Advanced Maths Documentation

Thomas Lower

January 2024

1 Task A

Figure 1: The starting image

Figure 2: The starting image set to greyscale

The image in figure 1 is a coloured image of our course's Christmas dinner. It is pre-cropped to fit the aspect ratio of the program is scaled within the program to fit the window.

In preparation for the task in section 2, the program will convert the image to greyscale before saving it back to disk. The algorithm used for this is a basic mean average of the 3 colour channels. The result of which is figure 2.

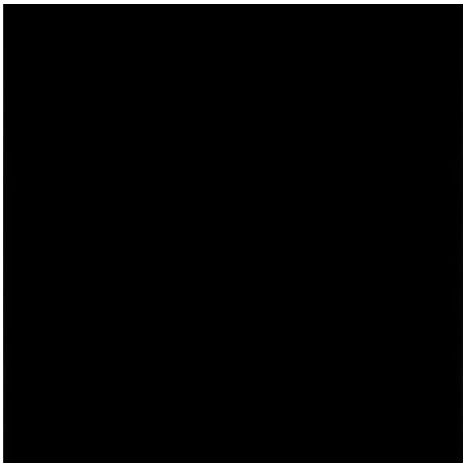
.BMP files are used throughout this project due to their lack of compression and clarity of pixel data that would allow maximum control over the given pixel

data.

All images are scaled to be 2000 x 2000.

2 Task B

Figure 3: The result of the first Fourier transform



When Fourier transforming figure 2, the resultant image can be seen in 3. When Inversely Fourier transformed, this recreated figure 2 almost exactly.

The system treats this process as a series of unrelated steps as to avoid data from a previous step contaminating that of the next step (such as a Fourier transform being affected by data from a previous Fourier transform). It reads each pixel from the greyscale screen and divides it by 255 to give a normalized value. This is then stored in a 2 dimensional array (an array of rows, each row being an array of floating point complex numbers) with the normalized values from the greyscale being used as the real value of a complex number which is passed into the Fourier transform function.

The Fourier function itself then reads in the entire row, calculates the angle for the given pixel in the row, applies the exponent of the pixel as the imaginary part of the complex number and scales the entire pixel by the size of the row (see section 1 for sizes). An output array is then created in the heap (rather than the call stack) so that a pointer to the array can be returned.

Afterwards, the Inverse Fourier transform function is applied in a similar fashion but with 2 differences: The angle is inverted, and the result is not scaled. I opted to write this as a separate function to avoid confusion and make it clearer to myself which way I was Fourier transforming.

Both transforms make sure to return the entire complex number and then only use the absolute value of this for rendering to the screen and image while saving the rest for future calculations to avoid truncation of data.

3 Task C

Figure 4: The result of the 2D Fourier transform

To swap the rows and columns of the Fourier transform result, the algorithm simply looks through the entire 2D array polynomially and swaps `result[x][y] with result[y][x]`. The result of this is then inversely Fourier transformed to simulate a 2D Fourier transform. The resultant image figure 4 features the same image but with a grey undertone as well as white artefacts in the top corners and black artefacts in the bottom corners.

4 Task D

Figure 5: The box convolution kernal

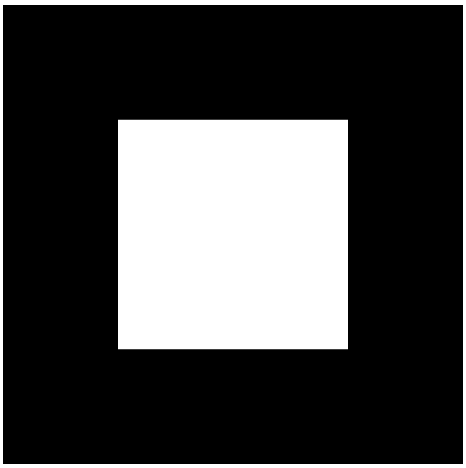


Figure 6: The result of Fourier transforming the box convolution kernal

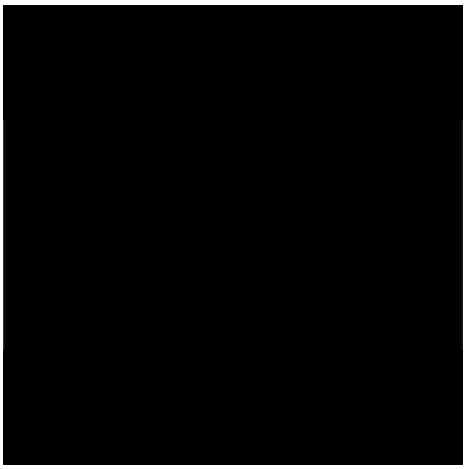


Figure 7: The result of multiplying the kernal with the fourier transformed image

Figure 8: The result of inversely transforming figure 7

This algorithm applies a convolution box to the image. The box is sized to take up `50%` of the screen and be centred in the middle, as visible in figure 5. The box is then fourier transformed to produce figure 6 and multiplied with the intermediate of the 2D Fourier transform in section 3. The result of this multiplication is 7 which when inversely Fourier transformed created 8.

