TMATROM: object-oriented T-MATrix Reduced Order Model software for efficient simulation of multi-parameter acoustic scattering

M. Ganesh,
Department of Applied Mathematics and Statistics,
Colorado School of Mines, Golden, CO 80401, USA
mganesh@mines.edu

Stuart C. Hawkins,
Department of Mathematics,
Macquarie University, Sydney, NSW 2109, Australia
stuart.hawkins@mq.edu.au

14th September 2016

Abstract

Efficient simulation of scattering cross sections of wave scattering configurations is an important tool for several applications. The transition matrix (T-matrix) based approach, that is independent of certain input parameters, provides an efficient framework for scattering simulations with varying parameters in the configurations, including dynamic and uncertain configurations. In recent years, the authors developed and mathematically analyzed a numerically stable T-matrix approach, solving a several decade open problem of developing *a priori* T-matrix truncation parameters that are independent of the shape of the particle. We develop and describe details of an associated computational counterpart that will provide an open-source software package for simulation of multi-parameter acoustic scattering models.

In this article, we describe an object oriented software package TMATROM which provides a class of reduced order model (ROM) tools for efficient simulation of multi-parameter acoustic scattering by obstacles in two dimensions. Such multi-parameter problems occur, for example, in monostatic cross section computations (involving thousands of input incident directions), in stochastic computations, and in multiple scattering simulations. The TMATROM package provides offline tools for computing the T-matrix of any two dimensional obstacle using *any* forward

wave propagation solver. Example solvers based on the Nyström method are provided along with a framework to add any other solver. The TMATROM package provides efficient and easy to use online tools for quickly solving multi-parameter scattering problems. Detailed examples are provided within the software package for monostatic cross section computations, and multiple scattering simulations.

1 Installation and verification

Download

Download the archive file `tmatrom.tar.gz` (Linux/Unix/OS X) or `tmatrom.zip` from

`http://www.romapp.org/`

Unpack

Your system may automatically unpack the archive file for you. If your system does not automatically unpack the archive you can unpack it using (Linux/Unix/OS X)

```
tar -xvzf tmatrom.tar.gz
```

or by opening it with Winzip (Windows).

Save unpacked software in a (TMATROM) directory

We recommend you save the files unpacked from the archive in a directory. Below we refer to this directory as the TMATROM root directory.

Add TMATROM subdirectories to Matlab path

After starting Matlab from the TMATROM root directory, type the command

```
setpath_tmatrom
```

Run example codes

Several examples are included in the TMATROM software package. Example codes (in directories prefixed EXAMPLE and with file names starting with the string `example_`) provide a quick way to test and use the package before looking into details of the objected-oriented ROM in this article.

After installation, we recommend that users type the following two commands in Matlab to test the installation of the TMATROM package. The first command solves and visualizes a bistatic (single parameter) sound-soft scattering simulation. The second command solves and visualizes the corresponding monostatic (multi-parameter) simulation with with 1000 input incident waves.

```
example_rom_nystrom_soundsoft_bistatic
example_rom_nystrom_soundsoft_monostatic
```

Before proceeding further, users may also like to try examples for scatterers with other material properties by replacing `soundsoft` in the above commands with `soundhard` or `absorbing`. Source codes for these and additional examples, including multiple particle configurations, are in the TMATROM subdirectory `EXAMPLE_ROM_NYSTROM_SOLVER`.

We recommend users follow the quick start guide in Section 3 to learn more about how to use the package.

Contents

2 Introduction

We describe the use of our object-oriented Matlab software package TMATROM. This package provides a class of reduced order model (ROM) software tools for simulation of multi-parameter acoustic scattering by obstacles in two dimensions. For details of various forward and inverse scattering models and associated applications, we refer to [2] and extensive list of references therein. The forward scattering model is governed by the Helmholtz equation and a set of parameters that provides information of the full model, such as the input incident wave that induces the scattered field, and how the output quantities of interest (QoIs) are measured (with a receiver direction parameter). Typical QoIs from the model are the scattered and far fields, and the associated acoustic cross section (ACS) of the scattering configuration [2].

Our ROM [5, 8] provides a very efficient framework when the scattering model needs to be simulated for multiple sets of parameters. Such multiple parameter sets occur, for example, when the QoI is the *monostatic* ACS, corresponding to the situation where the receiver direction is opposite to the incident direction (backscattering) and the receiver direction parameter varies from zero to 360 degrees.

Our computationally stable T-matrix approach [5, 8] is an optimization-free reduced basis framework to setup up a matrix that characterizes the scattering properties of an obstacle. The matrix is independent of the incident and receiver directions. If the location and orientation of the obstacle are changed, the associated new T-matrix can be quickly computed using techniques based on the translation-addition theorem and the old T-matrix of the obstacle for the original location and orientation. The T-matrix can be used any number of times to simulate scattered and far-fields after setting up appropriate vector

representations of several input incident fields. The numerical stability of TMATROM is comprehensively demonstrated with numerical examples in [7].

Multiple sets of parameters also arise in multiple particle scattering configurations, where the location and orientation of each particle in the configuration should be treated as parameters. The TMATROM framework is efficient for the multiple particle configuration based scattering model, even without prior knowledge of the location and orientation of each particle in the configuration, and hence it is also appropriate for moving particle configurations.

The TMATROM package provides an *offline* software framework to build a T-matrix for any two dimensional obstacle with a user's choice of obstacle geometry and appropriate material property (such as sound-soft, sound-hard, absorbing, penetrable). Subsequently the user can efficiently develop an *online* approach by assembling various types of multiple particle deterministic or stochastic configurations [6] comprising the obstacles.

A major advantage of our TMATROM framework is that it utilizes standard numerical methods for solving the scattering problem (for a fixed parameter set) *but* it is independent of any specific numerical method. Consequently, the TMATROM package can be used in conjunction with any existing forward wave propagation solver that a user is already familiar with. Easy incorporation of a solver into our object-oriented TMATROM core framework requires only that the solver can compute the far-fields associated with incident circular waves [8, Equation (2.15)].

To illustrate the easy customization of our object-oriented framework, we provide examples with two distinct families of forward solvers with the TMATROM package. The first is developed by the authors and uses the Nyström method [2]. The second is based on an open-source forward solver using non-polynomial finite elements and the method of fundamental solutions [1]. The forward solver does not need to use Matlab's object-oriented framework, but it does need to use the Matlab environment (and may use Mex files to incorporate procedures written using other computer languages).

The package has two components. The first component is the TMATROM kernel which contains object-oriented classes and functions associated with the ROM. The kernel component is described in Part I of this manual. The second component includes several example solvers for the scattering problem that will be sufficient for most users. The example solvers are described in Part II. We will assume that the user is familiar with the basic features of Matlab. The class structure of TMATROM is visualized in Figures 1–3.

As described above, the TMATROM model is independent of the solver used and users can provide their own solvers by extending templates provided. Details required for coding user defined solvers are described in Part III. In Part III we assume that the user is familiar with object-oriented programming in Matlab. The authors will be happy to collaborate with any user or group to customize their forward solver within our TMATROM

framework.

While we restrict the TMATROM package to the 2D models, our reduced basis T-matrix framework with *a priori* analytical truncation parameter estimates (without the expensive optimization techniques, such as the greedy approach, that are required in most reduced basis methods, see [9] and references therein) and exponential convergence analysis are applicable for both two and three dimensional scattering models [8]. In future we plan to develop a 3D version of the TMATROM package.

We request that any publications that make use of our TMATROM package cite our key papers on the ROM [7, 5, 6, 8] and this article. This article includes the minimal mathematical details required to understand the underlying algorithm and the package can be used as a *black-box* without knowledge of these details. For complete mathematical details of our stable ROM algorithm and convergence analysis with application to stochastic multiple particle configurations, we ask the user of this manual and TMATROM package to refer to [8].

