1. Introduction.

COLOUR

Colour System Library

John Walker

This program provides tools for representing colours, interconverting colour spaces, and transforming physical definitions of colour (for example, spectra or black body emission) into perceptual colour metrics such as the CIE tristimulus values, and thence to computer approximations such as RGB values for various phosphors.

This program is not presently used by the analysis suite; it is only exercised by its own built-in test program. It is provided as part of the eventual goal of supporting internal plotting without the need to invoke GNUPLOT, and also facilitate visual presentation of data in the form of colour (for example, one might wish to express a z score as a colour representing how "hot" it was, expressing z as a black body temperature).

References

Hunt, R.W.G. *Measuring Colour.* West Sussex England: Ellis Horwood Ltd., 1987. (Distributed in the U.S. by John Wiley **S**ons). ISBN 0-470-20986-0.

Adobe Systems, Inc. *PostScript Language Reference Manual, 3rd ed.* Reading Massachusetts: Addison-Wesley, 1999. ISBN 0-201-37922-8.

Rossotti, Hazel. *Colour: Why the World Isn't Grey.* Princeton: Princeton University Press, 1983. ISBN 0-691-02386-7.

Judd, Deane B. and Günter Wyszecki. *Color in Business, Science, and Industry.* New York: John Wiley **S**ons, 1975.

Arvo, James ed. *Graphics Gems II.* San Diego: Academic Press, Inc., 1991, section III.6, "Television Color Encoding". ISBN 0-12-064480-0.

Foley, J.D., A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics: Principles and Practice, 2nd ed. in C.* Reading Massachusetts: Addison-Wesley, 1996. ISBN 0-201-84840-6.

Pantone, Inc. *PANTONE Process Color Imaging Guide.* Moonachie NJ: Pantone, Inc., 1990.

⟨ colour_test.c 1 ⟩ ≡

```
#define REVDATE "1st␣February␣2002"
```

See also section 40.

2. Program global context.

```
#include "config.h"     /* System-dependent configuration */
  ⟨ Preprocessor definitions ⟩
  ⟨ Application include files 4 ⟩
  ⟨ Colour system data tables 37 ⟩
  ⟨ Class implementations 5 ⟩
```

3. We export the class definitions for this package in the external file `colour.h` that programs which use this library may include.

```
⟨ colour.h 3 ⟩ ≡
#ifndef COLOUR_HEADER_DEFINES
#define COLOUR_HEADER_DEFINES
#include <math.h>     /* Make sure math.h is available */
#include <iostream>
#include <exception>
#include <stdexcept>
#include <string>
#include <vector>
#include <algorithm>
  using namespace std;
#include "bitmap.h"
#include <stdio.h>     /* Needed to define FILE for gifout.h */
#include "gifout.h"
#include "graphics.h"
#include "pstampr.h"
#include "psrtext.h"
  ⟨ Class definitions 6 ⟩
#endif
```

4. The following include files provide access to external components of the program not defined herein.

```
⟨ Application include files 4 ⟩ ≡
#include "colour.h"     /* Class definitions for this package */
```

This code is used in section 2.

5. The following classes are defined and their implementations provided.

```
⟨ Class implementations 5 ⟩ ≡
  ⟨ Colour systems 17 ⟩
```

This code is used in section 2.

6. Colour system parent class: csColour.
This is the parent class of all colour classes.

```
⟨ Class definitions 6 ⟩ ≡
  class csColour {
  public:
    virtual string colourSystemName(void) = 0;      /* Return colour system name */
    virtual void writeParameters(ostream &of)
    {      /* Write description of colour system to output stream */
      of ≪ "Colour␣system:␣" ≪ colourSystemName( ) ≪ "\n";
    }
  };
```

See also sections 7, 8, 9, 10, 11, 12, and 33.

This code is used in section 3.

7. Spectral colour systems parent class: csSpectralColour.

The most general of all our notions of colour is that of an function that maps wavelengths (given in metres) onto intensities. This form allows specification of any radiation in the electromagnetic spectrum. Intensities are normalised to the range zero to one by dividing the intensity for a given frequency by the integrated intensity over the entire electromagnetic spectrum.

```
⟨ Class definitions 6 ⟩ +≡
  class csSpectralColour : public csColour {
  public:     /* Return normalised intensity for a given wavelength in metres */
    virtual double getIntensity(double waveLength) = 0;
      /* Return true if this is a pure (monochromatic) radiator, false */     /* otherwise. */
    virtual bool isMonochromatic(void) = 0;
  };
```

8. Monochromatic spectral colour systems: csMonochromaticColour.

A monochromatic colour is a abstract notion of a source which radiates all its intensity at a single wavelength. These don't really exist in the real world, since fundamental quantum processes will always spread the spectrum of any real radiator.

```
⟨ Class definitions 6 ⟩ +≡
  class csMonochromaticColour : public csSpectralColour {
  private:
    double wavelength;
  public:
    virtual string colourSystemName(void)
    {
      return "monochromatic␣spectral";
    }
    double getIntensity(double waveLength)
    {
      return (waveLength ≡ wavelength) ? 1.0 : 0.0;
    }
    bool isMonochromatic(void)
    {
      return true;
    }     /* Constructors and destructors */
    csMonochromaticColour(void)
    {
      wavelength = 0.0;
    }
    csMonochromaticColour(double waveLength)
    {
      wavelength = waveLength;
    }     /* Class-specific methods */
    void setWavelength(double waveLength)
    {
      wavelength = waveLength;
    }
    double getWavelength(void)
    {
      return wavelength;
    }
    void writeParameters(ostream &of)
    {
      csSpectralColour::writeParameters(of);
      of ≪ "␣␣␣␣Wavelength␣=␣" ≪ getWavelength( ) ≪ "\n";
    }
  };
```

