BEM.jl

This package is the result of work related to scattering in underwater acoustics. It provides implementations of the Boundary Element Method, currently for Dirichlet and Neumann boundary conditions. It also includes an accelerated algorithm based on the precorrected Fast Fourier Transform method.

Table of contents

1 Getting started

Currently, the package is unregistered and it depends on another unregistered package Mesh.jl. To get started, first manually add the dependency to the environment with `import Pkg; Pkg.add(;url="https://github.com/RuiRojo/Mesh.jl")`. Then proceed to add this package with `Pkg.add(;url="https://github/RuiRojo/BEM.jl")`.

```
>>> using BEM
```

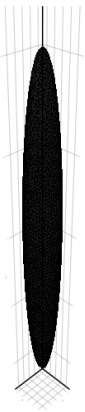
The following Examples section provides simple short examples to get a sense of how the package works. Then, Section 3 gives an overview how to manage meshes, points, directions, etc (it's currently in Spanish, my apologies). In Section 4 the basic usage of this package is explained, and more advanced details are presented in Section 5.

2 Examples

2.1 Far-field scattering pattern of a prolate spheroid

Let's load a sample 5k element mesh of a prolate spheroid.

```
>>> mesh = loadmesh(meshfile("Esferoide_5k"))
    meshplot(mesh)
```



The function `meshplot` is just a small function relying on `Makie.jl` (see Section 3).

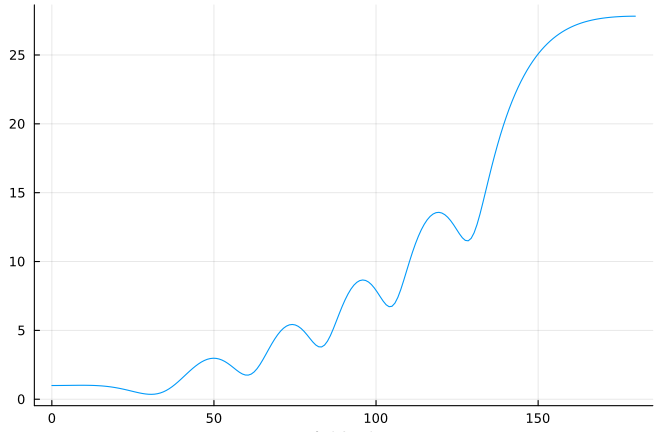
To compute the far-field angular scattering pattern under Dirichlet boundary conditions, when hit from above with a wave of wavenumber $k = 0.2$, such that the observation angle $\theta = 0$ corresponds to backscattering and $\theta = \pi$ to forward scattering, we could use the following code.

```
⇑ Define the run parameters
```

```
>>> idir = Versor([0, 0, -1]); # Incidence direction -z
>>> θs = range(0; stop=pi, length=200); # Angles of observation
>>> odirs = axis_sweep(:x, θs); # Directions of observation
>>> first(odirs) # first is z
3-element Versor:
 0.0
 0.0
 1.0
>>> last(odirs) # last is -z
3-element Versor:
  7.498798913309288e-33
  1.2246467991473532e-16
 -1.0
>>> out = scattering_farfield(mesh, idir, odirs; BC=Dirichlet, k=0.2);
```

This returns the values of f_∞, which are complex numbers. To convert to TS, use the `target_strength` function.

```
>>> plot(θs .* 360 / 2pi, target_strength.(out);
            label=false,
            ylabel="TS␣(dB)",
            xlabel="θ␣(deg)"
            )
```



As expected, TS is low at $\theta = 0$ since the acoustic wave is impinging against the tip of the spheroid and it grows larger until a maximum when it hits from the side.

3 Basic handling of meshes

This section mostly pertains to the `Mesh.jl` package.

Cargar mallas desde archivos Puede cargarse una malla en formato STL con las funciones `loadmesh_bin` o `loadmesh_ascii`, para STL binarios o ASCII respectivamente, e.g., `loadmesh_bin("path/to/mesh.stl")`. Adicionalmente, las funciones `savemesh(malla)` y `loadmesh("path/to/mesh.xxxx")` guardan y cargan mallas en formatos propios con los que puede resultar más eficiente trabajar en algunos casos. El manejo de mallas es parte del paquete propio `Mesh.jl` en https://github.com/RuiRojo/Mesh.jl.

