Departamento de Ingeniería de Sistemas y Automática

PROYECTO FIN DE CARRERA

ESTABILIZACIÓN DE LA VISIÓN ARTIFICIAL DEL ROBOT HUMANOIDE HOAP-3 DURANTE SU MOVIMIENTO

Autor: Ana Paula Mateo Garrido

Tutores: Paolo Pierro y Miguel González-Fierro

Leganés, Octubre 2010

Título: Estabilización de la visión artificial del robot humanoide HOAP-3 durante su movimiento.

Autor: Ana Paula Mateo

Director:

EL TRIBUNAL

Presidente: Santiago Garrido

Vocal: Rosa María de la Cruz

Secretario: César A. Arismendi

Realizado el acto de defensa y lectura del Proyecto Fin de Carrera el día 5 de Octubre de 2010 en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de

VOCAL

SECRETARIO PRESIDENTE

AGRADECIMIENTOS

En el momento en que empiezo a pensar a cuantas personas he de agradecer donde estoy ahora, me es inevitable pensar en el primer día…

A mi familia, que empezó, continuó y ha terminado conmigo, dándome fuerzas todos, todos los días.

A mis compañeros de clase, por todo lo que hemos pasado y espero que pasemos.

A David López, por la casualidad de haberte sentado el primer día de clase a mi lado y ofrecerme tanto durante todos estos años.

A Alberto, porque la recta final ha sido la más intensa y con alguien al lado es diferente.

A mi tutor, Paolo Pierro, porque desde el primer día me ofreció su apoyo.

Un especial agradecimiento a Miguel González-Fierro y Daniel Hernández, por el tiempo y esfuerzo que me han dedicado y por los días que se nos ha hecho de noche en el laboratorio.

A todos aquellos que no puedo nombrar uno por uno, pero que igualmente me han aportado pequeños detalles a lo largo de estos años.

Gracias.

RESUMEN

La robótica humanoide se prevé como una de las mayores revoluciones tecnológicas de la industria. En la próxima década se espera que puedan formar parte de nuestra vida cotidiana, como por ejemplo en asistencia de menores y ancianos, o realización de tareas domésticas y entretenimiento. Económicamente provocará un gran movimiento mundial, permitiendo su adquisición a precios mucho más asequibles y por lo tanto se producirá una gran difusión en cualquier ámbito, ya sea doméstico o empresarial.

Previo a este desarrollo es necesario un proceso de investigación en las distintas tareas que pueden realizar los humanoides de forma que el uso de estos sea de manera sencilla, cómoda y precisa.

Este proyecto trata sobre la obtención imágenes estables tomadas por el robot cuando se encuentra en movimiento. Esto permite que la imagen no fluctúe, obteniendo una secuencia nítida y preparada para posteriores procesos, como identificación o manipulación de precisión de objetos o simplemente para el desarrollo de tareas implicadas en visión o la telemática.

El software utilizado se basará principalmente en programación en C++ y en el uso de librerías OpenCV, así mismo trata de integrar los conocimientos adquiridos a lo largo de la carrera ya que contiene aspectos matemáticos, físicos, informáticos y de visión. Realizando un programa robusto que permita su utilización de una manera sencilla y en cualquier ámbito.

Palabras clave: Humanoides, impacto social-económico, estabilidad y nitidez visual.

ABSTRACT

The humanoid robot is expected to be one of the greatest technological revolutions in the industry; day by day is getting more common in households and society. In the next decade humanoid robots are expected to be part of our daily lives, such as in child and elderly care or domestic tasks and entertainment. Economically lead a large global movement, allowing them to purchase much more affordable prices and therefore will be a large spread in any field, whether home or business.

Prior to this development requires research process in different tasks that humanoids can perform this way the use of these robots is simple, convenient and accurate. This project consist in obtain stable images taken by the robot when it is in motion. This allows not fluctuating the image, resulting in a clear stream and prepared for subsequent processes, such as identification for precise objects manipulation or simply for the development of tasks involved in vision or telematics.

The software used mainly consists of programming in C++ and use OpenCV libraries. Furthermore, this project seeks to integrate the knowledge acquired throughout the degree because it contains aspects related to mathematics, physics and computer vision.

Keywords: Humanoids, socio-economic impact, stability and visual sharpness.

INDICE

Índice de figuras

Índice de tablas

1.INTRODUCCIÓN Y OBJETIVOS

1.1 INTRODUCCIÓN Y MOTIVACIÓN

En los últimos años se ha dado un fuerte impulso a la investigación en robots humanoides, desarrollándose robots con alta capacidad de manipulación, interacción y movilidad. Algunos ejemplos son el WABIAN-2 de la Universidad de Waseda (Ogura et al., 2006), el ASIMO de Honda (Sakagami et al., 2002), el HRP-2 del National Institute of Advanced Industrial Science and Technology of Japan (Kaneko et al., 2004) o el RH-1 diseñado en la Universidad Carlos III de Madrid (Arbulú, Kaynov, Cabas, y Balaguer, 2009). Pero para que estos robots interaccionen con el ser humano y se introduzcan plenamente en su entorno, es necesario aumentar su seguridad, manipulabilidad, estabilidad y su capacidad de interacción.

Para llegar a este nivel de adaptación de los robots al mundo humano, se requiere una sustancial mejora de los sistemas de visión artificial, tanto desde el punto de vista del hardware como de algoritmos, avanzando en aspectos como localización e identificación de objetos, nuevas tecnologías de video, detección de personas o estabilización de la imagen.

En los seres humanos, la visión, es seguramente el principal sistema de percibir el mundo que nos rodea. Nuestro sistema de visión, evolucionado durante millones de años, permite obtener imágenes nítidas y claras aunque estemos en movimiento. Es en el cerebro donde se procesa la información visual, que automáticamente y sin darnos cuenta, estabiliza la imagen cuando nos movemos.

Para los robots humanoides, debido a las inclinaciones laterales que sufren al andar, la estabilización de la imagen es un elemento fundamental para realizar muchas tareas complejas que impliquen movilidad. Con este sistema se aumenta la capacidad de interacción con el entorno; si el robot se encuentra en un escenario desestructurado, la estabilización de la imagen permite detectar objetos con el robot en movimiento que de otra forma se perderían de vista continuamente. Así mismo, puede mejorar la manipulabilidad, con un sistema de estabilización de la imagen se puede fácilmente agarrar un objeto mientras el robot camina. Permite mejorar la experiencia visual de un observador que esté teleoperando al robot mediante algún HRI. Es, en definitiva, un sistema necesario y beneficioso para robot humanoide, sin el cual está ciego cuando realiza alguna tarea en movimiento.

En este proyecto se presenta un nuevo algoritmo llamado Robust Real Time Stabilization (RRTS), con él se consigue controlar y corregir las perturbaciones producidas en la imagen obtenida por una cámara en movimiento. Para probar el comportamiento del algoritmo, se ha implementado en el robot humanoide HOAP-3 mientras camina. En un momento inicial se ha grabado la imagen de la cámara del robot mientras andaba y se ha pasado el algoritmo a este video en un ordenador. Finalmente, se ha implementado en el robot directamente consiguiendo la establización de la imagen en tiempo real.

1.2. OBJETIVOS

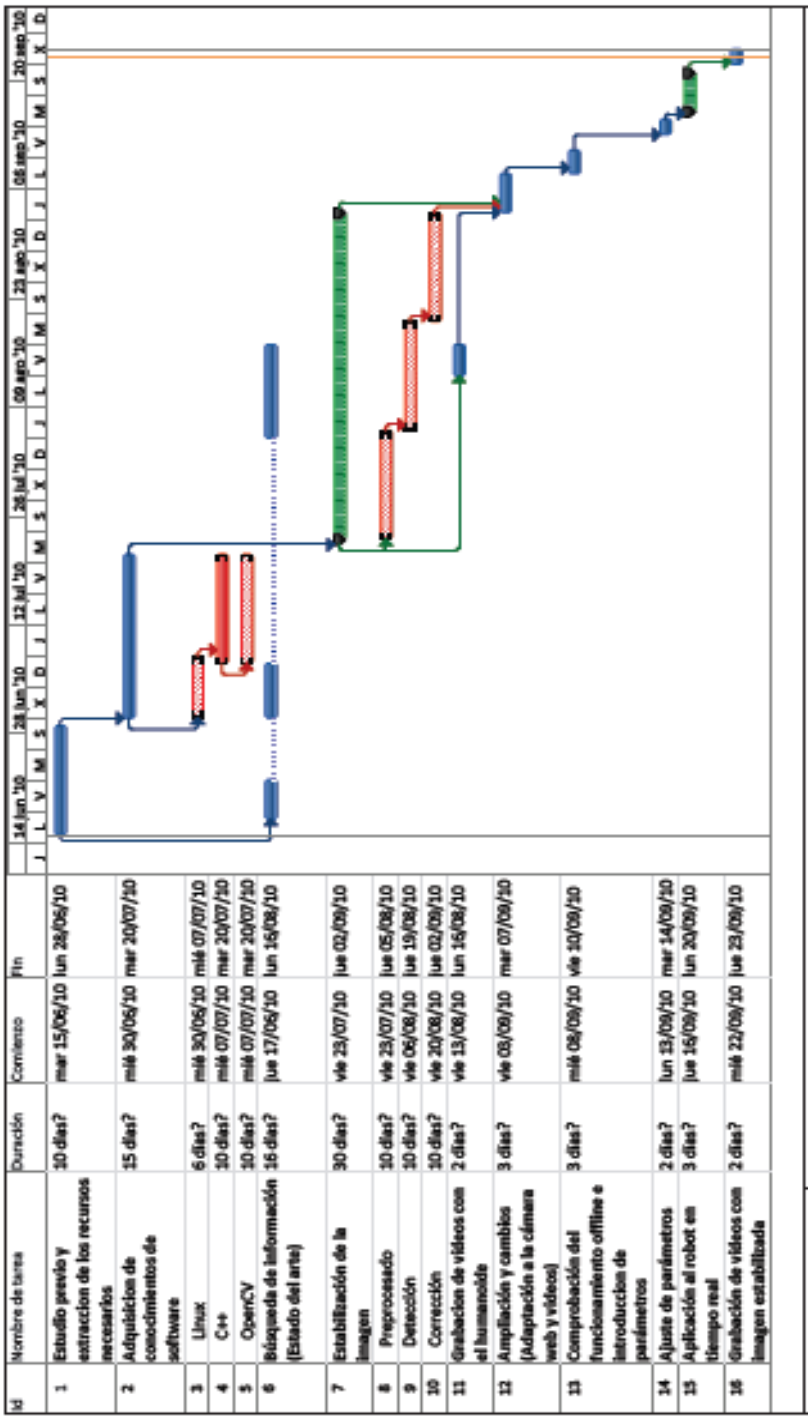
1. El objetivo principal del proyecto es la estabilización de las imágenes que ofrece la cámara estereoscópica del humanoide HOAP – 3 cuando se encuentra en movimiento.
2. Realizar la estabilización de un video de un conjunto de imágenes en movimiento de forma offline, es decir, procesar esa imagen desde un PC con el video previamente grabado.
3. Una vez se ha comprobado que el algoritmo funciona en un video previamente grabado, implementar ese mismo algoritmo directamente en el robot, creando un sistema cliente-servidor para obtener la secuencia de imágenes del robot en un PC y ejecutar el algoritmo de estabilización de forma que se muestre al usuario.

1.3. FASES DE DESARROLLO

Para el desarrollo del proyecto se ha seguido el siguiente proceso:

- Búsqueda de información sobre otras aplicaciones que tienen algún tipo de relación con el proyecto, contrastar y ampliar información sobre el tema de visión.
- Toma de confianza con los recursos existentes y planteamiento de la idea para el óptimo desarrollo del proyecto, teniendo en cuenta las recomendaciones.
- Toma de confianza con las librerías OpenCV.
- Realización del algoritmo usando las librerías de Opencv y en C++.
- Grabación de videos con el humanoide para la implementación offline.
- Ajuste de parámetros y del código con los videos grabados.
- Aplicación en el robot en tiempo real.
- Pruebas correspondientes que demuestren su correcto funcionamiento.

A continuación se muestra gráficamente un diagrama de Gantt con la previa organización del proyecto, en el se muestran las distintas etapas y el tiempo estimado para su realización:



Id	Nombre de tarea	Duración	Comienzo	Fin
1	Estudio previo y extraccion de los recursos necesarios	10 días?	mar 15/06/10	lun 28/06/10
2	Adquisicion de conocimientos de software	15 días?	mié 30/06/10	mar 20/07/10
3	Linux	6 días?	mié 30/06/10	mié 07/07/10
4	C++	10 días?	mié 07/07/10	mar 20/07/10
5	OpenCV	10 días?	mié 07/07/10	mar 20/07/10
6	Búsqueda de información (Estado del arte)	15 días?	jue 17/05/10	lun 16/08/10
7	Estabilización de la imagen	30 días?	vie 23/07/10	jue 02/09/10
8	Preprocesado	10 días?	vie 23/07/10	jue 05/08/10
9	Detección	10 días?	vie 06/08/10	jue 19/08/10
10	Corrección	10 días?	vie 20/08/10	jue 02/09/10
11	Grabacion de videos con el humanoide	2 días?	vie 13/08/10	lun 16/08/10
12	Ampliación y cambios (Adaptación a la cámara web y vídeos)	3 días?	vie 03/09/10	mar 07/09/10
13	Comprobación del funcionamiento offline e introduccion de parámetros	3 días?	mié 08/09/10	vie 10/09/10
14	Ajuste de parámetros	2 días?	lun 13/09/10	mar 14/09/10
15	Aplicación al robot en tiempo real	3 días?	jue 16/09/10	lun 20/09/10
16	Grabación de videos con imagen estabilizada	2 días?	mié 22/09/10	jue 23/09/10

1.4 ESTRUCTURA DE LA MEMORIA

Para facilitar la lectura de la memoria, se incluye a continuación un breve resumen de cada capítulo:

El capítulo 1 consta de una representación abreviada, objetiva y precisa del contenido del proyecto.

En el capítulo 2 se realiza la descripción general del proyecto, objetivos que se persiguen, y una planificación previa para su desarrollo.

En el capítulo 3 se describe el estado de la técnica en los campos que afectan al proyecto. Comienza con una breve introducción sobre las investigaciones desarrolladas hasta el momento con respecto a la estabilización de imágenes y a continuación se hace un análisis sobre las librerías utilizadas así como sus posibles aplicaciones en otros campos.

El capítulo 4 se centra en las plataformas de referencia, proporcionando una descripción acerca del robot, webcam y hardware utilizados y haciendo un breve análisis sobre sus posibles limitaciones.

El capítulo 5 versa sobre las técnicas y algoritmos empleados en visión para el proyecto, prestando especial interés en aquellas dedicadas a la estabilización de imágenes.

El capítulo 6 muestra los resultados obtenidos a lo largo de la realización del proyecto, desde la grabación de videos con la webcam hasta los resultados obtenidos en la estabilización en tiempo real con el robot.

En el capítulo 7 se explican las conclusiones extraídas del desarrollo de la aplicación y los caminos a seguir en la mejora y desarrollo futuros.

En el capítulo 8 se hace referencia a la bibliografía utilizada en los diferentes medios empleados.

Por último el capítulo 9 contiene documentación a la que se ha hecho referencia en el proyecto o que sea necesaria consultar durante su lectura.

2. ESTADO DEL ARTE

El ámbito de la estabilización de imágenes por software está en constante desarrollo, es muy fácil encontrar información en cualquier medio de comunicación, ya sea en artículos de prensa, aplicaciones en cámaras de fotos y video, medicina, etc., es decir, en todos los campos que sea necesario el uso de la visión artificial en el campo audiovisual.

El uso más frecuente se encuentra en la estabilización de imágenes tomadas con cámaras de video. Existen numerosos programas para realizar la estabilización, en algunos casos la estabilización se realiza al mismo tiempo que se crea el video, mientras que en otras ocasiones se realiza posteriormente con un programa específico.

La estabilización de imágenes en tiempo real aplicadas a cámaras de video pueden ser de tres formas: estabilización de imagen óptico (Cardani, 2006), estabilizador de imagen mecánico con sensor CCD (Oshima et al., 1989), (Kinugasa et al., 1990) y estabilizador de imagen digital (Xu & Lin (2006)), (Pan & Ngo (2005)).

El primer caso se trata de un sistema mecánico que consta de dos superficies ópticas paralelas formando un prisma flexible, cuando se produce un movimiento en la cámara, se detecta electrónicamente y mediante una variación de tensión se mueven las lentes. La luz que incide en el prisma varía y se envía la imagen al sensor en la dirección opuesta al movimiento realizado por la cámara. En este tipo de estabilización no se pierde calidad en la imagen puesto que esta es compensada antes de ser procesada.

El segundo caso es similar al anterior, la única diferencia es que es el sensor lo que se mueve y no las lentes, si lo comparamos con el anterior se trata de un modelo menos efectivo, ya que en situaciones con poca luz no realiza la estabilización.

El tercer caso es un sistema electrónico que actúa sobre la imagen obtenida en el sensor. Cuando la cámara se mueva a un lado el encuadre digital se moverá en la dirección contraria, de esta manera se cancela el efecto del movimiento del sensor. El inconveniente es que la imagen que se obtiene es ligeramente menor que la original afectando a la resolución y a la claridad de la imagen, pero tiene una gran ventaja, y es que no necesita ningún aparato electrónico o mecánico ya que compensa el movimiento de la imagen mediante algoritmos de visión. El esquema típico de esta técnica de estabilización se puede observar en la Figura 1.

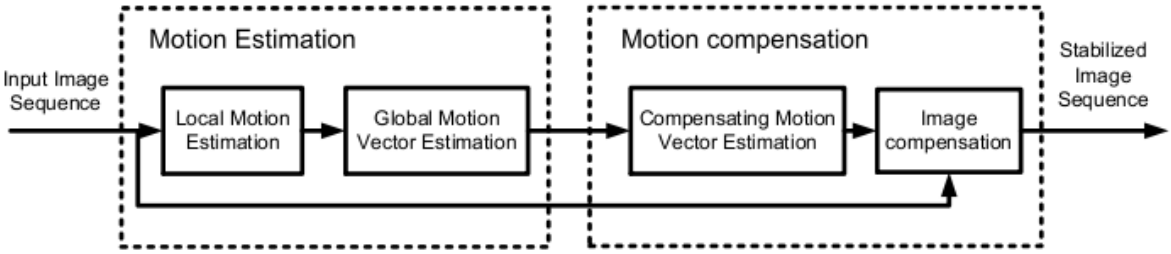


Figura 1: Esquema de estabilización de imagen digital

Estos sistemas de estabilización son adoptados hoy en día por algunos de los grandes fabricantes mundiales: Canon y Nikon usando el sistema de estabilizador óptico, sin embargo el estabilizador de imagen mecánico es utilizado por Sony, Casio y Fuji entre otros.

Uno de los últimos desarrollos lo produce canon el 22 de Julio de 2009 creando un sistema de estabilización de imagen híbrido, el cual es capaz de compensar ópticamente tanto el movimiento angular como el de desplazamiento. (Figura 2).

