CIMAT Centro de Investigación en Matemáticas, A.C.

AUTO-CALIBRACIÓN DE RASTREADOR OCULAR CON ANÁLISIS DE PATRONES Y TÉCNICAS PROBABILISTAS

T E S I S

Que para obtener el grado de

Maestro en Ingeniería de Software

Presenta

ISC. Carlos Alberto Pinedo García

Director de Tesis:

Dr. Carlos Alberto Lara Álvarez

Autorización de la versión

Zacatecas, Zac., 2 de febrero de 2018

Agradecimientos

A mi familia por todo el apoyo que me han brindado durante mis estudios.

A mi director de tesis Dr. Carlos Alberto Lara Álvarez por todos los conocimientos compartidos durante mi trabajo de investigación.

De igual manera, al Centro de Investigación en Matemáticas Unidad Zacatecas por el apoyo otorgado para mi estancia de investigación.

Finalmente, al Consejo Nacional de Ciencia y Tecnología (CONACYT) por el financiamiento mediante la beca de posgrado para la realización de estudios.

Resumen

El rastreo ocular es una técnica que mide los movimientos oculares para que un investigador sepa a donde está mirando una persona en un momento dado y la secuencia con la que sus ojos están cambiando de un lugar a otro. Es importante que un rastreador ocular esté calibrado para asegurar que las entradas y salidas están configuradas de forma óptima, y así evitar muchos problemas desde degradación del desempeño e incremento en los errores. En los rastreadores oculares sin compensación los movimientos de la cabeza del participante causan que la calibración deje de ser aceptable. En el mercado existen rastreadores oculares que proporcionan compensación del movimiento de cabeza; sin embargo, tienen un costo mayor. En ésta tesis se propone un método que aprovecha la lectura de un texto para ocultar el proceso de calibración; además, se utiliza calibración incremental y se comprueba si la calibración no excede un margen determinado, en caso contrario, el método propuesto es capaz de re-calibrar el rastreador ocular usando los datos que se obtienen de la lectura de texto y la interacción del usuario con el sistema. Para realizar el proceso de calibración se resuelve un problema de mínimos cuadrados usando factorización QR y rotaciones de Givens, de este modo se pueden añadir restricciones al sistema sin resolver todo el problema cada vez, reduciendo así el costo computacional. Se llevó a cabo un experimento para cuantificar el error en la calibración a través de las diferentes iteraciones. Al usar una resolución de $1440 \times 900 pxeles$ la calibración inicial con la técnica propuesta muestra errores absolutos en x de media$(e_x) = 278.50$píxeles y errores absolutos en y de media$(e_y) = 52.22$píxeles; es decir,la calibración es más eficiente en el eje vertical que en el eje horizontal. En iteraciones posteriores se observa una reducción del error en la calibración a media$(e_x) = 139.92$píxeles y media$(e_y) = 26.95$píxeles.

Palabras clave: rastreador ocular, calibración, compensación de movimientos de cabeza, incremental, factorización QR, rotaciones de Givens.

Índice general

Índice de figuras

Índice de tablas

1. Introducción

El rastreo ocular es una técnica mediante la cual se miden los movimientos oculares para que un investigador sepa a donde está mirando una persona en un momento dado y la secuencia con la que sus ojos están cambiando de un lugar a otro. El seguimiento de los movimientos oculares de las personas puede ayudar a los investigadores de Interacción Humano-Computadora (HCI del inglés *Human–Computer Interaction*) a entender el procesamiento de información visual y los factores que pueden afectar la usabilidad de las interfaces del sistema (Poole and Ball, 2005). Duchowski (2007) propone la siguiente taxonomía tecnológica para dispositivos de rastreo ocular:

1. Primera generación: medición directa del ojo (montado en la cabeza) que consiste en técnicas como lente de contacto escleral / bobina de búsqueda, electrooculograma.
2. Segunda generación: foto y video-oculografía.
3. Tercera generación: reflexión combinada basada en vídeo analógico / reflejo corneal.
4. Cuarta generación: reflexión combinada de pupila / córnea basada en vídeo digital, aumentada por técnicas de visión por computadora y Procesadores de Señal Digital (DSP del inglés *Digital Signal Processor*).

Debido al aumento en la capacidad de cómputo y a las mejoras de las técnicas de visión por computadora, los fabricantes de rastreadores oculares están desarrollando dispositivos que por lo general caen dentro de la cuarta generación, debido a que se utiliza una cámara; además, el seguimiento ocular puede realizarse remotamente (lo que significa que la cámara está colocada en algún lugar al frente del participante), o puede hacerse montado a la cabeza (lo que significa que está por debajo del eje visual del ojo, generalmente en el marco de unos lentes).

Ambos tipos de rastreadores oculares, remotos o montados en la cabeza, tienen dos inconvenientes importantes si se van a utilizar en sistemas HCI:

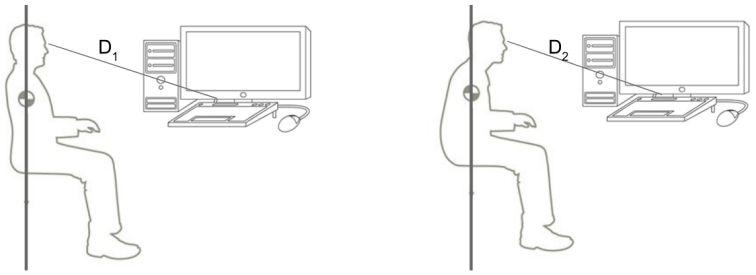


Figura 1.1: Distancia D_1, es diferente a la distancia D_2 cuando el participante se acerca a la pantalla lo que ocasiona un desfase en los datos obtenidos por el rastreador ocular.

- El cambio de posición de cabeza (Fig. 1.1). Esto se puede resolver para los rastreadores remotos usando dos cámaras estéreo o una cámara gran angular para buscar a la persona delante de ella y otra para apuntar la cara de la persona. Se necesitan características como la orientación 3D cara de la cara del sujeto y distancia para compensar el movimiento de la cabeza (Lupu and Ungureanu, 2013).
- Deben ser calibrados para cada individuo (Zhu and Ji, 2007).

Dentro de la cuarta generación de rastreadores oculares, se pueden usar dos tipos de imágenes, en el espectro visible y en el espectro infrarrojo. Debido al uso de técnicas de visión por computadora, es posible usar fuentes de video comunes como lo son las cámaras integradas en teléfonos celulares y computadoras portátiles, también es posible el uso de dispositivos de rastreo ocular de bajo costo, con el inconveniente de que no se tiene el hardware necesario para realizar la compensación de los movimientos de cabeza, lo cual conlleva inconvenientes como los que se mencionan a continuación.

1.1 Antecedentes

En el año 2015, el Centro de Investigación en Matemáticas unidad Zacatecas (CIMAT) adquiere dos rastreadores oculares, con el objetivo de utilizarlos en las investigaciones de:

1. Evaluación de Objetos de Aprendizaje a través de Seguimiento Ocular (Mitre-Hernandez et al, 2016; Alvarado Hernández, 2016; Lara-Alvarez et al, 2016).
2. Evaluación de Carga Cognitiva en Videojuegos con Seguimiento Ocular (Covarrubias, 2016).

En estas investigaciones se observó que, aunque el dispositivo entregaba datos útiles, al carecer de compensación de movimiento de cabeza, cualquier cambio en la posición del participante provocaba

Figura 1.2: Error en las fijaciones producido por el movimiento de la cabeza.

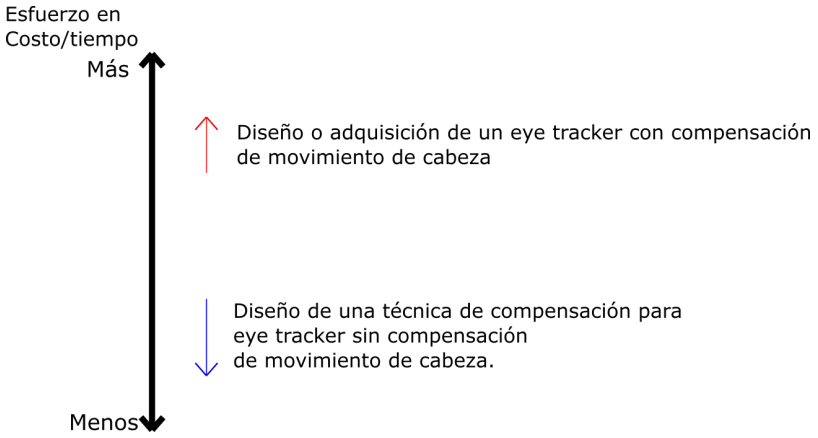


Figura 1.3: Relación costo/esfuerzo para resolver el problema de compensación de movimientos de cabeza.

un error en los datos como el que se observa en la Fig. 1.2. Para compensar dicho error Covarrubias (2016) propone una técnica que usa la distancia de la cabeza al sensor medida con un sensor ultrasónico.

1.2 Motivación

Durante un experimento de "*eye tracking*" es importante adaptarse a los movimientos de la cabeza del participante, ya que estos causan que la calibración requerida deje de ser válida. En el mercado existen rastreadores oculares que proporcionan compensación del movimiento de cabeza; sin embargo, tienen un costo mayor ya sea en precio o tiempo que se les debe invertir.

El incremento en el uso de computadoras portátiles y teléfonos inteligentes, permitiría utilizar sus cámaras como rastreadores oculares (Cheng et al, 2014), pero carecen del hardware necesario para realizar la compensación de movimientos de cabeza; existen propuestas de software que permiten

Figura 1.4: Sistema de sujeción de barbilla (Fixxl Ltd., 2017).

realizar esta compensación (Krafka et al, 2016). Las soluciones sin compensación de movimiento de cabeza tienden a ser más económicas (Fig. 1.3), pero requieren al participante permanecer en la misma posición por la duración del experimento, para reducir este problema se han usado sistemas de sujeción de barbilla (Fig. 1.4), pero es una solución incómoda para el participante y que requiere destinar un espacio y equipo especializado para el experimento, es por eso que en esta tesis se propone un método de calibración continúa para reducir el error de las mediciones del rastreador.

1.3 Definición del problema

Dadas las mediciones obtenidas por un rastreador ocular basado en video, sin corrección de movimiento de cabeza, el problema consiste en obtener una calibración en tiempo real y re-calibrar cuando existan cambios de posición de la cabeza de un participante.

1.4 Objetivos

Con el propósito de dar solución al problema presentado, este trabajo de tesis propone el siguiente objetivo general:

- Calibrar en tiempo real un rastreador ocular y detectar una calibración no válida.

El cual se descompone en los siguientes objetivos específicos.

- Estudiar los tipos de transformaciones geométricas en 2D.
- Estudiar técnicas de calibración de rastreadores oculares.
- Proponer una técnica de calibración basada en factorización QR y rotaciones de Givens.
- Probar la técnica de calibración con un rastreador ocular.

1.5 Hipótesis

En esta tesis se evalúa la siguiente hipótesis:

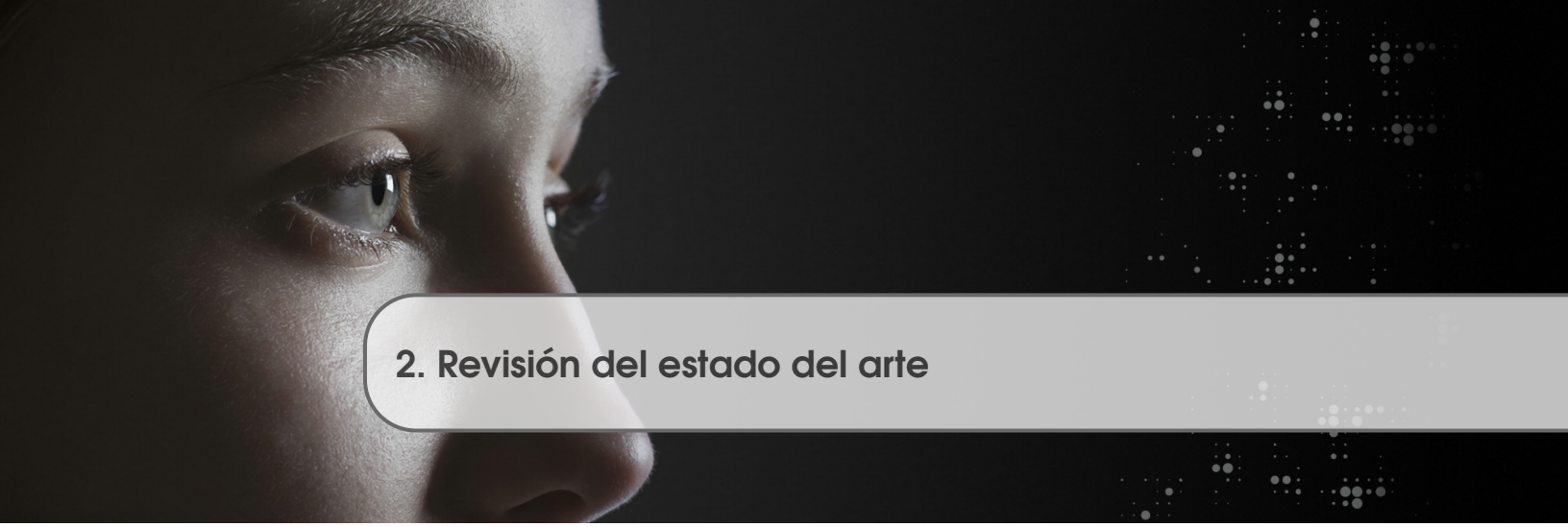
- Se puede generar una técnica de calibración en tiempo real para rastreador ocular basado en video capaz de compensar movimientos de cabeza naturales.

1.6 Alcance y limitaciones

Primero, el rastreador ocular usado es "*The Eye Tribe Tracker*" en su revisión de firmware 293; segundo, se requiere que el usuario lea un texto; tercero, la detección de fijaciones, calibración, actualización y validación de la calibración se deben realizar en tiempo real.