El pequeño paquete `MeshDataset.jl` provee acceso fácil a mallas de uso interno frecuente (un subconjunto de las cuales se comparten públicamente en https://github.com/RuiRojo/MeshDataset.jl, paquete que también debería agregarse al entorno de `Julia` si se desea usar).

```
>>> using MeshDataset
```

La función `meshnames()` lista los nombres de las mallas disponibles.

```
>>> meshnames()
```

```
132-element Vector{Any}:
 "Esfera_5034"
 "Esfera_982874"
 "Esfera_320"
 "Esferoide_1_7_1495082"
 ⋮
 "Submarino_Simple_Betssi_77K"
 "Vejiga_11cm"
```

Con la función `meshfile(name)` se obtiene el *path* al archivo asociado a la malla.

```
>>> function meshplot(m; edges=false, kwargs...)
     fig, ax, = CairoMakie.plot(m; kwargs...) # Plot the boundary
     edges && CairoMakie.wireframe!(ax, m; color=:blue) # Highlight the edges
     fig
    end;
```

```
>>> meshfile("Vejiga_11cm")
```

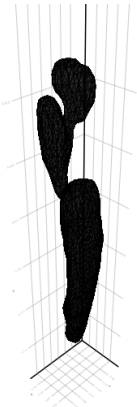
```
/mnt/main/stripe-3/data/julia/datadeps/mesh-Vejiga_11cm/Vejiga_11cm.lmesh
```

```
>>> vejiga = loadmesh(meshfile("Vejiga_11cm"));
          # Cargo la malla, guardada en formato propio lmesh
```

⇑ Visualizo

```
>>> using CairoMakie: plot, wireframe!
         # Cambiar por GLMakie para visualizar con OpenGL
```

```
>>> plot(vejiga)
```

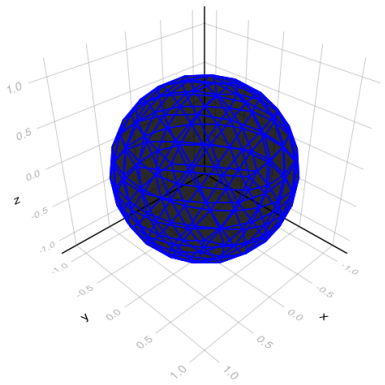


Para mayor claridad en las demostraciones, se usará una malla pequeña de 320 caras representando una esfera unitaria.

```
>>> sphere = loadmesh(meshfile("Esfera_320"));
```

⇑ Visualizo la malla

```
>>> let
       fig, ax, = plot(sphere) # Grafico la frontera
       wireframe!(ax, sphere; color=:blue) # Superpongo los aristas en azul
       fig
    end
```



Operar con mallas, caras, y vértices Las mallas importadas pueden usarse como un vector de caras, cada cara como un vector de tres vértices, y cada vértice como vector de tres puntos.

```
>>> face = sphere[1]
```

```
3-element Face:
 [0.0, 0.0, -1.0]
 [0.203181, -0.147618, -0.96795]
 [-0.077607, -0.238853, -0.96795]
```

```
>>> face[2]
```

```
3-element SVector{3, Float32} with indices SOneTo(3):
  0.203181
 -0.147618
 -0.96795
```

```
>>> sphere[1:3] # una "sub-malla" con las primeras 3 caras
```

```
3-element LightMesh:
 Float32[0.0, 0.0, -1.0]
 Float32[0.723607, -0.525725, -0.44722]
 Float32[0.0, 0.0, -1.0]
```

Los puntos como los vértices se representan con precisión simple como vectores de tamaño fijo, `SVector{3, Float32}`, del paquete `StaticArrays.jl` del registro general de `Julia`. Esto es más eficiente que usar vectores tradicionales, tanto términos de tiempo como memoria. Puede operarse con estos puntos de manera intuitiva:

```
>>> pt1, pt2, pt3 = face;
```

```
>>> pt1 + pt2
```

```
3-element SVector{3, Float32} with indices SOneTo(3):
  0.203181
 -0.147618
 -1.96795
```

```
>>> using Statistics, LinearAlgebra
```

