Cone-Beam Geometry X-ray Simulator for STL Files in MATLAB

This package performs simulated, 2 dimensional (2D) x-ray projections on 3 dimensional objects (3D), described in stereolithography format (STL) utilizing MATLAB. Cone-beam geometry was chosen as the x-ray model due to the utility advantages it has over parallel and fan beam x-ray geometry (simplest real world scenario, does not require collimater in order to parallelize beams, less equipment etc.).

1. Getting Started

Add the folder X-ray Simulation to your MATLAB path. Inside the folder you will find the main function, *XraySim.m*, along with several other functions that are necessary to run the simulations.

Make sure the STL file you are planning on simulation x-ray projections from is in the same folder as the main function *XraySim.m*, along with the functions: *READ_stl.m*, *VOXELISE.m*, and *projection2D.m*.

Prerequisites

Make sure the STL file you are planning on simulation x-ray projections from is in the same folder as the main function *XraySim.m*, along with the functions: *READ_stl.m*, *VOXELISE.m*, and *projection2D.m*.

Simulation Overview

This 2D x-ray projection process from this package can be summarized by the following steps:

1. Place the STL file containing the 3D mesh object that you want to obtain simulated x-rays from into the **"X-ray Simulation"** folder.
2. Use the matlab command, *Help XraySim* to determine the desired function inputs for the wrapper function *XraySim.m*. The simplest functionality requires just the *STLFILE.stl* as input.
3. Run the *XraySim.m* function.

4. *XraySim.m* then uses the functions *READ_stl.m* and *VOXELISE.m* to read in the 3D mesh data of the object, and uses three dimensional raycasting to voxelize it.
5. *XraySim.m* sets various simulation parameters based on the input at step 3.
6. The 3D object array is then fed into the *projection2D.m* function, which computes the slice by slice 2D Conical X-ray projection in the Z-direction.
7. Lastly, the projection is then saved as a Bitmap file and saved to the current folder.

2. Package Functions

This section gives a brief overview of the package functions and their functionality. Use the MATLAB help command to receive a more in depth description on the input parameter options, if needed.

XraySim.m

This is the wrapper function for the package and at minimum requires one input, the STL file. Other optional input options include the grid size for the voxelization (integer or 1xN array), a string input (to name the output BMP file), and several input argument pairs. This wrapper function calls *READ_stl.m* to read in the 3D STL file, and uses *VOXELISE.m* in order to turn the mesh into a 3D image array (voxelization). After various imaging simulation parameters are set, it then feeds the 3D image array into *projection2D.m* which ouputs a 2D x-ray image that is the cone beam projection of the 3D STL mesh object in the Z-direction. Finally it saves that image to the current directory the code is operating from.

READ_stl.m

This function and *VOXELISE.m* are taken from the voxelisation toolbox by Adam A[1]. This function specifically reads the mesh data from an STL file into MATLAB that can be both binary and ascii formats. It uses textscan to read ascii files all at once, rather than one line at a time, which makes the overall speed considerably faster. Lastly, it uses fread for binary files also making its computation speed fast. An example of STL ASCII file structure is below:

```
Solid object name
Facet normal x y z
        Outer loop
                Vertex x y z
```

```
                Vertex x y z
                Vertex x y z
        End Loop
End facet
(Repeat for all facets in file)
```

VOXELISE.m

This function utilizes *READ_stl.m*. to read in the 3D triangular-polygon mesh data into MATLAB in order to voxelize it into a 3D image array. This code utilizes a ray intersection method similar to the method described by Patil S., and Ravi B.[2].

There are multiple options for raycasting directions, but for reliability and accuracy we run it in all three cartesian directions, XYZ. To summarize, pixel by pixel rays are computed in all three Cartesian directions directions and wherever a ray hits the surface of a mesh, that coordinate is given a *1*. This process continues for all rays, creating a 3D binary mask representing the surface object. The inside is then "filled" with *1*'s finishing the process off and leaving a 3D logical array representing the object's shape.