1.7 Organización de la tesis

Este trabajo está organizado de la siguiente manera: El capítulo 2 describe las técnicas de calibración de rastreadores oculares reportadas en la literatura. El capítulo 3 detalla la propuesta realizada. El capítulo 4 aborda las pruebas realizadas y los resultados obtenidos. El capítulo 5 detalla las conclusiones obtenidas y describe el trabajo a futuro.



2. Revisión del estado del arte

Como se menciona en el capítulo 1, el seguimiento ocular es una técnica donde la posición del ojo se usa para determinar la dirección de la mirada de una persona en un momento dado (Poole and Ball, 2005). A través de los años, se han desarrollado diferentes técnicas de seguimiento ocular, de acuerdo con la tecnología disponible a la época.

Emile Java (Oftalmólogo francés, 1839 - 1907) fue uno de los primeros en describir en 1879 los movimientos del ojo durante la lectura de textos. Se observa con la ayuda de un espejo, que los movimientos oculares no son de forma continua a lo largo de la frase, sino compuestos de movimientos rápidos nombrados movimientos sacádicos combinados con paradas cortas nombradas fijaciones (Lupu and Ungureanu, 2013).

La aparición de la computadora personal en los años 80 se comparó con una bocanada de aire para las investigaciones de seguimiento ocular. Ahora, los científicos tienen un instrumento importante para el procesamiento de datos a alta velocidad. También comienzan a investigar cómo el seguimiento de los ojos se puede utilizar para la interacción entre el ser humano y la computadora. Al principio, esto se hizo para ayudar a las personas con discapacidad a tener acceso a la nueva tecnología. Luego, los grupos de marketing vieron la oportunidad de usar el seguimiento de los ojos para mejorar sus anuncios en las revistas, observando qué páginas se leen en realidad y cuántas veces. En el mismo contexto, a principios de los años 90, el rastreador ocular fue utilizado por el analista Joe Theismann de la NFL y una serie de aficionados al fútbol para determinar qué partes de la pantalla eran más vistas y qué partes menos. Debido al éxito de este enfoque, EURO RSCG, la mayor agencia de publicidad y marketing, utilizó la tecnología de seguimiento ocular para evaluar y medir las reacciones a la información de sitios web (Lupu and Ungureanu, 2013).

Como se menciona en el Capitulo 1, los rastreadores oculares basados en video tienen dos

Tabla 2.1: Comparativa de métodos de seguimiento ocular en función de la compensación de movimientos de cabeza y calibración.

		Compensación del movimiento de cabeza	
		SI	NO
Calibración	SI	▪ Cámaras estéreo. ▪ Cámara gran angular y cámara zoom. ▪ Seguimiento de vista en 3D con mapeo en 2D.	▪ Cámara simple. ▪ Montado a la cabeza.
	NO	▪ "*GazeCapture*" con "*iTracker*"	–

principales inconvenientes:

- Los movimientos de cabeza del participante, los cuales generan errores en los datos del rastreador ocular.
- Se deben calibrar para cada participante.

La tabla 2.1 muestra una comparativa entre los métodos de calibración y compensación de movimiento de cabeza, de los cuales se observa que todos los métodos estudiados que requieren de calibración y compensan movimientos de cabeza hacen uso de dos cámaras lo cual incrementa su costo. Este trabajo es útil para dispositivos de seguimiento ocular que requieren calibración pero no compensan los movimientos de cabeza. Ademas la técnica permite detectar cuando una calibración deja de ser válida.

2.1 Métodos de calibración

Como se menciona anteriormente, Duchowski (2007) propone una taxonomía de cuatro generaciones para rastreadores oculares, la tercera y cuarta generación hacen uso de un sistema de puntos distribuidos en la pantalla que el sujeto debe mirar para realizar una calibración. Estos sistemas se diferencian generalmente entre sí por:

- La tercera generación utiliza de cinco a nueve puntos y es controlado por el rastreador ocular.

- La cuarta generación utiliza un número cualquiera de puntos, y es controlado por la aplicación.

Para la cuarta generación de rastreadores oculares generalmente se utilizan nueve puntos (Fig. 2.1) ya que representa un intermedio entre la precisión de la calibración sin volverse cansado para el sujeto (Cheng et al, 2014), desde el punto de vista del usuario, la calibración de los rastreadores oculares ha mejorado considerablemente, siendo la mayor mejora la ausencia de un sistema de sujeción de barbilla.

Figura 2.1: Calibración del rastreador ocular con nueve puntos (The Eye Tribe, 2017).

Flatla et al (2011) hacen hincapié en la importancia del proceso de calibración en sistemas interactivos para asegurar que las entradas y salidas están configuradas de forma óptima, desde degradación del desempeño, incremento en los errores y algunas interacciones que pueden ser imposibles si no se realiza una calibración; sin embargo, estos procesos son tediosos y poco agradables para los participantes, quienes pueden acabar evitándolos del todo. Para dar solución a este problema Flatla et al (2011) proponen "*juegos de calibración*"; los cuales capturan los datos requeridos para la calibración de una manera atractiva y entretenida, para facilitar esto, presentan guías de diseño para mapear tipos comunes de calibración a tareas clave y de allí, a mecánicas de juego bien conocidas.

La propuesta de esta tesis no usa la matriz clásica de nueve (o doce) puntos para calibración de rastreadores oculares (Cheng et al, 2014; Duchowski, 2007; The Eye Tribe, 2017). Tampoco se puede usar un juego de calibración como el sugerido por Flatla et al (2011). Sin embargo, se pueden aprovechar las actividades que el usuario realiza cuando usa objetos de aprendizaje. Álvarez et al (2010) define objeto de aprendizaje como: un recurso de información o software interactivo

utilizado en el aprendizaje online. Una actividad primordial al usar objetos de aprendizaje es la lectura de texto; por este motivo se pensó en usar un patrón base que consiste de líneas de texto complementadas con clics en la pantalla. De tal manera que el proceso de calibración queda oculto al participante y se puede utilizar para re-calibrar el dispositivo cuando se detecta una calibración no válida.

2.2 Métodos de compensación de movimiento de cabeza

Zhu and Ji (2007) proponen dos técnicas de compensación de movimientos de cabeza. La primera es una técnica de seguimiento de la vista en tres dimensiones, que permite estimar el eje óptico de la vista sin necesidad de conocer ningún parámetro dependiente del ojo del sujeto. La segunda es una técnica basada en mapeo de la mirada en dos dimensiones para permitir movimiento libre de cabeza, cuando la cabeza se mueve, la función puede ser actualizada automáticamente.

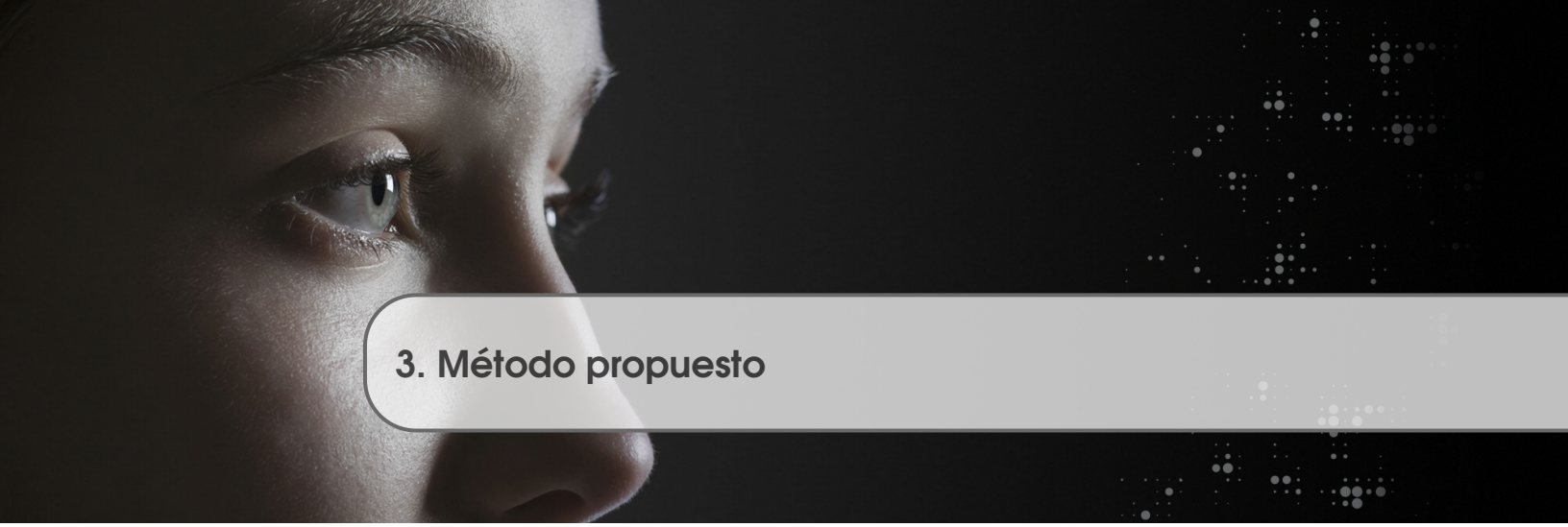
Lupu and Ungureanu (2013) lista el uso de dos cámaras estéreo o una cámara gran angular para buscar a la persona delante de ella y otra para apuntar la cara de la persona debido a que se requieren las características como la orientación en tres dimensiones de la cara del sujeto y distancia para compensar el movimiento de la cabeza.

Krafka et al (2016) proponen un método que hace uso de un conjunto de datos al que nombran "*GazeCapture*" y una red neuronal convolucional llamada "*iTracker*", que no recae en ningún sistema pre-existente de estimación de la posición de la cabeza.

2.3 Resumen de métodos

En la mayoría de métodos analizados se observa que aún requieren un proceso de calibración por participante. Para compensar los movimientos de cabeza, usualmente se requieren técnicas de visión por computadora para detectar las características en tres dimensiones del ojo y el rostro, debido a esto los rastreadores que hacen uso de una sola cámara tienen problemas en la compensación. En el capítulo 3 se propone un método que hace uso de la lectura de un texto para ocultar el proceso de calibración; además, para compensar los movimientos de cabeza se utiliza calibración incremental y válida si el error en la calibración y las posiciones de los datos en pantalla no exceden un margen

determinado, en caso contrario solícita un nuevo proceso de calibración.



3. Método propuesto

3.1 Generalidades del método

Como se describe en el capítulo anterior los métodos de calibración convencionales utilizan una secuencia de puntos para generar una calibración inicial; en los dispositivos de seguimiento ocular con compensación de cabeza o cuando se usa un dispositivo de sujeción de babilla, esa calibración se mantiene por el resto del experimento. Sin embargo, como describe la Fig. 1.1, esta calibración se puede invalidar para sensores sin compensación de movimiento de cabeza. En este capítulo se describe la técnica propuesta que usa la menor cantidad de puntos de fijación posibles para la calibración inicial y re-calibrar en caso de ser necesario.

El diagrama de la Fig. 3.1 ilustra el funcionamiento del método propuesto. En primer lugar, se requiere de un proceso de calibración inicial. En consecuencia, el rastreador ocular se puede usar para realizar mediciones. Con el fin de que la calibración sea válida en todo momento, el algoritmo realiza una validación de calibración de forma periódica. En caso de que la calibración no sea válida, el sistema realiza un proceso de re-calibración.

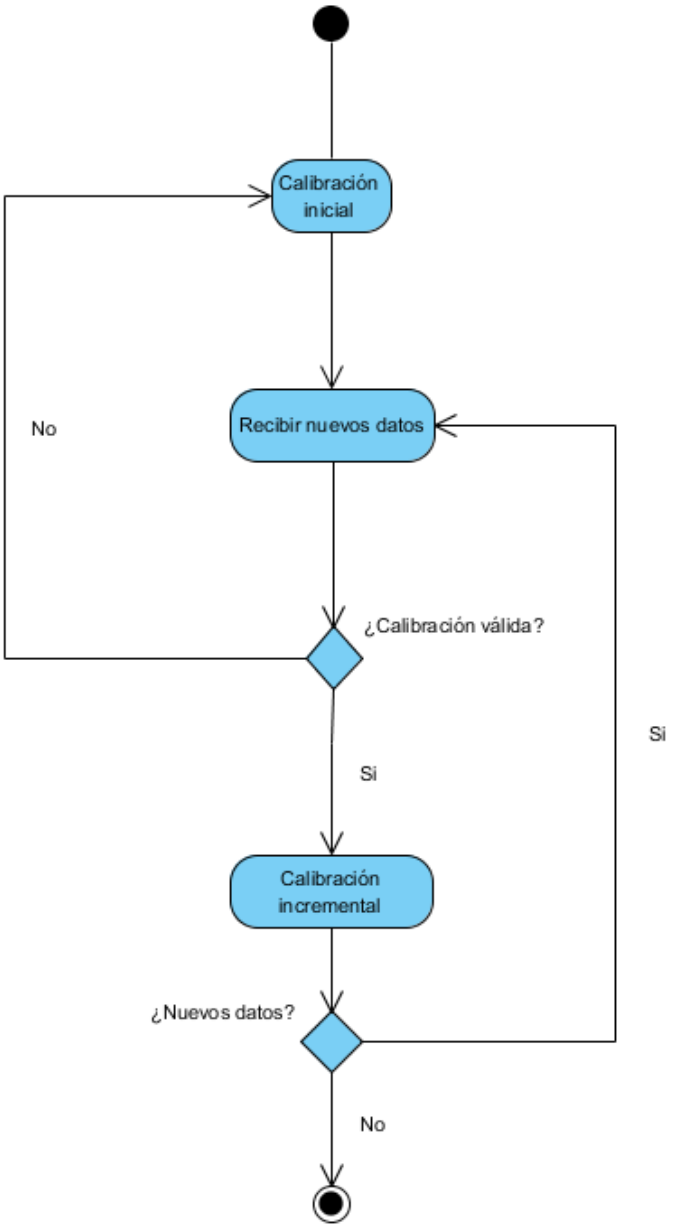


Figura 3.1: Diagrama de actividad que representa el flujo del método.

