UNIVERSIDADE DE SÃO PAULO
ESCOLA POLITÉCNICA

CAIO GARCIA CANCIAN
FELIPE CORRÊA MEYER MORÁN

# *Structure from Montion* a partir de uma sequência de vídeo

São Paulo
2020

CAIO GARCIA CANCIAN
FELIPE CORRÊA MEYER MORÁN

# *Structure from Montion* a partir de uma sequência de vídeo

Versão original

Monografia apresentada ao Departamento de Engenharia Mecatrônica e Sistemas Mecânicos da Escola Politécnica da Universidade de São Paulo para obtenção do título de Engenheiro.

Área de concentração:
Engenharia Mecatrônica

Orientador:
Prof. Dr. Jun Okamoto Junior

São Paulo

2020

Catalogação-na-publicação

Structure from motion a partir de uma sequência de vídeo / C. G. Cancian, F. Moran -- São Paulo, 2020. 78 p.

Trabalho de Formatura - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos.

1.Structure from Motion 2.Reconstrução 3.Processamento de vídeos I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos

Este relatório é apresentado como requisito parcial para obtenção do grau de engenheiro mecatrônico na Escola Politécnica da Universidade de São Paulo. É o produto do meu próprio trabalho, exceto onde indicado no texto. O relatório pode ser livremente copiado e distribuído desde que a fonte seja citada.

Caio Garcia Cancian

Felipe Corrêa Meyer Morán

**Agradecimentos**

Gostaríamos de agradecer primeiramente aos nossos pais por todas as oportunidades que nos forneceram e o apoio incondicional ao longo das nossas trajetórias. Agradecemos também a todos os amigos, familiares, professores e demais pessoas que nos acompanharam e contribuíram para o nosso percurso até esse importante momento. E também, ao nosso orientador, Prof. Dr. Jun Okamoto Junior, muito obrigado pelo apoio, incentivo e confiança durante todo o desenvolvimento do projeto.

Por último, gostaríamos de agradecer a Enstartiflette por organizar a viagem de ski em fevereiro de 2018 que, num momento de grande alegria, nos permitiu ter a ideia de fazermos esse trabalho juntos.

**Resumo**

O presente trabalho visa abordar o problema de visão computacional conhecido como *Structure from Motion* em que, essencialmente, busca-se reconstruir, a partir de imagens de uma cena estática adquiridas em posições distintas, tanto o modelo tridimensional dessa cena, quanto a trajetória da câmera responsável por registrar as imagens. Além disso, no caso específico desse trabalho, as imagens adquiridas são provenientes de uma sequência de vídeo registrada continuamente, o que introduz particularidades em seu tratamento. Esse tipo de problema possui interesse em diferentes áreas do conhecimento, sendo parte constituinte de aplicações cujo escopo vão de imagens médicas a realidade virtual e aumentada. Dada a vasta literatura e diferentes abordagens possíveis para o tema, a primeira etapa do projeto consistiu em um estudo teórico dos diferentes algoritmos que tipicamente constituem um *pipeline* de *Structure from Motion*. Em seguida, a partindo-se dos algoritmos melhor adaptados às particularidades do processamento de um vídeo, a segunda etapa do projeto consistiu na proposição de *pipeline* próprio. Por último, a terceira e última etapa do projeto foi a validação do *pipeline* proposto utilizando-se tanto dados sintéticos quanto vídeos reais em diferentes situações, de modo a se determinar os pontos fortes e as eventuais limitações da proposta. Dessa forma, esse texto discorre tanto sobre os aspectos teóricos de cada algoritmo utilizado, como também sobre aspectos práticos de implementação e que foram levados em conta em cada uma dessas etapas da elaboração da solução para o problema proposto.

Palavras-chaves: Structure from motion. Reconstrução 3D. Processamento de vídeos.

**Abstract**

The present work aims to address the computer vision problem known as Structure from Motion in which, essentially, it is necessary to reconstruct, from images of a static scene acquired in different positions, both the three-dimensional model of this scene, as well as the trajectory of the camera responsible for registering the images. In addition, in the specific case of this work, the acquired images come from a continuously recorded video sequence, which introduces particularities in their treatment. This type of problem has an interest in different areas of knowledge, being a constituent part of applications whose scope ranges from medical images to virtual and augmented reality. Given the vast literature and different possible approaches to the theme, the first stage of the project consisted of a theoretical study of the different algorithms that are typically used in a Structure from Motion pipeline. Then, based on the algorithms that are best adapted to the video processing difficulties, the second stage of the project consisted of the proposition of a new pipeline. Finally, the third and final stage of the project was the validation of the proposed pipeline using both synthetic data and real videos acquired in different situations, in order to determine the strengths and possible limitations of the proposed method. Thus, this text discusses both the theoretical aspects of each algorithm used, as well as practical aspects of implementation, which were taken into account in each of these steps followed when solving the proposed problem.

Key-words: Structure from motion. 3D Reconstruction. Video processing.

## Lista de figuras

**Sumário**

# 1 Introdução

*Structure from Motion* representa uma classe de problemas em que se deseja recuperar as informações da estrutura geométrica tridimensional de uma cena estática, a partir de um conjunto de imagens bidimensionais, utilizando essencialmente informações sobre a estimativa do movimento da câmera entre as imagens, como representado no esquema da figura (1.1).

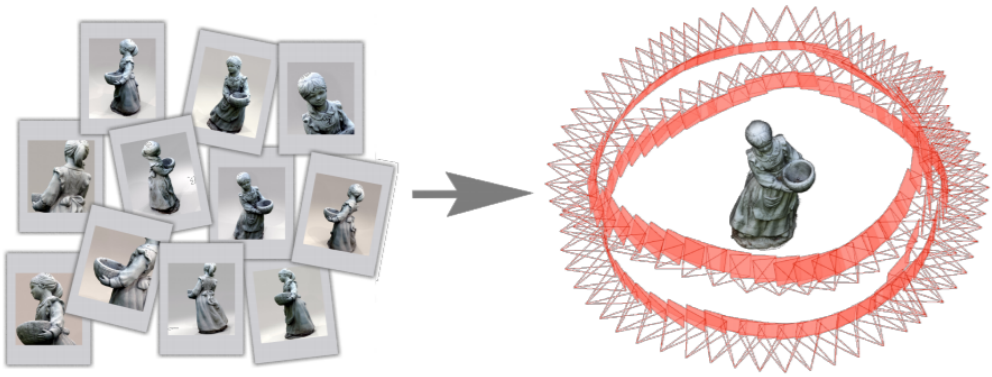


Figura 1.1 – Exemplo de reconstrução 3D e estimativa do movimento da câmera a partir de imagens 2D, retirado de (BIANCO; CIOCCA; MARELLI, 2018).

Esse tipo de problema tem historicamente despertado interesse em trabalhos de visão computacional, uma vez que possui diversas aplicações: preservação e digitalização de patrimônio cultural (REMONDINO, 2011; RONCELLA; RE; FORLANI, 2011), como a criação de modelos 3D de construções históricas e peças de museus; geociências e topografia (MANCINI et al., 2013; JAVERNICK; BRASINGTON; CARUSO, 2014), com a criação de modelos digitais detalhados de relevos; realidade virtual e aumentada (QUAN; WU, 2013; KROEGER; GOOL, 2014); e aplicações médicas (CARLBOM; TERZOPOULOS; HARRIS, 1994; LEONARD et al., 2016), através da reconstrução de volumes 3D de tecidos e órgãos humanos, utilizados durante o diagnóstico ou em intervenções cirúrgicas.

Essencialmente, o problema de SfM é resolvido através de um *pipeline* básico de, pelo menos, 3 etapas: extração de características importantes das imagens (*feature extraction*), especialmente pontos, linhas e outras estruturas geométricas, assim como a realização da correspondência dessas características entre as diferentes imagens (*feature matching*); estimativa do movimento da câmera, a partir da evolução das características escolhidas entre as imagens; e, finalmente, a reconstrução da estrutura 3D da cena, usando as informações das etapas anteriores, como resumido no diagrama da figura (1.2).

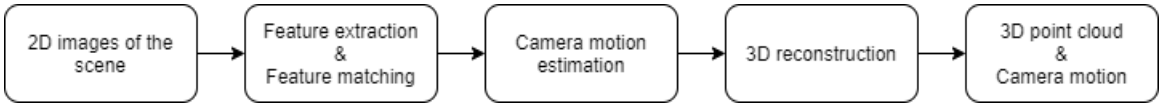



Figura 1.2 – Pipeline básico de SfM.

De maneira mais formal, seguindo o que é feito por exemplo em (HARTLEY; ZISSERMAN, 2003), o problema de *Structure from Motion* pode ser definido da seguinte forma: dado um conjunto de $k$ pontos bidimensionais $\mathbf{u}_1, \mathbf{u}_2, ..., \mathbf{u}_k$ projetados em $m$ imagens, o objetivo é encontrar as $m$ matrizes de projeção $\mathbf{P}_1, ..., \mathbf{P}_m$, bem como os $n$ pontos tridimensionais originais $\mathbf{X}_1, ..., \mathbf{X}_n$ que compõem a estrutura da cena.

Usando a notação de coordenadas homogêneas [1] é possível relacionar um ponto 3D $\tilde{\mathbf{X}} \sim [X \quad Y \quad Z \quad 1]^\top$ com seu pixel correspondente projetado $\tilde{\mathbf{u}} = [x \quad y \quad 1]^\top$ através da matriz de projeção $\mathbf{P}$:

$$\tilde{\mathbf{u}} \sim \mathbf{P}\tilde{\mathbf{X}} \tag{1}$$

em que a matriz de projeção $\mathbf{P} \in \mathbb{R}^{3\times4}$, usando o modelo de câmera *pinhole*, é definida pela igualdade a menos de escala $\mathbf{P} \sim \mathbf{K}[\mathbf{R} \mid \mathbf{t}]$, em que $\mathbf{K}$ é a chamada matriz de parâmetros intrínsecos da câmera ou matriz de calibração, uma vez que está relacionada com parâmetros como distância focal e distorção introduzida pela câmera; $\mathbf{R} \in \mathbb{R}^{3\times3}$ é a matriz de rotação, que representa a orientação da câmera, e $\mathbf{t} \in \mathbb{R}^{3\times1}$ é o vetor de translação, que representa a posição da câmera. As matrizes de rotação e translação juntas formam os chamados parâmetros extrínsecos e descrevem o posicionamento da câmera no espaço tridimensional. Para uma explicação mais detalhada, é possível consultar (HARTLEY; ZISSERMAN, 2003).

Como mencionado anteriormente, o problema de SfM é resolvido, então, em 3 etapas essenciais. A primeira delas é extração de características das imagens e a realização da correspondência dessas características entre duas imagens distintas, como mostrado na figura (1.3). Isso porque, como será explicitado posteriormente, através de um conjunto de imagens obtidas em posições diferentes, com projeções de um mesmo ponto do espaço, é possível recuperar não só as matrizes de rotação e translação que levam a câmera de uma posição de aquisição à outra, mas também as coordenadas desse ponto.

[1] Se $\mathbf{x} \in \mathbb{R}^n$, em coordenadas homogêneas $\tilde{\mathbf{x}} \in \mathbb{R}^{n+1}$, sendo o último elemento um fator de escala. Assim, se $\tilde{\mathbf{x}} = [\tilde{x_1} \ \tilde{x_2} \ w]^\top$, então $\mathbf{x} = [\tilde{x_1}/w \quad \tilde{x_2}/w]^\top$ e a igualdade a menos do fator de escala $w$ é escrita $\tilde{\mathbf{x}} \sim [x_1 \ x_2 \ 1]^\top$

Figura 1.3 – Exemplo de correspondência entre pontos de duas imagens, retirado de (SNAVELY; SEITZ; SZELISKI, 2008).

Existem inúmeros algoritmos na literatura que são capazes de detectar e realizar correspondências entre pontos de diferentes imagens (KARAMI; PRASAD; SHEHATA, 2017). Um algoritmo que é comumente aplicado é o chamado *Scale Invariant Feature Transform* (SIFT) (LOWE, 2004), responsável por encontrar características locais que são invariantes à escala e à rotações. Entretanto, mesmo possuindo boa performance, esse algoritmo, assim como outros derivados, é computacionalmente exigente e seu uso pode se tornar impeditivo em aplicações que devam funcionar em tempo real ou que precisem tratar um volume grande de imagens, como é o caso de um conjunto de *frames* que constituem um vídeo. Por isso, é interessante considerar uma outra classe de algoritmos de correspondência que possuem complexidade computacional mais baixa. Uma solução apresentada foram algoritmos baseados em fluxo óptico, sendo o mais comum deles conhecido como Kanade-Lucas-Tomasi *feature tracker* (KLT) (LUCAS; KANADE et al., 1981; TOMASI; KANADE, 1991).

Tendo sido realizada a correspondência entre pontos de imagens distintas, a segunda etapa do pipeline consiste em recuperar as matrizes de projeção que, como mencionado, possuem informações sobre a orientação (rotação) e posicionamento (translação) das câmeras. Isso é feito através usando a chamada matriz essencial $\mathbf{E}$ que relaciona duas câmeras $\mathcal{C}$ e $\mathcal{C}'$ de matrizes de projeção $\mathbf{P}$ e $\mathbf{P}'$, respectivamente (LONGUET-HIGGINS, 1981).

De maneira resumida, um ponto $\tilde{\mathbf{x}}'$ capturado por $\mathcal{C}'$ será descrito em $\mathcal{C}$ por

$$\tilde{\mathbf{x}} = \mathbf{R}\tilde{\mathbf{x}}' + \mathbf{t} \tag{2}$$

de modo que multiplicando a equação pela direita por $\tilde{\mathbf{x}}^\top [\mathbf{t}]_\times$ obtem-se:

$$\tilde{\mathbf{x}}^\top [\mathbf{t}]_\times \mathbf{R}\tilde{\mathbf{x}}' = \tilde{\mathbf{x}}^\top \mathbf{E}\tilde{\mathbf{x}}' = 0 \tag{3}$$

em que $\mathbf{E} \sim [\mathbf{t}]_\times \mathbf{R}$ é a matriz essencial, que introduz uma restrição algébrica, cuja interpretação geométrica está relacionado com o fato de que as projeções de um mesmo ponto devem estar na mesma linha epipolar.

Na prática, nós temos acesso às projeções modificadas pelos parâmetros intrísecos da câmera $\tilde{\mathbf{u}} \sim \mathbf{K}\tilde{\mathbf{x}}$, de modo que a equação 3 normalmente é reescrita em termos da matriz fundamental $\mathbf{F} \sim (\mathbf{K}^{-1})^\top \mathbf{E} \mathbf{K}'^{-1}$

$$\tilde{\mathbf{u}}^\top \mathbf{F} \tilde{\mathbf{u}}' = 0 \tag{4}$$

sendo que $\mathbf{F} \in \mathbb{R}^{3\times3}$, com posto igual a 2. Dessa forma, dado um par de imagens e pelo menos 8 pontos que se correspondem em cada uma das imagens, em teoria é possível estimar a matriz fundamental de maneira linear (LONGUET-HIGGINS, 1981). Estimada $\mathbf{F}$ e conhecida as matrizes de calibragem $\mathbf{K}$ é possível realizar as transformações inversas: $\mathbf{E} \sim \mathbf{K}'^\top \mathbf{F} \mathbf{K}$ e $\mathbf{E} = [\mathbf{t}]_\times \mathbf{R}$, de modo a se recuperar as matrizes de rotação $\mathbf{R}$ e translação $\mathbf{t}$ e, portanto, as matrizes de projeção $\mathbf{P} \sim \mathbf{K}[\mathbf{R} \mid \mathbf{t}]$ e $\mathbf{P}' \sim \mathbf{K}'[\mathbf{R} \mid \mathbf{t}]$

Desde a solução ao problema acima, conhecida como algoritmo de 8 pontos e introduzido em (LONGUET-HIGGINS, 1981), uma extensa literatura foi produzida para obter melhores estimativas do movimento das câmeras através de melhoras na estimação da matriz essencial. Inicialmente, isso foi feito melhorando-se o algoritmo de 8 pontos original, como a versão normalizada apresentada em (HARTLEY, 1997), mas depois utilizando-se cada vez menos pontos, até a versão otimizada com apenas 5 proposta em (NISTÉR, 2004).

Tendo as matrizes de projeção estimadas, a última etapa essencial para o *pipeline* é a reconstrução da estrutura 3D da cena, o que é feito em um processo conhecido como triangulação. Em teoria, tendo-se 2 vistas de um ponto $\mathbf{X}$ no espaço, ele deveria estar na intersecção dos raios reprojetados a partir de $\mathbf{u}$ e $\mathbf{u}'$, o que pode ser feito utilizando a matriz pseudo-inversa de $\mathbf{P}$ e $\mathbf{P}'$ respectivamente, como descrito em (HARTLEY; ZISSERMAN, 2003). Contudo, devido a presença de ruído nas imagens, de maneira geral esses raios reprojetados não se interceptam, de modo que, em geral, os pontos 3D são recuperados minimizando-se uma métrica apropriada, como por exemplo, o erro empírico de reprojeção

$$\mathbf{X}^\star = \arg\min_{\mathbf{X}} \sum_i \mathcal{L}_i(\mathbf{u}_i, \hat{\mathbf{u}}_i(\mathbf{P}_i, \mathbf{X})) \tag{5}$$

em que $\mathbf{u}_i$ é o pixel observado, $\hat{\mathbf{u}}_i$ é o pixel estimado e a somatória é feita nas $i$ imagens que contem o ponto $\mathbf{X}$. Assumindo que o ruído presente na imagem é Gaussiano e

branco, a escolha da métrica como sendo a norma $L_2$ entre os pixels estimado e observado $\mathcal{L}_i = ||\mathbf{u}_i - \hat{\mathbf{u}}_i(\mathbf{P}_i, \mathbf{X})||_2^2$ implica que $\mathbf{X}^\star$ é o estimador de máxima verossimilhança do ponto real.

O resultado final dessas 3 etapas deve ser, portanto, uma nuvem de pontos no espaço 3D, bem como as diferentes posições e orientações das câmeras que geraram as imagens 2D, como representado na figura (1.4).

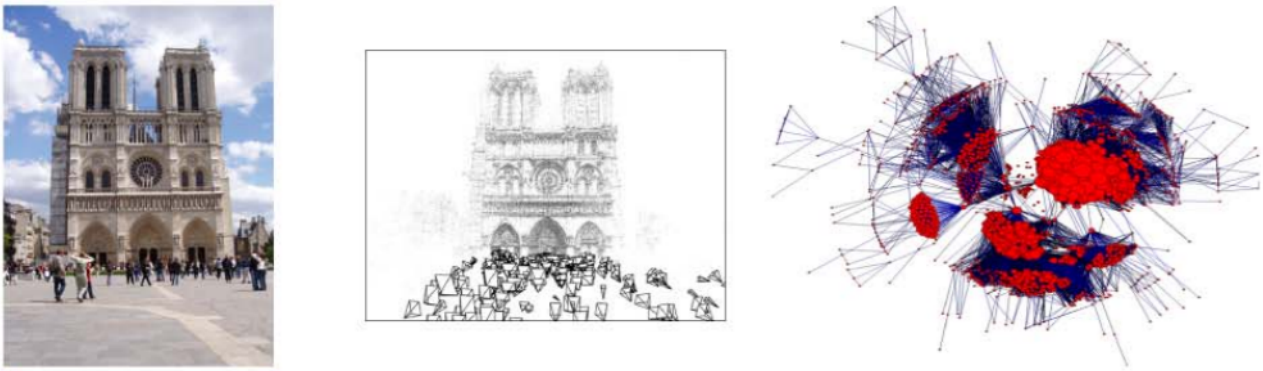


Figura 1.4 – Resultado de um pipeline de SfM, retirado de (SNAVELY; SEITZ; SZELISKI, 2008).

Esse trabalho de conclusão de curso se insere, dessa forma, nesse contexto. O nosso objetivo é estabelecer um *pipeline* de SfM capaz de receber uma sequência de vídeo de uma cena estática e reconstruir uma nuvem de pontos 3D, bem como as estimar a trajetória da câmera. Para tanto, serão implementadas as diferentes etapas apresentadas anteriormente, levando-se em conta as particularidades impostas pelo tratamento sequencial dos *frames* de um vídeo. Nas seções que seguem, uma revisão bibliográfica com os principais desenvolvimentos no tema será apresentada. Partindo-se dela, o *pipeline* desenvolvido será detalhado e os resultados obtidos serão analisados e discutidos.

# 2 Estado da arte

Nessa seção os avanços mais recentes na resolução do problema de reconstrução da estrutura 3D de uma cena estática a partir do movimento da câmera serão apresentados. Para tanto, em primeiro lugar, as técnicas gerais mais recentes das diferentes etapas do *pipeline* geral serão apresentadas e, em seguida, os principais avanços para a resolução do problema quando restrito ao caso específico em que as entradas representam *frames* de um vídeo.

É importante explicitar que, como em muitas outras tarefas de visão computacional, os trabalhos mais recentes na área utilizam técnicas de aprendizado de máquina e, em especial, de aprendizagem profunda e redes neurais. Mesmo que essas abordagens apresentem resultados promissores, elas fogem do escopo desse trabalho e, portanto, não serão discutidas nessa seção.

