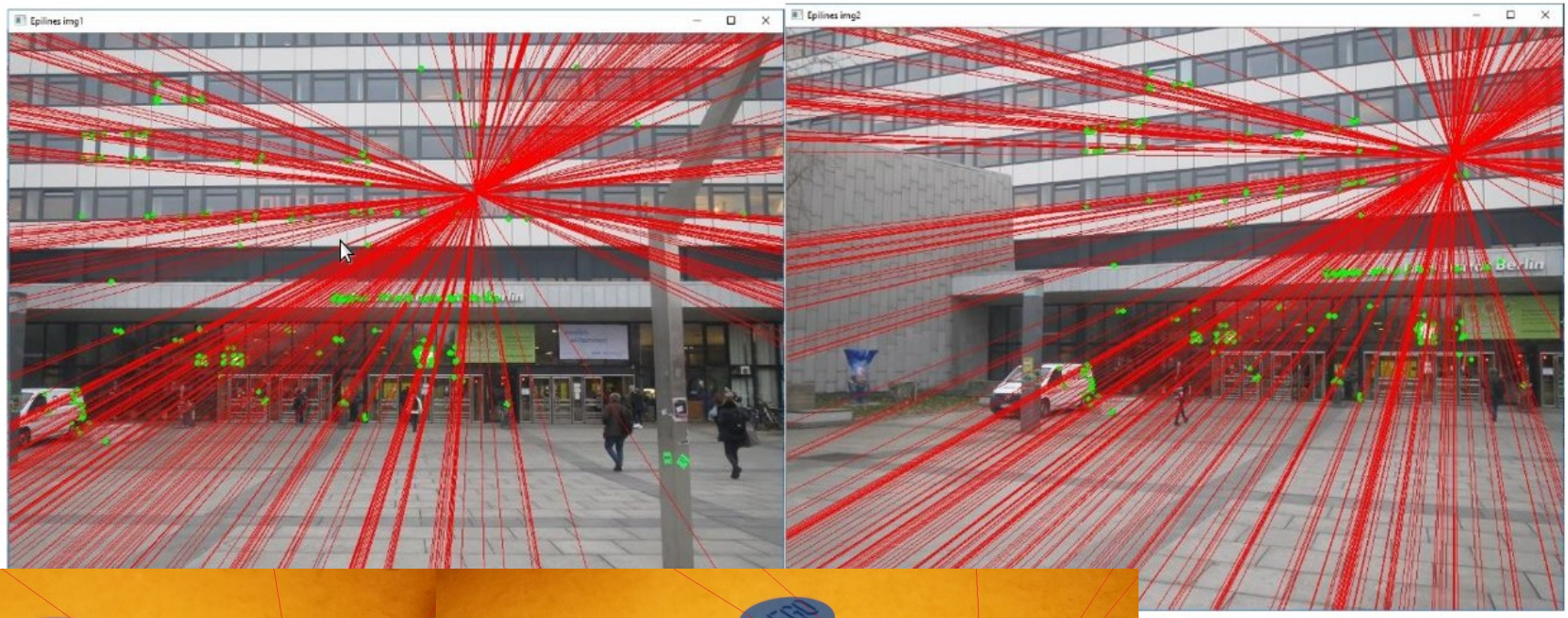
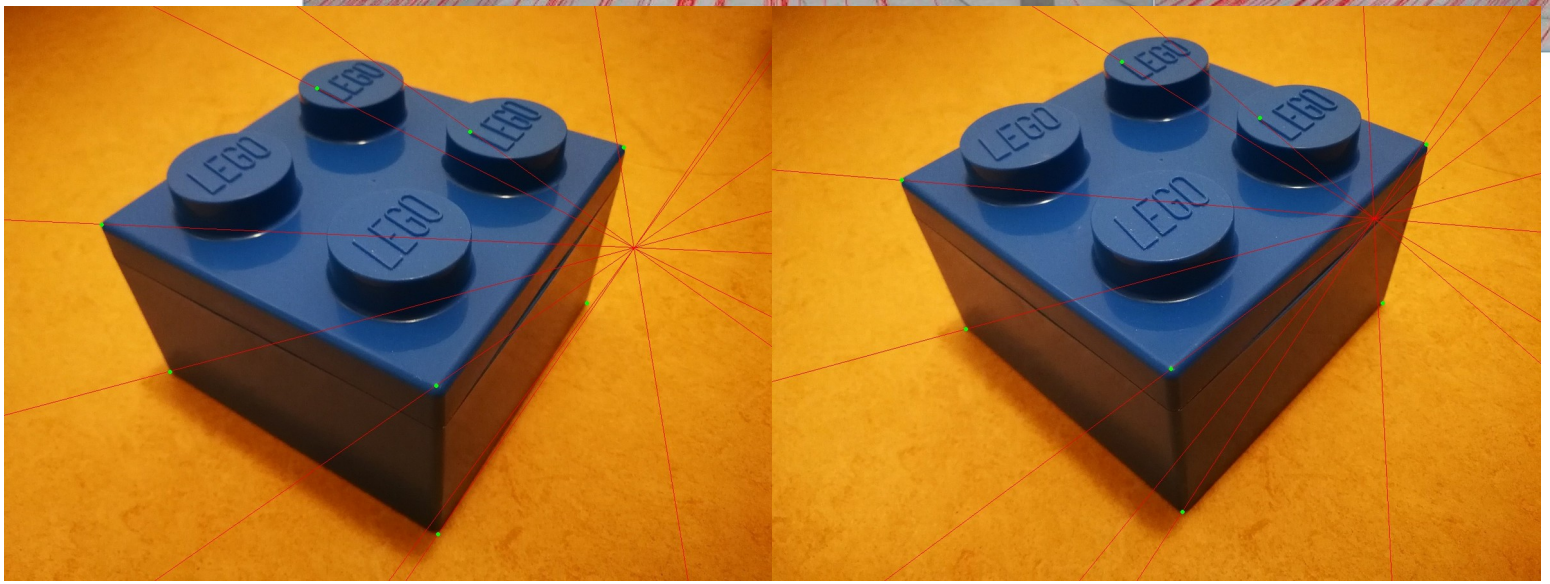
Photogrammetric Computer Vision

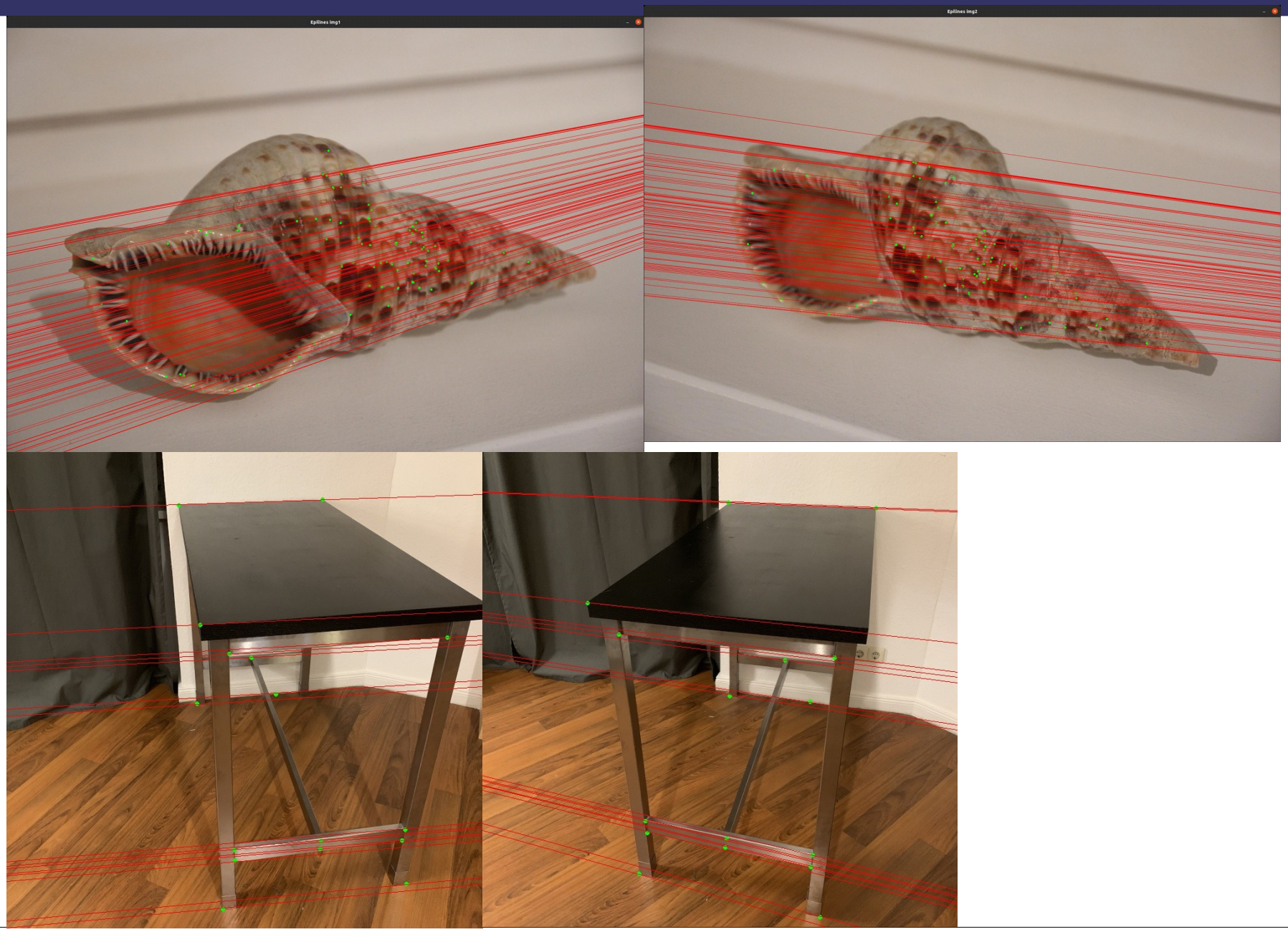
Berlin University of Technology (TUB),
Computer Vision and Remote Sensing Group
Berlin, Germany

