Schema for Sonar Image Generation

Chris Moorhead

March 27, 2023

Contents

1 Introduction

The purpose of this document is to explain the structure of the sonar imaging simulation process for the project and provide documentation for the adjustable parameters.

A bare-bones description of the process is that the simulator produces a bundle of rays that are emitted from a single point in space (the transmitter) and sent out into a 3D scene containing objects usually defined by an .obj file. Each ray has its own strength determined by its direction relative to the direction of the peak signal of the transmitter. As it passes through the environment it loses energy (acoustic attenuation) and may or may not encounter an object. If an object is encountered, the echo back in the direction of the transmitter (also the location of the receiver) is calculated. The distance travelled is proportional to the time taken and an aggregate of the received echoes is collected up to a predefined time. The number of discrete time periods into which the returning rays are sorted is the same as the resolution of the resulting image in the radial directon.

This process is repeated for each pulse as the sonar array rotates to scan the environment, each pulse corresponding to one line of the resulting image. The image then undergoes a post-processing stage in order to improve contrast. The image resolution in the axial direction is equivalent to the number of time steps needed in the simulation to complete one sweep of the scene, being determined by the scan speed.

2 Distinguishing Features of Sonar Imaging

There are several features of sonar imaging modelled by ray tracing which make it different to that of the more conventional optical ray tracing process:

- Where optical ray tracing gathers a whole scene at once by receiving information from each visible point, sonar imaging is built up one line at a time by sending out narrow pulses of acoustic energy and gathering information about the echoes.
- For scanning sonar used in this project, the data is gathered using a polar coordinate system, i.e. the resulting image shows distance changing with angle, rather than the usual Cartesian coordinate system. A transformation can be applied to change coordinate system, but this would be for the benefit of human interpretation and results in information loss.
- Sonar imaging creates monochrome images using the intensity of gathered signals.[1] In optical imaging, this is expressed on a scale of 0 to 255 (2 orders of magnitude). In contrast, the received signals for sonar imaging can be over a much larger range (4+ orders of magnitude). The incoming signals are measured using a log scale, specifically the deciBel system which includes comparison with a base reference signal. The resulting image, therefore, can be highly sensitive to whatever scaling is used to convert from the actual signal to one that can be fit into the scale 0-255. This conversion is done using a TVG table defined by the manufacturer. Note also that these additional levels of computation can attribute to higher levels of noise due to rounding.
- Acoustic signals dissipate according to both an inverse square law and an additional absorption factor. In other words, a signal at 10m will be at 100 times weaker at 100m, even neglecting the additional absorption. We can easily see how this contributes to the above range, but the additional absorption factor must be modelled differently than the optical domain, where it is known as volume absorption. It is dependent on both environmental factors and the frequency used and carries a much heavier influence than in the optical domain.

[1]Often sonar imaging software converts to a colour palette for ease of interpretation by a human.