Multiplication is used rather than direct convolution due to the idea that "Convolution in time domain is multiplication in the frequency domain" (Stephenson, 2024).

5 Task E

Figure 9: The box convolution kernal

Figure 10: The result of Fourier transforming the box convolution kernal

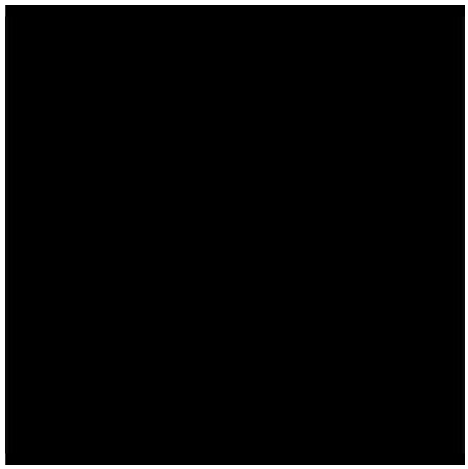


Figure 11: The result of multiplying the kernal with the fourier transformed image

Figure 12: The result of inversely transforming figure 11

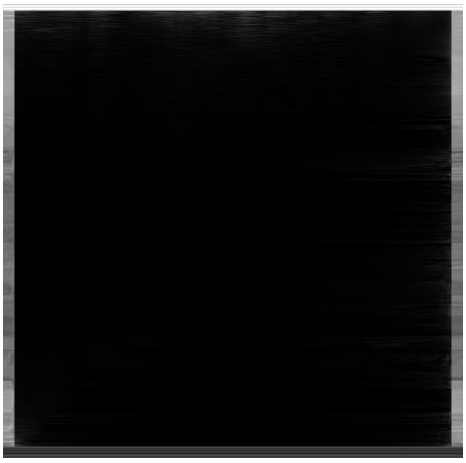
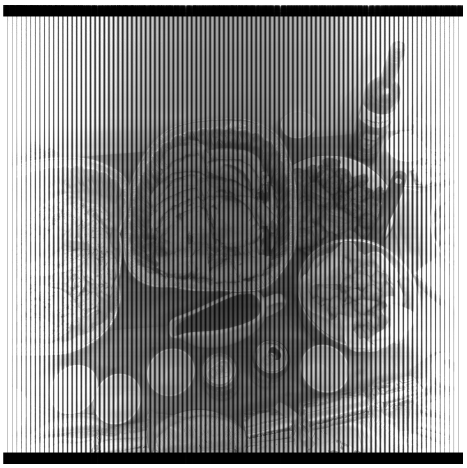


Figure 13: The result of inversely transforming figure 12



The convolution kernal I have created is a border image as shown in figure 9. I chose this because I noticed that the transformed image saw the pixels appearing to group on the edge, therefore I multiplied these to give them more focus. The result is visible in 13

6 Final Code

```
#include <SDL2/SDL.h>
#include <SDL2/SDL_image.h>
#include <complex.h>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>


//utility function to save images
void saveImage(SDL_Renderer *ren, char *filename, int width, int height)
{
  SDL_Surface *sectionSurface = SDL_CreateRGBSurface(0, width, height, 32, 0xff000000, 0x00ff0000, 0x0000ff00, 0x000000ff);
  SDL_RenderReadPixels(ren, NULL, SDL_PIXELFORMAT_RGBA8888, sectionSurface->pixels, sectionSurface->pitch);
  SDL_SaveBMP(sectionSurface, filename);
  SDL_FreeSurface(sectionSurface);
  printf("Image saved to %s\n", filename);
  fflush(stdout);
}

//basic color struct with alpha component
typedef struct color
{
  Uint8 r;
  Uint8 g;
  Uint8 b;
  Uint8 a;
} color;

//utility function to read the colour of a given pixel from the screen
color readPixel(SDL_Renderer *ren, int x, int y)
{
  SDL_Rect rect = {x, y, 1, 1};
  SDL_Surface *sectionSurface = SDL_CreateRGBSurface(0, 1, 1, 32, 0xff000000, 0x00ff0000, 0x0000ff00, 0x000000ff);
  SDL_RenderReadPixels(ren, &rect, SDL_PIXELFORMAT_RGBA8888, sectionSurface->pixels, sectionSurface->pitch);
  color c = {0, 0, 0, 0};
  Uint32 pixel = ((Uint32 *)sectionSurface->pixels)[0];
  SDL_GetRGBA(pixel, sectionSurface->format, &c.r, &c.g, &c.b, &c.a);
  SDL_FreeSurface(sectionSurface);
  return c;
}

//discrete fourier transform for a given row of pixels
float _Complex *ft(float _Complex *input, int y, int width)
{
  //data is kept separate to avoid contamination
  float _Complex data[width];
  float _Complex result[width];

