#ifndef \_GNU\_SOURCE

#define \_GNU\_SOURCE

#endif

#include "utils.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#ifndef \_USE\_MATH\_DEFINES

#define \_USE\_MATH\_DEFINES

#endif

#include <math.h>

#include <assert.h>

#include <float.h>

#include <limits.h>

#include "darkunistd.h"

#ifdef WIN32

#include "gettimeofday.h"

#else

#include <sys/time.h>

#include <sys/stat.h>

#endif

#ifndef USE\_CMAKE\_LIBS

#pragma warning(disable: 4996)

#endif

void \*xmalloc(size\_t size) {

void \*ptr=malloc(size);

if(!ptr) {

malloc\_error();

}

return ptr;

}

void \*xcalloc(size\_t nmemb, size\_t size) {

void \*ptr=calloc(nmemb,size);

if(!ptr) {

calloc\_error();

}

return ptr;

}

void \*xrealloc(void \*ptr, size\_t size) {

ptr=realloc(ptr,size);

if(!ptr) {

realloc\_error();

}

return ptr;

}

double what\_time\_is\_it\_now()

{

struct timeval time;

if (gettimeofday(&time, NULL)) {

return 0;

}

return (double)time.tv\_sec + (double)time.tv\_usec \* .000001;

}

int \*read\_map(char \*filename)

{

int n = 0;

int \*map = 0;

char \*str;

FILE \*file = fopen(filename, "r");

if(!file) file\_error(filename);

while((str=fgetl(file))){

++n;

map = (int\*)xrealloc(map, n \* sizeof(int));

map[n-1] = atoi(str);

free(str);

}

if (file) fclose(file);

return map;

}

void sorta\_shuffle(void \*arr, size\_t n, size\_t size, size\_t sections)

{

size\_t i;

for(i = 0; i < sections; ++i){

size\_t start = n\*i/sections;

size\_t end = n\*(i+1)/sections;

size\_t num = end-start;

shuffle((char\*)arr+(start\*size), num, size);

}

}

void shuffle(void \*arr, size\_t n, size\_t size)

{

size\_t i;

void\* swp = (void\*)xcalloc(1, size);

for(i = 0; i < n-1; ++i){

size\_t j = i + random\_gen()/(RAND\_MAX / (n-i)+1);

memcpy(swp, (char\*)arr+(j\*size), size);

memcpy((char\*)arr+(j\*size), (char\*)arr+(i\*size), size);

memcpy((char\*)arr+(i\*size), swp, size);

}

free(swp);

}

void del\_arg(int argc, char \*\*argv, int index)

{

int i;

for(i = index; i < argc-1; ++i) argv[i] = argv[i+1];

argv[i] = 0;

}

int find\_arg(int argc, char\* argv[], char \*arg)

{

int i;

for(i = 0; i < argc; ++i) {

if(!argv[i]) continue;

if(0==strcmp(argv[i], arg)) {

del\_arg(argc, argv, i);

return 1;

}

}

return 0;

}

int find\_int\_arg(int argc, char \*\*argv, char \*arg, int def)

{

int i;

for(i = 0; i < argc-1; ++i){

if(!argv[i]) continue;

if(0==strcmp(argv[i], arg)){

def = atoi(argv[i+1]);

del\_arg(argc, argv, i);

del\_arg(argc, argv, i);

break;

}

}

return def;

}

float find\_float\_arg(int argc, char \*\*argv, char \*arg, float def)

{

int i;

for(i = 0; i < argc-1; ++i){

if(!argv[i]) continue;

if(0==strcmp(argv[i], arg)){

def = atof(argv[i+1]);

del\_arg(argc, argv, i);

del\_arg(argc, argv, i);

break;

}

}

return def;

}

char \*find\_char\_arg(int argc, char \*\*argv, char \*arg, char \*def)

{

int i;

for(i = 0; i < argc-1; ++i){

if(!argv[i]) continue;

if(0==strcmp(argv[i], arg)){

def = argv[i+1];

del\_arg(argc, argv, i);

del\_arg(argc, argv, i);

break;

}

}

return def;

}

char \*basecfg(char \*cfgfile)

{

char \*c = cfgfile;

char \*next;

while((next = strchr(c, '/')))

{

c = next+1;

}

if(!next) while ((next = strchr(c, '\\'))) { c = next + 1; }

c = copy\_string(c);

next = strchr(c, '.');

if (next) \*next = 0;

return c;

}

int alphanum\_to\_int(char c)

