbf{H}_{\times}^{-1/2}(\mathrm{div}_\Gamma, \Gamma) := \left\{ \mathbf{v} \in \mathbf{H}_{\times}^{1/2}(\Gamma) \middle| \mathrm{div}_\Gamma\, \mathbf{v} \in H^{1/2} \times (\Gamma) \right\}$$

$$\mathbf{H}_{\times}^{-1/2}(\mathrm{curl}_\Gamma, \Gamma) := \left\{ \mathbf{v} \in \mathbf{H}_{\times}^{1/2}(\Gamma) \middle| \mathrm{curl}_\Gamma\, \mathbf{v} \in H^{1/2} \times (\Gamma) \right\}$$

In these definitions, div_Γ and curl_Γ are the scalar surface div and curl operators. Using the definition of these, it can be seen that

$$\mathbf{H}_{\times}^{-1/2}(\mathrm{curl}_\Gamma, \Gamma) = \left\{ \mathbf{n} \times \mathbf{v} \middle| \mathbf{v} \in \mathbf{H}_{\times}^{-1/2}(\mathrm{div}_\Gamma, \Gamma) \right\}.$$

8.2 Discrete Function Spaces

When approximating the solution of a problem with the boundary element method, finite-dimensional subspaces of the relevant Sobolev spaces are used to discretise the problem.

In this section of the Bempp Handbook, we look at how these discrete spaces are defined.

8.2.1 Degrees of Freedom (DOFs)

An abstract finite element is defined by:

- A reference element $R \subset \mathbb{R}^d$. In Bempp, R is always a triangle with vertices at $(0, 0)$, $(1, 0)$ and $(0, 1)$. - A finite dimensional polynomial space $\mathcal{V}$. Inside each triangle in the mesh, the solution will be approximated by a function in this space. - A set of functionals $\mathcal{L} = f_1, ...f_n$ that form a basis of the dual space $\mathcal{V}^* = \{f : \mathcal{V} \to \mathbb{R}\}$.

Given a functional $f_i \in \mathcal{L}$, a corresponding polynomial basis function $\phi_i \in \mathcal{V}$ is defined as the function such that

$$f_j(\phi_i) = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}.$$

Example: P1 space

As an example, for a P1 (continuous piecewise linear space) the following are used:

- R is the reference triangle. - $\mathcal{V} = \mathrm{span}\{1, x, y\}$. - $\mathcal{L}$ is the set of point evaluations at the vertices of R.

In this case, it is common to say that the space has a DOF at each vertex of the mesh.

Spaces used by Bempp

The definitions of the spaces available in Bempp are summarised in the following table. In each case, R is the unit triangle.

Space	$\mathcal{V}$	$\mathcal{L}$
DP0	span$\{1\}$	Point evaluation at centre of R
P1	span$\{1, x, y\}$	point evaluations at vertices of R
RWG1	$\text{span}\left\{\begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}, \begin{pmatrix}x\\y\end{pmatrix}\right\}$	Point evaluations at the midpoints of edges of R in a d
SNC1	$\text{span}\left\{\begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}, \begin{pmatrix}y\\-x\end{pmatrix}\right\}$	Point evaluations at the midpoints of edges of R in a d

The spaces defined on the barycentric dual grid are defined as subspaces of the spaces in the table above. Their definitions can be found in *A dual finite element complex on the barycentric refinement* (2007) by A. Buffa and S. Christiansen.

8.2.2 Inf-sup Stability

8.2.3 Interpolation and Projection

We now take a closer look at what happens in the initialisation of this Grid-Function. Denote the global basis functions of the space by ψ_j, for $j = 1, \ldots, N$. The computation of the grid function consists of two steps:

+ Compute the projection coefficients $p_j = \int_\Gamma \overline{\psi_j(\mathbf{y})} f(\mathbf{y}) \mathrm{d}\mathbf{y}$, where f is the analytic function to be converted into a grid function and Γ is the surface defined by the grid. + Compute the basis coefficients c_j from $Mc = p$, where M is the mass matrix defined by $M_{ij} = \int_\Gamma \overline{\psi_i(\mathbf{y})} \psi_j(\mathbf{y}) \mathrm{d}\mathbf{y}$.

This is an orthogonal $\mathcal{L}^2(\Gamma)$-projection onto the basis $\{\psi_1, ..., \psi_N\}$.

Chapter 9

Operators

9.1 Potential Operators

9.2 Boundary Operators

9.3 The Calderón Projector

Chapter 10

Linear Solvers

10.1 Direct Solvers

10.2 Iterative Solvers

10.2.1 Condition Numbers and Preconditioning

Chapter 11

Deriving BEM Formulations

11.1 Deriving BEM Formulations for Laplace and Helmholtz Problems

11.2 Deriving BEM Formulations for Maxwell Problems

Chapter 12

Avoiding Dense Matrices

12.1 The Fast Multipole Method

12.2 Hierarchical Matrices

Chapter 13

References and Further Reading