## *2.1 Feature Extraction, Matching e Tracking*

A extração automática de caracteristicas de interesse em uma cena (*feature extraction*), bem como a realização da correspondência dessas características entre imagens distintas dessa cena, obtidas de diferentes pontos de vista (*feature matching*), são etapas essenciais em várias áreas de visão computacional e, portanto, possuem uma vasta literatura própria. Assim, apenas alguns dos principais métodos serão apresentados, enquanto uma revisão extensiva pode ser encontrada em (TUYTELAARS; MIKOLAJCZYK et al., 2008).

Um primeiro algoritmo de extração de características que recebeu bastante atenção é conhecido como *Scale Invariant Feature Transform* (SIFT), apresentado em (LOWE, 2004). De modo geral, esse algoritmo busca encontrar características que são invariantes a rotações e escala, além de parcialmente invariantes à diferença de luminosidade e mudança de ponto de vista da aquisição. Para tanto, existem 4 passos fundamentais na criação das características SIFT: em primeiro lugar, possíveis pontos de interesse são obtidos encontrando-se extremos da representação no espaço de escala da imagem, usando diferença de Gaussianas; em segundo lugar, a localização, a escala e o ratio das curvaturas principais são calculados, de modo a eliminar candidatos com baixo contraste ou mal localizados;

o terceiro passo consiste em calcular gradientes locais para atribuir uma orientação aos pontos candidatos; por último, uma descrição local de cada um desses pontos é calculada, utilizando-se informações da magnitude e orientação do gradiente no entorno. Apesar de ter sido aplicado com sucesso em diferentes aplicações de visão computacional, inclusive em *pipelines* de SfM, o algoritmo para a extração de características SIFT é computacionalmente exigente, se tornando impraticável em aplicações em tempo real ou com grande volume de dados, como vídeos em que *frames* de alta resolução são amostrados em taxas relativamente elevadas.

Para tentar solucionar esse problema de eficiência computacional, diferentes algoritmos foram propostos, dos quais recebeu bastante atenção o conhecido como *Speed Up Robust Features* (SURF), proposto em (BAY; TUYTELAARS; GOOL, 2006). Esse algoritmo é inspirado nas ideias utilizadas nas características SIFT, porém com modificações para acelerar cada uma das diferentes etapas descritas anteriormente. Assim, a diferença de Gaussianas é aproximada por filtros quadrados (*box filters*) calculados na imagem integral (COOPER, 1989), que são mais rápidos e podem ser aplicados paralelamente em diferentes escalas; além disso, a localização e escala dos pontos de interesse é baseada no determinante da matriz Hessiana e, finalmente, tanto a orientação quanto o descritor local são obtidos usando-se *wavelets* nas direções horizontal e vertical, que também são calculadas de maneira eficiente com a imagem integral.

Uma outra alternativa proposta foi o algoritmo *Features from Accelerated Segment Test* (FAST) (ROSTEN; DRUMMOND, 2006), que buscava obter um aumento de performance em velocidade grande o suficiente para aplicações em tempo real ou com poder computacional limitado. Diferente em natureza do SIFT e SURF, o algoritmo FAST possui essencialmente 2 etapas: em primeiro lugar, possíveis pontos de interesse são encontrados aplicando-se um *threshold* a uma vizinhança de cada pixel da imagem e comparando a intensidade luminosa desses pixels adjacentes; em seguida, aprendizado de máquina é utilizado para aumentar a robustez dos pontos de interesse selecionados, através do uso de uma árvore de decisão, seguido de um procedimento de eliminação de múltiplos pontos de interesse adjacentes. Entretanto, apesar de ser sensivelmente mais rápido, esse algoritmo não é tão robusto à presença de ruído e, além disso, sua performance depende da escolha do valor do *threshold*.

Mais recentemente, o algoritmo *Oriented FAST and Rotated BRIEF* (ORB) foi proposto em (RUBLEE et al., 2011), como uma tentativa de se combinar a robustez do algoritmo SIFT, porém mantendo o ganho de velocidade do algoritmo FAST. Para tanto, os autores em (RUBLEE et al., 2011) combinaram o detector de pontos de interesse do FAST, modificado de modo que uma orientação é atribuída a esses pontos, com uma nova versão do descritor local chamado *Binary Robust Independent Elementary Features* (BRIEF), apresentado originalmente em (CALONDER et al., 2010), para recuperar a invariância à rotação que se observava nos algoritmos SIFT e SURF. Uma análise e comparação aprofundadas da performance de alguns desses algoritmos podem ser encontradas em (GAUGLITZ; HÖLLERER; TURK, 2011).

Esses algoritmos citados servem para que pontos de interesse sejam selecionados em uma imagem de modo que, em seguida, seja possível realizar a correspondência entre os mesmos pontos em um par de imagens distintas, que em geral representam grandes variações de posição e rotação da câmera, o que é chamado de *wide-baseline matching*. Porém, no caso de *frames* de um vídeo, em geral, vale a hipótese que a vizinhança dos pontos de interesse não variam tanto e é possível aplicar uma correspondência do tipo *narrow-baseline matching*. Uma outra categoria de algoritmos desse tipo busca, então, encontrar esses pontos de interesse e acompanhar sua evolução a cada *frame* utilizando o que é conhecido como fluxo óptico, sendo o Kanade-Lucas-Tomasi (KLT) *feature tracker* (LUCAS; KANADE et al., 1981; TOMASI; KANADE, 1991; SHI et al., 1994) o de maior destaque entre eles.

Originalmente, Lucas e Kanade desenvolveram um algoritmo capaz de alinhar uma imagem de *tamplate* com uma imagem de entrada, através de um método de descida de gradiente responsável por encontrar o movimento entre uma imagem e outra que minimiza o erro quadrático entre a imagem *tamplate* observada e a imagem estimada pelo deslocamento, sendo possível assim acompanhar a evolução dos pixels em cada imagem. Como o cálculo do gradiente envolve a inversão da matriz Hessiana da imagem, é essencial para a estabilidade do método que ela seja bem-condicionada.

Partindo dessa observação, Tomasi e Shi propuseram um método para selecionar os pontos de interesse a serem acompanhados entre as imagens, levando em conta os autovalores da matriz Hessiana: quando um, outro ou ambos são muito maiores do que zero, isso significa que essa matriz é não singular e, portanto, inversível. Mais especificamente, o algoritmo seleciona pontos tais que os autovalores sejam maiores que um determinado valor mínimo, associado ao ruído presente nas imagens. Dado sua robustez e velocidade, o algoritmo KLT ganhou grande popularidade e segue apresentando performance de estado-da-arte.

## *2.2 Estimativa do movimento da câmera*

A estimativa da matriz de projeção de uma câmera partindo-se de duas imagens distintas da mesma cena também recebeu grande atenção desde sua introdução (LONGUET-HIGGINS, 1981), em que foi proposto o algoritmo de 8 pontos. Basicamente, a tarefa consiste em se encontrar a matriz fundamental $\mathbf{F}$ que satisfaz equação (4) para quaisquer par de pontos correspondentes em duas imagens. Como $\mathbf{F}$ possui 9 elementos e tem posto igual a 2, basta utilizar 8 pontos não-coplanares para formular um sistema linear e resolvê-lo, de modo a se determinar completamente a matriz.

Entretanto, esse problema só pode ser resolvido analiticamente no caso em que não há qualquer tipo de ruído nas imagens ou erro de correspondência entre os pontos, o que não se observa em imagens reais. Dessa forma, diferentes modificações foram propostas para o algoritmo original. Em (HARTLEY, 1997), Hartley propôs uma versão normalizada do algoritmo, que busca uma solução do tipo mínimos quadrados do sistema linear obtido, levando-se em conta que o problema em geral é mal-condicionado: a matriz do sistema, teoricamente, deveria possuir apenas um autovalor nulo e todos os outros diferentes de zero; entretanto, devido à má-distribuição das coordenadas homogêneas dos pixels das imagens, isso em geral não é verdade. Assim, propôs-se uma transformação de coordenadas para normalizar as coordenadas dos pontos, de modo a melhorar o condicionamento numérico da matriz do sistema, obtendo-se assim resultados mais robustos.

Além disso, uma análise teórica mais detalhada mostra que o problema pode ser resolvido com menos do que 8 pontos. Usando diretamente o fato que a matriz do sistema linear possui um autovalor nulo, um algoritmo usando 7 pontos foi proposto em (TORR; MURRAY, 1997). Além do interesse teórico em explorar a estrutura matemática do

problema, o uso de menos pontos na estimação da matriz fundamental possui interesses práticos. Em (NISTÉR, 2004), Nìster apresentou uma solução eficiente de estimação usando apenas 5 pontos, utilizada em um *pipeline* de SfM em tempo real, obtendo performance de estado da arte à época, com resultados mais velozes e robustos do que aqueles obtidos com algoritmos de 8 e 7 pontos. Entretanto, esse algoritmo exige que as câmeras estejam calibradas, de modo que todos os parâmetros intrínsecos devem ser conhecidos, o que não é o caso para os algoritmos que usam mais pontos. Uma alternativa foi, então, proposta em (STEWÉNIUS et al., 2008), através de um algoritmo que utiliza 6 pontos, mas que relaxa a condição sobre os parâmetros intrínsecos, uma vez que a distância focal é considerada desconhecida, de modo a encontrar um compromisso entre a robustez obtida com menos pontos e a flexibilidade da aplicação.

## *2.3 Reconstrução 3D*

A reconstrução da estrutura 3D da cena consiste em resolver o problema de otimização apresentado pela equação (5). De maneira geral, quando mais de duas imagens são utilizadas, é possível identificar duas categorias gerais de métodos de empregados: métodos sequenciais (ou incrementais) e métodos por *batch*.

Os métodos sequenciais são os mais comuns na literatura e, essencialmente, funcionam através da incorporação de novas imagens uma a uma, de modo que reconstruções parciais são feitas e aprimoradas a cada nova imagem adicionada. Diferentes estratégias podem ser adotadas nesse processo sequencial. Por exemplo, (HARTLEY, 1992) utiliza a informação 3D dos pontos já reconstruídos para estimar o movimento da câmera das novas imagens e continuar a reconstrução do novos pontos. Outra possibilidade é calcular duas reconstruções distintas, a partir de diferentes pares, e unir as soluções, através de pontos 3D em comum, como é feito em (FITZGIBBON; ZISSERMAN, 1998), em que sequências de imagens cada vez mais longas são reconstruídas de maneira hierárquica. Entretanto, métodos sequenciais apresentam algumas limitações, como a necessidade de grande superposição entre imagens, o uso de muitos pontos de interesse e a falta de robustez a determinados tipos de cenas ou movimentos.

A segunda categoria de métodos de reconstrução é conhecida como reconstrução por *batch*, uma vez que utiliza múltiplas imagens simultaneamente para realizar as estimativas de movimento e estrutura 3D da cena, de modo a diminuir o erro de reconstrução. Igualmente, existem diversas formas de realizar esse tipo de reconstrução: em (TOMASI; KANADE, 1992) foi introduzido o chamado método por fatorização, mas com a limitação de um modelo simplificado de câmera, que exclui movimentos mais gerais. Alguns trabalhos tentaram corrigir essa limitação, como o método apresentado em (STURM; TRIGGS, 1996) que utiliza uma técnica de rebalanceamento das escalas dos pontos das imagens, de modo a aumentar a robustez da reconstrução.

Finalmente, a grande maioria dos *pipelines* de SfM mais recentes utilizam uma técnica de refinamento da solução encontrada (tanto movimento da câmera, quanto estrutura 3D) chamada de *Bundle Adjustment* (BA). De modo geral, BA consiste em minimizar uma função de perda que leva em conta o erro de reprojeção obtido. Dessa forma, através do uso de algoritmos de otimização não-linear, como por exemplo variações do algoritmo de Gauss-Newton, é possível refinar a solução e encontrar matrizes de projeção e pontos tridimensionais mais adequados. Existe uma extensa literatura com as diferentes técnicas comumente empregadas para a realização desse refinamento e uma revisão bastante extensa pode ser encontrada em (TRIGGS et al., 1999).

## *2.4 SfM a partir de vídeo*

Grande parte da literatura de SfM se concentra no caso em que as entradas são imagens de uma cena obtidas de maneira discreta e espaçada. Assim, dado um conjunto de imagens, cada par formado apresenta chances de possuir muitos pontos em comum, ou absolutamente nenhum, de modo que, geralmente, é preciso checar todas as combinações. Além disso, existe sempre a possibilidade de que, mesmo para um par onde efetivamente há pontos em comum, ocorram potencialmente grandes diferenças fotométricas entre as imagens, com oclusões, diferença de iluminação, entre outras características.

Por outro lado, para o caso de *frames* de um vídeo, existem algumas particularidades que precisam ser levadas em conta: similaridade entre as imagens de entrada em *frames* adjacentes, a existência de uma relação sequencial entre as entradas, a necessidade de processar grandes volumes de dados rapidamente, o controle de acúmulo de erro, entre

outras características. Assim, uma literatura própria se desenvolveu para tratar dessa classe de problema.

Um dos primeiros trabalhos a receber destaque foi (TOMASI; KANADE, 1992), em que foi introduzido o que os autores chamam de método por fatorização, isto é, em que todas as imagens de um subconjunto (*batch*) são utilizadas ao mesmo tempo para realizar as estimativas de movimento da câmera e estrutura 3D, de modo a reduzir o erro de reconstrução, que é distribuído por todas as aquisições. Dessa forma, o método foi um dos primeiros que se mostraram robusto à presença de ruído nos *frames*, de modo a obter boa performance em sequências de imagens reais.

Outro trabalho de destaque é apresentado em (BEARDSLEY; TORR; ZISSERMAN, 1996), em que se buscava recuperar a estrutura 3D de uma cena através de uma longa sequência de imagens obtidas com uma câmera cujos parâmetros são desconhecidos. Para tanto, os autores propuseram um método sequencial, que reconstrói a estrutura 3D iterativamente, usando uma trinca de *frames* como unidade básica, ao invés de pares, mas com o mesmo princípio de funcionamento: pontos de interesse são encontrados nas imagens e a correspondência entre eles é realizada usando-se uma estratégia de correlação cruzada, assumindo que as mudanças em uma vizinhança são essencialmente estáticas; o tensor focal (no lugar da matriz fundamental) é estimado usando um algoritmo de 6 pontos, associado a um esquema do tipo RANSAC. De maneira similar, (FITZGIBBON; ZISSERMAN, 1998) usa trincas de imagens como ponto de partida para se obter estimativas do movimento da câmera e representa um dos primeiros trabalhos que consideram diferentes tipos de cena: interior e exterior, filmadas com um movimento controlado ou à mão livre.

Esses trabalhos iniciais buscam uma reconstrução 3D esparsa, porém trabalhos mais recentes tentam obter performances em tempo real também para reconstrução densa da cena. O método apresentado em (NEWCOMBE; DAVISON, 2010) utiliza um pipeline de SfM em tempo real introduzido em (KLEIN; MURRAY, 2007), que combina o acompanhamento de milhares de pontos de interesse por *frame* com a construção de um mapa da cena, responsável por aumentar a precisão das estimativas de deslocamentos dos pontos e de movimento da câmera, atingindo performance de estado da arte em cenas com limitação de profundidade e pouca variação de características. A partir dessas estimações robustas, a reconstrução densa é obtida então através da criação de uma malha com base na seleção de um conjunto de *frames* similares que se superpõem. Outro método de reconstrução 3D densa é apresentado em (PIZZOLI; FORSTER; SCARAMUZZA,

2014), que tenta diminuir as limitações de profundidade nas cenas de métodos como o anterior através do uso de métodos probabilísticos bayesianos, obtendo assim uma melhor performance em diferentes tipos de cena.

Além disso, com a melhora dos sistemas de aquisição, foram evidenciadas dificuldades para gerenciar vídeos com elevadas taxas de *frames* e com resoluções cada vez maiores. O método proposto em (RESCH et al., 2015) busca atender esses requisitos, obtendo performance robusta para *framerates* de 25-120Hz e com resoluções de 2-20 megapixels. Para tanto, os autores estabelecem um *pipeline* de SfM que combina o uso do algoritmo KLT entre *frames* adjacentes, com características SIFT entre *frames* espaçados da sequência, bem como utilizam métodos de refinamento intermediário e global para produzir uma estimativa robusta dos movimento e da estrutura.

Finalmente, é importante destacar que, ao longos dos anos e sobretudo mais recentemente, passou a haver uma intersecção natural entre os trabalhos de SfM a partir de um vídeo e aqueles de *Simultaneous localization and mapping* (SLAM), sendo possível encontrar trabalhos que tratam de maneira indistinta o problema de SLAM monocular e o de SfM em tempo real. Entretanto, como o problema geral de SLAM apresenta por si só uma vasta literatura que foge ligeiramente do escopo desse trabalho, ele não será detalhado.

# 3 *Pipeline* proposto

O *pipeline* desenvolvido nesse trabalho será apresentado nas seções a seguir. Em primeiro lugar, serão apresentados suas características gerais, bem como os objetivos e as justificativas para cada escolha de *design*. Em seguida, as etapas essenciais do pipeline serão detalhadas: o processamento preliminar dos dados e calibração da câmera; o algoritmo de detecção de pontos de interesse ao longo dos diferentes *frames* do vídeo; os diferentes aspectos da reconstrução incremental; e, finalmente, a etapa de otimização.

## *3.1 Visão geral*

Como mencionado anteriormente, o objetivo desse trabalho consiste em desenvolver um *pipeline* que seja capaz de reconstruir a estrutura tridimensional de uma cena estática, bem como os movimentos de translação e rotação da câmera, a partir de uma sequência de *frames* de um vídeo. O grande interesse de se desenvolver um *pipeline* desse tipo seria, futuramente, tentar utilizá-lo em aplicações de realidade aumentada, em que a sequência de *frames* seria capturada por dispositivos como *smartphones* ou óculos de realidade aumentada e, ao mesmo tempo, ela seria processada de forma a produzir para o usuário, em tempo real, as informações de estrutura 3D e movimentos recuperadas pelos algoritmos implementados. O interesse do trabalho apresentado poderá possivelmente se estender, portanto, para além da construção do *pipeline* em si, uma vez que também poderá ser eventualmente empregado como base para que se estabeleça uma análise da performance e limitações dessa abordagem e, assim, a viabilidade de sua utilização.

Dessa forma, mesmo que esse trabalho seja preliminar e anterior a essas aplicações futuras, possíveis requisitos impostos por elas foram considerados durante o desenvolvimento. Concretamente, buscou-se propor um *pipeline* que, conceitualmente, fosse capaz de funcionar em tempo real. Para tanto, não só foram escolhidos algoritmos de menor complexidade computacional nas diferentes etapas do processamento para que se tivesse um ganho em velocidade, mas principalmente foi necessário garantir que a reconstrução da cena pudesse ser feita de maneira sequencial, conforme novos *frames* fossem sendo adquiridos pela câmera. Por isso, abordagens que implicariam a necessidade de se processar o vídeo em sua integralidade para seu funcionamento foram descartadas. É importante

frisar, entretanto, que a implementação efetiva de uma versão em tempo real foge do escopo desse trabalho e que apenas foram feitos esforços para que uma eventual adaptação pudesse ser realizada de maneira mais simples.

Portanto, tendo em vista esses objetivos, um *pipeline* de *Structure from Motiom* incremental foi desenvolvido. Cada etapa será detalhada nas próximas seções, porém, de maneira geral, o *pipeline* pode ser dividido inicialmente em duas etapas principais: ***feature tracking***, que corresponde à detecção e ao acompanhamento de pontos de interesse entre os diferentes *frames* do vídeo; e **reconstrução incremental**, que permite efetivamente que seja criada sequencialmente uma nuvem de pontos 3D da cena observada, assim como a trajetória da câmera, com suas mudanças de orientação e posição ao longo dos *frames*, a partir da evolução desses pontos de interesse detectados.

Dada a natureza sequencial de um vídeo, com pequenos deslocamentos fotométricos entre os *frames*, a etapa de detecção e acompanhamento dos pontos de interesse foi realizada através do algoritmo KLT. Essencialmente, a implementação utiliza o detector de Shi-Tomasi para encontrar os pontos de interesse no primeiro *frame* útil do vídeo e uma versão do algoritmo de fluxo óptico de Lucas-Kanade, que utiliza representação de imagem em pirâmides, para acompanhá-los nos *frames* subsequentes. Além disso, de modo a ser possível tratar vídeos com duração arbitrária, conforme as *features* seguidas vão sumindo da cena, também foi proposta uma forma de reinicializá-las, de modo a se adicionar outras e, ao mesmo tempo, manter a coerência ao longo de todo vídeo. Os detalhes serão explicitados nas seções a seguir.

Uma vez detectados pontos de interesse nos diferentes *frames*, é possível realizar a reconstrução dos pontos 3D e da trajetória da câmera. Tendo em vista os objetivos desse trabalho, optou-se pelo que é conhecido como reconstrução incremental. Nesse tipo de abordagem, a partir de dois *frames* que sejam considerados bons, é preciso inicializar a nuvem de pontos e as posições da câmera. Isso é feito utilizando-se o algoritmo de 5 pontos para encontrar a matriz essencial e, consequentemente, as matrizes de projeção da câmera em cada um dos dois *frames*; através da triangulação dos pontos reprojetados por essas matrizes, é possível encontrar a nuvem de pontos 3D inicial. Como será mostrado posteriormente, uma inicialização adequada é fundamental para a obtenção de uma boa reconstrução.