3 Flow

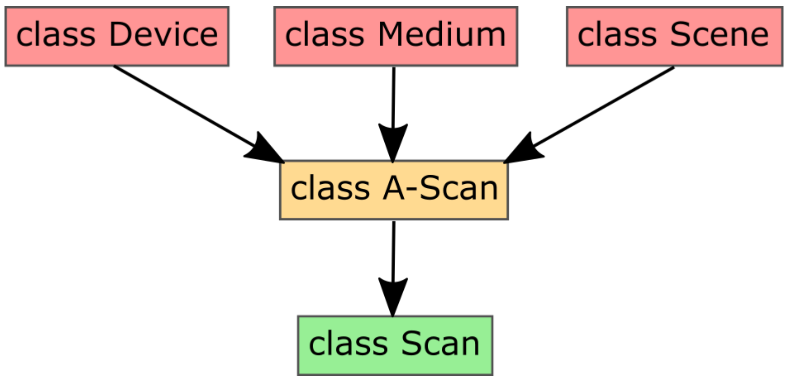


Figure 1: Flow Showing class Dependencies

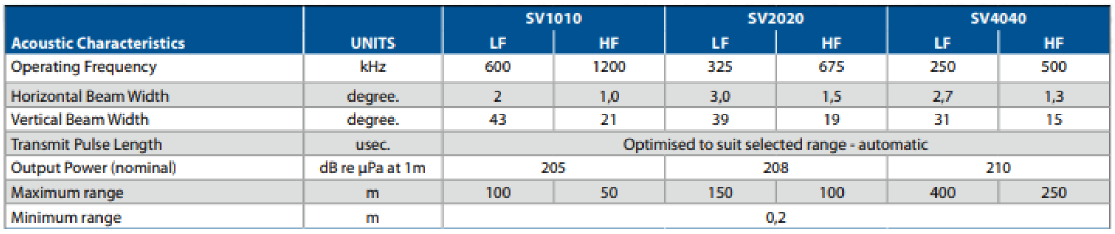
Figure 1 shows the main classes and their dependencies. These are described more fully in the subsections, but as an overview:

- **Device**: Controls the power and shape of each sonar pulse using the sonar physics formulae.
- **Medium**: The conditions of the marine environment control the attenuation of the acoustic signal over distance. Variables include salinity, temperature, depth and pH.
- **Scene**: Representation of the solid objects which the acoustic signal can potentially interact with.
- **A_scan**: The basic class used to build one line of a sonar scan. It sets the sonar device within the scene and determines the image realism in terms of number of rays emitted into the scene.
- **Scan**: Uses A_scan to build up a complete image one line at a time for a variety of scenarios.

3.1 Sonar Device

The sonar device we are attempting to model is the Sonavision SV1010. At 600 kHz it has a specified beam width of 43 degrees in the vertical direction and 2 degrees in the horizontal direction. At 1000 kHz, it has a specified beam width of 21 degrees in the vertical direction and 1 degree in the horizontal direction. See Figure 2. for the technical specifications.

Note that beam width is defined as the angular width of the primary lobe of the acoustic pulse that is above 50% of the nominal output power. The frequency can be adjusted, but for test purposes it will be fixed at 1200 kHz.



		SV1010		SV2020		SV4040	
Acoustic Characteristics	UNITS	LF	HF	LF	HF	LF	HF
Operating Frequency	kHz	600	1200	325	675	250	500
Horizontal Beam Width	degree.	2	1,0	3,0	1,5	2,7	1,3
Vertical Beam Width	degree.	43	21	39	19	31	15
Transmit Pulse Length	usec.	Optimised to suit selected range - automatic					
Output Power (nominal)	dB re µPa at 1m	205		208		210	
Maximum range	m	100	50	150	100	400	250
Minimum range	m	0,2					

Figure 2: Specification from the Sonavision Catalogue

3.2 Scene

The Scene class is perhaps of the most important in the simulation process as it contains all of the objects in the environment. The structure of a Scene can be seen in Figure 3. The Scene class is instantiated using the keywords "background", which must be a single

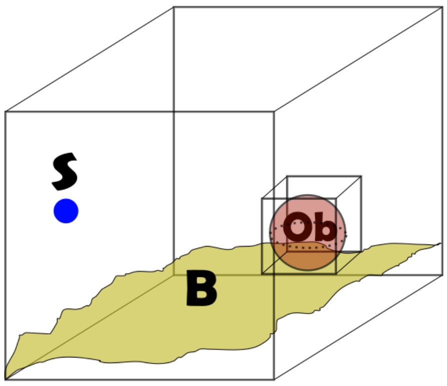


Figure 3: An instance of Scene contains a background and/or a selection of objects. Usually these are instances of Mesh.

instance of an Object subclass (usually a Mesh), and "objects" which must be passed a subclass of Object or a list/dictionary of these. Neither is a mandatory keyword, but if both are empty, then the Scene will be empty and will result in an empty image.

The optional "accelerator" keyword will be a method of parsing the scene and/or distributing the computation load and modifying the Scene class methods. An instance of Scene can also be ascribed a name using the "name" keyword, useful when using a pattern for scene descriptions eg "tyre-90-deg-r1-1". Two additional properties exist: rays,, and object_count, an integer storing the number of non-background objects of interest in the scene.

