
DSMRayTracerDoc Documentation

Release 0

Hayley

Aug 30, 2019

1	Main Program	3
2	Input the parameters	7
3	Rays	9
4	Room	13
5	Mesh	17
6	Reflection	25
7	Indices and tables	27
	Python Module Index	29
	Index	31

Contents:

Main Program

Code to trace rays around a room. This code computes the trajectories only.

RayTracerMainProgram.**MeshProgram**()

Refect rays and output the Mesh containing ray information.

Parameters for the raytracer are input in *ParameterInput* The raytracing parameters defined in this module are saved and then loaded.

- ‘Raytracing.npy’ - An array of 4 floats which is saved to [Nra (number of rays), Nre (number of reflections), h (relative meshwidth), L (room length scale, the longest axis has been rescaled to 1 and this is it’s original length)]
- ‘Obstacles.npy’ - An array for 3x3x1 arrays containing co-ordinates forming triangles which form the obstacles. This is saved to Oblist (The obstacles which are within the outerboundary)
- ‘Origin.npy’ - A 3x1 array for the co-ordinate of the source. This is saved to Tx (The location of the source antenna and origin of every ray)
- ‘OuterBoundary.npy’ - An array for 3x3x1 arrays containing co-ordinates forming triangles which form the obstacles. This is saved to OuterBoundary (The Obstacles forming the outer boundary of the room)

Put the two arrays of obstacles into one array

```
Oblast=[Oblast,OuterBoundary]
```

- ‘Directions.npy’ - An Nrx3x1 array containing the vectors which correspond to the initial direction of each ray. This is save to Direc.

A room is initialised with *Oblast* using the *room* class in *Room*.

The number of obstacles and the number of x, y and z steps is found

```
Nob=Room.Nob
Nx=int(Room.maxxleng()/h)
Ny=int(Room.maxyleng()/h)
Nz=int(Room.maxzleng()/h)
```

Initialise a *DictionarySparseMatrix*. DS with the number of spaces in the x, y and z axis Nx, Ny, Nz, the number of obstacles Nob, the number of reflections Nre and the number of rays Nra.

```
Mesh=DSM.DS(Nx,Ny,Nz,int(Nob*Nre+1),int((Nre)*(Nra)+1))
```

Find the reflection points of the rays and store the distance and reflection angles of the rays in the Mesh. Use the *room*. *ray_mesh_bounce*(Tx,Nre,Nra,Dirac,Mesh)() function.

```
Rays, Mesh=Room.ray_mesh_bounce(Tx,int(Nre),int(Nra),Direc,Mesh)
```

Save the reflection points in Rays to 'RayMeshPointsNraRefsNren.npy' making the substitution for Nra and Nre with their parameter values.

Returns Mesh

RayTracerMainProgram.**RayTracer**()

Refect rays and output the points of reflection.

Parameters for the raytracer are input in *ParameterInput* The raytracing parameters defined in this module are saved and then loaded.

- 'Raytracing.npy' - An array of 4 floats which is saved to [Nra (number of rays), Nre (number of reflections), h (relative meshwidth), L (room length scale, the longest axis has been rescaled to 1 and this is it's original length)]
- 'Obstacles.npy' - An array for 3x3x1 arrays containing co-ordinates forming triangles which form the obstacles. This is saved to Oblist (The obstacles which are within the outerboundary)
- 'Origin.npy' - A 3x1 array for the co-ordinate of the source. This is saved to Tx (The location of the source antenna and origin of every ray)
- 'OuterBoundary.npy' - An array for 3x3x1 arrays containing co-ordinates forming triangles which form the obstacles. This is saved to OuterBoundary (The Obstacles forming the outer boundary of the room)

Put the two arrays of obstacles into one array

```
Oblist=[Oblist,OuterBoundary]
```

- 'Directions.npy' - An Nrx3x1 array containing the vectors which correspond to the initial direction of each ray. This is save to Direc.

A room is initialised with *Oblist* using the *room* class in *Room*.

Find the reflection points of the rays using *room.ray_bounce*(Tx,Nre,Nra)() function.

```
Rays, Mesh=Room.ray_bounce(Tx,Nre,Nra,Direc)
```

Save the reflection points in Rays to 'RayPointsNraRefsNren.npy' making the substitution for Nra and Nre with their parameter values.

Returns 0 to mark a successful run

RayTracerMainProgram.**RefCoefComputation**(Mesh)

Compute the mesh of reflection coefficients.

Parameters Mesh – The DS mesh which contains the angles and distances rays have travelled.

Load the physical parameters using *ParameterInput*. *ObstacleCoefficients*() * Znobrat - is the vector of characteristic impedances for obstacles divided by the characteristic impedance of air. * reindex - if the vector of refractive indexes for the obstacles.

Compute the Reflection coefficients (RefCoefper,Refcoefpar) using: *DictionarySparseMatrix.ref_coef*(Mesh,Znobrat,reindex)():return: (RefCoefper,Refcoefpar)

RayTracerMainProgram.**RefCombine**(Rper,Rpar)

Combine reflection coefficients to get the loss from reflection coefficient for each ray segment.