Un punto de este tipo puede crearse con el constructor `SVector(x, y, z)` o `SA[...]`, a partir de un `Array` con la macro `@SVector`. Para más información ver la documentación de `StaticArrays.jl` [].

```
>>> SVector(1.2, 2, 3.5), SA[1.2, 2, 3.5], @SVector [1.2, 2, 3.5]
```

```
([1.2, 2.0, 3.5], [1.2, 2.0, 3.5], [1.2, 2.0, 3.5])
```

```
>>> @SVector [ i / 2 for i in 1:3 ]
```

```
3-element SVector{3, Float64} with indices SOneTo(3):
 0.5
 1.0
 1.5
```

Versores Un versor se representa por `Versor(φ, θ)`

```
>>> vers = Versor(0, π/2) # (φ=0, θ=π/2) equivale al versor (1, 0, 0)
```

```
3-element Versor:
 1.0
 0.0
 6.123233995736766e-17
```

```
>>> vec = SVector(vers) # convierto un Versor a SVector
```

```
3-element SVector{3, Float64} with indices SOneTo(3):
 1.0
 0.0
 6.123233995736766e-17
```

```
>>> Versor(SVector(0, 10, 0)) # el versor en la dirección de un SVector
```

```
3-element Versor:
 6.123233995736766e-17
 1.0
 6.123233995736766e-17
```

Algunos ejemplos de operaciones sobre las caras:

```
>>> mean(face) # centroide: promedio de los vértices de la cara
```

```
3-element SVector{3, Float32} with indices SOneTo(3):
  0.041858
 -0.12882365
 -0.97863334
```

```
>>> area(face) # área
```

```
0.03036415f0
```

```
>>> norm(cross(pt2 - pt1, pt3 - pt1)) / 2 # área calculada con el producto
    vectorial
```

```
0.03036415f0
```

```
>>> maximum( area.(sphere) ) # área máxima de cara
```

```
0.045715254f0
```

```
>>> let as = area.(sphere) # Relación entre el área 95% mayor y la 5% menor
        quantile( as, 0.95) / quantile( as, 0.05 )
    end
```

```
1.5054925004642263
```

Siempre pueden convertirse los puntos de `SVector` a vectores tradicionales usando la función `collect`.

```
>>> collect(pt1) # convertir un punto a un vector tradicional
```

```
3-element Vector{Float32}:
  0.0
  0.0
 -1.0
```

```
>>> collect.(face) # convertir la cara a un vector tradicional de 3 vectores
    tradicionales
```

```
3-element Vector{Vector{Float32}}:
 [0.0, 0.0, -1.0]
 [0.203181, -0.147618, -0.96795]
 [-0.077607, -0.238853, -0.96795]
```

La malla funciona también como iterador sobre las caras:

```
>>> mean( mean(f) for f in sphere ) # promedio de los centroides
```

```
3-element SVector{3, Float32} with indices SOneTo(3):
  1.21071935f-8
 -3.2410025f-8
  6.575137f-8
```

```
>>> for f in vejiga[1:4] # itero sobre las caras de la sub-malla
        l1 = f[3] - f[2] |> norm # longitud de aristas
        l2 = f[2] - f[1] |> norm
        l3 = f[3] - f[1] |> norm
        println( (l1 + l2 + l3) / 3 ) # imprimo promedio de longitud de
    aristas
    end
```

```
7.733009e-5
0.00025584974
0.00029891744
0.0003006349
```

La función `vertices(malla)` devuelve un vector con los vértices de la malla.

```
>>> vertices(sphere)
```

```
162-element Vector{SVector{3, Float32}}:
 [0.0, 0.0, -1.0]
 [0.203181, -0.147618, -0.96795]
 [-0.077607, -0.238853, -0.96795]
 [0.723607, -0.525725, -0.44722]
 [0.609547, -0.442856, -0.657519]
 [0.812729, -0.295238, -0.502301]
 [-0.251147, 0.0, -0.967949]
 [-0.077607, 0.238853, -0.96795]
 [0.203181, 0.147618, -0.96795]
 [0.860698, -0.442858, -0.251151]
 ⋮
 [-0.162456, -0.499995, -0.850654]
 [-0.232822, -0.716563, -0.657519]
 [0.670817, 0.162457, -0.723611]
 [0.670818, -0.162458, -0.72361]
 [0.447211, -1.0f-6, -0.894428]
 [0.425323, -0.309011, -0.850654]
 [0.05279, -0.688185, -0.723612]
 [0.138199, -0.425321, -0.894429]
 [0.361805, -0.587779, -0.723611]
```

```
>>> mean( vertices(sphere) ) # centroide de todos los vértices
```

```
3-element SVector{3, Float32} with indices SOneTo(3):
  2.2811655f-8
 -3.532127f-8
  1.1773757f-8
```