Scene has the following properties:

- **.background**: The assigned background object or None.
- **.objects**: The objects in the scene. This will be a short list.
- **.labels**: The names of the objects in the scenes, either provided by the user, or extracted from the object's properties.
- **.name**: None or an assigned string.
- **.accelerator**: An Accelerator instance.
- **.rays**: Used for temporary storage of ray arrays passed by Device during interaction via the A_scan class.
- **.pov**: Used to store the point in space representing the transmitter/receiver of the sonar device.
- **.object_count**: An integer storing the number of non-background objects of interest in the scene.

The methods in this class are as follows:

- **intersection_params**(rays, pov): This accesses the intersection_params method of the scene objects given an array of rays with a common origin in 3D space, pov. The inputs are fixed as properties of the Scene object after this is called for easy reuse.
- **collect_min(indices, mins, new_array, current_index)**: This is a means of updating intersection information as the scene is processed by comparing with the closest intersection seen so far.
- **process_background**(): Called during scene processing after all objects of interest have been tested.
- **scatter_loss**(): This calls upon intersection information and uses the gathered index information to calculate the signal loss per ray depending on incident angle with object and the material it interacts with.

3.2.1 Scene processing

The default algorithm for processing the scene is simply to iterate through the objects one by one and keep a record of the closest point of intersection. The steps are as follows:

1. Inputs are an array of rays of shape (w,h,3) and pov.
2. If the Accelerator calls of any preprocessing, this is applied here.
3. Create a 2D index array, obj_idxs, of shape (w,h) filled with value -1, which represents every ray missing all objects in the scene and the background object.
4. Create a 2D array, dists, of shape (w,h) of distances until first intersection filled with a high value outside of the range.
5. Iterate through the objects, first calculating the distance of intersections of rays with the current object to output an array of shaoe (w, h). Where the rays do not interact, the corresponding position will contain NaN.
6. These results are compared with the result so far using collect_min, which updates both dists and obj_idxs with the closest distance and relevant object index.
7. Create an empty dictionary, idx_dict, which will store the indices of locations in the input, rays, which intersect with a given object index first.
8. Iterate through the object indices to populate idx_dict.
9. If there are rays that don't hit any object, then these are tested against the background object. If they don't hit the background object either, we ignore those. The rays which hit the background are assigned index -1.
10. The output is a dictionary with object indices as keys and an array of vectors which interact with that object as values. These indices are needed for scatter_loss, which generates incident angles and absorption levels for each object using only those rays which pertain to that object. This process also ignores all rays that do not produce any echoes

3.2.2 Objects

The scene objects have their own classes and class hierarchies. The abstract base class is Object from which two groupings of subclasses are derived. The first is a group of primitive objects which have their own custom intersection and other methods. The second is a Composite class which is broken down into primitive components of the first group. The second is of most importance to this project as it is complicated objects consisting of many parts which we want to parse during the ray tracing process. The details of each classes are not so important except where they concern the ray tracing elements. These include, but are not restricted to:

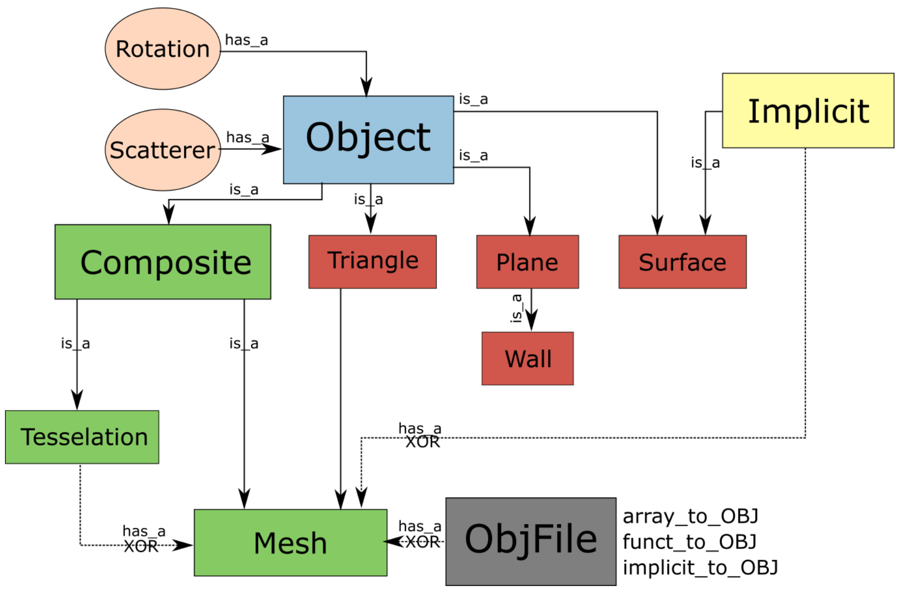


Figure 4: Some Class relationships

- **anchor**: A fixed point on the object that defines its location in space.
- **Rotation**: Its rotation expressed in Tait-Bryan notation. The object has a default position which can be adjusted and reset as needed to experiment with different orientations.
- **bounding_box**: AABB (Axis-Aligned Bounding Box) that contains the object. Where this exists, it may be used to filter incoming rays.
- **Scatterer**: Which defines the absorption properties of the material. It is used to calculated the energy dissipation of intersecting rays, a quantity which is dependent on angle of approach.
- **PARAMS**: A dictionary containing the type-specific parameters for that object.
- **intersection_params**: Called during the modelling of a single burst of acoustic energy and calculates the distances to intersection, if any, of an array of rays.
- **apply_rotation**: Allows us to rotate the object from its default orientation.

Composite objects have an extra level of depth in that they have an additional property COMPONENTS which is a dictionary of named components which are themselves objects. When applying intersection_params to this object, the incoming rays must be allocated to the individual smaller parts which have different normal properties.

Figure 4 shows examples of some objects created in the model along with some dependencies. Short descriptions are as follows:

- **Object**: The abstract base class that contains methods and properties to be inherited. It is basically a point in space with no area or volume.
- **Composite**: As noted, it is a piecewise collection of objects (normally faces) that come together to make a recognisable shape.
- **Plane/Wall**: A surface defined by $z + ax + by + c = 0$. Wall is the degenerate case where the plane is vertical.
- **Triangle**: A portion of a plane bound by three lines contained in it. This applies a different, more efficient algorithm to calculate intersections.
- **Implicit**: A function, $\mathbb{R}^3 \mapsto \mathbb{R}$ that defines a surface where it maps to zero. eg a sphere would be $f(x, y, z) : z^2 + y^2 + x^2 - r^2$ with $r \in \mathbb{R}_+$.
- **Surface**: Requires an instance of Implicit. It uses minimisation functions and derivatives to determine the points of intersections and the surface normals at those points. This is not an optimised process, however, and requires a known formula for the surface.
- **Tesselation**: Takes a function, $f(x, y) : \mathbb{R}^2 \mapsto \mathbb{R}$, which defines a height map and samples it to create a Composite of triangular faces that approximate the continuous surface of the formula.
- **ObjFile**: Allows us to interact with shapes stored in standard .OBJ format. It also allows for saving and loading of previously created objects, either within this framework or .OBJ files available from other resources eg. Human mesh examples. Three helper functions exist to convert data or another object to this type in order to save them to disc:
  - **array_to_OBJ**: Takes a 3D array of shape (m, n, 3) where m and n are the number of divisions in the x and y directions. The first two layers represent the x and y coordinates and the third is the output of a height map function, $f(x, y)$. The array itself is provided by the function funct_to_array.
  - **funct_to_OBJ**: Takes a height map function $f(x, y)$ with additional arguments that control the sampling grid. Internally, this creates the array used as an input above.
  - **implicit_to_ OBJ**: Samples an instance of Implicit and extracts a polygonal surface from it using the marching cubes algorithm. These surfaces should be representative of the surface if the correct resolution is chosen, although the resulting surface may not be optimal in terms of polygon count.
- **Mesh**: The main object used in this project, it can be instantiated by providing an instance of Tesselation, Surface or ObjFile. The result is a Composite object comprising of interlocking triangular faces.