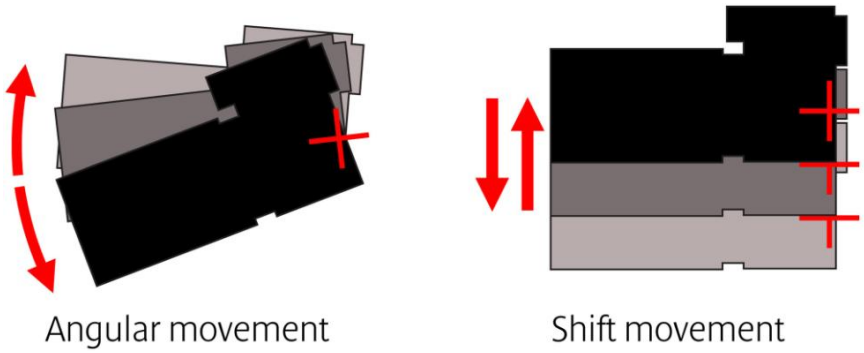


Figura 2: Nuevo sistema de estabilización de imagen híbrido de Canon

La estabilización ofrecida por Canon incorpora un sensor de velocidad angular que es capaz de detectar el ángulo de movimiento, así como un sensor de aceleración que determina el desplazamiento de la imagen basándose en la intensidad del movimiento de la cámara. También utiliza un nuevo algoritmo que combina la información de los dos sensores y desplaza los elementos del objetivo para compensar ambos tipos de movimiento.

Este sistema presenta grandes resultados, especialmente para tomas macro, algo que resulta muy difícil para las tecnologías de estabilización de la imagen habituales.

También cabe destacar la nueva técnica desarrollada por investigadores de la universidad de Wisconsin-Madison Content-Preserving Warps for 3D Video Stabilization (Liu et al. 2009).

El software funciona detectando el entorno del vídeo, calculando las distancias tridimensionalmente y aplicando un reajuste completo de las secuencias, fotograma por fotograma, cuadrando el objeto principal que se quiera destacar en el centro de la pantalla..

Figura 3: Resultados del algoritmo de estabilización Content-Preserving Warps for 3D Video Stabilization

Otro de los usos más extendidos es en la medicina, donde el cirujano mediante la simulación programa al robot para que lo ejecute de manera perfecta sobre el paciente. Con estos sistemas es posible realizar operaciones de microcirugía con suave precisión, siendo necesaria la estabilización de la imagen para que la operación pueda llevarse a cabo con exactitud. Un ejemplo de esta aplicación es el AESOP 3000 (Figura 4) , robot que conduce la cámara en cirugía laparoscópica, otro de ellos es el ARTEMIS de origen alemán que trabaja con visión en tres dimensiones, este entiende las órdenes verbales del cirujano y las ejecuta con movimientos exactos, también está dotado de instrumentos poliarticulados que le permite alcanzar movimientos que nunca se han logrado en cirugía laparoscópica y por último el ZEUS (Computer Motion), un robot que permite al cirujano operar a distancia al paciente, este consta de una consola con dos controles y un monitor que suministra la imagen en dos dimensiones.

Figura 4: AESOP 3000

Respecto a la utilización de sistemas de estabilización de la imagen aplicados a la robótica, existen varios casos en la literatura. En (Amanatiadis et al, 2010) un sistema de estabilización tanto electrónico como digital, diseñado para implementarlo en un rover, en (Hayakashi, 2005) se presenta un sistema de control de la imagen para interfaces en robots de rescate, en (Choi et al., 2010) se presenta un sistema de detección de perturbaciones mediante inclinómetros que posteriormente se trata con el filtro de Kalman Extendido.

El autor de este PFC no ha encontrado ningún artículo o algoritmo relacionado con la estabilización de la imagen en robots humanoides.

3.HERRAMIENTAS Y PLATAFORMA UTILIZADA

3.1. LIBRERIAS OPENCV

La librería OpenCV (Open source computer vision library) desarrollada en 1999 por Intel es una librería de uso público, con una biblioteca escrita en C optimizado (C++), es compatible con IPL (Intel Processin Library) y utiliza IPP (Intel Integrated Primitives) por lo que puede ser utilizada con procesadores Intel sin que ellos excluya su uso en otros procesadores. Esta funciona bajo Linux, Windows y Mac OS X.

Uno de los objetivos de esta librería es que pueda usarse en ordenadores sin grandes prestaciones y que a la vez puedan realizarse aplicaciones de visión sofisticadas y a buena velocidad.

Contiene más de 500 funciones de alto nivel (Figura 5) para el procesado de imágenes que abarcan áreas en visión, incluyendo imágenes médicas, seguridad, interfaces de usuario, calibración de cámaras, robótica, etc.

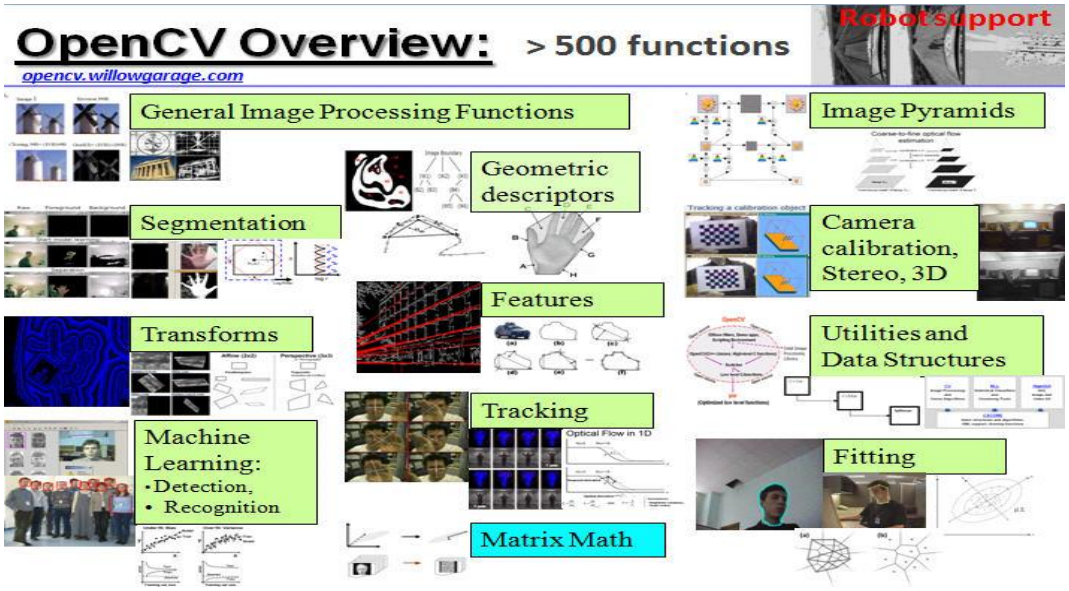


Figura 5: Funciones OpenCV

Desde que se desarrolló el número de aplicaciones se va incrementando diariamente, algunas de estas aplicaciones pueden ser:

- Interacción entre computadores y personas (HCI)
- Identificación de objetos, segmentación y reconocimiento
- Reconocimiento de gestos faciales
- Estructuras de movimiento (SFM)
- Aplicaciones en robots móviles
- Control de procesos
- Reconstrucción 3D y calibración de la cámara
- Estabilización de imágenes
- Etcétera.

Si especificamos con aplicaciones realizadas recientemente podemos encontrar:

- OpenCV ha sido usada en el sistema de visión del vehículo no tripulado Stanley de la Universidad de Stanford, el ganador en el año 2005 del Gran desafío DARPA.
- OpenCV se usa en sistemas de vigilancia de vídeo
- OpenCV es la clave en el programa *Swistrack*, una herramienta de seguimiento distribuida
- AVSR - Audio Visual Speech Recognition, software que permite a los ordenadores detector la cara de un orador y seguir los movimientos de su boca. Este software forma parte de la librería OpenCV de Intel.
- Existen numerosas aplicaciones ya sea con robots, o proyectos destinados a personas con discapacidad, estas aplicaciones no están realizadas por empresas multinacionales, sino que están realizadas por estudiantes que realizan su proyecto fin de carrera o desarrollos de investigación personales no patentados, los cuales pueden ser encontrados fácilmente en internet.

3.1.1 Requerimientos de software y hardware

Las librerías OpenCv funcionan bajo Linux, Windows y Mac, están pensadas para integrarse en programas escritos en C o C++ y es necesario tener un procesador Pentium o compatible. Si se trabaja en Windows será necesario el uso de Microsoft Visual C++, DirectShow o graphedt.exe, entre otros.

Este Proyecto de Fin de Carrera ha sido planteado en Ubuntu, por lo que tanto los algoritmos programados como el sistema de comunicación con el robot han sido implementados en Linux para ello ha sido necesario instalar las librerías GTK 2.0.

También son necesarios lo paquetes pkgconfig, libpng, zlib, libjpeg, libtiff, libjasper, Python 2.3, 2.4 o 2.5 con sus cabeceras correspondientes instaladas, libavcodec.

Después de la instalación se haya completado, la ruta de instalación por defecto es /usr/local/lib/ y /usr/local/include/OpenCV/. Por lo tanto es necesario agregar /usr/local/lib/ a /etc/ld.so.conf (y ejecutar después ldconfig) o añadirlo a la variable de entorno LD_LIBRARY_PATH en bashrc.

ExportLD_LIBRARY_PATH=$LD_LIBRARY_PATH:
/usr/local/lib/:/usr/local/include/OpenCV/

Para más información consultar (Learning OpenCV de *Gary Bradski and Adrian Kaehler, 2008*).

3.1.2 Visión general de la librería

La librería está dividida en áreas temáticas clásicas de la visión por computador. En cada una de ellas disponemos de una breve descripción teórica de las técnicas que se utilizan para implementar las funciones. Aquí solo haremos un apunte de los temas tratados en cada área.

Estructuras y funciones básicas

- Primero tenemos una serie de funciones para crear imágenes, reservar y liberar memoria, cambiar de parámetros, etc.
- Estructuras de daros dinámicas. Las matrices son alojadas en memoria dinámica.
- Operaciones con matrices, que pueden ser sencillas o cálculo de autovalores, transformaciones, etc.
- Realización de dibujos sencillos o introducción de textos.
- Operaciones auxiliares. También tenemos funciones para realizar operaciones como sumas, restas, raíces cuadradas, etc.

Análisis del movimiento

- Modelos de movimiento usados para describir las variaciones entre imágenes consecutivas en el tiempo, clasificarlas y calcular los diferentes parámetros.
- Calibración de la cámara.
- Flujo óptico (optical Flow) se usa un mñetodo para dividir la imagen en bloques para después buscarlos en imágenes sucesivas.

Análisis de imágenes

- Extracción de características. Filtros de cajas (Box filters), y otros métodos para encontrar bordes y esquinas.
- Estadísticos. Si consideramos el nivel de gris de cada pixel de una imagen como una observación de una variable estocástica podemos calcular parámetros como la media, desviación típica, etc.
- Pirámides. Disponemos de funciones para generar y reconstruir pirámides Gaussianas y Laplacianas. También tenemos funciones que se basan en pirámides para segmentar imágenes o para calcular el flujo óptico.
- Cambios morfolçogicos. Se incluyen erosiones, dilataciones, etc. para filtrar ruido o separar o unir regiones.

- Crecimiento de regiones. Partiendo de unos puntos iniciales se etiquetan progresivamente todos los puntos adyacentes que tengan unas características cercanas a los puntos iniciales.

Análisis estructural

- Tratamiento de contornos. Tenemos diferentes métodos para detectar contornos existiendo tipos de datos específicos para facilitar las operaciones.

Reconstrucción en 3D

- Calibración de cámaras. Las funciones para calibrar la cámara determinan los parámetros intrínsecos como la distancia focal, el coeficiente de distorsión radial, etc. Y los parámetros extrínsecos que relacionan la cámara con el entorno.

3.1.3 Tipos de datos

La librería OpenCV dispone de datos específicos para desenvolverse con facilidad en cada área de la visión por computador. Además de las imágenes con formato propio (IplImage *) los tipos de datos más importantes son estructuras de memoria dinámica.

- Almacenes de memoria (CVMemStorage)

Estos almacenes proporcionan los bloques de memoria necesarios para guardar las demás estructuras. Están compuestos por una cabecera seguida de una lista de direcciones y cantidades de memoria disponible. Los bloques de memoria apuntados son todos del mismo tamaño. En la cabecera tenemos un puntero al último bloque de memoria con espacio disponible. Según se van llenando los bloques el puntero cambia al siguiente. (Figura 6)

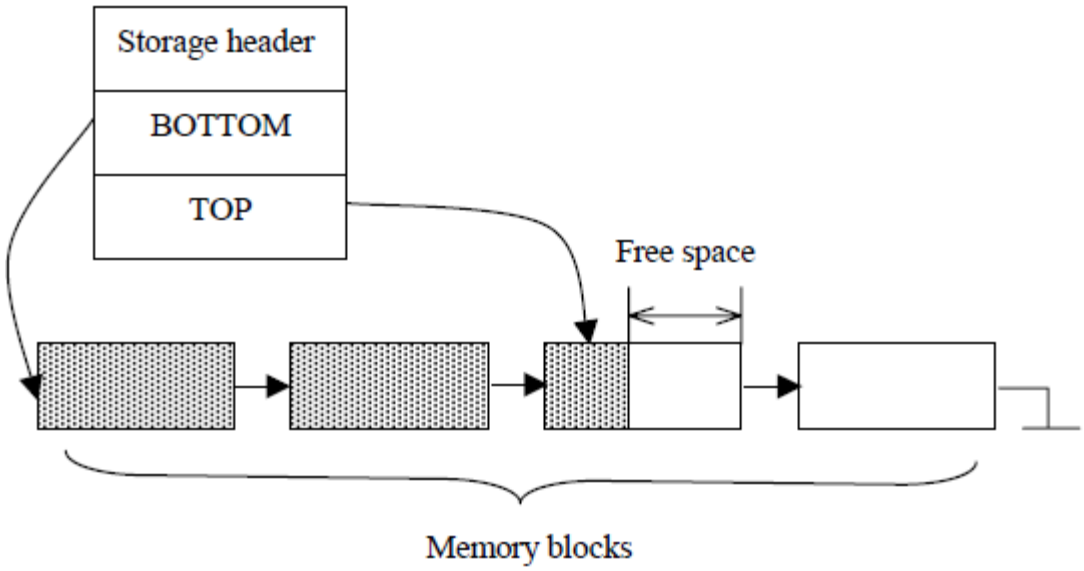


Figura 6: CvMemStorage

- Secuencias (CvSeq)

Una secuencia es un array de tamaño variable de diferentes tipos de elementos almacenado en un almacén de memoria. La secuencia es dividida en partes y cada una se almacena en un bloque de memoria. Cada uno, apunta al siguiente y al anterior. Podemos tener una secuencia muy larga en varios bloques de memoria o varias secuencias cortas en un solo bloque. Existen funciones para manejar con rapidez los elementos del principio y final de la secuencia, sin embargo, las funciones para operar elementos intermedios son bastante lentas.

Las secuencias son la base de otros tipos de estructuras como los contornos o los grafos. En la Figura 7 observamos la organización del almacén de memoria para una secuencia.

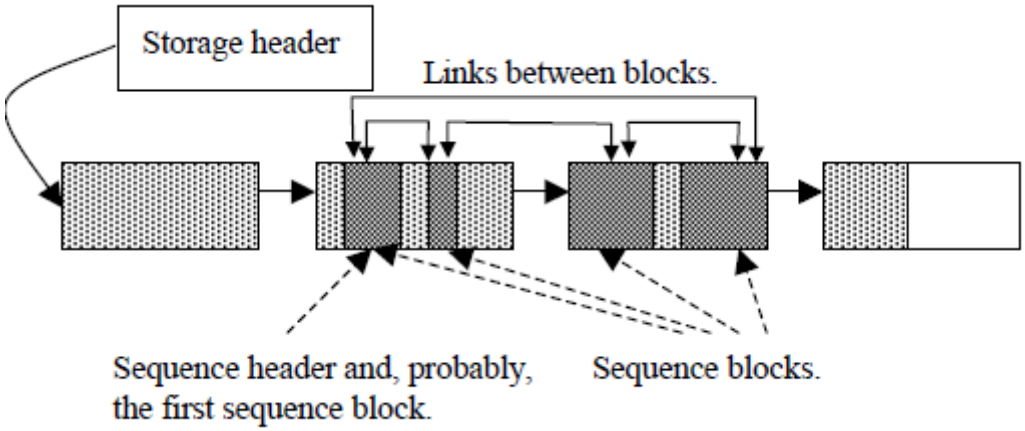


Figura 7: CvSeq

- Set (CvSet)

Estas estructuras están basadas en secuencias, aunque su uso es bastante diferente. El problema de las secuencias se encuentra en el manejo de los elementos del medio. Sise quiere borrar un elemento se debe tener cuidado de no romper la cadena de punteros. Además este tipo de operaciones son bastante lentas. Las estructuras set parten de un secuencia en la que en realidad los elementos no siguen un orden, e incluso se pueden tener espacios de memoria a mitad de un bloque. Lo que tenemos es un conjunto de celdas de memoria en las cuales se pueden alojar los elementos que se quieran. Se pueden eliminar o insertar elementos en cualquier hueco. En la cabecera de la estructura se tiene una lista de celdas vacías, admás se pueden añadir una lista de elementos y punteros para saber lo que se tiene guardado y donde. En la Figura 8 vemos como se estructura la secuencia de celdas sin orden físico, aunque sí lógico.

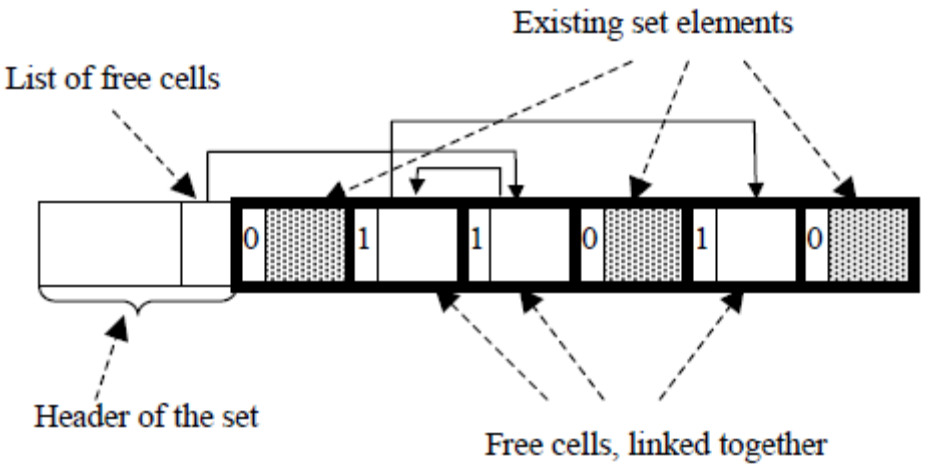


Figura 8: CvSet

- Matrices (CvMat y CvMatArray)

Estas estructuras están pensadas para guardar matrices de números reales de precisión simple o doble. Hay funciones que permiten hacer operaciones aritméticas sencillas y otras mñas complejas que pueden calcular los autovalores o la descomposicón en valores singulares. También existe una estructura específica para facilitar el manejo de arrays de matrices.