3.1.1 Patrón de calibración

El método de calibración propuesto utiliza el patrón que se describe en la Fig. 3.2. El patrón asocia los conjuntos $P = \{p_1, p_2\}$ y $L = \{l_1, l_2, l_3\}$ que se muestran en la pantalla con los conjuntos $P' = \{p'_1, p'_2\}$ y $L' = \{l'_1, l'_2, l'_3\}$ que se obtienen por el sensor. Los conjuntos P y P' son puntos, mientras que L y L' son líneas.

Para poder realizar la calibración se requiere un conjunto de puntos de la pantalla asociado a un conjunto de puntos del sensor. Los dos puntos de P asociados con los dos puntos P' no son suficientes para generar un modelo de calibración. Por lo tanto, la técnica propuesta obtiene más puntos a partir las primitivas P, L y P', L' como se describe a continuación:

Puntos de proyección. Estos puntos se obtienen proyectando puntos en líneas; es decir, se obtiene el conjunto P_Q:

$$P_Q = \text{proy}(P,L) = \{q_{ij} = \text{proy}(p_i, l_j) \mid p_i \in P, l_j \in L\}$$

Como se ilustra en la figura 3.2, se obtienen los puntos de proyección $P_Q = \text{proy}(P,L)$ y $P'_Q = \text{proy}(P', L')$

Puntos de intersección. Sea l_{ab} la línea que une los puntos p_1, p_2. El conjunto de puntos de intersección P_R es:

$$P_R = \text{intersect}(P,L) = \{r_i = \text{intersect}(l_{ab}, l_i) \mid l_i \in L\}$$

Como se ilustra en la figura 3.2, se obtienen los puntos de intersección $P_R = \text{intersect}(P,L)$ y $P'_R = \text{intersect}(P', L')$

De tal manera que la calibración se puede realizar con los puntos de la pantalla $P_S = P \cup P_Q \cup P_R$ asociados con los puntos del sensor ocular $P_E = P' \cup P'_Q \cup P'_R$.

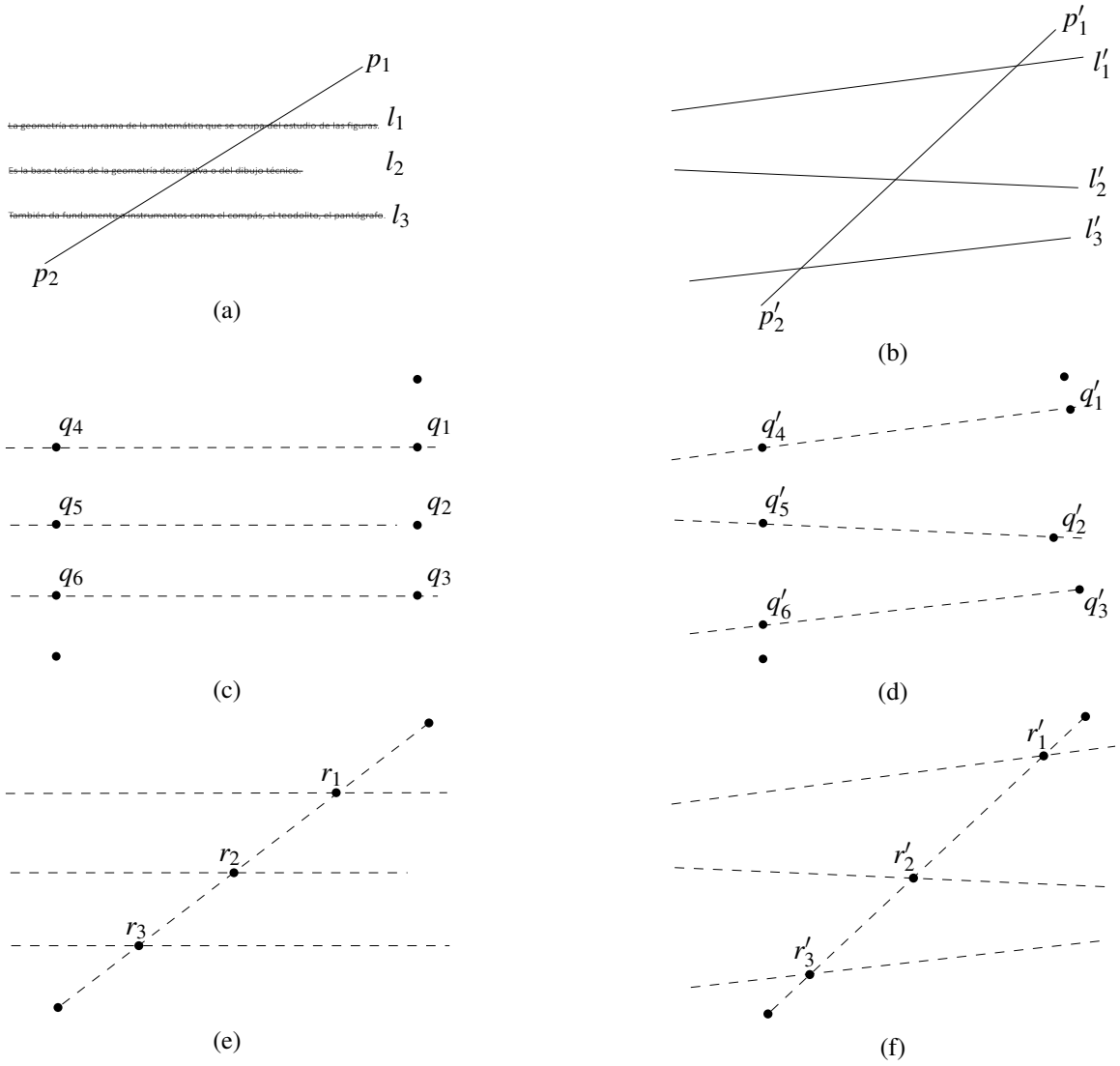


Figura 3.2: Patrón de calibración inicial. (a) Patrón que se obtiene de los puntos de fijación inicial en pantalla $P = \{p_1, p_2\}$ y las lineas de texto $L = \{l_1, l_2, l_3\}$. (b) Patrón que se obtiene de los puntos de fijación inicial del sensor $P' = \{p'_1, p'_2\}$ y las lineas de lectura $L = \{l'_1, l'_2, l'_3\}$. (c) Puntos que se obtienen de la proyección de los puntos de fijación inicial en pantalla a las lineas de texto $P_Q = \text{proy}(P, L)$. (d) Puntos que se obtienen de la proyección de los puntos de fijación inicial del sensor a las lineas de lectura $P'_Q = \text{proy}(P', L')$. (e) Puntos que se obtienen de la intersección entre los puntos de fijación inicial en la pantalla con las lineas de texto $P_R = \text{intersect}(P, L)$. (f) Puntos que se obtienen de la intersección entre los puntos de fijación inicial del sensor con las lineas de lectura $P'_R = \text{intersect}(P', L')$.

3.1.2 Obtención de primitivas de calibración

El proceso inicia mostrando un punto en la esquina superior derecha de la pantalla, espera un clic en la pantalla por parte del participante y solicita las lecturas de fijaciones al controlador del rastreador ocular, después muestra un punto en la parte inferior izquierda de la pantalla y nuevamente espera un clic en la pantalla antes de solicitar las fijaciones al controlador del rastreador ocular, con este proceso se obtienen las fijaciones iniciales que serán requeridas para proyectar las lineas necesarias para calcular los puntos de intersección. Una vez se tienen los puntos de fijación inicial se muestra un texto pequeño que el usuario debe leer (Fig. 3.3), nuevamente se espera por un clic en la pantalla antes de solicitar las lecturas de fijaciones.

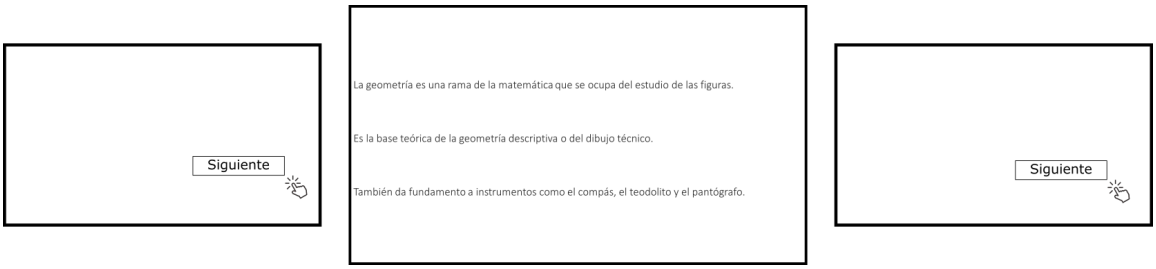


Figura 3.3: De izquierda a derecha: pantalla de espera por un clic, texto que se muestra al participante para calibración inicial, pantalla de espera por un clic.

3.2 Marco teórico

3.2.1 Representación en coordenadas homogéneas

Como se describe en Hartley and Zisserman (2004) una línea en el plano cartesiano está representada por una ecuación tal como $ax+by+c=0$, donde diferentes opciones de a, b y c dan lugar a diferentes líneas. Así, una línea puede ser representada naturalmente por el vector $(a,b,c)^\top$. La correspondencia entre las líneas y los vectores $(a,b,c)^\top$ no es uno a uno, ya que las líneas $ax+by+c=0$ y $(ka)x+(kb)y+(kc)=0$ son las mismas, Para cualquier constante k diferente de 0

.

Un punto $x=(x,y)^\top$ cae en la línea $l=(a,b,c)^\top$ si y solo si $ax+by+c=0$. Esto puede ser escrito en términos del productor interno de vectores que representan el punto como $(x,y,1)(a,b,c)^\top=(x,y,1)l=0$; esto significa que el punto $(x,y)^\top$ en $\mathbb{R}^2$ se representa en tres

dimensiones añadiendo una coordenada final de 1.

3.2.2 Mínimos cuadrados

Mínimos cuadrados es un método muy popular para calcular estimaciones y ajustar datos. Es una de las técnicas más antiguas de la estadística moderna (Abdi, 2003), un sistema de ecuaciones lineales se puede representar en forma matricial como:

$$A\theta = \mathbf{b} \tag{3.1}$$

donde A es la matriz de los coeficientes, $\mathbf{b}$ es la columna de términos constantes, y θ la columna de las incógnitas.

La solución general al problema de mínimos cuadrados es:

$$\begin{aligned} A^\top A\theta &= A^\top \mathbf{b} \\ \theta &= (A^\top A)^{-1} A^\top \mathbf{b} \end{aligned} \tag{3.2}$$

Ejemplo: Calibrar un rastreador ocular

Suponga que tenemos n puntos del sensor $((x_{s_1}, y_{s_1}), \ldots, (x_{s_n}, y_{s_n}))$ y n puntos de la pantalla $((x_{p_1}, y_{p_1}), \ldots, (x_{p_n}, y_{p_n}))$ de forma que:

$$\begin{aligned} (x_{s_1}, y_{s_1}) &\mapsto (x_{p_1}, y_{p_1}) \\ &\vdots \\ (x_{s_n}, y_{s_n}) &\mapsto (x_{p_n}, y_{p_n}) \end{aligned} \tag{3.3}$$

Se desea obtener la calibración con el modelo definido por Cheng et al (2014):

$$\hat{x}_p = a + bx_s + cy_s + dx_s^2 + ey_s^2 + fx_s y_s \tag{3.4}$$

$$\hat{y}_p = g + hx_s + iy_s + jx_s^2 + ky_s^2 + lx_s y_s \tag{3.5}$$

Es decir, para obtener la coordenada de la pantalla (x_p) a partir de una medición (x_s, y_s) se requiere conocer los parámetros $(a, b, \ldots, f)$ y $(g, h, \ldots, l)$ para una coordenada (y_p). En este ejemplo se resuelve únicamente para x_p; sin embargo, siguiendo el mismo proceso se puede dar solución para y_p.

Con las asociaciones (3.3) y el modelo (3.4) se puede definir el siguiente sistema de ecuaciones:

$$\begin{aligned} \hat{x}_{p_1} &= a + bx_{s_1} + cy_{s_1} + dx_{s_1}^2 + ey_{s_1}^2 + fx_{s_1}y_{s_1} \\ &\vdots \\ \hat{x}_{p_n} &= a + bx_{s_n} + cy_{s_n} + dx_{s_n}^2 + ey_{s_n}^2 + fx_{s_n}y_{s_n} \end{aligned} \tag{3.6}$$

Que se puede representar en la forma (3.1) con:

$$A = \begin{bmatrix} 1 & x_{s_1} & y_{s_1} & x_{s_1}^2 & y_{s_1}^2 & x_{s_1}y_{s_1} \\ & & & \vdots & & \\ 1 & x_{s_n} & y_{s_n} & x_{s_n}^2 & y_{s_n}^2 & x_{s_n}y_{s_n} \end{bmatrix}$$

$$\theta_x = [a, b, \ldots, f]^\top$$

$$\mathbf{b} = [\hat{x}_{p_1}, \ldots, \hat{x}_{p_n}]^\top$$

Y se puede resolver con mínimos cuadrados (3.2).

3.2.3 Factorización QR

La factorización QR es otra forma de resolver un problema de mínimos cuadrados. La factorización QR de una matriz es una descomposición de la misma como producto de una matriz ortogonal por una matriz triangular superior (Press et al, 2007).

Si $A \in \mathbb{R}^{m \times n}$ tiene columnas linealmente independientes, entonces se puede factorizar como:

$$A = QR \tag{3.7}$$

donde: Q es de $m \times n$ con columnas ortonormales y R es de $n \times n$, triangular superior, sin elementos 0 en la diagonal.