9. Black: csSpectralBlack. Black is the absence of colour. We define it as a spectral colour with zero intensity at any wavelength.

```
⟨ Class definitions 6 ⟩ +≡
  class csSpectralBlack : public csSpectralColour {
  public:
    virtual string colourSystemName(void)
    {
      return "black␣spectral";
    }
    double getIntensity(double waveLength)
    {
      return 0.0;
    }
    bool isMonochromatic(void)
    {
      return false;
    }
  };
```

10. White: csSpectralWhite. A theoretical source of white noise has equal power at all wavelengths. Such a source is impossible in reality since the energy flux would be infinite.

```
⟨ Class definitions 6 ⟩ +≡
  class csSpectralWhite : public csSpectralColour {
  public:
    virtual string colourSystemName(void)
    {
      return "white␣spectral";
    }
    double getIntensity(double waveLength)
    {
      return 1.0;
    }
    bool isMonochromatic(void)
    {
      return false;
    }
  };
```

11. Planckian black body: csBlackBody. Define the colour of a Planckian black body radiator with a given colour temperature.

```
⟨ Class definitions 6 ⟩ +≡
  class csBlackBody : public csSpectralColour {
  private:    /* Change temperature to class with system conversions */
    double temperature;    /* Temperature of radiator (°K) */
  public:
    double getIntensity(double waveLength)
    {
      return 0.0;
    }
    bool isMonochromatic(void)
    {
      return false;
    }    /* Constructors and destructors */
    csBlackBody(void)
    {
      temperature = 0.0;
    }
    csBlackBody(double temp)
    {
      temperature = temp;
    }    /* Class-specific methods */
    void setTemperature(double temp)
    {
      temperature = temp;
    }
    double getTemperature(void)
    {
      return temperature;
    }
    double totalFlux(void)    /* Total energy flux in W/m^2 */
    {    /* Total energy flux in Watts per square metre of a black body with temperature T (degrees
        Kelvin) is given by:
                                        C_s T^4
        where C_s is the Stefan-Boltzman constant:
                                    5.67051 × 10^-8 W/(m^2 K^4)
        which, in fundamental terms is:
                                    (π^2 k^4)/(60(h/(2π))^3 c^2)
          */
      return 5.67051 · 10^-8 * (temperature * temperature * temperature * temperature);
    }
    double flux(double wl)    /* Energy flux at a given wavelength */
    {    /* Energy flux from a black body with temperature T (degrees Kelvin), in Watts per square
        metre at wavelength λ (metres) is given by:
                                    M_e = C_1 λ^-5 (e^(C_2/(λ*T)) − 1)^-1
```

where:

$$C_1 = 3.74183e - 16Wm^2$$

$$C_2 = 1.4388e - 2m^\circ K$$

```
    */
  return 3.74183 · 10^-16/((wl * wl * wl * wl * wl) * (exp(1.4388 · 10^-2/(wl * temperature)) - 1));
}
void writeParameters(ostream &of)
{
  csSpectralColour::writeParameters(of);
  of ≪ "␣␣␣␣Temperature␣=␣" ≪ getTemperature( ) ≪ "␣K\n";
}
};
```

12. Device colour systems parent class: csDeviceColour. This class is the parent of all classes which specify colour in a device-specific way, assuming a particular set of illuminants which are summed to form the colour.

```
⟨ Class definitions 6 ⟩ +≡
  class csDeviceColour : public csColour {
  public:
    ⟨ Device colour fundamental methods 13 ⟩
    ⟨ Device colour derived methods 14 ⟩
    ⟨ Device colour system conversion utilities 15 ⟩
  };
```

13. The following three must-implement methods allow retrieving the colour in any of the three fundamental device colour spaces: RGB (additive), CMYK (subtractive), or Greyscale. Typically, a specific device colour class will store the colour in one of these forms and implement the other retrieval methods by converting the colour representation to that form.

```
⟨ Device colour fundamental methods 13 ⟩ ≡
  virtual void asRGB(double &r, double &g, double &b) = 0;
  virtual void asCMYK(double &c, double &m, double &y, double &k) = 0;
  virtual void asGreyScale(double &g) = 0;
```

This code is used in section 12.

14. The following methods allow retrieval of a device colour in other colour mapping spaces. The **csDeviceColour** class implements methods for these functions which provide default definitions which use the subclass *asRGB* method to obtain RGB components which are the converted into the requested colour space.

```
⟨ Device colour derived methods 14 ⟩ ≡
  virtual void asHSV(double &h, double &s, double &v);
  virtual void asHLS(double &h, double &l, double &s);
  virtual void asYIQ(double &y, double &i, double &q);
  virtual void asYUV(double &y, double &u, double &v);
  virtual void asSMPTE(double &y, double &Pb, double &Pr);
  virtual void asCMY(double &c, double &m, double &y);
```

This code is used in section 12.

15. The following **static** methods of the **csDeviceColour** class provide interconversions of common colour systems using the convention that colour components c are **double** values $0 \le c \le 1$.

