Algorithm 1: Capturing synchronous images at precise timestamp

```
Initialize camera 1
Initialize camera 2
Get the frame rate of camera 1 and camera 2
Calculate the time interval between frames
Get the current time
Calculate the timestamps for the frames based on the time interval and
  the start time
for each timestamp do
    Set the positions of the video streams to the corresponding
      timestamp
    Read frames from camera 1 and camera 2
    if frames were successfully captured then
        Save the frames with the corresponding timestamp
    end
    else
        Break out of the loop
    end
end
Release camera 1
Release camera 2
```

Algorithm 2: Masker detection: Extracting image coordinates

```
input synchronous image: .pngfile
hsv.img ← convert img to HSV color space
upper_HSV ← [130, 255, 255]
lower_HSV ← [90, 70, 0]
mask ← apply color mask to hsv.img with upper_HSV and lower_HSV
kernel ← create 5 × 5 kernel with all ones
dilate ← apply dilation operation to mask with kernel
closing ← apply morphological closing operation to dilate with kernel
contours ← find contours in closing
centers := empty list [ ]
for contour in contours do
    area ← calculate area of contour
    if area > 1000 then
        draw contour on mask with [0, 255, 0]
        M ← calculate moments of contour
        cx ← intM['m10']/M['m00']
        cy ← intM['m01']/M['m00']
        if (cx − cy) < 500 then
            append (cx, cy) to centers
            draw circle on img at center (cx, cy) with color [0, 0, 255]
output Marker detection image coordinates: C1,C2
```

Algorithm 3: Extracting object points : left-right image triangulation

left_img_pts ← [C1x, C1y], [C2x, C2y]
right_img_pts ← [C1x, C1y], [C2x, C2y]
Triangulate the object points in 3D space
proj_matrix_left = $\begin{bmatrix} fx & 0 & cx & 0 \\ 0 & fy & cy & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
proj_matrix_right = $\begin{bmatrix} fx & 0 & cx & tx \\ 0 & fy & cy & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
obj_pts_3d_homogeneous = triangulate[proj_matrix_left, proj_matrix_right, left_img_pts, right_img_pts]
$translation\_left, rotation\_left = decomposeProjectionMatrix(proj_{m}atrix_{l}eft)$
M = concatenate((rotation_left, translation_left.reshape(-1, 1)), axis=1)
obj_pts_3d_world_homogeneous = dot(M.T, obj_pts_3d_homogeneous) + translation_left.reshape(-1, 1)
obj_pts_3d_world = obj_pts_3d_world_homogeneous[:3] / obj_pts_3d_world_homogeneous[3]
output$Objectcoordinatesw.r.t.worldC.O.S$: *X, Y, Z*

Algorithm 4: Estimate_F_matrix

Input: img1_pts, img2_pts
Output: F
normalize points
$x1$ = img1_pts[:,0]
$y1$ = img1_pts[:,1]
$x1dash$ = img2_pts[:,0]
$y1dash$ = img2_pts[:,1]
A = np.zeros((len(x1),9))
for *i in range(len(x1))* **do**
    $A[i]$ = np.array([$x1dash[i] * x1[i]$, $x1dash[i] * y1[i]$, $x1dash[i]$, $y1dash[i] * x1[i]$, $y1dash[i] * y1[i]$, $y1dash[i]$, $x1[i]$, $y1[i]$, 1])
end
U, E, V = SVD(A)
$F\_est = V[-1, :]$
$F\_est = F\_est.reshape(3, 3)$
ua, sa, va = SVD(F_est)
sa = diag(sa)
$sa[2, 2] = 0$
$F = \text{dot}(ua, \text{dot}(sa, va))$
F = F / F[2,2]