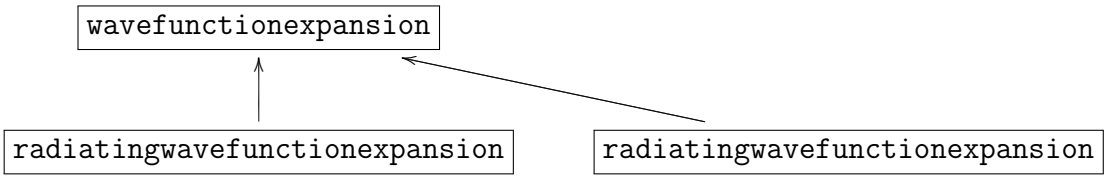

Figure 1: Figure showing the class dependencies for the wavefunction expansion classes.

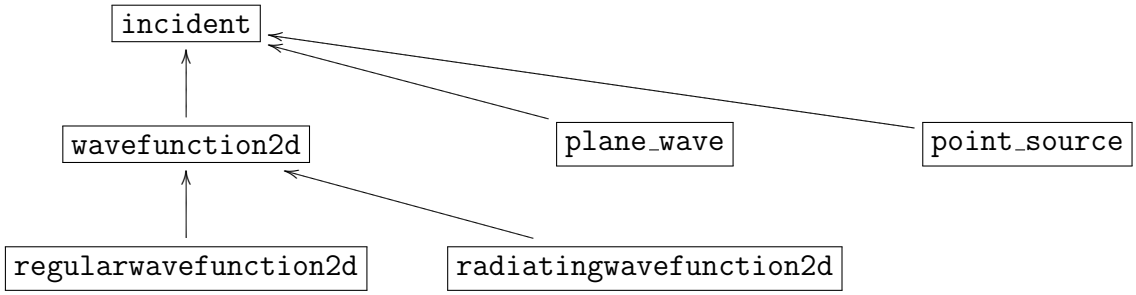

Figure 2: Figure showing the class dependencies for the incident wave classes.

A note on position vectors in TMATROM In our TMATROM code, and correspondingly in this manual, we represent real valued vectors in $\mathbb{R}^2$ (such as position vectors) by complex numbers with the real part and imaginary parts of the complex number corresponding to the x- and y-coordinates of the vector respectively.

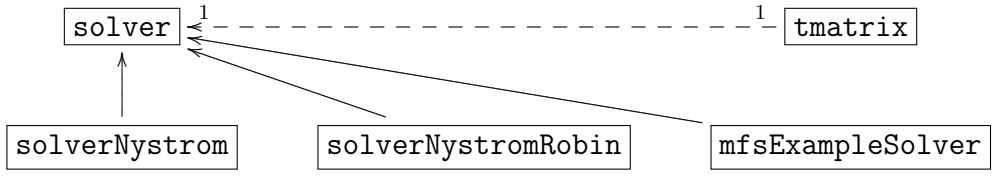

Figure 3: Figure showing the class dependencies for the solver and T-matrix classes.

3 Quick start guide

In this section we first describe how to setup, simulate and visualize the scattered and far-field induced by a plane wave impinging on a sound-soft kite shaped scatterer using the TMATROM package. Then, using the T-matrix of the kite scatterer, we demonstrate how to efficiently efficiently simulate the monostatic ACS of the scatterer using hundreds of incident plane wave directions.

First we setup the scatterer illustrated in Figure 4:

```
g = obstacleKite();
```

For a fixed wavenumber, say `kwave = 1`, it requires only a few lines (which are explained in detail later) to setup the incident wave direction independent T-matrix, which exponentially accurately characterizes the scattering properties of the scatterer:

```
kwave = 1;
solver = solverNystrom(kwave,[],g);
solver.setup(15);
nmax = suggestedorder(kwave,solver.getRadius());
tmat = tmatrix(nmax,kwave,solver,0);
```

The dimension of the T-matrix is suggested by the wavelength of the problem and associated convergence analysis [8].

A few more lines are required to simulate and visualize (see Figure 5) the intensity of the far field induced by an incident plane wave with direction $\widehat{\boldsymbol{d}} = (\cos\theta, \sin\theta)^T$ with, say, $\theta = \pi/4$ impinging on the scatterer.

```
p = plane_wave(pi/4,kwave);
b = tmat * regularwavefunctionexpansion(nmax,0,p);
b.visualizeFarField()
```

The monostatic ACS of the scatterer obtained at $m = 1000$ (co-located transmitter and receiver) angles in $[0, 2\pi]$ is easily and quickly simulated and visualized.

```
m = 1000;
farfield = zeros(m,1);
```

```
parfor j = 1:m
    p = plane_wave(pi+2*pi*j/m,kwave);
    c = tmat * regularwavefunctionexpansion(nmax,0,p);
    farfield(j) = c.evaluateFarField(exp(1i*2*pi*j/m));
end
plot(2*pi*(1:m)/m,10*log10(2*pi*abs(farfield).^2),'r')
```

The above commands will produce the plot in Figure 6.

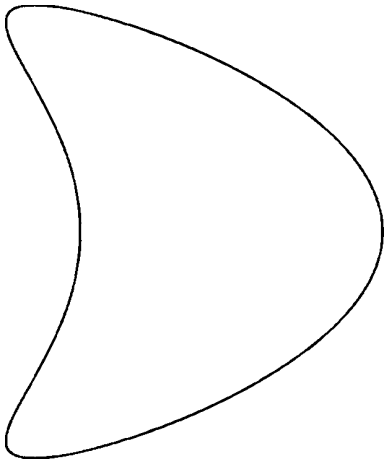

Figure 4: One of the TMATROM built-in example scatterers: a kite shape.

4 Mathematical Model

In this section we briefly describe the exterior scattering problem, in which an incident field interacts with a configuration to produce a scattered field. The scattering configuration may comprise one or more obstacles with various material properties (for example sound hard, sound soft, penetrable). In all cases, we denote the configuration by D.

The incident wave may or not penetrate inside the obstacles in D. In the case of a dielectric penetrable obstacle, the interior wavenumber is different from the exterior wavenumber k; the medium inside the obstacle may also be heterogeneous.

The scattered field exterior to D satisfies the constant coefficient Helmholtz equation [2]

$$\triangle u(\boldsymbol{x}) + k^2 u(\boldsymbol{x}) = 0, \qquad \boldsymbol{x} \in \mathbb{R}^2 \setminus \overline{D}, \tag{1}$$

and the Sommerfeld radiation condition [2, Equation (3.85)]

$$\lim_{|\boldsymbol{x}|\to\infty} \sqrt{|\boldsymbol{x}|} \left(\frac{\partial u}{\partial \boldsymbol{x}}(\boldsymbol{x}) - iku(\boldsymbol{x}) \right) = 0, \tag{2}$$

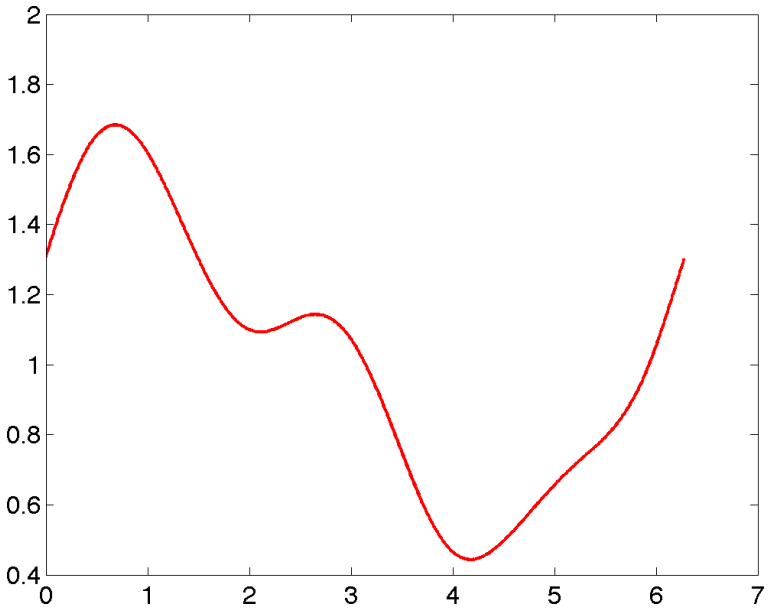

Figure 5: Absolute value of the far field induced by a plane wave impinging on the kite.