⟨ Device colour system conversion utilities 15 ⟩ ≡

```
protected:
  static double hlsval(double n1, double n2, double hue);
public:
  static void hsv_rgb(double h, double s, double v, double *r, double *g, double *b);
    /* HSV → RGB */
  static void rgb_hsv(double r, double g, double b, double *h, double *s, double *v);
    /* RGB → HSV */
  static void rgb_hls(double r, double g, double b, double *h, double *l, double *s);
    /* RGB → HLS */
  static void hls_rgb(double h, double l, double s, double *r, double *g, double *b);
    /* HLS → RGB */
  static void rgb_yiq(double r, double g, double b, double *y, double *i, double *q);
    /* RGB → YIQ */
  static void yiq_rgb(double y, double i, double q, double *r, double *g, double *b);
    /* YIQ → RGB */
  static void rgb_yuv(double r, double g, double b, double *y, double *u, double *v);
    /* RGB → YUV */
  static void yuv_rgb(double y, double u, double v, double *r, double *g, double *b);
    /* YUV → RGB */
  static void rgb_smpte_204M(double r, double g, double b, double *y, double *Pb, double *Pr);
    /* RGB → SMPTE 204M */
  static void smpte_204M_rgb(double y, double Pb, double Pr, double *r, double *g, double *b);
    /* SMPTE 204M → RGB */
  static void rgb_cmy(double r, double g, double b, double *c, double *m, double *y);
    /* RGB → CMY */
  static void cmy_rgb(double c, double m, double y, double *r, double *g, double *b);
    /* CMY → RGB */
  static void cmy_cmyk(double c, double m, double y, double *oc, double *om, double *oy, double
      *ok);      /* CMY → CMYK */
  static void cmyk_cmy(double c, double m, double y, double k, double *oc, double *om, double
      *oy);      /* CMYK → CMY */
```

This code is used in section 12.

16. Colour system conversion utilities. The following **static** methods of the **csDeviceColour** class implement interconversions of common colour systems using the convention that colour components c are **double** values $0 \le c \le 1$.

17. `HSV_RGB`: Convert HSV colour specification to RGB intensities. Hue (h) is specified as a **double** value from 0 to 360°, Saturation (s) and Intensity (v) as **double**s from 0 to 1. The (r, g, b) components are returned as **double**s from 0 to 1.

```
⟨ Colour systems 17 ⟩ ≡
  void csDeviceColour::hsv_rgb(double h, double s, double v, double *r, double *g, double *b)
  {
    int i;
    double f, p, q, t;
    if (s ≡ 0) {
      *r = *g = *b = v;
    }
    else {
      if (h ≡ 360.0) h = 0;
      h /= 60.0;
      i = (int) h;
      f = h − i;
      p = v * (1.0 − s);
      q = v * (1.0 − (s * f));
      t = v * (1.0 − (s * (1.0 − f)));
      assert(i ≥ 0 ∧ i ≤ 5);
      switch (i) {
      case 0: *r = v;
        *g = t;
        *b = p;
        break;
      case 1: *r = q;
        *g = v;
        *b = p;
        break;
      case 2: *r = p;
        *g = v;
        *b = t;
        break;
      case 3: *r = p;
        *g = q;
        *b = v;
        break;
      case 4: *r = t;
        *g = p;
        *b = v;
        break;
      case 5: *r = v;
        *g = p;
        *b = q;
        break;
      }
    }
  }
```

See also sections 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 34, 35, and 36.

This code is used in section 5.

18. `RGB_HSV`: Map r, g, b intensities in the range from 0 to 1 into Hue (h), Saturation (s), and Value (v): Hue from 0 to 360°, Saturation from 0 to 1, and Value from 0 to 1. Special case: if Saturation is 0 (it's a grey scale tone), Hue is undefined and is returned as -1.

This follows Foley *et al.*, section 13.3.4.

```
⟨ Colour systems 17 ⟩ +≡
  void csDeviceColour::rgb_hsv(double r, double g, double b, double *h, double *s, double *v)
  {
    double imax = max(r, max(g, b)), imin = min(r, min(g, b)), rc, gc, bc;
    *v = imax;
    if (imax ≠ 0) *s = (imax − imin)/imax;
    else *s = 0;
    if (*s ≡ 0) {
      *h = −1;
    }
    else {
      rc = (imax − r)/(imax − imin);
      gc = (imax − g)/(imax − imin);
      bc = (imax − b)/(imax − imin);
      if (r ≡ imax) *h = bc − gc;
      else if (g ≡ imax) *h = 2.0 + rc − bc;
      else *h = 4.0 + gc − rc;
      *h *= 60.0;
      if (*h < 0.0) *h += 360.0;
    }
  }
```

19. `RGB_HLS`: Map r, g, b intensities in the range from 0 to 1 into Hue (h), Lightness (l), and Saturation (s): Hue from 0 to 360°, Lightness from 0 to 1, and Saturation from 0 to 1. Special case: if Saturation is 0 (it's a grey scale tone), Hue is undefined and is returned as -1.

