MPSpack user manual

Alex Barnett* and Timo Betcke†

June 4, 2010

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

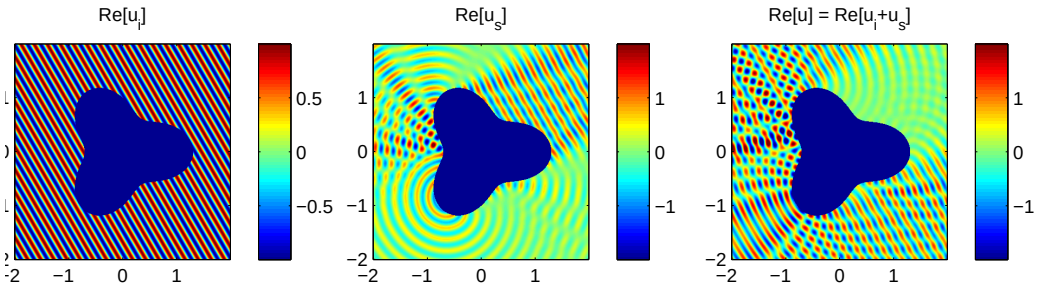

Figure 1: Sound-hard (homogeneous Neumann boundary condition) scattering from a smooth obstacle using a fundamental solutions basis in `MPSpack`. Left: incident wave. Center: scattered wave. Right: total solution field.

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. 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,