Take in the DS's (*DictionarySparseMatrix*. DS) corresponding to the reflection coefficients for all the ray interactions (*DictionarySparseMatrix.ref_coef*(Mesh)()).

Use the function `DictionarySparseMatrix.DS.dict_col_mult()` to multiple reflection coefficients in the same column.

```
Combper=[
[prod(nonzero terms in column 0 in Rper[0,0,0]),
prod(nonzero terms in column 1 in Rper[0,0,0]),
...,
prod(nonzero terms in column nb in Rper[0,0,0]
],
...,
[prod(nonzero terms in column 0 in Rper[Nx-1,Ny-1,Nz-1]),
prod(nonzero terms in column 1 in Rper[Nx-1,Ny-1,Nz-1]),
...,
prod(nonzero terms in column nb in Rper[Nx-1,Ny-1,Nz-1]
]
]
```

```
Combpar=[
[prod(nonzero terms in column 0 in Rpar[0,0,0]),
prod(nonzero terms in column 1 in Rpar[0,0,0]),
...,
prod(nonzero terms in column nb in Rpar[0,0,0]
],
...,
[prod(nonzero terms in column 0 in Rpar[Nx-1,Ny-1,Nz-1]),
prod(nonzero terms in column 1 in Rpar[Nx-1,Ny-1,Nz-1]),
...,
prod(nonzero terms in column nb in Rpar[Nx-1,Ny-1,Nz-1]
]
]
```

Parameters

- **Rper** – The mesh corresponding to reflection coefficients perpendicular to the polarisation.
- **Rpar** – The mesh corresponding to reflection coefficients parallel to the polarisation.

Return type DS(Nx,Ny,Nz,1,nb)

Returns Combper, Combpar

RayTracerMainProgram.**power_grid()**

Calculate the field on a grid using enviroment parameters and the ray Mesh.

This is something I want to say that is not in the docstring.

Input the parameters

The code saves the values for the parameters in a ray tracer

`ParameterInput.DeclareParameters()`

All input parameters for the ray-launching method are entered in this function which will then save them inside a Parameters folder.

- Nra - Number of rays

Note: Due to need of integer steps the input number of rays can not always be used if everything is equally spaced.

- Nre - Number of reflections
- Ns - Number of steps to split longest axis.
- l1 - Interior obstacle scale
- l2 - Boundary scale.
- triangle1 - First interior obstacle
- ...
- triangleN - Last interior obstacle
- OuterBoundary1 - First obstacle forming the boundary of the environment
- ...
- OuterBoundaryN - Last obstacle forming the boundary of the environment.

Returns 0 if successfully completed.

`ParameterInput.ObstacleCoefficients()`

Input the parameters for obstacles and the antenna. To ensure arrays are of the right length for compatibility for the ray-launcher retrieve the ray-launching parameters in `DeclareParameters()`

Load:

- ‘Obstacles.npy’ -Co-ordinates of obstacles in the room
- ‘OuterBoundary.npy’ - Co-ordinates of the walls of the room
- ‘Raytracing.npy’ -[Nra (number of rays), Nre (number of reflections), h (relative meshwidth)]

Calculate:

•Nob=len([Obstacles,OuterBoundary])

Input:

- Freespace* -[mu0 (permeability of air), eps0 (permittivity of air),Z0 (characteristic impedance of air), c (speed of light)]
- frequency* - ω angular frequency of the wave out the antenna.
- mur* - μ_r The relative permeability for all obstacles. This should be an array with the same number of terms as the number of obstacles Nob.
- epsr* - ϵ_r The relative permittivity for each obstacle. This should be an array with the same number of terms as the number of obstacles Nob.
- sigma* - σ The electrical conductivity of the obstacles. This should be an array with the same number of terms as the number of obstacles.
- Gt* - The gains of the antenna. The should be an array with the same number of terms as the number of rays Nra.

Calculate:

- eps0* - $\epsilon_0 = \frac{1}{\mu_0 c^2}$ permittivity of freespace.
- Z0* - $Z_0 = \sqrt{\frac{\mu_0}{\epsilon_0}}$ characteristic impedance of freespace.
- refindex* - The refractive index $n = \sqrt{\mu_r \epsilon_r}$
- Znobrat*- The relative impedance of the obstacles given by, $\hat{Z}_{Nob} = \frac{Z_{Nob}}{Z_0}$. The impedance of each obstacle Z_{Nob} is given by $Z_{Nob} = \sqrt{\frac{i\omega\mu_0\mu_r}{\sigma+i\epsilon_0\epsilon_r}}$.

The Znobrat and refindex terms are then reformatted so that they repeat Nre times with an extra term. The extra term corresponds to the line of sight path. This makes them the same length as a column in a matrix in a *DictionarySparseMatrix.DS*. Each term corresponds to a possible obstacle reflection combination.

The Gains matrix is also reformatted to that it repeats (Nre+1) times. This corresponds to every possible ray reflection number combination This makes them the same length as a row in a matrix in a *DictionarySparseMatrix.DS*. Each term corresponds to a possible obstacle reflection combination.