La función `boundingbox(mesh)` devuelve la extensión de la malla en formato `((xmin, xmax), (ymin, ymax), (zmin, zmax))` representando las coordenadas de la caja mínima que contiene a la malla y está alineada a los ejes coordenados:

```
>>> boundingbox(vejiga)
```

```
((-0.004953303f0, 0.001229969f0), (-0.002669825f0, 0.001681417f0),
(-0.01641069f0, 0.01347215f0))
```

Internamente, la malla es un objeto de tipo `LightMesh` que contiene dos campos:

- `:vertices`, conteniendo el vector de vértices, tal como devuelve `vertices(malla)`.
- `:selv`, conteniendo el vector de caras cada una representada como una terna (i_1, i_2, i_3) con los índices de sus vértices según `:vertices`. Los índices se guardan como enteros de 4 *bytes* sin signo — `UInt32` — para ahorrar memoria, pues el valor máximo de $2^{32} > 4 \times 10^9$ excede el cantidad de vértices con el que se puede aspirar a trabajar.

```
>>> sphere |> typeof
```

```
LightMesh
```

```
>>> fieldnames(LightMesh)
```

```
(:vertices, :selv)
```

```
>>> sphere.selv[17] # Los índices de la cara 17 como UInt32
```

```
(0x0000001f, 0x00000020, 0x00000021)
```

```
>>> selv = convert.(NTuple{3, Int}, sphere.selv); # Convierto los índices a
    enteros tradicionales Int64
>>> selv[17]
```

```
(31, 32, 33)
```

Aristas de la malla La función `edges(malla)` devuelve un iterador sobre las aristas que también puede usarse como vector:

```
>>> length(edges(vejiga)) # cantidad de aristas
```

```
7404
```

```
>>> e = edges(vejiga)[1] # tomo la primer arista
```

```
2-element Edge:
 [-0.002501244, -0.002140007, -0.01372092]
 [-0.002325444, -0.002003637, -0.01372092]
```

```
>>> e[1] # el primer punto
```

```
3-element SVector{3, Float32} with indices SOneTo(3):
 -0.002501244
 -0.002140007
 -0.01372092
```

```
>>> norm(e) # la longitud
```

```
0.019919474f0
```

```
>>> mean( norm.(edges(vejiga)) ) # el promedio de las longitudes
```

```
0.011276887f0
```

Se provee también la función `edgelenstats(mesh)` para un resumen rápido de las estadísticas principales de las longitudes de las aristas de la malla: promedio (`:mean`), mínima (`:minimum`), máxima (`:maximum`), y los nueve deciles del 0.1 al 0.9 (`.deciles`).

```
>>> edgelenstats(vejiga)
```

```
(deciles = [0.00015344667190220207, 0.00024589489912614287,
0.00030955811962485313, 0.0003346722514834255, 0.000424931168678105,
0.0005927901947870851, 0.0006003672606311738, 0.0006186825456097725,
0.000672905589453876], mean = 0.00042851715f0, minimum = 5.4105085f-6, maximum =
0.00072077557f0)
```

Normales — orientación de las caras La orientación de las caras es parte esencial de la geometría del dispersor. En un cara, un lado se va a corresponder al exterior del dispersor y otro al interior. Dos caras pueden ser idénticas en cuanto a los puntos del espacio que ocupan pero apuntar en sentidos opuestos. Por convención, el vector normal a una cara se interpreta como apuntando hacia el exterior del dispersor.

El vector normal a una cara puede obtenerse con la función `normal(cara)`.

```
>>> normal(face)
```

```
3-element SVector{3, Float32} with indices SOneTo(3):
  0.04815025
 -0.14818889
 -0.9877862
```

Las normales a las caras se computan a partir del orden de sus tres vértices según la regla de la mano derecha. Incluso si la malla se cargó a partir de un archivo STL por medio de `loadmesh_bin` o `loadmesh_ascii` y el archivo incluye información independiente sobre las normales, esto se descarta inmediatamente bajo la asunción de que el archivo es válido[1] y por lo tanto el ordenamiento de los vértices es consistente con las normales.

