PNGwriter Quick Reference Manual

Version 0.5.6 (December 2015)

http://pngwriter.sourceforge.net/

Introduction

This is the PNGwriter Quick Reference Manual. It is mostly a summary of the functions that PNGwriter provides, but also contains useful information on compilation and usage tips. Note that the `pngwriter.h` header file is also well commented. Be sure to look at the Examples section on the website, and to check the examples in `/usr/local/share/doc/pngwriter/`. Ultimately, if this documentation is insufficient or you suspect it may be mistaken, you should check the source code—most of it isn't that hard to understand!

Note: This document assumes that PNGwriter was installed in `/usr/local` (in `/usr/local/lib`, `/usr/local/share`, etc.). This is the default location. If you chose another installation location, then please keep this in mind as you read the Manual. This is important for knowing what header and library directories you must include at compile time, where the documentation and examples are, where the included fonts are, etc. For example, if you installed PNGwriter from the Debian package, it is installed under `/usr` (in `/usr/lib`, `/usr/share`, etc.). Or if when installing from source, you gave the command `make install PREFIX=$HOME` for example, where `$HOME` is the path to your home directory, you'll find PNGwriter's files installed under `$HOME/lib`, `$HOME/share`, etc.

Summary

PNGwriter is a very easy to use open source graphics library that uses PNG as its output format. The interface has been designed to be as simple and intuitive as possible. It supports plotting and reading in the RGB (red, green, blue), HSV (hue, saturation, value/brightness) and CMYK (cyan, magenta, yellow, black) colour spaces, basic shapes, scaling, bilinear interpolation, full TrueType antialiased and rotated text support, bezier curves, opening existing PNG images and more. Documentation in English and Spanish. Runs under Linux, Unix, Mac OS X and Windows. Requires libpng and optionally FreeType2 for the text support.

Contents

This document is divided into the following main sections:

- Text
- Image Size
- File and PNG-specific functions

Help and Support

If you have a problem, question or suggestion, you can post on the PNGwriter forums, since the mailing list is no longer in use. You can also email me directly.

PNGwriter's documentation consists of this PDF Manual, the README, the `pngwriter.h` header file, the two complete examples included with the source code, and of course, the website, which contains a Frequently Asked Questions section and quite a few interesting examples, among other things.

General Notes

Basic Usage

The most basic use of PNGwriter would be the following:

```
#include <pngwriter.h>

int main()
{
     pngwriter image(200, 300, 1.0, "out.png");
     image.plot(30, 40, 1.0, 0.0, 0.0);
     image.close();

     return 0;
}
```

This would plot a red dot (1 pixel) in the lower-left corner of the image, on a white background. The file created will be called "out.png", and will be 200 pixels wide and 300 pixels tall.

To get started, read about the constructor, plot(), dread() and close(). Explore the other functions when you feel confident with these. This is the most basic set of functions you will probably use.

It is important to remember that all functions that accept an argument of type "`const char *`" will also accept "`char *`". This is done so you can have a changing filename (to make many PNG images in series with a different name, for example), and to allow you to use `string` type objects which can be easily turned into `const char *` (if `theString` is an object of type `string`, then it can be used as a `const char *` by calling `theString.c_str()`).

It is also important to remember that whenever a function has a colour coeffiecient as its argument, that argument can be either an `int` from `0` to `65535` or a `double` from `0.0` to `1.0`. You must make sure that you are calling the function with the type that you want. Remember that `1` is an `int`, while `1.0` is a `double`, and will thus determine what version of the function will be used. Do not make the mistake of calling (for example) `plot(x, y, 0.0, 0.0, 65535)`, because there is no `plot(int, int, double, double, int)`. Also, please note that `plot()` and `read()` (and the functions that use them internally) are protected against entering, for example, a colour coefficient that is over `65535` or over `1.0`. Similarly, they are protected against negative coefficients. `read()` will return `0` when called outside the image range. This is actually useful as zero-padding should you need it.

Compilation

A typical compilation, assuming PNGwriter was installed in its default location (under `/usr/local`) would look like this:

```
g++ my_program.cc -o my_program `freetype-config --cflags`
     -I/usr/local/include  -L/usr/local/lib -lpng -lpngwriter -lz -lfreetype
```

Otherwise, replace `/usr/local` above with whatever location you chose at install time.
If you did not compile PNGwriter with FreeType support, then remove the FreeType-related flags and add
`-DNO_FREETYPE` above.

