deltaBEM: tools and toys for 2d BIE

Coordinator	:	Francisco–Javier Sayas (University of Delaware)
Developers	:	Sijiang Lu (University of Delaware, 2011–2013)
		Víctor Domínguez (Universidad Pública de Navarra, Spain)
		Douglas Freeman (University of Delaware, 2013)
		Andrea Carosso (University of Delaware, 2013)
		Matthew Hassell (University of Delaware, 2013–)
		Tonatiuh Sanchez–Vizuet (University of Delaware, 2013–)

Last update: March 29, 2016

Contents

Chapter 1

Geometry

1.1 Continuous and discrete geometries

Simple closed curves

A simple closed curve in the plane is given by a parametrization $\mathbf{x} : \mathbb{R} \to \Gamma \subset \mathbb{R}^2$, that is a 1-periodic function. *We assume that the parametrization gives a positive orientation of the curve*, so that normal vectors (as defined below) always point outwards. The vector

$$\mathbf{n}(t) = (x_2'(t), \, -x_1'(t))$$

is the non-normalized normal vector at $\mathbf{x}(t)$. For the sake of coding we will need the function $\mathbf{x}(t)$ and its first derivative $\mathbf{x}'(t)$.

A discrete version of the geometry

The discrete version of a curve is based on two parameters: an integer N will give the level of refinement, a number $\varepsilon \in \mathbb{R}$ will give a starting point for counting the parameter. For all effects, ε is defined modulo integers: a discrete grid for ε and the discrete grid for $\varepsilon + 1$ contain the same elements, although they are numbered in a different way.
We start by choosing a positive integer N, $h := 1/N$, and $\varepsilon \in \mathbb{R}$. We next choose parametric coordinates at

$$t_i^\varepsilon := (i + \varepsilon)\, h \qquad i \in \mathbb{Z}_N := \{1, \ldots, N\}, \qquad (i \in \mathbb{Z}).$$

The discrete geometry is described by several elements: they will be stored for values $i = 1, \ldots, N$

- breakpoints $\mathbf{b}_i^\varepsilon := \mathbf{x}(t_i^\varepsilon - \frac{h}{2})$
- midpoints $\mathbf{m}_i^\varepsilon := \mathbf{x}(t_i^\varepsilon)$
- normal vectors $\mathbf{n}_i^\varepsilon := h\mathbf{n}(t_i^\varepsilon)$
- We also need a function to find the next index:

$$n(i) := \begin{cases} i+1, & 1 \le i \le N-1, \\ 1, & i = N. \end{cases}$$

  This function will be very useful when we deal with collections of curves.

- Finally we need a pointer to where each of the connected components of the geometric object starts. For multiple curves, this index will tell us where each of the components starts within the data structure. *For collections of curves, this feature requires that all nodes of a curve are grouped together.*

Remark. In the background, there is a Boundary Element mesh. We can consider the elements

$$\Gamma_i^\varepsilon := \{\mathbf{x}(t) \,:\, t_i^\varepsilon - \tfrac{h}{2} < t < t_i^\varepsilon + \tfrac{h}{2}\}.$$

What we call length is just a simple approximation of the value of $|\Gamma_i^\varepsilon|$. The element Γ_i^ε is delimited by the break points $\mathbf{b}_i^\varepsilon$ and $\mathbf{b}_{n(i)}^\varepsilon$. The discrete methods we will be using consider piecewise constant functions on this mesh as a non-conforming approximation of the space $H^{1/2}(\Gamma)$. We also consider the space spanned by the Dirac deltas $\delta(\,\cdot\, - \mathbf{m}_i^\varepsilon)$, as a non-conforming approximation of $H^{-1/2}(\Gamma)$.

The data structure. A fully discrete mesh is stored in a single data structure. If `g` represents a discrete geometry with N elements, then

- `g.midpt` is an $N \times 2$ matrix with the coordinates of the points $\mathbf{m}_i^\varepsilon$ stored by rows,
- `g.brkpr` is an $N \times 2$ matrix with the coordinates of the points $\mathbf{b}_i^\varepsilon$ stored by rows,
- `g.normal` is an $N \times 2$ matrix with the vectors $\widehat{\boldsymbol{\nu}}_i^\varepsilon$ sotred by rows,
- `g.next` is a $1 \times N$ row vector with the next function; it is a permutation of the numbers $\{1, \ldots, N\}$.
- `g.comp` is a $1 \times N_{\rm comp}$ vector pointing at the index number of the first point of each connected component.

In the following example, we can observe a mesh with 10 points corresponding to a single closed obstacle, as can be seen in the `g.next` vector.

```
g =
      midpt: [10x2 double]
      brkpt: [10x2 double]
     normal: [10x2 double]
       next: [2 3 4 5 6 7 8 9 10 1]
       comp: 1
```

There's an additional optional field `g.parity` that is used only in the case of open arcs. (See Chapter 4.)

The discrete Calderón Calculus uses three samples of the geometry, corresponding to $\varepsilon \in \{0, 1/6, -1/6\}$. The central one will be called the **main grid**, and the other ones will be called the **companion grids**. At the time of discretization of integral operators they will be often denoted `g`, `gp`, `gm` respectively.

Merging geometries

The function `joinGeometry` picks two discrete geometries and places them one after another. This is meant to be used for discretization of separate curves. This process can be applied to already merged geometries, so that to join three discretized closed curves, we can first join the first two and then add the last one to the group. (The function `merge` carries out the task of multiple merges.)
The only places to be careful at are the `g.next` field (next index function) as well as `g.comp`. The `g.parity` field is taken into account at the time of merger. If it is not present in either of the merging geometries, it is kept non–existent. Explanations about the merging process for this field are postponed to Chapter 4.

```
function g = joinGeometry(g1,g2)

% function g = joinGeometry(g1,g2)
% Input
```

```
%            g1 : 1st geometry
%            g2 : 2nd geometry
% Output
%             g : merged geometry
% Last Modified: August 2, 2013

N=size(g1.midpt,1);
g.midpt = [g1.midpt; g2.midpt];
g.brkpt = [g1.brkpt; g2.brkpt];
g.normal = [g1.normal; g2.normal];
g.next = [g1.next,N+g2.next];
g.comp = [g1.comp,N+g2.comp];

if (¬isfield(g1,'parity'))&&(¬isfield(g2,'parity'))
    return
elseif (isfield(g1,'parity'))&&(isfield(g2,'parity'))
    g.parity = blkdiag(g1.parity,g2.parity);
elseif ¬isfield(g1,'parity')
    g1.parity = sparse(size(g1.midpt,1),size(g1.midpt,1));
else
    g2.parity = sparse(size(g2.midpt,1),size(g2.midpt,1));
end
g.parity = blkdiag(g1.parity,g2.parity);

return
```

This is an example of how to create and merge two discretized ellipses:

```
>> g1=ellipse(10,0,[1 2],[0 0]);
>> g2=ellipse(10,0,[1 2],[3 3]);
>> g=joinGeometry(g1,g2)
g =
      midpt: [20x2 double]
      brkpt: [20x2 double]
     normal: [20x2 double]
       next: [2 3 4 5 6 7 8 9 10 1 12 13 14 15 16 17 18 19 20 11]
       comp: [1 11]
```

Unpacking merged geometries

If a domain contains several components, they can be unpacked to separate data structures using **unpackGeometry**. The input is a single sampled geometry. The output is a cell array whose elements are the separated data structures (with original numbering for the **g.next** field and component numbers).

```
function UG = unpackGeometry(MG)

% UG = unpackGeometry(MG)
% Input:
%       MG    :    merged geometry (one structure array)
% Output:
%       UG    :    cell array (several elements; one structure array per
%                  element)
%
% Last modified: May 14, 2014

K = length(MG.comp);
UG{K} = [];
MG.comp=[MG.comp length(MG.next)+1];

for i=1:K
    next=MG.next(MG.comp(i):MG.comp(i+1)-1)-MG.comp(i)+1;
    UG{i} = struct('midpt',MG.midpt(MG.comp(i):MG.comp(i+1)-1,:),...
```

```
                   'brkpt',MG.brkpt(MG.comp(i):MG.comp(i+1)−1,:),...
                   'normal',MG.normal(MG.comp(i):MG.comp(i+1)−1,:),...
                   'next',next,...
                   'comp',1);
    if ¬isfield(MG,'parity')
        continue
    elseif MG.parity(MG.comp(i),MG.comp(i))==1
            UG{i}.parity = speye(length(UG{i}.midpt));
            UG{i}.parity = UG{i}.parity−UG{i}.parity(end:−1:1,:);
    end
end
return
```

Restriction to subgeometries

The following function picks a composite sampled geometry with components $\Gamma_1, \ldots, \Gamma_K$ and a vector $\mathbf{v} = (v_1, \ldots, v_L)$ with $v_j \in \{1, \ldots, K\}$ and returns a vector with the indices of points corresponding to $\Gamma_{v_1}, \ldots, \Gamma_{v_L}$ in that order. The function also returns the restriction matrix. If $N_j l$ is the number of discrete elements of Γ_j, the restriction matrix is an $(N_{v_1} + \ldots + N_{v_L}) \times (N_1 + \ldots + N_K)$ sparse matrix with only one 1 per row.

```
function [indices,restr] = selectComponents(g,v)

% [indices,restr] = selectComponents(g,v)
% Input:
%       g   :   joined geometry composed of several geometries
%       v   :   a row vector with numbers corresponding to geometries
% Output:
%       indices  :  a row vector containing the indices corresponding to
%                   the geometries selected
%       restr    : restriction matrix
% Last Modified: August 2 2013

indices = [];
N = length(g.midpt);
g.comp=[g.comp, N+1];
for i = v
    indices = [indices g.comp(i):(g.comp(i+1)−1)];
end
restr=sparse(1:length(indices),indices,1,length(indices),N);
return
```

1.2 Geometric utilities

Affine transformations

This function generates the discrete geometry corresponding to an affine transformation

$$\mathbf{y} = \mathrm{A}\mathbf{x} + \mathbf{b}$$

where $\det(\mathrm{A}) > 0$. The case $\det(\mathrm{A}) < 0$ requires renumbering nodes, since it generates negatively oriented curves. It is not supported at the current stage of the code.

```
function gnew=affine(g,A,b)

% gnew = affine(g,[a11 a12; a21 a22],[b2;b2])
% Input:
%     g    : discrete sampled geometry
```

```
%     A    : 2 x 2 matrix
%     b    : 2 x 1 vector
% Output:
%     gnew : discrete sampled geometry
%            corresponding the the affine transformation xnew=A*x+b
% Last modified: August 2, 2013

if det(A)≤0
    disp('det A ≤0 non supported')
    gnew=g;
end

gnew.midpt=bsxfun(@plus,b',g.midpt*A');
gnew.brkpt=bsxfun(@plus,b',g.brkpt*A');
gnew.normal=g.normal*([0 -1; 1 0]*A'*[0 1;-1 0]);
gnew.next=g.next;
gnew.comp=g.comp;
if isfield(g,'parity')
    gnew.parity = g.parity;
end

return
```

Given a discretized geometry g, and a collection of centers

$$\begin{bmatrix} c_1^x & c_1^y \\ \vdots & \vdots \\ c_N^x & c_N^y \end{bmatrix}$$

we use affine transformations to **translate** the original geometry g by means of the transformation $\mathbf{x} \mapsto \mathbf{x} + (c_j^x, c_j^y)^\top$, and we merge the resulting geometries in a single data structure.

```
function L = latticeGeometry(g,centers)

% function L = latticeGeometry(g,centers)
% Input:
%       g       :     the geometry from which the lattice will be made
%       centers :     N by 2 matrix of desired centers for the geometries
% Output:
%       L       :     a single structure of the combined geometries
% Last Modified: July 11, 2013

N = length(centers(:,1));
Larray = [];
for i = 1:N
    Larray = [Larray affine(g,eye(2),centers(i,:)')];
end
l = length(Larray);
L = Larray(1);
for i = 2:l
    L = joinGeometry(L,Larray(i));
end
return
```

Mirroring geometries

Suppose we are given discrete geometric structure with the additional requirement that all points lie on one side of the horizontal axis. The function `mirror` produces a mirror copy on the other side of the axis. Similarly `mirrorLR` takes a geometric structure corresponding to a curve lying on one side of the vertical axis and mirrors it across this axis. Note that the only difficulty of this process is the correct renumbering of the nodes, so that the positive orientation is preserved.

```
function gnew = mirror(g)

% gnew = mirror(g)
%
% Input :
%
%    g   : ΔBEM geometry
%
% Output:
%
%    gnew : geometry mirrored across the x-axis
%
% Last Modified: May 13, 2014

g.brkpt=g.brkpt(g.next,:);

N=size(g.midpt,1);

gnew.midpt=[g.midpt(:,1),-g.midpt(:,2)];
gnew.brkpt=[g.brkpt(:,1),-g.brkpt(:,2)];
gnew.comp=g.comp;
gnew.normal(:,1)=g.normal(:,1);
gnew.normal(:,2)=-g.normal(:,2);

prev=zeros(1,N);

prev(g.next)=1:N;
gnew.next=prev;

return
```

```
function gnew = mirrorLR(g)

% gnew = mirrorLR(g)
%
% Input :
%
%    g   : ΔBEM geometry
%
% Output:
%
%    gnew : geometry mirrored across the y-axis
%
% Last Modified: May 23, 2014

g.brkpt=g.brkpt(g.next,:);

N=size(g.midpt,1);

gnew.midpt=[-g.midpt(:,1),g.midpt(:,2)];
gnew.brkpt=[-g.brkpt(:,1),g.brkpt(:,2)];
gnew.comp=g.comp;
gnew.normal(:,1)=-g.normal(:,1);
gnew.normal(:,2)=g.normal(:,2);

prev=zeros(1,N);

prev(g.next)=1:N;
gnew.next=prev;

return
```

1.3 Avoiding repetition

In order to avoid having to sample the geometry for $\varepsilon = 0, \pm 1/6$ by calling the curve function three times, `sample` does this automatically by invoking the function three times. It works with curves with need zero or two additional parameters (see examples below).

```
function [g,gp,gm]=sample(curve,N,varargin)

% [g,gp,gm]=sample(@curve,N,varargin)
% Input:
%     @curve    : handle to one of the geometry functions
%     N         : number of points
%     varargin : other parameters needed by curve
% Output:
%     g,gp,gm   : sampled geometries with eps=0,1/2,-1/6
% Last modified: January 12, 2015

switch nargin
    case 2
        g =curve(N,0);
        gp=curve(N,1/6);
        gm=curve(N,-1/6);
    case 4
        g =curve(N,0,varargin{1},varargin{2});
        gp=curve(N,1/6,varargin{1},varargin{2});
        gm=curve(N,-1/6,varargin{1},varargin{2});
end
end
```

For the utilities `affine`, `mirror`, `mirrorLR`, and `latticeGeometry`, the function `threetimes` applies the function to the main and companion grids.

```
function [gnew,gpnew,gmnew]=threetimes(utility,g,gp,gm,varargin)

% [gnew,gpnew,gmnew]=threetimes(@utility,g,gp,gm,varargin)
% Input:
%     @utility : handle to one of the geometric utilities
%     varargin : other parameters needed by utility
% Output:
%     gnew      : utility(g,varargin)
%     gpnew     : utility(gp,varargin)
%     gmnew     : utility(gm,varargin)
% Last modified: January 12, 2015

switch nargin
    case 4
        gnew =utility(g);
        gpnew=utility(gp);
        gmnew=utility(gm);
    case 5
        gnew =utility(g,varargin{1});
        gpnew=utility(gp,varargin{1});
        gmnew=utility(gm,varargin{1});
    case 6
        gnew =utility(g,varargin{1},varargin{2});
        gpnew=utility(gp,varargin{1},varargin{2});
        gmnew=utility(gm,varargin{1},varargin{2});
end
end
```

The function `merge` simplifies the process of using `joinGeometry` by:

- doing it for multiple merges,

- doing it for the main and companion grids.

```
function [g,gp,gm]=merge(G,Gp,Gm)

% [g,gp,gm]=merge({g1,g2,...,gM},{gp1,gp2,...,gpM},{gm1,...,gmM});
% Input:
%     g1,g2,...    : geometry data structures (collected in cell array)
%     gp1,gp2,... : (same)
%     gm1,gm2,... : (same)
% Output:
%     g  : geometry data structure
%          join(join(....(join(g1,g2),g3),...),gM) [joinGeometry]
% Last modified: January 12, 2015

M=length(G);
g =G{1};
gp=Gp{1};
gm=Gm{1};
for c=2:M
    g =joinGeometry(g,G{c});
    gp=joinGeometry(gp,Gp{c});
    gm=joinGeometry(gm,Gm{c});
end
end
```

1.4 Examples of geometries

Ellipse

For a given center (c_1, c_2) and semiaxes (R_1, R_2), we consider the ellipse:

$$\mathbf{x}(t) := (c_1 + R_1 \cos(2\pi\, t), c_2 + R_2 \sin(2\pi\, t)).$$

```
function  g = ellipse(N,ep,R,c)

% g = ellipse(N,eps,[a,b],[cx,cy])
% Input:
%    N       : discrete interval number
%    ep      : epsilon parameter
%    [a b]   : semiaxes
%    [cx,cy] : center
% Output:
%    g       : sampled discrete geometry
%
% Last modified: August 2, 2013

h = 1/N;
t = h*(0:N-1); t = t+ep*h;
cost = cos(2*pi*t);
sint = sin(2*pi*t);

g.midpt = [c(1)+R(1)*cost;...
           c(2)+R(2)*sint]';
g.brkpt = [c(1)+R(1)*cos(2*pi*(t-0.5*h));...
           c(2)+R(2)*sin(2*pi*(t-0.5*h))]';
xp = [-R(1)*2*pi*sint;...
       R(2)*2*pi*cost]';
g.normal  = h*[xp(:,2) -xp(:,1)];
g.next    = [2:N 1];
g.comp    = [1];
```

```
return
```

General star-shaped domain

Given a 2π-periodic function r (and its first derivative r'), we consider the curve

$$\begin{aligned}\mathbf{x}(t) &= r(2\pi t)(\cos(2\pi t), \sin(2\pi t)),\\ \mathbf{x}'(t) &= (2\pi)\Big(r'(2\pi t)(\cos(2\pi t), \sin(2\pi t)) + r(t)(-\sin(2\pi t), \cos(2\pi t))\Big)\end{aligned}$$

```
function  g = starshape(N,ep,r,rp)

% g = starshape(N,ep,r,rp)
% Input:
%    N       : discretization parameter
%    ep      : epsilon parameter
%    r       : 2-Pi periodic radius function
%   rp       : first derivative of r
% Output:
%    g       : discrete geometry
%
% Last modified: August 2, 2012

h = 1/N;
t = h*(0:N-1); t = t+ep*h; t=t(:);
tau  = 2*pi*(t-0.5*h);
t    = 2*pi*t;
cost = cos(t);
sint = sin(t);
rt   = r(t);
rpt  = rp(t);

g.midpt = bsxfun(@times,[cost sint], rt);
g.brkpt = bsxfun(@times,[cos(tau) sin(tau)], r(tau));
xp = 2*pi*[rpt.*cost-rt.*sint,...
           rpt.*sint+rt.*cost];
g.normal  = h*[xp(:,2) -xp(:,1)];
g.next    = [2:N 1];
g.comp    = [1];

return
```

A TV-shaped domain

This domain is a slightly non-convex smoothened square centered at the origin. The length of its side is less than 3.06.

$$\mathbf{x}(t) = \left[\begin{array}{cc}(1+\cos(2\pi t)^2)\cos(2\pi t) & (1+\sin(2\pi t)^2)\sin(2\pi t)\end{array}\right]\left[\begin{array}{cc}\cos\pi/4 & \sin\pi/4\\ -\sin\pi/4 & \cos\pi/4\end{array}\right]$$

```
function  g = tvshape(N,ep)

% function  g = tvshape(N,ep)
% Input:
%        N   : number of space intervals
%        ep  : epsilon parameter
% Output:
```

```
%        g   : discrete sampling of smoothened square
%
% Last modified: August 2, 2013

h = 1/N;
t = h*(0:N-1); t = t+ep*h;

g.midpt  = [(1+cos(2*pi*t).^2).*cos(2*pi*t);...
            (1+sin(2*pi*t).^2).*sin(2*pi*t)]';
g.brkpt  = [(1+cos(2*pi*(t-0.5*h)).^2).*cos(2*pi*(t-0.5*h));...
            (1+sin(2*pi*(t-0.5*h)).^2).*sin(2*pi*(t-0.5*h))]';
R=[cos(pi/4) sin(pi/4); -sin(pi/4) cos(pi/4)]';
g.midpt = g.midpt*R;
g.brkpt = g.brkpt*R;

xp = [6.*pi.*sin(2.*pi.*t).^3 - 8.*pi.*sin(2.*pi.*t);...
      2.*pi.*(4.*cos(2.*pi.*t) - 3.*cos(2.*pi.*t).^3)]';
xp = xp*R;
g.normal  = h*[xp(:,2) -xp(:,1)];
g.next    = [2:N 1];
g.comp    = [1];
return
```

A kite

The parametrization

$$\mathbf{x}(t) = (\cos(2\pi t) + \cos(4\pi t)),\ 2\sin(2\pi t))$$

corresponds to a kite-shaped domain ponting towards the positive x axis. The domain fits in the rectangle $[-1.13, 2] \times [-2, 2]$.

```
function  g = kite(N,ep)

% function  g = kite(N,ep)
% Input:
%        N   = number of space intervals
%        ep  = epsilon parameter
% Output:
%          g  : discrete sampled geometry for a kite-shaped domain
% Last modified: August 2, 2013


h = 1/N;
t = h*(0:N-1); t = t+ep*h;

g.midpt  = [cos(2*pi*t)+cos(4*pi*t);...
            2*sin(2*pi*t)]';
g.brkpt  = [cos(2*pi*(t-0.5*h))+cos(4*pi*(t-0.5*h));...
            2*sin(2*pi*(t-0.5*h))]';
xp= [-2*pi*sin(2*pi*t)-4*pi*sin(4*pi*t);...
      4*pi*cos(2*pi*t)]';
g.normal  = h*[xp(:,2) -xp(:,1)];
g.next    = [2:N 1];
g.comp    = [1];
return
```

Chapter 2

The Helmholtz Calderón Calculus

This part contains the implementation details of the methods in [1], restricted to the particular choices of the parameter $\alpha = 1$ and $\alpha = 5/6$. The default value is $\alpha = 1$ (we will refer to this case as *averaging*). The case $\alpha = 5/6$ will be called *mixing*. It is related to the combination of Dirac deltas named the *fork* in [1].

> Everything is written so that we can work with Convolution Quadrature techniques, so instead of working with the Helmholtz equation $\Delta + k^2$, we use the resolvent equations $\Delta - s^2$. Given any of the transfer functions $\mathrm{A}(s)$ (operator or potential for the resolvent equation), to obtain the corresponding element for the Helmholtz equation use $\mathrm{A}(-\imath k)$, that is plug in $s = -\imath k$.