Projection2D.m

This code is adapted from *projection.m*, which takes a 3D image array (not logical) and computes the 2D cone x-ray projections in the Z-direction. The output of *VOXELISE.m* is a logical array so the 3D array must converted into either a single/double or grayscale image. These projections are computed using geometric re-sampling meshes, and 2-dimensional interpolation in order to determine the projection of each Z-slice onto the detector. The mathematics are explained later in the section 4.

The original *projection.m* function applies the projection in the Y-direction compared to my adaptation which projects in the Z-direction. This was done in order to maintain the correct geometry in the simulation. Future versions can include adding object rotations in order to examine different angled projections, but for the sake of code readability and efficiency it was not added.

3. Assumptions

There are multiple assumptions are made for this implementation to function smoothly. These range from experiment parameters, to the cartesian geometry in the setup. The main assumptions are described below.

STL File

STL files can be in either the ascii or binary file formats. The main assumption with the STL mesh is that triangular facets are utilized instead of any 2D shape. The voxelization code specifically is written with this assumption in mind. Additionally, the mesh that is ray-traced must be properly closed (“watertight”), to avoid artifacts during voxelization.

Simulation Setup

This implementation assumes the object is being imaged from x-rays in the z-direction. The setup has the center (x,y,z = 0,0,z) of the source, object, and detector plate all aligned along the Z-axis. The centre of the object is considered the origin (0,0,0). The x-rays can be visualized as if they are coming from a source above the object, projecting x-rays upon a 2D X-Y detector array beneath it. Visualizing further, the x-rays could be taken during total hip replacement surgery, where the hip joint lies over the x-y detector. **Figure 1** at the end of the ReadMe gives a representation of the setup.

Parameters

There a multitude of parameters that are used in the simulation process (specifically *projection2D.m*) that are assumed and set automatically to make computation easier. It should be noted that the option to set them any of them manually will be available in future versions for testing:

Gridsize: The x,y,z grid voxelization of the 3D STL file object is assumed to have enough voxels to fully represent the 3D model properly. It is recommended to have at least 100x100x100 voxelization (to be sure). It is recommended to use equal dimensions but it is not necessary.

Raycasting Direction: *VOXELISATION.m* is set to use raycasting in all three cartesian directions in order to provide the most accurate voxelization of the 3D mesh.

Object real size: The real size parameters for the 3D array encompassing the object is assumed to have a 1 to 1 size correlation with the grid parameters (1 pixel = 1 mm).

Detector panel pixel density and real size: To simulate a real detector, 256x256 pixels was selected as the pixel resolution, and calculated the approximate real size dimensions of the detector in order to achieve 300 PPI[2], as 375x375 mm:

$$\frac{(256^2\ pixels^2)}{\left(\frac{(375\ mm)}{\left(25.4\frac{mm}{inch}\right)}\right)^2} = \sim 300\ PPI^2$$

Distance of X-ray source to Detector: The distance of the x-ray source to the detector was kept constant at 1000 mm for this implementation, in order to simplify object distance calculations.

Distance of X-ray source to Object: This parameter is normally automatically computed in order to make sure the projection fits on the 2D detector properly, with no image cut-off. In other words, the distance is computed to make sure the 3D object array fits in the 'observation cone' the conical ray makes. This idea is demonstrated by **Figure 2**. With earlier parameters set, the calculation depends on the size of the z-direction 3D object array and is as follows:

$$tan(\theta) = \frac{\left(\frac{DD}{2}\right)}{D_D}$$

$$Radius = \sqrt{\left(\frac{Dx}{2}\right)^2 + \left(\frac{Dy}{2}\right)^2}$$

$$D_O = \frac{Radius}{\tan(\theta)} + \frac{Dz}{2}$$

Where, DD represents the detector dimension 375 mm, $D_{x,y,z}$ represents the dimensions of the 3D object array, D_D is the distance to the detector (along the z-axis) and D_O is the calculated distance to the object. This equation ensures that the proximal 3D array face to the x-ray source will always fit within the conic rays (by adjusting for half of the z-size of the array). Additionally, D_O can also be set manually in the *XraySim.m* function, in order to observe the change in 2D projection due to changing source-object distance. However, it is no longer guaranteed for the object to be fully in the x-ray cone-beam, and 2D projection cutoff may occur.

```
Example: 100x100x100 object

tan(θ) = (375/2) / (1000) = 0.1875
Radius = sqrt( (100/2)^2 + (100/2)^2 ) = 70.71 mm
```

```
D0 = 70.71/0.1875 + 100/2 = 427.12 mm away -> rounded to 427.
```

Sampling Parameters: The remaining parameters to describe are the scaling parameters, and geometric sampling vectors. The sampling parameters are calculated from the ratio of the voxel size vs real life size of the 3D object. These scaling parameters are then used to compute geometric sampling vectors that represent the object and detector in real space. For example, if the object is 100x100x100:

```
Example: 100x100x100 object

dx,dy,dz = 1; % object sampling factor
du,dv = 1.465 % detector sampling factor

xs,ys,zs = [-49.5, -48.5, … 48.5, 49.5] % object sampling vectors 1x100 size
us,vs    = [-186.77, -185.3, … 185.3, 186.77] % detector sampling vector 1x256 size
```

This results in vectors that are 1x100 and 1x256 for the object and detector respectively. The sampling is based upon the centre of the detector and object being 0, and aligned along the Z-axis, with the center of the 3D object array being the origin of the system.

Geometry

With the parameters described in the above section, this section describes the cartesian geometric setup of the simulation in more detail. As stated earlier, the projections are performed on the 3D object in the z-direction, with the x-ray source, object and detector plate being center aligned on the z-axis (x,y =0,0). For simplicity of calculations the major geometric assumptions for this setup include assuming that there is no detector, source or object shifts and rotations. This can be implemented in future versions.

From the sampling parameters description, the center of the 3D object array is the origin of the cartesian system having coordinates (x,y,z) = (0,0,0). It is known that the source and detector plate are at a fixed distance from each other at 1000 mm. An object placed at exactly 500 mm for example, would make the point source at coordinates (0,0,-500) and the detector plate origin at (0,0,500). The calculations in the previous section adjust where this origin point of the system is along the z-axis based upon the size of the 3D object array in order to make sure it fits within the conical view. With our current static parameters you can use trigonometry to determine the largest 3D array that can fit inside the cone before contacting the detector plate, the same way the object distance is calculated. This value is 209x209x209 mm^3 as any larger, means the 3D array would have to ‘pass through’ the detector to ensure fit.

The mathematics behind the slice by slice 2D projections is described in section 4.

4. 2D Conic X-ray Projection Mathematics

The mathematics behind the 2D cone-beam projection implementation is based off of linear algebra and perspective projection. Perspective projection is a linear projection of a 3 dimensional object that is projected onto a picture plane. This effect makes distant objects appear smaller than nearer objects. This effect can be visualized by examining **Figure 1**.

Using the variables in **Figure 1**, the 3D position of a known point can be used to calculate the 2D projection of it on to a display (detector) surface at a known distance by use of the following steps and equations:

- Let the origin be the centre of a 3D object
- Let (*x,y,z*) represent the 3D position of the point inside the object
- Let Z_{DSO} represent the distance from the source to the centre of the object
- Let Z_{DSD} represent the distance from the source to the centre of the detector
- Let (*Xd,Yd*) represent the 2D position of the projected point on the detector
- Let θ_x represent the (x,z) plane ray angle
- Let θ_y represent the (y,z) plane ray angle