Algorithm 5: Estimate E_Matrix from F_Matrix and K

Input: K, F

Output: E

$E_{est} \leftarrow K^T \cdot F \cdot K$

$U, S, V \leftarrow \text{SVD}(E_{est})$

$S \leftarrow \text{diag}(S)$

$S_{0,0}, S_{1,1}, S_{2,2} \leftarrow 1, 1, 0$

$E \leftarrow U \cdot S \cdot V$

Algorithm 6: Estimating the camera pose

Input: E

Output: R, T

$U, S, V \leftarrow \text{SVD}(E)$

$W \leftarrow \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

$R_1 \leftarrow U \cdot W \cdot V$

$R_2 \leftarrow U \cdot W \cdot V$

$R_3 \leftarrow U \cdot W^T \cdot V$

$R_4 \leftarrow U \cdot W^T \cdot V$

$T_1 \leftarrow U[:, 2]$

$T_2 \leftarrow -U[:, 2]$

$T_3 \leftarrow U[:, 2]$

$T_4 \leftarrow -U[:, 2]$

$R \leftarrow [R_1, R_2, R_3, R_4]$

$T \leftarrow [T_1, T_2, T_3, T_4]$

for $i \leftarrow 0$ **to** 3 **do**

  if $\det(R[i]) < 0$ **then**

    $R[i] \leftarrow -R[i]$

    $T[i] \leftarrow -T[i]$

Algorithm 7: point_triangulation(k, $pt1$, $pt2$, $R1$, $T1$, $R2$, $T2$)

Input: Camera intrinsics matrix k, image points $pt1$ and $pt2$, rotation matrices $R1$, $R2$, and translation vectors $T1$, $T2$

Output: 3D points of the object

Initialize an empty list $points\_3d$
Create a 3x3 identity matrix I
Reshape $T1$ and $T2$ to a 3x1 matrix
Calculate projection matrices $P1$ and $P2$ using k, $R1$, $T1$, and $R2$, $T2$ respectively
Create a homogeneous coordinate system for image points xy and xy_cap by concatenating ones to $pt1$ and $pt2$ matrices
for i *in range* $(0, length(xy))$ **do**
    Initialize an empty list A
    $x = xy[i][0]$, $y = xy[i][1]$,
    $x\_cap = xy\_cap[i][0]$, $y\_cap = xy\_cap[i][1]$
    Append $(y * p3 - p2)$ to A
    Append $(x * p3 - p1)$ to A
    Append $(y\_cap * p3\_cap - p2\_cap)$ to A
    Append $(x\_cap * p3\_cap - p1\_cap)$ to A
    Create a 4x4 array A from list A
    Compute the Singular Value Decomposition (SVD) of A to obtain u, s, and v
    Extract the last row of v to obtain $x\_$
    Normalize $x\_$ by dividing by its last element
    Append $x\_$ to $points\_3d$
end
Return the $points\_3d$ array

Algorithm 8: linear_triangulation(R_Set, T_Set, $pt1$, $pt2$, k)

Input: Rotation matrices R_Set, translation vectors T_Set, image points $pt1$ and $pt2$, camera intrinsics matrix k

Output: 3D point set of the object

Create a 3x3 identity matrix $R1\_$ and a 3x1 zero matrix $T1\_$
Initialize an empty list $points\_3d\_set$
for i *in range* $(0, length(R\_Set))$ **do**
    Compute the 3D points of the object using $R\_Set[i]$, $T\_Set[i]$, $pt1$, $pt2$, k and $R1\_$, $T1\_$ as inputs and append the points to $points\_3d\_set$
end
Return the $points\_3d\_set$ array

Algorithm 9: Non-linear triangulation

Input: Rotation matrices R_1, R_2, translation vectors T_1, T_2, 2D image points $pt1$, $pt2$, 3D object points X, camera intrinsic matrix K, and number of iterations k

