

TDA: Imagen

Salvador Romero Cortés

Toda la práctica está documentada con Doxygen y se puede generar con el script

`scripts/gen_doc.sh`

TDA

Esta práctica consiste en realizar un proyecto en C++ con un TDA Imagen que representa una imagen en formato PGM. Este formato es capaz del almacenar imágenes digitales en una escala de grises que va desde 0 (mínima luminosidad, negro) hasta 255 (máxima luminosidad, blanco).

Las operaciones básicas asociadas son las siguientes:

1. Creación de una imagen
2. Destrucción de una imagen
3. Consultar el número de filas de una imagen
4. Consultar el número de columnas de una imagen
5. Asignar un valor a un punto de la imagen
6. Consultar el valor de un punto de la imagen.

El tipo rep escogido es una clase en C++ con los siguientes campos:

```
typedef unsigned char byte;
//...
class Imagen{
private:
    int filas;
    int columnas;
    byte **imagen;

public:
    Imagen(int filas, int columnas); //1

    Imagen(const Imagen & otra); //1

    Imagen (int filas, int columnas, byte * vector); // 1

    ~Imagen(); // 2

    Imagen & operator=(const Imagen &otra);

    int num_filas() const; // 3

    int num_columnas() const; // 4

    void asigna_pixel(int fila, int columna, byte valor); // 5

    byte valor_pixel(int fila, int columna) const; // 6

private:
    void reservar(int filas, int columnas);
```

```
void liberar();

void copiar(const Imagen &otra);
};
```

Este código está documentado en `inc/imagenES.h` y se puede consultar su documentación generándola con el script mencionado al principio del documento.

Los métodos públicos de la clase representan las operaciones descritas anteriormente. El único añadido es el constructor

```
Imagen (int filas, int columnas, byte * vector);
```

Este constructor admite como parámetros el número de filas y de columnas de la imagen junto con un vector unidimensional de los píxeles de esta. Se recorre el vector con dos índices, `i` para las filas y `j` para las columnas. De esta manera podemos interpretar el vector como una matriz y crear nuestra imagen.

```
asigna_pixel(i,j,vector[i*columnas+j])
```

Funciones auxiliares

Estas son una serie de funciones que sirven para facilitar la implementación de otras funciones más complejas o para facilitar la entrada/escritura de las imágenes.

```
void error(std::string mensaje);
```

Esta función implementa la posibilidad de mostrar un mensaje de error por pantalla y abandonar el programa. Es importante liberar la memoria dinámica antes de llamar a la función.

```
Imagen leerVectorPPM(byte * vector, int filas, int columnas);
```

Esta función crea un objeto imagen a partir de un vector de imagen **PPM**. Esta función se explicará con más detalle más adelante.

```
void escribirVectorPGM(const Imagen &img, byte *vector, int filas, int columnas)
```

Esta función hace el proceso inverso a lo que hacía el constructor mencionado anteriormente. Modifica el parámetro `byte * vector` con los valores de la imagen pasada como parámetro. Al no devolver nada y modificar el parámetro es necesario crear el vector antes de llamar a la función.

```
bool escribirImagen(const Imagen &img, const char *nombre_archivo)
```

Esta función se basa en la función anterior (`void escribirVectorPGM(...)`) para escribir en disco la imagen pasada como parámetro.

```
double transformacion_morph(byte s, byte d, double a_i){
```

Esta función sirve para simplificar el uso de la función de `morphing`. Calcula y devuelve la siguiente expresión:

$$P(x, y) = a_i * O(x, y) + (1 - a_i) * D(x, y)$$

Donde $O(x, y)$ es el parámetro `byte s` y $D(x, y)$ es `byte d`.

Todas estas funciones están documentadas en `inc/funciones_imagen.h` y se puede consultar su documentación generándola con el script mencionado al principio del documento.

Funciones

Se definen una serie de funciones extra sobre el TDA Imagen. Estas son:

1. Umbralizar una imagen usando una escala de grises.
2. Zoom de una porción de la imagen.
3. Aumento de contraste de una imagen mediante una transformación lineal.
4. Simular un efecto especial de morphing.

Todas las funciones siguen el mismo esquema de :

1. Comprobar que los argumentos son correctos
2. Crear imagenes con los parámetros pasados
3. Realizar la operación sobre la(s) imagen(es).
4. Escribir en el disco el resultado y liberar la memoria dinámica usada.