Es **responsabilidad del usuario** que las **normales** calculadas apunten **hacia afuera** del dispersor. Como ayuda, se provee la función `BEM.check_normals(malla)` que verifica que todas las normales apunten en el mismo sentido del vector que une el centro de la cara con el centro del objeto. Esto es una condición suficiente pero, en geometrías complejas, no necesaria. Eventualmente, se podría implementar una corrección completamente automática que determine la orientación deseada en base a criterios de que el dispersor debe ser una geometría cerrada y acotada y las caras adyacentes deben siempre apuntar en un mismo sentido (Figure 1).

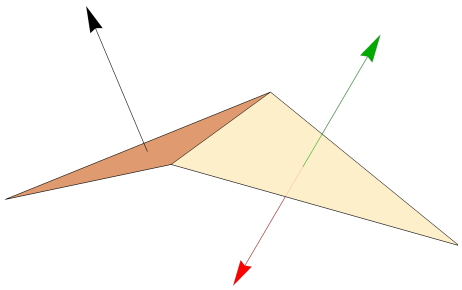


Figure 1. Las normales de la malla cargada deben consistentemente apuntar hacia el exterior. En la figura, si la flecha negra es la normal del triángulo de la izquierda, sólo la verde es una normal aceptable.

4 The basic API

The highest level functions provided to compute scattering are `backscattering`, `forward_scattering`, `scattering_nearfield` and `scattering_farfield`. They all have similar signatures:

- `backscattering(mesh, idir; k, BC, kwargs...)`
- `forward_scattering(mesh, idir; k, BC, kwargs...)`

1. El formato STL exige que se interprete la normal como apuntando al exterior del objeto y que los vértices se listen en el orden contrario a las agujas del reloj cuando se observa al objeto desde el exterior, siguiendo la regla de la mano derecha [].

- `scattering_nearfield(mesh, idir, outpt; k, BC, kwargs...)`
- `scattering_farfield(mesh, idir, outdir; k, BC, kwargs...)`

The incidence direction `idir` is represented as a `Versor`[2] (see Section 3). Similarly, `outdir` is a versor specifying the direction of observation, and `outpt` is the point of observation (assumed to be outside of the mesh but not necessarily far from it).

The boundary condition is specified through the `BC` keyword argument, and can currently take one of two valuse: `Dirichlet`, or `Neumann`.

The result will be f_∞ and can be converted to TS with the function `target_strength`.

These four functions accept alternative forms for repeated calculations that share the mesh and wavenumber k, such as when performing an angular sweep. These are not only for convenience but they may provide some optimizations and internal caching of shared partial results.

So, for example, say `idirs` is a vector with the desired incidence directions, and `k` and `BC` are already defined, this would be a better way to compute the backscattering or forward scattering over all the `idirs`:

```
# These produce the same results
[ backscattering(mesh, idir; k, BC) for idir in idirs ] # direct way

backscattering(mesh, idirs; k, BC) # better

# and so do these
[ forward_scattering(mesh, idir; k, BC) for idir in idirs ] # direct way

forward_scattering(mesh, idirs; k, BC) # better
```

The functions `scattering_nearfield` and `scattering_farfield` also support similar alternative forms:

```
[ scattering_nearfield(mesh, idir, outpt; k, BC) for outpt in outpts ]

scattering_nearfield(mesh, idir, outpts; k, BC)

map(
   scattering_nearfield(mesh, idir; k, BC), # Curried form
   outpts
   )
```

All of these functions are just syntactic sugar. Behind the scenes, they all go through one main high level function: `scattering_boundary`. This function computes the boundary values of the BVP, which is where the bulk of the hard computation lies. The result is a data structure of type `BoundaryValues` which can then be used to compute the scattering at any desired point efficiently.

