```
    1: #include "float16.hpp"
    2:
    3: float f16ToFloat(f16 y)
    4: {
    5:     union { float f; uint32_t i; } v;
    6:     v.i = f16ToFloatI(y);
    7:     return v.f;
    8: }
    9:
   10: uint32_t static f16ToFloatI(f16 y)
   11: {
   12:     int s = (y >> 15) & 0x00000001;                                 // sign
   13:     int e = (y >> 10) & 0x0000001f;                                 // exponent
   14:     int f =  y        & 0x000003ff;                                 // fraction
   15:
   16:     // need to handle 7c00 INF and fc00 -INF?
   17:     if (e == 0) {
   18:         // need to handle +-0 case f==0 or f=0x8000?
   19:         if (f == 0)                                                 // Plus or minus zero
   20:             return s << 31;
   21:         else {                                                      // Denormalized number -
- renormalize it
   22:             while (!(f & 0x00000400)) {
   23:                 f <<= 1;
   24:                 e -=  1;
   25:             }
   26:             e += 1;
   27:             f &= ~0x00000400;
   28:         }
   29:     } else if (e == 31) {
   30:         if (f == 0)                                                  // Inf
   31:             return (s << 31) | 0x7f800000;
   32:         else                                                         // NaN
   33:             return (s << 31) | 0x7f800000 | (f << 13);
   34:     }
   35:
   36:     e = e + (127 - 15);
   37:     f = f << 13;
   38:
   39:     return ((s << 31) | (e << 23) | f);
   40: }
   41:
   42: f16 floatToF16(float i)
   43: {
   44:     union { float f; uint32_t i; } v;
   45:     v.f = i;
   46:     return floatToF16I(v.i);
   47: }
   48:
   49: f16 static floatToF16I(uint32_t i)
   50: {
   51:     int s =  (i >> 16) & 0x00008000;                    // sign
   52:     int e = ((i >> 23) & 0x000000ff) - (127 - 15);      // exponent
   53:     int f =   i        & 0x007fffff;                    // fraction
   54:
   55:     // need to handle NaNs and Inf?
   56:     if (e <= 0) {
   57:         if (e < -10) {
   58:             if (s)                                       // handle -0.0
   59:                return 0x8000;
   60:             else
   61:                return 0;
   62:         }
```

```
63:         f = (f | 0x00800000) >> (1 - e);
64:         return s | (f >> 13);
65:     } else if (e == 0xff - (127 - 15)) {
66:         if (f == 0)                                             // Inf
67:             return s | 0x7c00;
68:         else {                                                  // NAN
69:             f >>= 13;
70:             return s | 0x7c00 | f | (f == 0);
71:         }
72:     } else {
73:         if (e > 30)                                             // Overflow
74:             return s | 0x7c00;
75:         return s | (e << 10) | (f >> 13);
76:     }
77: }
78:
79: double f16ToDouble(f16 i)
80: {
81:     return (double)f16ToFloat(i);
82: }
83:
84: f16 doubleToF16(double i)
85: {
86:     return floatToF16((float) i);
87: }
```

```
#pragma once
// From https://blog.fpmurphy.com/2008/12/half-precision-floating-point-format_14.html
/*
**  This program is free software; you can redistribute it and/or modify it under
**  the terms of the GNU Lesser General Public License, as published by the Free
**  Software Foundation; either version 2 of the License, or (at your option) any
**  later version.
**
**   IEEE 758-2008 Half-precision Floating Point Format
**   --------------------------------------------------
**
**   | Field    | Last | First | Note
**   |----------|------|-------|----------
**   | Sign     | 15   | 15    |
**   | Exponent | 14   | 10    | Bias = 15
**   | Fraction | 9    | 0     |
*/

#include <stdio.h>
#include <inttypes.h>

typedef uint16_t f16;

/* ----- prototypes ------ */
float f16ToFloat(f16);
f16 floatToF16(float);
double f16ToDouble(f16);
f16 doubleToF16(double);
static uint32_t f16ToFloatI(f16);
static f16 floatToF16I(uint32_t);
```

```
    1: #include <fstream>
    2: #include <getopt.h>
    3: #include <iostream>
    4: #include <string>
    5: #include <unistd.h>
    6: #include <string.h>
    7: #include <regex>
    8: #include <ctime>
    9:
   10: #include "convert.hpp"
   11: #include "program_options.hpp"
   12: #include "svd.hpp"
   13:
   14: void random_pgm(string filename, int xsize, int ysize, int maxlimit, int minlimit)
   15: {
   16:     srand(time(NULL));
   17:     ofstream newpgm;
   18:     int num;
   19:     filename = filename + ".pgm";
   20:     newpgm.open(filename);
   21:     pgma_write_header(newpgm,filename,xsize,ysize,255);
   22:     for(int i=0;i<xsize;i++)
   23:     {
   24:         for(int z=0;z<ysize;z++)
   25:         {
   26:             num = rand()%(maxlimit-minlimit)+1+minlimit;
   27:             newpgm << num << " ";
   28:         }
   29:         newpgm << "\n";
   30:     }
   31: }
   32:
   33:
   34: int main(int argc, char **argv)
   35: {
   36:   try {
   37:     ProgramOptions::parse(argc, argv);
   38:
   39:     switch(ProgramOptions::selected_algorithm())
   40:     {
   41:     case ProgramOptions::AlgorithmSelection::TO_BINARY:
   42:     {
   43:         //std::string input_file = ProgramOptions::text_pgm_filepath();
   44:         //std::string output_file = ProgramOptions::binary_pgm_filepath();
   45:         //std::cout << input_file << " " << output_file;
   46:         bool error;
   47:         char file_in_name[80];
   48:         char file_out_name[80];
   49:         int length = strlen(argv[2]);
   50:         strcpy(file_in_name, argv[2]);
   51:         strncpy (file_out_name,file_in_name,length-4);
   52:         file_out_name[length-4] = '\0';
   53:         std::cout << file_out_name << std::endl;
   54:         strcat (file_out_name,"_b.pgm");
   55:         std::cout << file_out_name;
   56:         error = pgma_to_pgmb(file_in_name,file_out_name);
   57:         return 0;
   58:     }
   59:     case ProgramOptions::AlgorithmSelection::FROM_BINARY:
   60:     {
   61:         //std::string input_file = ProgramOptions::binary_pgm_filepath();
   62:         //std::string output_file = ProgramOptions::text_pgm_filepath();
   63:         bool error;
```

```
   64:         char file_in_name[80];
   65:         char file_out_name[80];
   66:         int length = strlen(argv[2]);
   67:         strcpy(file_in_name, argv[2]);
   68:         strncpy (file_out_name,file_in_name,length-4);
   69:         file_out_name[length-4] = '\0';
   70:         strcat (file_out_name,"_copy.pgm");
   71:         //cout << file_out_name;
   72:         error = pgmb_to_pgma ( file_in_name, file_out_name );
   73:         return 0;
   74:     }
   75:     case ProgramOptions::AlgorithmSelection::COMPRESSED_SVD:
   76:     {
   77:         std::string input_file = ProgramOptions::svd_matrices_filepath();
   78:         std::string header_file = ProgramOptions::pgm_header_filepath();
   79:         std::string output_file = ProgramOptions::binary_pgm_filepath();
   80:         int rank = ProgramOptions::approximation_rank();
   81:
   82:         output_file += "_" + std::to_string(rank);
   83:
   84:         std::ifstream header(header_file);
   85:         std::ifstream pgm(input_file);
   86:         SVD::decomp decomposition = SVD::pgmSvdToHalfStream(header, pgm, rank);
   87:         header.close();
   88:         pgm.close();
   89:
   90:         SVD::writePgmAsSvd(output_file, decomposition);
   91:
   92:         std::cout << "Wrote compressed image to \"" << output_file << "\"" << std::endl
;
   93:         return 0;
   94:     }
   95:     case ProgramOptions::AlgorithmSelection::FROM_COMPRESSED_SVD:
   96:     {
   97:         std::string input_file = ProgramOptions::binary_pgm_filepath();
   98:         std::string output_file = ProgramOptions::text_pgm_filepath();
   99:
  100:         auto [pgm, rank] = SVD::svdToPGMString(input_file);
  101:
  102:         output_file = std::regex_replace(output_file, std::regex("_k"), "_" + std::to_s
tring(rank));
  103:         std::ofstream output(output_file);
  104:         output << pgm;
  105:
  106:         output.close();
  107:         std::cout << "Wrote decompressed image to \"" << output_file << "\"" << std::en
dl;
  108:         return 0;
  109:     }
  110:     case ProgramOptions::AlgorithmSelection::RANDOM_IMAGE:
  111:     {
  112:         std::string output_file = ProgramOptions::text_pgm_filepath();
  113:         bool check = false;
  114:         string filename;
  115:         int maxn = 255,minn = 0;
  116:         //std::cout << argc;
  117:         if(argc == 7 && atoi(argv[5]) >= 0 && atoi(argv[5]) <= 255 && atoi(argv[6]) >=
0 && atoi(argv[6]) <= 255 && atoi(argv[6]) < atoi(argv[5]))
  118:         {
  119:             maxn = atoi(argv[5]);
  120:             minn = atoi(argv[6]);
  121:             //std::cout << "limit";
  122:         }
```

```
  123:         int xsize = atoi(argv[3]);
  124:         int ysize = atoi(argv[4]);
  125:         random_pgm(output_file,xsize,ysize,maxn,minn);
  126:         std::cout << "pass";
  127:         return 0;
  128:     }
  129:     case ProgramOptions::AlgorithmSelection::QUICK:
  130:     {
  131:         std::string input_file = ProgramOptions::svd_matrices_filepath();
  132:         std::string header_file = ProgramOptions::pgm_header_filepath();
  133:         std::string output_file = ProgramOptions::binary_pgm_filepath();
  134:         std::ifstream header(header_file);
  135:         std::ifstream pgm(input_file);
  136:
  137:         SVD::writeAllDecomps(header, pgm, output_file);
  138:         header.close();
  139:         pgm.close();
  140:         return 0;
  141:     }
  142:     default:
  143:         throw std::runtime_error("Unknown algorithm type.");
  144:     }
  145:
  146:   }
  147:   catch (const std::runtime_error &e) { // if the user mis-entered anything, just print
 the help.