1. Umbralizar una imagen usando una escala de grises

Esta función discrimina (pone como blanco) los píxeles que no se encuentren en un umbral determinado de luminosidad.

La cabecera de la función es la siguiente.

```
void umbralizar_escala_grises(const char *original, const char *salida,
                             int umbral_min, int umbral_max);
```

- Comprobar que los argumentos son correctos

```
if (umbral_min >= umbral_max || umbral_min < 0 || umbral_max > 255)
    error("Intervalo de escala de grises incorrecto");
```

Comprobamos que el mínimo del umbral sea inferior al máximo así como que sea al menos 0. En el caso del máximo, que sea superior al mínimo y menor o igual a 255.

- Creación de las imágenes

```
int filas, columnas;
byte *vector_original = LeerImagenPGM(original, filas, columnas);
Imagen img_salida(filas, columnas, vector_original);
```

Primero obtenemos el vector unidimensional de bytes, usando la función proporcionada por el profesor. Usamos este vector para construir la imagen.

- Operaciones

```

for (int i = 0; i < filas; i++){
    for (int j = 0; j < columnas; j++){
        if (!(img_salida.valor_pixel(i, j) >= umbral_min &&
            img_salida.valor_pixel(i, j) <= umbral_max))
            img_salida.asigna_pixel(i, j, BLANCO);
    }
}

```

La operación consiste en recorrer los píxeles de la imagen y sustituir aquellos que no se encuentren en el intervalo `[umbral_min, umbral_ax]`.

- Escribir el resultado y liberar memoria

```

if (!escribirImagen(img_salida, salida)){
    delete [] vector_original;
    error("Error al escribir la imagen");
}
delete[] vector_original;

```

Se llama a la función auxiliar comentada previamente y se libera la memoria dinámica ocupada por el vector del principio. La memoria de la imagen se libera automáticamente con el destructor de la clase.

En caso de error se avisa usando la función auxiliar.

2. Zoom de una porción de una imagen

Esta función hace zoom sobre una zona *cuadrada* de la imagen. El recorte de tamaño $N \cdot N$ es aumentado a un tamaño de $2(N - 1)$. Se interpolan los píxeles con los de alrededor para intentar evitar la pérdida de información.

La cabecera de la función es la siguiente.

```

void zoom(const char *entrada, const char *salida, int x1, int y1, int x2, int y2);

```

- Comprobar que los argumentos son correctos

```

if (x1 >= x2 || y1 >= y2)
    error("Posiciones zoom incorrectas");
else if ((x2 - x1) != (y2 - y1))
    error("El recorte no es cuadrado");

```

Se comprueba que realmente el par `x1,y1` sea la esquina superior izquierda y que el par `x2,y2` sea la inferior derecha. También se comprueba los tamaños verticales y horizontales de la imagen ya que el recorte debe ser cuadrado.

- Creación de las imágenes

```

byte *vector_original = LeerImagenPGM(entrada, filas, columnas);
Imagen original(filas, columnas, vector_original);
Imagen interpo_colum(tam_recorte, dimension);
Imagen final(dimension, dimension);

```

Primero obtenemos el vector unidimensional de bytes, usando la función proporcionada por el profesor. Usamos este vector para construir la imagen.

Creamos además dos imágenes más: `interpo_colum` que tiene el ancho final $2(N - 1)$ y la altura N . Esta es una imagen intermedia que tiene los valores de la columna y la interpolación de estas.

La imagen `final` es el resultado de la función, es la que se escribe en disco y tiene un tamaño de $2(N - 1)$. Contiene la información de `interpo_colum` y las filas (y su interpolación) restante.

- Operaciones

```
//copiar columnas pares
for (int i = 0, x_orig = x1; i < tam_recorte; i++){
    for (int j = 0, y_orig = y1; j < dimension; j += 2){
        interpo_colum.asigna_pixel(i, j, original.valor_pixel(x_orig,
y_orig++));
    }
    x_orig++;
}
//interpolacion
for (int i = 0; i < tam_recorte; i++){
    for (int j = 1; j < dimension; j += 2){
        int media = (interpo_colum.valor_pixel(i, j - 1) +
                    interpo_colum.valor_pixel(i, j + 1)) / 2.0;
        interpo_colum.asigna_pixel(i, j, media);
    }
}
```

Primero trabajamos sobre la imagen `interpo_colum`. Copiamos las columnas pares de la imagen original. Usamos dos pares de contadores, uno para la imagen nueva y el otro para recorrer la imagen original.