It presents a similar signature as the others.

```
bv = scattering_boundary(mesh, idir; k, BC)
```

The type of the output, `BoundaryValues`, has been provided definitions so that it can be used as a functor to compute the scattering. When called with point, it will return the nearfield, and when called with a versor, it will return the farfield.

```
bv(SVector(10, 2.3, 54)) # near-field calculation
bv(Versor(0, 0)) # far-field at z (φ=0, θ=0)
```

2. In some cases, if a vector is provided instead of a versor, the function converts it internally.

`scattering_boundary` also accepts multiple incidence directions at once. This forms returns an iterator (a `Channel`) that generates the boundary values only as needed and, when possible, reuses the calculations that depend on the mesh but not the incidence direction. Under the hood, it's calling a curried form of `scattering_boundary` where the incidence direction is not specified.

```
bvs = scattering_boundary(mesh, idirs; k, BC)
[ bvfun(Versor(0, 0))) for bv in bvs ]
```

5 Advanced usage

All the high-level functions described in Section 4 can be fine tuned through similar optional keyword agruments.

The difference is that, now, the fine-tuning is done with these keyword arguments:

- `formulation[=IndirectFormulation]`: possible values are can be `DirectFormulation`, `IndirectFormulation`
    - `DirectFormulation`
    - `IndirectFormulation`
    - One of the surface integral operators `L/M/Mt/N`
- `algorithm[=Exact()]`: How to produce the surface integral operators.
    - `Exact(;quad[=])`
    - `Exact{LinearMap}(;quad[=])`
    - `Accelerated(;kwargs...)`
- These next two arguments, if not passed, will take automatic defaults that depend on the other arguments, but can be fine-tuned if desired:
    - μ: This can be a number or a a function of `k`. It currently defaults to $i/(1+k)$ where $i^2=-1$.
    - `solver`: This defaults to GMRES for `Accelerated` and `Exact{LinearMap}`, and to a direct solver through LU decomposition for `Exact{Matrix}` (see Section 5.1).

5.1 Algorithm

By default, the code uses an "exact" BEM method that fully computes the system matrix for each of the surface integral operators required by the formulation (two out of `L/M/Mt/N`). This is represented by an object of tpye `Exact{Matrix}`.

Alternatively, one can provide an algorithm of type `Exact{LinearMap}` which will do the same computations but without pre-computing the whole matrix of each of the operators. This reduces the memory footprint, but it can be prohibitively slower since each element has to be computed every time it is required.

The behaviour of these both algorithms can be constructed and customized through the following two keyword arguments:

- `quad[=@quad_gcuts 5]`: The quadrature rule used for the numerical integrations involved in the computation. This is given as a `QuadratureRule` object containing the points and their respective weights in a normalized triangle. The typical usage is to generate valid quadrature rules by one of the two macros:
    - `@quad_gcuts n`: generates quadrature rules of $n \times n$ points.

- `@quad_gcutm n`: generates quadrature rules of $n \times (n+1)/2 - 1$ points.

They are also provided as functions, but since the macro form might prove more efficient in some circumstances, given that it's equivalent to hard coding the rule. These are part of the `Mesh.jl` package.

- `verbose[=false]`: When set to `true`, progress reports will be issued to the command line.

```
# Example of algorithm customization
algorithm = Exact{LinearMap}(
   quad = @quad_gquts 8,
   verbose = true
   )

backscattering(mesh, idirs; k, BC, algorithm)
```

5.1.1 Accelerated algorithm: pFFT

The other algorithm provided performs an approximate calculation, but can be significantly more efficient for larger meshes. It is represented by the `Accelerated` type, and offers more customization options than the `Exact` algorithms, in addition to their `quad` and `verbose` options, which have similar meanings here.

The most important ones are:

- `gridfactor[=1]`. This determines the dimensions of the grid cells in relation to the largest edge lengths in the mesh (taken as the 90% percentile). The larger the `gridfactor`, the smaller the grid cells and the grid dimensions increase, which could improve the quality of the results at the cost of a higher computational burden.

- `r_near[=0]`. This is the radius of the neighbourhood of each face that will be computed exactly instead of using the accelerated approximation. Regardless of this number, the closest neighbouring faces will always be computed exactly since the approximation fails when face pairs share neighbouring nodes (unless the option `nearest_neighbours` is manually set to `false`). Higher values can improve the quality of the results at the expense of compute.

