INDEX

Proyecto en Inglés

1. Introduccion

La memoria del proyecto esta dividida en 4 partes:

-La primera es un resumen en español de las 3 partes siguientes, que componen el proyecto que se realizó en Munich para la Fachhochschule, Universidad de Ciencias Aplicadas.

-La segunda es donde comienza el proyecto. Consta de una introducción, así como de la explicación de todo lo realizado en el proyecto: contenido, dónde se realizó, material utilizado, tests realizados, conclusiones, etc.

-La tercera parte la compone el código comentado de las 2 funciones realizadas en el proyecto.

-La cuarta y última parte se compone de un manual de Khoros Cantata. De este modo se puede entender un poco con que se realizo el proyecto. Se adjuntan capturas del programa para que tener una idea de cómo aparece la información en él. A parte será de mucha utilidad para el alumno que siga con este proyecto. Le facilitará la toma de contacto con el programa.

2. El Proyecto

Mi proyecto consistió en hacer un programa en “C” para reconocer caras. El proyecto se realizó bajo entorno Linux, y con un programa dedicado al tratamiento de imágenes Khoros Cantata. La técnica usada fue la correlación cruzada. Esta técnica consiste en la comparación píxel a píxel de 2 imágenes. En nuestro caso las imágenes eran fotografías de caras de personas. Con este proceso de comparación se obtienen unos coeficientes. El valor de estos coeficientes oscila entre “0” y “1”. Cuando el valor de los píxeles comparados es el mismo, se obtiene un “1”. En caso contrario, cuando el valor de los píxeles es opuesto pues se obtiene un “0”. Este valor que se obtiene se denomina “Coeficiente de Correlación” y se representa con una “r”. Hay varias fórmulas para realizar esta función. En este proyecto hemos elegido la siguiente:

$$r = \frac{\frac{1}{n}\sum_{i=1}^{n}\left[\left(X_i - \overline{X}\right)\left(Y_j - \overline{Y}\right)\right]}{\sqrt{\frac{1}{n}\sum_{1=1}^{n}\left(X_i - \overline{X}\right)^2}\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \overline{Y}\right)^2}}$$

La variable “X” representa el valor de los píxeles de la Foto 1, e “Y” el valor del píxeles de la Foto 2. La variable “X” con una barra en su parte superior representa la media de la Foto. Es decir, la suma del valor de todos los píxeles de la Foto, dividido por el número total de píxeles que tiene la Foto. El valor del coeficiente de correlación se puede representar también con un porcentaje. Para esto bastará con multiplicar por 100 el valor de “r”. Usamos esta representación de “r” por considerar que era más clara.

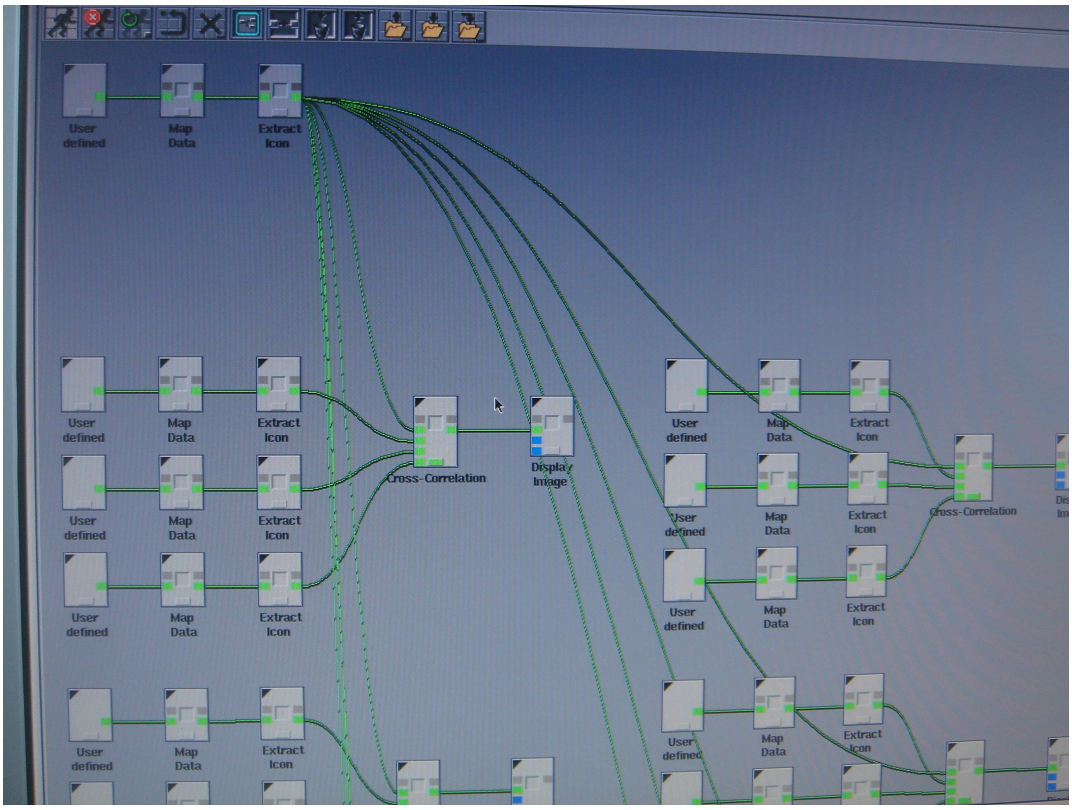
El primer paso es la toma de la Fotos para la realización de la Base de Datos. En nuestro caso ésta estaba compuesta por 30 Fotos. De éstas cada 3 correspondían al mismo individuo. Las Fotos se tomaban en color con una cámara Sony, pero para trabajar con ellas en el PC era necesario convertirlas a 256 tonos de gris. Este proceso de conversión se hacía con el programa KS-400 bajo entorno Windows. El programa KS-400 pertenece a Carl-Zeiss. Esta empresa es muy conocida por sus lentes para cámaras. La conversión se hace con una función de este programa. Una vez están todas las fotos convertidas, se la gira con un programa, de manera que aparezcan las caras en sentido vertical. Esto último se realiza para facilitar el visionado de las Fotos.

Una vez las fotos están en el ordenador con Linux, comenzamos a comentar la primera de las funciones en “C” realizadas. La función se denomina “**Icon.c**”. El propósito de esta función es recortar la cara de la foto original. Pero lo más importante es que la cara estará centrada, porque puede que en la foto original, el individuo no este en el centro de la imagen. El funcionamiento del algoritmo lo podríamos dividir en 2 pasos. El primer paso consiste en encontrar la cara y el segundo en extraer la porción de imagen seleccionada. El tamaño de la zona recortada es fijo y es 310*410 pixeles. Este tamaño se hizo probando diferentes tamaños y viendo cual de ellos se ajustaba

mejor. El segundo paso, consistente en recortar la zona deseada, se realiza con una función ya programada del programa Khoros Cantata. El único dato que hay que pasarle a la función es el punto de comienzo de la zona deseada.

En estas fotos se puede ver el resultado de la Función Icon:

La siguiente foto muestra la pantalla principal del programa Khoros Cantata



En ella se pueden apreciar diferentes “cajas”. Cada una de ellas representa una función. En la parte superior de la foto aparecen 3 de estas “cajas”. Corresponden con la entrada de la foto del individuo que se desea conocer su identidad. El resto de las “cajas” se encargan de procesar la base de datos. Volviendo a las 3 cajas de la parte superior, tienen como nombres: User defined, Map Data y Extract Icon. La función User defined es usada para introducir una foto en el programa. En el grupo de cajas de la parte superior, esta función se encargará de cargar la foto de la persona desconocida. En los otros grupos de cajas, irá cargando las fotos de cada uno de los miembros de la base de datos. Podemos distinguir grupos de varias cajas, dispuestos de una manera simétrica por toda la pantalla del programa. Cada grupo corresponde a un individuo de la base de datos. En la ejecución del programa solo hay que ir cambiando la foto del individuo desconocido, el resto ya se encuentra configurado. Después de haber cargado todas las fotos, éstas pasan a través de una función denominada Map Data. Como se comentó anteriormente las

fotos se encuentran en 256 tonos de grises. El problema surge cuando ese valor entre 0-255 no aparece en la foto. En su lugar aparece otro número. Éste hace referencia a una posición de memoria donde se encuentra en verdadero valor del píxel. Para poder leer los valores correctos de los píxeles de una forma eficaz y sin complicar mucho el algoritmo hay que invertir el proceso de Mapeo. De esto se encarga esta función. A la salida ya tenemos las fotos listas para ser procesadas por la primera función del proyecto. Las otras 2 funciones ya vienen en el programa.

La función Extract Icon se ejecuta 31 veces, 1 vez por cada Foto. A partir de ahora nos referiremos con el nombre de Icono a la parte de foto que extraeremos de la fotografía original. La función como ya comenté consta de 2 pasos: encontrar el Punto de Extracción y extraer el Icono. La búsqueda de ese punto, se dividirá en encontrar sus 2 coordenadas.

La más fácil de las 2 es encontrar la coordenada Y. Consiste en encontrar el primer píxel de la foto con un valor menor o igual a 175. En ese momento se detiene la búsqueda. El motivo de la elección de ese valor es que todas las fotos se realizaron sobre un fondo blanco (el blanco tiene un valor de 255). El valor de los píxeles del fondo de la foto oscila en torno a 200. En resumen, se busca el pelo de la persona. Una vez, el algoritmo ha encontrado ese punto, se le resta 10, porque deseábamos un poco de espacio entre la cara y el borde del icono.

Y' es el valor original de la coordenada y, donde se encontró el píxel con un valor menor a 175. Y corresponde con el valor final de la coordenada y:

$$Y = Y' - 10$$

Ahora el segundo paso: encontrar la coordenada x. Este paso es un poco mas laborioso que la búsqueda de la coordenada y. Esta vez tendremos que localizar 2 puntos, uno en el lado derecho de la cara (X'') y otro en el lado izquierdo (X').

X representa la diferencia entre estos 2 valores. En otras palabras, será el ancho de la cara:

$$X = X'' - X'$$

Con este valor, calcularemos, lo que hemos denominado en el proyecto, el off-set. Este valor será la distancia entre la cara y los bordes del icono. Todo este proceso se hace para mantener la misma distancia a ambos lados de la cara. Buscamos una simetría, de esta manera será más eficiente la correlación posteriormente. La fórmula usada para calcular la coordenada x es la siguiente:

$$Width = X'' - X'$$

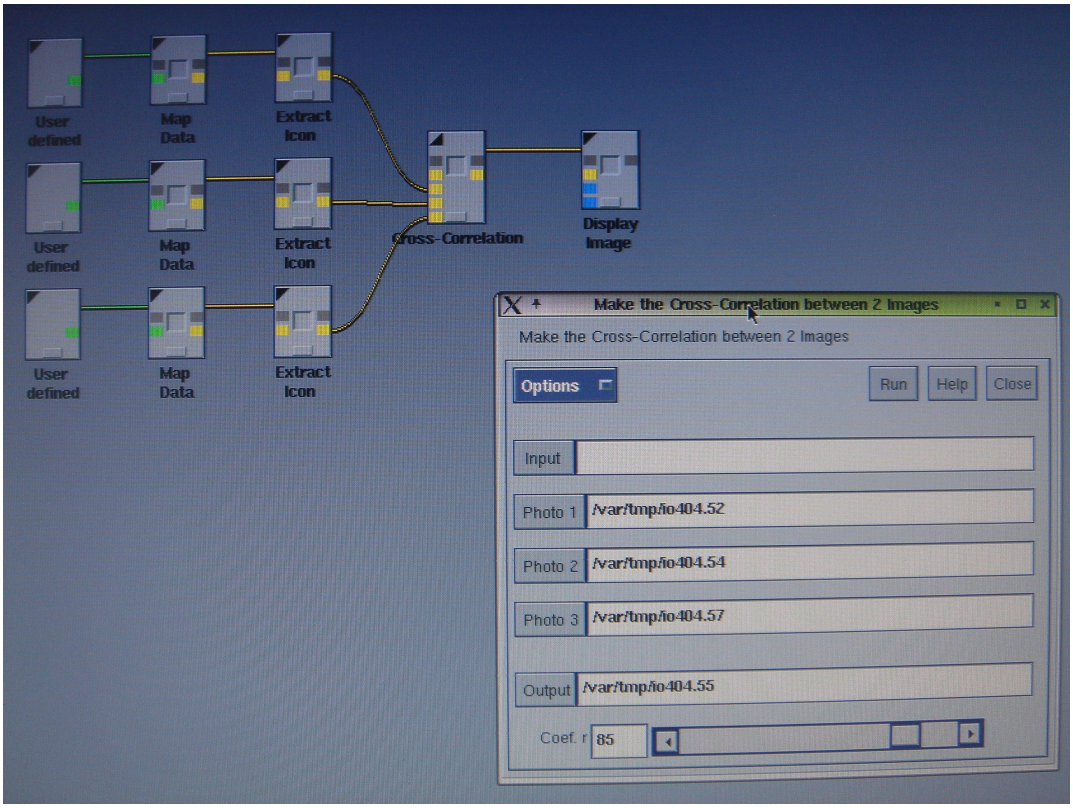
$$Offset = (310 - Width)/2$$

$$X = X' - Offset$$

El resultado del proceso es:

Z = Extraction Point

El siguiente bloque corresponde con la segunda función desarrollada en el proyecto, la función correlación. Se puede ver en la siguiente foto, el aspecto de la interfaz de la función:



Solo hay un parámetro modificable. Aparece con el nombre de “Coef r” y tiene un valor en la foto de 85 (este valor es el que se puede modificar).

Esta función tiene 4 entradas (4 iconos). 3 de estos iconos corresponden con una persona de la base de datos. El otro icono es de la persona desconocida que se desea identificar.

La fórmula usada fue:

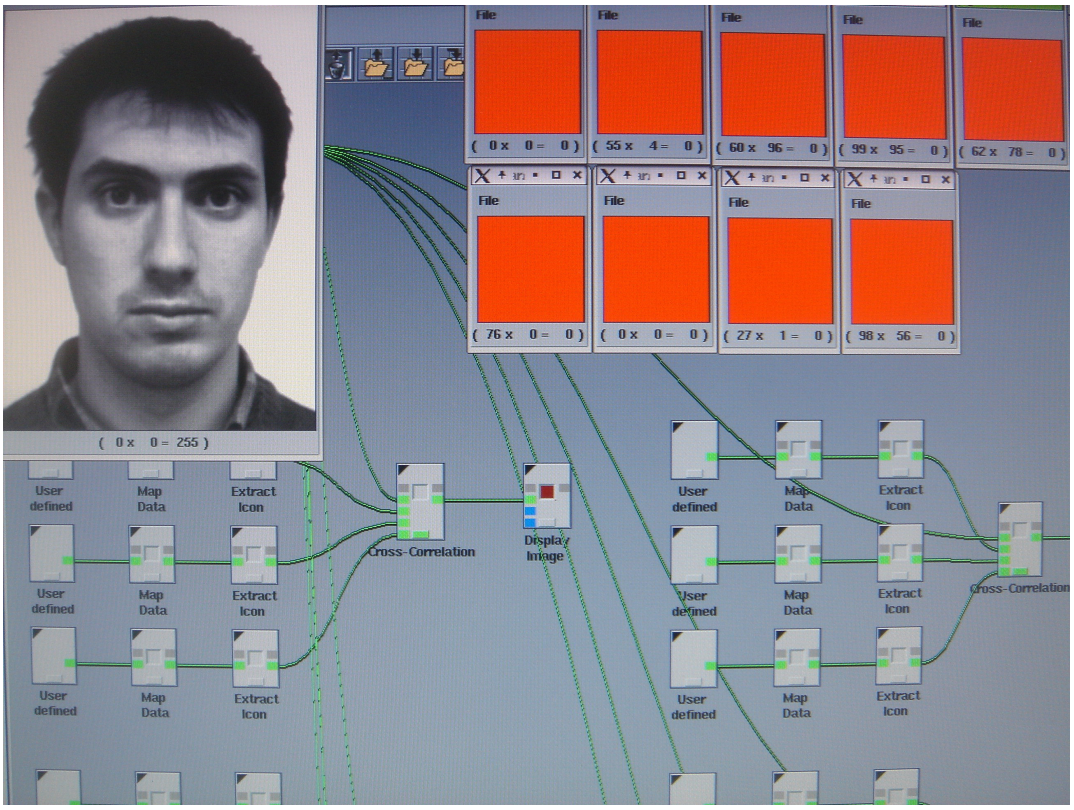
$$r = \frac{\frac{1}{n}\sum_{i=1}^{n}\left[\left(X_i - \overline{X}\right)\left(Y_j - \overline{Y}\right)\right]}{\sqrt{\frac{1}{n}\sum_{1=1}^{n}\left(X_i - \overline{X}\right)^2}\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \overline{Y}\right)^2}}$$

La variable “X” representa el valor de los pixeles de la Foto 1, e “Y” el valor del pixeles de la Foto 2. La variable “X” con una barra en su parte superior representa la media de la Foto. Es decir, la suma del valor de todos los píxeles de la Foto, dividido por el número total de píxeles que tiene la Foto.

El primer paso es calcular la media de los iconos. Este proceso consiste en sumar el valor de todos los píxeles de la foto y dividir entre el número total de píxeles. Obtenemos un número de este proceso.

El segundo paso es calcular el numerador y denominador de la función. Este proceso se hace para cada una de los iconos. En total, hacemos 3 correlaciones por cada individuo de la base de datos. Y obtenemos 3 coeficientes de correlación. Estos valores obtenidos se comparan con el límite fijado en la función. Cuando al menos 1 de estos valores es superior al límite, tenemos una comparación positiva. En caso contrario, obtenemos una comparación negativa.

El resultado del programa completo, se representa en pantalla de la siguiente manera:



La persona desconocida en este caso era Jose Juan. Cuando el programa encuentra una persona, cuyo coeficiente “r” es mayor que el límite que nosotros impusimos, entonces el resultado de la comparación es positivo y la foto de la persona de la base de datos es mostrada. En caso contrario, aparece un cuadrado rojo. En total aparecen en pantalla un numero de objetos (cuadrados o fotos) igual al número de individuos de la base de datos.

Pusimos un valor de r = 85%. Con este valor, el programa no fallaba, y solo mostraba al final la foto de un individuo. Se empezó el testeo del programa con un valor de 75%. Pero con este valor, había fallos de identificación con algunos individuos de la base de datos.

3. Conclusiones

Con estos tests que hicimos llegamos a las siguientes conclusiones:

- La Base de Datos debe ser tan grande como se pueda.
- Las fotos que tiene cada individuo en la Base de Datos deben representar diferentes situaciones de luz, proximidad del individuo a la cámara, ángulos de ejecución de las fotos, etc…

Con estas 2 ideas, la identificación será mejor. Aunque conlleve un aumento considerable en la carga computacional.

INDEX

1. Introduction

Humans have used body characteristics such as face, voice, etc. for thousands of years to recognize each other. Alphonse Bertillon, chief of the criminal identification division of the police department in Paris, developed and then practiced the idea of using a number of body measurements to identify criminals in the mid 19th century. Just as his idea was gaining popularity, it was obscured by a far more significant and practical discovery of the distinctiveness of the human fingerprints in the late 19th century. Soon after this discovery, many major law enforcement departments embraced the idea of first "booking" the fingerprints of criminals and storing it in a database. Later, the leftover (typically, fragmentary) fingerprints (commonly referred to as *latents*) at the scene of crime could be "lifted" and matched with fingerprints in the database to determine the identity of the criminals. Although biometrics emerged from its extensive use in law enforcement to identify criminals (e.g., security clearance for employees for sensitive jobs, fatherhood determination, forensics, positive identification of convicts and prisoners), it is being increasingly used today to establish person recognition in a large number of civilian applications.

What biological measurements qualify to be a biometric? Any human physiological and/or behavioural characteristic can be used as a biometric characteristic as long as it satisfies the following requirements:

• *Universality*: each person should have the characteristic.
• *Distinctiveness*: any two persons should be sufficiently different in terms of the characteristics.
• *Permanence*: the characteristic should be sufficiently invariant (with respect to the matching criterion) over a period of time;
• *Collectability*: the characteristic can be measured quantitatively.

However, in a practical biometric system (i.e., a system that employs biometrics for personal recognition), there are a number of other issues that should be considered, including:

• *Performance*, which refers to the achievable recognition accuracy and speed, the resources required to achieve the desired recognition accuracy and speed, as well as the operational and environmental factors that affect the accuracy and speed;
• *Acceptability*, which indicates the extent to which people are willing to accept the use of a particular biometric identifier (characteristic) in their daily lives.
• *Circumvention*, which reflects how easily the system can be fooled using fraudulent methods.

A practical biometric system should meet the specified recognition accuracy, speed, and resource requirements, be harmless to the users, be accepted by the intended population, and be sufficiently robust to various fraudulent methods and attacks to the system.