3.3 Medium

The medium defines the part of the environment regarded as empty space in the simulation. The strength of the acoustic signal dissipates with distance which will affect the strength of the collected signal for each pulse. While the physical properties of the medium (salinity, temperature, depth, pH) can be changed, they will remain constant for testing purposes.

3.4 A-Scan

The A-Scan is the most important part of the simulation in terms of the accelerator. It is instantiated as follows (note optional arguments and keyword arguments).

```
A-Scan(device, centre, direction, declination=0, res=200,
       threshold=0.3, *args, **kwargs)
```

The parameters are as follows:

- **device**: An instance of Device class. Note that this must have the frequency set before it can be used.
- **centre**: The (x, y, z) co-ordinates of the sonar transmitter/receiver.
- **direction**: The angle in radians of the beam centre in the x-y plane measured in a clockwise direction from the y-axis.[2]
- **declination**: The angle in radians the beam centre makes with the x-y plane. By default this is zero, i.e. the scanner is looking straight ahead. A positive value will be looking down.[2]
- **res**: The number of rays generated per degree of pulse. Be default this is 200.
- **threshold**: The max strength of rays we will consider in the main pulse. Be default this is 0.3. Note that the standard beam width would have a threshold of 0.5, so this will be a little wider than the spec in both directions. Lowering to 0.1 makes this a little wider, but by experimentation it was found that less than 0.1 did not affect the beam widths. This will form a lower bound for this argument.
- **radial_res**=RADIAL_RES: The number of buckets the ray responses will be collected into. This corresponds to the number of pixels per line in the image. By default it is 400.
- **test_plane**=TEST_PLANE: This optional keyword argument allows for the scanning procedure to be tested on a simple plane (an instance of the Plane) that must be defined by the user.
- **scene**=SCENE: If a test plane is not given, the model must be supplied with an instance of Scene.

[2]Use the extra argument "degs" to supply this parameter in degrees instead.

- **medium**=MEDIUM: The user can supply a Medium instance to the scan. If not given, an instance modelling fresh water will be used instead.
- **sos**=SPEED_OF_SOUND: This is the speed of sound in the medium will be 1450 m/s by default. Alternatively, it can be calculated using the same parameters used to create the Medium instance, although differences should be negligible.
- **range**=RANGE_IN_METERS: This is the range of the device in metres and defines the limit of the scene. This is combined with the speed of sound to give the upper limit in the receiver time. This is given a default of 20m.
- **rx_step**=RECEIVER_TIME_INTERVAL: Defines the range in terms of the maximum listening time at the receiver.
- **scan_speed**=SPEED: Must be one of "slow", "normal", "fast", "superfast", "stop" or "custom". This will affect both the radial and axial resolutions to match those of the SV1010 device under these settings. The exception is "custom" for which an extra keyword argument "angle_resolution" must be supplied. Note that this is not an available option in the actual sonar device.

The A-Scan object then generates a number of key properties used in the generation of the pulse shape. They are listed below:

- **.strength_threshold**: Stores the threshold variable described above.
- **.horiz_span**: Horizontal beam width in radians according to above threshold.
- **.vert_span**: Vertical beam width in radians according to above threshold.
- **.beam_widths**: The beam widths in degrees of the device according to specification (i.e. when threshold is 0.5).
- **.min_vert, .max_vert, .min_horiz, .max_horiz**: Maximum and minumum angles of rays in horizontal and vertical directions in radians.
- **.direction_angle**: Same as the direction variable. Value is in radians.
- **.direction**: Unit vector in $\mathbb{R}^3$ of the maximum ray strength in centre of primary lobe.
- **.res, .centre**: Same as the parameters defined when creating the instance.
- **.epsilon**: Offset to prevent division by zero. Default is 0.001.
- **.theta_divs, .phi_divs**: The number of rays in the horizontal and vertical directions. This is the same as the resolution multiplied by the horizontal and vertical span in degrees.
- **.directivity**: Directivity function that outputs an array of relative strengths in the emitted pulse using an array of rays as an input.

