

UNIVERSIDADE DE SÃO PAULO – ESCOLA POLITÉCNICA

Aplicação em tempo real de *optical flow* utilizando algoritmo de Lucas-Kanade

Andre de O. Botelho*

*POLI-USP, São Paulo, Brazil

e-mail: andre.botelho@usp.br

Abstract – Este artigo tem como objetivo apresentar aplicação em tempo real do clássico algoritmo de Lucas-Kanade em plataforma embarcadas atuais. Com isso pode ser mostrado que algoritmos clássicos podem ainda ser utilizados uma vez que o poder de processamento atual é muito superior ao da época.

Palavras-chave: Lucas-Kanade, *optical flow*, aplicação em tempo real

Introdução

A área de visão computacional teve grandes avanços nas décadas de 70 e 80 em termos de eficientização de algoritmos. Em 1981 o algoritmo de *optical flow* proposto por Lucas-Kanade[1] foi um deles.

Este algoritmo foi extremamente inovador e provou-se mais eficiente que as soluções anteriores ([2],[3]). O algoritmo se tornou tão importante que uma série de cursos de graduação e pós pedem para que seus alunos implementem-o e, inclusive, já está disponível em uma série de bibliotecas para visão computacional, OpenCV[4] é um desses exemplos.

Neste artigo eu apresento uma aplicação em tempo real utilizando a plataforma Raspberry Pi e a biblioteca OpenCV (*Open Computer Vision*), linguagem Python para *tracking* de objetos utilizando o algoritmo de Lucas-Kanade [LK].

Com isso pretendo demonstrar que algoritmos clássicos podem ser utilizados mais facilmente hoje em dia e que sua aplicação ainda gera bons resultados e robustos

Materiais e métodos

Para processamento das imagens e apresentação de resultados utilizei a biblioteca OpenCV. Esta é uma das bibliotecas padrões mais usadas pela comunidade (não necessariamente científica) para processamento de imagens e visão computacional. Dentre os muitos algoritmos já

implementados por *default* nesta biblioteca está o algoritmo de Lucas-Kanade.

Para o desenvolvimento do software utilizei o software *Thonny Python IDE* (V2.1.16) na plataforma RaspberryPi 3. Esta plataforma conta com 4 CPUs RM Cortex-A53 com clock de 1.2GHz. É uma plataforma potente para seu tamanho, porém bem aquém da tecnologia de ponta disponível hoje em dia.

Esta plataforma foi escolhida por também ser bastante utilizada globalmente e demonstra a facilidade de aplicar algoritmos clássicos em plataformas simples em tempo real.

Foi utilizado um vídeo artificial de uma circunferência branca em fundo preto para analisar a eficiência do *script* e, por fim, um vídeo gravado e processado em tempo real pela própria RaspberryPi.

Em um caso de translação 2D o algoritmo LK quer estimar um vetor $h = (u,v)$ de deslocamento entre duas imagens, como visto abaixo:

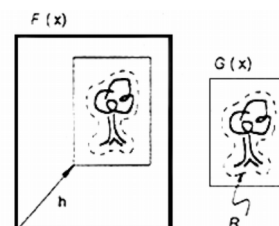


Figura 1: Vetor de Deslocamento h

Este vetor é obtido à partir das suas duas derivadas espaciais (aqui chamadas de f_x e f_y) e a derivada temporal (f_t) segundo a equação:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum_i f_{x_i}^2 & \sum_i f_{x_i} f_{y_i} \\ \sum_i f_{x_i} f_{y_i} & \sum_i f_{y_i}^2 \end{bmatrix}^{-1} * \begin{bmatrix} -\sum_i f_{t_i} f_{x_i} \\ -\sum_i f_{t_i} f_{y_i} \end{bmatrix}$$

Esta formula, apesar de parecer simples, é o que demonstra uma robustez no algoritmo. Apesar de ter a inversão de uma matriz nos cálculos ela é facilmente calculada por se tratar de uma matriz 2×2 .