There are other parameters that can be tuned, but they are considered less stable, e.g., those related to the specific implementation of interpolator provided (`h_nterms` relates to the number of elements of the polynomial basis used to interpolate around each face, and `scaleinterp` determines whether node scale normalization is applied) and to the convolver, projector and fixer.

```
# Example of algorithm customization
algorithm = Accelerated(
   r_near = 5,
   gridfactor = 0.5,
   h_nterms = [ 8, 17, 25 ],
   quad = @quad_gquts 8,
   verbose = true
   )

backscattering(mesh, idirs; k, BC, algorithm)
```

5.2 GMRES

The `solver` argument should be redefined when fine-tuning the GMRES parameters (or when the default is not an iterative solver as with the `Exact{Matrix}` algorithm).

Typically, one re-uses the function `gmres` provided by the package, which itself is a slightly tuned wrapper around the functionality provided by the package `IterativeSolvers`. For example,

```
solver(A, b) = gmres(A, b; reltol_posta=1e-4)

backscattering(mesh, idirs; k, BC, algorithm)
```

In this way, one can tune things like

- `restart[=min(200, size(A, 2))]`: the number of iterations such that, if the algorithm hasn't converged up to that point, it is restarted from the residual, i.e., the partial solution is used as initial guess in a new run. This prevents runaway memory usage, since the GMRES method has to store the partial Krylov basis.

- `maxiter[=200]` : the maximum number of iterations before giving up on full convergence.

- `reltol_posta[=1e-3]`: the relative error between $b = Ax$ and $\hat{b} = A\hat{x}$, obtained with the current best estimate of $\hat{x} \approx x$.

- `guess[=nothing]`: This currently can take the values `nothing` (default), which just starts from a zero guess; or it can take the value `guess=:last`, which makes it start from the last solution. This can be useful when doing a fine angular sweep where successive runs could be expected to converge to similar results.

- Logging related:

  - `each_iter[=identity]`: a monitoring function to be called on x on each run of Ax.

  - `verbose[=false]` / `very_verbose[=false]`.

6 Resource usage, monitoring, and other final points

- Progress monitoring for backscattering and forward scattering sweeps can be activated by defining a logger as described in `ProgressLogging.jl` (VSCode already comes with a default logger).

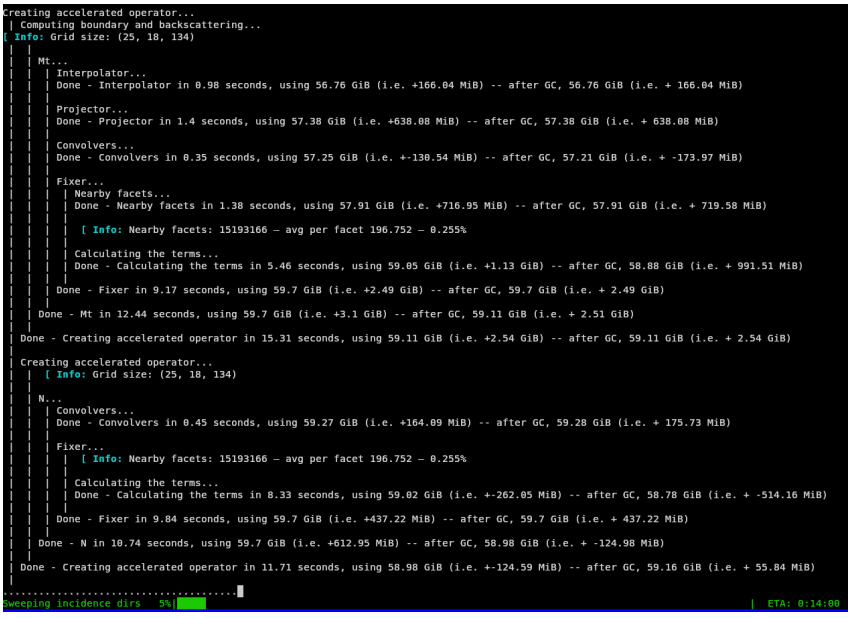


Figure 2. Example of the progess logs for a backscattering sweep given certain verbosity settings.

- A timer is provided, using the package `TimerOutputs.jl`, and can be displayed and reset at the global variable `BEM.to`. Currently, it misreports when running sweeps.

