

# Multiview Onboard Computaional Imager

## Algorithm Theoretical Basis Document

Caleb, Adams  
CalebAshmoreAdams@gmail.com

LastName2, FirstName2  
first2.last2@xxxxxx.com

LastName3, FirstName3  
first3.last3@xxxxxx.com

October 30, 2019

# Contents

<b>section</b>	<b>1</b>
<b>1 Introduction</b>	<b>2</b>
1.1 File Formats . . . . .	2
1.2 Feature Detection . . . . .	2
1.3 Feature Discription . . . . .	2
1.4 Feature Matching . . . . .	2
1.5 2 View Reprojection . . . . .	2
1.6 N View Reprojection . . . . .	2
1.7 Surface Reconstruction . . . . .	2
<b>2 Feature Detection</b>	<b>3</b>
<b>3 Feature Extraction</b>	<b>4</b>
<b>4 Feature Matching</b>	<b>5</b>
4.1 Overview . . . . .	5
4.2 2 View Matching . . . . .	5
4.3 N View Matching . . . . .	5
<b>5 Two View Reprojection</b>	<b>6</b>
5.1 Overview . . . . .	6
5.2 Getting Equations of Lines . . . . .	6
5.3 Minimum Distance Between Skew Lines . . . . .	8
5.4 Least Square Approximation for Two Lines . . . . .	9
<b>6 N/Multiview Reprojection</b>	<b>10</b>
6.1 Overview . . . . .	10
6.2 3 by 3 Inversion Method . . . . .	10
<b>7 Reconstruction</b>	<b>11</b>
7.1 Overview . . . . .	11
7.2 Orienting the Point Cloud . . . . .	11

# Chapter 1

## Introduction

1.1 File Formats

1.2 Feature Detection

1.3 Feature Discription

1.4 Feature Matching

1.5 2 View Reprojection

1.6 N View Reprojection

1.7 Surface Reconstruction

## Chapter 2

# Feature Detection

## Chapter 3

# Feature Extraction

## Chapter 4

# Feature Matching

### 4.1 Overview

### 4.2 2 View Matching

### 4.3 N View Matching

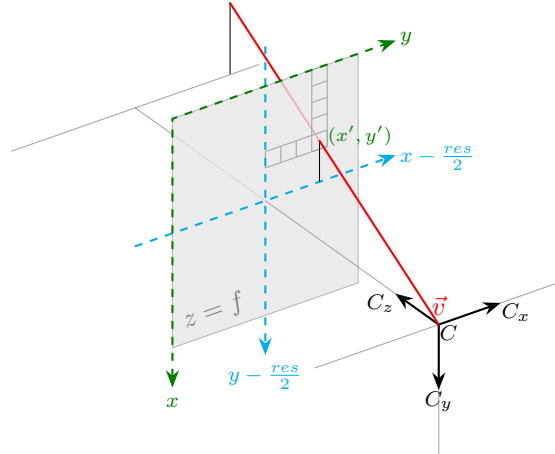
# Chapter 5

## Two View Reprojection

### 5.1 Overview

The goal with the 2-view reprojection is to take the pairs of matched points from the SIFT steps and move them from  $\mathbb{R}^2$  into  $\mathbb{R}^3$  so that equations of lines can be generated from the focal point of the camera into each matched point. Then, we want to find the minimum distance between those lines and choose the midpoint of that line segment at our reprojected point.