Save: * *frequency.npy*- The angular frequency ω . * *refindex.npy* - The refractive index of the obstacles. * *Znobrat.npy* - The relative characteristic impedance. * *TxGains.npy* - The gains of the antenna. * *Freespace.npy*- The freespace parameters.

```
Freespace=np.array([mu0,eps0,Z0,c])
```

Returns 0 if successfully completed.

Rays

Code to construct the ray-tracing objects rays

class Rays.**Ray** (*s, origin, direc*)

A ray is a representation of the the trajectory of a reflecting line and its reflections. Ray.points is an array of co-ordinates representing the collision points with the last term being the direction the ray ended in. And Ray.reflections is an array containing tuples of the angles of incidence and the number referring to the position of the obstacle in the obstacle list

mesh_multiref (*s, room, Nre, Mesh, Nra, nra*)

Takes a ray and finds the first Nre reflections within a room. As each reflection is found the ray is stepped through and information added to the Mesh. :param room: Obstacle co-ordinates, *Room. room*. :param Nre: Number of reflections, integer. :param Mesh: A grid with corresponding sparse matrices, this is a *DictionarySparseMatrix*. DS object. :param Nra: Total number of rays, integer.

Method:

- Create a temporary vector vec.
- For each ray segment use `mesh_singleray(room, Mesh, dist, vec, Nra, Nre, nra)()` to segment storing `r*calvec` in the Mesh. With `r` being the distance ray travelled to get centre of the grid point the ray has gone through.

Returns Mesh

mesh_singleray (*s, room, Mesh, dist, calvec, Nra, Nre, nra*)

Iterate between two intersection points and store the ray information in the Mesh

Parameters

- **room** – *Room. room* object which contains the co-ordinates of the obstacles.
- **Mesh** – *DictionarySparseMatrix*. DS which will store all of the ray information.
- **dist** – A scalar variable which is the distance the ray travelled at the start of the ray segment.
- **calvec** – A vector containing $e^{i\theta}$ terms for reflection angles :math: 'heta'. These terms are stored in row `nre*Nob+nob` with `nre` being the current reflection number, `Nob` the maximum obstacle number and `nob` the number of the obstacle which was hit with the corresponding angle.
- **Nra** – Total number of rays.
- **Nre** – Maximum number of reflections.
- **nra** – Current ray number.

Method:

- Calculate θ the reflection angle using `reflect_angle(room)()`.
- Find the number of steps N_s to the end of the ray segment using `number_steps(meshwidth)()`.
- Compute an array of normal vectors representing the ray cone.
- Check the reflection number: * If 0 then the `calcvec[0]` term is 1. * ElseIf 1 then set `calcvec[0] = 0` and `calcvec[nre * Nob + nob] = e^{i\theta}`. * Else set `calcvec[nre * Nob + nob] = e^{i\theta}`.
- Step along the ray, For $m1 \in [0, N_s)$: * Check if the ray point is outside the domain. * Calculate the co-ordinate of the centre. * Recalculate distance to be for the centre point `Mesh[i1, j1, k1, :, col] = np.sqrt(np.dot((p0 - p2), (p0 - p2))) * calcvec`. * For each normal:
 - Find the next cone point $p3$ from the previous point $p1$, using the distance through a grid cube α . This is given by: $p3 = p1 + m2 * \alpha * norm$.
 - Find the co-ordinate for the centre of the grid point z corresponding to the $p3$'s. * Find the distance to this centre point. * Set the column `nra * Nre + nre` of the mesh term at these grid points to the distance times the vector of reflection angles.

```
Mesh[cpos[0][j], cpos[1][j], cpos[2][j], :, col] = r2[j] * calcvec
```

- Find the co-ordinate for the next ray point. $p1 = p1 + \alpha * direc$.

Returns Mesh, dist, calcvec

multiref(*s, room, Nre*)

Takes a ray and finds the first five reflections within a room.

Parameters

- **room** – `Room.room` object which contains all the obstacles in the room.
- **Nre** – The number of reflections. Integer value.

Using the function `reflect_calc(room)()` find the co-ordinate of the reflected ray. Store this in `s.points` and return whether the function was successful.

Return type 3x1 numpy array.

Returns end=1 if unsuccessful, 0 is successful.

normal_mat(*s, Ncones, Nra, d, dist, h*)

Form a matrix of vectors representing the plane which is normal to d

- Normalise the direction of the ray $d = d / ||d||$
- Calculate angle spacing between rays :math: \delta\theta = 2\arcsin(\frac{h}{d})

rac[1]{Ncones}

- Calculate the number of normals. :math: Nnor = 1 + \frac{Ncones}{\delta\theta}

rac{(2\pi)}{\delta\theta}

- Create an array of all the angles.

```
anglevec = np.linspace(0.0, 2 * np.pi, num=int(Nnor), endpoint=False)
```

- The dot product of the direction and the normal needs to be 0. Choose $(1, 1, -(dx + dy)/dz)$ as the first vector (unless $dz==0$). Use this to compute another vector in the plane. This forms a co-ordinate axis for the normal vectors.