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

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 simple 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 three 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/tags/1.0 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-1.0.tar.gz
   ```

   This creates the directory `mpspack` containing the toolbox.

3. Get a zip-compressed archive from

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

There are some optional fast basis and other math libraries (C and Fortran with MEX interfaces), which although not needed for `MPSpack` to work, should improve efficiency (see Section 4). These can be compiled in a UNIX environment as follows (we have not yet tried them in other operating systems): from the `mpspack` directory type `make`. You may first need to adjust the library locations in `make.inc`, as explained in that file (If GNU Scientific Library [4] is not installed, you will need to remove the lines in `@utils/Makefile` which compile codes using GSL 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 start MATLAB and use `MPSpack` !

1.2 What problems can `MPSpack` solve?

Here we give a general framework (for examples see [1, 2]). Let $\Omega_j \subset \mathbb{R}^2$, $j = 1, \ldots, D$ 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 $u : \Omega \to \mathbb{C}$. In each domain we have,

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

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

$$a_j u + b_j u_n \;=\; f_j \qquad \text{on } \Gamma_j \;, \tag{3}$$

where $a_j, b_j \in \mathbb{C}$ are complex numbers (currently; in future they may be functions on the boundary), and $f : \Gamma_j \to \mathbb{C}$ are (possibly identically zero) driving functions. u_n is short for $\boldsymbol{n}\cdot\nabla u$, the normal derivative on the boundary.[1] 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,

$$a^+_{ij} u^+ + a^-_{ij} u^- \;=\; f_{ij} \qquad \text{on } \Gamma_{ij} \;, \tag{4}$$

$$b^+_{ij} u^+_n + b^-_{ij} u^-_n \;=\; g_{ij} \qquad \text{on } \Gamma_{ij} \;, \tag{5}$$

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 Γ_{ij} from its positive (negative) normal side.

We assume all the boundaries Γ_j and Γ_{ij} are piecewise smooth, and each smooth piece we will build from one or more *segments*. If Ω_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,

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

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

[1] 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 Γ_j.

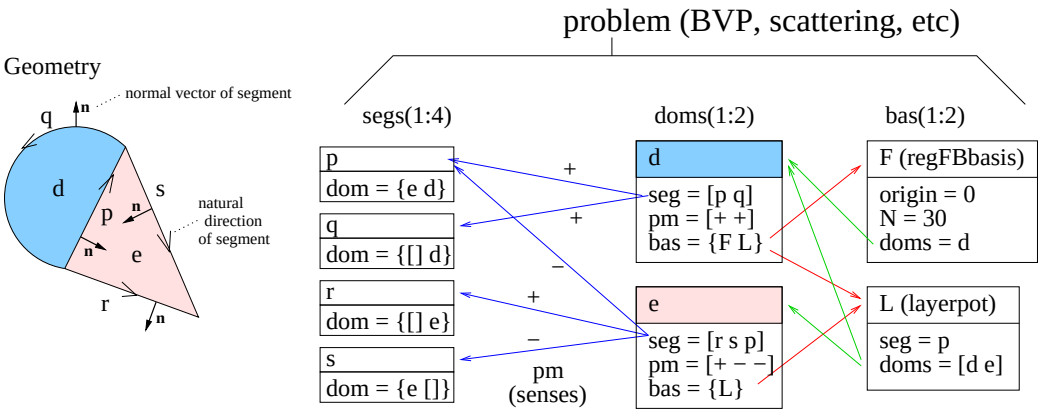

Figure 2: Relationship between segments, domains, and basis sets in a simple example. The physical geometry is shown on the left. There is a regular Fourier-Bessel basis in domain `d`, and a layer-potential density on segment `p` which affects both domains `d` and `e`. The corresponding code objects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

2. build domains using various of these segments as their boundaries
3. set up (in)homogeneous boundary or matching conditions on each segment
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)[2] 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.[3] 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

[2] A *property* is a variable stored inside the object instance, just like a field `a.b` inside a struct `a`.

[3] 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`.

z' is reasonable.[4] 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 = {[] []}`.

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* (± 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. It is a helpful exercise for the reader to check that 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

[4] 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.

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`.[5] 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

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

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

  where (r,θ) 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,

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

  where $w := z - z_0$.

- Fractional-order (wedge of angle π/ν) 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,θ) 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

[5] In fact, `doms` may be a row vector of more than one domains, but currently the only command which can create this is `segment.addinoutlayerpots`. This adds a 'two-sided' layer potential which influences domains on both its sides, generally with different wavenumbers. Therefore we discuss only the simpler case of one affected domain.

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,

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

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

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

  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,

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

  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

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

  Note that the normal derivative of Φ 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,

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

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

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

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

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

  where Γ is a segment (property `seg`), and σ and τ are functions on Γ 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 $\{\mathbf{x}_i\}_{i=1,\ldots,M}$ 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 $\{\xi_j\}_{j=1,\ldots,N}$, 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 $\boldsymbol{c}$, 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[6] 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 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.[7] 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 `p.sqrtwei` the row vector of (square-roots 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)

[6] For instance, segments are extracted from the domains passed in, then duplicate segments are removed.

[7] This inhomogeneity vector is in fact multiplied elementwise by the square-root of the quadrature weight vector `sqrtwei`, so that L^2 and l_2 norms become equal.

- `pointsolution(z)` which evaluates u at the pointset `z` (by checking into which domain each point in the pointset falls[8]), 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.

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 square-root of quadrature weights is chosen so 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,

```
A*co = 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

[8]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`.

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 $\exp(ik\boldsymbol{n}_{inc}\cdot\mathbf{x})$ in each air domain and 0 in all remaining problem domains. The incident wave direction is $\boldsymbol{n}_{inc} = (\cos\theta, \sin\theta)$, and θ 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 the 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.
- `testscattering.m` shows the main steps for solution of a scattering problem on a variety of domains.
- `testinpolywrapper.m` validates the MEX interface to point-in-polygon checking.
- `testdielscatrokh.m` calculates a transmission scattering problem using Rokhlin's hyper-singular cancelling scheme, via setting up layer potentials which each affect *two* domains.

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. High-order quadrature for self-evaluation of layer-potentials.
- 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 J-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 Engineering and Physical Sciences Research Council Grant EP/F06795X/1.