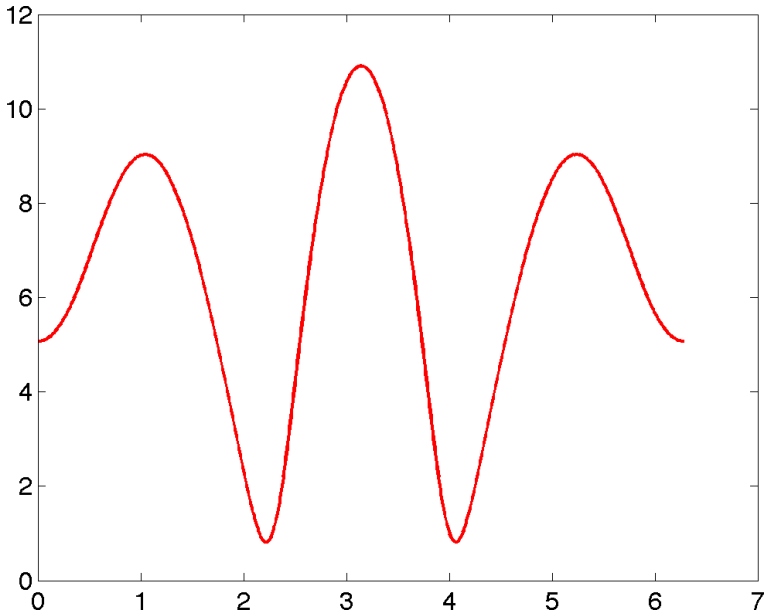

Figure 6: Monostatic ACS (dB) of the kite shaped sound-soft scatterer simulated using 1000 plane waves with 1000 incident direction angles in $[0, 2\pi]$.

uniformly as $|\boldsymbol{x}| \to \infty$. Here k is the wavenumber of the incident field, which satisfies $k = 2\pi/\lambda$, where λ is the wavelength.

For each impenetrable scatterer in D the above system needs to be augmented with a Dirichlet, Neumann, or Robin boundary condition, depending respectively on whether the obstacle is sound-soft or sound-hard or absorbing [2, 8]. For each penetrable scatterer in D the above system needs to augmented with the interior (variable or constant coefficient) Helmholtz equation in the interior of the obstacle and an associated interface boundary condition on the obstacle surface.

The incident wave may be a plane wave

$$u^{\text{inc}}(\boldsymbol{x}) = e^{ik\boldsymbol{x}\cdot\widehat{\boldsymbol{d}}}, \tag{3}$$

with direction specified by the unit vector $\widehat{\boldsymbol{d}}$, or a point source field

$$u^{\text{inc}}(\boldsymbol{x}) = H_0^{(1)}(k|\boldsymbol{x} - \boldsymbol{y}_0|), \tag{4}$$

where $\boldsymbol{y}_0$ is the point source location.

In many problems the quantities of interest (QoI) are the far field and associated acoustic cross section (ACS), in decibels (dB), as functions of the unit vectors $\widehat{\boldsymbol{x}} = \boldsymbol{x}/|\boldsymbol{x}|$ (representing the observation or receiver directions), defined respectively as [2, 8]:

$$u^{\infty}(\widehat{\boldsymbol{x}}) = \lim_{|\boldsymbol{x}|\to\infty} \sqrt{|\boldsymbol{x}|}e^{-ik|\boldsymbol{x}|}u(\boldsymbol{x}), \qquad \sigma^{\text{ACS}}(\widehat{\boldsymbol{x}}) = 10\log_{10}(2\pi\,|u^{\infty}(\widehat{\boldsymbol{x}})|^2). \tag{5}$$

Part I

The core TMATROM classes

Our T-matrix ROM [8] is based on expanding the regular incident field and the radiating scattered field using two distinct classes of basis functions. The TMATROM package kernel provides associated object-oriented classes. The kernel is independent of the solver used to simulate the mathematical model described in the previous section.

5 The `regularwavefunctionexpansion` class

The regular circular wavefunction with wavenumber k

$$\widetilde{e}_\ell(\boldsymbol{z}) = J_{|\ell|}(k|\boldsymbol{z}|)e^{i\ell\theta(\boldsymbol{z})}, \tag{6}$$

satisfies the Helmholtz equation (1) for all $\boldsymbol{z} \in \mathbb{R}^2$. Here J_l denotes the Bessel function of order l and $(r, \theta) = (|\boldsymbol{z}|, \theta(\boldsymbol{z}))$ are polar coordinates for the point $\boldsymbol{z}$. For fixed $n \in \mathbb{N}$ and coefficients $a_{-n}, \ldots, a_n$, the regular wavefunction series expansion with origin $\boldsymbol{x}_0$,

$$u(\boldsymbol{x}) = \sum_{j=-n}^{n} a_j \widetilde{e}_j(\boldsymbol{x} - \boldsymbol{x}_0) \tag{7}$$

is a regular solution of the Helmholtz equation (1).

Instantiation using coefficients

The `regularwavefunctionexpansion` class represents regular wavefunction expansions of the form (7). Given a vector of coefficients `cof` of length $2\mathtt{n} + 1$, an instance of the `regularwavefunctionexpansion` class is created using

```
u = regularwavefunctionexpansion(n,x0,k,cof);
```

where `k` is the wavenumber and `x0` is the expansion origin. Here `cof(j)` contains the coefficient $a_{\mathtt{j}-\mathtt{n}-1}$ for $\mathtt{j} = 1, \ldots, 2\mathtt{n} + 1$.

Instantiation from a plane wave

If a function u is regular in a neighbourhood of $\boldsymbol{x}_0 \in \mathbb{R}^2$ then u can be approximated by a regular wavefunction expansion of the form (7). If u is a plane wave (3) then there is an analytical expression for the coefficients [8]. An instance of the `regularwavefunctionexpansion` class approximating a `plane_wave` object `p` is created using

```
u = regularwavefunctionexpansion(n,x0,p);
```

where `n` is the order of the expansion and `x0` is the expansion origin.

Instantiation from a point source

Similarly, if u is the field induced by a point source (4) there is an analytical expression for the coefficients [8]. An instance of the `regularwavefunctionexpansion` class approximating a `point_source` object `q` is created using

```
u = regularwavefunctionexpansion(n,x0,q);
```

where `n` is the order of the expansion and `x0` is the expansion origin. This expansion is valid only in neighbourhoods about `x0` that do not contain the point source origin $\boldsymbol{y}_0$.

Evaluation

A `regularwavefunctionexpansion` object `u` is evaluated at points `z` using

```
val = u.evaluate(z);
```

Here `z` may be a scalar, vector or matrix. The `evaluate` method may be used to visualize the wavefunction expansion. For example, to visualize `u` in $[-10, 10] \times [-10, 10]$ with 500×500 mesh points

```
t = linspace(-10,10,500);
[x,y] = meshgrid(t);
z = x + 1i*y;
surf(x,y,real(u.evaluate(z)))
```

Visualization

The `regularwavefunctionexpansion` class also provides a convenient method for quickly visualizing the field. For example

```
u.visualize([-10 10 -10 10])
```

produces a similar figure to the example above.

Addition and subtraction

Two `regularwavefunctionexpansion`s `u` and `v` that are compatible are added using

```
w = u + v;
```

and subtracted using

```
w = u - v;
```

The `regularwavefunctionexpansion`s are compatible if they have the same origins, wavenumbers and orders.

6 The `radiatingwavefunctionexpansion` class

The radiating wavefunction

$$e_l(\boldsymbol{z}) = H_{|l|}^{(1)}(k|\boldsymbol{z}|)e^{il\theta(\boldsymbol{z})}, \tag{8}$$

satisfies the Helmholtz equation (1) for all $\boldsymbol{z} \in \mathbb{R}^2 \setminus \{\boldsymbol{0}\}$ and the radiation condition (2). Here $H_l^{(1)}$ denotes the Hankel function of order l and $(r, \theta) = (|\boldsymbol{z}|, \theta(\boldsymbol{z}))$ are polar coordinates for the point $\boldsymbol{z}$.