A continuación se rellenan las columnas impares con las medias de los píxeles anteriores y siguientes.

```
//copiar filas
for (int i=0; i < dimension; i+=2){
    for (int j=0; j < dimension; j++){
        final.asigna_pixel(i,j,interpo_colum.valor_pixel(i/2,j));
    }
}
//interpolacion
for (int i=1; i < dimension; i+=2){
    for (int j=0; j < dimension; j++){
        int media = (final.valor_pixel(i-1,j) + final.valor_pixel(i+1,j)) /
2.0;
        final.asigna_pixel(i,j,media);
    }
}
```

Primero se copian las filas pares de la imagen creada anteriormente. En este caso, trabajamos con un único par de contadores puesto que en este caso se simplifica más si simplemente dividimos (división truncada) entre 2 el índice de las filas.

[Puede que se pueda hacer con contadores, pero esta es la manera más sencilla que encontré de hacerlo]

Luego se rellenan las filas impares con las medias de los píxeles anteriores y siguientes.

- Escribir el resultado y liberar memoria

```
if (!escribirImagen(final, salida)){
    delete [] vector_original;
    error("Error al escribir la imagen");
}
delete[] vector_original;
```

Se llama a la función para escribir en el disco y se libera la memoria del vector. En caso de error se avisa con la función correspondiente.

3. Aumento de contraste de una imagen mediante una transformación lineal.

Esta función realiza una transformación lineal sobre los píxeles de la imagen con el objetivo de modificar el contraste.

La cabecera de la función es la siguiente.

```
void contrastar(const char *original, const char *salida, int minimo, int
maximo);
```

- Comprobar que los argumentos son correctos

```
if (minimo >= maximo)
    error("Intervalo de contraste incorrecto");
```

Se comprueba que el mínimo sea menor al máximo.

- Creación de las imágenes

```
int filas, columnas;
byte *vector_original = LeerImagenPGM(original, filas, columnas);
Imagen img_salida(filas, columnas, vector_original);
```

Primero obtenemos el vector unidimensional de bytes, usando la función proporcionada por el profesor. Usamos este vector para construir la imagen.

- Operaciones

```
byte a = 255;
byte b = 0;
for (int i = 0; i < img_salida.num_filas(); i++){
    for (int j = 0; j < img_salida.num_columnas(); j++){
        if (img_salida.valor_pixel(i, j) < a)
            a = img_salida.valor_pixel(i, j);
        if (img_salida.valor_pixel(i, j) > b)
            b = img_salida.valor_pixel(i, j);
    }
}
```

Primero obtenemos el mínimo y el máximo valor de la imagen.

A continuación calculamos la parte constante de la transformación lineal (al ser constante, nos ahorramos cálculos al hacerlo antes de recorrer la imagen).

```
double constante = (maximo * 1.0 - minimo * 1.0) / (b * 1.0 - a * 1.0);
```

Ahora aplicamos la transformación lineal

```
for (int i = 0; i < img_salida.num_filas(); i++){
    for (int j = 0; j < img_salida.num_columnas(); j++) {
        double actual = img_salida.valor_pixel(i, j);
        double nuevo = (actual - a) * constante + minimo;
        if (nuevo > 255)
            nuevo = 255;
        img_salida.asigna_pixel(i, j, round(nuevo));
    }
}
```

Le vamos aplicando a cada pixel la transformación correspondiente. En caso de que sea superior a 255 el resultado, le asignamos el valor 255 (blanco) puesto que es el límite de la imagen PGM.

- Escribir el resultado y liberar memoria

```
if (!escribirImagen(img_salida, salida)){
    delete [] vector_original;
    error("Error escribiendo la imagen");
}
delete[] vector_original;
```

Se llama a la función para escribir en el disco y se libera la memoria del vector. En caso de error se llama a la función correspondiente.

4. Simular un efecto especial de morphing.

Esta función realiza una transformación lineal sobre los píxeles de la imagen de manera que una imagen va lentamente transicionando hacia otra.