A *biometric system* is essentially a pattern recognition system that operates by acquiring biometric data from an individual, extracting a feature set from the acquired data, and comparing this feature set against the template set in the database. Depending on the application context, a biometric system may operate either in *verification* mode or *identification* mode:

• In the **verification mode**, the system validates a person's identity by comparing the captured biometric data with her own biometric template(s) stored system database. In such a system, an individual who desires to be recognized claims an identity, usually via a PIN (Personal Identification Number), a user name, a smart card, etc., and the system conducts a one comparison to determine whether the claim is true or not (e.g., "*Does this biometric data belong to Jose Juan?*"). Identity verification is typically used for *positive recognition,* where the aim is to prevent multiple people from using the same identity.

• In the **identification mode**, the system recognizes an individual by searching the templates of all the users in the database for a match. Therefore, the system conducts a one-to-many comparison to establish an individual's identity (or fails if the subject is not enrolled in the system database) without the subject having to claim an identity (e.g., "*Whose biometric data is this?*").

Identification is a critical component in *negative recognition* applications where the system establishes whether the person is who he/she (implicitly or explicitly) denies to be. The purpose of negative recognition is to prevent a single person from using multiple identities. Identification may also be used in positive recognition for convenience (the user is not required to claim an identity). While traditional methods of personal recognition such as passwords, PINs, keys, and tokens may work for positive recognition, negative recognition can only be established through biometrics.

Given the requirement for determining people's identity, the obvious question is what technology is best suited to supply this information? There are many different identification technologies available, many of which have been in wide-spread commercial use for years. The most common person verification and identification methods today are Password/PIN (Personal Identification Number) systems, and Token systems (such as your driver's license). Because such systems have trouble with forgery, theft, and lapses in users' memory, there has developed considerable interest in biometric identification systems, which use pattern recognition techniques to identify people using their physiological characteristics. Fingerprints are a classic example of a biometric; newer technologies include retina and iris recognition.

While appropriate for bank transactions and entry into secure areas, such technologies have the disadvantage that they are intrusive both physically and socially. They require the user to position their body relative to the sensor, and then pause for a second to `declare' themselves. This `pause and declare' interaction is unlikely to change because of the fine-grain spatial sensing required. Moreover, there is an `oracle-like' aspect to the interaction: since people can't recognize other people using this sort of data, these types of identification do not have a place in normal human interactions and social structures.

While the “pause and present” interaction and the oracle-like perception are useful in high-security applications (they make the systems look more accurate), they are exactly the opposite of what is required when building a store that recognizes its best customers, or an information kiosk that

remembers you, or a house that knows the people who live there. Face recognition from video and voice recognition have a natural place in these next-generation smart environments -- they are unobtrusive (able to recognize at a distance without requiring a `pause and present' interaction), are usually passive (do not require generating special electro-magnetic illumination), do not restrict user movement, and are now both low-power and inexpensive. Perhaps most important, however, is that humans identify other people by their face and voice, therefore are likely to be comfortable with systems that use face and voice recognition

Face recognition is a task that the human vision system seems to perform almost effortlessly, yet the goal of building computer-based systems with comparable capabilities has proven to be difficult. The task implicitly requires the ability to locate and track faces through often complex and dynamic scenes. Recognition is difficult because of variations in factors such as lighting conditions, viewpoint, body movement and facial expression. Although evidence from psychophysical and neurobiological experiments provides intriguing insights into how we might code and recognise faces, its bearings on computational and engineering solutions are far from clear. The study of face recognition has had an almost unique impact on computer vision and machine learning research at large. It raises many challenging issues and provides a good vehicle for examining some difficult problems in vision and learning. Many of the issues raised are relevant to object recognition in general.

2. Methods of Face-Recognition

In this section, we try to expose the principal methods to make Face-Recognition. We explain the methods, what benefit has each method and the algorithm used.

2.1. The Direct Correlation Method

The direct correlation method of face recognition involves the direct comparison of pixel intensity values taken from facial images. We convert, for example, bitmap images of 65 by 82 pixels into a vector of 5330 elements,

describing a point within a 5330 dimensional image space. By measuring the distance between these points, we gain an indication of image similarity. Similar images are located close together within the image space, while dissimilar images are spaced far apart. Extending this idea to faces, calculating the Euclidean distance d, between two facial image vectors (often referred to as the query image q, and gallery image g), we get an indication of similarity. A threshold is then applied to make the final verification decision.

$$d = \| q - g \|$$

$$d \leq \text{threshold} \rightarrow \text{accept}$$

$$d \geq \text{threshold} \rightarrow \text{reject}$$

2.2. The Eigenface Method

In this section we give a brief explanation of the eigenface method of face recognition. We compute the covariance matrix C, of facial images from a set of M (60) training images: { Γ_1 Γ_2 Γ_3 ...)

$$C = \frac{1}{M}\sum_{n=1}^{M}\Phi_n\Phi_n^T = AA^T$$

$$A = [\Phi_1\Phi_2\Phi_3...\Phi_M]$$

$$\Phi_n = \Gamma_n - \Psi$$

$$\Psi = \frac{1}{M}\sum_{n=1}^{M}\Gamma_n$$

The eigenvectors and eigenvalues of this covariance matrix are calculated using Standard linear methods and the M` eigenvectors with the highest eigenvalues chosen to formulate the projection matrix u. For the sake of consistency with the fisherface method, we use the first 59 principal components when testing the eigenface method.

An ace-key ω (image vector projected into ace space) can then be produced by the following equation.

$$\omega_k = u_k^T (\Gamma - \Psi)$$

for k = 1 to M

These face-keys (vectors of 59 principal component coefficients) can then be compared using the Euclidian distance measure as with the direct correlation method.

2.3. The Fisherface Method

The fisherface method of face recognition uses both principal component analysis and linear discriminated analysis to produce a subspace project in matrix, similar to that used in the eigenface method.

To accomplish this we expand the training set to contain multiple images of each person, providing examples of how a person-face may change from one image to another due to variations in lighting conditions, facial expressions and even small changes in orientation. We define the training set as,

$$Training - set = \{\Gamma_1 \Gamma_2 \Gamma_3 ... \Gamma_M\}$$

Where Γ_i is a facial image and the training set is partitioned into c classes, such that all the images in each class X_i are of the same person and no single person is present in more than one class.

We begin by computing three scatter matrices, representing the within-class (S_w), between-class (S_b) and total (S_t) distribution of the training set throughout image space.

$$S_T = \sum_{n=1}^{M} (\Gamma_n - \Psi)(\Gamma_n - \Psi)^T$$

$$S_B = \sum_{i=1}^{c} |X_i| (\Psi_i - \Psi)(\Psi_i - \Psi)^T$$

$$S_W = \sum_{i=1}^{c} \sum_{\Gamma \in X} (\Gamma_K - \Psi_i)(\Gamma_k - \Psi_i)^T$$

Where $\Psi = \frac{1}{M} \sum_{n=1}^{M} \Gamma_n$, is the average image vector of the entire training set, and $\Psi_i = \frac{1}{|X_i|} \sum_{\Gamma_i \in X_i} \Gamma_i$, the average of each individual class X_i. By performing PCA on the total scatter matrix S_t, and taking the top M-c principal components, we produce a projection matrix U_{PCA}, which is used to reduce the dimensionality of the within-class scatter matrix, ensuring it is non-singular, before computing the top c-1 (in this example 59) eigenvectors of the reduced scatter matrices, U_{fld} as shown below.

$$U_{fld} = \arg\max_U \left(\frac{\left| U^T U_{pca}^T S_B U_{pca} U \right|}{\left| U^T U_{pca}^T S_W U_{pca} U \right|} \right)$$

Finally, the matrix U_{ff} is calculated as shown in next equation, such that it will project a facial image into a reduced image space of c-1 dimensions, in which the between-class scatter is maximised for all c classes, while the within-scatter is minimised for each class X_i

$$U_{ff} = U_{fld} U_{pca}$$

Once the matrix U_{ff} has been constructed it is used in much the same way as the projection matrix in the eigenface system, reducing the dimensionality of the image vectors from 5330 to just (c-1) elements. Again, like the eigenface system, the components of the projection matrix can be viewed as images, referred to as fisherfaces.

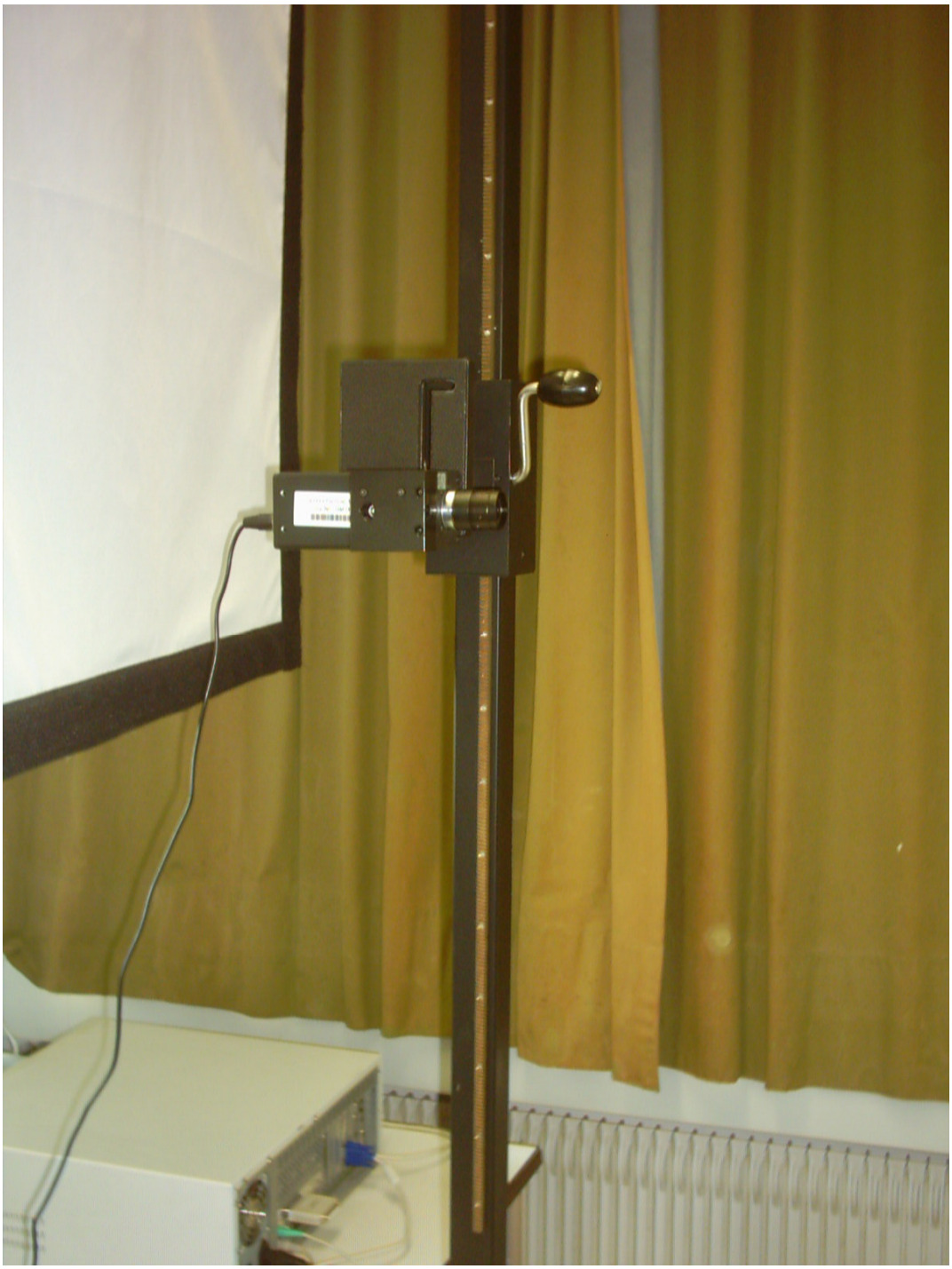
3. The Laboratory

This work has been performed in the IA Laboratory of Professor Johannes Jaschul. This Laboratory was located in the Fachhochschule, in Lothstrasse in München.

The Laboratory has around 30 square meters. It has several computers and other devices.

We worked with a Sony camera. It is a colour camera, but after taking the photographs, we transformed them into grey scale values in order to simplify the work.

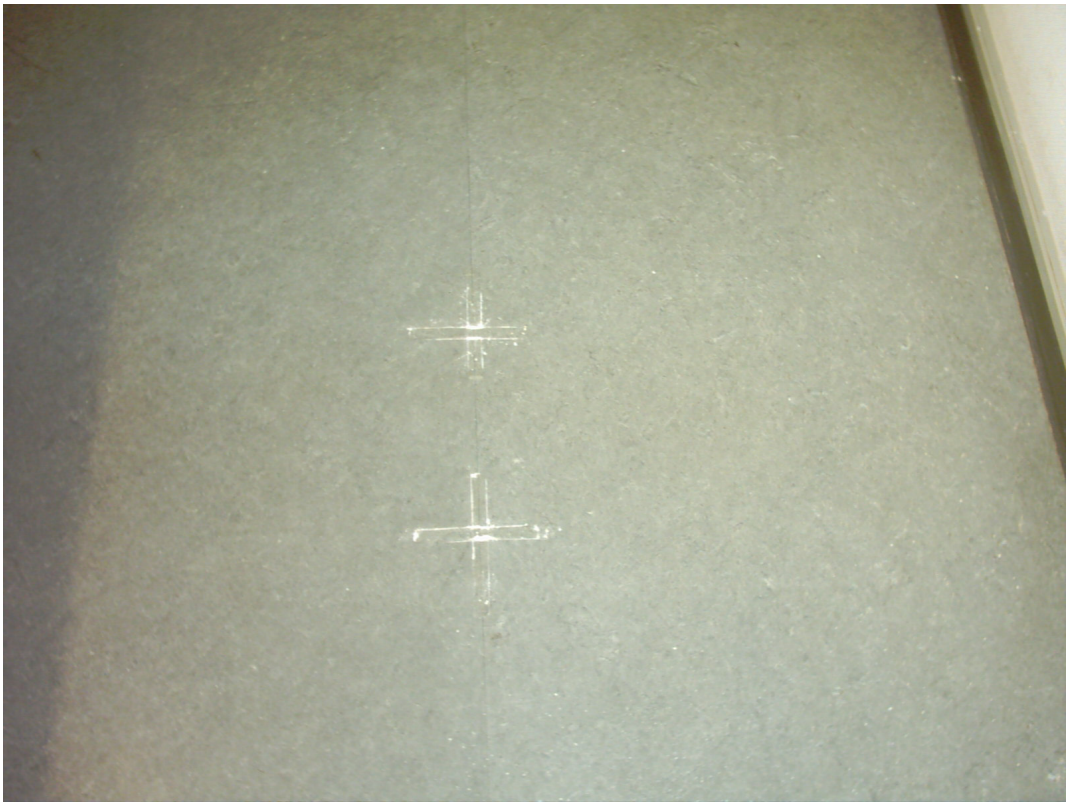
Some photos we took in the Laboratory are shown below:



In the last photo, we can see the part of our Laboratory, where we took the photos. This corresponds to a corner of the Laboratory. The black structure is where the camera is. With the crank we could adjust the height of the camera.

That depended of the person. The camera is not in the good position and took the photos with a turn of 90º. We solved this problem with the Paint program in the computer.

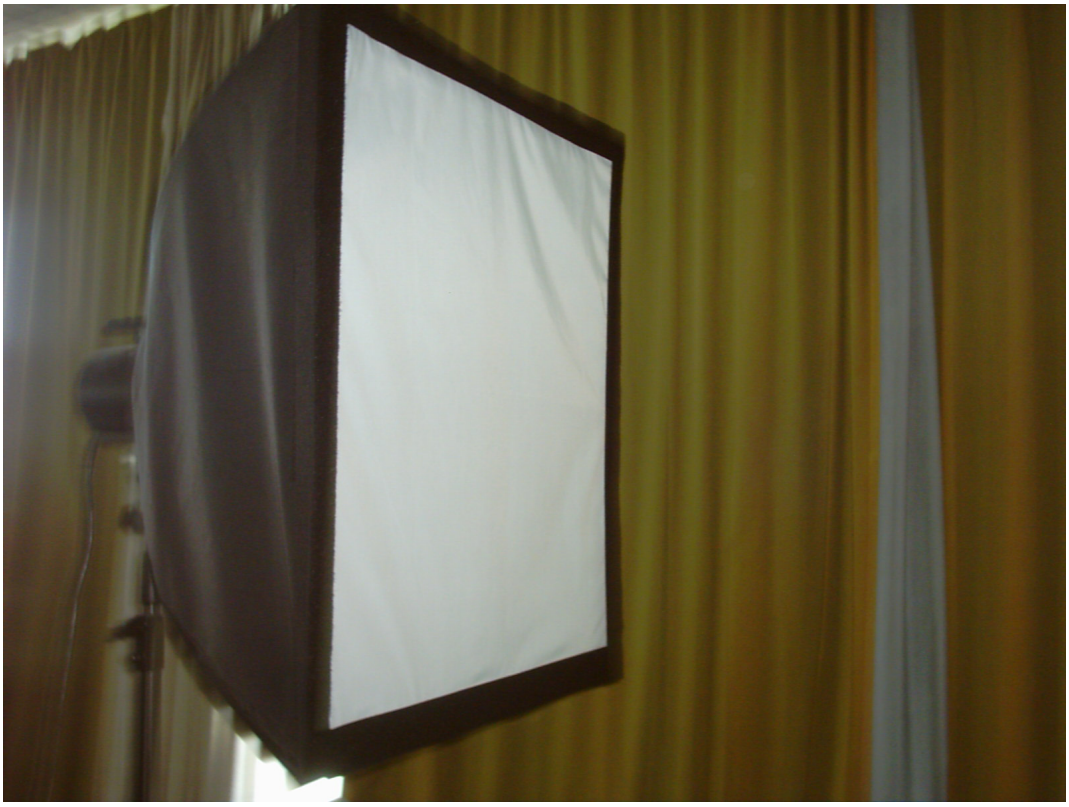
This photo shows where we took the photos too, but from other angle. We worked with the screen used for the projections. This way we had a white background in our photos and the face was shown clearly.



This photo is quite interesting. Here we can appreciate the marks on the floor that we had to make. This way all the photos that we took had the same distance to the camera. For this project it represents the difference between a positive or a negative result. We were very careful with this.

Here is one of the lights of the Laboratory. To get good photographs, which we could appreciate, all the details of the face, we needed more light because only with natural light was not enough.

This image represents the Spotlight used to take photographs. It is a professional Focus with a lot of power.



The Spotlight photographed from another angle.

4. Data Base

In this point we introduce the Date Base used to test the program. It consists of ten people, and we took 3 different photos of each one. The sizes of the photos were 512x512 pixels with 256 levels of grey. Here one photo of each person is shown:

5. Equipment

We worked with a computer with Linux. The way we showed the results in the Pc-Screen was very visual. When the program recognized the person, it showed this person. And in the Screen was shown one of the photos of the data-base. When the Recognition-Process was negative a little red frame was shown. So, after each test, ten frames were in the screen. The best situation was when only one photograph was shown and the rest of them were red frames.

6. My Project

My project consists in make a program to recognize Faces. I worked in Linux with a program call Khoros Cantata. The technique used was Cross-Correlation. This technique consists in compare pixel by pixel two pictures. In our case this pictures were photos. And with this process it is obtained coefficients. The value of these coefficients is between “0” and “1”. When the value of the two pixels, that they are being compared, is the same, we obtained a value of “1”. A value of “0” is obtained in the opposite case. This value that we obtained is called “Correlation Coefficient” and it is represented wit a letter “r”. There are several formulas to get the correlation coefficient. I used this:

$$r = \frac{\frac{1}{n}\sum_{i=1}^{n}\left[\left(X_i - \overline{X}\right)\left(Y_j - \overline{Y}\right)\right]}{\sqrt{\frac{1}{n}\sum_{1=1}^{n}\left(X_i - \overline{X}\right)^2}\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \overline{Y}\right)^2}}$$

Variable “X” corresponds with the values of the pixels of Photo 1, and “Y” of the Photo 2. The variable “X” with a hyphen on it is the mean value of the entire photo 1. The variable “Y” with a hyphen on it is the mean value of the entire photo 2. We have to calculate them before use this formula. The parameter “r” could be represented with a percent. To do this, we have only to multiply the coefficient “r” with the number 100. I used this representation of “r” because I think it is clearer.

We start with the photos that we took in the Laboratory. The first step is to convert these photos in colour to grey-scale photos (256 tones of grey). We do this with the program KS-400 in a PC with Windows. KS-400 is a program of Carl-Zeiss. It is prepared to work with photos. There is a function to do this process. Then we take the photos to the PC with Linux.

When we have the photos ready, we start with the first C-function. The first function that I had to program is called Icon.c. The purpose of this function is to take a frame (with a specific size) of the original photo. With this operation we get only the face of the person of the photo. But the most important thing, it is that we get the face centred. How works the algorithm? It does two steps. The first step is to find the face and the second step is to extract the frame. The size of this frame is set to 310*410 pixels. This size was chosen because it fits very well whit the size of the faces of the Data Base. The second step is very easy because the program Khoros Cantata gives us a function that it makes all. The only thing that we have to get for this function is a point. I call this point “Extract Point”. This point is in the left-upper corner of the frame.