### 5.2 Getting Equations of Lines



The goal here is to generate a parametric equation of a line given camera position and orientation coordinates  $C$ , the camera focal length  $f$ , and the position of a coordinate in  $\mathbb{R}^2$  on the image plane. We wish to generate vector  $v$  that can be used to make the parametric equation. The 2-view reprojection takes the matched points between 2 images and places them into  $\mathbb{R}^3$ . To place each set of points into  $\mathbb{R}^3$  some trigonometry and matrix transformations need to take place. The first step to moving a keypoint into  $\mathbb{R}^3$  is to place it onto a plane in  $\mathbb{R}^2$ . the coordinates  $(x', y')$  in  $\mathbb{R}^2$  require the size of a pixel  $dpix$ , the location of the keypoint  $(x, y)$ , and the resolution of the image  $(xres, yres)$  to yield:

$$x' = dpix(x - \frac{xres}{2}) \quad y' = dpix(y - \frac{yres}{2})$$

This is repeated for the other matching keypoint. The coordinate  $(x', y', z')$  in  $\mathbb{R}^3$  of the keypoint  $(x', y')$  in  $\mathbb{R}^2$  is given by three rotation matrices and one translation matrix. First we treat  $(x', y')$  in  $\mathbb{R}^2$  as a homogenous vector in  $\mathbb{R}^3$  to yield  $(x', y', 1)$ . Given a unit vector representing the camera, in our case the spacecraft's camera's, orientation  $(r_x, r_y, r_z)$  we find the angle to rotate in each axis  $(\theta_x, \theta_y, \theta_z)$ . in our simple case we find the angle in the  $xy$  plane with:

$$\theta_z = \cos^{-1} \frac{([1 \ 0 \ 0] \cdot [r_x \ r_y \ r_z])}{\sqrt{[r_x \ r_y \ r_z] \cdot [r_x \ r_y \ r_z]}}$$

Future software will generate rotations for all planes in an identical way. Now, given a rotation in each plane  $(\theta_x, \theta_y, \theta_z)$  we calculate the homogeneous coordinate  $(r_x, r_y, r_z, 1)$  in  $\mathbb{R}^4$  using linear transformations. The values  $(T_x, T_y, T_z)$  represent

a translation in  $\mathbb{R}^3$  and use camera position coordinates  $(C_x, C_y, C_z)$ , the camera unit vectors representing orientation  $(u_x, u_y, u_z)$ , and focal length  $f$ :

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta_x) & -\sin(\theta_x) & 0 \\ 0 & \sin(\theta_x) & \cos(\theta_x) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\theta_y) & 0 & \sin(\theta_y) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta_y) & 0 & \cos(\theta_y) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\theta_z) & -\sin(\theta_z) & 0 & 0 \\ 0 & \cos(\theta_x) & -\sin(\theta_x) & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x_r \\ y_r \\ z_r \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} C_x - (x_r + f * u_x) \\ C_y - (y_r + f * u_y) \\ C_z - (z_r + f * u_z) \\ 1 \end{bmatrix} = \begin{bmatrix} T_x \\ T_y \\ T_z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_r \\ y_r \\ z_r \\ 1 \end{bmatrix} = \begin{bmatrix} x'_r \\ y'_r \\ z'_r \\ 1 \end{bmatrix}$$

The above process should happen for both points that have been matched. This should result in 2 homogeneous points that we will call  $[x_0 \ y_0 \ z_0 \ 1]^T$  and  $[x_1 \ y_1 \ z_1 \ 1]^T$ . Each point has a corresponding camera vector, which is already known thanks to the camera coordinates,  $[C_{x0} \ C_{y0} \ C_{z0} \ 1]^T$  and  $[C_{x1} \ C_{y1} \ C_{z1} \ 1]^T$ . From this we can make parametric lines  $L_0$  and  $L_1$  with the parametric variables  $t_0$  and  $t_1$ :

$$L_0 = \begin{bmatrix} x_0 - C_{x0} \\ y_0 - C_{y0} \\ z_0 - C_{z0} \end{bmatrix} \begin{bmatrix} t_0 \\ t_0 \\ t_0 \end{bmatrix} + \begin{bmatrix} C_{x0} \\ C_{y0} \\ C_{z0} \end{bmatrix}$$

$$L_1 = \begin{bmatrix} x_1 - C_{x1} \\ y_1 - C_{y1} \\ z_1 - C_{z1} \end{bmatrix} \begin{bmatrix} t_1 \\ t_1 \\ t_1 \end{bmatrix} + \begin{bmatrix} C_{x1} \\ C_{y1} \\ C_{z1} \end{bmatrix}$$

The Host functions are simple and will not be addressed here, refer to standard CUDA memory management. For the Kernel, we must insure that we have a few global variables about the camera data/parameters available to the GPU. These variables are the focal length  $foc$ , the feild of view  $fov$ , and the resolution of the imager  $res$ . This assumes that the cameras are in the  $xy$  plane, which can only be assumed for this special 2-view case. It may be helpful to read the section on file formats, which is after the main introduction.

