Project 01: Image Interpolation

Assigned: 2019-02-13 12:00:00

Due: 2019-02-22 23:59:00

Points: 100 pts is full credit (25 pts extra available)

Instructions: All of this assignment will be submitted via Canvas. Handwritten solutions will receive 0 credit.

Deliverables: For this assignment, you are to submit all code through code.vt.edu. You are also expected to upload a PDF file to canvas with your writeup.

Collaboration: This assignment is to be completed by yourself, however, you make seek assistance from your classmates. In your submission you must indicate from whom you received assistance.

Honor Code: By submitting this assignment, you acknowledge that you have adhered to the Virginia Tech Honor Code and attest to the following:

> I have neither given nor received unauthorized assistance on this assignment. The work I am presenting is ultimately my own.

References

- Writing pseudo-code in Latex: https://en.wikibooks.org/wiki/LaTeX/Algorithms
- Making tables in Latex: https://en.wikibooks.org/wiki/LaTeX/Tables
- Basics of 1D interpolation: https://ocw.mit.edu/courses/mechanical-engineering/2-086-numerical-computatic nutshells-guis/MIT2_086F14_Interpolation.pdf
- Basics of 2D linear interpolation: http://www.aip.de/groups/soe/local/numres/bookcpdf/c3-6. pdf
- 2D arrays (row vs column major): https://en.wikipedia.org/wiki/Row-_and_column-major_order (This may help with address calculation.)
- PGM image format: http://netpbm.sourceforge.net/doc/pgm.html

Code Environment

You are **strongly** encouraged to explore the code environemt. Look at the files, code, and structure. See what is provided. Ask questions when they arise!

1. **Task:** Acquire the code environment. The project code environment is in the public assignments repository (your upstream repository). Fetch this and
2. **Task:** Space has been made in `homework/hw01/` for you to add your homework tex source and any images you generated for that. Add and commit your Homework 01 files now.
3. **Task:** Install the required dependencies for this project.

```
> sudo apt install libpng-dev

> sudo apt install liblapacke-dev

> sudo apt install imagemagick
```

Project Structure

This section describes the `projects/proj01/` directory.

- `projects/proj01/code/` is where the provided code is and where your code will go.
- `projects/proj01/data/` is where any project specific data that we provide may be found. Unless you are told specifically to do so, do not add to this directory. Under no circumstances should you add large image files to your repository.
- `projects/proj01/report/` is where any project report source (tex files, etc) is to be stored.
- `projects/proj01/results/` is where any result data (e.g., timings) generated by your project should be stored and committed.
- `projects/proj01/spec/` is where this document may be found.

The `projects/proj01/code` Directory

This directory is primarily where you will work. Inside it, you will find the following:

- `makefile` is the makefile for this project. See the next section for details.
- `apps/` contains the source code for individual programs.
  - Each `.c` file in this directory is a program.
  - Any `.c` file in this directory must contain a `main()` function.
  - The makefile will automatically detect these programs and build them if the `make all` rule is run.
  - The project library, `libimageproc` must be compiled first before any program in this directory will link correct.
  - The contents of many of these programs have been commented out so that they will compile with incomplete solutions. You will need to uncomment the code to use it for tests.
  - These programs are provided as examples. They are insufficient for you to completely test your code. Grading will be based on a different set of programs.
- `extra/` contains some additional code for testing some of the required libraries. This may be useful for debugging your system.
- `include/` contains the header files for the code in the library. Each header corresponds to a file in `src/`, with minor exceptions.

- `scripts/` contains any scripts that we thought might be useful.
- `src/` contains the source files for the library. Each source file has a corresponding header file in `include`. You will be modifying files here.

The makefile

The makefile that we provide is more advanced than the ones that you have seen so far. The following is a short summary of its output.

- Any programs (`apps/`) are compiled to the `bin/` directory. To compile a program, run

  ```
  > make program_name
  ```

  . You can run them from here.

  ```
  > ./bin/program
  ```

- The image processing code is built into a static library, `libimageproc`, which is in the `lib/` directory after compilation. To compile the library, run

  ```
  > make libimageproc
  ```

- Any intermediate build artifacts (object files) are placed in the `build/` directory.
- To compile the library and all programs, simply run `make`.
- The command `make clean` removes these three directories.
- Your project will be partially graded by having some of our codes link against your implementation if `libimageproc`. Therefore, it is critical that you do not change the function signatures.
- This makefile supports a debug mode, which turns off optimizations and turns on the `-g` option. This is achieved by running

  ```
  > make DEBUG=1
  ```

- Do not modify the makefile under any circumstance.

