Método para detección y seguimiento de objetos con aplicaciones en Realidad Aumentada

Christian Nicolás Pfarher

Punto de control N° 3:

Métodos de extracción de características en imágenes parte 2

Matching de características entre imágenes

Director
Dr. Enrique Marcelo Albornoz

Codirector
Dr. César Martínez

29 de junio de 2012

Ingeniería Informática
Facultad de Ingeniería y Ciencias Hídricas
UNIVERSIDAD NACIONAL DEL LITORAL

Índice

1. Métodos de extracción de características en imágenes parte 2

Para aplicar el método SURF descripto en el informe anterior, se utiliza el código que ofrece la librería OpenCV (Open Source Computer Vision)[1]. La función que lo implementa está descrita a continuación:

```
void cvExtractSURF(const CvArr* image,
                   const CvArr* mask,
                   CvSeq** keypoints,
                   CvSeq** descriptors,
                   CvMemStorage* storage,
                   CvSURFParams params)
```

los parámetros[2] de está función son los que se presentan a continuación:

- **image** (entrada): imagen de entrada en escala de grises de 8 bits.
- **mask** (entrada opcional): máscara de 8 bits. Las características sólo son buscadas en áreas que contienen más de un 50 % de los píxeles distintos de cero.
- **keypoints** (salida): puntero doble a una secuencia que representa los puntos claves. La secuencia de la estructura CvSURFPoint es la siguiente:

```
typedef struct CvSURFPoint {
   CvPoint2D32f pt;
   int laplacian;
   int size;
   float dir;
   float hessian;
} CvSURFPoint;
```

donde:

`pt:` es la posición de la característica en la imagen.

`laplacian:` representa el signo del laplaciando en el punto, los valores posibles son: -1, 0 o +1. Puede ser usado para la comparación de características (normalmente características con laplacianos de diferentes signos no coincidirán).

`size:` es el tamaño de la característica.

`dir:` es la orientación de la característica (0-360 grados).

[1] `http://SourceForge.net/projects/opencvlibrary`

[2] `http://opencv.willowgarage.com/documentation/feature_detection.html#cvExtractSURF`

`hessian`: es el valor del hessiano (puede ser usado para estimar aproximadamente la intensidad de las características). Relacionado con `params.hessianThreshold`.

- `descriptors` (salida opcional): puntero doble a una secuencia que representa el descriptor. Dependiendo del valor de *params.extended*, cada elemento de la secuencia contendrá un vector de punto flotante (CV_32F) de 64 o 128 elementos. Si el parámetro es NULL, el descriptor no es calculado.

- `storage`: almacenamiento de memoria donde serán guardados los puntos claves y descriptores.

- `params`: otros parámetros del algoritmo con la estructura de CvSURFParams:

```
typedef struct CvSURFParams
{   int extended;
    double hessianThreshold;
    int nOctaves;
    int nOctaveLayers;
} CvSURFParams;

CvSURFParams cvSURFParams(double hessianThreshold,
                          int extended=0);
```

donde:

`extended`: tamaño del descriptor. 0 indica un descriptor básico (64 elementos), mientras que 1 significa un descriptor extendido (128 elementos).

`hessianThreshold`: umbral para el hessiano del punto clave de las características, se extraen sólo aquellas cuyo hessiano supere el umbral. Los valores recomendables por defectos están entre 300 y 500 (dependen del promedio local de contraste y nitidez de la imagen).

`nOctaves`: es el número de octavas a ser usadas en la extracción. Con cada octava adicional, el número de características es duplicado. (3 por defecto).

`nOctaveLayers`: es el número de capas en cada octava. (4 por defecto).

El objetivo de la función *cvExtractSURF* es buscar características robustas en la imagen, como se ha descripto en el informe anterior y en [BETV08]. Por cada característica hallada retorna su localización, tamaño, orientación y opcionalmente el descriptor (básico o extendido). Su uso puede ser la localización y seguimiento de objetos, emparejamiento de imágenes, entre otras aplicaciones.

2. Correspondencia de características entre imágenes

Una vez que han sido identificados los puntos claves y sus descriptores asociados sobre la imagen, se procede con la búsqueda de correspondencia (en caso de existir) entre la imagen patrón y la imagen del flujo de vídeo. De esta forma se busca identificar si el objeto está presente en la imagen capturada.