2.1 Geometric elements and mixing matrices

Take parametric curves (see Chapter 1) $t \mapsto \{\mathbf{x}_1(t), \ldots, \mathbf{x}_L(t)\}$, integers $\{N_1, \ldots, N_L\}$, meshsizes $h_\ell := 1/N_\ell$, and compute:

- points on the main and companion grids

  $$\mathbf{x}_{\ell,i} := \mathbf{x}_\ell(i\, h_\ell), \qquad \mathbf{n}_{\ell,i} := h_\ell \mathbf{x}'(i\, h_\ell)^\perp, \qquad \mathbf{b}_{\ell,i} := \mathbf{x}((i - \tfrac{1}{2})h_\ell), \qquad i = 1, \ldots, N_\ell,$$

  (for $\ell = 1, \ldots, L$).

- the corresponding next-index function $n_\ell : \mathbb{Z}_{N_\ell} \to \mathbb{Z}_{N_\ell}$ given $n_\ell(i) = i + 1 \pmod{N_\ell}$,

- equally sized samples on two companion grids

  $$\mathbf{x}^\pm_{\ell,i} := \mathbf{x}_\ell((i \pm \tfrac{1}{6})\, h_\ell), \qquad \mathbf{n}^\pm_{\ell,i} := h_\ell \mathbf{x}'((i \pm \tfrac{1}{6})\, h_\ell)^\perp, \qquad \mathbf{b}^\pm_{\ell,i} := \mathbf{x}((i \pm \tfrac{1}{6} - \tfrac{1}{2})h_\ell),$$

  (for $i = 1, \ldots, n_\ell, \ell = 1, \ldots, L$)

Each of the three groups of discrete elements are merged in a single geometric structure with $N = N_1 + \ldots + N_L$ elements: on the main grid we have $\mathbf{m}_i$, $\mathbf{n}_i$, $\mathbf{b}_i$ (and the corresponding next-index function), and we have two companion grids with $\mathbf{m}^\pm_i$, $\mathbf{n}^\pm_i$, $\mathbf{b}^\pm_i$.
We consider the $N \times N$ matrix

$$\mathrm{Q}_{ij} = \tfrac{1}{24} \begin{cases} 22 & i = j, \\ 1 & i = n(j) \quad \text{or} \quad j = n(i), \\ 0 & \text{otherwise}, \end{cases}$$

Finally we consider the mass matrix

$$\mathrm{M}_{ij} = \tfrac{1}{18}\begin{cases} 16 & i=j, \\ 1 & i=n(j) \quad \text{or} \quad j=n(i), \\ 0 & \text{otherwise,} \end{cases} \qquad \text{or} \qquad \mathrm{M}_{ij} = \tfrac{1}{9}\begin{cases} 7 & i=j, \\ 1 & i=n(j) \quad \text{or} \quad j=n(i), \\ 0 & \text{otherwise,} \end{cases}$$

The first case corresponds to averaging, and the second case corresponds to mixing. They can be coded together as

$$\mathrm{M}_{ij} = \tfrac{1}{18}\begin{cases} 4+12\alpha & i=j, \\ 7-6\alpha & i=n(j) \quad \text{or} \quad j=n(i), \\ 0 & \text{otherwise,} \end{cases} \qquad \alpha = \begin{cases} 1, & \text{(averaging)} \\ 5/6 & \text{(mixing)} \end{cases}$$

There are two more matrices only for the *mixing* case:

$$\mathrm{P}^{+}_{ij} = \tfrac{1}{2}\begin{cases} \frac{5}{6} & i=j, \\ \frac{1}{6} & i=n(j), \\ 0 & \text{otherwise,} \end{cases} \qquad \mathrm{P}^{-} = (\mathrm{P}^{+})^{\top}.$$

In the averaging case $\mathrm{P}^{+} = \mathrm{P}^{-} = \frac{1}{2}\mathrm{I}$, and we will produced just two scalars instead of two multiples of the identity matrix.
The sparse matrices $\mathrm{Q}, \mathrm{M}, \mathrm{P}^{\pm}$ are produced in the function `CalderonCalculusMatrices`:

- The default case is averaging, producing two scalars for $\mathrm{P}^{\pm}$. This case can be invoked by not using a final (variable–argument–in) parameter, or by taking this parameter to be zero.
- If the additional parameter is not zero, mixing is used.

The additional parameter is referred to as **`fork`** in all pieces of code. This is justified by the presentation of the methods in [1].

```
function [Q,M,Pp,Pm] = CalderonCalculusMatrices(g,varargin)

% [Q,M,Pp,Pm] = CalderonCalculusMatrices(g)
% [Q,M,Pp,Pm] = CalderonCalculusMatrices(g,fork)
% Input :
%        g : geometry
%     fork : 0 or absent (averaged method) ≠0 (mixed method)
% Output:
%       Q   : lookaround quadrature matrix
%       M   : mass matrix
%    Pp,Pm : mixing matrices (=1/2 if fork=0)
%
% Last Modified: October 31, 2013

fork=0;
if nargin==2
    fork=varargin{1};
end
if fork
    alpha=5/6;
else
    alpha=1;
end

N = size(g.midpt,1);

% quadrature matrix

Q = sparse(1:N,1:N,22/24)...
    +sparse(g.next,1:N,1/24)+sparse(1:N,g.next,1/24);
```

```
% mass matrix

M = sparse(1:N,1:N,(4+12*alpha)/18)...
    +sparse(g.next,1:N,(7-6*alpha)/18)+sparse(1:N,g.next,(7-6*alpha)/18);

% mixing matrices or average values

if fork
    Pp=sparse(1:N,1:N,alpha/2)+sparse(g.next,1:N,(1-alpha)/2);
    Pm=Pp';
else
    Pp=1/2;
    Pm=1/2;
end

return

% singular quadrature

S = sparse(1:N,1:N,349/432)...
    +sparse(1:N,g.next,5/1296)...
    +sparse(g.next,1:N,89/432)...
    +sparse(g.next(g.next),1:N,-23/1296);
```

2.2 Operators

Consider the three pairs of discrete operators (depending on the complex frequency number s):

$$\begin{aligned}
\mathrm{V}^{\pm}_{ij}(s) &:= \frac{\imath}{4}H^{(1)}_0(\imath s|\mathbf{m}^{\pm}_i - \mathbf{m}_j|),\\
\mathrm{K}^{\pm}_{ij}(s) &:= -\frac{s}{4}H^{(1)}_1(\imath s|\mathbf{m}^{\pm}_i - \mathbf{m}_j|)\,\frac{(\mathbf{m}^{\pm}_i - \mathbf{m}_j)\cdot\mathbf{n}_j}{|\mathbf{m}^{\pm}_i - \mathbf{m}_j|},\\
\mathrm{J}^{\pm}_{ij}(s) &:= -\frac{s}{4}H^{(1)}_1(\imath s|\mathbf{m}^{\pm}_i - \mathbf{m}_j|)\,\frac{(\mathbf{m}_j - \mathbf{m}^{\pm}_i)\cdot\mathbf{n}^{\pm}_i}{|\mathbf{m}^{\pm}_i - \mathbf{m}_j|}\\
&= \frac{s}{4}H^{(1)}_1(\imath s|\mathbf{m}^{\pm}_i - \mathbf{m}_j|)\,\frac{(\mathbf{m}^{\pm}_i - \mathbf{m}_j)\cdot\mathbf{n}^{\pm}_i}{|\mathbf{m}^{\pm}_i - \mathbf{m}_j|}
\end{aligned}$$

We also consider

$$\mathrm{V}^{\pm}_{\mathbf{n},ij}(s) := s^2(\mathbf{n}^{\pm}_i\cdot\mathbf{n}_j)\mathrm{V}^{\pm}_{ij}(s),$$

and finally

$$\widetilde{\mathrm{W}}^{\pm}_{ij}(s) := \widetilde{\mathrm{V}}^{\pm}_{n(i),n(j)}(s) - \widetilde{\mathrm{V}}^{\pm}_{n(i),j}(s) - \widetilde{\mathrm{V}}^{\pm}_{i,n(j)}(s) + \widetilde{\mathrm{V}}^{\pm}_{ij}(s), \quad \text{where} \quad \widetilde{\mathrm{V}}^{\pm}_{ij}(s) := \frac{\imath}{4}H^{(1)}_0(\imath s|\mathbf{b}^{\pm}_i - \mathbf{b}_j|).$$

A subfunction computes the five matrix-valued functions above for a main grid (no superscript) and one of the companion grids ($\pm$ superscript).
In the final step, the discrete operators for the Calderón Calculus are built with the expressions

$$\begin{aligned}
\mathrm{V}(s) &:= \mathrm{P}^+\mathrm{V}^+(s) + \mathrm{P}^-\mathrm{V}^-(s),\\
\mathrm{K}(s) &:= (\mathrm{P}^+\mathrm{K}^+(s) + \mathrm{P}^-\mathrm{K}^-(s))\mathrm{Q},\\
\mathrm{J}(s) &:= \mathrm{Q}(\mathrm{P}^+\mathrm{J}^+(s) + \mathrm{P}^-\mathrm{J}^-(s)),\\
\mathrm{W}(s) &:= \mathrm{P}^+\widetilde{\mathrm{W}}^+(s) + \mathrm{P}^-\widetilde{\mathrm{W}}^-(s) + \mathrm{Q}(\mathrm{P}^+\mathrm{V}^+_{\mathbf{n}}(s) + \mathrm{P}^-\mathrm{V}^-_{\mathbf{n}}(s))\mathrm{Q}.
\end{aligned}$$

The output is four matrix-valued functions of the variable s.

```
function [V,K,J,W] = CalderonCalculusHelmholtz(g,gp,gm,varargin)

% [V,K,J,W] = CalderonCalculusHelmholtz(g,gp,gm)
% [V,K,J,W] = CalderonCalculusHelmholtz(g,gp,gm,fork)
% Input:
%     g : principal geometry
%    gp : companion geometry with epsilon = 1/6
%    gm : companion geometry with epsilon = -1/6
%    fork : 0 or absent (averaged method) ≠0 (mixed method)
% Output:
%     V : single layer operator V
%     K : double layer operator K
%     J : transpose of operator K
%     W : hypersingular operator W
%
% Last Modified: October 31, 2013

[Vp,Kp,Jp,Wpp,Vnp] = CalderonCalculusHelmholtzHalf(g,gp);
[Vm,Km,Jm,Wpm,Vnm] = CalderonCalculusHelmholtzHalf(g,gm);

fork=0;
if nargin==4
    fork=varargin{1};
end
[Q,~,Pp,Pm] = CalderonCalculusMatrices(g,fork);

V =@(s) Pp*Vp(s) + Pm*Vm(s);
K =@(s) (Pp*Kp(s) + Pm*Km(s))*Q;
J =@(s) Q*(Pp*Jp(s) + Pm*Jm(s));
W =@(s) Pp*Wpp(s) + Pm*Wpm(s) + Q*(Pp*Vnp(s) + Pm*Vnm(s))*Q;

return

% Subfunction computing the two halves of the Calculus

function [V,K,J,Wp,Vn] = CalderonCalculusHelmholtzHalf(g,gp)

% [V,K,J,Wp,Vn] = CalderonCalculusHelmholtzHalf(g,gp)
% Input:
%     g : principal geometry
%    gp : companion geometry
% Output:
%     V : single layer operator
%     K : double layer operator
%     J : transpose of double layer operator
%    Wp : principal part of hypersingular operator W
%    Vn : regular part of W
%
% Last Modified: September 5, 2013

% V(s)  = Single layer operator
% Vn(s) = Regular part of W(s)

DX = bsxfun(@minus,gp.midpt(:,1),g.midpt(:,1)');
DY = bsxfun(@minus,gp.midpt(:,2),g.midpt(:,2)');
D = sqrt(DX.^2+DY.^2);                          % |m_i^ep-m_j|
H  = gp.normal(:,1)*g.normal(:,1)'...
     +gp.normal(:,2)*g.normal(:,2)'; % n_i . n_j

V = @(s) 1i/4.*besselh(0,1,1i*s*D);
Vn = @(s) s^2.*H.*1i/4.*besselh(0,1,1i*s*D);

% Principal part of W(s)

DX = bsxfun(@minus,gp.brkpt(:,1),g.brkpt(:,1)');
DY = bsxfun(@minus,gp.brkpt(:,2),g.brkpt(:,2)');
```

```
D1 = sqrt(DX.^2+DY.^2);                                % |b_i^ep-b_j|
D2 = D1(gp.next,:);                                   % |b_i+1^ep-b_j|
D3 = D1(:,g.next);                                     % |b_i^ep-b_j+1|
D4 = D1(gp.next,g.next);                              % |b_i+1^ep-b_j+1|
Wp = @(s) 1i/4*besselh(0,1,1i*s*D1)-1i/4*besselh(0,1,1i*s*D2)...
          -1i/4*besselh(0,1,1i*s*D3)+1i/4*besselh(0,1,1i*s*D4);

% K(s) = Double layer operator

DX = bsxfun(@minus,gp.midpt(:,1),g.midpt(:,1)');
DY = bsxfun(@minus,gp.midpt(:,2),g.midpt(:,2)');
D  = sqrt(DX.^2+DY.^2);
N = bsxfun(@times,DX,g.normal(:,1)')...
    +bsxfun(@times,DY,g.normal(:,2)');
N = N./D;
K = @(s) -s/4.*besselh(1,1,1i*s*D).*N;

% J(s) = Transposed double layer operator

N = bsxfun(@times,gp.normal(:,1),DX)...
     +bsxfun(@times,gp.normal(:,2),DY);
N = N./D;
J = @(s) s/4.*besselh(1,1,1i*s*D).*N;

return
```

2.3 Potentials and right-hand sides

Given a collection of observation points $\{\mathbf{z}_1, \ldots, \mathbf{z}_M\}$, we compute the $M \times N$ matrix of monopoles

$$\mathrm{S}_{ij}(s) := \frac{\imath}{4} H_0^{(1)}(\imath s |\mathbf{z}_i - \mathbf{m}_j|)$$

and the $M \times N$ matrix of dipoles

$$\mathrm{D}_{ij}(s) := -\frac{s}{4} H_1^{(1)}(\imath s |\mathbf{z}_i - \mathbf{m}_j|) \frac{(\mathbf{z}_i - \mathbf{m}_j) \cdot \mathbf{n}_j}{|\mathbf{z}_i - \mathbf{m}_j|}.$$

```
function [SL,DL]=HelmholtzPotentials(g,z)

% [SL,DL]=HelmholtzPotentials(g,z)
%
% Input:
%     g : geometry
%     z : K x 2 matrix with points where potentials are evaluated
% Output:
%   SL(s)      : transfer function for SL
%   DL(s)      : transfer function for DL
% Last modified: May 28, 2014.

% SL(s) : single layer potential

RX = bsxfun(@minus,z(:,1),g.midpt(:,1)');
RY = bsxfun(@minus,z(:,2),g.midpt(:,2)');
R = sqrt(RX.^2+RY.^2);
SL = @(s) 1i/4*besselh(0,1,1i*s*R);

% DL(s) : double layer potential

RN = bsxfun(@times,RX,g.normal(:,1)')...
    +bsxfun(@times,RY,g.normal(:,2)');
RN = RN./R;
```

```
DL = @(s) -s/4*besselh(1,1,1i*s*R).*RN;

return
```

Given a function u and its gradient ∇u, we start by computing the column vectors

$$(\boldsymbol{\beta}_0^{\pm})_i := u(\mathbf{m}_i^{\pm}), \qquad (\boldsymbol{\beta}_1^{\pm})_i := \nabla u(\mathbf{m}_i^{\pm}) \cdot \mathbf{n}_i^{\pm}, \qquad i = 1, \ldots, N,$$

and then mix them to obtain the observation vectors

$$\boldsymbol{\beta}_0 = \mathrm{P}^+ \boldsymbol{\beta}_0^+ + \mathrm{P}^- \boldsymbol{\beta}_0^-, \qquad \boldsymbol{\beta}_1 = \mathrm{Q}(\mathrm{P}^+ \boldsymbol{\beta}_1^+ + \mathrm{P}^- \boldsymbol{\beta}_1^-).$$

The function u is given as a vectorized function of the variables x and y. The gradient has to be input as a vectorized function of two variables returning a $N_{\text{points}} \times 2$ matrix.

```
function [beta0,beta1] = CalderonCalculusTest(u,gradu,gp,gm,varargin)

% [beta0,beta1] = CalderonCalculusTest(u,gradu,gp,gm)
% [beta0,beta1] = CalderonCalculusTest(u,gradu,gp,gm,fork)
% Input:
%       u : vectorized scalar function of x,y
%   gradu : vectorized vector-valued function of x,y (returns K x 2 matrix)
%   gp,gm : two companion meshes
%    fork : 0 or absent (averaged method) ≠0 (mixed method)
% Output:
%   beta0 : Dirichlet boundary data
%   beta1 : Neumann boundary data
%
% Last Update: October 31, 2013

% Tests on two meshes

beta0p = u(gp.midpt(:,1),gp.midpt(:,2));
beta1p = sum(gradu(gp.midpt(:,1),gp.midpt(:,2)).*gp.normal,2);

beta0m = u(gm.midpt(:,1),gm.midpt(:,2));
beta1m = sum(gradu(gm.midpt(:,1),gm.midpt(:,2)).*gm.normal,2);

% averaging/mixing and quadrature

if nargin==5
    fork=varargin{1};
else
    fork=0;
end
[Q,¬,Pp,Pm] = CalderonCalculusMatrices(gp,fork);
beta0 = Pp*beta0p+Pm*beta0m;
beta1 = Q*(Pp*beta1p+Pm*beta1m);

return
```

2.4 Decoupled Calculus

Let $\Gamma = \Gamma_1 \cup \ldots \cup \Gamma_M$ be a collection of curves. The goal of this function is the creation of block diagonal functions with blocks

$$\alpha_\ell^{-1} \mathrm{V}_\ell(s/c_\ell), \qquad \alpha_\ell \mathrm{W}_\ell(s/c_\ell), \qquad \mathrm{K}_\ell(s/c_\ell), \qquad \mathrm{J}_\ell(s/c_\ell),$$

for given vectors of parameters $(c_1, \ldots, c_M)$ and $(\alpha_1, \ldots, \alpha_M)$. These operators are needed for the implementation of transmission problems with interior obstacles with different material properties. The lack of connexion between the different operators is related to the fact that the interior problems will not be related.

```
function [V,K,J,W]=CalderonCalculusHelmholtzDecoupled(g,gp,gm,c,alpha,varargin)

% [V,K,J,W]=CalderonCalculusHelmholtzDecoupled(g,gp,gm,c,alpha)
% [V,K,J,W]=CalderonCalculusHelmholtzDecoupled(g,gp,gm,c,alpha,fork)
%
% Input:
%    g,gp,gm    : geometry structures for main and companion meshes
%                 for a geometry with K components
%    c, alpha   : K x 1 vectors
%    fork       : 0 or absent (averaged method) ≠0 (mixed method)
% Output:
%    V,K,J,W    : block diagonal operators (Costabel-Stephan formulation)
%                 chi^{-1} V(s/c), K(s/c), J(s/c), chi W(s/c)
% Last modified: October 31, 2013

G=unpackGeometry(g);
Gp=unpackGeometry(gp);
Gm=unpackGeometry(gm);
nComp=length(G);

fork=0;
if nargin==6
    fork=varargin{1};
end
for comp=1:nComp
    [VV,KK,JJ,WW]=CalderonCalculusHelmholtz(G{comp},Gp{comp},Gm{comp},fork);
     V{comp}=VV;
     K{comp}=KK;
     J{comp}=JJ;
     W{comp}=WW;
end
V=@(s) attachBlockDiag(V,s,c,alpha.^(-1));
W=@(s) attachBlockDiag(W,s,c,alpha);
J=@(s) attachBlockDiag(J,s,c,ones(size(alpha)));
K=@(s) attachBlockDiag(K,s,c,ones(size(alpha)));
return

% Subfunction to merge functions in a diagonal block function

function F=attachBlockDiag(f,s,speed,factor)
a=cell(1,length(f));
for i=1:length(f);
    a{i}=factor(i)*f{i}(s/speed(i));
end
F=blkdiag(a{:});
return
```

Chapter 3

Use of the Helmholtz Calderón Calculus

3.1 Exterior Dirichlet and Neumann problems

The geometric layout for exterior problems is considering the exterior of a collection of closed curves Γ_ℓ with non-intersecting interior. As we are parametrizing with positive orientation, normal vectors point in the correct direction.

Continuous equations

Consider the equation

$$\Delta U - s^2 U = 0 \qquad \text{in } \Omega_+, \tag{3.1}$$

with boundary condition

$$\gamma^- U = \beta_0 \qquad \text{or} \qquad \partial_{\mathbf{n}}^- U = \beta_1. \tag{3.2}$$

Problem (3.1)-(3.2) is uniquely solvable for the resolvent equations ($s \in \mathbb{C}_+ := \{s \in \mathbb{C} : \mathrm{Re}s > 0\}$) and also in the acoustic case $s = -\imath k$. In the first case $U \in H^1(\Omega_+)$ decays exponentially fast at infinity, while in the second case, the Sommerfeld radiation condition

$$\nabla U(\mathbf{z}) \cdot \left(\tfrac{1}{|\mathbf{z}|}\mathbf{z}\right) - \imath k U(\mathbf{z}) = o(\frac{1}{\sqrt{|\mathbf{z}|}}) \qquad \text{as } |\mathbf{z}| \to \infty,$$

is satisfied. If $\mathbf{x}_0$ is in the interior of Γ, then

$$U(\mathbf{z}) = H_0^{(1)}(\imath s\, |\mathbf{z} - \mathbf{x}_0|), \qquad \mathbf{z} \in \Omega_+,$$

is a (radiating) solution of (3.1). In scattering problems, there is an incident wave U^{inc} and the boundary condition is

$$\beta_0 := -\gamma U^{\mathrm{inc}}, \qquad \beta_1 := -\partial_{\mathbf{n}} U^{\mathrm{inc}}, \tag{3.3}$$

corresponding to sound-soft and sound-hard obstacles respectively. The total wave is $U + U^{\mathrm{inc}}$.
We can use a **direct formulation** using the potential representation for the solution of (3.1)-(3.2)

$$U = \mathrm{D}(s)\varphi - \mathrm{S}(s)\lambda, \qquad \varphi = \gamma^+ U, \qquad \lambda = \partial_{\mathbf{n}}^+ U.$$

The following table shows the four associated integral equations:

Dirichlet	$\mathrm{V}(s)\lambda = -\frac{1}{2}\varphi + \mathrm{K}(s)\varphi$	$\varphi = \beta_0$	(dD01)
	$\frac{1}{2}\lambda + \mathrm{J}(s)\lambda = -\mathrm{W}(s)\varphi,$	$\varphi = \beta_0$	(dD02)
Neumann	$-\frac{1}{2}\varphi + \mathrm{K}(s)\varphi = \mathrm{V}\lambda,$	$\lambda = \beta_1$	(dN01)
	$-\mathrm{W}(s)\varphi = \frac{1}{2}\lambda + \mathrm{J}(s)\lambda,$	$\lambda = \beta_1$	(dN02)

We can also use a single or double **layer potential ansatz**

$$U = \mathrm{S}(s)\eta \qquad \text{or} \qquad U = \mathrm{D}(s)\psi,$$

or a combined-field representation

$$U = \mathrm{D}(s)\eta + s\mathrm{S}(s)\eta,$$

leading to the following collection of equations:

Dirichlet	$\mathrm{V}(s)\eta = \beta_0,$	$U = \mathrm{S}(s)\,\eta$	(iD01)
	$\frac{1}{2}\psi + \mathrm{K}(s)\psi = \beta_0,$	$U = \mathrm{D}(s)\,\psi$	(iD02)
	$\frac{1}{2}\eta + \mathrm{K}(s)\eta + s\mathrm{V}(s)\eta = \beta_0,$	$U = \mathrm{D}(s)\eta + s\mathrm{V}(s)\eta$	(iD03)
Neumann	$-\frac{1}{2}\eta + \mathrm{J}(s)\eta = \beta_1,$	$U = \mathrm{S}(s)\,\eta$	(iN01)
	$\mathrm{W}(s)\psi = -\beta_1,$	$U = \mathrm{D}(s)\,\psi$	(iN02)
	$\mathrm{W}(s)\eta + \frac{1}{2}s\eta - s\mathrm{J}(s)\eta = -\beta_1,$	$U = \mathrm{D}(s)\eta + s\mathrm{V}(s)\eta$	(iN03)

The **Burton-Miller integral equations** are based on the fact that the incident wave is a solution of the interior problem and therefore

$$\mathrm{S}(s)\gamma U^{\mathrm{inc}} - \mathrm{D}(s)\partial_{\mathbf{n}} U^{\mathrm{inc}} = 0 \qquad \text{in } \Omega_+,$$

and therefore, using (3.3), we can write

$$\mathrm{S}(s)\beta_1 = \mathrm{D}(s)\beta_0 \qquad \text{in } \Omega_+.$$

For the Dirichlet problem we then represent

$$U = -\mathrm{S}(s)\xi = \mathrm{D}(s)\varphi - \mathrm{S}(s)\lambda, \qquad \lambda - \xi = \beta_1, \qquad \varphi = \beta_0$$

and use the integral equation

$$\tfrac{1}{2}\xi + \mathrm{J}(s)\xi + s\mathrm{V}(s)\xi = -\beta_1 - s\beta_0.$$

For the Neumann problem, we represent

$$U = \mathrm{D}(s)\psi = \mathrm{D}(s)\varphi - \mathrm{S}(s)\lambda, \qquad \varphi - \psi = \beta_0, \qquad \lambda = \beta_1$$

and use the integral equation

$$\mathrm{W}(s)\psi + \tfrac{1}{2}s\psi - s\mathrm{K}(s)\psi = -\beta_1 - s\beta_0.$$

Note that the Burton-Miller formulations work only when the data β_0 and β_1 are the Cauchy data of an interior solution. They will not work for point-source solutions that are often used for testing.