{

return (c < 58) ? c - 48 : c-87;

}

char int\_to\_alphanum(int i)

{

if (i == 36) return '.';

return (i < 10) ? i + 48 : i + 87;

}

void pm(int M, int N, float \*A)

{

int i,j;

for(i =0 ; i < M; ++i){

printf("%d ", i+1);

for(j = 0; j < N; ++j){

printf("%2.4f, ", A[i\*N+j]);

}

printf("\n");

}

printf("\n");

}

void find\_replace(const char\* str, char\* orig, char\* rep, char\* output)

{

char\* buffer = (char\*)calloc(8192, sizeof(char));

char \*p;

sprintf(buffer, "%s", str);

if (!(p = strstr(buffer, orig))) { // Is 'orig' even in 'str'?

sprintf(output, "%s", buffer);

free(buffer);

return;

}

\*p = '\0';

sprintf(output, "%s%s%s", buffer, rep, p + strlen(orig));

free(buffer);

}

void trim(char \*str)

{

char\* buffer = (char\*)xcalloc(8192, sizeof(char));

sprintf(buffer, "%s", str);

char \*p = buffer;

while (\*p == ' ' || \*p == '\t') ++p;

char \*end = p + strlen(p) - 1;

while (\*end == ' ' || \*end == '\t') {

\*end = '\0';

--end;

}

sprintf(str, "%s", p);

free(buffer);

}

void find\_replace\_extension(char \*str, char \*orig, char \*rep, char \*output)

{

char\* buffer = (char\*)calloc(8192, sizeof(char));

sprintf(buffer, "%s", str);

char \*p = strstr(buffer, orig);

int offset = (p - buffer);

int chars\_from\_end = strlen(buffer) - offset;

if (!p || chars\_from\_end != strlen(orig)) { // Is 'orig' even in 'str' AND is 'orig' found at the end of 'str'?

sprintf(output, "%s", buffer);

free(buffer);

return;

}

\*p = '\0';

sprintf(output, "%s%s%s", buffer, rep, p + strlen(orig));

free(buffer);

}

void replace\_image\_to\_label(const char\* input\_path, char\* output\_path)

{

find\_replace(input\_path, "/images/train2014/", "/labels/train2014/", output\_path); // COCO

find\_replace(output\_path, "/images/val2014/", "/labels/val2014/", output\_path); // COCO

find\_replace(output\_path, "/JPEGImages/", "/labels/", output\_path); // PascalVOC

find\_replace(output\_path, "\\images\\train2014\\", "\\labels\\train2014\\", output\_path); // COCO

find\_replace(output\_path, "\\images\\val2014\\", "\\labels\\val2014\\", output\_path); // COCO

find\_replace(output\_path, "\\JPEGImages\\", "\\labels\\", output\_path); // PascalVOC

//find\_replace(output\_path, "/images/", "/labels/", output\_path); // COCO

//find\_replace(output\_path, "/VOC2007/JPEGImages/", "/VOC2007/labels/", output\_path); // PascalVOC

//find\_replace(output\_path, "/VOC2012/JPEGImages/", "/VOC2012/labels/", output\_path); // PascalVOC

//find\_replace(output\_path, "/raw/", "/labels/", output\_path);

trim(output\_path);

// replace only ext of files

find\_replace\_extension(output\_path, ".jpg", ".txt", output\_path);

find\_replace\_extension(output\_path, ".JPG", ".txt", output\_path); // error

find\_replace\_extension(output\_path, ".jpeg", ".txt", output\_path);

find\_replace\_extension(output\_path, ".JPEG", ".txt", output\_path);

find\_replace\_extension(output\_path, ".png", ".txt", output\_path);

find\_replace\_extension(output\_path, ".PNG", ".txt", output\_path);

find\_replace\_extension(output\_path, ".bmp", ".txt", output\_path);

find\_replace\_extension(output\_path, ".BMP", ".txt", output\_path);

find\_replace\_extension(output\_path, ".ppm", ".txt", output\_path);

find\_replace\_extension(output\_path, ".PPM", ".txt", output\_path);

find\_replace\_extension(output\_path, ".tiff", ".txt", output\_path);

find\_replace\_extension(output\_path, ".TIFF", ".txt", output\_path);

// Check file ends with txt:

if(strlen(output\_path) > 4) {

char \*output\_path\_ext = output\_path + strlen(output\_path) - 4;

if( strcmp(".txt", output\_path\_ext) != 0){

fprintf(stderr, "Failed to infer label file name (check image extension is supported): %s \n", output\_path);