Este artigo trata de aplicação em tempo real deste algoritmo e, portanto, algumas restrições tem que ser consideradas. A primeira delas é que não necessariamente todos os pontos são interessantes para *tracking*. Segundo que objetos em uma imagem são delimitados por “cantos”, detectar cantos de um objeto sólido e calcular o movimento destes representa o movimento do sólido como um todo. Terceiros: o movimento entre frames pode ser maior que um pixel, algum método deve ser utilizado para tratar esses casos.

A partir da suposição 2 podemos resolver o primeiro problema utilizando o algoritmo proposto por Shi-Tomasi [5]. Neste artigo os autores demonstram a razão e como escolher bons pontos em uma imagem para *tracking*. Esta função está implementada, também, na biblioteca OpenCV.

Uma vez que temos um número menor de pontos para processar podemos considerar agora o terceiro problema, movimentos que são maiores que um pixel. Para este problema foi utilizado o recurso de pirâmides de uma imagem. Essa técnica consiste em pegar a imagem original, diminuí-la L vezes, calcular o *optical flow* nestes níveis e ir subindo, sempre levando em conta os níveis mais baixos. Desta maneira, movimentos que são maiores que um pixel no tamanho original podem passar a ser sub-pixel em um dos níveis e, com isso, o algoritmo de LK consegue contabilizá-los. Uma figura que exemplifica isto pode ser visto na Figura 2, uma outra aplicação do mesmo método pode ser visto em [7].

O algoritmo descrito foi implementado em Python utilizando, principalmente, a biblioteca OpenCV e foi baseado em um de seus exemplos [6]. Foi utilizada a câmera da Raspberry pi para captura das imagens no vídeo ao vivo, as imagens foram armazenadas no disco interno e o tempo entre elas para estatísticas foram imprimidos na tela enquanto o *script* rodava. Durante o processamento foram traçados os movimentos recentes dos pontos de interesse encontrados e o resultado mostrado em tempo real.

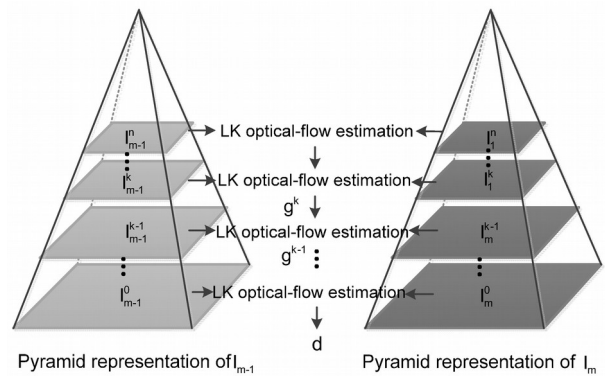


Figura 2: Exemplo do algoritmo de Pirâmide

Resultados

O primeiro teste foi feito com um vídeo de um objeto branco em movimento pseudo-aleatório em um fundo preto. Exemplos de *frames* do vídeo pode ser visto na Figura 3, o vídeo na íntegra pode ser visto através do link: <https://www.youtube.com/watch?v=LmcZc8DPPLY>. O resultado do *tracking* pode ser visto em <https://www.youtube.com/watch?v=f2MOtBxYBKo>, exemplos de *frames* visto na Figura 4.



Figura 3: Exemplo de vídeo processado

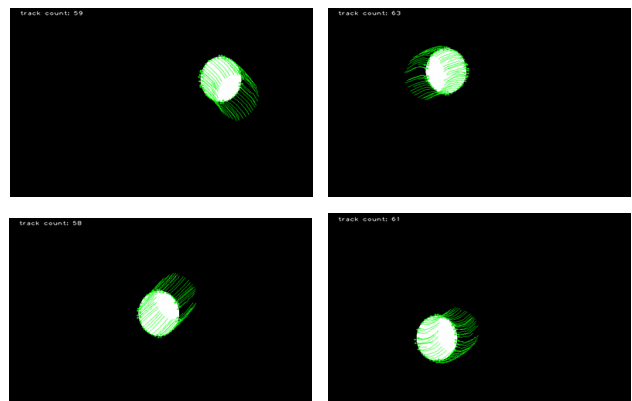


Figura 4: Vídeo processado e *tracking*

Como podemos ver na no vídeo o *tracking* é feito com sucesso e o algoritmo funciona com precisão. Analisando o log, Figura 5, vemos que o *framerate* (quantidade de frames processado por segundo) não é muito satisfatório, cerca de 2 fps (*frames per second*). Isso se deve ao fato de que as imagens originais são muito grandes e muito

pesadas para o hardware em questão, o que faz com que o hardware demore muito para ler as imagens do disco.