Discrete equations

After sampling the boundary data $\boldsymbol{\beta}_0$ and $\boldsymbol{\beta}_1$, and constructing integral operators ($\mathrm{V}(s)$, $\mathrm{K}(s)$, $\mathrm{J}(s)$, and $\mathrm{W}(s)$), the mass matrix M and the quadrature matrix Q, we can think in the following terms.

Direct methods. We write

$$U_h = \mathrm{D}(s)\boldsymbol{\phi} - \mathrm{S}(s)\boldsymbol{\lambda}, \qquad \boldsymbol{\phi} = \mathrm{Q}\boldsymbol{\varphi}.$$

We are expecting

$$\max_i N_i|\lambda_i - \nabla U^+(\mathbf{m}_i)\cdot \mathbf{n}_i| = \mathcal{O}(h^3), \qquad \max_i |\phi_i - U^+(\mathbf{m}_i)| = \mathcal{O}(h^3).$$

(Note the discrepancy of our code with [1], where effectively we are using $\mathrm{D}(s)\mathrm{Q}$ from the beginning.)

Dirichlet	$\mathrm{V}(s)\boldsymbol{\lambda} = -\frac{1}{2}\mathrm{M}\boldsymbol{\varphi} + \mathrm{K}(s)\boldsymbol{\varphi}$	$\mathrm{M}\boldsymbol{\varphi} = \boldsymbol{\beta}_0$	(dD01)
	$\frac{1}{2}\mathrm{M}\boldsymbol{\lambda} + \mathrm{J}(s)\boldsymbol{\lambda} = -\mathrm{W}(s)\boldsymbol{\varphi}$,	$\mathrm{M}\boldsymbol{\varphi} = \boldsymbol{\beta}_0$	(dD02)
	$\frac{1}{2}\mathrm{M}\boldsymbol{\xi} + \mathrm{J}(s)\boldsymbol{\xi} + s\mathrm{V}(s)\boldsymbol{\xi} = -\boldsymbol{\beta}_1 - s\boldsymbol{\beta}_0$,	$U_h = -\mathrm{S}(s)\boldsymbol{\xi}$ $\mathrm{M}\boldsymbol{\varphi} = \boldsymbol{\beta}_0, \quad \mathrm{M}(\boldsymbol{\lambda}-\boldsymbol{\xi}) = \boldsymbol{\beta}_1$	(dD03)
Neumann	$-\frac{1}{2}\mathrm{M}\boldsymbol{\varphi} + \mathrm{K}(s)\boldsymbol{\varphi} = \mathrm{V}\boldsymbol{\lambda}$,	$\mathrm{M}\boldsymbol{\lambda} = \boldsymbol{\beta}_1$	(dN01)
	$-\mathrm{W}(s)\boldsymbol{\varphi} = \frac{1}{2}\mathrm{M}\boldsymbol{\lambda} + \mathrm{J}(s)\boldsymbol{\lambda}$,	$\mathrm{M}\boldsymbol{\lambda} = \boldsymbol{\beta}_1$	(dN02)
	$\mathrm{W}(s)\boldsymbol{\psi} + \frac{1}{2}s\mathrm{M}\boldsymbol{\psi} - s\mathrm{K}(s)\boldsymbol{\psi} = -\boldsymbol{\beta}_1 - s\boldsymbol{\beta}_0$,	$U_h = \mathrm{D}(s)\mathrm{Q}\boldsymbol{\psi}$ $\mathrm{M}\boldsymbol{\lambda} = \boldsymbol{\beta}_1, \quad \mathrm{M}(\boldsymbol{\varphi}-\boldsymbol{\psi}) = \boldsymbol{\beta}_0$	(dN03)

Indirect methods. In the case of indirect methods there is no approximation of the Cauchy data. The discretized integral equations are

Dirichlet	$\mathrm{V}(s)\boldsymbol{\eta} = \boldsymbol{\beta}_0$,	$U_h = \mathrm{S}(s)\,\boldsymbol{\eta}$	(iD01)
	$\frac{1}{2}\mathrm{M}\boldsymbol{\psi} + \mathrm{K}(s)\boldsymbol{\psi} = \boldsymbol{\beta}_0$,	$U_h = \mathrm{D}(s)\mathrm{Q}\boldsymbol{\psi}$	(iD02)
	$\frac{1}{2}\mathrm{M}\boldsymbol{\eta} + \mathrm{K}(s)\boldsymbol{\eta} + s\mathrm{V}(s)\boldsymbol{\eta} = \boldsymbol{\beta}_0$,	$U_h = \mathrm{D}(s)\mathrm{Q}\boldsymbol{\eta} + s\mathrm{V}(s)\boldsymbol{\eta}$	(iD03)
Neumann	$-\frac{1}{2}\mathrm{M}\boldsymbol{\eta} + \mathrm{J}(s)\boldsymbol{\eta} = \boldsymbol{\beta}_1$,	$U_h = \mathrm{S}(s)\,\boldsymbol{\eta}$	(iN01)
	$\mathrm{W}(s)\boldsymbol{\psi} = -\boldsymbol{\beta}_1$,	$U_h = \mathrm{D}(s)\mathrm{Q}\boldsymbol{\psi}$	(iN02)
	$\mathrm{W}(s)\boldsymbol{\eta} + \frac{1}{2}s\mathrm{M}\boldsymbol{\eta} - s\mathrm{J}(s)\boldsymbol{\eta} = -\boldsymbol{\beta}_1$,	$U_h = \mathrm{D}(s)\mathrm{Q}\boldsymbol{\eta} + s\mathrm{V}(s)\boldsymbol{\eta}$	(iN03)

3.2 Interior Dirichlet and Neumann problems

For quite obvious reasons, in the case of interior problem we are assuming that the domain Ω_- is connected. (Otherwise, we can just solve separate problems on each connected component.) Additional care has to be taken to the case of domains with holes, since parametrization gives inward pointing normals on the holes.

Consider now the problem

$$\Delta U - s^2 U = 0 \qquad \text{in } \Omega_-$$

with Dirichlet or Neumann boundary conditions

$$\gamma^- U = \beta_0 \qquad \text{or} \qquad \partial_{\mathbf{n}}^- U = \beta_1.$$

Note that the interior problem has resonant cases (all of the in the acoustic case $s = -\imath k$) where existence and uniqueness of solution are compromised. If $\mathbf{x}_0 \in \Omega_+$, then

$$U(\mathbf{z}) = H_0^{(1)}(\imath s|\mathbf{z} - \mathbf{x}_0|), \qquad \mathbf{z} \in \Omega_-,$$

is a solution pf the interior equation. If $|\mathbf{d}| = 1$, then

$$U(\mathbf{z}) = \exp(s\,\mathbf{d} \cdot \mathbf{z}), \qquad \mathbf{z} \in \Omega_-,$$

is also a solution. (These plane solutions are not valid for exterior problems, since they are not radiating.) If we write

$$U = \mathrm{S}(s)\lambda - \mathrm{D}(s)\varphi, \qquad \varphi = \gamma^- U, \qquad \lambda = \partial_{\mathbf{n}}^- U,$$

we can get to four different direct formulations:

Dirichlet	$\mathrm{V}(s)\lambda = \frac{1}{2}\boldsymbol{\varphi} + \mathrm{K}(s)\varphi$	$\varphi = \beta_0$	(dD01int)
	$-\frac{1}{2}\lambda + \mathrm{J}(s)\lambda = -\mathrm{W}(s)\varphi,$	$\varphi = \beta_0$	(dD02int)
Neumann	$\frac{1}{2}\varphi + \mathrm{K}(s)\varphi = \mathrm{V}\lambda,$	$\lambda = \beta_1$	(dN01int)
	$-\mathrm{W}(s)\varphi = -\frac{1}{2}\lambda + \mathrm{J}(s)\lambda,$	$\lambda = \beta_1$	(dN02int)

Note that we have just changed the signs of identity operators in the integral equations and the sign of the representation formula. With potential representations, we reach the following equations (where, once again, only identity matrices in integral equations change sign):

Dirichlet	$\mathrm{V}(s)\eta = \beta_0,$	$U = \mathrm{S}(s)\,\eta$	(iD01int)
	$-\frac{1}{2}\psi + \mathrm{K}(s)\psi = \beta_0,$	$U = \mathrm{D}(s)\,\psi$	(iD02int)
	$-\frac{1}{2}\eta + \mathrm{K}(s)\eta + s\mathrm{V}(s)\eta = \beta_0,$	$U = \mathrm{D}(s)\eta + s\mathrm{V}(s)\eta$	(iD03int)
Neumann	$\frac{1}{2}\eta + \mathrm{J}(s)\eta = \beta_1,$	$U = \mathrm{S}(s)\,\eta$	(iN01int)
	$\mathrm{W}(s)\psi = -\beta_1,$	$U = \mathrm{D}(s)\,\psi$	(iN02int)
	$\mathrm{W}(s)\eta - \frac{1}{2}s\eta - s\mathrm{J}(s)\eta = -\beta_1,$	$U = \mathrm{D}(s)\eta + s\mathrm{V}(s)\eta$	(iN03int)

3.3 Transmission problems

The case of a single scatterer. Consider now the problem

$$\Delta U - s^2 U = 0 \quad \text{in } \Omega_+, \qquad \Delta V - (s/c)^2 V = 0 \quad \text{in } \Omega_-,$$

with transmission conditions

$$\gamma^+ U - \beta_0 = \gamma^- V, \qquad \partial_{\mathbf{n}}^+ U - \beta_1 = \alpha \partial_{\mathbf{n}}^- V.$$

(Note the funny minus sign in the TC, consistent with our desire of having $\beta_0 = -\gamma U^{\rm inc}$, $\beta_1 = -\partial_{\mathbf{n}} U^{\rm inc}$). We choose

$$\varphi^- := \gamma^- V, \qquad \lambda^- := \alpha \partial_{\mathbf{n}}^- V,$$

as unknowns of the problem. The integral representations are

$$\begin{aligned} U &= \mathrm{D}(s)\varphi^+ - \mathrm{S}(s)\lambda^+, \qquad \varphi^+ = \varphi^- + \beta_0, \quad \lambda^+ = \lambda^- + \beta_1, \\ V &= \alpha^{-1}\mathrm{S}(s/c)\lambda^- - \mathrm{D}(s/c)\varphi^-, \end{aligned}$$

and the corresponding system of integral equations is

$$\begin{bmatrix} \mathrm{W}(s) + \alpha \mathrm{W}(\frac{s}{c}) & \mathrm{J}(s) + \mathrm{J}(\frac{s}{c}) \\ -\mathrm{K}(s) - \mathrm{K}(\frac{s}{c}) & \mathrm{V}(s) + \frac{1}{\alpha}\mathrm{V}(\frac{s}{c}) \end{bmatrix} \begin{bmatrix} \varphi^- \\ \lambda^- \end{bmatrix} = \begin{bmatrix} \mathrm{W}(s) & \frac{1}{2}\mathrm{I} + \mathrm{J}(s) \\ \frac{1}{2}\mathrm{I} - \mathrm{K}(s) & \mathrm{V}(s) \end{bmatrix} \begin{bmatrix} \tau_0 \\ \tau_1 \end{bmatrix}, \qquad \begin{aligned} \tau_0 &= -\beta_0, \\ \tau_1 &= -\beta_1. \end{aligned}$$

At the discrete level, we have to be careful to understand that data under the action of integral operators have to be projected on the space using the mass matrix. This is the reason where we have added two artificial equations. Recovery of the exterior approximations leads to the equations

$$\varphi^+ - \varphi^- = \beta_0, \qquad \lambda^+ - \lambda^- = \beta_1.$$

$$\begin{bmatrix} \mathrm{W}(s) + \alpha \mathrm{W}(\frac{s}{c}) & \mathrm{J}(s) + \mathrm{J}(\frac{s}{c}) \\ -\mathrm{K}(s) - \mathrm{K}(\frac{s}{c}) & \mathrm{V}(s) + \frac{1}{\alpha}\mathrm{V}(\frac{s}{c}) \end{bmatrix} \begin{bmatrix} \boldsymbol{\varphi}^- \\ \boldsymbol{\lambda}^- \end{bmatrix} = \begin{bmatrix} \mathrm{W}(s) & \frac{1}{2}\mathrm{M} + \mathrm{J}(s) \\ \frac{1}{2}\mathrm{M} - \mathrm{K}(s) & \mathrm{V}(s) \end{bmatrix} \begin{bmatrix} \boldsymbol{\tau}_0 \\ \boldsymbol{\tau}_1 \end{bmatrix}, \qquad \begin{aligned} \mathrm{M}\boldsymbol{\tau}_0 &= -\boldsymbol{\beta}_0, \\ \mathrm{M}\boldsymbol{\tau}_1 &= -\boldsymbol{\beta}_1. \end{aligned}$$

Recovery of the exterior Cauchy data can be done with the equations

$$\mathrm{M}\boldsymbol{\varphi}^+ - \mathrm{M}\boldsymbol{\varphi}^- = \boldsymbol{\beta}_0, \qquad \mathrm{M}\boldsymbol{\lambda}^+ - \mathrm{M}\boldsymbol{\lambda}^- = \boldsymbol{\beta}_1,$$

or equivalently with

$$\boldsymbol{\varphi}^+ - \boldsymbol{\varphi}^- = -\boldsymbol{\tau}_0, \qquad \boldsymbol{\lambda}^+ - \boldsymbol{\lambda}^- = -\boldsymbol{\tau}_1.$$

The corresponding potentials are

$$U_h = \mathrm{D}(s)\mathrm{Q}\boldsymbol{\varphi}^+ - \mathrm{S}(s)\boldsymbol{\lambda}^+, \qquad V_h = \alpha^{-1}\mathrm{S}(s/c)\boldsymbol{\lambda}^- - \mathrm{D}(s/c)\mathrm{Q}\boldsymbol{\varphi}^-$$

The case of multiple scatterers. Imagine that we have scatterers $\Gamma_1, \ldots, \Gamma_K$, where the equations

$$\Delta V_\ell - (s/c_\ell)^2 V = 0 \qquad \text{in } \Omega_\ell,$$

are satisfied. On the boundaries Γ_ℓ, we impose transmission conditions

$$\gamma^+ U - \beta_0 = (\gamma_1^- V_1, \ldots, \gamma_K^- V_K), \qquad \partial_{\mathbf{n}} U^+ - \beta_1 = (\partial_{\mathbf{n},1}^- V_1, \ldots, \partial_{\mathbf{n},L}^- V_K) \qquad \text{on } \Gamma = \Gamma_1 \cup \Gamma_K).$$

The integral operators corresponding to the interior domains are not coupled to each other. This leads to the matrices:

$$\begin{aligned}
\mathrm{V}_{\mathrm{int}}(s) &:= \mathrm{diag}(\alpha_1^{-1}\mathrm{V}_1(s/c_1), \ldots, \alpha_K^{-1}\mathrm{V}_K(s/c_K)),\\
\mathrm{K}_{\mathrm{int}}(s) &:= \mathrm{diag}(\mathrm{K}_1(s/c_1), \ldots, \mathrm{K}_K(s/c_K)),\\
\mathrm{J}_{\mathrm{int}}(s) &:= \mathrm{diag}(\mathrm{J}_1(s/c_1), \ldots, \mathrm{J}_K(s/c_K)),\\
\mathrm{W}_{\mathrm{int}}(s) &:= \mathrm{diag}(\alpha_1\mathrm{W}_1(s/c_1), \ldots, \alpha_K\mathrm{W}_K(s/c_K)),
\end{aligned}$$

where $(\mathrm{V}_\ell(s), \mathrm{K}_\ell(s), \mathrm{J}_\ell(s), \mathrm{W}_\ell(s))$ are the matrices for the Calderón Calculus on the obstacle Γ_ℓ. The corresponding integral system is

$$\begin{bmatrix} \mathrm{W}(s) + \mathrm{W}_{\mathrm{int}}(s) & \mathrm{J}(s) + \mathrm{J}_{\mathrm{int}}(s) \\ -\mathrm{K}(s) - \mathrm{K}_{\mathrm{int}}(s) & \mathrm{V}(s) + \mathrm{V}_{\mathrm{int}}(s) \end{bmatrix} \begin{bmatrix} \boldsymbol{\varphi}^- \\ \boldsymbol{\lambda}^- \end{bmatrix} = \begin{bmatrix} \mathrm{W}(s) & \frac{1}{2}\mathrm{M} + \mathrm{J}(s) \\ \frac{1}{2}\mathrm{M} - \mathrm{K}(s) & \mathrm{V}(s) \end{bmatrix} \begin{bmatrix} \boldsymbol{\tau}_0 \\ \boldsymbol{\tau}_1 \end{bmatrix}, \qquad \begin{aligned} \mathrm{M}\boldsymbol{\tau}_0 &= -\boldsymbol{\beta}_0, \\ \mathrm{M}\boldsymbol{\tau}_1 &= -\boldsymbol{\beta}_1. \end{aligned}$$

Reconstruction of the exterior fields is done with the same rule

$$\boldsymbol{\varphi}^+ = \boldsymbol{\varphi}^- - \boldsymbol{\tau}_0, \qquad \boldsymbol{\lambda}^+ = \boldsymbol{\lambda}^- - \boldsymbol{\tau}_1.$$

3.4 Mixed problems

Let now Γ_S and Γ_H be respective geometries (each of them can be composed of several curves), where Dirichlet and Neumann conditions will be imposed respectively. Let then

$$\boldsymbol{\beta}_{0,S}, \qquad \boldsymbol{\beta}_{1,H}$$

be the sampling of the minus incident wave and its normal derivative on the separate geometries. A discrete model can be built using a single layer potential on Γ_S and a double layer potential on Γ_H

$$U_h = \mathrm{S}_S(s)\boldsymbol{\lambda}_S + \mathrm{D}_H(s)\mathrm{Q}_H\boldsymbol{\varphi}_H,$$

where

$$\begin{bmatrix} \mathrm{V}_S(s) & \mathrm{R}_S\mathrm{K}(s)\mathrm{R}_H^\top \\ \mathrm{R}_H\mathrm{J}(s)\mathrm{R}_S^\top & -\mathrm{W}_H(s) \end{bmatrix} \begin{bmatrix} \boldsymbol{\lambda}_S \\ \boldsymbol{\varphi}_H \end{bmatrix} = \begin{bmatrix} \boldsymbol{\beta}_{0,S} \\ \boldsymbol{\beta}_{1,H} \end{bmatrix}.$$

Here:

- Operators and potentials subscripted with $\circ \in \{S, H\}$ correspond to the Calderón Calculus on $\Gamma_\circ$.
- $\mathrm{R}_\circ$ is the restriction matrix to $\Gamma_\circ$ with $\circ \in \{S.H\}$
- Unsubscripted operators correspond to the Calderón Calculus on $\Gamma = \Gamma_S \cup \Gamma_H$.

Chapter 4

Open arcs

4.1 Cosine transform sampling

Parametrization

Let $\mathbf{x} : [0,1] \to \mathbb{R}^2$ be the parametrization of a smooth simple open arc. Let also

$$\phi(t) := \tfrac{1}{2} + \tfrac{1}{2}\cos(\pi(2t-1)),$$

which is a 1-periodic even function $\phi : \mathbb{R} \to [0,1]$. We then define

$$\mathbf{a}(t) := (\mathbf{x} \circ \phi)(t) = \mathbf{x}\big(\tfrac{1}{2} + \tfrac{1}{2}\cos(\pi(2t-1))\big),$$

and note that

$$\mathbf{a}(0) = \mathbf{x}(0) = \mathbf{a}(1), \qquad \mathbf{a}(\tfrac{1}{2}) = \mathbf{x}(1), \qquad \mathbf{a}(1-t) = \mathbf{a}(t) \quad \forall t.$$

The normal vector field

$$\mathbf{n}(t) := \phi'(t)(\mathbf{x}' \circ \phi)^\perp(t) = -\pi\sin(\pi(2t-1))\mathbf{x}'\big(\tfrac{1}{2} + \tfrac{1}{2}\cos(\pi(2t-1))\big)^\perp, \qquad (c_1, c_2)^\perp = (c_2, -c_1),$$

satisfies

$$\mathbf{n}(0) = \mathbf{n}(\tfrac{1}{2}) = \mathbf{0}, \qquad \mathbf{n}(1-t) = -\mathbf{n}(t), \quad \forall t,$$

and the latter property implies that the orientation of the normal vector depends on whether we are going from $\mathbf{x}(0)$ to $\mathbf{x}(1)$ or back.