Como se ha visto, un descriptor SURF [BETV08, BETV08] es un vector de 64 dimensiones que caracteriza localmente una vecindad de un punto. En una aplicación típica, gran cantidad de descriptores SURF son extraídos de la imagen y la consulta consiste en buscar los vectores que mejor coincidan, entre los de la imagen objeto y los de la imagen del flujo de video. Para ello, existen diversas técnicas, entre las cuales se presenta la denominada *búsqueda del vecino más cercano* (del inglés, Nearest Neighbor Search o NNS) [AMN+98].

2.1. Búsqueda del vecino más cercano

El método de búsqueda del vecino más cercano es de utilidad en una gran variedad de aplicaciones como el reconocimiento de imágenes, la compresión de datos, el reconocimiento de patrones y su clasificación, el aprendizaje máquina, los sistemas de recuperación de documentos, estadísticas y análisis de datos, entre otros.

El método de la búsqueda del vecino más cercano, es un problema de optimización que intenta buscar los puntos más cercanos en un espacio métrico. Éste, puede definirse de la siguiente forma: dado un conjunto de puntos $P = \{p_{1,...,}p_n\}$ en un espacio métrico M y un punto de consulta $q \in M$, encontrar el/los punto/s más cercano/s a q en P donde M es un espacio euclídeo d-dimensional y las distancia es medida mediante la distancia euclídea o Manhattan.

Resolver este tipo de problemas no resulta trivial en espacios de grandes dimensiones, además no es usual encontrar algoritmos que posean un rendimiento mayor al de la búsqueda lineal (también conocida como "búsqueda por fuerza bruta"), la cual resulta ser muy costosa y aveces inaplicable para muchas aplicaciones. Es por esto, que se ha generado un gran interés en algoritmos [3] que puedan realizar la búsqueda del vecino más cercano de forma aproximada, con lo cual es posible lograr mejoras significativas en tiempo de ejecución con errores de precisión relativamente pequeños [BL97] .

Existe una extensa variedad de publicaciones [ML09, AMN+98, BL97, LMGY04] que abordan el algoritmo de búsqueda del vecino más cercano de forma aproximada, en los que su rendimiento, varía dependiendo de las pro-

[3] `http://www.cs.umd.edu/~mount/ANN/`

piedades del conjunto de datos, tales como: la dimensionalidad, correlación y características de agrupación y tamaño. Uno de los algoritmos más usados para la búsqueda del vecino más cercano es KD-tree [BL97, FBF77] (básicamente se trata de un árbol binario balanceado de búsqueda), el cual ha dado buenos resultados para la búsqueda exacta en datos de bajas dimensiones. Este algoritmo, ha sido modificado en diversos trabajos [MM07, AMN+98, BL97, Bri95, FN75, NS06, LMS06, LMGY04, AH08] para lograr reducir los tiempos de ejecución con datos de grandes dimensiones, a coste de obtener resultados aproximados. En el trabajo de Slipa-Anan y Hartley [Ric04, AH08], se propuso el uso de múltiples árboles KD aleatorios conocidos por su término en inglés como **Randomized KD-Tree**, donde se ha encontrado que dicho método, obtiene resultados satisfactorios en un amplio rango de problemas [ML09].

2.1.1. K-vecinos más cercanos

Existe una variante al algoritmo NNS denominada k-NN (k-Nearest Neighbor). A diferencia del anterior, se debe especificar un parámetro K que establece cuantos vecinos influencian la clasificación de un punto. Estos vecinos son definidos utilizando una distancia métrica, por ejemplo la euclídea. Así, en el caso $K = 1$ se está en presencia del algoritmo NNS. Si $K > 1$ se utilizan los K vecinos más próximos al punto y se lo clasifica como perteneciente a la clase mayoritaria.

2.2. El algoritmo de árboles KD aleatorio

Como se mencionó anteriormente, el algoritmo KD-tree clásico [FBF77] resulta eficiente con datos de bajas dimensiones, pero su rendimiento se ve afectado rápidamente al aumentar la dimensionalidad. Para obtener una velocidad mayor a la de la búsqueda lineal se hace necesario establecer una búsqueda aproximada del vecino más cercano. Esto mejora el tiempo de búsqueda pero, como contrapartida, el algoritmo no siempre da como resultado el vecino más cercano.