Saving Image 107	on time: 55.7493133509426, from previous: 0.3441099262200049, fps = 1.637006297490490
Saving Image 121	on time: 56.35718883381653, from previous: 0.41104722023610254, fps = 2.40328516125627
Saving Image 122	on time: 56.78945654296875, from previous: 0.4232757891522217, fps = 2.362563311757645
Saving Image 123	on time: 57.2797911671446, from previous: 0.4993457374572754, fps = 2.002605230274499
Saving Image 124	on time: 57.7315666471741, from previous: 0.4517955780929297, fps = 2.213390410614325
Saving Image 125	on time: 58.151297092437744, from previous: 0.4197163977203369, fps = 2.3825952580937177
Saving Image 126	on time: 58.71673893926529, from previous: 0.5054418466475342, fps = 1.765264624302352
Saving Image 127	on time: 59.1430944786987, from previous: 0.410150805845947, fps = 2.3914551168013076
Saving Image 128	on time: 59.55486536026001, from previous: 0.4199765123901367, fps = 2.381115568019772
Saving Image 129	on time: 60.11119747161865, from previous: 0.5562543899018555, fps = 2.1877398310059578
Saving Image 130	on time: 60.540042472457896, from previous: 0.4358227252980205, fps = 2.294519914549519
Saving Image 131	on time: 60.940484681259155, from previous: 0.35710620880126953, fps = 2.8002873524848244
Saving Image 132	on time: 61.4722395440674, from previous: 0.5681846141815186, fps = 1.75999139513948
Saving Image 133	on time: 61.92709992050171, from previous: 0.4553762506135516, fps = 2.125994477380307
Saving Image 134	on time: 62.376821756362915, from previous: 0.4492183586120695, fps = 2.2262122118989026
Saving Image 135	on time: 62.8353212776184, from previous: 0.5585603713989258, fps = 1.799316770848476
Saving Image 136	on time: 63.388018521499634, from previous: 0.433436393777937, fps = 2.3071435958027773
Saving Image 137	on time: 63.79313926560974, from previous: 0.424319744110074, fps = 2.356713336722085
Saving Image 138	on time: 64.26715652503967, from previous: 0.48361825842983164, fps = 2.06774657594307
Saving Image 139	on time: 64.7104423046112, from previous: 0.4336857795715332, fps = 2.305816900461807
Saving Image 140	on time: 65.1337503606033, from previous: 0.42331606189209694, fps = 2.3623014300404501
Saving Image 141	on time: 65.6830585029992, from previous: 0.5493061937866211, fps = 1.8204981744252868
Saving Image 142	on time: 66.120579247063, from previous: 0.43752074241630184, fps = 2.285605922399199
Saving Image 143	on time: 66.57468196182225, from previous: 0.45410695347595215, fps = 2.20211405923063
Saving Image 144	on time: 67.1410466194153, from previous: 0.560164857592773, fps = 1.7639283940768434
Saving Image 145	on time: 67.5103172559204, from previous: 0.368712636505127, fps = 2.7121390138849644
Saving Image 146	on time: 67.9567928314209, from previous: 0.446475568208574, fps = 2.239764526709152
Saving Image 147	on time: 68.51567544743877, from previous: 0.5508627133170711, fps = 1.7892941846294543
Saving Image 148	on time: 68.94566321372966, from previous: 0.42998766899188887, fps = 2.3256480874123024
Saving Image 149	on time: 69.37793445587156, from previous: 0.43227124214172863, fps = 2.313362311694429
Saving Image 150	on time: 69.9806940832679, from previous: 0.6080724246521, fps = 1.6454917851484
Saving Image 151	on time: 70.4218528707214, from previous: 0.4358462873535156, fps = 2.294386654354971
Saving Image 152	on time: 70.8568014827576, from previous: 0.4348498212036133, fps = 2.2991205432416018
Saving Image 153	on time: 71.3747865104675, from previous: 0.5166786027709661, fps = 1.935445262783595
Saving Image 154	on time: 71.820818955866267, from previous: 0.4407621899551931, fps = 2.260185861861319
Saving Image 155	on time: 72.27088165283203, from previous: 0.4567008014836523, fps = 2.189617351120466
Saving Image 156	on time: 72.84327483177185, from previous: 0.5723931789398195, fps = 1.7470508678181482
Saving Image 157	on time: 73.2842782695606, from previous: 0.44095454788286, fps = 2.2875976873671886