Sampling

Let N be a positive integer, $h := 1/(2N)$ and

$$t_j := j\,h + \tfrac{1}{2}\,h, \qquad t_j^\pm := (j \pm \tfrac{1}{6})h + \tfrac{1}{2}h, \qquad j \in \mathbb{Z}_{2N}^* = \{0, \ldots, 2N-1\} \qquad (j \in \mathbb{Z}).$$

Note that

$$t_{2N-1-j} = 1 - t_j, \qquad t_{2N-1-j}^\pm = 1 - t_j^\mp.$$

Therefore the midpoints

$$\mathbf{m}_j := \mathbf{a}(t_j) \qquad \mathbf{m}_j^\pm := \mathbf{a}(t_j^\pm)$$

satisfy

$$\mathbf{m}_{2N-1-j} = \mathbf{m}_j, \qquad \mathbf{m}_{2N-1-j}^\pm = \mathbf{m}_j^\mp.$$

The normal vectors

$$\mathbf{n}_j := h\mathbf{n}(t_j), \qquad \mathbf{n}_j^\pm := h\mathbf{n}(t_j^\pm)$$

satisfy

$$\mathbf{n}_{2N-1-j} = -\mathbf{n}_j, \qquad \mathbf{n}^{\pm}_{2N-1-j} = -\mathbf{n}^{\mp}_j.$$

To sample the break points, we define

$$s_j := t_j - \tfrac{1}{2}h = j\,h, \qquad s_j := t^{\pm}_j - \tfrac{1}{2}h = (j \pm \tfrac{1}{6})h, \qquad j \in \mathbb{Z}^*_{2N}$$

and

$$\mathbf{b}_j := \mathbf{a}(s_j), \qquad \mathbf{b}^{\pm}_j := \mathbf{a}(s^{\pm}_j).$$

(Note that $\mathbf{b}_0 = \mathbf{x}(0)$ and $\mathbf{b}_N = \mathbf{x}(1)$.) Since

$$s_{2N-j} = 1 - s_j, \qquad s^{\pm}_{2N-j} = 1 - s^{\mp}_j,$$

it follows that

$$\mathbf{b}_{2N-j} = \mathbf{b}_j, \qquad \mathbf{b}^{\pm}_{2N-j} = \mathbf{b}^{\mp}_j.$$

Parity matrix

Let H be a $(2N) \times (2N)$ *sparse* matrix, defined by

$$\mathrm{H}_{i,i} = 1, \qquad \mathrm{H}_{i,2N-1-i} = -1, \qquad \mathrm{H}_{i,j} = 0 \quad \text{otherwise}.$$

Given a vector $\boldsymbol{\xi} \in \mathbb{C}^{2N}$, we say that the vector is even when $\xi_{2N-1-j} = \xi_j$, and therefore $\mathrm{H}\boldsymbol{\xi} = \mathbf{0}$. A vector is odd when $\xi_{2N-1-j} = -\xi_j$, and therefore $\mathrm{H}\boldsymbol{\xi} = 2\boldsymbol{\xi}$. For an open arc sampled as `g`, this matrix is stored in the field `g.parity`. Its absolute value `abs(g.parity)` corresponds to the matrix $|\mathrm{H}|$ that satisfies $|\mathrm{H}|\boldsymbol{\xi} = \mathbf{0}$ for odd vectors and $|\mathrm{H}|\boldsymbol{\xi} = 2\boldsymbol{\xi}$ for even vectors.
Any vector can be written as

$$\boldsymbol{\xi} = \boldsymbol{\xi}_{\text{even}} + \boldsymbol{\xi}_{\text{odd}}, \qquad \boldsymbol{\xi}_{\square} \in \mathbb{C}^{2N}_{\square} \qquad \square \in \{\text{even}, \text{odd}\}.$$

How to input data

The functions $\mathbf{x}$ and $\mathbf{x}'$ have to be given in a way that when the input is a column vector of N values of t, the output is an $N \times 2$ matrix.

```
function g = openarc(N,ep,x,xp)

% function g = openarc(N,ep,x,x')
% Input:
%       N      :      number of segments
%       ep     :      epsilon parameter
%       x      :      parameterization vector of the curve
%       xp     :      first derivative of x
% Output:
%       g      :      discrete geometry
% Last Modified: August 2, 2013

h = 1/(2*N);
t = h*(0:2*N-1);
t = t+(ep+0.5)*h;

T = @(tau) 0.5*cos(pi*(2*tau-1))+0.5;
TP = @(tau) -pi*sin(pi*(2*tau-1));
xT = x(T(t'));
xpT = xp(T(t'));

g.midpt = xT;
g.brkpt = x(T(t'-h/2));
g.normal = h*[xpT(:,2).*TP(t'), -xpT(:,1).*TP(t')];
```

```
g.next = [2:2*N 1];
g.comp = [1];

g.parity = speye(2*N);
g.parity = g.parity-g.parity(end:-1:1,:);

return
```

Here are two examples of how to produce discrete geometries with this function.

```
v=[1 0];w=[3 2];   % a segment
g=openarc(10,0,@(t) (1-t)*v+t*w, @(t) (1+0*t)*(w-v));

part=1.8;   % part=1 is a half circle; part=0.5 is a quarter
xs   = @(s)[cos(part*pi*s),sin(part*pi*s)];
xps  = @(s) part*[-pi*sin(part*pi*s),pi*cos(part*pi*s)];
g = openarc(10,0,xs,xps);
```

4.2 Helmholtz Calderón Calculus on open arcs

The Calderón Calculus for open arcs is limited to the layer potentials and the operators $\mathrm{V}(s)$ and $\mathrm{W}(s)$.

The single layer potential

Given points $\mathbf{z}_i \in \mathbb{R}^2 \setminus \Gamma$, the single layer potential from a sampled open arc is represented by the matrix

$$\mathrm{S}_{i,j}(s) = \frac{\imath}{4} H_0^{(1)}(\imath s|\mathbf{z}_i - \mathbf{m}_j|), \qquad j \in \mathbb{Z}_{2N}^*.$$

Since $\mathbf{m}_{2N-1-j} = \mathbf{m}_j$, then

$$\mathrm{S}_{i,2N-1-j}(s) = \mathrm{S}_{i,j}(s),$$

and it follows that

$$\mathrm{S}(s)\boldsymbol{\eta} = \mathrm{S}(s)\boldsymbol{\eta}_{\mathrm{even}}.$$

The single layer operator

Note that

$$\mathbf{m}_{2N-1-j} = \mathbf{m}_j, \qquad \mathbf{m}^{\pm}_{2N-1-i} = \mathbf{m}^{\mp}_i, \qquad \mathbf{m}^{+}_{2N-2-i} = \mathbf{m}^{-}_{i+1}, \qquad \mathbf{m}^{-}_{2N-i} = \mathbf{m}^{+}_{i-1}.$$

This implies that the matrix

$$\mathrm{V}_{ij}(s) = \mathrm{P}^{+}\mathrm{V}^{+}_{ij}(s) + \mathrm{P}^{-}\mathrm{V}^{-}_{ij}(s), \qquad \mathrm{V}^{\pm}_{ij}(s) = \frac{\imath}{4} H_0^{(1)}(\imath s|\mathbf{m}^{\pm}_i - \mathbf{m}_j|),$$

satisfies

$$\mathrm{V}_{i,2N-1-j}(s) = \mathrm{V}_{i,j}(s) = \mathrm{V}_{2N-1-i,j}(s)$$

which implies

$$\mathrm{V}(s)\boldsymbol{\eta} = \mathrm{V}(s)\boldsymbol{\eta}_{\mathrm{even}} \in \mathbb{C}^{2N}_{\mathrm{even}} \qquad \forall \boldsymbol{\eta} \in \mathbb{C}^{2N}.$$

The double layer potential

Given points $\mathbf{z}_i \in \mathbb{R}^2 \setminus \Gamma$, the double layer potential from a sampled open arc is represented by the matrix

$$\mathrm{D}_{i,j}(s) = -\frac{s}{4} H_1^{(1)}(\imath s|\mathbf{z}_i - \mathbf{m}_j|)\frac{(\mathbf{z}_i - \mathbf{m}_j)\cdot \mathbf{n}_j}{|\mathbf{z}_i - \mathbf{m}_j|}, \qquad j \in \mathbb{Z}_{2N}^*.$$

Since $\mathbf{m}_{2N-1-j} = \mathbf{m}_j$ and $\mathbf{n}_{2N-1-j} = -\mathbf{n}_j$, then

$$\mathrm{D}_{i,2N-1-j}(s) = \mathrm{D}_{i,j}(s),$$

and it follows that

$$\mathrm{D}(s)\boldsymbol{\varphi} = \mathrm{D}(s)\boldsymbol{\varphi}_{\mathrm{odd}}.$$

Moreover, it is easy to show that

$$\mathrm{Q}\boldsymbol{\varphi} \in \mathbb{C}^{2N}_{\square} \qquad \forall \boldsymbol{\varphi} \in \mathbb{C}^{2N}_{\square}, \qquad \square \in \{\mathrm{even}, \mathrm{odd}\},$$

and therefore

$$\mathrm{D}(s)\mathrm{Q}\boldsymbol{\varphi} = \mathrm{D}(s)\mathrm{Q}\boldsymbol{\varphi}_{\mathrm{odd}}.$$

The hypersingular operator

The hypersingular operator on a sampled open arc is a sum of two operators: the principal part and the regular part. Consider first the matrices

$$\mathrm{V}_{\mathbf{n}}(s) = \mathrm{P}^+\mathrm{V}^+_{\mathbf{n}}(s) + \mathrm{P}^-\mathrm{V}^-_{\mathbf{n}}(s), \qquad \mathrm{V}_{\mathbf{n},ij}(s) := s^2\mathbf{n}_i^{\pm}\cdot \mathbf{n}_j \mathrm{V}^{\pm}_{ij}(s),$$

then

$$\mathrm{V}_{\mathbf{n},i,2N-1-j}(s) = -\mathrm{V}_{\mathbf{n},i,j}(s) = \mathrm{V}_{\mathbf{n},2N-1-i,j}(s),$$

and finally

$$\mathrm{Q}\mathrm{V}_{\mathbf{n}}(s)\mathrm{Q}\boldsymbol{\varphi} = \mathrm{Q}\mathrm{V}_{\mathbf{n}}(s)\mathrm{Q}\boldsymbol{\varphi}_{\mathrm{odd}} \in \mathbb{C}^{2N}_{\mathrm{odd}} \qquad \forall \boldsymbol{\varphi} \in \mathbb{C}^{2N}.$$

Consider next the matrices

$$\widetilde{\mathrm{W}}(s) = \tfrac{1}{2}\widetilde{\mathrm{W}}^+(s) + \tfrac{1}{2}\widetilde{\mathrm{W}}^-(s), \qquad \widetilde{\mathrm{W}}^{\pm}(s) = (\mathrm{E}^\top - \mathrm{I})\widetilde{\mathrm{V}}^{\pm}(s)(\mathrm{E} - \mathrm{I}),$$

where

$$\widetilde{\mathrm{V}}^{\pm}_{ij}(s) = (\imath s|\mathbf{b}_i^{\pm} - \mathbf{b}_j|),$$

and

$$\mathrm{E}_{i,j} = \delta_{i,n(j)} = \delta_{n^{-1}(i),j}, \qquad (\mathrm{E}\boldsymbol{\xi})_i = \xi_{n^{-1}(i)}.$$

Since $\mathbf{b}_{2N-j} = \mathbf{b}_j$, then $\widetilde{\mathrm{V}}^{\pm}_{i,2N-j}(s) = \widetilde{\mathrm{V}}^{\pm}_{ij}(s)$, which implies that

$$\widetilde{\mathrm{V}}^{\pm}_{i,2N-1-j+1}(s) - \widetilde{\mathrm{V}}_{i,2N-1-j}(s) = -\big(\widetilde{\mathrm{V}}^{\pm}_{i,j+1}(s) - \widetilde{\mathrm{V}}^{\pm}_{i,j}(s)\big)$$

and therefore

$$\widetilde{\mathrm{W}}(s)\boldsymbol{\varphi} = \widetilde{\mathrm{W}}(s)\boldsymbol{\varphi}_{\mathrm{odd}}.$$

With similar arguments, we can prove that

$$\widetilde{\mathrm{W}}(s)\boldsymbol{\varphi} \in \mathbb{C}^{2N}_{\mathrm{odd}} \qquad \forall \boldsymbol{\varphi} \in \mathbb{C}^{2N}.$$

Chapter 5

The Laplace Calderón Calculus

The sampling of closed curves, mass matrix M, the quadrature matrix Q, and the testing device for right-hand sides are taken verbatim from the Helmholtz Calderón Calculus.

5.1 Potentials and operators

After sampling the curves (this does not change from the Helmholtz Calderón Calculus), the single and double layer potentials observec at points $\{\mathbf{z}_1, \ldots, \mathbf{z}_M\}$ are given by

$$\begin{aligned}
\mathrm{S}_{ij} &:= -\frac{1}{2\pi} \log |\mathbf{z}_i - \mathbf{m}_j| \\
\mathrm{D}_{ij} &:= -\frac{1}{2\pi} \frac{(\mathbf{z}_i - \mathbf{m}_j) \cdot \mathbf{n}_j}{|\mathbf{z}_i - \mathbf{m}_j|^2}
\end{aligned}$$

These are computed with the `LaplacePotentials` function.

```
function [SL,DL]=LaplacePotentials(g,z)

% [SL,DL]=LaplacePotentials(g,z)
%
% Input:
%     g : geometry
%     z : K x 2 matrix with points where potentials are evaluated
% Output:
%   SL  : matrix for SL
%   DL  : matrix for DL
% Last modified: September 4, 2013

% SL : single layer potential

DX = bsxfun(@minus,z(:,1),g.midpt(:,1)');
DY = bsxfun(@minus,z(:,2),g.midpt(:,2)');
D = sqrt(DX.^2+DY.^2);
SL = -1/(2*pi)*log(D);

% DL : double layer potential

N = bsxfun(@times,DX,g.normal(:,1)')...
    +bsxfun(@times,DY,g.normal(:,2)');
DL = 1/(2*pi)*N./D.^2;

return
```

The one sided (half) integral operators for the Laplacian are given by

$$
\begin{aligned}
\mathrm{V}^{\pm}_{ij} &:= -\frac{1}{2\pi}\log|\mathbf{m}^{\pm}_i - \mathbf{m}_j|,\\
\mathrm{K}^{\pm}_{ij} &:= \frac{1}{2\pi}\frac{(\mathbf{m}^{\pm}_i - \mathbf{m}_j)\cdot \mathbf{n}_j}{|\mathbf{m}^{\pm}_i - \mathbf{m}_j|^2},\\
\mathrm{J}^{\pm}_{ij} &:= \frac{1}{2\pi}\frac{(\mathbf{m}_j - \mathbf{m}^{\pm}_i)\cdot \mathbf{n}^{\pm}_i}{|\mathbf{m}^{\pm}_i - \mathbf{m}_j|^2},\\
\mathrm{W}^{\pm}_{ij} &:= \widetilde{\mathrm{V}}^{\pm}_{n(i),n(j)} - \widetilde{\mathrm{V}}^{\pm}_{n(i),j} - \widetilde{\mathrm{V}}^{\pm}_{i,n(j)} + \widetilde{\mathrm{V}}^{\pm}_{i,j},
\end{aligned}
$$

where

$$\widetilde{\mathrm{V}}^{\pm}_{ij} := -\frac{1}{2\pi}\log|\mathbf{b}^{\pm}_i - \mathbf{m}_j|.$$

We also consider matrices

$$\mathrm{C}^{\pm}_{ij} := \begin{cases} |\mathbf{n}^{\pm}_i|\,|\mathbf{n}_j| & \text{if } (i,j) \text{ in the same component,}\\ 0 & \text{otherwise.}\end{cases}$$

These matrices are built in a subfunction of the main function. The final version of the operators is given by

$$
\begin{aligned}
\mathrm{V} &:= \mathrm{P}^+\mathrm{V}^+ + \mathrm{P}^-\mathrm{V}^-,\\
\mathrm{K} &:= (\mathrm{P}^+\mathrm{K}^+ + \mathrm{P}^-\mathrm{K}^-)\mathrm{Q},\\
\mathrm{J} &:= \mathrm{Q}(\mathrm{P}^+\mathrm{J}^+ + \mathrm{P}^-\mathrm{J}^-),\\
\mathrm{W} &:= \mathrm{P}^+\mathrm{W}^+ + \mathrm{P}^-\mathrm{W}^-,\\
\mathrm{C} &:= \mathrm{Q}(\mathrm{P}^+\mathrm{C}^+ + \mathrm{P}^-\mathrm{C}^-)\mathrm{Q}.
\end{aligned}
$$

The goal of C is to help to make $\mathrm{W}+\mathrm{C}$ coercive. The exact operator W has a kernel with the constant functions on each of the connected components of the curve, and is coercive in the quotien space. The matrix C is used to impose zero integral on each of the connected components of the curve.

```
function [V,K,J,W,C] = CalderonCalculusLaplace(g,gp,gm,varargin)

% [V,K,J,W,C] = CalderonCalculusLaplace(g,gp,gm)
% [V,K,J,W,C] = CalderonCalculusLaplace(g,gp,gm,fork)
% Input:
%     g : principal geometry
%    gp : companion geometry with epsilon = 1/6
%    gm : companion geometry with epsilon = −1/6
%    fork : 0 or absent (averaged method) ≠0 (mixed method)
% Output:
%     V : single layer operator V
%     K : double layer operator K
%     J : transpose of operator K
%     W : hypersingular operator W
%     C : rank Ncomp perturbation
%
% Last Modified: October 31, 2013

[Vp,Kp,Jp,Wp,Cp] = CalderonCalculusLaplaceHalf(g,gp);
[Vm,Km,Jm,Wm,Cm] = CalderonCalculusLaplaceHalf(g,gm);

fork=0;
if nargin==2
    fork=varargin{1};
end
[Q,¬,Pp,Pm] = CalderonCalculusMatrices(g,fork);
```

```
V = Pp*Vp+Pm*Vm;
K = (Pp*Kp+Pm*Km)*Q;
J = Q*(Pp*Jp+Pm*Jm);
W = Pp*Wp+Pm*Wm;
C = Q*(Pp*Cp+Pm*Cm)*Q;

return

% Subfunction computing the two halves of the operators

function [V,K,J,W,C] = CalderonCalculusLaplaceHalf(g,gp)

% [V,K,J,W,C] = CalderonCalculusLaplaceHalf(g,gp)
% Input:
%     g : principal geometry
%    gp : companion geometry
% Output:
%     V : single layer operator
%     K : double layer operator
%     J : transpose of double layer operator
%     W : hypersingular operator
%     C : rank Ncomp perturbation
%
% Last Modified: September 4, 2013

% V = Single layer operator

DX = bsxfun(@minus,gp.midpt(:,1),g.midpt(:,1)');
DY = bsxfun(@minus,gp.midpt(:,2),g.midpt(:,2)');
D = sqrt(DX.^2+DY.^2);                    % |m_i^ep-m_j|
V = -1/(2*pi)*log(D);

% K = Double layer operator

N = bsxfun(@times,DX,g.normal(:,1)')...
    +bsxfun(@times,DY,g.normal(:,2)');
K = 1/(2*pi)*N./D.^2;

% J = Transposed double layer operator

N = bsxfun(@times,gp.normal(:,1),DX)...
     +bsxfun(@times,gp.normal(:,2),DY);
J = -1/(2*pi)*N./D.^2;

% W = Hypersingular operator

DX = bsxfun(@minus,gp.brkpt(:,1),g.brkpt(:,1)');
DY = bsxfun(@minus,gp.brkpt(:,2),g.brkpt(:,2)');
D  = sqrt(DX.^2+DY.^2);                          % |b_i^ep-b_j|
W  = -1/(2*pi)*(log(D(gp.next,g.next))+log(D)...
                -log(D(gp.next,:))-log(D(:,g.next)));

% C = rank Ncomp perturbation

lengths=sum(g.normal.^2,2);
lengthsp=sum(gp.normal.^2,2);
N = size(g.midpt,1);
g.comp=[g.comp N+1];
C =sparse(N,N);
for c=1:length(g.comp)-1
    list=g.comp(c):g.comp(c+1)-1;
    C(list,list)=lengthsp(list)*lengths(list)';
end

return
```

5.2 Examples

If

$$\Delta U = 0 \qquad \text{in } \Omega^+, \qquad U = c_\infty + a r^{-1} + \mathcal{O}(r^{-2}) \qquad \text{as } r \to \infty,$$

then we can write the following representations:

- As a single layer potential when c_∞
$$U = \mathrm{S}\lambda,$$
- As a corrected single layer potential
$$U = \mathrm{S}\lambda + c_\infty,$$
with λ having vanishing integral over Γ.
- As a double layer potential (with density defined up to a constant on each connected component of the curve), only in the decaying case $c_\infty = 0$
$$U = \mathrm{D}\varphi.$$
- Using Green's representation formula
$$U = c_\infty - \mathrm{S}\partial_\nu^+ U + \mathrm{D}\gamma^+ U.$$

Here is a collection of already discretized integral formulations for the exterior Dirichlet problem:

- Indirect SL formulation (valid for the decaying case, and invertible when the logarithmic capacity of Γ is not one)
$$\mathrm{V}\boldsymbol{\lambda} = \boldsymbol{\beta}_0, \qquad U_h = \mathrm{S}\boldsymbol{\lambda}.$$
- Indirect SL formulation
$$\begin{bmatrix} \mathrm{V} & \mathbf{1} \\ \mathbf{1}^\top & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{\lambda} \\ c_\infty \end{bmatrix} = \begin{bmatrix} \boldsymbol{\beta}_0 \\ 0 \end{bmatrix}, \qquad U_h = \mathrm{S}\boldsymbol{\lambda} + c_\infty.$$
Here $\mathbf{1}$ is a vector of ones. With the same matrix, if the last component of the RHS is set to c_0, we get the asymptotic behavior
$$U_h = -\frac{|\Gamma|}{2\pi}\log r + c_\infty + \mathcal{O}(r^{-1}).$$
- Direct formulation based on the first BIE
$$\mathrm{M}\boldsymbol{\varphi} = \boldsymbol{\beta}_0, \qquad \begin{bmatrix} \mathrm{V} & -\mathbf{1} \\ \mathbf{1}^\top & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{\lambda} \\ c_\infty \end{bmatrix} = \begin{bmatrix} -\frac{1}{2}\mathrm{M}\boldsymbol{\varphi} + \mathrm{K}\boldsymbol{\varphi} \\ 0 \end{bmatrix}, \qquad U_h = \mathrm{DQ}\boldsymbol{\varphi} - \mathrm{S}\boldsymbol{\lambda} + c_\infty.$$
In this case $\boldsymbol{\lambda}_j \approx \nabla U(\mathbf{m}_j)\cdot \mathbf{n}_j$.
- Symmetric formulation
$$\begin{bmatrix} \mathrm{V} & -\frac{1}{2}\mathrm{M} - \mathrm{K} & -\mathbf{1} \\ \frac{1}{2}\mathrm{M} + \mathrm{J} & \mathrm{W} + \mathrm{C} & \mathbf{0} \\ \mathbf{1}^\top & \mathbf{0}^\top & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{\lambda} \\ \boldsymbol{\varphi} \\ c_\infty \end{bmatrix} = \begin{bmatrix} -\boldsymbol{\beta}_0 \\ \mathbf{0} \\ 0 \end{bmatrix}, \qquad U_h = \mathrm{DQ}\boldsymbol{\varphi} - \mathrm{S}\boldsymbol{\lambda} + c_\infty.$$
In this case $\boldsymbol{\lambda}_j \approx \nabla U(\mathbf{m}_j)\cdot\mathbf{n}_j$ and $(\mathrm{Q}\boldsymbol{\varphi})_j \approx U(\mathbf{m}_j) + c_\ell$, where c_ℓ depends on the connected component and cannot be computed unless we use another method to adjust. In any case, this is the Dirichlet problem and we do not need an approximation of the Dirichlet data.

