Macros

```
.macro displaymsg msg
 #store
        addi x10, x10, 0
        addi sp, sp, -16
        sw a4, 0(sp)
        sw t3, 4(sp)
        sw t5, 8(sp)
        sw t6, 12(sp)
#program
        li t6, 0x10000100 #address of UART for nexys a7
        li t5, 0x10000114 #address of its status register
        print_characters_\@:
        lb a4, 0(\msg)
        beqz a4, exit
        sb a4, 0(t6)
        cont_\@:
        lw t3, 0(t5)                # Load UART status register
        andi t3, t3, 0x20     # Check if it can print
        beqz t3, cont_\@
        addi \msg, \msg, 1
        j print_characters_\@
#loading
        lw a4, 0(sp)
        lw t3, 4(sp)
        lw t5, 8(sp)
        lw t6, 12(sp)
        addi sp, sp, 16
.endm
```

```
.macro displaynum num, buffer
# This macro divides the number by 10 and stores the remainder in a buffer,
# effectively storing the number in reverse order which can then be converted to ASCII later
#storing previous data in stack to be restored after program is finished
	addi x10, x10, 0
	addi sp, sp, -28
	sw t2, 0(sp)
	sw t4, 4(sp)
	sw t1, 8(sp)
	sw t6, 12(sp)
	sw t5, 16(sp)
	sw a3, 20(sp)
	sw a2, 24(sp)
#program
	li t2, 10
	addi \buffer, \buffer, 35 # number can be 35 digits long
	mv t4, \buffer
	conversion_\@:
	remu t1, \num, t2 #Remainder after dividing by 10
	addi t1, t1, '0' #adding ASCII value of 0, to convert said number to correct ASCII code
	addi \buffer, \buffer, -1 #storing digit at the top of stack
	sb t1, 0(\buffer) #storing the remainder
	divu \num, \num, t2 #perform integer division by 10, to move onto next number
	bnez \num, conversion_\@ #if it is 0 we have reached last digit
#Adds space between numbers
	addi t1, x0, 32  #ASCII code for space at the very top of stack for good presentation
	addi \buffer, \buffer, -1
	sb t1, 0(\buffer)
#Sends characters to display same procedure displaymsg
	li t6, 0x10000100 # Address of UART for nexys a7
	li t5, 0x10000114 # Address of its status register
	print_\@:
	lb a3, (\buffer)
	sb a3, 0(t6)
	can_print_\@:
	lw a2, 0(t5)      # Load UART status register
	andi a2, a2, 0x20 # Check if it can print
	beqz a2, can_print_\@ # If not, wait
	addi \buffer, \buffer, 1 # Move buffer pointer to next digit
	bne \buffer, t4, print_\@
#loading back values stored in registers as program is finished executing
	lw a2, 24(sp)
	lw a3, 20(sp)
	lw t5, 16(sp)
	lw t6, 20(sp)
	lw t1, 16(sp)
	lw t4, 12(sp)
	lw t2, 0(sp)
```

```
        addi sp, sp, 28
.endm




.macro prime input, answer
#storing
        Add x10, x10, x0
        Add sp, sp, -16
        Sw t1, 0(sp)
        Sw t2, 4(sp)
        Sw t3, 8(sp)
        Sw t4, 12(sp)
        li t1, 2            # setting it to 2
        li t2, 3            # starting number for loop
        bltz \input, not_prime_\@ # check if negative
        beq \input, t1, set_prime_\@ # check if it is 2; if so jump to set_prime to set \answer
to 1
        andi t3, \input, 1   # check if it is odd or even by checking the least significant bit
        beq t3, x0, not_prime_\@ # if even, it's not prime
Loop_\@: # start a loop from 3
        rem t4, \input, t2   # check if the number divided by the counter is remainder 0
        beqz t4, not_prime_\@   # jump to not_prime if REM = 0 not prime
        mul t4, t2, t2    # get counter squared
        bgt t4, \input, set_prime_\@ # check if the counter squared is above the limit
        addi t2, t2, 2    # increment counter
        j Loop_\@
set_prime_\@:           # set prime to 1
        li \answer, 1
        j end_prime_\@
not_prime_\@:           # set not prime to 0
        li \answer, 0
end_prime_\@:
 #loading
     lw t1 0(sp)
     lw t2, 4(sp)
     lw t3, 8(sp)
     lw t3, 12(sp)
    Add sp, sp, 16
.endm
```

Resize jpegs - needs libjpeg-dev library, to install use *sudo apt-get libjpeg-dev*

