AERE 361 Lab 10

Jordan Reese

March 2018

A lot of work

Multiplying matrices in C or any low level language is very different than if you were to do it with a high level language. I used matlab to double check my math. My C code was about 300 lines of code and in matlab it took me...4 lines. Now I understand that someone got paid a lot of money to make it so I could only use 4 lines. This lab I had to write all of that underlying code myself, and I didn't even really scratch the surface because I am still using csvparse. It is still nice to develop a greater understanding of my underlying high level programs.

CSVParse

We became familiar with this program in lab 8. There are a few different tools that CSVParse has to offer, but like in lab 8 I used code from csvinfo.c. This code parse through a csv file and checks to see how many rows and columns there is. With a few manipulations to the callback function it has then you can get it to store. Below are the original callbacks and the 3rd and 4th functions are ones I created. I talked about the functions is the lab 8 report so I won't waste your time.

```
void *data) {
  ((struct counts *)data)->fields++;
}
void cb2 (int c, void *data) {
  ((struct counts *)data)->rows++;
}
void storfield (void *s, size_t len, void *data){
  struct matrix *mw = ((struct matrix *)data);
  double v = atof(s);
  mw->matf[mw->r][mw->c] = v;
  mw->c++;
}
void countrow (int c, void *data){
  struct matrix *mw = ((struct matrix *)data);
```

```
  mw->r++;
  mw->c = 0;
}
```

LOTS O' FUNCTIONS

This code has a lot of functions in it. My main is very clean because of that, but there was still much coding that needed to be done. After I declared my structs at the top. I started to think what needed to be done before I could do any math. I need to allocate, initialize, store, and print my two imported CSV matrices. The only complex function was the read/write function. That one took some thinking to get all the way through. I even called function INSIDE ANOTHER FUNCTION. That is like func-ception. Anyway I digress. I will list that code below and talk a little about it.

```
*file, struct matrix *mw){
  FILE *fp;
  struct csv_parser p;
  char buf[1024];
  size_t bytes_read;
  unsigned char options = 0;
  struct counts co = {0, 0};
  if (csv_init(&p, options) != 0) {
    fprintf(stderr, "Failed to initialize csv parser\n");
    return 1;
  }

  csv_set_space_func(&p, is_space);
  csv_set_term_func(&p, is_term);
  fp = fopen(file, "rb");
  if (!fp) {
    fprintf(stderr, "Failed to open %s: %s\n", file, strerror(errno));
    return 2;
  }
  while ((bytes_read=fread(buf, 1, 1024, fp)) > 0) {
    if (csv_parse(&p, buf, bytes_read, cb1, cb2, &co) != bytes_read) {
      fprintf(stderr, "Error while parsing file: %s\n",
          csv_strerror(csv_error(&p)));
      return 7;
    }
  }
  csv_fini(&p, cb1, cb2, &co);
  if (ferror(fp)) {
    fprintf(stderr, "Error while reading file %s\n", file);
    fclose(fp);
    return 3;
  }
```

```
printf("%s: %lu fields, %lu rows\n", file, co.fields, co.rows);
co.columns = co.fields / co.rows;
allocate_matrix(co.rows, co.columns, mw);
rewind(fp);
csv_free(&p);
options = CSV_APPEND_NULL;
if (csv_init(&p, options) != 0) {
  fprintf(stderr, "Failed to initialize csv parser\n");
  return 4;
}
//printf("about to do the while\n");
//INVAR this is the csv code, but we are chunking through the file by
    1024 bytes and parsing it.
mw->r = 0;
mw->c = 0;
while ((bytes_read=fread(buf, 1, 1024, fp)) > 0) {
  if (csv_parse(&p, buf, bytes_read, storfield, countrow, mw) !=
      bytes_read) {
    fprintf(stderr, "Error while parsing file: %s\n",
        csv_strerror(csv_error(&p)));
    return 6;
  }
}
//printf("after the while\n");
csv_fini(&p, storfield, countrow, mw);
mw->r = co.rows;
mw->c = co.columns;

if (ferror(fp)) {
  fprintf(stderr, "Error while reading file %s\n", file);
  fclose(fp);
  return 5;
}


csv_free(&p);
//printf("right before return\n");
return 0;
```

As you can see quite a lengthy fucntion compared to what we are used to in this class. Most of this fucntion is made up of csvinfo.c. Luckily for me csvinfo.c does a fair amount of error checking for me. The first pass through the imported CSV is to know how many rows it has so I can properly allocate the memory needed to store that information. The second pass (the one where I use my callbacks) is the one that does all of the storing into the aforementioned allocated matrix.

Actually doing "Math"

Doing math with large arrays in C is nothing like you would do on paper or even a higher level programming language. Because C doesn't know how to just multiply matrices together, you have to do it manually. I actually used Wikipedia for this part. They have a nice graphic that make visualizing the process very easy. I knew I needed a nest for loop 3 deep. One for the rows of my first matrix, one for the columns of my second matrix, and the last one for the fields in those rows and columns. I needed to do a dot product, so I needed some way to count those values. The loops have one interesting thing I did though. In the loops you can see a loc variable. This is short for local, and all it does is saves me from indexing my multiplied matrix is a weird way. I found it works a little better for the cross product, and I don't have to worry about matrix 3 having weird stuff in it from memory. Below is the nest looping I used and a little error checking.

```
if (mat1->c != mat2->r){
   printf("Your matrix dimension do not match, therefore cannot do
       multiplication\n");
   return 1;
 }

 allocate_matrix(mat1->r, mat2->c, mat3);
 int r = 0;
 int c = 0;
 int k = 0;
 double loc = 0;
 for(r = 0; r < mat3->r; r++){
   for(c = 0; c < mat3->c; c++){
     loc = 0;
     for(k = 0; k < mat1->c; k++){
  loc += mat1->matf[r][k] * mat2->matf[k][c];
     }
     mat3->matf[r][c] = loc;
   }
 }
 return 0;
```

Main

Main is soooo pretty. Because I used function for most of the things I did, main came out nicely. I do a little bit of error checking below my function to ensure they return a value stating they worked as intended.

Performance

I think I made this code fairly well performing. When I used valgrind to find the cache misses. It came out with .4 percent misses. To me that seems like a fairly respectable number. I didn't text super huge matrices, but they weren't just tiny little ones either. The matrices in my repository now are only integers, but I did text the program with decimals as well. Because of the way I used nested for loops, the code will have a large big O complexity.