1. First compute the (x,z) and (y,z) plane angles:

$$\tan(\theta_x) = \frac{x}{(Z_{DSO} + z)} \qquad (1)$$

$$\tan(\theta_y) = \frac{y}{(Z_{DSO} + z)} \qquad (2)$$

2. The same angles, are related to the projected 2D points by the next equations:

$$\tan(\theta_x) = \frac{Xd}{Z_{DSD}} \qquad (3)$$

$$\tan(\theta_y) = \frac{Yd}{Z_{DSD}} \qquad (4)$$

3. Sub equations (1) and (2) into (3) and (4) respectively, and solve for *Xd* and *Yd* respectively.

$$\frac{x}{(Z_{DSO} + z)} = \frac{Xd}{Z_{DSD}} \qquad (5)$$

$$Xd = x * \frac{Z_{DSD}}{(Z_{DSO} + z)} \qquad (6)$$

$$\frac{y}{(Z_{DSO} + z)} = \frac{Yd}{Z_{DSD}} \qquad (7)$$

$$Yd = y * \frac{Z_{DSD}}{(Z_{DSO} + z)} \qquad (8)$$

4. This solution demonstrates that the (*x,y,z*) point is projected onto the 2D plane by use of the factor $\frac{Z_{DSD}}{(Z_{DSO}+z)}$ which is the ratio between the source-detector source-object distance. This factor is adjusted by which z-slice the projection algorithm is in (the z value) which causes the further slices (larger z) to result in smaller projections.

The inverse of this factor is applied to the geometric re-sampling meshes before using them to interpolate the 2D projection in the function *projection2D.m*. The inverse of the factor is applied because the relationship between x and Z_{DSD} is not linear (required for linear interpolation). However, x is linearly related to $\frac{1}{Z_{DSD}}$, which is why we apply the inverse of the function before linearly interpolating. Further explanation for why the inverse of the factor is applied is explained by page 4 of the article:

Perspective Texture Mapping Part 1: Foundations by Chris Hecker

A pdf of this article can be found in the **Appendix** folder of the package. For a more in depth read on the topic of perspective projection refer to the following link:

https://en.wikipedia.org/wiki/3D_projection#Perspective_projection

5. Cone-beam Projection vs. Parallel Projection

This section details the reasoning for selecting a cone-beam projection implementation vs. implementing parallel beams.

First, the main advantage of a parallel x-ray beam implementation is that it gives a geometrically correct 2D projection no matter the object's rotation, distance (from x-ray source or detector) and size. By considering the x-ray beams like ray-tracing (done by *VOXELISE.m*) in a specific direction, the projection can be computed quite simply. For example, in the package folder labelled "**Simulated X-ray Image Projections**", there are examples of utilizing *VOXELISE.m* in order to compute the parallel projections of the example STL files as a comparison to the cone-beam projections. This can be also visualized by **Figure 3**. The code used to create the parallel projection in those images and **Figure 3,** is as follows:

```
% Parallel Beam Implementation of STL Object using VOXELISE.m
[Object] = VOXELISE(200,200,200,'Cylinder.stl','xyz');
Projection = mat2gray(squeeze(sum(Object,3)));
imwrite(Projection, 'Cylinder Parallel.bmp')
```

Comparatively, the cone-beam 2D projections have geometric distortion due to the change in projection angle based upon the object distance from the x-ray source. This is due to the geometry behind the perspective projection, explained in the section before; image slices closer to the x-ray source have a larger projection (on the detector) than the slices closer due to the transformation of the (x,y) coordinates of the object. This results in a more proximal object having a larger projection even if it has the same size as another object further away from the point source (this is not observed in parallel implementation as it will have the same projection).

While this is an issue when concerned with exact anatomical representation (2D projection) of the object, these images can still have great value. For example, if the object of interest has a small z-size, the cone-beam 2D projections will provide a fairly accurate representation of the actual geometric truth with minimal distortions (almost identical to parallel). While this was out of the scope of the implementation, these distortions can be corrected utilizing perspective correction (homography matrix).