Output: Reconstructed 3D object points X

$I \leftarrow$ identity matrix of size 3×3
$P_1 \leftarrow K \cdot [R_1| - T_1]$
$P_2 \leftarrow K \cdot [R_2| - T_2]$
$points3D\_new\_set \leftarrow$ empty list
for $i \leftarrow 1$ **to** $len(X)$ **do**
    $opt \leftarrow$ least squares optimization of $loss$ function with initial guess $X[i]$ and arguments $pt1[i]$, $pt2[i]$, P_1, P_2
    $points3D\_new \leftarrow$ optimized 3D point
    append $points3D\_new$ to $points3D\_new\_set$
end
return $points3D\_new\_set$

Algorithm 10: Calculate the mean error of the 3D points

Input: $R1, T1, R2, T2, pt1, pt2, X, k$

Output: e

$R1 \leftarrow \text{reshape}(R1, (3, 3))$
$T1 \leftarrow \text{reshape}(T1, (3, 1))$
$R2 \leftarrow \text{reshape}(R2, (3, 3))$
$T2 \leftarrow \text{reshape}(T2, (3, 1))$
$I \leftarrow \text{identity}(3)$
Calculate projection matrices
$P1 \leftarrow k \times R1 \times \text{hstack}(I, -T1)$
$P2 \leftarrow k \times R2 \times \text{hstack}(I, -T2)$
$e \leftarrow []$
for $i \leftarrow 1$ *to* $len(X)$ **do**
$error \leftarrow \text{loss}(X[i], pt1[i], pt2[i], P1, P2)$
$e.\text{append}(error)$
return $\text{mean}(e)$

Algorithm 11: Reprojection error loss function

Input : 3D point X, image point (u_1, v_1) in camera 1, image point (u_2, v_2) in camera 2, projection matrices P_1 and P_2

Output: Reprojection error

`#Reshape projection matrices to` 3×4

$p_{11}, p_{12}, p_{13} \leftarrow P_1$

$p_{21}, p_{22}, p_{23} \leftarrow P_2$

$p_{11}, p_{12}, p_{13} \leftarrow \text{reshape}(p_{11}, p_{12}, p_{13})$

$p_{21}, p_{22}, p_{23} \leftarrow \text{reshape}(p_{21}, p_{22}, p_{23})$

`#Calculate the image points in camera 1`

$u_1' \leftarrow \frac{p_{11}X}{p_{13}X}$

$v_1' \leftarrow \frac{p_{12}X}{p_{13}X}$

`#Calculate the image points in camera 2`

$u_2' \leftarrow \frac{p_{21}X}{p_{23}X}$

$v_2' \leftarrow \frac{p_{22}X}{p_{23}X}$

`#Calculate the reprojection error`

$\mathbf{error}_1,\mathbf{e1} = \|(u_1 - u_1')\|^2 + \|(v_1 - v1')\|^2$

$\mathbf{error}_2,\mathbf{e2} = \|(u_2 - u_2')\|^2 + \|(v_2 - v2')\|^2$

$total\_error \leftarrow error_1 + error_2$

$\mathbf{lossFunc}, error\_function = error\_mat.append[(total\_error)/(len(imagepoint)]$

$$error\_average = \frac{\sum_{i=0}^{n} \sum_{j=0}^{m} error\_mat[i][j]}{(len(imagepoint) * X)}$$

$error\_reprojection \leftarrow \sqrt{error\_average}$

return $error\_reprojection$

Algorithm 12: Bundle Adjustment : Levenberg-Marquardt Optimization

Input : pose_set, X_world_all, map_2d_3d, K

Output: pose_set_opt, X_world_all_opt

n_cam ← number of cameras

n_3d ← number of 3D points

indices ← list of indices of 3D points

pts_2d ← 2D points of all cameras

indices_cam ← list of camera indices

*x*0 ← initial estimate of parameters

A ← sparse matrix with sparsity pattern defined by *indices* and *indices_cam*

result ← least_squares(fun=loss, x0=x0, jac_sparsity=A, verbose=2, x_scale='jac', ftol=1e-4, method='trf', args=(n_cam, n_3d, indices, pts_2d, indices_cam, K))