		Time			Allocations		
Tot / % measured:		1173s / 36.7%			410GiB / 99.4%		
Section	ncalls	time	%tot	avg	alloc	%tot	avg
Computing boundary	2	430s	100.0%	215s	407GiB	100.0%	204GiB
Creating operator A	1	324s	75.4%	324s	395GiB	96.9%	395GiB
Creating accelerated operator	2	323s	75.0%	161s	395GiB	96.9%	197GiB
N	1	183s	42.6%	183s	199GiB	48.8%	199GiB
Fixer	1	178s	41.5%	178s	197GiB	48.4%	197GiB
Calculating the terms	1	152s	35.3%	152s	182GiB	44.6%	182GiB
Creating sparse operator	1	25.5s	5.9%	25.5s	15.2GiB	3.7%	15.2GiB
Convolvers	1	3.98s	0.9%	3.98s	1.82GiB	0.4%	1.82GiB
Convolver	9	3.69s	0.9%	410ms	1.80GiB	0.4%	205MiB
Mt	1	138s	32.1%	138s	196GiB	48.1%	196GiB
Fixer	1	111s	25.7%	111s	188GiB	46.3%	188GiB
Calculating the terms	1	88.3s	20.5%	88.3s	171GiB	41.9%	171GiB
Creating sparse operator	1	16.6s	3.9%	16.6s	13.1GiB	3.2%	13.1GiB
Nearby facets	1	4.50s	1.0%	4.50s	4.78GiB	1.2%	4.78GiB
Interpolator	1	12.5s	2.9%	12.5s	1.36GiB	0.3%	1.36GiB
Projector	1	8.86s	2.1%	8.86s	5.26GiB	1.3%	5.26GiB
Convolvers	1	3.53s	0.8%	3.53s	684MiB	0.2%	684MiB
Convolver	3	2.49s	0.6%	829ms	617MiB	0.1%	206MiB
Solving system	1	105s	24.4%	105s	12.5GiB	3.1%	12.5GiB
Applying gmres	1	105s	24.4%	105s	12.5GiB	3.1%	12.5GiB
Applying	60	103s	23.9%	1.71s	11.9GiB	2.9%	204MiB
Convolving	360	47.1s	10.9%	131ms	1.33GiB	0.3%	3.80MiB
IFFT	360	30.8s	7.2%	85.7ms	58.9MiB	0.0%	167KiB
FFT	360	3.41s	0.8%	9.47ms	669KiB	0.0%	1.86KiB
Projecting	60	15.7s	3.6%	261ms	6.62GiB	1.6%	113MiB
Interpolating	180	5.23s	1.2%	29.0ms	2.79GiB	0.7%	15.9MiB
Computing incident field in boundary	1	25.0ms	0.0%	25.0ms	3.53MiB	0.0%	3.53MiB

Figure 3. Example of the provided timer output.

- Many of the package functions support setting `verbose=true` as an argument, in which case progress logs and other information are issued.

- There are many things left to test and optimize in the code.

    - The convolutions are done by zero padding the 3D arrays (in all three dimensions), which can be improved.

    - Copies are stored for optimization puruposes, like of the Green function's FFT, which are not strictly necessary, and result from the typical compromise between memory and speed. However, speed hasn't been thoroughly tested to know if the compromise is worth it.

    - The garbage collector has been observed to not reliably kick in when it could, and the codebase is sprinkled with explicit calls to collect the garbage. However, this hasn't been tested in the latests versions on `Julia`, which have made related improvemens.

- It's been observed that sometimes (like while computing the fixer) it uses more memory than one would think its necessary, though it often drops after finishing. This needs further debugging.

- Each operator is treated independently (though there's some memoization going on), even though they could be grouped as a single operator. In the `Exact{Matrix}` algorithm, this is an obvious needless waste. This also means that the operator `N` take 9 times more resources than operator `L`, which makes some formulations much more computationally intensive than others.

- The codebase still containes "extra fat" —remains from the dirty process of learning and prototyping and iterating and learning again— which hasn't been trimmed and presently serves no purpose. Similarly, comments and function in-code documentation may not be fully complete or up to date. This will be improved soon.

- The package provides multiple additional small undocumented functions that can help in the typical workflow. These include `target_strength`, `intensity`; `BEM.axis_sweep` to generate versors to sweep around axis; `area`, `edgelenstats`, `boundingbox`, `mesh_a`, and others to perform common statistics on meshes, `refineFlat` to refine a mesh while retaining its shape and minimizing variance in edge length, etc.

- It also exposes (currently undocumented) functions to work with the isolated surface integral operators.