Em seguida, conforme novos *frames* são adquiridos, os deslocamentos da câmera correspondentes a essas novas imagens, bem como eventuais novos pontos 3D que são

reconstruídos, são incorporados, respectivamente, na trajetória e na nuvem de pontos existentes até então. Isso é feito de maneira análoga à inicialização: a matriz essencial é estimada e em seguida, os pontos 3D são recuperados através de triangulação. Entretanto, como o algoritmo de 5 pontos fornece uma igualdade a menos de escala, simplesmente repetir o procedimento anterior forneceria apenas a direção de translação da câmera, bem como não garantiria que os pontos 3D estariam na mesma escala em diferentes reconstruções. Para resolver esses problemas é preciso introduzir uma etapa intermediária para recuperação do movimento da câmera, conhecido algoritmo *Perspective-n-Point* (PnP), em que a informação 3D reconstruída até então é utilizada para estimar a translação e rotação da câmera, de modo a manter a coerência de escala. Assim, o algoritmo de 5 pontos serve como estimativa inicial de movimento da câmera, que é corrigido em seguida pelo algoritmo PnP. A partir dessa estimativa do movimento em escala correta, a triangulação pode ser então realizada pra se obter os pontos 3D.

Finalmente, a última etapa do algoritmo representa um processo de otimização dos pontos 3D obtidos e, ao mesmo tempo, da trajetória da câmera, conhecido como *Bundle Adjustment*. Partindo-se da nuvem de pontos 3D e das matrizes de projeção da câmera estimadas ao longo da trajetória, é possível reprojetar esses pontos e comparar com o que foi observado originalmente nos *frames* do vídeo. Assim, definindo-se uma função objetivo que calcula o erro total dessas reprojeções, é possível estabelecer um problema de otimização, em que se busca minimizar esse erro para que sejam obtidos os pontos 3D e a trajetória da câmera que melhor explicam os dados observados. Essa etapa é fundamental para a correção de erros de estimação que inevitavelmente são introduzidos devido às mais diversas fontes de erro, sobretudo ruído nos *frames* que podem acarretar erros de correspondência entres os pontos de interesse seguidos.

O diagrama abaixo representa, portanto, um resumo do *pipeline* proposto e descrito brevemente acima. Cada uma das etapas será discutida em maiores detalhes nas próximas seções, ou seja, os problemas serão apresentados formalmente com sua formulação matemática correspondente e os algoritmos utilizados para resolvê-los serão descritos.

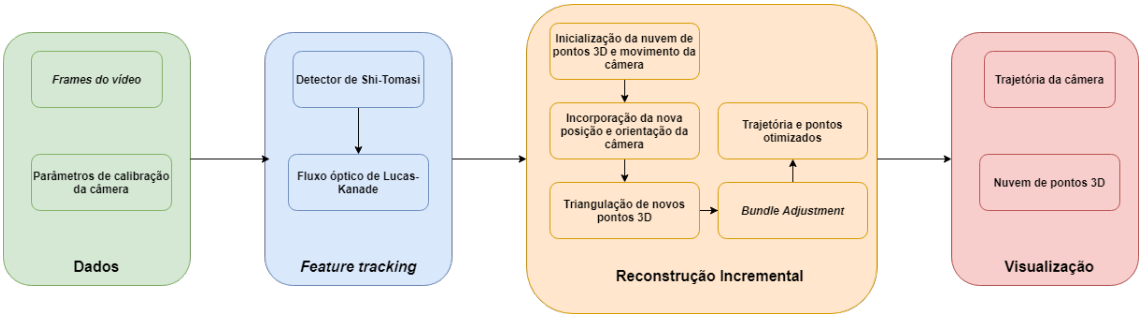



Figura 3.1 – Diagrama do *pipeline* de SfM a partir de um vídeo

## *3.2 Algoritmo KLT e feature tracking*

### 3.2.1 Descrição teórica

O algoritmo de Kanade-Lucas-Tomasi (KLT) para *feature tracking* é constituído de duas etapas fundamentais: em um *frame* inicial são detectados pontos de interesse usando o que é conhecido como detector de Shi-Tomasi e, em seguida, esses pontos são encontrados nos *frames* subsequentes através da estimativa do fluxo óptico pelo método de Lucas-Kanade, que permite encontrar o deslocamento dos pontos de interesse entre *frames*. Como o detector de Shi-Tomasi é baseado nas equações de Lucas-Kanade, é preciso entender inicialmente o conceito de fluxo óptico.

De maneira geral, o fluxo óptico representa a distribuição de velocidades aparentes dos elementos de uma imagem causados pelo movimento relativo entre a câmera e a cena observada. Dessa forma, sendo $I(u_x, u_y, t)$ a intensidade do pixel em $(x, y)$ no instante $t$, estimar o fluxo óptico entre dois *frames* observados com um intervalo de $\Delta t$ entre eles, significa encontrar essencialmente o campo de velocidades $(v_x, v_y)$ que transforma $I(u_x, u_y, t)$ em $I(u_x + \Delta u_x, u_y + \Delta u_y, t + \Delta t)$. O método de estimação de Lucas-Kanade pertence à categoria dos chamados métodos diferenciais, isto é, métodos que assumem pequenos deslocamentos $\Delta u_x$ e $\Delta u_y$ entre os dois *frames*.

Sob a hipótese diferencial, é possível aproximar em primeira ordem a nova intensidade $I(u_x + \Delta u_x, u_y + \Delta u_y, t + \Delta t)$ através da sua expansão em série de Taylor

$$I(u_x+\Delta u_x, u_y+\Delta u_y, t+\Delta t) = I(u_x, u_y, t)+\frac{\partial I}{\partial u_x}\Delta u_x+\frac{\partial I}{\partial u_y}\Delta u_y+\frac{\partial I}{\partial t}\Delta t+\mathcal{O}(\Delta u_x^2, \Delta u_y^2, \Delta t^2) \tag{6}$$

Assim, linearizando a equação em torno de $(x, y, t)$ tem-se

$$\frac{\partial I}{\partial u_x}\Delta u_x + \frac{\partial I}{\partial u_y}\Delta u_y + \frac{\partial I}{\partial t}\Delta t = 0 \tag{7}$$

e dividindo-a por $\Delta t$

$$\frac{\partial I}{\partial u_x}v_x + \frac{\partial I}{\partial u_y}v_y + \frac{\partial I}{\partial t} = 0 \tag{8}$$

de modo que definindo $\mathbf{v} = (v_x, v_y)$, $\nabla I = (I_x, I_y) = (\partial_x I, \partial_y I)$ e $I_t = \partial_t I$, pode ser reescrita de forma compacta como

$$\nabla I \cdot \mathbf{v} = -I_t \tag{9}$$

sendo que a expressão na forma acima é conhecida como equação do fluxo óptico.

É possível notar que, de maneira geral, conhecida a distribuição de intensidades nos dois *frames*, temos uma única equação com as duas incógnitas $v_x$ e $v_y$, de modo que é impossível estimar o fluxo óptico diretamente a partir de um único pixel e, portanto, é necessário introduzir alguma restrição extra para a resolução do problema. O método proposto por Lucas e Kanade assume, então, que o fluxo óptico é aproximadamente constante em uma vizinhança do pixel de interesse e resolve as equações (9) com um método do tipo mínimos quadrados para encontrar $v_x$ e $v_y$.

Mais explicitamente, dado uma vizinhança de pixels $\mathcal{U} = (\mathbf{u}_1, \mathbf{u}_2, ..., \mathbf{u}_n)$, sendo $\mathbf{u}_k = (x_k, y_k)$, o método assume que o fluxo óptico $\mathbf{v} = (v_x, v_y)$ é o mesmo para todos os pixels e, portanto, $\forall k \in \{1, 2, ..., n\}$ vale a equação do fluxo óptico

$$I_x(\mathbf{u}_k)v_x + I_y(\mathbf{u}_k)v_y = -I_t(\mathbf{u}_k) \tag{10}$$

de modo que escrevendo-se as $n$ equações obtidas em forma matricial, obtém-se

$$\begin{bmatrix} I_x(\mathbf{u}_1) & I_y(\mathbf{u}_1) \\ I_x(\mathbf{u}_2) & I_y(\mathbf{u}_2) \\ \vdots & \vdots \\ I_x(\mathbf{u}_n) & I_y(\mathbf{u}_n) \end{bmatrix} \begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} -I_t(\mathbf{u}_1) \\ -I_t(\mathbf{u}_2) \\ \vdots \\ -I_t(\mathbf{u}_n) \end{bmatrix} \tag{11}$$

que é um sistema de equações do tipo $\mathbf{Av} = \mathbf{b}$ sobre-determinado, isto é, com mais equações do que incógnitas.

Sendo assim, é possível resolver esses sistema de equações através do método de mínimos-quadrados. Para tanto, transforma-se a matriz $\mathbf{A}$ em matriz quadrada multiplicando-a por sua transposta $\mathbf{A}^\top\mathbf{A}\mathbf{v} = \mathbf{A}^\top\mathbf{b}$, de modo que $\mathbf{v} = (\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top\mathbf{b}$, ou seja

$$\begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} \sum_k I_x(\mathbf{u}_k)^2 & \sum_k I_x(\mathbf{u}_k)I_y(\mathbf{u}_k) \\ \sum_k I_y(\mathbf{u}_k)I_x(\mathbf{u}_k) & \sum_k I_y(\mathbf{u}_k)^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_k I_x(\mathbf{u}_k)I_t(\mathbf{u}_k) \\ -\sum_k I_y(\mathbf{u}_k)I_t(\mathbf{u}_k) \end{bmatrix} \tag{12}$$

o que permite recuperar os valores de $v_x$ e $v_y$ e, consequentemente, calcular o deslocamento de um pixel dessa vizinhança de um *frame* ao seguinte.

Dessa forma, para que o método de estimação do fluxo óptico de Lucas-Kanade funcione, é necessário que a matriz $\mathbf{S} = \mathbf{A}^\top\mathbf{A}$ seja inversível. Partindo-se dessa observação, Shi e Tomasi propuseram então um detector de pontos de interesse que garantiria um bom funcionamento do método de estimação do fluxo óptico, isto é, que impusesse a não-singularidade da matriz $\mathbf{S}$. De um ponto de vista puramente teórico, sendo $\mathbf{S} \in \mathbb{R}^{2\times2}$, ela possui dois autovalores $\lambda_1$ e $\lambda_2$, de modo que para ser inversível bastaria garantir que $\lambda_1, \lambda_2 > 0$. Entretanto, na prática, devido a presença de ruído, é necessário impor que os autovalores sejam superiores a um *threshold* $\lambda \in \mathbb{R}^+$, relacionado a esse nível de ruído, para garantir estabilidade numérica da matriz na hora da inversão. Além disso, outra exigência numérica para a inversibilidade da matriz seria seu bom condicionamento, isto é, que os dois autovalores possuíssem a mesma ordem de grandeza, de modo que $\lambda_1/\lambda_2 \sim 1$.

É interessante notar que $\mathbf{S} = (\nabla I)(\nabla I)^\top$, de modo que os autovalores $\lambda_1$ e $\lambda_2$ representam a magnitude do gradiente $(\nabla I)$ local da imagem em cada uma das duas direções. Mais precisamente, se $\lambda_1 > 0$ e $\lambda_2 = 0$, isso significa que a imagem só varia em uma única direção, aquela do autovetor associado a $\lambda_1$, enquanto tem variação nula na outra (e de modo análogo para $\lambda_1 = 0$ e $\lambda_2 > 0$). Em termos práticos, na imagem, isso se traduz pela existência de uma linha reta. Por outro lado, se $\lambda_1 = \lambda_2 = 0$, não há variação do gradiente em nenhuma direção e, assim, isso significa a existência de uma zona plana e continua na imagem. Finalmente, se $\lambda_1 = \lambda_2 > 0$, o gradiente varia igualmente em todas as direções e podemos associá-lo a presença de um vértice na região da imagem.

O detector de Shi-Tomasi consiste, então, em um detector de regiões com vértices na imagem. Para isso, basta percorrê-la por janelas de vizinhança, estimar a matriz $\mathbf{S}$ e calcular seus autovalores. Caso $\min(\lambda_1, \lambda_2) > \lambda$ e $\lambda_1/\lambda_2 \sim 1$, essa janela pode ser considerada como possuindo um vértice de interesse, cuja evolução ao longo dos *frames*

poderá ser bem estimada pelo método do fluxo óptico de Lucas-Kanade. Isto é, o pixel $\mathbf{u}_k$ de uma janela detectada no instante $t$ estará na posição $(u_x^k + v_x\Delta t, u_y^k + v_y\Delta t)$ no frame seguinte $t + \Delta t$. Assim, a cada *frame*, basta partir da posição anterior conhecida, calcular o fluxo óptico nas janelas detectadas usando as equações (12) e atualizar as novas posições.

### 3.2.2 Implementação piramidal

A hipótese fundamental para o funcionamento do algoritmo KLT descrito acima é a condição de pequenos deslocamentos, a partir da qual é possível obter a equação do fluxo óptico linearizada (9) e todas as outras relações que regem o funcionamento do algoritmo. Assim, seria necessário produzir um vídeo cujos deslocamentos entre *frames* fossem da ordem de menos de 1 pixel, o que não é viável considerando-se uma aquisição habitual de um vídeo, mesmo com *framerates* elevados. Para atacar esse problema e aumentar a robustez geral do algoritmo, em (BOUGUET et al., ) foi proposta uma implementação do método de estimação do fluxo óptico de Kanade-Lucas utilizando a representação da imagem em pirâmide.

De maneira geral, uma pirâmide de imagens consiste em uma sequência de réplicas dessa imagem que são geradas aplicando-se filtros de suavização e, em seguida, sub-amostrando a imagem resultante. O objetivo de tal representação é obter uma sequência de imagens com níveis decrescentes de resolução. Assim, partindo-se da imagem original $I$, de tamanhos $n_x$ e $n_y$ em cada uma das suas direções, e considerando-a como sendo o nível zero, isto é, $I^0 = I$, é possível construir recursivamente essa sequência de imagens $I^1, I^2, ..., I^L$ de modo que no nível $L$ tenha-se imagens com o mesmo conteúdo e de dimensões

$$n_x^L \leq \frac{n_x^{L-1} + 1}{2} \quad \text{e} \quad n_y^L \leq \frac{n_y^{L-1} + 1}{2} \tag{13}$$

isto é, grosso modo, as dimensões são divididas por 2 a cada nível.

O interesse desse tipo de abordagem é que o fluxo óptico pode ser estimado em qualquer um dois níveis superiores da pirâmide de imagem e, em seguida, propagado aos níveis inferiores e de maior resolução. Como mostrado no trabalho original, no nível $L$ da pirâmide, a relação entre o fluxo óptico nessa resolução $\mathbf{v}^L$ e o fluxo óptico real $\mathbf{v}$ é dado por:

$$\mathbf{v}^L = \frac{\mathbf{v}}{2^L} \tag{14}$$

o que significa que, para o típico valor de $L = 4$, a magnitude do fluxo óptico no quarto nível é 16 vezes menor do que na imagem com resolução original.

Portanto, esse tipo de abordagem permite recolocar vídeos com deslocamentos arbitrários entre *frames* nas condições de pequenos deslocamentos que são exigidas por hipótese pelo método de Lucas-Kanade, bastando-se aumentar o número de níveis da pirâmide calculados. Evidentemente, essa abordagem possui limitações e, iniciar estimativas a partir de imagens com resolução muito baixa pode acabar degradando o resultado final e, como apontado no trabalho original (BOUGUET et al., ), tipicamente reduções de 3 a 4 níveis são suficientes para a obtenção de um bom compromisso entre redução significativa da magnitude deslocamentos e qualidade da imagem para a estimativa.

### 3.2.3 Gerenciamento das *features*

Finalmente, um último problema com a implementação básica do KLT que precisou ser adaptado foi o gerenciamento e inclusão de novas *features* detectadas e acompanhadas. Isso porque, em sua versão inicial, o algoritmo em primeiro lugar detecta vértices de interesse no primeiro *frame* através do método de Shi-Tomasi e os acompanha nos *frames* seguintes, estimando seus deslocamentos através do fluxo óptico de Lucas-Kanade.

Entretanto, para vídeos mais longos, é possível que todas *features* detectadas no primeiro *frame* eventualmente saiam de quadro, de modo que se não for incluída nenhuma regra de atualização e inclusão de novas *features*, o restante da sequência de *frames* não terá pontos observados e, consequentemente, não será possível realizar as etapas seguintes de reconstrução. Dessa forma, foi necessário desenvolver essa regra de atualização e inclusão de novos pontos de interesse que passariam a ser acompanhados, de modo a manter a coerência entre os pontos antigos e os novos. Isso foi feito baseando-se em (ESCRIVá; MENDONçA, 2019), onde foi desenvolvido um método para se estabelecer essa correspondência de pontos

detectados a cada atualização.

Seja $\mathbf{L}_1$ a matriz cujas linhas $\mathbf{l}_1^1, \mathbf{l}_2^1, ..., \mathbf{l}_n^1$ representam as $n$ *features* acompanhadas inicialmente pelo KLT. Assim que dado o sinal de *reset*, uma nova matriz $\mathbf{L}_2$ será formada, com $m$ novas *features*. A correspondência entre $\mathbf{L}_1$ e $\mathbf{L}_2$ é feita, então, através da matriz $\mathbf{C}$, com $n$ linhas e $m$ colunas, cuja entrada $c_{ij}$ será dada por

$$c_{ij} = \begin{cases} 1, & \text{se } ||\mathbf{l}_i^1 - \mathbf{l}_j^2|| \leq \lambda_p \\ 0, & \text{caso contrário} \end{cases} \tag{15}$$

isto é, caso as *features* $\mathbf{l}_i^1$ e $\mathbf{l}_j^2$ estejam à uma distância em pixels menor que um determinado *threshold* $\lambda_p$ definido, elas são consideradas iguais; caso contrário, é possível afirmar que elas estão distantes o suficiente para serem classificadas como diferentes.

Em seguida, para cada coluna $j$ da matriz de correspondência, é possível calcular a soma $\sigma_j$ em todas as linhas de $\mathbf{C}$

$$\sigma_j = \sum_{i=1}^{n} c_{ij} \tag{16}$$

de modo a se obter um vetor $\mathbf{\Sigma} = [\sigma_1 \ \sigma_2 \ ... \ \sigma_m]$, cujas componentes nulas correspondem exatamente às *features* de $\mathbf{L}_2$ consideradas suficientemente distantes de todas as *features* já existentes em $\mathbf{L}_1$ e que, portanto, podem ser incorporadas.

Assim, a matriz $\mathbf{L}_1$ é incrementada com as linhas $j_k$ de $\mathbf{L}_2$, tais que $\sigma_{j_k} = 0$, formando uma nova matriz $\mathbf{L}_1'$ dada por

$$\mathbf{L}_1' = \begin{bmatrix} \mathbf{l}_1^1 \\ \vdots \\ \mathbf{l}_n^1 \\ \mathbf{l}_{j_1}^2 \\ \vdots \\ \mathbf{l}_{j_k}^2 \end{bmatrix} \tag{17}$$

É importante notar que, naturalmente, as *features* do KLT vão deixando de ser acompanhadas conforme vão sumindo do quadro e que, adicionalmente, o sinal de *reset* para incrementar a matriz ocorre quando o número de linhas fica abaixo de um certo limite inferior. Com isso, é possível garantir que o número de *features* acompanhadas esteja sempre aproximadamente constante, sem aumentar muito o número ou, inversamente, ficar sem nenhuma.

### 3.2.4 Etapa de *feature tracking* resultante

Dessa forma, a etapa de *feature tracking* pode ser resumida através dos seguintes passos descritos a seguir:

1. No primeiro *frame* a matriz $\mathbf{L}$ de *features* é inicializada com as coordenadas $\mathbf{u}_k = (u_x^k, u_y^k)$ dos vértices encontrados pelo detector de Shi-Tomasi;
2. Para os *frames* subsequentes, o fluxo óptico correspondente $\mathbf{v}_k = (v_x^k, v_y^k)$ de Lucas-Kanade é estimado em cada vizinhança dos vértices encontrados;
3. As posições $\mathbf{u}_k$ da matriz $\mathbf{L}$ são atualizadas com $\mathbf{u}'_k = (u_x^k + v_x^k \Delta t, u_y^k + v_y^k \Delta t)$. Caso a nova posição seja fora do quadro da câmera, a *feature* é descartada de $\mathbf{L}$;
4. Caso o número de *features* acompanhadas caia abaixo de um determinado valor, o detector de Shi-Tomasi é usado novamente e novas *features* são incorporadas à matriz $\mathbf{L}$ como descrito na seção anterior;
5. Os passos 2 a 4 são repetidos até que todos os *frames* do vídeo tenham sido processados.

## *3.3 Structure from Motion incremental*

De maneira formal, o *framework* de SfM incremental pode ser descrito da seguinte maneira: dado um conjunto de pixels de interesse $\mathcal{U} = \{\mathbf{u}_1, \mathbf{u}_2, ..., \mathbf{u}_k\}$, obtidos na etapa de *feature tracking* a partir de $m$ *frames* de matrizes de projeção $\mathcal{P} = \{\mathbf{P}_1, \mathbf{P}_2, ..., \mathbf{P}_m\}$ e $n$ pontos tridimensionais $\mathcal{X} = \{\mathbf{X}_1, \mathbf{X}_2, ..., \mathbf{X}_n\}$ inicialmente desconhecidos. O objetivo é estabelecer um *pipeline* capaz de produzir estimativas de matrizes e pontos conforme os *frames* vão sendo observados.