Last Exercise





Last Exercise



SFM Pipeline

Image Acquisition

SFM Pipeline

Image Acquisition

Keypoint Detection

Keypoint Detection

SFM Pipeline

Image Acquisition

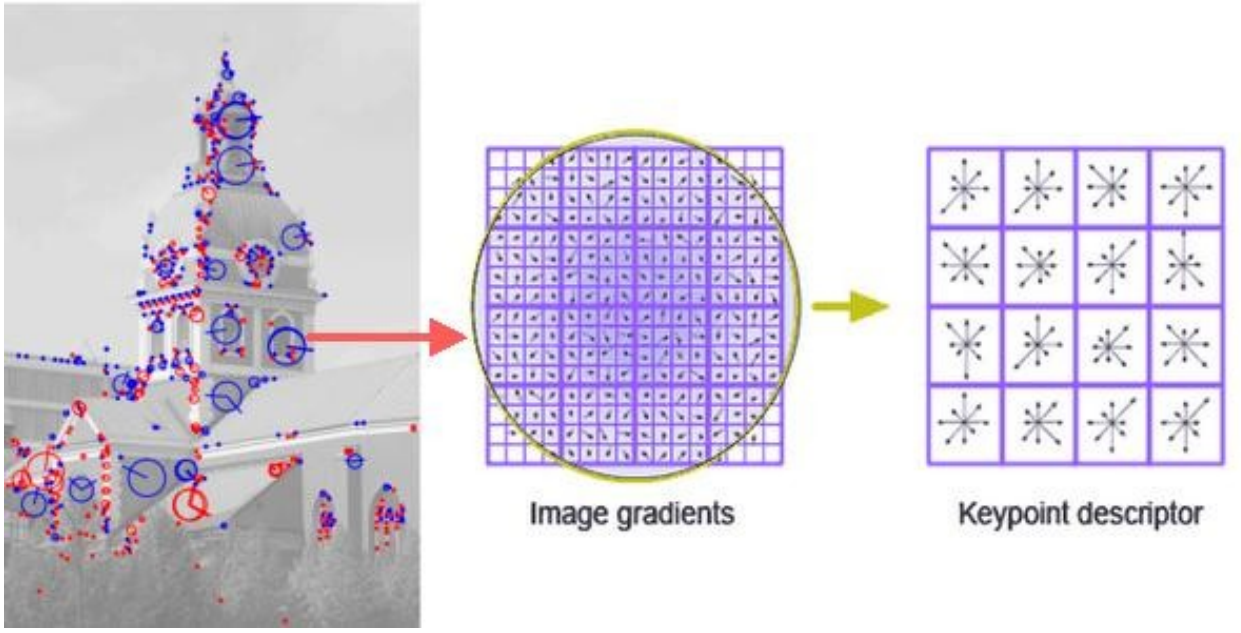
Keypoint Detection

SFM Pipeline

Image Acquisition

Keypoint Detection

Keypoint Description

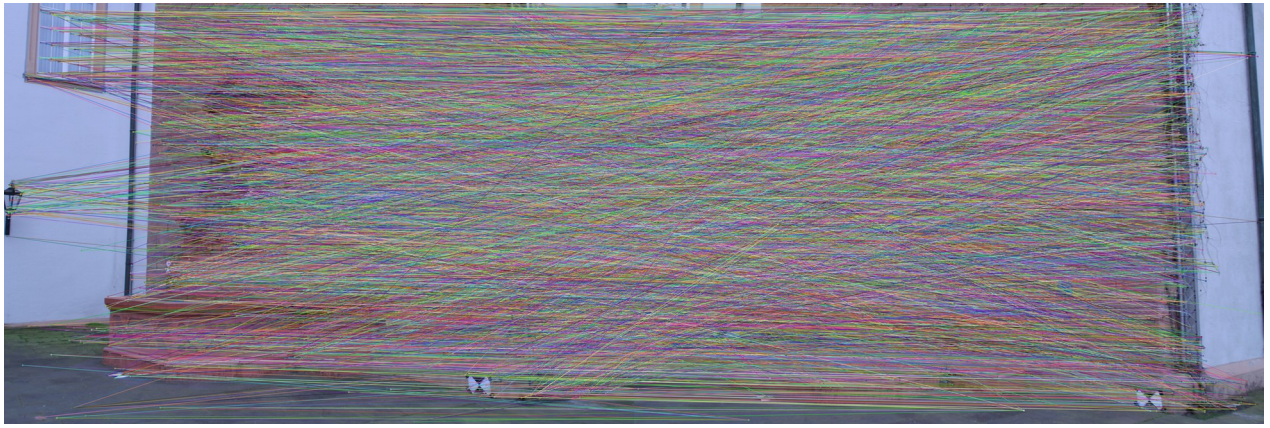


SIFT Detection
→ Position
→ Scale
→ Orientation

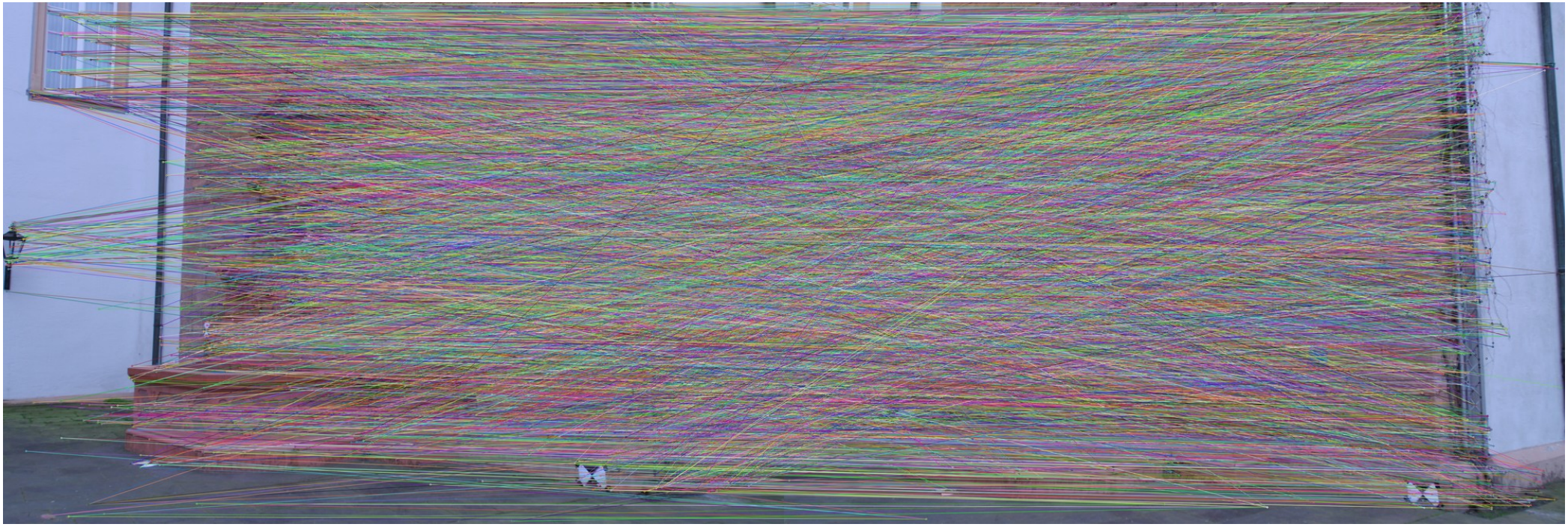
SIFT Description
→ gradient histogram
→ 128 dimensional vector

SFM Pipeline

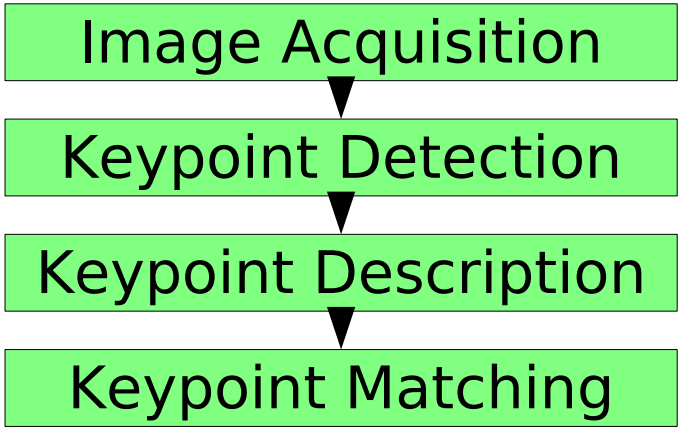
- Image Acquisition
- Keypoint Detection
- Keypoint Description
- Keypoint Matching



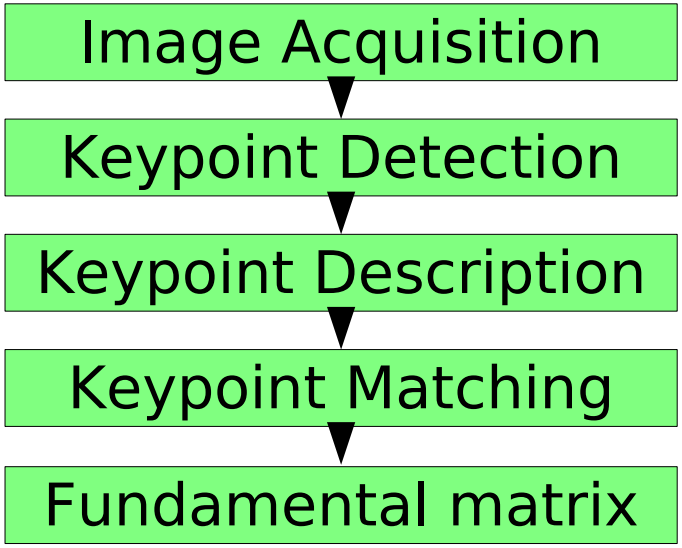
Keypoint Matching



SFM Pipeline



SFM Pipeline



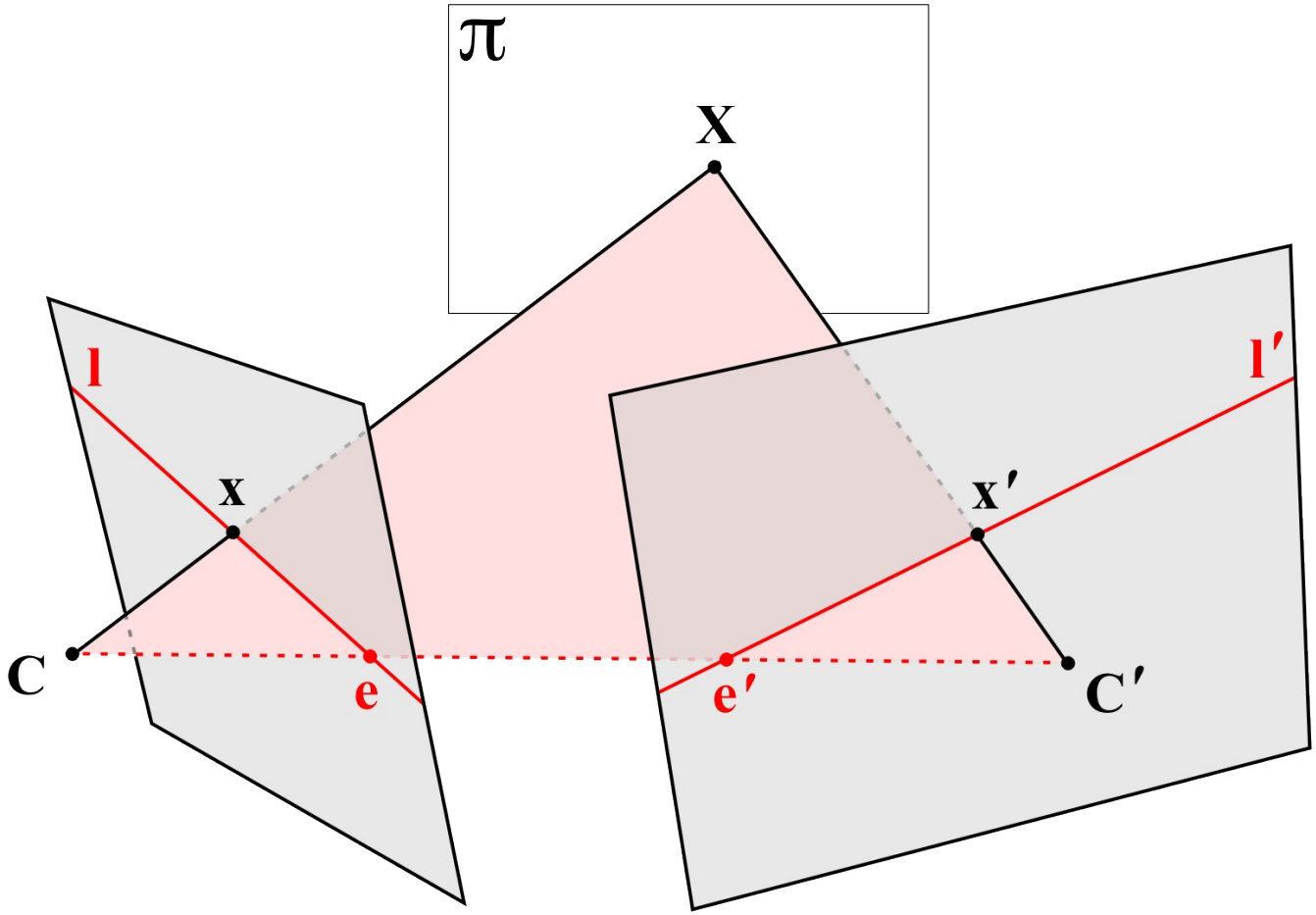
Fundamental matrix

$$x' = H_\pi x$$

$$\begin{aligned} l' &= e' \times x' \\ &= [e']_\times x' \\ &= [e']_\times H_\pi x \\ &= Fx \end{aligned}$$

$$\begin{aligned} 0 &= x'^T \cdot l' \\ &= x'^T F x \end{aligned}$$

Epipolar constraint



Fundamental matrix

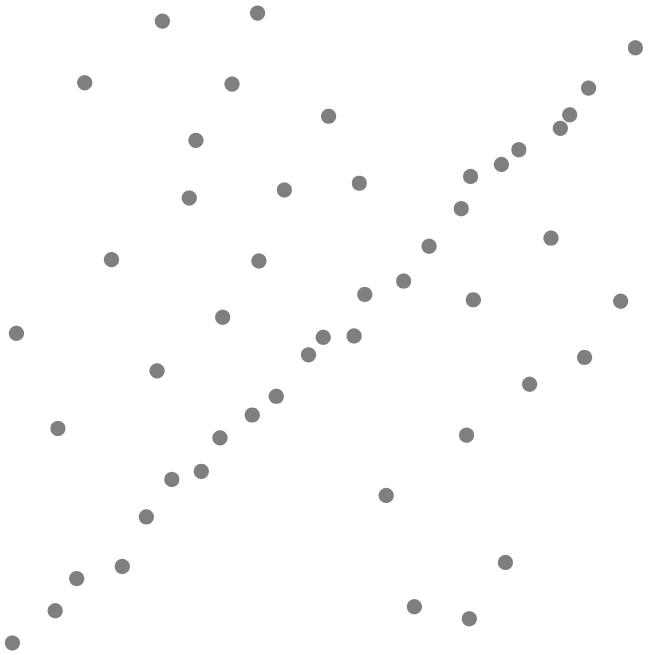
Usually first via RANSAC and a fast direct method

→ Select random minimal set of data points and estimate model
  - e.g. minimal 7-point or normalized 8-point algorithm