Los elementos guardados en el árbol KD-tree, son vectores de altas dimensiones. En el primer nivel (la raíz del árbol), los datos son divididos en dos mitades por un hiper plano ortogonal a una dimensión elegida con un valor de umbral. Generalmente, esta división se realiza con la media, en la dimensión con la mayor varianza del conjunto de datos. Se utiliza la media ya que con características visuales provistas por SIFT o SURF, es la medida que presenta el mejor rendimiento [ML09]. Mediante la comparación del vector de consulta con el "valor de partición", es fácil determinar a que mitad del conjunto de datos pertenece el vector de consulta. Cada una de las mitades de los datos es dividida de igual manera y en forma recursiva, para lograr crear un árbol binario completamente balanceado. Cada nodo

de la parte inferior del árbol corresponde a un punto simple del conjunto de datos. Sin embargo, en algunas aplicaciones los nodos hojas pueden tener más de un punto.

Slipa-Anan y Hartley han propuesto una versión del algoritmo de árbol KD en el que múltiples árboles KD aleatorios son creados [Ric04, AH08]. El algoritmo de árboles KD original, divide los datos por la mitad en cada nivel del árbol sobre la dimensión para la cual los datos exhiben la mayor varianza. A diferencia de éstos, los árboles aleatorios son construidos seleccionando la dimensión de división de forma aleatoria, sobre las primeras D dimensiones en las que los datos tienen mayor varianza. Se usa el valor fijo $D = 5$ que resulta el más adecuado para diferentes datos [ML09].

Cuando se realiza la búsqueda en el árbol, una cola con prioridad es mantenida a través de todos los arboles aleatorios, por lo que la búsqueda estará ordenada mediante el incremento de la distancia a cada nodo del borde. El grado de aproximación, se determina mediante el examen de un número fijo de nodos hoja, en cuyo momento la búsqueda termina y devuelve los mejores candidatos.

Se debe tener en cuenta que la cantidad de memoria utilizada aumenta linealmente con el número de árboles aleatorios, una característica poco feliz y cuya importancia no resulta menor a la hora de monitorizar la sobrecarga del sistema.

2.3. Búsqueda de correspondencia - Implementación

La librería Rápida para aproximación de vecinos más cercanos (FLANN - Fast library for Approximate Nearest Neighbors) [ML09], contiene una colección de algoritmos optimizados para la búsqueda rápida del vecino más cercano, en grandes conjuntos de datos y con características de altas dimensiones. OpenCV, posee una interfaz[4] a la librería FLANN[5] la cual será utilizada para poder resolver el problema de la búsqueda. A continuación se presenta una descripción de los principales métodos de FLANN.

La clase index de FLANN se utiliza para abstraer los diferentes tipos de índices a usar para las búsqueda del vecino más cercano.

```
namespace cv
{
namespace flann
{
    template <typename T>
    class Index_
    {
    public:
            Index_(const Mat& features, const IndexParams& params);
            ~Index_();
            ...
```

[4] http://opencv.willowgarage.com/documentation/cpp/flann_fast_approximate_nearest_neighbor_search.html

[5] http://people.cs.ubc.ca/~mariusm/index.php/FLANN/FLANN

```
        void knnSearch(const Mat& queries,
                       Mat& indices,
                       Mat& dists,
                       int knn,
                       const SearchParams& params);
        ...
        const IndexParams* getIndexParameters();
    };
    typedef Index_<float> Index;
} } // namespace cv::flann
```

El constructor de la clase

```
Index_<T>::Index_(const Mat& features, const IndexParams& params)
```

construye un índice de búsqueda del vecino más cercano para un conjunto de datos dado. Los parámetros [6] son:

- **features:** matriz que contiene las características (puntos) a indexar. Se almacena un punto por cada fila de la matriz; el tamaño de la matriz es CxD, donde C es la cantidad de características y D es la dimensionalidad.
- **params:** estructura que contiene los parámetros del índice. El tipo del índice será construido dependiendo del tipo de este parámetro. Los parámetros[7] posibles son:
  - **LinearIndexParams**: el índice lleva a cabo una búsqueda de fuerza bruta lineal.
  - **KDTreeIndexParams**: el índice construido consiste en un conjunto de árboles aleatorios KD (randomized KD-trees) que se buscará en paralelo.