If $dz==0$ then instead choose the vector $(0, 0, 1)$ and repeat the process for the additional axis vector.
.. code:

```
N[0]=(0,0,1)
y=N[0] x d
y=(1/||y||)*y
```

- Use the axis vectors as multiples of $\cos(\theta) \sin(\theta)$ to form equally space vectors.

```
N=cos(anglevec)* N[0]+sin(anglevec)*y
```

returns N

number_cone_steps (*s, h, dist, Nra*)

find the number of steps taken along one normal in the cone

number_steps (*s, meshwidth*)

The number of steps along the ray between intersection points

obst_collision_point (*s, surface*)

intersection of the ray with a wall_segment

ray_length (*s, inter*)

The length of the ray upto the intersection

raytest (*s, room, err*)

Checks the reflection function for errors

ref_angle (*s, room*)

Find the reflection angle of the most recent intersected ray.

Parameters **room** – *Room*. room object which contains all the obstacles in the room.

Use the ray number stored in *s.points[-2][-1]* to retrieve the obstacle number then retrieve that obstacle from room.

```
norm=edge1 x edge2

c = (ray_direc . norm) / (||ray_direc|| ||norm||)

theta=arccos(c)
```

Return type float

Returns theta

reflect_calc (*s, room*)

Finds the reflection of the ray inside a room.

Method:

•If: the previous collision point was None then don't find the /
next one. * Else:

room_collision_point (*s*, *room*)

The closest intersection out of the possible intersections with the `wall_segments` in `room`. Returns the intersection point and the wall intersected with

Room

Code to construct the mesh of the room

class Room.**room** (*s, obst*)

A room is where the obstacle co-ordinates are contained.

Parameters **obst** – is a $\text{Nob} \times 3 \times [3 \times 1]$ array, where Nob is the number of obstacles.

obst[j] is a $3 \times [3 \times 1]$ array which is 3, 3D co-ordinates which form a triangle.

This array of triangles forms the obstacles in the room.

Attributes of room:

- **s.obst=obst**
- **.points[3*j]=obst[j][0]**
- **s.maxlength** is a 4×1 array initialised as empty. Once assigned this is the maximum length in the room and in the x, y, and z axis.
- **s.bounds** is a 3×2 array $s.bounds = [[minx, miny, minz], [maxx, maxy, maxz]]$
- **s.inside_points** is an initial empty array. Points which are known to be inside obstacles are added to this array later.
- **s.time** is an array with the time the room was created.
- **s.meshwidth** is initialised as zero but is stored once asked for using **get_meshwidth**.

coordinate (*s, h, i, j, k*)

Find the co-ordinate of the point at the centre of the element.

Parameters

- **h** – the meshwidth. Once assigned this matches **s.meshwidth**
- **i** – the first index or an array corresponding to the first index for multiple points.
- **j** – the second index or an array corresponding to the second index for multiple points.
- **k** – the third index or an array corresponding to the third index for multiple points.

If there is only 1 **i**, 1 **j**, and 1 **k**,

$$p = [minx, miny, minz] + h * [i + 0.5, j + 0.5, k + 0.5]$$

Else if there's arrays for **i, j**, and **k**,

```
p=[ [minx,miny,minz] +h*[i0+0.5,j0+0.5,k0+0.5], ...,
    [minx,miny,minz] +h*[in+0.5,jn+0.5,kn+0.5] ]
```

Returns p

maxlength (s, a=0)

Get the maximum length in the room or axis.

Parameters **a** – the axis or room. a=0 maximum length in room, a=1 for x-axis, a=2 for y-axis a=3 for z-axis.

If the maxlength[a] hasn't been found yet find it by comparing the between points in s.points.

Returns s.maxlength[a]

position (s, p, h)

Find the indexing position in a mesh with width h for point p lying in the room s.

Parameters **p** – =[x,y,z] the co-ordinate of the point p or an array of

pointsp = [[x0,y0,z0], ..., [xn,yn,zn]] :param h: is the meshwidth, once assigned this matches s.meshwidth

If p is one point,

```
out=(p-[minx,miny,minz])/h,
```

If p is an array of points,

```
out=[ (p0-[minx,miny,minz])/h, ..., (pn-[minx,miny,minz])/h]
```

Returns out

ray_bounce (s, Tx, Nre, Nra, directions)

Trace ray's uniformly emitted from an origin around a room.

Parameters

- **Nra** – Number of rays
- **Nre** – number of reflections Nre
- **directions** – A Nra*3 array of the initial directions for each ray.

The multiref function is used to find the Nre reflections for the Nra rays with the obstacles s.obst.

raylist = [[p00,p01, ..., p0Nre], [p10, ..., p1Nre], ..., [pNra0, ..., pNraNre]]

Return type An array of the ray points.

Returns raylist

ray_mesh_bounce (s, Tx, Nre, Nra, directions, Mesh)

Traces ray's uniformly emitted from an origin around a room.

Parameters

- **Tx** – the co-ordinate of the transmitter location
- **Nra** – Number of rays
- **Nre** – number of reflections
- **directions** – Nra*3 array of the initial direction for each ray.