This follows Foley *et al.*, section 13.3.5.

```
⟨ Colour systems 17 ⟩ +≡
  void csDeviceColour::rgb_hls(double r, double g, double b, double *h, double *l, double *s)
  {
    double imax = max(r, max(g, b)), imin = min(r, min(g, b)), rc, gc, bc;
    *l = (imax + imin)/2;
    if (imax ≡ imin) {
      *s = 0;
      *h = −1;
    }
    else {
      if (*l ≤ 0.5) *s = (imax − imin)/(imax + imin);
      else *s = (imax − imin)/(2.0 − imax − imin);
      rc = (imax − r)/(imax − imin);
      gc = (imax − g)/(imax − imin);
      bc = (imax − b)/(imax − imin);
      if (r ≡ imax) *h = bc − gc;
      else if (g ≡ imax) *h = 2.0 + rc − bc;
      else *h = 4.0 + gc − rc;
      *h *= 60.0;
      if (*h < 0) *h += 360.0;
    }
  }
```

20. `HLS_RGB`: Convert HLS colour specification to r, g, b intensities. Hue (h) is specified as a **double** value from 0 to 360°; Lightness (l) and Saturation (s) as **double**s from 0 to 1. The RGB components are returned as **double**s from 0 to 1.

```
⟨ Colour systems 17 ⟩ +≡
  double csDeviceColour::hlsval(double n1, double n2, double hue)
  {
    if (hue > 360.0) hue -= 360.0;
    else if (hue < 0.0) hue += 360.0;
    if (hue < 60.0) {
      return n1 + ((n2 - n1) * hue)/60.0;
    }
    else if (hue < 180.0) {
      return n2;
    }
    else if (hue < 240.0) {
      return n1 + ((n2 - n1) * (240.0 - hue))/60.0;
    }
    else {
      return n1;
    }
  }
  void csDeviceColour::hls_rgb(double h, double l, double s, double *r, double *g, double *b)
  {
    double m1, m2;
    if (l ≤ 0.5) m2 = l * (1.0 + s);
    else m2 = l + s - (l * s);
    m1 = 2 * l - m2;
    if (s ≡ 0) {
      *r = *g = *b = l;
    }
    else {
      *r = hlsval(m1, m2, h + 120.0);
      *g = hlsval(m1, m2, h);
      *b = hlsval(m1, m2, h - 120.0);
    }
  }
```

21. `RGB_YIQ` Convert RGB colour specification, r, g, b ranging from 0 to 1, to Y, I, Q colour specification. YIQ is the encoding used in NTSC television.

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.2989 & 0.5866 & 0.1144 \\ 0.5959 & -0.2741 & -0.3218 \\ 0.2113 & -0.5227 & 0.3113 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

⟨ Colour systems 17 ⟩ +≡

```
void csDeviceColour::rgb_yiq(double r, double g, double b, double *y, double *i, double *q)
{
  double ay = (r * 0.2989 + g * 0.5866 + b * 0.1144), ai = (r * 0.5959 + g * -0.2741 + b * -0.3218),
      aq = (r * 0.2113 + g * -0.5227 + b * 0.3113);
  *y = ay;
  if (ay == 1.0) {      /* Prevent round-off on grey scale */
    ai = aq = 0.0;
  }
  *i = ai;
  *q = aq;
}
```

22. `YIQ_RGB`: Convert YIQ colour specification, Y, I, Q given as **double**s: $0 \le Y \le 1$, $-0.6 \le I \le 0.6$, $-0.52 \le Q \le 0.52$, to R, G, B intensities in the range from 0 to 1. The matrix below is the inverse of the `RGB_YIQ` matrix above. YIQ is the encoding used in NTSC television.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.0000 & 0.9562 & 0.6210 \\ 1.0000 & -0.2717 & -0.6485 \\ 1.0000 & -1.1053 & 1.7020 \end{bmatrix} \cdot \begin{bmatrix} Y \\ I \\ Q \end{bmatrix}$$

⟨ Colour systems 17 ⟩ +≡

```
void csDeviceColour::yiq_rgb(double y, double i, double q, double *r, double *g, double *b)
{
  double ar = (y + i * 0.9562 + q * 0.6210), ag = (y + i * -0.2717 + q * -0.6485),
      ab = (y + i * -1.1053 + q * 1.7020);
  *r = max(0.0, min(1.0, ar));
  *g = max(0.0, min(1.0, ag));
  *b = max(0.0, min(1.0, ab));
}
```

23. `RGB_YUV`: Convert RGB colour specification, R, G, B ranging from 0 to 1, to Y, U, V colour specification. YUV is the encoding used by PAL television.

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.2989 & 0.5866 & 0.1144 \\ -0.1473 & -0.2717 & 0.4364 \\ 0.6149 & -0.5145 & -0.1004 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

⟨ Colour systems 17 ⟩ +≡

```
void csDeviceColour::rgb_yuv(double r, double g, double b, double *y, double *u, double *v)
{
  double ay = (r * 0.2989 + g * 0.5866 + b * 0.1144), au = (r * -0.1473 + g * -0.2891 + b * 0.4364),
      av = (r * 0.6149 + g * -0.5145 + b * -0.1004);
  *y = ay;
  if (ay == 1.0) {      /* Prevent round-off on grey scale */
    au = av = 0.0;
  }
  *u = au;
  *v = av;
}
```

24. `YUV_RGB`: Convert YUV colour specification, Y, U, V given as **double**s, to R, G, B intensities in the range from 0 to 1. The matrix below is the inverse of the *rgb_yuv* matrix above. YUV is the encoding used by PAL television.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.0000 & 0.0000 & 1.1402 \\ 1.0000 & -0.3959 & -0.5810 \\ 1.0000 & 2.0294 & 0.0000 \end{bmatrix} \cdot \begin{bmatrix} Y \\ U \\ V \end{bmatrix}$$