```

```
  for (int x = 0; x < width; x++)
  {
    data[x] = input[x];
  }

  for (int x = 0; x < width; x++)
  {
    result[x] = 0.0 + 0.0 * I;
    for (int xx = 0; xx < width; xx++)
    {
      float angle = 2.0 * M_PI * xx / (float)width;
      result[x] += data[xx] * cexp(I * angle * x);
    }
    result[x] /= width;
  }

  //output is placed onto the heap to avoid stack overflow and
    allow access to the array outside of the function without
    having to return all values.
  float _Complex *result2 = (float _Complex *)malloc(sizeof(float
    _Complex) * width);

  for (int x = 0; x < width; x++)
  {
    result2[x] = result[x];
  }

  return result2;
}

//inverse discrete fourier transform for a given row of pixels
float _Complex *Ift(float _Complex *input, int y, int width)
{
  float _Complex data[width];
  float _Complex result[width];

  for (int x = 0; x < width; x++)
  {
    data[x] = input[x];
  }

  for (int x = 0; x < width; x++)
  {
    result[x] = 0.0 + 0.0 * I;
    for (int xx = 0; xx < width; xx++)
    {
      float angle = 2.0 * M_PI * xx / (float)width;
      result[x] += data[xx] * cexp(-I * angle * x);
    }
  }

  float _Complex *result2 = (float _Complex *)malloc(sizeof(float
    _Complex) * width);

  for (int x = 0; x < width; x++)
  {
    result2[x] = result[x];
```

```
  }

  return result2;
}

int main(int argc, char *argv[])
{
  if (SDL_Init(SDL_INIT_EVERYTHING) != 0)
  {
    printf("error initializing SDL: %s\n", SDL_GetError());
    return 1;
  }

  const int SCREEN_WIDTH = 2000;
  const int SCREEN_HEIGHT = 2000;

  SDL_Window *win = SDL_CreateWindow("GAME",
                    SDL_WINDOWPOS_CENTERED,
                    SDL_WINDOWPOS_CENTERED,
                    SCREEN_WIDTH, SCREEN_HEIGHT, 0);
  //create and clear screen
  SDL_Renderer *ren = SDL_CreateRenderer(win, -1, 0);
  SDL_RenderClear(ren);
  SDL_SetRenderDrawColor(ren, 255, 255, 255, 255);




  //load image
  SDL_Surface *image = IMG_Load("start_image.bmp");
  SDL_Texture *texture = SDL_CreateTextureFromSurface(ren, image);
  SDL_RenderCopy(ren, texture, NULL, NULL);
  SDL_RenderPresent(ren);




  //convert image to grayscale
  for (int y = 0; y < SCREEN_HEIGHT; y++)
  {
    for (int x = 0; x < SCREEN_WIDTH; x++)
    {
      color c = readPixel(ren, x, y);
      float temp = (c.r + c.g + c.b) / 3;
      SDL_SetRenderDrawColor(ren, temp, temp, temp, 255);
      SDL_RenderDrawPoint(ren, x, y);
    }
  }
  SDL_RenderPresent(ren);
  saveImage(ren, "./grayscale_image.bmp", SCREEN_WIDTH,
    SCREEN_HEIGHT);




  //fourier transform
  float _Complex *result[SCREEN_HEIGHT];
```

```
158   for (int y = 0; y < SCREEN_HEIGHT; y++)
159   {
160     //generate array
161     float _Complex *data = (float _Complex *)malloc(sizeof(float
        _Complex) * SCREEN_WIDTH);
162     for (int x = 0; x < SCREEN_WIDTH; x++)
163     {
164       data[x] = readPixel(ren, x, y).r / 255.0;
165     }
166     //perform fourier transform on generated array
167     result[y] = ft(data, y, SCREEN_WIDTH);
168   }
169   //display result and save to file
170   for (int y = 0; y < SCREEN_HEIGHT; y++)
171   {
172     for (int x = 0; x < SCREEN_WIDTH; x++)
173     {
174       float temp = cabs(result[y][x]);
175       if (temp > 1)
176       {