}

}else{

fprintf(stderr, "Label file name is too short: %s \n", output\_path);

}

}

float sec(clock\_t clocks)

{

return (float)clocks/CLOCKS\_PER\_SEC;

}

void top\_k(float \*a, int n, int k, int \*index)

{

int i,j;

for(j = 0; j < k; ++j) index[j] = -1;

for(i = 0; i < n; ++i){

int curr = i;

for(j = 0; j < k; ++j){

if((index[j] < 0) || a[curr] > a[index[j]]){

int swap = curr;

curr = index[j];

index[j] = swap;

}

}

}

}

void error(const char \*s)

{

perror(s);

assert(0);

exit(EXIT\_FAILURE);

}

void malloc\_error()

{

fprintf(stderr, "xMalloc error\n");

exit(EXIT\_FAILURE);

}

void calloc\_error()

{

fprintf(stderr, "Calloc error\n");

exit(EXIT\_FAILURE);

}

void realloc\_error()

{

fprintf(stderr, "Realloc error\n");

exit(EXIT\_FAILURE);

}

void file\_error(char \*s)

{

fprintf(stderr, "Couldn't open file: %s\n", s);

exit(EXIT\_FAILURE);

}

list \*split\_str(char \*s, char delim)

{

size\_t i;

size\_t len = strlen(s);

list \*l = make\_list();

list\_insert(l, s);

for(i = 0; i < len; ++i){

if(s[i] == delim){

s[i] = '\0';

list\_insert(l, &(s[i+1]));

}

}

return l;

}

void strip(char \*s)

{

size\_t i;

size\_t len = strlen(s);

size\_t offset = 0;

for(i = 0; i < len; ++i){

char c = s[i];

if(c==' '||c=='\t'||c=='\n'||c =='\r'||c==0x0d||c==0x0a) ++offset;

else s[i-offset] = c;

}

s[len-offset] = '\0';

}

void strip\_args(char \*s)

{

size\_t i;

size\_t len = strlen(s);

size\_t offset = 0;

for (i = 0; i < len; ++i) {

char c = s[i];

if (c == '\t' || c == '\n' || c == '\r' || c == 0x0d || c == 0x0a) ++offset;

else s[i - offset] = c;

}

s[len - offset] = '\0';

}

void strip\_char(char \*s, char bad)

{

size\_t i;

size\_t len = strlen(s);

size\_t offset = 0;

for(i = 0; i < len; ++i){

char c = s[i];

if(c==bad) ++offset;

else s[i-offset] = c;

}

s[len-offset] = '\0';

}

void free\_ptrs(void \*\*ptrs, int n)

{

int i;

for(i = 0; i < n; ++i) free(ptrs[i]);

free(ptrs);

}

char \*fgetl(FILE \*fp)

{

if(feof(fp)) return 0;

size\_t size = 512;

char\* line = (char\*)xmalloc(size \* sizeof(char));

if(!fgets(line, size, fp)){

free(line);

return 0;

}

size\_t curr = strlen(line);

while((line[curr-1] != '\n') && !feof(fp)){

if(curr == size-1){

size \*= 2;

line = (char\*)xrealloc(line, size \* sizeof(char));

}

size\_t readsize = size-curr;

if(readsize > INT\_MAX) readsize = INT\_MAX-1;

fgets(&line[curr], readsize, fp);

curr = strlen(line);

}

if(curr >= 2)

if(line[curr-2] == 0x0d) line[curr-2] = 0x00;

if(curr >= 1)

if(line[curr-1] == 0x0a) line[curr-1] = 0x00;

return line;

}

int read\_int(int fd)

{

int n = 0;

int next = read(fd, &n, sizeof(int));

if(next <= 0) return -1;

return n;

}

void write\_int(int fd, int n)

{

int next = write(fd, &n, sizeof(int));

if(next <= 0) error("read failed");

}

int read\_all\_fail(int fd, char \*buffer, size\_t bytes)

{

size\_t n = 0;

while(n < bytes){

int next = read(fd, buffer + n, bytes-n);

if(next <= 0) return 1;

n += next;

}

return 0;

}

int write\_all\_fail(int fd, char \*buffer, size\_t bytes)

{

size\_t n = 0;

while(n < bytes){

size\_t next = write(fd, buffer + n, bytes-n);

if(next <= 0) return 1;

n += next;