⚠ **SE NECESITA CREAR UN DIRECTORIO LLAMADO `res_morphing` EN EL DIRECTORIO ACTUAL PARA GUARDAR LAS IMÁGENES INTERMEDIAS** ⚠

La cabecera de la función es la siguiente.

```
void morphing(const char * fuente, const char * destino,
              const char * basename, int pasos);
```

- Comprobar que los argumentos son correctos

```
if (pasos <= 0){
    error("El número de pasos debe ser mayor que 0");
}
```

Se comprueba que el número de pasos sea mayor que 0 puesto que en caso contrario no se haría nada.

- Creación de las imágenes

```

int filas1, filas2, columnas1, columnas2;
byte * v_fuente = LeerImagenPGM(fuente,filas1,columnas1);
byte * v_destino = LeerImagenPGM(destino,filas2,columnas2);
if (filas1 != filas2 || columnas1 != columnas2){
    delete [] v_fuente;
    delete [] v_destino;
    error ("Las imagenes no son del mismo tamaño");
}

Imagen source(filas1,columnas1,v_fuente);
Imagen target(filas2,columnas2,v_destino);
Imagen intermedia(source);

```

Leemos las dos imágenes y comprobamos que sean del mismo tamaño. Después se crean los objetos de imagen. Uno para la imagen desde la cual comienza la transición, otra es a la que se dirige la transición y otra para almacenar los pasos intermedios. Esta última es una copia de la primera porque se parte de esa para el morphing.

- Operaciones

```

double incremento = 1.0/pasos;
int contador = 0;
char extension [] = ".pgm";
char carpeta [] = "res_morphing/";
char * salida = new char
[strlen(basename)+sizeof(int)+strlen(extension)+strlen(carpeta)];

for (double a_i = 1.0; a_i >= 0.0; a_i-=incremento){
    for (int i=0; i < filas1; i++){
        for (int j=0; j < columnas1; j++){
            double trans =
tranformacion_morph(source.valor_pixel(i,j),target.valor_pixel(i,j),a_i);
            intermedia.asigna_pixel(i,j,trans);
        }
    }
    sprintf(salida,"%s%s%d%s",carpeta,basename,contador,extension);
    if (!escribirImagen(intermedia,salida)){
        delete [] salida;
        delete [] v_fuente;
        delete [] v_destino;
        error("Error escribiendo las imágenes");
    }
    contador++;
}

```

Primero creamos una serie de variables que nos serán útil más adelante.

Recorremos la imagen tantas veces como pasos y como imágenes intermedias vamos a generar.

Estas iteraciones usan un contador que parte de 1 y disminuye $\frac{1}{pasos}$ en cada iteración hasta llegar a 0. Este índice es también el usado para calcular la transformación de los píxeles.

En cada iteración simplemente se llama a la función que calcula la transformación y se le asigna el nuevo valor a la imagen intermedia. Cuando se termina de cambiar todos los píxeles, se escribe la imagen en el disco. En caso de error se usa la función correspondiente.

- Liberar memoria


```
delete [] salida;
delete [] v_fuente;
delete [] v_destino;
```

En este caso no se escribe porque se hace durante la ejecución del bucle. Por tanto, solo nos queda eliminar la memoria dinámica usada.

Todo el código está documentado en `inc/funciones_imagen.h` y se puede consultar su documentación generándola con el script correspondiente.

Programa principal - `main`

El objetivo del programa principal es proporcionar al usuario un menú sencillo en el que elegir que operación desea realizar sobre las imágenes que ha debido pasar como parámetros.

El `main` también se encarga de comprobar que las imágenes proporcionadas por el usuario sean realmente imágenes PGM con las que trabaja el programa.

Scripts

Para facilitar el uso y añadir funcionalidad externa al proyecto se han diseñado una serie de scripts. Estos son:

- `a_gris.sh`

Este script se encarga de compilar el archivo `src/color_a_gris.cpp` y ejecutar el archivo correspondiente. Para ello existe una regla de construcción concreta en el Makefile para que sólo se encargue de ese archivo.

El programa se encarga de llamar a la función `void colorAGris(const char * nombre_ppm, const char * nombre_pgm);` que se encarga de transformar una imagen a color a una imagen en escala de grises.

Para ello usa las siguientes constantes (definidas como macros en el código)

```
const double ROJO_GRIS = 0.2989;
const double VERDE_GRIS = 0.587;
const double AZUL_GRIS = 0.114;
```

El cambio de color a gris se hace mediante la siguiente fórmula:

$$G(i, j) = 0.2989 \cdot R(i, j) + 0.587 \cdot G(i, j) + 0.114 \cdot B(i, j)$$

El script recorre todos los archivos de tipo PPM en el directorio `imagenes_entrada` y guarda su versión PGM en el directorio `imagenes_salida`.