Isto é, após a inicialização da nuvem de pontos estimada $\hat{\mathcal{X}} = \{\hat{\mathbf{X}}_1, \hat{\mathbf{X}}_2, ..., \hat{\mathbf{X}}_r\}, r \leq n$, e das matrizes de projeção dos dois primeiros *frames* $\hat{\mathcal{P}} = \{\hat{\mathbf{P}}_1, \hat{\mathbf{P}}_2\}$, a cada frame observado, uma nova matriz de projeção necessariamente será estimada e incorporada no conjunto de estimativas $\hat{\mathcal{P}}$ e, caso algum ponto 3D novo seja reconstruído, ele também será incorporado no seu respectivo conjunto $\hat{\mathcal{X}}$.

Assim, no $k$-ésimo *frame* processado após a inicialização, teremos $\hat{\mathcal{P}} = \{\hat{\mathbf{P}}_1, \hat{\mathbf{P}}_2, ..., \hat{\mathbf{P}}_k\}$ e $\hat{\mathcal{X}} = \{\hat{\mathbf{X}}_1, \hat{\mathbf{X}}_2, ..., \hat{\mathbf{X}}_{r+l}\}, 0 \leq l \leq n - r$, com cada nova estimativa incorporada sendo feita apenas com base nas informações reconstruídas até então. Esse processo de incorporação de matrizes de projeção e pontos ocorre, então, até que todos os *frames* tenham sido processados. A seguir, cada uma das etapas necessárias para se estabelecer esse processo de reconstrução incremental serão detalhadas.

### 3.3.1 Inicialização

Na etapa de inicialização não existe nenhuma estrutura tridimensional já reconstruída e, assim, a única informação disponível é aquela dos pontos projetados 2D e suas correspondências entre dois *frames*. Como mencionado anteriormente, esse problema pode ser resolvido então utilizando-se a restrição epipolar da equação (4), o que é feito através de uma classe de algoritmos conhecidos como algortimos de $n$ pontos. Devido a maior precisão, menor sensibilidade ao ruído e por não sofrer do problema conhecido como degeneração planar (basicamente, existência de ambiguidades na solução da restrição epipolar) o algoritmo de 5 pontos introduzido por Nistér em (NISTÉR, 2004) foi utilizado.

De maneira simplificada, partindo-se da equação epipolar entre os pixels correspondentes $\tilde{\mathbf{u}}$ e $\tilde{\mathbf{u}}'$ de dois *frames* distintos

$$\tilde{\mathbf{u}}^\top (\mathbf{K}^{-1})^\top \mathbf{E} \mathbf{K}^{-1} \tilde{\mathbf{u}}' = 0 \tag{18}$$

em que $\mathbf{K}$ é a matriz de calibração da câmera suposta conhecida, o algoritmo de 5 pontos busca estimar a matriz essencial $\mathbf{E}$ considerando duas restrições adicionais sob sua estrutura

$$\begin{aligned} \det(\mathbf{E}) &= 0 \\ \mathbf{E}\mathbf{E}^\top \mathbf{E} - \frac{1}{2}\mathrm{tr}(\mathbf{E}\mathbf{E}^\top)\mathbf{E} &= 0 \end{aligned} \tag{19}$$

demonstradas, por exemplo, em (FAUGERAS; FAUGERAS, 1993). Essas restrições diminuem os 9 graus de liberdade originais para apenas 5, de modo que teoricamente apenas 5 pontos correspondentes entre as imagens precisam ser utilizados. Na prática, mais do que 5 correspondências são utilizadas de modo a se aumentar a robustez da solução, o que gera um sistema de equações sobre-determinado e, portanto, garante uma solução no sentido de mínimos-quadrados.

A equação (18) pode ser vista, basicamente, como um sistema linear homogêneo nos parâmetros da matriz essencial, cuja solução pode ser encontrada, portanto, por decomposição em valores singulares (SVD). Assim, dado o sistema de $m \times 9$ equações introduzido por (18), com $m \geq 5$, dependendo do número de correspondências utilizado, sua solução será uma combinação linear das 4 matrizes $\mathbf{M}_i$ associadas aos 4 valores singulares de menor magnitude:

$$\mathbf{E} = \sum_{i=1}^{4} \alpha_i \mathbf{M}_i \tag{20}$$

e como na prática as estimativas realizadas são obtidas a menos de um fator de escala, é possível impor $\alpha_4 = 1$ e reescrever a equação em termo dos coeficientes restantes $\tilde{\alpha}_i = \alpha_i / \alpha_4$

$$\mathbf{E} = \tilde{\alpha}_1 \mathbf{M}_1 + \tilde{\alpha}_3 \mathbf{M}_3 + \tilde{\alpha}_3 \mathbf{M}_3 + \mathbf{M}_4 \tag{21}$$

Assim, substituindo-se (21) em (19), obtém-se um sistema de equações polinomiais nos coeficientes $\tilde{\alpha}_i$, cuja solução permite determinar completamente $\mathbf{E}$. Nistér propõe, então, um método eficiente para encontrar as raízes desse sistema de equações polinomiais, cujo detalhamento foge do escopo desse texto e pode ser consultado em seu trabalho original (NISTÉR, 2004).

Uma vez encontrada a matriz essencial, é necessário recuperar a matriz de rotação $\mathbf{R}$ e o vetor de translação $\mathbf{t}$ associados ao movimento que leva a câmera de um *frame* ao seguinte, usando a definição da matriz essencial apresentada anteriormente

$$\mathbf{E} \sim [\mathbf{t}]_{\times}\mathbf{R} \tag{22}$$

em que $[\mathbf{t}]_{\times}$ é a matriz de produto externo associada ao vetor de translação e dada por

$$[\mathbf{t}]_{\times} = \begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix} \tag{23}$$

Isso é feito realizando-se a decomposição em valores singulares da matriz essencial $\mathbf{E} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\top}$, com $\mathbf{\Sigma} = \text{diag}(1, 1, 0)$ e $\mathbf{U}$ e $\mathbf{V}$ tais que $\det(\mathbf{U}) > 0$ e $\det(\mathbf{V}) > 0$. Como mostrado em (HARTLEY; ZISSERMAN, 2003), definindo-se

$$\mathbf{D} = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{24}$$

tem-se que $\mathbf{t} \sim \mathbf{t}_u = [u_{13}\ u_{23}\ u_{33}]^{\top}$ e $\mathbf{R} \sim \mathbf{R}_a = \mathbf{U}\mathbf{D}\mathbf{V}^{\top}$ ou $\mathbf{R} \sim \mathbf{R}_b = \mathbf{U}\mathbf{D}^{\top}\mathbf{V}^{\top}$. Consequentemente, qualquer uma das combinações de translação e rotação anteriores produz uma matriz essencial que satisfaz a equação epipolar (18).

Para resolver essas ambiguidades e encontrar uma solução única, é preciso fixar a matriz de projeção do primeiro *frame* como sendo $\mathbf{P} \sim K[\mathbf{I}_3 \mid \mathbf{0}]$ e arbitrar uma translação $\mathbf{t}_u$ unitária entre os *frames*. Com isso, existem basicamente 4 possibilidades para a matriz de projeção do segundo *frame*: $\mathbf{P}'_A \sim K[\mathbf{R}_a \mid \mathbf{t}_u]$, $\mathbf{P}'_B \sim K[\mathbf{R}_a \mid -\mathbf{t}_u]$, $\mathbf{P}'_C \sim K[\mathbf{R}_b \mid \mathbf{t}_u]$, $\mathbf{P}'_D \sim K[\mathbf{R}_b \mid -\mathbf{t}_u]$, sendo apenas uma delas correspondente ao movimento real. A outra solução representa aquela obtida se uma das projeções for rotacionada de $180^o$ e as duas restantes são as soluções espelhadas correspondentes às duas anteriores.

A escolha da solução correta deve ser feita utilizando o chamado *cheirality check* que, basicamente, a partir de um ponto 3D triangulado usando uma das possíveis soluções, permite checar qual matriz de projeção garante que o ponto esteja de frente com a câmera e na orientação projetada correta (NISTÉR, 2004). Como resultado, obtém-se portanto as duas primeiras matrizes de projeção estimadas $\hat{\mathbf{P}}_1 \sim K[\mathbf{I}_3 \mid \mathbf{0}]$ e $\hat{\mathbf{P}}_2 \sim K[\mathbf{R}_{1\to2} \mid \mathbf{t}_{1\to2}]$, com $||\mathbf{t}_{1\to2}|| = 1$

Estimadas a matrizes de projeção, a última etapa da inicialização consiste em triangular os pixels para obter os pontos tridimensionais correspondentes. O algoritmo de triangulação básico, descrito em (HARTLEY; ZISSERMAN, 2003), é conhecido como Direct Linear Transform (DLT) e consiste, novamente, na solução de um sistema linear homogêneo sobre-determinado, através de uma decomposição em valores singulares da matriz do sistema.

Esse sistema é obtido da seguinte forma: dado um ponto tridimensional $\mathbf{X}$ e suas projeções em dois *frames* distintos $\mathbf{u} = [u\ v]$ e $\mathbf{u'} = [u'\ v']$, tem-se que $\mathbf{u} = \mathbf{PX}$ e $\mathbf{u'} = \mathbf{P'X}$. Realizando um produto vetorial dos dois lados de cada equação de projeção, obtém-se então $\mathbf{u} \times (\mathbf{PX}) = \mathbf{0}$ e $\mathbf{u'} \times (\mathbf{P'X}) = 0$, que são equações lineares nas componentes de $\mathbf{X}$ da forma:

$$\begin{aligned} (u\mathbf{p}^{3\top} - \mathbf{p}^{1\top})\mathbf{X} = 0 \\ (v\mathbf{p}^{3\top} - \mathbf{p}^{2\top})\mathbf{X} = 0 \\ (u\mathbf{p}^{2\top} - v\mathbf{p}^{1\top})\mathbf{X} = 0 \end{aligned} \tag{25}$$

Como a terceira equação é uma combinação linear das duas primeiras, uma vez que temos $u\mathbf{p}^{2\top} - v\mathbf{p}^{1\top} = -u(v\mathbf{p}^{3\top} - \mathbf{p}^{2\top}) + v(u\mathbf{p}^{3\top} - \mathbf{p}^{1\top})$, cada par de pontos projetados correspondentes fornece 4 equações que podem ser escritas na forma homogênea $\mathbf{AX} = 0$

$$\mathbf{A} = \begin{bmatrix} u\mathbf{p}^{3\top} - \mathbf{p}^{1\top} \\ v\mathbf{p}^{3\top} - \mathbf{p}^{2\top} \\ u'\mathbf{p}'^{3\top} - \mathbf{p}'^{1\top} \\ v'\mathbf{p}'^{3\top} - \mathbf{p}'^{2\top} \end{bmatrix} \tag{26}$$

cuja solução $\hat{\mathbf{X}}$ corresponde ao vetor associado ao menor valor singular de $\mathbf{A}$ e, portanto, fornece a melhor estimativa de $\mathbf{X}$ no sentido de mínimos-quadrados.

Assim, considerando-se $r$ pontos tridimensionais $\mathbf{X}_1, \mathbf{X}_2, ..., \mathbf{X}_r$ e seus respectivos pares de projeção nos dois *frames* iniciais $\mathbf{u}_{11}, \mathbf{u}_{12}, \mathbf{u}_{21}, \mathbf{u}_{22}, ..., \mathbf{u}_{r1}, \mathbf{u}_{r2}$, em que $\mathbf{u}_{ij}$ representa o i-ésimo ponto tridimensional projetado no j-ésimo *frame*, obtidos na etapa de *feature tracking*, bem como as matrizes de projeção $\hat{\mathbf{P}}_1$ e $\hat{\mathbf{P}}_2$ estimadas anteriormente pelo algoritmo de 5 pontos, basta aplicar o algoritmo DLT descrito acima para que se obtenha então a estimativa dos $r$ pontos 3D triangulados $\hat{\mathbf{X}}_1, \hat{\mathbf{X}}_2, ..., \hat{\mathbf{X}}_r$

### 3.3.2 Refinamento da inicialização e incorporação de novos *frames*

Uma vez inicializadas as estimativas da nuvem de pontos 3D $\hat{\mathcal{X}} = \{\hat{\mathbf{X}}_1, \hat{\mathbf{X}}_2, ..., \hat{\mathbf{X}}_r\}$ e das matrizes de projeção *frames* $\hat{\mathcal{P}} = \{\hat{\mathbf{P}}_1, \hat{\mathbf{P}}_2\}$, o próximo passo consiste na adição incremental de novos *frames* $\hat{\mathbf{P}}_2$ para a composição de trajetória da câmera e, em seguida, a triangulação de eventuais novos pontos $\hat{\mathbf{X}}_{r+l}$ nunca antes vistos na cena.

De maneira análoga à inicialização, a estimativa da matriz de projeção poderia ser feita através do algoritmo de 5 pontos descrito anteriormente. Entretanto, como mostrado, a solução do algoritmo impõe $\hat{\mathbf{P}}_{k-1} \sim K[\mathbf{I}_3 \mid \mathbf{0}]$ e $\hat{\mathbf{P}}_k \sim K[\mathbf{R}_{(k-1)\to k} \mid \mathbf{t}_{(k-1)\to k}]$, sendo $||\mathbf{t}_{(k-1)\to k}|| = 1$. Isso significa que só seria possível recuperar a direção do movimento relativo entre os *frames*, sendo impossível distinguir translações de magnitudes diferentes e, consequentemente, a trajetória recuperada não corresponderia a realidade.

Uma forma de contornar esse problema é utilizar o algoritmo de *Perspective-n-Points*, que utiliza a informação da nuvem 3D já reconstruída para estimar a matriz de projeção, uma vez que baseia-se na correspondência entre os pontos tridimensionais e os pixels observados. Com a nuvem 3D inicializada na etapa anterior, esse tipo de abordagem torna-se viável. Partindo-se dos $r$ pontos tridimensionais em $\hat{\mathcal{X}}$, que estão expressos em um sistema de referência global, cada projeção do i-ésimo ponto no j-ésimo *frame* $\mathbf{u}_{ij} = [u_x^{ij}\ u_y^{ij}]$ será dada pela chamada equação de colinearidade

$$\begin{aligned} u_x^{ij} &= \frac{(\mathbf{r}_1^j)^\top \mathbf{X}_i + t_x^j}{(\mathbf{r}_3^j)^\top \mathbf{X}_i + t_z^j} \\ u_y^{ij} &= \frac{(\mathbf{r}_2^j)^\top \mathbf{X}_i + t_y^j}{(\mathbf{r}_3^j)^\top \mathbf{X}_i + t_z^j} \end{aligned} \tag{27}$$

sendo $\mathbf{r}_k^j$ a k-ésima linha da matriz de rotação $\mathbf{R}_j$ do j-ésimo *frame*

$$\mathbf{R}_j = \begin{bmatrix} \mathbf{r}_1^j \\ \mathbf{r}_2^j \\ \mathbf{r}_3^j \end{bmatrix} \tag{28}$$

e $\mathbf{t}_j = [t_x\ t_y\ t_z]$ o seu vetor de translação associado. Dessa forma, conhecidos os pontos e suas projeções correspondentes, as equações acima fornecem uma forma de achar os parâmetros de rotação e translação desconhecidos.

Como mostrado em (LEPETIT; MORENO-NOGUER; FUA, 2009), métodos iterativos garantem as melhores performances nessa tarefa. Essencialmente, partindo-se de uma estimativa inicial para $\mathbf{R}_j$ e $\mathbf{t}_j$ é possível definir a seguinte função objetivo

$$f(\mathbf{R}_j, \mathbf{t}_j) = \sum_{i=1}^{r} \left[ \left( u_x^{ij} - \frac{(\mathbf{r}_1^j)^\top \mathbf{X}_i + t_x^j}{(\mathbf{r}_3^j)^\top \mathbf{X}_i + t_z^j} \right)^2 + \left( u_y^{ij} - \frac{(\mathbf{r}_2^j)^\top \mathbf{X}_i + t_y^j}{(\mathbf{r}_3^j)^\top \mathbf{X}_i + t_z^j} \right)^2 \right] \tag{29}$$

que representa, essencialmente, o erro de reprojeção do ponto 3D no *frame*. Definindo $\mathbf{\Theta} = (\mathbf{R}_j, \mathbf{t}_j)$, a solução ótima para o problema $\hat{\mathbf{\Theta}} = (\hat{\mathbf{R}}_j, \hat{\mathbf{t}}_j)$ será dada minimizando esse erro de reprojeção

$$\hat{\mathbf{\Theta}} = \arg\min_{\mathbf{\Theta}} f(\mathbf{\Theta}) \tag{30}$$

Usualmente, o método de otimização empregado nessa classe de problemas é conhecido como Levenberg-Marquardt, ou mínimos-quadrados amortecido, que será explicitado na próxima seção. Como será visto, sendo uma otimização não-linear, a qualidade da solução depende fortemente da estimativa inicial. Uma possibilidade para uma boa inicialização do problema, sobretudo para a matriz de rotação $\mathbf{R}_j$, é a utilização do algoritmo de 5 pontos apresentado anteriormente. Entretanto, sendo o vetor de translação unitário, ele pode estar muito longe de um bom chute inicial, principalmente no caso em que há um grande deslocamento da câmera entre *frames*.

De modo a se obter estimativas mais robustas, portanto, foi utilizado o algoritmo conhecido como *Efficient Perspective-n-Points* (EPnP), proposto em (LEPETIT; MORENO-NOGUER; FUA, 2009). Essencialmente, o método representa um algoritmo não-iterativo de complexidade linear capaz de resolver o problema de recuperação das matrizes de rotação e translação a partir de pontos 3D e suas respectivas projeções. O detalhamento do método foge do escopo desse trabalho e pode ser consultado em sua publicação original. De toda forma, os autores mostram que a combinação desse método com uma otimização simples como a descrita acima permite que se obtenham resultados comparáveis ao estado-da-arte que algoritmos iterativos mais complexos produzem. Dessa forma, dado o equilíbrio entre performance e velocidade de processamento, esse método foi escolhido.

O resultado da otimização permite, portanto, estimar a matriz de projeção do j-ésimo *frame* $\hat{\mathbf{P}}_j \sim \mathbf{K}[\hat{\mathbf{R}}_j \mid \hat{\mathbf{t}}_j]$ e, consequentemente, triangular eventuais novos pontos tridimensionais $\hat{\mathbf{X}}$, conforme descrito na seção anterior, incrementando, assim, a cada *frame*, os conjuntos de estimativas $\hat{\mathcal{P}}$ e $\hat{\mathcal{X}}$ sequencialmente.

É importante notar que, ao utilizar essa abordagem, todos os *frames* serão processados baseando-se nas informações tridimensionais recuperadas, essencialmente, nos dois primeiros, de modo que é preciso garantir que o primeiro passo garantirá um bom comportamento do *pipeline* ao longo do resto do vídeo. Assim, caso necessário, é possível refinar a inicialização com alguns dos *frames* processados com o método descrito acima e, em seguida, avaliar a evolução do erro de reconstrução para garantir que a inicialização foi adequada e, caso contrário, recomeçar o processo.

Mais precisamente, uma vez feita a inicialização como descrita na seção anterior, é possível processar $k$ *frames* utilizando o PnP como descrito acima e, por fim, aplicando a otimização global que será detalhada na próxima seção. De um ponto de vista teórico, isso significa que os pontos reconstruídos e as $k + 2$ matrizes de projeção estimadas são tais que minimizam o erro empírico de reprojeção. Como a função objetivo depende da quantidade de pontos observados, é possível que a solução encontrada corresponda a um mínimo local e específico desses *frames* iniciais, de modo que não necessariamente causará uma melhora nos resultados dos demais *frames* seguidos.

Para lidar com esse problema, outros $p$ *frames* seguintes são processados como anteriormente, mas com a nova nuvem de pontos otimizada, e o erro de reprojeção de cada um deles é calculado. Caso o erro médio fique abaixo de um determinado *threshold*, a inicialização e demais reconstruções realizadas até então são consideradas satisfatórias e os demais *frames* são processados e as respectivas matrizes de projeção e pontos 3D são incluídos normalmente nos seus respectivos conjuntos. Caso contrário, a inicialização é considerada ruim e desloca-se de um a janela dos $k + p + 2$ *frames* processados, jogando-se fora, portanto, o primeiro e reiniciando-se as reconstruções com os demais. Esse processo é repetido até que a condição de erro mínimo seja satisfeita, momento a partir do qual o algoritmo segue normalmente como explicado até que se terminem os *frames* a serem processados.

### 3.3.3 *Bundle Adjustment*

A última etapa do *pipeline* consiste na otimização conjunta dos pontos tridimensionais e das matrizes de projeção estimadas, num processo conhecido como *Bundle Adjustment* (BA). Essa etapa deve ser entendida como um refinamento global de $\hat{\mathcal{P}}$ e $\hat{\mathcal{X}}$, diferindo portanto dos demais processos iterativos e de otimização descritos, uma vez que busca encontrar o parâmetros da câmera e os pontos que melhor descrevem os dados observados em sua totalidade.

Formalmente, dado o conjunto de parâmetros $\mathbf{\Theta} = (\hat{\mathcal{P}}, \hat{\mathcal{X}})$, formado pelas estimativas iniciais das matrizes de projeção e dos pontos reconstruídos, bem como os pixels projetados originais $\mathcal{U}$, o BA pode ser definido como o seguinte problema de encontrar $\mathbf{\Theta}^\star$ tal que

$$\mathbf{\Theta}^\star = \arg\min_{\mathbf{\Theta}} \mathcal{L}(\mathcal{U}, \mathbf{\Theta}) \tag{31}$$

sendo $\mathcal{L}$ uma função que fornece uma medida do erro empírico, ou resíduo, entre o modelo utilizando os parâmetros estimados $\mathbf{\Theta}$ e os dados observados $\mathcal{U}$.

