**Overview Flowchart**

\

**Person Signature Creation**

**Section 4**

**Triangulation**

**Section 3**

**Person Points Identification**

**(one image)**

**Section 2**

**Camera Calibration**

**Section 1**

**SECTION 1: Camera Calibration**

**SECTION 1: Camera Calibration**

**Rectification Homography Computation**

**Section 1 Step 3**

**Registration**

**Section 1 Step 4**

**Extrinsic Matrix**

**Section 1 Step 2**

**Intrinsic Matrix Calculation**

**Section 1 Step 1**

* This entire section is done in VIPER-REGISTRAR/initialization\_calibration\_script.py

**Step 1: Intrinsic Matrix Calculation**

* Currently using VIPER-REGISTRAR/lib/camera\_calibration/camera\_calibration.py
* Get images of checkerboard, ensure to have images evenly spread out across entire screen
* Pass images along with other argument to function
* Do this separately for both camera (need not be stereo images) and save the outputs to cameras json file and matrices to settings file
* NOTE: Consider detecting images with high reprojection error and removing them, using calib.io software this demonstrated noticeable improvements but was not looked at

**Step 2: Extrinsic Matrix**

* Currently simply hard coded extrinsic matrices known from setup are used
* These are saved to the settings file

**Step 3: Rectification Homography Computation**

* Computed using VIPER-REGISTRAR/lib/camera\_calibration/rectification\_point\_method.py
* Take undistorted stereo images (can be of anything, but not only white wall as noticeable keypoints must be detectable to match)
* Feed images into function and get out two, left and right, rectification homographies
* These are saved to settings file

**Step 4: Registration**

* Uses VIPER-REGISTRAR/python\_registrar/registrar.py and VIPER-REGISTRAR/python\_registrar/client.py
* See previously mentioned initialization\_calibration\_script.py

**SECTION 2: Person Points Identification (one cam)**

**Right Image Rectification**

**Section 2 Step 1**

**Bounding Box Identification**

**Section 2 Step 2**

**Bounding Box Extraction and Scaling**

**Section 2 Step 3**

**Pose Identification**

**Section 2 Step 4**

**Right Body Points Remapping**

**Section 2 Step 5**

**SECTION 2: Person Points Identification (one cam)**

**Step 1: Right Image Rectification:**

* Using cv2.warpPerspective(img, H, (cols,rows)), where img is the image read in with cv2.imread(), H is the rectification homography matrix as numpy array for the given camera, cols is the number of columns in image (ex: 1920) and rows is the number of rows (ex: 1080)

**Step 2: Bounding Box Identification:**

* Uses the run\_yolo from VIPER-RECOGNITION/PERSON\_IDENTIFICATION/YOLOV8/yolov8.py
* Note that the model needs to be specified in the code for now
* Takes in a list of image paths and return a list of list of bounding boxes

**Step 3: Bounding Box Extraction and Scaling:**

* The bounding box plus 5% width and height to give a tolerance on the box is cropped out of the image. The location it is cropped from is retained for later. The box is passed to the next step where the scaling is done along with the pose detection

**Step 4: Pose Identification:**

* Uses run\_thunder function from VIPER-RECOGNITION/PERSON\_IDENTIFICATION/THUNDER/thunder.py
* The function first scales the image (should be already cropped using bounding box from earlier step) to be of required size for thunder (256,256). The scaling factors are kept.
* Pose points are gotten using the thunder model

**Step 5: Right Body Points Remapping:**

* The points obtained from the thunder model in the previous step are then rescaled to the original fed in box size.
* The points are then sent back to the script that called run\_thunder where the points are translated according to where the bounding box was cut from
* This gives the pose points in the rectified image
* These points are then converted back to the original image by applying the inverse of the rectification homography

**SECTION 3: Triangulation**

**Left Image Rectification**

**Section 3 Step 1**

**Point by Point Line Search**

**Section 3 Step 2**

**Left Body Points Remapping**

**Section 3 Step 3**

**Triangulation**

**Section 3 Step 4**

**SECTION 3: Triangulation**

**Step 1: Left Image Rectification:**

* Using cv2.warpPerspective(img, H, (cols,rows)), where img is the image read in with cv2.imread(), H is the rectification homography matrix as numpy array for the given camera,

**Step 2: Point by Point Line Search:**

* Uses sweep\_line\_block function from VIPER-RECOGNITION/PERSON\_IDENTIFICATION/LINE\_MATCHING/line\_matching\_4.py
* This function uses ssim function from VIPER-RECOGNITION/PERSON\_IDENTIFICATION/LINE\_MATCHING/my\_ssim.py
* The sweep line block function takes in rectified source and target images, the pixel coordinate in the source image you wish to locate in the target image and the width and height box radius (half total size) to use in ssim when comparing boxes for similarity

**Step 3: Left Body Points Remapping:**

* The identified body points are then remapped to the original (non rectified image) using, as for the right point, the rectification homography. (Note: the rectification homography is specific to each camera)

**Step 4: Triangulation:**

* Uses the triangulate\_points function from VIPER-RECOGNITION/PERSON\_IDENTIFICATION/TRIANGULATION/triangulate\_points.py
* The function uses the intrinsic and extrinsic matrices as well as the pixel coordinates in the left and right images to compute the 3D world coordinate point.

**SECTION 4: Person Signature Creation**

**Body Measurement Computation**

**Section 4 Step 1**

**Temporary List Search**

**Section 4 Step 2**

**Temporary List Addition**

**Section 4 Step 3**

**Global List Search**

**Section 4 Step 4**

**Global Signature Addition**

**Section 4 Step 5**

**SECTION 4: Person Signature Creation**

**Step 1: Body Measurement Computation:**

* For each measurement of interest, the length is computed by taking the norm of the difference of the two points in the 3D world space
* These measurements, along with information on the frame they came from, etc. are put together to create an object of class Frame\_data.
* The Frame\_data class comes from VIPER-RECOGNITION/DATABASE\_SORTING/store\_temp\_people.py

**Steps 2-5:**

* The sort\_frame\_data function from VIPER-RECOGNITION/DATABASE\_SORTING/store\_temp\_people.py is run on the Frame\_data object and the temporary list, (The global list is a global variable and is also accessed, the database structure in C++ will need to be modified, most likely will be stored in a fila and accessed there when needed).
* The above function will compare all people in the Frame\_data object to the people in the temporary list, it will then either add them to the best match or create a new person if there is no match.
* Once a person in the temporary list has enough entries (reaches maturity) they are compared to the global list where they are either merged with their best match or a new person is created