Figura 5: Log do processamento do vídeo

Uma vez processado o vídeo começamos a analisar o processamento do vídeo em tempo real. Pequenas alterações no código tiveram que ser feitas para obtenção da imagem, agora as imagens já são adquiridas no formato (640,480) e, com isso, o processamento fica mais rápido. Na Figura 6 vemos exemplo de imagens processadas e, na Figura 7, o log. Vemo que o *fps* fica próximo de 6 a 7 frames por segundo, com picos de 10. Tal *fps* não é alto para muitas aplicações, porém para algumas, como *tracking* de pessoas e objetos por camera de segurança, é melhor do que muitas marcas no mercado. O vídeo pode ser visto em https://www.youtube.com/watch?v=7l_sMI177ag



Figura 6: Vídeo processado em tempo real

Saving Image 37	on time: 6.460829496383667, from previous: 0.17494726181030273, fps = 5.716008296742084
Saving Image 38	on time: 6.6155908019775757, from previous: 0.15476313438208084, fps = 6.46156311044559
Saving Image 39	on time: 6.836912631988525, from previous: 0.22132121212127855, fps = 4.518307298034776
Saving Image 40	on time: 6.972068786621094, from previous: 0.13515615463256836, fps = 7.39849151128486
Saving Image 41	on time: 7.115329768757324, from previous: 0.1432000213623047, fps = 6.880315406173653
Saving Image 42	on time: 7.335241794586182, from previous: 0.12991300508285742, fps = 4.547252656708392
Saving Image 43	on time: 7.497310983078083, from previous: 0.1620711049181213, fps = 6.169999720653544
Saving Image 44	on time: 7.661039530929656, from previous: 0.1637115478519525, fps = 6.10393455922835
Saving Image 45	on time: 7.8765764236450195, from previous: 0.2155458927154541, fps = 4.63938320188095
Saving Image 46	on time: 8.020312671076247, from previous: 0.1515562534322754, fps = 6.59821006603339
Saving Image 47	on time: 8.170114517211914, from previous: 0.14191840133667, fps = 7.0431542446453905
Saving Image 48	on time: 8.36622373884892, from previous: 0.1961890564727832, fps = 5.099293565536551
Saving Image 49	on time: 8.593203392028809, from previous: 0.1369798183441162, fps = 7.30934546679358
Saving Image 50	on time: 8.640110731124878, from previous: 0.13697339066934, fps = 7.30421325266232
Saving Image 51	on time: 8.780471801578121, from previous: 0.1403610706293457, fps = 7.124482561236068
Saving Image 52	on time: 8.87216591835022, from previous: 0.0916941165024073, fps = 10.80502511902197
Saving Image 53	on time: 8.82262469484227, from previous: 0.1504590113220215, fps = 6.646326641534787
Saving Image 54	on time: 9.21137928627075, from previous: 0.10875432014485332, fps = 5.297891959508079
Saving Image 55	on time: 9.352434746185303, from previous: 0.14486446582754, fps = 6.088642670842007
Saving Image 56	on time: 9.504241466522217, from previous: 0.14789772833691406, fps = 6.761429437292928
Saving Image 57	on time: 9.602325039986330, from previous: 0.1780843734741211, fps = 5.615315821860631
Saving Image 58	on time: 9.82398533821196, from previous: 0.1416594821472168, fps = 7.05918077866707
Saving Image 59	on time: 9.962024927139282, from previous: 0.1380395889282266, fps = 7.244298594079242
Saving Image 60	on time: 10.151008728179932, from previous: 0.1090738014064941, fps = 5.2088406507109515
Saving Image 61	on time: 10.285516738891602, from previous: 0.1344101071166992, fps = 7.439475230723039
Saving Image 62	on time: 10.424899995025635, from previous: 0.1393742561340332, fps = 7.174926186696267
Saving Image 63	on time: 10.63114047059476, from previous: 0.20624947547912598, fps = 4.84849717858002
Saving Image 64	on time: 10.78120231628418, from previous: 0.15060184577941895, fps = 6.663910908195915
Saving Image 65	on time: 10.93378233909607, from previous: 0.13258002281188965, fps = 6.553913737424898
Saving Image 66	on time: 11.134312723484877, from previous: 0.2054688725289762, fps = 4.88613537620625
Saving Image 67	on time: 11.308948858261108, from previous: 0.16661763192123145, fps = 6.001765770664454
Saving Image 68	on time: 11.458142518997192, from previous: 0.15719368073608398, fps = 6.361379697955448
Saving Image 69	on time: 11.613009780513184, from previous: 0.2028672895159912, fps = 4.9235139182938
Saving Image 70	on time: 11.82908821109597, from previous: 0.1680784254638672, fps = 5.49603672202584
Saving Image 71	on time: 11.98704949591044, from previous: 0.1584617494918075, fps = 6.312463879791948
Saving Image 72	on time: 12.19044423103325, from previous: 0.2023927132687988, fps = 4.9275827458073705