- Imágenes (IplImage)

La librería OpenCV comparte formato de imagen con la IPL de Intel. Las imágenes están compuestas por una cabecera y una zona de datos. La definición de la estructura es la siguiente:

```
typedef struct _IplImage {

    int nSize;

    int ID;

    int nChannels;

    int alphaChannel;

    int depth;

    char colorModel[4];

    char channelSeq[4];
```

```
    int dataOrder;
    int origin;
    int align;
    int width;
    int height;
    struct _IplROI* roi;
    struct _IplImage* maskROI;
    void* imageId;
    struct _IplTileInfo* tileInfo;
    int imageSize;
    char* imageData;
    int widthStep;
    int BorderMode[4];
    int BorderConst[4];
    char* imageDataOrigin;
} IplImage;
```

Sólo se van a describer los campos más importantes:

- Width y heigh. Ancho y alto de la imagen.
- Depth. Indica el tipo de valor que puede tomar cada pixel), las posibilidades pueden ser:

  IPL_DEPTH_8U Unsigned 8-bit integer (8u)

IPL_DEPTH_8S Signed 8-bit integer (8s)

IPL_DEPTH_16S Signed 16-bit integer (16s)

IPL_DEPTH_32S Signed 32-bit integer (32s)

IPL_DEPTH_32F 32-bit fl oating-point single-precision (32f)

IPL_DEPTH_64F 64-bit fl oating-point double-precision (64f)

- NChanels. Numero de planos que tiene la imagen, 1 para imágenes monocromo y 3 para imágenes a color.
- Origin (Origen de los ejes de coordenadas de la imagen)
- DataOrder. En las imágenes a color nos dice si los datos de cada plano de la imagen están separados o entrelazados.
- WidthStep. Contiene el número de bytes que separan dos puntos de la misma columna situados en filas sucesivas.
- ImageData (Puntero a la primera final de la imagen.
- Roi. Esta estructura nos permite operar solo sobre una región de la imagen (región de interés) sobre un solo plano o ambas cosas a la vez. La estructura Roi está compuesta por cinco enteros que indican los planos, además del origen y tamaño de la región de interés.

- Otro tipo de datos

  CvPoint. Define las coordenadas de un punto. Campos:

  int x – coordenada x

  int y – coordenada y

  CvRect. Define un rectángulo. Campos:

  int x – coordenada x del vértice superior izquierdo

  int y – coordenada y del vértice superior izquierdo

  int weidth – ancho del rectángulo

  int height – alto del rectángulo

  CvSize. Estructura para definir el tamaño de una imagen. Campos:

int weidth – ancho de la imagen

int height – alto de la imagen

CvFont. Estructura para caracterizar las fuentes de texto. Campos:

const int* data – datos y medidas de la fuente

CvSize – factores de escala vertical y horizontal

int italic_scale – inclinación para cursivas

int thickness – grosor de las letras

int dx – separación entre letras

3.1.4 Funciones

En este apartado se definirán algunas de las funciones utilizadas:

cvCreateImage: Inicializa una imagen, crea la cabecera y reserva memoria. Devuelve un puntero a la cabecera de la imagen creada.

IplImage* cvCreateImage(CvSize size, int depth, int channels);

size – tamaño de la imagen

depth - tipo de datos para los pixeles

channels – número de canals

cvReleaseImage: libera la cabecera y memoria de una imagen.

void cvReleaseImage(IplImage** image)

image – puntero con doble indirección a la cabecera de la imagen.

cvSetImageRoi: establece la región de interñes de una imagen.

void cvSetImageCOI(IplImage* image, int coi);

image – puntero a la imagen

rect – rectángulo que determina la región de interés

cvCopyImage: copia una imagen en otra sin considerar las ROIs.

void cvCopyImage(IplImage* src, IplImage* dst);

src – puntero a la imagen de referencia

dst – puntero a la imagen de destino

cvPutText: dibuja una cadena de texto

void cvPutText(IplImage* img, const char* text, CvPoint org, CvFont* font, int color)

img – puntero a la imagen

text – texto a dibujar

org – coordenadas de la esquina inferior izquierda donde introducimos el texto

font – puntero a la estructura que caracteriza la fuente

color – color de las letras (nivel de gris)

cvNamedWindow: crea una ventana para visualizar imágenes

cvShowImage: dibuja una imagen en una ventana

cvDestroyWindow: destruye una ventana

3.1.5 Conclusiones

La gran ventaja de esta librería es la gran cantidad de áreas que abarca, sin embargo, hay muchas de ellas que están tratadas muy superficialmente. A pesar de todo, en las áreas menos desarrolladas sí están sentadas las bases para poder realizar implementaciones complejas.

Otro punto a favor es que está desarrollada tanto para Linux como para Windows, aunque el código no es portable directamente, ya que las librerías no son exactamente iguales.

Sin embargo, el punto débil de la librería es la falta de documentación proporcionada, varias veces se ha tenido que recurrir al código fuente para encontrar información adicional.

3.2. EL ROBOT HOAP-3

La plataforma usada para la implementación del algoritmo RRTS es el robot humanoide HOAP-3 (Figura 9). Se trata de un humanoide de tamaño y peso medio, 60 cm y 9 kg aproximadamente, diseñado y fabricado por Fujitsu.

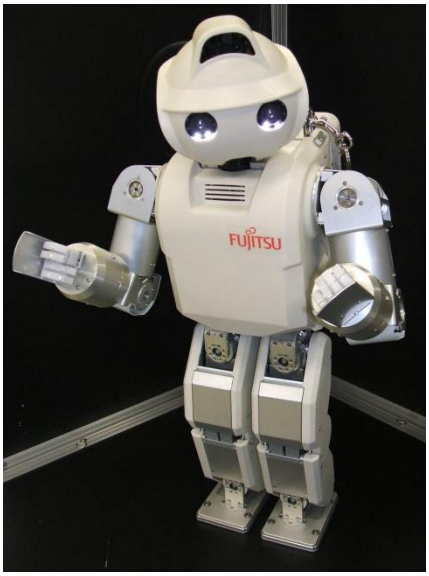


Figura 9: Robot humanoide HOAP - 3

El robot HOAP-3 posee 28 grados de libertad lo que le dota de gran capacidad de movimiento. Posee 6 grados de libertad en cada pierna, 6 en cada brazo, 3 en la cabeza y 1 en la cadera. Estos motores disponen de encoders relativos y existe la posibilidad de controlarlos tanto en posición como en velocidad.

El robot lleva incorporado un PC-104 embebido en su interior, exactamente en la parte de atrás. Se trata de un Pentium de 1.1 Ghz con 512 Mb de RAM y una memoria Compact Flash de 1 Gb. Posee conexión Wifi IEEE802.11g y 4 puertos USB.

El sistema operativo que incorpora el robot es un Linux en tiempo real, cuya base es un Fedora Core 1 con el kernel 2.4. Este sistema operativo en tiempo real permite un mayor dominio de la plataforma.

Con el robot, el fabricante proporciona un ordenador que es un clon exacto del que lleva incorporado el HOAP-3 con mismo hardware y software instalado. Para controlar al robot existen dos métodos de conexión, el primero es a través de una conexión inalámbrica, gracias a

un router que también proporciona el fabricante, la segunda opción es una conexión directa por medio de un cable usb que conecta el PC externo directamente con los motores y los drivers del robot.

Además, el robot se puede alimentar directamente por cable o mediante una batería de NiMH de 24V, esto último ofrece la ventaja de una gran versatilidad en el movimiento.

Para completar las funcionalidades de este humanoide se han añadido un conjunto de sensores que le permiten desenvolverse con naturalidad en cualquier entorno. Como se puede observar en la Figura 10, el robot dispone de dos cámaras para visión estereoscópica, un micrófono, un altavoz, sensores infrarrojos de distancia, y sensores de fuerza en pies y manos.

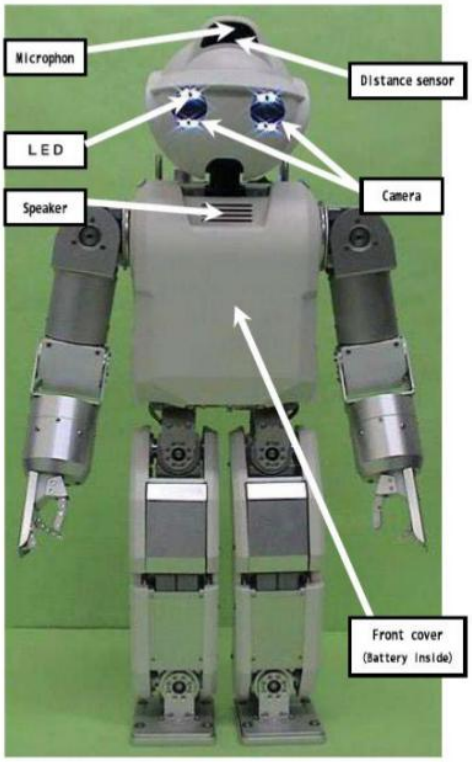


Figura 10: Sensores disponibles en el HOAP-3

3.3 WEBCAM Y HARDWARE DE VISIÓN

Además del humanoide HOAP – 3, para la realización del proyecto ha sido necesario el uso de distintos elementos hardware. A continuación, se detallarán cada uno de ellos:

- WebCam: Cámara Web con micrófono digital integrado, del portátil Serie HP Mini 110-1000 PC.
- Ordenador HP Mini 110: Con procesador Intel Atom N270 de 1,60 GHz, Cache de nivel 2, 512 KB, y tarjeta gráfica Intel Graphics Media Accelerator 950.

3.4 LIMITTACIONES DE HARDWARE

Tras la realización de las pruebas experimentales se han podido detectar las siguientes limitaciones de hardware:

- La mayor limitación se produce con la tarjeta gráfica, ya que la velocidad al reproducir el video mientras se implementa el algoritmo es bastante limitada.
- En tiempo real, la velocidad mejora, ya que no tiene almacenado ningún video.
- Con respecto a la webcam, no ha habido ningún problema significativo, aunque cabe destacar que al inicio de la implementación se bajó la resolución de la imagen para ganar velocidad de procesado.
- Con el humanoide la única limitación que se puede producir es la dificultad de uso para realizar cualquier prueba o simplemente grabación de videos, ya que se tiene que ajustar a las distintas especificaciones requeridas.

4.ESTABILIZACIÓN DE LA IMAGEN

4.1 BASES TEÓRICAS

Para estabilizar imágenes tomadas por el humanoide es necesaria la estabilización software. Existen algunos métodos, que independientemente del ámbito en el que se utilicen son adaptables a robots en movimiento, para detectar el tipo de estabilización más adecuado es necesario un estudio previo de los diferentes modelos de movimiento que se pueden producir, también es necesario conocer qué translaciones pueden darse en una imagen, pues los parámetros que se obtengan, serán los necesarios para poder realizar la estabilización. En la mayoría de los casos se mantiene una estructura similar en la estabilización que consta de dos partes fundamentales: estimación y compensación del movimiento.

4.1.1 Modelos de movimiento

Para describir un movimiento, es necesario basarse en técnicas de estimación, todas ellas basadas en un modelo.

Hay tres tipos de modelos de movimiento que se dan en este proyecto que son, traslación (posición), rotación (orientación) y zoom (escalado).

- Traslación

La traslación (Figura 11) se define como un movimiento directo sin cambios de orientación manteniendo la forma y el tamaño del objeto trasladado, estos se deslizan mediante un vector determinado.

Siendo P y Q dos puntos cualesquiera del plano, se cumple que (1):

$$d(P,Q) = d(T(P),T(Q)) = d(P',Q') \qquad (1)$$

Entonces (2):

$$\overrightarrow{PQ} = \overrightarrow{P'Q'} \qquad (2)$$

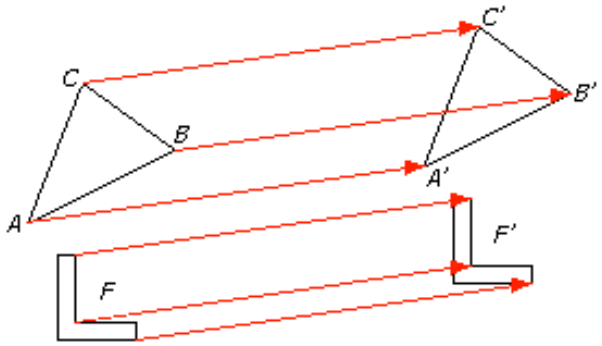


Figura 11: Traslación

En este modelo solo hay que tener en cuenta dos parámetros, considerándose únicamente las variaciones en la dirección vertical y horizontal, por lo tanto solo se hablará de movimiento en x e y.

- Rotación

Es el movimiento que se produce cuando un objeto gira alrededor de un eje que se sitúa dentro del cuerpo de este. Se trata de un movimiento directo que no afecta a la forma ni al tamaño de las figuras (Figura 12).

Para definir este modelo son necesarios dos parámetros, el ángulo y el punto del eje de giro.

Si se produce una rotación en el objeto con centro *O* y ángulo *á*, es una transformación geométrica que hace corresponder a cada punto *P* otro punto *Pð* tal que (3):

$$\overline{OP} = \overline{OP'} \text{ y } \widehat{POP'} = \alpha \qquad (3)$$

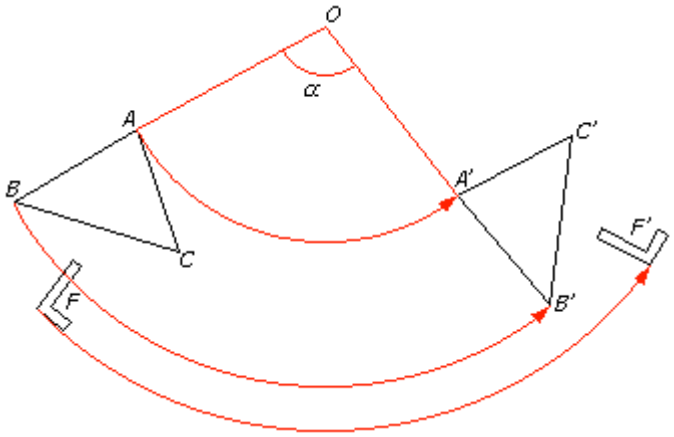


Figura 12: Rotación

- Zoom

Escalado o zoom (Figura 13) es la variación que sufre la imagen al agrandarse o reducirse, reduciendo su tamaño físico y por tanto cambiando la cantidad de píxeles que contiene. En el escalado, cambia el tamaño de la imagen pero no la forma.

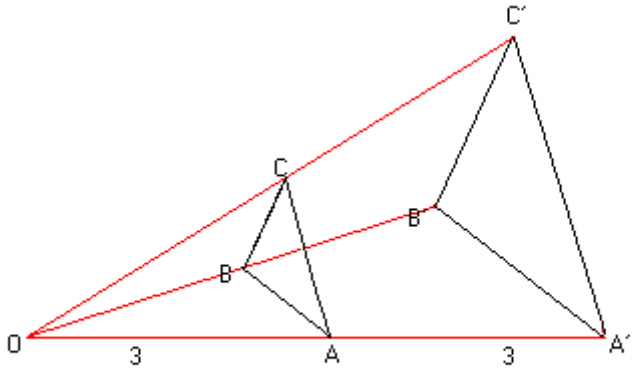


Figura 13: Zoom o Escalado

4.1.2 Transformaciones

En el apartado anterior, se ha dado información básica sobre los distintos tipos de movimiento que puede sufrir un objeto. En este apartado se verá con más profundidad las variaciones que puede haber en una imagen

Se puede definir transformación como el cambio de forma o posición que sufre un objeto, representando o moviendo cada punto del objeto original a una posición diferente mediante un procedimiento especificado.

Las principales transformaciones geométricas pueden ser: transformación afín, bilineal y perspectiva, entre otras. En este apartado, solo se explicarán estos dos tipos de transformaciones, ya que son las que pueden darse en nuestro caso.

- Transformación afín

Se puede definir una transformación afín como traslación, rotación, reflexión, escalado o combinación de las mismas (Figura 14).

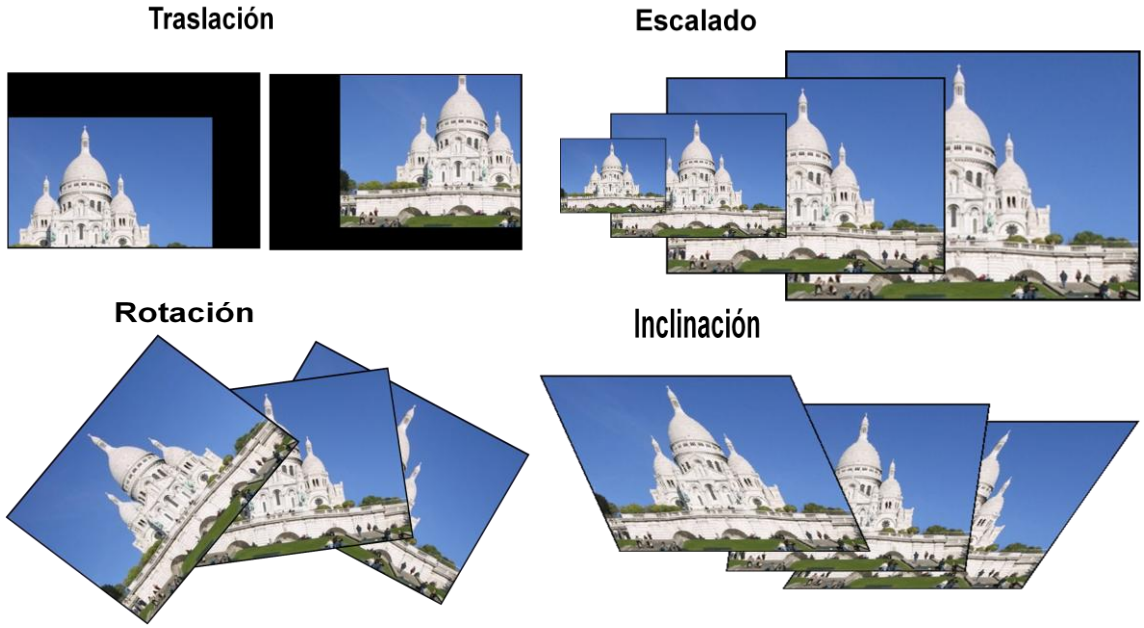


Figura 14: Transformaciones afines

Todas estas transformaciones se pueden expresar de la siguiente forma matricial, que de forma genérica corresponde a (4):

$$R(x,y) = A(\begin{pmatrix} C11 & C12 & C13 \\ C21 & C22 & C23 \end{pmatrix}\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}) \quad (4)$$

Los parámetros ((c_{11},c_{12},c_{13}),(c_{21},c_{22},c_{23}))corresponden a los parámetros de transformación. Estos pueden implicar traslaciones, rotaciones, etc.

Para cada caso obtendremos una matriz A diferente siendo los valores C_{11}, C_{12}, etc. Diferentes según se produzca una traslación **(5)**, rotación **(6)**, escalado **(7)** o inclinación **(8)**.

$$R(x,y) = A(\begin{pmatrix} 1 & 0 & dx \\ 0 & 1 & dy \end{pmatrix}\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}) \quad \textbf{(5)}$$

$$R(x,y) = A(\begin{pmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \end{pmatrix}\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}) \quad \textbf{(6)}$$

$$R(x,y) = A(\begin{pmatrix} ex & 0 & 0 \\ 0 & ey & 0 \end{pmatrix}\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}) \quad \textbf{(7)}$$

$$R(x,y) = A(\begin{pmatrix} 1 & -ix & 0 \\ -iy & 1 & 0 \end{pmatrix}\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}) \quad \textbf{(8)}$$

Es posible tener una transformación afín con varios tipos de movimientos a la vez (9), para realizar la transformación equivalente a dos transformaciones afines, se realiza el producto entre matrices y posteriormente se añade una fila de valor (0, 0, 1).

$$\begin{pmatrix} C11 & C12 & C13 \\ C21 & C22 & C23 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} d11 & d12 & d13 \\ d21 & d22 & d23 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} r11 & r12 & r13 \\ r21 & r22 & r23 \\ 0 & 0 & 1 \end{pmatrix} \quad (9)$$

- Transformación bilineal y perspectiva

Una transformación bilineal o perspectiva se puede entender como una generalización de una transformación afín. En el caso de una transformación afín la imagen original y girada mantenían la misma forma, un rombo. En el caso de la transformación bilineal o perspectiva, ya no se trata de un rombo, sino que cualquier cuadrilátero convexo se transforma en otro cuadrilátero convexo. Véase Figura 15.