Tipicamente, considera-se $\mathcal{L}$ como sendo a soma do erro quadrático entre os pixels observados $\mathbf{u}_{ij}$ e os reprojetados a partir dos parâmetros estimados $\hat{\mathbf{u}}_{ij}(\mathbf{\Theta}) = \hat{\mathbf{P}}_j \hat{\mathbf{X}}_i$, de forma análoga à equação (29), que pode ser reescrita de forma mais sucinta como

$$\mathcal{L}(\mathcal{U}, \mathbf{\Theta}) = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{m} \left[ (u_{ij} - \hat{u}_{ij}(\mathbf{\Theta}))^2 + (v_{ij} - \hat{v}_{ij}(\mathbf{\Theta}))^2 \right] = \frac{1}{2} ||\mathbf{r}(\mathbf{\Theta})||^2 \tag{32}$$

em que $||.||$ representa a norma $l^2$ do chamado vetor de resíduos $\mathbf{r}(\mathbf{\Theta})$, que possui, em cada uma das suas coordenadas, o erro de reprojeção para cada um dos pixels observados, o que é dado por $\mathbf{r}(\mathbf{\Theta}) = [\mathbf{u}_{11} - \hat{\mathbf{u}}_{11}(\mathbf{\Theta}), ..., \mathbf{u}_{nm} - \hat{\mathbf{u}}_{nm}(\mathbf{\Theta})]$.

O método de otimização comumente empregado na resolução do problema acima é chamado algoritmo de Levenberg-Marquardt (LM) (MORÉ, 1978), que essencialmente representa uma combinação de dois algoritmos de otimização: método de Gauss-Newton e o método de descida de gradiente. Nesses dois métodos, a ideia central consiste em encontrar uma regra iterativa de atualização dos parâmetros do tipo $\mathbf{\Theta}^{k+1} = \mathbf{\Theta}^k + \mathbf{h}^k$, que faça $\mathbf{\Theta}$ convergir para $\mathbf{\Theta}^\star$ a partir de uma estimativa inicial $\mathbf{\Theta}^0$.

No caso do algoritmo de descida do gradiente, a regra de atualização na k-ésima iteração $\mathbf{h}^k$ é feita na direção oposta do gradiente de $\mathcal{L}(\mathcal{U}, \mathbf{\Theta})$ em relação aos parâmetros $\mathbf{\Theta}^k$ obtidos até então. Isso porque o gradiente de uma função representa sua direção

de maior variação e, ao se caminhar em sua direção oposta, busca-se uma região que representa um mínimo ao menos local. Temos então que

$$\mathbf{h}_{dg} = -\frac{\partial \mathcal{L}}{\partial \mathbf{\Theta}} = -\frac{\partial \mathbf{r}}{\partial \mathbf{\Theta}}\mathbf{r}(\mathbf{\Theta}) \tag{33}$$

em que $\partial \mathbf{r}/\partial \mathbf{\Theta}$ é obtida derivando-se cada um dos resíduos em relação a cada um dos parâmetros. Isso define uma matriz que possui $P = n \times m$ linhas, isto é, uma por pixel observado, e $Q = \dim(\Theta)$ colunas, isto é, uma para cada parâmetro da otimização. Essa matriz recebe o nome de matriz jacobina $\mathbf{J}$ cujas entradas são dadas por

$$J_{pq}(\mathbf{\Theta}) = \frac{\partial r_p}{\partial \Theta_q}(\mathbf{\Theta}) \tag{34}$$

de forma que, no caso do método da descida de gradiente, tem-se, finalmente

$$\mathbf{h}_{dg} = -\mathbf{J}\mathbf{r}(\mathbf{\Theta}) \tag{35}$$

o que evidencia que é um método de primeira ordem, em que apenas o gradiente da função objetivo precisa ser calculado, que apresenta baixa complexidade computacional e boas propriedades de convergência para funções objetivas mais simples. Porém, para problemas mais complexos, a convergência pode demorar a ocorrer e se tornar proibitiva.

O algoritmo de Gauss-Newton, por outro lado, representa um algoritmo de ordem superior, uma vez que sua regra de atualização depende de derivadas de segunda ordem. Ele baseia-se na hipótese de que o vetor de resíduos é aproximadamente quadrática perto do ótimo. Como mostrado em (MORÉ, 1978), isso significa que

$$[\mathbf{J}^\top \mathbf{J}]\mathbf{h}_{gn} = -\mathbf{J}\mathbf{r}(\mathbf{\Theta}) \tag{36}$$

e devido à existência de uma inversão de matrizes, o algoritmo de Gauss-Newton é relativamente mais complexo, porém garante convergência robusta de uma maior classe de problemas.

O algoritmo LM introduz um parâmetro de amortecimento $\lambda$ que faz com que a otimização seja interpolada entre a descida de gradiente e o método de Gauss-Newton. Essencialmente, sua direção de otimização é dada por

$$[\mathbf{J}^\top \mathbf{J} + \lambda \mathbf{I}]\mathbf{h}_{lm} = -\mathbf{J}\mathbf{r}(\mathbf{\Theta}) \tag{37}$$

em que no início, $\lambda$ é grande de modo a favorecer a descida de gradiente e, conforme o algoritmo se aproxima de um mínimo, seu valor é aumentado aos poucos, de modo a acelerar a convergência usando o método de Gauss-Newton.

Além de utilizar o algoritmo LM, para que seja computacionalmente viável aplicar o BA em larga escala, isto é, com potencialmente centenas ou milhares de *frames* e, ainda assim, obter resultados rápidos, é possível utilizar a estrutura de matrizes esparsas para armazenar o jacobiano, uma vez que a grande maioria das derivadas parciais será igual a zero. Isso porque cada componente do vetor de resíduos $\mathbf{u}_{ij} - \hat{\mathbf{u}}_{ij}(\mathbf{\Theta})$ na verdade depende apenas dos 6 parâmetros de $\hat{\mathbf{P}}_j$ (3 rotações e 3 translações) e dos 3 parâmetros de $\hat{\mathbf{X}}_i$.

Isto é, para $n$ pontos 3D vistos todos em $m$ *frames*, a matriz jacobiana possui $Q = 6m + 3n$ colunas, das quais apenas 9 são não-nulas por linha. Dessa forma, é possível guardar apenas as entradas não-nulas do jacobiano, o que não só reduz a complexidade em memória do método, mas principalmente acelera as operações matriciais do jacobiano envolvidas no cálculo da direção de otimização $\mathbf{h}_{lm}$.

O algoritmo resultante é conhecido, portanto, como *Sparse Bundle Adjustment* (SBA) e é particularmente adaptado para aplicações com um grande volume de dados que precisam ser processados de maneira otimizada, o que é o caso dos *frames* de um vídeo. É importante notar que essa etapa pode ser aplicada tanto ao longo do processamento do vídeo, ao se acumular um determinado número de *frames*, quanto no final quando todos os *frames* já foram adquiridos.

### 3.3.4 Etapa de reconstrução incremental resultante

Resumidamente, a reconstrução incremental pode ser, então, descrita através da sequência de passos a seguir:

1. Dados dois *frames* iniciais, o algoritmo de 5 pontos é utilizado para se se estimar as respectivas matrizes de projeção $\hat{\mathbf{P}}_1$ e $\hat{\mathbf{P}}_2$. Com elas, o algoritmo DLT é aplicado para triangular os $r$ pontos 3D observados $\hat{\mathbf{X}}_1, \hat{\mathbf{X}}_2, ..., \hat{\mathbf{X}}_r$ e inicializar a nuvem de pontos;
2. A inicialização é, então, refinada utilizando-se a nuvem de pontos inicial para estimar as matrizes de projeção dos $k$ *frames* seguintes através do algoritmo EPnP, cuja solução é refinada por uma otimização com o algoritmo LM;

3. O erro de reconstrução é então calculado ao se obter a matriz de projeção dos $p$ *frames* subsequentes: caso esse erro seja menor do que um *threshold*, a inicialização é considerada adequada e todas as $k + p + 2$ matrizes de projeção estimadas são mantidas, bem como eventuais novos pontos 3D são adicionados; caso contrário, os passos 1 e 2 são repetidos deslocando-se um *frame* da janela até que o *threshold* seja atendido;
4. Os *frames* seguintes são todos processados utilizando-se EPnP, seguido do refinamento iterativo, para se estimar as novas matrizes de projeção e, em seguida, triangular eventuais novos pontos para serem incluídos na nuvem existente;
5. A solução é refinada utilizando-se o algoritmo de *Sparse Bundle Adjustment* para otimizar as posições da câmera e os pontos 3D reconstruídos. Dependendo da configuração escolhida, essa etapa pode ser feita a cada n *frames* processados pelo passo 4 ou apenas no fim do vídeo.

## *3.4 Processamento dos dados*

Além das duas grandes etapas do *pipeline* proposto, *feature tracking* e SfM incremental, existem etapas de pré e pós-processamento dos dados que são necessárias para o funcionamento do método proposto. Mais especificamente, tanto o algoritmo de 5 pontos para estimação da matriz essencial, quanto o algoritmo de triangulação e, finalmente, o algoritmo PnP, assumem que a câmera utilizada para aquisição esteja calibrada, isto é, que a matriz de parâmetros intrínsecos $\mathbf{K}$ seja conhecida. Por isso, para cada novo dispositivo utilizado para aquisição do vídeo, a etapa de calibração que será descrita na próxima seção precisa ser realizada antes de se aplicar o restante do *pipeline*.

Além disso, após o processamento dos *frames* do vídeo, tanto a nuvem de pontos 3D quanto a trajetória precisam ser visualizadas. Para tanto, é necessário garantir que todas as posições e orientações das câmeras, assim como as coordenadas dos pontos reconstruídos, estejam escritas no mesmo referencial. Entretanto, em geral, os algoritmos empregados no *pipeline* fornecem referenciais relativos, de modo que uma etapa de conversão entre bases precisa ser aplicada, como será mostrado adiante.

### 3.4.1 Calibração da câmera

O processo de calibração de uma câmera consiste em recuperar os chamados parâmetros intrínsecos dessa câmera, isto é, a matriz $\mathbf{K}$, que em sua forma geral, é dada por

$$\mathbf{K} = \begin{bmatrix} f_u & \gamma & c_u \\ 0 & f_v & c_v \\ 0 & 0 & 1 \end{bmatrix} \tag{38}$$

em que $f_u$ e $f_v$ são as chamadas distancias focais em cada uma das direções, $c_u$ e $c_v$ são as coordenadas do centro óptico e $\gamma$ é a chamada torção entre os eixos de projeção. Usualmente, os pixels são quadrados, de modo que $f_x = f_y$ e $\gamma = 1$, e, além disso, em geral o centro óptico coincide com o centro da imagem.

O algoritmo de calibração é baseado na trabalho apresentado em (ZHANG, 2000). Essencialmente, partindo-se de uma série de imagens com pontos de interesse cujas coordenadas 2D, bem como as coordenadas 3D originais, são conhecidas, é possível obter os parâmetros intrínsecos. Para que essas coordenadas sejam conhecidas, um padrão de calibração conhecido e facilmente detectável deve ser usado. Em geral, esse padrão corresponde a um quadriculado preto e branco, como um tabuleiro de xadrez. Assim, a primeira etapa da calibração, que corresponde à detecção dos pontos de interesse nas diferentes imagens, fica muito mais precisa: utilizando um detector de vértices, como o descrito na seção 3.2, é possível realizar a correspondência dos vértices do padrão quadriculado com precisão sub-pixel; além disso, ao se utilizar um tabuleiro plano, é possível saber *a priori* que todos os pontos 3D estão no mesmo plano.

Uma vez detectados os pontos em cada uma das imagens, o processo de calibração consiste em resolver a equação de projeção $\mathbf{u} = \mathbf{K}[\mathbf{R} \mid \mathbf{t}]\mathbf{X}$ em que $\mathbf{u}$ e $\mathbf{X}$ foram determinados na detecção e $\mathbf{K}$ e $[\mathbf{R} \mid \mathbf{t}]$ precisam ser encontrados. Ao se combinar as equações de todos os vértices do tabuleiro para todas as imagens, é possível obter um sistema de equações sobre-determinado e linear nos parâmetros intrínsecos e extrínsecos da câmera, de modo que é possível obter uma solução ótima no sentido de mínimos quadrados.

### 3.4.2 Visualização

Como discutido nas seções anteriores, os diferentes algoritmos empregados para a obtenção da matriz de projeção da câmera e dos pontos tridimensionais não fornecem os resultados em um sistema de coordenadas absoluto e comum a todos. Ao contrário, o algoritmo de 5 pontos fornece um deslocamento relativo entre um *frame* e outro e o algoritmo PnP fornece as informações de deslocamento e orientação no referencial da câmera.

Dessa forma, para que seja possível visualizar os pontos 3D e a trajetória da câmera de maneira coerente, é necessário arbitrar um sistema de referência considerado o global e, a cada etapa, realizar uma transformação de coordenadas, para que todos os pontos e matrizes de projeção estejam descritos nesse mesmo sistema de coordenadas. No *pipeline* proposto, isso é feito fixando-se o sistema de coordenadas da primeira câmera como sendo o global e descrevendo as demais câmeras e pontos nesse mesmo sistema.

Para isso, feita a inicialização da nuvem de pontos, que estará naturalmente descrita no sistema de coordenadas da primeira câmera, basta passar a matriz de rotação $\mathbf{R}_k$ e o vetor de rotação $\mathbf{t}_k$ estimados no k-ésimo *frame* no sistema de coordenadas da câmera para o sistema de coordenada dos pontos 3D.

Isso significa, essencialmente, inverter a matriz homogênea dos parâmetros extrínsecos da câmera $[\mathbf{R} \mid \mathbf{t}]$ (uma vez que os parâmetros intrínsecos de calibração independem do sistema de coordenadas). Isto é, obtém-se $[\mathbf{R}' \mid \mathbf{t}'] = ([\mathbf{R} \mid \mathbf{t}])^{-1}$. Usando-se o fato que a matriz de rotação é ortogonal, sua inversa é igual sua transposta, de modo que tem-se finalmente $[\mathbf{R}' \mid \mathbf{t}'] = [\mathbf{R}^\top \mid -\mathbf{R}^\top \mathbf{t}]$. Aplicando-se essa transformação, portanto, tem-se todas as matrizes e pontos no mesmo sistema de coordenadas.

# 4 Implementação

Assim como a etapa de concepção teórica do *pipeline*, a etapa de implementação é crucial para o bom andamento do projeto e pode ser a diferença entre um projeto fracassado ou fora das especificações técnicas e administrativas e um projeto de sucesso. Nessa sessão são abordadas algumas das práticas utilizadas no auxílio da implementação do projeto na seguinte ordem: explicação do código desenvolvido e sua estrutura; explicação sobre o ambiente de desenvolvimento, dependências e utilização do software desenvolvido; boas práticas de desenvolvimento utilizadas e outras considerações.

## *4.1 Execução*

Como descrito na seções anteriores, o *pipeline* proposto tem como objetivo a reconstrução esparsa de cenas e da trajetória de uma câmera a partir de um vídeo monocular. Dessa descrição podemos extrair o arquivo de vídeo como a primeira entrada do *software* e a reconstrução como saída, que pode ser tanto um aquivo com todos os dados gerados quando uma representação gráfica do resultado.

Porém, visto que os algoritmos utilizados necessitam de diversos parâmetros de configuração, temos um arquivo contendo esses valores como a segunda entrada do *pipeline*. Dessa forma, a execução do pipeline se resume a escolher o vídeo a ser reconstruído, escolher os parâmetros de configuração adequados, executar o código e, por fim, consumir a saída da forma adequada.

## *4.2 Estrutura do código*

Visto que diversos algoritmos foram implementados, cada um com diferentes entradas, saídas e condições de utilização, além dos outros componentes que realizam a junção de todas as partes, a seguinte estrutura foi proposta:

- *main.py*: ponto de entrada do usuário, responsável por carregar o arquivo de configurações, instanciar o pipeline, lançar a execução e eventual visualização da reconstrução ou análise dos dados de erro gerados;
- *video_pipeline.py*: contém a classe *VideoPipieline*, responsável por orquestrar a

resconstrução desde a extração de pontos de interesse do vídeo e etapa de inicialização da reconstrução até as otimizações finais após o tratamento do vídeo por completo;

- *synthetic_pipeline.py*: contém a classe *SyntheticPipieline*, classe filha de *VideoPipieline*. Essa classe permite, em suma, a substituição do algoritmo KLT pela criação de câmeras e cenas sintéticas, de forma que as etapas da reconstrução possam ser validadas e exploradas sem a necessidade de um vídeo real junto às dificuldades associadas ao pipeline completo;
- *config.py*: contém a definição de todos os itens do arquivo de configurações e seus respectivos tipos. Também contém funções para a análise do arquivo de configurações e conversão para a estrutura de dados usada internamente;
- *video_algorithms.py*: contém todos os algoritmos relacionados diretamente ao tratamento do vídeo e extração de informações como o algoritmo KLT e de junção de conjuntos de pontos de interesse distintos;
- *reconstruction_algorithms.py*: contém os algoritmos responsáveis pela reconstrução da cena e trajetória a partir dos pontos de interesse 2D detectados. Dentre os algoritmos estão o algoritmo de 5 pontos, as variações do algoritmo *solvePnP* e o algoritmo de triangulação e de reprojeção;
- *init_agorithms.py*: neste arquivo está implementada a rotina utilizada para iniciar a reconstrução utilizando os algoritmos descritos acima;
- *bundle_adjusment.py*: contém as funções relacionadas ao *Bundle Adjustment*, como estruturação dos dados antes e depois da otimização, funções de apoio à otimização e a própria função que realiza a otimização;
- *utils.py*: contém funções diversas que auxiliam no desenvolvimento nas demais partes do código.

Podemos notar pela lista acima que, apesar do *Python* ser uma linguagem orientada a objetos, há um baixo número de classes novas, sendo a maior parte do código implementada sob a forma de funções. Essa estrutura foi consequência do ponderamento das práticas adotadas no projeto (descritas abaixo) com o objetivo de melhorar as métricas de performance do *pipeline*, diminuir a incidência de erros de programação e permitir que futuras modificações no código e sua compreensão sejam facilitas.

## *4.3 Bibliotecas utilizadas*

Nessa seção são abordadas e explicadas algumas das bibliotecas utilizadas durante o desenvolvimento com o intuito de familiarizar o leitor com as ferramentas disponíveis no ecossistema de desenvolvimento em *Python* e mostrar como as suas utilizações contribuíram para o desenvolvimento do projeto.

- *opencv*: possivelmente umas das mais conhecidas bibliotecas de tratamento de imagem. Sua escolha foi devido à extensa documentação disponível online, por ser implementada em *C++* e portanto apresentar uma performance aceitável e também pelo fato de que está disponível para uso tanto em *Python* quanto em *C++*, facilitando assim uma futura portabilidade do *pipeline* para outras linguagens. Uma restrição importante à sua utilização, porém, é a dificuldade no processo de instalação, que muitas vezes envolve a compilação a partir do código fonte não só da própria biblioteca, como também de diversas dependências. Porém, uma vez devidamente instalada sua utilização é fácil.
- *numpy*: como dependência do *opencv* e ótima ferramenta para operações com matrizes, estruturas de dados multi-dimensionais e computação numérica, temos a biblioteca *numpy*. Seu uso, assim como o da biblioteca anterior é amplamente difundido, com uma documentação extensa e uma comunidade bastante ativa. Também implementada em *C++* e compilada para ser utilizada em *Python*, essa biblioteca apresenta ganhos de performance consideráveis quando comparada com uma implementação puramente em *Python*.
- *scipy*: definida como um conjunto de bibliotecas (dentre elas *numpy* e *pandas*) para o ramo da matemática, ciência e engenharia, essa biblioteca disponibiliza diversas ferramentas para tratamento de dados. No contexto desse projeto foi utilizada na etapa de otimização do *Bundle Adjustment*.
- *pandas*: utilizada nesse projeto para o tratamento dos dados oriundos da execução, *pandas* é também uma biblioteca bastante difundida e utilizada por conta de sua performance e flexibilidade.

- *itertools*: apesar de não ser uma biblioteca externa, vale ressaltar o módulo *itertoos* por conta do papel que tomou nas etapas de simplificação e otimização do código. Esse módulo permite a escrita de código mais legível e "pythonico" de forma que construções mais complexas de laços fossem substituídas, o que diminuiu a incidência de certos tipos de erros de programação e aumentou a velocidade da implementação de funcionalidades novas.
- *ruamel.yaml*: como descrito abaixo, o formato de arquivos *YAML* apresenta diversas vantagens em relação a outros formatos de arquivos, especialmente quando o usuário modifica-o manualmente, portanto uma biblioteca adequada é necessária.
- *dacite*: juntamente com a utilização de *ruamel.yaml*, essa biblioteca permite a criação de *dataclasses* com bastante simplicidade, realizando ao mesmo tempo a verificação do tipo de dados sendo processados contra o tipo esperado.
- *seaborn*: por fim, mas ainda importante, temos *seabon*, uma biblioteca de visualização de dados bastante flexível e simples de ser utilizada, que durante o desenvolvimento do projeto simplificou consideravelmente a geração de gráficos e tornou a análise visual de dados mais dinâmica.