→ Evaluate model
  - e.g. Sampson error

→ Iterate and keep model with smallest error

→ Data points with small error to selected model are inlier



Fundamental matrix

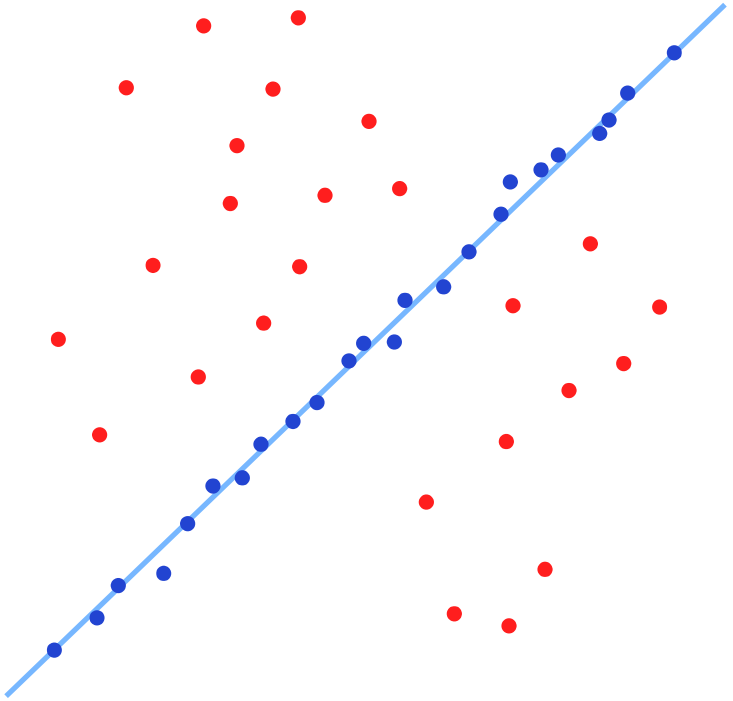
Usually first via RANSAC and a fast direct method

→ Select random minimal set of data points and estimate model
  - e.g. minimal 7-point or normalized 8-point algorithm

→ Evaluate model
  - e.g. Sampson error

→ Iterate and keep model with smallest error

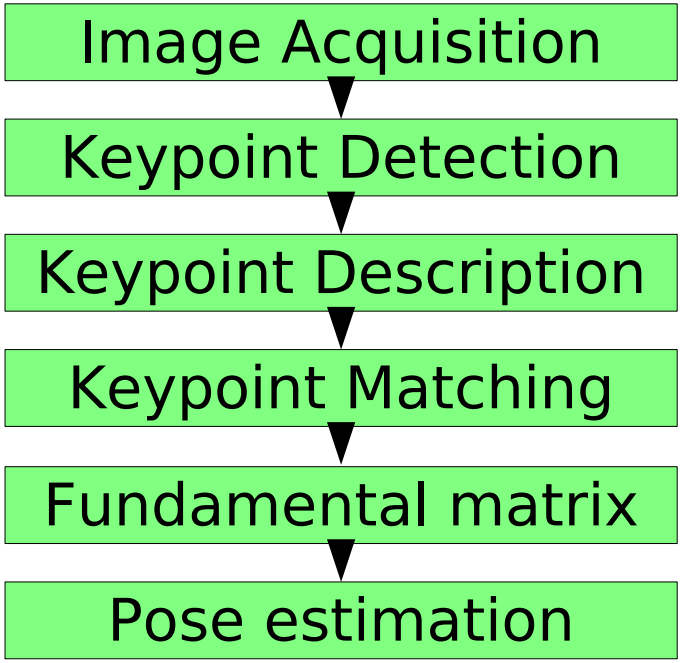
→ Data points with small error to selected model are inlier



SFM Pipeline

- Image Acquisition
- Keypoint Detection
- Keypoint Description
- Keypoint Matching
- Fundamental matrix

SFM Pipeline



Pose estimation

$$E = K_2^T \cdot F \cdot K_1$$

Or: “Calibrate” point pairs by multiplying them with K^{-1} and compute F on those.

Pose estimation

$$E = K_2^T \cdot F \cdot K_1$$

$$SVD(E) = UDV^T \quad det(U) > 0 \wedge det(V) > 0$$

$$t = U_3$$

$$R = UWV^T$$

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

Pose estimation

$$E = K_2^T \cdot F \cdot K_1$$

$$SVD(E) = UDV^T$$

$$t = U_3$$

$$R = UWV^T$$

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

$$det(U) > 0 \wedge det(V) > 0$$

Enforce by negating last rows of U and V if necessary.

$$E = R''_{abs}[t]_\times R'_{abs} = [t]_\times R'_{rel}$$

Pose estimation

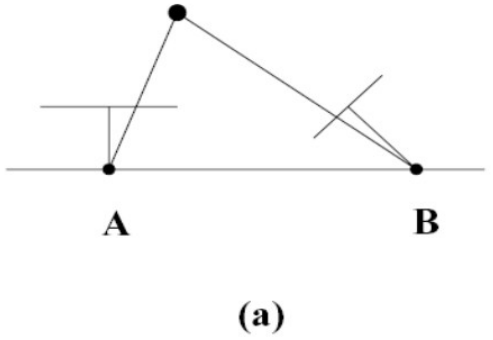
$$E = K_2^T \cdot F \cdot K_1$$

$$SVD(E) = UDV^T$$

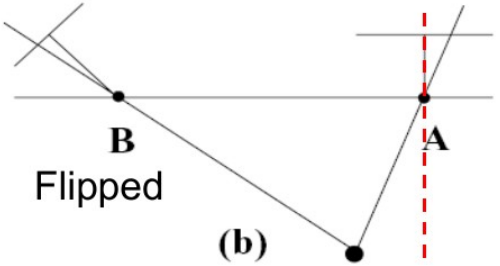
$$t = U_3$$

$$R = UWV^T$$

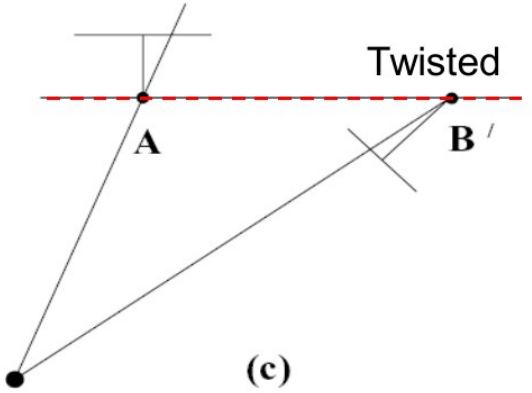
$$W = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



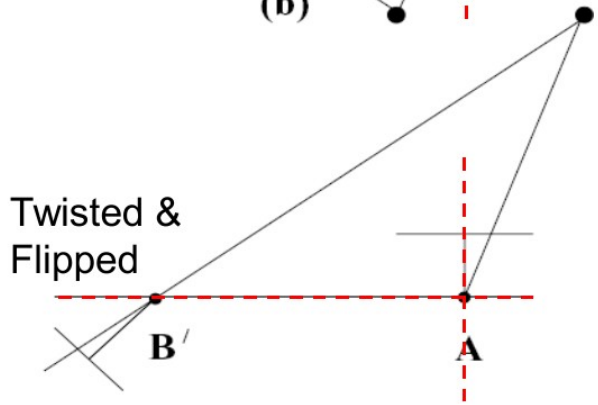
(a)



(b)



(c)

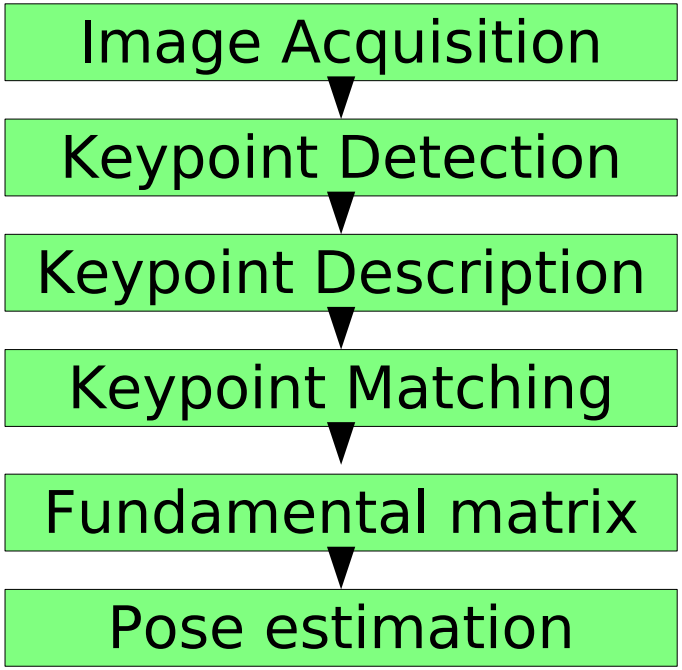


(d)

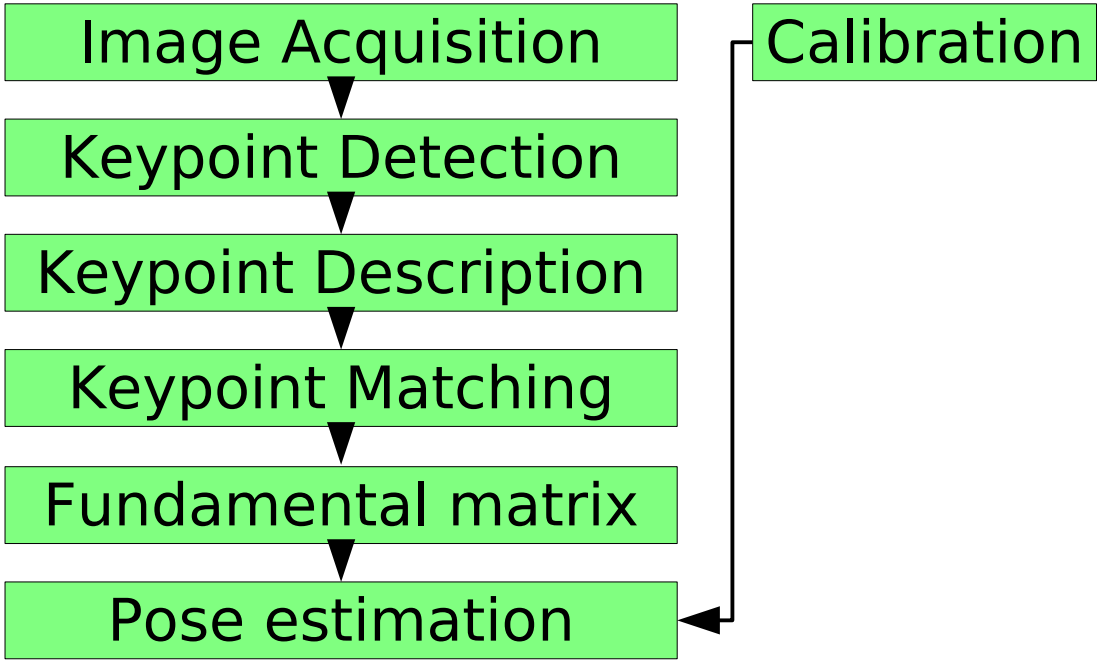
The results are algebraically correct also with -t and W^T.

Try all 4 possibilities, choose the one with most valid points.