Provided Modules

- `image_types.h`: Provides the core image data types.
- `linalg.h/c`: Provides a wrapper for LAPACKE for solving linear systems.
- `math_utilities.h/c`: Provides some handy mathematical routines that are not in the C standard library.
- `pam_io.h/c`: Provides routines for loading and saving PGM/PPM images, using the core image data types defined above.
- `png_utilties.h/c`: Provides code to allow you to output png images directly.
- `python_utilities.h/c`: Provides utilities for writing images as numpy arrays (text).
- Any other header file likely contains the function signatures for functions that you will be expected to implement.

Data Types

You will primarily interact with two image data types that we have provided. Both are single-channel or monochrome image data types.

- `MImage8` is a Monochrome Image format which stores data in 8-bit integers (`unsigned char` type).
- `MImageF` is a Monochrome Image format which stores data in Floating-point format (`float` type).
- Each type has a data field, which is a pointer to user-allocated memory. This field is for a 1D array, which will will treat as a 2D array using *strided array access*.
- Each type has integer height and width fields for the size of the x (width) and y (height) dimensions.

Requirements

Remember: commit early, commit often, commit after minor changes, commit after major changes. Push your code to `code.vt.edu` frequently.

1. **Allocators/deallocators for floating-point images** In `include/image_memory.h`, you will find function signatures for nullifiers, allocators, initializers, and deallocators for the `MImage8` and `MImageF` types. In `src/image_memory.c`, you will see implementations of these routines for `MImage8`.

   (a) **Task: (5 pts)** In `src/image_memory.c`, implement the following routines, using the signatures **precisely** as written in the header file. Each routine should return an integer error code, 0 for success and 1 for failure.

   - `nullify_MImageF()`: safely nullify an instance of `MImageF`
   - `allocate_MImageF()`: allocate an instance of `MImageF` sized `width` by `height`
   - `initialize_MImageF()`: safely initialize an instance of `MImageF` to have all 0.0 data
   - `deallocate_MImageF()`: safely deallocate an instance of `MImageF`

2. **Convert 8-bit images to floating-point data (and vice versa)** Images are often stored using 8 bit integer values in each channel, thus the dynamic range is 0-255. We will perform a number of transformations that are best handled with more resolution that 256 values, so we will use floating-point numbers to represent our working images. The function signatures for this task are in `include/image_convert.h`. Your implementation must be in `src/image_convert.c`.

   (a) **Task: (5 pts)** Write functions to convert images between `MImage8` and `MImageF` formats. You should assume that the output images are already allocated. Each routine should return an integer error code, 0 for success and 1 for failure. You must account for failure modes.

   - `convert_MImage8_to_MImageF()`: convert data with range 0-255 to have `float` values 0.0-1.0. E.g., 0 maps to 0.0, 255 maps to 1.0, 1 maps to 1/255, etc.
   - `convert_MImageF_to_MImage8()`: convert data with range 0.0-1.0 to have `unsigned char` values 0-255. E.g., 0.0 maps to 0, 1.0 maps to 255, etc. You will have floating point numbers that do not map directly to an integer, so use the `floor` function to account for this.

3. **1D Interpolation** As a stepping-stone to full 2D interpolation, you will implement the various 1D interpolation schemes that you derived on Homework 01. We will use the same image structs for the 1D problems as we will for the 2D problems. For these tasks, we will assume that the `height` of the image is 1.

   These routines should use `MImageF` types for both input and output. The function signatures for this task are in `include/image_interp1d.h`. Your implementations must be in `src/image_interp1d.c`. You should assume that the images are already allocated. Each routine should return an integer error code, 0 for success and 1 for failure. You must account for failure modes, where present.

   (a) **Task: (1 pts)** `compute_downsample_size()`: In `include/image_utilities.h` there is a signature for a routine to compute the number of pixels in one dimension of the downsampled image. Implement this routine in `src/image_utilities.c`.

   (b) **Task: (4 pts)** `interp1d_downsample()`: Implement 1D downsampling following the algorithm developed in Homework 01.

   (c) **Task: (5 pts)** `interp1d_nearest()`: Implement 1D interpolation following the nearest left-neighbor algorithm developed in Homework 01.

   (d) **Task: (10 pts)** `interp1d_linear()`: Implement 1D interpolation following the piecewise linear algorithm developed in Homework 01.

