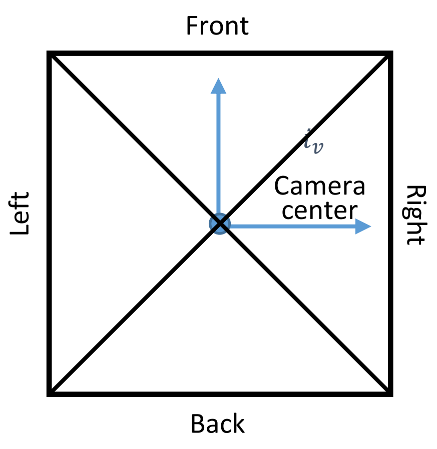
Image based rendering from 360° photos

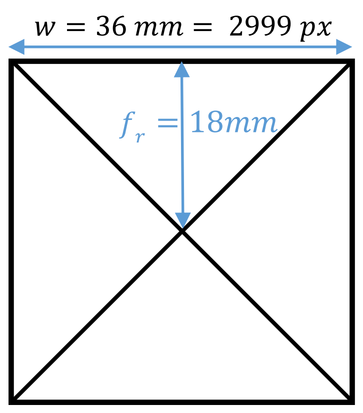
The objective of this assignment is to create a virtual view renderer, which takes as input 4 photos that show a 360 degree view around the camera, and outputs a new view at an arbitrary direction. The catch is that the perspective distortion in the input images has to be corrected, otherwise objects do not preserve their shapes. This kind of processing can be used e.g. to render the correct view when wearing virtual reality headset and watching a 360 degree video.

Your implementation should run without errors or warnings on a 2017a Matlab install by executing *"run LW1.m"*

1. Familiarize yourself with the data set [Mandatory]

Figure out in which order the images are. Designate one of them as the "front" view, and name the others accordingly as "back", "left" and "right" in reference to the front view. Convert them to grayscale (`help rgb2gray`)





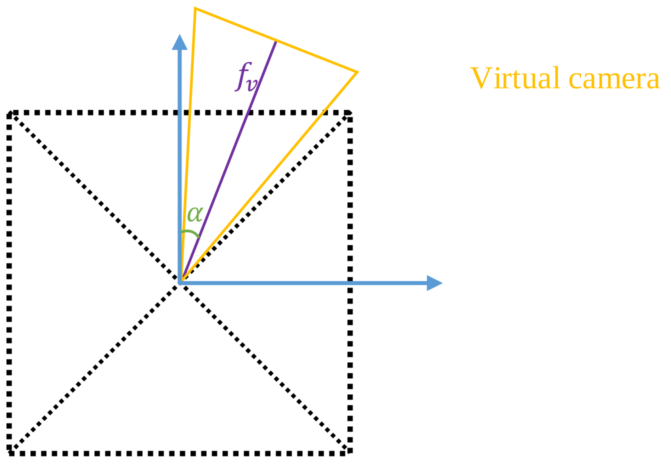
2. Combine the images side by side to form a panorama [Mandatory]

The resulting image has the same height as the input images, and 4 times the width. Make sure the order of the images is correct, i.e. objects on the edge of the image continue in the adjacent image. The capturing process is hardly ideal, so don't expect the images to align perfectly. Observe what happens to straight lines at the borders between the images. This panorama image is just for reference (i.e. "what happens if the projections are not considered correctly") and not used for further processing.

3. Make a function that computes the global space coordinates of the image plane pixels of a **virtual** camera [Mandatory]

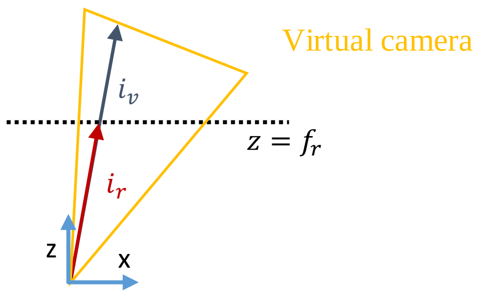
1. The inputs to the function are the virtual camera focal length f_v and the angle α in which the camera is facing.
2. List all the pixel coordinates (from 1 to image size for both dimensions) on the image plane (`help meshgrid`), and then shift the coordinate system so the value in the middle position becomes (0,0).

3. Add f_v (for instance 24 mm, but feel free to experiment with other values as well) as the *z* coordinate to the pixels to give them a location in 3D space. It should also be given in pixels for the sake of consistency, so figure out the pixel size from the width of the reference image in pixels and the width of the sensor and convert. The pixel size of the virtual camera is considered to be the same as the reference camera.
4. **Keep only every 4th coordinate point in both dimensions.** This will reduce the dimensions of the resulting image to 1/4.
5. Orient the points to the correct direction using a rotation matrix *R*. *R* should be the matrix that rotates around the axis which points up (`help roty`) with α as the parameter.