- **.range**, **.receiver_time**: Range in metres of the simulation and corresponding receiver time in seconds over which we listen for returning signals. Related by the formula .range = .sos * .receiver_time.
- **.angles**: Array of shape (2, .phi_divs, .theta_divs) where index 0 contains values of theta for the 2D distribution of rays and index 1 the values of phi. Further indexing gives the row and then the column of the array of rays.
- **.scan_speed**: Set by the "scan_speed" keyword or "normal" by default.
- **.step_rotation**: A Rotation object that is used to increment timesteps.[3]
- **.max_intersection, .min_intersection**: The maximum and minimum distances detectable by the simulation. This truncates some responses and parallels limitations of the real device. Max intersection may differ from range due to rounding.

The following methods are essential in the operation of this class:

- **angles_to_vec(angles)**: Returns an array of normalised vectors of shape (3, .phi_divs, .theta_divs).
- **perpenticularity()**: Quantity used in calculating intersections and angles of intersection. Equivalent to the dot product of the ray direction vector and plane normal. Used only in the test plane case and stored in .ray_plane_product.
- **angles_with_plane()**: Returns a 2D array of shape (.phi_divs, .theta_divs) containing angles of intersection of the array of ray vectors with the test plane.
- **intersection_params()**: Standard method present in all object classes. This calls this method and collates the responses from objects in the scene. The output will be a 2D array of shape (.phi_divs, .theta_divs) containing distances in metres until intersection and is stored for reuse as a property of the scene class. Note that entries corresponding to rays having no intersections, or intersections outside of the range .min_intersection to .max_intersection, are assigned NaN.
- **directivity_filter(*args,**kwargs)**: Returns the loss of signal due to the directivity of the pulse. This is equivalent to assigning a strength loss in terms of dB to each ray in the 3d array.
- **apply_attenuation(*args, **kwargs)**: Calls intersection_params Uses the transmission loss method in the Medium class to return signal loss per ray in dB. The output will be a 2D array of shape (.phi_divs, .theta_divs). The optional argument "show" calls an external function visualise_2D_array. See XXX.
- **total_strength_field(*args, **kwargs)**: Calculates the total strength of each ray in dB, returning an two arrays. The first the array of distances to intersection points, where applicable, and the second an array of the strength of the returning signal along the same path as the emitted signal, being a combination of the

[3]See scipy.spatial.transform documentation.

directivity, transmission loss and absorption by the intersected surface. Both are arrays of shape (.phi_divs, .theta_divs).

- **gather(dist_array, strength_array, *args, **kwargs)**: Gathers the returns of all rays[4] and aggregates them into discrete time intervals. It flattens the 2D arrays, then filters out all echo responses recorded as NaN. The algorithm then creates an array of the bin indexes for the valid responses and aggregates this with weights equivalent to the corresponding strength of each ray response. The number of time intervals is fixed as radial_resolution. The optional "scaled" argument will shift the bin indices such that the first bin will be assigned to the minimum detected object.
- **scan_line(*args, **kwargs)**: This will produce one line of the sonar image and can be used to test the smoothness of the aggregate echo response. Using the optional argument "show_echoes" will show a histogram of the bins with aggregate strengths. Autogain (see below) will be applied unless the "no_gain" is used. The "visualise" argument can be applied to view a square image that shows the response. This image can be saved to a chosen directory using the "save_dir" keyword argument and assigned a custom name ("untitled.png" if not) using the "image_name" keyword argument. It returns the same array as *gathered* above.
- **auto_gain(arr, **kwargs)**: The internal workings of the SV1010 is such that intensity is given on a scale of 7 bits (0-127). A fixed mapping from intensity of returns at each time period to this scale does not make sense as the range of intensities can differ greatly. We therefore imitate the autogain on the device which adaptively changes this mapping. The code converts non-zero values in the array to a log scale and then . To apply a fixed gain the keyword "gain" can be applied which will multiply all values by a fixed value, saturating the image where it increases the maximum threshold.
- **advance_timestep(**kwargs)**: Advances the timestep one iteration by rotating the device direction and corresponding array of rays.