177         temp = 1;
178       }
179       if (temp < 0)
180       {
181         temp = 0;
182       }
183       SDL_SetRenderDrawColor(ren, temp * 255, temp * 255, temp *
      255, 255);
184       SDL_RenderDrawPoint(ren, x, y);
185     }
186   }
187   SDL_RenderPresent(ren);
188   saveImage(ren, "./fourier_image.bmp", SCREEN_WIDTH, SCREEN_HEIGHT
      );
189
190
191   //clear screen
192   SDL_SetRenderDrawColor(ren, 0, 0, 0, 255);
193   SDL_RenderClear(ren);
194
195
196   //flip the fourier transform between columns and rows
197   for (int y = 0; y < SCREEN_HEIGHT; y++)
198   {
199     for (int x = 0; x < SCREEN_WIDTH; x++)
200     {
201       result[y][x] = result[x][y];
202     }
203   }
204
205
206   //perform inverse fourier transform
207   float _Complex *result2[SCREEN_HEIGHT];
208   for (int y = 0; y < SCREEN_HEIGHT; y++)
209   {
210     result2[y] = Ift(result[y], y, SCREEN_HEIGHT);
211   }
```

```
212   for (int y = 0; y < SCREEN_HEIGHT; y++)
213   {
214     for (int x = 0; x < SCREEN_WIDTH; x++)
215     {
216       float temp = cabs(result2[y][x]);
217       if (temp > 1)
218       {
219         temp = 1;
220       }
221       if (temp < 0)
222       {
223         temp = 0;
224       }
225       SDL_SetRenderDrawColor(ren, temp * 255, temp * 255, temp *
      255, 255);
226       SDL_RenderDrawPoint(ren, x, y);
227     }
228   }
229   SDL_RenderPresent(ren);
230   saveImage(ren, "./double_fourier_image.bmp", SCREEN_WIDTH,
      SCREEN_HEIGHT);
231
232
233   //clear screen
234   SDL_SetRenderDrawColor(ren, 0, 0, 0, 255);
235   SDL_RenderClear(ren);
236
237
238   //generate kernals for convolution
239   SDL_SetRenderDrawColor(ren, 255, 255, 255, 255);
240
241   bool const usingBoxKernal = false;
242
243   if (usingBoxKernal)
244   {
245     SDL_Rect rect = {SCREEN_WIDTH / 4, SCREEN_HEIGHT / 4,
      SCREEN_WIDTH / 2, SCREEN_HEIGHT / 2};
246     SDL_RenderFillRect(ren, &rect);
247   }
248   else
249   {
250     SDL_Rect rect = {0, 0, 50, SCREEN_HEIGHT};
251     SDL_RenderFillRect(ren, &rect);
252     SDL_Rect rect2 = {SCREEN_WIDTH - 50, 0, 50, SCREEN_HEIGHT};
253     SDL_RenderFillRect(ren, &rect2);
254     SDL_Rect rect3 = {0, 0, SCREEN_WIDTH, 50};
255     SDL_RenderFillRect(ren, &rect3);
256     SDL_Rect rect4 = {0, SCREEN_HEIGHT - 50, SCREEN_WIDTH, 50};
257     SDL_RenderFillRect(ren, &rect4);
258   }
259   SDL_RenderPresent(ren);
260   saveImage(ren, "./rect_image.bmp", SCREEN_WIDTH, SCREEN_HEIGHT);
261
262
263   //perform fourier transform on kernal
264   float _Complex *box_result[SCREEN_HEIGHT];
265
```

```
266   for (int y = 0; y < SCREEN_HEIGHT; y++)
267   {
268     float _Complex *data = (float _Complex*)malloc(sizeof(float
      _Complex) * SCREEN_WIDTH);
269     for (int x = 0; x < SCREEN_WIDTH; x++)
270     {
271       data[x] = readPixel(ren, x, y).r / 255.0;
272     }
273     box_result[y] = ft(data, y, SCREEN_WIDTH);
274   }
275   for (int y = 0; y < SCREEN_HEIGHT; y++)
276   {
277     for (int x = 0; x < SCREEN_WIDTH; x++)
278     {
279       float temp = cabs(box_result[y][x]);
280       if (temp > 1)
281       {
282         temp = 1;
283       }
284       if (temp < 0)
285       {
286         temp = 0;
287       }
288       SDL_SetRenderDrawColor(ren, temp * 255, temp * 255, temp *
