How the Reconstruction Script Works

How the Reconstruction Script Works

Introduction

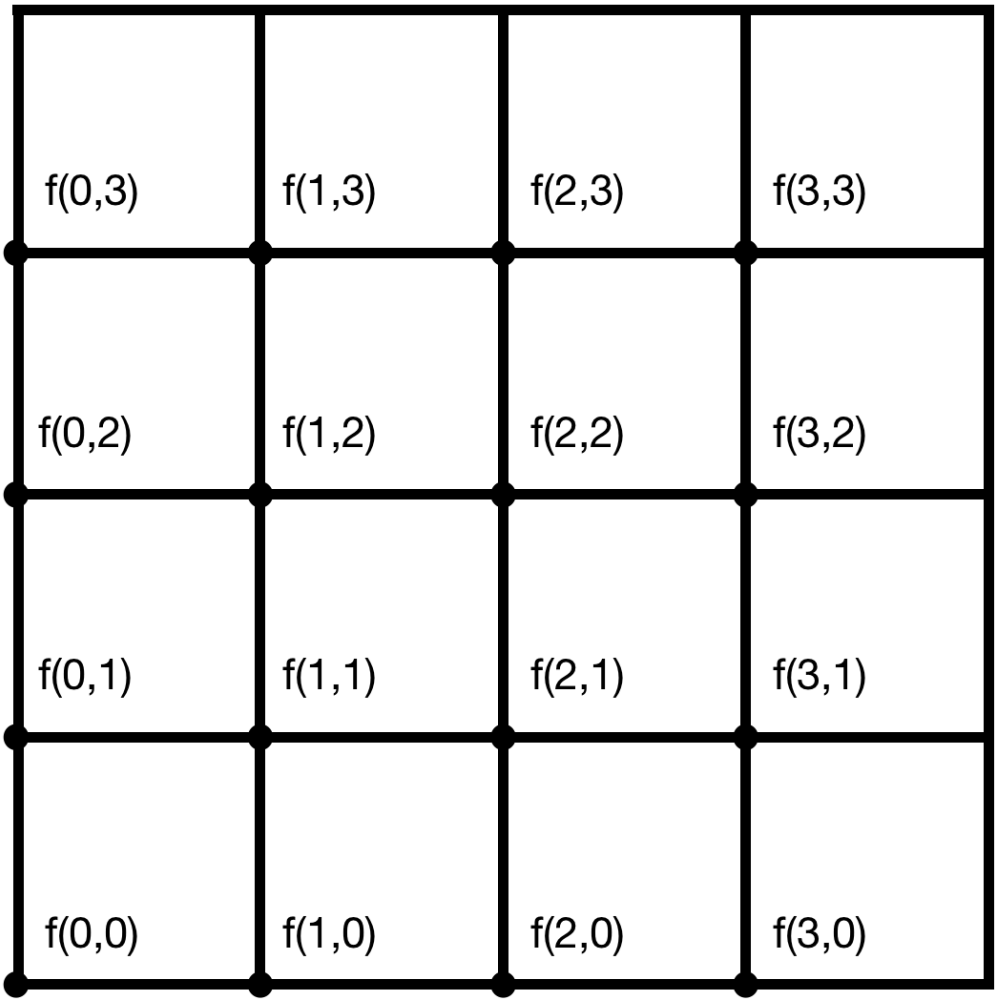
This document serves to explain how the script works to implement the formulas discussed. It will show a brief overview of the formulas and their corresponding implementations. The program is written in MATLAB with C++ in performance-critical components. Only the MATLAB components will be discussed in this document. All the code discussed, unless stated otherwise, is in the **Main_Script.m** file.

Part 1: Generating the Phantoms

The phantoms are mathematical functions defined over 3D domains. They are represented as a discrete 3D grid in memory. Each value in the grid corresponds to a value of the phantom. As with any computer, memory is finite so we need to define the bounds of the grid. So we start by defining:

```
d_half = 200;
[x, y, z] = meshgrid(-d_half:d_half, -d_half:d_half, -
d_half:d_half);
```

This creates a 400x400x400 grid, which is deemed suitable. Keep in mind that an increase in a grid dimension will cubically scale memory occupation and computing operations. Remember that they're simply mathematical functions. To fit in computer memory, they're discretised on a 3D domain. We can imagine this as a rectangular cluster of voxels, with each voxel carrying a value of the above-mentioned function. Below is a visualisation of a 2D slice of the voxel grid:



Notice how the coordinates of each voxel lie at their **bottom-left**. Now let's generate the phantoms, in the form of a mathematical function. We have different options: uncomment the one you want to use. First, we have the spheres:

```
% Gaussian sphere
% phantom_grid = (1 - heaviside(sqrt(x.^2+y.^2+z.^2) - 70)) .*
exp(-(x.^2+y.^2+z.^2) / 1000);

% Monochromatic sphere
% phantom_grid = (1 - heaviside(sqrt(x.^2+y.^2+z.^2) - 20));
% phantom_grid = heaviside(sqrt(x.^2+y.^2+z.^2) - 30);
```