  148:     std::cerr << "ERROR: " << e.what() << std::endl;
  149:     ProgramOptions::print_help();
  150:   }
  151: }
```

```
    1: //The convert part are based on this website : https://people.sc.fsu.edu/~jburkardt/cpp_src/pgma_to_pgmb/pgma_to_pgmb.html
    2: #include "program_options.hpp"
    3:
    4: #include <unistd.h>
    5: #include <climits>
    6: #include <getopt.h>
    7: #include <iostream>
    8: #include <sstream>
    9: #include <experimental/filesystem>
   10: #include <regex>
   11:
   12: namespace fs = std::experimental::filesystem;
   13:
   14: ProgramOptions* ProgramOptions::s_instance = nullptr;
   15:
   16: ProgramOptions* ProgramOptions::instance()
   17: {
   18:     if(s_instance == nullptr)
   19:     {
   20:         s_instance = new ProgramOptions();
   21:     }
   22:
   23:     return s_instance;
   24: }
   25:
   26: ProgramOptions::AlgorithmSelection ProgramOptions::selected_algorithm()
   27: {
   28:     return instance()->m_algorithm;
   29: }
   30:
   31: const std::string& ProgramOptions::program_name()
   32: {
   33:     return instance()->m_program_name;
   34: }
   35:
   36: const std::string& ProgramOptions::text_pgm_filepath()
   37: {
   38:     return instance()->m_text_pgm_filepath;
   39: }
   40:
   41: const std::string& ProgramOptions::binary_pgm_filepath()
   42: {
   43:     return instance()->m_binary_pgm_filepath;
   44: }
   45:
   46: const std::string& ProgramOptions::pgm_header_filepath()
   47: {
   48:     return instance()->m_pgm_header_filepath;
   49: }
   50:
   51: const std::string& ProgramOptions::svd_matrices_filepath()
   52: {
   53:     return instance()->m_svd_matrices_filepath;
   54: }
   55:
   56: int ProgramOptions::approximation_rank()
   57: {
   58:     return instance()->m_approximation_rank;
   59: }
   60:
   61: void ProgramOptions::clear()
   62: {
```

```
   63:     if(s_instance)
   64:     {
   65:         delete s_instance;
   66:     }
   67:     s_instance = new ProgramOptions();
   68: }
   69:
   70: void ProgramOptions::print_help()
   71: {
   72:     //TODO UPDATE THIS!
   73:     std::cout << std::endl;
   74:     std::cout << " Usage: " << instance()->m_program_name << " [1|2|3|4] [pgm-file|pgm-
header-file|SVD-compressed-file]" << std::endl;
   75:     std::cout << "                                    [SVD-matrices-file] [approximation
-rank]" << std::endl;
   76:     std::cout << std::endl;
   77:     std::cout << " Advance Algorithms Project #2: Image Compression Using SVD and Dimen
sionality" << std::endl;
   78:     std::cout << "                                  Reduction Using PCA" << std::endl;
   79:     std::cout << std::endl;
   80:     std::cout << " One and only one of the following parameters must be selected." << s
td::endl;
   81:     std::cout << "     1                  Convert ASCII PGM file to a binary PGM file.
" << std::endl;
   82:     std::cout << "                             Requires: pgm-file (ASCII)" << std::end
l;
   83:     std::cout << "                             Outputs:  pgm-file (binary)" << std::en
dl;
   84:     std::cout << std::endl;
   85:     std::cout << "     2                  Convert a binary PGM file to an ASCII PGM fi
le." << std::endl;
   86:     std::cout << "                             Requires: pgm-file (binary)" << std::en
dl;
   87:     std::cout << "                             Outputs:  pgm-file (ASCII)" << std::end
l;
   88:     std::cout << std::endl;
   89:     std::cout << "     3                  Store a compressed image file using SVD appr
oximation" << std::endl;
   90:     std::cout << "                             Requires: pgm-header-file (ASCII)," <<
std::endl;
   91:     std::cout << "                                       SVD-matrices-file," << std::e
ndl;
   92:     std::cout << "                                       approximation-rank" << std::e
ndl;
   93:     std::cout << "                             Outputs:  SVD-compressed-file" << std::
endl;
   94:     std::cout << std::endl;
   95:     std::cout << "     4                  Revert an SVD-compressed image to a binary p
gm-file" << std::endl;
   96:     std::cout << "                             Requires: SVD-compressed-file" << std::
endl;
   97:     std::cout << "                             Outputs:  pgm-header-file (ASCII)," <<
std::endl;
   98:     std::cout << "                                       SVD-matrices-file," << std::e
ndl;
   99:     std::cout << "                                       pgm-file (binary)" << std::en
dl;
  100:     std::cout << std::endl;
  101:     std::cout << "     5                  Create a randomly generated PGM image to the
 given file" << std::endl;
  102:     std::cout << "                             Requires: pgm-file" << std::endl;
  103:     std::cout << "                             Outputs:  pgm-file (ASCII)," << std::en
dl;
```

```
  104:     std::cout << std::endl;
  105:     std::cout << std::endl;
  106:     std::cout << " The following options are required as stated above:" << std::endl;
  107:     std::cout << "    pgm-file (ASCII)     Textual PGM file conforming to the" << std::
endl;
  108:     std::cout << "                           PGM P2 specification" << std::endl;
  109:     std::cout << std::endl;
  110:     std::cout << "    pgm-file (binary)    Binary PGM file conforming to the" << std::e
ndl;
  111:     std::cout << "                           PGM P2 specification" << std::endl;
  112:     std::cout << std::endl;
  113:     std::cout << "    pgm-header-file      Text file containing the PGM P2 specificatio
n header" << std::endl;
  114:     std::cout << "                           (width height max-value)" << std::endl;
  115:     std::cout << std::endl;
  116:     std::cout << "    SVD-compressed-file  File that has been approximated using SVD" <
< std::endl;
  117:     std::cout << std::endl;
  118:     std::cout << "    SVD-matrices-file    File containing U, \u03A3, and V matrices" <
< std::endl;
  119:     std::cout << std::endl;
  120:     std::cout << "    approximation-rank   An integer representing the rank of the appr
oximation" << std::endl;
  121:     std::cout << std::endl;
  122:     std::cout << std::endl;
  123:     std::cout << " AUTHORS:" << std::endl;
  124:     std::cout << "          Quansu Lu    <ql21@zips.uakron.edu>" << std::endl;
  125:     std::cout << "       Edwin Sarver    <els40@zips.uakron.edu>" << std::endl;
  126:     std::cout << "          Ying Wang    <yw73@zips.uakron.edu>" << std::endl;
  127:
  128: }
  129:
  130: void ProgramOptions::parse(int argc, char** argv)
  131: {
  132:     //TODO UPDATE THIS!
  133:     optind = 1; // reset getopt()
  134:     opterr = 0; // Don't let getopt() print errors
  135:
  136:     instance()->m_program_name = argv[0];
  137:
  138:     // Handle flag arguments
  139:     // 'getopt(argc, argv, <expected flags>)' only parses flag arguments (like -b)
  140:     // it returns an int that is one of the following:
  141:     // =>  -1 if there are no more flag args
  142:     // =>  the char of the expected flag, if found
  143:     // =>  '?' if a flag arg was found but was not in the expected flags.
  144:     // The returned value is as an int, so it will need to be converted to a char
  145:     // before it can be checked properly.
  146:     char c;
  147:     while ((c = (char)getopt(argc, argv, "h")) != -1)
  148:     {
  149:         switch (c)
  150:         {
  151:             // The only acceptable flag argument is "-h" for the help.