}

return 0;

}

void read\_all(int fd, char \*buffer, size\_t bytes)

{

size\_t n = 0;

while(n < bytes){

int next = read(fd, buffer + n, bytes-n);

if(next <= 0) error("read failed");

n += next;

}

}

void write\_all(int fd, char \*buffer, size\_t bytes)

{

size\_t n = 0;

while(n < bytes){

size\_t next = write(fd, buffer + n, bytes-n);

if(next <= 0) error("write failed");

n += next;

}

}

char \*copy\_string(char \*s)

{

if(!s) {

return NULL;

}

char\* copy = (char\*)xmalloc(strlen(s) + 1);

strncpy(copy, s, strlen(s)+1);

return copy;

}

list \*parse\_csv\_line(char \*line)

{

list \*l = make\_list();

char \*c, \*p;

int in = 0;

for(c = line, p = line; \*c != '\0'; ++c){

if(\*c == '"') in = !in;

else if(\*c == ',' && !in){

\*c = '\0';

list\_insert(l, copy\_string(p));

p = c+1;

}

}

list\_insert(l, copy\_string(p));

return l;

}

int count\_fields(char \*line)

{

int count = 0;

int done = 0;

char \*c;

for(c = line; !done; ++c){

done = (\*c == '\0');

if(\*c == ',' || done) ++count;

}

return count;

}

float \*parse\_fields(char \*line, int n)

{

float\* field = (float\*)xcalloc(n, sizeof(float));

char \*c, \*p, \*end;

int count = 0;

int done = 0;

for(c = line, p = line; !done; ++c){

done = (\*c == '\0');

if(\*c == ',' || done){

\*c = '\0';

field[count] = strtod(p, &end);

if(p == c) field[count] = nan("");

if(end != c && (end != c-1 || \*end != '\r')) field[count] = nan(""); //DOS file formats!

p = c+1;

++count;

}

}

return field;

}

float sum\_array(float \*a, int n)

{

int i;

float sum = 0;

for(i = 0; i < n; ++i) sum += a[i];

return sum;

}

float mean\_array(float \*a, int n)

{

return sum\_array(a,n)/n;

}

void mean\_arrays(float \*\*a, int n, int els, float \*avg)

{

int i;

int j;

memset(avg, 0, els\*sizeof(float));

for(j = 0; j < n; ++j){

for(i = 0; i < els; ++i){

avg[i] += a[j][i];

}

}

for(i = 0; i < els; ++i){

avg[i] /= n;

}

}

void print\_statistics(float \*a, int n)

{

float m = mean\_array(a, n);

float v = variance\_array(a, n);

printf("MSE: %.6f, Mean: %.6f, Variance: %.6f\n", mse\_array(a, n), m, v);

}

float variance\_array(float \*a, int n)

{

int i;

float sum = 0;

float mean = mean\_array(a, n);

for(i = 0; i < n; ++i) sum += (a[i] - mean)\*(a[i]-mean);

float variance = sum/n;

return variance;

}

int constrain\_int(int a, int min, int max)

{

if (a < min) return min;

if (a > max) return max;

return a;

}

float constrain(float min, float max, float a)

{

if (a < min) return min;

if (a > max) return max;

return a;

}

float dist\_array(float \*a, float \*b, int n, int sub)

{

int i;

float sum = 0;

for(i = 0; i < n; i += sub) sum += pow(a[i]-b[i], 2);

return sqrt(sum);

}

float mse\_array(float \*a, int n)

{

int i;

float sum = 0;

for(i = 0; i < n; ++i) sum += a[i]\*a[i];

return sqrt(sum/n);

}

void normalize\_array(float \*a, int n)

{

int i;

float mu = mean\_array(a,n);

float sigma = sqrt(variance\_array(a,n));

for(i = 0; i < n; ++i){

a[i] = (a[i] - mu)/sigma;

}

mu = mean\_array(a,n);

sigma = sqrt(variance\_array(a,n));

}

void translate\_array(float \*a, int n, float s)

{

int i;

for(i = 0; i < n; ++i){

a[i] += s;

}

}

float mag\_array(float \*a, int n)

{

int i;

float sum = 0;

for(i = 0; i < n; ++i){

sum += a[i]\*a[i];

}

return sqrt(sum);

}

// indicies to skip is a bit array

float mag\_array\_skip(float \*a, int n, int \* indices\_to\_skip)