Con esto en consideración podemos realizar el siguiente despeje para resolver el problema de mínimos cuadrados:

$$\begin{aligned} Ax &= b \\ A^\top Ax &= A^\top b \\ (QR)^\top (QR)x &= (QR)^\top b \\ R^\top Q^\top QRx &= R^\top Q^\top b \\ R^\top Rx &= R^\top Q^\top b \\ Rx &= Q^\top b \end{aligned} \tag{3.8}$$

Entonces se procede a resolver utilizando sustitución hacia atrás.

$$\begin{aligned} x_n &= \frac{b_n}{A_{nn}} \\ x_{n-1} &= \frac{b_{n-1} - A_{n-1,n}\, x_n}{A_{n-1,n-1}} \\ x_{n-2} &= \frac{b_{n-2} - A_{n-2,n-1}\, x_{n-1} - A_{n-2,n}\, x_n}{A_{n-2,n-2}} \\ &\vdots \\ x_1 &= \frac{b_1 - A_{12}x_2 - A_{13}x_3 - \cdots - A_{1n}x_n}{A_{11}} \end{aligned} \tag{3.9}$$

3.2.4 Rotaciones de Givens

La factorización QR también se puede calcular con una serie de rotaciones de Givens (Press et al, 2007). Cada rotación hace cero un elemento en la subdiagonal de la matriz, formando de este modo la matriz R. La concatenación de todas las rotaciones de Givens realizadas, forma la matriz ortogonal Q.

Por ejemplo la matriz de rotación para i,k es:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & & & & \cdots & 0 \\ 0 & \cdots & q_{i,i} & \cdots & q_{i,k} & \cdots & 0 \\ 0 & \vdots & \ddots & & & \cdots & 0 \\ 0 & \cdots & q_{k,i} & \cdots & q_{k,k} & \cdots & 0 \\ 0 & \vdots & \ddots & & & \cdots & 0 \\ 0 & 0 & 0 & \cdots & & \ddots & 1 \end{bmatrix} \tag{3.10}$$

El procedimiento de rotación de Givens es útil en situaciones donde pocos elementos fuera de la diagonal necesitan ser anulados. De esta manera podemos usar una solución rápida para obtener una calibración inicial y rotaciones de Givens cuando se tenga una solución parcial. Es decir cuando un rastreador ocular se re-calibra.

3.3 Descripción del método de calibración seleccionado

3.3.1 Calibración inicial

Para calcular la calibración, se toma el texto que se mostró al participante (Fig. 3.3) y se somete al siguiente proceso (Algoritmo 1):

Una vez se tienen los grupos de centroides y de fijaciones, se utiliza mínimos cuadrados con la ecuación para las rectas $y = mx + b$ para obtener una linea que se ajusta a el grupo de centroides y otra que ajusta al grupo de fijaciones, a partir de estas lineas y los puntos de fijación, se calculan los puntos de intersección usando la linea ajustada a los centroides en su forma de coordenadas homogéneas y la linea en coordenadas homogéneas producto de los dos puntos de fijación, después, se calculan los puntos de intersección con los datos del rastreador (Fig. 3.2).

Con las líneas en coordenadas homogéneas se calculan los puntos más cercanos a los puntos de

Algoritmo 1: Algoritmo de Calibración Inicial

Datos: Datos de fijación del rastreador ocular DF, imagen con texto I
Resultado: Modelo de calibración del rastreador ocular θ_x, θ_y, Q, R

```
Generar patrón de calibración;
inicio
    Convertir I a escala de grises;
    Binarizar I (Otsu, 1979);
    Realizar operación morfológica "closening" sobre I;
    Obtener componentes conectados de I;
    Obtener centroides en I;
    Eliminar la coordenada en x de los centroides y de las fijaciones del rastreador;
    Agrupar usando la técnica de agrupación "kmeans";
    Restaurar la coordenada en x a los grupos de centroides y de fijaciones resultantes;
    Ajustar líneas (L, L') usando mínimos cuadrados;
    Calcular puntos de intersección (P_R, P'_R);
    Calcular puntos de proyección (P_Q, P'_Q);
fin
Obtener calibración inicial;
inicio
    Crear matrices para calibración usando el modelo de Cheng et al (2014);
    Calcular calibración inicial (θ_x, θ_y, Q, R);
fin
```

fijación (Fig. 3.2) utilizando las ecuaciones (3.11)

$$\begin{aligned} x' &= \frac{b(bx-ay)-ac}{a^2+b^2} \\ y' &= \frac{a(-bx+ay)-bc}{a^2+b^2} \end{aligned} \tag{3.11}$$

Para realizar el proceso de calibración se tomó el modelo propuesto en Cheng et al (2014) el cual es un modelo cuadrático de la forma:

$$\begin{aligned} \hat{x}_p &= a+bx_s+cy_s+dx_s^2+ey_s^2+fx_sy_s \\ \hat{y}_p &= g+hx_s+iy_s+jx_s^2+ky_s^2+lx_sy_s \end{aligned} \tag{3.12}$$

De esta forma los datos de los puntos de fijación, unión y proyección basados en las lecturas del rastreador ($P_E = P' \cup P'_Q \cup P'_R$) son representados en una matriz denominada A donde todos los coeficientes son puestos a 1.

Algoritmo 2: Algoritmo de Calibración Incremental

Datos: Modelo de calibración inicial θ_x, θ_y, Matrices de factorización Q, R, Nuevo punto P
Resultado: Modelos de calibración actualizados θ'_x, θ'_y, Q', R'

```
inicio
    Agregar P a R representándolo en base al modelo de Cheng et al (2014);
    Usando R y Q calcular con rotaciones de Givens Q' y R';
    Calcular calibración (θ'x,θ'y);
fin
```

Usando factorización QR sobre la matriz A y con los datos de los putos de la imagen con el texto que el participante lee ($P_S = P \cup P_Q \cup P_R$), los cuales son ordenados para coincidir con su correspondiente en los datos del sensor en la matriz A, se calculan una θx y θy que contienen el modelo de calibración inicial.

3.3.2 Calibración incremental

A partir de θ_x y θ_y obtenidas de la calibración inicial y de las matrices Q y R de la factorización se van añadiendo puntos de fijación a la calibración (Algoritmo 2).

Se toma a la matriz R como base se agrega el nuevo punto de fijación representado en el modelo cuadrático, y mediante rotaciones de Givens se calculan una nueva matriz R y una matriz Q que tiene como base la matriz Q de la factorización, y nuevamente son calculadas θ_x y θ_y. Este proceso se repite mientras haya nuevos puntos de fijación.

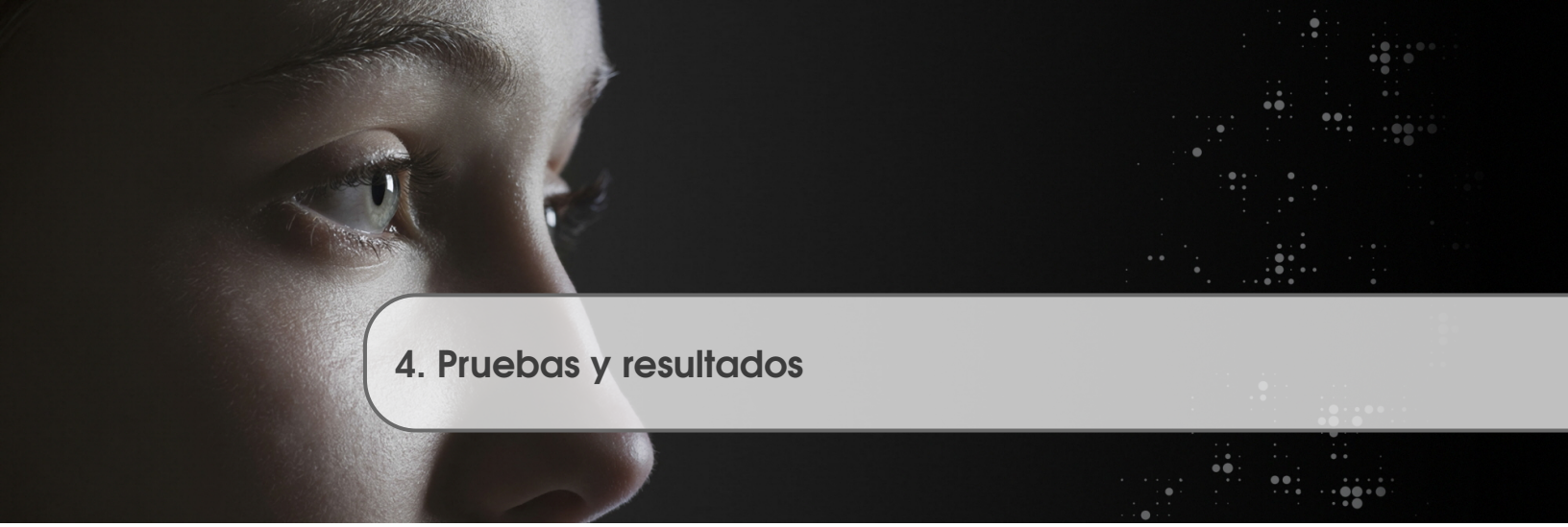
3.3.3 Detección de calibración válida

Para la detección de calibración no válida se propone usar un limite de tres veces el error anterior, de esta forma si durante la calibración incremental se inserta un error superior a tres veces el error que se tenia de la calibración anterior el sistema solicita un proceso de re-calibración, provocando así la substitución del modelo de calibración anterior.

3.4 Resumen

En este capítulo se propone una técnica de calibración que hace uso de la lectura de un texto a modo de ocultar el proceso de calibración y que permite compensar los movimientos de cabeza

mediante el uso de una calibración incremental; además, permite validar si el error en la calibración excede un limite de tolerancia predeterminado.



4. Pruebas y resultados

4.1 Objetivos de la investigación

Este estudio desea resolver las siguientes preguntas de investigación:

RQ1 ¿La técnica de calibración propuesta logra errores bajos para el eje de las ordenadas?

RQ2 ¿La técnica de calibración propuesta logra errores bajos para el eje abscisas?

RQ3 ¿La técnica de calibración propuesta logra errores similares para los ejes vertical y horizontal?

RQ4 ¿La técnica de calibración propuesta es tolerante a errores?

4.2 Metodología

Para evaluar el método propuesto, se realiza un procedimiento de calibración inicial (Algoritmo 1), acto seguido, se muestra una matriz de nueve puntos (Fig. 2.1) la cual se utiliza para medir el error inicial de la calibración. Se inicia un proceso de calibración incremental mediante el cual se añaden nuevos puntos a la calibración (Algoritmo 2) el Algoritmo 4 se usa para evaluar la calibración en cada iteración.

Algoritmo 3: Algoritmo del experimento

Datos: Datos de fijación del rastreador ocular
Resultado: Modelo de calibración del rastreador ocular

```
1 inicio
2     Realizar calibración inicial (Algoritmo 1);
3     Evaluar calibración (Algoritmo 4);
4     para i=0; i < 3; i++ hacer
5         Realizar calibración incremental (Algoritmo 2);
6         Evaluar calibración (Algoritmo 4);
7     fin
8 fin
```

Algoritmo 4: Algoritmo de Evaluación

Datos: θ_x, θ_y, Q, R
Resultado: e_x, e_y

```
inicio
    Mostrar matriz de nueve puntos;
    Obtener fijaciones;
    Calcular estimaciones de los puntos de fijacion usando θx y θy (3.12);
    Calcular ex y ey (4.1);
fin
```

4.3 Métricas

Para la obtención de resultados se tomaron los datos de todos los participantes y se dividieron en las siguientes categorías: error en calibración inicial, error en iteración uno, error en iteración dos y error en iteración tres. Con los datos obtenidos se generan varias tablas que contienen el error absoluto en las coordenadas x y y calculadas por la calibración respecto a las mostradas en pantalla.Para calcular el error absoluto (4.1) en x (e_x) y en y (e_y), de cada categoría se obtiene el error medio en x ($\hat{x}_i$) y en y ($\hat{y}_i$). A continuación, para cada una de las iteraciones se compara con las coordenadas base (x_i, y_i).

$$\begin{aligned} e_{x_i} &= |\hat{x} - x_i| \\ e_{y_i} &= |\hat{y} - y_i| \end{aligned} \tag{4.1}$$

donde $i = 1 \ldots 9$ es el número de puntos en la matriz de evaluación (x_i, y_i) son las coordenadas de cada punto y $\hat{x}_i$ y $\hat{y}_i$ se calculan con (3.12) con los parámetros de calibración actualizados θ_x y θ_y.

4.4 Procedimiento

Para evaluar los métodos propuestos se diseñó un experimento con los pasos que se describen en el Algoritmo 3.

En el experimento participaron nueve voluntarios con un rango de edad de 23 a 33 años, a todos los participantes se les instruyó a dar clic en el punto que estaban observando con el objetivo de asegurar que su mirada estaba fija en ese punto y no en otra dirección, a continuación debían leer un

texto pequeño en la pantalla, dar clic en la pantalla al terminar de leer y continuar haciendo clic en los puntos conforme aparecían en la pantalla.

4.5 Análisis estadístico

Se realizaron tres pruebas t para muestras emparejadas donde se comparan los errores e_x con la iteración previa y entre la primera iteración y la última, de igual forma para los errores e_y además de de los errores e_x contra los errores e_y, para las pruebas se usa un valor de $p < 0.05$ para considerar diferencias significativas.