⟨ Colour systems 17 ⟩ +≡

```
void csDeviceColour::yuv_rgb(double y, double u, double v, double *r, double *g, double *b)
{
  double ar = (y + u * 0.0000 + v * 1.1402), ag = (y + u * -0.3959 + v * -0.5810),
      ab = (y + u * 2.0294 + v * 0.0000);
  *r = max(0.0, min(1.0, ar));
  *g = max(0.0, min(1.0, ag));
  *b = max(0.0, min(1.0, ab));
}
```

25. `RGB_SMPTE_204M`: Convert RGB colour specification, R, G, B ranging from 0 to 1, to Y, P_b, P_r colour specification according to the SMPTE 204M (1988) specification for HDTV using the SMPTE set of phosphors.

$$\begin{bmatrix} Y \\ P_b \\ P_r \end{bmatrix} = \begin{bmatrix} 0.2122 & 0.7013 & 0.0865 \\ -0.1162 & -0.3838 & 0.5000 \\ 0.6149 & -0.4451 & -0.0549 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

⟨ Colour systems 17 ⟩ +≡

```
void csDeviceColour::rgb_smpte_204M(double r, double g, double b, double *y, double
        *Pb, double *Pr)
{
  double ay = (r * 0.2122 + g * 0.7013 + b * 0.0865), aPb = (r * -0.1162 + g * -0.3838 + b * 0.5000),
      aPr = (r * 0.5000 + g * -0.4451 + b * -0.0549);
  *y = ay;
  if (ay == 1.0) {      /* Prevent round-off on grey scale */
    aPb = aPr = 0.0;
  }
  *Pb = aPb;
  *Pr = aPr;
}
```

26. `SMPTE_204M_RGB`: Convert a colour specified using the SMPTE reference phosphors as given in the SMPTE 204M (1988) specification for HDTV to R, G, B intensities in the range from 0 to 1. The matrix below is the inverse of the `RGB_SMPTE_204M` matrix above.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.0000 & 0.0000 & 1.5755 \\ 1.0000 & -0.2254 & -0.4768 \\ 1.0000 & 1.8270 & 0.0000 \end{bmatrix} \cdot \begin{bmatrix} Y \\ P_b \\ P_r \end{bmatrix}$$

⟨ Colour systems 17 ⟩ +≡

```
void csDeviceColour::smpte_204M_rgb(double y, double Pb, double Pr, double *r, double
        *g, double *b)
{
  double ar = (y + Pb * 0.0000 + Pr * 1.5755), ag = (y + Pb * -0.2254 + Pr * -0.4768),
      ab = (y + Pb * 1.8270 + Pr * 0.0000);
  *r = max(0.0, min(1.0, ar));
  *g = max(0.0, min(1.0, ag));
  *b = max(0.0, min(1.0, ab));
}
```

27. `RGB_CMY`: Convert RGB colour specification, R, G, B ranging from 0 to 1, to C, M, Y colour specification, also ranging from 0 to 1.

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

⟨ Colour systems 17 ⟩ +≡

```
void csDeviceColour::rgb_cmy(double r, double g, double b, double *c, double *m, double *y)
{
  *c = 1.0 - r;
  *m = 1.0 - g;
  *y = 1.0 - b;
}
```

28. `CMY_RGB`: Convert CMY colour specification, C, M, Y ranging from 0 to 1, to R, G, B colour specification, also ranging from 0 to 1.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} C \\ M \\ Y \end{bmatrix}$$

⟨ Colour systems 17 ⟩ +≡

```
void csDeviceColour::cmy_rgb(double c, double m, double y, double *r, double *g, double *b)
{
  *r = 1.0 - c;
  *g = 1.0 - m;
  *b = 1.0 - y;
}
```

29. *cmy_cmyk*: Convert CMY colour specification, C, M, Y ranging from 0 to 1, to C, M, Y, K colour specification, also ranging from 0 to 1. K is the black ink component in four colour printing processes. We convert C, M, Y by computing $K = \min(C, M, Y)$, then subtracting that value from each of the C, M, Y components.

⟨ Colour systems 17 ⟩ +≡

```
void csDeviceColour::cmy_cmyk(double c, double m, double y, double *oc, double *om, double
        *oy, double *ok)
{
  double k = min(c, min(m, y));
  *oc = c - k;
  *om = m - k;
  *oy = y - k;
  *ok = k;
}
```

30. *cmyk_cmy*: Convert CMYK colour specification, C, M, Y, K ranging from 0 to 1, to C, M, Y colour specification, also ranging from 0 to 1. K is the black ink component in four colour printing processes. We simply add the K component to each of the C, M, Y components.

```
⟨ Colour systems 17 ⟩ +≡
  void csDeviceColour::cmyk_cmy(double c, double m, double y, double k, double *oc, double
          *om, double *oy)
  {
    *oc = c + k;
    *om = m + k;
    *oy = y + k;
  }
```