Now a collection of already discretized integral formulations for the Neumann problem. In this case, $c_\infty = 0$, since otherwise, there are more solutions.

- Indirect DL formulation:

$$(\mathrm{W}+\mathrm{C})\boldsymbol{\varphi} = -\boldsymbol{\beta}_1, \qquad U_h = \mathrm{DQ}\boldsymbol{\varphi}.$$

- Direct formulation based on the 2nd BIE:

$$\mathrm{M}\boldsymbol{\lambda} = \boldsymbol{\beta}_1, \qquad (\mathrm{W}+\mathrm{C})\boldsymbol{\varphi} = -(\tfrac{1}{2}\mathrm{M}+\mathrm{J})\boldsymbol{\lambda}, \qquad U_h = \mathrm{DQ}\boldsymbol{\varphi} - \mathrm{S}\boldsymbol{\lambda}.$$

Chapter 6

The Navier-Lamé Calderón Calculus

The methods for the linear elasticity and time-harmonic linear elasticity equations follow from [2]. Note that this reference deals only with elastic waves. However, all the arguments are applicable to quasistatic elasticity.

There are some significant differences between the Calderón Calculus for the Navier–Lamé equations

$$-\mathrm{div}(\mu(\mathrm{D}\mathbf{U} + (\mathrm{D}\mathbf{U})^\top) + \lambda \mathrm{div}\mathbf{U}\,\mathrm{I}) = 0$$

and the Laplace related calculus.

- At the organization level, every scalar entry is automatically subtituted by a 2×2 block. However, these blocks will be spread so that matrices have $(N + N) \times (N + N)$ form.
- From the point of view of discretization, only the mixing method $\alpha = \frac{5}{6}$ gives a third order scheme. The averaging method gives only first order.

6.1 Potentials and operators

Given M observation points $\{\mathbf{z}_1, \ldots, \mathbf{z}_M\}$, the Lamé single and double layer potentials are given by

$$\begin{aligned}
\underline{\mathrm{S}}_{ij} :=& -\frac{1+A}{4\pi\mu} \log |\mathbf{z}_i - \mathbf{m}_j| \mathrm{I}_{2\times 2} + \frac{1-A}{4\pi\mu} \frac{1}{|\mathbf{z}_i - \mathbf{m}_j|^2} (\mathbf{z}_i - \mathbf{m}_j) \otimes (\mathbf{z}_i - \mathbf{m}_j),\\
\underline{\mathrm{D}}_{ij} :=& \frac{A}{2\pi} \left(\frac{(\mathbf{z}_i - \mathbf{m}_j) \cdot \mathbf{n}_j}{|\mathbf{z}_i - \mathbf{m}_j|^2} \mathrm{I}_{2\times 2} + \frac{(\mathbf{z}_i - \mathbf{m}_j) \cdot \mathbf{n}_j^\perp}{|\mathbf{z}_i - \mathbf{m}_j|^2} \mathrm{J}_{2\times 2} \right)\\
&+ \frac{1-A}{\pi} \frac{(\mathbf{z}_i - \mathbf{m}_j) \cdot \mathbf{n}_j}{|\mathbf{z}_i - \mathbf{m}_j|^4} (\mathbf{z}_i - \mathbf{m}_j) \otimes (\mathbf{z}_i - \mathbf{m}_j),
\end{aligned}$$

where

$$A = \frac{\mu}{\lambda + 2\mu}, \qquad \mathrm{J}_{2\times 2} = \left[\begin{array}{cc} 0 & 1 \\ -1 & 0 \end{array} \right] \qquad \text{and} \qquad \mathbf{n}_j^\perp = -\mathrm{J}_{2\times 2} \mathbf{n}_j = (-n_j^y, n_j^x).$$

Note that on a single curve $\mathbf{n}_j = h\mathbf{x}'(t_j)$ is the scaled tangential vector. As already mentioned above, these blocks are stored in an $(M + M) \times (N + N)$ matrix, that is, the matrix blocks are of size $M \times N$, and there are 2×2 of them.

```
function [SL,DL]=LamePotentials(g,z,mu,lambda)

% [SL,DL]=LamePotentials(g,z,mu,lambda)
```

```
%
% Input:
%     g : geometry
%     z : K x 2 matrix with points where potentials are evaluated
%     mu, lambda : Lame parameters
% Output:
%    SL  : matrix for SL
%    DL  : matrix for DL
% Last modified: November 5, 2013

A = mu/(lambda+2*mu);

RX   = bsxfun(@minus,z(:,1),g.midpt(:,1)');
RY   = bsxfun(@minus,z(:,2),g.midpt(:,2)');
RXNX = bsxfun(@times,RX,g.normal(:,1)');
RXNY = bsxfun(@times,RX,g.normal(:,2)');
RYNX = bsxfun(@times,RY,g.normal(:,1)');
RYNY = bsxfun(@times,RY,g.normal(:,2)');
RN   = RXNX+RYNY;
RT   = -RXNY+RYNX;
R = sqrt(RX.^2+RY.^2);
O = zeros(size(R));

SL = (1+A)/(4*pi*mu)*[-log(R) O; O -log(R)]...
     +(1-A)/(4*pi*mu)*repmat(1./R.^2,[2 2])...
                     .*[RX.*RX  RX.*RY; RY.*RX RY.*RY];
DL = A/(2*pi)*repmat(1./R.^2,[2 2]).*([RN O; O RN]+[O RT; -RT O])...
     +(1-A)/pi*repmat(RN./R.^4,[2 2])...
              .*[RX.*RX RX.*RY; RY.*RX RY.*RY];

return
```

The four one-sided discrete integral operators are given by:

$$
\begin{aligned}
\underline{\mathrm{V}}_{ij}^{\pm} &:= -\frac{1+A}{4\pi\mu}\log|\mathbf{m}_i^{\pm}-\mathbf{m}_j|\,\mathrm{I}_{2\times2} + \frac{1-A}{4\pi\mu}\frac{1}{|\mathbf{m}_i^{\pm}-\mathbf{m}_j|^2}(\mathbf{m}_i^{\pm}-\mathbf{m}_j)\otimes(\mathbf{m}_i^{\pm}-\mathbf{m}_j)\\
\underline{\mathrm{K}}_{ij}^{\pm} &:= \frac{A}{2\pi}\frac{(\mathbf{m}_i^{\pm}-\mathbf{m}_j)\cdot\mathbf{n}_j}{|\mathbf{m}_i^{\pm}-\mathbf{m}_j|^2}\,\mathrm{I}_{2\times2} + \frac{A}{2\pi}\frac{(\mathbf{m}_i^{\pm}-\mathbf{m}_j)\cdot\mathbf{n}_j^{\perp}}{|\mathbf{m}_i^{\pm}-\mathbf{m}_j|^2}\,\mathrm{J}_{2\times2}\\
&\quad + \frac{1-A}{\pi}\frac{(\mathbf{m}_i^{\pm}-\mathbf{m}_j)\cdot\mathbf{n}_j}{|\mathbf{m}_i^{\pm}-\mathbf{m}_j|^4}(\mathbf{m}_i^{\pm}-\mathbf{m}_j)\otimes(\mathbf{m}_i^{\pm}-\mathbf{m}_j)\\
\underline{\mathrm{J}}_{ij}^{\pm} &:= \frac{A}{2\pi}\frac{(\mathbf{m}_j-\mathbf{m}_i^{\pm})\cdot\mathbf{n}_i^{\pm}}{|\mathbf{m}_i^{\pm}-\mathbf{m}_j|^2}\,\mathrm{I}_{2\times2} - \frac{A}{2\pi}\frac{(\mathbf{m}_j-\mathbf{m}_i^{\pm})\cdot(\mathbf{n}_i^{\pm})^{\perp}}{|\mathbf{m}_i^{\pm}-\mathbf{m}_j|^2}\,\mathrm{J}_{2\times2}\\
&\quad + \frac{1-A}{\pi}\frac{(\mathbf{m}_j-\mathbf{m}_i^{\pm})\cdot\mathbf{n}_i^{\pm}}{|\mathbf{m}_i^{\pm}-\mathbf{m}_j|^4}(\mathbf{m}_i^{\pm}-\mathbf{m}_j)\otimes(\mathbf{m}_i^{\pm}-\mathbf{m}_j)\\
\underline{\widetilde{\mathrm{V}}}_{ij}^{\pm} &:= \frac{A(\lambda+\mu)}{\pi}\Big(-\log|\mathbf{b}_i^{\pm}-\mathbf{b}_j|\,\mathrm{I}_{2\times2} + \frac{1}{|\mathbf{b}_i^{\pm}-\mathbf{b}_j|^2}(\mathbf{b}_i^{\pm}-\mathbf{b}_j)\otimes(\mathbf{b}_i^{\pm}-\mathbf{b}_j)\Big)\\
\underline{\mathrm{W}}_{ij}^{\pm} &:= \underline{\widetilde{\mathrm{V}}}_{n(i),n(j)}^{\pm} - \underline{\widetilde{\mathrm{V}}}_{n(i),j}^{\pm} - \underline{\widetilde{\mathrm{V}}}_{i,n(j)}^{\pm} + \underline{\widetilde{\mathrm{V}}}_{i,j}^{\pm}
\end{aligned}
$$

A rank $3N$ perturbation is used to cancel out the kernel of the hypersingular operator (the rigid motions):

$$
\mathrm{C}_{ij}^{\pm} := \begin{cases} |\mathbf{n}_i^{\pm}|\,|\mathbf{n}_j|\big(\mathrm{I}_{2\times2} + (\mathbf{m}_i^{\pm})^{\perp}\otimes\mathbf{m}_j^{\perp}\big), & \text{if } (i,j) \text{ in the same component,}\\ 0, & \text{otherwise.}\end{cases}
$$

Mixing and quadrature is done with the usual formulas, after having expanded the matrices to

$$
\begin{bmatrix} \mathrm{P}^{\pm} & \mathrm{O} \\ \mathrm{O} & \mathrm{P}^{\pm} \end{bmatrix}, \qquad \text{and} \qquad \begin{bmatrix} \mathrm{Q} & \mathrm{O} \\ \mathrm{O} & \mathrm{Q} \end{bmatrix}.
$$

Note that the mass matrix has to be expanded in the same form.

```
function [V,K,J,W,C] = CalderonCalculusLame(g,gp,gm,mu,lambda)

% [V,K,J,W,C] = CalderonCalculusLame(g,gp,gm,mu,lambda)
% Input:
%     g : principal geometry
%    gp : companion geometry with epsilon = 1/6
%    gm : companion geometry with epsilon = -1/6
%    mu,lambda : Lame parameters
% Output:
%     V : single layer operator
%     K : double layer operator K
%     J : transpose of operator K
%     W : hypersingular operator W
%     C : projection on local rigid motions
%
% Last Modified: November 4, 2013

[Vp,Kp,Jp,Wp,Cp] = CalderonCalculusLameHalf(g,gp,mu,lambda);
[Vm,Km,Jm,Wm,Cm] = CalderonCalculusLameHalf(g,gm,mu,lambda);

[Q,~,Pp,Pm] = CalderonCalculusMatrices(g,1);
O = zeros(size(Q));
Q  = [Q O; O Q];
Pp = [Pp O;O Pp];
Pm = [Pm O;O Pm];
V = Pp*Vp+Pm*Vm;
K = (Pp*Kp+Pm*Km)*Q;
J = Q*(Pp*Jp+Pm*Jm);
W = Pp*Wp+Pm*Wm;
C = Q*(Pp*Cp+Pm*Cm)*Q;

return

% Subfunction computing the two halves of the operators

function [V,K,J,W,P] = CalderonCalculusLameHalf(g,gp,mu,lambda)

% [V,K,J,W,P] = CalderonCalculusLameHalf(g,gp,mu,lambda)
% Input:
%     g : principal geometry
%    gp : companion geometry
%    mu,lambda : Lame parameters
% Output:
%     V : single layer operator
%     K : double layer operator
%     J : transpose of double layer operator
%     W : hypersingular operator
%     P : projection on rigid motions
%
% Last Modified: November 4, 2013

A = mu/(lambda+2*mu);

% V and K = First row of Calderon Projector

RX   = bsxfun(@minus,gp.midpt(:,1),g.midpt(:,1)');
RY   = bsxfun(@minus,gp.midpt(:,2),g.midpt(:,2)');
RXNX = bsxfun(@times,RX,g.normal(:,1)');
RXNY = bsxfun(@times,RX,g.normal(:,2)');
RYNX = bsxfun(@times,RY,g.normal(:,1)');
RYNY = bsxfun(@times,RY,g.normal(:,2)');
RN   = RXNX+RYNY;
RT   = -RXNY+RYNX;
R = sqrt(RX.^2+RY.^2);
O = zeros(size(R));
```

```matlab
V = (1+A)/(4*pi*mu)*[-log(R) O; O -log(R)]...
    +(1-A)/(4*pi*mu)*repmat(1./R.^2,[2 2])...
                     .*[RX.*RX  RX.*RY; RY.*RX RY.*RY];
K = A/(2*pi)*repmat(1./R.^2,[2 2]).*([RN O; O RN]+[O RT; -RT O])...
    +(1-A)/pi*repmat(RN./R.^4,[2 2]).*[RX.*RX  RX.*RY; RY.*RX  RY.*RY];

% J = Transposed double layer operator

RX = -RX; RY = -RY;
NXRX = bsxfun(@times,gp.normal(:,1),RX);
NYRX = bsxfun(@times,gp.normal(:,2),RX);
NXRY = bsxfun(@times,gp.normal(:,1),RY);
NYRY = bsxfun(@times,gp.normal(:,2),RY);
NR   = NXRX+NYRY;
TR   = -NYRX+NXRY;

J = A/(2*pi)*repmat(1./R.^2,[2 2]).*([NR O; O NR]+[O -TR; TR O])...
    +(1-A)/pi*repmat(NR./R.^4,[2 2]).*[RX.*RX  RX.*RY; RY.*RX  RY.*RY];

% W = Hypersingular operator

RX = bsxfun(@minus,gp.brkpt(:,1),g.brkpt(:,1)');
RY = bsxfun(@minus,gp.brkpt(:,2),g.brkpt(:,2)');
R  = sqrt(RX.^2+RY.^2);                          % |b_i^ep-b_j|
W  = A*(lambda+mu)/pi*([-log(R) O;O -log(R)]...
         +repmat(1./R.^2,[2 2]).*[RX.*RX  RX.*RY; RY.*RX  RY.*RY]);
next = [g.next length(g.next)+g.next];   % next index in two components
nextp = [gp.next length(gp.next)+gp.next];

W = W(nextp,next)-W(nextp,:)-W(:,next)+W;

% Projection onto rigid motions

l=sum(g.normal.^2,2); lp=sum(gp.normal.^2,2);
x=l.*g.midpt(:,1); xp=lp.*gp.midpt(:,1);
y=l.*g.midpt(:,2); yp=lp.*gp.midpt(:,2);
N = size(g.midpt,1);
g.comp=[g.comp N+1];
C = sparse(N,N);
XX = C; XY = C; YX = C; YY= C;
for c=1:length(g.comp)-1
    list=g.comp(c):g.comp(c+1)-1;
    C(list,list)=lp(list)*l(list)';
    XX(list,list)=xp(list)*x(list)';
    YY(list,list)=yp(list)*y(list)';
    XY(list,list)=xp(list)*y(list)';
    YX(list,list)=yp(list)*x(list)';
end
O = sparse(N,N);
P = [C O; O C]+[YY -YX; -XY XX];

return
```

We also include a source solution and its associated stress tensor. Given a source points $\mathbf{x}_0$ and a direction $\mathbf{d}$, then

$$\mathbf{u}(\mathbf{z};\mathbf{x}_0,\mathbf{d}) = C_1 \log|\mathbf{z}-\mathbf{x}_0|\mathbf{d} + C_2\frac{(\mathbf{z}-\mathbf{x}_0)\cdot\mathbf{d}}{|\mathbf{z}-\mathbf{x}_0|^2}(\mathbf{z}-\mathbf{x}_0),$$

where

$$C_1 = -\frac{1+A}{4\pi\mu}, \qquad C_2 = \frac{1-A}{4\pi\mu},$$

is a solution of the Lamé equations in $\mathbb{R}^d \setminus \{\mathbf{x}_0\}$. In particular, given two different points $\mathbf{x}_0 \neq \mathbf{x}_1$,

$$\mathbf{u}(\cdot;\mathbf{x}_0,\mathbf{d}) - \mathbf{u}(\cdot;\mathbf{x}_1,\mathbf{d})$$

is a decaying solution of the Lamé equations that can be used as test for exterior problems. The Jacobian matrix for $\mathbf{u}$ is

$$\begin{aligned}\mathrm{D}\mathbf{u}(\mathbf{z}) =& C_1\frac{1}{|\mathbf{z}-\mathbf{x}_0|^2}\mathbf{d}\otimes(\mathbf{z}-\mathbf{x}_0)+C_2\frac{1}{|\mathbf{z}-\mathbf{x}_0|^2}(\mathbf{z}-\mathbf{x}_0)\otimes\mathbf{d}\\ &-2C_2\frac{(\mathbf{z}-\mathbf{x}_0)\cdot\mathbf{d}}{|\mathbf{z}-\mathbf{x}_0|^4}(\mathbf{z}-\mathbf{x}_0)\otimes(\mathbf{z}-\mathbf{x}_0)+C_2\frac{(\mathbf{z}-\mathbf{x}_0)\cdot\mathbf{d}}{|\mathbf{z}-\mathbf{x}_0|^2}\,\mathrm{I}_{2\times 2},\end{aligned}$$

and the associated stress is

$$\mu(\mathrm{D}\mathbf{u}+(\mathrm{D}\mathbf{u})^\top)+\lambda\mathrm{trace}(\mathrm{D}\mathbf{u})\,\mathrm{I}_{2\times 2}.$$

```
function [u,v,sxx,sxy,syy]=sourceSolutionLame(mu,lambda,x0,d)

% [u,v,sxx,sxy,syy]=sourceSolutionLame(mu,lambda,[x0,y0],[d1 d2])
% Input:
%     mu,lambda : Lame parameters
%     [x0,y0]   : source point
%     [d1,d2]   : direction of displacement vector
% Output:
%    Five vectorized functions of (x,y), corresponding to a source solution
%    of the Lame equation and its corresponding stress tensor
% Last modified: October 31, 2013

xx = x0(1);
yy = x0(2);
d1 = d(1);
d2 = d(2);

rr = @(x,y) (x-xx).^2+(y-yy).^2;
rd = @(x,y) (x-xx)*d1+(y-yy)*d2;

A  = mu/(lambda+2*mu);
C1 = -(1+A)/(4*pi*mu);
C2 = (1-A)/(4*mu*pi);

u = @(x,y) C1*0.5*log(rr(x,y))*d1...
           +C2*rd(x,y)./rr(x,y).*(x-xx);
v = @(x,y) C1*0.5*log(rr(x,y))*d2...
           +C2*rd(x,y)./rr(x,y).*(y-yy);
ux = @(x,y) (C1+C2)*d1*(x-xx)./rr(x,y)...
            -2*C2*rd(x,y)./(rr(x,y).^2).*(x-xx).^2 ...
            +C2*rd(x,y)./rr(x,y);
uy = @(x,y) C1*d1*(y-yy)./rr(x,y)...
            +C2*d2*(x-xx)./rr(x,y)...
            -2*C2*rd(x,y)./(rr(x,y).^2).*(x-xx).*(y-yy);
vx = @(x,y) C1*d2*(x-xx)./rr(x,y)...
            +C2*d1*(y-yy)./rr(x,y)...
            -2*C2*rd(x,y)./(rr(x,y).^2).*(y-yy).*(x-xx);
vy = @(x,y) (C1+C2)*d2*(y-yy)./rr(x,y)...
            -2*C2*rd(x,y)./(rr(x,y).^2).*(y-yy).^2 ...
            +C2*rd(x,y)./rr(x,y);

sxx = @(x,y) 2*mu*ux(x,y)+lambda*(ux(x,y)+vy(x,y));
sxy = @(x,y) mu*(uy(x,y)+vx(x,y));
syy = @(x,y) 2*mu*vy(x,y)+lambda*(ux(x,y)+vy(x,y));
return
```

6.2 Some examples of use

Finding good exact solutions for the Lamé equations is not entirely obvious. If $\underline{U} : \Omega_+ \to \mathbb{R}^2$ is a bounded exterior solution of the Lamé equations, then

$$\underline{U} = \underline{c}_\infty - \mathrm{S}\underline{\sigma}^+ + \mathrm{D}\gamma^+\underline{U} = \underline{c}_\infty + \mathrm{S}\underline{\lambda},$$

and

$$\int_\Gamma \underline{\lambda} = \underline{0} = \int_\Gamma \underline{\sigma}^+.$$

Here $\underline{\sigma}^+ : \Gamma \to \mathbb{R}^2$ is the exterior normal stress. For the **exterior Dirichlet problem**, we can try the following formulations:

- A straightforward single layer representation

  $$\underline{U}_h = \mathrm{S}\underline{\lambda}, \qquad \mathrm{V}\underline{\lambda} = \underline{\beta}_0.$$

  The operator V might not be invertible. This formulation can represent decaying $\mathcal{O}(1/r)$ solutions of the problem, and also unbounded point-source solutions.

- A well-posed single layer representation

  $$\begin{bmatrix} \mathrm{V} & \underline{1} \\ \underline{1}^\top & \mathrm{O} \end{bmatrix} \begin{bmatrix} \underline{\lambda} \\ \underline{c}_\infty \end{bmatrix} = \begin{bmatrix} \underline{\beta}_0 \\ \underline{0} \end{bmatrix},$$

  where

  $$\underline{1}^\top = \begin{bmatrix} 1 & \dots & 1 & 0 & \dots & 0 \\ 0 & \dots & 0 & 1 & \dots & 1 \end{bmatrix}$$

- A direct formulation

  $$\mathrm{M}\underline{\varphi} = \underline{\beta}_0, \qquad \mathrm{V}\underline{\lambda} = (-\tfrac{1}{2}\mathrm{M} + \mathrm{K})\underline{\varphi}, \qquad \underline{U}_h = \mathrm{DQ}\underline{\varphi} - \mathrm{S}\underline{\lambda}.$$

  The operator equation might not be solvable. Also, this representation can reproduce solutions that behave like a point source solution plus a decaying solution at infinity.