4.6 Resultados

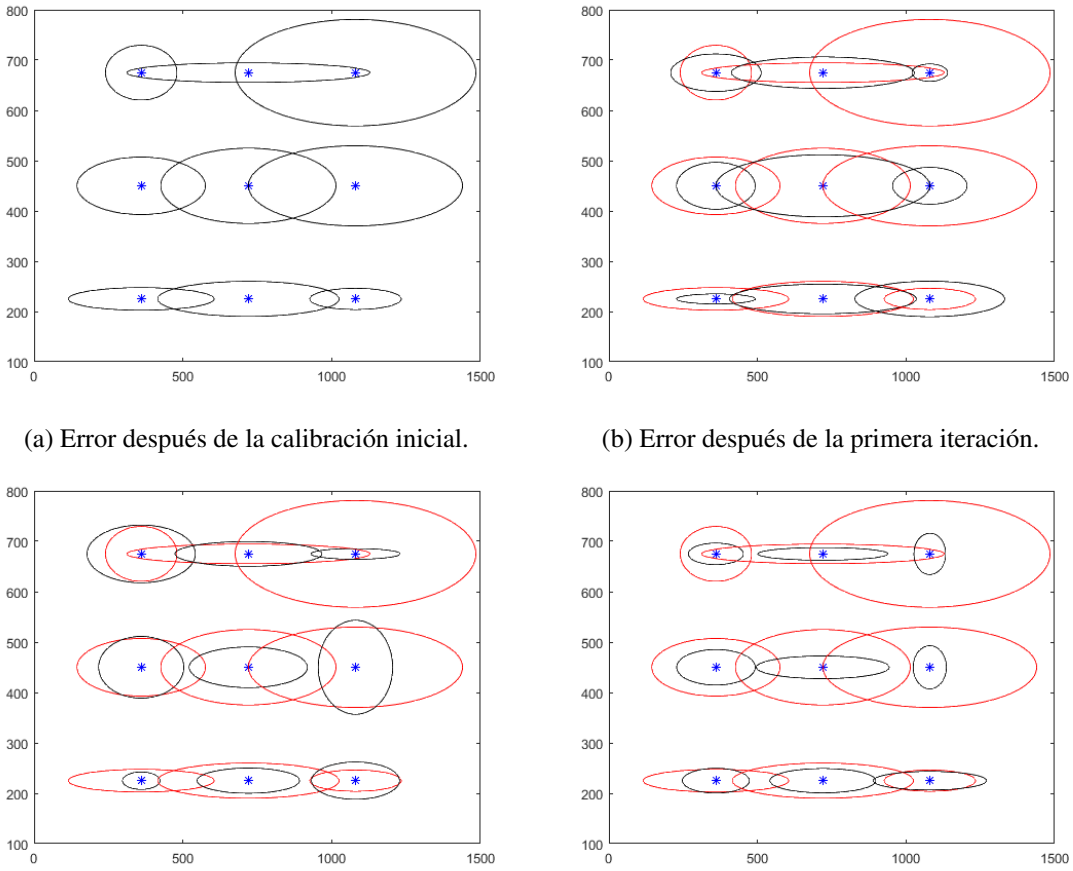
Como se mencionó anteriormente, los resultados se dividieron en cuatro categorías y se tomó el error medio de cada una en x y en y, las figuras 4.1 ilustran los resultados obtenidos, en estas figuras se muestran los nueve puntos de fijación (como asteriscos azules) junto al error obtenido en esa calibración (como óvalos negros), los óvalos rojos en las figuras 4.1b a 4.1d muestran el error de la calibración inicial como comparación.

La tabla 4.1 muestra los errores absolutos obtenidos (e_x y e_y) para cada punto de la matriz utilizada para medir el error por cada iteración usando una resolución de 1440×900píxeles.

La primera prueba evalúa los errores e_x, en ella se observa como la media decrece de la iteración 0 en 278.50píxeles a la iteración 3 con 139.92píxeles; para la desviación estándar pasa de 103.66 en la iteración 0 a 66.09 en la iteración 3, entre la iteración 0 y la 3 se tiene $t(8) = 3.31$, $p = 0.011$. Por lo que se puede concluir que esta diferencia es estadísticamente significativa.

La segunda prueba evalúa los errores e_y, en ésta se tiene un descenso de la media de 52.22píxeles a 26.95píxeles de la iteración 0 a la 3, la desviación estándar desciende de 30.41 a 10.36, entre la iteración 0 y la 3 se tiene $t(8) = 3.26$, $p = 0.011$. Por lo que se puede concluir que esta diferencia es estadísticamente significativa.

La tercer prueba compara el error entre e_x y e_y, en ésta se observa como los errores absolutos en x (media$(e_x) = 278.50$píxeles) y los errores absolutos en y (media$(e_y) = 52.22$píxeles) arrojaron diferencias significativas en la calibración inicial, $t(8) = 6.98$, $p < 0.001$. En las siguien-



(a) Error después de la calibración inicial.

(b) Error después de la primera iteración.

(c) Error después de la segunda iteración.

(d) Error después de la tercera iteración.

Figura 4.1: Error en la calibración después de cada iteración.

tes iteraciones esta diferencia se reduce. Por ejemplo, en la tercera re-calibración las medias son media$(e_x) = 139.92$píxeles y media$(e_y) = 26.95$píxeles; pero, la diferencia aún es estadísticamente significativa, $t(8) = 4.55$, $p = 0.002$. Por lo que podemos concluir que los errores e_y tienden a ser significativamente menores respecto a los de e_x.

Tabla 4.1: Error de los resultados del algoritmo propuesto en iteraciones de calibración.

	iteración 0		iteración 1		iteración 2		iteración 3	
Punto	e_x	e_y	e_x	e_y	e_x	e_y	e_x	e_y
1	244.9350	22.4530	132.6230	9.9630	62.9760	17.2140	112.7540	24.564
2	304.9400	34.9440	314.6100	29.3810	171.9960	25.0270	178.3970	24.1490
3	153.9100	21.0690	251.9570	35.3360	147.7460	36.9850	190.9390	18.3480
4	215.6030	57.1830	132.1900	46.6390	142.9340	61.3260	132.1320	35.2410
5	294.4100	74.8350	360.0900	61.7670	198.7830	40.3310	223.7760	22.5250
6	359.8000	79.6910	125.3680	36.5040	125.1300	93.2740	55.8400	42.7440
7	120.1500	54.4320	152.3550	37.2990	181.8790	57.2260	92.7590	21.3920
8	408.3400	19.4910	308.0390	31.3420	246.8590	24.5010	219.1150	12.5990
9	404.4440	105.8530	60.0900	17.2790	150.0850	10.6620	53.5500	40.9540

5. Conclusiones y trabajo futuro

5.1 Discusión de resultados

Como se observó en las pruebas t para muestras emparejadas la técnica de calibración propuesta logra errores bajos para el eje de las ordenadas, pero no así para el eje de las abscisas donde se observan errores significativamente mas grandes, esto se debe a el diseño del patrón de calibración generado el cual causa que se tenga una mayor cantidad de puntos de referencia verticales respecto a los horizontales. En comparativa entre los errores en e_x y e_y, en base a la desviación estándar, el error típico y las medias se observa como los errores en el eje horizontal son mayores a los del eje vertical. En base a los resultados obtenidos, se observa que el la técnica de calibración propuesta es capaz de mejorar la precisión de la calibración a través de varias iteraciones, además de que el proceso de calibración inicial queda oculto al participante y permite recuperarse de un error desechando la calibración anterior y re-calibrando.

5.2 Conclusiones

Este trabajo presenta un método de calibración que hace énfasis en rastreadores oculares sin compensación de movimiento de cabeza. Mediante los resultados obtenidos es posible ver que la técnica propuesta permite calibrar un rastreador ocular. Para llegar a este resultado fue necesario cumplir con los siguientes objetivos:

- Estudiar los tipos de transformaciones geométricas en 2D.
- Estudiar técnicas de calibración de rastreadores oculares.

  Cumplir este objetivo nos permite observar que la técnica propuesta puede usarse en rastreadores oculares más económicos los cuales suelen de carecer de cámaras estéreo o no depender de largos conjuntos de datos de rostros para hacer el seguimiento.

- Proponer una técnica de calibración basada en factorización QR y rotaciones de Givens.

  El uso de factorización QR permite que sólo durante la calibración inicial o la re-calibración se calcule un modelo de calibración completo permitiendo añadir puntos de fijación a la calibración sin tener que realizar el cálculo completo.

- Probar la técnica de calibración con un rastreador ocular.

  Durante las pruebas se pudo observar que la técnica propuesta, debido a que tiene un error más bajo en el eje vertical, permite detectar si una línea de texto fue leída; pero también que si se desea utilizarla en otras aplicaciones es necesario mejorar o desarrollar otro patrón de calibración que permita mejorar la precisión en el eje horizontal.

La realización de estos objetivos hizo posible cumplir con el objetivo general:

- Calibrar en tiempo real un rastreador ocular y detectar una calibración no válida.

  Al calibrar en tiempo real un rastreador ocular y detectar una calibración no válida se puede reducir el tiempo que toman los experimentos ya que en muchas situaciones cuando un participante se mueve hay que interrumpir un experimento y ejecutarlo nuevamente o adquirir dispositivos con la capacidad de corregir ese movimiento lo cual incrementa los costos.

Debido a esto, es posible decir que se cumplió con la hipótesis planteada en el capítulo 1 de esta tesis, la cual tenía como objetivo el comprobar si se podía generar una técnica de calibración en tiempo real para rastreador ocular basado en video capaz de compensar movimientos de cabeza naturales.

5.3 Trabajo futuro

En esta sección se muestran las posibles mejoras que pueden realizarse a futuro:

- Mejorar la precisión de la calibración inicial; esto se podría lograr mediante la inclusión de más texto o de imágenes simples que sirvan como puntos de fijación.
- Comparar el método propuesto con otros para observar cómo se desempeña en cuestión de precisión, sin embargo los algoritmos de calibración suelen ser mantenidos en secreto por lo que realizar esta comparación podría ser difícil.
- Reducir el error para las coordenadas en x, ya que para calcular la calibración inicial se utilizan varias proyecciones y estas en su mayoría se ven mas reflejadas en el eje vertical, incrementar

los puntos y proyecciones horizontales podría beneficiar a la precisión de las coordenadas en x.

Bibliografía

Abdi H (2003) Least Squares. In: M. Lewis-Beck, A. Bryman, T. Futing (Eds): Encyclopedia for research methods for the social sciences. Thousand Oaks (CA): Sage., pp 559–561

Alvarado Hernández MG (2016) Evaluación de Objetos de Aprendizaje a través de Seguimiento Ocular. PhD thesis, Centro de Investigación en Matemáticas A.C. (Zacatecas, Zacatecas)

Álvarez J, Otamendi A, Belfer K, Nesbit J, Leacock T (2010) Instrumento para la evaluación de objetos de aprendizaje (lori_esp) manual de usuario

Cheng D, An M, Yu W, Fang L (2014) Learning Dynamic Models of Gaze Point Mapping from Eye Movement. In: Proceedings of International Conference on Internet Multimedia Computing and Service, ACM, New York, NY, USA, ICIMCS '14, pp 289:289—-289:294, DOI 10.1145/2632856.2632873, URL http://doi.acm.org/10.1145/2632856.2632873

Covarrubias R (2016) Evaluación de Carga Cognitiva en Videojuegos con Seguimiento Ocular. PhD thesis

Duchowski A (2007) Eye Tracking Methodology: Theory and Practice, 2nd edn. Springer, URL http://gen.lib.rus.ec/book/index.php?md5=77AAC8F1DDFA6ED76E69AC6E5DB452E8

Fixxl Ltd (2017) Chin and head rest. Webpage, https://www.rehacom.co.uk/chin-and-head-rest/

Flatla DR, Gutwin C, Nacke LE, Bateman S, Mandryk RL (2011) Calibration games: Making calibration tasks enjoyable by adding motivating game elements. In: Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology, ACM, New York, NY, USA, UIST '11, pp 403–412, DOI 10.1145/2047196.2047248, URL http://doi.acm.org/10.1145/2047196.2047248

Hartley R, Zisserman A (2004) Multiple View Geometry in Computer Vision, 2nd edn. Cambridge University Press, ISBN: 0521540518

Krafka K, Khosla A, Kellnhofer P, Kannan H, Bhandarkar S, Matusik W, Torralba A (2016) Eye tracking for everyone. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp 2176–2184

Lara-Alvarez C, Mitre-Hernandez H, Alvarado-Hernandez M (2016) Entropy of eye fixations: a tool for evaluation of learning objects. Advances in Computational Linguistics p 89

Lupu RG, Ungureanu F (2013) A survey of eye tracking methods and applications. Mathematics Subject Classification:68U35,68N19,94A12 LXIII(3):72–86

Mitre-Hernandez H, Alvarado-Hernandez M, Lara-Alvarez C (2016) Evaluation of learning objects through eye tracking. In: Software Process Improvement (CIMPS), International Conference on, IEEE, pp 1–8

Otsu N (1979) A threshold selection method from gray-level histograms. IEEE Transactions on Systems, Man, and Cybernetics 9(1):62–66, DOI 10.1109/TSMC.1979.4310076

Poole A, Ball LJ (2005) Eye tracking in human-computer interaction and usability research: Current status and future. In: Prospects", Chapter in C. Ghaoui (Ed.): Encyclopedia of Human-Computer Interaction. Pennsylvania: Idea Group, Inc

Press WH, Teukolsky SA, Vetterling WT, Flannery BP (2007) Numerical Recipes 3rd Edition: The Art of Scientific Computing, 3rd edn. Cambridge University Press, New York, NY, USA

The Eye Tribe (2017) Calibration. Webpage, https://s3.eu-central-1.amazonaws.com/theeyetribe.com/theeyetribe.com/dev/general/index.html

Zhu Z, Ji Q (2007) Novel Eye Gaze Tracking Techniques Under Natural Head Movement. IEEE Transactions on Biomedical Engineering 54(12):2246–2260, DOI 10.1109/TBME.2007.895750

Glosario

CIMAT Centro de Investigación en Matemáticas unidad Zacatecas. 2

cámaras estéreo son cámaras capaces de capturar imágenes (fotografías) en tres dimensiones. 2

DSP Digital Signal Processor. 1

electrooculograma es un examen que consiste en colocar pequeños electrodos cerca de los músculos de los ojos para medir el movimiento de estos. 1

firmware es el programa informático que tiene directa interacción con los circuitos electrónicos de un dispositivo, siendo así el encargado de controlarlo para ejecutar correctamente las instrucciones externas. 5