31. Colour system conversion methods.

The following methods express a colour in a given colour system in a variety of others. The class must implement the *asRGB* method required for each of these conversions.

```
⟨ Colour systems 17 ⟩ +≡
  void csDeviceColour::asHSV(double &h, double &s, double &v)      /* To HSV */
  {
    double r, g, b;
    asRGB(r, g, b);
    rgb_hsv(r, g, b, &h, &s, &v);
  }
  void csDeviceColour::asHLS(double &h, double &l, double &s)      /* To HLS */
  {
    double r, g, b;
    asRGB(r, g, b);
    rgb_hls(r, g, b, &h, &l, &s);
  }
  void csDeviceColour::asYIQ(double &y, double &i, double &q)      /* To YIQ */
  {
    double r, g, b;
    asRGB(r, g, b);
    rgb_yiq(r, g, b, &y, &i, &q);
  }
  void csDeviceColour::asYUV(double &y, double &u, double &v)      /* To YUV */
  {
    double r, g, b;
    asRGB(r, g, b);
    rgb_yuv(r, g, b, &y, &u, &v);
  }
  void csDeviceColour::asSMPTE(double &y, double &Pb, double &Pr)
    /* To SMPTE 204M */
  {
    double r, g, b;
    asRGB(r, g, b);
    rgb_smpte_204M(r, g, b, &y, &Pb, &Pr);
  }
  void csDeviceColour::asCMY(double &c, double &m, double &y)      /* To CMY */
  {
    double r, g, b;
    asRGB(r, g, b);
    rgb_cmy(r, g, b, &c, &m, &y);
  }
```

32. Perceptual colour systems.
The following sections define data used in manipulating colour systems.

33. CIE colour systems.

A colour system is defined by the CIE x and y coordinates of its three primary illuminants and the x and y coordinates of the white point.

```
⟨ Class definitions 6 ⟩ +≡
  class CIEColourSystem {
  private:
    string name;      /* Colour system name */
    double xRed, yRed,      /* Red primary illuminant */
    xGreen, yGreen,      /* Green primary illuminant */
    xBlue, yBlue,      /* Blue primary illuminant */
    xWhite, yWhite;      /* White point */
  public:
    CIEColourSystem(string c_name, double c_xRed, double x_yRed, double c_xGreen, double
            c_yGreen, double c_xBlue, double c_yBlue, double c_xWhite, double c_yWhite)
    {
      name = c_name;
      xRed = c_xRed;
      yRed = x_yRed;
      xGreen = c_xGreen;
      yGreen = c_yGreen;
      xBlue = c_xBlue;
      yBlue = c_yBlue;
      xWhite = c_xWhite;
      yWhite = c_yWhite;
    }
    void xyz_to_rgb(double xc, double yc, double zc, double *r, double *g, double *b);
    static bool inside_gamut(double r, double g, double b);
    bool constrain_rgb(double *x, double *y, double *z, double *r, double *g, double *b, bool
        interpwp = false);
  protected:
    static double clamp(double v, double l, double h)
    {
      return (v < l) ? l : ((v > h) ? h : v);      /* Constrain l ≤ v ≤ h */
    }
  };
  extern CIEColourSystem NTSCsystem, EBUsystem, SMPTEsystem, HDTVsystem, CIEsystem;
    /* Predefined standard colour systems */
```

34. *xyz_to_rgb*: Given an additive tricolour system defined by the CIE x and y chromaticities of its three primaries (z is derived trivially as $1-(x+y)$), and a desired chromaticity (x_c, y_c, z_c) in CIE space, determine the contribution of each primary in a linear combination which sums to the desired chromaticity. If the requested chromaticity falls outside the Maxwell triangle (colour gamut) formed by the three primaries, one of the r, g, or b weights will be negative. Use *inside_gamut* to test for a valid colour and *constrain_rgb* to desaturate an outside-gamut colour to the closest representation within the available gamut.

```
⟨ Colour systems 17 ⟩ +≡
  void CIEColourSystem::xyz_to_rgb(double xc, double yc, double zc, double *r, double *g, double
          *b)
  {
    double xr, yr, zr, xg, yg, zg, xb, yb, zb;
    xr = xRed;
    yr = yRed;
    zr = 1 - (xr + yr);
    xg = xGreen;
    yg = yGreen;
    zg = 1 - (xg + yg);
    xb = xBlue;
    yb = yBlue;
    zb = 1 - (xb + yb);
    *r = (-xg * yc * zb + xc * yg * zb + xg * yb * zc - xb * yg * zc - xc * yb * zg + xb * yc * zg)/(xr * yg *
        zb - xg * yr * zb - xr * yb * zg + xb * yr * zg + xg * yb * zr - xb * yg * zr);
    *g = (xr * yc * zb - xc * yr * zb - xr * yb * zc + xb * yr * zc + xc * yb * zr - xb * yc * zr)/(xr * yg * zb -
        xg * yr * zb - xr * yb * zg + xb * yr * zg + xg * yb * zr - xb * yg * zr);
    *b = (xr * yg * zc - xg * yr * zc - xr * yc * zg + xc * yr * zg + xg * yc * zr - xc * yg * zr)/(xr * yg * zb -
        xg * yr * zb - xr * yb * zg + xb * yr * zg + xg * yb * zr - xb * yg * zr);
  }
```

35. *inside_gamut*: Test whether a requested colour is within the gamut achievable with the primaries of the current colour system. This amounts simply to testing whether all the primary weights are non-negative.

```
⟨ Colour systems 17 ⟩ +≡
  bool CIEColourSystem::inside_gamut(double r, double g, double b)
  {
    return (r ≥ 0) ∧ (g ≥ 0) ∧ (b ≥ 0);
  }
```