3.5 Scan

The Scan class uses A_scan as a component and controls recording behaviour exhibited by the SV1010 device. It is instantiated with an instance of A_scan and a mode that corresponds to a recording setting on the SV1010. Mode must be a string and must be one of the following:

- **scan**: Corresponding to scanning a sector of the environment. The "span" keyword must be supplied with a value in radians of the size of the sector. Using the additional "degs" argument allows this to be in degrees instead.

[4]Note that the strength array input uses the real strength values in Pa, not on a dB scale, as we cannot aggregate the log values involved with the dB scale using a summation to obtain meaningful results.

- **rotate**: Corresponding to a single 360 degree sweep of the environment.
- **stop**: Corresponds to no rotation of the device - this happens when the sonar device itself is being dragged and its centre changes instead of its direction. This is used in so-called Side Scan Sonar imaging. This requires an extra duration keyword argument in seconds.

It has the following properties:

- **.a_scan**: The instance of A_Scan used to create the Scan object.
- **.device, .scan_speed, .angle_resolution, .angle_step, .radial_rsolution**: Inherited from the same properties in A_Scan.
- **.start**: The same as the direction property in the A_Scan which sets the start point for the beam.
- **.steps**: The total number of steps in the scan.
- **.current**: The angle at the current position in the scan in radians.
- **.scan_idx**: Current scan step.

And the following methods:

- **get_line**: Obtains one line from A_Scan at current position.
- **full_scan**: Compiles all lines according to the scan parameters. The optional verbosity keyword can be assigned different integer values indicating a higher level of visibility in how the program will run.

4 Project Utilities

The classes and some default instances are included in the project_utils.py file.

- Choice of device under high and low frequency: SV1010_1200Hz, SV1010_600Hz
- Salt water Medium under high and low frequency: SALT_WATER_1200, SALT_WATER_600

A Explicit Function Extraction Tools

To create .OBJ files of a function that describes a height map at different levels of sampling, three functions were made.

- **scale_sample**(funct, x_range, y_range, levels, low=1): Given an explicit function, tuples/lists representing the range in x and y axes and the number of levels of complexity to generate, this function will produce a list of arrays representing a grid of shape $(2^i + 1, 2^i + 1)$ where $low \geq i \geq low + levels$. By default, this will start at $low = 1$ resulting in a grid of 9 sampled points at the lowest level. Each subsequent level increases the number of sampled points by a factor of 4.
- **nested_tesselations**(funct, x_range, y_range, levels, low=1,[name=NAME]): Takes the above function and instead produces a dictionary of tesselations. Using the optional keyword argument, it's possible to name each Tesselation object which is then appended by the number of subdivisions of the original area.
- **nested_sample_to_OBJ**(funct, x_range, y_range, levels, low=1,[name=NAME],[save_dir=DIR]): Takes the output of the above function and saves the objects in .OBJ format. The optional keyword argument will save to a chosen directory, otherwise the objects are saved in the current working directory. Recall that these objects have all triangular faces. The number of faces at complexity level i is $2 * *i + 2$ and this will be added as a descriptor in the filename.

The steps to apply are simple:

1. Write a function that takes in two variables, x and y, and outputs a single value. There may be other parameters you want to tinker with in order to get the desired property of the surface.
2. Call nested_sample_to_OBJ with the relevant arguments.
3. Use software to examine the output to get something at a suitable scale and complexity for a scene or background object. Recall the Nyquist Sampling Theorem and consider that it may affect representation of some functions. Recursively design prospective surfaces.