

# MPSpack user manual

Alex Barnett\* and Timo Betcke†

July 26, 2009

## Abstract

MPSpack is a fully object-oriented MATLAB toolbox for solving Laplace, Helmholtz, wave scattering, and related PDE boundary-value problems on piecewise-homogeneous 2D domains, including those with corners. The philosophy is to use basis functions which are particular solutions to the PDE in some region; solving is thus reduced to matching on the boundary (or on boundaries of subregions). This idea is known as the Method of Particular Solutions, or as Trefftz, ultra-weak, or non-polynomial methods in the FEM community. Basis functions include plane-wave, Fourier-Bessel, corner-adapted expansions, and fundamental solutions. Layer potential representations and associated singular quadrature schemes are also available. It is designed to be simple to use, and to enable highly-accurate solutions. This is the user manual; for a more hands-on approach and worked examples see the accompanying tutorial.

## 1 Overview

In numerical analysis there has been recent excitement in methods for solving linear PDEs where solutions are approximated by linear combinations of *particular solutions* to the PDE. These methods are high-order (often exponentially convergent), efficient at high frequencies (the number of degrees of freedom  $N$  scales linearly with wavenumber, in 2D), and are quite simple to

---

\*Department of Mathematics, Dartmouth College, Hanover, NH, 03755, USA

†Department of Mathematics, University of Reading, Berkshire, RG6 6AX, UK

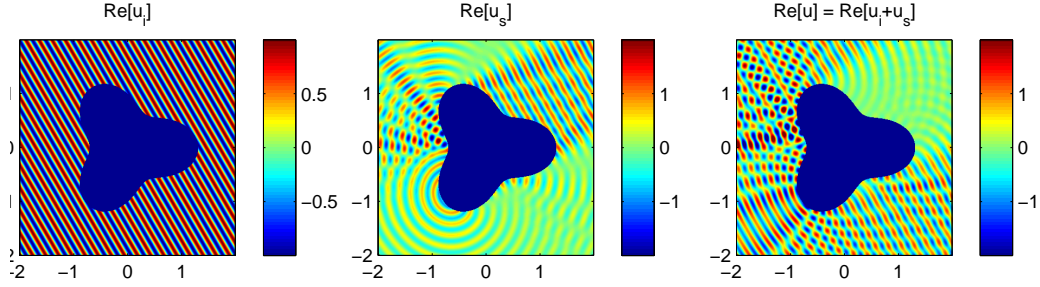


Figure 1: Sound-hard (homogeneous Neumann boundary condition) scattering from a smooth obstacle using a fundamental solutions basis in **MPSpack**. Left: incident wave. Center: scattered wave. Right: total solution field.

implement. When geometries become more complicated, the domain needs to be split up into multiple subdomains, e.g. one for each corner, and the implementation and matrix construction becomes cumbersome. The goal of this software toolbox is to make implementation of these methods simple and transparent, and create an intuitive, unifying framework in which many types of boundary-value problems, boundary conditions, and domain geometries may be solved, explored and visualized with ease. In these methods there is either no mesh (as in fundamental solutions methods), or the number of subdomains is small and fixed (so-called  $p$ -FEM). Thus we leave the job of specifying subdomains, i.e. ‘meshing’, to the user—this contrasts with the large number of finite-element packages already in existence. The benefit of such methods is their rapid convergence and efficiency. The solution boils down to dense least-squares linear algebra, which despite the  $O(N^3)$  cost enables some quite high-frequency problems to be solved.

We focus on the scalar homogeneous Helmholtz equation in the plane,

$$(\Delta + k^2)u = 0 \quad \text{in } \Omega, \quad (1)$$

where  $\Omega \subset \mathbb{R}^2$  is an interior or exterior domain,  $k \geq 0$  is the wavenumber, and certain linear boundary conditions are imposed. Such problems arise in wave scattering and cavity resonances. In **MPSpack** we use the (recently-acquired) power of MATLAB [5] *object-oriented* programming to represent the mathematical objects such as segments, domains, basis sets, and BVPs, by software objects that may be manipulated just like variables. The result is that complicated problems may be set up and solved in a few lines of quite-readable code. For example, to solve and plot the scattering of plane-wave

incident from a smooth sound-hard (Neumann) domain 10 wavelengths in size, we can type,

```
s = segment.radialfunc(250, {@(q) 1 + 0.3*cos(3*q), @(q) -0.9*sin(3*q)});
d = domain([], [], s, -1);
s.setbc(1, 'N', []);
opts.tau = 0.04; d.addmfsbasis(s, 210, opts);
p = scattering(d, []);
p.setoverallwavenumber(30);
p.setincidentwave(pi/6);
p.solvecoeffs;
p.showthreefields;
```

which produces Fig. 1 in about 1 sec of CPU time, to an accuracy of  $10^{-9}$  (boundary condition  $L^2$ -norm).