In these photos we could see the result of the Function Icon.

(FOTO DEL ESPACIO DE TRABAJO DEL PROGRAMA, APARECE EN EL RESUMEN EN ESPAÑOL)

This Photo shows how the workspace of the whole program is. But now I will use to explain the Extract Icon Box.

In the upper side of the photo appear three boxes: User defined, Map Data and Extract Icon. The User defined box is to put a photo in the program. You have to choose which Photo you want to compare with the photos of the Data Base. The Map Data Photo was not in the original program but I had to use it. It is a function to transform the values that they appear in the pixel. In the beginning the value of the pixel was not a number between 0 and 255. Instead of these values there were the values of a memory position. With this function we change this. After that is the Extract Icon Function. As can be see here we used this function for each photo. Totally this function is used 31 times, one time pro Photo.

Now I describe the function in detail. As I said before, the function has two steps. The first step is to find the Extraction Point. This action takes 2 steps more: to find “X” coordinate and “Y” coordinate.

The Search of the Y coordinate is easy. It consists in to find a pixel, whose value is lower than 175. When this happens, it adds 10 to this value and this is the final value of the coordinate Y of the Extraction Point.

Y’ is the original value of the Y coordinate, where the value lower than 175 was found. Y is the final value of the Y coordinate and it is calculated:

$$Y = Y' - 10$$

The number 10 was chosen because we wanted a bit of space between the upper border of the frame and the face.

Now the second step, find the X coordinate. This step is a bit more complicate than the Y coordinate. We find 2 points this time: the Minimum Value in the Left Side of the Face (X') and the Minimum Value of the Right Side of the Face (X'').

X represents the difference between X' and X'':

$$X = X'' - X'$$

With this value we calculate the off-set (Distance between the face and the border of the frame) that we will put in the frame. This way the face appears in the centre of it. We make that to keep the same distance between the frame and the face, in both sides of the Icon. All this process is to try to keep symmetry in the Icon (This is the name that we use for the frame).

The formula used to calculate the Final X Coordinate is:

$$Width = X'' - X'$$

$$Offset = (310 - Width)/2$$

$$X = X' - Offset$$

The result of this process is:

Z = Extraction Point

The next is the function Correlation.

(FOTO DEL ASPECTO DE LA INTERFAZ DE LA FUNCIÓN CORRELACION, APARECE EN EL RESUMEN EN ESPAÑOL)

There is only one modifiable parameter. It appears in the photo with the label Coef. r and it has a value of 85 (this value could be modified).

This function hat four inputs (four icons). Three of these icons correspond with a person of the data base (each person of the data base hat three different photos). The other icon corresponds with the unknown person.

The formula used was:

$$r = \frac{\frac{1}{n}\sum_{i=1}^{n}\left[\left(X_i - \overline{X}\right)\left(Y_j - \overline{Y}\right)\right]}{\sqrt{\frac{1}{n}\sum_{1=1}^{n}\left(X_i - \overline{X}\right)^2}\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \overline{Y}\right)^2}}$$

Variable “X” corresponds with the values of the pixels of Photo 1, and “Y” of the Photo 2. The variable “X” with a hyphen on it is the mean value of the entire photo 1. The variable “Y” with a hyphen on it is the mean value of the entire photo 2.

The First Step is calculating the mean values of the Icons. This process consists in add all the values of the pixels of the photo and then divide between the total numbers of pixels. We get a number.

The Second Step is calculating the Numerator and Denominator of the fraction.

This process does the same with the 3 photos of the Data Base. We do three correlation pro person of the Data Base. At the end we have 3 correlation coefficients. We compare these values with the limit that we have fixed. When one of these values is bigger than the limit, we have a positive match. In other case we have a negative match.

The result of the entire program (icon + correlation) has the following representation:

(FOTO DEL RESULTADO DE LA EJECUCIÓN DEL PROGRAMA, APARECE EN EL RESUMEN EN ESPAÑOL)

The person unknown was Jose Juan and the program recognised him. When the program hat a person, whose “r” coefficient is bigger than the limit, that we set, then the match is positive and the photo of this person of the Data Base is

shown. When the match is negative (lower than “r”) then a red frame is shown. Totally always appear a number of frames that it corresponds with the number of persons of the Data Base.

We set r = 85%. With this value there are no mistakes, and the person unknown is correctly identified. We started wit a value of 75%. We thought that this value would be enough. But in the tests appeared mistakes of identification. That means that we got more than 1 picture at the end. In the last test we had to reduce the coefficient because due to the light conditions and the area of the face the matching between the images was not too good.

With these tests we probed that the Data Base hat to be as big as it can. And the photos of this Data Base would be very different. With these two ideas the matching will be bigger.

7. Tests

The method was tested with the photographs of the Database. To make these tests we introduce one of the three photos like a new one. Then I see the results. Of course that one of the photos had 100 % of matching but we put more attention in the other percentages.

Before to introduce the results I explain how I present it:

-Name of the Person

1st Photo: {Persons that they are not the person that I put like "new person", that is to say, when the program fails}.

2nd Photo: {Persons that they are not the person that I put like "new person", that is to say, when the program fails}.

3rd Photo: {Persons that they are not the person that I put like "new person", that is to say, when the program fails}.

A "0" means that the recognition was successful.

a) Test with r = 75 % (r = coefficient of correlation)

-**DAVID**: {JOSE1, JOSE3, VICTOR2, VICTOR3}, {JOSE3, VICTO2, VICTOR3}, {JOSE1, JOSE3, VICTOR2, VICTOR3}.

-**JASCHUL**: {0}, {0}, {0}.

-**JOSE**: {DAVID1, DAVID3, JUAN1, JUAN3}, {0}, {DAVID1, DAVID2, DAVID3}.

-**JUAN**: {JOSE1}, {0}, {JOSE1}.

-**PAUL**: {0}, {0}, {0}.

-**SCHNEIDER**: {0}, {0}, {0}.

-**THOMAS**: {0}, {0}, {0}.

-**VICTOR**: {0}, {DAVID1, DAVID2, DAVID3}, {DAVID1, DAVID2, DAVID3}.

-**VOLLMANN**: {0}, {0}, {0}.

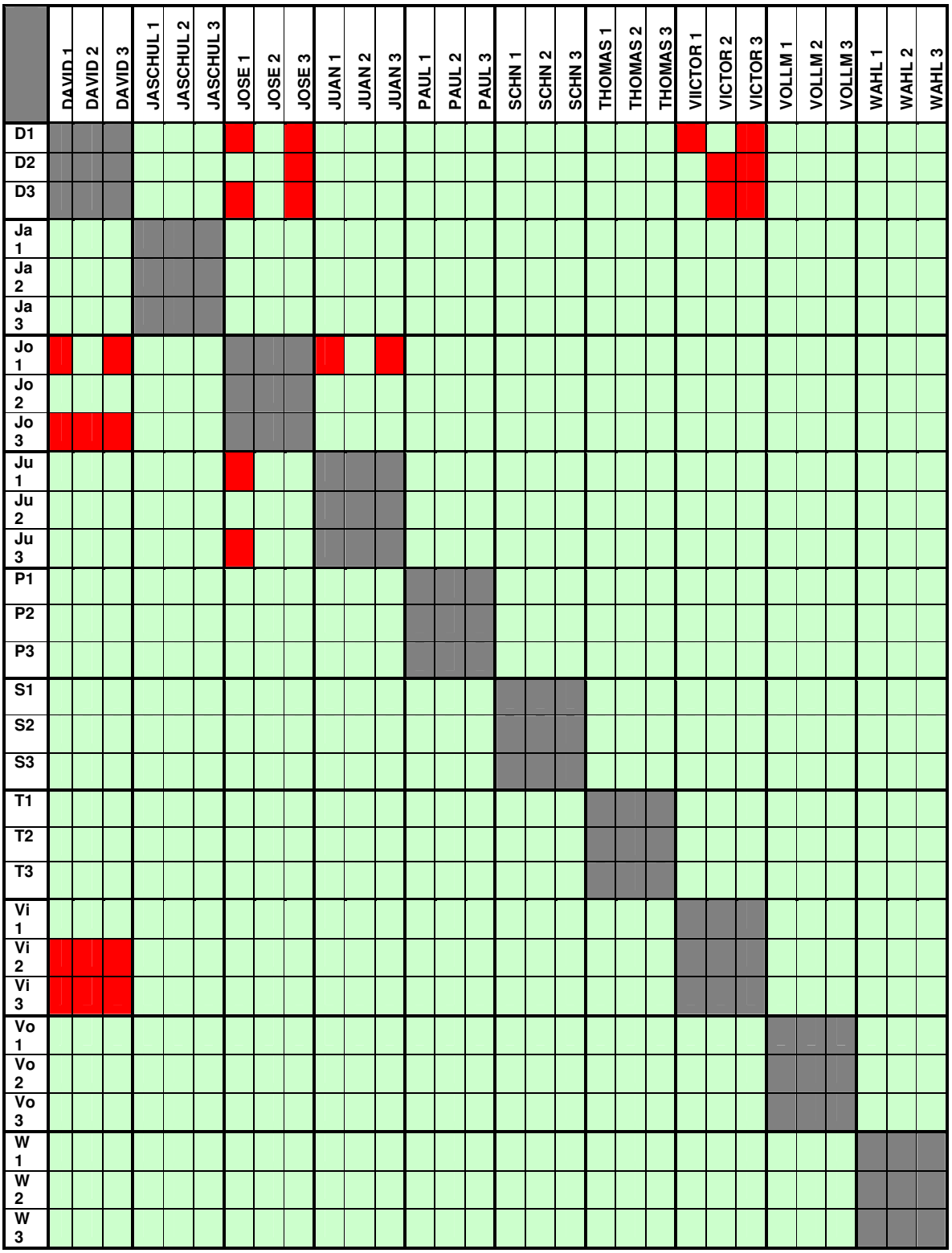
-**WAHL**: {0}, {0}, {0}.

Looking at these results, we increased the "r" coefficient. It was funny that we only had problems with the Spanish people.

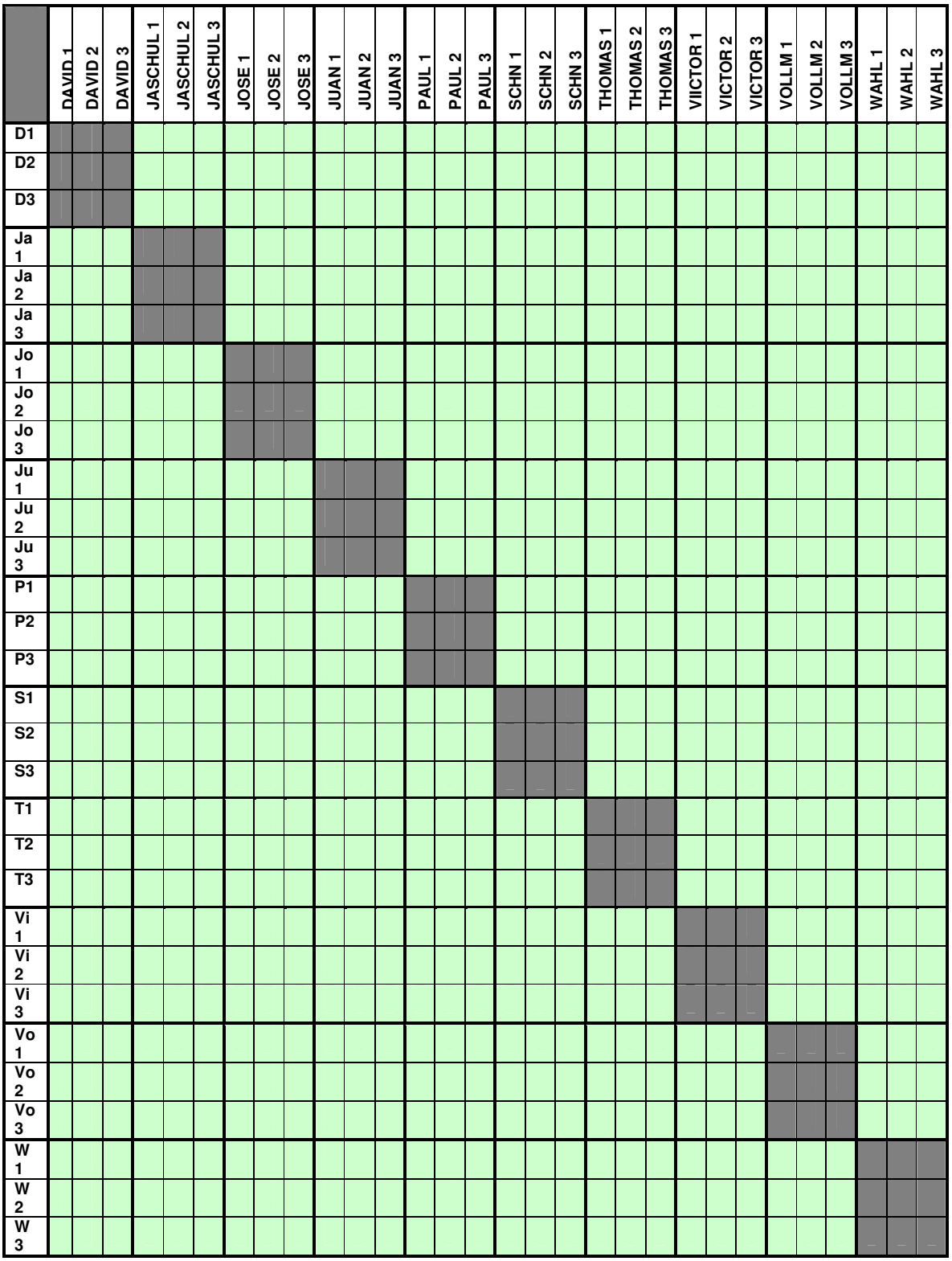
b) Test with r = 80 % (r = coefficient of correlation)

-**DAVID**: {0}, {0}, {0}.
-**JASCHUL**: {0}, {0}, {0}.
-**JOSE**: {0}, {0}, {0}.
-**JUAN**: {0}, {0}, {0}.
-**PAUL**: {0}, {0}, {0}.
-**SCHNEIDER**: {0}, {0}, {0}.
-**THOMAS**: {0}, {0}, {0}.
-**VICTOR**: {0}, {0}, {0}.
-**VOLLMANN**: {0}, {0}, {0}.
-**WAHL**: {0}, {0}, {0}.

RESULTS OF TEST A (r = 75%):



RESULTS OF TEST B (r = 80%):



With this value of “r”, the program had no problem to recognize all the people. The Test was successful.

The Last Test was to test new photos a probe the program. The problem was that we could not do the photos similar like the photos from the data base. The faces were a little bigger or smaller. Only we could test with Jose Juan and me.

With my photo the Test was successful, but I had to reduce the limit of a positive matching to 60%. With Jose Juan appeared five more persons. The Test failed.

8. Conclusions

This method, alone, does not obtain good percentage of successful. With the data base worked well, but these were taken at the same time, with the same light conditions. With news photos the percentage falls. But this method in combination with other methods can get better results.

Other problem was that you need a lot of power of calculation, with 10 people each test take a bit of time (1 min). With more people more time.

In short the project was very interesting and the software a good surprise. Khoros Cantata is a very good suite to work with images. It allows you a lot of freedom and many useful tools.

C Code

of the

Functions:

Icon
&
Correlation

ICON.C

With this function, we pretend to extract a frame of the image. This new image, that we call Icon, will use to make the Correlation. It is easy to see with an example:

1) Define Variables

```
int run_Icon(void)
{
```

These two variables aren't necessary, but sure are handy. Several error/warning routines (kerror, kwarn, etc) expect two parameters which are the library and function where the error occurred. Instead of passing them explicitly in each call to kerror or kwarn, it is better to define them for the whole program/routine so it will be easier to change them in case the name of the program/routine changes. For this case (a main routine) we will the "lib" will be the name of the program and the "rtn" will be "main".

```
  char *lib = "Icon";
 char *rtn = "main";
```

This is a kobject - a polymorphical data object that allows representation of data in five dimensions with additional segments in different data types. Also a kobject can be read/save in different formats. Every time you want to make an input, output or processing of a data object, use a kobject. When first declaring it, assign NULL to it, will make error checking easier.

```
 kobject in_obj = NULL;
 kobject out_obj = NULL;
```

The width and height of the input image, which will determine the size of the output kobject

```
  int h,w,d,t,e;
 int ch,cw;
```

In “inplane”, we will store the image. This way we can manipulate it.

```
double *inplane = NULL;
double value;
```

A list of objects

```
klist *objlist = NULL;
```

Several variables that we will use later

```
int datatype;
int flag = 0;
int x_ext,y_ext;
int temp = 0;
int width= 0;
int min1,min2 = 256;
int offset = 0;
```

2) Create input object and check it

The kobjects must be created with special commands: for input, usually with *kpds_open_input_object*, for output usually with *kpds_open_output_object*, and you can create temporary kobjects with *kpds_create_object*. When a kobject is open, all its data and attributes can be set or read. When a kobject is created, you must define a minimum set of attributes to work with it. In the five lines above, we attempt to open a kobject, and in case of failure we use kerror to display an error message and kexit to finish the program (wouldn't make any sense to continue if we cannot create the output object). The call to kexit with *KEXIT_FAILURE* means an error, while calling it with *KEXIT_SUCCESS* means that the program terminated successfully. Note the usage of the clui_info structure to get the user interface parameter (clui_info->o_file).

```
if ((in_obj = kpds_open_input_object(clui_info->i_file)) == KOBJECT_INVALID)
{
        kerror(lib,rtn, "Cannot open input object %s",clui_info->i_file);
        kexit(KEXIT_FAILURE);
}
```

3) Create output object and check it

```
if ((out_obj = kpds_open_output_object(clui_info->o_file)) == KOBJECT_INVALID)
{

        kerror(lib,rtn, "Cannot open outout object %s",clui_info->i_file);
        kpds_close_object(in_obj);
        kexit(KEXIT_FAILURE);
}
```

4) Get & set some input parameters

Here we first get the original data type of the input object in the variable datatype and then set the data type to double. This is done to simplify the processing; otherwise we would need to duplicate the code to deal with the several supported data types.

```
kpds_get_attribute(in_obj,KPDS_VALUE_DATA_TYPE,&datatype);
kpds_set_attribute(in_obj,KPDS_VALUE_DATA_TYPE,KDOUBLE);
```

This function gets the five parameters of the input object

```
kpds_get_attribute(in_obj,KPDS_VALUE_SIZE,&w,&h,&d,&t,&e);
```

We copy the input object in the output object. This way, the output object hat the same attributes. And we check it after that.

```
if (!kpds_copy_object(in_obj,out_obj))
{
        kerror(lib,rtn,"Unable to copy input to output object");
        kexit(KEXIT_FAILURE);
}
```

5) Alloc memory for the planes

We use the k functions to guarantee portability. And we check it after that.

```
inplane=(double *)kmalloc(w*h*sizeof(double));

if (inplane == NULL)
{
        kerror(lib,rtn,"Unable to alloc memory for plane");
        kexit(KEXIT_FAILURE);
}
```

The List of objects is made to make easy free them at the end of the program.

```
objlist = klist_add(objlist,inplane,"KMALLOC");
```

6) Get a plane of the input object

With this function we obtain a plane (the whole image) with only one function. And the pointer is in the beginning of the image.

```
kpds_get_data(in_obj,KPDS_VALUE_PLANE,(kaddr)inplane);
```

7) Search of the Coordinate Y of the Point of Extraction

This part of the code tries to find the Y coordinate of the punt of extraction. The frame that we extract has a set size. We only need where begin this frame to

extract it. The process of the extraction will be made for a function of the program. We find the first pixel whose value is lower than 175. When this happens, we stop the search.

```
for(ch=0;ch<h;ch++)
{
        for(cw=0;cw<w;cw++)
        {
                value = inplane[PIXEL(cw,ch)];
                if ((value < 175)&&(flag == 0))
                {
                        flag = 1;
                        y_ext = ch-10;
                }

        }
}
```

8) Search of the Coordinate X of the Extraction-Point

Step 1: Search of Minimum Value in the Left Side of the Face

The search of the X coordinate is a bit complicate. The first step is to find a point in the left side of the face.

```
for (ch=0;ch<h;ch++)
{
        for(cw=0;cw<w;cw++)
    {
                if((ch>100)&&(ch<412))
                {
                        value = inplane[PIXEL(cw,ch)];
                        if(value<175)
                        {
                                temp = cw;
                                if(temp<min1)
                                        min1 = temp;
                        }
                }
        }
}
```

Step 2: Search of the Minimum Value of the Right Side of the Face

After that we search a point in the other side of the face. And we save it.

```
flag = 0;
for (ch=0;ch<h;ch++)
{
        for(cw=0;cw<w;cw++)
        {
                if((ch>(y_ext)+15)&&(ch<412)&&(cw>256))
                {
                        value = inplane[PIXEL(cw,ch)];
                        if((value>175)&&(flag==0))
                        {
                                flag = 1;
                                temp = cw;
                                if(temp>min2)
                                        min2 = temp;
                        }
                }
        }
        flag = 0;
}
```