The website has additional information about compiling under Windows, though I cannot offer support for this.

Constructor

The constructor requires the width and the height of the image, the background colour for the image and the filename of the file (a pointer or simply "`myfile.png`"). The default constructor creates a PNGwriter instance that is `250x250`, white background, and filename `"out.png"`. **Tip**: The filename can be given as easily as: pngwriter `mypng(300, 300, 0.0, "myfile.png");` **Tip**: If you are going to create a PNGwriter instance for reading in a file that already exists, then width and height can be 1 pixel, and the size will be automatically adjusted once you use readfromfile().

```
pngwriter();
pngwriter(const pngwriter &rhs);
pngwriter(int width, int height, int backgroundcolour, char * filename);
pngwriter(    int width, int height, double backgroundcolour,
     char * filename);
pngwriter(    int width, int height, int backgroundcolour,
     const char * filename);
pngwriter(    int width, int height, double backgroundcolour,
     const char * filename);
```

Assignment Operator

PNGwriter overloads the assignment operator =.

```
pngwriter & operator = (const pngwriter & rhs);
```

Version Number

Returns the PNGwriter version number.

```
double version(void);
```

. . . . . Plotting . . . . . .

Plot

The pixels are numbered starting from (1, 1) and go to (width, height). If the colour coefficients are of type `int`, they go from `0` to `65535`. If they are of type `double`, they go from `0.0` to `1.0`. **Tip**: To plot using red, then specify `plot(x, y, 1.0, 0.0, 0.0)`. To make pink, just add a constant value to all three coefficients, like this: `plot(x, y, 1.0, 0.4, 0.4)`. **Tip**: If nothing is being plotted to your PNG file, make sure that you remember to close() the instance before your program is finished, and that the x and y position is actually within the bounds of your image. If either is not, then PNGwriter will not complain-- it is up to you to check for this! **Tip**: If you try to plot with a colour coefficient out of range, a maximum or minimum coefficient will be assumed, according to the given coefficient. For example, attempting to plot `plot(x, y, 0.5,-0.2,3.7)` will set the green coefficient to `0.0` and the blue coefficient to `1.0`.

```
void plot(int x, int y, int red, int green, int blue);
void plot(int x, int y, double red, double green, double blue);
```

Plot Blend

Plots the colour given by red, green blue, but blended with the existing pixel value at that position. opacity is a `double` that goes from `0.0` to `1.0`. For example, `0.0` will not change the pixel at all, and `1.0` will plot the given colour. Anything in between will be a blend of both pixel levels. **Please note**: This is neither alpha channel nor PNG transparency chunk support. This merely blends the plotted pixels.

```
void plot_blend(   int x, int y, double opacity,
                   int red, int green, int blue);
void plot_blend(   int x, int y, double opacity,
                   double red, double green,   double blue);
```