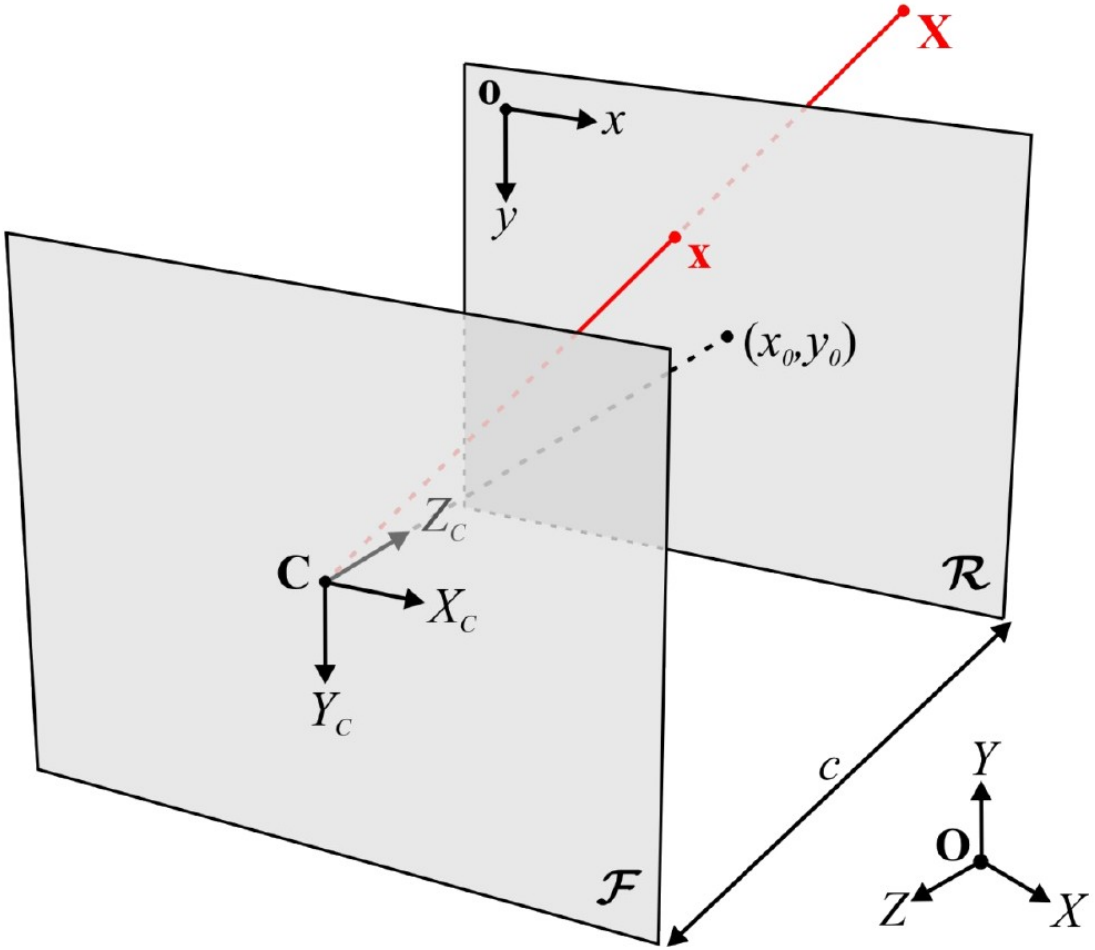
SFM Pipeline



SFM Pipeline



Calibration



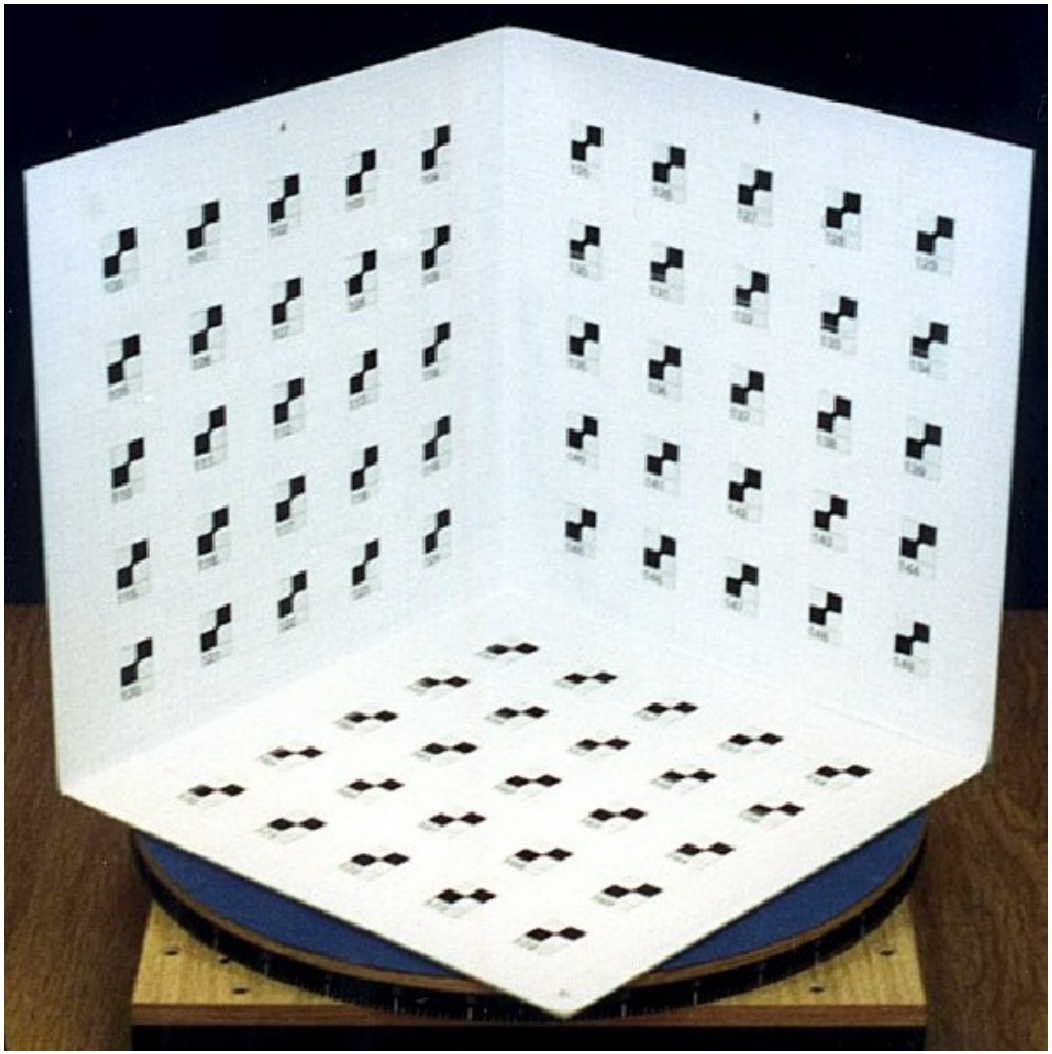
The three-dimensional reconstruction of objects from images requires that the **interior** and **exterior** orientation of the cameras are known.

$$P = K \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} R & t \\ 0^T & 1 \end{bmatrix} = K R [\, I \,|\, -C \,]$$

- Principal distance c
- Principle point (x_0, y_0)
- Aspect ratio γ
- Shearing s

- Projection center $(X_0,Y_0,Z_0)^T$
- Rotation angles ω, ϕ, κ

Calibration



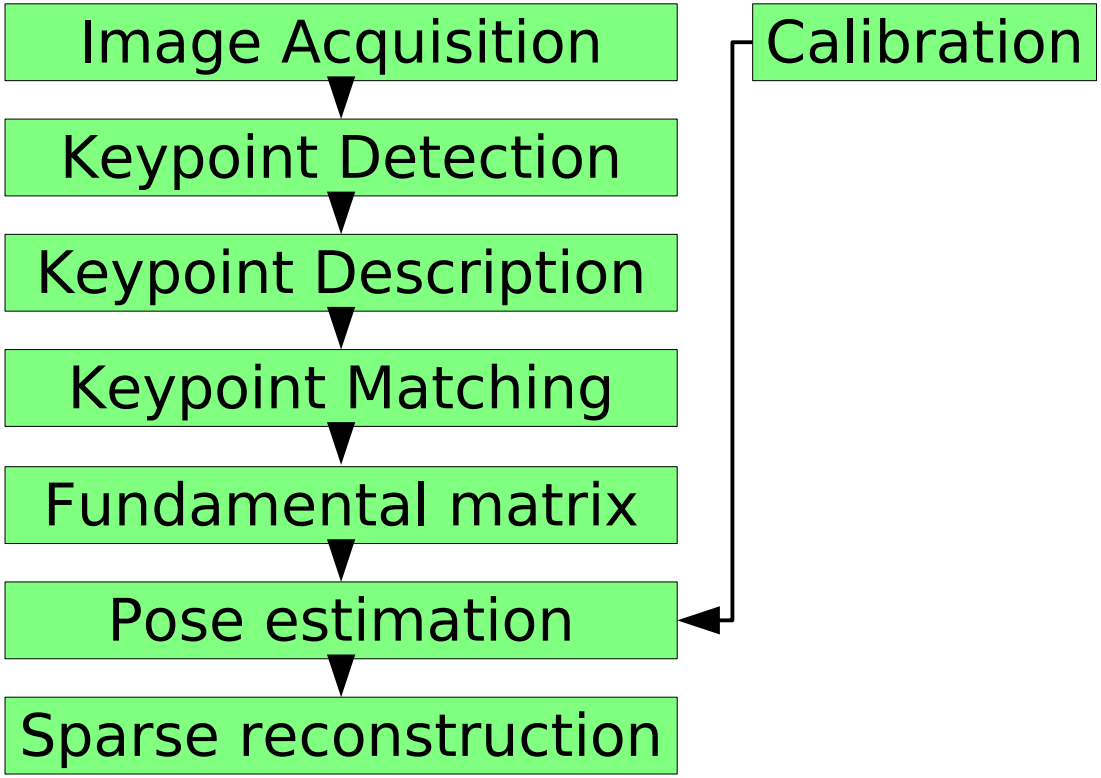
The three-dimensional reconstruction of objects from images requires that the **interior** and **exterior** orientation of the cameras are known.

$$P = K \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} R & t \\ \mathbf{0}^T & 1 \end{bmatrix} = K R \left[I \mid -C \right]$$

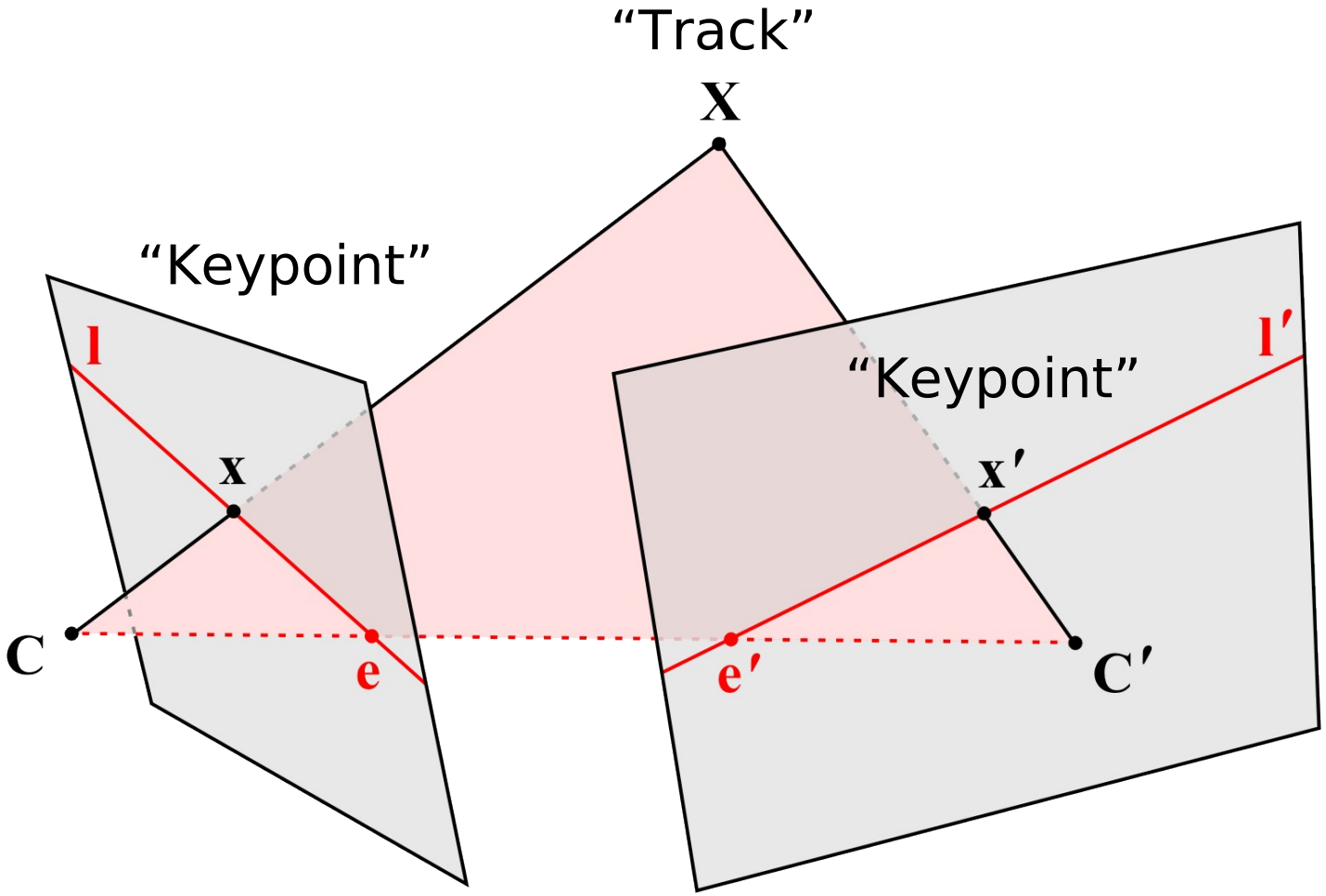
- Principal distance c
- Principle point (x_0, y_0)
- Aspect ratio γ
- Shearing s