More generally we may have multiple domains with different wavenumbers connected by homogeneous or inhomogeneous boundary conditions, as in transmission, dielectric-coated, acoustic or photonics problems. With  $k = 0$  we have Laplace's equation, with applications to electrostatics, steady-state heat flow, and probability. In this release we discuss only boundary-value problems (BVPs). Extensions to eigenvalue and periodic problems are already in progress; we will document these in future releases.

The accompanying tutorial is the best way to leap right in to using the package. The rest of this rather brief manual is more of a 'top-down' document, describing installation, the PDEs that may be solved, our data structures and design choices, limitations, and acknowledgments. As usual you may also get help on any **MPSPack** command by typing **help** followed by the command name at the MATLAB prompt.

## 1.1 Installation

Requirements: MATLAB version 7.6 (2008a) or newer is needed, since we make heavy use of recent object-oriented programming features. No other MATLAB toolboxes are needed. If you wish to use faster regular Bessel functions you may want to install the GNU Scientific Library [4].

The project is hosted at the repository

<http://code.google.com/p/mpspack>

There are two alternative methods to download and unpack:

1. Ensure you have subversion (`svn`) installed. This is available from <http://subversion.tigris.org>. Anonymous check out (download) of MPSPack is then via the subversion command:

```
svn co http://mpspack.googlecode.com/svn/branches/release
mpspack
```

This creates a directory `mpspack` containing the toolbox.

You might prefer a more user-friendly graphical subversion client such as those listed at <http://subversion.tigris.org/links.html#clients>

2. Get a gzip-compressed tar archive from

```
http://code.google.com/p/mpspack/downloads/list
```

In a UNIX environment you may now unpack this with

```
tar zxvf mpspack.tar.gz
```

This creates the directory `mpspack` containing the toolbox.

There are some optional fast basis and other math libraries (C and Fortran with MEX interfaces), which should be compiled in a UNIX environment as follows: from the `mpspack` directory type `make`. (If GSL [4] is not installed, you will need to remove the references to `gsl` codes in `@utils/Makefile` before executing `make`).

You should now add the `mpspack` directory to your MATLAB path, for instance by adding the line

```
addpath 'path/to/mpspack';
```

to your MATLAB `startup.m` file. You are now ready to use MPSPack !

## 1.2 What problems can MPSPack solve?

Here we give a general framework (for examples see [1, 2]). Let  $\Omega_j \subset \mathbb{R}^2$ ,  $j = 1, \dots, D$  be a set of (possibly multiply connected) domains. One of the domains may be an exterior domain. The solution domain is  $\Omega := \bigcup_{j=1}^D \Omega_j$ , and we seek a solution  $u : \Omega \rightarrow \mathbb{C}$ . In each domain we have,

$$(\Delta + n_j^2 k^2)u = 0 \quad \text{in } \Omega_j, \quad (2)$$

where the ‘overall wavenumber’ (or frequency)  $k$  has been scaled by  $n_j$  for each domain. In the optical application  $n_j$  is interpreted as a *refractive index* (with  $n_j = 1$  vacuum or ‘air’) and we will use this name.

For all boundaries  $\Gamma_j := \partial\Omega_j \cap \partial\Omega$  at the edge of the solution domain we have boundary conditions

$$a_j u + b_j u_n = f_j \quad \text{on } \Gamma_j , \quad (3)$$

where  $a_j, b_j \in \mathbb{C}$  are complex numbers (currently; in future they may be functions on the boundary), and  $f : \Gamma_j \rightarrow \mathbb{C}$  are (possibly identically zero) driving functions.  $u_n$  is short for  $\mathbf{n} \cdot \nabla u$ , the normal derivative on the boundary.<sup>1</sup> If there is a nonempty *common* boundary  $\Gamma_{ij} := \partial\Omega_i \cap \partial\Omega_j$  then it has value and derivative matching (continuity) conditions,

$$a_{ij}^+ u^+ + a_{ij}^- u^- = f_{ij} \quad \text{on } \Gamma_{ij} , \quad (4)$$

$$b_{ij}^+ u_n^+ + b_{ij}^- u_n^- = g_{ij} \quad \text{on } \Gamma_{ij} , \quad (5)$$

where  $a_{ij}^+, a_{ij}^-, b_{ij}^+, b_{ij}^-$  are numbers and  $f_{ij}, g_{ij}$  are driving functions. The notation  $u^+$  ( $u^-$ ) means the limiting value approaching the boundary  $\Gamma_{ij}$  from its positive (negative) normal side.

We assume all the boundaries  $\Gamma_j$  and  $\Gamma_{ij}$  are piecewise smooth, and each smooth piece we will build from one or more *segments*. If  $\Omega_j$  is the exterior domain (with  $n_j = 1$ ), we may wish to impose additional boundary conditions at infinity, such as Sommerfeld's radiation condition,

$$iku - \frac{\partial u}{\partial r} = o(r^{1/2}) , \quad (6)$$

where  $r$  is the radial coordinate. This occurs in the scattering context, where the unknown satisfying the above BVP is now usually renamed  $u_s$ , and the total field is then  $u = u_{inc} + u_s$  with  $u_{inc}$  the incident wave [3]. As is standard with integral equation methods, this is achieved by choosing basis sets (MFS, layer potentials, etc.) satisfying the radiation condition.