Blended Functions

All these functions are identical to their non-blended types. They take an extra argument, opacity, which is a double from 0.0 to 1.0 and represents how much of the original pixel value is retained when plotting the new pixel. In other words, if opacity is 0.7, then after plotting, the new pixel will be 30% of the original colour the pixel was, and 70% of the new colour, whatever that may be. As usual, each function is available in `int` or `double` versions. **Please note**: This is neither alpha channel nor PNG transparency chunk support. This merely blends the plotted pixels.

```
void plotHSV_blend(     int x, int y, double opacity,
                        double hue, double saturation, double value);
void plotHSV_blend(     int x, int y, double opacity,
                        int hue, int saturation, int value);

void line_blend(   int xfrom, int yfrom, int xto, int yto,
                   double opacity, int red, int green,int  blue);
void line_blend(   int xfrom, int yfrom, int xto, int yto,
                   double opacity, double red, double green,double  blue);

void square_blend( int xfrom, int yfrom, int xto, int yto,
                   double opacity, int red, int green,int  blue);
```

```
void square_blend( int xfrom, int yfrom, int xto, int yto,
                   double opacity, double red, double green, double  blue);

void filledsquare_blend(    int xfrom, int yfrom, int xto, int yto,
                            double opacity,
                            int red, int green,int  blue);
void filledsquare_blend(    int xfrom, int yfrom, int xto, int yto,
                            double opacity,
                            double red, double green,double  blue);



void circle_blend( int xcentre, int ycentre, int radius,
                   double opacity, int red, int green, int blue);
void circle_blend( int xcentre, int ycentre, int radius,
                   double opacity, double red, double green, double blue);

void filledcircle_blend(    int xcentre, int ycentre, int radius,
                            double opacity,
                            int red, int green, int blue);
void filledcircle_blend(    int xcentre, int ycentre, int radius,
                            double opacity,
                            double red, double green, double blue);
void bezier_blend( int startPtX, int startPtY,
                   int startControlX, int startControlY,
                   int endPtX, int endPtY,
                   int endControlX, int endControlY,
                   double opacity,
                   double red, double green, double blue);

void bezier_blend( int startPtX, int startPtY,
                   int startControlX, int startControlY,
                   int endPtX, int endPtY,
                   int endControlX, int endControlY,
                   double opacity,
                   int red, int green, int blue);

void plot_text_blend(  char * face_path, int fontsize,
                       int x_start, int y_start, double angle, char * text,
                       double opacity,
                       double red, double green, double blue);
void plot_text_blend(  char * face_path, int fontsize,
                       int x_start, int y_start, double angle, char * text,
                       double opacity,
                       int red, int green, int blue);

void plot_text_utf8_blend(char * face_path, int fontsize,
                          int x_start, int y_start, double angle, char * text,
                          double opacity,
                          double red, double green, double blue);
void plot_text_utf8_blend(char * face_path, int fontsize,
                          int x_start, int y_start, double angle, char * text,
                          double opacity,
                          int red, int green, int blue);
```

```
void boundary_fill_blend(int xstart, int ystart,
         double opacity,
         double boundary_red, double boundary_green, double boundary_blue,
         double fill_red, double fill_green, double fill_blue);
void boundary_fill_blend(int xstart, int ystart,
         double opacity,
         int boundary_red, int boundary_green, int boundary_blue,
         int fill_red, int fill_green, int fill_blue);

void flood_fill_blend( int xstart, int ystart, double opacity,
                       double fill_red, double fill_green, double fill_blue);
void flood_fill_blend( int xstart, int ystart, double opacity,
                       int fill_red, int fill_green, int fill_blue);

void polygon_blend(int * points, int number_of_points, double opacity,
                       double red, double green, double blue);
void polygon_blend(int * points, int number_of_points, double opacity,
                       int red, int green, int blue);
void plotCMYK_blend(int x, int y, double opacity,
                  double cyan, double magenta, double yellow, double black);
void plotCMYK_blend(int x, int y, double opacity,
                  int cyan, int magenta, int yellow, int black);
```

Plot HSV

With this function a pixel at coordinates `(x, y)` can be set to the desired colour, but with the colour coefficients given in the Hue, Saturation, Value colourspace.

```
void plotHSV(int x, int y, double hue, double saturation, double value);
void plotHSV(int x, int y, int hue, int saturation, int value);
```

Plot CMYK

Plot a point in the Cyan, Magenta, Yellow, Black colourspace. Please note that this colourspace is lossy, i.e. it cannot reproduce all colours on screen that RGB can. The difference, however, is barely noticeable. The algorithm used is a standard one. The colour components are either doubles from `0.0` to `1.0` or ints from `0` to `65535`.

```
void plotCMYK(     int x, int y,
                   double cyan, double magenta, double yellow, double black);
void plotCMYK(     int x, int y,
                   int cyan, int magenta, int yellow, int black);
```

Clear

The whole image is set to black.

```
void clear(void);
```

Invert

Inverts the image in RGB colourspace.

```
void invert(void);
```

Boundary Fill

All pixels adjacent to the start pixel will be filled with the fill colour, until the boundary colour is encountered. For example, calling `boundary_fill()` with the boundary colour set to red, on a pixel somewhere inside a red circle, will fill the entire circle with the desired fill colour. If, on the other hand, the circle is not the boundary colour, the rest of the image will be filled. The colour components are either `double` from `0.0` to `1.0` or `int` from `0` to `65535`.

```
void boundary_fill(int xstart, int ystart,
         double boundary_red, double boundary_green, double boundary_blue,
         double fill_red, double fill_green, double fill_blue);
void boundary_fill( int xstart, int ystart,
                     int boundary_red, int boundary_green, int boundary_blue,
                     int fill_red, int fill_green, int fill_blue) ;
```