- Projection center $(X_0, Y_0, Z_0)^T$
- Rotation angles ω, ϕ, κ

SFM Pipeline



Sparse reconstruction

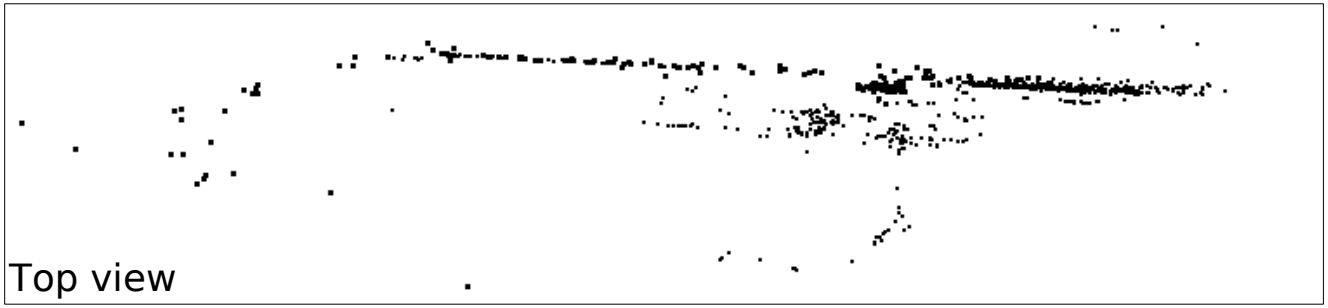


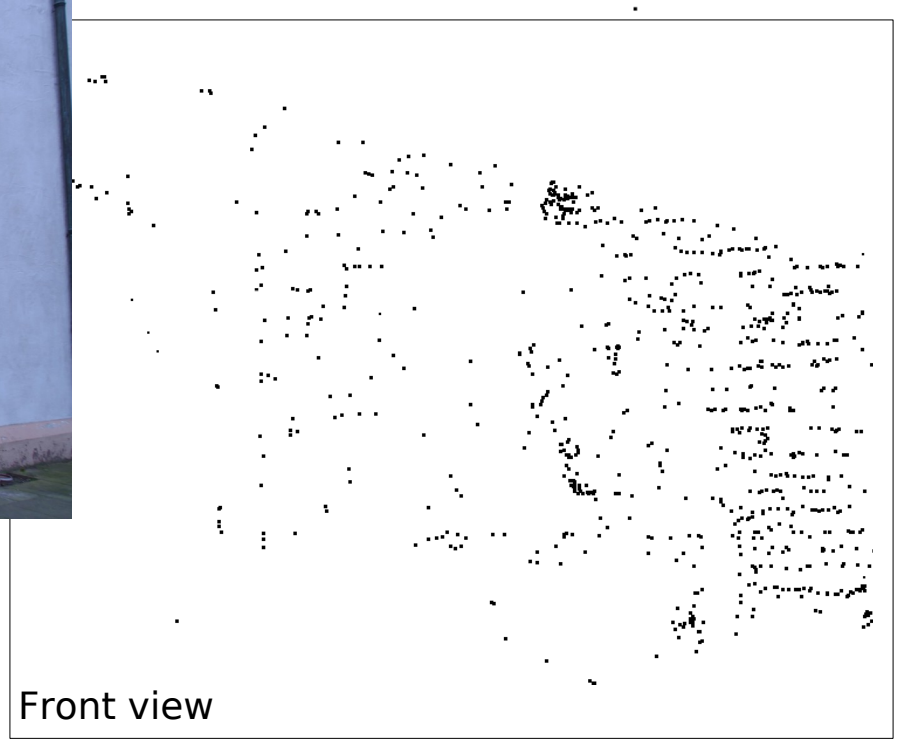
$$\boldsymbol{P}_1 = K_1[\boldsymbol{I}|\boldsymbol{0}]H_1$$
$$\boldsymbol{P}_2 = K_2[\boldsymbol{I}|\boldsymbol{0}]H_2$$

$$A = \begin{bmatrix} x\,\boldsymbol{p}_1^3 - \boldsymbol{p}_1^1 \\ y\,\boldsymbol{p}_1^3 - \boldsymbol{p}_1^2 \\ x'\,\boldsymbol{p}_2^3 - \boldsymbol{p}_2^1 \\ y'\,\boldsymbol{p}_2^3 - \boldsymbol{p}_2^2 \end{bmatrix}$$

$\boldsymbol{p}^i$ is i-th row of $\boldsymbol{P}$

Sparse reconstruction

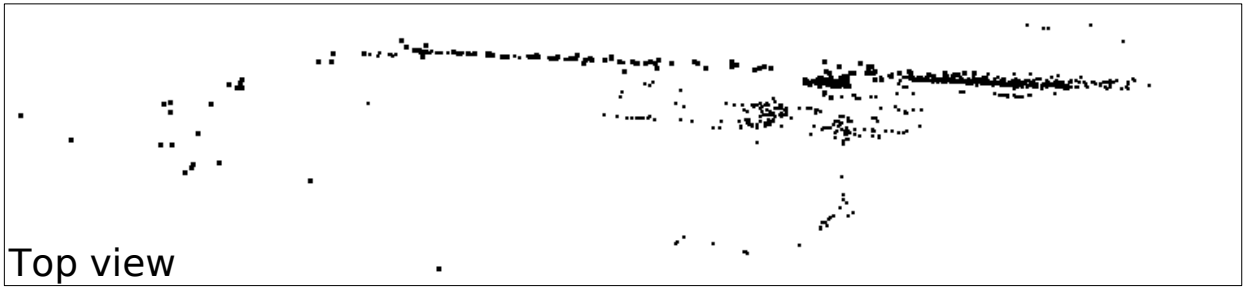




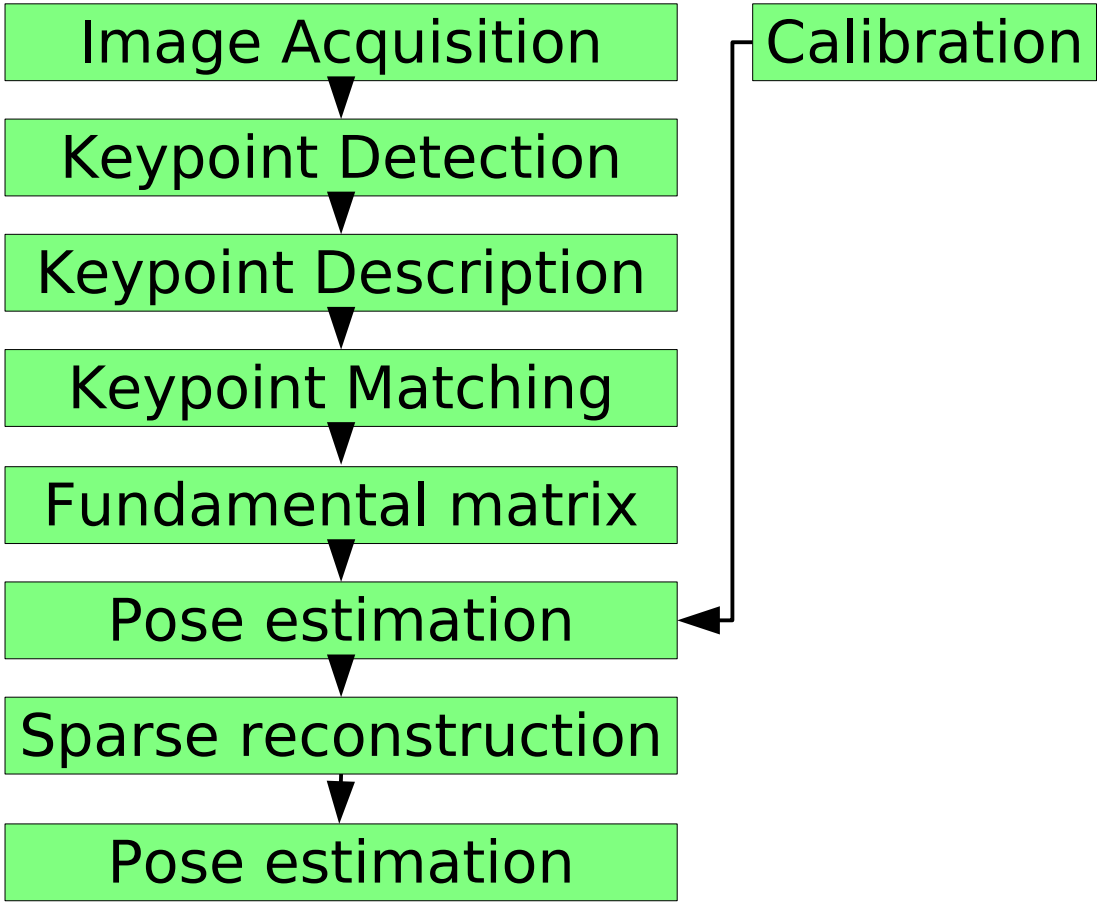
SFM Pipeline

- Image Acquisition
- Keypoint Detection
- Keypoint Description
- Keypoint Matching
- Fundamental matrix
- Pose estimation
- Sparse reconstruction