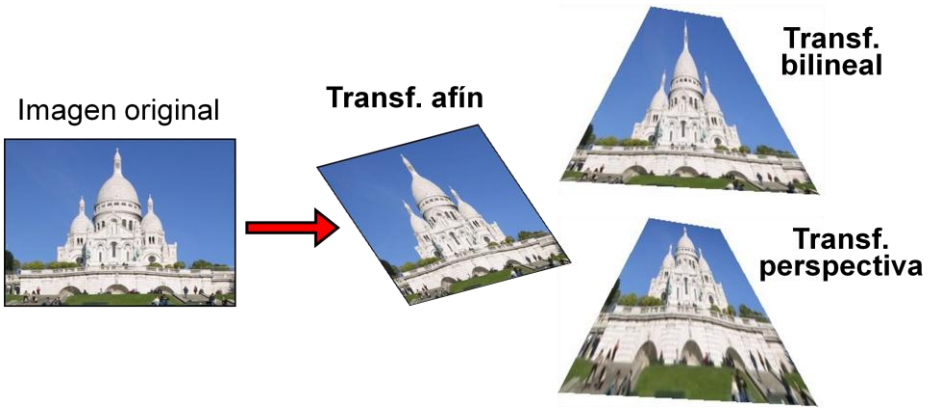


Figura 15: Comparación de transformación afín con transformación bilineal o perspectiva

Como en las transformaciones afines, se puede definir una transformación perspectiva de forma matricial (10), (11):

$$R(x,y) = A\left(\frac{x'}{z'}, \frac{y'}{z'}\right) \qquad (10)$$

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} C11 & C12 & C13 \\ C21 & C22 & C23 \\ C31 & C32 & C33 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \qquad (11)$$

Sin embargo, cuando tenemos una transformación bilineal, la matriz de transformación (12) es:

$$R(x,y) = A\left(\begin{pmatrix} C11 & C12 & C13 & C14 \\ C21 & C22 & C23 & C24 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ x \cdot y \\ 1 \end{pmatrix}\right) \qquad (12)$$

En todos estos casos, la matriz de transformación será necesaria para realizar la compensación de la imagen. Por cada matriz presentada (según el movimiento que se produzca), se hace un análisis y se resuelven incógnitas, los valores que se obtienen, serán los necesarios para compensar la imagen de salida con respecto a la de entrada.

4.1.3 Estimación y compensación del movimiento

La estabilización de una imagen se realiza en dos partes: estimación y compensación del movimiento. La estimación se puede definir como, el proceso a partir del cual se obtienen los vectores de movimiento de los distintos bloques de la imagen a codificar, respecto a una o más imágenes de referencia.

Hay dos tipos de estimación: Backward y Forward.

- Estimación/compensación Forward: Si se tiene una imagen de referencia (analizada mediante bloques) y la siguiente (transformada), se calcularán los vectores de movimiento correspondientes, a partir del análisis de la imagen original. Para cada bloque de la imagen de referencia se busca donde se sitúa en la imagen que se quiere codificar, y a partir de la nueva posición se extraen los vectores de movimiento. Es posible que en algunos bloques o píxeles de la imagen de referencia no aparezcan en la imagen original, si esto sucede algunos pixeles de la imagen compensada se sobrescribirán y otros quedarán sin definir generando pixeles de valor "0" (Figura 16).

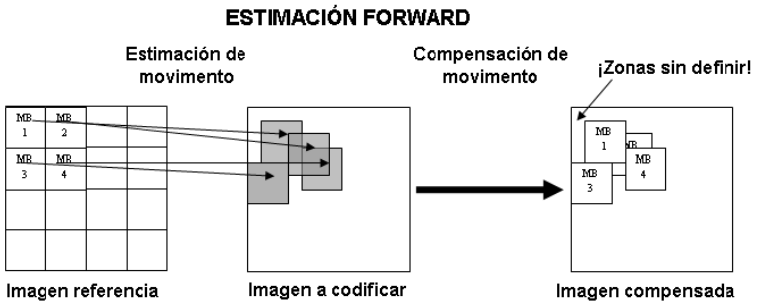


Figura 16: Estimación Forward

- Estimación/compensación Backward: En la estimación backward se definen los bloques en la imagen a codificar (calculando los vectores de movimiento) y busca donde se encuentran en la imagen de referencia. En este caso se no se deja ningún pixel sin definir ya que los bloques se van llenando según los vectores encontrados (Figura 16).

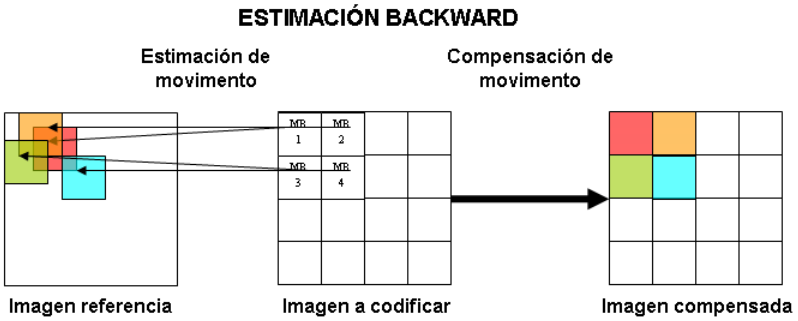


Figura 17: Estimación Backward

4.1.4 Algoritmos

Hay una gran variedad de algoritmos desarrollados para estimar los vectores de movimiento, estos, compiten entre sí para mejorar la velocidad de cómputo pero a cambio se introducen pérdidas en la secuencia final.

- Block Matching

Se trata del método más utilizado, con una implementación sencilla y con resultados de alta calidad. En este método se asumen que los movimientos solo son de traslación y no se producen cambios de iluminación.

En este algoritmo cada una de las imágenes de la secuencia de video se dividirá en bloques rectangulares. Este método detecta el movimiento entre dos imágenes con respecto a los bloques que las constituyen.

Los bloques del frame actual serán comparados con los bloques de la imagen de referencia, creando un vector que une ambas regiones (correspondencia).

Cuando se determine la máxima similitud entre bloques (minimiza un error medio) entre todos los candidatos dentro de la ventana de búsqueda, se reañizará la correspondencia. Si el bloque elegido no se encuentra en la misma posición en ambas imágenes, significa que se ha

movido. La distancia que hay entre ambos bloques será el vector de desplazamiento estimado, este se le asignará a todos los píxeles de un mismo bloque (Figura 18).

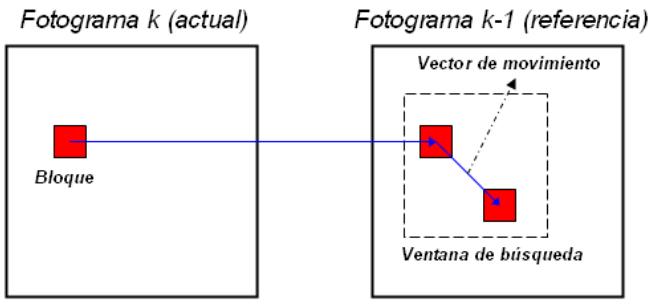


Figura 18: Block Matching

- Aproximación multiresolución

Se trata de un método para realizar la estimación del movimiento de manera rápida, para ello se reducen las imágenes a un décima parte, por lo que habrá menos puntos característicos que comparar. A la hora de hacer la compensación del movimiento, sí se utilizarán las imágenes con sus dimensiones originales, utilizando los vectores de movimiento obtenidos en las imágenes reducidas, pero escalados. De esta manera el tiempo de cómputo se reduce significativamente, aunque la calidad en la imagen procesada se modifica. (Figura 19).

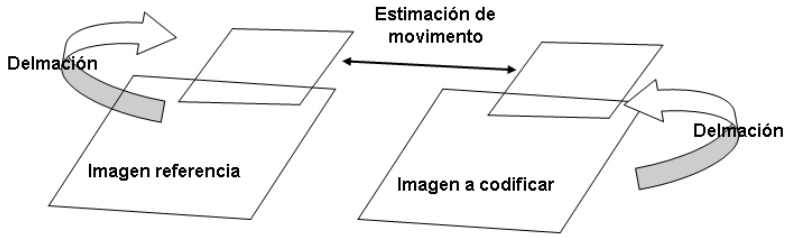


Figura 19: Estimación multiresolución

Aunque existen multitud de métodos encontrados para determinar los vectores de movimiento entre una secuencia de imágenes, solo se han especificado aquellos que guardan alguna relación con la estabilización en el humanoide, es decir aquellas que van a ser utilizadas en el algoritmo desarrollado.

4.2 ROBUST REAL TIME STABILIZATION ALGORITHM

La estabilización de la imagen se va a realizar para compensación de movimientos de traslación y rotación de las imágenes tomadas por el humanoide en movimiento.

La estabilización se realizará mediante el algoritmo RRTS (Robust Real Time Stabilization) basado en SURF (Speeded Up Robust Features).

El algoritmo RRTS tendrá dos partes fundamentales, la detección de parámetros x, y, θ para detectar cual es la variación entre las distintas imágenes y posteriormente la corrección de parámetros, donde el movimiento que se ha producido se invierte hasta que se obtenga una imagen estable.

El segundo subapartado comenzará con la explicación de la estructura general del algoritmo, para ello se inserta un diagrama de bloques de la estructura general del sistema de estabilización, en él hay tres partes, el preprocesado, donde se binarizan las imágenes y el procesado, donde se encuentra el algoritmo que realiza la estabilización, por último se realiza un ajuste de la ventana que mostrará las imágenes estabilizadas.

En el tercer subapartado se explicará detalladamente el software empleado y en que otros algoritmos o sistemas se ha basado el SSTR para su realización.

En cuarto lugar, se detallará como se ha realizado la implementación del sistema que consta de cuatro partes: binarización, detección de parámetros, corrección de parámetros y por último el ajuste de pantalla.

4.3 ESTRUCTURA GENERAL

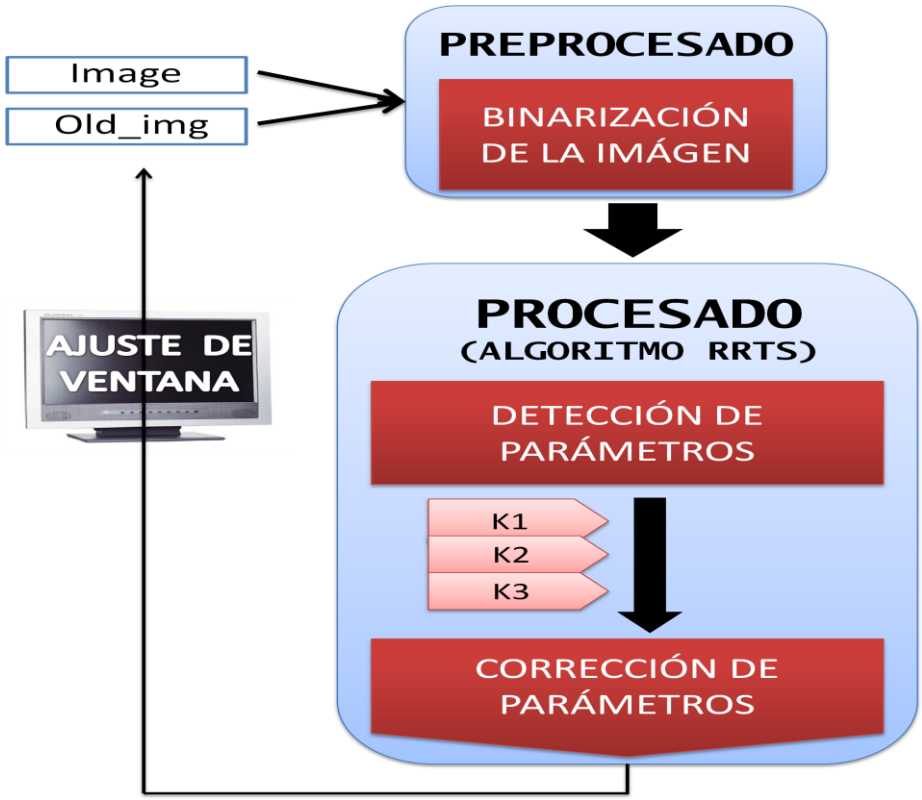


Figura 20: Esquema general de la estabilización

En el diagrama de bloques de la Figura 20 se muestra el esquema general de la estabilización. Este consta de dos partes fundamentales:

- Preprocesado
- Procesado

En la parte de preprocesado se realiza una binarización de la imagen a tratar para poder llevar a cabo el algoritmo principal.

En el procesado es donde se realiza la estabilización de la imagen. Ésta a su vez se puede dividir en dos etapas fundamentales, según se puede ver en el diagrama, detección y corrección de parámetros.

Entre estas dos etapas se introducen un conjunto de valores de ajuste (K1, K2, K3), que corresponden a tres reguladores proporcionales, el objetivo fundamental es mejorar el comportamiento dinámico del sistema de forma que se adecúe a las a las especificaciones deseadas para su funcionamiento. En la Tabla 1 se observa las respuestas que puede tomar la estabilización en función de los valores de los reguladores.

VALOR K	AMORTIGUACIÓN	DESCRIPCIÓN DEL SISTEMA
0<K<1	Sistema subamortiguado	Controla el sistema y obtiene una imagen estable.
K=1	Críticamente amortiguado	Sistema críticamente controlado
K>1	Sistema sobreamortiguado	Sistema inestable

Tabla 1: Valores de K

Si los reguladores toman valores por debajo de 1, se obtiene un sistema subamortiguado, es decir, la imagen se corresponde con un sistema estable. Se darán valores por debajo de 1 cuando la estabilización se realice en el movimiento de traslación (ya sea en X o en Y), esto hará que la secuencia de imágenes fluya de manera progresiva y suave.

Cuando el valor es 1, el sistema se encuentra críticamente amortiguado, es decir, ante cualquier perturbación puede producirse una inestabilidad. Se dará valor 1 al regulador usado en la estabilización del ángulo, porque cualquier pequeña variación por debajo de la unidad produce errores significativos en la salida del sistema.

Si por el contrario, toman valores por encima de la unidad, la estabilización se produce de manera descontrolada, y el comportamiento dinámico del sistema de acuerdo a las especificaciones establecidas no es el deseado.

En el procesado hay dos partes, en la primera se lleva a cabo la identificación del desplazamiento y giro que se producen entre una imagen y la siguiente (Figura 21), para posteriormente corregirlos. Para ello se analizan las imágenes inicial y la que llega a continuación.

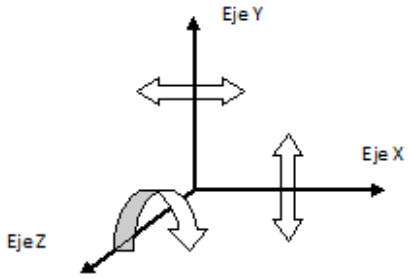


Figura 21: Variaciones que puede sufrir la imagen con respecto a los ejes de coordenadas

Tomando un primer frame de referencia estable, donde el robot tiene que encontrarse en posición erguida y parado, se compara el siguiente frame con el anterior (imagen inicial), la diferencia en los ejes X, Y y el ángulo de rotación θ entre las dos imágenes, serán los parámetros que se buscan.

Tras la identificación de los datos necesarios, se realiza la segunda parte del procesado, la corrección del desplazamiento y giro, es decir, una transformación afín genérica en 2D (

Figura 22).

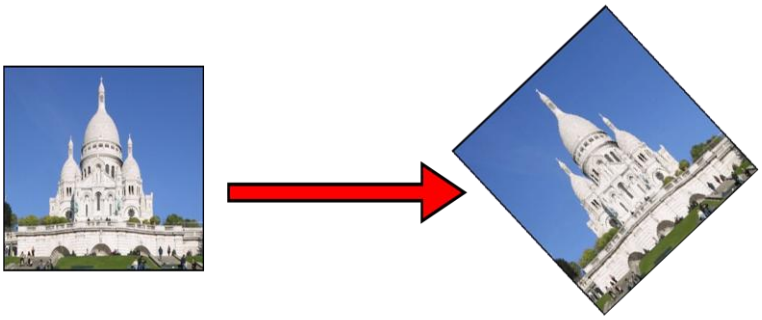


Figura 22: Transformación afín genérica.

Esta corrección solo se producirá cuando las condiciones iniciales sean las adecuadas y no en todos los casos, pues solo se busca que el sistema estabilice cuanto el robot se encuentra en

movimiento y no si se producen cambios muy grandes de ángulo, como podría ser el caso de que el humanoide se cayera.

Una vez realizada la estabilización, se saca por pantalla la imagen estabilizada. Al realizar las trasformaciones aparecerá un borde negro, que se produce a causa de píxeles no definidos, que toman valor 0 en el proceso de estabilización. Estos bordes serán eliminados recortando la imagen y que posteriormente será ampliada hasta que tenga el mismo tamaño que la imagen sin estabilizar. Esto permitirá una visualización del video mucho más cómoda obteniendo un resultado muy parecido al que vería un ser humano al caminar.

Estos pasos se irán produciendo a lo largo de todas las imágenes formando un ciclo, tomando como referencia siempre la imagen anterior, que ya estará estabilizada. Con el algoritmo presentado se consigue una imagen estable con el humanoide en movimiento.

4.4 IMPLEMENTACIÓN

4.4.1 Binarización

Se entiende como binarización de una imagen como convertir una imagen con 256 niveles de grises a una imagen en blanco y negro. La binarización se usa como un proceso previo al OCR (reconocimiento óptico de caracteres).

Primero se realizará la conversión de la imagen a escala de grises para posteriormente realizar la binarización (Figura 23). La binarización se realizará para establecer la correspondencia entre los distintos puntos característicos detectados mediante comparación de contraste.

Figura 23: Izquierda imagen en escala de grises, a la derecha binariazación de la imagen

4.4.2 Detección de parámetros

Después de la binarización realizada en la parte del preprocesado, se pasa a la parte de procesado, compuesto por dos grandes bloques: la detección de parámetros de estabilización (X,Y,θ) y la corrección de estos para obtener la imagen estabilizada.

El objetivo es utilizar un detector que encuentre los mismos puntos en el espacio a pesar de que la escena sea observada desde diferentes posiciones, ángulos o que experimente cambios de posición. Tras realizar el seguimiento de puntos, estos son caracterizados por medio de descriptores locales.

La detección y análisis de parámetros de estabilización está basada en el algoritmo SURF (Speeded Up Robust Teatures) (Bay et al., 2006) el cual es una adaptación eficiente y rápida del algoritmo SIFT (Lowe, 1999). El algoritmo SURF consta de tres partes, un detector de puntos característicos, un descriptor que asocia con vectores esos puntos, para definir el contenido de la imagen y finalmente, se realiza la correspondencia de puntos entre una imagen y la siguiente.