Flood Fill

All pixels adjacent to the start pixel will be filled with the fill colour, if they are the same colour as the start pixel. For example, calling `flood_fill()` somewhere in the interior of a solid blue rectangle will colour the entire rectangle the fill colour. The colour components are either `double` from `0.0` to `1.0` or `int` from `0` to `65535`.

```
void flood_fill(   int xstart, int ystart,
                   double fill_red, double fill_green, double fill_blue);
void flood_fill(   int xstart, int ystart,
                   int fill_red, int fill_green, int fill_blue) ;
```

Laplacian

This function applies a discrete laplacian to the image, multiplied by a constant factor. The kernel used in this case is:

```
1.0  1.0  1.0
1.0 -8.0  1.0
1.0  1.0  1.0
```

Basically, this works as an edge detector. The current pixel is assigned the sum of all neighbouring
pixels, multiplied by the corresponding kernel element. For example, imagine a pixel and its 8 neighbours:

```
1.0      1.0      1.0      0.0      0.0
1.0      1.0    ->1.0<-    0.0      0.0
1.0      1.0      1.0      0.0      0.0
```

This represents a border between white and black, black is on the right. Applying the laplacian to he pixel specified above pixel gives:

```
1.0*1.0   +  1.0*1.0    +  0.0*1.0   +
1.0*1.0   +  1.0*-8.0   +  0.0*1.0   +
1.0*1.0   +  1.0*1.0    +  0.0*1.0   =  -3.0
```

Applying this to the pixel to the right of the pixel considered previously, we get a sum of `3.0`. That is, after passing over an edge, we get a high value for the pixel adjacent to the edge. Since PNGwriter limits the colour components if they are off-scale, and the result of the laplacian may be negative, a scale factor and an offset value are included. This might be

useful for keeping things within range or for bringing out more detail in the edge detection. The final pixel value will be given by:

```
final value =  laplacian(original pixel)*k + offset
```

Tip: Try a value of 1.0 for k to start with, and then experiment with other values. **Tip**: It would be difficult to foresee all possible uses of this function, so please feel free to look at the source code and perhaps implement your own if it does not suit your needs.

```
void laplacian(double k, double offset);
```

. . . . . Reading . . . . . .

Read
With this function we find out what colour the pixel `(x, y)` is. If "`colour`" is `1`, it will return the red coefficient, if it is set to `2`, the green one, and if it set to `3`, the blue colour coefficient will be returned, and this returned value will be of type `int` and be between 0 and 65535.

```
int read(int x, int y, int colour);
```

Read, Average
Same as the above, only that the average of the three colour coefficients is returned.

```
int read(int x, int y);
```

dRead
Same as `read()`, but the returned value will be of type `double` and be between `0.0` and `1.0`.

```
double dread(int x, int y, int colour);
```

dRead, Average
Same as the above, only that the average of the three colour coefficients is returned.

```
double dread(int x, int y);
```

Bilinear Interpolation of Image
Given a floating point coordinate (`x` from 0.0 to width, `y` from 0.0 to height), this function will return the interpolated colour intensity specified by colour (where red = `1`, green = `2`, blue = `3`). `bilinear_interpolate_read()` returns an `int` from 0 to 65535, and `bilinear_interpolate_dread()` returns a `double` from 0.0 to 1.0. **Tip**: Especially useful for enlarging an image.

```
int bilinear_interpolation_read(double x, double y, int colour);
double bilinear_interpolation_dread(double x, double y, int colour);
```

Read HSV
With this function we find out what colour the pixel (x, y) is, but in the Hue, Saturation, Value colourspace. If "colour" is `1`, it will return the Hue coefficient, if it is set to `2`, the Saturation one, and if it set to `3`, the Value colour coefficient will be returned, and this returned value will be of type int and be between 0 and 65535. **Important**: If you attempt to read the Hue of a pixel that is a shade of grey, the value returned will be nonsensical or even NaN. This is just the way the RGB -> HSV algorithm works: the Hue of grey is not defined. You might want to check whether the pixel you are reading is grey before attempting a readHSV().

```
int readHSV(int x, int y, int colour);
```

dRead HSV

Same as the above, but the returned value will be of type double and be between `0.0` and `1.0`.

```
double dreadHSV(int x, int y, int colour);
```

Read CMYK, double version

Get a pixel in the Cyan, Magenta, Yellow, Black colourspace. if 'colour' is `1`, the Cyan component will be returned as a double from `0.0` to `1.0`. If 'colour is `2`, the Magenta colour component will be returned, and so on, up to `4`.

```
double dreadCMYK(int x, int y, int colour);
```

Read CMYK

Same as the above, but the colour components returned are an int from `0` to `65535`.

```
int readCMYK(int x, int y, int colour);
```

Figures

These functions draw basic shapes. Available in both `int` and `double` versions.