36. *constrain − rgb*: If the requested RGB shade contains a negative weight for one of the primaries, it lies outside the colour gamut accessible from the given triple of primaries. Desaturate it by mixing with the white point of the colour system so as to reduce the primary with the negative weight to zero. This is equivalent to finding the intersection on the CIE diagram of a line drawn between the white point and and the requested color and the edge of the Maxwell triangle formed by the three primaries. If *interpwp* is nonzero, the white point defined by the sum of the three primary illuminants is used instead of the colour system's actual white point. While indefensible from the standpoint of colour theory, this produces much better looking charts on computer monitors, since the white points of most colour systems are well to the blue of the white defined by the $R = G = B = 1$.

```
⟨ Colour systems 17 ⟩ +≡
  bool CIEColourSystem::constrain_rgb(double *x, double *y, double *z, double *r, double
      *g, double *b, bool interpwp)
{    /* Is the contribution of one of the primaries negative ? */
  if (¬inside_gamut(*r, *g, *b)) {
    double par, wr, wg, wb, xw, yw;
      /* Determine the white point used in interpolating out of gamut colours. If interpwp is set, the
        colour system's white point is used. Otherwise, the white defined by an equal mix of the three
        illuminants is taken as the origin of the interpolation line drawn to the out of gamut colour. */
    if (interpwp) {
      xw = xWhite;
      yw = yWhite;
    }
    else {
      xw = (xRed + xGreen + xBlue)/3;
      yw = (yRed + yGreen + yBlue)/3;
    }    /* Yes. Find the RGB mixing weights of the white point (we assume the white point is in the
        gamut!). */
    xyz_to_rgb(xw, yw, 1 − (xw + yw), &wr, &wg, &wb);
      /* Find the primary with negative weight and calculate the parameter of the point on the vector
        from the white point to the original requested colour in RGB space. */
    if (*r < *g ∧ *r < *b) {
      par = wr/(wr − *r);
    }
    else if (*g < *r ∧ *g < *b) {
      par = wg/(wg − *g);
    }
    else {
      par = wb/(wb − *b);
    }    /* Since XYZ space is a linear transformation of RGB space, we can find the XYZ space
        coordinates of the point where the edge of the gamut intersects the vector from the white
        point to the original colour by multiplying the parameter in RGB space by the difference
        vector in XYZ space. */
    *x = clamp(xw + par * (*x − xw), 0, 1);
    *y = clamp(yw + par * (*y − yw), 0, 1);
    *z = clamp(1 − (*x + *y), 0, 1);    /* Now finally calculate the gamut-constrained RGB weights. */
    *r = clamp(wr + par * (*r − wr), 0, 1);
    *g = clamp(wg + par * (*g − wg), 0, 1);
    *b = clamp(wb + par * (*b − wb), 0, 1);
    return true;    /* Colour modified to fit RGB gamut */
  }
  return false;    /* Colour within RGB gamut */
}
```

37. CIE colour matching functions.

The following table gives the CIE colour matching functions $\bar{x}(\lambda)$, $\bar{y}(\lambda)$, and $\bar{z}(\lambda)$, for wavelengths λ at 5 nanometre increments from 380 nm through 780 nm. This table is used in conjunction with Planck's law for the energy spectrum of a black body at a given temperature to plot the black body curve on the CIE chart.

⟨ Colour system data tables 37 ⟩ ≡

```
static double cie_colour_match[][3] = {{0.0014, 0.0000, 0.0065},      /* 380 nm */
{0.0022, 0.0001, 0.0105}, {0.0042, 0.0001, 0.0201}, {0.0076, 0.0002, 0.0362}, {0.0143, 0.0004, 0.0679},
    {0.0232, 0.0006, 0.1102}, {0.0435, 0.0012, 0.2074}, {0.0776, 0.0022, 0.3713}, {0.1344, 0.0040, 0.6456},
    {0.2148, 0.0073, 1.0391}, {0.2839, 0.0116, 1.3856}, {0.3285, 0.0168, 1.6230}, {0.3483, 0.0230, 1.7471},
    {0.3481, 0.0298, 1.7826}, {0.3362, 0.0380, 1.7721}, {0.3187, 0.0480, 1.7441}, {0.2908, 0.0600, 1.6692},
    {0.2511, 0.0739, 1.5281}, {0.1954, 0.0910, 1.2876}, {0.1421, 0.1126, 1.0419}, {0.0956, 0.1390, 0.8130},
    {0.0580, 0.1693, 0.6162}, {0.0320, 0.2080, 0.4652}, {0.0147, 0.2586, 0.3533}, {0.0049, 0.3230, 0.2720},
    {0.0024, 0.4073, 0.2123}, {0.0093, 0.5030, 0.1582}, {0.0291, 0.6082, 0.1117}, {0.0633, 0.7100, 0.0782},
    {0.1096, 0.7932, 0.0573}, {0.1655, 0.8620, 0.0422}, {0.2257, 0.9149, 0.0298}, {0.2904, 0.9540, 0.0203},
    {0.3597, 0.9803, 0.0134}, {0.4334, 0.9950, 0.0087}, {0.5121, 1.0000, 0.0057}, {0.5945, 0.9950, 0.0039},
    {0.6784, 0.9786, 0.0027}, {0.7621, 0.9520, 0.0021}, {0.8425, 0.9154, 0.0018}, {0.9163, 0.8700, 0.0017},
    {0.9786, 0.8163, 0.0014}, {1.0263, 0.7570, 0.0011}, {1.0567, 0.6949, 0.0010}, {1.0622, 0.6310, 0.0008},
    {1.0456, 0.5668, 0.0006}, {1.0026, 0.5030, 0.0003}, {0.9384, 0.4412, 0.0002}, {0.8544, 0.3810, 0.0002},
    {0.7514, 0.3210, 0.0001}, {0.6424, 0.2650, 0.0000}, {0.5419, 0.2170, 0.0000}, {0.4479, 0.1750, 0.0000},
    {0.3608, 0.1382, 0.0000}, {0.2835, 0.1070, 0.0000}, {0.2187, 0.0816, 0.0000}, {0.1649, 0.0610, 0.0000},
    {0.1212, 0.0446, 0.0000}, {0.0874, 0.0320, 0.0000}, {0.0636, 0.0232, 0.0000}, {0.0468, 0.0170, 0.0000},
    {0.0329, 0.0119, 0.0000}, {0.0227, 0.0082, 0.0000}, {0.0158, 0.0057, 0.0000}, {0.0114, 0.0041, 0.0000},
    {0.0081, 0.0029, 0.0000}, {0.0058, 0.0021, 0.0000}, {0.0041, 0.0015, 0.0000}, {0.0029, 0.0010, 0.0000},
    {0.0020, 0.0007, 0.0000}, {0.0014, 0.0005, 0.0000}, {0.0010, 0.0004, 0.0000}, {0.0007, 0.0002, 0.0000},
    {0.0005, 0.0002, 0.0000}, {0.0003, 0.0001, 0.0000}, {0.0002, 0.0001, 0.0000}, {0.0002, 0.0001, 0.0000},
    {0.0001, 0.0000, 0.0000}, {0.0001, 0.0000, 0.0000}, {0.0001, 0.0000, 0.0000}, {0.0000, 0.0000, 0.0000}
  /* 780 nm */
};
```