- **Mesh** – a $N_x \times N_y \times N_z \times n_a \times n_b$ array (actually a dictionary of sparse matrices using class DS but built to have similar structure to an array).

```
na=int(Nob*Nre+1), nb=int((Nre)*(Nra)+1)
```

The rays are reflected Nre times with the obstacles s.obst. The points of intersection with the obstacles are stored in z raylist. This is done using the mesh_multiref function. As each intersection is found the mesh_multiref function forms the line segment between intersection points and the corresponding ray cone. All mesh elements in the ray cone store the reflection angles and the distance along the ray cone from the source to the centre of each mesh element. This is stored in Mesh. See Rays.mesh_multiref() for more details on the reflections and storage.

When complete the time in s.time() is assigned to the time taken to complete the function.

Returns raylist, Mesh

Mesh

class DictionarySparseMatrix.**DS**(*s*, *Nx=1*, *Ny=1*, *Nz=1*, *na=1*, *nb=1*, *dt=<class 'numpy.complex128'>*)

The DS class is a dictionary of sparse matrices. The keys for the dictionary are (i,j,k) such that i is in [0,Nx], j is in [0, Ny], and k is in [0,Nz]. SM=DS[x,y,z] is a na*nb sparse matrix, initialised with complex128 data type. $na = (Nob * Nre + 1)$ $nb = ((Nre) * (Nra) + 1)$

asin(*s*)

Finds

$\theta = \arcsin(x)$ for all terms $x \neq 0$ in the DS *s*. Since all angles θ are in $[0, \pi/2]$, $\arcsin(x)$ is not a problem.

Returns DSM with the same dimensions as *s*, with $\arcsin(s) = \theta$ in the same positions as the corresponding theta terms.

cos(*s*)

Finds $\cos(\theta)$ for all terms $\theta \neq 0$ in the DS *s*.

Returns A DSM with the same dimensions with $\cos(\theta)$ in the same position as the corresponding theta terms.

dense(*s*)

Fills the DSM *s*.

Returns A dense $Nx \times Ny \times Nz \times na \times nb$ array with matching nonzero terms to the sparse matrix *s* and zeroes elsewhere.

dict_DSM_divideby_vec(*s*, *vec*)

Divide every column of the DSM *s* elementwise with the vector *vec*.

Parameters **vec** – a row vector with length *na*.

For integers *x*, *y*, *z*, *k* and *j* such that, $x \in [0, Nx)$, $y \in [0, Ny)$, $z \in [0, Nz)$, $k \in [0, na)$, $j \in [0, nb)$,

$$\text{out}[x, y, z, k, j] = \text{DSM}[x, y, z, k, j] / \text{vec}[k]$$

Return type a DSM 'out' with the same dimensions as *s*.

Returns out

dict_col_mult(*s*)

Multiply all nonzero terms in a column.

In every grid point x,y,z of s there is a sparse matrix SM . Take the product of all nonzero terms in each column and keep these in a vector v . Construct a new DS of size $N_x \times N_y \times N_z \times 1 \times nb=s.shape[1]$. Call this out. $out[x,y,z]$ should be the v corresponding to the SM in s at x,y,z .

Method:

- Find the `DS.nonzero()` indices of s .
- For each nonzero x,y,z grid point find the nonzero() indices of the SM . Do this by column so that the output has pairs going through each nonzero column and matching the nonzero row number. Use function `nonzero_bycol()`.
- Go through each of these indice pairs for the SM . Check if the column index is new. If so assign the column in out to the matching value in the SM . If the column number is not new then multiply the value in the column in out by the corresponding value in the SM .

```
out=[
[prod(nonzero terms in column 0 in s[0,0,0]),
prod(nonzero terms in column 1 in s[0,0,0]),
...,
prod(nonzero terms in column nb in s[0,0,0]
],
...,
[prod(nonzero terms in column 0 in s[Nx-1,Ny-1,Nz-1]),
prod(nonzero terms in column 1 in s[Nx-1,Ny-1,Nz-1]),
...,
prod(nonzero terms in column nb in s[Nx-1,Ny-1,Nz-1]
]
]
```

Return type DS of size $N_x \times N_y \times N_z \times 1 \times nb$

Returns out

dict_row_vec_multiply (s, vec)

Multiply every row of the DSM s elementwise with the vector vec .

Parameters **vec** – a row vector with length na .

For integers x, y, z, k and j such that, $x \in [0, Nx), y \in [0, Ny), z \in [0, Nz), k \in [0, na), j \in [0, nb)$,

```
out[x,y,z,k,j]=vec[j]*DSM[x,y,z,k,j]
```

Multiplication is done using `DS.dict_vec_multiply(vec)()`

Return type A DSM ‘out’ with the same dimensions as s .

Returns out

dict_scal_mult ($s, scal$)

Multiply every term of the DSM s by $scal$.

Parameters **scal** – scalar variable

For integers x, y, z, k and j such that, $x \in [0, Nx), y \in [0, Ny), z \in [0, Nz), k \in [0, na), j \in [0, nb)$,

```
out[x,y,z,k,j]=scal*DSM[x,y,z,k,j]
```

Return type DS(N_x, N_y, N_z, na, nb)

Returns out

dict_vec_divideby_DSM(*s*, *vec*)

Every column of the DSM *s* divides elementwise the vector *vec*.