- A well posed direct formulation

  $$\mathrm{M}\underline{\varphi} = \underline{\beta}_0, \qquad \begin{bmatrix} \mathrm{V} & \underline{1} \\ \underline{1}^\top & \mathrm{O} \end{bmatrix} \begin{bmatrix} \underline{\lambda} \\ \underline{c}_\infty \end{bmatrix} = \begin{bmatrix} (-\tfrac{1}{2}\mathrm{M} + \mathrm{K})\underline{\varphi} \\ \underline{0} \end{bmatrix}, \qquad \underline{U}_h = \mathrm{DQ}\underline{\varphi} - \mathrm{S}\underline{\lambda} + \underline{c}_\infty.$$

Tow options are now given for the **exterior Neumann problem**. Note that for this problem, only decaying solutions can be computed (otherwise, there are too many solutions).

- A direct formulation

  $$\mathrm{M}\underline{\lambda} = \underline{\beta}_1, \qquad (\mathrm{W} + \mathrm{C})\underline{\varphi} = -(\tfrac{1}{2}\mathrm{M} + \mathrm{J})\underline{\lambda}, \qquad \underline{U}_h = \mathrm{DQ}\underline{\varphi} - \mathrm{S}\underline{\lambda}.$$

  Note that $\underline{\varphi}$ is not a good approximation of the exterior trace, since the operator W has been stabilized.

- A symmetric (but way too complicated) formulation. This is the kind of formulation that is used for coupling purposes:

  $$\begin{bmatrix} \mathrm{V} & \tfrac{1}{2}\mathrm{M} - \mathrm{K} \\ -\tfrac{1}{2}\mathrm{M} + \mathrm{J} & \mathrm{W} \end{bmatrix} \begin{bmatrix} \underline{\lambda} \\ \underline{\varphi} \end{bmatrix} = \begin{bmatrix} \underline{0} \\ -\underline{\beta}_1 \end{bmatrix}, \qquad \underline{U}_h = \mathrm{DQ}\underline{\varphi} - \mathrm{S}\underline{\lambda}.$$

  In this case $\mathrm{Q}\underline{\varphi}$ is an approximation of the trace. This formulation is well posed when V is invertible.

Chapter 7

The elastodynamic Calderón Calculus

Just as in the static case, every scalar entry is automatically subtituted by a 2×2 block so that matrices have $(N+N) \times (N+N)$ form. Only the mixing method $\alpha = \frac{5}{6}$ gives a third order scheme with the averaging method giving only first order.

7.1 Potentials and operators

Let $K_n(\cdot)$ be the modified Bessel function of the second kind of order n, also known as the Macdonald function. Then

$$c_L = \sqrt{(\lambda + 2\mu)/\rho}\ , \qquad c_T = \sqrt{\mu/\rho}$$

are the respective speeds of pressure (longitudinal) and shear (transverse) waves. The following combinations of the Lamé parameters will be useful:

$$\xi := c_T/c_L, \qquad \nu = \frac{\lambda}{2(\lambda+\mu)}, \qquad C = 2(1-\nu).$$

We introduce the auxiliary functions

$$\begin{aligned}
\psi(r) &:= K_0(r/c_T) + \frac{c_T}{r}\big(K_1(r/c_T) - \xi K_1(r/c_L)\big). \\
\partial_r \psi(r) &= -(1/c_T)K_1(r/c_T) - \frac{2c_T}{r^2}\Big(K_1(r/c_T) - \xi K_1(r/c_L)\Big) - \frac{1}{r}\Big(K_0(r/c_T) - \xi^2 K_0(r/c_L)\Big). \\
\chi(r) &:= K_2(r/c_T) - \xi^2 K_2(r/c_L). \\
\partial_r \chi(r) &= -(1/2c_T)\Big(K_1(r/c_T) + K_3(r/c_T) - \xi^3\big(K_1(r/c_L) + K_3(r/c_L)\big)\Big).
\end{aligned}$$

Given M observation points $\{\mathbf{z}_1, \ldots, \mathbf{z}_M\}$, the time-harmonic elastic single and double layer potentials are given by:

$$\underline{\mathrm{S}}_{ij}(s) := \frac{1}{2\pi\mu}\left(\psi(s|\mathbf{z}_i-\mathbf{m}_j|)\mathbf{I}_{2\times 2} - \frac{\chi(s|\mathbf{z}_i-\mathbf{m}_j|)}{|\mathbf{z}_i-\mathbf{m}_j|^2}(\mathbf{z}_i-\mathbf{m}_j)\otimes(\mathbf{z}_i-\mathbf{m}_j)\right),$$

$$\begin{aligned}\underline{\mathrm{D}}_{ij}(s) := &-\frac{s\partial_r\psi(s|\mathbf{z}_i-\mathbf{m}_j|)}{2\pi|\mathbf{z}_i-\mathbf{m}_j|}\Big(((\mathbf{z}_i-\mathbf{m}_j)\cdot\mathbf{n}_j)\mathbf{I}_{2\times 2}+\mathbf{n}_j\otimes(\mathbf{z}_i-\mathbf{m}_j)+(\lambda/\mu)(\mathbf{z}_i-\mathbf{m}_j)\otimes\mathbf{n}_j\Big)\\ &+\frac{1}{2\pi}\chi(s|\mathbf{z}_i-\mathbf{m}_j|)\Big(-\frac{4(\mathbf{z}_i-\mathbf{m}_j)\cdot\mathbf{n}_j}{|\mathbf{z}_i-\mathbf{m}_j|^4}(\mathbf{z}_i-\mathbf{m}_j)\otimes(\mathbf{z}_i-\mathbf{m}_j)+\frac{1}{|\mathbf{z}_i-\mathbf{m}_j|^2}\,\mathbf{n}_j\otimes(\mathbf{z}_j-\mathbf{m}_i)\\ &\qquad+\frac{(\mathbf{z}_i-\mathbf{m}_j)\cdot\mathbf{n}_j}{|\mathbf{z}_i-\mathbf{m}_j|^2}\mathbf{I}_{2\times 2}+\frac{(2+\lambda/\mu)}{|\mathbf{z}_i-\mathbf{m}_j|^2}(\mathbf{z}_i-\mathbf{m}_j)\otimes\mathbf{n}_j\Big)\\ &+\frac{s}{2\pi}\partial_r\chi(s|\mathbf{z}_i-\mathbf{m}_j|)\left(\frac{2(\mathbf{z}_i-\mathbf{m}_j)\cdot\mathbf{n}_j}{|\mathbf{z}_i-\mathbf{m}_j|^3}(\mathbf{z}_i-\mathbf{m}_j)\otimes(\mathbf{z}_i-\mathbf{m}_j)+\frac{(\lambda/\mu)}{|\mathbf{z}_i-\mathbf{m}_j|}(\mathbf{z}_i-\mathbf{m}_j)\otimes\mathbf{n}_j\right).\end{aligned}$$

```
function [SL,DL]=ElasticWavePotentials(g,z,mu,lambda,rho)

% [SL,DL]=ElasticWavePotentials(g,z,mu,lambda,rho)
%
% Input:
%     g : geometry
%     z : K x 2 matrix with points where potentials are evaluated
%     mu, lambda : Lame parameters
%     rho : density
% Output:
%   SL  : matrix valued function of the variable s
%   DL  : matrix valued function of the variable s
%
% Last modified: July 15, 2014.

% Parameters

cT = sqrt(mu/rho);
cL = sqrt((lambda+2*mu)/rho);
xi = sqrt(mu/(lambda+2*mu));    % (cT/cL)


% Basic Matrix blocks

R1   = bsxfun(@minus,z(:,1),g.midpt(:,1)');
R2   = bsxfun(@minus,z(:,2),g.midpt(:,2)');
RaRb = [R1.*R1 R1.*R2; R2.*R1 R2.*R2];
R1N1 = bsxfun(@times,R1,g.normal(:,1)');
R1N2 = bsxfun(@times,R1,g.normal(:,2)');
R2N1 = bsxfun(@times,R2,g.normal(:,1)');
R2N2 = bsxfun(@times,R2,g.normal(:,2)');
RbNa = [R1N1 R2N1; R1N2 R2N2];
RaNb = [R1N1 R1N2; R2N1 R2N2];
RN   = R1N1+R2N2;
R = sqrt(R1.^2+R2.^2);
O = zeros(size(R));

% Utilities

Id = @(A)  [A O; O A]; % Block Identity
Sc = @(A)  [A A; A A]; % "Scalar" Matrix

% Auxiliary functions

psi   = @(r) besselk(0,r/cT)...
          +(cT./r).*(besselk(1,r/cT)-xi*besselk(1,r/cL));
```

```
psi_r = @(r) -(1/cT)*besselk(1,r/cT)...
             -(2*cT./r.^2).*(besselk(1,r/cT)-xi*besselk(1,r/cL))...
             -(1./r).*(besselk(0,r/cT)-xi^2*besselk(0,r/cL));

chi   = @(r) besselk(2,r/cT)-xi^2*besselk(2,r/cL);

chi_r = @(r) -0.5/cT*...
            (besselk(1,r/cT)+besselk(3,r/cT)...
             -xi^3*(besselk(1,r/cL)+besselk(3,r/cL)));

% Elastic wave Potentials

SL = @(s) 1/(2*pi*mu)*(Id(psi(s*R))-Sc(chi(s*R)./R.^2).*RaRb);

DL = @(s) -s/(2*pi)*Sc(psi_r(s*R)./R).*( Id(RN)+RbNa+(lambda/mu)*RaNb )...
          +1/(2*pi)*Sc(chi(s*R)).*( -4*Sc(RN./R.^4).*RaRb...
                                    +Sc(1./R.^2).*RbNa + Id(RN./R.^2)...
                                    +(2+lambda/mu)*Sc(1./R.^2).*RaNb )...
          +s/(2*pi)*Sc(chi_r(s*R)).*( 2*Sc(RN./R.^3).*RaRb ...
                                       +lambda/mu*Sc(1./R).*RaNb );


return
```

The first three Calderón Calculus operators, $\underline{\mathrm{V}}_{ij}^{\pm}(s)$, $\underline{\mathrm{K}}_{ij}^{\pm}(s)$ and $\underline{\mathrm{J}}_{ij}^{\pm}(s)$ are given by:

$$\underline{\mathrm{V}}_{ij}^{\pm}(s) := \frac{1}{2\pi\mu}\left(\psi(s|\mathbf{m}_i^{\pm}-\mathbf{m}_j|)\mathbf{I}_{2\times 2} - \frac{\chi(s|\mathbf{m}_i^{\pm}-\mathbf{m}_j|)}{|\mathbf{m}_i^{\pm}-\mathbf{m}_j|^2}(\mathbf{m}_i^{\pm}-\mathbf{m}_j)\otimes(\mathbf{m}_i^{\pm}-\mathbf{m}_j)\right),$$

$$\begin{aligned}
\underline{\mathrm{K}}_{ij}^{\pm}(s) := &-\frac{s\partial_r\psi(s|\mathbf{m}_i^{\pm}-\mathbf{m}_j|)}{2\pi|\mathbf{m}_i^{\pm}-\mathbf{m}_j|}\Big((\mathbf{m}_i^{\pm}-\mathbf{m}_j)\cdot\mathbf{n}_j\,\mathbf{I}_{2\times 2} + \mathbf{n}_j\otimes(\mathbf{m}_i^{\pm}-\mathbf{m}_j) + (\lambda/\mu)(\mathbf{m}_i^{\pm}-\mathbf{m}_j)\otimes\mathbf{n}_j\Big)\\
&+\frac{1}{2\pi}\chi(s|\mathbf{m}_i^{\pm}-\mathbf{m}_j|)\Big(-\frac{4(\mathbf{m}_i^{\pm}-\mathbf{m}_j)\cdot\mathbf{n}_j}{|\mathbf{m}_i^{\pm}-\mathbf{m}_j|^4}(\mathbf{m}_i^{\pm}-\mathbf{m}_j)\otimes(\mathbf{m}_i^{\pm}-\mathbf{m}_j) + \frac{1}{|\mathbf{m}_i^{\pm}-\mathbf{m}_j|^2}(\mathbf{m}_i^{\pm}-\mathbf{m}_j)\otimes\mathbf{n}_i\\
&\qquad + \frac{(\mathbf{m}_i^{\pm}-\mathbf{m}_j)\cdot\mathbf{n}_j}{|\mathbf{m}_i^{\pm}-\mathbf{m}_j|^2}\mathbf{I}_{2\times 2} + \frac{(2+\lambda/\mu)}{|\mathbf{m}_i^{\pm}-\mathbf{m}_j|^2}(\mathbf{m}_i^{\pm}-\mathbf{m}_j)\otimes\mathbf{n}_j\Big)\\
&+\frac{s}{2\pi}\partial_r\chi(s|\mathbf{m}_i^{\pm}-\mathbf{m}_j|)\Big(\frac{2(\mathbf{m}_i^{\pm}-\mathbf{m}_j)\cdot\mathbf{n}_j}{|\mathbf{m}_i^{\pm}-\mathbf{m}_j|^3}(\mathbf{m}_i^{\pm}-\mathbf{m}_j)\otimes(\mathbf{m}_i^{\pm}-\mathbf{m}_j) + \frac{(\lambda/\mu)}{|\mathbf{m}_i^{\pm}-\mathbf{m}_j|}(\mathbf{m}_i^{\pm}-\mathbf{m}_j)\otimes\mathbf{n}_j\Big).
\end{aligned}$$

$$\begin{aligned}
\underline{\mathrm{J}}_{ij}^{\pm}(s) := &-\frac{s\partial_r\psi(s|\mathbf{m}_j-\mathbf{m}_i^{\pm}|)}{2\pi|\mathbf{m}_i^{\pm}-\mathbf{m}_j|}\Big((\mathbf{m}_j-\mathbf{m}_i^{\pm})\cdot\mathbf{n}_i^{\pm}\,\mathbf{I}_{2\times 2} + (\mathbf{m}_j-\mathbf{m}_i^{\pm})\otimes\mathbf{n}_i^{\pm} + (\lambda/\mu)\,\mathbf{n}_i^{\pm}\otimes(\mathbf{m}_j-\mathbf{m}_i^{\pm})\Big)\\
&+\frac{1}{2\pi}\chi(s|\mathbf{m}_j-\mathbf{m}_i^{\pm}|)\Big(-\frac{4(\mathbf{m}_j-\mathbf{m}_i^{\pm})\cdot\mathbf{n}_i^{\pm}}{|\mathbf{m}_j-\mathbf{m}_i^{\pm}|^4}(\mathbf{m}_j-\mathbf{m}_i^{\pm})\otimes(\mathbf{m}_j-\mathbf{m}_i^{\pm}) + \frac{1}{|\mathbf{m}_j-\mathbf{m}_i^{\pm}|^2}\,\mathbf{n}_i^{\pm}\otimes(\mathbf{m}_j-\mathbf{m}_i^{\pm})\\
&\qquad + \frac{(\mathbf{m}_j-\mathbf{m}_i^{\pm})\cdot\mathbf{n}_i^{\pm}}{|\mathbf{m}_j-\mathbf{m}_i^{\pm}|^2}\mathbf{I}_{2\times 2} + \frac{(2+\lambda/\mu)}{|\mathbf{m}_j-\mathbf{m}_i^{\pm}|^2}\,\mathbf{n}_i^{\pm}\otimes(\mathbf{m}_j-\mathbf{m}_i^{\pm})\Big)\\
&+\frac{s}{2\pi}\partial_r\chi(s|\mathbf{m}_j-\mathbf{m}_i^{\pm}|)\Big(\frac{2(\mathbf{m}_j-\mathbf{m}_i^{\pm})\cdot\mathbf{n}_i^{\pm}}{|\mathbf{m}_j-\mathbf{m}_i^{\pm}|^3}(\mathbf{m}_j-\mathbf{m}_i^{\pm})\otimes(\mathbf{m}_j-\mathbf{m}_i^{\pm}) + \frac{(\lambda/\mu)}{|\mathbf{m}_j-\mathbf{m}_i^{\pm}|}\,\mathbf{n}_i^{\pm}\otimes(\mathbf{m}_j-\mathbf{m}_i^{\pm})\Big).
\end{aligned}$$

The hypersingular operator $\underline{\mathrm{W}}_{ij}^{\pm}(s)$ is slightly more complicated and requires splitting into a regular and singular part, $\underline{\mathrm{WR}}_{ij}^{\pm}(s)$ and $\underline{\mathrm{WS}}_{ij}^{\pm}(s)$, such that

$$\underline{\mathrm{W}}_{ij}^{\pm}(s) = \underline{\mathrm{WR}}_{ij}^{\pm}(s) + \underline{\mathrm{WS}}_{ij}^{\pm}(s).$$

We introduce the following auxiliary functions:

$$G(r) := \frac{1}{2\pi\rho}\big(K_0(r/c_T) - K_0(r/c_L)\big)\ ,$$
$$F(\mathbf{x}) := \frac{1}{s^2}G(sr), \qquad r := |\mathbf{x}|.$$

The operator requires higher order derivatives of this functions, since they involve the special MacDonald functions, we provide them explicitly:

$$G'(r) = \frac{-1}{2\pi\rho c_T}\big(K_1(r/c_T) - \xi K_1(r/c_L)\big),$$

$$G''(r) = \frac{1}{4\pi\rho c_T^2}\bigg(K_0(r/c_T) + K_2(r/c_T) - \xi^2\big(K_0(r/c_L) + K_2(r/c_L)\big)\bigg),$$

$$G'''(r) = \frac{-1}{4\pi\rho c_T^3}\bigg(3K_1(r/c_T) + K_3(r/c_T) - \xi^3\big(3K_1(r/c_L) + K_3(r/c_L)\big)\bigg),$$

$$G^{(iv)}(r) = \frac{1}{2\pi\rho c_T^4}\bigg(\big(3(c_T/r)^2 + 1\big)K_2(r/c_T) - \xi^4\big(3(c_L/r)^2 + 1\big)K_2(r/c_L)\bigg).$$

$$\Delta G(r) = \frac{1}{r}G'(r) + G''(r),$$

$$\Delta^2 G(r) = G^{(iv)}(r) + \frac{2}{r}G'''(r) - \frac{1}{r^2}G''(r) + \frac{1}{r^3}G'(r).$$

For the Hessian matrix $\mathbf{D}^2F(\mathbf{x})$ we define two more auxiliary functions

$$a(r) := G''(r) - \frac{1}{r}G'(r),$$

$$b(r) := \frac{1}{r}G'(r).$$

So that the Hessian of F is given by

$$\mathbf{D}^2F(\mathbf{x}) = \frac{a(sr)}{r^2}\mathbf{x}_\alpha\mathbf{x}_\beta + b(sr)\,\mathbf{I}_{2\times 2}.$$

Finally, we will also need the following:

$$\mathbf{M}_1 := (\mathbf{n}_i^\pm \otimes \mathbf{n}_j)\big((\mathbf{m}_i^\pm - \mathbf{m}_j) \otimes (\mathbf{m}_i^\pm - \mathbf{m}_j)\big) + \big((\mathbf{m}_i^\pm - \mathbf{m}_j) \otimes (\mathbf{m}_i^\pm - \mathbf{m}_j)\big)(\mathbf{n}_i^\pm \otimes \mathbf{n}_j),$$
$$\mathbf{M}_2 := \Big((\mathbf{n}_i^\pm \otimes \mathbf{n}_j)\big((\mathbf{m}_i^\pm - \mathbf{m}_j) \otimes (\mathbf{m}_i^\pm - \mathbf{m}_j)\big)\Big)^\top + \Big(\big((\mathbf{m}_i^\pm - \mathbf{m}_j) \otimes (\mathbf{m}_i^\pm - \mathbf{m}_j)\big)(\mathbf{n}_i^\pm \otimes \mathbf{n}_j)\Big)^\top,$$
$$\mathrm{M}_3 := (\mathbf{n}_i^\pm \otimes \mathbf{n}_j) : \big((\mathbf{m}_i^\pm - \mathbf{m}_j),$$

where ":" corresponds to the Frobenius inner product. Note that, at the implementation level, $\mathbf{M}_1$ and $\mathbf{M}_2$ are *block* transposes of each other and not transposes.

With all the previous definitions in mind, we have

$$\underline{\mathrm{WR}}_{ij}^{\pm}(s) = Cs^2\Bigg(\mu\Delta^2 G(s|\mathbf{m}_i^{\pm}-\mathbf{m}_j|)\Big(\lambda\,\mathbf{n}_i^{\pm}\otimes\mathbf{n}_j+\mu(\mathbf{n}_j\otimes\mathbf{n}_i^{\pm}+(\mathbf{n}_i^{\pm}\cdot\mathbf{n}_j)\,\mathbf{I}_{2\times 2})\Big)$$
$$-\frac{1}{c_L^2}\Bigg(\lambda^2\Delta G(s|\mathbf{m}_i^{\pm}-\mathbf{m}_j|)\,\mathbf{n}_i^{\pm}\otimes\mathbf{n}_j+2\lambda\mu\Big(\frac{a(s|\mathbf{m}_i^{\pm}-\mathbf{m}_j|)}{|\mathbf{m}_i^{\pm}-\mathbf{m}_j|^2}\mathbf{M}_1+2b(s|\mathbf{m}_i^{\pm}-\mathbf{m}_j|)\,\mathbf{n}_i^{\pm}\otimes\mathbf{n}_j\Big)$$
$$+\mu^2\Big(\big(\frac{a(s|\mathbf{m}_i^{\pm}-\mathbf{m}_j|)}{|\mathbf{m}_i^{\pm}-\mathbf{m}_j|^2}\mathrm{M}_3+b(s|\mathbf{m}_i^{\pm}-\mathbf{m}_j|)(\mathbf{n}_i^{\pm}\cdot\mathbf{n}_j)\big)\mathbf{I}_{2\times 2}+(\mathbf{n}_i^{\pm}\cdot\mathbf{n}_j)\,\mathbf{D}^2F(\mathbf{m}_i^{\pm}-\mathbf{m}_j)$$
$$+\frac{a(s|\mathbf{m}_i^{\pm}-\mathbf{m}_j|)}{|\mathbf{m}_i^{\pm}-\mathbf{m}_j|^2}\mathbf{M}_2+sb(s|\mathbf{m}_i^{\pm}-\mathbf{m}_j|)\,\mathbf{n}_j\otimes\mathbf{n}_i^{\pm}\Big)\Bigg)\Bigg),$$

$$\underline{\mathrm{WS}}_{ij}^{\pm}(s) =\mathbf{D}^{\top}\Big(4\mu^2\Delta G(s|\mathbf{b}_i^{\pm}-\mathbf{b}_j|)-\mathbf{D}^2F(\mathbf{b}_i^{\pm}-\mathbf{b}_j)\Big)\mathbf{D}.$$

In the last expression $\mathbf{D}$ is a differentiation matrix.