  152:         case 'h':
  153:             print_help();
  154:             return;
  155:         case '?':
  156:             std::string error_string = "Unknown option \"";
  157:             error_string += ((char)optopt);
  158:             error_string += "\".";
  159:             throw std::runtime_error(error_string.c_str());
  160:         }
```

```
  161:     }
  162:
  163:     int required_positional_args = 2; // this is true for 1, 2, and 4
  164:
  165:     int remaining_args = argc - optind;
  166:
  167:     if(remaining_args < required_positional_args)
  168:     {
  169:         throw std::runtime_error("Too few positional arguments.");
  170:     }
  171:
  172:     // Check the first positional argument.
  173:     try
  174:     {
  175:         // Assume the first positional argument is in reference to the
  176:         // algorithm to be used. This converts directly to the AlgorithmSelection
  177:         // enum if it is between 1 and 4, inclusive.
  178:         int selection = std::stoi(argv[optind]);
  179:         if( selection >= 1 && selection <= 6)
  180:         {
  181:             instance()->m_algorithm = (AlgorithmSelection) selection;
  182:         }
  183:         else
  184:         {
  185:             // Throw an exception if the algorithm selection number was not 1, 2, 3, or
 4.
  186:             std::string error_string = "The number used to select the algorithm must be
 between 1 and 5, inclusive. Got \"";
  187:             error_string += argv[optind];
  188:             error_string += "\".";
  189:             throw std::runtime_error(error_string);
  190:         }
  191:         optind++;
  192:         remaining_args--;
  193:     }
  194:     catch(const std::exception& e)
  195:     {
  196:         // If we caught an exception, it was because whatever was in the algorithm sele
ction spot
  197:         // couldn't be read by std::stoi as an integer. Throw an exception of our own m
aking.
  198:         std::string error_string = "Expected a number for the first argument, got \"";
  199:         error_string += argv[optind];
  200:         error_string += "\".";
  201:         throw std::runtime_error(error_string);
  202:     }
  203:
  204:     if (instance()->m_algorithm == AlgorithmSelection::COMPRESSED_SVD && remaining_args
 < 3)
  205:     {
  206:         throw std::runtime_error("Too few positional arguments to use the selected algo
rithm.");
  207:     }
  208:
  209:     // Assume that the positional arguments will be in the correct order
  210:     // Assume that the given filepath is valid. Don't check it.
  211:     for (int offset_optind = 0; optind + offset_optind < argc; offset_optind++)
  212:     {
  213:         remaining_args--;
  214:         switch (instance()->m_algorithm)
  215:         {
  216:         case AlgorithmSelection::TO_BINARY:
  217:         {
```

```
  218:             instance()->m_text_pgm_filepath = argv[optind + offset_optind];
  219:             std::string extension = fs::path(instance()->m_text_pgm_filepath).extension
();
  220:             std::string filename = fs::path(instance()->m_text_pgm_filepath).stem();
  221:             filename.append("_b.");
  222:             filename.append(extension);
  223:             instance()->m_binary_pgm_filepath = fs::path(instance()->m_text_pgm_filepat
h).replace_filename(filename);
  224:             return;
  225:         }
  226:         case AlgorithmSelection::FROM_BINARY:
  227:         {
  228:             instance()->m_binary_pgm_filepath = argv[optind + offset_optind];
  229:             std::string extension = fs::path(instance()->m_binary_pgm_filepath).extensi
on();
  230:             std::string filename = fs::path(instance()->m_binary_pgm_filepath).stem();
  231:
  232:             if(const auto pos = filename.rfind("_b") != std::string::npos)
  233:             {
  234:                 filename.erase(pos, 2);
  235:             }
  236:             filename.append("_copy.");
  237:             filename.append(extension);
  238:             instance()->m_text_pgm_filepath = fs::path(instance()->m_binary_pgm_filepat
h).replace_filename(filename);
  239:             return;
  240:         }
  241:         case AlgorithmSelection::QUICK:
  242:         case AlgorithmSelection::COMPRESSED_SVD:
  243:         {
  244:             if(offset_optind == 0)
  245:             {
  246:                 instance()->m_pgm_header_filepath = argv[optind + offset_optind];
  247:             } else if (offset_optind == 1) {
  248:                 instance()->m_svd_matrices_filepath = argv[optind + offset_optind];
  249:                 std::string filename = fs::path(instance()->m_svd_matrices_filepath).st
em();
  250:                 std::string extension = fs::path(instance()->m_svd_matrices_filepath).e
xtension();
  251:                 filename.append("_b");
  252:                 filename.append(".pgm");
  253:                 filename.append(".SVD");
  254:                 instance()->m_binary_pgm_filepath = fs::path(instance()->m_svd_matrices
_filepath).replace_filename(filename);
  255:             }
  256:             else
  257:             {
  258:                 try
  259:                 {
  260:                     instance()->m_approximation_rank = std::stoi(argv[optind + offset_o
ptind]);
  261:                 }
  262:                 catch(const std::exception& e)
  263:                 {
  264:                     std::string error_string = "Expected a number for the approximation
 rank argument, got \"";
  265:                     error_string += argv[optind];
  266:                     error_string += "\".";
  267:                     throw std::runtime_error(error_string);
  268:                 }
  269:                 return;
  270:             }
  271:
```

```
  272:             break;
  273:         }
  274:         case AlgorithmSelection::FROM_COMPRESSED_SVD:
  275:         {
  276:             instance()->m_binary_pgm_filepath = argv[optind + offset_optind];
  277:             std::string filename = fs::path(instance()->m_binary_pgm_filepath).stem();
  278:             instance()->m_text_pgm_filepath = fs::path(instance()->m_binary_pgm_filepat
h).replace_extension(); // Remove ".SVD"
  279:             std::string extension = fs::path(instance()->m_binary_pgm_filepath).extensi
on();
  280:             instance()->m_text_pgm_filepath = fs::path(instance()->m_binary_pgm_filepat
h).replace_extension().replace_extension(); // Remove ".pgm"
  281:             instance()->m_text_pgm_filepath = std::regex_replace(instance()->m_text_pgm
_filepath, std::regex("_b"), "");
  282:             instance()->m_text_pgm_filepath.append("_k");
  283:             instance()->m_text_pgm_filepath.append(".pgm");
  284:             return;
  285:         }
  286:         case AlgorithmSelection::RANDOM_IMAGE:
  287:         {
  288:             instance()->m_text_pgm_filepath = argv[optind + offset_optind];
  289:             return;
  290:         }
  291:         }
  292:     }
  293: }
  294:
  295: ProgramOptions::ProgramOptions()
  296:     : m_algorithm(ProgramOptions::AlgorithmSelection::NONE),
  297:       m_program_name(""),
  298:       m_text_pgm_filepath(""),
  299:       m_binary_pgm_filepath(""),
  300:       m_pgm_header_filepath(""),
  301:       m_svd_matrices_filepath(""),
  302:       m_approximation_rank(0)
  303: {}
```

```
    1: #include <cstdlib>
    2: #include <iostream>
    3: #include <iomanip>
    4: #include <fstream>
    5: #include <ctime>
    6: #include <cstring>
    7: #include <string.h>
    8: #include "convert.hpp"
    9:
   10: using namespace std;
   11:
   12: void i4vec_to_ucvec ( int n, int *a, unsigned char *b )
   13: {
   14:   int i;
   15:
   16:   for ( i = 0; i < n; i++ )
   17:   {
   18:     *b = ( unsigned char ) *a;
   19:     a++;
   20:     b++;
   21:   }
   22:   return;
   23: }
   24:
   25:
   26: void pgma_check_data ( int xsize, int ysize, int maxg, int *g )