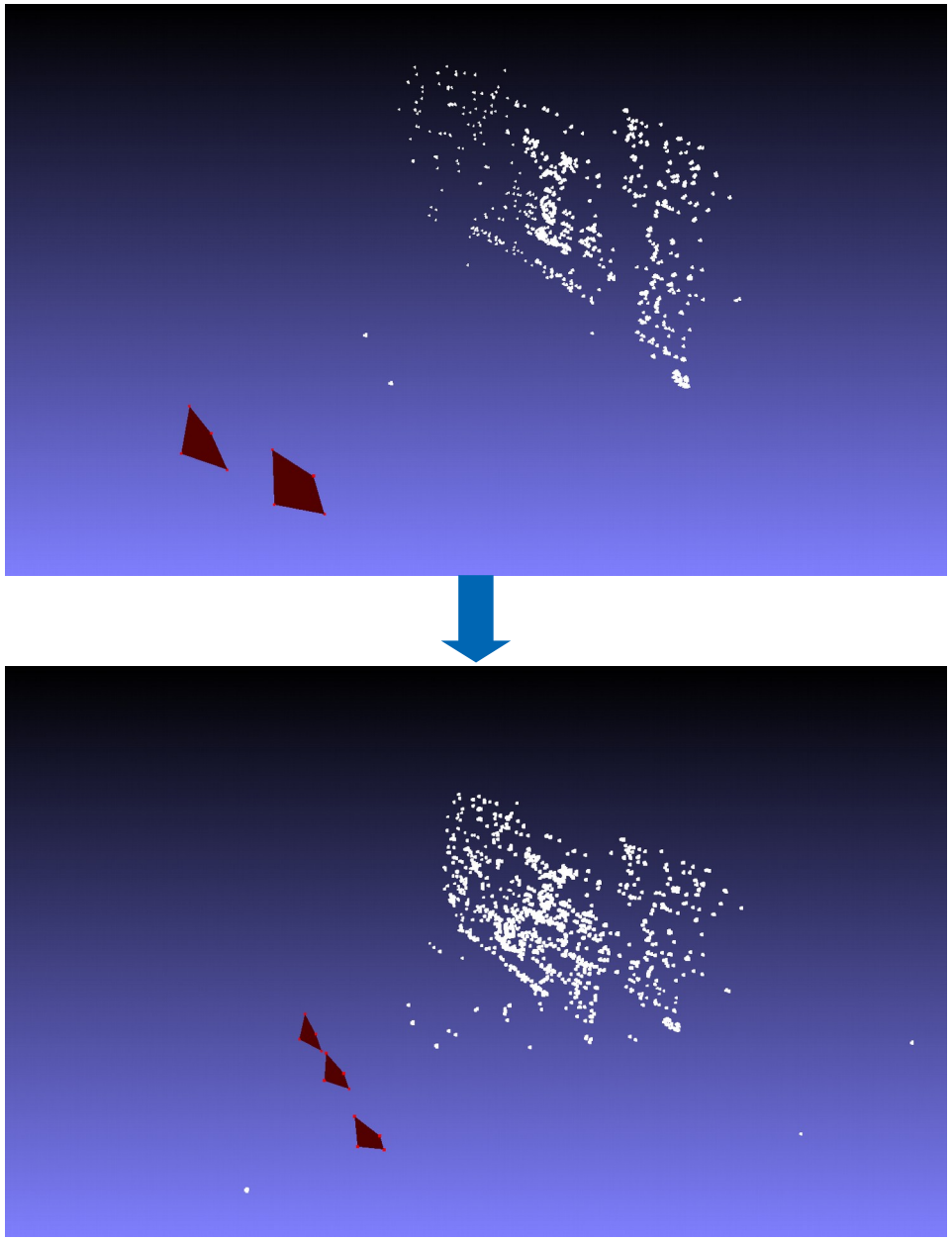
Calibration



SFM Pipeline



Pose Estimation 3rd Camera

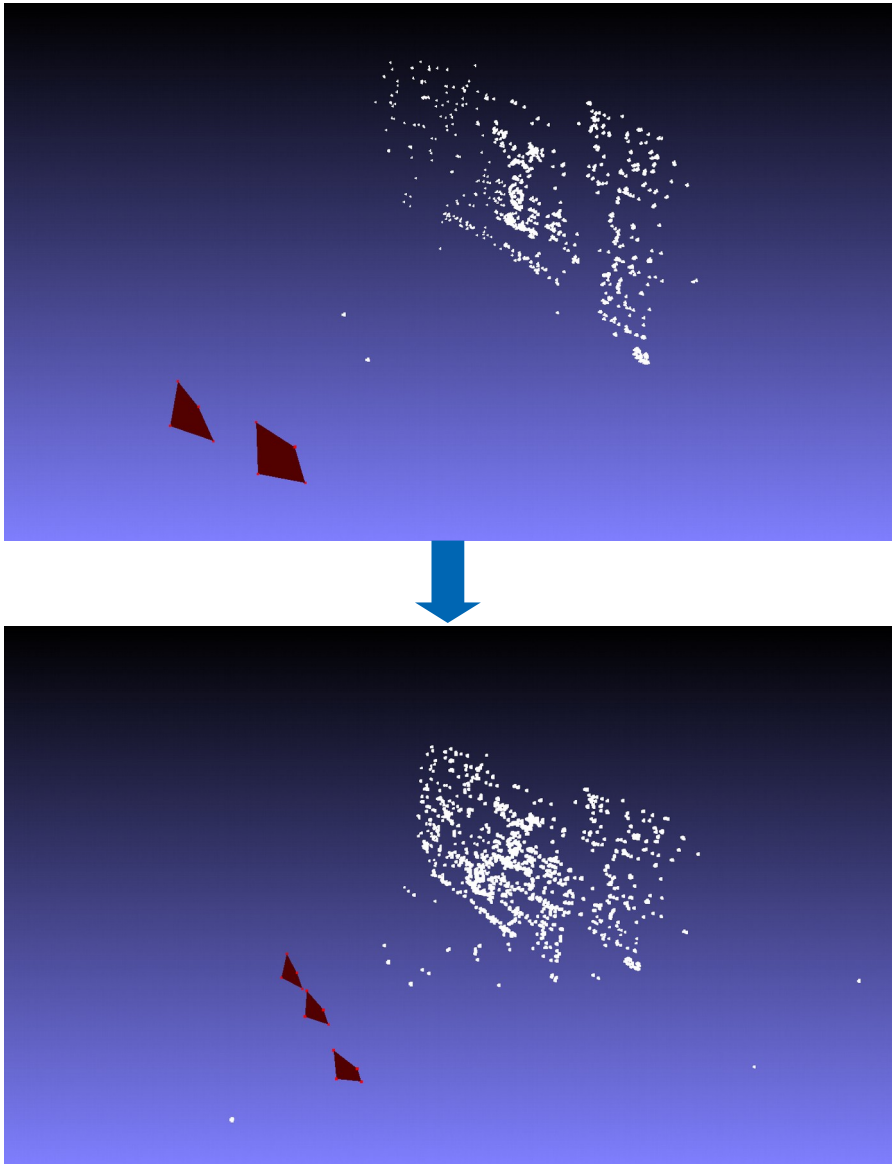


- Some 3D tracks have matches in 3rd camera
- Pair up 3D track positions with 2D keypoint positions in 3rd camera
- Estimate projection matrix
- Factorize into internal and external parts
- Use external calibration as pose of 3rd camera

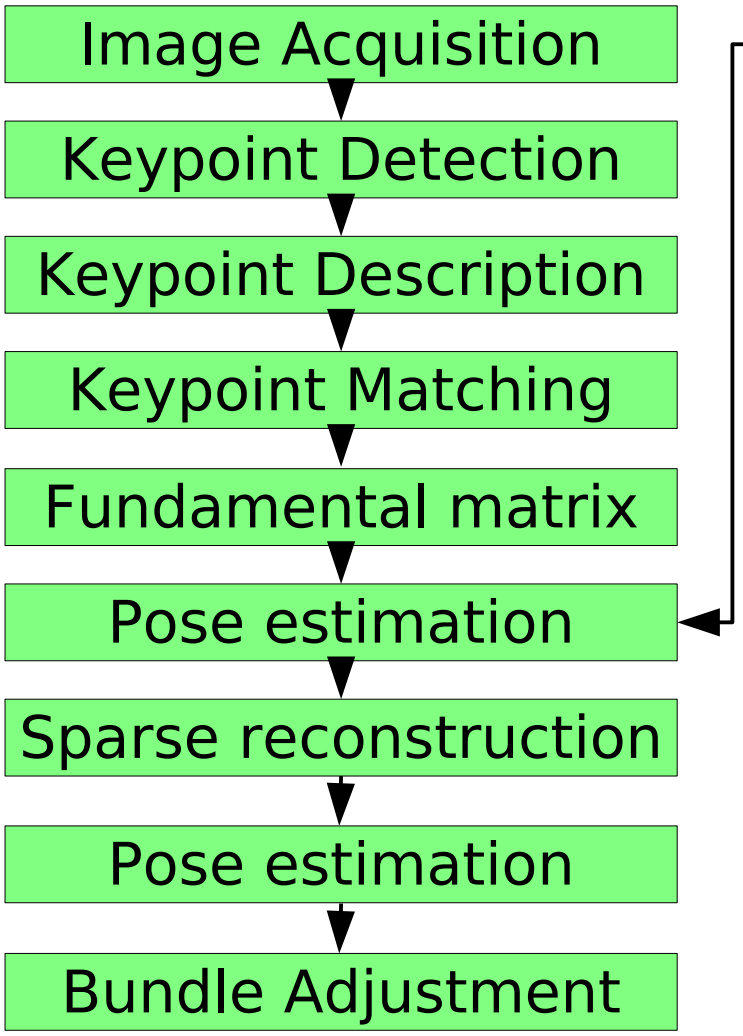
SFM Pipeline

- Image Acquisition
- Keypoint Detection
- Keypoint Description
- Keypoint Matching
- Fundamental matrix
- Pose estimation
- Sparse reconstruction
- Pose estimation

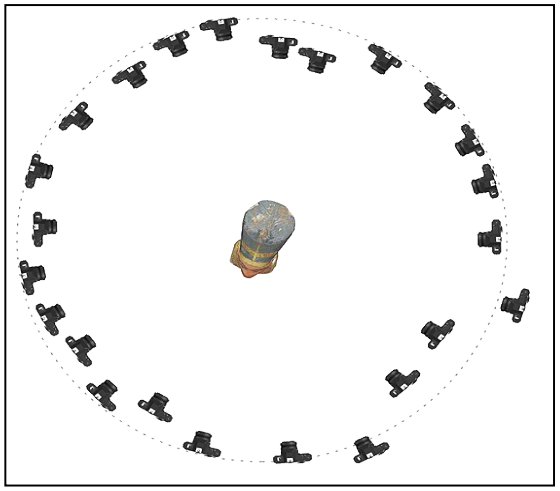
Calibration



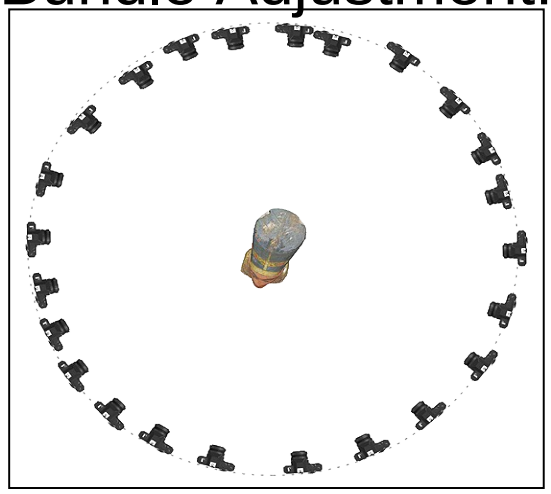
SFM Pipeline



Initial Solution:



Bundle Adjustment:



Reconstruction Ambiguity

Without global information the reconstruction is not unique:

$$x_i = (PH^{-1})HX_i = PX_i$$

$$x'_i = (PH^{-1})HX'_i = PX'_i$$

Both (P, P', X_i, X'_i) and $(PH^{-1}, P'H^{-1}, HX_i, HX'_i)$ are valid reconstructions

$$P \rightarrow PH^{-1}$$

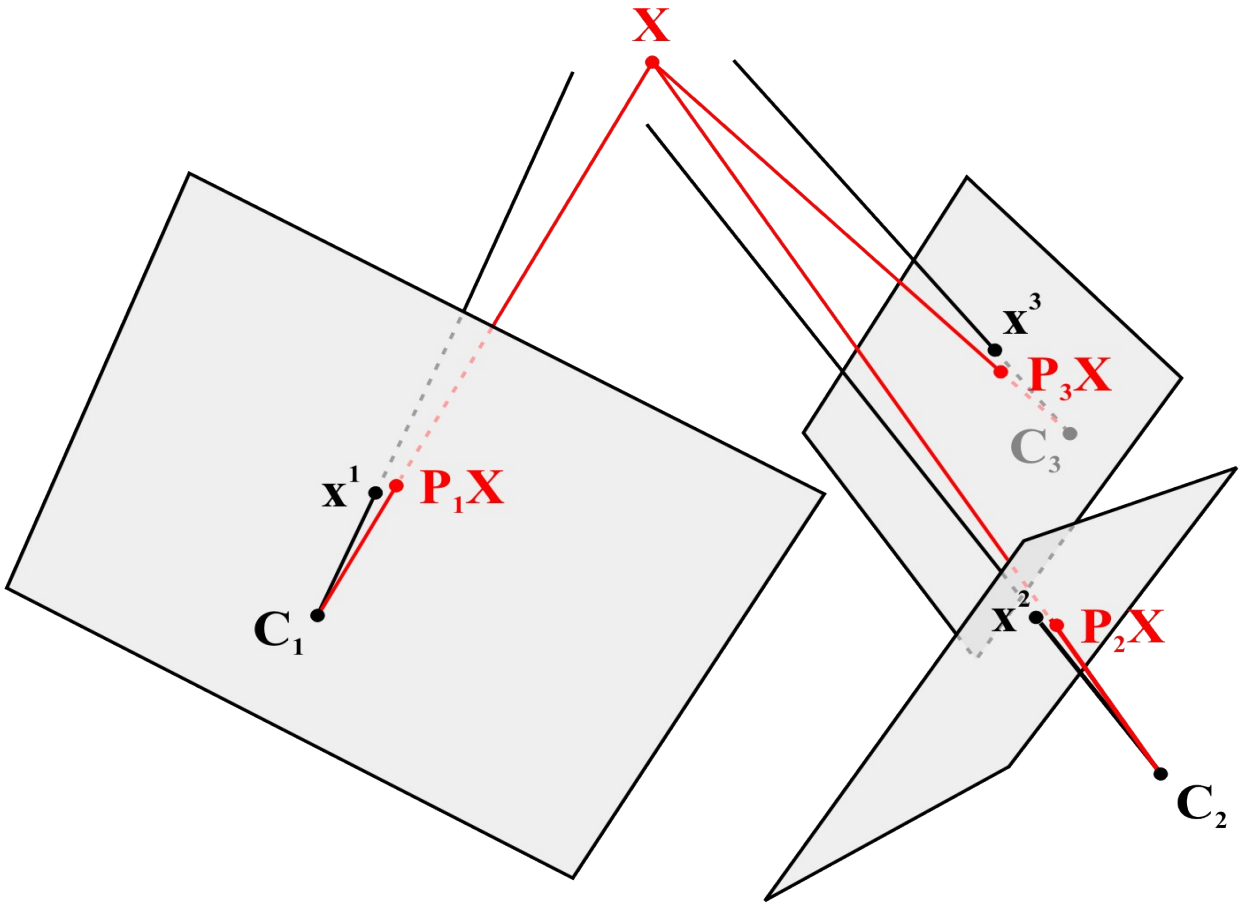
$$P' \rightarrow P'H^{-1}$$

$$X_i \rightarrow HX_i$$

Bundle Adjustment

Global error minimization
over all m points and n images:

$$\min_{\mathbf{P}_i, \mathbf{X}_j} \sum_{i=1}^{n} \sum_{j=1}^{m} d\left(\mathbf{P}_i \mathbf{X}_j, x_j^i\right)^2$$



Optimization Problems

Assume the following optimization problem:

$\vec{y}$ Measurements (keypoint locations)

$\vec{\beta}$ Tweakable parameters (focal lengths, orientations, ...)