## *4.4 Práticas de desenvolvimento*

Por fim, estão descritas aqui algumas das práticas adotadas durante o desenvolvimento do projeto. Essas práticas foram adotadas com os seguintes objetivos:

- Diminuir a ocorrência de erros de programação;
- Melhorar a qualidade e organização do código;
- Facilitar e acelerar a introdução de novas funcionalidades;
- Facilitar a manutenção do código;
- Facilitar a utilização do código;
- Melhorar a comunicação entre membros da equipe;
- Facilitar o desenvolvimento em mais de uma pessoa.

E as principais práticas adotadas são:

- Utilização de ambiente virtual: permite padronizar o ambiente de desenvolvimento diminuindo assim problemas que a equipe pode encontrar por conta de versões diferentes da linguagem ou dependências, por exemplo;
- Controle de versões: através de ferramentas como *git* e *mercurial*, o controle de versões permite um melhor controle sobre as modificações no código e o desenvolvimento simultâneo por mais de um integrante da equipe;
- Padronização do estilo do código: facilita o desenvolvimento, pois faz com que o código desenvolvido por cada desenvolvedor seja mais facilmente integrado ao resto e analisado ou estendido quando necessário. Para auxiliar nessa tarefa a ferramenta *Black* foi utilizada, que enforça a utilização do estilo *PEP 8*, o guia de estilo oficial de *Python*;
- Utilização de boa *IDE*: fornece grande auxílio no desenvolvimento indicando possíveis erros, facilitando a refatoração do código, utilização do controle de versionamento entre outras funções. No caso desse projeto a *IDE* escolhida foi *PyCharm* desenvolvido pela *JetBrains*;
- Modularidade e reutilização do código: auxilia no desenvolvimento pode diminuir a complexidade das funções a serem implementadas, diminuir a quantidade de código escrito e auxilia também na fase de testes, pois permite separar o pipeline em partes menores e bem definidas para serem testadas individualmente;
- Gerenciamento de configurações centralizado: importante principalmente na utilização do *pipeline*, o bom gerenciamento de configurações alerta usuários quando o arquivo de configurações está formatado incorretamente ou apresenta dados em formatos não compatíveis. Auxilia também durante o desenvolvimento, pois centraliza as validações das entradas em um único ponto e torna desnecessária a revalidação dos dados em outros pontos do código;
- Entregas pequenas, periódicas e bem definidas: talvez uma das práticas que mais contribuiu para o andamento do projeto, ao separar o trabalho a ser realizado foi possível não só acompanhar o desenvolvimento com a granularidade adequada, como também seguir fielmente o cronograma proposto.

# 5 Resultados

Os resultados obtidos com o *pipeline* proposto serão apresentados a seguir. Os testes realizados foram divididos em, essencialmente, duas etapas. Em um primeiro momento, os diferentes algoritmos utilizados durante a reconstrução incremental foram testados utilizando-se um conjunto de dados sintéticos. Isto é, foram gerados pontos tridimensionais, bem como uma trajetória para a câmera, ambos sendo, portanto, conhecidos. Em seguida, os pontos foram projetados através das diferentes matrizes da câmera em cada posição da trajetória. Essas projeções foram, então, colocadas como entrada do *pipeline* e a reconstrução obtida foi comparada com os dados originais.

Conforme será detalhado a seguir, essa validação foi realizada de duas maneiras. A primeira delas usando os dados projetados ideias, com o objetivo de validar o *pipeline*, uma vez que nessa situação era necessário recuperar exatamente os dados gerados originalmente. Em seguida, ruído gaussiano branco foi adicionado às projeções para simular de maneira mais realista as projeções ruidosas dos *frames* de um vídeo. Nesse caso, foi possível atestar, em particular, o efeito corretivo do *Bundle Adjustment* e sua capacidade de fazer a solução retornar aos valores esperado.

Uma vez feita a validação com esses dados sintéticos, a segunda etapa consistiu na utilização de diferentes vídeos reais. A performance do *pipeline* em cada uma dessas situações foi analisada, bem como o impacto de algumas variações nos parâmetros dos diversos algoritmos envolvidos na reconstrução. Em especial, o impacto da inicialização será evidenciado, mostrando de que forma o processo de refinamento proposto auxilia na melhoria da performance geral da reconstrução.

## *5.1 Validação do pipeline*

Como mencionado anteriormente, antes de aplicar o *pipeline* a vídeos reais, a primeira etapa de testes consistiu em uma validação dos diferentes algoritmos utilizados através do uso de projeções de pontos tridimensionais gerados sinteticamente em posições e orientações conhecidas. Mais especificamente, foi gerado um paralelepípedo de dimensões $4 \times 5 \times 5$ com pontos apenas nas faces externas e 25 posições de câmera que realizam um círculo completo em torno desse paralelepípedo, como mostrado na imagem abaixo:

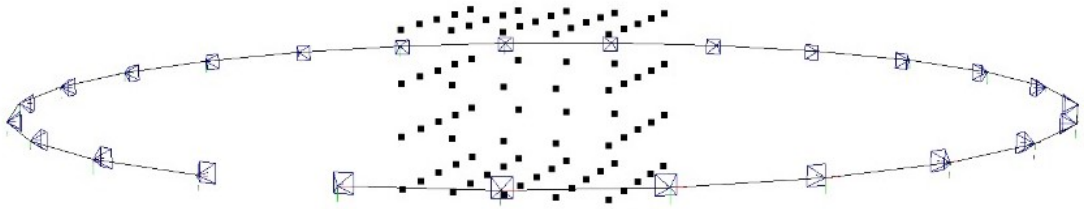
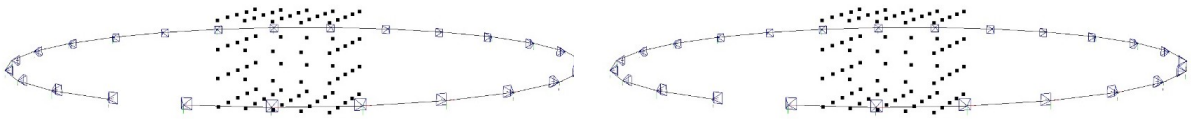


Figura 5.1 – Paralelepípedo e trajetória da câmera gerados para validação do *pipeline*

Como esperado, a versão dos dados sem qualquer fonte de ruído é recuperada de maneira idêntica pelas versões do *pipeline* sem e com a otimização do *Bundle Adjustment*, como pode ser inspecionado visualmente nas imagens abaixo. Isso permite validar, ao mesmo tempo, dois aspectos importantes do trabalho: em primeiro lugar, a reconstrução básica funciona como deveria, uma vez que na ausência de ruído ela retorna exatamente o esperado; além disso, o fato do *Bundle Adjustment* não alterar em nada a solução nesse cenário, em que o erro de reprojeção é nulo, é um indicativo inicial de seu bom funcionamento, dado que qualquer otimização é desnecessária.



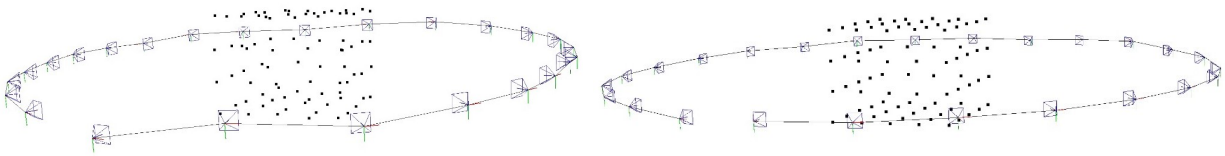
(a) (b)

Figura 5.2 – Comparação dos resultados do *pipeline*: sem *Bundle Adjustment* à esquerda (a) e com à direita (b). Como esperado, as reconstruções obtidas são idênticas entre si e com os dados originais.

Por outro lado, assim que algum tipo de ruído é introduzido nas projeções dos dados sintéticos, é possível perceber a deterioração da reconstrução quando feita pelo *pipeline* de base sem otimização e, de maneira análoga, a capacidade de correção imposta pelo *Bundle Adjustment*. De maneira mais precisa, no caso dos dados sintéticos, são conhecidos os pontos tridimensionais $\mathbf{X}_i$, bem como as matrizes de projeção $\mathbf{P}_j$. Assim, para alterar cada ponto $i$ projetado em uma posição de câmera $j$ dado por $\mathbf{u}_{ij} = \mathbf{P}_j\mathbf{X}_i$, ruído branco gaussiano aditivo (AGWN na sigla em inglês) foi utilizado. Isto é, em cada componente das projeções foi adicionado o termo gaussiano de média zero e matriz de covariância $\mathbf{\Sigma}$ diagonal e constante $\mathbf{\Sigma} = \sigma\mathbf{I}$:

$$\mathbf{u}_{ij}^{\text{noisy}} = \mathbf{u}_{ij} + \mathcal{N}(0, \sigma\mathbf{I}) \tag{39}$$

No caso dos testes realizados, a variância do erro utilizada foi de $\sigma = 5$ pixels. Na prática, isso significa que cada pixel projetado estará em sua posição original com um desvio de mais ou menos 10 pixels em aproximadamente 95% dos casos, o que representa um nível de ruído relativamente elevado. Novamente, os resultados do *pipeline* com e sem a aplicação do *Bundle Adjustment* são mostrados na figura a seguir.



(a) (b)

Figura 5.3 – Comparação dos resultados do *pipeline* quando rúido é adicionado: sem *Bundle Adjustment* à esquerda (a) e com à direita (b). O BA é capaz de recuperar o resultado original.

É possível notar que, no caso do *pipeline* básico, o ruído introduz dificuldades na recuperação tanto da estrutura tridimensional, que fica deformada, quanto da trajetória da câmera, que passa a apresentar oscilações inexistentes nos dados originais. Isso é bastante razoável, uma vez que os algoritmos da reconstrução possuem certa sensibilidade ao ruído. De toda forma, é interessante notar que a estrutura geral dos dados foi recuperada: ainda é possível entender que o objeto é um paralelepípedo e que a câmera realizou uma trajetória circular em torno dele.

Por outro lado, nessa situação com ruído, o interesse do *Bundle Adjustment* fica muito mais evidente. É possível notar que, ao menos visualmente, a solução é praticamente idêntica aos dados originais apresentados na figura (5.1): o paralelepípedo não apresenta mais a distorção observada no caso sem BA e a trajetória da câmera volta a ser um círculo sem qualquer desvio como antes.

No caso dos dados sintéticos, é possível analisar cada uma das situações descritas anteriormente de um ponto de vista quantitativo também, ao se calcular o erro entre a solução encontrada e os dados originais (o que não é possível em geral para vídeos reais, uma vez que as posições dos pontos e câmeras não são conhecidos). O gráfico abaixo apresenta um resumo desses resultados: para cada *frame*, são comparados os ângulos e as translações estimados com os dados originais, nos gráficos de cima, bem como os erros médios dos pontos tridimensionais estimados e da reprojeção desses pontos, nos gráficos da segunda linha. Devido ao caráter estocástico do ruído, cada caso é rodado 200 vezes e

a parte hachurada do gráfico representa o desvio padrão dos erros obtidos.

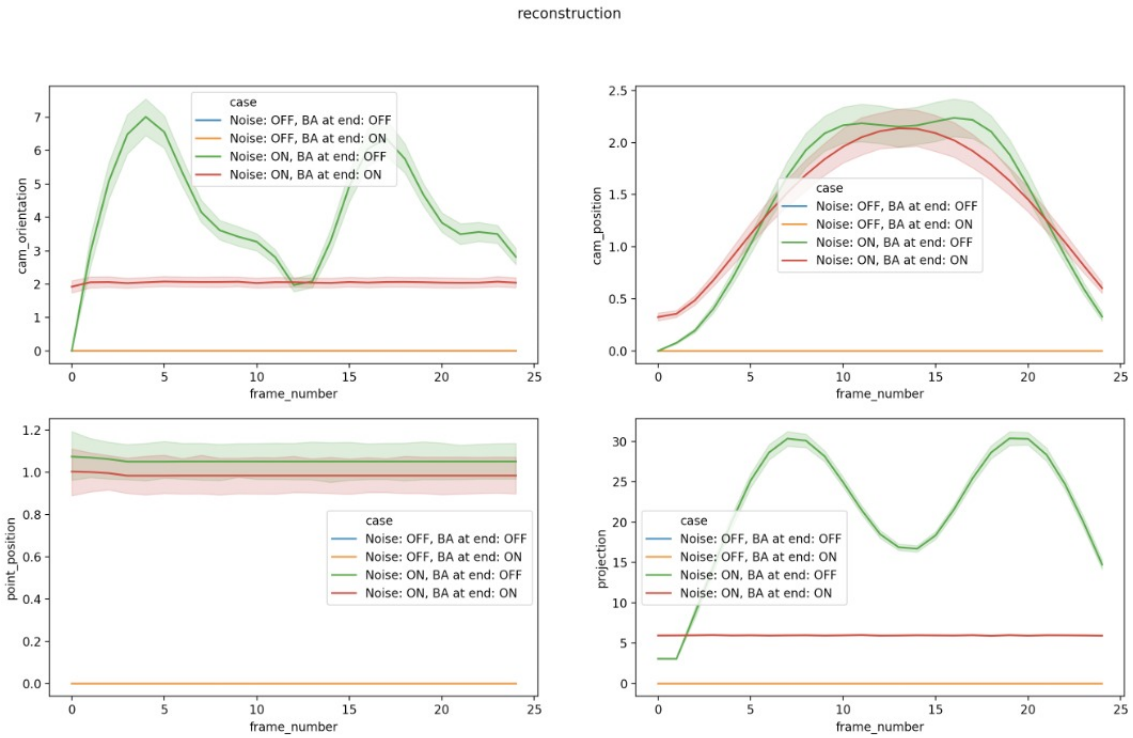



Figura 5.4 – Erros, ao longo dos *frames*, nas posições dos pontos, na translação e orientação da câmera e na reprojeção para os dados com e sem rúido e *pipelines* com e sem BA. A otimização aproxima a solução do resultado original.

Como havia sido inspecionado visualmente, é possível notar que no caso sem ruído, os *pipelines* com e sem BA, representados pelas curvas azul e amarela, respectivamente, garantem a mesma solução sem qualquer tipo de erro: em todos os gráficos as duas curvas estão sobrepostas e valem zero. Assim que o ruído gaussiano é introduzido, é possível notar um grande crescimento do erro no *pipeline* sem BA (curva verde), enquanto que a presença do BA permite uma reconstrução muito mais próxima da original. Como é possível notar no gráfico do canto inferior esquerdo, o erro de reprojeção médio fica próximo de 5 pixels no caso com BA, enquanto oscila em torno de 25 quando ele é desligado, o que representa, portanto, uma redução média do erro de cinco vezes.

Em último lugar, além da validação da qualidade da reconstrução, também foi analisada a velocidade de processamento do *pipeline* nesse caso mais simples, com o objetivo de se estimar o quão distante estaria a performance de um processamento em tempo real. Para o caso mais simples, sem ruído ou qualquer otimização final, o *pipeline* é capaz de processar 300 *frames* por segundo em um processador *Intel Core* i7-8850H 2.6GHz de 6 núcleos. Evidentemente, essa situação é muito distante de um vídeo real,

porém, de toda forma, o fato do *pipeline* ser bastante eficiente nessa situação simples mostra que existe a possibilidade de sua performance não ser impeditiva em casos mais complexos.

## *5.2 Análise do efeito da otimização e inicialização*

Uma vez validados os constituintes básicos do *pipeline*, uma segunda etapa de testes foi realizada para se comparar diferente versões possíveis, dependendo da escolha de alguns dos componentes. Mais precisamente, em primeiro lugar foi analisada a influência do momento em que o *Bundle Adjustment* era aplicado: como mencionado anteriormente, essa etapa de otimização da reconstrução incremental poderia ser feita apenas quando todos os *frames* tivessem sido processados ou ao longo da reconstrução, através de uma janela deslizante toda vez que um determinado número de *frames* é processado. Isso porque, usualmente, a literatura indica o uso da otimização apenas no final do *pipeline*. Porém, dado as características sequenciais de um vídeo, bem como o interesse de eventualmente poder processá-lo em tempo real, a solução com a janela deslizante parece ser mais adequada.

Dessa forma, foram utilizados 4 configurações diferentes: a primeira, sem qualquer *Bundle Adjustment* sendo aplicado; a segunda, com BA apenas no final; a terceira, com BA sendo aplicado com uma janela deslizante; e a quarta, em que o BA é aplicado tanto com a janela deslizante, quanto no final do vídeo. Para todos os casos, ruído branco gaussiano foi adicionado nas projeções do mesmo modo que descrito na seção anterior e com $\sigma = 5$. É importante notar que, no caso em que a janela deslizante é utilizada, são considerados sempre 6 *frames* anteriores, sendo que a diferença entre eles é variável. Isto é, utiliza-se sempre o *frame* atual, um *frame* para trás, 3 *frames* para trás, 6 *frames* pra trás e assim sucessivamente. O objetivo de uma tal estratégia é aumentar a velocidade de processamento e diminuir um potencial *overfitting*. Isso porque o BA otimiza o erro empírico de reprojeção, isto é, a otimização depende dos dados observados. Sendo feito com os dados parciais, existe a possibilidade da solução convergir para um mínimo local que não corresponde, portanto, à melhor configuração possível.

As figuras abaixo mostram os resultados para cada uma das quatro configurações descritas anteriormente. Visualmente é possível perceber que todas as versões com BA ((b), (c) e (d)) performam melhor do que a versão básica, o que está de acordo com os

resultados apresentados anteriormente.

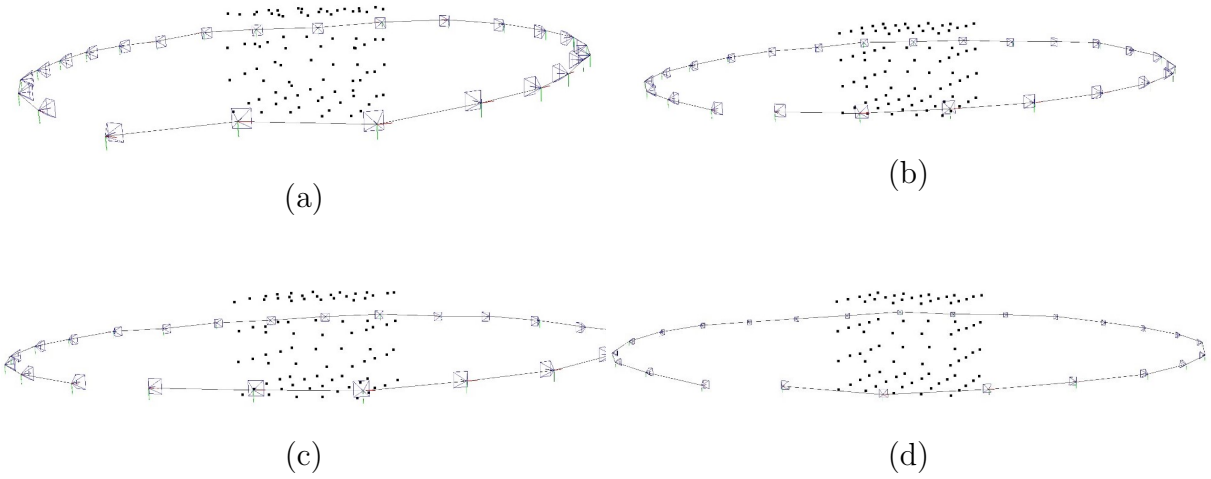



Figura 5.5 – Diferentes versões do *pipeline*: (a) sem BA algum, (b) BA apenas no final, (c) BA com janela deslizante, (d) BA com janela deslizante e no final.

O efeito do momento em que o BA é aplicado pode ser melhor analisado através dos gráficos de erro que foram introduzidos na seção anterior e que são reproduzidos abaixo.