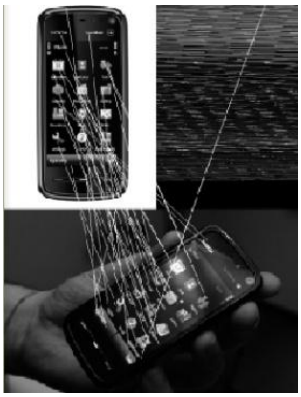


Figura 24: Apicación del algoritmo SURF

Este algoritmo se suele usar en visión por computador en tareas como reconocimiento o identificación de objetos. En la parte superior de la Figura 24 se puede observar la imagen del objeto que se desea detectar, detrás de esta, vemos una imagen general que contiene a ese objeto. Los puntos característicos en ambas imágenes serán detectados, se hará la correspondencia entre ellas y por último se unirán mediante líneas.

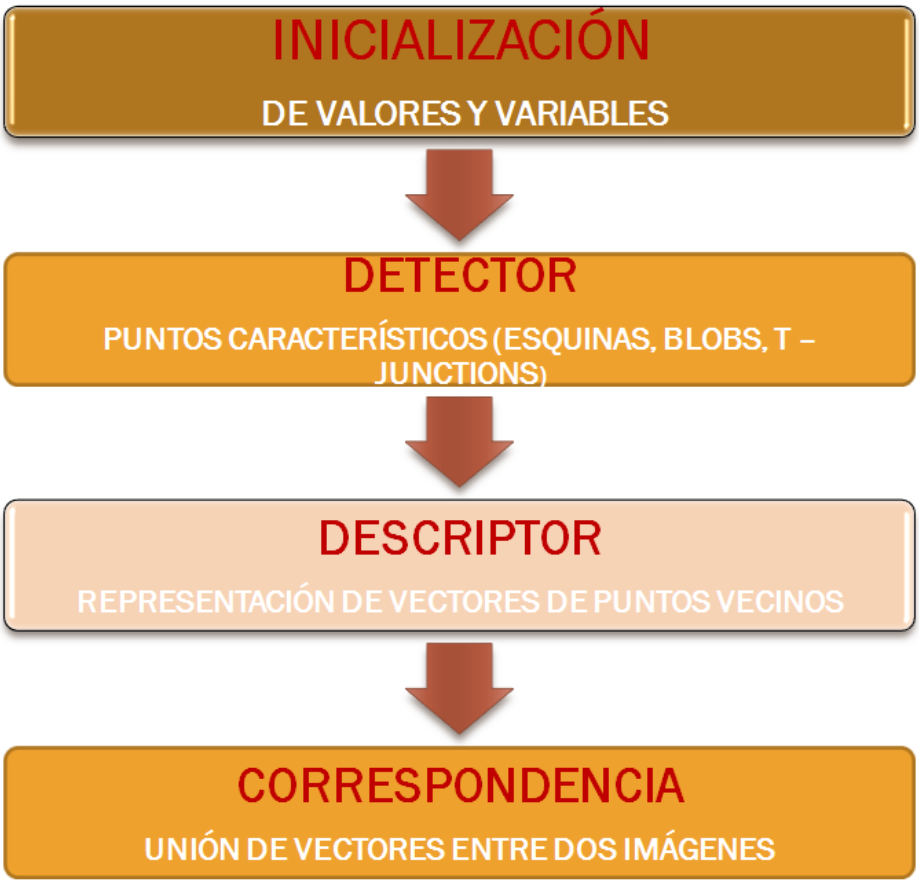


Figura 25: Diagrama del algoritmo RRTS

El nuevo algoritmo RRTS (Robust Real Time Stabilization) se basa en SURF para estabilizar una imagen en tiempo real. En este algoritmo no sólo se va a realizar una identificación, sino que también se calculan las variaciones en el eje X, Y y el ángulo θ entre dos imágenes, por último se hace una traslación afín genérica. En la Figura 25 se observan las fases del algoritmo que a continuación se explicarán.

Inicialización

En la inicialización se definen las funciones y variables que van a ir formando parte del algoritmo:

El algoritmo comienza con la inicialización de dos imágenes, la primera imagen que cumpla los criterios para comenzar la estabilización y la que llega a continuación. Estas, serán inicializadas como:

IplImage *oldImage (Primera imagen tomada)

IplImage *actualImage (Imagen que llega a continuacion)

Generalmente la imagen “actualImage” será la que se encuentra desestabilizada y la imagen “oldImage” será la que se tome como referencia.

- A continuación se selecciona una región de interés en la imagen de referencia “oldImage”, en este recuadro será donde se detecten los puntos característicos. Para ello, se inicializan los siguientes datos:

  CvPoint centro (centro del rectángulo de la región de interés)
  CvRect rect (región de interés que será un rectángulo
  int ancho, alto (dimensiones de la región)

  Las dimensiones del rectángulo serán las correspondientes al 80% de la imagen de referencia (oldImage).

  Se hace así para que a la hora de analizar la imagen y tomados los puntos característicos, la región sea recuadrada con vectores, estos servirán posteriormente para analizar y corregir el giro detectado. Para realizarlo se determinará:

  Que el centro de la imagen sea el punto de intersección de las líneas que definen el ancho y el alto de la imagen. Tomando este valor como punto de giro en la rotación.

- Se definen las matrices de rotación y traslación para poder realizar la translación afín. Estas se definen como:

  CvMat *m_rot, *m_tras

  CvPoint2D32f centro32f

  Para realizar una translación es necesario definir su matriz correspondiente y su centro de giro, la matriz variará según el tipo de movimiento que se quiera corregir, en nuestro caso, se tendrán en cuenta las traslación en x e y, así como el angulo de rotación de la imagen.

- Se inicializan los valores de los reguladores proporcionales, K1, K2 y K3 con un valor de 1, 0.5 y 0.7 respectivamente. Estos valores se han estimado de forma heurística de acuerdo a la situación de la cámara, ángulo de giro del robot al andar y factores del

terreno. Para la aplicación del algoritmo con un movimiento diferente a caminar sería necesario reajustar estos valores.

- Para que la imagen estabilice, el ángulo de giro se supone no mayor de 30°, por lo tanto este será el valor por defecto que se establezca. Cuando se produzcan giros muy elevados, como por ejemplo una caída del humanoide, no interesa estabilizar la imagen. El ángulo se inicializa de la siguiente forma:

```
int umbralPhi;
umbralPhi = 30;
```

Detector

La primera parte consiste en detectar los puntos característicos de la región de interés seleccionada.

La mayoría de las imágenes contienen muchos datos, aunque la mayoría no proporciona suficiente información para interpretar la escena. En este caso, se busca como en cualquier sistema de visión artificial, extraer las máximas características posibles y que proporcionen la máxima información posible de la imagen analizada y además que se haga de una forma robusta y eficaz. Estas características, deben ser:

- Que no suponga un tiempo de cómputo excesivo en la extracción.
- Que tengan una buena localización y con poco error de estimación.
- Deben ser características estables (que permanezcan en toda la secuencia).
- Que sean características con la mayor información posible.

Estas condiciones las cumplen las aristas, esquinas y uniones. Estas, aparecen en la mayoría de las escenas.

Una unión es la intersección de dos o más aristas en un punto de la escena. Sin embargo, debemos distinguir unión de punto característico, un punto característico se define como punto de la imagen donde el contorno de una arista tiene curvatura alta y/o el centro de una unión.

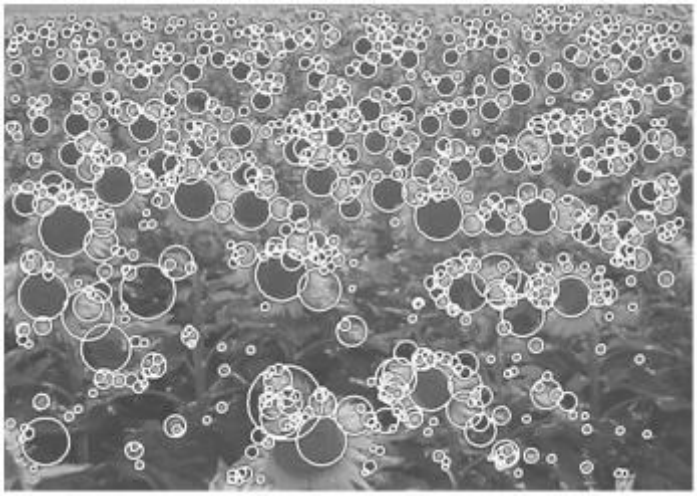


Figura 26: Regiones de los puntos característicos detectados

En este algoritmo, es necesario al menos encontrar dos puntos característicos en la escena (Figura 26), con ello, sería suficiente para recuadrar con vectores la región seleccionada en la estabilización.

El método utilizado para la detección de puntos característicos es el utilizado SURF (Bay et al. 2006). Para ello se utiliza una aproximación básica de la matriz Hessiana, su determinante será el necesario para la localización de puntos y determinación de la escala.

Dado un punto X = (x, y) en una imagen I, la matriz Hessiana H (x, σ) en x con escala σ se define como (13):

$$\mathcal{H}(\mathbf{x}, \sigma) = \begin{bmatrix} L_{xx}(\mathbf{x}, \sigma) & L_{xy}(\mathbf{x}, \sigma) \\ L_{xy}(\mathbf{x}, \sigma) & L_{yy}(\mathbf{x}, \sigma) \end{bmatrix} \quad (13)$$

donde Lxx(x, σ) es la convolución de la derivada de segundo orden de la Gaussiana, con la imagen I en el punto x, y similarmente para los demás elementos de la matriz. Para el cálculo del determinante, se realizan aproximaciones a las derivadas de segundo orden de la Gaussiana de modo que se obtienen tres aproximaciones: Dxx, Dxy y Dyy. De esta forma, el determinante de la Hessiana que nos indica la escala del punto se calcula con la siguiente fórmula (14):

$$det(\mathcal{H}_aprox) = D_{xx}D_{yy} - (0.9D_{xy})^2 \quad (14)$$

Descriptor

El descriptor se utiliza para caracterizar la región de interés de la imagen que rodea a los puntos característicos. Estos descriptores son invariantes ante traslación, rotación y cambios en la resolución de la imagen. RRTS se basa en el descriptor SURF que es parecido al descriptor SIFT, este algoritmo es computacionalmente rápido y da la posibilidad de implementarlo en tiempo real en el robot.

Para obtener el descriptor el primer paso es calcular la escala, esta es el resultado del determinante de la matriz Hessiana que se ha calculado en el apartado anterior, luego se calcula la orientación del punto de interés y por último se calculan los vectores, mediante la unión de estos se obtiene finalmente el descriptor.

Para obtener un punto invariante a la orientación, se calcula la transformada de Haar (Haar-Wavelet) (Haar, 1909) para las direcciones x e y, con una región circular de radio 6s (se entiende por s la escala correspondiente a ese punto de interés). Esto se calcula para todos sus puntos vecinos. Una vez calculado, se estima la orientación dominante mediante la suma de todos los resultados.

El descriptor se calculará mediante la obtención de puntos invariantes a la orientación, pero en vez de usar regiones circulares se estudia en regiones cuadradas de tamaño 20s (Figura 27) realizadas a partir de la orientación de cada punto invariante de radio 6s. Las regiones se dividirán regularmente en 4 subregiones cada una, en cada subregión se calcularán algunas características simples.

Figura 27: Regiones cuadradas

A continuación, se calcula la transformada de Haar para las direcciones en x e y, y se suavizan los resultados con una matriz Gaussiana obteniendo d_x y d_y. Los resultados obtenidos de cada subregión se suman y se calcula el valor de cada uno de ellos |dx| y |dy|. Por lo tanto, cada subregión proporciona un vector v de componentes v = (Σdx,Σdy,Σ|dx|,Σ|d_y|) (Figura 28).

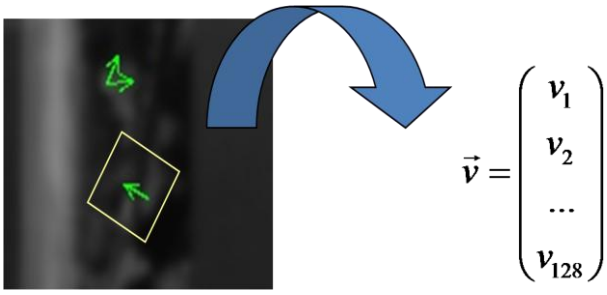


Figura 28: Vector que proporciona una subregión y su valor

El último paso para obtener el descriptor, corresponde a la unión de los distintos vectores de las subregiones.

Correspondencia

Se puede definir una correspondencia como una transformación geométrica que lleva un punto de interés de la imagen prototipo a la imagen de prueba. Siendo el objetivo encontrar un gran grupo de correspondencias que compartan la misma transformación. En este caso tenemos dos tipos de correspondencias, traslación (Figura 29) y rotación (Figura 30):

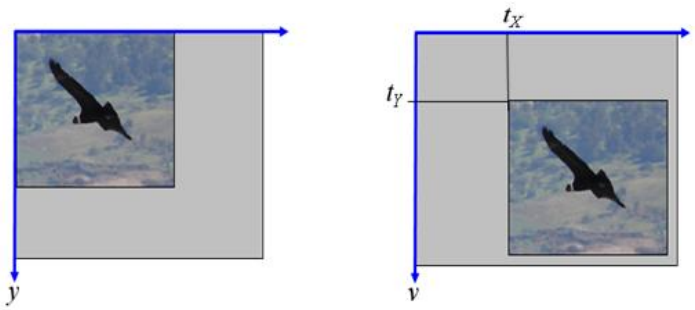


Figura 29: Traslación en una imagen

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_X \\ t_Y \end{pmatrix} \tag{15}$$

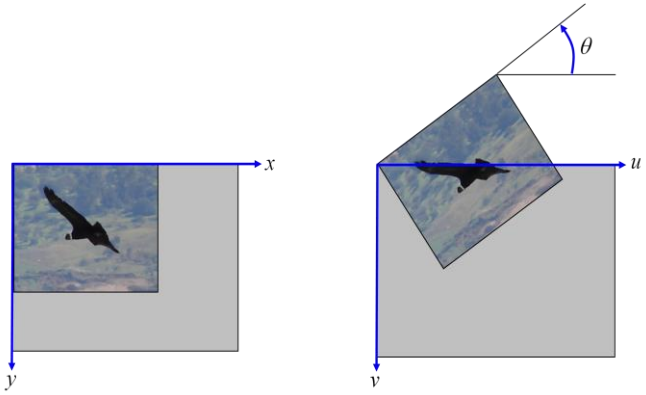


Figura 30: Rotación en una imagen

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{16}$$

Uniendo estas dos correspondencias, obtenemos una imagen con translación afín genérica (Figura 31). Obteniendo gráficamente el siguiente resultado con sus correspondientes matrices de transformación (17), (18).

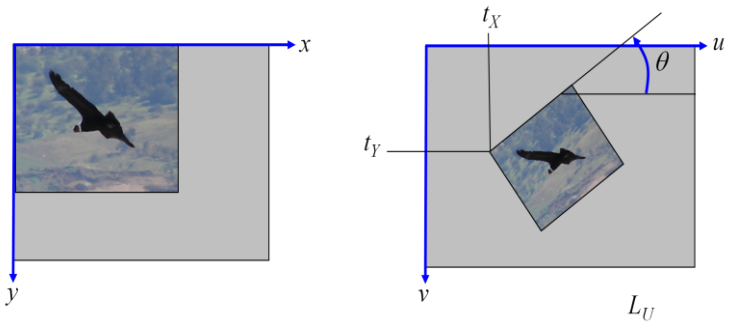


Figura 31: Traslación afín genérica

$$\begin{pmatrix} u \\ v \end{pmatrix} = e \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_X \\ t_Y \end{pmatrix} \tag{17}$$

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} e\cos\theta & e\sin\theta & t_X \\ -e\sin\theta & e\cos\theta & t_Y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \quad (18)$$

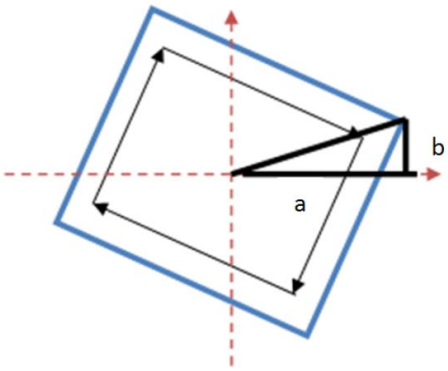


Figura 32: Cálculo del ángulo de una imagen rotada

El ángulo de rotación (19) será calculado, según la Figura 32 como:

$$\theta = tg^{-1}\frac{p.y - centro.y}{p.x - centro.x} = tg^{-1}\frac{b}{a} \quad (19)$$

siendo p.x y p.y es la distancia X e Y del punto desplazado.

Para detectar si se realiza la correspondencia o no, se compara el contraste de cada punto característico de modo que si este no coincide, no se realiza el emparejamiento (Figura 33).

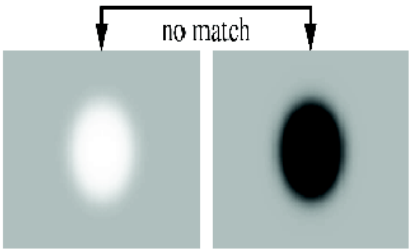


Figura 33: No matching

4.4.3 Corrección de parámetros

La última parte del algoritmo RRTS será la inversión de los valores obtenidos, mediante una trasformación afín genérica en 2D. En esta parte la variación de valores en X, Y y θ serán corregidas hasta que los ejes de la imagen siguiente coincidan con los ejes de la imagen anterior. Los valores a compensar serán θ, X=a y Y=b (Figura 32).

La transformación genérica en 2D se realiza mediante la función:

cvGetAffineTransform(a, b, m_tras); (Cálculo de la matriz de traslación)

cv2DRotationMatrix(centro32f, pos.phi*K.phi, 1.0, m_rot); (Cálculo de la matriz de rotación.)

cvWarpAffine(input, compensada, m_tras); (compensación del movimiento de traslación)

cvWarpAffine(input, compensada, m_rot); (compensación del giro)

En la corrección de parámetros se sigue el siguiente diagrama de flujo (Figura 34):

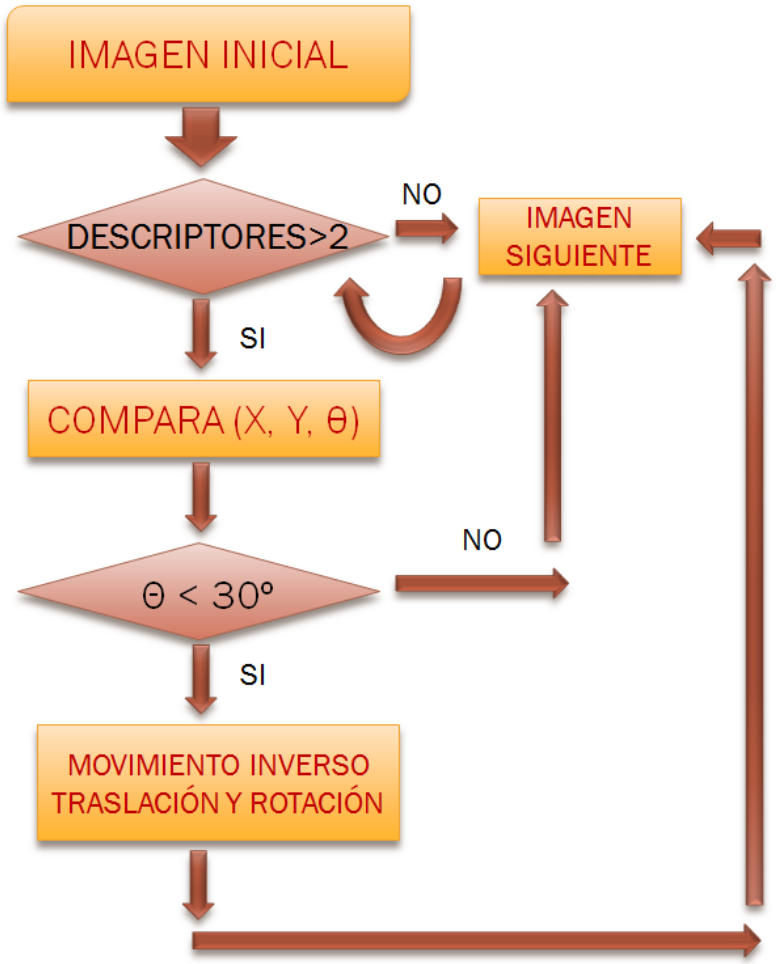


Figura 34: Corrección de parámetros

La realización de la corrección de parámetros solo se realizará cuando los puntos característicos detectados sean como mínimo dos, sino, pasa a la imagen siguiente y vuelve a buscarlos, esto se repetirá, hasta que sean encontrados. Se establece esta restricción ya sino el sistema no recuadra con vectores la región de interés establecida (80 % de la imagen de referencia).