In terms of general usage, the parallel implementation requires a parallel collimator in order to make parallel rays. The x-ray beam divergence heavily depends on the quality and resolution of the collimator making it an expensive requirement. The cheaper, smaller and more portable cone-beam implementation of the x-ray source is a major advantage for real-life applications.

Lastly in terms of real life application, a cone-beam implementation has multiple advantages over standard Computed Tomography (CT) fan beam implementation. Cone-beam CT produces similar quality images compared to conventional CT (fan beam) with the advantage of requiring a smaller and less expensive machine/setup. Cone-beam CT application delivers far less x-ray dosage, and utilizes a medical

fluoroscopy tube as the x-ray source (compared to a high-output anode x-ray tube in conventional CT).

6. References

1. **VOXELISE.m** and **READ_stl.m** are both from the mesh voxelisation toolbox by Adam A.: https://www.mathworks.com/matlabcentral/fileexchange/27390-mesh-voxelisation
2. The raycasting method used in the function **VOXELISE.m** from the voxelization toolbox by Adam A. is based on:
   Patil S and Ravi B. Voxel-based representation, display and thickness analysis of intricate shapes. Ninth International Conference on Computer Aided Design and Computer Graphics (CAD/CG 2005)
3. **projection2D.m** is adapted using code from **projection.m**, **XraySim.m** has code adapted from ParamSetting.m. Both are from the 3D Cone beam CT (CBCT) projection backprojection FDK, iterative reconstruction toolbox by:
   Kyungsang Kim
   https://www.mathworks.com/matlabcentral/fileexchange/35548-3d-cone-beam-ct-cbct-projection-backprojection-fdk-iterative-reconstruction-matlab-examples

7. Authors

Michael Behr – ReadMe + MATLAB coding + Testing Protocol
Contact - MichaelBehr13@gmail.com
Insitution - Toronto Rehabilitation Institute, University Health Network

8. Figures

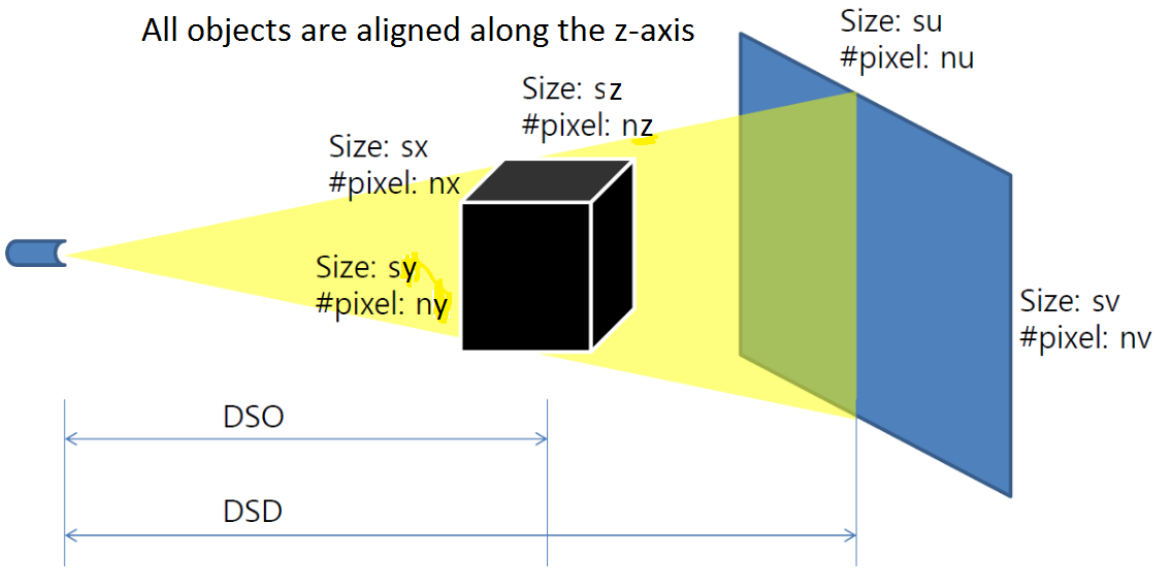

Figure 1: Cone-beam implementation demonstrating the simulation setup. The object isocenter is considered the origin for mathematical calculations. This figure is adapted from the Cone-beam toolbox by Kyungsang Kim.