To solve a BVP the flow using **MPSPack** is often as follows (look back to the code given in Sec. 1):

1. define piecewise-smooth segments forming all boundaries
2. build domains using various of these segments as their boundaries
3. set up (in)homogeneous boundary or matching conditions on each segment

---

<sup>1</sup>Within this framework it is possible to have domains with ‘slits’ or cracks, as long as each side of the crack is defined to be a different part of  $\Gamma_j$ .

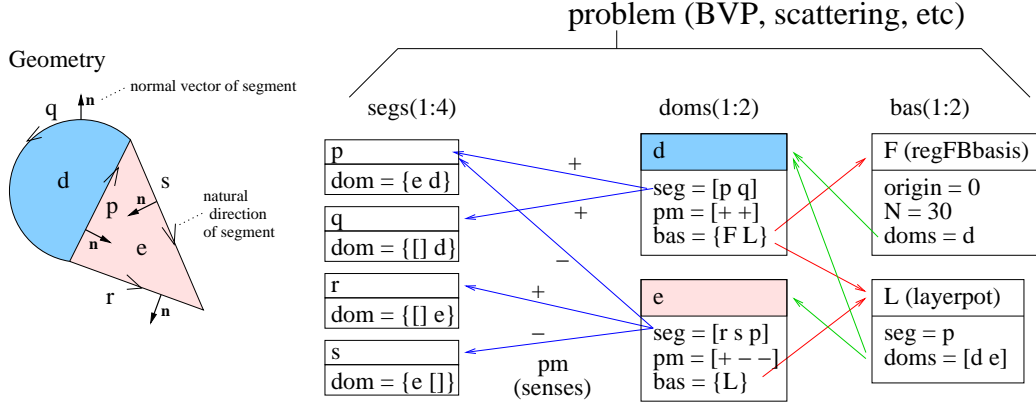


Figure 2: Relationship between segments, domains, and basis sets in a simple example. The physical geometry is shown on the left. There is a regular Fourier-Bessel basis in domain  $d$ , and a layer-potential density on segment  $p$  which affects both domains  $d$  and  $e$ . The corresponding code objects are on the right. Pointers to the four segments, two domains, and two basis sets are stored within the problem instance.

4. choose basis set(s) within each domain
5. build then solve a dense least-squares linear system to get the basis coefficient vector
6. check residual error in satisfying boundary and matching conditions
7. evaluate and plot solution on desired points or grid

The accompanying tutorial document is a good way to explore these steps in the context of examples. We now discuss how software data structures represent the above objects.

## 2 Objects: segments, domains, and basis sets

Fig. 2 overviews how segments, domains and basis sets are represented in **MPSPack** in a simple example. For all these objects we use MATLAB's **handle** class, which means that only a single copy of any object instance is stored, and any time an instance is copied or passed as a function argument, it is a *pointer* to that instance that is duplicated. (This contrasts the *value*

class, such as numeric variables, in which the *actual data* is duplicated when copied or passed as an argument.) Thus each domain stores (as one of its properties)<sup>2</sup> an array of pointers to the segments forming its boundary. It also stores a cell array of the basis sets that contribute to the solution inside it. Each basis set stores a list of the domains it affects—usually this is a single domain. There are also boundary conditions stored in each segment, which are not shown. The ‘problem’ object (BVP, scattering problem, etc) contains arrays of (pointers to) all relevant segments, domains, and basis sets. *Methods* (i.e. commands available which act on the problem class) then use this internal information to, for instance, construct a matrix and solve the problem.

## 2.1 Segments

All coordinates in the plane are stored as complex double-precision numbers. In other words the point  $(2, 3)$  is represented by  $2+3i$  in `MPSpack`. A segment is specified as a parametrized complex-valued function  $z(t)$  for  $0 \leq t \leq 1$ , and  $z'(t)$  is also needed. If `s` is a segment, these are stored as the properties `s.Z` and `s.Zp` respectively. Given a list `s.t` of quadrature points on  $[0, 1]$  (and their weights), the parametrization  $z$  is used to compute quadrature point locations `s.x` and weights `s.w` for approximating integrals with respect to arc-length on the segment. Unit normals `s.nx` are computed when needed via the expression  $iz'/|z'|$ , which shows that a segment’s normal always points to the *right* when the segment is traversed in its natural direction (increasing  $t$ ); see Fig. 2.<sup>3</sup> Since our focus is on high-order and accurate methods, we feel that forcing the user to go to the trouble of providing  $z$  and  $z'$  is reasonable.<sup>4</sup> The benefit is that highly accurate quadrature is possible, and the user may simply switch between quadrature schemes (`s.quadr`) and numbers of points.