Parameters *vec* – a row vector with length *na*.

For integers *x*, *y*, *z*, *k* and *j* such that, $x \in [0, Nx)$, $y \in [0, Ny)$, $z \in [0, Nz)$, $k \in [0, na)$, $j \in [0, nb)$,

```
out[x, y, z, k, j] = vec[k] / DSM[x, y, z, k, j]
```

Return type a DSM ‘out’ with the same dimensions as *s*.

Returns out

dict_vec_multiply(*s*, *vec*)

Multiply every column of the DSM *s* elementwise with the vector *vec*.

Parameters *vec* – a row vector with length *na*.

For integers *x*, *y*, *z*, *k* and *j* such that, $x \in [0, Nx)$, $y \in [0, Ny)$, $z \in [0, Nz)$, $k \in [0, na)$, $j \in [0, nb)$,

```
out[x, y, z, k, j] = vec[k] * DSM[x, y, z, k, j]
```

Multiplication is done using `DS.dict_vec_multiply(vec)()`

Return type A DSM ‘out’ with the same dimensions as *s*.

Returns out

nonzero(*s*)

Find the indices of the nonzero terms in the DSM *s*.

Note: The indices are found by iterating through all keys (x,y,z) for the DSM *s* and finding the nonzero indices of the corresponding sparse matrix. These indices are then combined with the x,y,z key and stacked to create a 5xN array of all the nonzero terms in the DSM, where N is the number of nonzero terms.

Returns indices=[[x1,y1,z1,k1,j1],...,[xn,yn,zn,kn,jn]]

nonzeroMat(*s*, *cor*)

Find the indices of the nonzero terms for part of the DSM *s*.

Parameters *cor* – the part of *s* that you want the nonzero indices for.

The indices are found by using the `nonzero()` function on *s*[*cor*]

Returns indices=[[x1,y1,z1,k1,j1],...,[xn,yn,zn,kn,jn]]

row_sum(*s*)

Sum all nonzero terms in a row.

In every grid point x,y,z of *s* there is a sparse matrix SM. Construct a new DS of size $N_x \times N_y \times N_z \times na = s.shape[0] \times 1$. Call this out. out[x,y,z] should be the corresponding *na* x 1 SM to the SM in *s* at x,y,z.

Method:

- Find the `DS.nonzero()` indices of *s*
- Go through each of these indices. Check if the row index is new. If so assign the row in out to the matching value in the SM. If the row number is not new then sum the value in the column in out by the corresponding value in the SM.

```
out=[
    [sum(nonzero terms in row 0 in s[0,0,0]),
     sum(nonzero terms in row 1 in s[0,0,0]),
     ...,
     sum(nonzero terms in row na in s[0,0,0]
    ],
    ...,
    [sum(nonzero terms in row 0 in s[Nx-1,Ny-1,Nz-1]),
     sum(nonzero terms in row 1 in s[Nx-1,Ny-1,Nz-1]),
     ...,
     sum(nonzero terms in row na in s[Nx-1,Ny-1,Nz-1]
    ]
]
```

Return type DS of size Nx x Ny x Nz x na x 1

Returns out

save_dict (*s*, *filename_*)

Save the DSM *s*.

Parameters **filename** – the name of the file to save to.

Returns nothing

sin (*s*)

Finds $\sin(\theta)$ for all terms $\theta! = 0$ in the DS *s*.

Returns A DSM with the same dimensions with $\sin(\theta)$ in the same position as the corresponding theta terms.

sparse_angles (*s*)

Finds the angles θ which are the arguments of the nonzero complex terms in the DSM *s*.

Returns A DSM with the same dimensions with θ in the same position as the corresponding complex terms.

stopcheck (*s*, *i*, *j*, *k*, *p1*, *h*)

Check if the index [*i*,*j*,*k*] is valid.

Parameters

- **i** – is the index for the x axis.
- **j** – is the index for the y axis.
- **k** – is the index for the z axis.
- **p1** – is the point at the end of the ray.
- **h** – is the mesh width

Returns 1 if valid, 0 if not.

Todo

add the inside check to this function

Todo

add the check for the end of the ray.

stopchecklist (*s, ps, p1, h, p3, n*)

Check if the list of points is valid.

Parameters

- **ps** – the indices for the points in the list
- **p1** – the end of the ray
- **h** – the meshwidth
- **p3** – the points on the cone vectors
- **n** – the normal vectors forming the cone.

start=0 if no points were valid if at least 1 point was valid, ps=[[i1,j1,k1],...,[in,jn,kn]] the indices of the valid points, p3=[[x1,y1,z1],...,[xn,yn,zn]] co-ordinates of the valid points, N=[n0,...,nN] the normal vectors corresponding to the valid points.

Returns start, ps, p3, N

togrid (*s*)

Computethe matrix norm at each grid point and return a 3d numpy array.

Return type Nx x Ny x Nz numpy array

Returns Grid

DictionarySparseMatrix.**load_dict** (*filename_*)

Load a DS as a dictionary and construct the DS again.