For fixed $n \in \mathbb{N}$ and coefficients $b_{-n}, \ldots, b_n$, the radiating wavefunction series expansion with origin $\boldsymbol{x}_0$,

$$u(\boldsymbol{x}) = \sum_{m=-n}^{n} b_m e_m(\boldsymbol{x} - \boldsymbol{x}_0) \tag{9}$$

is a radiating solution of the scattering problem (1)–(2).

Instantiation using coefficients

The `radiatingwavefunctionexpansion` class represents radiating wavefunction expansions of the form (9). Given a vector of coefficients `cof` of length $2\mathtt{n} + 1$, an instance of the `radiatingwavefunctionexpansion` class is created using

```
u = radiatingwavefunctionexpansion(n,x0,k,cof);
```

where `k` is the wavenumber and `x0` is the expansion origin. Here `cof(j)` contains the coefficient $b_{\mathtt{j}-\mathtt{n}-1}$ for $\mathtt{j} = 1, \ldots, 2\mathtt{n}+1$.

Typically `radiatingwavefunctionexpansion` objects are created by applying a transformation to a `regularwavefunctionexpansion` object. This is described in detail in the next section.

Evaluation

A `radiatingwavefunctionexpansion` object `u` is evaluated at points `z` using

```
val = u.evaluate(z);
```

Here `z` may be a scalar, vector or matrix. The `evaluate` method may be used to visualize the wavefunction expansion. For example to visualize `u` in $[-10, 10] \times [-10, 10]$ with 500×500 mesh points,

```
t = linspace(-10,10,500);
[x,y] = meshgrid(t);
z = x + 1i*y;
surf(x,y,real(u.evaluate(z)))
```

Visualization

The `radiatingwavefunctionexpansion` class also provides a convenient method for quickly visualizing the field. For example

```
u.visualize([-10 10 -10 10])
```

produces a similar figure to the example above.

A radiating wave function expansion may blow up near its origin. The expansion may be visualized only outside a neighborhood of its origin by applying a mask. For example, to visualize `u` exterior to a disk of radius, say `rad`,

```
mask = abs(z) > 1.1 * rad;
surf(x,y,real(u.evaluate(z,mask)))
```

Evaluating the far field

The far field of a `radiatingwavefunctionexpansion` object `u` is evaluated at receiver direction points `z` (on the unit circle) using

```
val = u.evaluateFarField(z);
```

Here `z` may be a scalar, vector or matrix. The `evaluate` method may be used to visualize the far field of a wavefunction expansion. For example

```
t = linspace(0,2*pi);
z = exp(1i*t);
plot(t,abs(u.evaluateFarField(z)))
```

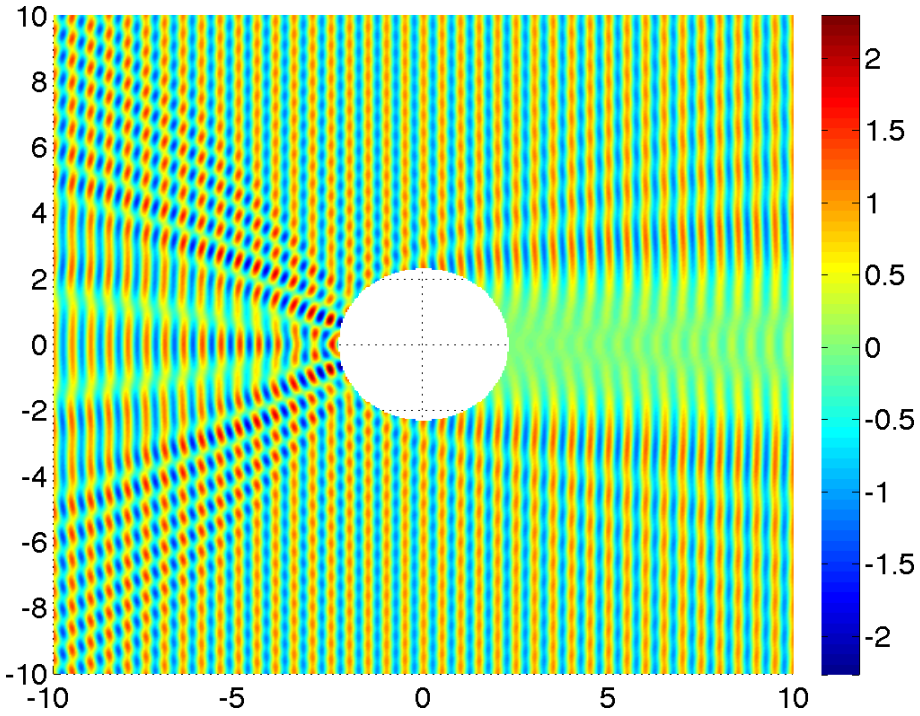

Figure 7: Visualization of the total field induced by a plane wave impinging on the kite.

Visualizing the far field

The `radiatingwavefunctionexpansion` class also provides a convenient method for quickly visualising the far field. For example

```
u.visualizeFarField()
```

produces a similar figure to the example above.

Addition and subtraction

Two `radiatingwavefunctionexpansions` `u` and `v` that are compatible are added using

```
w = u + v;
```

and subtracted using

```
w = u - v;
```

The `radiatingwavefunctionexpansions` are compatible if they have the same origins, wavenumbers and orders.

Example

The visualization in Figure 7 is readily plotted by combining details in this and the previous section with the quick start guide in Section 3. For details, see the part of the code labelelled `visualize the total field` in `example_rom_nystrom_soundhard_bistatic.m`.

7 Other `wavefunctionexpansion` operations

The additional features of the TMATROM kernel classes described in this section are useful for efficient online simulation of multiple particle configurations with dynamic locations and orientations of obstacles.

Copying

A `regularwavefunctionexpansion` object `u` is copied using

```
v = regularwavefunctionexpansion(u);
```

Similarly, a `radiatingwavefunctionexpansion` object `u` is copied using

```
v = radiatingwavefunctionexpansion(u);
```

Changing the expansion origin (regular to regular)

A regular wave function expansion (7) may be expanded about a new origin using the translation-addition theorem [3]. The origin of a `regularwavefunctionexpansion` object `u` is changed to a new origin `x1` using

```
u.changeorigin(x1)
```

An alternative is to create a new `regularwavefunctionexpansion` object `v` with origin `x1` using

```
v = regularwavefunctionexpansion(u,x1);
```

A new `regularwavefunctionexpansion` object `v` with order `n1` and origin `x1` is created using

```
v = regularwavefunctionexpansion(u,x1,n1);
```

In all three cases the expansion coefficients with respect to the new origin are computed using the translation addition theorem [3].

Changing the expansion origin (radiating to radiating)

A radiating wave function expansion (9) may be expanded about a new origin using the translation-addition theorem. The origin of a `radiatingwavefunctionexpansion` object `u` is changed to a new origin `x1` using

```
u.changeorigin(x1)
```

An alternative is to create a new `radiatingwavefunctionexpansion` object `v` with origin `x1` using

```
v = radiatingwavefunctionexpansion(u,x1);
```

A new `radiatingwavefunctionexpansion` object `v` with order `n1` and origin `x1` is created using

```
v = radiatingwavefunctionexpansion(u,x1,n1);
```

In all three cases the expansion coefficients with respect to the new origin are computed using the translation addition theorem [3].

Changing the expansion origin (radiating to regular)

If a radiating wave function expansion (9) is regular in a neighborhood of the new origin `x1` then it may also be approximated using a regular wavefunction expansion about the new origin

```
v = regularwavefunctionexpansion(u,x1);
```

A new `regularwavefunctionexpansion` object `v` with order `n1` and origin `x1` is created using

```
v = regularwavefunctionexpansion(u,x1,n1);
```

In both cases the coefficients with respect to the new origin are computed using the translation addition theorem [3].

Rotating the coordinate system

A regular wave function expansion (7) or radiating wave function expansion (9) may be expanded in a new coordinate system obtained by rotating the old axes (with origin at the expansion origin). A `regularwavefunctionexpansion` or `radiatingwavefunctionexpansion` object u is transformed to a coordinate system rotated by angle `theta` in the positive direction using

```
u.rotatecoordinates(theta)
```

Example

In this example we use a `radiatingwavefunctionexpansion` object `v` with origin 0 representing the field scattered by a kite shaped object. The field is visualized in Figure 8 (left).