   27: {
   28:   int i;
   29:   int *index;
   30:   int j;
   31:   int k;
   32:
   33:   if ( xsize <= 0 )
   34:   {
   35:     cerr<< "\n";
   36:     cerr << "PGMA_CHECK_DATA: Error!\n";
   37:     cerr << "  XSIZE <= 0.\n";
   38:     cerr << "  XSIZE = " << xsize << "\n";
   39:     exit ( 1 );
   40:   }
   41:
   42:   if ( ysize <= 0 )
   43:   {
   44:     cerr << "\n";
   45:     cerr << "PGMA_CHECK_DATA: Error!\n";
   46:     cerr << "  YSIZE <= 0.\n";
   47:     cerr << "  YSIZE = " << ysize << "\n";
   48:     exit ( 1 );
   49:   }
   50:
   51:   if ( g == NULL )
   52:   {
   53:     cerr << "\n";
   54:     cerr << "PGMA_CHECK_DATA: Error!\n";
   55:     cerr << "  Null pointer to g.\n";
   56:     exit ( 1 );
   57:   }
   58:
   59:   index = g;
   60:
   61:   for ( j = 0; j < ysize; j++ )
   62:   {
   63:     for ( i = 0; i < xsize; i++ )
```

```
    64:     {
    65:       if ( *index < 0 )
    66:       {
    67:         cerr << "\n";
    68:         cerr << "PGMA_CHECK_DATA - Fatal error!\n";
    69:         cerr << "  Negative data.\n";
    70:         cerr << "  G(" << i << "," << j << ")=" << *index << "\n";
    71:         exit ( 1 );
    72:       }
    73:       else if ( maxg < *index )
    74:       {
    75:         cerr << "\n";
    76:         cerr << "PGMA_CHECK_DATA - Fatal error!\n";
    77:         cerr << "  Data exceeds MAXG = " << maxg << "\n";
    78:         cerr << "  G(" << i << "," << j << ")=" << *index << "\n";
    79:         exit ( 1 );
    80:       }
    81:       index = index + 1;
    82:     }
    83:   }
    84:   return;
    85: }
    86:
    87:
    88: void pgma_read ( string input_name, int *xsize, int *ysize, int *maxg,
    89:   int **g )
    90:
    91: {
    92:   ifstream input;
    93:   int numbytes;
    94:
    95:   input.open ( input_name.c_str ( ) );
    96:
    97:   if ( !input )
    98:   {
    99:     cerr << "\n";
   100:     cerr << "PGMA_READ - Fatal error!\n";
   101:     cerr << "  Cannot open the input file \"" << input_name << "\".\n";
   102:     exit ( 1 );
   103:   }
   104:
   105:   pgma_read_header ( input, xsize, ysize, maxg );
   106:
   107:   numbytes = (*xsize) * (*ysize) * sizeof ( int );
   108:
   109:   *g = new int[numbytes];
   110:
   111:   pgma_read_data ( input, *xsize, *ysize, *g );
   112:
   113:   input.close ( );
   114:
   115:   return;
   116: }
   117:
   118:
   119: void pgma_read_data ( ifstream &input, int xsize, int ysize, int *g )
   120: {
   121:   int i;
   122:   int j;
   123:
   124:   for ( j = 0; j < ysize; j++ )
   125:   {
   126:     for ( i = 0; i < xsize; i++ )
```

```
  127:     {
  128:       input >> *g;
  129:       if ( input.eof ( ) )
  130:       {
  131:         exit ( 1 );
  132:       }
  133:       g = g + 1;
  134:     }
  135:   }
  136:
  137:   return;
  138: }
  139:
  140:
  141: void pgma_read_header ( ifstream &input, int *xsize, int *ysize, int *maxg )
  142: {
  143:   int count;
  144:   string line;
  145:   string rest;
  146:   int step;
  147:   int width;
  148:   string word;
  149:
  150:   step = 0;
  151:
  152:   while ( 1 )
  153:   {
  154:     getline ( input, line );
  155:
  156:     if ( input.eof ( ) )
  157:     {
  158:       cerr << "\n";
  159:       cerr << "PGMA_READ_HEADER - Fatal error!\n";
  160:       cerr << "  End of file.\n";
  161:       exit ( 1 );
  162:     }
  163:
  164:     if ( line[0] == '#' )
  165:     {
  166:       continue;
  167:     }
  168:
  169:     if ( step == 0 )
  170:     {
  171:       s_word_extract_first ( line, word, rest );
  172:
  173:       if ( s_len_trim ( word ) == 0 )
  174:       {
  175:         continue;
  176:       }
  177:       line = rest;
  178:
  179:       if ( ( word[0] != 'P' && word[0] != 'p' ) ||
  180:            word[1] != '2' )
  181:       {
  182:         cerr << "\n";
  183:         cerr << "PGMA_READ_HEADER - Fatal error.\n";
  184:         cerr << "  Bad magic number = \"" << word << "\".\n";
  185:         exit ( 1 );
  186:       }
  187:       step = 1;
  188:     }
  189:
```

```
  190:     if ( step == 1 )
  191:     {
  192:       s_word_extract_first ( line, word, rest );
  193:
  194:       if ( s_len_trim ( word ) == 0 )
  195:       {
  196:         continue;
  197:       }
  198:       *xsize = atoi ( word.c_str ( ) );
  199:       line = rest;
  200:       step = 2;
  201:     }
  202:
  203:     if ( step == 2 )
  204:     {
  205:       s_word_extract_first ( line, word, rest );
  206:
  207:       if ( s_len_trim ( word ) == 0 )
  208:       {
  209:         continue;
  210:       }
  211:       *ysize = atoi ( word.c_str ( ) );
  212:       line = rest;
  213:       step = 3;
  214:     }
  215:
  216:     if ( step == 3 )
  217:     {
  218:       s_word_extract_first ( line, word, rest );
  219:
  220:       if ( s_len_trim ( word ) == 0 )
  221:       {
  222:         continue;
  223:       }
  224:       *maxg = atoi ( word.c_str ( ) );
  225:       break;
  226:     }
  227:
  228:   }
  229:
  230:   return;
  231: }
  232:
  233:
  234: bool pgma_to_pgmb ( char *file_in_name, char *file_out_name )
  235: {
  236:   int *g;
  237:   unsigned char *g2;
  238:   bool error;
  239:   int maxg;
  240:   int xsize;
  241:   int ysize;
  242:
  243:   pgma_read ( file_in_name, &xsize, &ysize, &maxg, &g );
  244:
  245:   if ( error )
  246:   {
  247:     cout << "\n";
  248:     cout << "PGMA_TO_PGMB: Fatal error!\n";
  249:     cout << "  PGMA_READ failed.\n";
  250:     return true;
  251:   }
  252:
```

```
  253:   pgma_check_data ( xsize, ysize, maxg, g );
  254:
  255:   g2 = new unsigned char [ xsize * ysize ];
  256:   i4vec_to_ucvec ( xsize * ysize, g, g2 );
  257:   delete [] g;
  258:
  259:   error = pgmb_write ( file_out_name, xsize, ysize, g2, maxg );
  260:
  261:   delete [] g2;
  262:
  263:   if ( error )
  264:   {
  265:     cout << "\n";
  266:     cout << "PGMA_TO_PGMB: Fatal error!\n";
  267:     cout << "  PGMB_WRITE failed.\n";
  268:     return true;
  269:   }
  270:
  271:   return false;
  272: }
  273:
  274: bool pgmb_write ( string output_name, int xsize, int ysize, unsigned char *g, int maxgt
 )
  275: {
  276:   bool error;
  277:   ofstream output;
  278:   int i;
  279:   unsigned char *indexg;
  280:   int j;
  281:   unsigned char maxg;
  282:
  283:   maxg = maxgt;
  284:   indexg = g;
  285:
  286:   for ( i = 0; i < xsize; i++ )
  287:   {
  288:     for ( j = 0; j < ysize; j++ )
  289:     {
  290:       /*if ( maxg < *indexg )
  291:       {
  292:         maxg = *indexg;
  293:       }*/
  294:       indexg = indexg + 1;
  295:     }
  296:   }
  297:
  298:   output.open ( output_name.c_str ( ), ios::binary );
  299:
  300:   if ( !output )
  301:   {
  302:     cout << "\n";
  303:     cout << "PGMB_WRITE: Fatal error!\n";
  304:     cout << "  Cannot open the output file " << output_name << "\n";
  305:     return true;
  306:   }
  307:
  308:   error = pgmb_write_header ( output, xsize, ysize, maxg );
  309:
  310:   if ( error )
  311:   {
  312:     cout << "\n";
  313:     cout << "PGMB_WRITE: Fatal error!\n";
  314:     cout << "  PGMB_WRITE_HEADER failed.\n";
```

```
315:     return true;
316:   }
317:
318:   error = pgmb_write_data ( output, xsize, ysize, g );
319:
320:   if ( error )
321:   {
322:     cout << "\n";
323:     cout << "PGMB_WRITE: Fatal error!\n";
324:     cout << "  PGMB_WRITE_DATA failed.\n";
325:     return true;
326:   }
327:
328:   output.close ( );
329:
330:   return false;
331: }
332:
333:
334: bool pgmb_write_data ( ofstream &output, int xsize, int ysize, unsigned char *g )
335: {
336:   int i;
337:   unsigned char *indexg;
338:   int j;
339:
340:   indexg = g;
341:
342:   for ( j = 0; j < ysize; j++ )
343:   {
344:     for ( i = 0; i < xsize; i++ )
345:     {
346:       output << *indexg;
347:       indexg = indexg + 1;
348:     }
349:   }
350:
351:   return false;
352: }
353:
354:
355: bool pgmb_write_header ( ofstream &output, int xsize, int ysize, unsigned char maxg )
356: {
357:   output << "P5" << " "
358:          << xsize << " "
359:          << ysize << " "
360:          << ( int ) maxg << "\n";
361:
362:   return false;
363: }
364:
365:
366: void s_word_extract_first ( string s, string &s1, string &s2 )
367: {
368:   int i;
369:   int mode;
370:   int s_len;
371:
372:   s_len = s.length ( );
373:   s1 = "";
374:   s2 = "";
375:   mode = 1;
376:
377:   for ( i = 0; i < s_len; i++ )
```

```
  378:   {
  379:     if ( mode == 1 )
  380:     {
  381:       if ( s[i] != ' ' )
  382:       {
  383:          mode = 2;
  384:       }
  385:     }
  386:     else if ( mode == 2 )
  387:     {
  388:       if ( s[i] == ' ' )
  389:       {
  390:         mode = 3;
  391:       }
  392:     }
  393:     else if ( mode == 3 )
  394:     {
  395:       if ( s[i] != ' ' )
  396:       {
  397:         mode = 4;
  398:       }
  399:     }
  400:     if ( mode == 2 )
  401:     {
  402:       s1 = s1 + s[i];
  403:     }
  404:     else if ( mode == 4 )
  405:     {
  406:       s2 = s2 + s[i];
  407:     }
  408:   }
  409:
  410:   return;
  411: }
  412:
  413:
  414: void timestamp ( )
  415: {
  416: # define TIME_SIZE 40
  417:
  418:   static char time_buffer[TIME_SIZE];
  419:   const struct tm *tm;
  420:   size_t len;
  421:   time_t now;
  422:
  423:   now = time ( NULL );
  424:   tm = localtime ( &now );
  425:
  426:   len = strftime ( time_buffer, TIME_SIZE, "%d %B %Y %I:%M:%S %p", tm );
  427:
  428:   cout << time_buffer << "\n";
  429:
  430:   return;
  431: # undef TIME_SIZE
  432: }
  433:
  434:
  435: int s_len_trim ( string s )
  436: {
  437:   int n;
  438:
  439:   n = s.length ( );
  440:
```

```
  441:   while ( 0 < n )
  442:   {
  443:     if ( s[n-1] != ' ' )
  444:     {
  445:       return n;
  446:     }
  447:     n = n - 1;
  448:   }
  449:
  450:   return n;
  451: }
  452:
  453: //****************************************************************************
  454:
  455: char ch_cap ( char ch )
  456: {
  457:   if ( 97 <= ch && ch <= 122 )
  458:   {
  459:     ch = ch - 32;
  460:   }
  461:
  462:   return ch;
  463: }
  464:
  465:
  466: void pgma_write ( string output_name, int xsize, int ysize, int *g, int maxgt )
  467: {
  468:   ofstream output;
  469:   int i;
  470:   int *indexg;
  471:   int j;
  472:   int maxg;
  473:
  474:   output.open ( output_name.c_str ( ) );
  475:
  476:   if ( !output )
  477:   {
  478:     cerr << "\n";
  479:     cerr << "PGMA_WRITE - Fatal error!\n";
  480:     cerr << "  Cannot open the output file \"" << output_name << "\".\n";
  481:     exit ( 1 );
  482:   }
  483:
  484:   maxg = maxgt;
  485:   indexg = g;
  486:
  487:   for ( j = 0; j < ysize; j++ )
  488:   {
  489:     for ( i = 0; i < xsize; i++ )
  490:     {
  491:       /*if ( maxg < *indexg )
  492:       {
  493:         maxg = *indexg;
  494:       }*/
  495:       indexg = indexg + 1;
  496:
  497:     }
  498:   }
  499:
  500:   pgma_write_header ( output, output_name, xsize, ysize, maxg );
  501:
  502:   pgma_write_data ( output, xsize, ysize, g );
  503:
```

```
  504:   output.close ( );
  505:
  506:   return;
  507: }
  508:
  509: void pgma_write_data ( ofstream &output, int xsize, int ysize, int *g )
  510: {
  511:   int i;
  512:   int *indexg;
  513:   int j;
  514:   int numval;
  515:
  516:   indexg = g;
  517:   numval = 0;
  518:
  519:   for ( j = 0; j < ysize; j++ )
  520:   {
  521:     for ( i = 0; i < xsize; i++ )
  522:     {
  523:       output << *indexg;
  524:       numval = numval + 1;
  525:       indexg = indexg + 1;
  526:
  527:       if ( numval % 12 == 0 || i == xsize - 1 || numval == xsize * ysize )
  528:       {
  529:         output << "\n";
  530:       }
  531:       else
  532:       {
  533:         output << " ";
  534:       }
  535:
  536:     }
  537:   }
  538:   return;
  539: }
  540:
  541: void pgma_write_header ( ofstream &output, string output_name, int xsize,int ysize, int
 maxg )
  542: {
  543:   output << "P2\n";
  544:   output << "# " << output_name << " created by PGMA_IO::PGMA_WRITE.\n";
  545:   output << xsize << "  " << ysize << "\n";
  546:   output << maxg << "\n";
  547:
  548:   return;
  549: }
  550:
  551:
  552: bool pgmb_check_data ( int xsize, int ysize, unsigned char maxg, unsigned char *g )
  553: {
  554:   int i;
  555:   unsigned char *index;
  556:   int j;
  557:   int k;
  558:
  559:   if ( xsize <= 0 )
  560:   {
  561:     cout << "\n";
  562:     cout << "PGMB_CHECK_DATA: Error!\n";
  563:     cout << "  xsize <= 0.\n";
  564:     cout << "  xsize = " << xsize << "\n";
  565:     return true;
```

```
  566:   }
  567:
  568:   if ( ysize <= 0 )
  569:   {
  570:     cout << "\n";
  571:     cout << "PGMB_CHECK_DATA: Error!\n";
  572:     cout << "  ysize <= 0.\n";
  573:     cout << "  ysize = " << ysize << "\n";
  574:     return true;
  575:   }
  576:
  577:   if ( g == NULL )
  578:   {
  579:     cout << "\n";
  580:     cout << "PGMB_CHECK_DATA: Error!\n";
  581:     cout << "  Null pointer to g.\n";
  582:     return true;
  583:   }
  584:
  585:   index = g;
  586:
  587:   for ( j = 0; j < ysize; j++ )
  588:   {
  589:     for ( i = 0; i < xsize; i++ )
  590:     {
  591:       if ( maxg < *index )
  592:       {
  593:         cout << "\n";
  594:         cout << "PGMB_CHECK_DATA - Fatal error!\n";
  595:         cout << "  Data exceeds MAXG = " << ( int ) maxg << "\n";
  596:         cout << "  G(" << i << "," << j << ")=" << ( int ) (*index) << "\n";
  597:         return true;
  598:       }
  599:
  600:       index = index + 1;
  601:     }
  602:   }
  603:
  604:   return false;
  605: }
  606:
  607:
  608: bool pgmb_read ( string input_name, int *xsize, int *ysize, unsigned char *maxg, unsign
ed char **g )
  609: {
  610:   bool error;
  611:   ifstream input;
  612:   int numbytes;
  613:
  614:   input.open ( input_name.c_str ( ), ios::binary );
  615:
  616:   if ( !input )
  617:   {
  618:     cout << "\n";
  619:     cout << "PGMB_READ: Fatal error!\n";
  620:     cout << "  Cannot open the input file " << input_name << "\n";
  621:     return true;
  622:   }
  623:
  624:   error = pgmb_read_header ( input, xsize, ysize, maxg );
  625:
  626:   if ( error )
  627:   {
```

```
628:     cout << "\n";
629:     cout << "PGMB_READ: Fatal error!\n";
630:     cout << "  PGMB_READ_HEADER failed.\n";
631:     return true;
632:   }
633:
634:   *g = new unsigned char [ (*xsize) * (*ysize) ];
635:
636:   error = pgmb_read_data ( input, *xsize, *ysize, *g );
637:
638:   if ( error )
639:   {
640:     cout << "\n";
641:     cout << "PGMB_READ: Fatal error!\n";
642:     cout << "  PGMB_READ_DATA failed.\n";
643:     return true;
644:   }
645:
646:   input.close ( );
647:
648:   return false;
649: }
650:
651:
652: bool pgmb_read_data ( ifstream &input, int xsize, int ysize, unsigned char *g )
653: {
654:   char c;
655:   bool error;
656:   int i;
657:   unsigned char *indexg;
658:   int j;
659:
660:   indexg = g;
661:
662:   for ( j = 0; j < ysize; j++ )
663:   {
664:     for ( i = 0; i < xsize; i++ )
665:     {
666:       input.read ( &c, 1 );
667:       *indexg = ( unsigned char ) c;
668:       indexg = indexg + 1;
669:       error = input.eof();
670:       if ( error )
671:       {
672:         cout << "\n";
673:         cout << "PGMB_READ_DATA - Fatal error!\n";
674:         cout << "  End of file reading pixel ("
675:           << i << ", " << j <<") \n";
676:         return true;
677:       }
678:     }
679:   }
680:   return false;
681: }
682:
683:
684: bool pgmb_read_header ( ifstream &input, int *xsize, int *ysize, unsigned char *maxg )
685: {
686:   int count;
687:   int fred;
688:   string line;
689:   int maxg2;
690:   string rest;
```

```
  691:   int step;
  692:   int width;
  693:   string word;
  694:
  695:   step = 0;
  696:
  697:   while ( 1 )
  698:   {
  699:     getline ( input, line );