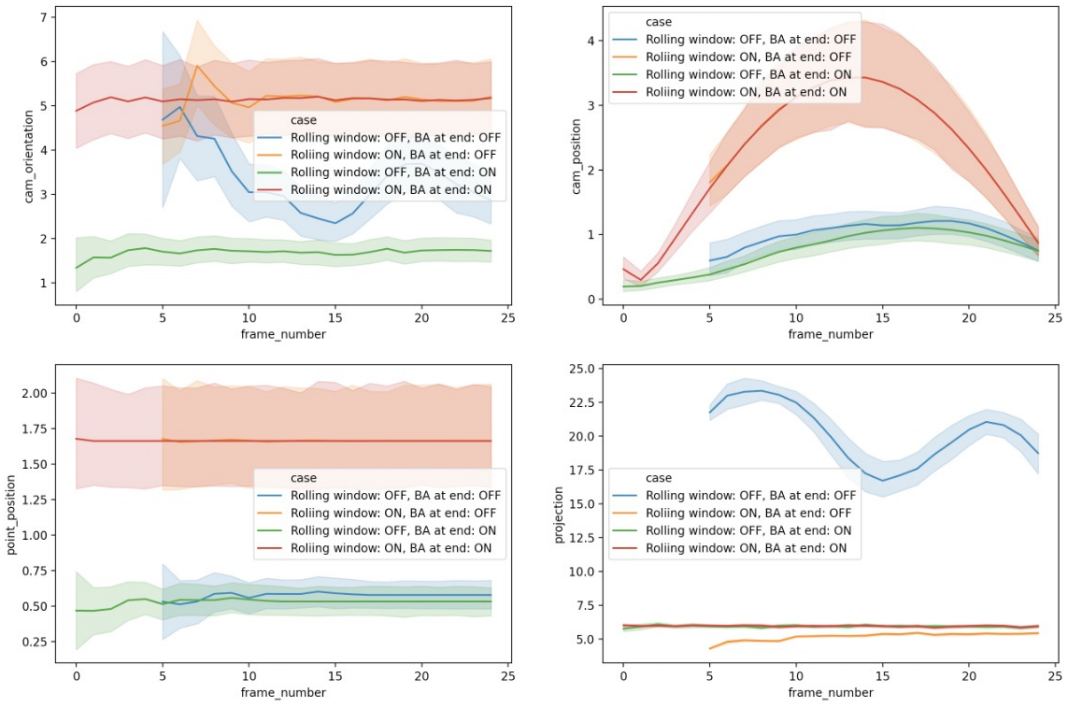
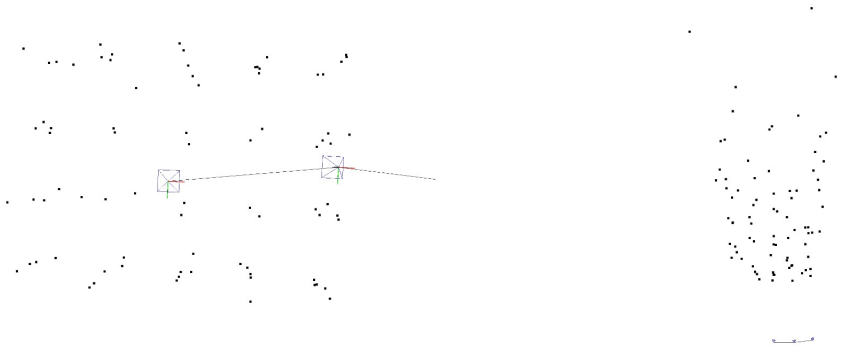



Figura 5.6 – Comparação das performances com as diferentes configurações do BA.

É possível notar que, de maneira geral, a presença do *Bundle Adjustment* no final (curvas verde e vermelha) garante os melhores resultados em termos de todas as métricas de erro, sendo que a versão em que a janela deslizante não é usada fornece os melhores resultados (curva verde). Isso pode ser interpretado pelo argumento fornecido anteriormente: a aplicação do BA com dados parciais pode resultar na otimização para mínimos locais, uma vez que nem todos os dados estão disponíveis. Isso fica ainda mais evidente quando o BA é realizado apenas com a janela deslizante (curva amarela): mesmo que o erro de reprojeção médio seja o menor de todos, que é justamente o que é minimizado pelo algoritmo, as demais métricas de erro são piores até mesmo do que a versão sem otimização alguma, o que é um forte indicativo de que a solução encontrada caminhou para um mínimo local com boa coerência entre os pontos tridimensionais e as posições da câmera, mas que não correspondia exatamente aos dados originais.

Esse potencial *overfitting* causado pela otimização com a janela deslizante fica bastante evidente quando analisadas a figura abaixo, em que o BA foi utilizado logo no início da reconstrução, quando apenas uma das faces do paralelepípedo havia sido vista pela câmera até então.



(a) (b)

Figura 5.7 – Resultado da otimização ao longo do *pipeline* para os *frames* iniciais: visão frontal à esquerda (a) e superior à direita (b). Os pontos da perspectiva das câmeras estão coerentes, mas a estrutura global do paralelepípedo fica deformada.

É possível perceber que a otimização produz pontos coerentes para a face do paralelepípedo que havia sido vista pela câmera até então, como está evidenciado na visão frontal (da perspectiva das câmeras) do paralelepípedo mostrado à direita: os pontos dessa face estão razoavelmente dispostos na forma de uma quadrilátero e os demais pontos em outros planos garantem uma perspectiva natural do objeto tridimensional. Por outro lado, a visão superior mostrada na figura à esquerda evidencia que os demais pontos estão completamente deformados e não formam a figura esperada. Como o erro minimizado pelo BA resulta da reprojeção entre pontos e câmeras disponíveis, o mínimo encontrado corresponde, visualmente, a uma disposição dos pontos que faça sentido apenas para a perspectiva das câmeras usadas.

Dada a pequena translação e rotação (e o ruído), a informação observada é muito parecida em todos os *frames* e, além disso, apresenta baixa qualidade para os pontos das demais faces. Como consequência, é possível obter um erro de reprojeção baixo para as câmeras utilizadas e, ao mesmo tempo, uma configuração global que não corresponde exatamente à estrutura tridimensional existente na realidade.

Além da otimização, outro constituinte crucial do *pipeline* proposto é a inicialização, uma vez que, quando realizada de maneira inadequada, pode acarretar na impossibilidade de realizar qualquer reconstrução que seja minimamente razoável. Para tanto, foram realizados alguns testes com diferentes números de *frames* utilizados para refinar a inicialização, como descrito na seção 3.3.2. Mais precisamente, os gráficos apresentados na figura anterior utilizam 5 *frames* para o refinamento e outros 5 para a validação do *threshold* de erro. Em seguida, foram realizados testes com o dobro do número de *frames* tanto para o refinamento, quanto para a validação, os resultados sendo mostrados nos gráficos a seguir.

É possível notar que, de maneira geral, o nível de erro foi menor em todas as configurações testadas, o que mostra a utilidade de se realizar uma inicialização mais robusta. O ponto negativo, entretanto é o aumento da complexidade de cálculo, o que resulta, necessariamente, em um menor número de *frames* processados por segundos. Dessa forma, é possível perceber que existe um compromisso entre a qualidade da reconstrução e a velocidade do processamento.

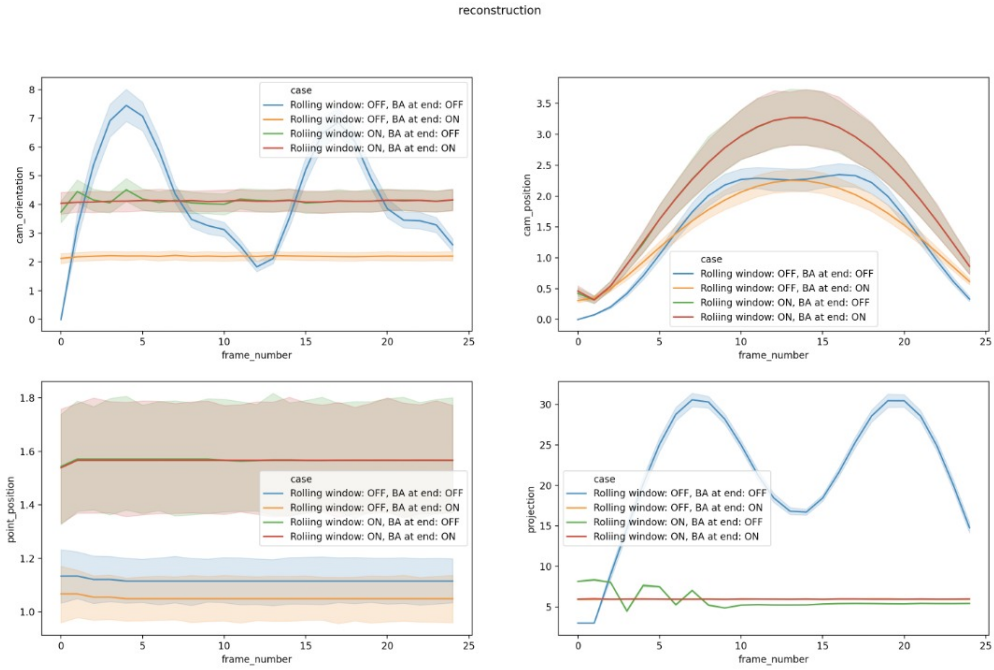



Figura 5.8 – Comparação das performances com as diferentes configurações do BA, 10 *frames* utilizados na inicialização.

Além disso, os testes realizados mostraram que existe uma influência da aplicação do *Bundle Adjustment* com a janela deslizante na performance da inicialização. Basicamente, a presença de uma otimização ao longo do processo permite que o *threshold* de erro considerado durante o refinamento seja atingido mais rápido, como é mostrado no gráfico abaixo, o que faz com que o número médio de *frames* descartados seja muito mais baixo.

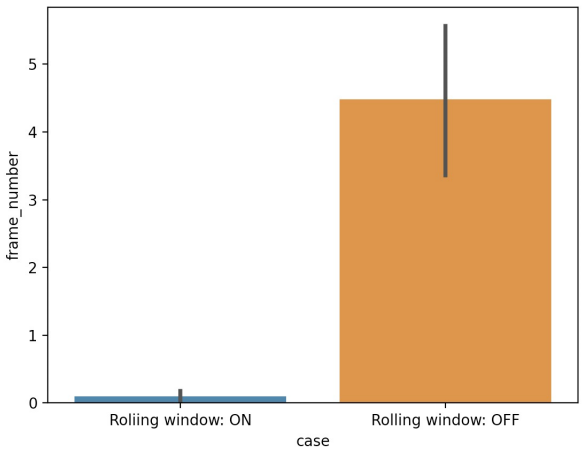



Figura 5.9 – Comparação do número de *frames* descartados na inicialização. O BA com janela deslizante permite um início mais rápido.

Mais precisamente, enquanto a ausência do BA implicou em média 3 a 6 *frames* descartados, de modo geral sua presença permitiu a inicialização diretamente a partir da primeira tentativa. Como cada inicialização realiza as contas para $k + p$ *frames*, com $k = p = 10$, isso significa que a são realizados 22 processamentos no caso com $BA$ e de 60 a 120 processamentos. Na prática, isso significa que o BA com janela deslizante seria capaz de aumentar a velocidade de processamento, mesmo utilizando mais *frames* para o refinamento da inicialização e, como o número de *frames* aumenta, o efeito potencial do *overfitting* discutido anteriormente pode ser suavizado, mesmo que ele ainda possa estar presente.

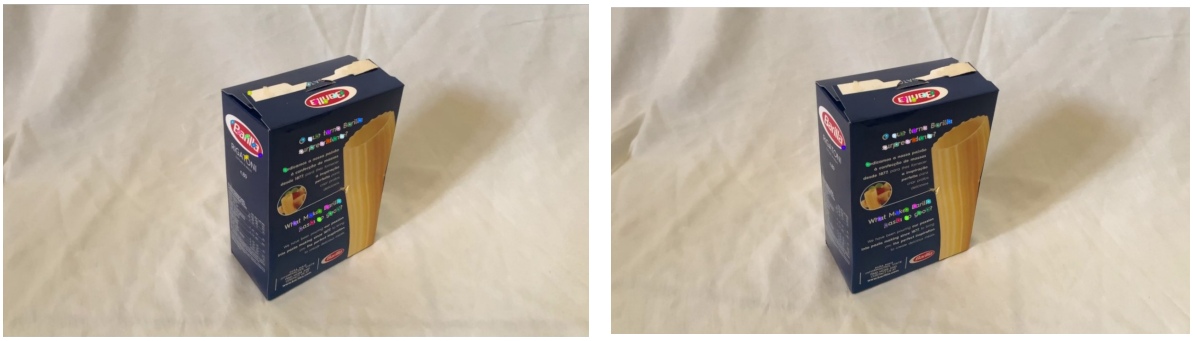
A partir desses testes, foi possível então selecionar o que foi considerada a melhor configuração do *pipeline* proposto antes de aplicá-lo a vídeos reais. Essencialmente, levando-se em conta o efeito da qualidade da reconstrução obtida, assim como a velocidade de processamento, a versão que utiliza um maior número de *frames* para a refinar a inicialização, associado ao *Bundle Adjustment* realizado através da janela deslizante foi, escolhido por fornecer o melhor compromisso entre essas duas métricas de performance. Como mencionado anteriormente, o maior problema de uma tal estratégia é causar uma espécie de *overfitting* e degradar ligeiramente a solução. Entretanto, a presença de mais *frames* na inicialização parece tender a compensar esse efeito. Por último, de um ponto de vista um pouco mais conceitual, essa escolha permite que o *pipeline* rode teoricamente em tempo real, uma vez que a realização do BA apenas no final do vídeo implica que ele seja processado por completo após sua aquisição. Novamente, mesmo que isso não seja um requisito central do projeto, a possibilidade de adaptá-lo para esse modo de funcionamento é interessante do ponto de vista de trabalhos futuros.

## *5.3 Vídeos reais*

### 5.3.1 Vídeo 1: caixa de macarrão

Uma vez realizados os testes com dados sintéticos, foi possível aplicar o *pipeline* proposto a vídeos reais. Os primeiros testes foram realizados com objetos geométricos simples, como por exemplo uma caixa de macarrão, disposto em um ambiente bem contrastado, o objetivo sendo facilitar a detecção de boas *features* com o algoritmo KLT. Como é possível observar com a sequência de *frames* a seguir, isso de fato acontece: as diferentes *features* seguidas pelo KLT estão marcadas nos diferentes momentos do vídeo com pontos coloridos e é possível notar que, de fato, existe uma coerência conforme a sequência de imagens avança.

(a) (b)



(c) (d)

Figura 5.10 – Sequência de *frames* do vídeo de uma caixa de macarrão e a evolução das *features* acompanhadas pelo KLT.

É interessante notar que, como esperado, as melhores *features* encontradas pelo algoritmo estão nos vértices de letras da caixa, onde há um forte gradiente entre regiões brancas e azuis, por exemplo. Em seguida, essas *features* foram alimentadas no algoritmo de reconstrução, o resultado sendo apresentado abaixo. É possível notar que a solução encontrada é bastante coerente com o esperado: a câmera realiza uma trajetória de cima da caixa contornando-a (como é possível perceber pela sequência de *frames* acima). Além disso, os pontos tridimensionais reconstruídos apresentam a mesma estrutura de paralelepípedo da caixa, estando em diferentes planos perpendiculares entre si.

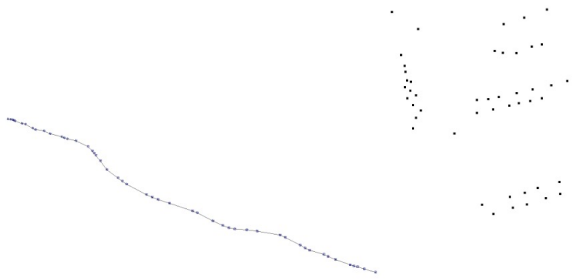


Figura 5.11 – Resultado da reconstrução para a sequência de vídeo da caixa de macarrão.

Além da inspeção visual do resultado, que é um pouco mais difícil de realizar de modo confiável nesse caso, é possível avaliar quantitativamente sua performance através do erro de reprojeção, uma vez que basta comparar as posições dos pixels nos *frames* originais com os estimados através dos pontos tridimensionais e matrizes de projeção encontradas. O resultado é apresentado no gráfico abaixo.

Como é possível observar, o erro médio obtido permanece abaixo de 0.6 pixel em todos os *frames*. De maneira natural, também é possível observar que esse erro tem uma tendência de aumento ao longo do tempo. Isso pode ser explicado por um acúmulo de erro durante a reconstrução, que é inerente ao método sequencial em que ela é feita.

Um segundo teste foi realizado utilizando a mesma caixa de macarrão. No vídeo mostrado acima, a aquisição foi realizada de maneira suave e contínua, isto é, a câmera realizou um movimento contornando lentamente a caixa de cima, sem qualquer movimento brusco ou outro tipo de ruído na aquisição. Para testar a robustez do *pipeline* uma segunda aquisição foi feita, dessa vez muito mais complexa: a caixa foi filmada por mais

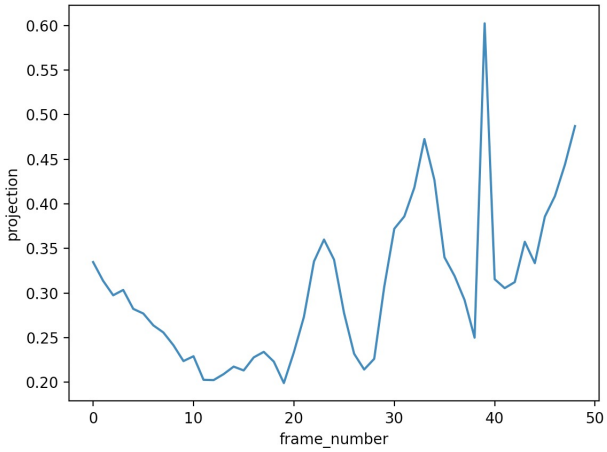



Figura 5.12 – Erro de reprojeção obtido com a sequência de vídeo da caixa de macarrão

tempo, com a câmera realizando um movimento aproximadamente circular (iniciando e terminando aproximadamente no mesmo ponto) e repleto de pausas e interrupções ao longo da trajetória.

O objetivo principal sendo, portanto, a introdução de diferentes tipos de ruídos na aquisição que, em tese, deveriam dificultar a reconstrução. O resultado obtido é mostrado abaixo: em primeiro lugar, é possível notar que a trajetória mais complexa de fato é recuperada pelo *pipeline* e a câmera de fato inicia e termina o seu movimento aproximadamente no mesmo ponto; além disso, os pontos tridimensionais obtidos também apresentam a mesma coerência de antes, estando nos diferentes planos como esperado.

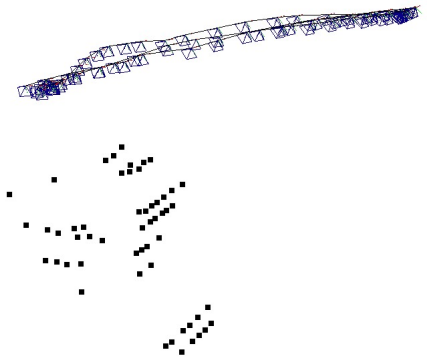


Figura 5.13 – Resultado da reconstrução da caixa de macarrão no caso de uma aquisição ruidosa.

É interessante notar que, olhando o erro médio de reprojeção, como esperado, há um aumento em relação a versão anterior: esse erro chega a superar o nível de 2.5 pixels, ou seja, praticamente 5 vezes maior do que o vídeo mais simples. De toda forma, tanto do ponto de vista qualitativo quanto quantitativo o resultado obtido continua aceitável.

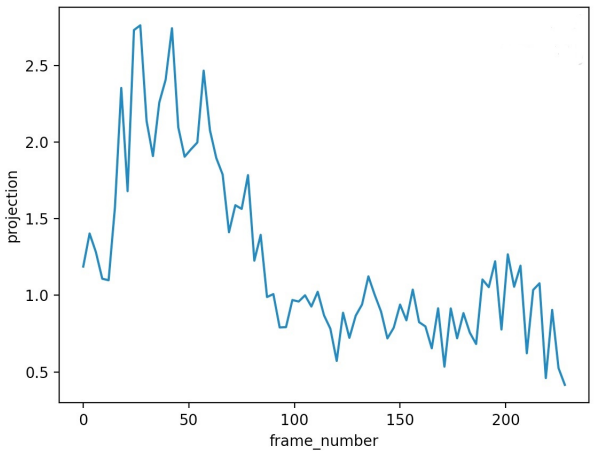
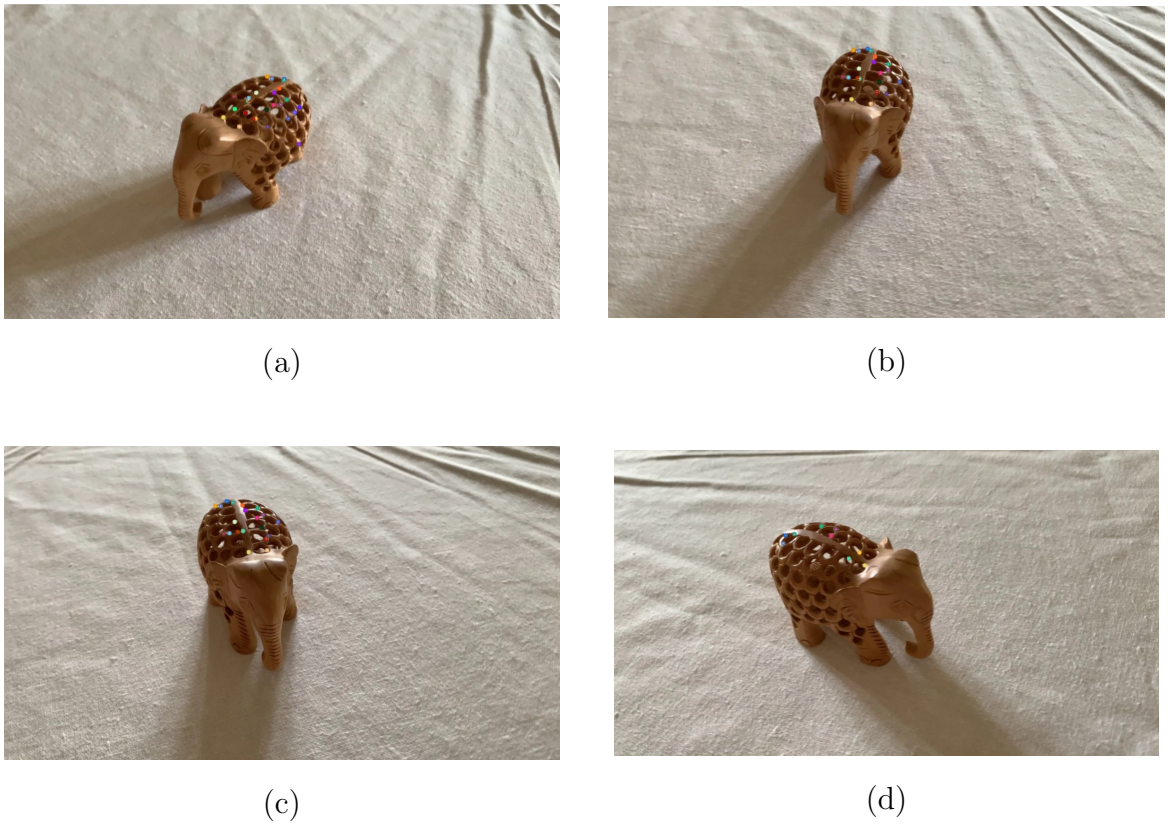



Figura 5.14 – Erro de reprojeção no caso de uma aquisição ruidosa: há um aumento significativo em relação a versão mais simples do vídeo.