Segments start out their lives *unconnected* to any domains, as evidenced by two empty elements as their domain connection property cell-array `s.dom = {[ ] [ ]}`.

---

<sup>2</sup>A *property* is a variable stored inside the object instance, just like a field `a.b` inside a struct `a`.

<sup>3</sup>A segment is in fact a subclass of a simple object we call a pointset. A pointset `p` contains only a list of locations `p.x` and possibly corresponding normal directions `p.nx`.

<sup>4</sup>In future releases we may allow  $z$  and  $z'$  to be automatically generated by high-order polynomial fits to a set of boundary points such as might be available from an engineering or CAD package.

## 2.2 Domains

A domain object `d` contains as properties an ordered list of segments `d.segs` which form its boundary, and an equal-sized list `d.pm` of *senses* ( $\pm 1$ ) specifying whether each segment is to be taken in its ‘forward’ (natural,  $+1$ ) direction, or ‘backward’ (reverse,  $-1$ ) direction. All segments taken in these (possibly reversed) senses must i) connect up head to tail in the correct order, in one or more connected closed loops, and ii) have the domain interior lying to its left side when traversed according to its sense. The latter ensures that the segment normals, *when multiplied by the corresponding senses*, always point *outwards* (away) from the domain. At this point the reader would be wise to check they are able to write down correctly from the geometry the `pm` arrays in Fig. 2.

In order to distinguish disconnected components of the boundary of `d`, the list `d.spiece` specifies which boundary piece each segment is part of. E.g. `spiece = [1 1 2 3]` means there are three boundary pieces, the first involving two segments, and the other two only one segment each. If `s.exterior = 0` then the domain is not an exterior domain, thus we could deduce that there is an outer boundary (which is always the first piece, and taking into account senses is traversed in a CCW direction) with two excluded regions (each traversed in a CW direction). On the other hand, if `s.exterior = 1` the domain is the whole plane with three disjoint excluded regions (each in a CW direction). We recommend now looking at the plots produced by `test/testdomain.m` and examining their object properties at the command prompt.

Each domain also contains a list of corners where segments meet: the first corner is where the end of the last segment of piece 1 joins the start of the first segment of piece 1. Subsequent corners follow in the same order as segments. The angles (in radians) subtended by each corner on the domain interior side are in `d.cang`, and their locations in `d.cloc`. The angle at which segment `d.seg(j)` ‘heads off in’ at its start (when taken in the sense given by `pm` as above) is `d.cangoff(j)`, expressed as a complex number on the unit circle. The advantage of domains containing corner information is that corner-adapted basis sets may automatically be added.

When a domain is constructed by passing in lists of segments and senses, the relevant segment’s side is *connected* to the domain. E.g. if the natural positive side of segment `s` has been used as a boundary (approaching from the interior) of domain `d`, we find `s.dom{1} = d`. This bookkeeping is used



to ensure that each side of a segment can only be connected to at most one domain. If a domain-connected segment `s` is to be reused making new domains, one must first run the method `s.disconnect` which empties both elements of the `s.dom` cell array.

## 2.3 Boundary conditions

These are stored on a per-segment basis, as properties of each segment instance. A boundary condition (BC) may reside on only *one* side of a segment (MPSPack checks that this side is connected to a domain), as specified by the `bcside` property. `s.bcside = +1` indicates a BC on the natural (positive normal) side of segment `s`, and `s.bcside = -1` a BC on the opposite (negative normal) side. (This is independent of any of the `pm` senses stored in the connected domains.) Then the numbers `s.a`, `s.b`, and function handle `s.f` give the  $a_j$  and  $b_j$  coefficients and function  $f_j$  from (3).

If `s.bcside = 0` this indicates a matching condition (4) and (5) rather than a BC. In this case, `s.a` contains a 2-element array with coefficients  $[a^+, a^-]$  from (4), `s.b` contains  $[b^+, b^-]$  from (5), and `s.f` and `s.g` contain the functions  $f$  and  $g$ .

In the above, function handles may be replaced by a list (column vector) of function *values* at the quadrature points (however, now the user must ensure that, if the quadrature points change, that these function values are updated accordingly).

## 2.4 Basis sets

A basis set object `b` affects the function values inside the domain in the pointer `b.doms`.<sup>5</sup> When a basis set is *added* to a basis-set-free domain `d`, two things happen: i) a new instance of the appropriate basis class is created, with `d.bas{1}` pointing to this basis object, and ii) this basis object has property `doms` set to `d`. These pointers in both directions are shown in Fig. 2, and help later bookkeeping to be rapid (i.e. free of searches).

The types of basis objects currently implemented are

---

<sup>5</sup>In fact, `doms` may be a list, but currently only obscure layer potential objects have the power to influence more than one domain. Therefore we discuss only the simpler case of one affected domain.