   (e) **Task: (15 pts)** `interp1d_cubic()`: Implement 1D interpolation following the piecewise cubic algorithm developed in Homework 01. Note, this requires you to solve a linear system. Use the wrapper provided in `linalg.h`.

4. **2D Interpolation** (45 pts) You will implement the various 2D interpolation schemes that you derived on Homework 01. These routines should use `MImageF` types for both input and output.

   The function signatures for this task are in `include/image_interp2d.h`. Your implementations must be in `src/image_interp2d.c`. You should assume that the images are already allocated. Each routine should return an integer error code, 0 for success and 1 for failure. You must account for failure modes, where present.

   (a) **Task: (5 pts)** `interp2d_downsample()`: Implement 2D downsampling following the algorithm developed in Homework 01.

   (b) **Task: (15 pts)** `interp2d_nearest()`: Implement 2D interpolation following the nearest left-neighbor algorithm. This algorithm is equivalent to applying the 1D algorithm in each dimension. Do not create any intermediate arrays.

   (c) **Task: (15 pts)** `interp2d_linear()`: Implement 2D interpolation following the second piecewise linear algorithm developed in Homework 01. Do not create any intermediate arrays.

   (d) **Task: (10 pts)** Write a new application, `apps/cameraman.c` which loads the cameraman image found in the root `test_data` directory and compares the results of using your piecewise constant and piecewise linear interpolation algorithms to **triple** the size of the image. (If you do the extra credit, include the 2D piecewise cubic results too!) Follow the example codes to see how to load `PGM` images, manipulate them, and save them. You can use the `convert` program to convert the output image to `PNG` format so that you can include it in your project writeup.

   ```
   > convert image.pgm image.png
   ```

   Part of your grade on this task comes from the writeup.

5. **Extra Credit** The following tasks are optional. Any and all can be completed for extra credit.

   (a) **Task: (5 pts)** Implement an image cropping routine. so far, we have assumed that all input and output images map to the box with upper left corner at $(a, c) = (0.0, 0.0)$ and lower right corner at $(b, d) = (1.0, 1.0)$. To crop an image, while changing resolution (or not) we can have the output image map all of its pixels to a subset of that box with upper left corner $(\hat{a}, \hat{c})$, $\hat{a} \geq 0.0$, $\hat{c} \geq 0.0$ and lower right corner $(\hat{b}, \hat{d})$, $\hat{b} \leq 1.0$, $\hat{d} \leq 1.0$.

   `crop_linear()`: Implement a routine which uses 2D piecewise linear interpolation, along with a sub-box defined above, to crop and resize an image. The signature for this routine is in `include/image_crop.h` and the implementation must be in `src/image_crop.h`.

   Write a program `apps/crop.c` demonstrating this routine. See `apps/test_crop.c` as an example.

   (b) **Task: (15 pts)** `interp2d_cubic()`: In `include/image_interp2d.h` there is a signature for a 2D piecewise cubic interpolation. Implement this routine as developed in the extra credit of Homework 01. Note, this requires you to solve a linear system. Use the wrapper provided in `linalg.h`. Include your results using this code in your programs.

   (c) **Task: (5 pts)** Write a new application, `apps/interesting.c` which loads the an interesting image and compares the results of using your piecewise constant and piecewise linear interpolation algorithms to increase the size of the image. Try a few different resolutions or change the aspect ratio. (If you do the extra credit, include the 2D piecewise cubic results too!) You can use the `convert` program to convert the output images to `PNG` format so that you can include it in your project writeup.

   You are free to choose your own image. You can use `convert` to change nearly any image format to `PGM`.

   ```
   > convert image.png/jpg/tif image.pgm
   ```

   Note, the image loader we have provided is not completely robust, so this may cause some problems with custom images. Any issue is likely due to the structure of the image header that `convert` generates. This can be fixed.

Questions

Answer the following questions using latex and place your tex source in the `proj01/report` directory of the assignment repository.

- Include the output of `cameraman.c` and describe any differences you see in the 2D interpolation schemes that you implemented. Which do you like best?
- Try running your program using the raytrace image instead of the cameraman. Try to time the execution time for the different interpolation schemes using this larger image. How does piecewise linear or cubic compare to nearest neighbor?

EC Include the output of `crop.c` and describe any differences you see in the 2D interpolation schemes that you implemented.

EC Include the output of `interesting.c` and describe any differences you see in the 2D interpolation schemes that you implemented.