Una vez se detectan más de dos puntos de interés, se determinan los parámetros x, y, θ que cuantifican la variación de movimiento entre imágenes, si el valor de θ es mayor de 30º, no se realiza la estabilización. Esta segunda restricción está establecida para que el sistema no estabilice cuando el humanoide no se encuentra en movimiento, por ejemplo si l robot sufre una caída.

Una vez realizada la estabilización en la imagen, esta pasará a ser la imagen de referencia (ya que se encuentra estabilizada) y la siguiente imagen pasará por el mismo proceso para ser corregida.

4.4.4 Ajuste de pantalla

En la última parte de la estabilización se realiza un ajuste de la pantalla donde se muestra la imagen corregida. Esta será recortada y ampliada hasta tener el tamaño de la imagen sin estabilizar.

Esto provoca una gran comodidad a la hora de visualizar los resultados obtenidos, de manera que las zonas que quedan sin definir tras el proceso de estabilización desaparecen.

Este ajuste se realiza mediante la siguiente función de OpenCV:

```
cvGetPerspectiveTransform();

cvWarpPerspective();
```

Mediante valores que se dan a estas funciones, es posible determinar cuánto se desea recortar la imagen estabilizada y posteriormente ampliada. Variará de unos casos a otros, ya que dependerá de las restricciones de entrada que se establezcan en el algoritmo (ángulo máximo de estabilización).

En este caso, el ajuste de la ventana se realiza de acuerdo a los valores de K establecidos en K1=1, K2=0.5, K3=0.7, para otros valores, se verán bordes negros, es decir, píxeles con valor nulo.

A continuación se muestra en la Figura 35 se puede ver el resultado de ajustar la ventana.

Figura 35: De izquierda a derecha, imagen original (sin estabilizar), imagen estabilizada, región (imagen estabilizada con ajuste de la ventana)

5.RESULTADOS EXPERIMENTALES

5.1. ADAPTACIÓN AL ROBOT HOAP-3 OFFLINE

Para poder ver los resultados del algoritmo, lo primero que se experimentado es la implementación offline sobre videos grabados con el humanoide. Esto permite mayor comodidad a la hora e realizar pruebas y posibles modificaciones del algoritmo.

A continuación se expondrán los resultados de la estabilización offline con distintos valores de K.

En la Tabla 2 se muestran los distintos resultados según los valores dados a K1, K2 y K3.

K1	Rotación	0<K1<1	Cuando vale 0 no se produce la corrección en el giro Cuando vale 1 se produce la corrección completamente
K2	Traslación en X	0<K2<1	Cuando vale 0 no se produce la corrección en la altura Cuando vale 1 se produce la corrección completamente
K3	Traslación en Y	0<K3<1	Cuando vale 0 no se produce la corrección en la posición Cuando vale 1 se produce la corrección completamente

Tabla 2: Corrección de los distintos parámetros según los valores de K

A continuación, se ven gráficamente los resultados obtenidos tras realizar la implementación del algoritmo con distintos valores de K.

Caso 1:

En este caso K1=1, K2=1, K3=1, es decir, se realiza la estabilización completamente en la rotación y traslación.

Como puede verse en Figura 36 , la imagen se encuentra estabilizada. El recuadro tomado como referencia en la imagen original será el que se mantenga durante toda la secuencia. Los

tramos nuevos en x e y que no formen parte de la zona recuadrada no aparecerán en la ventana "ESTABILIZADA".

Dar valores demasiado altos a las constantes K, produce una pérdida de información muy grande como puede verse en la ventana "REGION", aunque se producen imágenes estables.

Figura 36: Estabilización completa

Caso 2:

Los valores dados son: K1= 0.5, K2= 1 y K3=1, en este caso, la corrección del giro no se realiza totalmente.

Como puede verse en la ventana "ESTABILIZADA" de la Figura 37 se produce la corrección en los ejes x e y completamente pero la imagen se encuentra un poco girada, esto es debido al valor de K1=0.5.

Por lo tanto la imagen no es estable. Para que la imagen lo sea, la condición fundamental es que K1 tenga siempre un valor unidad, ya que este parámetro es el que gráficamente determina si la imagen es estable o no.

Figura 37: Corrección en la traslación y el 50% en el giro

Caso 3:

Una vez, determinado que el valor de K1 debe tener valor 1 para que los resultados cumplan las especificaciones mínimas. Damos valores a distintos a K2 y K3 para visualizar los distintos resultados.

En este caso con K1=1, K2=1 y K3=0, no se realiza la corrección en la altura aunque sí en el giro y traslación en x, como puede verse en la Figura 38.

Puede verse que la pérdida de información en este caso es considerable, ya que tras un cambio grande se posición en el eje x como se sucede en la imagen original, la zona derecha se ve recortada para ajustarse a la región señalada. Por lo que, sería recomendable, dar un valor menos restrictivo a K3, ya que un valor por debajo de 1 no influye en la estabilización final de la secuencia y no se pierde tanta información en la imagen.

Figura 38: Imagen sin corrección en altura

Caso 4:

En el caso 4, al igual que el anterior se dará valor 0 a una de las constantes que corrigen las traslación, siendo los valores de K: K1=1, K2=0 y K3=1.

La corrección en la posición no se produce, pero si lo hace en el giro y en la altura, tras el ajuste con estos valores, se puede observar en la Figura 39 que la imagen es estable y la pérdida de información que se produce no es grande.

Figura 39: Imágen sin corrección en posición

Caso5:

Teniendo en cuenta los resultados de los casos anteriores, en este, se dan valores de acuerdo a lo establecido al caso 3, siendo K1= 1, K2=1 y K3=0.7.

Dando valores a K3 por debajo de 1 se produce poca pérdida de información, y el resultado es una imagen estable (Figura 40). Sin embargo, al dar un valor de 1 a K2 sigue produciéndose mucha pérdida de información en la imagen, la solución, sería dar valores a K2 y K3 por debajo de 1.

Figura 40: Estabilización con K3<1

Caso 6:

En este caso de establece un valor de 0.5 para K2, 1 para K1 y 1 para K3, teniendo en cuenta los resultados del caso 4. Al darle un valor por encima de 0 la pérdida de información es mucho menor, aunque como puede verse en la ventana “ESTABILIZADA” en la Figura 41 es necesario dar valores por debajo de 1 a K3.

Figura 41: Estabilización con K2<1

Caso 7:

En este caso, como se determinaba en el caso anterior, aparte de dar valores por debajo de 1 a la constante K3, también se hará para K2.

En este caso vemos en la Figura 42 que los resultados en la ventana ”REGION” son los esperados, mostrándose una imagen estable y recortada, de manera que no puede visualizarse el proceso realizado para la implementación, como se muestra en la ventana “ESTABILIZADA”, donde aparecen bordes negros.

Figura 42: Imagen estabilizada según el estudio de los distintos valores de K

A continuación se muestran en una tabla los resultados obtenidos en los diferentes casos propuestos.

Caso	K1	K2	K3	Estabilidad	Resultado
1	1	1	1	Estable	Mucha pérdida de información en la imagen estable
2	0.5	1	1	Inestable	-
3	1	1	0	Estable	Mucha pérdida de información en la imagen estable
4	1	0	1	Estable	Poca pérdida de información
5	1	1	0.7	Estable	Poca pérdida de

					información
6	1	0.5	1	Estable	Menos pérdida de información que en el caso 4
7	**1**	**0.5**	**0.7**	**Estable**	**Casi sin pérdida de información**

Tabla 3: Estudio de la estabilidad según el valor de las distintas constantes K

Como puede verse en la Tabla 3, el único parámetro que puede producir la inestabilidad de la imagen al estar por debajo de 1 es K1, por lo tanto este valor no se modificará. Para los distintos valores de K2 y K3 se tomarán los valores de los casos más favorables, que son K2=0.5 y K3=0.7.

5.2. ADAPTACION EN TIEMPO REAL

Para realizar la adaptación el tiempo real al robot, lo primero que hay que establecer es la comunicación con el humanoide. Posteriormente se detallan los resultados experimentales obtenidos.

Comunicación con el robot

El algoritmo RRTS para la estabilización de la imagen descrito anteriormente fue implementado en el robot humanoide HOAP-3 para estabilizar la imagen percibida por un operador remoto en la interfaz de teleoperación, mejorando la comodidad y el desempeño del teleoperador del robot.

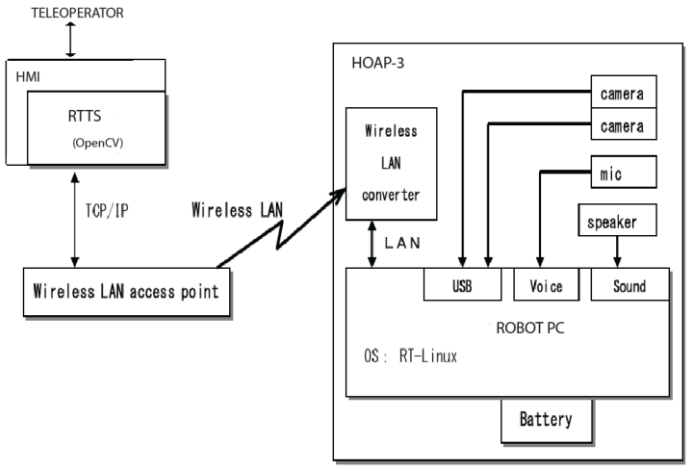


Figura 43. Sistema de teleoperación del robot HOAP-3

La Figura 43 muestra la muestra la configuración del sistema de teleoperación para el robot HOAP-3. La comunicación entre y el robot y la interfaz de control del teleoperador se realiza mediante de enlaces de TCP/IP a través de una red inalámbrica independiente.

La imagen capturada de las cameras del robot es previamente procesada mediante el algoritmo RRTS antes de su visualización por el teleoperador. El sistema de estabilización de la imagen recibe del servidor del HOAP-3 imágenes de la cámara del robot, a un ratio de 6-10 fps

aproximadamente., en formato YUV240p del cual es transformado al formato de imagen IplImage de openCV para su adecuado procesamiento mediante el algoritmo RRTS. Finalmente una la imagen estabilizada resultante se presenta al teleoperador.

5.3. RESULTADOS Y DISCUSIÓN

En este aparatado se muestran y explican los resultados de la estabilización obtenidos en el humanoide. Para ello primero se ha hecho un estudio offline. Esto quiere decir que se ha ejecutado el algoritmo en un video previamente grabado, con el fin de disponer de material con el que trabajar. Posteriormente, como se detalla en el siguiente apartado se implementa en tiempo real.

Discusión de resultados en la implementación offline

A continuación, se muestran tres figuras en las que aparece la imagen original y el resultado de la aplicación del algoritmo. En la Figura 44 se puede apreciar que el algoritmo ha realizado una corrección en la rotación, es decir, se compensa el ángulo de giro sufrido en el eje z.

Figura 44: Estabilización en la rotación

En el siguiente caso, no solo se produce una compensación el movimiento de rotación, sino que también se puede observar una corrección en el eje y. El algoritmo ha sido implementado para que también compense el movimiento ascendente/descendente de la imagen, como puede verse en la Figura 45.

Figura 45: Estabilización en rotación y traslación en y

Finalmente, tal y como se muestra en la Figura 46 se compensan los dos tipos de movimientos: rotación en Z y traslación en X e Y.

Figura 46: Estabilización en la rotación y traslación en x e y

Los resultados obtenidos en la mayoría de los casos son los esperados. Hay una mejoría notable comparando la imagen original y la estabilizada. En este caso ha sido fácil de analizar ya que nos encontramos con un entorno fijo, sin que se presente ninguna interferencia externa en el análisis (correcto funcionamiento del robot, intrusión de elementos dinámicos en la imagen, etcétera).

Discusión de resultados en la implementación online

En este apartado, se exponen los resultados obtenidos tras la implementación online del algoritmo en el humanoide.

Como se puede ver a continuación se han capturado tres imágenes donde se pueden apreciar los resultados obtenidos. En el primer caso se realiza una compensación en la rotación Figura 47, en el segundo no solo se compensa la rotación, también la traslación en el eje y

Figura 48 , en el último caso se corrige el movimiento en los tres ejes

Figura 49.

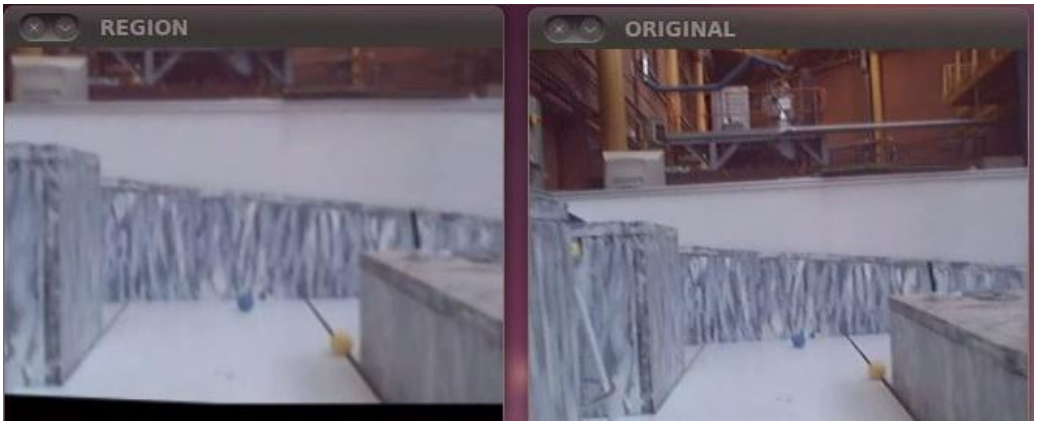


Figura 47: Estabilización en la rotación

Figura 48: Estabilización en la rotación y en y

Figura 49: Estabilización en la rotación y traslación en x e y

Tras la evaluación de las imágenes obtenidas en tiempo real, el resultado es satisfactorio pero menos preciso que en el caso de la implementación a partir de un video grabado. Esto se debe a que en tiempo real hay que asegurarse de que los primeros fotogramas que se tomen sean estáticos, ya que si no corrige con respecto a una primera imagen desestabilizada.

Otro problema presentado es el hecho de que al trabajar con un mismo video no es necesario el ajuste de parámetros del algoritmo, sin embargo en el desarrollo en tiempo real, las condiciones en el entorno (estado del robot, iluminación, terreno, etc.) no siempre son las mismas. El único inconveniente posible que se puede presentar es que se trabaje en un entorno que no posea puntos característicos suficientes para el análisis de la imagen por parte del algoritmo.

Con todo esto la estabilización obtenida resulta altamente satisfactoria, ya que se consigue estabilizar movimientos muy bruscos. Siendo el método teórico estudiado el adecuado en todos los casos que se nos han presentado.

6.CONCLUSIONES Y TRABAJOS FUTUROS

6.1. CONCLUSIONES

Tras la realización e implementación del algoritmo RRTS los resultados obtenidos han sido los esperados en la mayoría de los casos.

Los objetivos planteados se han cumplido totalmente y de forma satisfactoria en todos los casos.

1. El objetivo principal del proyecto es la estabilización de las imágenes que ofrece la cámara estereoscópica del humanoide HOAP – 3 cuando se encuentra en movimiento.

Se cumple el objetivo principal que es la realización de un sistema de estabilización que se ajuste a los requisitos establecidos, corregir las fluctuaciones que se producen en la imagen cuando el robot se encuentra en movimiento. No solo se estabiliza la imagen para que esta no esté girada, sino que se corrigen posibles cambios de altura y posición en la secuencia de imágenes. Este sistema estabiliza y además es posible adaptarlo a distintos escenarios mediante el ajuste de las constantes K introducidas.

2. Realizar la estabilización de un video de un conjunto de imágenes en movimiento de forma offline, es decir, procesar esa imagen desde un PC con el video previamente grabado.

Tras la grabación de videos se implementa sobre estos y los resultados son positivos como puede verse en los resultados experimentales del capítulo 5.

3. Una vez se ha comprobado que el algoritmo funciona en un video previamente grabado, implementar ese mismo algoritmo directamente en el robot, creando un sistema cliente-servidor para obtener la secuencia de imágenes del robot en un PC y ejecutar el algoritmo de estabilización de forma que se muestre al usuario.

Este objetivo se cumple en su totalidad, ya que tras la evaluación del algoritmo puede verse que la imagen aparece estabilizada independientemente que se haya implementado en videos o en tiempo real.

Con respecto al funcionamiento del algoritmo los resultados son en general satisfactorios, ya que se trata de un sistema robusto y con una velocidad de implementación buena.

Con respecto a la inicialización hubiese sido más cómodo haber creado variables dinámicas donde los valores de K tomasen valores automáticamente según el escenario en que nos encontremos, esto permite que sea adaptable a cualquier medio sin un previo análisis del caso, aunque supone dar un planteamiento diferente al código realizado.

En determinados casos, el método de detección de puntos no es el adecuado, ya que o no se obtienen suficientes (debido a la iluminación de la imagen o que se elige un escenario con poco contraste) o los puntos detectados no tienen suficiente información (no se obtienen una correspondencia clara).

La única limitación que tiene, es que solo funciona en escenarios con elementos estáticos. En el caso de tener objetos en movimiento en la escena o personas que se crucen implica posibles errores en la estabilización, pues los puntos que se detectan van cambiando entre una etapa y otra del algoritmo y por lo tanto la correspondencia no se realiza adecuadamente.

Antes de plantearnos posibles mejoras debemos destacar el buen funcionamiento del algoritmo, comentar que se trata de un sistema innovador en la robótica humanoide y que es necesaria e incluso para el funcionamiento de otros algoritmos.

Para finalizar, cabe destacar que este Proyecto de Fin de Carrera ha producido una publicación en el 7º Workshop de Robocity2030 (A. P. Mateo et al., 2010).

6.2. TRABAJOS FUTUROS

Un posible trabajo futuro sería la realización de un estudio más profundo sobre iluminación y factores externos que producen que no se implemente correctamente el algoritmo, buscando métodos adicionales con mejores resultados para cualquier iluminación por ejemplo.

Además se podría realizar un análisis exhaustivo en los distintos valores que las constantes K pueden tomar, podría haber supuesto una pequeña mejora de la estabilización en el escenario propuesto, y tener la posibilidad de disponer de un sistema ajustado finamente.

Otro trabajo futuro es la utilización del algoritmo en otros humanoides o aplicaciones y comprobar su funcionamiento offline y en tiempo real. De hecho, como trabajo futuro a corto plazo se plantea implementar el algoritmo en el manipulador móvil MANFRED de la Universidad Carlos III de Madrid.