HCI Human–Computer Interaction. 1

lente de contacto escleral son un tipo de lentes semirrígidas con un tamaño mayor de lo habitual. Su diámetro es similar al de las lentes blandas; gracias a ello, se pueden apoyar en la esclera (la parte blanca del ojo), que es una zona inervada, por lo que resultan muy cómodas de colocar. 1

mapeo es la realización de un mapa o conjunto de elementos de un mismo tipo o categoría que tienen una distribución espacial determinada. 8, 10

marketing es la actividad, conjunto de instituciones y procesos para crear, comunicar, distribuir e intercambiar ofertas que tengan valor para los consumidores, clientes, socios y la sociedad en general. 7

matriz es un arreglo bidimensional de números. 9

matriz ortogonal es una matriz cuadrada cuya matriz inversa coincide con su matriz traspuesta. 17

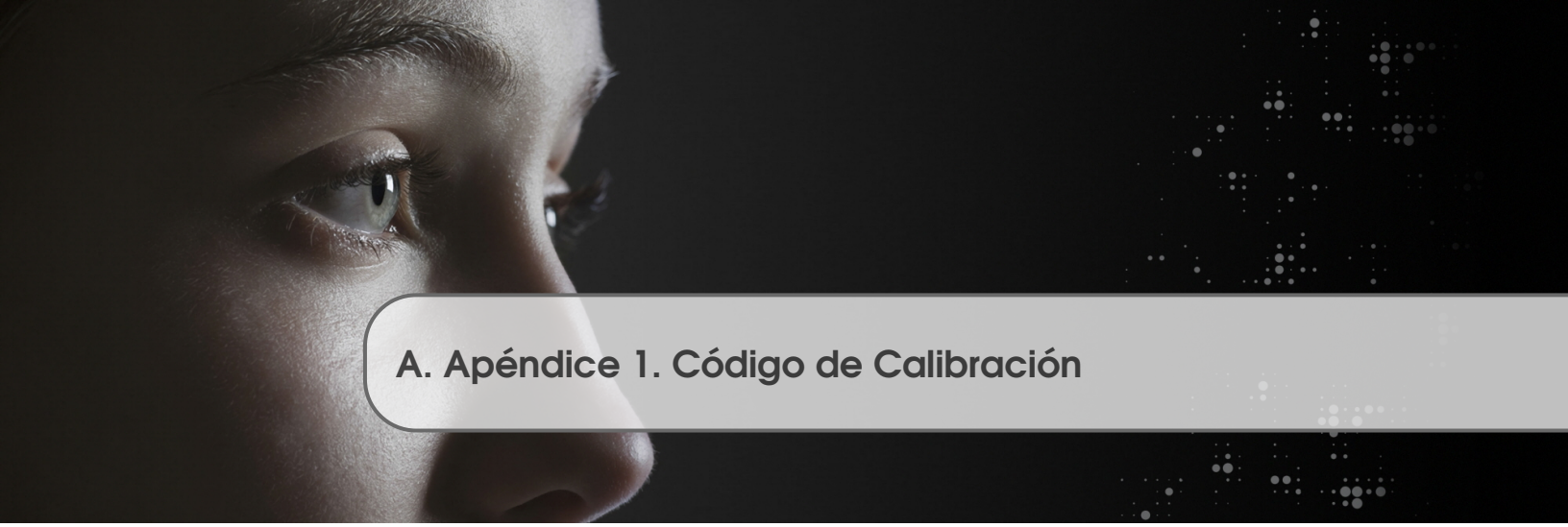
matriz triangular es un tipo especial de matriz cuadrada cuyos elementos por encima o por debajo de su diagonal principal son cero. 17

NFL National Football League. 7

plano cartesiano es un diagrama que permite localizar puntos específicamente dentro de un sistema de coordenadas que se conocen como Coordenadas Rectangulares. 15

taxonomía clasificación u ordenación en grupos de cosas que tienen unas características comunes. 1

video-oculografía es un procedimiento de exploración de la motilidad ocular. 1



A. Apéndice 1. Código de Calibración

Librerías

OpenCV Es una biblioteca libre de visión artificial.

Eigen Es una biblioteca de C ++ de alto nivel para álgebra lineal, operaciones matriciales y de vectores, transformaciones geométricas, solucionadores numéricos y algoritmos relacionados.

Windows.h Es un archivo cabecera específico de Windows para la programación en lenguaje C/C++ que contiene las declaraciones de todas las funciones de la biblioteca Windows API

gazeapi.h Proporciona una interfaz C ++ para comunicarse con el servidor EyeTribe a través de la API abierta de EyeTribe.

iostream Es un componente de la biblioteca estándar (STL) del lenguaje de programación C++ que es utilizado para operaciones de entrada/salida.

numeric Provee algoritmos para el procesamiento numérico.

list Provee la plantilla clase contenedora std::list, una lista doblemente enlazada.

iterator Provee clases y plantillas para trabajar con iteradores.

vector Provee la plantilla clase contenedora std::vector, un arreglo dinámico.

fstream Provee facilidades para la entrada y salida basada en archivos.

sstream Provee la clase plantilla std::sstream y otras clases para la manipulación de cadenas de caracteres.