Finally, recalling the mixing and quadrature matrices $\mathbf{Q}$, $\mathbf{P}^+$ and $\mathbf{P}^-$, the operators are given by mixing as follows:

$$\begin{aligned}
\mathbf{V}(s)_{ij} =&(\mathbf{P}^+\otimes\mathbf{I}_{2\times 2})\mathbf{V}(s)_{ij}^{+}+(\mathbf{P}^-\otimes\mathbf{I}_{2\times 2})\mathbf{V}(s)_{ij}^{-}\ ,\\
\mathbf{K}(s)_{ij} =&\Big((\mathbf{P}^+\otimes\mathbf{I}_{2\times 2})\mathbf{K}(s)_{ij}^{+}+(\mathbf{P}^-\otimes\mathbf{I}_{2\times 2})\mathbf{K}(s)_{ij}^{-}\Big)(\mathbf{Q}\otimes\mathbf{I}_{2\times 2})\ ,\\
\mathbf{J}(s)_{ij} =&(\mathbf{Q}\otimes\mathbf{I}_{2\times 2})\Big((\mathbf{P}^+\otimes\mathbf{I}_{2\times 2})\mathbf{J}(s)_{ij}^{+}+(\mathbf{P}^-\otimes\mathbf{I}_{2\times 2})\mathbf{J}(s)_{ij}^{-}\Big)\ ,\\
\mathbf{W}(s)_{ij} =&(\mathbf{Q}\otimes\mathbf{I}_{2\times 2})\Big((\mathbf{P}^+\otimes\mathbf{I}_{2\times 2})\mathbf{WR}(s)_{ij}^{+}+(\mathbf{P}^-\otimes\mathbf{I}_{2\times 2})\mathbf{WR}(s)_{ij}^{-}\Big)(\mathbf{Q}\otimes\mathbf{I}_{2\times 2})\\
&+(\mathbf{P}^+\otimes\mathbf{I}_{2\times 2})\mathbf{WS}(s)_{ij}^{+}+(\mathbf{P}^-\otimes\mathbf{I}_{2\times 2})\mathbf{WS}(s)_{ij}^{-}.
\end{aligned}$$

```
function [V,K,J,W] = CalderonCalculusElasticWave(g,gp,gm,mu,lambda,rho)

% [V,K,J,W] = CalderonCalculusElasticWave(g,gp,gm,mu,lambda,rho)
% Input:
%     g : principal geometry
%    gp : companion geometry with epsilon = 1/6
%    gm : companion geometry with epsilon = -1/6
%    mu,lambda : Lame parameters
%    rho : density
% Output:
%     V : single layer operator (function of s)
%     K : double layer operator K
%     J : transpose double layer operator K
%     W : hypersingular operator W
%
% Last Modified: March 29, 2016.

[Vp,Kp,Jp,WRp,WSp] = CalderonCalculusEWHalf(g,gp,mu,lambda,rho);
[Vm,Km,Jm,WRm,WSm] = CalderonCalculusEWHalf(g,gm,mu,lambda,rho);

[Q,¬,Pp,Pm] = CalderonCalculusMatrices(g,1);  % Fork = 1
O = zeros(size(Q));
Q  = [Q O; O Q];
Pp = [Pp O;O Pp];
Pm = [Pm O;O Pm];

V = @(s) Pp*Vp(s)+Pm*Vm(s);
K = @(s) (Pp*Kp(s)+Pm*Km(s))*Q;
```

```
J = @(s) Q*(Pp*Jp(s)+Pm*Jm(s));
W = @(s) Q*(Pp*WRp(s)+Pm*WRm(s))*Q + Pp*WSp(s)+Pm*WSm(s);

end

% Subfunction computing the two halves of the operators

function [V,K,J,WR,WS] = CalderonCalculusEWHalf(g,gp,mu,lambda,rho)

% [V,K,J,WR,WS] = CalderonCalculusEWHalf(g,gp,mu,lambda, rho)
% Input:
%     g : principal geometry
%    gp : companion geometry
%    mu,lambda : Lame parameters
%   rho : density
% Output:
%     V : single layer operator (function of s)
%     K : double layer operator
%     J : transpose double layer operator
%     WR: regular part of the hypersingular operator
%     WS: singular part of the hypersingular operator
%
% Last Modified: May 22, 2014.

cT = sqrt(mu/rho);
cL = sqrt((lambda+2*mu)/rho);
nu = .5*lambda/(lambda+mu);
CC = 2*(1-nu);  % CC=(lambda+2*mu)/(lambda+mu)
xi = sqrt(mu/(lambda+2*mu));

% Some basic blocks

R1  = bsxfun(@minus,gp.midpt(:,1),g.midpt(:,1)');
R2  = bsxfun(@minus,gp.midpt(:,2),g.midpt(:,2)');
R = sqrt(R1.^2+R2.^2);
R11 = R1.*R1;
R12 = R1.*R2;
R21 = R12;
R22 = R2.*R2;
RIRJ = [R11 R12; R21 R22];
O = zeros(size(R));

% Utilities

Id = @(A) [A O; O A];
Sc = @(A) [A A; A A];

% Four functions

psi  = @(r) besselk(0,r/cT)...
          +(cT./r).*(besselk(1,r/cT)-xi*besselk(1,r/cL));
psi_r = @(r) -(1/cT)*besselk(1,r/cT)...
             -(2*cT./r.^2).*(besselk(1,r/cT)-xi*besselk(1,r/cL))...
             -(1./r).*(besselk(0,r/cT)-xi^2*besselk(0,r/cL));

chi   = @(r) besselk(2,r/cT)-xi^2*besselk(2,r/cL);
chi_r = @(r) -0.5/cT*(besselk(1,r/cT)+besselk(3,r/cT)...
                        -xi^3*(besselk(1,r/cL)+besselk(3,r/cL)));

% Single layer operator

V = @(s) 1/(2*pi*mu)*(Id(psi(s*R))-Sc(chi(s*R)./R.^2).*RIRJ);

% Double layer operator

R1N1 = bsxfun(@times,R1,g.normal(:,1)');
```

```
R1N2 = bsxfun(@times,R1,g.normal(:,2)');
R2N1 = bsxfun(@times,R2,g.normal(:,1)');
R2N2 = bsxfun(@times,R2,g.normal(:,2)');
RN   = R1N1+R2N2;
RINJ = [R1N1 R1N2; R2N1 R2N2];
NIRJ = [R1N1 R2N1; R1N2 R2N2];


K = @(s) -s/(2*pi)*Sc(psi_r(s*R)./R).*(Id(RN)+NIRJ+(lambda/mu)*RINJ)...
          +1/(2*pi)*Sc(chi(s*R)).*...
                (-4*Sc(RN./R.^4).*RIRJ + Sc(1./R.^2).*NIRJ...
                 +Id(RN./R.^2) + (2+lambda/mu)*Sc(1./R.^2).*RINJ)...
          +s/(2*pi)*Sc(chi_r(s*R)).*...
                (2*Sc(RN./R.^3).*RIRJ+lambda/mu*Sc(1./R).*RINJ);

% Transposed double layer operator

R1 = -R1; R2 = -R2;  % For J,   r = y-x
RJRI = RIRJ;

N1R1 = bsxfun(@times,gp.normal(:,1),R1);
N2R1 = bsxfun(@times,gp.normal(:,2),R1);
N1R2 = bsxfun(@times,gp.normal(:,1),R2);
N2R2 = bsxfun(@times,gp.normal(:,2),R2);
NR   = N1R1+N2R2;
NJRI = [N1R1 N2R1; N1R2 N2R2];
RJNI = [N1R1 N1R2; N2R1 N2R2];

J = @(s) -s/(2*pi)*Sc(psi_r(s*R)./R).*(Id(NR)+NJRI+(lambda/mu)*RJNI)...
          +1/(2*pi)*Sc(chi(s*R)).*...
                (-4*Sc(NR./R.^4).*RJRI+Sc(1./R.^2).*NJRI...
                 +Id(NR./R.^2)+(2+lambda/mu)*Sc(1./R.^2).*RJNI)...
          +s/(2*pi)*Sc(chi_r(s*R)).*...
                (2*Sc(NR./R.^3).*RJRI+lambda/mu*Sc(1./R).*RJNI);

% Hypersingular operator: auxiliary computations

N11 = gp.normal(:,1)*g.normal(:,1)';
N12 = gp.normal(:,1)*g.normal(:,2)';
N21 = gp.normal(:,2)*g.normal(:,1)';
N22 = gp.normal(:,2)*g.normal(:,2)';
NINJ = [N11 N12; N21 N22];
NJNI = [N11 N21; N12 N22];
NdotN = N11+N22;

M1M2 = [N11.*R11+N12.*R21+R11.*N11+R12.*N21,...
            N11.*R12+N12.*R22+R11.*N12+R12.*N22;...
        N21.*R11+N22.*R21+R21.*N11+R22.*N21,...
            N21.*R12+N22.*R22+R21.*N12+R22.*N22];
M3M4 = [N11.*R11+N12.*R21+R11.*N11+R12.*N21,...
            N21.*R11+N22.*R21+R21.*N11+R22.*N21;...
        N11.*R12+N12.*R22+R11.*N12+R12.*N22,...
            N21.*R12+N22.*R22+R21.*N12+R22.*N22];
M5   = N11.*R11+N12.*R12+N21.*R21+N22.*R22;

Gp = @(r) -1/(2*pi*rho*cT)*(besselk(1,r/cT) - xi*besselk(1,r/cL));

Gpp = @(r) 1/(4*pi*rho*cT^2)*...
             ( besselk(0,r/cT)+besselk(2,r/cT)...
                -xi^2*(besselk(0,r/cL)+besselk(2,r/cL)) );

Gppp = @(r) -1/(8*pi*rho*cT^3)*...
            ( 3*besselk(1,r/cT)+besselk(3,r/cT)...
               -xi^3*(3*besselk(1,r/cL)+besselk(3,r/cL)) );

Gpppp = @(r) 1/(2*pi*rho*cT^4)*...
```

```
                ( (3*cT^2./r.^2+1).*besselk(2,r/cT)...
                  -xi^4*(3*cL^2./r.^2+1).*besselk(2,r/cL));
LapG   = @(r) Gp(r)./r+Gpp(r);
BiLapG = @(r) Gpppp(r)+2*Gppp(r)./r-Gpp(r)./r.^2+Gp(r)./r.^3;
a = @(r) Gpp(r)-Gp(r)./r;
b = @(r) Gp(r)./r;

% Regular part of the hypersingular operator

HessF = @(s) Sc(a(s*R)./R.^2).*RIRJ +Id(b(s*R));
WR = @(s) CC*s^2*...
           (mu*Sc(BiLapG(s*R)).*(lambda*NINJ + mu*(NJNI + Id(NdotN)))...
            -(1/cL^2)*(...
                lambda^2*Sc(LapG(s*R)).*NINJ...
                +2*lambda*mu*(Sc(a(s*R)./R.^2).*M1M2+2*Sc(b(s*R)).*NINJ)...
                +mu^2*(Id(a(s*R)./R.^2.*M5+b(s*R).*NdotN)...
                       +Sc(NdotN).*HessF(s)...
                       +Sc(a(s*R)./R.^2).*M3M4+2*Sc(b(s*R)).*NJNI)));

% Principal part of the hypersingular operator

R1  = bsxfun(@minus,gp.brkpt(:,1),g.brkpt(:,1)');
R2  = bsxfun(@minus,gp.brkpt(:,2),g.brkpt(:,2)');
R   = sqrt(R1.^2+R2.^2);

HessFbrk = @(s) Sc(a(s*R)./R.^2).*[R1.*R1 R1.*R2; R2.*R1 R2.*R2]...
                 +Id(b(s*R));

Nelt = size(g.brkpt,1);
Dt = -speye(Nelt)+sparse(1:Nelt,g.next,1);
Dt = [Dt O; O Dt];
WS = @(s) Dt*( 4*mu^2*(Id(LapG(s*R)) - HessFbrk(s)) )*Dt';

end
```

Chapter 8

Plotting utilities

8.1 Meshing around obstacles

This subroutine makes use of the PDE toolbox of Matlab to mesh the exterior (and the interior if required) of a prescribed curves.
The input is

- `r` an exterior rectangle containing the curves in the following format `[xmin xmax ymax ymin]` so that `[xmin ymin]` and `[ xmax ymax ]` are the lower left and the upper right vertices of the rectangle.
- `g` geometry of the curve(s). The subroutine uses `g.midpt`, `g.normal`, and `g.comp`.
- `d_abs,d_rel` (absolute and relative distance) parameters which control how much the grid approaches to the curves. If both of them are set to be zero, the points `g.midpt` are nodes of the grid.
- `hgrid` meshsize of the grid
- `intQ` Optional. If `intQ` is set to be 1, the interior of the curves are meshed also. Otherwise, the grid is constructed only for the exterior.

The output is the constructed mesh:

- `X` and `Y` are $N_{\rm nodes} \times 1$ vectors with the coordinates of the nodes of the triangularion
- `T` is an $N_{\rm elts}\times$. Then first three rows are the connectivity matrix of the mesh. The last row is a index indicating the subdomain number. This number is chosen by MATLAB using its own criteria.
- `sub` is a vector relating the MATLAB choice for the subdomain number with the original numbering: `sub(1)` gives the tag for the exterior domain, `sub(2)` gives the tag for the domain interior to the first obstacle, etc.
- `gBn` is a vector containing the list of nodes located at the boundary of the domain counted globally (i.e. with respect to the full triangulation). Note that the list includes the both nodes located on the original geometry `g` as well as those along the sides of the bounding box.

As mentioned above, the parameters `d_abs,d_rel` control how much the grid approaches to the curve(s). For the exterior grid this is done by constructing first a polygon which is a dilated copy of the curves

$$\mathbf{m}_i + \big(\frac{\texttt{d\_abs}}{|\mathbf{n}_i|} + \texttt{d\_rel}\big)\mathbf{n}_i$$

and next including these points in the boundary nodes of the mesh.

If an interior triangulation of the curves are required, a similar procedure is followed by using now

$$\mathbf{m}_i - \Big(\frac{\texttt{d\_abs}}{|\mathbf{n}_i|} + \texttt{d\_rel}\Big)\mathbf{n}_i$$

as a smaller copy of the curve(s).
Note that both polygons, the bigger and smaller copies of the curves, must not intersect. Otherwise, an error message is displayed by Matlab and the grid is not constructed.

```
function [X,Y,T,sub,gBn] = triangulateGeometry(r, g,d_abs,d_rel,hgrid,varargin)

% [X,Y,T,sub,gBn]=triangulateGeometry([xmin xmax ymax ymin],g,dabs,drel,hgrid,fill)
%
% Input:
%
% [xmin xmax ymax ymin] : framing rectangle
% g                     : geometry of the curve (discrete geometry)
% dabs                  : absolute distance of the grid to the curve
% drel                  : relative distance of the grid to the curve
% hgrid                 : meshgrid size
% fill                  : 1 generate grid for the interior
%                         0 or missing -> interior not meshed
%
% Output:
%
%   X,Y   : Npt x 1 vectors with X and Y coordinates of points
%   T     : Nelements x 4 matrix with elements (nodes and subdomains)
%   sub   : Subdomain ordering
%   gBn   : Column vector containig the -global- list of boundary nodes.
%
%
% Note that you need the PDE toolbox in your Matlab distribution
%
% Last modified: August 29, 2014.

if ¬isempty(varargin)
    intQ=varargin{1};
    intQ=double(intQ);
    intQ=intQ(1);
else
    intQ=0;
end

[pde_fig,ax]=pdeinit;

n_curves=length(g.comp);
n=length(g.midpt);
index=[g.comp n+1];
d_abs=abs(d_abs); d_rel=abs(d_rel);

lengths=sqrt(g.normal(:,1).^2+g.normal(:,2).^2);

counter=1;
pderect(r,'R1');
Op='R1'; Op2=[];

for j=1:n_curves
    counter=counter+1;
    label2=['R' num2str(counter)];
    indAux=[index(j):index(j+1)-1];
    curve=g.midpt(indAux,:)+...
        bsxfun(@times,g.normal(indAux,:),d_abs./lengths(indAux)+d_rel);
    pdepoly(curve(:,1).', curve(:,2).', label2);
    Op=[Op '-' label2];
    if intQ==1
```

```
        curve=g.midpt(indAux,:)-...
            bsxfun(@times,g.normal(indAux,:),d_abs./lengths(indAux)+d_rel);
        counter=counter+1;
        label2=['R' num2str(counter)];
        pdepoly(curve(:,1).', curve(:,2).',label2);
        Op2=[Op2 '+' label2];
        object(j,:)=curve(1,1:2);
    end
end


% Generating the mesh
Op=['(' Op ')' Op2];

pdetool('appl_cb',1);
set(ax,'XLim',[r(1) r(2)]);
set(ax,'YLim',[r(4) r(3)]);
set(ax,'XTickMode','auto');
set(ax,'YTickMode','auto');

set(findobj(get(pde_fig,'Children'),'Tag','PDEEval'),'String',Op)

% Mesh generation:
setappdata(pde_fig,'trisize',hgrid);
setappdata(pde_fig,'Hgrad',1.25+hgrid);
setappdata(pde_fig,'refinemethod','regular');
setappdata(pde_fig,'jiggle',char('on','mean',''));
pdetool('initmesh')

% Extracting the information about the triangulation
h = findobj(get(pde_fig,'Children'),'flat','Tag','PDEMeshMenu');

hp = findobj(get(h,'Children'),'flat','Tag','PDEInitMesh');
p = get(hp,'UserData');  % Node coordinates

ht = findobj(get(h,'Children'),'flat','Tag','PDEMeshParam');
t = get(ht,'UserData');  % Delauney triangulation

he=findobj(get(h,'Children'),'flat','Tag','PDERefine');
e = get(he,'UserData');   % Nodes that define the (boundary) edges
gBn = unique(e(1:2,:));   % Boundary Nodes


X=p(1,:)';
Y=p(2,:)';
T=t';


if intQ==1
    for j=1:n_curves
        vertex=intersect(find(object(j,1)==X),find(object(j,2)==Y));
        [elt,v]=find(T(:,1:3)==vertex);
        sub(j)=T(elt(1),4);
    end
    ext = setdiff(1:n_curves+1,sub);
    sub=[ext sub];
else
    sub=1;
end

return
```

The following script shows how to create, transform and merge three geometries and the create a triangulation around it. The subdomain order is `[ 1 4 2 3]`.

```
% SCRIPT TO TEST THE GENERATE DOMAIN UTILITIES

g1=ellipse(60,0,[0.5 0.4],[0.2,0.5]);

g2=tvshape(60,0);
g2=affine(g2,0.5*[1/sqrt(2) 1/sqrt(2); -1/sqrt(2) 1/sqrt(2)],[-1.5;-2]);

g3=kite(30,0);
A=[cos(pi/4) -sin(pi/4); sin(pi/4) cos(pi/4)]*diag([0.3 0.3]);
g3=affine(g3,A,[1;-1.5]);

g=merge(g1,g2,g3);

indaux=[g.comp length(g.midpt)+1];
clf
hold on
for j=1:length(g.comp)
    aux=g.midpt(indaux(j):indaux(j+1)-1,:);
    plot(aux(:,1),aux(:,2),'o-')
end
axis equal
Box=[-3, 2, 2, -4 ];
d_abs=0.02;
d_rel=0;
h=0.2;
[X,Y,T,sub]=triangulateGeometry(Box,g,d_abs,d_rel,h,1);
disp(sub)

trimesh(T(:,1:3),X,Y,'color','k')
```

Additionally, and thinking of problems where different representations are used in different subdomains, we have the function that separates a triangulation into several subtriangulations, each of them corresponding to a subdomain. The first triangulation corresponds to the exterior domain generated by **triangulateGeometry**. Each of the triangulations is renumbered so that evaluation is only carried out at points that are relevant to the corresponding subdomain. The output is collected in four cell arrays.

```
function [XX,YY,TT,LBN] = separateTriangulation(X,Y,T,sub,gBn)

% [XX,YY,TT,LBN] = separateTriangulation(X,Y,T,sub,gBn)
%
% Input:
%
%   X,Y   : nVert x 1 vectors with coordinates of vertices
%   T     : nElt x 4 with triangulation (4th column is subdomain).
%   sub   : Vector containing the ordering of the subdomains.
%   gBn   : Vector containing the -global- list of boundary nodes.(optional)
%
% Output:
%
%     XX, YY : cell array with X,Y coordinates separated by subdomain.
%     TT     : cell array with triangulation separated by subdomain.
%     LBN    : cell array containing the -local- list of boundary nodes
%              separated by subdomain. (optional only available if gBn is
%              provided as input).
%
% Last modified: August 29, 2014.

nVert = size(X,1);
nSubd = max(T(:,4));

for j = 1:nSubd

    i = find(T(:,4)==sub(j));
    TT{j} = T(i,1:3);
```

```
    vert = TT{j}(:);
    vert = unique(vert);
    XX{j} = X(vert);
    YY{j} = Y(vert);
    transp = zeros(nVert,1);
    transp(vert) = 1:length(vert);
    TT{j} = transp(TT{j});

    % Optional list of boundary nodes
    if nargin == 5
        LBN{j} = vert(ismember(vert,gBn));
        LBN{j} = transp(LBN{j});
    end

end

end
```

In the following example, samples of the circles $x^2 + y^2 = 1$ and $(x-2)^2 + (y-2)^2 = (1.5)^2$ are created and merged (in that particular order). We next triangulate the domain, MATLAB decides to mesh the domains in the order shown in `sub`: first the exterior domain (the part of the box $[-2, 4] \times [-2, 4]$ lying outside the obstacles), then the interior of the circle of radius 1.5 and finally the interior of the smaller circle. After running `separateTriangulation`, the subdomains are numbered as originally: subdomain number one will be the part of the box around the obstacles, subdomain number two will be inside the first circle, and subdomain number three insited the second circle.

```
>> g1=ellipse(20,0,[1 1],[0 0]);
>> g2=ellipse(40,0,[1.5 1.5],[2 2]);
>> g=joinGeometry(g1,g2);
>> [X,Y,T,sub]=triangulateGeometry([-2 4 4 -2],g,0.1,0.1,0.2,1);
>> sub
sub =
     1     3     2
>> [XX,YY,TT]=separateTriangulation(X,Y,T,sub)
XX =
    [1150x1 double]    [129x1 double]    [315x1 double]
YY =
    [1150x1 double]    [129x1 double]    [315x1 double]
TT =
    [2056x3 double]    [216x3 double]    [548x3 double]
```

8.2 Dynamic plots in the frequency domain

Consider a triangulation described with two vectors with the x and y components of the vertices (called `X` and `Y`) and a matrix representing the elements. We are given a complex vector `U` with as many components as `X`. We then plot on the triangulation the values

$$U_{i,n} = \cos(\frac{n}{2\pi N})\mathrm{Re}\, U_i + \sin(\frac{n}{2\pi N})\mathrm{Im}\, U_i \qquad n = 0, \ldots, N-1.$$

The results are then stored in png correlative files. There is an option to include the shape of the obstacles in the picture (the obstacles are represented by a sampled geometry data structure).

```
function frequencyDomainPlot(X,Y,Tri,U,Nsnap,name,g,yes)

% frequencyDomainPlot(X,Y,Tri,U,Nsnap,name,g,yes)
% Input:
%     [X,Y,Tri] : triangulation in the plane
%      U        : complex values of a function on (X,Y)
```

```
%      Nsnap    : number of desired snapshots
%      name     : name of file for plot
%      g        : scatterer
%      yes      : 1 (include the scatterers), 0 (do not)
%
% Last modified: August 6, 2013

% Names of files

indices={'01','02','03','04','05','06','07','08','09',...
         '10','11','12','13','14','15','16','17','18','19',...
         '20','21','22','23','24','25','26','27','28','29',...
         '30','31','32','33','34','35','36','37','38','39',...
         '40','41','42','43','44','45','46','47','48','49',...
         '50','51','52','53','54','55','56','57','58','59',...
         '60','61','62','63','64','65','66','67','68','69',...
         '70'};
for i=1:length(indices)
    longname{i}=[name,indices{i},'.png'];
end
Nsnap=min(Nsnap,length(indices));

% Unpacking the geometries and fixing heights

G=unpackGeometry(g);
nComp=length(G);

Umax=max(real(U)); Umin=min(real(U));
U(1:2)=[];
T=linspace(0,2*pi,Nsnap+1); T(end)=[];

% Running times

i=0;
for t=T
    trisurf(Tri,X,Y,...
            [Umax;Umin;real(U)*cos(t)+imag(U)*sin(t)]);
    view(2), shading interp, axis equal, axis off
    hold on
    if yes
        for k=1:nComp
            plot3(G{k}.midpt(:,1),G{k}.midpt(:,2),0*G{k}.midpt(:,1),'-');
        end
    end
    i=i+1;
    saveas(gcf,longname{i})
    hold off
    pause(0.02)
end
```

Chapter 9

Time domain tools

We outline here the basic time domain tools for deltaBEM. They are implementations of Lubich's Convolution Quadrature (CQ) method [4]. For details about the algorithms, see [3].