Line

The line functions use the fast Bresenham algorithm.

```
void line(    int xfrom, int yfrom,
              int xto, int yto,
              int red, int green, int blue);
void line(    int xfrom, int yfrom,
              int xto, int yto,
              double red, double green,double blue);
```

Triangle

Draws a triangle specified by the three pairs of points `(x1, y1)`, `(x2, y2)`, `(x3, y3)`. The colour components are either `double`s from 0.0 to 1.0 or `int`s from `0` to `65535`.

```
void triangle(     int x1, int y1, int x2, int y2, int x3, int y3,
                   int red, int green, int blue);
void triangle(     int x1, int y1, int x2, int y2, int x3, int y3,
                   double red, double green, double blue);
```

Filled Triangle, and Filled Triangle, Blended

Draws the triangle specified by the three pairs of points in the colour specified by the colour coefficients. `filledtriangle_blend()` does the same, but blended with the background. See the description for Blended Functions. The colour components are either `double`s from 0.0 to 1.0 or `int`s from `0` to `65535`.

```
void filledtriangle(    int x1,int y1,int x2,int y2,int x3,int y3,
                        int red, int green, int blue);
void filledtriangle(    int x1,int y1,int x2,int y2,int x3,int y3,
                        double red, double green, double blue);
void filledtriangle_blend(  int x1,int y1,int x2,int y2,int x3,int y3,
                   double opacity, int red, int green, int blue);
void filledtriangle_blend(  int x1,int y1,int x2,int y2,int x3,int y3,
                   double opacity, double red, double green, double blue);
```

Square

Despite the name, these functions draw rectangles.

```
void square(  int xfrom, int yfrom,
              int xto, int yto,
              int red, int green, int blue);
void square(  int xfrom, int yfrom,
              int xto, int yto,
              double red, double green, double blue);
void filledsquare( int xfrom, int yfrom,
                   int xto, int yto,
```

```
                  int red, int green,int blue);
void filledsquare( int xfrom, int yfrom,
                  int xto, int yto,
                  double red, double green,double blue);
```

Circle

The circle functions use a fast integer math algorithm. The filled circle functions make use of sqrt().

```
void circle(  int xcentre, int ycentre, int radius,
              int red, int green, int blue);
void circle(  int xcentre, int ycentre, int radius,
              double red, double green, double blue);
void filledcircle( int xcentre, int ycentre, int radius,
                  int red, int green, int blue);
void filledcircle( int xcentre, int ycentre, int radius,
                  double red, double green, double blue);
```

Bezier Curve

(After Frenchman Pierre Bézier from Regie Renault) A collection of formulae for describing curved lines and surfaces, first used in 1972 to model automobile surfaces. (from the The Free On-line Dictionary of Computing) See http://www.moshplant.com/direct-or/bezier/ for one of many available descriptions of bezier curves. There are four points used to define the curve: the two endpoints of the curve are called the anchor points, while the other points, which define the actual curvature, are called handles or control points. Moving the handles lets you modify the shape of the curve.

```
void bezier(  int startPtX, int startPtY,
              int startControlX, int startControlY, int endPtX, int endPtY,
              int endControlX, int endControlY,
              double red, double green, double blue);


void bezier(  int startPtX, int startPtY,
              int startControlX, int startControlY, int endPtX, int endPtY,
              int endControlX, int endControlY,
              int red, int green, int blue);
```

Polygon

This function takes an array of integer values containing the coordinates of the vertexes of a polygon. Note that if you want a closed polygon, you must repeat the first point's coordinates for the last point. It also requires the number of points contained in the array. For example, if you wish to plot a triangle, the array will contain `6` elements, and the number of points is `3`. Be very careful about this; if you specify the wrong number of points, your program will either segfault or produce points at nonsensical coordinates. The colour components are either `double`s from `0.0` to `1.0` or `int`s from `0` to `65535`.

```
void polygon( int * points, int number_of_points,
              double red, double green, double blue);
void polygon( int * points, int number_of_points,
```

```
              int red, int green, int blue);
```