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

```
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. Gimbutas.
- `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.

A Class Descriptions

`POINTSET` Create a pointset object with locations and normal vectors as complex numbers.

A pointset is simple object containing a list of points in 2D, plus possibly associated normal directions. It is used to store quadrature points on a segment, and also evaluation point lists. Coordinates are stored as complex numbers.

Constructors

`p = POINTSET()` creates an empty object.

`p = POINTSET(x)` where x is m-by-1 array, creates pointset with m points, where the ith point has Cartesian coordinates (Re x(i), Im x(i)).

`p = POINTSET(x, nx)` where x is above and nx has same size as x, creates pointset with coordinates x (interpreted as above) and associated normals nx (interpreted in the same way). The Euclidean lengths of the vectors in nx are not required to be, nor changed to, unity.

See also: `POINTSET/plot`, `SEGMENT` which builds on `POINTSET`

Methods

none

`SEGMENT` create segment object

Constructors

`s = SEGMENT(M, [xi xf])` creates a line segment object from xi to xf, both complex numbers.

`s = SEGMENT(M, [xc R ti tf])` creates a circular arc segment with center xi, radius R and angles from ti to tf. The order is important. If tf¿ti the orientation is counter-clockwise, otherwise clockwise.

`s = SEGMENT(M, Z, Zp)` creats an analytic curve given by the image of the analytic function Z:[0,1]-¿C. Zp must be the derivative of Z. Z and Zp are function handles.

`s = SEGMENT(M, Z, Zp, Zpp)` works as above but also takes the second derivative Z" of Z. This is useful for layer potentials.

`s = SEGMENT(M, p, qtype)` where p is any of the above, chooses quadrature type
`qtype = 'p'`: periodic trapezoid (appropriate for periodic segments, M pts)
`'t'`: trapezoid rule (ie, half each endpoint, M+1 pts)
`'c'`: Clenshaw-Curtis (includes endpoints, M+1 pts)
`'g'`: Gauss (takes $O(M^3)$ to compute, M pts)

If M is empty, a default value of 20 is used.

`s = SEGMENT()` creates an empty segment object.

See also: `POINTSET`, `segment/PLOT`

Methods

`requadrature(segs, M, qtype)` change a segment's quadrature scheme or number of points

`setbc(seg, pm, a, b, f)` set a (in)homogeneous boundary condition on a

segment

`setmatch(seg, a, b, f, g)` set (in)homogeneous matching conditions across a segment

`newseg = scale(seg, fac)` rescale (dilate) a segment (or list) about the origin

`newseg = translate(seg, a)` translate a segment (or list of segments)

`newseg = rotate(seg, t)` rotate a segment (or list of segments) about the origin

`newseg = reflect(seg, ax)` reflect a segment (or list of segments) about x or y axis

`disconnect(segs)` disconnect a segment or segment list from any domains

`h = plot(s, pm, o)` plots a directed segment on current figure, using its quadrature pts

`s = polyseglist(M, p, qtype)`
`s = polyseglist(M, p)`
(Static Method) create closed list of segment objects from CCW polygon vertices

`s = radialfunc(M, fs)` (Static Method) closed segment from radial function r=f(theta) and derivatives

`s = smoothstar(M, a, w)` (Static Method) single-freq oscillatory radial function closed segment

`[a b] = dielectriccoeffs(pol, np, nm)` (Static Method) give a and b coeff pairs (1-by-2) from refractive indices

`DOMAIN` create an interior/exterior domain possibly with excluded subregions

A domain is an ordered connected list of segments defining the exterior boundary, with zero or more ordered connected lists of segments defining the boundaries of any interior excluded regions. If the exterior boundary is empty the domain from which interior regions are possibly excluded is taken to be the whole plane, resulting in an unbounded exterior domain.

Constructors