  700:
  701:     if ( input.eof ( ) )
  702:     {
  703:       cout << "\n";
  704:       cout << "PGMB_READ_HEADER - Fatal error!\n";
  705:       cout << "  End of file.\n";
  706:       return true;
  707:     }
  708:
  709:     if ( line[0] == '#' )
  710:     {
  711:       continue;
  712:     }
  713:
  714:     if ( step == 0 )
  715:     {
  716:       s_word_extract_first ( line, word, rest );
  717:
  718:       if ( s_len_trim ( word ) <= 0 )
  719:       {
  720:         continue;
  721:       }
  722:
  723:       if ( !s_eqi ( word, "P5" ) )
  724:       {
  725:         cout << "\n";
  726:         cout << "PGMB_READ_HEADER - Fatal error.\n";
  727:         cout << "  Bad magic number = \"" << word << "\".\n";
  728:         return true;
  729:       }
  730:       line = rest;
  731:       step = 1;
  732:     }
  733:
  734:     if ( step == 1 )
  735:     {
  736:       s_word_extract_first ( line, word, rest );
  737:
  738:       if ( s_len_trim ( word ) <= 0 )
  739:       {
  740:         continue;
  741:       }
  742:       *xsize = atoi ( word.c_str ( ) );
  743:       line = rest;
  744:       step = 2;
  745:     }
  746:
  747:     if ( step == 2 )
  748:     {
  749:       s_word_extract_first ( line, word, rest );
  750:
  751:       if ( s_len_trim ( word ) <= 0 )
  752:       {
  753:         continue;
```

```
754:       }
755:       *ysize = atoi ( word.c_str ( ) );
756:       line = rest;
757:       step = 3;
758:     }
759:
760:     if ( step == 3 )
761:     {
762:       s_word_extract_first ( line, word, rest );
763:
764:       if ( s_len_trim ( word ) <= 0 )
765:       {
766:         continue;
767:       }
768:       fred = atoi ( word.c_str ( ) );
769:       *maxg = ( unsigned char ) fred;
770:       line = rest;
771:       break;
772:     }
773:   }
774:
775:   return false;
776: }
777:
778:
779: bool pgmb_to_pgma ( char *file_in_name, char *file_out_name )
780: {
781:   bool error;
782:   unsigned char *g;
783:   int *g2;
784:   unsigned char maxg;
785:   int xsize;
786:   int ysize;
787:
788:   error = pgmb_read ( file_in_name, &xsize, &ysize, &maxg, &g );
789:
790:   if ( error )
791:   {
792:     cout << "\n";
793:     cout << "PGMB_TO_PGMA: Fatal error!\n";
794:     cout << "  PGMB_READ failed.\n";
795:     return true;
796:   }
797:
798:   error = pgmb_check_data ( xsize, ysize, maxg, g );
799:
800:   if ( error )
801:   {
802:     cout << "\n";
803:     cout << "PGMB_TO_PGMA: Fatal error!\n";
804:     cout << "  PGMB_CHECK_DATA reports bad data from the file.\n";
805:
806:     delete [] g;
807:     return true;
808:   }
809:
810:   g2 = new int [ xsize *  ysize ];
811:   ucvec_to_i4vec ( xsize * ysize, g, g2 );
812:   delete [] g;
813:
814:   pgma_write ( file_out_name, xsize, ysize, g2, maxg );
815:
816:   delete [] g2;
```

```
  817:
  818:   if ( error )
  819:   {
  820:     cout << "\n";
  821:     cout << "PGMB_TO_PGMA: Fatal error!\n";
  822:     cout << "  PGMA_WRITE failed.\n";
  823:     return true;
  824:   }
  825:
  826:   return false;
  827: }
  828:
  829:
  830: bool s_eqi ( string s1, string s2 )
  831: {
  832:   int i;
  833:   int nchar;
  834:   int s1_length;
  835:   int s2_length;
  836:
  837:   s1_length = s1.length ( );
  838:   s2_length = s2.length ( );
  839:
  840:   if ( s1_length < s2_length )
  841:   {
  842:     nchar = s1_length;
  843:   }
  844:   else
  845:   {
  846:     nchar = s2_length;
  847:   }
  848:
  849:   for ( i = 0; i < nchar; i++ )
  850:   {
  851:
  852:     if ( ch_cap ( s1[i] ) != ch_cap ( s2[i] ) )
  853:     {
  854:       return false;
  855:     }
  856:   }
  857:
  858:   if ( nchar < s1_length )
  859:   {
  860:     for ( i = nchar; i < s1_length; i++ )
  861:     {
  862:       if ( s1[i] != ' ' )
  863:       {
  864:         return false;
  865:       }
  866:     }
  867:   }
  868:   else if ( nchar < s2_length )
  869:   {
  870:     for ( i = nchar; i < s2_length; i++ )
  871:     {
  872:       if ( s2[i] != ' ' )
  873:       {
  874:         return false;
  875:       }
  876:     }
  877:   }
  878:
  879:   return true;
```

```
  880: }
  881:
  882:
  883: void ucvec_to_i4vec ( int n, unsigned char *a, int *b )
  884: {
  885:   int i;
  886:
  887:   for ( i = 0; i < n; i++ )
  888:   {
  889:     *b = ( int ) *a;
  890:     a++;
  891:     b++;
  892:   }
  893:
  894:   return;
  895: }
```

```
    1: #include <cstdlib>
    2: #include <iostream>
    3: #include <iomanip>
    4: #include <fstream>
    5: #include <ctime>
    6: #include <cstring>
    7: #include <string.h>
    8:
    9: using namespace std;
   10:
   11: void i4vec_to_ucvec ( int n, int *a, unsigned char *b );
   12: void pgma_check_data ( int xsize, int ysize, int maxg, int *g );
   13: void pgma_read ( string input_name, int *xsize, int *ysize, int *maxg, int **g );
   14: void pgma_read_data ( ifstream &input, int xsize, int ysize, int *g );
   15: void pgma_read_header ( ifstream &input, int *xsize, int *ysize, int *maxg );
   16: bool pgma_to_pgmb ( char *file_in_name, char *file_out_name );
   17: bool pgmb_write ( string output_name, int xsize, int ysize, unsigned char *g, int maxgt
 ) ;
   18: bool pgmb_write_data ( ofstream &output, int xsize, int ysize, unsigned char *g );
   19: bool pgmb_write_header ( ofstream &output, int xsize, int ysize, unsigned char maxg );
   20: int s_len_trim ( string s );
   21: void s_word_extract_first ( string s, string &s1, string &s2 );
   22: void timestamp ( );
   23:
   24: char ch_cap ( char ch );
   25: void pgma_write ( string output_name, int xsize, int ysize, int *g, int maxgt );
   26: void pgma_write_data ( ofstream &output, int xsize, int ysize, int *g );
   27: void pgma_write_header ( ofstream &output, string output_name, int xsize, int ysize, in
t maxg );
   28: bool pgmb_check_data ( int xsize, int ysize, unsigned char maxg, unsigned char *g );
   29: bool pgmb_read ( string input_name, int *xsize, int *ysize, unsigned char *maxg, unsign
ed char **g );
   30: bool pgmb_read_data ( ifstream &input, int xsize, int ysize, unsigned char *g );
   31: bool pgmb_read_header ( ifstream &input, int *xsize, int *ysize, unsigned char *maxg );
   32: bool pgmb_to_pgma ( char *file_in_name, char *file_out_name );
   33: bool s_eqi ( string s1, string s2 );
   34: void ucvec_to_i4vec ( int n, unsigned char *a, int *b );
```

```
    1: #pragma once
    2: #include <string>
    3:
    4: /**
    5:  * @brief The ProgramOptions class is a singleton class with a
    6:  * fully static interface. It contains all the information the user
    7:  * provided on the command-line.
    8:  */
    9: class ProgramOptions
   10: {
   11: public:
   12:
   13:     /**
   14:      * @brief The AlgorithmSelection enum defines the 3 different supported
   15:      * Algorithms used in this program.
   16:      *
   17:      *            BFS: Breadth-First Search
   18:      * FORD_FULKERSON: The Ford-Fulkerson algorithm implemented using the BFS
   19:      *    CIRCULATION: The solution for the circulation problem
   20:      */
   21:     enum AlgorithmSelection
   22:     {
   23:         NONE = 0,
   24:         TO_BINARY = 1,
   25:         FROM_BINARY = 2,
   26:         COMPRESSED_SVD = 3,
   27:         FROM_COMPRESSED_SVD = 4,
   28:         RANDOM_IMAGE = 5,
   29:         QUICK = 6
   30:     };
   31:
   32:     /**
   33:      * @brief Clears the current ProgramOptions instance
   34:      *
   35:      * There isn't much normal use for this method besides
   36:      * in testing.
   37:      */
   38:     static void clear();
   39:
   40:     /**
   41:      * @brief Get the provided filepath of the graph text-file
   42:      * @return The provided filepath string
   43:      */
   44:     static const std::string graph_filepath();
   45:
   46:     /**
   47:      * @brief Get the singleton instance of ProgramOptions
   48:      *
   49:      * Check to see if an instance already exists.