//DOES NOT WORK ON REPL but does work after command
//sudo apt-get install libjpeg-dev in linux. this opens and edits the colour data of a jpeg. to resize it. I made the resize algorithm and mapped the colour data using linear interpolation. but the specifics to decompress and write to a jpeg was from chatgpt. I edited it to fit my program.

```
#include <stdio.h>
#include <jpeglib.h>
#include <stdlib.h>

typedef struct {
    unsigned char r, g, b;
} Pixel;

Pixel** read_jpeg_file(const char* filename, int* width, int* height) {
    //surfs through header to get information and reach where pixel data is stored
    struct jpeg_decompress_struct cinfo;
    struct jpeg_error_mgr jerr;
    FILE *infile = fopen(filename, "rb");
    if (!infile) {
        printf("ERROR wrong/invalid file path %s\n", filename);
        exit(0);
        return 0;
    }
    cinfo.err = jpeg_std_error(&jerr);
    jpeg_create_decompress(&cinfo);
    jpeg_stdio_src(&cinfo, infile);
    jpeg_read_header(&cinfo, TRUE);
    jpeg_start_decompress(&cinfo);
    *width = cinfo.output_width;
    *height = cinfo.output_height;
    //initialises 2d array of pixel structure to store colour information
    Pixel** image = (Pixel**)malloc(cinfo.output_height * sizeof(Pixel*));
    for (int i = 0; i < cinfo.output_height; i++) {
        image[i] = (Pixel*)malloc(cinfo.output_width * sizeof(Pixel));
    }

    JSAMPARRAY buffer = (*cinfo.mem->alloc_sarray)
        ((j_common_ptr) &cinfo, JPOOL_IMAGE, cinfo.output_width *
cinfo.output_components, 1);
    //loads colour data
    while (cinfo.output_scanline < cinfo.output_height) {
        jpeg_read_scanlines(&cinfo, buffer, 1);
        for (int x = 0; x < cinfo.output_width; x++) {
            image[cinfo.output_scanline-1][x].r = buffer[0][x * cinfo.output_components];
```

```
            image[cinfo.output_scanline-1][x].g = buffer[0][x * cinfo.output_components + 1];
            image[cinfo.output_scanline-1][x].b = buffer[0][x * cinfo.output_components + 2];
        }
    }

    jpeg_finish_decompress(&cinfo);
    jpeg_destroy_decompress(&cinfo);
    fclose(infile);

    return image;
}
void write_file(const char* filename, Pixel** image, int width, int height){
    struct jpeg_compress_struct cinfo;
    struct jpeg_error_mgr jerr;
    //writes data to header about image
    FILE *output = fopen(filename, "wb");
    cinfo.err = jpeg_std_error(&jerr);
    jpeg_create_compress(&cinfo);
    jpeg_stdio_dest(&cinfo, output);
    cinfo.image_width = width;
    cinfo.image_height = height;
    cinfo.input_components = 3;
    cinfo.in_color_space = JCS_RGB;

    jpeg_set_defaults(&cinfo);
    jpeg_start_compress(&cinfo, TRUE);

    JSAMPROW row_pointer;
    int row_stride = width * 3; // RGB channels
    //writes colour
    while (cinfo.next_scanline < cinfo.image_height) {
        row_pointer = (JSAMPROW) &image[cinfo.next_scanline][0];
        jpeg_write_scanlines(&cinfo, &row_pointer, 1);
    }

    jpeg_finish_compress(&cinfo);
    fclose(output);
    jpeg_destroy_compress(&cinfo);

}

    int main() {
    char filename[100]; // Allocate memory for filename
    int x, y, ox, oy, owidth, oheight, nwidth, nheight;
    float scaleh, scalew;

    // Getting file path of the image
    printf("Enter the file path of the image: ");
```

```
    scanf("%s", filename);

    Pixel** image = read_jpeg_file(filename, &owidth, &oheight);

    // Getting dimensions of the new image
    printf("Enter the dimensions of the new image (height width): ");
    scanf("%d %d", &nheight , &nwidth);

    scaleh = (float) oheight / nheight;
    scalew = (float) owidth / nwidth;

    printf("%f %f\n", scaleh, scalew);

    // Allocate memory for new_image
    Pixel** new_image = (Pixel**) malloc(nheight * sizeof(Pixel*));
    for (int i = 0; i < nheight; i++) {
        new_image[i] = (Pixel*) malloc(nwidth * sizeof(Pixel));
    }

    float oxf, oyf; // Used to check if we need to interpolate
    f       for (int y = 0; y < nheight; y++) {
            for (int x = 0; x < nwidth; x++) {
            oxf = scalew * x;
            oyf = scaleh * y;
            int ox = (int)oxf;
            int oy = (int)oyf;
            float fx = oxf - ox;
            float fy = oyf - oy;

            if (ox >= owidth - 1) ox = owidth - 2;
            if (oy >= oheight - 1) oy = oheight - 2;

            Pixel p1 = image[oy][ox];
            Pixel p2 = image[oy][ox + 1];
            Pixel p3 = image[oy + 1][ox];
            Pixel p4 = image[oy + 1][ox + 1];

            new_image[y][x].r = (unsigned char)((1 - fx) * (1 - fy) * p1.r + fx * (1 - fy) * p2.r + (1 -
fx) * fy * p3.r + fx * fy * p4.r);
            new_image[y][x].g = (unsigned char)((1 - fx) * (1 - fy) * p1.g + fx * (1 - fy) * p2.g + (1 -
fx) * fy * p3.g + fx * fy * p4.g);
            new_image[y][x].b = (unsigned char)((1 - fx) * (1 - fy) * p1.b + fx * (1 - fy) * p2.b + (1 -
fx) * fy * p3.b + fx * fy * p4.b);
            }
            }

    // Write the scaled image to a new file
    write_file("scaled.jpg", new_image, nwidth, nheight);
```

```
    // Free memory allocated for both image and new_image
    for (int i = 0; i < oheight; i++) {
        free(image[i]);
    }
    free(image);

    for (int i = 0; i < nheight; i++) {
        free(new_image[i]);
    }
    free(new_image);

    return 0;
}
```