      255, 255);
289       SDL_RenderDrawPoint(ren, x, y);
290     }
291   }
292   SDL_RenderPresent(ren);
293   saveImage(ren, "./fourier_rect_image.bmp", SCREEN_WIDTH,
      SCREEN_HEIGHT);
294
295
296   //multiply the two fourier transforms to simulate convolution
297   float _Complex *convolution_result[SCREEN_HEIGHT];
298   for (int y = 0; y < SCREEN_HEIGHT; y++) {
299     convolution_result[y] = (float _Complex*)malloc(sizeof(float
      _Complex) * SCREEN_WIDTH);
300     for (int x = 0; x < SCREEN_WIDTH; x++) {
301       convolution_result[y][x] = box_result[y][x] * result2[y][x];
302     }
303   }
304   SDL_RenderPresent(ren);
305   saveImage(ren, "./convolution_image.bmp", SCREEN_WIDTH,
      SCREEN_HEIGHT);
306
307
308
309   //perform inverse fourier transform on the convolution result
310   float _Complex *Ift_result[SCREEN_HEIGHT];
311   for (int y = 0; y < SCREEN_HEIGHT; y++)
312   {
313     Ift_result[y] = Ift(convolution_result[y], y, SCREEN_WIDTH);
314   }
315   for (int y = 0; y < SCREEN_HEIGHT; y++)
316   {
317     for (int x = 0; x < SCREEN_WIDTH; x++)
```

```
318     {
319       float temp = cabs(Ift_result[y][x]);
320       if (temp > 1)
321       {
322         temp = 1;
323       }
324       if (temp < 0)
325       {
326         temp = 0;
327       }
328       SDL_SetRenderDrawColor(ren, temp * 255, temp * 255, temp *
      255, 255);
329       SDL_RenderDrawPoint(ren, x, y);
330     }
331   }
332   SDL_RenderPresent(ren);
333   saveImage(ren, "./inverse_fourier_image.bmp", SCREEN_WIDTH,
      SCREEN_HEIGHT);
334
335
336
337   //perform inverse fourier transform on the inverse fourier
      transform result to return to true image
338   float _Complex *Ift_result2[SCREEN_HEIGHT];
339   SDL_SetRenderDrawColor(ren, 0, 0, 0, 255);
340   SDL_RenderClear(ren);
341
342   for (int y = 0; y < SCREEN_HEIGHT; y++)
343   {
344     Ift_result2[y] = Ift(Ift_result[y], y, SCREEN_WIDTH);
345   }
346   for (int y = 0; y < SCREEN_HEIGHT; y++)
347   {
348     for (int x = 0; x < SCREEN_WIDTH; x++)
349     {
350       float temp = cabs(Ift_result2[y][x]);
351       if (temp > 1)
352       {
353         temp = 1;
354       }
355       if (temp < 0)
356       {
357         temp = 0;
358       }
359       SDL_SetRenderDrawColor(ren, temp * 255, temp * 255, temp *
      255, 255);
360       SDL_RenderDrawPoint(ren, x, y);
361     }
362   }
363   SDL_RenderPresent(ren);
364   saveImage(ren, "./double_inverse_fourier_image.bmp", SCREEN_WIDTH
      , SCREEN_HEIGHT);
365
366
367   //wait for user to close window
368   while (1)
369   {
```

```
    SDL_Event e;
    if (SDL_PollEvent(&e))
    {
      if (e.type == SDL_QUIT)
      {
        break;
      }
    }
  }

  saveImage(ren, "./end_image.bmp", SCREEN_WIDTH, SCREEN_HEIGHT);

  // Free allocated memory
  for (int y = 0; y < SCREEN_HEIGHT; y++) {
    free(result[y]);
    free(result2[y]);
    free(box_result[y]);
    free(convolution_result[y]);
    free(Ift_result[y]);
    free(Ift_result2[y]);
  }
  SDL_DestroyTexture(texture);
  SDL_FreeSurface(image);
  SDL_DestroyRenderer(ren);
  SDL_DestroyWindow(win);

  IMG_Quit();
  SDL_Quit();

  return 0;
}
```

References

Stephenson, I. (2024). Advanced mathematics for computer graphics, lecture 10.