Using the command

```
v.rotatecoordinates(pi/6)
```

we rotate the coordinate system by angle $\pi/6$ in the positive direction. Subsequent calls to the `evaluate` method will be with respect to the new coordinate system. The field is visualised with respect to the new coordinate system in Figure 8 (center).

Now

```
u = regularwavefunctionexpansion(v,4-2i);
```

constructs a `regularwavefunctionexpansion` object `u` that represents the field in a neighbourhood of $(4,-2)^T$ (with respect to the rotated coordinate system). This regular wavefunction expansion is visualised in Figure 8 (right) in a ball around $(4,-2)^T$.

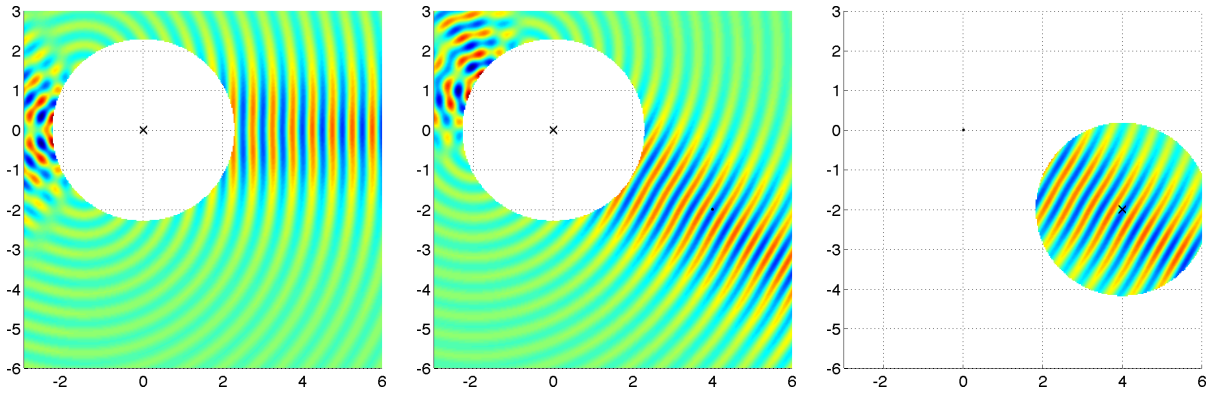

Figure 8: Visualizations of a field scattered by a kite shaped object in the original coordinate system (left); in a rotated coordinate system (center); and with a new expansion center $(4, -2)^T$. The expansion centers are marked with $\times$.

8 The `tmatrix` class

Linearity of the Helmholtz equation (1) implies that when an incident wave (7) is scattered by a configuration, the radiating wave expansion coefficients (9) satisfy

$$\underline{b} = T\underline{a}, \tag{10}$$

where T is a matrix, called the T-matrix. Here we assume that the series are not truncated (so $n = \infty$) and $\underline{a} = (a_m)_{m\in\mathbb{N}}$, $\underline{b} = (b_m)_{m\in\mathbb{N}}$. This relation holds approximately when the series expansions are truncated to a finite order n, and the corresponding T-matrix is a $(2n+1) \times (2n+1)$ matrix, denoted below as T_n.

Instantiation

The T-matrix depends on the scatterer, and on the center chosen for the wavefunction expansions of the incident and scattered waves, but is independent of their expansion coefficients (and hence the incident field). The `tmatrix` class represents the T-matrix of a scatterer. An instance of the `tmatrix` class for expansions of order `n` is created using

```
tmat = tmatrix(n,k,s,x0);
```

where `x0` is the expansion origin and `k` is the wavenumber. The solver `s` is described below.

Solver

The parameter `s` represents a solver (based, for example, on various types of boundary and finite element methods) for the two dimensional wave scattering problem described in Section 4 for a fixed set of parameters determining the problem. The shape (and properties of the scatterer) enter the T-matrix through the solver. Example solvers are provided with the TMATROM package but users can also setup a solver class based on their own solver for the two dimensional wave scattering problem. The solver is explained in more detail in the next section.

Error measure to quantify the accuracy of the truncated T-matrix

The infinite T-matrix T satisfies the symmetry relation [5, Theorem 1]

$$T + T^* + 2TT^* = 0$$

in the case of sound-soft, sound-hard or transmission boundary conditions applied on the scatterer surface. The symmetry condition also holds in the Robin boundary condition case under certain conditions on the impedance. In the case of a truncated T-matrix T_n, the quantity

$$\max_{\ell,m=-n,\ldots,n} |(T_n + T_n^* + 2T_nT_n^*)_{\ell,m}|$$

gives a useful measure of the error. This error measure is computed for a `tmatrix` object `T` using

```
T.error()
```

The error measure above depends on the solver `s` and also on the order `n` chosen for the wavefunction expansions. A suitable expression for the order in the case of spherical scatterers is given in [10] and our experience [4, 8] suggests that this choice is usually suitable for two dimensional circular scatterers and scatterers of other shapes. This order can be calculated using

```
n = suggestedorder(k,r);
```

where `r` is the radius of the scatterer. We refer to [8] for convergence results for the T-matrix.

Scattered field computation

The radiating scattered field induced by scattering of an incident field is given by

```
b = T * a;
```

Here `a` is a `regularwavefunctionexpansion` representing the incident field, `T` is a `tmatrix` and `b` is a `radiatingwavefunctionexpansion` representing the scattered field. The objects `a` and `T` are required to have the same wavenumber, origin and order.

Single particle configuration examples with multiple parameters

Our source codes

```
example_rom_nystrom_soundsoft_bistatic.m
example_rom_nystrom_soundsoft_monostatic.m
example_rom_nystrom_soundhard_bistatic.m
example_rom_nystrom_soundhard_monostatic.m
example_rom_nystrom_absorbing_bistatic.m
example_rom_nystrom_absorbing_monostatic.m
```

in the TMATROM subdirectory `EXAMPLE_ROM_NYSTROM_SOLVER` illustrate the usage of the `tmatrix` class for multi-parameter scattering by configurations containing single obstacles with various material properties.

Multiple particle configuration simulations using the T-matrix

Having assembled the T-matrix ROM for each particle in the configuration we can efficiently simulate scattering by many configurations (with the same particles but varying locations and orientations of the particles) using the additional object oriented features described in Section 7.

To briefly describe the method, we consider scattering of an incident field `p` by a configuration containing three individual particles with centers `x{1}`, `x{2}` and `x{3}` respectively. We assume that the T-matrices `tmat{1}`, `tmat{2}` and `tmat{3}` of the three particles have already been computed with origin 0. Using a logical origin at 0 corresponds to using local coordinates about the center of each particle.

We let `a{1}`, `a{2}` and `a{3}` be the radiating wavefunction expansions of the field scattered by the three particles. Then the scattered field is `a{1} + a{2} + a{3}` and the total field is `p + a{1} + a{2} + a{3}`.

Let us consider what happens on the first scatterer, which has center `x{1}`. The incident field on the first particle consists of the incident wave `p` and the field `a{2} + a{3}` scattered by the other particles. We can combine these into a single regular wavefunction expansion with center `x{1}`

```
regularwavefunctionexpansion(n,x{1},p) ...
      + regularwavefunctionexpansion(a{2},x{1}) ...
      + regularwavefunctionexpansion(a{3},x{1})
```

where `n` is the order of the wavefunction expansions.

The scattered field can then be computed from the incident field using the T-matrix. To facilitate this we first set the logical origin of the T-matrix to be the center of the first particle

```
tmat{1}.setOrigin(x{1});
```

Then the scattered field is computed from the incident field in the usual way

```
a{1} = tmat{1} * ( regularwavefunctionexpansion(n,x{1},p) ...
      + regularwavefunctionexpansion(a{2},x{1}) ...
      + regularwavefunctionexpansion(a{3},x{1}) );
```

In practice the radiating wavefunction expansions `a{1}`, `a{2}` and `a{3}` are unknown. However, the expression above, and similar expressions derived for the other scatterers, give a linear system for the radiating wavefunction expansion coefficients that can be solved iteratively. In the example

```
example_rom_nystrom_multiple_config.m
```

the linear system is solved using GMRES.