Para terminar, el cambio más fuerte que se podría dar al algoritmo sería que estabilizase no solo para escenarios estáticos, sino que se implementara correctamente con objetos móviles en escena.

BIBLIOGRAFÍA

A. Amanatiadis, A. Gasteratos, S. Papadakis and V. Kaburlasos. "Image Stabilization in Active Robot Vision" in Robot Vision, edited by: Ales Ude. Vukovar, Croatia, 2010.

M. Arbulu, D. Kaynov, L. M. Cabas, and C. Balaguer: The RH-1 full-sizehumanoid robot: design, walking pattern generation and control. Journal of Applied Bionics and Biomechanics., 6(3):301–344, 2009.

H. Bay, T. Tuytelaars and Luc Van Gool. SURF: Speeded Up Robust Features. In book "Computer Vision - ECCV 2006". Lecture Notes in Computer Science. Springer Berlin/Heidelberg. 2006.

B. Cardani: Optical image stabilization for digital cameras, IEEE Control Syst. Mag. 26(2): 21–22., 2006.

K. Hayashi, Y. Yokokohji, T. Yoshikawa: Tele-existence Vision System with Image Stabilization for Rescue Robots. Pp. 50 – 55. Proceedings of IEEE International Conference on Robotics and Automation.2005

K. Kaneko, F. Kanehiro, S. Kajita, H. Hirukawa, T. Kawasaki, M. Hirata, K. Akachi, y T. Isozumi: Humanoid robot HRP-2. Proceedings of IEEE International Conference on Robotics and Automation, 2004.

T. Kinugasa, N. Yamamoto, H. Komatsu, S. Takase. & T. Imaide: Electronic image stabilizer for video camera use, IEEE Trans. Consum. Electron. 36(3): 520–525, 1990.

David G. Lowe: Distinctive image features from scale-invariant key-points. International Journal of Computer Vision, 2:91–110, 2004.

Y. Ogura, H. Aikawa, H. Kondo, A. Morishima, H. Lim, and A. Takanishi: Development of a new humanoid robot WABIAN-2. Pages 76–81. International Conference on Proceedings of IEEE Robotics and Automation, 2006.

M. Oshima, T. Hayashi, S. Fujioka, T. Inaji, H. Mitani, J. Kajino, K. Ikeda & K. Komoda: VHS camcorder with electronic image stabilizer, IEEE Trans. Consum. Electron. 35(4): 749–758., 1989.

C. Pan, Z. & Ngo: Selective object stabilization for home video consumers, IEEE Trans. Consum. Electron. 51(4): 1074–1084, 2005.

Y. Sakagami, R. Watanabe, C. Aoyama, S. Matsunaga, N. Higaki, and K. Fujimura: The intelligent asimo: System overview and integration. pages 2478–2483. IEEE/RSJ international conference on intelligent robots and systems, 2002.

X. Xu, L. & Lin: Digital image stabilization based on circular block matching, IEEE Trans. Consum. Electron. 52(2): 566–574, 2006.

Alan C. Brooks: Real-Time Digital Image Stabilization, EE 420 Image Processing Computer Project Final Paper. Electrical Engineering Department, Northwestern University, Evanston, IL 60208 USA, 2003.

Y. Matsushita, E. Ofek, X. Tang and H. Shum:. "Full-frame Video Stabilization", IEEE International Conference on Computer Vision and Pattern Recognition (CVPR), Vol. 1, pp. 50-57, 2005.

A. Censi, A. Fusiello, y V. Roberto: Image Stabilization by Features Tracking. In Proceedings of the 9th International Conference on Image Analysis and Processing, Venice, Italy, September 1999. IAPR. To appear. 1999.

R. Hu, R. Shi, I. Shen, W. Chen: Video Stabilization Using Scale-Invariant Features, iv, pp.871-877, 11th International Conference Information Visualization (IV '07), 2007

A. M. Romero and M. Cazorla: Comparativa de detectores de características visuales y su aplicación al SLAM, 2009

F. Liu, M. Gleicher, H. Jin y A. Agarwala: Content-Preserving Warps for 3D Video Stabilization, Computer Sciences Department, University of Wisconsin-Madison and Adobe Systems Inc., 2009.

H. Sevilla: Estabilización de imágenes de video tomadas desde una cámara en movimiento, Universidad Carlos III de Madrid, 2003.

A.P. Mateo et al.: Robust Real Time Stabilization: Estabilización de la imagen con aplicación en el robot humanoide HOAP-3, 2010.

http://www.canon.es/About_Us/Press_Centre/Press_Releases/Consumer_News/News/Hybrid_IS_Press_Release.asp consultado en Julio 2010

Gary R Learning openCV: computer vision with the OpenCV library *1st. ed.* Bradski.

L. Shapiro and G. Stockman, Computer Vision, University of Washington, 2000.

ANEXO

CalculoErrores.cpp

```
#include "CalculoErrores.h"


Comparador::Comparador(IplImage* init)
{
	ancho=init->width/2;
	alto=init->height/2;

	oldImage = cvCloneImage(init);
}

double
compareSURFDescriptors( const float* d1, const float* d2, double best, int length )
{
   double total_cost = 0;
   assert( length % 4 == 0 );
   for( int i = 0; i < length; i += 4 )
   {
      double t0 = d1[i] - d2[i];
      double t1 = d1[i+1] - d2[i+1];
      double t2 = d1[i+2] - d2[i+2];
      double t3 = d1[i+3] - d2[i+3];
      total_cost += t0*t0 + t1*t1 + t2*t2 + t3*t3;
      if( total_cost > best )
         break;
   }
   return total_cost;
}


int
naiveNearestNeighbor( const float* vec, int laplacian,
                      const CvSeq* model_keypoints,
                      const CvSeq* model_descriptors )
{
   int length = (int)(model_descriptors->elem_size/sizeof(float));
   int i, neighbor = -1;
   double d, dist1 = 1e6, dist2 = 1e6;
   CvSeqReader reader, kreader;
   cvStartReadSeq( model_keypoints, &kreader, 0 );
   cvStartReadSeq( model_descriptors, &reader, 0 );

   for( i = 0; i < model_descriptors->total; i++ )
   {
      const CvSURFPoint* kp = (const CvSURFPoint*)kreader.ptr;
      const float* mvec = (const float*)reader.ptr;
```

```
            CV_NEXT_SEQ_ELEM( kreader.seq->elem_size, kreader );
        CV_NEXT_SEQ_ELEM( reader.seq->elem_size, reader );
        if( laplacian != kp->laplacian )
            continue;
        d = compareSURFDescriptors( vec, mvec, dist2, length );
        if( d < dist1 )
        {
            dist2 = dist1;
            dist1 = d;
            neighbor = i;
        }
        else if ( d < dist2 )
            dist2 = d;
    }
    if ( dist1 < 0.6*dist2 )
        return neighbor;
    return -1;
}

void
findPairs( const CvSeq* objectKeypoints, const CvSeq* objectDescriptors,
           const CvSeq* imageKeypoints, const CvSeq* imageDescriptors, vector<int>& ptpairs )
{
    int i;
    CvSeqReader reader, kreader;
    cvStartReadSeq( objectKeypoints, &kreader );
    cvStartReadSeq( objectDescriptors, &reader );
    ptpairs.clear();

    for( i = 0; i < objectDescriptors->total; i++ )
    {
        const CvSURFPoint* kp = (const CvSURFPoint*)kreader.ptr;
        const float* descriptor = (const float*)reader.ptr;
        CV_NEXT_SEQ_ELEM( kreader.seq->elem_size, kreader );
        CV_NEXT_SEQ_ELEM( reader.seq->elem_size, reader );
        int nearest_neighbor = naiveNearestNeighbor( descriptor, kp->laplacian, imageKeypoints,
imageDescriptors );
        if( nearest_neighbor >= 0 )
        {
            ptpairs.push_back(i);
            ptpairs.push_back(nearest_neighbor);
        }
    }
}


void
flannFindPairs( const CvSeq*, const CvSeq* objectDescriptors,
           const CvSeq*, const CvSeq* imageDescriptors, vector<int>& ptpairs )
{
            int length = (int)(objectDescriptors->elem_size/sizeof(float));

    cv::Mat m_object(objectDescriptors->total, length, CV_32F);
            cv::Mat m_image(imageDescriptors->total, length, CV_32F);


            // copy descriptors
    CvSeqReader obj_reader;
            float* obj_ptr = m_object.ptr<float>(0);
```

```
    cvStartReadSeq( objectDescriptors, &obj_reader );
    for(int i = 0; i < objectDescriptors->total; i++ )
    {
        const float* descriptor = (const float*)obj_reader.ptr;
        CV_NEXT_SEQ_ELEM( obj_reader.seq->elem_size, obj_reader );
        memcpy(obj_ptr, descriptor, length*sizeof(float));
        obj_ptr += length;
    }
    CvSeqReader img_reader;
            float* img_ptr = m_image.ptr<float>(0);
    cvStartReadSeq( imageDescriptors, &img_reader );
    for(int i = 0; i < imageDescriptors->total; i++ )
    {
        const float* descriptor = (const float*)img_reader.ptr;
        CV_NEXT_SEQ_ELEM( img_reader.seq->elem_size, img_reader );
        memcpy(img_ptr, descriptor, length*sizeof(float));
        img_ptr += length;
    }

    // find nearest neighbors using FLANN
    cv::Mat m_indices(objectDescriptors->total, 2, CV_32S);
    cv::Mat m_dists(objectDescriptors->total, 2, CV_32F);
    cv::flann::Index flann_index(m_image, cv::flann::KDTreeIndexParams(4));  // using 4 randomized
kdtrees
    flann_index.knnSearch(m_object, m_indices, m_dists, 2, cv::flann::SearchParams(64) ); // maximum
number of leafs checked

    int* indices_ptr = m_indices.ptr<int>(0);
    float* dists_ptr = m_dists.ptr<float>(0);
    for (int i=0;i<m_indices.rows;++i) {
            if (dists_ptr[2*i]<0.6*dists_ptr[2*i+1]) {
                    ptpairs.push_back(i);
                    ptpairs.push_back(indices_ptr[2*i]);
            }
    }
}


/* a rough implementation for object location */
int
locatePlanarObject( const CvSeq* objectKeypoints, const CvSeq* objectDescriptors,
                const CvSeq* imageKeypoints, const CvSeq* imageDescriptors,
                const CvPoint src_corners[4], CvPoint dst_corners[4] )
{
    double h[9];
    CvMat _h = cvMat(3, 3, CV_64F, h);
    vector<int> ptpairs;
    vector<CvPoint2D32f> pt1, pt2;
    CvMat _pt1, _pt2;
    int i, n;

#ifdef USE_FLANN
    flannFindPairs( objectKeypoints, objectDescriptors, imageKeypoints, imageDescriptors, ptpairs );
#else
    findPairs( objectKeypoints, objectDescriptors, imageKeypoints, imageDescriptors, ptpairs );
#endif

    n = ptpairs.size()/2;
```

```
    if( n < 4 )
        return 0;

    pt1.resize(n);
    pt2.resize(n);
    for( i = 0; i < n; i++ )
    {
        pt1[i] = ((CvSURFPoint*)cvGetSeqElem(objectKeypoints,ptpairs[i*2]))->pt;
        pt2[i] = ((CvSURFPoint*)cvGetSeqElem(imageKeypoints,ptpairs[i*2+1]))->pt;
    }

    _pt1 = cvMat(1, n, CV_32FC2, &pt1[0] );
    _pt2 = cvMat(1, n, CV_32FC2, &pt2[0] );
    if( !cvFindHomography( &_pt1, &_pt2, &_h, CV_RANSAC, 5 ))
        return 0;

    for( i = 0; i < 4; i++ )
    {
        double x = src_corners[i].x, y = src_corners[i].y;
        double Z = 1./(h[6]*x + h[7]*y + h[8]);
        double X = (h[0]*x + h[1]*y + h[2])*Z;
        double Y = (h[3]*x + h[4]*y + h[5])*Z;
        dst_corners[i] = cvPoint(cvRound(X), cvRound(Y));
    }

    return 1;
}

int calcular_errores(IplImage* entradaOld, IplImage* entrada)
{

    //const char* object_filename = argc == 3 ? argv[1] : "parte-foto.png";
    //const char* scene_filename = argc == 3 ? argv[2] : "foto-completa.png";


    CvMemStorage* storage = cvCreateMemStorage(0);

  cvNamedWindow("Object", 1);
  cvNamedWindow("Object Correspond", 1);

    static CvScalar colors[] =
    {
        {{0,0,255}},
        {{0,128,255}},
        {{0,255,255}},
        {{0,255,0}},
        {{255,128,0}},
        {{255,255,0}},
        {{255,0,0}},
        {{255,0,255}},
        {{255,255,255}}
    };



    IplImage* object = cvCloneImage(entradaOld);//cvLoadImage( object_filename,
CV_LOAD_IMAGE_GRAYSCALE );
    IplImage* image = cvCloneImage(entrada);//cvLoadImage( scene_filename,
CV_LOAD_IMAGE_GRAYSCALE );
```

```
    /*if( !object || !image )
    {
        fprintf( stderr, "Can not load %s and/or %s\n"
            "Usage: find_obj [<object_filename> <scene_filename>]\n",
            object_filename, scene_filename );
        exit(-1);
    }
*/
    IplImage* object_color = cvCreateImage(cvGetSize(object), 8, 3);
    cvCvtColor( object, object_color, CV_GRAY2BGR );



    CvSeq *objectKeypoints = 0, *objectDescriptors = 0;
    CvSeq *imageKeypoints = 0, *imageDescriptors = 0;
    int i;
    CvSURFParams params = cvSURFParams(500, 1);



    double tt = (double)cvGetTickCount();
    cvExtractSURF( object, 0, &objectKeypoints, &objectDescriptors, storage, params );
    printf("Object Descriptors: %d\n", objectDescriptors->total);
    cvExtractSURF( image, 0, &imageKeypoints, &imageDescriptors, storage, params );
    printf("Image Descriptors: %d\n", imageDescriptors->total);
    tt = (double)cvGetTickCount() - tt;
    printf( "Extraction time = %gms\n", tt/(cvGetTickFrequency()*1000.));
    CvPoint src_corners[4] = {{0,0}, {object->width,0}, {object->width, object->height}, {0, object-
>height}};
    CvPoint dst_corners[4];
    IplImage* correspond = cvCreateImage( cvSize(image->width, object->height+image->height), 8, 1 );
    cvSetImageROI( correspond, cvRect( 0, 0, object->width, object->height ) );
    cvCopy( object, correspond );
    cvSetImageROI( correspond, cvRect( 0, object->height, correspond->width, correspond->height ) );
    cvCopy( image, correspond );
    cvResetImageROI( correspond );



#ifdef USE_FLANN
    printf("Using approximate nearest neighbor search\n");
#endif

    if( locatePlanarObject( objectKeypoints, objectDescriptors, imageKeypoints,
        imageDescriptors, src_corners, dst_corners ))
    {
        for( i = 0; i < 4; i++ )
        {
            CvPoint r1 = dst_corners[i%4];
            CvPoint r2 = dst_corners[(i+1)%4];
            cvLine( correspond, cvPoint(r1.x, r1.y+object->height ),
                cvPoint(r2.x, r2.y+object->height ), colors[8] );
        }
    }
    vector<int> ptpairs;
#ifdef USE_FLANN
```

```
    flannFindPairs( objectKeypoints, objectDescriptors, imageKeypoints, imageDescriptors, ptpairs );
#else
    findPairs( objectKeypoints, objectDescriptors, imageKeypoints, imageDescriptors, ptpairs );
#endif
    for( i = 0; i < (int)ptpairs.size(); i += 2 )
    {
        CvSURFPoint* r1 = (CvSURFPoint*)cvGetSeqElem( objectKeypoints, ptpairs[i] );
        CvSURFPoint* r2 = (CvSURFPoint*)cvGetSeqElem( imageKeypoints, ptpairs[i+1] );
        cvLine( correspond, cvPointFrom32f(r1->pt),
            cvPoint(cvRound(r2->pt.x), cvRound(r2->pt.y+object->height)), colors[8] );
    }

    cvShowImage( "Object Correspond", correspond );
    for( i = 0; i < objectKeypoints->total; i++ )
    {
        CvSURFPoint* r = (CvSURFPoint*)cvGetSeqElem( objectKeypoints, i );
        CvPoint center;
        int radius;
        center.x = cvRound(r->pt.x);
        center.y = cvRound(r->pt.y);
        radius = cvRound(r->size*1.2/9.*2);
        cvCircle( object_color, center, radius, colors[0], 1, 8, 0 );
    }
    cvShowImage( "Object", object_color );

    cvWaitKey(0);

    cvDestroyWindow("Object");
    cvDestroyWindow("Object SURF");
    cvDestroyWindow("Object Correspond");

    return 0;
}
```

CalculoErrores.h

```
#include <cv.h>
#include <highgui.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>

#include <iostream>
#include <vector>

using namespace std;


// define whether to use approximate nearest-neighbor search
#define USE_FLANN



typedef struct Posicion

{
 int x, y;
 double phi;

};

class Comparador
{
private:
	IplImage* oldImage;
	int ancho, alto;

public:
	Comparador(IplImage* init);

	int calcular_errores(IplImage* entradaOld, IplImage* entrada);

};
```