{

int i;

float sum = 0;

for (i = 0; i < n; ++i) {

if (indices\_to\_skip[i] != 1) {

sum += a[i] \* a[i];

}

}

return sqrt(sum);

}

void scale\_array(float \*a, int n, float s)

{

int i;

for(i = 0; i < n; ++i){

a[i] \*= s;

}

}

int sample\_array(float \*a, int n)

{

float sum = sum\_array(a, n);

scale\_array(a, n, 1. / sum);

float r = rand\_uniform(0, 1);

int i;

for (i = 0; i < n; ++i) {

r = r - a[i];

if (r <= 0) return i;

}

return n - 1;

}

int sample\_array\_custom(float \*a, int n)

{

float sum = sum\_array(a, n);

scale\_array(a, n, 1./sum);

float r = rand\_uniform(0, 1);

int start\_index = rand\_int(0, 0);

int i;

for(i = 0; i < n; ++i){

r = r - a[(i + start\_index) % n];

if (r <= 0) return i;

}

return n-1;

}

int max\_index(float \*a, int n)

{

if(n <= 0) return -1;

int i, max\_i = 0;

float max = a[0];

for(i = 1; i < n; ++i){

if(a[i] > max){

max = a[i];

max\_i = i;

}

}

return max\_i;

}

int top\_max\_index(float \*a, int n, int k)

{

if (n <= 0) return -1;

float \*values = (float\*)xcalloc(k, sizeof(float));

int \*indexes = (int\*)xcalloc(k, sizeof(int));

int i, j;

for (i = 0; i < n; ++i) {

for (j = 0; j < k; ++j) {

if (a[i] > values[j]) {

values[j] = a[i];

indexes[j] = i;

break;

}

}

}

int count = 0;

for (j = 0; j < k; ++j) if (values[j] > 0) count++;

int get\_index = rand\_int(0, count-1);

int val = indexes[get\_index];

free(indexes);

free(values);

return val;

}

int int\_index(int \*a, int val, int n)

{

int i;

for (i = 0; i < n; ++i) {

if (a[i] == val) return i;

}

return -1;

}

int rand\_int(int min, int max)

{

if (max < min){

int s = min;

min = max;

max = s;

}

int r = (random\_gen()%(max - min + 1)) + min;

return r;

}

// From http://en.wikipedia.org/wiki/Box%E2%80%93Muller\_transform

float rand\_normal()

{

static int haveSpare = 0;

static double rand1, rand2;

if(haveSpare)

{

haveSpare = 0;

return sqrt(rand1) \* sin(rand2);

}

haveSpare = 1;

rand1 = random\_gen() / ((double) RAND\_MAX);

if(rand1 < 1e-100) rand1 = 1e-100;

rand1 = -2 \* log(rand1);

rand2 = (random\_gen() / ((double)RAND\_MAX)) \* 2.0 \* M\_PI;

return sqrt(rand1) \* cos(rand2);

}

/\*

float rand\_normal()

{

int n = 12;

int i;

float sum= 0;

for(i = 0; i < n; ++i) sum += (float)random\_gen()/RAND\_MAX;

return sum-n/2.;

}

\*/

size\_t rand\_size\_t()

{

return ((size\_t)(random\_gen()&0xff) << 56) |

((size\_t)(random\_gen()&0xff) << 48) |

((size\_t)(random\_gen()&0xff) << 40) |

((size\_t)(random\_gen()&0xff) << 32) |

((size\_t)(random\_gen()&0xff) << 24) |

((size\_t)(random\_gen()&0xff) << 16) |

((size\_t)(random\_gen()&0xff) << 8) |

((size\_t)(random\_gen()&0xff) << 0);

}

float rand\_uniform(float min, float max)

{

if(max < min){

float swap = min;

min = max;

max = swap;

}

#if (RAND\_MAX < 65536)

int rnd = rand()\*(RAND\_MAX + 1) + rand();

return ((float)rnd / (RAND\_MAX\*RAND\_MAX) \* (max - min)) + min;

#else

return ((float)rand() / RAND\_MAX \* (max - min)) + min;

#endif

//return (random\_float() \* (max - min)) + min;

}

float rand\_scale(float s)

{

float scale = rand\_uniform\_strong(1, s);

if(random\_gen()%2) return scale;

return 1./scale;

}

float \*\*one\_hot\_encode(float \*a, int n, int k)

{

int i;

float\*\* t = (float\*\*)xcalloc(n, sizeof(float\*));

for(i = 0; i < n; ++i){

t[i] = (float\*)xcalloc(k, sizeof(float));

int index = (int)a[i];

t[i][index] = 1;