- Regular Fourier-Bessel expansion (**regfbasis**) of degree  $N$ , comprising the set of  $2N + 1$  functions, for Helmholtz ( $k > 0$ ),

$$\begin{aligned} & \{J_n(kr) \cos(n\theta)\}_{n=0,\dots,N} \cup \{J_n(kr) \sin(n\theta)\}_{n=1,\dots,N} && \text{(real case)} \\ \text{or} & \{J_n(kr) e^{in\theta}\}_{n=-N,\dots,N} && \text{(complex case)} \end{aligned}$$

where  $(r, \theta)$  are polar coordinates relative to an origin  $z_0$  (basis property **origin**). If the **real** property is true (default), the real set is used, otherwise complex. Or, for the Laplace ( $k = 0$ ) case, as a function of coordinate  $z$ ,

$$\begin{aligned} & \{1, \operatorname{Re} w, \operatorname{Re} w^2, \dots, \operatorname{Re} w^N, \operatorname{Im} w, \dots, \operatorname{Im} w^N\} && \text{(real case)} \\ \text{or} & \{w^{-N}, w^{-N+1}, \dots, w^N\} && \text{(complex case)} \end{aligned}$$

where  $w := z - z_0$ .

- Fractional-order (wedge of angle  $\pi/\nu$ ) Fourier-Bessel expansion (**nufbbasis**) of degree  $N$ , for Helmholtz ( $k > 0$ ),

$$\begin{aligned} & \{J_{\nu n}(kr) \cos(\nu n\theta)\}_{n=0,\dots,N} && \text{(cos type)} \\ & \{J_{\nu n}(kr) \sin(\nu n\theta)\}_{n=1,\dots,N} && \text{(sin type)} \end{aligned}$$

where  $(r, \theta)$  are polar coordinates relative to the wedge corner  $z_0$  such that  $\theta = 0$  is aligned with the most clockwise edge of the wedge interior. The basis set may be of three types (property **type**): '**c**' cos type only ( $N+1$  functions), '**s**' sin type only ( $N$  functions), and '**cs**' cos and sin types ( $2N + 1$  functions). The cos type satisfies zero Neumann BCs on the wedge boundary, the sin type zero Dirichlet BCs. Since they have a singularity at  $z_0$ , the branch cut is chosen (angle property **branch**) by default to point symmetrically away from the wedge interior.

No complex version is implemented. Or, for the Laplace ( $k = 0$ ) case,

$$\begin{aligned} & \{r^{\nu n} \cos(\nu n\theta)\}_{n=0,\dots,N} && \text{(cos type)} \\ & \{r^{\nu n} \sin(\nu n\theta)\}_{n=1,\dots,N} && \text{(sin type)} \end{aligned}$$

- Set of real (as opposed to evanescent) plane waves (**rpwbasis**), with  $N$  travel directions  $\mathbf{n}_j = (\cos \theta_j, \sin \theta_j)$ , where  $\theta_j = \pi j/N$ ,  $j = 1, \dots, N$ , for  $k > 0$ ,

$$\begin{aligned} & \{\cos(k\mathbf{n}_j \cdot \mathbf{x})\}_{j=1,\dots,N} \cup \{\sin(k\mathbf{n}_j \cdot \mathbf{x})\}_{j=1,\dots,N} && \text{(real case)} \\ & \{e^{ik\mathbf{n}_j \cdot \mathbf{x}}\}_{j=1,\dots,N} \cup \{e^{-ik\mathbf{n}_j \cdot \mathbf{x}}\}_{j=1,\dots,N} && \text{(complex case)} \end{aligned}$$

Here the coordinate is  $\mathbf{x} := (x, y)$ , and the usual dot product in  $\mathbb{R}^2$  is used. Note the travel directions are equally spaced in  $(0, \pi]$ . If the **real** property is true (default), the real set is used, otherwise complex. There are  $2N$  functions in each case. The  $k = 0$  limit does not give a useful basis set, so it left undefined.

- Set of fundamental solutions (**mfsbasis**), with  $N$  origins (charge points)  $\mathbf{y}_j \in \mathbb{R}^2$ ,  $j = 1, \dots, N$ , a linear combination of monopole and dipole,

$$\{i\eta\Phi(\mathbf{x}, \mathbf{y}_j) + \frac{\partial\Phi}{\partial n_{\mathbf{y}_j}}(\mathbf{x}, \mathbf{y}_j)\}_{j=1, \dots, N} \quad (7)$$

where  $\eta \in \mathbb{C}$  is a parameter (property **eta**), and the fundamental solution is defined for  $\mathbf{x} \in \mathbb{R}^2 \setminus \{\mathbf{y}\}$  as

$$\Phi(\mathbf{x}, \mathbf{y}) = \Phi(\mathbf{x} - \mathbf{y}) = \begin{cases} \frac{i}{4} H_0^{(1)}(k|\mathbf{x} - \mathbf{y}|), & k > 0 \\ \frac{1}{2\pi} \log \frac{1}{|\mathbf{x} - \mathbf{y}|}, & k = 0 \end{cases} \quad (8)$$