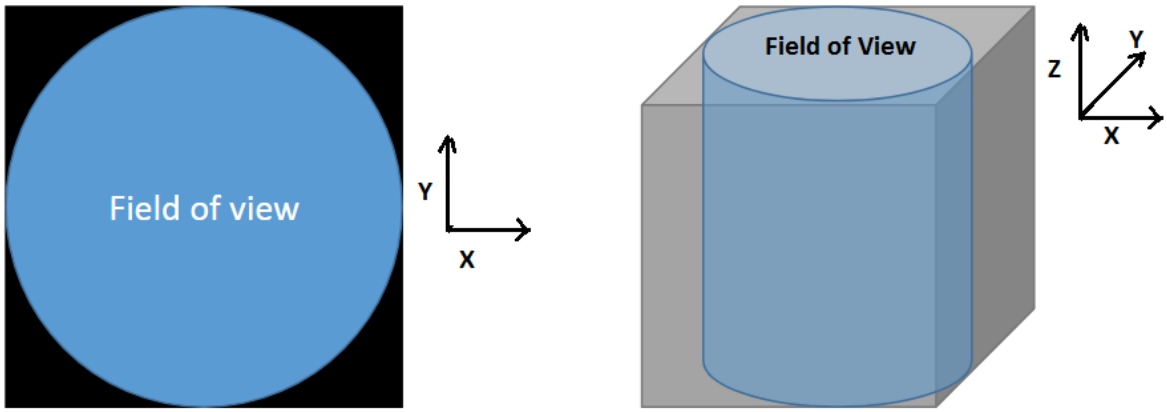

Figure 2: The object should fit within the field of view in order to have its projection not cutoff on the detector. This figure is adapted from the Cone-beam toolbox by Kyungsang Kim.

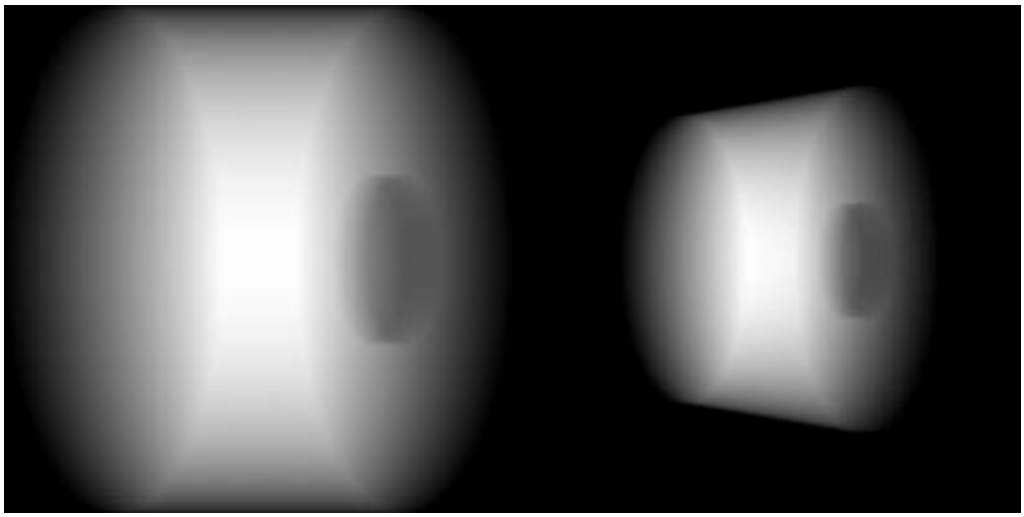

Figure 3: Parallel X-ray implementation vs Cone-beam implementation. Notice the perspective distortion in the 2nd image due to cone geometry.