   50:      * If it does return that.
   51:      * Otherwise, instantiate an instance and return that one.
   52:      *
   53:      * @return The current singleton instance of ProgramOptions
   54:      *
   55:      */
   56:     static ProgramOptions* instance();
   57:
   58:     /**
   59:      * @brief Parse the input options sent on the command line.
   60:      *
   61:      * Perform some basic error-checking.
   62:      *
   63:      * Example inputs:
```

```
64:       * `<program_name> -b file/path/name.ext 0 5`
65:       * `<program_name> -f file/path/name.ext`
66:       * `<program_name> -c file/path/name.ext`
67:       *
68:       * @throws std::string with the error-text.
69:       *
70:       * @param argc - The argc that is sent down from the system
71:       * @param argv - The argv that is sent down from the system
72:       */
73:      static void parse(int argc, char** argv);
74:
75:      /**
76:       * @brief Prints the help message.
77:       */
78:      static void print_help();
79:
80:      /**
81:       * @brief Get the name of the program as it was run by the user
82:       * @return The name of the program as it was run by the user
83:       */
84:      static const std::string& program_name();
85:
86:      /**
87:       * @brief Get which algorithm was selected
88:       * @return The enum the IDs the selected algorithm
89:       */
90:      static AlgorithmSelection selected_algorithm();
91:
92:      /**
93:       * @brief Get the filepath given for the ASCII pgm
94:       * file
95:       *
96:       * @return The filepath string for the ASCII pgm file
97:       */
98:      static const std::string& text_pgm_filepath();
99:
100:     /**
101:      * @brief Get the filepath given for the binary pgm
102:      * file
103:      *
104:      * @return The filepath string for the binary pgm file
105:      */
106:     static const std::string& binary_pgm_filepath();
107:
108:     /**
109:      * @brief Get the filepath given for the pgm header
110:      * file
111:      *
112:      * @return The filepath string for the pgm header file
113:      */
114:     static const std::string& pgm_header_filepath();
115:
116:     /**
117:      * @brief Get the filepath given for the SVD matrices
118:      * file
119:      *
120:      * @return The filepath string for the SVD matrices file
121:      */
122:     static const std::string& svd_matrices_filepath();
123:
124:     /**
125:      * @brief Get the approximation rank
126:      *
```

```
127:      * @return the integer given for the approximation rank
128:      */
129:     static int approximation_rank();
130:
131:
132: private:
133:     /**
134:      * @brief The hidden constructor for the ProgramOptions singleton.
135:      */
136:     ProgramOptions();
137:
138:     AlgorithmSelection m_algorithm;
139:     std::string m_program_name;
140:     std::string m_text_pgm_filepath;
141:     std::string m_binary_pgm_filepath;
142:     std::string m_pgm_header_filepath;
143:     std::string m_svd_matrices_filepath;
144:     int m_approximation_rank;
145:
146:     static ProgramOptions* s_instance;
147:
148: };
```

```
    1: #include "svd.hpp"
    2:
    3: #include <sstream>
    4: #include <string>
    5: #include <fstream>
    6: #include <iostream>
    7:
    8: std::string trim(const std::string &str)
    9: {
   10:     size_t first = str.find_first_not_of(' ');
   11:      if (std::string::npos == first)
   12:      {
   13:          return str;
   14:      }
   15:      size_t last = str.find_last_not_of(' ');
   16:      return str.substr(first, (last - first + 1));
   17: }
   18:
   19: SVD::decomp SVD::pgmSvdToHalfStream(std::istream &header, std::istream &pgm, int rank)
   20: {
   21:     int line_count = 0;
   22:     unsigned int width = 0;
   23:     unsigned int height = 0;
   24:     unsigned char max_value = 0;
   25:
   26:     std::vector<double> original_values;
   27:
   28:     for(std::string line; std::getline(header, line);line_count++) {
   29:         if(line.find('#') != std::string::npos
   30:             || line.find('P') != std::string::npos)
   31:         {
   32:             line_count--;
   33:             continue;
   34:         }
   35:         if(line_count == 0)
   36:         {
   37:            std::stringstream ss(line);
   38:            std::string width_string;
   39:            std::string height_string;
   40:
   41:            std::getline(ss, height_string, ' ');
   42:            std::getline(ss, width_string, ' ');
   43:
   44:            width = std::stoi(trim(width_string));
   45:            height = std::stoi(trim(height_string));
   46:         }
   47:         else if (line_count == 1)
   48:         {
   49:             max_value = std::stoi(line);
   50:         }
   51:     }
   52:     for(std::string line; std::getline(pgm, line);)
   53:     {
   54:         std::stringstream ss(line);
   55:         for(std::string value; std::getline(ss, value, ' ');)
   56:         {
   57:             if(trim(value).length() > 0)
   58:             {
   59:                 original_values.push_back(std::stoi(trim(value)));
   60:             }
   61:         }
   62:     }
   63:
```

```
   64:     Eigen::MatrixXd M(height, width);
   65:
   66:     unsigned int column = 0;
   67:     unsigned int row = 0;
   68:     unsigned int count = 0;
   69:     for(const auto &value: original_values)
   70:     {
   71:         M(row, column) = value;
   72:         ++count;
   73:         column = count % width;
   74:         row = count / width;
   75:     }
   76:
   77:     M /= max_value;
   78:
   79:     Eigen::BDCSVD<Eigen::MatrixXd> svd(M, Eigen::ComputeFullU | Eigen::ComputeFullV);
   80:
   81:     auto U = svd.matrixU();
   82:     auto S = svd.singularValues();
   83:     auto V = svd.matrixV();
   84: //    std::cout << "M:" << std::endl << M << std::endl;
   85: //    std::cout << "U:" << std::endl << U << std::endl;
   86:     std::cout << "S:" << std::endl << S << std::endl;
   87: //    std::cout << "V:" << std::endl << V << std::endl;
   88:
   89:     std::vector<half_float::half> U_vec;
   90:     std::vector<half_float::half> S_vec;
   91:     std::vector<half_float::half> V_vec;
   92:
   93:     for (unsigned int column = 0; column < rank; column++)
   94:     {
   95:         for(unsigned int row = 0; row < U.rows(); row++)
   96:         {
   97:             double value = U(row, column);
   98:             half_float::half half_p = half_float::half_cast<half_float::half>(value);
   99:             U_vec.push_back(half_p);
  100:         }
  101:     }
  102:
  103:     for(unsigned int row = 0; row < rank; row++)
  104:     {
  105:         double value = S(row, 0);
  106:         half_float::half half_p = half_float::half_cast<half_float::half>(value);
  107:         S_vec.push_back(half_p);
  108:     }
  109:
  110:     for (unsigned int column = 0; column < rank; column++)
  111:     {
  112:         for(unsigned int row = 0; row < V.rows(); row++)
  113:         {
  114:             double value = V(row, column);
  115:             half_float::half half_p = half_float::half_cast<half_float::half>(value);
  116:             V_vec.push_back(half_p);
  117:         }
  118:     }
  119:
  120:     return {
  121:         {
  122:             U.rows(),
  123:             V.rows(),
  124:             rank,
  125:             max_value
  126:         },
```

```
  127:         U_vec,
  128:         S_vec,
  129:         V_vec
  130:     };
  131:
  132: }
  133:
  134: SVD::decomp SVD::rankDecomp(const Eigen::MatrixXd &U, const Eigen::MatrixXd &S, const E
igen::MatrixXd &V, int rank, unsigned char max_value)
  135: {
  136:
  137:     std::vector<half_float::half> U_vec;
  138:     std::vector<half_float::half> S_vec;
  139:     std::vector<half_float::half> V_vec;
  140:
  141:     for (unsigned int column = 0; column < rank; column++)
  142:     {
  143:         for(unsigned int row = 0; row < U.rows(); row++)
  144:         {
  145:             double value = U(row, column);
  146:             half_float::half half_p = half_float::half_cast<half_float::half>(value);
  147:             U_vec.push_back(half_p);
  148:         }
  149:     }
  150:
  151:     for(unsigned int row = 0; row < rank; row++)
  152:     {
  153:         double value = S(row, 0);
  154:         half_float::half half_p = half_float::half_cast<half_float::half>(value);
  155:         S_vec.push_back(half_p);
  156:     }
  157:
  158:     for (unsigned int column = 0; column < rank; column++)
  159:     {
  160:         for(unsigned int row = 0; row < V.rows(); row++)
  161:         {
  162:             double value = V(row, column);
  163:             half_float::half half_p = half_float::half_cast<half_float::half>(value);
  164:             V_vec.push_back(half_p);
  165:         }
  166:     }
  167:
  168:     return {
  169:         {
  170:             U.rows(),
  171:             V.rows(),
  172:             rank,