param_cam ← camera parameters from *result*

X_world_all_opt ← optimized 3D points from *result*

pose_set_opt ← dictionary of optimized camera poses

for *each cp in param_cam* **do**

- R ← rotation matrix from quaternion in $cp[:4]$
- C ← translation vector from $cp[4:]$
- append (R, C) to *pose_set_opt*

return *pose_set_opt*, *X_world_all_opt*

Algorithm 13: Reprojection points and Error

Input: Matrices A, kc, all_RT, $all\_image\_corners$ and $world\_corners$

Output: $all\_reprojected\_points$, $error\_reprojection$ from the triangulation image points

error_mat = [ ]
all_reprojected_points = [ ]
for *i, image_ corners* ***in*** *enumerate(all_ image_ corners)* **do**
  $RT \leftarrow all\_RT[i]$
  $\mathrm{RT3} \leftarrow \left[\mathrm{RT}{:},0\ \mathrm{RT}{:},1\ \mathrm{RT}{:},3\right].\mathrm{reshape}(3,3)$
  $RT3 \leftarrow RT3^T$
  $\mathrm{ART3} = A \cdot \mathrm{RT3}$
  $\mathrm{image}_total\_error = 0$
  reprojected_points = [ ]
  for j *in range*$(world_corners.shape[0])$ **do**
    world_ point_ 2d = world_ corners[j]
    world_ point_ 3d_ homo = ([world_ point_ 2d[0], world_ point_ 2d[1], 0, 1]).reshape(4,1)
    $XYZ = RT \cdot world\_point\_3d\_homo$
    $x = \frac{XYZ[0]}{XYZ[2]}$
    $y = \frac{XYZ[1]}{XYZ[2]}$
    $radiusofdistortion, r \leftarrow \sqrt{x^2 + y^2}$
    observed image co-ordinates
    $mij \leftarrow image\_corners[j]$
    $mij \leftarrow np.array([mij[0], mij[1], 1], dtype='float').reshape(3, 1)$
    projected image co-ordinates
    $uvw = ART3 \cdot world\_point\_2d\_homo$
    $u = \frac{uvw[0]}{uvw[2]}$
    $v = \frac{uvw[1]}{uvw[2]}$
    $u\_dash \leftarrow u + (u - u_0) \times (k_1 \times r^2 + k_2 \times r^4)$
    $v\_dash \leftarrow v + (v - v_0) \times (k_1 \times r^2 + k_2 \times r^4)$
    reprojected_ points.append([u_ dash, v_ dash])
    mij_ dash = ([u_ dash, v_ dash, 1]
    $\boldsymbol{error}, e = |mij - mij\_dash|^2$
    image_ total_ error = image_ total_ error + e
  end
  all_ reprojected_ points.append(reprojected_ points)
  error_ mat.append(image_ total_ error)
  $\boldsymbol{lossFunc}, error\_function = error\_mat.append[(image\_total\_error)/(len(all\_image\_corners)]$
end

$$error\_average = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} error\_mat[i][j]}{(len(all\_image\_corners) * world\_corners.shape[0])}$$

$error\_reprojection \leftarrow \sqrt{error\_average}$

Algorithm 14: Levenberg-Marquardt Optimization

Input:

A_{init}, kc_{init}, all_RT_{init}, $all\_image\_corners$, $world\_corners$

Output:

A_{new}, kc_{new}

$x0 \leftarrow extractParamFromA(A_{init}, kc_{init})$

res$\leftarrow$ $scipy.optimize.least\_squares(fun = lossFunc, x0 = x0, method = "lm", args = [all\_RT_{init}, all\_image\_corners, world\_corners])$

$x1 \leftarrow res.x$

$AK \leftarrow retrieveA(x1)$

$A_{new} \leftarrow AK[0]$

$kc_{new} \leftarrow AK[1]$