Step 3: The Calculation of the Offset

Now we calculate the offset (Distance between the face and the border of the frame). We make that to keep the same distance between the frame and the face, in both sides of the Icon. All this process is to try to keep symmetry in the Icon.

```
width = min2-min1;
offset = (310-width)/2;
x_ext = min1-offset;
```

9) Extraction of the frame

To make the extraction of the frame, we use a function of the library. To use this function we only need the coordinates of the point of beginning. This point is already calculated.

```
lkextract(in_obj,x_ext,y_ext,0,0,0,310,410,t,d,e,FALSE,out_obj);
```

10) Restore the original datatype from out_obj

```
kpds_set_attribute(in_obj,KPDS_VALUE_DATA_TYPE,datatype);
```

11) Free memory

We free all the memory use in the function. We have to do only once, because we use the function objlist.

```
 (void)klist_free(objlist,(kfunc_void)lkcall_free);
```

12) Close object

```
  kpds_close_object(in_obj);
  kpds_close_object(out_obj);

   return TRUE;
}
```

CORRELATION.C

This function does the Correlation between de Icon of an unknown person and the three icons of a person of the data base. It does them and after that compares the results. When the comparison is positive, the image of the person is shown; when the comparison is negative a red frame is shown. The formula used to do the correlation is:

$$r = \frac{\frac{1}{n}\sum_{i=1}^{n}\left[\left(X_i - \overline{X}\right)\left(Y_j - \overline{Y}\right)\right]}{\sqrt{\frac{1}{n}\sum_{1=1}^{n}\left(X_i - \overline{X}\right)^2}\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \overline{Y}\right)^2}}$$

```
int run_Correlation(void)
{
```

1) Define Variables

```
 char *lib = "Icon";
char *rtn = "main";
kobject icon_new = NULL;
kobject icon_1 = NULL;
kobject icon_2 = NULL;
kobject icon_3 = NULL;
kobject out_obj = NULL;
int h,w,d,t,e;
int ch,cw;
int datatype;
int datatype1;
int datatype2;
int datatype3;
double *icon_plane = NULL;
double *photo1_plane = NULL;
double *photo2_plane = NULL;
double *photo3_plane = NULL;
klist *objlist = NULL;
double aver_i = 0;
double aver_ph1 = 0;
double aver_ph2 = 0;
double aver_ph3 = 0;
double value1,value2;
double var1,var2;
double num=0;
double den=0;
double r1,r2,r3;
double den1=0;
double den2=0;
int h1=100;
int w1=100;
unsigned char map[3]; /* a map entry = row */
unsigned char pix=0; /* a value for the value segment */
int r;
```

1) Create input objects and check them

This first corresponds with the icon of the person unknown

```
if ((icon_new = kpds_open_input_object(clui_info->i_file)) == KOBJECT_INVALID)
{
        kerror(lib,rtn, "Cannot open input object %s",clui_info->i_file);
        kexit(KEXIT_FAILURE);
}
```

Now the three icons of a person of the data base

```
if ((icon_1 = kpds_open_input_object(clui_info->i0_file)) == KOBJECT_INVALID)
{
        kerror(lib,rtn, "Cannot open input object %s",clui_info->i0_file);
        kpds_close_object(icon_new);
        kexit(KEXIT_FAILURE);
}

if ((icon_2 = kpds_open_input_object(clui_info->i01_file)) == KOBJECT_INVALID)
{
        kerror(lib,rtn, "Cannot open input object %s",clui_info->i01_file);
        kpds_close_object(icon_new);
        kpds_close_object(icon_1);
        kexit(KEXIT_FAILURE);
}

if ((icon_3 = kpds_open_input_object(clui_info->i012_file)) == KOBJECT_INVALID)
{
        kerror(lib,rtn, "Cannot open input object %s",clui_info->i012_file);
        kpds_close_object(icon_new);
        kpds_close_object(icon_1);
        kpds_close_object(icon_2);
        kexit(KEXIT_FAILURE);
}
```

The variable “r” contains the value of the correlation coefficient. This value will be the limit between a positive or a negative result

```
r=clui_info->int1_int;
```

3) Create output object and check it

```
if ((out_obj = kpds_open_output_object(clui_info->o_file)) == KOBJECT_INVALID)
{
        kerror(lib,rtn, "Cannot open outout object %s",clui_info->o_file);
        kpds_close_object(icon_new);
        kpds_close_object(icon_1);
        kpds_close_object(icon_2);
        kpds_close_object(icon_3);
        kexit(KEXIT_FAILURE);
}
```

4) Get & set some input parameters

We obtain some information of the Icon of the unknown person and we save it in variables

```
kpds_get_attribute(icon_new,KPDS_VALUE_DATA_TYPE,&datatype);
kpds_set_attribute(icon_new,KPDS_VALUE_DATA_TYPE,KDOUBLE);

kpds_get_attribute(icon_new,KPDS_VALUE_SIZE,&w,&h,&d,&t,&e);

kpds_get_attribute(icon_1,KPDS_VALUE_DATA_TYPE,&datatype1);
kpds_set_attribute(icon_1,KPDS_VALUE_DATA_TYPE,KDOUBLE);


kpds_get_attribute(icon_2,KPDS_VALUE_DATA_TYPE,&datatype2);
kpds_set_attribute(icon_2,KPDS_VALUE_DATA_TYPE,KDOUBLE);


kpds_get_attribute(icon_3,KPDS_VALUE_DATA_TYPE,&datatype3);
kpds_set_attribute(icon_3,KPDS_VALUE_DATA_TYPE,KDOUBLE);
```

Here we copy the structure of the input icon into the output object

```
 if (!kpds_copy_object(icon_new,out_obj))
{
        kerror(lib,rtn,"Unable to copy input to output object");
        kexit(KEXIT_FAILURE);
}
```

5) Alloc memory for the planes

Now, we alloc memory for the planes. We use one plane for each input object. We work with planes.

```
icon_plane=(double *)kmalloc(w*h*sizeof(double));

if (icon_plane == NULL)
{
        kerror(lib,rtn,"Unable to alloc memory for plane");
        kexit(KEXIT_FAILURE);
}

objlist = klist_add(objlist,icon_plane,"KMALLOC");

photo1_plane=(double *)kmalloc(w*h*sizeof(double));

if (photo1_plane == NULL)
{
        kerror(lib,rtn,"Unable to alloc memory for plane");
        kexit(KEXIT_FAILURE);
}

objlist = klist_add(objlist,photo1_plane,"KMALLOC");

photo2_plane=(double *)kmalloc(w*h*sizeof(double));

if (photo2_plane == NULL)
{
```

```
        kerror(lib,rtn,"Unable to alloc memory for plane");
        kexit(KEXIT_FAILURE);
}

objlist = klist_add(objlist,photo2_plane,"KMALLOC");

photo3_plane=(double *)kmalloc(w*h*sizeof(double));

if (photo3_plane == NULL)
{
        kerror(lib,rtn,"Unable to alloc memory for plane");
        kexit(KEXIT_FAILURE);
}

objlist = klist_add(objlist,photo3_plane,"KMALLOC");
```

6) Get a plane of the input objects

We obtain the plane of the input objects; all the work with the pointer is very easy with this program

```
kpds_get_data(icon_new,KPDS_VALUE_PLANE,(kaddr)icon_plane);

kpds_get_data(icon_1,KPDS_VALUE_PLANE,(kaddr)photo1_plane);

kpds_get_data(icon_2,KPDS_VALUE_PLANE,(kaddr)photo2_plane);

kpds_get_data(icon_3,KPDS_VALUE_PLANE,(kaddr)photo3_plane);
```

7) Calculate the average of the input objects

We calculate the average of the four Icons. The process consists in to sum the values of all the pixels and after that divide between the total numbers of pixels.

```
for(ch=0;ch<h;ch++)
{
        for(cw=0;cw<w;cw++)
        {
                aver_i+=icon_plane[PIXEL(cw,ch)];
        }
}
aver_i=aver_i/(w*h);
kprintf("media icon = %g\n",aver_i);


for(ch=0;ch<h;ch++)
{
        for(cw=0;cw<w;cw++)
        {
                aver_ph1+=photo1_plane[PIXEL(cw,ch)];
        }
}
aver_ph1=aver_ph1/(w*h);
kprintf("media foto1 = %g\n",aver_ph1);
```

```
for(ch=0;ch<h;ch++)
 {
        for(cw=0;cw<w;cw++)
        {
                aver_ph2+=photo2_plane[PIXEL(cw,ch)];
        }
 }
 aver_ph2=aver_ph2/(w*h);
 kprintf("media foto2 = %g\n",aver_ph2);


 for(ch=0;ch<h;ch++)
 {
        for(cw=0;cw<w;cw++)
        {
                aver_ph3+=photo3_plane[PIXEL(cw,ch)];
        }
 }
 aver_ph3=aver_ph3/(w*h);
 kprintf("media foto3 = %g\n",aver_ph3);
```

8) Calculate the correlation-coefficient

First we calculate the numerator and then the denominator

```
 for(ch=0;ch<h;ch++)
 {
        for(cw=0;cw<w;cw++)
        {
                value1=icon_plane[PIXEL(cw,ch)];
                value2=photo1_plane[PIXEL(cw,ch)];
                var1=value1-aver_i;
                var2=value2-aver_ph1;
                num+=(var1*var2);
        }
 }

 num=num/(w*h);

 for(ch=0;ch<h;ch++)
 {
        for(cw=0;cw<w;cw++)
        {
                value1=icon_plane[PIXEL(cw,ch)];
                value2=photo1_plane[PIXEL(cw,ch)];
                var1=value1-aver_i;
                var2=value2-aver_ph1;
                den1+=(var1*var1);
                den2+=(var2*var2);
        }
 }
 den=sqrt(den1*den2);
 den=den/(w*h);

 r1=(num/den)*100;
 kprintf("coef r1 (%)= %g\n",r1);

 num=0;
```

```
den1=0;
den2=0;
for(ch=0;ch<h;ch++)
{
        for(cw=0;cw<w;cw++)
        {
                value1=icon_plane[PIXEL(cw,ch)];
                value2=photo2_plane[PIXEL(cw,ch)];
                var1=value1-aver_i;
                var2=value2-aver_ph2;
                num+=(var1*var2);
        }
}
num=num/(w*h);

for(ch=0;ch<h;ch++)
{
        for(cw=0;cw<w;cw++)
        {
                value1=icon_plane[PIXEL(cw,ch)];
                value2=photo2_plane[PIXEL(cw,ch)];
                var1=value1-aver_i;
                var2=value2-aver_ph2;
                den1+=(var1*var1);
                den2+=(var2*var2);
        }
}
den=sqrt(den1*den2);
den=den/(w*h);

r2=(num/den)*100;
kprintf("coef r1 (%)= %g\n",r2);


num=0;
den1=0;
den2=0;
for(ch=0;ch<h;ch++)
{
        for(cw=0;cw<w;cw++)
        {
                value1=icon_plane[PIXEL(cw,ch)];
                value2=photo3_plane[PIXEL(cw,ch)];
                var1=value1-aver_i;
                var2=value2-aver_ph3;
                num+=(var1*var2);
        }
}
num=num/(w*h);
```

```
 for(ch=0;ch<h;ch++)
 {
        for(cw=0;cw<w;cw++)
        {
                value1=icon_plane[PIXEL(cw,ch)];
                value2=photo3_plane[PIXEL(cw,ch)];
                var1=value1-aver_i;
                var2=value2-aver_ph3;
                den1+=(var1*var1);
                den2+=(var2*var2);
        }
 }
 den=sqrt(den1*den2);
 den=den/(w*h);

 r3=(num/den)*100;
 kprintf("coef r3 (%)= %g\n",r3);
```

9) Moment of the Comparison

When the comparison is negative we create a red frame

```
  if((r1<r)&&(r2<r)&&(r3<r))
  {

        kpds_create_value(out_obj);

        kpds_set_attributes(out_obj,KPDS_VALUE_SIZE,w1,h1,1,1,1,KPDS_VALUE_DATA_T
YPE,KUBYTE,NULL);

        kpds_create_map(out_obj);

        kpds_set_attributes(out_obj,KPDS_MAP_SIZE,3,6,1,1,1,KPDS_MAP_DATA_TYPE,KU
BYTE,NULL);

        map[0]=255;
        map[1]=0;
        map[2]=0;

        for(ch=0;ch<h1;ch++)
        {
                for(cw=0;cw<w1;cw++)
           {
                        kpds_put_data(out_obj,KPDS_VALUE_POINT,(kaddr)&pix);
                        kpds_put_data(out_obj,KPDS_MAP_LINE,(kaddr)map);
                }
        }
        kpds_close_object(out_obj);
  }
```

When the comparison is positive, we copy the Photo of the Data Base which obtained a positive result

```
if(r1>r)
{
        if(!kpds_copy_object(icon_1,out_obj))
        {
                kerror(lib,rtn,"Unable to copy icon1 in out_obj");
                kexit(KEXIT_FAILURE);
        }
}
else if(r2>r)
{
        if(!kpds_copy_object(icon_2,out_obj))
        {
                kerror(lib,rtn,"Unable to copy icon2 in out_obj");
                kexit(KEXIT_FAILURE);
        }
}
else if(r3>r)
{
        if(!kpds_copy_object(icon_3,out_obj))
        {
                kerror(lib,rtn,"Unable to copy icon3 in out_obj");
                kexit(KEXIT_FAILURE);

        }
}
```

10) Free Memory

We free the memory used

```
(void)klist_free(objlist,(kfunc_void)lkcall_free);
```

11) Close the objects

```
kpds_close_object(icon_new);
kpds_close_object(icon_1);
kpds_close_object(icon_2);
kpds_close_object(icon_3);
```

return TRUE;

}

Manual

Of

Khoros

Cantata

INDEX

2. Visual Programming: Cantata.

2.1. Introduction.

2.2. Overview of Graphical User Interface.

2.3. The Visual Programming Workspace.

2.3.1. Introduction tit h Glyph.

2.3.2. Standard Glyph Components.

2.3.2.1. Input Data Connection Node.

2.3.2.2. Output Data Connection Node.

2.3.2.3. Data Available (DAV) Input

2.3.2.4. Data Available (DAV) Output

2.3.2.5. Pane Access Control.

2.3.2.6. Run Button.

2.3.2.7. Input Control Connection Node.

2.3.2.8. Output Control Connection Node.

2.3.2.9. Operator Name.

2.3.2.10. “Selected” Indicator.

2.3.2.11. Open Workspaces.

2.3.2.12. Control Structure Pixmap.

2.3.2.13. Error Indicator.

2.3.2.14. Info Indicator.

2.3.3. Basic Glyph Operations.

2.3.3.1. Selecting a Glyph.

2.3.3.2. Moving a Glyph.

2.3.3.3. Destroying a Glyph.

2.3.3.4. Executing a Glyph.

This manual is a resume of the manuals which appear in:

http://www.cab.u-szeged.hu/local/doc/khoros/Tutorial/index.html

http://rab.ict.pwr.wroc.pl/khoros_root/topmost_toc.html

1. What is Khoros?

Khoros is a software integration and development environment that emphasizes information processing and data exploration. The goal of the Khoros software is to provide a complete application development environment that redefines the software engineering process to include all members of the project group, from the application end-user to the infrastructure programmer. Khoros is a comprehensive system that may be viewed in different ways, depending on your scientific needs and objectives.

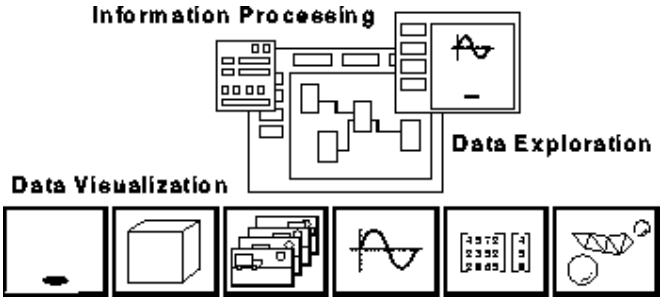


Figure 1: Khoros provides a large variety of programs for information processing, data exploration, and data visualization.

For those who need end user solutions to scientific problems, Khoros may be used as it stands, providing a rich set of programs for information processing, data exploration, and data visualization. Multidimensional data manipulation operators include point wise arithmetic, statistic calculations, data conversions, histograms, data organization, and size operators; image processing routines and matrix manipulation are also provided. Interactive data visualization programs include an image display and manipulation package, an animation program, a 2D/3D plotting package, a colormap alteration tool, and an interactive image/signal classification application. In addition, 3D visualization capabilities are also offered; a number of data processing routines for 3D visualization are provided, along with a software rendering application. The Khoros operators are generalized, such that each can solve problems in a variety of specific areas such as medical imaging, remote sensing, process control, signal processing, and numerical analysis.

All information processing and visualization programs in Khoros are available via the *visual programming environment*, **Cantata**. **Cantata** is a graphically expressed, event-driven, data flow visual language which provides a visual programming environment within the Khoros system. Data flow is a "naturally visible" approach in which a visual program is described as a directed graph, where each node represents an operator or function and each directed arc represents a path over which data flows. By providing a natural environment which is similar to the block diagrams that are already familiar to practitioners in the field, the visual language provides support to both novice and experienced programmers. **Cantata** supports coarse grain distributed processes; it can handle both stream and block data. Its visual hierarchy, iteration, flow control, and expression-based parameters make it a powerful simulation and prototyping system.

For application developers, the Khoros Toolbox Programmer's system consists of programming services and software development tools that support all aspects of developing new engineering and scientific applications. Applications written to Khoros can take advantage of the same capabilities offered by the Khoros data processing and visualization routines, including the ability to transparently access large data sets distributed across a network, operate on a variety of data and file formats without conversion, simultaneously support different widget sets, and maintain a consistent presentation with a standardized user interface. The software development environment provides developers with a direct manipulation graphical user interface design tool, automatic code generation, standardized user interface and documentation, and interactive configuration management. The Khoros software development system can also be used for software integration, where existing programs can be brought together into a consistent, standardized, and cohesive environment.

Khoros provides a powerful working environment for the engineering and scientific communities, addressing many of the issues associated with quickly developing X Window based applications, prototyping solutions to complex problems, and utilizing the resources of a distributed network. The layered approach of the Khoros infrastructure and the concept of program services

provide developers with the flexibility to create complex applications, while at the same time hiding the intimidating details of operating systems and X Window systems.

A common misperception is that there is a single application named "Khoros." In fact, "Khoros" is the name referring to hundreds of programs and thousands of library calls, available in several discrete sets which are referred to as "toolboxes." Khoros is a complete data exploration and software development environment that reduces time in solving complex problems, allows free sharing of ideas and information, and promotes portability.

1.1. Organization of the Khoros Software

This section introduces concepts used by all of the application toolboxes and describes the data model used in Khoros and how the application toolbox operators behave with respect to the model. It also contains information on useful Khoros utilities, instructions for invoking the operators from the command line, and instructions on how to access and use the operators within the visual programming language**.**

Each toolbox section contains an introduction consisting of more specific information which applies to that toolbox and tables listing the available operators. In some toolboxes, such as Envision and Geometry, full chapters are devoted to the interactive applications in the toolbox.

Operator tables are provided in each toolbox in two formats. The first shows the hierarchical organization of the operators by category and subcategory. The second lists operators alphabetically. The Khoros software system is divided into several *toolboxes*. A toolbox is a collection of programs and/or libraries that are managed as a single entity, or *object*. A toolbox imposes a predefined directory structure on its contents, to provide consistency and predictability in software and documentation organization.

Typically, a toolbox contains programs and libraries which have a similar function or common objective. The Application Toolboxes contains six application-specific toolbox chapters for the Khoros data processing and visualization software. These toolboxes are Datamanip, Envision, Geometry, Image, Matrix, and Retro.

Khoros Application Toolboxes

- DATAMANIP -- Polymorphic Data Processing Operators
- ENVISION -- Interactive Data Exploration Tools
- GEOMETRY -- 3D Scalar and Vector Visualization
- IMAGE -- Image Processing & Analysis Operators
- MATRIX -- Matrix Operators
- RETRO -- Khoros 1.0 Image Processing & Analysis Operators
- SAMPLE DATA -- Sample Workspaces and Data

Note that the only *required* toolboxes are *Bootstrap*, *Dataserv*, and *Design*; however, the *Datamanip* toolbox is essential for doing any kind of information processing, while the *Envision* toolboxes is necessary for data visualization..

1.1.1. The Design Toolbox

The Design toolbox contains the applications that make up the Khoros toolbox programmer's system. It also contains the libraries that comprise GUI and Visualization Services. In contrast to the Bootstrap and Dataserv toolboxes, the Design toolbox requires that X11 (revision 5, or 6) be installed. All applications and libraries in the Design toolbox depend on X Window.

The major applications in the Design toolbox include: **Cantata**, the visual programming language; **Craftsman**, the toolbox management program; **Composer**, the software object editor; **Guise**, the direct manipulation user interface design tool; and **khelp**, which displays online help pages.