In the future, only the first 3 components, the position coordinates, of  $C_0$  and  $C_1$  are used. Vectors  $v_0$  and  $v_1$  are used to parateterizeds lines.



---

**Algorithm 1** Device Kernel

---

```
1: Let:  $dpix = foc * \tan \frac{fov/2}{res/2}$ 
2: Let:  $i = blockIdx.x * blockDim.x + threadIdx.x$ 
3: procedure GENERATELINES(R2POINTS, R3CAMERAS)
4:    $C_0[6] = (R3cameras[0], R3cameras[1], R3cameras[2], R3cameras[3], R3cameras[4], R3cameras[5])$ 
5:    $C_1[6] = (R3cameras[6], R3cameras[7], R3cameras[8], R3cameras[9], R3cameras[10], R3cameras[11])$ 
6:    $x_0 = (R2points[i] - res/2.0)$ 
7:    $y_0 = (-R2points[i + 1]) + res/2.0$ 
8:    $x_1 = (R2points[i + 2] - res/2.0)$ 
9:    $y_1 = (-R2points[i + 3]) + res/2.0$ 
10:   $kp_0 = [x_0, y_0, 0.0]$ 
11:   $kp_1 = [x_1, y_1, 0.0]$ 
12:   $\theta_{x0} = \cos^{-1}(\frac{kp_0 \cdot [1 \ 0 \ 0]^T}{\sqrt{\|kp_0\|}})$ 
13:   $\theta_{x1} = \cos^{-1}(\frac{kp_1 \cdot [1 \ 0 \ 0]^T}{\sqrt{\|kp_1\|}})$ 
14:   $kp_0 = kp_0 \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\frac{\pi}{2}) & -\sin(\frac{\pi}{2}) \\ 0 & \sin(\frac{\pi}{2}) & \cos(\frac{\pi}{2}) \end{bmatrix} \begin{bmatrix} \cos(\theta_{x0} + \frac{\pi}{2}) & -\sin(\theta_{x0} + \frac{\pi}{2}) & 0 \\ \sin(\theta_{x0} + \frac{\pi}{2}) & \cos(\theta_{x0} + \frac{\pi}{2}) & 0 \\ 0 & 0 & 1 \end{bmatrix}$ 
15:   $kp_1 = kp_1 \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\frac{\pi}{2}) & -\sin(\frac{\pi}{2}) \\ 0 & \sin(\frac{\pi}{2}) & \cos(\frac{\pi}{2}) \end{bmatrix} \begin{bmatrix} \cos(\theta_{x1} + \frac{\pi}{2}) & -\sin(\theta_{x1} + \frac{\pi}{2}) & 0 \\ \sin(\theta_{x1} + \frac{\pi}{2}) & \cos(\theta_{x1} + \frac{\pi}{2}) & 0 \\ 0 & 0 & 1 \end{bmatrix}$ 
16:   $kp_0[0] = C_0[0] - (kp_0[0](C_0[3] * foc))$ 
17:   $kp_0[1] = C_0[1] - (kp_0[1](C_0[4] * foc))$ 
18:   $kp_1[0] = C_1[0] - (kp_1[0](C_1[3] * foc))$ 
19:   $kp_1[1] = C_1[1] - (kp_1[1](C_1[4] * foc))$ 
20:   $v_0 = kp_0 - C_0$ 
21:   $v_1 = kp_1 - C_1$ 
```