  173:             max_value
  174:         },
  175:         U_vec,
  176:         S_vec,
  177:         V_vec
  178:     };
  179: }
  180:
  181: void SVD::writeAllDecomps(std::istream &header, std::istream &pgm, const std::string &f
ilename)
  182: {
  183:     int line_count = 0;
  184:     unsigned int width = 0;
  185:     unsigned int height = 0;
  186:     unsigned char max_value = 0;
  187:
```

```
188:     std::vector<double> original_values;
189:
190:     for(std::string line; std::getline(header, line);line_count++) {
191:         if(line.find('#') != std::string::npos
192:             || line.find('P') != std::string::npos)
193:         {
194:             line_count--;
195:             continue;
196:         }
197:         if(line_count == 0)
198:         {
199:            std::stringstream ss(line);
200:            std::string width_string;
201:            std::string height_string;
202:
203:            std::getline(ss, height_string, ' ');
204:            std::getline(ss, width_string, ' ');
205:
206:            width = std::stoi(trim(width_string));
207:            height = std::stoi(trim(height_string));
208:         }
209:         else if (line_count == 1)
210:         {
211:             max_value = std::stoi(trim(line));
212:         }
213:     }
214:     for(std::string line; std::getline(pgm, line);)
215:     {
216:         std::stringstream ss(line);
217:         for(std::string value; std::getline(ss, value, ' ');)
218:         {
219:             if(trim(value).length() > 0)
220:             {
221:                 original_values.push_back(std::stoi(trim(value)));
222:             }
223:         }
224:     }
225:
226:     Eigen::MatrixXd M(height, width);
227:
228:     unsigned int column = 0;
229:     unsigned int row = 0;
230:     unsigned int count = 0;
231:     for(const auto &value: original_values)
232:     {
233:         M(row, column) = value;
234:         ++count;
235:         column = count % width;
236:         row = count / width;
237:     }
238:
239:     M /= max_value;
240:
241:     Eigen::BDCSVD<Eigen::MatrixXd> svd(M, Eigen::ComputeFullU | Eigen::ComputeFullV);
242:
243:     auto U = svd.matrixU();
244:     auto S = svd.singularValues();
245:     auto V = svd.matrixV();
246:
247:     unsigned int max_rank = std::min(height, width);
248:
249:     for(unsigned int r = 1; r <= max_rank; r++)
250:     {
```

```
  251:         SVD::writePgmAsSvd(filename + "_" + std::to_string(r), rankDecomp(U, S, V, r, m
ax_value));
  252:         std::cout << "Compressed " << r << " of " << max_rank << std::endl;
  253:     }
  254:
  255:     for(unsigned int r = 1; r <= max_rank; r++)
  256:     {
  257:         auto [text, rank] = SVD::svdToPGMString(filename + "_" + std::to_string(r));
  258:         std::ofstream out(filename + "_" + std::to_string(r) + ".pgm");
  259:         out << text;
  260:         out.flush();
  261:         out.close();
  262:         std::cout << "Decompressed " << r << " of " << max_rank << std::endl;
  263:     }
  264:
  265: }
  266:
  267: void SVD::writePgmAsSvd(const std::string &output_path, decomp decomposition)
  268: {
  269:     std::ofstream file(output_path, std::ios::out | std::ios::binary);
  270:
  271:     file.write((char*)&(decomposition.meta), sizeof (metadata));
  272:
  273:     for(const auto &value : decomposition.U)
  274:     {
  275:         file.write((char*)&value, sizeof (half_float::half));
  276:     }
  277:
  278:     for (const auto &value : decomposition.S)
  279:     {
  280:         file.write((char*)&value, sizeof (half_float::half));
  281:     }
  282:
  283:     for (const auto &value : decomposition.V)
  284:     {
  285:         file.write((char*)&value, sizeof (half_float::half));
  286:     }
  287:
  288:     file.flush();
  289:
  290:     file.close();
  291: }
  292:
  293: std::tuple<std::string, long> SVD::svdToPGMString(const std::string &input_filename)
  294: {
  295:     std::ifstream file(input_filename, std::ios::in | std::ios::binary);
  296:
  297:     metadata *sizes = new metadata;
  298:
  299:     file.read((char*)sizes, sizeof (metadata));
  300:
  301:     std::vector<half_float::half> values;
  302:     half_float::half temp;
  303:     while(file.read((char*)&temp, sizeof(half_float::half)))
  304:     {
  305:         values.push_back(temp);
  306:     }
  307:
  308:     file.close();
  309:
  310:     // Calculate approximated PGM
  311:     unsigned long U_size =  sizes->rank * sizes->U_height;
  312:     unsigned long V_size = sizes->V_width * sizes->rank;
```

```
313:     Eigen::MatrixXd U = Eigen::MatrixXd::Zero(sizes->U_height, sizes->U_height);
314:     Eigen::MatrixXd S = Eigen::MatrixXd::Zero(sizes->U_height, sizes->V_width);
315:     Eigen::MatrixXd V = Eigen::MatrixXd::Zero(sizes->V_width, sizes->V_width);
316:     long count = 0;
317:     for(const auto &value: values)// TODO ensure correct order...
318:     {
319:         if(count < (U_size))
320:         {
321:             long row = count%sizes->U_height;
322:             long col = (count < sizes->U_height)? 0 : (count)/sizes->U_height;
323: //            std::cout << "U(" << row << ", " << col << ")" <<std::endl;
324:             U(row, col) = half_float::half_cast<double>(value);
325:
326:         }
327:         else if(count < (U_size) + sizes->rank )
328:         {
329:             S(count - U_size, count - U_size) = half_float::half_cast<double>(value);
330:         }
331:         else
332:         {
333:             long v_count = count - (U_size + sizes->rank);
334:             long row = v_count%sizes->V_width;
335:             long col = (count < sizes->V_width) ? 0 : (v_count)/sizes->V_width;
336: //            std::cout << "V(" << row << ", " << col << ")" <<std::endl;
337:             V(row, col) = half_float::half_cast<double>(value);
338:         }
339:         count++;
340:     }
341: //    std::cout << "U: " << std::endl << U << std::endl;
342: //    std::cout << "S: " << std::endl << S << std::endl;
343: //    std::cout << "V: " << std::endl << V << std::endl;
344:
345:     Eigen::MatrixXd pgm_approx = U * (S * V.transpose());
346:     pgm_approx *= sizes->max_value;
347:
348: //    std::cout << "APPROXIMATED PICTURE" << std::endl << pgm_approx << std::endl;
349:
350:     // round to nearest integer value and get other info.
351:     unsigned long width = pgm_approx.cols();
352:     unsigned long height = pgm_approx.rows();
353:
354:     std::vector<unsigned char> pgm_values;
355:
356:     for(unsigned int row = 0; row < pgm_approx.rows(); row++)
357:     {
358:         for (unsigned int column = 0; column < pgm_approx.cols(); column++)
359:         {
360:             long value = std::lround(pgm_approx(row, column));
361:             if(value > UCHAR_MAX)
362:             {
363:                 value = UCHAR_MAX;
364:             }
365:             else if (value < 0)
366:             {
367:                 value = 0;
368:             }
369:
370:             pgm_values.push_back((unsigned char)value);
371:         }
372:     }
373:
374:     std::stringstream out;
375:     out << "P2" << std::endl;
```

```
376:     out << std::to_string(height) << " " << std::to_string(width) << std::endl;
377:     out << std::to_string(sizes->max_value) << std::endl;
378:
379:     int val_count = 0;
380:     for(const auto &value: pgm_values)
381:     {
382:         out << std::to_string(value);
383:         if(val_count > 20)
384:         {
385:             val_count = -1;
386:             out << std::endl;
387:         }
388:         else
389:         {
390:             out << " ";
391:         }
392:         val_count++;
393:     }
394:
395:     out.flush();
396:     return {out.str(), sizes->rank};
397: }
```

```
    1: #pragma once
    2: #include <istream>
    3: #include <vector>
    4: #include <tuple>
    5: #include <utility>
    6: #include <include/half.hpp>
    7: #include <Eigen/SVD>
    8:
    9: class SVD {
   10: public:
   11:     struct metadata
   12:     {
   13:         long U_height;
   14:         long V_width;
   15:         long rank;
   16:         unsigned char max_value;
   17:     };
   18:
   19:     struct decomp
   20:     {
   21:         metadata meta;
   22:         std::vector<half_float::half> U;
   23:         std::vector<half_float::half> S;
   24:         std::vector<half_float::half> V;
   25:     };
   26:
   27:     static decomp pgmSvdToHalfStream(std::istream &header, std::istream &pgm, int rank)
;
   28:
   29:     static void writePgmAsSvd(const std::string &output_path, decomp decomposition);
   30:
   31:     static std::tuple<std::string, long> svdToPGMString(const std::string &input_filena
me);
   32:     static SVD::decomp rankDecomp(const Eigen::MatrixXd &U, const Eigen::MatrixXd &S, c
onst Eigen::MatrixXd &V, int rank, unsigned char max_value);
   33:     static void writeAllDecomps(std::istream &header, std::istream &pgm, const std::str
ing &filename);
   34: };
```