`d = DOMAIN(s, pm)` creates an interior domain whose boundary is the list of handles of segment objects s, using the list of senses pm (each element is the number +1 or -1). A warning is given if the segments do not appear to connect up at corners. Normals should all point outwards, ie away from the domain, otherwise a warning is given. If pm has only 1 element, it will be duplicated as necessary to match the size of s.

`d = DOMAIN()`
`d = DOMAIN([], [])`
creates an exterior domain equal to the whole plane $\mathbb{R}^2$.

`d = DOMAIN([], [], si, pmi)` creates an exterior domain equal to the whole plane minus an excluded region whose (non-intersecting) boundary is given by segment list si and sign list pmi (which have the same format as s, pm above). If si and pmi are instead cell arrays of segment lists and corresponding sign lists, each cell element is an excluded region. As above, warnings are given if intersections or incorrect normals are found.

`d = DOMAIN(s, pm, si, pmi)` combines the above features, creating a bounded domain with excluded region(s).

See also: `SEGMENT`, `domain/PLOT`

Methods

`norout = normalscheck(d)` check that senses of normals point away from a domain

`i = inside(d, p)` return true (false) for points inside (outside) a domain

`x = x(d)` return column vector of quadrature points on a domain boundary

`nx = nx(d)` return column vector of unit outward normals on a domain boundary

`w = w(d)` return row vector of quadrature weights for a domain boundary

`bb = boundingbox(d)` return bounding [xmin xmax ymin ymax] for interior domain

`xc = center(d)` return center x (as complex number) of bounding box of domain

`diam = diam(d)` approximate diameter of an interior domain

`[zz ii gx gy] = grid(d, dx)` make grid covering interior domain, or some of exterior domain

`deletecorner(d,j)` remove a given corner number from a domain

`Nf = Nf(d)` total number of basis functions (dofs) associated with domain

`clearbases(d)` remove all basis set associations from a domain

`showbasesgeom(d)` show geometry of basis objs

`addconnectedsegs(d, s, pm, o)` append a domain's params, corners given conn seg list

`addregfbbasis(d, origin, N, opts)` create a regular Fourier-Bessel basis object in a domain

`addnufbbasis(d,origin,nu,offset,branch,N,opts)` create a fractional-order Fourier-Bessel basis set

`addcornerbases(d, N, opts)` add irreg Fourier-Bessel basis to each corner

of a domain

`addrpwbasis(d, N, opts)` create a real plane wave basis object in a domain

`addmfsbasis(d, varargin` Add an MFS (fundamental solutions) basis to a domain

`b = addlayerpot(d, a, segs, opts)` create a layer-potential basis set object in a domain

`[A A1 A2] = evalbases(d, p, opts)` evaluate all basis sets in a domain object, on a pointset

`h = plot(d, o)` show domain (or list of domains) on current figure

`setrefractiveindex(doms, n)` sets n for a list of domains

`v = approxpolygon(seg, pm)` (Static Method) stack together approximating polygon vertices of segment list

`h = showsegments(seg, pm, o)` plot signed segment list to current figure (domain helper)

`x = stackquadpts(seg, pm)` helper routine, ordered quad pts from signed connected seg list

`h = showdomains(dlist, opts)` plot all domains on current figure using a color for each

`BASIS (ABSTRACT)` Abstract class that defines the interfaces which are common for all basis objects

Methods

`[A, A1, A2] = eval(b, pts)` (abstract) evaluate a basis on a set of points

`Nf = Nf(b, opts)` (abstract) method returning number of degrees of freedom

`updateN(b,N)` set the number of basis functions

`k = k(b, opts)` looks up the basis wavenumber

`MFSBASIS : BASIS` create a fundamental solutions (MFS) basis set.

Constructors