Note that the normal derivative of  $\Phi$  is taken with respect to its second argument, as in a double-layer kernel. The linear combination of monopoles and dipoles is useful to prevent interior resonances of the MFS curve, hence poor conditioning [2]. The case **eta=Inf** is interpreted as monopoles only, that is,

$$\{\Phi(\mathbf{x}, \mathbf{y}_j)\}_{j=1, \dots, N} \quad (9)$$

- Single- and double-layer potentials (**layerpot**) lying on a segment,

$$u(\mathbf{x}) = \mathcal{S}\sigma(\mathbf{x}) := \int_{\Gamma} \Phi(\mathbf{x}, \mathbf{y})\sigma(\mathbf{y})ds_{\mathbf{y}} \quad (10)$$

where  $ds$  is arclength, or double-layer potential (DLP) density

$$u(\mathbf{x}) = \mathcal{D}\tau(\mathbf{x}) := \int_{\Gamma} \frac{\partial\Phi(\mathbf{x}, \mathbf{y})}{\partial n_{\mathbf{y}}} \tau(\mathbf{y})ds_{\mathbf{y}} \quad (11)$$

where  $\Gamma$  is a segment (property **seg**), and  $\sigma$  and  $\tau$  are functions on  $\Gamma$  represented by their values at the segment's  $M$  quadrature points.

Regular and fractional Fourier-Bessels also may be *rescaled*, i.e. basis fuctions normalized to all have a similar-sized value at a given radius, in order to improve numerical stability and keep coefficient sizes similar. This radius is the basis property **rescale\_rad** (default is 0, no rescaling).

## 2.5 Basis set evaluation

Any basis set **b** can be *evaluated* at a column vector **x** of points  $\{\mathbf{x}_i\}_{i=1,\dots,M}$  by making from them a pointset object **z** = `pointset(x)`, then using one the following,

```
A = b.eval(z)
[A An] = b.eval(z)
[A Ax Ay] = b.eval(z)
```

If the set of basis functions in **b** are labeled  $\{\xi_j\}_{j=1,\dots,N}$ , then the matrix **A** has the *ij*-th element  $\xi_j(\mathbf{x}_i)$ . The second and third methods above also return first-derivatives of the basis functions, either the normal (**An**) or Cartesian (**Ax**, **Ay**) derivatives. As you might expect, for the normal derivative case only the pointset must also contain a list of normals, i.e. **z** = `pointset(x, nx)`.

All these dense basis set matrices are evaluated in a reasonably efficient manner (see Sec. 4). With a coefficient vector **c**, the linear combinations  $\sum_{j=1}^N c_j \xi_j(\mathbf{x}_i)$  are given by the product of **A** with the column vector. We chose to have the basis evaluation interface return these dense matrices, with cost  $O(NM)$ , since they will be stacked together as blocks of a larger matrix (see next section), and all our linear algebra is dense. However, for large-scale problems Fast Multipole method for computing a weighted sum of fundamental solutions would be better, for instance costing  $O(N \log N)$  when  $M = N$ . We do not implement these yet, since our focus is on small to medium problems.

## 3 Solution methods and problem classes

A problem class object contains as properties lists of (pointers to) basis sets, domains, and segments; see Fig. 2. These are collected<sup>6</sup> when domains are passed into a problem constructor (see tutorial for examples). The main method `fillbcmatrix` common to all problem classes fills the block-structured matrix **A** which maps the coefficient vector **co** (stacked column vectors of basis coefficients for all bases or degrees of freedom in the problem) to the boundary condition *inhomogeneity vector*. The latter is defined

---

<sup>6</sup>For instance, segments are extracted from the domains passed in, then duplicate segments are removed.

as the left-hand sides of (3), or (4) and (5) as appropriate, evaluated on the quadrature points of each segment of the problem and stacked together to give one column vector, when  $u$  is given by its basis representation in each domain.<sup>7</sup> The matrix **A** has block structure [2], where the order of the block columns matches that of the problem basis objects **bas**, and the order of the block rows matches that of the problem segments **segs**.

All problem classes contain various helper methods such as

- **updateN(N)** which sets the degree of all basis sets in the problem to  $N$  times their scale factor **bas.nmultiplier** (by default this is 1 for each basis object). Note that the total number of degrees of freedom is often several times larger than  $N$ .
- **fillquadwei** which fills the column vector of quadrature weights (in the same order as the inhomogeneity vector)
- **setoverallwavenumber(k)** which sets the problem wavenumber to  $k$ , and hence the wavenumbers in domain  $j$  to  $n_j k$  as in (2)
- **pointsolution(z)** which evaluates  $u$  at the pointset **z** (by checking into which domain each point in the pointset falls<sup>8</sup>), and utilities which build on this such as **gridsolution** and **showsolution**. Currently these routines discard the basis evaluation matrices used on the plotting grids—clearly this design choice could be improved when multiple problems at the same wavenumber are to be solved.
- methods which plot problem geometry rather than fields, such as **showbdry**, **showbasesgeom**, and **plot**. These are most useful for checking that the problem has been set up correctly.