}

return t;

}

static unsigned int x = 123456789, y = 362436069, z = 521288629;

// Marsaglia's xorshf96 generator: period 2^96-1

unsigned int random\_gen\_fast(void)

{

unsigned int t;

x ^= x << 16;

x ^= x >> 5;

x ^= x << 1;

t = x;

x = y;

y = z;

z = t ^ x ^ y;

return z;

}

float random\_float\_fast()

{

return ((float)random\_gen\_fast() / (float)UINT\_MAX);

}

int rand\_int\_fast(int min, int max)

{

if (max < min) {

int s = min;

min = max;

max = s;

}

int r = (random\_gen\_fast() % (max - min + 1)) + min;

return r;

}

unsigned int random\_gen()

{

unsigned int rnd = 0;

#ifdef WIN32

rand\_s(&rnd);

#else // WIN32

rnd = rand();

#if (RAND\_MAX < 65536)

rnd = rand()\*(RAND\_MAX + 1) + rnd;

#endif //(RAND\_MAX < 65536)

#endif // WIN32

return rnd;

}

float random\_float()

{

unsigned int rnd = 0;

#ifdef WIN32

rand\_s(&rnd);

return ((float)rnd / (float)UINT\_MAX);

#else // WIN32

rnd = rand();

#if (RAND\_MAX < 65536)

rnd = rand()\*(RAND\_MAX + 1) + rnd;

return((float)rnd / (float)(RAND\_MAX\*RAND\_MAX));

#endif //(RAND\_MAX < 65536)

return ((float)rnd / (float)RAND\_MAX);

#endif // WIN32

}

float rand\_uniform\_strong(float min, float max)

{

if (max < min) {

float swap = min;

min = max;

max = swap;

}

return (random\_float() \* (max - min)) + min;

}

float rand\_precalc\_random(float min, float max, float random\_part)

{

if (max < min) {

float swap = min;

min = max;

max = swap;

}

return (random\_part \* (max - min)) + min;

}

#define RS\_SCALE (1.0 / (1.0 + RAND\_MAX))

double double\_rand(void)

{

double d;

do {

d = (((rand() \* RS\_SCALE) + rand()) \* RS\_SCALE + rand()) \* RS\_SCALE;

} while (d >= 1); // Round off

return d;

}

unsigned int uint\_rand(unsigned int less\_than)

{

return (unsigned int)((less\_than)\* double\_rand());

}

int check\_array\_is\_nan(float \*arr, int size)

{

int i;

for (i = 0; i < size; ++i) {

if (isnan(arr[i])) return 1;

}

return 0;

}

int check\_array\_is\_inf(float \*arr, int size)

{

int i;

for (i = 0; i < size; ++i) {

if (isinf(arr[i])) return 1;

}

return 0;

}

int \*random\_index\_order(int min, int max)

{

int \*inds = (int \*)xcalloc(max - min, sizeof(int));

int i;

for (i = min; i < max; ++i) {

inds[i - min] = i;

}

for (i = min; i < max - 1; ++i) {

int swap = inds[i - min];

int index = i + rand() % (max - i);

inds[i - min] = inds[index - min];

inds[index - min] = swap;

}

return inds;

}

int max\_int\_index(int \*a, int n)

{

if (n <= 0) return -1;

int i, max\_i = 0;

int max = a[0];

for (i = 1; i < n; ++i) {

if (a[i] > max) {

max = a[i];

max\_i = i;

}

}

return max\_i;

}

// Absolute box from relative coordinate bounding box and image size

boxabs box\_to\_boxabs(const box\* b, const int img\_w, const int img\_h, const int bounds\_check)

{

boxabs ba;

ba.left = (b->x - b->w / 2.)\*img\_w;

ba.right = (b->x + b->w / 2.)\*img\_w;

ba.top = (b->y - b->h / 2.)\*img\_h;

ba.bot = (b->y + b->h / 2.)\*img\_h;

if (bounds\_check) {

if (ba.left < 0) ba.left = 0;

if (ba.right > img\_w - 1) ba.right = img\_w - 1;

if (ba.top < 0) ba.top = 0;

if (ba.bot > img\_h - 1) ba.bot = img\_h - 1;

}

return ba;

}

int make\_directory(char \*path, int mode)

{

#ifdef WIN32

return \_mkdir(path);

#else

return mkdir(path, mode);

#endif

}