4. Back-project the virtual image points to the front image [Mandatory]

Find the intersection between the side and a conceptual ray of light which starts from the optical center of the virtual camera, and is aimed at each of the pixels on its camera plane. The intersection points will give the locations in which the input image should be sampled from to assign a color for the virtual camera pixels. Hint: consider the 3D pixel location as a vector i_v from the origin to the pixel. Find how much the vector should be scaled so it reaches the image plane of the front facing real camera, i.e. $si_v = i_r$. Since the front plane is aligned with the coordinate axes, its representation is simply $z = f_r$.



5. Sample the image cube [Mandatory]

Shift the intersection points from the previous task back to default Matlab coordinates where *(1, 1)* is in the corner of the image. Sample the front image (`help interp2`) from those coordinates.

6. Repeat the resampling for all sides of the cube [+1]

Repeat the back projection and resampling from tasks 4 and 5 for all sides of the cube. The other image planes are $x = f_r$, $x = -f_r$ and $z = -f_r$. If the scaling factor for a pixel vector is negative, it isn't facing that particular side, and thus should not contribute to the output image. Such intersection coordinates should be set to NaN, so they won't interfere with the proper image. For each side, you'll get one sub image which is either partially covered in pixels, or is blank, depending on to viewing direction of the camera. Hint: note that for some directions, the coordinates become flipped in the horizontal direction, so compensate that (`help fliplr`) to avoid mirroring in the results.

7. Combine the valid pixels from the different sides of the cube [+1]

From each sub-image, pick the pixels that have proper values. Sampling from coordinates (NaN, NaN) produces NaN as the output in the previous task, which can be used to assist in discarding the extra pixels (`help isnan`). If everything went according to plan, for each pixel, there is only one sub-image that has a valid (non-NaN) value for each pixel in the new image. Collect those values in a single image. This is the virtual camera image from the middle towards the requested viewing direction

8. Color video [+1]

Apply the processing to RGB images instead of grayscale. Make a loop that calls the function repeatedly with α iterating over the range $[-180,\ 180]$, so that it looks like the camera is rotating. The step should be at most 5 degrees, but the smaller you make it, the smoother the video will be. Place the frames in a single matrix in a format that the Matlab video player understands and is able to play (`help implay`)

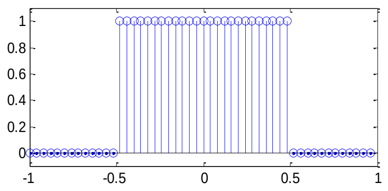
9. Apply anti-aliasing filtering [+1]

Task 5 is a typical case of sampling a signal with a sampling rate lower than that of the original signal. Therefore it may cause high frequency content to be aliased. In order to prevent this, design and apply a 2D anti-aliasing filter using the windowing method.

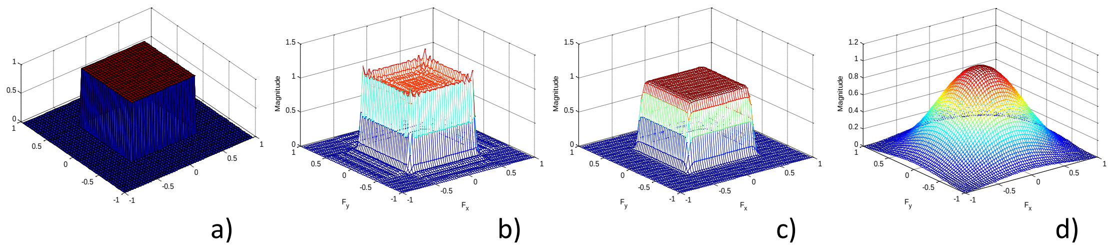
1. Create an enumeration of horizontal and vertical frequencies in 2D (`help freqspace`) of size 50x50

Threshold the frequency lists with the cutoff frequency, i.e. make a vector where frequencies whose **absolute** value is above the cutoff are zeros, and those below are ones. The cutoff frequency should be calculated keeping in mind both the decimation factor, and the change in the sampling density resulting from the ratio between f_v and f_r. E.g. if the decimation factor would be 2 and the virtual focal length twice the reference focal length, the sampling rate is effectively the same and no filtering would be needed. The following figures are created with an effective sampling rate of 0.5. For your implementation, calculate the rate for the f_v you are using.

2.



3. Multiply the two 1D binary vectors via matrix multiplication to create a 2D binary matrix. The result should be an ideal response for a filter that cuts off the necessary amount of frequencies in both dimensions, i.e. should look like box (a) below (plotted by `surf`).



4. Make a 2D Gaussian window of the same size (`help fspecial`). Pick some value for the parameter sigma; it will be refined in step 6.
5. Create the filter from the ideal response and the window function (`help fwind2`) and normalize it's overall energy (sum) to one
6. Check the frequency response of the filter (`help freqz2`). Make sure that it resembles closely the ideal response, i.e. doesn't have notable ripple on the passband (b) and the transition band doesn't become too wide (d). Adjust the sigma parameter of the Gaussian window if necessary to reach something like (c).
7. Apply the filter to the input images before resampling them in task 5 (`help imfilter`)