For the full algorithm and the corresponding mathematical description, including full details of the linear system and its solution, we refer to [6, Section 2.2]. Details in the source code

`example_rom_nystrom_multiple_config.m`

in the TMATROM subdirectory `EXAMPLE_ROM_NYSTROM_SOLVER` illustrate the ROM simulation and reproduce the visualisations of the scattered and total field in Figure 9 for the multiple particle configuration model. The preliminary off line step is to build the T-matrix ROM for each distinctly shaped template particle in the configuration. This approach is easily adapted for simulating dynamic multiple particle configurations.

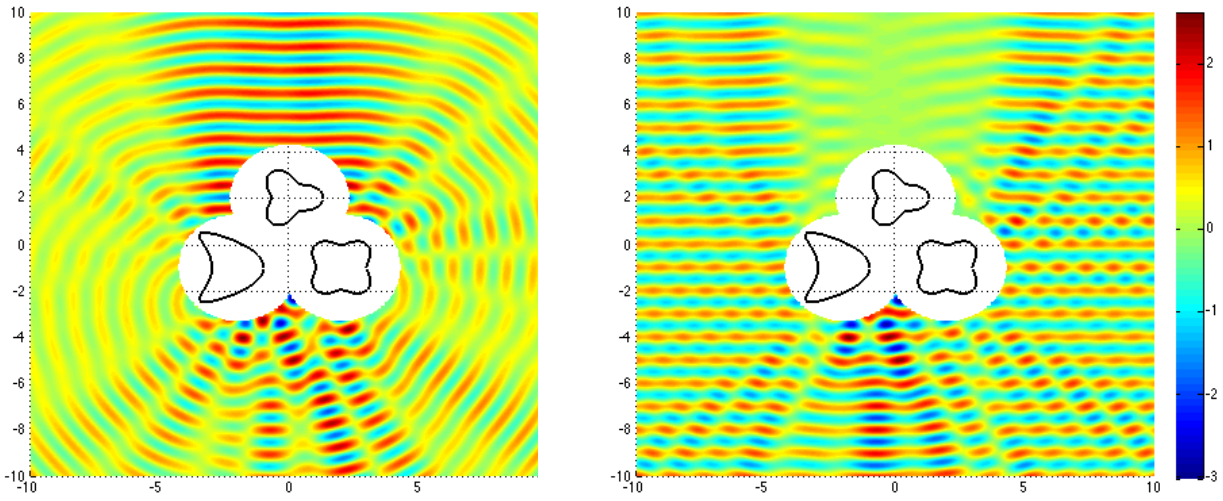

Figure 9: Multiple-parameter acoustic computer model using ROM scattering characterization (independent of the location, orientation, input incident wave, and output observation direction) of each distinctly shaped particle. Real part of scattered field (left) and total field (right) of a multiple particle configuration.

Part II

Using the provided solvers

9 The `solverNystrom` sound soft scatterer solver

The `solverNystrom` class solves the scattering problem (1)–(2) for a single particle D subject to the sound soft boundary condition

$$u(\boldsymbol{x}) + u^{\mathrm{i}}(\boldsymbol{x}) = 0, \qquad \boldsymbol{x} \in \partial D,$$

where ∂D denotes the boundary of the particle. The boundary integral equation reformulation of the scattering problem and its solution using the high order Nyström method is described in [2, Section 3.5].

Example boundaries

The boundary ∂D is represented by an object of class `obstacle`. Some example obstacle classes are provided in the TMATRIX package, including the circle with radius `r`,

```
b = obstacleCircle(r);
```

the kite shaped obstacle [2, Page 79],

```
b = obstacleKite();
```

the pinched ball [2, Page 94]

```
b = obstaclePinchedBall();
```

the cylinder with height `h` and width `w` capped with semi-circles

```
b = obstacleCappedCylinder(h,w);
```

and the trefoil [1]

```
b = obstacleTrefoil();
```

Polar coordinate boundaries

Obstacles described by polar coordinates can easily be represented using

```
b = obstaclePolar(r,dr,ddr);
```

where the anonymous functions `r`, `dr` and `ddr` represent the radius function and its first and second derivatives with respect to `t`. For example, the pinched ball is given by

```
r = @(t) sqrt(1.44 - 0.5*cos(-4*t));
dr = @(t) -sin(-4*t)./sqrt(1.44 - 0.5*cos(-4*t));
```

```
ddr = @(t) -sin(-4*t).^2./(1.44 - 0.5*cos(-4*t)).^1.5 ...
          + 4*cos(-4*t)./sqrt(1.44 - 0.5*cos(-4*t));
b = obstaclePolar(r,dr,ddr);
```

Instantiation

A `solverNystrom` object is created using

```
S = solverNystrom(k,[],b);
```

where `k` is the wavenumber and `b` is an obstacle. The incident field(s) are then set using

```
S.setIncidentField(inc)
```

where `inc` is a cell array of `incident` objects. For example

```
inc{1} = plane_wave(0,k);
inc{2} = point_source(3+2i,k);
S.setIncidentField(inc);
```

sets two incident fields, the first induced by a plane wave and the second induced by a point source.

Solving

The integral operators are discretized using the Nyström scheme with `2q` quadrature points using

```
S.setup(q)
```

and the corresponding linear systems are solved using

```
S.solve();
```

Far field computation

The far field is then obtained at points with angle specified in the vector `theta` using

```
F = S.getFarField(theta,[1 2]);
```

Here `F(:,1)` is the far field corresponding to `inc{1}` and `F(:,2)` is the far field corresponding to `inc{2}`.

Example

A full example to simulate scattering of a plane wave and spherical wave induced by a point source impinging on a sound soft pinched ball shaped scatterer is below. The resulting simulated intensity of far fields are visualized in Figure 10.

```
b = obstaclePinchedBall();
k = 1;
q = 15;
S = solverNystrom(k,[],b);
inc{1} = plane_wave(0,k);
```

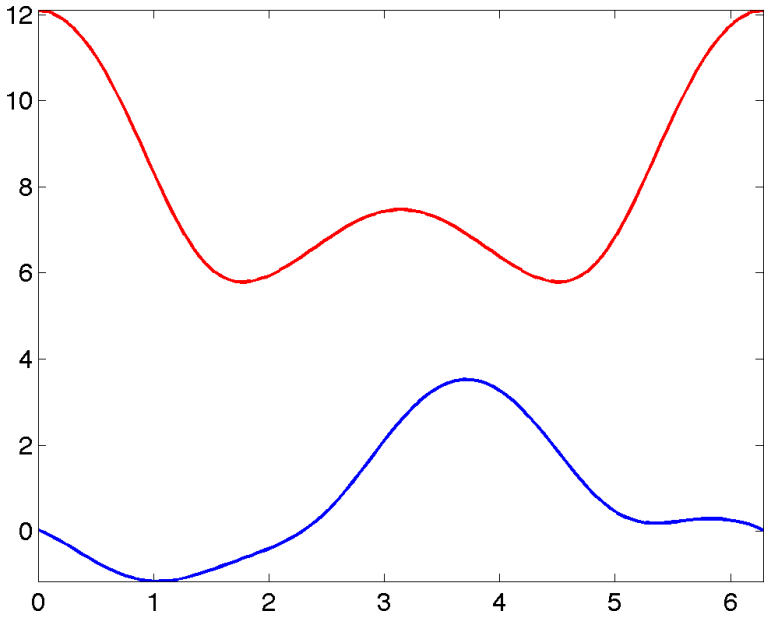

Figure 10: Visualization of the ACS of a pinched ball with incident plane wave and point source circular wave.

```
inc{2} = point_source(3+2i,k);
S.setIncidentField(inc);
S.setup(q)
S.solve();
theta = linspace(0,2*pi);
F = S.getFarField(theta,[1 2]);
plot(theta,10*log10(2*pi*abs(F(:,1)).^2),'r*')
hold on
plot(theta,10*log10(2*pi*abs(F(:,2)).^2),'mo')
legend('input:planewave', 'input:pointsource')
axis([0 2*pi -2 14])
```

10 The `solverNystromRobin` for simulating scattering models with sound-hard (Neumann) or absorbing (Robin) boundary conditions

The `solverNystromRobin` class solves the scattering problem (1)–(2) for a single particle D subject to the Robin boundary condition

$$\frac{\partial}{\partial n}(u+u^{\mathrm{i}})(\boldsymbol{x})+i\mu(u+u^{\mathrm{i}})(\boldsymbol{x})=0, \qquad \boldsymbol{x}\in\partial D,$$

where ∂D denotes the boundary of the particle. Here μ is the Robin boundary condition parameter. The case $\mu = 0$ is equivalent to the sound-hard boundary condition.