`b = MFSBASIS(Z, N, opts)` creates an MFS basis using charge points at Z(t), where t are equally distributed in [0,1]. Z is a handle to a function. If the charge curve is to be closed, then Z must be 1-periodic. N may be empty.

`b = MFSBASIS(Z, Zp, N, opts)` is as above. But Zp additionally specifies the derivative of Z. This is necessary if dipole-type charges (double layer potentials) are used. N may be empty.

`b = MFSBASIS(p, opts)` describes MFS charge points in terms of a point set object p. If double layer potentials are used then p also needs to contain normal directions.

`b = MFSBASIS(s, N, opts)` uses the function handles from segment s to choose N charge points. It is equivalent to b = MFSBASIS(s.Z, s.Zp, N, opts).

In each of the above, opts is an optional structure with optional fields:
`opts.eta` - (inf) If eta=inf use single layer potential MFS basis. For eta $\neq$ inf use linear combination of double and single layer potential MFS basis
`opts.fast` - (0) Hankel evaluation method (0=matlab; 1,2=faster)
`opts.real` - (false) If true, use real part of Hankel functions only
`opts.tau` - (0) Creates charge curve Z(t+i.tau) rather than Z(t)

Methods

`showgeom(bas, opts)` plots geometry of MFS basis charge points on current figure

`NUFBBASIS : BASIS` create a fractional-order Fourier-Bessel basis set

Constructors

`b = NUFBBASIS(origin, nu, offset, branch, N)` creates a basis of fractional-order Fourier-Bessel functions appropriate for expansion of the Helmholtz equation in a wedge of angle pi/nu ¿ 0. The orders are nu*(1:N) (for sine angular functions) or nu*(0:N) (for cosine angular functions). The other arguments are:
`origin`: The origin of the Fourier-Bessel functions (as complex number)
`offset`: The direction that corresponds to the angular variable theta=0
`branch`: Direction of the branch cut (point on the unit circle);
it must not point into the affected domain.
`N` : degree of the basis set

The wedge lies counterclockwise from offset, spanning pi/nu in angle. As with all basis types, the wavenumber k is determined by that of the affected domain. k=0 gives fractional-power Laplace equation solutions.

`b = NUFBBASIS(origin, nu, offset, branch, N, opts)` as above but permits optional arguments to be selected. Currently supported is
`opts.type`:
's' Create basis of Fourier-Bessel sine fct.
'c' Create basis of Fourier-Bessel cosine fct.
'cs' Create basis of Fourier-Bessel cosine and sine functions (default)
The reason for having 'cs' rather than using separate 's' and 'c' objects is a factor of 2 in speed: the set of Bessel evaluations is reused.

See also: DOMAIN/ADDNUFBBASIS

Methods

`h = showgeom(nufb, opts)` show corner wedge location, geometry and branch cut

`sc = Jrescalefactors(nufb, n)` given list of orders, return FB J-rescaling factors

`REGFBBASIS : BASIS` create a regular Fourier-Bessel basis set

Constructors

`b = REGFBBASIS(origin, N)` creates a regular Fourier-Bessel basis object with given origin, and order N. As with all basis types, the wavenumber k is determined by that of the affected domain, as follows:

$k = 0$ gives harmonic polynomials,
$\{1, \mathrm{Re}z, \mathrm{Re}z^2, ..., \mathrm{Re}z^N, \mathrm{Im}z, ..., \mathrm{Im}z^N\}$ (real case)
or $\{z^{-N}, z^{-N+1}, ..., z^N\}$ (complex case)
$k > 0$ gives generalized harmonic polynomials, $\{J_n(kr)\cos(n\theta)\}$ $n = 0, .., N$ and $\{J_n(kr)\sin(n\theta)\}$ $n = 1, .., N$ (real case)
or $\{J_n(kr)\exp(1in\theta)\}$ $n = -N, .., N$ (complex case)

`b = REGFBBASIS(origin, N, opts)` is as above, with options:
`opts.real`: if true (default), use real values (cos, sin type), otherwise use complex exponentials with orders -N through N.
`opts.rescale_rad`: if positive, rescales the basis coefficients i.e. columns of evaluation matrix, such that the value at radius rescale_rad is O(1). (default is 0, giving no rescaling)
`opts.besselcode`: math library to use for J Bessel evaluation ($k > 0$)