In order to solve a problem, it is intended that the user create one of the convenient classes **bvp** or **scattering**, outlined below.

---

<sup>7</sup>This inhomogeneity vector is in fact multiplied by the square-root of the quadrature weight vector **sqrwei**, so that  $L^2$  and  $l_2$  norms become equal.

<sup>8</sup>Note that this is currently quite approximate, though usually adequate for plotting purposes. It is computed using an approximating polygon for the domain, with 50–100 or so sides per curved segment. See **domain.inside**.

### 3.1 Boundary-value problem class: `bvp`

This class is a subclass of `problem`, and solves the BVP from the first part of Sec. 1.2. All domains in the problem are passed to the constructor as a single array, as in `p = bvp(doms)`. When the solution is requested via `p.solvecoeffs`, the right-hand side vector `p.rhs` is first created which contains the right-hand sides of (3), or (4) and (5), for each problem segment, stacked in order, multiplied elementwise by the square-roots of the quadrature weights `p.sqrtwei`. This ‘left diagonal preconditioning’ of linear system by the quadrature weights means that the minimum  $L^2$  boundary error in solving the BVP is given by the minimum  $l_2$  residual norm in solving the linear system,

$$\mathbf{A} * \mathbf{co} = \mathbf{rhs}$$

This least-squares solution is done internally via `p.linsolve` which simply calls MATLAB’s dense solver (`backslash` command).

Once the least-squares coefficient vector has been found, the solution  $u$  is usually evaluated for plotting; see the tutorial for examples.

### 3.2 Scattering problem class: `scattering`

This class is a subclass of `bvp`. It solves the frequency-domain scattering problem mentioned in the middle of Sec. 1.2 and defined in Sec. 8 of the tutorial, in [2], or in many standard texts [3]. A BVP is solved for  $u_s$ , with right-hand side data `rhs` deriving from the incident wave field  $u_{inc}(\mathbf{x})$  which is  $\exp(ik\mathbf{n}_{inc} \cdot \mathbf{x})$  in each air domain and 0 in all remaining problem domains. The incident wave direction is  $\mathbf{n}_{inc} = (\cos \theta, \sin \theta)$ , and  $\theta$  is set by the `p.setincidentwave(theta)` command. More precisely, the right-hand side data for  $u_s$  is the negative of the boundary-condition mismatches that  $u_{inc}$  possesses at each boundary, so that the total field  $u = u_{inc} + u_s$  obeys homogeneous boundary or matching conditions on all segments.

This class includes a useful plotting routine `p.showthreefields` which uses `p.gridsolution` and `p.gridincidentwave` to compute and show separate subplots of  $u_{inc}$ ,  $u_s$  and  $u$ , as in Fig. 1.

## 4 Tweaks and test routines

By default **MPSPack** uses robust rather than fast math libraries. It is simple to switch to the following faster alternatives, but the user should be careful to check that the answers agree to their accuracy requirements.

1. Regular ( $J$ -type) Bessel functions, needed for **regfbbasis.eval**. Each **regfbbasis** object contains a property **besselcode** that may be changed to 'r' for Barnett's recurrence-relation implementation (2-3 times faster than MATLAB but not guaranteed relative accuracy in the deep evanescent region), or 'g' for MEX interface to GSL (nearly as fast as the recurrence version, and robust).
2. Hankel functions of order 0 and 1, needed for **mfsbasis.eval** and **layerpot.eval**. Both these basis objects have a property **fast** which may be 0 (MATLAB implementation), or 1 or 2 (a MEX interface to Greengard-Rokhlin's Fortran code, roughly 5 times faster than MATLAB). The user may change this property for each relevant basis object.
3. Point-in-polygon checking is implemented by a MEX interface to a C code which was found to be about 70 times faster than MATLAB. The implementation may be switched only by uncommenting lines in **@utils/inpolywrapper.m**

In the **test/** directory you will find some routines that we use to validate **MPSPack**. These are of variable quality, but contain coding examples that you may find useful, in particular,

- **testsegment.m** builds and plots line, arc, and analytic function segments.
- **testdomain.m** builds and plots nine domains of increasing complexity.
- **testbasis.m** plots all basis function types over a grid and uses the finite-difference grid approximation to validate their derivatives.
- **testbvp.m** shows the main steps for solution of a BVP on a variety of domains (FIX THIS CODE).
- **testscattering.m** shows the main steps for solution of a scattering problem on a variety of domains (FIX THIS CODE).

- `testinpolywrapper.m` validates the MEX interface to point-in-polygon checking.

There are several other more specific test routines in the directory that we do not list here.

## 5 Known bugs and limitations

Current bugs and issues are listed at the repository site,

<http://code.google.com/p/mpspack/issues/list>

Please alert the authors to any new bugs that you discover, including a description of how to reproduce the behavior, using this interface.