Arrow, Filled Arrow

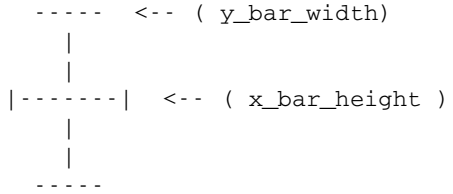
Plots an arrow from `(x1, y1)` to `(x2, y2)` with the arrowhead at the second point, given the size in pixels and the angle in radians of the arrowhead. The plotted arrow consists of one main line, and two smaller lines originating from the second point. Filled Arrow plots the same, but the arrowhead is a solid triangle. **Tip**: An angle of `10` to `30` degrees (`0.1745` to `0.5236` radians) looks OK.

```
void arrow( int x1, int y1, int x2, int y2,
         int size, double head_angle, double red, double green, double blue);
void arrow( int x1, int y1, int x2, int y2,
         int size, double head_angle, int red, int green, int blue);

void filledarrow(  int x1, int y1, int x2, int y2,
                   int size, double head_angle,
                   double red, double green, double blue);
void filledarrow(  int x1, int y1, int x2, int y2,
                   int size, double head_angle,
                   int red, int green, int blue);
```

Cross, Maltese Cross

Plots a simple cross at `(x, y)`, with the specified height and width, and in the specified colour. Maltese cross plots a cross, as before, but adds bars at the end of each arm of the cross. The size of these bars is specified with `x_bar_height` and `y_bar_width`. The cross will look something like this:




```
void cross(   int x, int y, int xwidth, int yheight,
              double red, double green, double blue);
void cross(   int x, int y, int xwidth, int yheight,
              int red, int green, int blue);

void maltesecross( int x, int y, int xwidth, int yheight,
                   int x_bar_height, int y_bar_width,
                   double red, double green, double blue);
void maltesecross( int x, int y, int xwidth, int yheight,
                   int x_bar_height, int y_bar_width,
                   int red, int green, int blue);
```

Diamond, Filled Diamond

Plots a diamond shape, given the (`x`, `y`) position, the width and height, and the colour. Filled diamond plots a filled diamond.

```
void diamond( int x, int y, int width, int height,
              int red, int green, int blue);
void diamond( int x, int y, int width, int height,
              double red, double green, double blue);

void filleddiamond(    int x, int y, int width, int height,
                       int red, int green, int blue);
void filleddiamond(    int x, int y, int width, int height,
                       double red, double green, double blue);
```

Text

Plot Text

Uses the Freetype2 library to plot text in the image. `face_path` is the file path to a TrueType font file (`.ttf`) (FreeType2 can also handle other types). `fontsize` specifices the approximate height of the rendered font in pixels. `x_start` and `y_start` specify the placement of the lower, left corner of the text string. `angle` is the text angle in radians. `text` is the text to be rendered. The colour coordinates can be `doubles` from 0.0 to 1.0 or `ints` from 0 to 65535. **Tip**: PNGwriter installs a few fonts in `/usr/local/share/pngwriter/fonts` to get you started. **Tip**: Remember to add `-DNO_FREETYPE` to your compilation flags if PNGwriter was compiled without FreeType support.

```
void plot_text(    char * face_path, int fontsize,
                   int x_start, int y_start,   double angle,
                   char * text,
                   double red, double green, double blue);
void plot_text(    char * face_path, int fontsize,
                   int x_start, int y_start, double angle,
                   char * text,
                   int red, int green, int blue);
```

Plot UTF-8 Text

Same as the above, but the text to be plotted is encoded in UTF-8. Why would you want this? To be able to plot all characters available in a large TrueType font, for example: for rendering Japenese, Chinese and other languages not restricted to the standard 128 character ASCII space. **Tip**: One quick way to get a string into UTF-8 is to write it in an adequate text editor, and save it as a file in UTF-8 encoding, which can then be read in in binary mode.

```
void plot_text_utf8(    char * face_path, int fontsize,
                        int x_start, int y_start, double angle,
                        char * text,
                        double red, double green, double blue);
void plot_text_utf8(    char * face_path, int fontsize,
                        int x_start, int y_start, double angle,
                        char * text,
                        int red, int green, int blue);
```