TestOpenCV.cpp

```
1 #include <opencv2/core.hpp>
2 #include <opencv2/imgcodecs.hpp>
3 #include <opencv2/imgproc.hpp>
```

```
#include <opencv2/highgui.hpp>
#include <iostream>
#include <iterator>
#include <vector>
#include <Eigen/Dense>
#include <fstream>
#include <Windows.h>
#include <sstream>
#include "MyGaze.h"


using namespace cv;
using namespace std;

vector<Point2f> coords;
int width = GetSystemMetrics(SM_CXSCREEN), height = GetSystemMetrics
   (SM_CYSCREEN);
Point2f scr_pnt_up((width / 5) * 4, 100), scr_pnt_down(width / 5,
   height - 100);
bool flag;

void waitClick()
{
        flag = true;
        while (flag)
        {
                int k = waitKey(10);
                if (k == 27) break;
        }
}
int save_to_file(string path, vector<Point2f> vec)
{
```

```
34          ofstream outfile;
35          outfile.open(path);
36          for (auto n : vec)
37          {
38                  outfile << n.x << ", " << n.y << endl;
39          }
40          outfile.close();
41          return 0;
42  }
43
44  vector<Point2f> mat_to_vec(Mat fix_mat)
45  {
46          vector<Point2f> fix;
47          for (int i = 0; i < fix_mat.rows; i++)
48          {
49                  double *ptr = fix_mat.ptr<double>(i);
50                  fix.push_back(Point2f(ptr[0], ptr[1]));
51          }
52          std::sort(fix.begin(), fix.end(), [=](const cv::Point2f &a,
                const cv::Point2f &b) {
53                  return (norm(a - scr_pnt_up) < norm(b - scr_pnt_up))
                        ;
54          });
55          return fix;
56  }
57
58  Mat read_file(string path)
59  {
60          /*Data set reading*/
61          ifstream inputfile(path);
62          vector<vector<double>> values;
63          vector<double> x, y;
```

```
64          for (string line; getline(inputfile, line);)
65          {
66                  replace(line.begin(), line.end(), ',', ' ');
67                  istringstream in(line);
68                  values.push_back(
69                          vector<double>(istream_iterator<double>(in),
70                                  istream_iterator<double>()));
71          }
72          int count = 0;
73          for (auto n : values)
74          {
75                  if (count != 1 || count != 2)
76                  {
77                          x.push_back(n[0]);
78                          y.push_back(n[1]);
79                  }
80                  else
81                  {
82                          count++;
83                  }
84          }
85          Mat my(y), mx(x), resoult;
86          hconcat(mx, my, resoult);
87          return resoult;
88  }
89
90  void on_mouse(int event, int x, int y, int, void *img)
91  {
92          if (event == EVENT_LBUTTONDOWN)
93          {
94                  cout << "Left button of the mouse is clicked -
                        position (" << x << ", " << y << ")" << endl;
```

```
95                  flag = false;
96          }
97  }
98
99  int display_line(Mat &line_ds, Mat &img)
100 {
101         for (int i = 0; i < line_ds.rows; i++)
102         {
103                 const double *p1 = line_ds.ptr<double>(i);
104                 if (i + 1 < line_ds.rows)
105                 {
106                         double *p2 = line_ds.ptr<double>(i + 1);
107                         line(img, Point(p1[0], p1[1]), Point(p2[0],
                               p2[1]), Scalar(0, 0, 0), 1, CV_AA);
108                 }
109         }
110         return 0;
111 }
112
113 Point2f intersect(Eigen::Vector3d line_h, vector<Point2f> pnt_in)
114 {
115         Eigen::Vector3d pnt1, pnt2, line_intersect, pnt_intersect;
116         pnt1 << pnt_in[0].x, pnt_in[0].y, 1;
117         pnt2 << pnt_in[1].x, pnt_in[1].y, 1;
118         line_intersect = pnt1.cross(pnt2);
119         cout << line_intersect << "linea de interserccion" << endl;
120         cout << pnt1 << endl << pnt2 << "Puntos de fijacion" << endl
               ;
121         pnt_intersect = line_h.cross(line_intersect);
122         pnt_intersect = pnt_intersect * (1 / pnt_intersect(2));
123         return Point2f(pnt_intersect(0), pnt_intersect(1));
124 }
```

```
125
126 Point2f proyect_line(Mat &theta, Mat &img, Point2f pnt)
127 {
128         double a = theta.at<double>(0), c = theta.at<double>(1), a2
               = a * a;
129         int b = -1, b2 = b * b;
130         int x = (b * (b * (pnt.x) - a * pnt.y) - (a * c)) / (a2 + b2
               );
131         int y = (a * (-b * (pnt.x) + a * pnt.y) - (b * c)) / (a2 +
               b2);
132         line(img, Point(pnt.x, pnt.y), Point(x, y), Scalar(0, 255,
               0), 1, CV_AA);
133         return Point2f(x, y);
134 }
135
136 Mat adjust_line(Mat &centroids, Mat &img, vector<Point2f> &pnt,
       vector<Point2f> pnt_in, double &b)
137 {
138         Mat y = centroids.col(1), x = centroids.col(0), one = Mat::
               ones(x.size(), x.type());
139         hconcat(x, one, one);
140         Mat theta = (one.t() * one).inv() * one.t() * y;
141         b = theta.at<double>(1);
142         Mat yp = (theta.at<double>(0) * x) + theta.at<double>(1);
143         Eigen::Vector3d line_h(theta.at<double>(0), -1, theta.at<
               double>(1));
144         vector<Point2f> vec_line, vec_proyect;
145         pnt.push_back(intersect(line_h, pnt_in));
146         for (auto i : pnt_in)
147         {
148                 pnt.push_back(proyect_line(theta, img, i));
149         }
```

```
150         return yp;
151 }
152
153 int get_centroids(Mat &img, vector<Point2f> &pnts)
154 {
155         Mat grey, binimg, labels, stats, centroids, best_labels,
               attemps, centers, centroids32f;
156         cvtColor(img, grey, COLOR_BGR2GRAY);
157         threshold(grey, binimg, 80, 255, THRESH_BINARY_INV |
               THRESH_OTSU);
158         connectedComponentsWithStats(binimg, labels, stats,
               centroids);
159         centroids.convertTo(centroids32f, CV_32F);
160         kmeans(centroids32f.col(1), 3, best_labels, TermCriteria(
               CV_TERMCRIT_ITER | CV_TERMCRIT_EPS, 10000, 0.0001), 5,
               KMEANS_PP_CENTERS, centers);
161         Mat red, green, blue;
162         for (int i = 0; i < centroids.rows; i++)
163         {
164                 const double *Mi = centroids.ptr<double>(i);
165                 int tmp = best_labels.at<int>(i);
166                 switch (tmp)
167                 {
168                 case 0:
169                         circle(img, Point(Mi[0], Mi[1]), 2.0, Scalar
                           (0, 0, 255), -1, CV_AA);
170                         red.push_back(centroids.row(i));
171                         break;
172                 case 1:
173                         circle(img, Point(Mi[0], Mi[1]), 2.0, Scalar
                           (0, 255, 0), -1, CV_AA);
174                         green.push_back(centroids.row(i));
```

```
175                         break;
176                 case 2:
177                         circle(img, Point(Mi[0], Mi[1]), 2.0, Scalar
                               (255, 0, 0), -1, CV_AA);
178                         blue.push_back(centroids.row(i));
179                         break;
180                 default:
181                         break;
182                 }
183         }
184         vector<Point2f> pnt_in, pnt_red, pnt_blue, pnt_green;
185         pnt_in.push_back(scr_pnt_up);
186         pnt_in.push_back(scr_pnt_down);
187         vector<double> b(3);
188         Mat yp_red = adjust_line(red, img, pnt_red, pnt_in, b[0]),
               yp_blue = adjust_line(blue, img, pnt_blue, pnt_in, b[1]),
                yp_green = adjust_line(green, img, pnt_green, pnt_in, b
               [2]), line_red, line_blue, line_green;
189         /*Sort*/
190         if (b[0] > b[1])
191         {
192                 swap(b[0], b[1]);
193                 swap(pnt_red, pnt_blue);
194         }
195         if (b[0] > b[2])
196         {
197                 swap(b[0], b[2]);
198                 swap(pnt_red, pnt_green);
199         }
200         if (b[1] > b[2])
201         {
202                 swap(b[1], b[2]);
```

```
203                 swap(pnt_blue, pnt_green);
204         }
205         /*sort*/
206         pnts.reserve(pnt_red.size() + pnt_blue.size() + pnt_green.
               size());
207         pnts.insert(pnts.end(), pnt_red.begin(), pnt_red.end());
208         pnts.insert(pnts.end(), pnt_blue.begin(), pnt_blue.end());
209         pnts.insert(pnts.end(), pnt_green.begin(), pnt_green.end());
210         hconcat(red.col(0), yp_red, line_red);
211         display_line(line_red, img);
212         hconcat(blue.col(0), yp_blue, line_blue);
213         display_line(line_blue, img);
214         hconcat(green.col(0), yp_green, line_green);
215         display_line(line_green, img);
216         imshow("Display window", img);
217         return 0;
218 }
219
220 Mat display_img(string path)
221 {
222         Mat image;
223         image = imread(path, IMREAD_COLOR);
224         if (image.empty())
225         {
226                 cout << "Could not open or find the image" << std::
                       endl;
227         }
228         resize(image, image, Size(width, height), 0, 0,
               CV_INTER_AREA);
229         namedWindow("Display window", CV_WINDOW_NORMAL);
230         setWindowProperty("Display window", CV_WND_PROP_FULLSCREEN,
               CV_WINDOW_FULLSCREEN);
```

```
231         setMouseCallback("Display window", on_mouse, &image);
232         imshow("Display window", image);
233         waitClick();
234         return image;
235 }
236
237 int display_point(float x, float y)
238 {
239         Mat matPoint = Mat::zeros(1000, 1000, CV_8UC3);
240         circle(matPoint,
241                 Point(x, y),
242                 5.0,
243                 Scalar(0, 0, 255),
244                 -1,
245                 CV_AA);
246         namedWindow("Display window", CV_WINDOW_NORMAL); // Create a
                window for display.
247         setWindowProperty("Display window", CV_WND_PROP_FULLSCREEN,
               CV_WINDOW_FULLSCREEN);
248         imshow("Display window", matPoint); // Show our image inside
                it.
249         waitKey(1 /*000*/);                     // Wait for a
               keystroke in the window
250         return 0;
251 }
252
253 int calibration_points(Mat &a, float x, float y, vector<Point2f> tmp
       )
254 {
255         display_point(x, y);
256         std::sort(tmp.begin(), tmp.end(), [=](const cv::Point2f &a,
               const cv::Point2f &b) {
```

```
257                 return (norm(a - Point2f(x, y)) < norm(b - Point2f(x
                        , y)));
258         });
259         int median = floorf(tmp.size() / 2) - 1;
260         vector<double> arrtmp = { 1, tmp[median - 2].x, tmp[median -
                2].y, tmp[median - 2].x * tmp[median - 2].x, tmp[median
               - 2].y * tmp[median - 2].y, tmp[median - 2].x * tmp[
               median - 2].y };
261         Mat row(arrtmp);
262         a.push_back(row.t());
263         arrtmp = { 1, tmp[median].x, tmp[median].y, tmp[median].x *
               tmp[median].x, tmp[median].y * tmp[median].y, tmp[median
               ].x * tmp[median].y };
264         row = Mat(arrtmp);
265         a.push_back(row.t());
266         arrtmp = { 1, tmp[median + 2].x, tmp[median + 2].y, tmp[
               median + 2].x * tmp[median + 2].x, tmp[median + 2].y *
               tmp[median + 2].y, tmp[median + 2].x * tmp[median + 2].y
               };
267         row = Mat(arrtmp);
268         a.push_back(row.t());
269         return 0;
270 }
271
272 int factorize_givens(Eigen::MatrixXd &Q, Eigen::MatrixXd &R)
273 {
274         Eigen::MatrixXd idnt = Eigen::MatrixXd::Identity(Q.rows(), Q
               .cols());
275         for (int i = 0; i < R.cols(); i++)
276         {
277                 Eigen::MatrixXd tmp = idnt;
278                 Eigen::Matrix2d rotation;
```

```
279                 double a = R(i, i), b = R(i + 1, i);
280                 double hipotenuse = sqrt((a * a) + (b * b));
281                 rotation << a / hipotenuse, b / hipotenuse,
282                         -b / hipotenuse, a / hipotenuse;
283                 tmp.block(i, i, 2, 2) = rotation;
284                 R = tmp * R;
285                 cout << R << endl;
286                 Q = Q * tmp.transpose();
287         }
288         return 0;
289 }
290
291 int draw_pnt(Mat &matPoint, Point2f pnt, Scalar color)
292 {
293         if (matPoint.empty())
294         {
295                 matPoint = Mat(Size(width, height), CV_8UC3);
296                 matPoint.setTo(Scalar(255, 255, 255));
297         }
298         circle(matPoint,
299                 pnt,
300                 10.0,
301                 color,
302                 -1,
303                 CV_AA);
304         imshow("Display window", matPoint); // Show our image inside
                it.
305         waitClick();
306         circle(matPoint,
307                 pnt,
308                 10.0,
309                 Scalar(0, 0, 255),
```

```
310                 -1,
311                 CV_AA);
312         imshow("Display window", matPoint);
313         waitKey(1000);
314         return 0;
315 }
316
317 int incremental(Eigen::MatrixXd &Q, Eigen::MatrixXd &R, Eigen::
    VectorXd &theta_x, Eigen::VectorXd &theta_y, Eigen::VectorXd &x,
    Eigen::VectorXd &y, vector<Point2f> pnts, int inc, Eigen::
    MatrixXd &a)
318 {
319         Scalar black(0, 0, 0), red(0, 0, 255), green(0, 255, 0);
320         vector<Point2f> fix;
321         Mat fix_mat, matPoint;
322         ofstream outfile;
323         for (int i = 0; i < pnts.size(); i++)
324         {
325                 draw_pnt(matPoint, pnts[i], black);
326                 stringstream ss;
327                 ss << "fix" << i << "_inc" << inc << ".csv";
328                 fix_mat = read_file(ss.str());
329                 fix = mat_to_vec(fix_mat);
330                 Point2f pnt = fix[static_cast<int>(fix.size() / 2)];
331                 Eigen::VectorXd vec(6);
332                 vec << 1, pnt.x, pnt.y, pnt.x * pnt.x, pnt.y * pnt.y
                        , pnt.x * pnt.y;
333                 double x_4 = vec.dot(theta_x), y_4 = vec.dot(theta_y
                        );
334                 cout << "x: " << x_4 << endl
335                         << "y: " << y_4 << endl;
336                 draw_pnt(matPoint, Point2f(x_4, y_4), red);
```

```
337
338                 /* Add point to calibration */
339                 Eigen::MatrixXd idnt = Eigen::MatrixXd::Identity(Q.
                       rows() + 1, Q.cols() + 1);
340                 Eigen::MatrixXd tmp = idnt;
341                 tmp.block(1, 1, Q.rows(), Q.cols()) = Q;
342                 Q = tmp;
343                 cout << R << endl;
344                 tmp = Eigen::MatrixXd(R.rows() + 1, R.cols());
345                 tmp << vec.transpose(),
346                         R;
347                 R = tmp;
348                 cout << endl << R << endl;
349                 cout << "doing givens..." << endl;
350                 factorize_givens(Q, R);
351                 theta_x = Eigen::VectorXd::Zero(6);
352                 theta_y = Eigen::VectorXd::Zero(6);
353                 Eigen::VectorXd vtmpx(x.size() + 1), vtmpy(y.size()
                       + 1);
354                 vtmpx << pnts[i].x, x;
355                 x = vtmpx;
356                 vtmpy << pnts[i].y, y;
357                 y = vtmpy;
358                 Eigen::MatrixXd QtX = Q.transpose() * vtmpx;
359                 Eigen::MatrixXd QtY = Q.transpose() * vtmpy;
360                 for (int i = (R.cols() - 1); i > -1; i--)
361                 {
362                         theta_x(i) = (QtX(i, 0) - R.row(i).dot(
                               theta_x)) / R(i, i);
363                         theta_y(i) = (QtY(i, 0) - R.row(i).dot(
                               theta_y)) / R(i, i);
364                 }
```

```
365                 cout << theta_x << endl
366                         << endl
367                         << theta_y << endl;
368                 x_4 = vec.dot(theta_x);
369                 y_4 = vec.dot(theta_y);
370                 draw_pnt(matPoint, Point2f(x_4, y_4), green);
371                 matPoint.release();
372
373                 /* Minimos cuadrados */
374                 Eigen::MatrixXd(a.rows() + 1, a.cols());
375                 tmp << vec.transpose(),
376                         a;
377                 Eigen::VectorXd thetaXMC(6), thetaYMC(6);
378                 thetaXMC = (tmp.transpose() * tmp).inverse() * tmp.
                       transpose() * x;
379                 thetaYMC = (tmp.transpose() * tmp).inverse() * tmp.
                       transpose() * y;
380                 cout << "Minimos cuadrados iteracion " << inc <<
                       endl << thetaXMC << endl << endl << thetaYMC <<
                       endl;
381                 a = tmp;
382         }
383         return 0;
384 }
385
386 int draw_matrix(Eigen::VectorXd &theta_x, Eigen::VectorXd &theta_y,
       int itr)
387 {
388         Mat fix_mat, matPoint;
389         ofstream outfile;
390         vector<Point2f> fix;
391         stringstream ss;
```

```
392         Point2f matrix_pnt;
393         for (int i = 0; i < 3; i++)
394         {
395                 for (int j = 0; j < 3; j++)
396                 {
397                         matrix_pnt = Point2f((width / 4)*(j + 1), (
                            height / 4)*(i + 1));
398                         draw_pnt(matPoint, matrix_pnt, Scalar(0, 0,
                            0));
399                         ss << "Matrix_row_" << i << "_col" << j << "
                            _itr_" << itr << ".csv";
400
401                         /* Closing stuff */
402                         matPoint.release();
403
404                         fix_mat = read_file(ss.str());
405                         cout << "Mat fix" << endl << fix_mat << endl
                            ;
406                         fix = mat_to_vec(fix_mat);
407                         Point2f pnt = fix[static_cast<int>(fix.size
                            () / 2)];
408                         Eigen::VectorXd vec(6);
409                         vec << 1, pnt.x, pnt.y, pnt.x * pnt.x, pnt.y
                             * pnt.y, pnt.x * pnt.y;
410                         double x = vec.dot(theta_x), y = vec.dot(
                            theta_y);
411                         ss.str(std::string());
412                 }
413         }
414         return 0;
415 }
416
```

```
417 int auto_calibrate()
418 {
419         ofstream outfile;
420         vector<Point2f> fix_1, fix_2, fix_3;
421         Mat fix1_mat, fix2_mat;
422         // Fixation points
423         Mat matPoint(Size(width, height), CV_8UC3);
424         matPoint.setTo(Scalar(255, 255, 255));
425         circle(matPoint,
426                 scr_pnt_up,
427                 10.0,
428                 Scalar(0, 0, 0),
429                 -1,
430                 CV_AA);
431         namedWindow("Display window", CV_WINDOW_NORMAL);
432         setWindowProperty("Display window", CV_WND_PROP_FULLSCREEN,
              CV_WINDOW_FULLSCREEN);
433         setMouseCallback("Display window", on_mouse, &matPoint);
434         imshow("Display window", matPoint);
435         waitClick();
436         circle(matPoint,
437                 scr_pnt_up,
438                 10.0,
439                 Scalar(0, 0, 255),
440                 -1,
441                 CV_AA);
442         imshow("Display window", matPoint);
443         waitKey(1000);
444         matPoint = Mat(Size(width, height), CV_8UC3);
445         matPoint.setTo(Scalar(255, 255, 255));
446         circle(matPoint,
447                 scr_pnt_down,
```

```
448                 10.0,
449                 Scalar(0, 0, 0),
450                 -1,
451                 CV_AA);
452         imshow("Display window", matPoint);
453         waitClick();
454         circle(matPoint,
455                 scr_pnt_down,
456                 10.0,
457                 Scalar(0, 0, 255),
458                 -1,
459                 CV_AA);
460         imshow("Display window", matPoint);
461         waitKey(1000);
462         Mat img = display_img("Diapositiva4.PNG");
463         vector<Point2f> pnts_img;
464         get_centroids(img, pnts_img);
465         Mat centroids, best_labels, centers, centroids32f;
466
467         centroids = read_file("cali.csv");
468         fix1_mat = read_file("fix1.csv");
469         fix2_mat = read_file("fix2.csv");
470         fix_1.clear();
471         fix_2.clear();
472         for (int i = 0; i < fix1_mat.rows; i++)
473         {
474                 double *ptr = fix1_mat.ptr<double>(i);
475                 fix_1.push_back(Point2f(ptr[0], ptr[1]));
476         }
477         for (int i = 0; i < fix2_mat.rows; i++)
478         {
479                 double *ptr = fix2_mat.ptr<double>(i);
```

```
480                 fix_2.push_back(Point2f(ptr[0], ptr[1]));
481         }
482         std::sort(fix_1.begin(), fix_1.end(), [=](const cv::Point2f
               &a, const cv::Point2f &b) {
483                 return (norm(a - scr_pnt_up) < norm(b - scr_pnt_up))
                       ;
484         });
485         std::sort(fix_2.begin(), fix_2.end(), [=](const cv::Point2f
               &a, const cv::Point2f &b) {
486                 return (norm(a - scr_pnt_down) < norm(b -
                       scr_pnt_down));
487         });
488         centroids.convertTo(centroids32f, CV_32F);
489         kmeans(centroids32f.col(1), 3, best_labels, TermCriteria(
               CV_TERMCRIT_ITER | CV_TERMCRIT_EPS, 10000, 0.0001), 5,
               KMEANS_PP_CENTERS, centers);
490         Mat red, green, blue;
491         for (int i = 0; i < centroids.rows; i++)
492         {
493                 double *Mi = centroids.ptr<double>(i);
494                 int tmp = best_labels.at<int>(i);
495                 switch (tmp)
496                 {
497                 case 0:
498                         circle(img, Point(Mi[0], Mi[1]), 2.0, Scalar
                               (0, 0, 255), -1, CV_AA);
499                         red.push_back(centroids.row(i));
500                         break;
501                 case 1:
502                         circle(img, Point(Mi[0], Mi[1]), 2.0, Scalar
                               (0, 255, 0), -1, CV_AA);
503                         green.push_back(centroids.row(i));
```

```
504                         break;
505                 case 2:
506                         circle(img, Point(Mi[0], Mi[1]), 2.0, Scalar
                               (255, 0, 0), -1, CV_AA);
507                         blue.push_back(centroids.row(i));
508                         break;
509                 default:
510                         break;
511                 }
512         }
513         vector<Point2f> pnt_in, pnt_red, pnt_blue, pnt_green;
514         pnt_in.push_back(fix_1[static_cast<int>(fix_1.size() / 2)]);
515         pnt_in.push_back(fix_2[static_cast<int>(fix_2.size() / 2)]);
516         for (auto n : pnt_in)
517         {
518                 cout << "punto est " << n.x << ", " << n.y << endl;
519         }
520         cout << endl;
521         vector<double> b(3);
522         Mat yp_red = adjust_line(red, img, pnt_red, pnt_in, b[0]),
               yp_blue = adjust_line(blue, img, pnt_blue, pnt_in, b[1]),
                yp_green = adjust_line(green, img, pnt_green, pnt_in, b
               [2]), line_red, line_blue, line_green;
523
524         /*Sort*/
525         if (b[0] > b[1])
526         {
527                 swap(b[0], b[1]);
528                 swap(pnt_red, pnt_blue);
529         }
530         if (b[0] > b[2])
531         {
```

```
532                 swap(b[0], b[2]);
533                 swap(pnt_red, pnt_green);
534         }
535         if (b[1] > b[2])
536         {
537                 swap(b[1], b[2]);
538                 swap(pnt_blue, pnt_green);
539         }
540         /*sort*/
541
542         hconcat(red.col(0), yp_red, line_red);
543         display_line(line_red, img);
544         hconcat(blue.col(0), yp_blue, line_blue);
545         display_line(line_blue, img);
546         hconcat(green.col(0), yp_green, line_green);
547         display_line(line_green, img);
548         imshow("Display window", img);
549         vector<Point2f> pnts_sensor;
550         pnts_sensor.reserve(pnt_red.size() + pnt_blue.size() +
              pnt_green.size());
551         pnts_sensor.insert(pnts_sensor.end(), pnt_red.begin(),
              pnt_red.end());
552         pnts_sensor.insert(pnts_sensor.end(), pnt_blue.begin(),
              pnt_blue.end());
553         pnts_sensor.insert(pnts_sensor.end(), pnt_green.begin(),
              pnt_green.end());
554
555         /*Asosiacion*/
556         for (int i = 0; i < pnts_sensor.size(); i++)
557         {
558                 line(img, pnts_sensor[i], pnts_img[i], Scalar(0, 0,
                      255), 1, CV_AA);
```

```
559         }
560         /*Asosiacion*/
561
562         Eigen::Matrix<double, 11, 6> a;
563         a << 1, pnts_sensor[0].x, pnts_sensor[0].y, pnts_sensor[0].x
                * pnts_sensor[0].x, pnts_sensor[0].y * pnts_sensor[0].y,
                pnts_sensor[0].x * pnts_sensor[0].y,
564                 1, pnts_sensor[1].x, pnts_sensor[1].y, pnts_sensor
                        [1].x * pnts_sensor[1].x, pnts_sensor[1].y *
                        pnts_sensor[1].y, pnts_sensor[1].x * pnts_sensor
                        [1].y,
565                 1, pnts_sensor[2].x, pnts_sensor[2].y, pnts_sensor
                        [2].x * pnts_sensor[2].x, pnts_sensor[2].y *
                        pnts_sensor[2].y, pnts_sensor[2].x * pnts_sensor
                        [2].y,
566                 1, pnts_sensor[3].x, pnts_sensor[3].y, pnts_sensor
                        [3].x * pnts_sensor[3].x, pnts_sensor[3].y *
                        pnts_sensor[3].y, pnts_sensor[3].x * pnts_sensor
                        [3].y,
567                 1, pnts_sensor[4].x, pnts_sensor[4].y, pnts_sensor
                        [4].x * pnts_sensor[4].x, pnts_sensor[4].y *
                        pnts_sensor[4].y, pnts_sensor[4].x * pnts_sensor
                        [4].y,
568                 1, pnts_sensor[5].x, pnts_sensor[5].y, pnts_sensor
                        [5].x * pnts_sensor[5].x, pnts_sensor[5].y *
                        pnts_sensor[5].y, pnts_sensor[5].x * pnts_sensor
                        [5].y,
569                 1, pnts_sensor[6].x, pnts_sensor[6].y, pnts_sensor
                        [6].x * pnts_sensor[6].x, pnts_sensor[6].y *
                        pnts_sensor[6].y, pnts_sensor[6].x * pnts_sensor
                        [6].y,
```

```
570                 1, pnts_sensor[7].x, pnts_sensor[7].y, pnts_sensor
                       [7].x * pnts_sensor[7].x, pnts_sensor[7].y *
                       pnts_sensor[7].y, pnts_sensor[7].x * pnts_sensor
                       [7].y,
571                 1, pnts_sensor[8].x, pnts_sensor[8].y, pnts_sensor
                       [8].x * pnts_sensor[8].x, pnts_sensor[8].y *
                       pnts_sensor[8].y, pnts_sensor[8].x * pnts_sensor
                       [8].y,
572                 1, pnt_in[0].x, pnt_in[0].y, pnt_in[0].x * pnt_in
                       [0].x, pnt_in[0].y * pnt_in[0].y, pnt_in[0].x *
                       pnt_in[0].y,
573                 1, pnt_in[1].x, pnt_in[1].y, pnt_in[1].x * pnt_in
                       [1].x, pnt_in[1].y * pnt_in[1].y, pnt_in[1].x *
                       pnt_in[1].y;
574         Eigen::VectorXd x(11);
575         x << pnts_img[0].x, pnts_img[1].x, pnts_img[2].x, pnts_img
               [3].x, pnts_img[4].x, pnts_img[5].x, pnts_img[6].x,
               pnts_img[7].x, pnts_img[8].x, scr_pnt_up.x, scr_pnt_down.
               x;
576         Eigen::VectorXd y(11);
577         y << pnts_img[0].y, pnts_img[1].y, pnts_img[2].y, pnts_img
               [3].y, pnts_img[4].y, pnts_img[5].y, pnts_img[6].y,
               pnts_img[7].y, pnts_img[8].y, scr_pnt_up.y, scr_pnt_down.
               y;
578         Eigen::MatrixXd Q, R;
579         Eigen::VectorXd thetaX_Eigen, thetaY_Eigen;
580         thetaX_Eigen = Eigen::VectorXd::Zero(6);
581         thetaY_Eigen = Eigen::VectorXd::Zero(6);
582         Eigen::HouseholderQR<Eigen::MatrixXd> qr(a);
583         Q = qr.householderQ();
584         R = qr.matrixQR().triangularView<Eigen::Upper>();
585         Eigen::MatrixXd QtX = Q.transpose() * x;
```

```
586         Eigen::MatrixXd QtY = Q.transpose() * y;
587         for (int i = (R.cols() - 1); i > -1; i--)
588         {
589                 thetaX_Eigen(i) = (QtX(i, 0) - R.row(i).dot(
                        thetaX_Eigen)) / R(i, i);
590                 thetaY_Eigen(i) = (QtY(i, 0) - R.row(i).dot(
                        thetaY_Eigen)) / R(i, i);
591         }
592         cout << thetaX_Eigen << endl
593                 << endl
594                 << thetaY_Eigen << endl;
595         vector<double> ex, ey;
596         for (int i = 0; i < a.rows(); i++)
597         {
598                 Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic
                        > tmpx = a.row(i) * thetaX_Eigen;
599                 ex.push_back(tmpx(0, 0));
600                 Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic
                        > tmpy = a.row(i) * thetaY_Eigen;
601                 ey.push_back(tmpy(0, 0));
602         }
603         for (auto n : pnts_img)
604         {
605                 cout << "Punto imagen: " << n.x << ", " << n.y <<
                        endl;
606                 circle(img, n, 5, Scalar(0, 0, 100), -1, CV_AA);
607         }
608         for (auto n : pnts_sensor)
609         {
610                 cout << "Punto sensor: " << n.x << ", " << n.y <<
                        endl;
611         }
```

```
612         for (int i = 0; i < ex.size(); i++)
613         {
614                 circle(img, Point(ex[i], ey[i]), 5, Scalar(255, 0,
                        255), -1, CV_AA);
615                 cout << "Punto estimado: " << ex[i] << ", " << ey[i]
                        << endl;
616         }
617         circle(img, scr_pnt_up, 5, Scalar(0, 0, 100), -1, CV_AA);
618         circle(img, scr_pnt_down, 5, Scalar(0, 0, 100), -1, CV_AA);
619
620         for (int i = 0; i < centroids.rows; i++)
621         {
622                 Eigen::VectorXd a_red(6);
623                 double *pred = centroids.ptr<double>(i);
624                 a_red << 1, pred[0], pred[1], pred[0] * pred[0],
                        pred[1] * pred[1], pred[0] * pred[1];
625                 double redx = a_red.dot(thetaX_Eigen);
626                 double redy = a_red.dot(thetaY_Eigen);
627                 circle(img, Point(redx, redy), 5, Scalar(0, 0, 0),
                        -1, CV_AA);
628                 cout << "\nEstimated point " << redx << ", " << redy
                        << endl;
629         }
630
631         imshow("Display window", img);
632         waitClick();
633
634         /* Minimos cuadrados */
635         Eigen::MatrixXd tmp = a;
636         Eigen::VectorXd thetaXMC(6), thetaYMC(6);
637         thetaXMC = (tmp.transpose() * tmp).inverse() * tmp.transpose
                () * x;
```

```
638         thetaYMC = (tmp.transpose() * tmp).inverse() * tmp.transpose
               () * y;
639         cout << "Minimos cuadrados calibracion inicial" << endl <<
               thetaXMC << endl << thetaYMC << endl;
640
641         draw_matrix(thetaX_Eigen, thetaY_Eigen, 0);
642         for (int i = 1; i <= 3; i++)
643         {
644                 vector<Point2f> pnts;
645                 pnts.push_back(Point2f((width / 4)*i, height / 4));
646                 pnts.push_back(Point2f((width / 4)*(4 - i), (height
                       / 4)*3));
647                 incremental(Q, R, thetaX_Eigen, thetaY_Eigen, x, y,
                       pnts, i, tmp);
648                 draw_matrix(thetaX_Eigen, thetaY_Eigen, i);
649         }
650         return 0;
651 }
652
653 int main(int argc, char **argv)
654 {
655         auto_calibrate();
656         return 0;
657 }
```