The GUI and Visualization Services libraries, all dependent on X11, are also included in the Design toolbox. These libraries handle creation and

management of graphical user interfaces, as well as creation and management of all GUI and visual objects.

The Design toolbox requires the Bootstrap and Dataserv toolboxes to be installed.

1.1.2. The Datamanip Toolbox

The Datamanip toolbox contains general data manipulation operators. Data manipulation operators include point wise arithmetic, statistics calculations, data conversion, histogram, data organization, and size operators.

All Datamanip programs are written to support the relationships defined by the polymorphic data model. This data model provides for general 5-dimensional data (width, height, depth, time, elements), with any combination of value, map, mask, location, and time data components. The Datamanip operators are implemented using polymorphic data services, which is a part of the *kappserv* library in the Dataserv toolbox.

1.2. Khoros Applications

There are hundreds of programs included in the various toolboxes that make up the Khoros scientific software environment. This section provides a few highlights of the Khoros system, by briefly summarizing some of the top level applications that are available.

1.2.1. Cantata

The Cantata visual language is one of the main applications offered in the Design toolbox. Cantata is a graphically expressed, data flow visual language which provides a visual programming environment within the Khoros system. Its visual hierarchy, iteration, flow control, and expression based parameters make it a powerful simulation and prototyping system. *Workspaces* can be captured into a customizable, simple front end graphical user interface that hides the complexity of the data flow diagram, and allows the encapsulated workspace to be delivered as a new, interactive application for a production environment.

1.2.2. Craftsman

The Craftsman toolbox management application, distributed in the Design toolbox, is used to create, delete, and copy toolboxes as well as libraries and programs.

1.2.3. Composer

The Composer software object editor, also distributed in the Design toolbox, works in conjunction with Craftsman to provide you with convenient access to all of the software object components and can invoke all of the operations needed to modify, compile, debug, and document software objects. Composer can be run directly from the command line, or can be accessed via Craftsman.

1.2.4. Guise

The Guise direct manipulation graphical user interface design tool (located in the Design toolbox) works in conjunction with Composer to allow you to interactively create and modify the GUI of your program. Guise outputs a User Interface Specification (UIS) file, which defines both the graphical user interface (GUI) and/or command line user interface (CLUI) of your program. Guise can be run independently, or can be accessed via Composer.

1.2.5. Kman

The Khoros version of man, distributed as part of the Bootstrap toolbox, this command allows you to access man pages for any Khoros program. The [-k] argument to kman allows you to search for programs based on a key word.

1.2.6 . Editimage

The Envision toolbox contains editimage, which is an interactive image display, examination, and manipulation tool. Its capabilities include zooming in on the image, printing of pixel values and map data values, direct manipulation of the image colormap, and region of interest operations. Editimage operates on images of any data type, including complex data. It can operate on very large data sets, in which case a viewport is used to display a portion of the image, and a pan icon is used to navigate about the image. In this case, the only portion of the image currently being displayed is read into memory. A variety of file formats are supported without the need for conversion.

1.3 The Software Development System

The concepts of *toolbox object* and *software object* are important for the use of the Khoros software development system.

A *toolbox object* is an encapsulation of programs and libraries that are managed as an entity. The toolbox imposes a predefined directory structure on its contents to provide consistency and predictability to software configuration.

If a toolbox object is an encapsulation of programs and libraries, the programs and libraries themselves can be also be considered objects. A *software object* consists of the files associated with a particular library or program. A software object is composed of source code, documentation, and a user interface specification (if applicable). There are different types of software objects: program objects, library objects, pane objects, and script objects. These objects encapsulate Khoros programs, libraries, wrappers for other programs, and differently on the graphical user interface.

1.3.1. The Graphical User Interface

1.3.1.1. GUI Representations of the CLUI (Panes of Programs)

While the CLUI of a program is sufficient to execute a program directly from the command line, a graphical user interface is required to execute the same program from the Cantata visual programming language.

Every argument of a program must have representation on both its CLUI and its GUI; thus, each program in the Khoros system has a graphical user interface which is the graphical counterpart of its command line user interface. The GUI counterpart to a program's CLUI is frequently referred to as the *pane* of the program, so called for the ".pane" postfix convention which is used in naming the User Interface Specification (UIS) files that define both the GUI and the CLUI of a program.

As mentioned previously, the graphical user interface for a program is also used when the program is accessed via the Cantata visual programming language.

1.3.1.2. Kroutines vs. Xvroutines

The majority of the programs in Khoros are data processing routines, called *kroutines*. While these programs *may* have a graphical user interface (their *pane*) displayed (with the use of the [-gui] option, or when they are accessed via cantata) they do not *require* use of a GUI. These programs may be run solely from the command line on a non-X Windows terminal at home, for example.

In contrast, Khoros X-Window-based interactive applications, or *xvroutines*, must *always* display a graphical user interface; they cannot be run without a workstation supporting the X Window system. Furthermore, the graphical user interface that they display when executed from the command line is *not* simply the graphical representation of their command line arguments, as when the [-gui] option is used; rather, the graphical user interface that will be displayed is almost always considerably more extensive and complicated than the graphical representation of their command line arguments (which may also be displayed, as they too have the [-gui] option).

1.3.1.3. Toolbox Design and Implementation

The objective of Khoros Pro application toolboxes is to facilitate domain-specific work while simultaneously enabling cross-domain collaboration. When designing the data processing and visualization operators for Khoros Pro, we addressed several issues that we feel are critical to increasing the productivity of the scientist in solving data processing and analysis problems. These issues include providing domain interoperability, which will promote the reuse of software solutions over diverse domains; providing format and system independence, including the capability to process very large data sets; providing the ability not only to visualize data using traditional methods, but to allow for data exploration; and instilling confidence in these tools by ensuring reliability

and stability. This approach provides flexibility & power to the end user, and promotes collaboration within and across application domains.

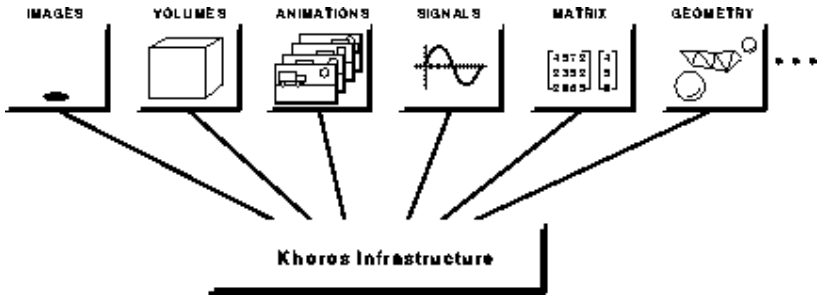


Figure 2: The data operators in the Khoros application toolboxes are designed to address the needs of many application domains, from image processing to signal processing; from geometry visualization to numerical analysis.

1.4. Overview of the Data Models

In order to better understand the functionality of the different Khoros operators, it is helpful to understand the data models to which the operators are written. The following two sections explain the *polymorphic data model* and the *geometry data model*.

1.4.1. The Polymorphic Data Model

The polymorphic data model is based on the premise that data sets are usually generated to model, or acquired from, real-world phenomena. The polymorphic model thus consists of data which exists in three-dimensional space and one-dimensional time. The model can be pictured most easily as a time-series of volumes in space. This time-series of volumes is represented by five different data segments. Each segment of data has a specific meaning dictating how it should be interpreted. Specifically, these five segments are value, location, time, mask, and map. All of these segments are optional; a data object may contain any combination of them and still conform to the polymorphic model.

The value segment is the primary data segment, consisting of data element vectors organized implicitly into a time-series of volumes. The value data may be given explicit positioning in space and time with the location and

time segments. The remaining two segments mask and map are provided for convenience. The mask segment is used to mark the validity of each point of value data. The map segment is provided as an extension to the value data; the value data can be used as an index into the map data. Figure 3 provides an overview of the polymorphic model. Each data segment is described in more detail below.

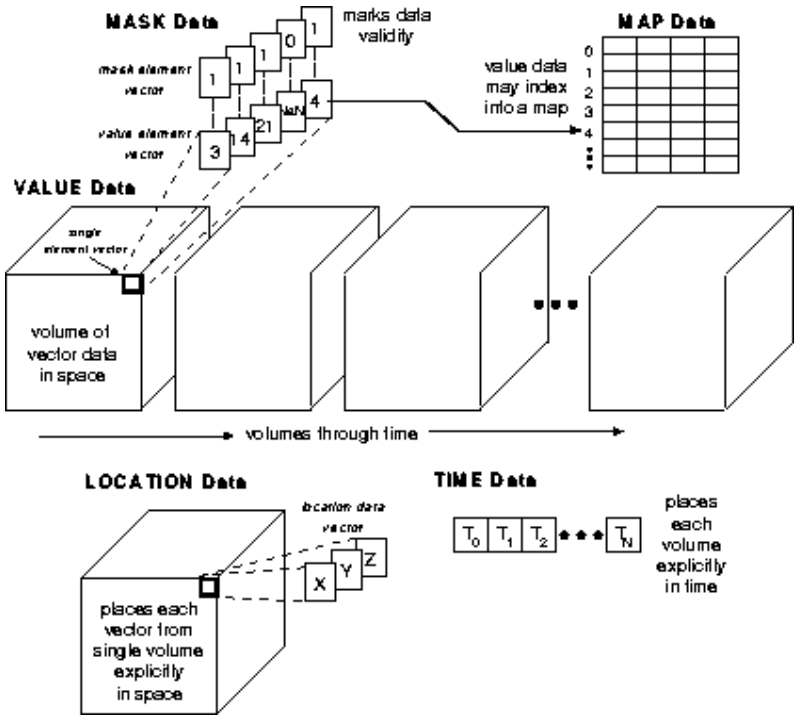


Figure 3: An overview of the Polymorphic Data Model. The polymorphic model consists of five data segments, with each segment serving a specific purpose. The value segment consists of data element vectors organized into a time-series of volumes. The volume of value data can be given explicit locations in space with the location segment; one location vector is provided for each value vector in a single volume. The volumes of value data can be given explicit locations in time with the time segment; a time-stamp may be given for each volume in time. A mask segment is available for marking value data validity. A map segment is also provided; the value data can be used as an index into the map data.

1.4.2. Value Data

The value data segment is the primary storage segment in the polymorphic data model. Most of the operators are specifically geared toward processing the data stored in this segment. For example, in an imaging context,

the individual pixel RGB values would be stored here. In a signal context, regularly sampled signal amplitudes would be stored here.

1.4.3. Location Data

The value points in the value segment are stored implicitly in a regularly gridded fashion. Explicit location information, such as longitude or latitude for map data, can be added using the location segment. If the value data is irregularly sampled in space, the explicit location of each sample can be stored here. Specifically, the information stored in this segment serves to position each of the value data in explicit space. Note that the location data only explicitly positions a single volume; the position then holds for each volume through time. A curvilinear grid allows for an independent position to be stored for each value vector in a volume. A rectilinear grid allows for explicit locations to be given for the *width*, *height*, and *depth* axes. A uniform grid allows for explicit location corner markers to be specified.

1.4.4. Time Data

Explicit time information can be added using the time segment. If each volume of value data is irregularly sampled in time, an explicit timestamp for each volume can be stored here. This is useful in animations where each frame of the animation occurs at a different time.

1.4.5. Mask Data

The mask segment is available for flagging invalid values in the value segment. If a processing routine produces invalid values, such as *NaN* or *Infinity*, these values can be flagged in the mask data so that later routines can avoid processing them. A mask point of *zero* is used to mark *invalid* value points, while a mask point of *one* is used to mark *valid* value points. The mask segment identically mirrors the value segment in size; that is, there is one mask point for each value point.

1.4.6. Map Data

In cases where the value data contains redundant vectors that are duplicated in different positions, the map segment may be used. The value vectors are replaced with values which are then indexed into the map. The map then contains the actual data vectors. Where representing the data explicitly in the value segment would mean that a redundant value could be represented many times, mapping allows the value to be represented once with indices to the relevant value vectors. In this sense, the map is an extension of the value segment.

1.4.7. Object Containing Value & Map

If the input object contains both map and value data, the program will operate exclusively on the map data whenever possible to maintain the compression provided by the map (see Figure 4). If there are multiple inputs, the data will most often be mapped before processing occurs. Also, operations that depend on data size, such as histogram or statistics calculations, must usually map the data before processing.

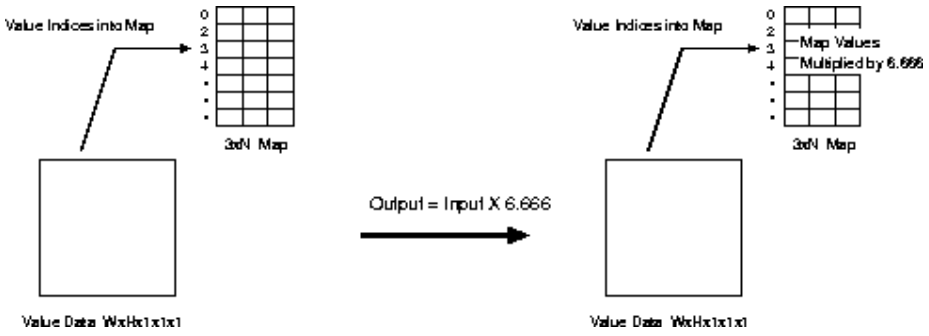


Figure 4: This figure illustrates the scaling operation performed on an object that has both map and value data. Since the value data acts as an index into the map data, the scaling operation should be performed on the map. Note that if the value data were scaled, indexing into the map would be corrupted. The output data object has the same dimensions as the input object.

2. Visual Programming: Cantata

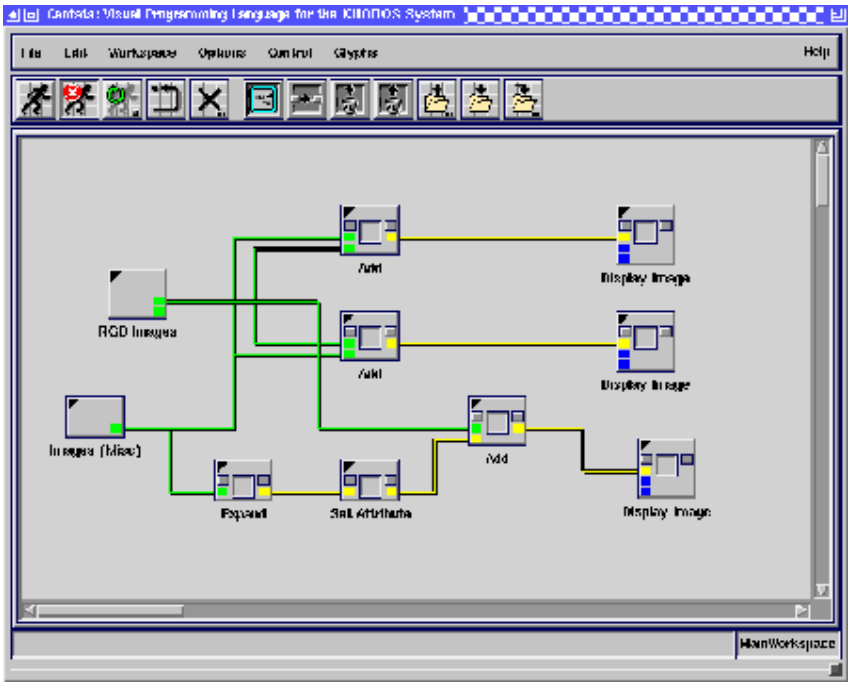


Figure 5: The cantata visual language allows you to build graphically expressed, data flow visual programs within the Khoros system.

2.1. Introduction

Cantata is a graphically expressed, data flow visual language which provides a visual programming environment within the Khoros system. Data flow is a "naturally visible" approach in which a visual program is described as a directed graph, where each node represents an operator or function and each directed arc represents a path over which data flows. The purpose in providing a visual language interface to the programs included in the Khoros system is to increase the productivity of researchers and application developers. By providing a more natural environment which is similar to the block diagrams that are already familiar to practitioners in the field, the visual language provides support to both novice and experienced programmers.

In Cantata, the icons (called *glyphs*) typically represent programs from the Khoros system. However, given the Khoros software integration

environment, they can also be used to represent non-Khoros programs that have been integrated into Khoros (see The Toolbox Programming Manual for information on creating a Khoros object and bringing the object into Cantata). Each of the hundreds of stand-alone data processing and scientific visualization programs in the Khoros system can be represented in the Cantata visual language as glyphs. To create a Cantata visual program, the user selects the desired programs (and control structures, as needed), places the corresponding glyphs on the Cantata workspace, and connects these glyphs to indicate the flow of data from program to program, forming a *network* within a *workspace*. Such workspaces can be executed, saved, and restored to be used again or modified later. Workspaces may also be encapsulated into stand-alone applications with a very simplified graphical user interface so that they may be treated as independent Khoros applications.

The visual hierarchy, iteration, flow control, and expression-based parameters make Cantata a powerful simulation and prototyping system. Cantata interprets the visual network dynamically to schedule glyphs and then dispatch them as processes. The Cantata scheduler is event driven rather than data driven or demand driven. Glyphs are referred to as *coarse grained* because each glyph corresponds to an entire process, not a code segment or a sub-procedure. Once a glyph has been scheduled, the dispatcher is responsible for determining the data transport, the communication protocol, and the process execution mode. Communication protocol between Cantata and the different glyphs can be as simple as just initiating process execution or more complicated if glyph parameters must be continuously updated as the process executes continuously.

Glyphs may be executed locally or remotely to efficiently utilize a heterogeneous network of computers. Cantata utilizes a network execution daemon to negotiate the remote data transport and to spawn processes on remote machines. The visual programmer assigns operators to specific machines interactively; this may be done both to optimize execution speed and to fully utilize available hardware. Note that the remote machines to be utilized need not have a full Khoros installation, but must at least have a copy of the

network execution daemon running in order to work with the remote transport mechanism.

Application specific domains, such as image processing and geometry visualization, typically process data as blocks. However, the domains of telecommunications and process control tend to process data as streams. Cantata glyphs can process data in both blocks and streams.

The Cantata visual language extends the basic data flow paradigm to make it a more powerful application prototyping or simulation environment. Data and control-dependent program flow is provided by *flow control glyphs* such as if/else, while, count, and trigger. Visual subroutines, or *procedures*, are available to support the development of hierarchical data flow graphs. Variables may be set interactively by the user, or calculated at run time via mathematical expressions tied to data values or control variables.

2.2. Overview of Graphical User Interface

The graphical user interface of the visual language consists of the Cantata *master form* surrounding a *canvas*, in which *networks* of connected *glyphs* may be created, as described in the following sections.

The Cantata *workspace* consists of a large canvas in which the visual programming network is constructed. By default, it is a gridded surface, although the screen images in this document show a plain canvas for clarity; glyphs will snap to the grid when created or moved, even if the grid is not visible. The actual Cantata workspace is several times larger than the size of the visible viewport. You can control which part of the workspace is visible in the viewport by moving the scroll bars on the left and bottom sides of the workspace frame. Along the top of the workspace is the workspace command bar, which contains *icons* on which you can click to perform certain operations. The workspace command bar provides easy access to the most frequently used Cantata features. It can be optionally hidden via the "Hide Command Bar" item on the Options menu, or by setting the command line option [-commandbar] to FALSE at startup.

The *master form*, which appears on the screen when Cantata is executed, consists of the following components:

1) The main Cantata menus, which provide access to the Khoros programs and utilities as well as Cantata utilities.

- "File" menu
- "Edit" menu
- "Workspace" menu
- "Options" menu
- "Control" menu
- "Glyphs" menu
- "Help" menu

2) The workspace command bar which contains icons representing the most commonly used commands from the Cantata inventory. The icons are displayed just below the main Cantata menu bar. The set of icons that appear is variable. You can choose to display just those icons you wish (including the entire set available), or you can make them all disappear.

3) The main workspace canvas. This is the area where the visual program is constructed. It takes up most of the surface area of the Cantata GUI. More than one workspace canvas can be open at a time. When new workspaces are created, the different canvases can be brought to the foreground by clicking on the Area tabs at the top-left of the canvas. By default, all actions will be performed on the top (i.e., current) workspace canvas.

