# Cassini Propeller Detector

## Algorithm description

pfr

## Introduction

There are many simple improvements that are sorely missing from this submission, such as dynamically constructing orbits from candidate objects, and various improvements to the local forest. In the end I didn't implement those features because I had a comfortable lead on the provisional tests and I discovered that there was another competition going on at the same time. I suspect that the system tests may have been much more reliant on those features than the provisional tests.

# Part I.
# Processing pipeline

## 1. Coordinate transform refinement – `Shot::adjust()`

### 1.1. Input images generation

As a first step, missing lines are removed from the input, and incomplete lines are filled from neighboring pixels. The outermost rows and columns are also removed, so that we manipulate images of size $1022 \times 1022$ when no lines are missing.

We then use the provided camera pointing angle to render a theoretical binary image of the outer A ring, based on the static array `split_radii` which contains the list of discontinuities that are relevant to the images we wish to process (specifically, the Encke gap and the Keeler gap).

This theoretical binary image allows us to determine the typical luminous flux for the rings and for outer space with a simple median, which now allows us to quantize the input image according to these two levels.

### 1.2. Image alignment

If we can find an affine transform that matches these two images, this will allow us to recover the correct pointing angle, rotation, and field of view for the camera.

This is done in `ImageAligner` by minimizing the squared error between the theoretical view and the captured image, first at a coarse 32× subsampling, and then gradually increasing resolution until the 2× subsampling level. Minimization is performed by fixed-step displacements on a grid.

Matching the theoretical view takes care of absolute radial positioning; for increased precision we also include an error term favoring correlation of pixels located at similar radii.

Note: Because the transform will generally require pixel values from the theoretical view that lie outside the original image boundaries, we actually render that image with extra margins on the side.

## 2. Filtering – `Shot::process()`

### 2.1. Flat field correction

We first multiply the image with a $256 \times 256$ correction image to account for non-uniformity of flat field images. The correction image is generated as described in part 2.

### 2.2. Warping

We warp the image into polar coordinates. We choose the dimensions so that one pixel in the input image approximately corresponds to one pixel in the warped image. This generally means that there will be anisotropy in the warped image, that is, one pixel in the radial direction may not represent the same distance as one pixel in the longitudinal direction.

Warping is done by bicubic sampling, after a light blurring step intended to reduce aliasing.

### 2.3. Ring removal

The rings can now be removed by subtracting the median of each row. We then improve the subtraction by fitting a piecewise-cubic $C^1$ function for each row, under an $L^1$ loss to avoid being affected by isolated bright objects.

We then use the original median ring brightness to normalize image brightness.

### 2.4. ADC parasite removal

Cassini images feature oscillations that appear as a horizontal band sinusoids. We remove them by considering the original row in which each warped pixel appeared, and subtracting an approximation of the median of each original row.

### 2.5. Noisy region attenuation

The region from 134,205 to 134,375 km often features high-amplitude medium frequency noise. Since the detection stage won't have access to radius data (in order to prevent overfit), we attenuate this region to avoid creating false positives.

# 3. Prediction – `Shot::process()`

## 3.1. Local analysis

We use the convolution kernels determined in part 2 and feed the results of the convolution as features to the local forest trained in part 2. This creates an image representing the probability that the pixel represents a propeller object, based on local features. Because of the class weights used during training, this posterior probability is based on a 200:1 prior probability ratio of non-objects to objects. We reject pixels where this posterior probability is lower than 50%.

## 3.2. Candidate extraction

We extract the local maxima of probability. We consider entire connected regions of probability greater than 20% as a single candidate. We extract the horizontal and vertical size of this region to use as a feature for the adjustment forest.

Finally, we normalize the probabilities of all candidate objects found in a given image so that their sum never exceeds 1. This is because the expected number of objects in any given image rarely exceeds 1.

## 3.3. Probability adjustment

We now feed all candidates to the adjustment forest trained in part 2, which computes an updated probability that takes into account the shape of the object.

# 4. Merging all candidates – `PropellerDetector::getAnswer()`

## 4.1. Static orbit computation – `Universe::initialize()`

We fit the ground truth data into orbit predictions by finding a slight perturbation $a$ of the mean radius so that the observed longitudes can be well predicted by the orbit of an object of semi-major axis $a$, which has constant period determined by Kepler's third law. Quality of prediction is measured by RMS error. This error level is remembered in order to create an axis-aligned multivariate normal distribution model for objects belonging to the orbit. This will later allow determination of the likelihood that an object belongs to a given orbit.

## 4.2. Final probability computation and linking

For each candidate object, we find the likeliest orbit and its likelihood. If the likelihood exceeds a threshold, we mark that object as belonging to that orbit. The final updated probability is the product of the original probability and the orbit probability, which is the sum of a fixed unknown-orbit probability and of the likeliest known-orbit probability.

Finally, some objects are present in the input data twice. This shouldn't improve the score much since this isn't very common, but as a nod to the previous NASA contest we emit all objects twice with a heavily reduced probability the second time.

# Part II.
# Training data generation

Training data is generated in three passes, each pass using the results of the previous one.

## Pass 1: Flat field image

The flat field correction factor is computed as the per-pixel median across all the training data of the ratio between a synthetic image where the brightness of each ring was made uniform, and the original image. We downsample both images by a factor of 4 to avoid excessive memory usage, and we discard samples where the ratio deviates by more than 10% from unity.

## Pass 2: Local convolution kernels

We apply K-means clustering to image patches of size $11 \times 5$ representing objects on the one hand, and non-objects on the other hand. We use 10 clusters for objects and 4 clusters for non-objects.

This gives us 14 representative $11 \times 5$ images, which will be used as convolution kernels.

## Pass 2: Local forest

We simply train a random forest based on the results of the convolution, which lets us predict a probability that a pixel represents a propeller based on its local neighborhood. No other features are used.

Class weights are chosen so that the non-objects have total weight 200 times greater than the objects.

## Pass 3: Adjustment forest

Finally, we train a random forest based on the candidate probability and the candidate width, height, and height-to-width ratio. The prediction is an adjusted probability that the candidate is an object.

We don't use any class weighting here.