Calculates coefficients of polynomial using only coordinates - repl finding coefficients(1) - C

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// Function to allocate memory for a matrix
double** allocateMatrix(int degree) {
    double** matrix = (double**)malloc(degree * sizeof(double*));
    for (int i = 0; i < degree; i++) {
        matrix[i] = (double*)malloc(degree * sizeof(double));
    }
    return matrix;
}
void gaussElimination(double** matrix, double** inverse, int degree) {
    for (int i = 0; i < degree; i++) { //to cycle through every row
        double pivot = matrix[i][i]; //is the value in matrix, with the position where a 1 will be in identity matrix
        if (pivot == 0) { //check if it is 0, as will use it to divide later
            for (int k = i + 1; k < degree; k++) { //if it is 0, we trying to swap this row with the one below
                if (matrix[k][i] != 0) {
                    for (int j = 0; j < degree; j++) {
                        double temp = matrix[i][j]; //swapping one element at a time in the row
                        matrix[i][j] = matrix[k][j];
                        matrix[k][j] = temp;

                        temp = inverse[i][j]; //copying the same action in indentity matrix
                        inverse[i][j] = inverse[k][j];
                        inverse[k][j] = temp;
```

```
                }
                pivot = matrix[i][i]; //updating pivot
                break;
            }
        }
        if (pivot == 0) { // if the pivot is still 0, then an inverse can't be calculated
            printf("Matrix is singular and cannot be inverted\n");
            return;
        }
    }
    for (int j = 0; j < degree; j++) { //dividing every element in the row by the pivot, to make
value at pivot position 1
        matrix[i][j] /= pivot;
        inverse[i][j] /= pivot;
    }
    for (int k = 0; k < degree; k++) { // we are subtracting pivot row from other rows, in an
effort to make values in the same column as pivot = 0
        if (k != i) {
            double factor = matrix[k][i]; //setting value that we want to become 0
            for (int j = 0; j < degree; j++) {
                matrix[k][j] -= factor * matrix[i][j]; //multiplying and subtracting to ensure the
factor value turns to 0, but the others are still non-zero. factor becomes 0, because the value
in pivot position is 1 from previous division which is selected in first iteration of j loop.
                inverse[k][j] -= factor * inverse[i][j];
            }
        }
    }
  }
}
int main() {
  int degree;
  double x;
  printf("Enter the degree of the polynomial: ");
  scanf("%d", &degree);
  degree += 1;
  //intialising matrices, as 2d arrays
  double** matrix = allocateMatrix(degree);
  double** inverse = allocateMatrix(degree);
  //intialising vectors as 1d arrays
  double* y = (double*)malloc(degree * sizeof(double));
  double* result =  (double*)malloc(degree * sizeof(double));
  //gathering inputs
  for (int i = 0; i < degree; i++) {
   printf("Enter the coordinates:\n");
    scanf("%lf %lf", &x, &y[i]);
    for (int j = 0; j < degree; j++) {
        matrix[i][(degree - 1) - j] = pow(x, j); //setting up matrix with X^n to X^0.
        if (i == j) { //setting up inverse matrix
```

```
                inverse[i][j] = 1.0;
            } else {
                inverse[i][j] = 0.0;
            }
        }
    }
    gaussElimination(matrix, inverse, degree);
    //perform multiplication with vector by calculating the dot product between each row in the
inverse matrix and the y coordinate vector
    for (int i = 0; i < degree; i++){
      for (int j = 0; j < degree; j++){
        result[i] += inverse[i][j] * y[j];
      }
    }
  printf("The coefficients are: \n");
    for (int i =0; i < degree; i++){
      printf("%.2lf ", result[i]);
    }
    return 0;
}
```