- `animar.sh`

Dependencias:

- Paquete `imagemagick`

Este script se encarga de convertir todas las imágenes intermedias generadas por el morphing en un gif para ver mejor la transición.

El uso de este script es el siguiente

```
scripts/animar.sh <directorio-imagenes> <archivo-salida> <tiempo-en-ms-entre-frame>
```

- `gen_doc.sh`

Dependencias:

- doxygen: para generar la documentación
- graphviz: paquete para generar grafos de dependencia entre los distintos objetos y ficheros
- pdflatex: paquete para generar la documentacion en pdf

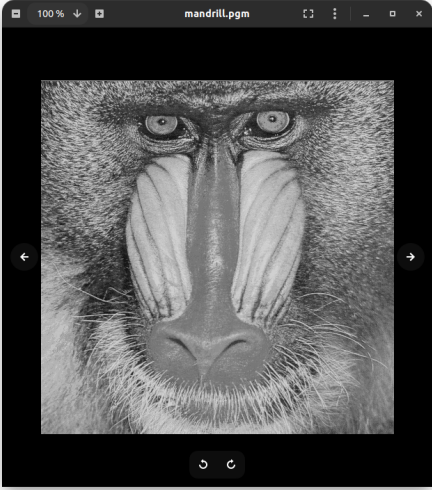
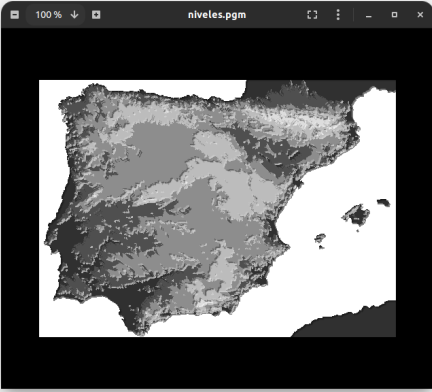

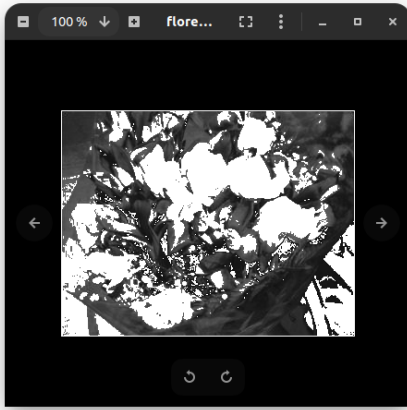
El script borra la documentación generada anteriormente en el directorio `doc/` y usa doxygen para generar la documentación en pdf y en html. Se crea un enlace simbólico al archivo `index.html` que se encuentra dentro de `doc/html`. En el caso de pdf copia el resultado en el directorio actual.

Imágenes de prueba

Primero hemos pasado todas las fotos de color a escala de grises con `scripts/a_gris.sh`.

- Pruebas umbral:




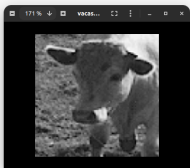


Probamos a hacer la operación de umbral en el rango 10,100 en las fotos: `mandrill.pgm`, `niveles.pgm` y `flores.pgm`

Original	Umbral
	
	
	

Podemos ver que los resultados son los esperados

- Pruebas zoom:

Probamos las imagenes: `alhambra.pgm`, `vacas.pgm` y `cameraman.pgm`

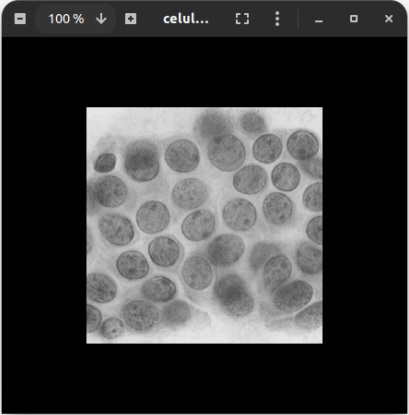
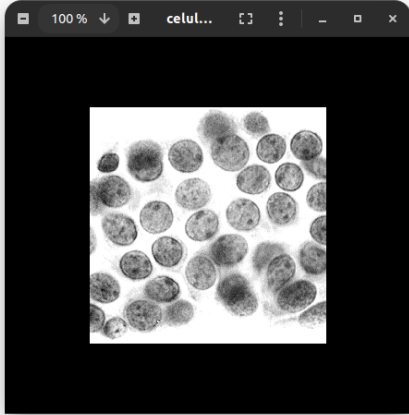
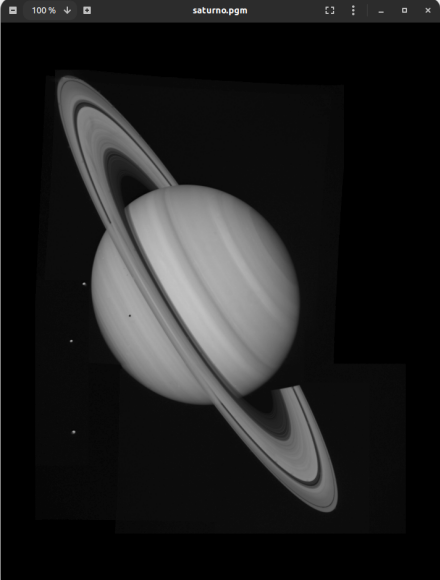
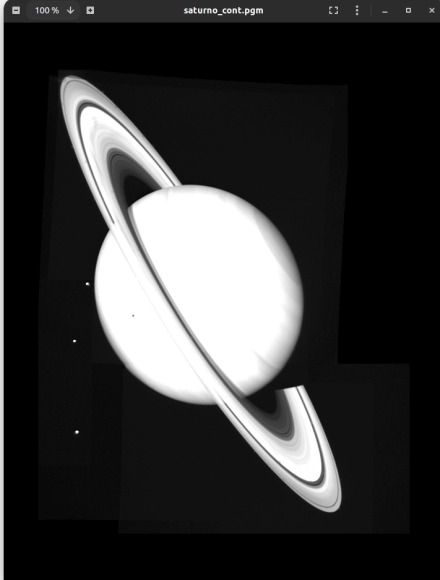
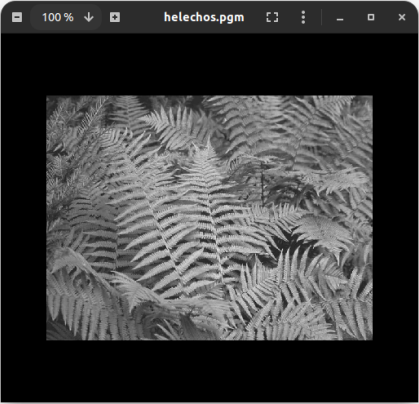
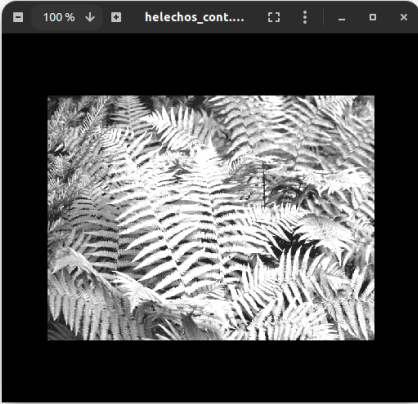
Original	Zoom	Coordenadas (argumentos)
		(220,540), (337,657)
		(81,139), (167,225)
		(32,90), (82,140)

Vemos que funciona correctamente.

- Pruebas contraste

Para probar la función de contraste elegiremos imágenes con poco contraste para comprobarlo: `celulas.pgm`, `saturno.pgm` y `helechos.pgm`.

El rango de la nueva foto es 0,400 en los tres ejemplos.

Original	Contraste 0,400
	
	
	

Vemos que las nuevas imágenes tienen más contraste.

- Pruebas morphing

Para el morphing necesitamos dos imágenes que sean del mismo tamaño. Usaremos `helechos.pgm` con `bosque.pgm` y `celulas.pgm` con `vacas.pgm`. Ambos con 100 pasos intermedios. Como no puedo ni poner 100 imagenes ni un gif en un pdf, puede comprobar el resultado en el directorio `resultados/res_morphing_helecho` y `resultados/res_morphing_celula`. Puede probar el script `scripts/animar.sh` para generar una animación gif.

Todas las imágenes generadas por el programa para estos ejemplos están en la carpeta `resultados`. En las tablas aparecen capturas de pantalla debido a que el programa que estoy utilizando para este pdf no admite imágenes del formato `pgm`.