Then we have the rectangles:

```
% Rectangle
% phantom_grid = (heaviside(x + 100) - heaviside(x - 100)) .*
(heaviside(y + 100) - heaviside(y - 100)) .* (heaviside(z + 100) -
heaviside(z - 100));
% phantom_grid = 1 - (heaviside(x + 10) - heaviside(x - 10)) .*
(heaviside(y + 10) - heaviside(y - 10)) .* (heaviside(y + 10) -
heaviside(y - 10));

% Dual x-pillars
% phantom_grid = (heaviside(y - 10) - heaviside(y + 10));
```

And if you want, we can have the famous Sheep-Logan image:

```
% phantom_grid = repmat(phantom('Modified Shepp-Logan', 2*d_half),
[1, 1, 2*d_half]);
```

And we plot it like:

```
figure(1)
imagesc(phantom_grid(:,:,d_half))
colormap gray
```

And that's it for the generation!

Part 2: The Forward Problem

Now that we've generated the phantoms, let's move to the forward problem. Here, we'll compute the Radon transforms over V-lines. The final output will also be defined over a grid; each voxel in the grid contains the value of the following formula:

$$f[k] = \int_{-a}^{a} \int_{0}^{\pi/2} \int_{l} \vec{V} \cdot \vec{\mathrm{d}L} \mathrm{d}\theta \mathrm{d}s$$

This is the one listed in the papers; I omitted some details. Anyways, since the voxel grid is finite, we change the integrals into summations and we discretise it. To perform the summation, we'll nest two loops: one for the spatial integration, and another for the angular integration. The line integral itself is folded into another function, which we won't discuss here.

First, declare some variables required for discretisation:

```
voxel_iterations = length(phantom_grid) - 1;

% The angular resolution...
d_theta = pi / 80;
angle_iterations = length(0:d_theta:(pi/2-d_theta));

forward_array  = zeros(voxel_iterations, angle_iterations);
```

So `voxel_iterations` is simply the summation end points used when discretising the station integrals. Then `d_theta` is the quantised angle that will be used in the angular summation, and `angle_iterations` is the number of quantised angles. You'll find that `angle_iterations * d_theta = pi / 2`. So declare the following loop first, for the spatial integral:

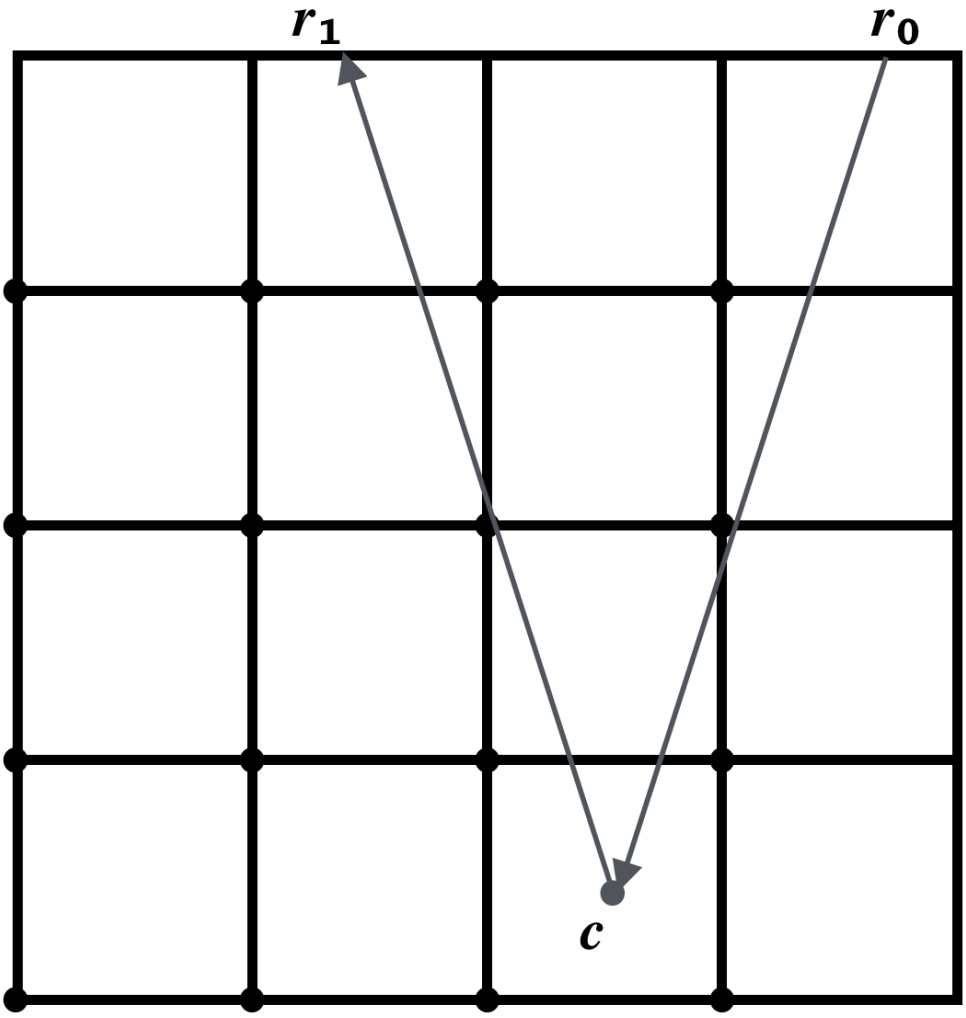
```
for i = 1:2*d_half

    voxel_centre = -d_half + ([i, 1, d_half] - 1) + 0.5;
    voxel_value = 0;

    ...

end
```