2.3. The Visual Programming Workspace

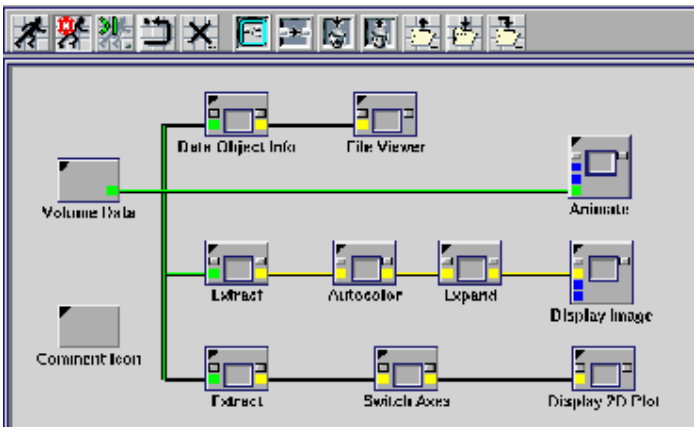


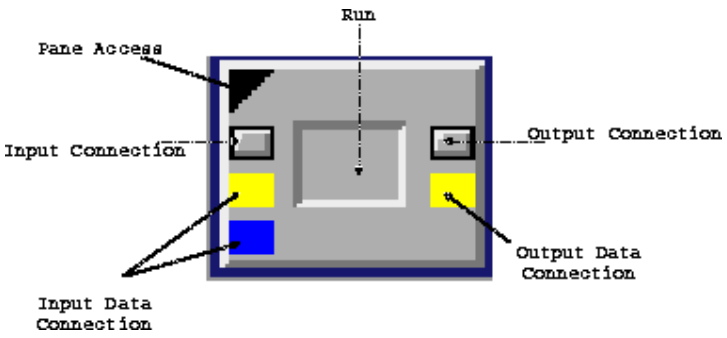
Figure 6: The visual programming workspace is made up of a viewport containing a canvas with a grid. A visual program, made up of a network of glyphs, can be placed on the canvas, either by restoring a saved program, or by creating a new one. To the right and bottom of the canvas are scrollbars (not pictured) which can be used to control the portion of the grid that is visible within the viewport. Along the top of the workspace is a workspace command bar, containing buttons displaying icons. These buttons provide shortcuts to a variety of visual programming operations that are also available from the main Cantata menus, shown at the top of the Cantata window.

There are a variety of functions supported by the Cantata workspace. Procedure creation and loop construct creation are accessible from the Control menu. Editing capabilities are accessed from the Edit menu. File manipulation features are accessed through the File menu. The Workspace menu provides control for starting, stopping, resetting, and checking visual programs. The most commonly used functions are also accessible from buttons/icons on the workspace command bar. With only a few exceptions (where the concept simply does not apply), the workspace functions always operate on the *currently selected* glyphs.

Functions in Cantata are available from several sources, such as the menus, the command bar, and keyboard accelerators. However, the most basic interface to Cantata functionality is through the menus; while only a subset of operations may be available through the command bar, for example, *all* functionality is available through the menus. It makes sense, then, to discuss menus first.

The following sections explain the general organization of the Cantata menus, with an explanation of the functionality provided by each item. Command bar counterparts to the menu items are provided where appropriate, as are keyboard accelerators. In those cases where a more detailed discussion is provided elsewhere, a reference to that discussion is given.

2.3.1. Introduction To The Glyph



A *glyph* is simply a visual representation of a program available from within Cantata. Typically, these are the programs in one of the Khoros 2 toolboxes you have installed at your site, but they can also be non-Khoros programs you have developed that have been given a Khoros pane interface (see The Toolbox Programming Manual for information on creating a Khoros object and bringing the object into Cantata). Each program may be run independently from the command line, or may be executed via Cantata. When accessed from Cantata, the program itself is referred to as an *operator*; the icon that represents the operator in the Cantata workspace is called a *glyph*. As stated earlier, a visual program simply consists of a number of glyphs connected together in a network.

2.3.2. Standard Glyph Components

A glyph has a number of components. Each component provides some sort of information about the glyph. In addition, many of the components are also *buttons* which you may use to perform an operation on the glyph. A summary of the various glyph components follows.

2.3.2.1. Input Data Connection Node

Each glyph may have one or more input data connections. The input data connection node is represented by a colored square at the left edge of the glyph. When a data input connection is *required*, the square will appear in yellow; when it is *optional*, it will appear in blue. Some operators are provided for the express purpose of providing input for other operators; their glyphs will have no input data connections, as they take no input.

2.3.2.2. Output Data Connection Node

Each glyph may have one or more output data connections. The output data connection node is represented by a colored square at the right edge of the glyph. When a data output connection is *required*, the square will appear in yellow; when it is *optional*, it will appear in blue. Some operators are provided specifically to visualize data produced by other operators; their glyphs will have no output data connections, as they produce no output.

2.3.2.3. Data Available (DAV) (associated with Input Data Connection)

When there is data available to the operator from a previous glyph connected to the input data connection, the input data connection will change to green. Read this as, "data has been made available to the glyph at this input data connection."

2.3.2.4. Data Available (DAV) (associated with Output Data Connection)

When data is made available by the operator to a subsequent glyph connected to the output data connection of the glyph, the data connection indicator will change to green. Read this as, "data has been made available by the glyph at this output data connection."

2.3.2.5. Pane Access Button

Every glyph has a pane access button in the upper-left corner in the shape of a black triangle. This button is used to display the graphical user interface, or *pane*, of the operator. The pane is used to specify values for the arguments of the operator. These values correspond to command line arguments when the operator is run outside of Cantata.

2.3.2.6. Run Button

Most glyphs have a square run button in the centre that is used to execute the operator represented by the glyph. The few glyphs that cannot be executed (i.e., the ones whose purpose is simply to provide input for other glyphs) will not have a run button. Note that the square run button glows red when the operator is executing.

2.3.2.7. Input Control Connection Node

Every glyph can have a input control connection, which is used to delay execution of the glyph until another glyph is executed. This control is represented by the small, grey square above the input data connection(s).

2.3.2.8. Output Control Connection Node

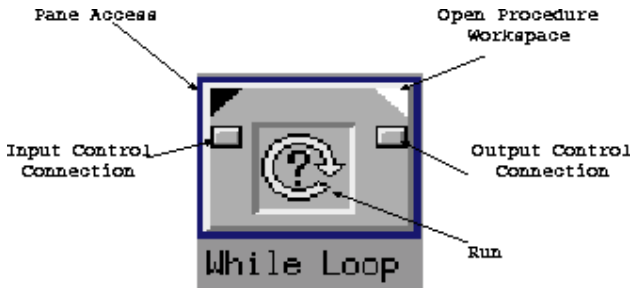
Every glyph can have an output control connection, which is used to delay execution of another glyph until this glyph is executed. This control is represented by the small, grey square above the output data connection(s).

2.3.2.9. Operator Name

Every glyph will display beneath it the name of the operator that it represents. The name can be changed by clicking on it, and then making the desired edits in the edit pop-up window.

2.3.2.10. 'Selected' Indicator

When a glyph has been selected, it will appear depressed and darker in colour than when not selected.

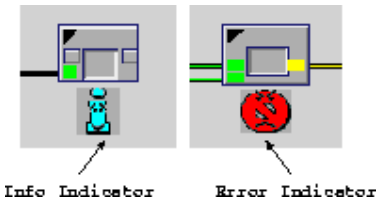


2.3.2.11. Open Workspace

Glyphs representing procedures and loop control structures have a special "Open Workspace" button which is used to open up the workspace associated with the procedure or the loop. The open workspace button is the white triangle that appears in the upper-right corner of the glyph.

2.3.2.12. Control Structure Pixmap

Glyphs representing procedures and control structures have a special pixmap displayed in the middle to indicate that the glyph in question is a procedure or a control structure; this helps to differentiate them from "regular" glyphs. Note that the pixmap is inside the square marking the run button.



2.3.2.13. Error Indicator

The error icon will appear under the glyph when the operator has encountered an error during execution. Clicking on the icon displays a message window that contains information on the error.

2.3.2.14. Info Indicator

The information icon will appear under the glyph when the operator has encountered information during execution. Clicking on the icon displays the glyph information window containing any message concerning execution of the glyph. When the mouse is moved slowly over the glyph, an identifier for each component of the glyph is printed in the status window below the workspace. When the text references an input or an output connection, the title of the input or output parameter is printed.

2.3.3. Basic Glyph Operations

2.3.3.1. Selecting a Glyph

Many of the workspace manipulation and editing capabilities in Cantata (see Section D) work on the *currently selected glyph(s)*. By *selecting* a glyph (or a set of glyphs), you are indicating to which glyphs you want a particular operation to be applied. In general, if *no* glyphs are selected when such an operation is initiated, the operation will apply to *all* glyphs in the workspace. For example, the "copy" operation will copy all the currently selected glyphs; if no glyphs are selected when the "copy" action is initiated, all the glyphs in the workspace will be copied.

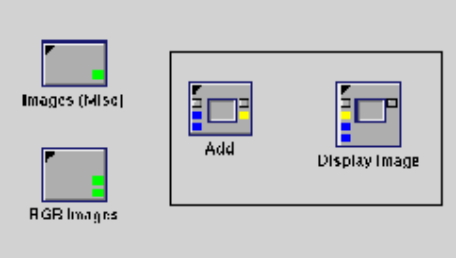


Figure 7: You can select a single glyph by clicking on it. Select a set of glyphs by outlining them with a box.

Figure 8: Selected glyphs will appear depressed and darker in color than unselected glyphs.

You can select a single glyph by clicking on it. You can select multiple glyphs by *rubber banding* (i.e., outlining with the mouse) a box around a set of glyphs. To rubber band a box, click in the workspace at the upper left corner of the area containing the glyphs that you wish to select. *Holding the mouse button down*, drag the mouse to the lower right hand corner of the area containing the glyphs that you wish to select; a box will appear following the cursor as you move the mouse. A glyph will appear depressed and darker in colour than other glyphs when selected

You can unselect selected glyphs by clicking anywhere in the workspace surface area. This will unselect all selected glyphs in the workspace. All glyphs in the workspace can be selected or unselected at once by choosing "Select All" or "Unselect All" from the Edit menu.

2.3.3.2. Moving A Glyph

First select the glyph(s) to be moved. Then, hold down the left mouse button while dragging the glyph to the desired position. Releasing the mouse button will place the glyph(s) at the new location. When moving a set of selected glyphs, simply choose one glyph to drag to the new location; all the other selected glyphs will follow along, maintaining their relative position.

2.3.3.3. Destroying A Glyph

To eliminate a glyph from the workspace, select it and then click the "Delete Selected Glyphs" button on the command bar. Multiple glyphs can be selected and deleted at the same time. You can also use the Delete item on the Edit menu. If the operator represented by the glyph is currently executing, the process will be interrupted. If you destroy a glyph by mistake, you can recover the glyph by using the "Undo Delete" feature of the Edit submenu. All the glyphs in a workspace can be deleted by clicking the Clear Workspace button in the menu bar.

2.3.3.4. Executing a Glyph

To execute the operator represented by a particular glyph individually, click on the run button of the glyph.

2.3.3.5. Renaming a Glyph

You can sometimes improve the readability of the visual program by renaming a glyph. To change the name of a glyph, click on the current name that appears underneath the glyph; a prompt in which you can enter a new name will pop up. Enter the desired name in the text box that appears in the pop-up prompt, and click "OK." The name that appears under the glyph will immediately change to the new name. While copies of the glyph will reflect the

name change, note that the change is valid only for a single instance. That is, the name for the glyph in the menu will not change.

2.3.3.6. Creation of the Glyph

Each Khoros program has an assigned *category*, *subcategory*, and *operator name* (also called the *icon name*). The use of the category/subcategory/name convention imposes a hierarchy on the Cantata operators and makes the process of finding a particular operator from the hundreds of available operators a much easier task. There are three ways to create a glyph for an existing operator in Cantata. The first way uses the category/subcategory/name approach to finding the desired operator; the second way uses a combination of category/subcategory/name organization with alphabetization; the third way uses a combination of alphabetization and key word scanning. This section explains the three methods of glyph creation in Cantata, and then goes on to describe how you can customize operators that are available as glyphs. Keep in mind that the operators available to you as glyphs in Cantata will vary according to which Khoros toolboxes you have installed at your site.

2.3.3.7. Customizing Which Operators are accessible as Glyphs

The operators available from the Glyph Menu, Accelerated Routines List, and Accelerated Finder List are dynamic. The list of available operators will change according to which toolboxes you have installed at your site. Each time Cantata is run, the items that appear in the Glyph Menu are dynamically created according to the contents of the Toolboxes file. For example, if the Design, Bootstrap, Datamanip, Envision, Geometry, Image, and Retro toolboxes are listed in the Toolboxes file, only operators from those toolboxes will be accessible from Cantata. You did not plan to use any of the programs in the Retro toolbox, for instance, deleting the Retro entry from your Toolboxes file would eliminate all references to programs in the Retro toolbox from within Cantata. Note that the categories and subcategories used by the Glyph Menu and the Accelerated Routines List can *span toolboxes*; the appearance of a particular category does not necessarily imply that its contents will come from

only one toolbox (although this is often the case). For example, the Retro and Image toolboxes both have operators in the "Image Proc" category and the "Transforms" subcategory. Thus, if you deleted the retro entry from your Toolboxes file, the "Image Proc" submenu button would still appear, since that category would still be referenced by the Image toolbox. If there are no toolboxes listed in your Toolboxes file which reference a particular category, that category will disappear from both the Toolbox Menu and the Accelerated Routines List.

2.3.4. Input/Output: Glyph Connections

2.3.4.1. Data Connections

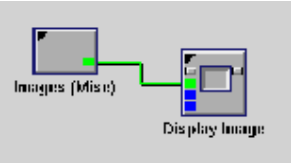


Figure 9: To become part of a network, two glyphs are connected with a data connection. Here, the data connection between the two glyphs causes the output of the "Images (Misc)" operator to become the input of the "Display Image" operator.

Data connections are an integral part of the visual program, and are required for the program's construction. Glyphs contain input and output *data connection nodes*, represented by colored squares located on the left and right sides of the glyph. To create a data connection between two glyphs, click with the mouse on the output data connection node of one glyph, and then on the input data connection node of another glyph (or vice versa). When a successful data flow connection is made, a connection line (yellow by default) will be drawn between the two glyphs. When two glyphs are connected with a data connection, it is implied that the output of the first will become the input of the second. As such, the data connection represents data flow in the visual program. If a data connection square is yellow, then its corresponding input/output parameter is a required argument for that operator. Data connection nodes representing required arguments of operators must *always* be connected to other glyphs with data connections. The "Check" item available on

the Workspace menu may be used to check a network for any missing input/output data connections.

2.3.4.2. Control Connections

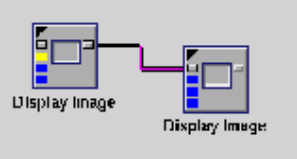


Figure 10: A control connection is made between two glyphs to prevent the second glyph from executing until the first glyph has already done so.

A visual program requires *data connections* between glyphs in order to form the network and to define where each process will obtain its data. In contrast, *control connections* are not necessary as part of a fully operational network. They do, however, allow you to constrain the operation of a visual program and provide additional control over the order in which processes are executed. Glyphs contain input and output *control connection nodes*, represented by small, grey squares just above the input/output data connection squares at either edge of the glyph. To create a control connection between two glyphs, click with the mouse on the output control connection node of one glyph, and then on the input control connection node of a second glyph. When a successful control flow connection is made, a connection line (purple by default) will be drawn between the two glyphs; the second glyph will now be *controlled* by the first glyph. Control connections simply cause the second, or *controlled* glyph, to "wait" on the execution of the operator represented by the first glyph. Thus, control connections provide a simple way of specifying an order for process execution when one is not already dictated by the data flow, as is frequently the case in networks with a number of parallel paths. Note that control connections can be created independently of data connections. In other words, a glyph that does not feed data to a second glyph can still have a control connection to that glyph.

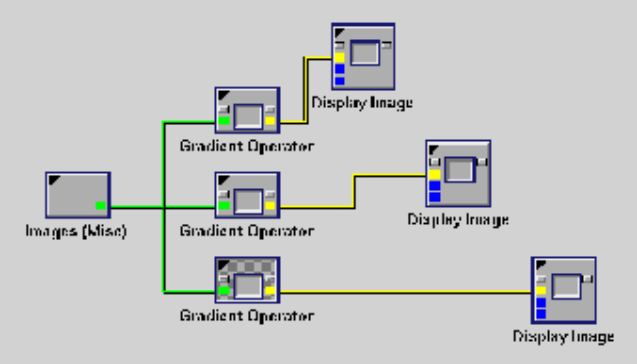


Figure 11: Without *control* connections, there is no way to predict the order in which the "Display Images" operators will be executed.

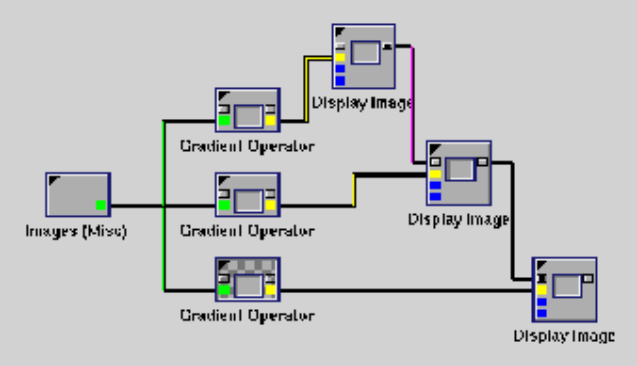


Figure 12: With the *control* connections in place, the second "Display Image" operator will not be scheduled for execution until the first "Display Image" operator has displayed its image, and the user has killed the image. In the same way, the third "Display Image" operator will be forced to wait for the second to be displayed and destroyed before it can display its image.

2.3.4.3. Manipulating Connections

Once a data flow or control connection has been made between two glyphs, it can be changed either by connecting that glyph to a different glyph or by deleting the connection altogether. Clicking the *left* mouse button on the connection between two glyphs will bring up a menu which you may use to delete the connection, save the file associated with that connection (for *data flow* connections only), or set connection options.

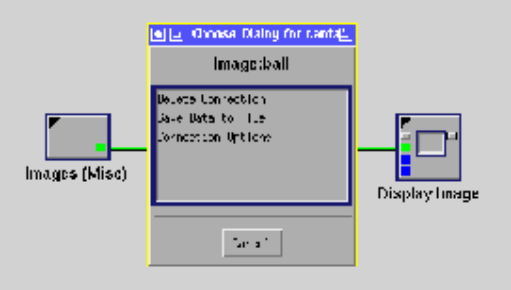


Figure 13: Clicking the *left* mouse button on a data flow or control connection between two glyphs will display a menu with which you may delete the connection, save the file associated with that connection (for *data flow* connections only), or set connection options.

2.3.4.4. Delete Connection

Choosing this selection from the menu will remove the connection.

2.3.4.5. Save Data to File

Offered as an option with data flow connections only, this item will bring up a prompt where you can enter the filename in which to save a copy of the file associated with that connection. Note that this option can only be used with permanent data transport mechanisms, specifically shared memory, standard UNIX files, or memory mapped files.

2.3.4.6. Operator Execution

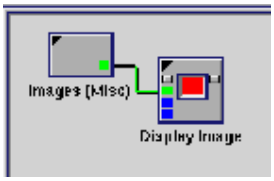


Figure 14: A single operator can be executed by clicking on the run button of its glyph. Alternatively, the entire visual program can be run by clicking on the "Run" button that appears at the far left of the workspace command bar, or by selecting "Run" from the Workspace submenu. Regardless of how the execution is initiated, both the "Run" button of the workspace command bar and the run button of the currently executing glyph(s) will be switched to the "on" position (the button on the glyph turns red) during execution.

Once a visual program has been constructed, there are two ways in which you can execute the operators represented by the glyphs:

1. You can run the entire visual program at once, where order of operator execution is determined by the data connections (and control connections, if any) of the network.
2. You can execute individually one or more glyphs of the visual program "manually".

3. Toolbox Programming

3.1. Introduction

You may interact with the Khoros software system on one or more of several levels. Each level represents a different interaction or level of programming. For example, an application user who simply wants to visualize a 2D data set will interact with the system at a different level than a toolbox programmer who will actually implement an imaging application or a data processing routine. Figure 1 below depicts the various programming or "user interaction" levels available within the Khoros development environment. Depending on the task at hand, the level at which you work with the Khoros system will vary.

This chapter will provide the toolbox programmer with an introduction to the software development tools available to facilitate the development of programs and applications within Khoros. The Khoros software development tools provide a complete environment which supports the iterative process of developing, maintaining, delivering, and sharing software. These tools act as the programmer's assistant by providing automation where possible, enforcing consistency as necessary, and hiding underlying complexity of software configuration, code generators, and documentation formatters.

Each program and library in the Khoros system is contained within a *toolbox*. A toolbox is a collection of programs and libraries that are managed as an entity. A toolbox imposes a predefined directory structure on its contents to provide consistency and predictability to software configuration. Typically, a toolbox contains programs and libraries that are characteristic of a given application domain. For example, programs that perform image processing operations might be contained within one toolbox, while programs that perform signal processing operations are contained within another toolbox.

A *toolbox object* is an encapsulation of programs and libraries; similarly, the programs and libraries themselves can also be considered as objects. A *software object* consists of the files associated with a particular library or program. There are several categories of software objects: *program objects* (which are categorized as *kroutines* or *xvroutines*), library objects, *pane objects*, and *script objects*.