You may also contact the authors with suggestions via their email addresses given on the source page <http://code.google.com/p/mpspack>

Limitations of, and planned future improvements to, the software include:

- Two dimensions. Quadrature, decomposition into subdomains, and corner singularities, the main ideas upon which this toolbox is based, become much more complicated in three dimensions. Implementing 3D problems would be a major undertaking.
- Eigenvalue problems, one of the main reasons the authors became interested in particular solutions methods, are not yet implemented. For efficiency in symmetric domains, such as Bunimovich's stadium, this would include classes which symmetrize basis sets for single reflection,  $C_4$ , etc symmetry, by wrapping the calls to basis evaluations using reflection points.
- The necessity to purchase commercial MATLAB software. However, there are no free mathematical environments that we know of of comparable numerical versatility, plotting, and object-oriented capability. One idea would be to port to the promising-looking **SAGE** and **python** environment; get in touch if you would like to be involved.
- Some better tools for problem set-up are needed, such as:
  - Automatic meshing, based on complex approximation theory
  - make `domain.setbc` which uses one BC data function on all segments



- segment methods to create analytic interpolant function from boundary point data, enabling user to specify a segment using points on a curve, as in `FMMToolbox`.
- Checking of whether a point is inside a domain is approximate, currently based on an approximating polygon of typically 50-100 sides per segment object. Better would be to use the segment parametrization function in an iterative scheme, since this could give machine precision.
- We have not tested complex wavenumber problems, which have applications to conductive media. We expect that some of the methods, such as layer potentials and fundamental solutions, will carry over without modification.
- Graded-index media problems, using Airy and other particular solutions.
- Better automated ways to choose MFS charge points given a domain, based on [1].
- Solution evaluation on boundary segments via `segment.bdrysolution` which evaluates  $u$ ,  $u_n$  on one or other side of a boundary. This should then be used by `problem.fillbcmatrix`
- Incorporation of Fast Multipole Methods for evaluation with MFS and layer potentials, and iterative methods for second-kind layer potential formulations.
- Saving plotting-evaluation matrices for use with multiple right-hand sides. Using QR factors of `problem.A` matrix for rapid solution with multiple right-hand sides.
- Computation of far-field distribution in a scattering problem from MFS or layer potential representation of a radiating solution. This needs a variant of a multipole-to-local matrix of regular bessel functions.

Please contact the authors if you implement any of these and/or want to join the project!

## 6 Acknowledgments and credits

Almost all the code in `MPSPack` is written by Alex Barnett and Timo Betcke. The concept and need for the package came out of our work in eigenvalue and scattering problems, using global basis methods. We have been influenced by, and learned useful tricks from, the Schwarz-Christoffel toolbox by Toby Driscoll, the `FMMToolbox` by MadMax Optics, Inc., and the `chebfun` system by L. N. Trefethen, Z. Battles, T. Driscoll, R. Pachón, and R. Platte. Alex Barnett's work is supported by National Science Foundation grants DMS-0507614 and DMS-0811005. Timo Betcke's work is supported by XXX.

`MPSPack` is released under the GNU Public License, as follows:

Copyright (C) 2008, 2009, Timo Betcke, Alex Barnett

`MPSPack` is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 3 of the License, or (at your option) any later version.

`MPSPack` is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with `MPSPack`; if not, see <http://www.gnu.org/licenses>

As part of the distribution (sometimes in order to improve performance over native MATLAB libraries), we include codes written by others, as follows:

- `hank103.f` fast Hankel function computation, by V. Rokhlin and L. Greengard.
- `kapurtrap.m` Kapur-Rokhlin periodic quadrature weights, by Z. Gambutas.
- `clencurt.m` and `gauss.m`, quadrature nodes and weights, by L. N. Trefethen [6].

- `polypn.c` algorithm to check if a point is in a polygon, Copyright (C) 1970-2003, Wm. Randolph Franklin.
- `inpoly.m` algorithm to check if a point is in a polygon, Darren Engwirda, 2005-2007.
- `copy.m` makes deep copy of MATLAB handle object, Doug M. Schwarz, 6/16/08.

## References

- [1] A. H. BARNETT AND T. BETCKE, *Stability and convergence of the Method of Fundamental Solutions for Helmholtz problems on analytic domains*, J. Comput. Phys., 227 (2008), pp. 7003–7026.
- [2] —, *An exponentially convergent non-polynomial finite element method for scattering from polygons*, 2009. in preparation.
- [3] D. COLTON AND R. KRESS, *Inverse acoustic and electromagnetic scattering theory*, vol. 93 of Applied Mathematical Sciences, Springer-Verlag, Berlin, second ed., 1998.
- [4] M. GALASSI *et al.*, *GNU Scientific Library Reference Manual*. <http://www.gnu.org/software/gsl>.
- [5] THE MATHWORKS, INC., *MATLAB software*, Copyright (c) 1984–2009. <http://www.mathworks.com/matlab>.
- [6] L. N. TREFETHEN, *Spectral methods in MATLAB*, vol. 10 of Software, Environments, and Tools, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2000.