▷  $kp_0$  now represents the keypoint location in  $\mathbb{R}^3$   
▷  $kp_1$  now represents the keypoint location in  $\mathbb{R}^3$   
▷ line  $L_0$ 's vector  
▷ line  $L_1$ 's vector

---

### 5.3 Minimum Distance Between Skew Lines

Now that we have lines  $L_0$  and  $L_1$ , the challenge is to find the points  $s_0$  and  $s_1$  of closest approach. First, we must test the assumption that our lines are skew, meaning they are not parallel and do not intersect. To start to think of this we take the forms of  $L_0$  and  $L_1$  and simplify them by thinking of them as parametric vectors where  $C_0$  and  $C_1$ , represent the camera position vectors and  $v_0$  and  $v_1$  represent the vector was previously calculated from the subtraction of match coordinates with the camera vector. We make the simple equations:

$$L_0 = v_0 t_0 + C_0 \quad L_1 = v_1 t_1 + C_1$$

To make sure that the lines are not parallel, which is unlikely, we must verify that their cross product is not zero. if  $v_0 \times v_1 = 0$  then we have a degenerate case with infinitely many solutions. As long as we know this is not the case we can proceed. We know that the cross product of the two vectors  $c = v_0 \times v_1$  is perpendicular to the lines  $L_0$  and  $L_1$ . We know that the plane  $P$ , formed by the translation of  $L_1$  along  $c$ , contains  $C_1$ . We also know that the point  $C_1$  is perpendicular to the vector  $n_0 = v_1 \times (v_0 \times v_1)$ . Thus, the intersection of  $L_0$  with  $P$  is also the point,  $s_0$ , that is nearest to  $L_1$ , given by the equation:

$$s_0 = C_0 + \frac{(C_1 - C_0) \cdot n_0}{v_0 \cdot n_0} \cdot v_0$$

This also holds for the second line  $L_1$ , the point  $s_1$ , and vector  $n_1 = v_0 \times (v_1 \times v_0)$ . with the equation:

$$s_1 = C_1 + \frac{(C_0 - C_1) \cdot n_1}{v_1 \cdot n_1} \cdot v_1$$

Now, given to points that represent the closest points of approach, we simply find the midpoint  $m$ :

$$m = \begin{bmatrix} (s_0[x] + s_1[x])/2 \\ (s_0[y] + s_1[y])/2 \\ (s_0[z] + s_1[z])/2 \end{bmatrix}$$

## 5.4 Least Square Approximation for Two Lines

## Chapter 6

# N/Multiview Reprojection

### 6.1 Overview

For the purpose of our software, we expect a set of points  $P_n = (p_x, p_y, p_z)$  and their corresponding orientation as a unit vector  $\hat{P}_n = (\hat{u}_x, \hat{u}_y, \hat{u}_z)$  orientations, at this stage we are exclusively in  $\mathbb{R}^3$

### 6.2 3 by 3 Inversion Method

Caleb made up the name to this method, so don't search it because you probably won't find anything about it.

## Chapter 7

# Reconstruction

### 7.1 Overview

### 7.2 Orienting the Point Cloud

# Chapter 8

## Bundle Adjustment

### 8.1 Introduction

Bundle adjustment is defined as the problem of refining a visual reconstruction of a scene geometry to produce jointly optimal 3D point cloud representation of the scene and viewing parameter (camera pose and calibration) estimates.

### 8.2 Notation

Here we list some notation used in this paper.