Comparador.cpp

```
/*************************************************************************************
******
*	ARCHIVO:	Comparador.cpp
*	AUTOR:		Ana Paula Mateo Garrido
*	FUNCION:	Comparador class implementation
*************************************************************************************
*****/
#include "Comparador.h"

Comparador::Comparador(IplImage* init)
{
	storage = cvCreateMemStorage(0);
	centro = cvPoint(init->width/2,init->height/2);
	ancho=init->width*0.8;
	alto=init->height*0.8;
	rect = cvRect(centro.x-ancho/2,centro.y-alto/2,ancho,alto);
	oldImage = cvCloneImage(init);
	actualImage = cvCloneImage(init);
	compensada = cvCloneImage(init);

	// Declaration of rotation and traslation matrix and initialize the references
	m_rot= cvCreateMat(2, 3, CV_64FC1);
	m_tras= cvCreateMat(2, 3, CV_64FC1);
	centro32f = cvPoint2D32f(actualImage->width/2.0, actualImage->height/2.0);  // Centre point of
rotation
	a[0] = cvPoint2D32f(0,0);				//Reference points for rotation
	a[1] = cvPoint2D32f(actualImage->width,0);
	a[2] = cvPoint2D32f(0,actualImage->height);

	// Values for proportional controller
	K.phi = 1.0;
	K.x = 0.5;
	K.y = 0.7;

	// Value of angle threshold
	umbralPhi = 30;
}

Comparador::~Comparador()
{
	cvReleaseImage(&actualImage);
	cvReleaseImage(&oldImage);
	cvReleaseImage(&compensada);
	cvReleaseMat(&m_rot);
	cvReleaseMat(&m_tras);
	cvReleaseMemStorage(&storage);
}

// Compare descriptors to evaluate the matching
double Comparador::compareSURFDescriptors( const float* d1, const float* d2, double best, int length )
{
   double total_cost = 0;
   assert( length % 4 == 0 );
   for( int i = 0; i < length; i += 4 )
   {
      double t0 = d1[i] - d2[i];
      double t1 = d1[i+1] - d2[i+1];
```

```
        double t2 = d1[i+2] - d2[i+2];
        double t3 = d1[i+3] - d2[i+3];
        total_cost += t0*t0 + t1*t1 + t2*t2 + t3*t3;
        if( total_cost > best )
            break;
    }
    return total_cost;
}

// Stablish matching between points in both secuences.
int Comparador::naiveNearestNeighbor(      const float* vec, int laplacian,
                                                                        const CvSeq*
model_keypoints,
                                                                        const CvSeq*
model_descriptors )
{
    int length = (int)(model_descriptors->elem_size/sizeof(float));
    int i, neighbor = -1;
    double d, dist1 = 1e6, dist2 = 1e6;
    CvSeqReader reader, kreader;
    cvStartReadSeq( model_keypoints, &kreader, 0 );
    cvStartReadSeq( model_descriptors, &reader, 0 );

    for( i = 0; i < model_descriptors->total; i++ )
    {
        const CvSURFPoint* kp = (const CvSURFPoint*)kreader.ptr;
        const float* mvec = (const float*)reader.ptr;
            CV_NEXT_SEQ_ELEM( kreader.seq->elem_size, kreader );
        CV_NEXT_SEQ_ELEM( reader.seq->elem_size, reader );
        if( laplacian != kp->laplacian )
            continue;
        d = compareSURFDescriptors( vec, mvec, dist2, length );
        if( d < dist1 )
        {
            dist2 = dist1;
            dist1 = d;
            neighbor = i;
        }
        else if ( d < dist2 )
            dist2 = d;
    }
    if ( dist1 < 0.6*dist2 )
        return neighbor;
    return -1;
}

// Looking for matching points and match
#ifdef USE_FLANN
void Comparador::flannFindPairs( const CvSeq*, const CvSeq* objectDescriptors,
                                                                        const CvSeq*, const
CvSeq* imageDescriptors,
                                                                        vector<int>& ptpairs )
{
            int length = (int)(objectDescriptors->elem_size/sizeof(float));

            cv::Mat m_object(objectDescriptors->total, length, CV_32F);
            cv::Mat m_image(imageDescriptors->total, length, CV_32F);
```

```
		// copy descriptors
		CvSeqReader obj_reader;
		float* obj_ptr = m_object.ptr<float>(0);
		cvStartReadSeq( objectDescriptors, &obj_reader );
		for(int i = 0; i < objectDescriptors->total; i++ )
		{
			const float* descriptor = (const float*)obj_reader.ptr;
			CV_NEXT_SEQ_ELEM( obj_reader.seq->elem_size, obj_reader );
			memcpy(obj_ptr, descriptor, length*sizeof(float));
			obj_ptr += length;
		}
		CvSeqReader img_reader;
		float* img_ptr = m_image.ptr<float>(0);
		cvStartReadSeq( imageDescriptors, &img_reader );
		for(int i = 0; i < imageDescriptors->total; i++ )
		{
			const float* descriptor = (const float*)img_reader.ptr;
			CV_NEXT_SEQ_ELEM( img_reader.seq->elem_size, img_reader );
			memcpy(img_ptr, descriptor, length*sizeof(float));
			img_ptr += length;
		}

		// find nearest neighbors using FLANN
		cv::Mat m_indices(objectDescriptors->total, 2, CV_32S);
		cv::Mat m_dists(objectDescriptors->total, 2, CV_32F);
		cv::flann::Index flann_index(m_image, cv::flann::KDTreeIndexParams(4));  // using 4
randomized kdtrees
		flann_index.knnSearch(m_object, m_indices, m_dists, 2, cv::flann::SearchParams(64) ); //
maximum number of leafs checked

		int* indices_ptr = m_indices.ptr<int>(0);
		float* dists_ptr = m_dists.ptr<float>(0);
		for (int i=0;i<m_indices.rows;++i) {
			if (dists_ptr[2*i]<0.6*dists_ptr[2*i+1]) {
				ptpairs.push_back(i);
				ptpairs.push_back(indices_ptr[2*i]);
			}
		}
}
#else
void Comparador::findPairs(	const CvSeq* objectKeypoints, const CvSeq* objectDescriptors,
					const CvSeq* imageKeypoints, const
CvSeq* imageDescriptors,
					vector<int>& ptpairs )
{
		int i;
		CvSeqReader reader, kreader;
		cvStartReadSeq( objectKeypoints, &kreader );
		cvStartReadSeq( objectDescriptors, &reader );
		ptpairs.clear();

		for( i = 0; i < objectDescriptors->total; i++ )
		{
			const CvSURFPoint* kp = (const CvSURFPoint*)kreader.ptr;
			const float* descriptor = (const float*)reader.ptr;
			CV_NEXT_SEQ_ELEM( kreader.seq->elem_size, kreader );
			CV_NEXT_SEQ_ELEM( reader.seq->elem_size, reader );
```

```
                int nearest_neighbor = naiveNearestNeighbor( descriptor, kp->laplacian,
imageKeypoints, imageDescriptors );
                if( nearest_neighbor >= 0 )
                {
                        ptpairs.push_back(i);
                        ptpairs.push_back(nearest_neighbor);
                }
        }
}
#endif

// Search planar objects in points secuences
int Comparador:: locatePlanarObject( const CvSeq* objectKeypoints, const CvSeq* objectDescriptors,
                        const CvSeq* imageKeypoints, const CvSeq* imageDescriptors,
                        const CvPoint src_corners[4], CvPoint dst_corners[4] )
{
   double h[9];
   CvMat _h = cvMat(3, 3, CV_64F, h);
   vector<int> ptpairs;
   vector<CvPoint2D32f> pt1, pt2;
   CvMat _pt1, _pt2;
   int i, n;

#ifdef USE_FLANN
   flannFindPairs( objectKeypoints, objectDescriptors, imageKeypoints, imageDescriptors, ptpairs );
#else
   findPairs( objectKeypoints, objectDescriptors, imageKeypoints, imageDescriptors, ptpairs );
#endif

   n = ptpairs.size()/2;
   if( n < 4 )
      return 0;

   pt1.resize(n);
   pt2.resize(n);
   for( i = 0; i < n; i++ )
   {
      pt1[i] = ((CvSURFPoint*)cvGetSeqElem(objectKeypoints,ptpairs[i*2]))->pt;
      pt2[i] = ((CvSURFPoint*)cvGetSeqElem(imageKeypoints,ptpairs[i*2+1]))->pt;
   }

   _pt1 = cvMat(1, n, CV_32FC2, &pt1[0] );
   _pt2 = cvMat(1, n, CV_32FC2, &pt2[0] );
   if( !cvFindHomography( &_pt1, &_pt2, &_h, CV_RANSAC, 5 ))
      return 0;

   for( i = 0; i < 4; i++ )
   {
      double x = src_corners[i].x, y = src_corners[i].y;
      double Z = 1./(h[6]*x + h[7]*y + h[8]);
      double X = (h[0]*x + h[1]*y + h[2])*Z;
      double Y = (h[3]*x + h[4]*y + h[5])*Z;
      dst_corners[i] = cvPoint(cvRound(X), cvRound(Y));
   }

   return 1;
}
```

```
// Locate the region defined by the interest rectangle in “oldImage” and fix it the new image.
// Calculate the position and the 2D relative orientation between oldImage and the new one.
int Comparador::calcular_errores()
{
	IplImage* image = cvCreateImage(cvGetSize(actualImage),actualImage->depth,1);	// 
Creating an Imagen in Gray Scale
	cvCvtColor(actualImage,image,CV_BGR2GRAY);  // Conversion in gray scale

   cvSetImageROI(oldImage,  rect );
   IplImage* oldRegion = cvCreateImage(cvGetSize(oldImage),oldImage->depth,oldImage->nChannels);
	IplImage* object = cvCreateImage(cvGetSize(oldImage),oldImage->depth,1);
   cvCopy(oldImage,oldRegion,NULL);		cvCvtColor(oldRegion,object,CV_BGR2GRAY);
	cvReleaseImage(&oldRegion);
   cvResetImageROI(oldImage);
	// Inicialization of points secuences
   CvSeq *objectKeypoints = 0, *objectDescriptors = 0;
   CvSeq *imageKeypoints = 0, *imageDescriptors = 0;

   CvSURFParams params = cvSURFParams(500, 1);		// Looking for 500 interest points

	// Stracting point secuences in each image
   cvExtractSURF( object, 0, &objectKeypoints, &objectDescriptors, storage, params );
   cvExtractSURF( image, 0, &imageKeypoints, &imageDescriptors, storage, params );

	if( objectDescriptors->total > 2 && imageDescriptors->total > 2 )
	{
		// Inicialization of rectangle corners
		CvPoint src_corners[4] = {{0,0}, {object->width,0}, {object->width, object->height}, 
{0, object->height}};
		CvPoint dst_corners[4] = 
{src_corners[0],src_corners[1],src_corners[2],src_corners[3]};

		if(	locatePlanarObject( objectKeypoints, objectDescriptors, imageKeypoints,
			imageDescriptors, src_corners, dst_corners ))
		{	// If an plane object is founded calculate the position error and the orientation 
from it vertex

			double vector[4]={0,0,0,0};
			double sumaX=0;
			double sumaY=0;

			for( int i = 0; i < 4; i++ )
			{
				CvPoint r1 = dst_corners[i%4];
				CvPoint r2 = dst_corners[(i+1)%4];

				sumaX+=r1.x;
				sumaY+=r1.y;
				vector[i]=calcular_angulo(r1);
			}
			pos.phi= devuelveGrados((vector[0]+vector[1]+vector[2]+vector[3])/4.0);
			pos.x = (sumaX/4.0) - centro.x;
			pos.y = (sumaY/4.0) - centro.y;
		}
		else
		{
			pos.phi = 0;
			pos.x = 0;
			pos.y = 0;
```

```
                }
        }
        else
        {
                pos.phi = 0;
                pos.x = 0;
                pos.y = 0;
        }

        // Free memory
        cvReleaseImage(&object);
        cvReleaseImage(&image);

        // Clear memory storage
        cvClearMemStorage(storage);
   return 0;
}


double Comparador::calcular_angulo(CvPoint p)
{
        double phi= atan2((float)(p.y-centro.y),(float)(p.x-centro.x));
        return phi;
}

void Comparador::escribePos()
{
        printf("\n +++ angulo->%f, pos-> %f, %f\n", pos.phi,pos.x,pos.y);
}

char* Comparador::getInformacion()
{
        sprintf(cad,"PHI: %f | X: %f | Y: %f",pos.phi,pos.x,pos.y);
        return cad;
}

void Comparador::compensarAngulo(IplImage* input)
{
        // Create rotation matrix with phi angle calulated
        cv2DRotationMatrix(centro32f, pos.phi*K.phi, 1.0, m_rot);   // Calculate rotation matrix
        cvWarpAffine(input, compensada, m_rot);              // Rotation the matrix with this matrix
}

void Comparador::compensarPosicion(IplImage* input)
{
        for( int i = 0; i < 3; i++ )
        {
                b[i].x = a[i].x - pos.x*K.x;
                b[i].y = a[i].y - pos.y*K.y;
        }
        cvGetAffineTransform( a, b, m_tras );                   // Calculation rtraslation matrix
        cvWarpAffine(input, compensada, m_tras );// Transformation with this matrix
}

void Comparador::compensarAngulo()
{
        compensarAngulo(actualImage);
}
```

```
void Comparador::compensarPosicion()
{
	compensarPosicion(actualImage);
}

void Comparador::estabilizarImagen(IplImage *input)
{
	setActualImage(input);		//current image stablished
	calcular_errores();			// Calculate the relative position from previous
image


	if( pos.phi < umbralPhi )
	{
		compensarAngulo();
		// feedback: puts stabilized image like reference for future comparations

		setOldImage(compensada);	// Stablish reference image

		compensarPosicion(oldImage);
		setOldImage(compensada);
	}
	else
		setOldImage(input);
}

// Method to return the region limited by r in input image in output image
void Comparador::devuelveRegion(IplImage *input, IplImage *output, CvRect r)
{

	CvPoint2D32f a[4],b[4];

	// origin points in warp
	a[0].x = r.x;				a[0].y = r.y;
	a[1].x = r.x + r.width;		a[1].y = r.y;
	a[2].x = r.x + r.width;		a[2].y = r.y + r.height;
	a[3].x = r.x;				a[3].y = r.y + r.height;

	// destination points in warp
	b[0].x = 0;					b[0].y = 0;
	b[1].x = output->width;		b[1].y = 0;
	b[2].x = output->width;		b[2].y = output->height;
	b[3].x = 0;					b[3].y = output->height;

	CvMat* t = cvCreateMat(3, 3, CV_64FC1);// transformation matrix

	cvGetPerspectiveTransform(a,b,t);	// calculate transform matrix
	cvWarpPerspective(input,output,t);	// Makes perspective transformation

	cvReleaseMat( &t );		// Free memory
}
```

Comparador.h

```
/*************************************************************************************
******
*	ARCHIVO:	Comparador.h
*	AUTOR:		Ana Paula Mateo Garrido
*	FUNCION:	Declaration of Comparador class wich permit to calculate the position and the
orientation between consecutive images
*************************************************************************************
*****/
#include <cv.h>
#include <highgui.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <iostream>
#include <vector>

using namespace std;


//#define USE_FLANN
#ifndef M_PI
	#define M_PI 3.14159265358979323846
#endif

//Data structure correction
struct Posicion
{
	double x, y;
	double phi;
};

// comparador class:Making the comparation between consecutiv images, calculate and keep angle error,
x and y.
class Comparador
{
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// declaration of variables
private:
	IplImage *actualImage, *oldImage;	// current image and previous image wich we do the
comparation
	IplImage *compensada;			// contasated image
	CvMemStorage* storage;			// memory storage to features points
	CvPoint centro;				// Centre of the rectangle stablished
for the comparation
	CvRect rect;				// Interest rectangle for the
comparation
	int ancho, alto;			//comparation rectangle Weight and heigh
	Posicion pos;				//relative position detected between
actual and previous images
	char cad[50];				//Relative position and orientation
in text mode
	CvMat *m_rot, *m_tras;			// Rotation and traslation matrix
	CvPoint2D32f centro32f, a[3], b[3];	// Reference points for rotation and traslation
```

```
        Posicion K;                                                     // Gain of proportional
controller
        int umbralPhi;                                          // Phi threshold
//-------------------------------------------------------------------------------------------

//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++
// class public methods
public:

                Comparador(IplImage* init);
        ~Comparador();
        //*************************************/

        Posicion getPos(){return pos;}
        char* getInformacion();
        IplImage* getOldImage(){return oldImage;}                       //returns the used image to
compare
        IplImage* getActualImage(){return actualImage;}         // returns the current image.
        IplImage* getImagenCompensada(){return compensada;}     // returns stabilized image.
        Posicion getGanancias(){return K;}                      // return proportional controller used gains.
        int getUmbralPhi(){return umbralPhi;}       // returns used threshold angle.

        void setOldImage(IplImage* old){oldImage = cvCloneImage(old);}
        // Stablished referenced image.
        void setActualImage(IplImage* actual){actualImage = cvCloneImage(actual);}// Stablished
current image.
        void setCentro(CvPoint c){centro = c;}      //Stablished the comparation ROI centre
        void setDimensiones(int w, int h){alto=h; ancho=w;}             // stablished ROI
dimensions.
        void setGanancias(double kphi, double kx, double ky)        // stablished proportional controller
gains.
        { K.phi = kphi; K.x = kx; K.y = ky; }
        void setUmbralPhi(int umbral){umbralPhi = umbral;}              // stablished angle
threshold.

        void escribePos();          //Write in the screen relative position
        //*************************************************************************************
************************/


        int calcular_errores();     // Calculate image angle and relative position and orientation between
images.
        void compensarAngulo();             // Correct the angle in the current image.
        void compensarPosicion();           // Correct the position in the current image.
        void compensarAngulo(IplImage* input);              // correct the angle in input image.
        void compensarPosicion(IplImage* input);  // Correct the angle in input image.

        void estabilizarImagen(IplImage* input);    // calculate and make the stabilization in images with
the values ok proportional controllers stablished.

        void devuelveRegion(IplImage* input, IplImage* output, CvRect r);   // Returns the input image
region
//-------------------------------------------------------------------------------------------

//private methods class
private:
```

```
        double calcular_angulo(CvPoint p);          // returns the angle formed by the line joining p and
the centre
        double devuelveGrados(double phi){return phi*180/M_PI;}   // returns the angle value in
degrees
        // Comparation of points descriptors to evaluate the correspondence.
        double compareSURFDescriptors( const float* d1, const float* d2, double best, int length );

        // stablished the correspondence between two image points.
        int naiveNearestNeighbor(       const float* vec, int laplacian, const CvSeq*
model_keypoints,
                                                                const CvSeq* model_descriptors );


        // Looking for correspond points pairs
#ifdef USE_FLANN
        void flannFindPairs(    const CvSeq*, const CvSeq* objectDescriptors,
                                                        const CvSeq*, const CvSeq*
imageDescriptors,
                                                        vector<int>& ptpairs );
#else
        void findPairs( const CvSeq* objectKeypoints, const CvSeq* objectDescriptors,
                                                const CvSeq* imageKeypoints, const CvSeq*
imageDescriptors,
                                                vector<int>& ptpairs );
#endif
        //Identify objects between correspond points secuence
        int locatePlanarObject( const CvSeq* objectKeypoints, const CvSeq* objectDescriptors,
                                                        const CvSeq* imageKeypoints, const
CvSeq* imageDescriptors,
                        const CvPoint src_corners[4], CvPoint dst_corners[4] );
};
```

```
/*************************************************************************************
******
*       ARCHIVO:        Principal.h
*       AUTOR:                  Ana Paula Mateo Garrido
*       FUNCION:        Implement main function
*************************************************************************************
*****/
#include "Comparador.h"

int main()
{
        // capture object creation and inicialitation   CvCapture* capture = cvCreateCameraCapture( 0 );
                // Webcam
        //CvCapture* capture = cvCreateFileCapture("out-6N.avi");   // Video

        IplImage *image, *muestra;          // declaratios of images used
```

```
	image = cvQueryFrame(capture);		// Takes an image to calibrate and initialized

		muestra = cvCreateImage(cvGetSize(image),8,3);
	CvRect rec = cvRect(image->width*0.1,image->height*0.1,image->width*0.8,image-
>height*0.8);

	Comparador compara = Comparador(image);		//comparador creation and initialization

	// window creation
	cvNamedWindow("ORIGINAL");
	cvNamedWindow("ESTABILIZADA");
	cvNamedWindow("REGION");


	while(true)
	{
		image = cvQueryFrame(capture);		// Takes a camera frame
		compara.estabilizarImagen(image);	// Making the stabilization.
		compara.devuelveRegion(compara.getImagenCompensada(),muestra,rec);

		//Shows the current image, the position and the orientation calculated.

	cvPutText(image,compara.getInformacion(),cvPoint(20,20),&cvFont(0.5),CV_RGB(255,0,100))
;

	cvPutText(image,compara.getInformacion(),cvPoint(21,21),&cvFont(0.5),CV_RGB(255,100,20
0));

		cvShowImage("ORIGINAL",image);
		cvShowImage("ESTABILIZADA",compara.getImagenCompensada());
		cvShowImage("REGION",muestra);

		// If the user press ESC the aplication finish.
		char c = cvWaitKey(33);
		if(c==27)
			break;

	}


	// free memory
	if( capture )
		cvReleaseCapture( &capture );
	//if( image )
	//	cvReleaseImage(&image);
	if( muestra )
		cvReleaseImage(&muestra);

	cvDestroyAllWindows();
}
```