Get Text Width, Get Text Width UTF8

Returns the approximate width, in pixels, of the specified *unrotated* text. It is calculated by adding each letter's width and kerning value (as specified in the TTF file). Note that this will not give the position of the farthest pixel, but it will give a pretty good idea of what area the text will occupy. **Tip**: The text, when plotted unrotated, will fit approximately in a box with its lower left corner at `(x_start, y_start)` and upper right at `(x_start + width, y_start + size)`, where width is given by `get_text_width()` and size is the specified size of the text to be plotted. **Tip**: Text plotted at position `(x_start, y_start)`, rotated with a given angle '`th`', and of a given '`size`' whose width is '`width`', will fit approximately inside a rectangle whose corners are at

```
(x_start, y_start)
(x_start + width*cos(th), y_start + width*sin(th))
(x_start + width*cos(th) - size*sin(th),y_start+ width*sin(th)+size*cos(th))
```

```
     (x_start - size*sin(th), y_start + size*cos(th))

int get_text_width(char * face_path, int fontsize,  char * text);
int get_text_width_utf8(char * face_path, int fontsize, char * text);
```

Image Size

Scale Proportional

Scale the image using bilinear interpolation. If `k` is greater than 1.0, the image will be enlarged. If `k` is less than 1.0, the image will be shrunk. Negative or null values of `k` are not allowed. The image will be resized and the previous content will be replaced by the scaled image. **Tip**: Use `getheight()` and `getwidth()` to find out the new width and height of the scaled image. **Note**: After scaling, all images will have a bit depth of 16, even if the original image had a bit depth of 8.

```
void scale_k(double k);
```

Scale Non-Proportional

Scale the image using bilinear interpolation, with different horizontal and vertical scale factors.

```
void scale_kxky(double kx, double ky);
```

Scale to Final Width and Height

Scale the image in such a way as to meet the target width and height. **Tip**: If you want to keep the image proportional, scale_k() might be more appropriate.

```
void scale_wh(int finalwidth, int finalheight);
```

Resize Image

Resizes the PNGwriter instance. Note: All image data is set to black (this is a resizing, not a scaling, of the image).

```
void resize(int width, int height);
```

File and PNG-Specific Functions

Read From File

Opens the existing PNG image, and copy it into this instance of the class. It is important to mention that only some PNG variants are supported. Very generally speaking, most PNG files can now be read (as of version 0.5.4), but if they have an alpha channel it will be completely stripped. If the PNG file uses GIF-style transparency (where one colour is chosen to be transparent), PNGwriter will not read the image properly, but will not complain. Also, if any ancillary chunks are included in the PNG file (chroma, filter, etc.), it will render with a slightly different tonality. For the vast majority of PNGs, this should not be an issue. **Note**: If you read an 8-bit PNG, the internal representation of that instance of PNGwriter will be 8-bit (PNG files of less than 8 bits will be upscaled to 8 bits). To convert it to 16-bit, just loop over all pixels, reading them into a new instance of PNGwriter. New instances of PNGwriter are 16-bit by default.

```
void readfromfile(char * name);
void readfromfile(const char * name);
```

Close

Close the instance of the class, and write the image to disk.

```
void close(void);
```

Rename

To rename the file once an instance of PNGwriter has been created. If the argument is a `long unsigned int`, for example `77`, the filename will be changed to `0000000077.png`

```
void pngwriter_rename(char * newname);
void pngwriter_rename(const char * newname);
void pngwriter_rename(long unsigned int index);
```

Get Height

When you open a PNG with `readfromfile()` you can find out its height with this function.

```
int getheight(void);
```

Get Width

When you open a PNG with `readfromfile()` you can find out its width with this function.

```
int getwidth(void);
```

Write PNG

Writes the PNG image to disk. You can still change the PNGwriter instance after this.

```
void write_png(void);
```

Set Compression Level
Set the compression level that will be used for the image. `-1` is default, `0` is none, `9` is best compression.

```
void setcompressionlevel(int level);
```

Get Bit Depth
When you open a PNG with `readfromfile()` you can find out its bit depth with this function.

```
int getbitdepth(void);
```

Get Colour Type
When you open a PNG with `readfromfile()` you can find out its colour type.

```
int getcolortype(void);
```

Set Gamma Coefficient
Set the image's gamma (file gamma) coefficient. The default value of `0.5` should be fine.

```
void setgamma(double gamma);
```

Get Gamma Coefficient
Get the image's gamma coefficient. This is experimental.

```
double getgamma(void);
```

Set Text
Sets the text information in the PNG header. If it is not called, the default is used.

```
void settext( char * title, char * author,
              char * description, char * software);
void settext( const char * title, const char * author,
              const char * description, const char * software);
```