MyGaze.cpp

```
1 #include "MyGaze.h"
2 #include <iostream>
3 #include <numeric>
4 #include <list>
5 #include <iterator>
```

```
#include <opencv2/core.hpp>
using namespace std;
using namespace cv;
// --- MyGaze implementation
MyGaze::MyGaze()
        : m_api(1), // verbose_level 0 (disabled)
        prev_fix(-1000, -1000)
{
        // Connect to the server on the default TCP port (6555)
        if (m_api.connect())
        {
                // Enable GazeData notifications
                m_api.add_listener(*this);
        }
}

MyGaze::~MyGaze()
{
        m_api.remove_listener(*this);
        m_api.disconnect();
        fixations.clear();
}

void MyGaze::on_gaze_data(gtl::GazeData const &gaze_data)
{
        if (gaze_data.state & gtl::GazeData::GD_STATE_TRACKING_GAZE)
        {
                gtl::Point2D const &smoothedCoordinates = gaze_data.
                    avg;
                // Move GUI point, do hit-testing, log coordinates,
                    etc.
```

```
35                  gd.push_back(Point2f(gaze_data.avg.x, gaze_data.avg.
                       y));
36                  if (gd.size() == 10)
37                  {
38                          analyze(gd);
39                  }
40          }
41  }
42
43  void MyGaze::analyze(list<Point2f> &points)
44  {
45          vector<Point2f> p{ begin(points), end(points) };
46          vector<float> x, y;
47          transform(p.begin(), p.end(), back_inserter(x), [](Point2f
               const &pnt) { return pnt.x; });
48          transform(p.begin(), p.end(), back_inserter(y), [](Point2f
               const &pnt) { return pnt.y; });
49          sort(x.begin(), x.end());
50          sort(y.begin(), y.end());
51          Point2f median(x[5], y[5]);
52          int counter = 0;
53          for (auto n : p)
54          {
55                  double dist = norm(n - median);
56                  if (dist <= 30)
57                  {
58                          counter++;
59                          if (counter == 5)
60                          {
61                                  double dist_prev = norm(median -
                                       prev_fix);
62                                  if (dist_prev >= 30)
```

```
                                {
                                        fixations.push_back(median);
                                        prev_fix = median;
                                }
                        }
                }
        }
        points.pop_front();
}

list<Point2f> MyGaze::get_fixations()
{
        return fixations;
}

vector<Point2f> MyGaze::get_readings()
{
        return{ begin(gd), end(gd) };
}

vector<Point2f> MyGaze::get_fixations_vec()
{
        return{ begin(fixations), end(fixations) };
}
```

MyGaze.h

```
#include <gazeapi.h>
#include <list>
#include <opencv2/core.hpp>
using namespace std;
using namespace cv;
```

```
6  // --- MyGaze definition
7  class MyGaze : public gtl::IGazeListener
8  {
9  public:
10         MyGaze();
11         ~MyGaze();
12         list<Point2f> get_fixations();
13         vector<Point2f> get_readings();
14         vector<Point2f> get_fixations_vec();
15
16 private:
17         // IGazeListener
18         void on_gaze_data(gtl::GazeData const &gaze_data);
19         void analyze(list<Point2f> &points);
20
21 private:
22         gtl::GazeApi m_api;
23         list<Point2f> gd;
24         list<Point2f> fixations;
25         Point2f prev_fix;
26 };
```