Parameters **filename** – the name of the DS saved

```
Nx=max(Keys[0])-min(Keys[0])
Ny=max(Keys[1])-min(Keys[1])
Nz=max(Keys[2])-min(Keys[2])
```

Returns nothing

DictionarySparseMatrix.**nonzero_bycol** (*SM*)

Find the index pairs for the nonzero terms in a sparse matrix. Go through each column and find the nonzero rows.

Parameters **SM** – sparse matrix.

Returns [[i(0j0),i(1j0),...,i(nj0),...,i(njn)], [j0,...,j0,...,jn,...,jn]]

DictionarySparseMatrix.**parnonzero** (*nj, DS*)

Parallel version of a program with a dummy DS and a function for finding the indices of the nonzero terms in a mesh.

Parameters

- **nj** – number of processes.
- **DS** – the mesh

Pool the nj processes Specify what needs to be done. Combine the information.

Returns 5xn array which n is the number of nonzero terms.

DictionarySparseMatrix.**phase_calc** (*RadMesh, khat, L*)

Compute $\exp(ik\hat{r}L^2)$ for a Mesh of r

The phase is usually expressed at $\exp(ikr)$. Since \hat{k} and \hat{r} are nondimensional lengths scaled by the room length L the power of L^{-2} must be used.

Exponentials are not defined on DS, instead use $\exp(iheta) = \cos(heta) + i \sin(heta)$.

```
S1=RadMesh.dict_scal_mult(khat)
S2=S1.dict_scal_mult(1.0/(L**2))
out=S1.cos()+S1.sin().dict_scal_mult(1j)
```

rtype DS of size $N_x \times N_y \times N_z \times n_a \times 1$

return out

DictionarySparseMatrix.**power_compute** (*Mesh, Grid, Znobrat, refindex, Antpar, Gt*)

Compute the field from a Mesh of ray information and the physical parameters.

Parameters

- **Mesh** – The *DS* mesh of ray information.
- **Znobrat** – An $N_{ob} \times N_{re}+1$ array containing tiles of the impedance of obstacles divided by the impedance of air.
- **refindex** – An $N_{ob} \times N_{re}+1$ array containing tiles of the refractive index of obstacles.

Method:

- First compute the reflection coefficients using `ref_coef(Mesh, Znobrat, refindex)()`
- Combine the reflection coefficients that correspond to the same ray using `DS.dict_col_mult()`. This multiplies reflection coefficients in the same column.
- Extract the distance each ray had travelled using `DS.__get_rad__()`
- Multiply by the gains for the corresponding ray.
- Multiply terms by the phases $\exp(i\hat{k}\hat{r})L^{-2}$. With L being the room length scale. \hat{r} being the relative distance travelled which is the actual distance divided by the room length scale, and \hat{k} is the relative wavenumber which is the actual wavenumber times the room length scale.
- Multiply by the gains corresponding to each ray.
- Divide by the distance corresponding to each ray segment.
- Sum all the ray segments in a grid point.
- Multiply the grid by the transmitted field times the wavelength divided by the room length scale. $\frac{\lambda}{L4\pi}$
- Multiply by initial polarisation vectors and combine.
- Ignore dividing by initial phi as when converting to power in db this disappears.
- Take the amplitude and square.
- Take $10\log_{10}()$ to get the db Power.

Return type $N_x \times N_y \times N_z$ numpy array of real values.

Returns Grid

DictionarySparseMatrix.**ref_coef** (*Mesh, Znobrat, refindex*)

Find the reflection coefficients.

Parameters Mesh – The DS mesh which contains terms $re^{i\theta}$ with θ the reflection angle of incidence.

Method:

- Gets the mesh of angles using `DS.sparse_angles()`
- Gets the indices of the nonzero terms using `DS.nonzero()`
- Initialise $\sin(\theta)$, $\cos(\theta)$ and $\cos(\theta)$ meshes.
- Compute $\cos(\theta)$, $\sin(\theta)$, $\cos(\theta)$

```
c thi=AngDSM.cos()
SIN=AngDSM.sin()
Div=SIN.dict_DSM_divideby_vec(refindex)
ctht=Div.asin().cos()
```

- Compute the reflection coefficients.

```
S1=(c thi).dict_vec_multiply(Znobrat)
S2=(ctht).dict_vec_multiply(Znobrat)
Rper=(S1-ctht)/(S1+ctht)
Rpar=(c thi-S2)/(c thi+S2)
```

Return type `Rper=DS(Nx,Ny,Nz,na,nb),Rpar=DS(Nx,Ny,Nz,na,nb)`

Returns `Rper, Rpar`

`DictionarySparseMatrix.test_14()`

This is a test of the reflection coefficient function. It sets test versions for the input parameters required and fills a DS with dummy values. It then computes the reflection coefficients associated with those dummy parameters and values.

`DictionarySparseMatrix.test_15()`

Testing multiplying nonzero terms in columns

`DictionarySparseMatrix.test_17()`

Test the `parnonzero()` function which should find `nonzero()` indices in parallel.

`DictionarySparseMatrix.test_18()`

Testing the save and load pickle functions.