`'r'` use downwards-recurrence in Matlab, fast, but relative accuracy not guaranteed.
`'m'` use Matlab's built-in besselj, is slower (default).
`'g'` use GNU Scientific Library via MEX interface, fast.

See also: `DOMAIN/ADDREGFBBASIS`

Methods

`showgeom(regfb, opts)` plot regular FB basis set geometry info (just origin now)

`sc = Jrescalefactors(regfb, n)` given list of orders, return FB J-rescaling factors

`RPWBASIS : BASIS` create a real (as opposed to evanescent) plane wave basis set

Constructors

`b = RPWBASIS(N)` creates a real plane wave basis object, with N directions θ_j equally spaced in $(0, \pi]$, i.e. $\theta_j = \pi j/N$. As with all basis types, the wavenumber k is determined by that of the affected domain. Basis functions are:
$\{\cos(kn_j z)\}\ j = 1, .., N$ and $\{\sin(kn_j.z)\}\ j = 1, .., N$ (real case)
$\{\exp(ikn_j z)\}\ j = 1, .., N$ and $\{\exp(-ikn_j z)\}\ j = 1, .., N$ (complex case)
where the direction unit vectors are $n_j = (\cos\theta_j, \sin\theta_j)$

If k=0 a warning is produced since this basis does not have a useful limiting form as k tends to zero (regfbbasis should be used instead).

`b = RPWBASIS(N, opts)` does the same, except allowing user options:
`opts.real`: if true, real case (cos/sin type), otherwise complex case.

See also: `DOMAIN/ADDRPWBASIS`

Methods

`showgeom(b, opts)` plot real plane wave basis set geometry information

`LAYERPOT : BASIS` create a layer potential basis set on a segment

Constructors

`b = LAYERPOT(seg, a, opts)` where `a = 'single'` or `'double'` creates a layer potential basis object with density on segment seg, with options:
`opts.real`: if true, real valued (Y_0), otherwise complex (H_0 outgoing).
`opts.fast`: if 0 use matlab Hankel, 1 use fast fortran Hankel, 2 faster.
If a is instead a 1-by-2 array, then a mixture of a(1) single plus a(2) double is created, useful for Brakhage, Werner, Leis & Panich type combined representations. As usual for basis sets, the wavenumber k is determined by that of the affected domain.

Note that the discretization of the layerpot is given by that of the seg, apart from periodic segments where new quadrature weights may be used.

See also: `DOMAIN/ADDLAYERPOT`,

Methods

`showgeom(b, opts)` show discrete points of segments

`[A Sker] = S(k, s, t, o)` (Static Method) double layer potential discretization matrix for density on a segment

`[A Dker_noang cosker] = D(k, s, t, o)` (Static Method) double layer potential discretization matrix for density on a segment

`[A Sker Dker_noang] = T(k, s, t, o)` (Static Method) deriv of double layer potential discr matrix for density on a segment

`A = localfromSLP(s, Jexp)` (Static Method) return matrix taking SLP density values on seg to J (local) exp

`A = localfromDLP(s, Jexp)` (Static Method) return matrix taking DLP density values on seg to J (local) exp

`A = fundsol(r, k)` (Static Method) return matrix of fundamental solutions

`[B radderivs] = fundsol_deriv(r, cosphi, k, radderivs)` (Static Method)
return matrix of normal derivatives of fundamental solutions

QUADR static class of quadrature rules

Methods

`[x w] = peritrap(N)` (Static Method) trapezoid quadrature rule for periodic functions

`[x w] = traprule(N)` (Static Method) quadrature points and weights for composite N+1-point trapezoid rule

`[x,w] = gauss(N)` (Static Method) nodes x (Legendre points) and weights w for Gauss quadrature

`[x w] = clencurt(N)` (Static Method) nodes x (Chebyshev points) and weights w for Clenshaw-Curtis quadrature

`[x,w,cs,ier]=kapurtrap(n,m)` (Static Method) nodes and weights of n-point corrected trapezoidal quadrature formula on [0,1]

`Rjn = kress_Rjn(n)` (Static Method) Kress weights

`D = perispecdiffrow(N)` (Static Method) row 1 of N-pt spectral 2pi-periodic differentiation matrix

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.