$\vec{f}(\vec{\beta})$ Function that should reproduce measurements

Find the set of parameters, that minimizes the error:

$$\vec{\beta}_{\text{opt}} = \arg\min_{\vec{\beta}} S(\vec{\beta}) = \arg\min_{\vec{\beta}} \left| \vec{y} - \vec{f}(\vec{\beta}) \right|^2$$

Idea: Start with some initial state $\vec{\beta}$ and iteratively compute an update $\vec{\delta}$ that improves the state.

$$\vec{\beta}_{\text{new}} = \vec{\beta} + \vec{\delta}$$

Gauss Newton

The effects of an update $\vec{\delta}_{\mathrm{GN}}$ can be approximated using the Jacobian $\mathbf{J}$ of $\vec{f}(\cdot)$:

$$\vec{f}(\vec{\beta}+\vec{\delta}_{\mathrm{GN}}) \approx \vec{f}(\vec{\beta}) + \mathbf{J}\cdot\vec{\delta}_{\mathrm{GN}} \qquad \mathbf{J}_{i,j} = \frac{\partial f_i(\vec{\beta})}{\partial \beta_j}$$

Approximation of error: $S(\vec{\beta}+\vec{\delta}_{\mathrm{GN}}) \approx \left|\vec{y} - \vec{f}(\vec{\beta}) - \mathbf{J}\cdot\vec{\delta}_{\mathrm{GN}}\right|^2$

Minimum where derivative is zero:

$$\frac{\partial \left|\vec{y} - \vec{f}(\vec{\beta}) - \mathbf{J}\cdot\vec{\delta}_{\mathrm{GN}}\right|^2}{\partial \vec{\delta}_{\mathrm{GN}}} = 2\cdot\mathbf{J}^T\left(\vec{y} - \vec{f}(\vec{\beta}) - \mathbf{J}\cdot\vec{\delta}_{\mathrm{GN}}\right) = \vec{0}$$

Update step to minimum from solution to equation system:

$$\mathbf{J}^T\mathbf{J}\cdot\vec{\delta}_{\mathrm{GN}} = \mathbf{J}^T\cdot\left(\vec{y} - \vec{f}\left(\vec{\beta}\right)\right)$$

Levenberg Marquardt

The approximation usually only holds locally, and the Gauss Newton update might be too large:

$$\mathbf{J}^T\mathbf{J} \cdot \vec{\delta}_{\mathrm{GN}} = \mathbf{J}^T \cdot \left(\vec{y} - \vec{f}\left(\vec{\beta}\right)\right)$$

Idea of Levenberg and Marquardt: Introduce a damping parameter

$$\left(\mathbf{J}^T\mathbf{J} + \lambda \cdot \mathrm{diag}\left(\mathbf{J}^T\mathbf{J}\right)\right) \cdot \vec{\delta} = \mathbf{J}^T \cdot \left(\vec{y} - \vec{f}\left(\vec{\beta}\right)\right)$$

For small λ identical to Gauss Newton, for large λ like regular gradient descend with small step size.

Levenberg Marquardt

Ingredients that we need:

1. Initial state $\vec{\beta}_0$ (you)

2. Code to compute residuals (you)

3. Code to compute $\mathbf{J}$ (you)

$$\left(\mathbf{J}^T\mathbf{J} + \lambda \cdot \mathrm{diag}\left(\mathbf{J}^T\mathbf{J}\right)\right) \cdot \vec{\delta} = \mathbf{J}^T \cdot \left(\vec{y} - \vec{f}\left(\vec{\beta}\right)\right)$$

4. Some matrix multiplications (given)

4. Linear equation solver to solve for $\vec{\delta}$ (given)

5. Code to update state (you)

$$\vec{\beta}_{\mathrm{new}} = \vec{\beta} + \vec{\delta}$$

State Representation

Internal Calibration:

- One shared internal calibration for all cameras
- Represented as matrix, but only (one) focal length and principal point are adapted

External Calibration:

- External calibration per camera stored as 4x4 matrix
- Usually needs reorthonormalization (Gram Schmidt), but skipped here

Tracks:

- Track position stored as normalized homogeneous coordinates

- Additionally, one weight for each keypoint (outlier control)

Residual Computation

For each camera:

- Compose internal and external calibration

$$\mathbf{P} = \mathbf{K} \cdot \mathbf{H}_{3\times 4}$$

For each keypoint of camera:

- Project observed track into image

$$x = \mathrm{hom2eucl}(\mathbf{P} \cdot X)$$

- Compare to keypoint position
- Scale difference by keypoint weight

$$\underbrace{r}_{\text{residual}} = (\overbrace{y}^{\text{keypoint location}} - x) \cdot \underbrace{w}_{\text{keypoint weight}}$$

State Update

Update is not just an addition!

Internal Calibration:

- Update focal length relative to previous one
- Similarly, update principal point relative to previous one

$$\mathbf{K}_{\text{new}} = \begin{pmatrix} K_{0,0} \cdot (1 + \delta_{\mathbf{K},0}) & 0 & K_{0,2} \cdot (1 + \delta_{\mathbf{K},1}) \\ 0 & K_{1,1} \cdot (1 + \delta_{\mathbf{K},0}) & K_{1,2} \cdot (1 + \delta_{\mathbf{K},2}) \\ 0 & 0 & 1 \end{pmatrix}$$

External Calibration:

- Rotation update as angles around x/y/z axis of **camera**
  - ➔ e.g. x rotation update always moves image content vertically
  - ➔ No gimbal lock
- Translation similarly relative to current camera orientation

Tracks:

- Simple addition of x, y, z, w

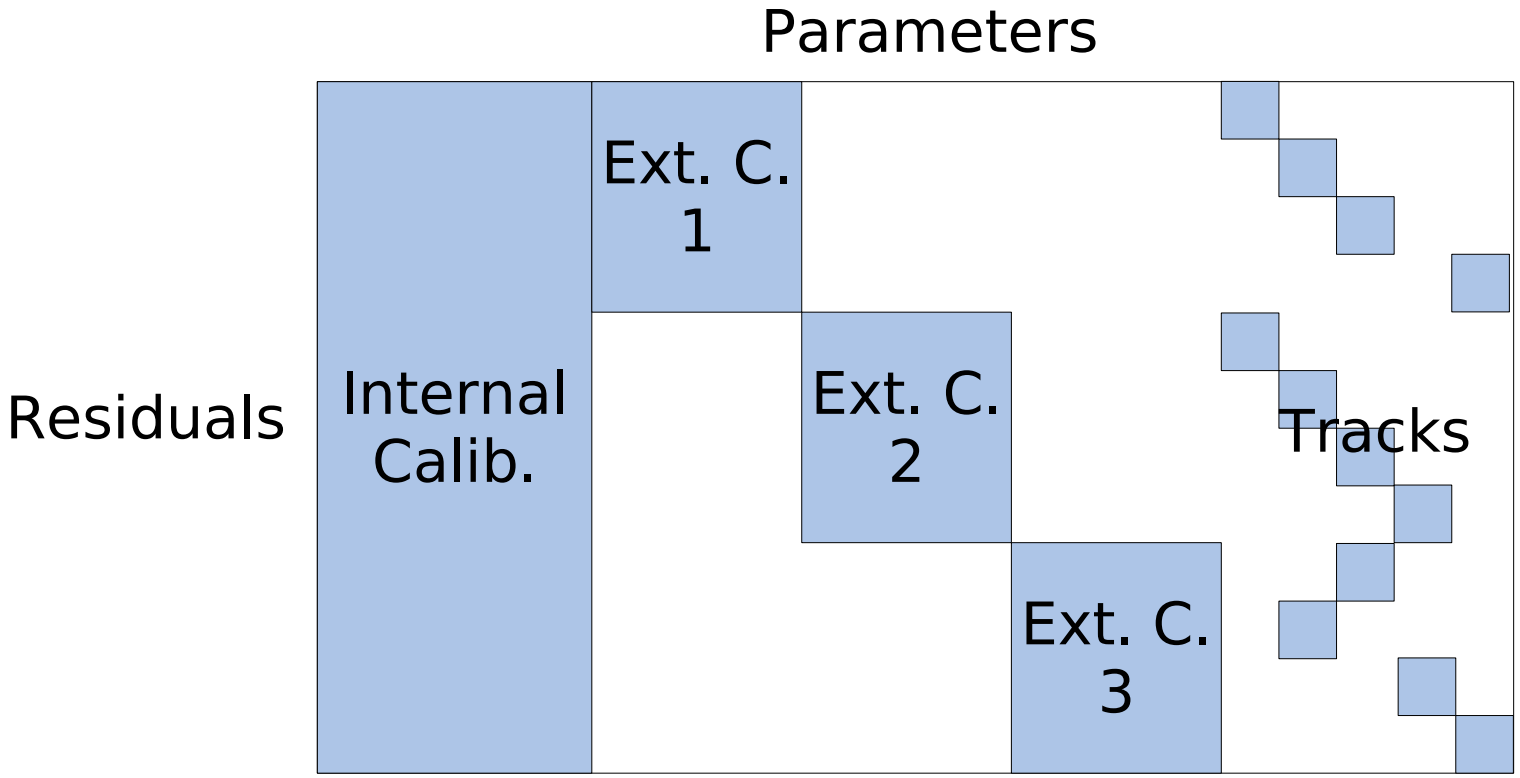
Sparse Jacobi

Assume 3 cameras all seeing 1000 tracks

6000 residuals
3009 parameters (+ internal calib)

Full $\mathbf{J}$ has 6000 x 3009 elements, but mostly empty

Sparse Jacobi



Code will only store a list of (double) rows, one for each keypoint
Each (double) row contains a block for:

- Internal calibration
- External calibration
- Track

Computing Jacobi Elements

Exploit the chain rule

$$x = \text{hom2eucl}(\mathbf{K} \cdot \mathbf{H}_{3\times 4} \cdot X)$$

Compute Jacobian for each step, and multiply them together

$$\frac{\partial x}{\partial X} = \mathbf{J}_{\text{hom2eucl}} \cdot \mathbf{J}_{\mathbf{K}} \cdot \mathbf{J}_{\mathbf{H}_{3\times 4}}$$

Example:

$$x = \text{hom2eucl}(u)$$
$$u = \mathbf{K} \cdot v$$
$$v = \mathbf{H}_{3\times 4} \cdot X$$

$$\frac{\partial x}{\partial \delta_{\mathbf{K}}} = \mathbf{J}_{\text{hom2eucl}} \cdot \frac{\partial u}{\partial \delta_{\mathbf{K}}}$$

Computing Jacobi Elements

$$x = \text{hom2eucl}(u)$$
$$u = \mathbf{K} \cdot v$$
$$v = \mathbf{H}_{3\times 4} \cdot X$$

Jacobi of conversion to Euclidean:

Operation:

$$\text{hom2eucl}\left(\begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix}\right) = \begin{pmatrix} \frac{u_0}{u_2} \\ \frac{u_1}{u_2} \end{pmatrix}$$

Jacobian:

$$\mathbf{J}_{\text{hom2eucl}} = \begin{pmatrix} \frac{1}{u_2} & 0 & -\frac{u_0}{u_2^2} \\ 0 & \frac{1}{u_2} & -\frac{u_1}{u_2^2} \end{pmatrix}$$

Computing Jacobi Elements

Jacobi of parameter update of internal calibration:

$$x = \text{hom2eucl}(u)$$
$$u = \mathbf{K} \cdot v$$
$$v = \mathbf{H}_{3\times 4} \cdot X$$

Update Step:

$$\mathbf{K}_{\text{new}} = \begin{pmatrix} K_{0,0} \cdot (1 + \delta_{\mathbf{K},0}) & 0 & K_{0,2} \cdot (1 + \delta_{\mathbf{K},1}) \\ 0 & K_{1,1} \cdot (1 + \delta_{\mathbf{K},0}) & K_{1,2} \cdot (1 + \delta_{\mathbf{K},2}) \\ 0 & 0 & 1 \end{pmatrix}$$

Jacobi:

$$\frac{\partial u}{\partial \delta_{\mathbf{K}}} = \begin{pmatrix} v_0 \cdot K_{0,0} & v_2 \cdot K_{0,2} & 0 \\ v_1 \cdot K_{1,1} & 0 & v_2 \cdot K_{1,2} \\ 0 & 0 & 0 \end{pmatrix}$$

$$\frac{\partial x}{\partial \delta_{\mathbf{K}}} = \mathbf{J}_{\text{hom2eucl}} \cdot \frac{\partial u}{\partial \delta_{\mathbf{K}}}$$

Computing Jacobi Elements

Jacobi of parameter update of external calibration:

$$x = \text{hom2eucl}(u)$$
$$u = \mathbf{K} \cdot v$$
$$v = \mathbf{H}_{3\times4} \cdot X$$

$$\frac{\partial x}{\partial \delta_{\mathbf{H}}} = \mathbf{J}_{\text{hom2eucl}} \cdot \mathbf{J}_{\mathbf{K}} \cdot \frac{\partial v}{\partial \delta_{\mathbf{H}}}$$

$$\boxed{\mathbf{J}_{\mathbf{K}} = \mathbf{K}}$$

$$\frac{\partial v}{\partial \delta_{\mathbf{H}}} = \begin{pmatrix} 0 & v_2 & -v_1 & X_3 & 0 & 0 \\ -v_2 & 0 & v_0 & 0 & X_3 & 0 \\ v_1 & -v_0 & 0 & 0 & 0 & X_3 \end{pmatrix}$$

Computing Jacobi Elements

Jacobi of parameter update of tracks:

$$x = \text{hom2eucl}(u)$$
$$u = \mathbf{K} \cdot v$$
$$v = \mathbf{H}_{3\times4} \cdot X$$

$$\frac{\partial x}{\partial \delta_X} = \mathbf{J}_{\text{hom2eucl}} \cdot \mathbf{J}_K \cdot \mathbf{J}_{\mathbf{H}_{3\times4}} \cdot \frac{\partial X}{\partial \delta_X}$$
$$= \mathbf{J}_{\text{hom2eucl}} \cdot \mathbf{K} \cdot \mathbf{H}_{3\times4}$$

$$\mathbf{J}_K = \mathbf{K}$$
$$\mathbf{J}_{\mathbf{H}_{3\times4}} = \mathbf{H}_{3\times4}$$

5. Exercise Theory

1. Why do we need the initial pose estimation? What happens, if we just run bundle adjustment?

2. How can more than three cameras be reconstructed? How can the reconstruction grow beyond what is visible in the first two cameras?