So here, we are iterating across voxels at the bottom of a 2D slice of the 3D domain. In each voxel, we calculate the centre coordinate. The `voxel_value` will be used to accumulate all the integral values. Next, replace the `...` with:

```
for t = 2:angle_iterations-1

        angle = d_theta * (t - 1);
```

```
        % Compute the rays...
        v_0 = [tan(angle), 1, 0];
        v_0 = v_0 / norm(v_0);

        v_1 = [-v_0(1), v_0(2), v_0(3)];

        % To compute the end points, let's throw them far away from
        % the domain and let the clipping take care of matters...
        r_0 = voxel_centre + d_half * sqrt(6) * v_0;
        r_1 = voxel_centre + d_half * sqrt(6) * v_1;

        % We now perform the ray integrals...
        [ray_01, ray_11] = clip_ray(r_0, voxel_centre, d_half);
        [ray_12, ray_02] = clip_ray(voxel_centre, r_1, d_half);

        voxel_value = SLLineIntegralApproximator(phantom_grid,
d_half, ray_01, ray_11);
        voxel_value = voxel_value +
SLLineIntegralApproximator(phantom_grid, d_half, ray_02, ray_12);

        forward_array(i, t) = voxel_value;

    end
```

As a brief overview, this code generates a V-line as a combination of 2 rays. Each ray is defined between 2 end points, shown in the diagram below. The vector $\vec{c}$ is the centre of the voxel, which is represented as `voxel_centre` in code.



When computing $\vec{r_0}$ and $\vec{r_1}$, we first calculate two unit vectors $\vec{v_0}$ and $\vec{v_1}$, which are the normalised displacement vectors of the two rays. We then add the voxel centre offset, and multiply them with a scale factor that guarantees that $\vec{r_0}$ and $\vec{r_1}$ lie outside the domain. Subsequently, we call `clip_ray()` to move $\vec{r_0}$ and $\vec{r_1}$ to the boundaries of the domain, as seen in the diagram above. The workings of this function are irrelevant for this document.

After generating the V-lines, we perform the integration using the `SLLineIntegralApproximator`. This is a function which computes a line integral for a ray in a function defined over a 3D grid. It's written in C++ for optimisation.

We then add it to the final array for the forward problem.

The Inverse Problem

We finally end at the inverse problem. Just like the forward problem, this is a simple implementation of the formula we discussed. So since it's a 2D reconstruction, we'll have a double-summation for each spatial dimension; each one iterating through voxels. Within these, we have another summation for the angular component. Start by defining the number of voxel iterations and the output data structures:

```
inverse_array = zeros(voxel_iterations, voxel_iterations);
```

Now let's create the loops for the spatial integral component:

```
for i = 1:2*d_half
   for j = 1:2*d_half

       voxel_value = 0;

       ...

       inverse_array(i, j) = voxel_value;

   end
end
```

This is a pair of nested loops representing a double summation. The `voxel_value` is simply the value of the reconstructed voxel at the pixel coordinate `(i,j)`. It starts at zero and gets successively incremented in the angular summation, which we'll see shortly. So replace the `...` with:

```
       for angle_index = 2:angle_iterations-1

           theta = d_theta * (angle_index - 1);
           t = tan(theta);

           if round(i + j * t) <= 2*d_half
               temp_val = forward_array(round(i + j * t),
angle_index) / sqrt(1 + t^2) * tan(d_theta);
```

```
                voxel_value = voxel_value + temp_val;
            end

            if round(i - j * t) >= 1
                temp_val = forward_array(round(i - j * t),
angle_index) / sqrt(1 + t^2) * tan(d_theta);
                voxel_value = voxel_value + temp_val;
            end

        end
```

Remember that the formula contains two summation components here. The use of the `round()` function guarantees that the loop boundaries are integers, as there will always be small errors when storing non-integer values in computers. Lastly, after all these loops, we plot the result. So at the very end of the file, add:

```
figure(2)
imagesc(inverse_array)
colormap gray
```

And that's it!