$I$	$\{I_1, \dots, I_N\}$ . Input set of $N$ images
$C_i$	camera center location corresponding to $I_i$ (world coordinates)
$c_{foc}$	camera focal length
$c_{fov}$	camera field of view
$K$	intrinsic camera matrix
$R_i$	$i$ -th camera rotation matrix
$M_i$	$i$ -th camera projection matrix
$PC$	$\{P_1, \dots, P_L\}$ . Point cloud set
$P_j$	$\{x_j, y_j, z_j\}$ . $i$ -th computed 3D feature point (world coordinates)
$p_{(i,j)}$	observed 2D image point on image $i$ corresponding to $P_j$ (pixel coordinates)
$s$	state vector
$\delta$	step vector
$f(s)$	cost/error function

### 8.3 Problem Formulation

We are given a set of  $N$  images along with the feature points for each image. As a prerequisite, a set of 3D points,  $PC$ , are estimated for each pixel point match set. It is convenient to define a “track” for each  $P_j \in PC$  as

$$T_j = \{P_j, Q(P_j)\}$$

where  $Q(P_j) = \{p_{(1,j)}, p_{(2,j)}, \dots, p_{(L,j)}\}$  is the set of observed image points that were previously used to predict  $P_j$ , and where  $p_{(i,j)} = null$  whenever no point on  $I_i$  contributed to estimation of  $P_j$  (i.e., when  $P_j$  is not visible on  $I_i$ ).

[[ reword this after writing earlier parts of pipeline to refer to ]]

Then as we previously defined, for a particular image point  $p_{(i,j)}$ , the projection error, or distance between the observed location and the projected image point location  $\bar{p}_{(i,j)} = M_i P_j$ , is denoted

$$d(p_{(i,j)}, \bar{p}_{(i,j)})$$

This distance function depends on all entries in the projection matrix  $M_i$ , namely  $R_i$ ,  $C_i$ ,  $c_{foc}$ , and  $c_{fov}$ , as well as the 3D coordinates of each feature point  $P_j = (x_j, y_j, z_j)$ . Now we organize all of these variables into one state vector  $s$ :

$$s = \left\{ \bigcup_i R_i, \bigcup_i C_i, c_{foc}, c_{fov}, \bigcup_j P_j \right\}$$

with  $i = 1, 2, \dots, N$  and  $j = 1, 2, \dots, L$ . Then our goal is to globally minimize the amount of projection error with respect to  $s$ . We define the cost function

$$f(s) = \sum_{i=1}^N \sum_{j=1}^L d(p_{(i,j)}, \bar{p}_{(i,j)}) = \sum_{i=1}^N \sum_{j=1}^L d(p_{(i,j)}, M_i P_j) \quad (8.1)$$

The goal of bundle adjustment then is to compute the optimal state vector parameters to minimize the cost function:

$$\min_s f(s) \quad (8.2)$$

## 8.4 BA Algorithm

This section explains the bundle adjustment algorithm in detail. First we explain the mathematical solution to the problem expressed in section 3. After that we describe the methods used to reduce computational cost so that the algorithm becomes feasible.

### 8.4.1 Numerical Optimization Estimate

We need to minimize the cost function  $f(s)$  over parameters  $s$  starting from the given initial estimate. Real valued cost functions are too complicated to minimize in closed form, and the parameter space is non linear, so instead we approximate  $f(s)$  by a local linear model - in our case a quadratic Taylor series expansion of  $f$  at the current state  $s$ . For a small step  $\delta$ ,

$$f(s + \delta) \approx f(s) + g(s)^T \delta + \frac{1}{2} \delta^T H(s) \delta \quad (8.3)$$

where  $g \equiv \frac{df}{ds}(s)$  and  $H \equiv \frac{d^2 f}{ds^2}(s)$  are the gradient vector and Hessian matrix of  $f$ , respectively. Assuming that  $H$  is positive definite, the local model is a simple quadratic with a unique global minimum, which can be found explicitly. To do this, we use the Gauss-Newton method to solve for step  $\delta$  in the following linear system.

$$H(s) \delta = -g(s) \quad (8.4)$$

### 8.4.2 Schur Complement

Note that calculating the second derivatives  $H$  and then inverting the matrix to solve (4) by

$$\delta = -H^{-1}g$$

is extremely computationally expensive for complex cost functions such as this  $f$ , so bundle adjustment methods typically estimate the hessian using clever techniques such as exploiting the sparse nature of  $H$ .

## 8.5 Pseudocode

## 8.6 Appendix