    ```
    struct KDTreeIndexParams : public IndexParams{
        KDTreeIndexParams( int trees = 4 );
    };
    ```

    trees: el número de árboles KD paralelos a usar. Valores que producen buenos resultados se encuentran entre 1 y 16, dependiendo del conjunto de datos.
  - **KMeansIndexParams**: el índice construido será un árbol k-means jerárquico.

[6] http://opencv.willowgarage.com/documentation/cpp/flann_fast_approximate_nearest_neighbor_search.html#cv::flann::Index_

[7] http://opencv.willowgarage.com/documentation/cpp/flann_fast_approximate_

- **CompositeIndexParams**: el índice creado combina árboles aleatorios KD y k-means jerárquicos.
- **AutotunedIndexParams**: el índice creado será seleccionado automáticamente para ofrecer el mejor rendimiento eligiendo el tipo de índice óptimo entre el uso de árboles aleatorios KD, k-means jerárquicos o lineales, como así también, los parámetros para el conjunto de datos proporcionados.
- **SavedIndexParams**: este tipo de objeto es usado para cargar un índice previamente guardado en el disco.

La función **knnSearch** lleva a cabo la búsqueda de los K-vecinos más cercanos para un punto dado, usando el índice. A continuación se presentan los parámetros[8] de dicha función:

```
void Index_<T>::knnSearch(const Mat& queries,
                          Mat& indices,
                          Mat& dists,
                          int knn,
                          const SearchParams& params)
```

Parámetros:

- **query:** matriz con puntos para realizar la consulta, su tamaño es CxD, donde C es la cantidad de características y D es la dimensionalidad (un punto por fila).
- **indices:** matriz que contendrá los índices de los K vecinos más cercanos encontrados.
- **dists:** matriz que contendrá las distancias a los K vecinos más cercanos encontrados.
- **knn:** número de vecinos más cercanos para los que se hará la búsqueda.
- **params:** parámetros de búsqueda

```
struct SearchParams {
        SearchParams(int checks = 32);
};
```

checks: especifica las hojas máximas a visitar en la búsqueda de los vecinos. Un valor alto para este parámetro, dará una mejor precisión, pero requerirá más tiempo. Si se utilizó la configuración automática

nearest_neighbor_search.html#id4

[8] http://opencv.willowgarage.com/documentation/cpp/flann_fast_approximate_nearest_neighbor_search.html#cv-cv-flann-index-t-knnsearch

cuando se creó el índice, el número de controles necesarios para alcanzar la precisión especificada también fue calculado, en cuyo caso se omite este parámetro.

Referencias

[AH08] C. Silpa Anan and R. I. Hartley. Optimised KD-trees for fast image descriptor matching. In *CVPR*, pages 1–8, 2008.

[AMN+98] Arya, Mount, Netanyahu, Silverman, and Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *JACM: Journal of the ACM*, 45, 1998.

[BETV08] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-up Robust Features (SURF). *Computer Vision and Image Understanding: CVIU*, 110(3):346–359, June 2008.

[BL97] Jeffrey S. Beis and David G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR '97)*, CVPR '97, pages 1000–, Washington, DC, USA, 1997. IEEE Computer Society.

[Bri95] S. Brin. Near neighbor search in large metric spaces. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB '95)*, pages 574–584, San Francisco, Ca., USA, September 1995. Morgan Kaufmann Publishers, Inc.

[FBF77] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, September 1977.

[FN75] Keinosuke Fukunaga and Patrenahalli M. Narendra. A branch and bound algorithm for computing k-nearest neighbors. *IEEE Transactions on Computers (TOC)*, C-24(7):750–753, July 1975.

[LMGY04] Ting Liu, Andrew W. Moore, Alexander G. Gray, and Ke Yang. An investigation of practical approximate nearest neighbor algorithms. In *NIPS*, 2004.

[LMS06] B. Leibe, K. Mikolajczyk, and B. Schiele. Efficient clustering and matching for object class recognition. In *BMVC*, page II:789, 2006.

[ML09] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *In VISAPP International Conference on Computer Vision Theory and Applications*, pages 331–340, 2009.

[MM07] K. Mikolajczyk and J. G. Matas. Improving descriptors for fast tree matching by optimal linear projection. In *ICCV*, pages 1–8, 2007.

[NS06] D. Nister and H. Stewenius. Scalable recognition with a vocabulary tree. In *CVPR*, pages II: 2161–2168, 2006.

[Ric04] Richard. Localisation using an image-map. 2004.