3. What would have to be changed, to also estimate radial distortion in the bundle adjustment?

5. Exercise

SFM Pipeline

Given:

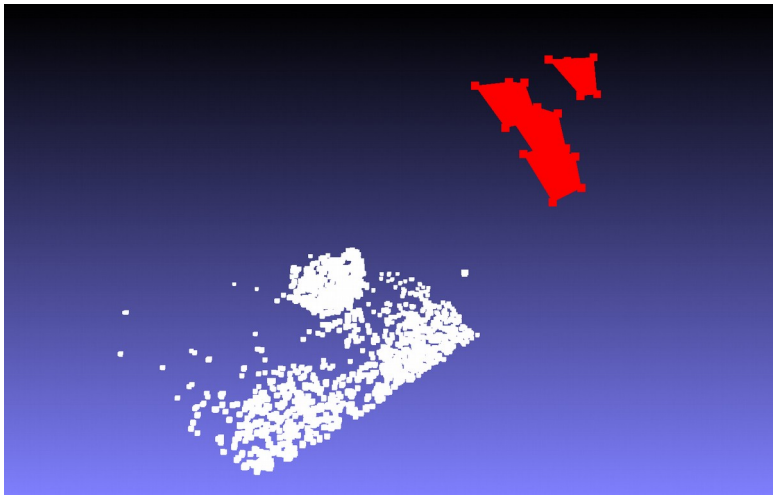
- Main function (main.cpp)
- Unit tests (unit_test.cpp)
- Header of individual functions
- Three images from Fountain-P11 with internal calibration

Todo:

- Individual functions
  - Fill in the necessary function body parts (Pcv5.cpp)
- Run first on the given images
- Then capture your own scene, 3-4 images, and run on that
- View produced .ply files in e.g. meshlab and include screenshots
- Look at the “precision” of the tracks before and after BA

Own Pictures

- Stick with small baseline
- Good light
- No blur
- Strongly textured non-flat object



5. Exercise - Given

FILE: main.cpp
int main(int argc, char argv)**

- Main function
  - Usage:
    - Focal length in argv[1]
    - Principal point in argv[2] and argv[3]
    - Image filenames in other arguments

    - Loads images builds scene and runs bundle adjustment
    - Writes .ply of the scene before and after bundle adjustment
      - View with e.g. meshlab

Also given:

- Matching
- Triangulation
- RANSAC for projection matrices
- Levenberg-Marquardt
- All kinds of plumbing

From Previous Exercises

```
Mat getCondition2D(Mat& p)
Mat getCondition3D(Mat& p)
```

```
cv::Matx33f estimateFundamentalRANSAC(
      vector<Vec3f>& p1, vector<Vec3f>& p2,
       unsigned numIterations, float threshold)
```

To Do

```
Matx44f computeCameraPose(Matx33f &K,
            vector<Vec3f> &p1, vector<Vec3f> &p2)
```

K: Internal calibration matrix
p1: N image points as array of 3D vectors from first camera
p2: N image points as array of 3D vectors from second camera
return: Output, 4x4 external calibration of second camera

- Estimates pose of second camera if first is in the origin
- Computes essential matrix from points and K
- Computes four possible camera matrices
- Chooses the one with most points in front of both cameras
- Points can be assumed to be outlier free (no RANSAC needed)

To Do

`void BundleAdjustment::BAState::computeResiduals(float *residuals) const`

- Computes residuals (w. weights)
- Project track positions into image and compare to keypoint location
- Details in source code

```
void BundleAdjustment::BAState::computeResiduals(float *residuals) const
{
    unsigned rIdx = 0;
    for (unsigned camIdx = 0; camIdx < m_cameras.size(); camIdx++) {
        const auto &calibState = m_internalCalibs[m_scene.cameras[camIdx].inte
        const auto &cameraState = m_cameras[camIdx];

        for (const KeyPoint &kp : m_scene.cameras[camIdx].keypoints) {
            const auto &trackState = m_tracks[kp.trackIdx];

            // Compute and store the residual in x direction multiplied by kp.weight
            // residuals[rIdx++] = ...
            // Compute and store the residual in y direction multiplied by kp.weight
            // residuals[rIdx++] = ...
        }
    }
}
```

```
struct InternalCalibrationState {
    cv::Matx33f K;
};

struct CameraState {
    cv::Matx44f H;
};

struct TrackState {
    cv::Vec4f location;
};
```

To Do

`void BundleAdjustment::BAState::computeJacobiMatrix( JacobiMatrix *dst) const`

- Computes Jacobi Matrix
- Exploit the chain rule!
- Build one small Jacobi matrix for each step, then multiply them together.
- Details in source code

```
void BundleAdjustment::BAState::computeJacobiMatrix(JacobiMatrix *dst) const
{
    BAJacobiMatrix &J = dynamic_cast<BAJacobiMatrix&>(*dst);
    unsigned rIdx = 0;
    for (unsigned camIdx = 0; camIdx < m_cameras.size(); camIdx++) {
        const auto &calibState = m_internalCalibs[m_scene.cameras[camIdx].internalCalibIdx];
        const auto &cameraState = m_cameras[camIdx];
        for (const KeyPoint &kp : m_scene.cameras[camIdx].keypoints) {
            const auto &trackState = m_tracks[kp.trackIdx];

            cv::Vec3f v = ???
            cv::Vec3f u = ???
            cv::Matx23f J_hom2eucl = ???
            cv::Matx33f du_dDeltaK = ???
            J.m_rows[rIdx].J_internalCalib = ???

            cv::Matx<float, 2, 4> J_v2eucl = ???
            cv::Matx<float, 3, 6> dv_dDeltaH = ???
            J.m_rows[rIdx].J_camera = ???

            cv::Matx24f J_worldSpace2eucl = ???
            J.m_rows[rIdx].J_track = ???
            rIdx++;
}}}
```

```
struct InternalCalibrationState {
    cv::Matx33f K;
};

struct CameraState {
    cv::Matx44f H;
};

struct TrackState {
    cv::Vec4f location;
};
```

To Do

`void BundleAdjustment::BAState::computeJacobiMatrix( JacobiMatrix *dst) const`

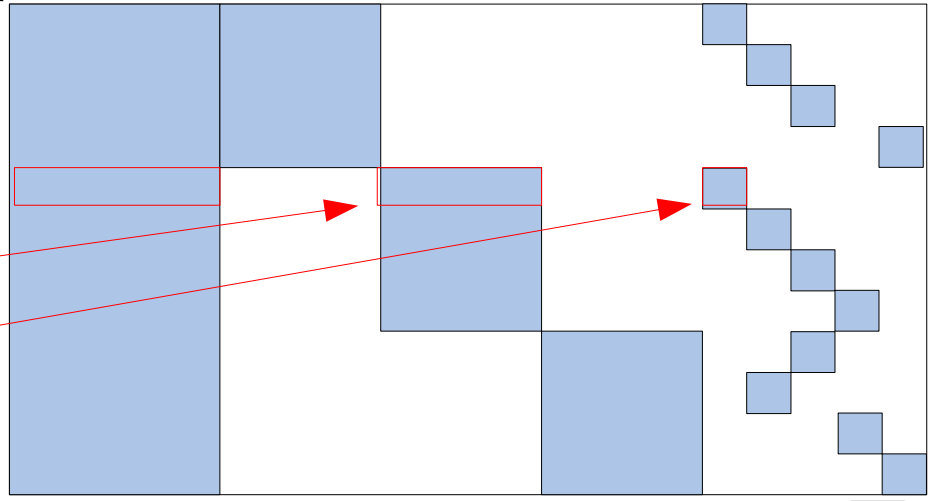
- Computes Jacobi Matrix
- Exploit the chain rule!
- Build one small Jacobi matrix for each step, then multiply them together.
- Details in source code

```
void BundleAdjustment::BAState::computeJacobiMatrix(JacobiMatrix *dst) const
{
    BAJacobiMatrix &J = dynamic_cast<BAJacobiMatrix&>(*dst);
    unsigned rIdx = 0;
    for (unsigned camIdx = 0; camIdx < m_cameras.size(); camIdx++) {
        const auto &calibState = m_internalCalibs[m_scene.cameras[camIdx].internalCalibIdx];
        const auto &cameraState = m_cameras[camIdx];
        for (const KeyPoint &kp : m_scene.cameras[camIdx].keypoints) {
            const auto &trackState = m_tracks[kp.trackIdx];

            cv::Vec3f v = ???
            cv::Vec3f u = ???
            cv::Matx23f J_hom2eucl = ???
            cv::Matx33f du_dDeltaK = ???
            J.m_rows[rIdx].J_internalCalib = ???

            cv::Matx<float, 2, 4> J_v2eucl = ???
            cv::Matx<float, 3, 6> dv_dDeltaH = ???
            J.m_rows[rIdx].J_camera = ???

            cv::Matx24f J_worldSpace2eucl = ???
            J.m_rows[rIdx].J_track = ???
            rIdx++;
}}}
```



To Do

`void BundleAdjustment::BAState::update(const float *update, State *dst) const`

- Computes the update of a state (= set of parameters)
- Updates Internal calibration, cameras' external calibrations, and tracks
- Details in source code

```
void BundleAdjustment::BAState::update(const float *update, State *dst) const
{
    [...]
    unsigned intCalibOffset = 0;
    for (unsigned i = 0; i < m_internalCalibs.size(); i++) {
        state.m_internalCalibs[i].K = m_internalCalibs[i].K;

// TO DO !!!
/*
 * Modify the new matrix K
 *
 * m_internalCalibs[i].K is the old matrix, state.m_internalCalibs[i].K is the new matrix.
 *
 * update[intCalibOffset + i * NumUpdateParams::INTERNAL_CALIB + 0]
 *     is how much the focal length is supposed to change (scaled by the old focal length)
 * update[intCalibOffset + i * NumUpdateParams::INTERNAL_CALIB + 1]
 *     is how much the principal point is supposed to shift in x direction (scaled by the old x position of the principal point)
 * update[intCalibOffset + i * NumUpdateParams::INTERNAL_CALIB + 2]
 *     is how much the principal point is supposed to shift in y direction (scaled by the old y position of the principal point)
 */
    }
    [...]
```

To Do

void BundleAdjustment::BAState::update(const float *update, State *dst) const

- Computes the update of a state (= set of parameters)
- Updates Internal calibration, cameras' external calibrations, and tracks
- Details in source code

```
void BundleAdjustment::BAState::update(const float *update, State *dst) const
{
    [...]
    unsigned cameraOffset = intCalibOffset + m_internalCalibs.size() * NumUpdateParams::INTERNAL_CALIB;
    for (unsigned i = 0; i < m_cameras.size(); i++) {
/*
 * Compose the new matrix H
 *
 * m_cameras[i].H is the old matrix, state.m_cameras[i].H is the new matrix.
 *
 * update[cameraOffset + i * NumUpdateParams::CAMERA + 0] rotation increment around the camera X axis (not world X axis)
 * update[cameraOffset + i * NumUpdateParams::CAMERA + 1] rotation increment around the camera Y axis (not world Y axis)
 * update[cameraOffset + i * NumUpdateParams::CAMERA + 2] rotation increment around the camera Z axis (not world Z axis)
 * update[cameraOffset + i * NumUpdateParams::CAMERA + 3] translation increment along the camera X axis (not world X axis)
 * update[cameraOffset + i * NumUpdateParams::CAMERA + 4] translation increment along the camera Y axis (not world Y axis)
 * update[cameraOffset + i * NumUpdateParams::CAMERA + 5] translation increment along the camera Z axis (not world Z axis)
 *
 * use rotationMatrixX(...), rotationMatrixY(...), rotationMatrixZ(...), and translationMatrix
 */

        //state.m_cameras[i].H = ...
    }

    [...]
```

To Do

`void BundleAdjustment::BAState::update(const float *update, State *dst) const`

- Computes the update of a state (= set of parameters)
- Updates Internal calibration, cameras' external calibrations, and tracks
- Details in source code

```
void BundleAdjustment::BAState::update(const float *update, State *dst) const
{
    [...]
    unsigned trackOffset = cameraOffset + m_cameras.size() * NumUpdateParams::CAMERA;
    for (unsigned i = 0; i < m_tracks.size(); i++) {
        state.m_tracks[i].location = m_tracks[i].location;

/*
 * Modify the new track location
 *
 * m_tracks[i].location is the old location, state.m_tracks[i].location is the new location.
 *
 * update[trackOffset + i * NumUpdateParams::TRACK + 0] increment of X
 * update[trackOffset + i * NumUpdateParams::TRACK + 1] increment of Y
 * update[trackOffset + i * NumUpdateParams::TRACK + 2] increment of Z
 * update[trackOffset + i * NumUpdateParams::TRACK + 3] increment of W
 */

            // Renormalization to length one
        float len = std::sqrt(state.m_tracks[i].location.dot(state.m_tracks[i].location));
        state.m_tracks[i].location *= 1.0f / len;
    }
}
```