Instantiation

A `solverNystromRobin` object is created using

```
S = solverNystromRobin(k,[],b,mu);
```

where `k` is the wavenumber, `b` is an obstacle and `mu` is the Robin boundary condition parameter.

Other methods

Usage of the `solverNystromRobin` object after it is created is identical to the usage of the `solverNystrom` object described in the previous section.

Part III

Incorporating a user defined solver

11 The solver class

Abstract methods

The `tmatrix` class is instantiated with an object of class `solver`. In practice this object must be an instance of a class derived from `solver` and the derived class must provide methods with interfaces

```
setup(self)
solve(self)
val = getFarField(self,points,index)
```

We have provided the `solver` class and we recommend that user defined solvers are derived from the `solver` class using

```
classdef my_solver < solver
```

syntax. The methods above are declared as abstract methods in the `solver` class and hence they must be overwritten in the child class. We have found that provision of `setup` and `solve` methods facilitates easy extension of solver classes. Before expanding on the `setup` and `solve` methods we discuss some other methods of the `solver` class.

Constructor

The constructor for the `solver` class has interface

```
obj = solver(kwave,incidentField)
```

Here `kwave` is the wavenumber. It is expected that the solver be able to solve several scattering problems arising from several incident fields. The incident fields are given as a cell array `incidentField` and each item in the cell array should be of type `incident`. Incident fields are discussed in the next section.

The command

```
obj = solver(kwave,[])
```

defers setting the incident field. The incident field can be set subsequently using

```
obj.setIncidentField(incidentField)
```

The incident fields are again given as a cell array `incidentField` and each item in the cell array should be of type `incident`. Incident fields are discussed in the next section.

Visualization

The `solver` class also provides methods for plotting the far field and cross section. We refer to `help solver` for full details.

`setup` method

The solver is setup using

```
obj.setup();
```

This method would typically perform tasks that are performed once only, such as computing a discretization matrix.

`solve` method

The solver is applied using

```
obj.solve();
```

This method would typically perform tasks that depend on the incident wave. For example, this method might solve a linear system (with the discretization matrix) for each of several right hand sides that are derived from several given incident fields.

`getFarField` method

The far field corresponding to the incident fields is computed from a `solver` object `obj` using

```
val = obj.getFarField(points,index)
```

Here `points` is a vector of angles in $[0, 2\pi]$ and `val` is matrix with `val(:,k)` being the values of the far field induced by the incident field `incidentField{index(k)}` at the

observation angles specified in `points`.

Example

The user is free to define the `setup`, `solve` and `getFarField` methods any way that they like as long as the above functionality is provided.

A typical structure is seen in the class `solverNystrom` which is provided as an example. The `setup` method creates a matrix representing the operators in an integral equation. The `solve` method solves the linear system associated with the integral equation for each incident field and stores the coefficients. The `getFarField` method computes the far field for each incident field using the stored coefficients.

Extension

Finally, we note that the user is free to specify further useful methods in their solver class. For example, in the `solverNystrom` class we have specified further methods for visualizing the scattering obstacle and determining its radius.

Online documentation

Full documentation of `solverNystrom` is obtained using

```
help solverNystrom
```

Template

A template for a user defined solver is given in `solver_template.m`.

12 Details of incorporating a user defined solver

In this section we work through an example of constructing a user defined `solver` class. All that is required of the solver is that it can compute the far field corresponding to a given incident field.

In this example we show how to construct a `solver` class using the MPSPACK [1] solver. Before describing our `solver` class, we give a simple example of using MPSPACK to solve a single scattering problem for a sound soft scatterer described using polar coordinates. For brevity, we will use this code with minimal explanation, and we refer to the MPSPACK manual [1] for full details of how to use MPSPACK. Understanding how to solve a single scattering problem using MPSPACK will make it easy to see how the associated solver class works.

We will mainly focus on the details related to incorporating the solver MPSPACK into the object oriented framework of TMATROM. Below we assume that the user has downloaded the MPSPACK solver in a directory and added the directory to the Matlab path. To verify

that MPSPACK is installed correctly we recommend that the user types the following two commands

```
example_rom_mpspack_soundsoft_bistatic
example_rom_mpspack_soundhard_bistatic
```

First we setup the MPSPACK solver. The main tasks are to describe the boundary of the scatterer by specifying function handles `f` and `df` that give the radius and its derivative as a function of angle. These functions can be defined as anonymous functions. Then an instance of the MPSPACK `scattering` class is created and stored as `scatteringObject` and the wavenumber is set.

```
kwave = 1;
tau = 0.1;
n = 100;
m = 2*n;
opts= struct('eta',kwave,'fast',2,'multiplier',2.1,'tau',tau);
boundary = segment.radialfunc(m, {f,df});
boundary.setbc(1,'D', []);
d = domain([], [], boundary, -1);
d.addmfsbasis(boundary,n,opts);
scatteringObject = scattering(d, []);
scatteringObject.setoverallwavenumber(kwave);
```

The next part depends on the incident wave. Here we consider an incident wave stored as `inc` which is of type incident. For example

```
inc = plane_wave(pi/4,kwave);
```

We define anonymous functions `ui`, `uix` and `uiy` that evaluate the incident field, and the x- and y-coordinates of its gradient respectively. These are required by MPSPACK for simulating scattering of the incident wave.

Using these we call the `setincidentwave` method of `scattering`. Finally we call the `solvecoeffs` method of `scattering`, which solves a linear system to obtain a vector of coefficients. This vector of coefficients depends on the incident wave. We need to know that this vector is stored as `scatteringObject.co`, but it is internal to MPSPACK and we do not need to understand its true meaning.

```
ui = @(x) inc.evaluate(x);
uix = @(x) inc.evaluateGradient(x);
f = @(x) inc.evaluateGradient(x);
uiy = @(x) getSecondOutput(f,x);
scatteringObject.setincidentwave(ui,uix,uiy);
scatteringObject.solvecoeffs;
```

Finally we compute the far field at points specified in the vector `points`. For example

```
points = linspace(0,2*pi,100);
```

This computation uses the MPSACK internal vector `scatteringObject.co`.

```
farfield = scatteringObject.gridfarfield(opts,points);
```

It is clear that the computation of the far field breaks down into three distinct stages. The first is a setup stage that is independent of the incident field. The second is a solve stage that computes some internal quantity that depends on the incident field. The third stage is the far field computation. These three stages form the basis of the `setup`, `solve` and `getFarField` methods in our solver class. These are the methods that are declared as abstract in the `solver` base class.

class definition

We begin by creating a file `mfsExampleSolver.m` that contains the class declaration for our `mfsExampleSolver` class.