The different types of software object are classified according to their purpose, the types of files associated with them, and the types of operations that can be performed on them. For example, a program object such as a kroutine embodies a program that the user will execute to perform a task; it has a user interface, source code, and documentation; operations that can be done on the program object include code generation, user interface design, source code modification, and so on. A library object, on the other hand, is simply a collection of functions which are used by programs; it has source code and documentation but there is no user interface involved; operations that can be performed on the library object exclude code generation and user interface design. The differences between the various types of software objects will be explained later in this chapter.

The Khoros software development environment supports the organization depicted above in a way that is designed to reduce the detail and complexity inherent in a large-scale software system. This environment is comprised primarily of two high-level tools, **craftsman** and **composer**, for managing toolbox and software objects respectively. **Craftsman** is used to create, delete, and copy toolbox objects and software objects; **Composer** provides the toolbox programmer with convenient access to all of the software object components and can invoke all of the operations needed to edit, and manage existing software objects. These two tools work together to provide a high level, visual environment in which toolboxes can be created and software can be written, documented, and installed..

A toolbox object provides a convenient way of presenting Khoros users with an encapsulated collection of information processing programs, interactive applications, and/or libraries designed for a specific application area. The toolbox object enforces a pre-defined directory structure in which its software objects are located; it manages both itself and its software objects via an object-

based interface to a software database. It should be noted that some of the directories below will not be created until they are needed.

A toolbox contains the following directories:

Bin

This directory is where all the executable programs from your toolbox are located.

Data

Data files that can be used with the programs in the toolbox are stored in this directory. Often, this directory may contain subdirectories indicating general categories of data, such as "images," "sequences," "signals," and so on. See the data directory in the *sampledata* toolbox for a good example of data directory organization.

Examples

This directory contains unsupported example programs that can be distributed. Example programs are generally used to demonstrate the proper use of public library calls for a library contained in the toolbox. Note that these programs are *not* formalized program objects created with **composer**, but simply manually-created directories containing a main program (no code generation involved), Imakefile (created with **kgenimake**), and Makefile (created with **kgenmake**). They generally have no documentation other than comments inside the code. For examples of the layout and implementation of example programs, see the *design* toolbox.

Include

This directory stores the public include files associated with libraries in the toolbox. The include directory contains one include file named "*toolbox*.h" for the entire toolbox. It will also have one subdirectory for each library object in the toolbox; the subdirectory is named for the respective library. Public include files for the library objects are located Inside the subdirectory. See the *design* toolbox for a model.

Lib

This directory contains the compiled archives and *.so's of any library objects that may exist in the toolbox.

Manual

This directory contains the manual for the toolbox. It will contain one subdirectory for each chapter in the manual, plus a README, an Imakefile, a Makefile, and directories for the glossary, index, and hardcopy. For examples of toolbox manuals, see the *envision* or the *datamanip* toolboxes. Do not use the *bootstrap, dataserv* or *design* toolboxes as models as they have a specialized configuration.

Objects

Software objects are located in this directory. There will be one "type" directory (named after the type) for each type of software object that exists in the toolbox; thus, there may be one or more of the *kroutine*, *xvroutine*, *script*, *library* or *pane* directories. Under the "type" directories will be one subdirectory for each software object of that type, named after the software object itself.

Repos

This directory is a repository for various files that need to be associated with your toolbox. Configuration files and the toolbox object database file are located here.

Testsuite

This directory is the location for any test suites that are created in order to test the correctness of programs or libraries in the toolbox.

3.2. Creating a Software Object (Kroutine) on a Toolbox

You will be creating a software object on a toolbox, so this toolbox must be selected in craftsman's left list. The right list will show the objects already in that toolbox (if any). To select the toolbox just click with the left mouse button on its name, then craftsman will show the toolbox objects in the right list.

3.2.1. Creating the kroutine for the first program

To create a software object in a toolbox, after selecting the toolbox click the craftsman's menu button "Object Operations" and select "Create Object". The following window will appear:

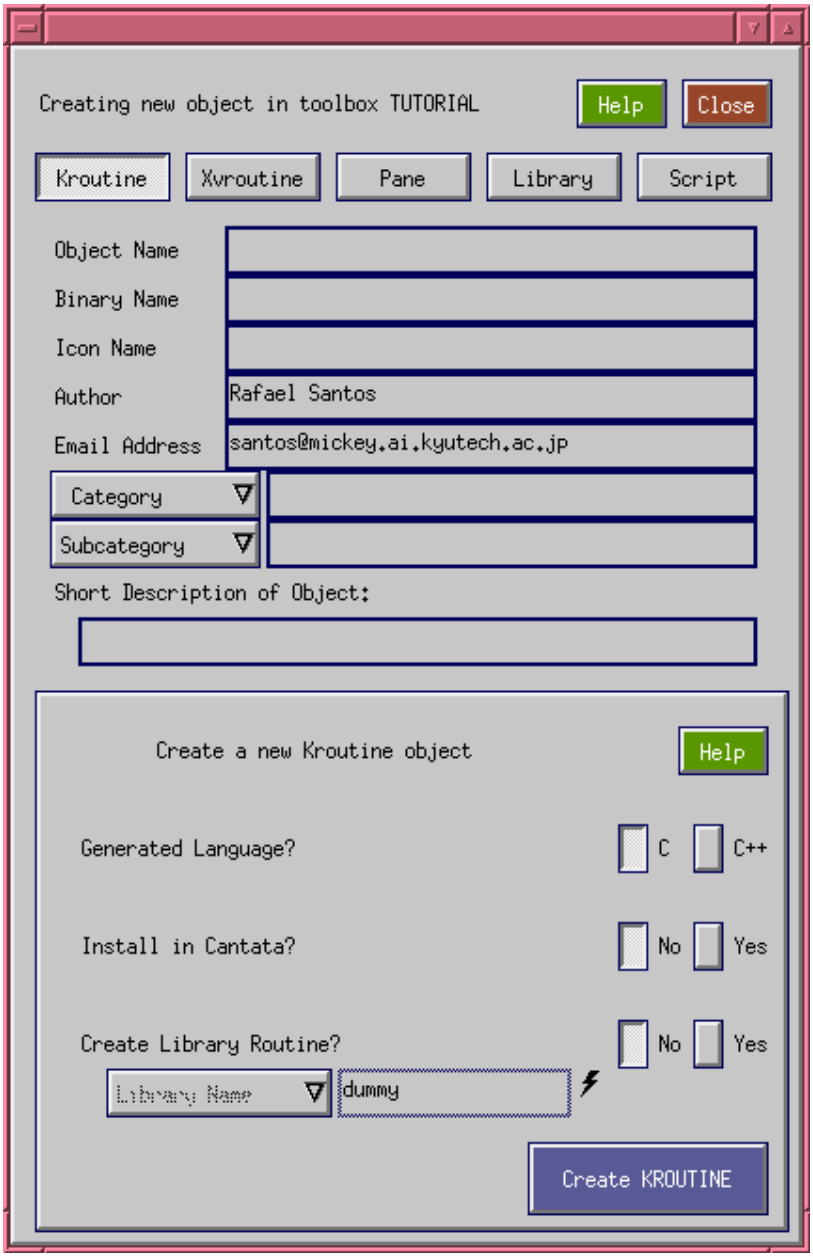


Figure 15: Creating a new object in a toolbox

First you must select which kind of object you want to create, using one of the five buttons (Kroutine, Xvroutine, Pane, Library, Script) on the top of the window. For this tutorial, we will be creating only a kroutine, so the default is OK. Please note that the options for the form changes depending on the object you're creating.

Select the Kroutine button and fill the form with the following information:

- **Object Name**: enter a single-string name for the object
- **Binary Name**: enter a single-string name for the binary (executable) object (hint: can be the same as the Object Name)
- **Icon Name**: enter a string for the glyph name for Cantata
- **Author**: your name
- **Email Address**: your e-mail
- **Category**: main menu for Cantata. Will appear when the "glyphs" button is pressed.
- **Subcategory**: main submenu for Cantata. Will appear when the "glyphs" button is pressed and the value entered in "Category" above is selected.
- **Short Description of Object**: enter a description of the object, will be useful for man pages and the finder in Cantata
- **Generated Language?** If you selected "Alpha C++ support" when compiling Khoros, it will allows you to select between C and C++, otherwise only C will be available (at least in version 2.1). For the examples in these pages, use always C.
- **Install in Cantata?** Select "Yes" so the object will be installed in Cantata.
- **Create Library Routine?** Useful if you want your object functionality to be callable from other objects. In other words, your kobject will be just an interface to a library function, and this library function will be available for other objects as well. If you select "Yes" craftsman will ask which library you want to create/add this object into. For this tutorial, select "No".

Entering some values in the fields the window will look like:

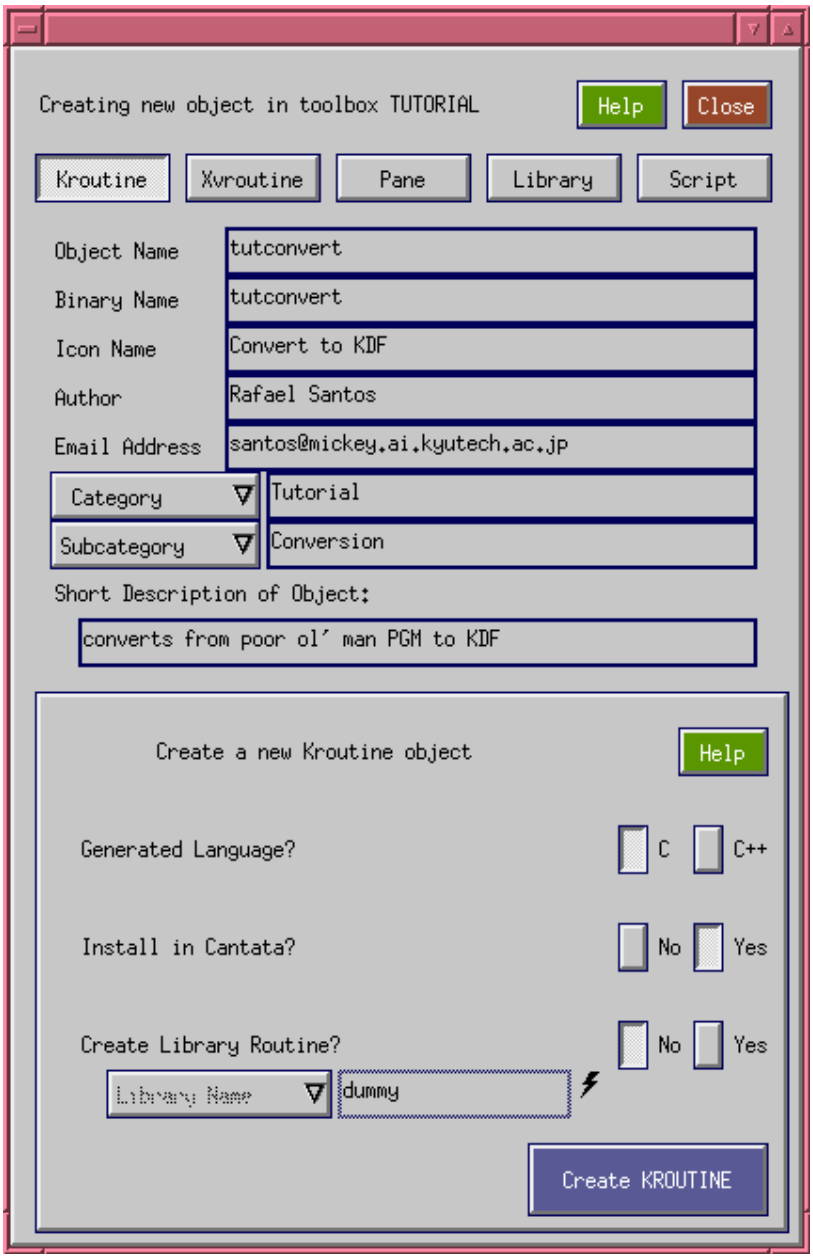


Figure 16: Creating a new object in a toolbox (filled)

To create the object, click on the "Create KROUTINE" button. After some seconds, the software object will be created and you can click the "Close" button.

With this step the object is created, with an user interface and skeleton C program. On the next step we will edit the object to add functionality to it.

3.2.2. Modifying the User Interface of the first program

The object created with craftsman can be compiled and executed - it just won't do nothing. To add functionality to the object, you must change (if needed) its user interface and add code to the skeleton code generated by craftsman and ghostwriter .

To edit an software object you need to select it (left-click the mouse on it) and select the option "Edit Object (Composer)" from the menu button "Object Operations" in craftsman. It will call composer, which will looks like:

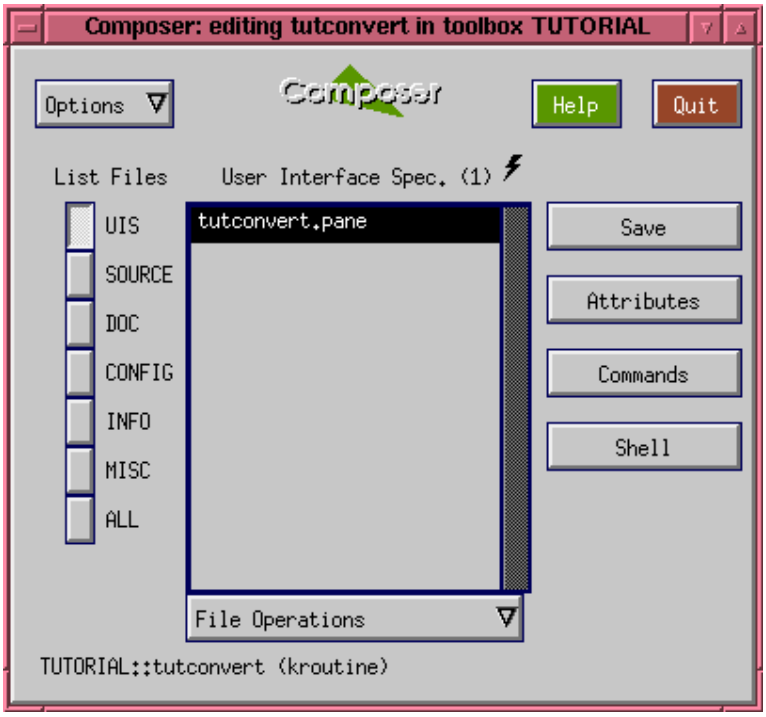


Figure 17: Composer editing the tutconvert kroutine

The left part shows a column of buttons that controls which kind of files will appear on the file list on the middle. The buttons and corresponding file types are:

- **UIS**: User Interface Specification files - files which are related to the graphical user interfaces of the programs. For panes, scripts and kroutines there will be a single file in this list (a pane file) for the user interface specification. For xvroutines, there will be at least two files, one for the user interface for parameter selection (pane) and other for the user interaction (form). For more details about user interfaces, see the User Interface section.
- **Source**: Source code for the software objects. Will depend on which kind of object is being created/edited, usually will be a list of .c and .h files for kroutines, libraries and xvroutines.
- **Doc**: The documentation for the software object. Will list man pages and help files for the objects.
- **Config**: Will list the makefiles for the object plus a configuration file (cms) with several keywords and information but the object that can be viewed but not edited.
- **Info**: Will contain a changelog file for the object.
- **Misc**: Will contain miscellaneous files associated to the object.
- **All**: Will list all files in all categories.

A "File Operation" button on the bottom of the list will allow you to view or edit these files (not all files can be edited or viewed). Depending on the object, it will be edited/viewed with a text editor or with a special program.

3.2.3. Editing the first kroutine's user interface

The first thing we can do is change the user interface of the object, which by default contains one input and one output object. For our sample kroutine, let's suppose we need another input, to allow the user the selection of which kind of data will be generated. For this, must change the default generated user interface that is defined in the file tutconvert.pane.

To edit the user interface, it must be highlited in the file list (the button "UIS" must be selected to show the list of user interface objects, click on the name of the object you want to edit). If it is selected, just click the "File Operations" button an select "Guise" to visually modify the user interface of the object. That should call guise , the Graphical User Interface Speficication Editor, and bring two different windows:

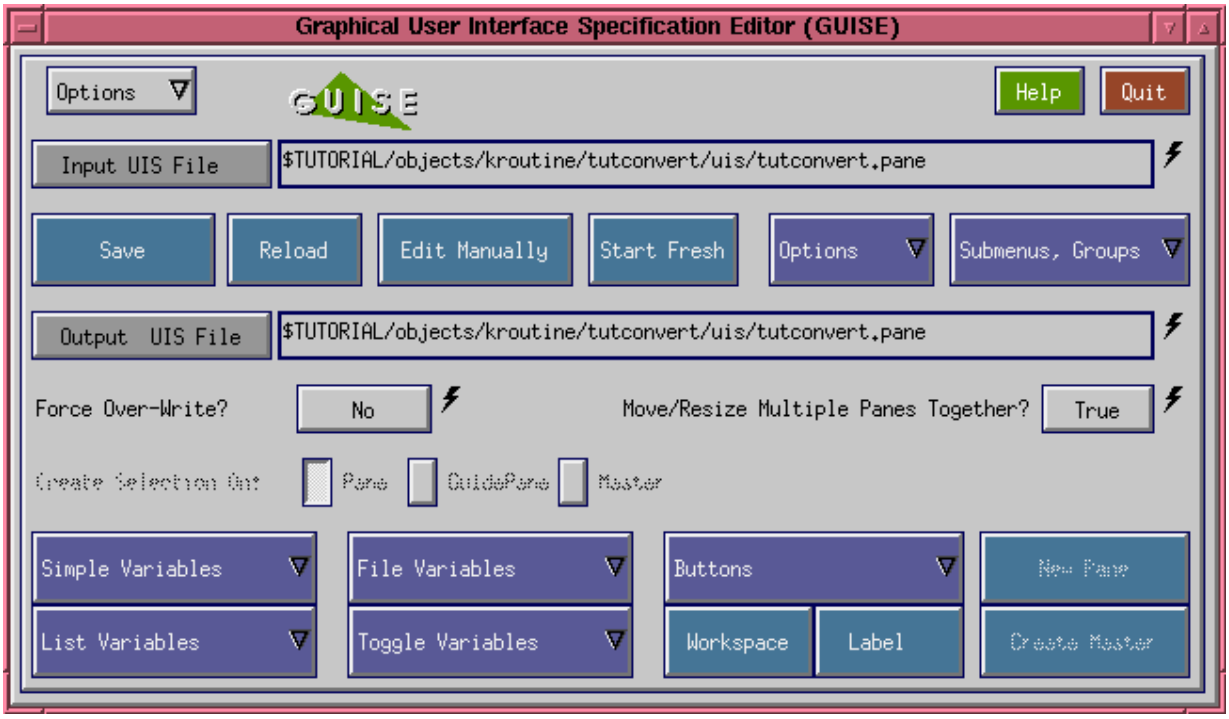


Figure 18: Guise Controls

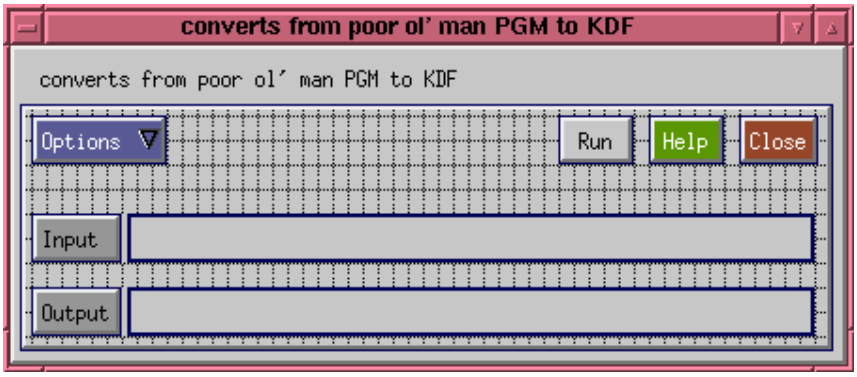


Figure 19: Initial pane for the tutconvert kroutine

As mentioned before, when the kroutine was created automatically, the user interface was also created automatically with only one input and output object. Of course, depending on the functionality of the program, there can be several (or none) input and output objects and other parameters that can be set from the pane. It is possible to add integers, flags, doubles, strings, toggles, lists, multiple choice items, input and output files, etc. objects to the user interface.