Figura 7: Log do vídeo processado em tempo real

Conclusões

Este artigo teve como objetivo demonstrar o uso do algoritmo de Lucas-Kanade em uma plataforma atual com o intuito de demonstrar que algoritmos tidos como clássicos podem ainda ser usados e consegue performar em plataformas atuais.

A aplicação em tempo real desejada foi de *tracking*, onde pontos de interesse tido como interessantes foram encontrados e seus movimentos estimados em tempo real.

Foi utilizada a biblioteca do OpenCV e a plataforma utilizada foi a Raspberry Pi, ambos são muito utilizados hoje em dia e possuem uma grande comunidade apoiando-os.

O resultado foi mostrado em tempo real, gravados em um vídeo e o log durante o processamento foi capturado. O vídeo em tempo real, cujas imagens já são capturadas em tamanho apropriado, possuíam tempo de processamento compatível com a aplicação desejada. Desta maneira conseguiu-se demonstrar com sucesso a implementação deste algoritmo em plataformas atuais e em tempo real.

Todos os códigos e imagens estão disponíveis no github.com/aobotelho/PTC5750

Referências

- [1] B.D. Lucas and T. Kanade, “An Iterative Image Registration Technique with an Application to Stereo Vision”, DARPA Image Understanding Workshop, 1981, pp121–130 (see also IJCAI’81, pp674–679).(disponível em: https://cseweb.ucsd.edu/classes/sp02/cse252/lucas_kanade81.pdf)
- [2] Barnea, Daniel I. and Silverman, Harvey F. "A Class of Algorithms for Fast Digital Image

Registration." IEEE Transactions on Computers C-21.2 (February 1972), 179- 186.

[3] Moravec, Hans. P. Visual Mapping by a Robot Rover. Sixth International Joint Conference on Artificial Intelligence, Tokyo, August, 1979, pp. 598-600

[4] <https://opencv.org/>

[5] J. Shi, C. Tomasi, "*Good Features to Track*", IEEE Conference in Computer Vision and Pattern Recognition, Seattle, June 1994

[6] https://docs.opencv.org/3.3.1/d7/d8b/tutorial_py_lucas_kanade.html

[7] <http://www.mdpi.com/2072-666X/6/4/487/htm>