First we declare our new class `mfsExampleSolver` as a child of the `solver` class.

```
classdef mfsExampleSolver < solver

    % subsequent code will be inserted here

end
```

This means that the child `mfsExampleSolver` inherits several methods and properties that are defined in the parent `solver` class. In addition to these, the `mfsExampleSolver` solver class needs several properties of its own, which are mainly parameters for the MPSPACK solver.

```
classdef mfsExampleSolver < solver

    properties
        f
        df
        scatteringObject
        tau
        m
        n
        coeffs
    end

    % subsequent code will be inserted here

end
```

The `mfsExampleSolver` class also needs several methods. For now we give a template code. We will expand the function definitions below.

```
classdef mfsExampleSolver < solver
```

```
        properties
           ...
        end

        methods

            function self = mfsExampleSolver(kwave,incidentField,f,df,n,tau,m)
                ...
            end

            function setup(self)
                ...
            end

            function solve(self)
                ...
            end

            function val = getFarField(self,points,index)
                ...
            end

        end

    end
```

`mfsExampleSolver` **constructor**

First we declare the class constructor function, which must have the same name as the class. In this example we take the MPSPACK parameters as parameters of the constructor function.

```
function self = mfsExampleSolver(kwave,incidentField,f,df,n,tau,m)
```

Our first task in the constructor is to create the object `self` by calling the parent class constructor function. The object `self` is then instantiated as `mfsExampleSolver` class.

```
self = self@solver(kwave,incidentField);
```

Our next task is to copy the given parameters into the class properties.

```
self.f = f;
self.df = df;
self.m = m;
self.n = n;
self.tau = tau;
```

Finally, we set to empty the `scatteringObject` and `coeffs` properties of `self`. These

properties will be set later by other methods.

```
self.scatteringObject = [];
self.coeffs = [];
```

`setup` method

We assume that all of the parameters for the `mfsExampleSolver` are set in the constructor, so that the setup method has no parameters.

```
function setup(self)
```

Now the function body contains essentially the same code we used to setup our MPSPACK example. Note that we now use our class properties for the parameters.

```
opts= struct('eta',self.kwave,'fast',2,'multiplier',2.1,'tau',self.tau);
boundary = segment.radialfunc(self.m, {self.f,self.df});
boundary.setbc(1,'D', []);
d = domain([], [], boundary, -1);
d.addmfsbasis(boundary,self.n,opts);
```

Finally, the MPSPACK `scattering` object is stored in the `scatteringObject` property of `self` so that it can be accessed from other methods.

```
self.scatteringObject = scattering(d, []);
self.scatteringObject.setoverallwavenumber(self.kwave);
```

`solve` method

Again we assume that all of the parameters for the `mfsExampleSolver` are set in the constructor, so that the setup method has no parameters.

```
function solve(self)
```

As before, we closely follow the code we used in the MPSPACK example above. The main difference is that we need to solve for each of several incident fields given in the cell array `self.incidentField`. This necessitates a loop through the cell array.

```
for k=1:length(self.incidentField)
```

Now we essentially repeat the code from the MPSPACK example above.

```
ui = @(x) self.incidentField{k}.evaluate(x);
uix = @(x) self.incidentField{k}.evaluateGradient(x);
f = @(x) self.incidentField{k}.evaluateGradient(x);
uiy = @(x) getSecondOutput(f,x);
self.scatteringObject.setincidentwave(ui,uix,uiy);
```

In the MPSPACK example above we solved for only a single incident field and we could assume that MPSPACK internal variables were in the correct state. In this code we must reset the MPSPACK internal variable `rhs` ourselves.

```
self.scatteringObject.rhs = [];
```

Now we can solve the MPSPACK linear system using the MPSPACK `solvecoeffs` method.

```
self.scatteringObject.solvecoeffs;
```

Finally we store the MPSPACK internal coefficients in our `coeffs` array so that we can access it later in other methods.

```
self.coeffs{k} = self.scatteringObject.co;
```

`getFarField` method

The `getFarField` method is called by the `tmatrix` class and so its interface cannot be changed. The interface is

```
function val = getFarField(self,points,index)
```

where `points` is a vector of angles in $[0, 2\pi]$ and index is an array of integers between 1 and `length(self.incidentField)`. The return variable `val` is an array of size `length(points)` by `length(index)` and on return, the `k`th column of `val` must contain the far field corresponding to `self.incidentField(index(k))` at the observation angles specified in `points`.

We begin looping through the vector `index`

```
for k=1:length(index)
```

Our first task is to extract the coefficients corresponding to the `index(k)`th incident wave from the `coeffs` property and set the MPSPACK internal variable `co` appropriately.

```
self.scatteringObject.co = self.coeffs{index(k)};
```

Then we follow the MPSPACK example above to compute the far field values.

```
opts = [];
val(:,k) = self.scatteringObject.gridfarfield(opts,points);
```

Example codes

The full code for this example is given in `mfsExampleSolver.m` and an example of its use is given in `example_user_solver.m` in the subdirectory `EXAMPLE_ROM_MPSPACK_SOLVER`. In the full code we include a small amount of additional code (to allow default parameter values and some checking of parameters in methods) that we omitted above.

Finally, we remark that reuse of code can be facilitated by using more complicated nesting of child and parent classes than in the example above. In the classes `mfsPolarSoftSolver.m`, `mfsPolarHardSolver.m`, `mfsPolarSolver.m`, and `mfsSolver.m` we provide an alternative implementation of MPSPACK solvers that facilitates code reuse for sound soft and sound hard scattering problems and also for geometries with more general description than polar coordinate parametrization. Using the above involved structure, we provide two examples in the subdirectory: `example_rom_mpspack_soundsoft_bistatic.m` and `example_rom_mpspack_soundhard_bistatic.m`.

13 Incident fields

We assume that the solver can compute the far field of a given incident field provided it can compute the value of the incident field and its gradient at appropriate points specified by the solver.

The classes `plane_wave`, `point_source` and `regularwavefunction2d` represent common incident fields.

Plane wave

A `plane_wave` object with wavenumber `k` and direction `exp(1i*theta)` is created using

```
p = plane_wave(theta,k);
```

Point source

A `point_source` object with wavenumber `k` and point source location `x` is created using

```
p = point_source(x,k);
```

Regular wave function (incident field)

A `regularwavefunction2d` object with wavenumber `k`, order `n` and expansion origin `x0` is created using

```
p = regularwavefunction2d(n,k,x0);
```

Evaluating

An incident field `p` is evaluated at points `z` using

```
val = p.evaluate(z);
```

Here `z` may be a scalar, vector or matrix.

Evaluating the gradient

The gradient of an incident field `p` is evaluated at points `z` using

```
[dx,dy] = p.evaluateGradient(z);
```

Here `dx` and `dy` are the first and second components of the gradient, that is, the partial derivatives of the incident field with respect to x and y respectively.

We remark here the gradient of the incident field is a complex valued vector and hence the convention that we use elsewhere, of using complex values to represent real valued vectors, cannot be used. This is why we split the components of the gradient into `dx` and `dy`.

Acknowledgments

Support of the National Science Foundation (NSF) and Colorado Golden Energy Computing Organization (GECO) are gratefully acknowledged.

References

[1] A. Barnett and T. Betcke. *MPSpack user manual*, 2013. `http://code.google.com/p/mpspack/`.

[2] D. Colton and R. Kress. *Inverse Acoustic and Electromagnetic Scattering Theory*. Springer, 2012.

[3] T. J. Dufva, J. Sarvas, and J. C.-E. Sten. Unified derivation of the translation addition theorems for the spherical and vector wave functions. *Progress in Electromagnetics Research B*, 4:79–99, 2008.

[4] M. Ganesh and S. C. Hawkins. A far-field based T-matrix method for three dimensional acoustic scattering. *ANZIAM J.*, 50:C121–C136, 2008.

[5] M. Ganesh and S. C. Hawkins. A far-field based T-matrix method for two dimensional acoustic scattering. *ANZIAM J.*, 51:C215–C230, 2010.

[6] M. Ganesh and S. C. Hawkins. A stochastic pseudospectral and T-matrix algorithm for acoustic scattering by a class of multiple particle configurations. *J. Quant. Spect. Radiative Transfer*, 123:41–52, 2013.

[7] M. Ganesh and S. C. Hawkins. Algorithm 975: TMATROM—a T-matrix reduced order model software. *ACM Trans. Math. Software*, 44(1), 2017.

[8] M. Ganesh, S. C. Hawkins, and R. Hiptmair. Convergence analysis with parameter estimates for a reduced basis acoustic scattering T-matrix method. *IMA J. Numer. Anal.*, 32:1348–1374, 2012.

[9] M. Ganesh, J. Hesthaven, and B. Stamm. A reduced basis method for multiple electromagnetic scattering in three dimensions. *J. Comput. Phys.*, 231:7756–7779, 2012.

[10] W. J. Wiscombe. Improved Mie scattering algorithms. *Applied Optics*, 1980.