In our case, let's consider another parameter, which will control the data type of the KDF output object, which can be either integer or double. For this, we can either add a toggle or list variable. Let's choose a toggle - click on the "Toggle Variables" button on the Guise Controls window and select "Flag". A pop-up window will appear and ask how many options you want on the flag - enter 2 instead of the default 3 and click "OK". A Flag Toggle variable will appear on the pane of the object:

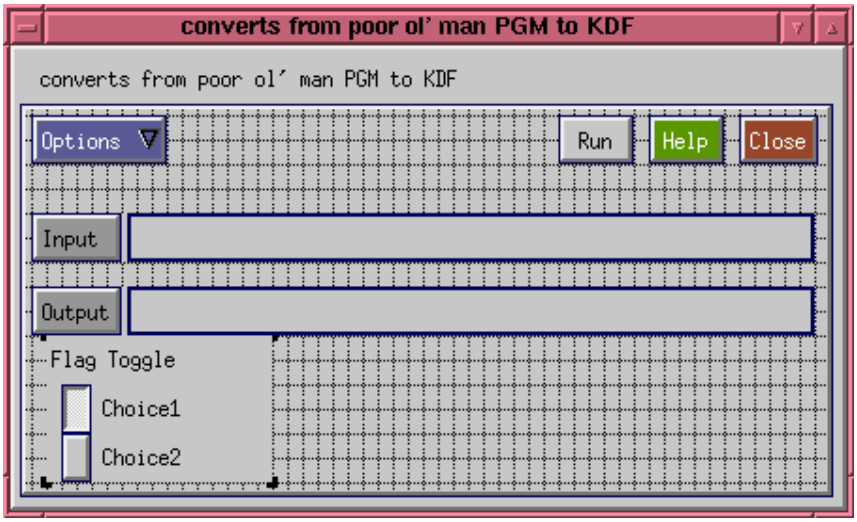


Figure 20: Pane for the tutconvert kroutine after adding a flag toggle variable

Note that the Flag Toggle object is *marked* on the pane - meaning that it can be moved, changed, etc. To mark another field in the pane just click with the left mouse on it. To edit a field click with the middle button on it. Clicking with the middle button on the Flag Toggle field will bring the window:

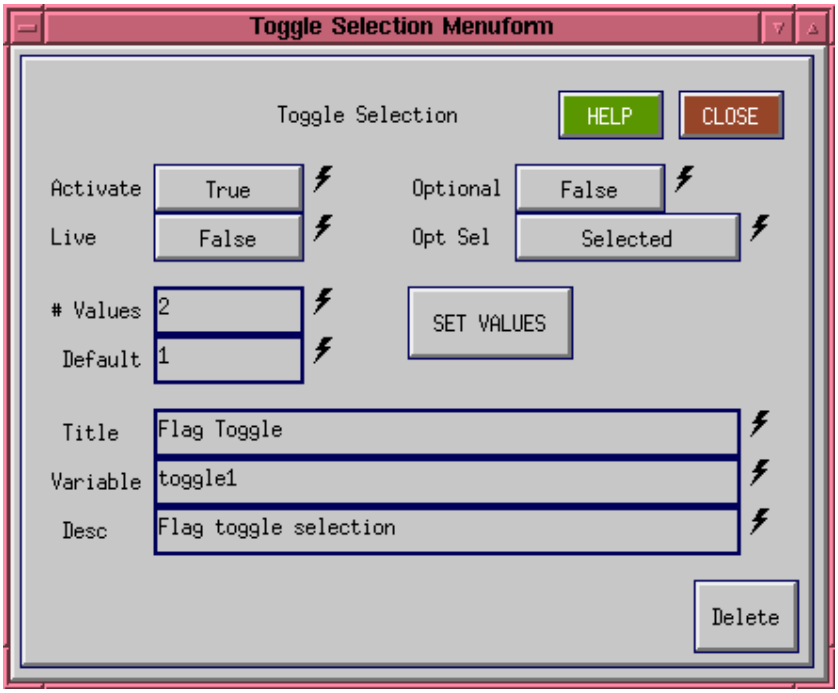


Figure 21: Toggle Selection Menuform

In this window we can change several of the parameters of the flag toggle field. For example, we will change the title of the field (Title), the variable that will be generated in the source code for that field's value (Variable), the description (Desc) and the toggle values (which appears as default as Choice 1 and Choice 2) (Set Values). Clicking the "Set Values" button will allow us to change the values, bringing the window:

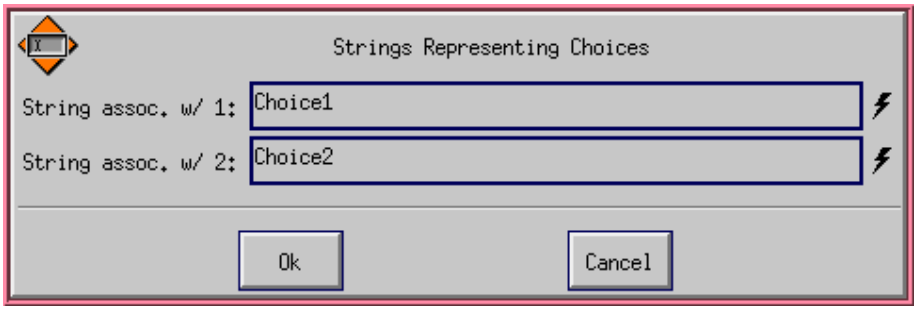


Figure 22: Set values window – default

On which we will enter the desired values:

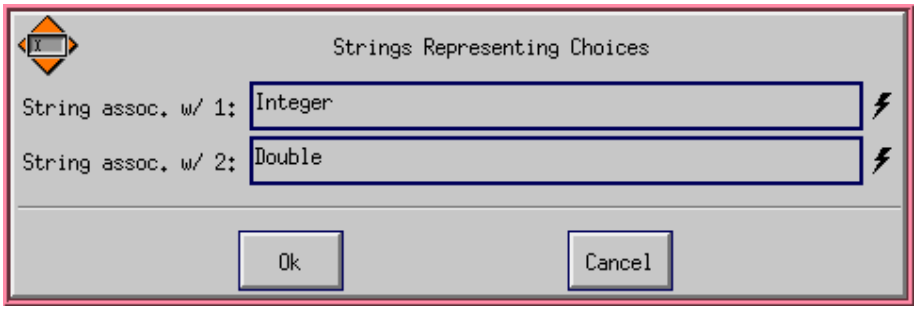


Figure 23: Set values window - filled with our choices

Clicking the button OK will bring another window with the strings that will be used for the toggle values, fill this window too:

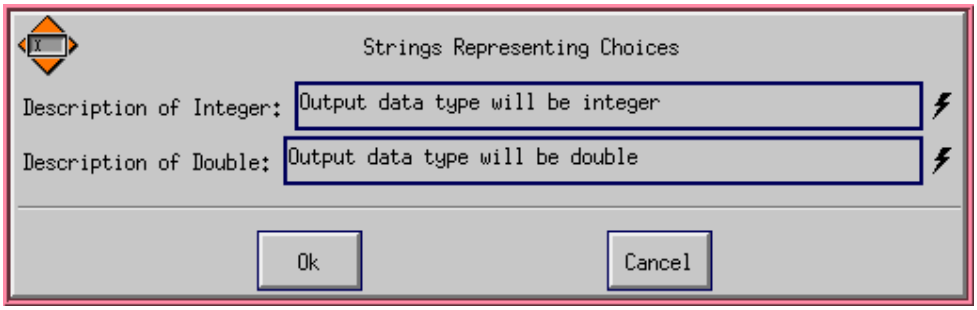


Figure 24: Set values window - filled with descriptions of our choices

Clicking the button OK will return us to the flag toggle field parameters editor, on which we can enter/modify the other parameters:

- **Title**: the title that will appear on the user interface
- **Variable**: the name of the variable that will be used for code generation
- **Desc**: the description of what is the variable

All these fields are important either for the user interface or for code generation, I suggest you fill all of them so your generated user interface and code will be easy to understand.

After filling the fields above the window should looks like:

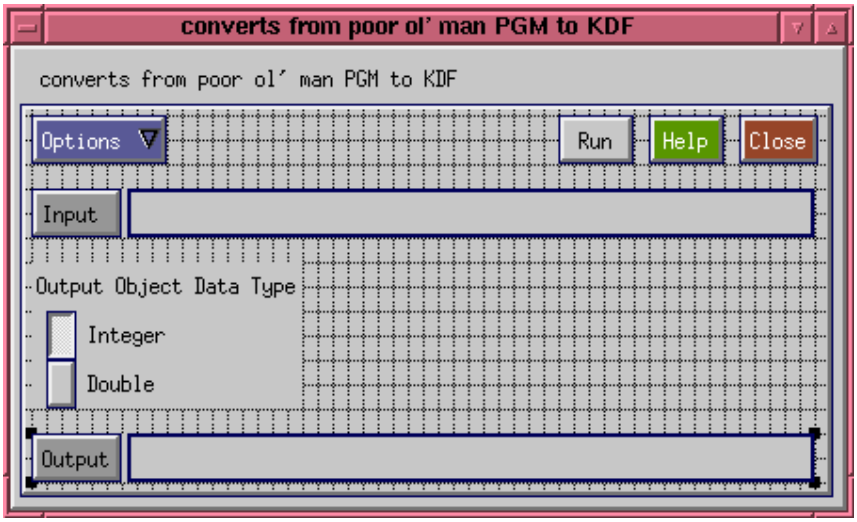


Figure 25: Toggle Selection Menuform filled with our choices

Click the Close button and the pane will be updated with the new information for the flag toggle object. After marking and dragging the fields so one will not overlap the others (or for aesthetic reasons), our pane should look like:

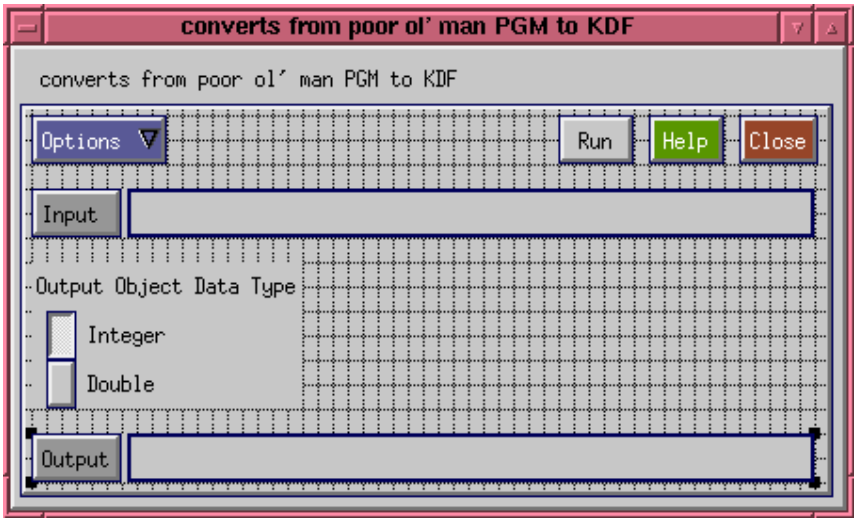


Figure 26: Pane for the tutconvert kroutine after edition

Note that you could also change the titles, variable names and descriptions of the variables associated with the input and output files, but this will not be necessary for our example.

If all modifications are done, we can close guise now - click on the "Save (Needed)" button on guise and after confirmation, click on the "Close" button on the guise panel - **not** on the software pane we were editing. We should be back on the composer window.

At this point, the code generated when the object was created is not synchronized with the modifications we made on the pane (user interface). To synchronize them, you **must** click the "Commands" button which will bring the window:

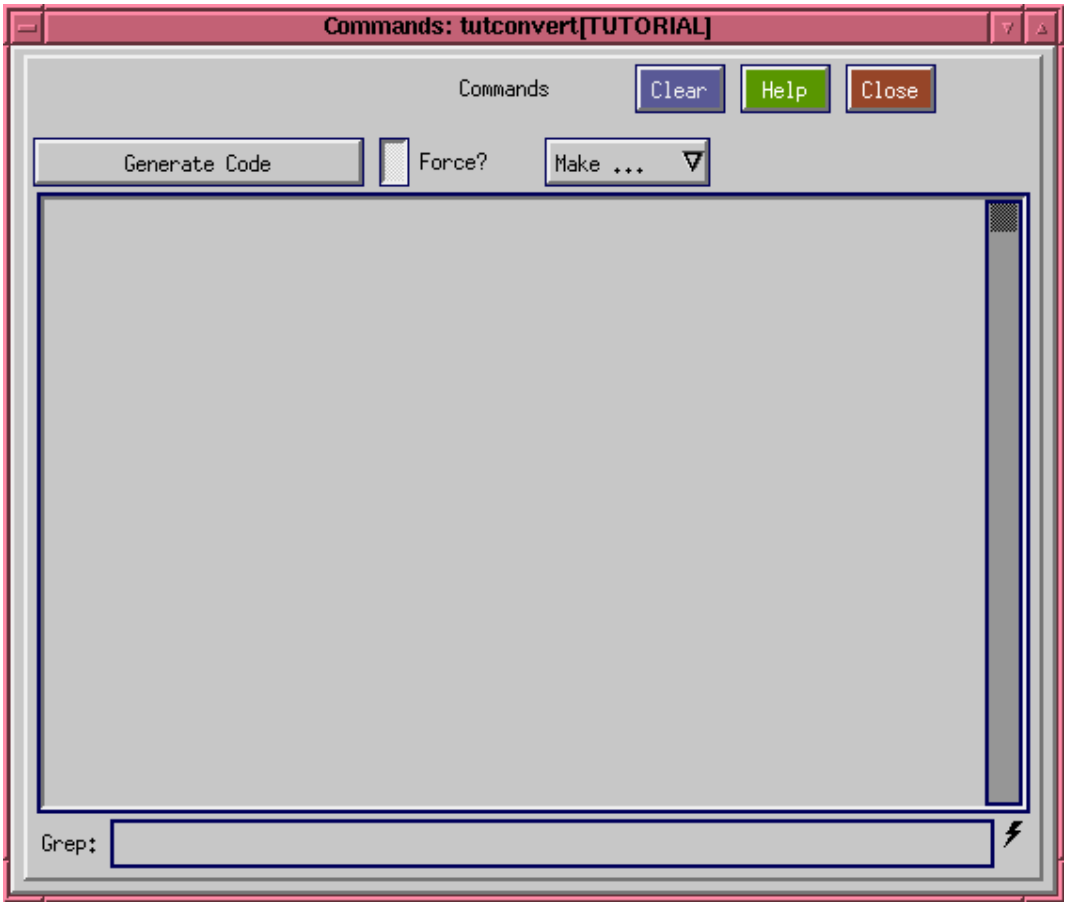


Figure 27: Composer commands

And then click on the button "Generate Code". It will regenerate all related code so the object will be synchronized - messages will be displayed in the Commands window. The next steps will cover examining and modifying the source code of the object to add functionality.

3.2.4. Examining the code of the first program

Composer / ghostwriter automatically generate the code for our kroutine. To see which files are generated, click on the "Source" button of composer's panel - three files will appear on the file list, as the image shows:

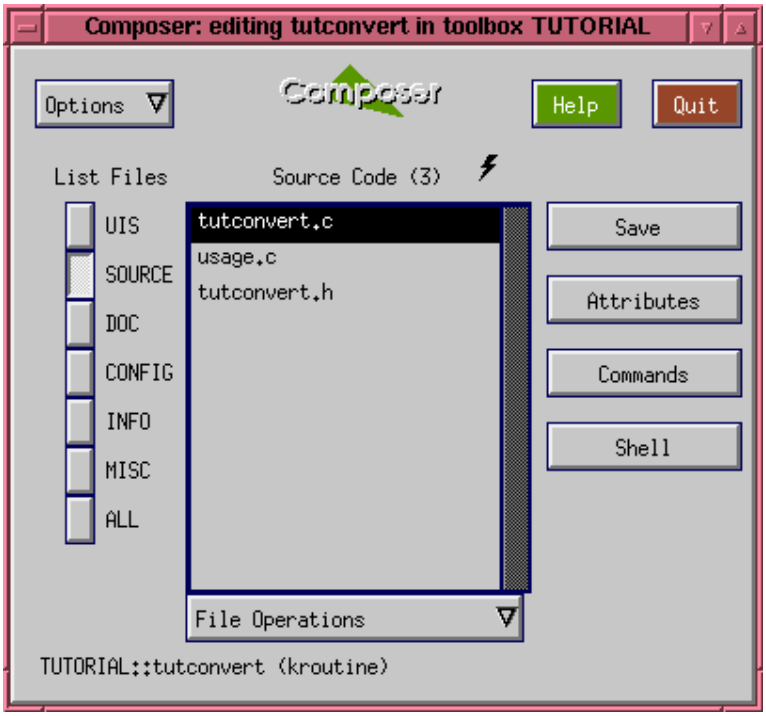


Figure 28: Source codes in Composer

The automatically generated files are:

- **convert.c** - the main program
- **usage.c** - routines for getting the command line or use interface parameters
- **convert.h** - header file

We will need to modify the main C program only, or possibly add some minor stuff on the header file. Before modifying any program, let's see the concept of **tag** - tags are predetermined strings on the code that are recognized by some programs to be of special significance. In this case, some tags allows the user to modify parts of the code that will not be touched case composer (actually ghostwriter, the code writer) need to rewrite the code. It means that you **must** write your code between the ghostwriter tags or it will be **lost** when the code is regenerated. It also means that almost all the code you will have to

write is located between the tags and that everything outside the tags was written by ghostwriter and should be left as is.

3.3. Short Introduction to the Polymorphic Data Model

This page is just a short introduction to the Polymorphic Data Model (PDM). The PDM can represent data in up to 5 dimensions, where three are spatial (*width*, *height* and *depth*), one is temporal (*time*) and each point of data can be considered as a vector of *elements*. This is shown in the figure below, where W=width, H=height, D=depth, T=time and E=elements:

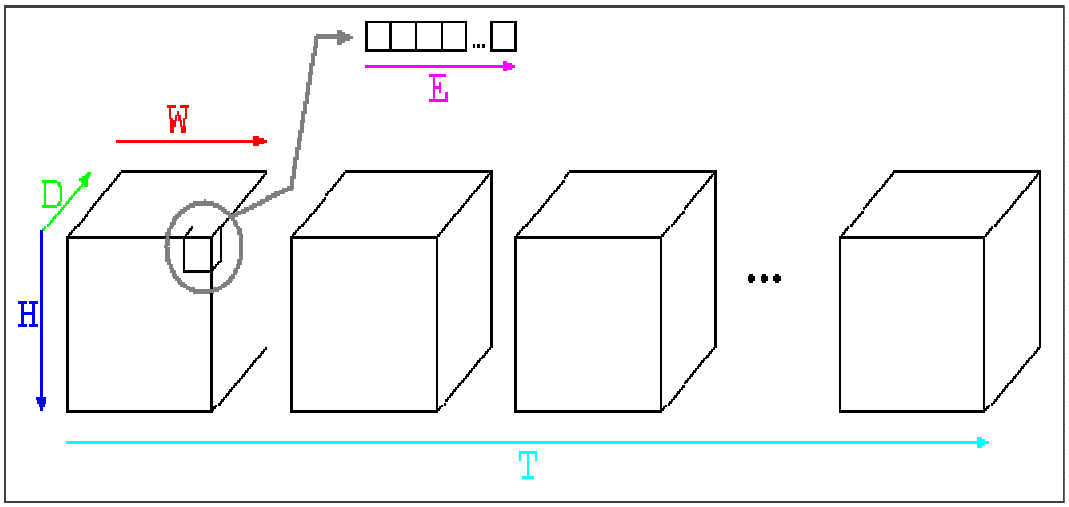


Figure 29: 5 dimensions of the Polymorphic Data Model

The data is represented in segments in the PDM: the *value* segment represents the data itself, the *mask* segment corresponds in dimensions to the value segment and serve to identify which points in the value segment are valid or not. The *map* segment serves to associate vectors of data to indexes, so a value in the value segment will be an index in the map table. The *location* segment explicitly locates a vector in space while the *time* segment explicitly

locates a volume in time. Not all segments has the same dimensions due to the information they are meant to represent:

- The *value* segment uses the 5 data dimensions of the PDM.
- The *mask* segment also uses the 5 data dimensions of the PDM since every point in the value segment can be masked as valid (1) or non-valid (0).
- The *map* segment will have independent dimensions of the value segment. The map segment can be considered as a table which entries (rows) can have several elements (columns). When using a map segment, each value in the value segment will point to a row in the map segment.
- The *location* segment will use the width, height and depth dimensions to explicitly locate a point in the value segment in space.
- The *time* segment will be a single vector with the same time dimension as the value segment, to explicitly locate a time volume in time.

Bibliography

"A wavelet-based Framework for Face Recognition" **Christophe Garcia, Giorgos Zikos, Giorgos Tziritas.**

"Face Recognition using the Discrete Cosine Transform" **Ziad M.Hafed, Martin D. Levine.**

"Face Recognition: A comparison of Appearance-Based Approaches" **Thomas Heseltine, Nick Pears, Jim Austin, Zezhi Chen.**

"Evaluation of Image pre-processing techniques for eigenface based face recognition" **Thomas Heseltine, Nick Pears, Jim Austin.**

"An efficient LDA Algorithm for Face Recognition" **Jie Yang, Hua Yu, William Kunz.**

"Face Image Resolution versus Face recognition performance based on two global methods" **Jingdong Wang, Changshui Zhang, Heng-Yeum Shum.**

"Thesis: Elastic Bunch Graph Matching" **David S. Bolme.**

"Three Approaches for Face recognition" **V.V. Starovoitov, D.I. Samel, D.V. Briliuk.**

"Information Access using Speech, Speaker and Face Recognition" **M. Viswanathan, H.S.M. Beigi.**

"Look who's talking: Speaker Detection using Video and Audio correlation" **Ross Cutler, Larry Davis.**

"Towards robust Face Recognition from Video" **Jeffery R. Price, Timothy F. Gee.**