### 5.3.2 Vídeo 2: estátua de um elefante

Ainda mantendo-se em vídeos de objetos bem contrastados em relação ao fundo, o segundo tipo de teste realizado foi utilizando objetos mais complexos. Da mesma forma que uma aquisição mais complexa, o objetivo desse teste era de introduzir dificuldades na reconstrução. Nesse caso, essa dificuldade é ainda mais representativa, uma vez que a complexidade do objeto poderia potencialmente deteriorar a qualidade dos pontos selecionados pelo algoritmo KLT e, como toda a reconstrução se baseia na correspondência entre esses pontos ao longo dos *frames*, o impacto poderia ser visto ao longo de todo *pipeline*.

Para realizar esse teste, foi utilizada uma estátua de um elefante. Como evidenciado na sequência de *frames* abaixo, ela foi filmada de maneira análoga ao vídeo da caixa de macarrão: a câmera realiza um movimento suave acima do objeto, contornando-o parcialmente. Além disso, a estátua em si apresenta uma estrutura muito mais complexa que a caixa de macarrão, uma vez que existem diferentes detalhes e uma região com muitas texturas e partes vazadas, com certo potencial de dificultar o detector de *features* do KLT.



(a) (b)

(c) (d)

Figura 5.15 – Sequência de *frames* do vídeo da estátua de um elefante: as *features* detectadas se concentram, sobretudo, na região com bastante textura.

Naturalmente, a região vazada e com textura recebeu quase que a totalidade das *features* detectadas pelo KLT. No entanto, um ponto positivo foi que elas se mantiveram coerentes ao longo do vídeo, sem que houvesse qualquer tipo de salto ou outra variação abrupta de um *frame* para outro, o que poderia acontecer em tal tipo de estrutura.

Realizando a reconstrução em seguida e inspecionando-a visualmente, é possível concluir que o *pipeline* foi capaz de produzir um resultado coerente do ponto de vista qualitativo, como é mostrado na figura abaixo com a saída da reconstrução. Isso porque a trajetória da câmera é claramente recuperada, com as diversas posições acima do objeto, contornando-o ao longo do vídeo. Além disso, a estrutura 3D parece bastante razoável: como os pontos detectados são todos pertencentes à região vazada do elefante, é possível reconhecer o formato arredondado dessa região, bem como a certa simetria presente tanto nos pontos detectados, quanto na própria estrutura real do elefante.

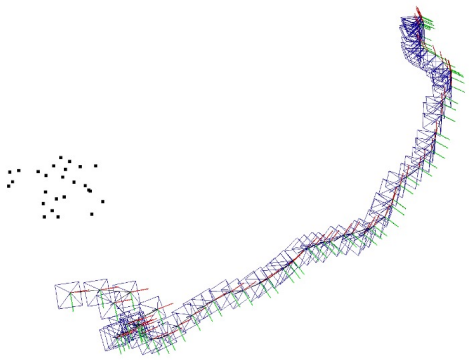


Figura 5.16 – Resultado da reconstrução do *pipeline* para a estátua de um elefante.

De um ponto de vista quantitativo, é possível analisar o erro de reprojeção mais uma vez. Como esperado, dada a maior complexidade desse caso, o erro observado é muito maior do que anteriormente, chegando a 14 pixels nos *frames* iniciais. Entretanto, a aplicação do *Bundle Adjustment* ao longo do vídeo permite que esse erro decresça e volte para níveis aceitáveis do meio para o fim do vídeo. De toda forma, é possível concluir que a performance foi ligeiramente afetada pelo aumento da complexidade do objeto observado.

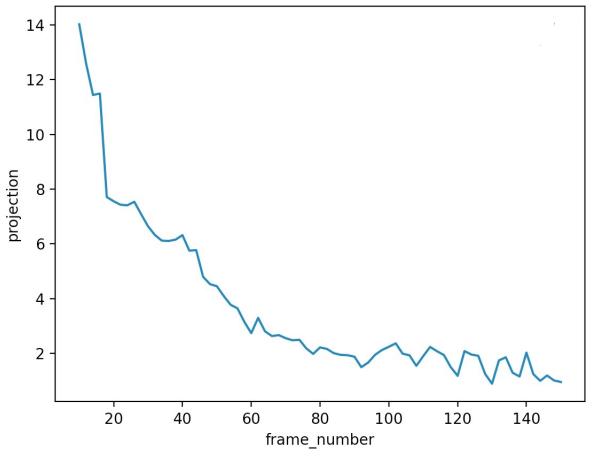



Figura 5.17 – Erro de reprojeção do vídeo da estátua de um elefante.

## *5.4 Limitações*

Em seguida, depois de realizados os testes em cenas internas, uma segunda categoria de vídeos foi testada. Nesse caso, foram escolhidas cenas mais amplas e em ambientes externos e, portanto, com muitos detalhes o que do que nos exemplos anteriores. Como já mencionado, a estrutura desse tipo de cena introduz uma dificuldade natural ao *pipeline* proposto: o fluxo óptico estimado é esparso, isto é, apenas alguns vértices da cena são seguidos, de modo que uma cena com muitos detalhes potencialmente será composta de pontos espalhados de maneira não uniforme por todos os objetos presentes, sem que apenas um único objeto seja completamente caracterizado.

Além disso, cenas externas introduzem outros tipos de dificuldade que, muitas vezes, não se adéquam às hipóteses de funcionamento dos algoritmos de reconstrução. A mais sensível dessas hipóteses é a necessidade da cena ser estática: isso é claramente obtido nos vídeos anteriores, em que os objetos detectados não se moviam; entretanto, no caso de uma cena com vegetação por exemplo, o efeito do movimento das folhas pelo vento pode acabar deslocando erroneamente as *features* detectadas. Outros exemplos de deslocamentos indesejados são introduzidos pela presença de sombras ou reflexos que, caso detectados pelo algoritmo KLT, introduzem um grande nível de ruído nos algoritmos de reconstrução.

Diferentes testes foram realizados nessa situação. Por exemplo, a área externa de uma casa foi filmada em um dia ensolarado e todos os elementos dificultadores estavam presentes no vídeo: sombras que se moviam com a câmera, uma grande quantidade de vegetação oscilando com o vento e janelas com reflexos de outros objetos da cena. Como esperado, não foi possível estabelecer uma correspondência robusta entre as *features* nos diferentes *frames* e, consequentemente, a reconstrução se tornou inviável, dado que todos os algoritmos subsequentes no *pipeline* são dependentes de que os vértices acompanhados sejam os mesmos conforme os vídeo avança. Um segundo teste foi realizado com estátuas e outros elementos, também em uma área externa e, novamente, os mesmos problemas foram observados.

Em todos esses testes, portanto, o *pipeline* foi incapaz de realizar a reconstrução da cena. Visualmente, os resultados obtidos mostravam nuvens de pontos e trajetórias de câmera incoerentes com os vídeos de entrada e, quantitativamente, o erro de reprojeção observado permaneceu oscilando na ordem de centenas a milhares de pixels. Uma possível solução a ser investigada seria alguma estratégia para aumentar a robustez das *features*, utilizando os métodos descritos anteriormente, como SIFT, aplicados com um espaçamento entre *frames* para permitir maiores variações fotométricas. No entanto, os potenciais movimentos das cenas externas ainda não seriam completamente tratados e, além disso, a performance em velocidade do *pipeline* seria potencialmente reduzida, dada a complexidade desses algoritmos.

# 6 Conclusão

Diante dos resultados expostos nas seções anteriores, foi possível, portanto, estabelecer um *pipeline* de *Structure from Motion* capaz de reconstruir a estrutura tridimensional de uma cena estática, bem como a trajetória da câmera realizada na aquisição da cena, a partir de uma sequência de *frames* que compõem um vídeo. Em primeiro lugar, é importante ressaltar que o seu funcionamento foi validado utilizando-se um conjunto de dados sintéticos, em que a estrutura 3D era completamente conhecida, bem como a posição e orientação da câmera em cada um dos *frames* gerados, de modo que o resultado esperado da reconstrução era conhecido.

Como mostrado anteriormente, o *pipeline* não só foi capaz de reconstruir a cena ideal, isto é, livre de qualquer ruído, mas principalmente que as etapas de otimização adicionadas para corrigir as principais fontes de erro foram bastante efetivas: a versão com BA é capaz de recuperar a resposta esperada mesmo com a presença de forte ruído gaussiano nas projeções, o que não é o caso do *pipeline* em sua versão básica sem essa etapa final. Essa etapa foi fundamental para validar as diferentes etapas da reconstrução e garantir que os diferentes algoritmos implementados, bem como as diferentes conversões de sistema de coordenadas e a biblioteca de visualização, estavam corretos e não eram os responsáveis por eventuais erros que seriam observados em vídeos reais.

Uma vez validada a implementação, foi possível então passar para o tratamento de vídeos reais. Como mostrado, a qualidade da reconstrução mostrou-se bastante dependente do tipo de vídeo considerado. Quando a aquisição é feita de forma suave e continua e a cena é constituída de poucos objetos com formas geométricas relativamente simples e bem definidas, o *pipeline* apresenta uma boa performance. Isso pode ser explicado pelo fato do vídeo, nessas condições, estar mais próximo das condições ideias dos dados sintéticos. É importante destacar que, mesmo com uma inicialização complexa e computacionalmente exigente, além de estar escrito em uma linguagem naturalmente mais lenta, a velocidade de processamento obtida é em torno de 26 *frames* por segundo. Isso mostra que uma eventual adaptação para uma versão capaz de processar o vídeo em tempo real não está tão distante, uma vez que, como mencionado anteriormente, ele foi concebido para ser teoricamente capaz de funcionar dessa forma. Assim, uma possibilidade de trabalho futuro seria a tradução do código para uma linguagem mais rápida, como C++, bem como o

*profiling* para se entender os gargalos de performance e a paralelização de algumas etapas que teriam grande potencial de elevar a velocidade do processamento.

Entretanto, é visível a degradação dos resultados nos casos em que os vídeos são adquiridos de forma menos fluída (com interrupções e retomadas, elevado tremor ou outros tipos de ruídos) ou em que a cena filmada é muita ampla e constituída de muitos detalhes. Esse último problema é inerente ao tipo de reconstrução feita: o algoritmo KLT seleciona uma quantidade limitada de pontos para acompanhar, resultando em um fluxo óptico esparso. Assim, em uma cena com diferentes detalhes, o número de pontos acompanhados não é suficiente para uma boa distinção da estrutura da cena, de modo que a utilização de um fluxo óptico denso (como uma mapa de profundidade) se mostra mais adequada.

O problema de robustez a vídeos mais ruidosos, entretanto, é mais complexo e é possível identificar diferentes caminhos que potencialmente poderiam resolvê-lo. Em primeiro lugar, como observado durante o desenvolvimento, a inicialização das matrizes de projeção e, sobretudo, da nuvem de pontos 3D apresenta um grande impacto no resultado final, a ponto de uma inicialização ruim inviabilizar completamente a reconstrução, mesmo com o uso do *Bundle Adjustment*. Dessa forma, métodos mais complexos do que o implementado e presentes em trabalhos mais recentes poderiam ser implementados com um grande potencial de impactarem positivamente no resultado. Além disso, um estudo mais aprofundado dos diversos parâmetros presentes no *pipeline* poderia ser realizado, para se estabelecer diferentes configurações adaptadas a diferentes tipos de vídeos, por exemplo.

Finalmente, uma outra possibilidade seria a mudança de abordagem. Ao invés de se basear apenas em informações visuais, como é o caso do problema de SfM, seria possível utilizar outras informações vindas de sensores como acelerômetros e giroscópios. Nesse caso, seria possível estimar a dinâmica completa da câmera, em uma abordagem conhecida como *Monocular SLAM*. Trabalhos mais recentes, inclusive, utilizam modelos probabilísticos para estimar a evolução da câmera ao longo do tempo, sendo capazes de captar e tratar as incertezas inerentes ao problema de maneira mais eficiente e robusta. Dessa forma, passar para abordagens desse tipo possui potencial de um grande salto de performance, com a desvantagem de consistir em modelos mais complexos de serem implementados.

## Referências


BAY, H.; TUYTELAARS, T.; GOOL, L. V. Surf: Speeded up robust features. In: SPRINGER. European conference on computer vision. [S.l.], 2006. p. 404–417. 17

BEARDSLEY, P.; TORR, P.; ZISSERMAN, A. 3d model acquisition from extended image sequences. In: SPRINGER. European conference on computer vision. [S.l.], 1996. p. 683–695. 22

BIANCO, S.; CIOCCA, G.; MARELLI, D. Evaluating the performance of structure from motion pipelines. Journal of Imaging, Multidisciplinary Digital Publishing Institute, v. 4, n. 8, p. 98, 2018. 7, 11

BOUGUET, J.-Y. et al. Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm. 30, 31

CALONDER, M. et al. Brief: Binary robust independent elementary features. In: SPRINGER. European conference on computer vision. [S.l.], 2010. p. 778–792. 18

CARLBOM, I.; TERZOPOULOS, D.; HARRIS, K. M. Computer-assisted registration, segmentation, and 3d reconstruction from images of neuronal tissue sections. IEEE Transactions on medical imaging, IEEE, v. 13, n. 2, p. 351–362, 1994. 11

COOPER, M. C. Formal Hierarchical Object Models for Fast Template Matching. The Computer Journal, v. 32, n. 4, p. 351–361, 01 1989. ISSN 0010-4620. Disponível em: ⟨https://doi.org/10.1093/comjnl/32.4.351⟩. 17

ESCRIVá, P. J. D. M.; MENDONçA, V. G. Building Computer Vision Projects with OpenCV 4 and C++: Implement Complex Computer Vision Algorithms and Explore Deep Learning and Face Detection. [S.l.]: Packet, 2019. 31

FAUGERAS, O.; FAUGERAS, O. A. Three-dimensional computer vision: a geometric viewpoint. [S.l.: s.n.], 1993. 35

FITZGIBBON, A. W.; ZISSERMAN, A. Automatic camera recovery for closed or open image sequences. In: SPRINGER. European conference on computer vision. [S.l.], 1998. p. 311–326. 20, 22

GAUGLITZ, S.; HÖLLERER, T.; TURK, M. Evaluation of interest point detectors and feature descriptors for visual tracking. International journal of computer vision, Springer, v. 94, n. 3, p. 335, 2011. 18

HARTLEY, R.; ZISSERMAN, A. Multiple view geometry in computer vision. [S.l.]: Cambridge university press, 2003. 12, 14, 36, 37

HARTLEY, R. I. Estimation of relative camera positions for uncalibrated cameras. In: SPRINGER. European conference on computer vision. [S.l.], 1992. p. 579–587. 20

HARTLEY, R. I. In defense of the eight-point algorithm. IEEE Transactions on pattern analysis and machine intelligence, IEEE, v. 19, n. 6, p. 580–593, 1997. 14, 19

JAVERNICK, L.; BRASINGTON, J.; CARUSO, B. Modeling the topography of shallow braided rivers using structure-from-motion photogrammetry. Geomorphology, Elsevier, v. 213, p. 166–182, 2014. 11

KARAMI, E.; PRASAD, S.; SHEHATA, M. Image matching using sift, surf, brief and orb: performance comparison for distorted images. arXiv preprint arXiv:1710.02726, 2017. 13

KLEIN, G.; MURRAY, D. Parallel tracking and mapping for small ar workspaces. In: IEEE. 2007 6th IEEE and ACM international symposium on mixed and augmented reality. [S.l.], 2007. p. 225–234. 22

KROEGER, T.; GOOL, L. V. Video registration to sfm models. In: SPRINGER. European Conference on Computer Vision. [S.l.], 2014. p. 1–16. 11

LEONARD, S. et al. Image-based navigation for functional endoscopic sinus surgery using structure from motion. In: INTERNATIONAL SOCIETY FOR OPTICS AND PHOTONICS. Medical Imaging 2016: Image Processing. [S.l.], 2016. v. 9784, p. 97840V. 11

LEPETIT, V.; MORENO-NOGUER, F.; FUA, P. Epnp: An accurate o (n) solution to the pnp problem. International journal of computer vision, Springer, v. 81, n. 2, p. 155, 2009. 39

LONGUET-HIGGINS, H. C. A computer algorithm for reconstructing a scene from two projections. Nature, Springer, v. 293, n. 5828, p. 133–135, 1981. 13, 14, 19

LOWE, D. G. Distinctive image features from scale-invariant keypoints. International journal of computer vision, Springer, v. 60, n. 2, p. 91–110, 2004. 13, 16

LUCAS, B. D.; KANADE, T. et al. An iterative image registration technique with an application to stereo vision. Vancouver, British Columbia, 1981. 13, 18

MANCINI, F. et al. Using unmanned aerial vehicles (uav) for high-resolution reconstruction of topography: The structure from motion approach on coastal environments. Remote Sensing, Multidisciplinary Digital Publishing Institute, v. 5, n. 12, p. 6880–6898, 2013. 11

MORÉ, J. J. The levenberg-marquardt algorithm: implementation and theory. In: Numerical analysis. [S.l.]: Springer, 1978. p. 105–116. 41, 42

NEWCOMBE, R. A.; DAVISON, A. J. Live dense reconstruction with a single moving camera. In: IEEE. 2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. [S.l.], 2010. p. 1498–1505. 22

NISTÉR, D. An efficient solution to the five-point relative pose problem. IEEE transactions on pattern analysis and machine intelligence, IEEE, v. 26, n. 6, p. 756–770, 2004. 14, 20, 34, 35, 36

PIZZOLI, M.; FORSTER, C.; SCARAMUZZA, D. Remode: Probabilistic, monocular dense reconstruction in real time. In: IEEE. 2014 IEEE International Conference on Robotics and Automation (ICRA). [S.l.], 2014. p. 2609–2616. 23

QUAN, H.; WU, M. A real-time sfm method in augmented reality. In: SPRINGER. Proceedings of the 2012 International Conference on Information Technology and Software Engineering. [S.l.], 2013. p. 841–848. 11

REMONDINO, F. Heritage recording and 3d modeling with photogrammetry and 3d scanning. Remote sensing, Molecular Diversity Preservation International, v. 3, n. 6, p. 1104–1138, 2011. 11

RESCH, B. et al. Scalable structure from motion for densely sampled videos. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. [S.l.: s.n.], 2015. p. 3936–3944. 23

RONCELLA, R.; RE, C.; FORLANI, G. Performance evaluation of a structure and motion strategy in architecture and cultural heritage. Int. Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences, v. 38, n. 5/W16, p. 285–292, 2011. 11

ROSTEN, E.; DRUMMOND, T. Machine learning for high-speed corner detection. In: SPRINGER. European conference on computer vision. [S.l.], 2006. p. 430–443. 17

RUBLEE, E. et al. Orb: An efficient alternative to sift or surf. In: IEEE. 2011 International conference on computer vision. [S.l.], 2011. p. 2564–2571. 18

SHI, J. et al. Good features to track. In: IEEE. 1994 Proceedings of IEEE conference on computer vision and pattern recognition. [S.l.], 1994. p. 593–600. 18

SNAVELY, N.; SEITZ, S. M.; SZELISKI, R. Modeling the world from internet photo collections. International journal of computer vision, Springer, v. 80, n. 2, p. 189–210, 2008. 7, 13, 15

STEWÉNIUS, H. et al. A minimal solution for relative pose with unknown focal length. Image and Vision Computing, Elsevier, v. 26, n. 7, p. 871–877, 2008. 20

STURM, P.; TRIGGS, B. A factorization based algorithm for multi-image projective structure and motion. In: SPRINGER. European conference on computer vision. [S.l.], 1996. p. 709–720. 21

TOMASI, C.; KANADE, T. Detection and tracking of point features. School of Computer Science, Carnegie Mellon Univ. Pittsburgh, 1991. 13, 18

TOMASI, C.; KANADE, T. Shape and motion from image streams under orthography: a factorization method. International journal of computer vision, Springer, v. 9, n. 2, p. 137–154, 1992. 21, 22

TORR, P. H.; MURRAY, D. W. The development and comparison of robust methods for estimating the fundamental matrix. International journal of computer vision, Springer, v. 24, n. 3, p. 271–300, 1997. 19

TRIGGS, B. et al. Bundle adjustment—a modern synthesis. In: SPRINGER. International workshop on vision algorithms. [S.l.], 1999. p. 298–372. 21

TUYTELAARS, T.; MIKOLAJCZYK, K. et al. Local invariant feature detectors: a survey. Foundations and trends® in computer graphics and vision, Now Publishers, Inc., v. 3, n. 3, p. 177–280, 2008. 16

ZHANG, Z. A flexible new technique for camera calibration. IEEE Transactions on pattern analysis and machine intelligence, IEEE, v. 22, n. 11, p. 1330–1334, 2000. 45