`DictionarySparseMatrix.test_19()`

Test the `nonzero_bycol(SM)()` function. Initialise a dummy sparse matrix SM.

In `nonzero_bycol(SM)()`:

- Transpose the matrix.
- Find the nonzero indices.
- Swap the rows and columns in the indices.
- Return the indices

Check these match the nonzero terms in SM.

Returns 0 if successful 1 if not.

`DictionarySparseMatrix.test_20()`

Test the `dict_col_mult()` function. Use a dummy DS with each matrix upper triangular with the number in every position the row. Check that the `col_mult` that comes out is the column number +1 factorial.

`DictionarySparseMatrix.test_21()`
Test if the `__get_rad__()` function works.

Reflection

Code to Reflect a line in an edge without using Shapely

`reflection.errorcheck(err, ray, ref, normedge)`

Take the input ray and output ray and the normal to the edge, check that both vectors have the same angle to the normal

`reflection.refangle(line, obst)`

Find the reflection angle for the line reflection on the surface obst

`reflection.test3()`

angle test

Indices and tables

- `genindex`
- `modindex`
- `search`

d

`DictionarySparseMatrix`, [17](#)

p

`ParameterInput`, [7](#)

r

`Rays`, [9](#)

`RayTracerMainProgram`, [3](#)

`reflection`, [25](#)

`Room`, [13](#)

A

asin() (DictionarySparseMatrix.DS method), 17

C

coordinate() (Room.room method), 13

cos() (DictionarySparseMatrix.DS method), 17

D

DeclareParameters() (in module ParameterInput), 7

dense() (DictionarySparseMatrix.DS method), 17

dict_col_mult() (DictionarySparseMatrix.DS method), 17

dict_DSM_divideby_vec() (DictionarySparseMatrix.DS method), 17

dict_row_vec_multiply() (DictionarySparseMatrix.DS method), 18

dict_scal_mult() (DictionarySparseMatrix.DS method), 18

dict_vec_divideby_DSM() (DictionarySparseMatrix.DS method), 18

dict_vec_multiply() (DictionarySparseMatrix.DS method), 19

DictionarySparseMatrix (module), 17

DS (class in DictionarySparseMatrix), 17

E

errorcheck() (in module reflection), 25

L

load_dict() (in module DictionarySparseMatrix), 21

M

maxleng() (Room.room method), 14

mesh_multiref() (Rays.Ray method), 9

mesh_singleray() (Rays.Ray method), 9

MeshProgram() (in module RayTracerMainProgram), 3

multiref() (Rays.Ray method), 10

N

nonzero() (DictionarySparseMatrix.DS method), 19

nonzero_bycol() (in module DictionarySparseMatrix), 21

nonzeroMat() (DictionarySparseMatrix.DS method), 19

normal_mat() (Rays.Ray method), 10

number_cone_steps() (Rays.Ray method), 11

number_steps() (Rays.Ray method), 11

O

obst_collision_point() (Rays.Ray method), 11

ObstacleCoefficients() (in module ParameterInput), 7

P

ParameterInput (module), 7

parnonzero() (in module DictionarySparseMatrix), 21

phase_calc() (in module DictionarySparseMatrix), 21

position() (Room.room method), 14

power_compute() (in module DictionarySparseMatrix), 22

power_grid() (in module RayTracerMainProgram), 5

R

Ray (class in Rays), 9

ray_bounce() (Room.room method), 14

ray_length() (Rays.Ray method), 11

ray_mesh_bounce() (Room.room method), 14

Rays (module), 9

raytest() (Rays.Ray method), 11

RayTracer() (in module RayTracerMainProgram), 4

RayTracerMainProgram (module), 3

ref_angle() (Rays.Ray method), 11

ref_coef() (in module DictionarySparseMatrix), 22

refangle() (in module reflection), 25

RefCoefComputation() (in module RayTracerMainProgram), 4

RefCombine() (in module RayTracerMainProgram), 4

reflect_calc() (Rays.Ray method), 11

reflection (module), 25

room (class in Room), 13

Room (module), 13

room_collision_point() (Rays.Ray method), 11

row_sum() (DictionarySparseMatrix.DS method), 19

S

`save_dict()` (DictionarySparseMatrix.DS method), [20](#)
`sin()` (DictionarySparseMatrix.DS method), [20](#)
`sparse_angles()` (DictionarySparseMatrix.DS method), [20](#)
`stopcheck()` (DictionarySparseMatrix.DS method), [20](#)
`stopchecklist()` (DictionarySparseMatrix.DS method), [21](#)

T

`test3()` (in module reflection), [25](#)
`test_14()` (in module DictionarySparseMatrix), [23](#)
`test_15()` (in module DictionarySparseMatrix), [23](#)
`test_17()` (in module DictionarySparseMatrix), [23](#)
`test_18()` (in module DictionarySparseMatrix), [23](#)
`test_19()` (in module DictionarySparseMatrix), [23](#)
`test_20()` (in module DictionarySparseMatrix), [23](#)
`test_21()` (in module DictionarySparseMatrix), [24](#)
`togrid()` (DictionarySparseMatrix.DS method), [21](#)