A short introduction to CQ. Consider the causal convolution

$$y(t) = \int_0^t f(t-\tau)g(\tau)d\tau \tag{9.1}$$

where y is unknown and f and g are known. We will assume that we are using causal data (i.e. $f(t)$ and $g(t)$ are zero for $t < 0$) and seek a causal solution y. The function f will be used through its Laplace transform $\mathrm{F}(s) = \mathcal{L}\{f(t)\}$. We fix a uniform time step $k > 0$ and a uniform time grid $t_n := nk$ for $n \geq 0$. CQ approximates the forward convolution (9.1) by a discrete convolution

$$y(t_n) = \sum_{m=0}^{n} \omega_m^{\mathrm{F}}(k) g(t_{n-m}),$$

where the convolution weights $\omega_m^{\mathrm{F}}(k)$ are the coefficients of the Taylor series

$$\mathrm{F}\left(\frac{\delta(\zeta)}{k}\right) = \sum_{m=0}^{\infty} \omega_m^{\mathrm{F}}(k)\zeta^m. \tag{9.2}$$

The function $\delta(\zeta)$ is called the transfer function for the CQ method, and is based on an underlying A-stable ODE solver. In the case of BDF2, the transfer function is $\delta(\zeta) = (1-z) + \frac{1}{2}(1-z)^2$. CQ also can be used to solve convolution equations. The continuous convolution equation (with y still unknown) and its CQ discretization are

$$g(t) = \int_0^t f(t-\tau)y(\tau)d\tau \quad \text{and} \quad g(t_n) = \sum_{m=0}^{n} \omega_m^{\mathrm{F}}(k) y(t_{n-m}),$$

respectively.

9.1 Computing forward convolutions

The function `CQforward` approximates the causal convolution $\mathrm{F}(\partial_t)g(t)$ where $g(t)$ is causal data taking values in a Hilbert space and $\mathrm{F}(s)$ is an operator valued distribution depending on the Laplace parameter $s \in \mathbb{C}$. We are using the operational notation of Lubich, i.e.

$$\mathrm{F}(\partial_t)g(t) = \int_0^t f(t-\tau)g(\tau)d\tau, \quad \mathrm{F}(s) = \mathcal{L}\{f(t)\}.$$

Once discretized in space, $\mathrm{F}(s)$ will be a matrix-valued function of s, taking values in $\mathbb{C}^{d_1 \times d_2}$, and $g(t)$ will be a $d_2 \times M+1$ matrix of data discretized in space and sampled at $M+1$ time steps. From this point on, we will assume F and g are already discrete. We will keep the same names for the discrete values. `CQforward` takes as input the following:

- $\mathrm{F}(s) : \mathbb{C} \to \mathbb{C}^{d_1 \times d_2}$, a matrix-valued function handle of the parameter $s \in \mathbb{C}$,
- g : a $d_2 \times M + 1$ matrix with the function g discretized in space and sampled at $M + 1$ time points,
- k: the time step size, $k = T/M$ where T is the final time of the simulation, and
- (optional) p : a function handle with the transfer function for CQ. If no option is passed, the default transfer function is BDF2.

`CQforward` outputs a $d_1 \times M + 1$ matrix with the result of the discrete convolution $u = F(\partial_t^k)g(t_n)$ for $n = 0, \ldots, M$. The forward convolution can be computed in parallel across the time steps (this is the so-called 'all-steps-at-once' method). If the user has Matlab's Parallel Toolbox, the main `parfor` loop in the code is executed in parallel, otherwise it is simply executed serially.

```
function u = CQforward(F,g,k,varargin)

% u = CQforward(F,g,k)
% u = CQforward(F,g,k,p)
%
% Input:
%    F : transfer function (d1 x d2 matrix-valued)
%    g : time values of input (d2 x (N+1) matrix)
%    k : time-step
%    p : transfer function of multistep method
% Output:
%    u : F(\partial_k) g              d1 x (N+1) matrix
%
% Last Modified : June 4, 2014

if nargin==3
    p = @(z) 1.5-2*z+0.5*z.^2;      % The default is BDF2
else
    p=varargin{1};
end

d1 = size(F(1),1);              % number of components of output vector
N  = size(g,2)-1;               % N = number of time-steps

omega = exp(2*pi*1i/(N+1));
R = eps^(0.5/(N+1));

h = bsxfun(@times,g,R.^(0:N));
h = fft(h,[],2);                % DFT by columns (\hat H)
u = zeros(d1,N+1);
parfor l=0:floor((N+1)/2)
    u(:,l+1)=F(p(R*omega^(-l))/k)*h(:,l+1);     % \hat v
end
u(:,N+2-(1:floor(N/2)))=conj(u(:,2:floor(N/2)+1));

u=real(ifft(u,[],2));                           % v
u=bsxfun(@times,u,R.^(-(0:N)));

return
```

9.2 Solving convolution equations

If we are solving a convolution equation, where now the unknown is under the action of a convolution operator, we can use `CQequation`. `CQequation` takes as input:

- F(s): a $d \times d$ matrix depending on the parameter $s \in \mathbb{C}$,
- y: a $d \times M + 1$ matrix with data for the problem,
- k: the time step size, and
- (optional) p: a function handle with the transfer function for CQ. If no option is passed, the default transfer function is BDF2.

`CQequation` outputs a $d \times M + 1$ matrix with the solution to the convolution equation computed at the $M + 1$ time steps. The convolution equation can be solved in parallel across the time steps (this is the so-called 'all-steps-at-once' method). If the user has Matlab's Parallel Toolbox, the main `parfor` loop in the code is executed in parallel, otherwise it is simply executed serially.

```
function g=CQequation(F,h,k,varargin)

% g=CQequation(F,h,k)
% g=CQequation(F,h,k,p)
%
% Input:
%    F : transfer function             d x d matrix valued
%    h : time values of RHS            d x (N+1) matrix
%    k : time-step
%    p : transfer function of multistep method
% Output:
%    g : F(\partial_k)^{-1} u          d x (N+1) matrix
%
% Last Modified : June 4, 2014

if nargin==3
    p = @(z) 1.5-2*z+0.5*z.^2;      % The default is BDF2
else
    p=varargin{1};
end

d = size(h,1);              % number of components of problem
N = size(h,2)-1;            % N = number of time-steps
omega = exp(2*pi*1i/(N+1));
R = eps^(0.5/(N+1));

h = bsxfun(@times,h,R.^(0:N));  % scaling
h = fft(h,[],2);                % dft by columns
g = zeros(d,N+1);
parfor l=0:floor((N+1)/2)       %compute half the sequence
    g(:,l+1)=F(p(R.*omega.^(-l))/k)\h(:,l+1);   % \hat w
end

g(:,N+2-(1:floor(N/2)))=conj(g(:,2:floor(N/2)+1));
                                % mirror the hermitian seq
g=real(ifft(g,[],2));                            % w
g=bsxfun(@times,g,R.^(-(0:N)));                  % g

return
```

9.3 Approximating cylindrical waves

A cylindrical wave around a point $\mathbf{x}_0$ is given by the formula

$$u(\mathbf{x},t) = \int_0^{t-|\mathbf{x}-\mathbf{x}_0|} \frac{f(\tau)}{2\pi\sqrt{(t-\tau)^2 - |\mathbf{x}-\mathbf{x}_0|^2}} \mathrm{d}\tau,$$

assuming that f is a causal function ($f(t) = 0$ for all $t \le 0$). This formula is difficult to evaluate, but we can take advantage of the fact that the operator $f \mapsto u(\mathbf{x},\cdot)$ is a convolution operator and the corresponding transfer function is

$$\frac{\imath}{4} H_0^{(1)}(\imath s|\mathbf{x}-\mathbf{x}_0|).$$

This means that evaluation of this function at points $\mathbf{x}_i$ can be done using samples of f at discrete times and Convolution Quadrature. If we want to compute

$$\nabla u(\mathbf{x},t)\cdot\mathbf{n},$$

we can again take advantage of the convolutional structure of the process. Since $H_0^{(1)}(z)' = -H_1^{(1)}(z)$, then the corresponding transfer function is

$$\frac{s}{4} H_1^{(1)}(\imath s|\mathbf{x}-\mathbf{x}_0|)\frac{(\mathbf{x}-\mathbf{x}_0)\cdot\mathbf{n}}{|\mathbf{x}-\mathbf{x}_0|}.$$

The code evaluates $u(\mathbf{x}_i, t_j)$ and $\nabla u(\mathbf{x}_i, t_j)\cdot\mathbf{n}_j$. The Convolution Quadrature method is applied with oversampling in time, using a time-step $T/(5M)$ and then eliminating intermediate steps.

The function `cylindricalwave` can also take as input a cell array of function handles $F = \{(@t)\mathrm{signal}_1(t), \ldots, @(t)\mathrm{signal}_N(t)\}$ corresponding to different signals, as well as an $N\times 2$ matrix of source terms, one for each signal. The output is then the superposition of these signals:

$$\sum_{j=1}^{N} \int_0^{t-|\mathbf{x}-\mathbf{x}_j|} \frac{f_j(\tau)}{2\pi\sqrt{(t-\tau)^2 - |\mathbf{x}-\mathbf{x}_j|^2}} \mathrm{d}\tau.$$

A final option is for the signal to be passed to `cylindricalwave` after being observed. If there are N observation points, signal may be passed as an $N\times(M+1)$ matrix. In this case, there is no oversampling in the computation of the convolution. Detection of the type of input is fully automatic to `cylindrical wave`.

```
function [u,dnu]=cylindricalwave(f,c,x0,x,normal,T,M,option)

% [u,dnu]=cylindricalwave(f,c,[x0 y0],x,normal,T,M,option)
%
% Input:
%       f       : vectorized function of one variable or
%                 cell array of vectorized functions of one variable or
%                 Nsrc x M+1 matrix of a sampled signal
%       c       : wave speed
%       [x0 y0]: source point or matrix of source points (corresponding
%                 to one source point per handle in f.
%       x       : N x 2 matrix (observation points)
%       normal  : N x 2 matrix (vectors for directional observations)
%       T       : final time
%       M       : number of time steps
%       option  : 1 (compute everything) 2 (compute only uinc)
% Output:
%       u       : N x (M+1) matrix
%       dnu     : N x (M+1) matrix
```

```
%
% Last modified: May 29, 2014

N=size(x,1);
Nsc=size(x0,1);

if isa(f,'function_handle');
    upsample=1;
    k=5;        %upsampling parameter
    t=linspace(0,T,k*M+1);
    F=f(t);
    F=repmat(F,Nsc,1);
    kappaCQ=T/(k*M);
elseif iscell(f)
    upsample=1;
    k=5;        %upsampling parameter
    F=zeros(Nsc,k*M+1);
    t=linspace(0,T,k*M+1);
    for j=1:length(f)
        fj=f{j};
        F(j,:)=fj(t);
    end
    kappaCQ=T/(k*M);
elseif ismatrix(f)
    upsample=0;
    kappaCQ=T/M;
    F=f;
end

u=zeros(N,size(F,2));

for j=1:Nsc
    diffs=bsxfun(@minus,x,x0(j,:));
    dist =sqrt(diffs(:,1).^2+diffs(:,2).^2);
    U=@(s) 1i/4*besselh(0,1,1i*s*dist);
    u=u+CQforward(@(s) U(s/c),F(j,:),kappaCQ);
end

if option==2
    dnu=[];
    if upsample
        u=u(:,1:k:end);
    end
    return
end

dnu=zeros(N,size(F,2));

for j=1:Nsc
    diffs=bsxfun(@minus,x,x0(j,:));
    dist =sqrt(diffs(:,1).^2+diffs(:,2).^2);
    dipoles=sum(diffs.*normal,2)./dist;
    dnU=@(s) s/4*besselh(1,1,1i*s*dist).*dipoles;
    dnu=dnu+CQforward(@(s)dnU(s/c),F(j,:),kappaCQ);
end

if upsample
    u=u(:,1:k:end);
    dnu=dnu(:,1:k:end);
end

return
```

9.4 Runge-Kutta CQ Tools

We also have two functions, `RKCQforward` and `RKCQeqation` that are the Runge-Kutta counterparts to the linear multistep CQ tools. This section presents some preliminaries to aid in the use of RKCQ tools. For more details, we refer the reader to [3]. Consider a Runge-Kutta method given by the Butcher table

$$\begin{array}{c|c} \mathbf{c} & \mathbf{A} \\ \hline & \mathbf{b}^T \end{array}$$

where $A \in \mathbb{R}^{S\times S}$, $b \in \mathbb{R}^S$, and $c \in \mathbb{R}^S$. We define the stability function associated to the Runge-Kutta method by

$$R(z) = 1 + zb^T(I - zA)^{-1}\mathbb{1}, \qquad \mathbb{1} := (1, 1, \ldots, 1)^T.$$

We demand that the Runge-Kutta method satisfy $\mathbf{b}^T = \mathbf{e}_S^T A$ where $\mathbf{e}_S^T = (0 \ldots 1) \in \mathbb{R}^S$. This is equivalent to requiring the last row of the matrix A be the same as the weight vector $\mathbf{b}$. Such RK methods are known as stiffly accurate methods. We let S denote the number of stages of the method. Note that s is reserved for the continuous variable in the Laplace domain (i.e. $\mathcal{L}(f(t)) = \mathrm{F}(s)$), but in these notes and code it will be unambiguous.

Two families of methods which can be used for RKCQ are the Radau IIa and Lobatto IIIC families of methods. The Butcher table for 3^{rd} order Radau IIa is

$$\begin{array}{c|cc} 1/3 & 5/12 & -1/12 \\ 1 & 3/4 & 1/4 \\ \hline & 3/4 & 1/4 \end{array}$$

and the Butcher table for 4^{th} order Lobotto IIIC is

$$\begin{array}{c|ccc} 0 & 1/6 & -1/3 & 1/6 \\ 1/2 & 1/6 & 5/12 & -1/12 \\ 1 & 1/6 & 2/3 & 1/6 \\ \hline & 1/6 & 2/3 & 1/6 \end{array}$$

The matrix convolution weights are given by a generating function analogous to the scalar case:

$$F\left(\frac{\Delta(\zeta)}{\kappa}\right) = \sum_{j=0}^{\infty} W_j^F(\kappa)\zeta^j$$

with

$$\Delta(\zeta) = A^{-1}(I - \zeta\mathbb{1} \otimes \mathbf{e}_S^T).$$

We must be able to compute the matrix-valued operators

$$F\left(\frac{\Delta(R\zeta_{N+1}^{-l})}{\kappa}\right) \quad \text{and} \quad F\left(\frac{\Delta(0)}{\kappa}\right).$$

These operator-valued functions of matrices can be properly understood in terms of the Dunford-Taylor calculus. Such an exposition is not given here.

Suppose that $\Delta(R\zeta_N^{-l})$ has a full basis of eigenvectors, so that there exists a diagonal matrix $\Lambda = \mathrm{diag}(\lambda_1, \cdots, \lambda_S)$ and a matrix P so that $\Delta(R\zeta_N^{-l}) = P\Lambda P^{-1}$, then the operators above are computed according to

$$F\left(\frac{\Delta(R\zeta_N^{-l})}{\kappa}\right) = (P\otimes I)\mathrm{diag}(F(\lambda_1/\kappa),\ldots,F(\lambda_S/\kappa))(P^{-1}\otimes I).$$

We make the assumption that data is given as a $S\,d\times N$ matrix, which is then acted on by the $Sd\times Sd$ matrix $P^{-1}\otimes I_d$ matrix at each time step. The block-diagonal operator $\mathrm{diag}(F(\lambda_1/\kappa),\ldots,F(\lambda_S/\kappa))$ is applied component by component. Finally, the result is acted on by $P\otimes I_d$ in each time step.

We make use of the vectorized notation

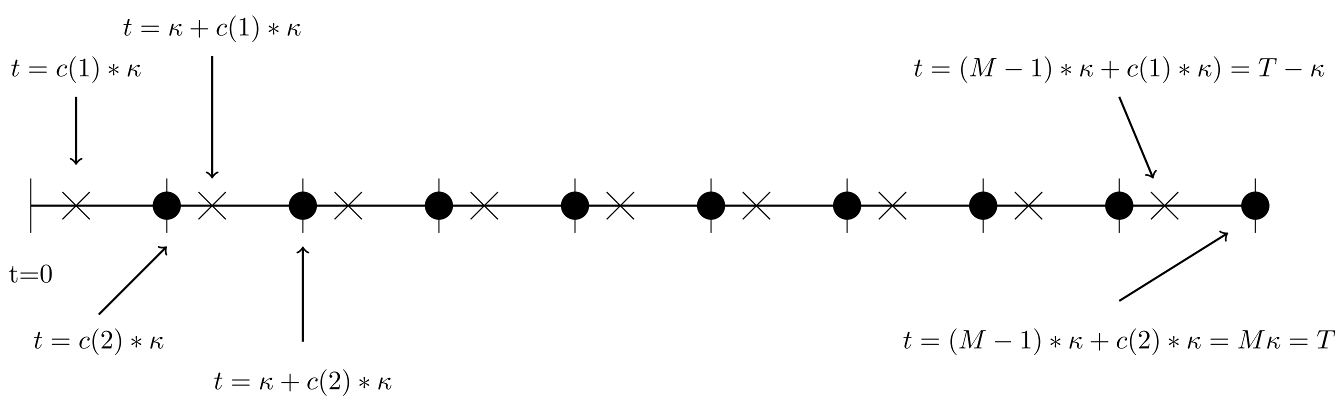
$$g(t_m+\mathbf{c}\kappa) := \begin{pmatrix} g(t_m+c_1\kappa) \\ g(t_m+c_2\kappa) \\ \vdots \\ g(t_m+c_S\kappa) \end{pmatrix}$$

to compactly represent quantities that are evaluated or stored at each stage of the RK method.

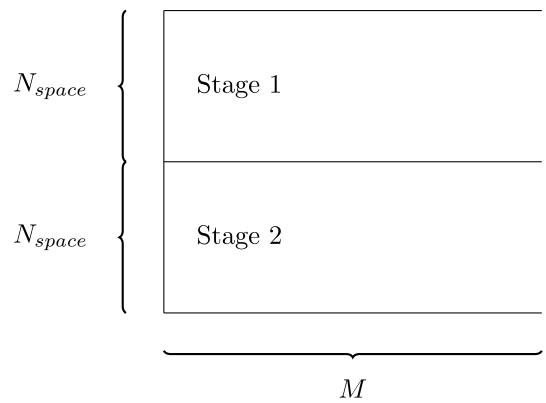
For a given final time T and number of time steps M, RKCQ approximates discrete convolutions on M intervals of length $\kappa = T/M$, as opposed to the scalar CQ which approximates the discrete convolution on $M+1$ equispaced points with $t_n - t_{n-1} = \kappa$ for $n = 1,\ldots,M+1$. For a concrete example, we diagram the approximation made by 2-stage Radau ii-a RKCQ. The Butcher array for Radau iia is

$$\begin{array}{c|cc} 1/3 & 5/12 & -1/12 \\ 1 & 3/4 & 1/4 \\ \hline & 3/4 & 1/4 \end{array}$$

We'll use Matlab notation for values in vectors. For time stepping, we need the vector c from the Butcher table, so in the case of Radau iia, we have $c(1) = 1/3$ and $c(2) = 1$. We will denote by an cross where the first stage is evaluated and a dot $\bullet$ where the second stage is evaluated. For Radau iia, data is evaluated and solutions are computed on the following points in the interval:



For computation with CQ routines, it is assumed that data is input as matrices of size $N_{space} * N_{stage} \times M$. In the case of a two stage method, it is stored as:



N_{space}
Stage 1
N_{space}
Stage 2
M

Bibliography

[1] V. Domínguez, S.L. Lu, F.J. Sayas. *A Nyström flavored Calderón Calculus of order three for two dimensional waves. Comput. Math. Appl.* **67** (2014) 217-236.

[2] V. Domínguez, T. Sánchez-Vizuet, F.J. Sayas. *A fully discrete Calderón Calculus for the two-dimensional wave equation.* To appear in *Comput. Math. Appl.*

[3] M. Hassell, F.–J. Sayas. *Convolution Quadrature for Wave Simulations. To appear SEMA SIMAI Springer Series.*

[4] C. Lubich. *Convolution Quadrature and discretized operational calculus I. Convolution quadrature and discretized operational calculus I. Numer. Math.* **52** (1988) 129-145.

Appendix A

Lists

A.1 Functions

Geometric module: curves

`bonegeometry`

`ellipse`

`kite`

`openarc`

`polygon`

`starshape`

`tvshape`

Geometric module: utilities

`affine`

`frequencyDomainPlot`

`joinGeometry`

`latticeGeometry`

`merge`

`mirror`

`mirrorLR`

`sample`

`selectComponents`

`separateTriangulation`

`threetimes`

`triangulateGeometry` (needs the PDETool)

`unpackGeometry`

General operators

- `CalderonCalculusMatrices`
- `CalderonCalculusTest`
  - `CalderonCalculusMatrices`

Resolvent-set (Helmholtz) Calderón Calculus

- `CalderonCalculusHelmholtz`
  - `CalderonCalculusMatrices`
- `HelmholtzPotentials`
- `CalderonCalculusHelmholtzDecoupled`
  - `CalderonCalculusHelmholtz`
  - `unpackGeometry`

Laplace Calderón Calculus

- `CalderonCalculusLaplace`
  - `CalderonCalculusMatrices`
- `LaplacePotentials`

Elasticity

- `CalderonCalculusElasticWave`
- `CalderonCalculusLame`
  - `CalderonCalculusMatrices`
- `ElasticWavePotentials`
- `LamePotentials`
- `sourceSolutionLame`

Time domain tools

- `CQforward`
- `CQequation`
- `cylindricalWave`
  - `CQforward`