See also sections 38 and 39.

This code is used in section 2.

38. Spectral chromaticity co-ordinates.

The following table gives the spectral chromaticity co-ordinates $x(\lambda)$ and $y(\lambda)$ for wavelengths in 5 nanometre increments from 380 nm through 780 nm. These co-ordinates represent the position in the CIE x–y space of pure spectral colours of the given wavelength, and thus define the outline of the CIE "tongue" diagram.

```
⟨ Colour system data tables 37 ⟩ +≡
  static double spectral_chromaticity[81][3] = {{0.1741, 0.0050},      /* 380 nm */
  {0.1740, 0.0050}, {0.1738, 0.0049}, {0.1736, 0.0049}, {0.1733, 0.0048}, {0.1730, 0.0048}, {0.1726, 0.0048},
      {0.1721, 0.0048}, {0.1714, 0.0051}, {0.1703, 0.0058}, {0.1689, 0.0069}, {0.1669, 0.0086}, {0.1644,
      0.0109}, {0.1611, 0.0138}, {0.1566, 0.0177}, {0.1510, 0.0227}, {0.1440, 0.0297}, {0.1355, 0.0399},
      {0.1241, 0.0578}, {0.1096, 0.0868}, {0.0913, 0.1327}, {0.0687, 0.2007}, {0.0454, 0.2950}, {0.0235,
      0.4127}, {0.0082, 0.5384}, {0.0039, 0.6548}, {0.0139, 0.7502}, {0.0389, 0.8120}, {0.0743, 0.8338},
      {0.1142, 0.8262}, {0.1547, 0.8059}, {0.1929, 0.7816}, {0.2296, 0.7543}, {0.2658, 0.7243}, {0.3016,
      0.6923}, {0.3373, 0.6589}, {0.3731, 0.6245}, {0.4087, 0.5896}, {0.4441, 0.5547}, {0.4788, 0.5202},
      {0.5125, 0.4866}, {0.5448, 0.4544}, {0.5752, 0.4242}, {0.6029, 0.3965}, {0.6270, 0.3725}, {0.6482,
      0.3514}, {0.6658, 0.3340}, {0.6801, 0.3197}, {0.6915, 0.3083}, {0.7006, 0.2993}, {0.7079, 0.2920},
      {0.7140, 0.2859}, {0.7190, 0.2809}, {0.7230, 0.2770}, {0.7260, 0.2740}, {0.7283, 0.2717}, {0.7300, 0.2700},
      {0.7311, 0.2689}, {0.7320, 0.2680}, {0.7327, 0.2673}, {0.7334, 0.2666}, {0.7340, 0.2660}, {0.7344, 0.2656},
      {0.7346, 0.2654}, {0.7347, 0.2653}, {0.7347, 0.2653}, {0.7347, 0.2653}, {0.7347, 0.2653}, {0.7347, 0.2653},
      {0.7347, 0.2653}, {0.7347, 0.2653}, {0.7347, 0.2653}, {0.7347, 0.2653}, {0.7347, 0.2653}, {0.7347, 0.2653},
      {0.7347, 0.2653}, {0.7347, 0.2653}, {0.7347, 0.2653}, {0.7347, 0.2653}, {0.7347, 0.2653}, {0.7347, 0.2653}
    /* 780 nm */
  };
```

39. Standard colour system definitions.

The following define the white point chromaticities for the NTSC, EBU, and SMPTE colour systems.

```
#define IlluminantC 0.3101, 0.3162      /* For NTSC television */
#define IlluminantD65 0.3127, 0.3291      /* For EBU and SMPTE */
```

```
⟨ Colour system data tables 37 ⟩ +≡
  CIEColourSystem NTSCsystem("NTSC", 0.67, 0.33, 0.21, 0.71, 0.14, 0.08, IlluminantC),
      EBUsystem("EBU␣(PAL/SECAM)", 0.64, 0.33, 0.29, 0.60, 0.15, 0.06, IlluminantD65),
      SMPTEsystem("SMPTE", 0.630, 0.340, 0.310, 0.595, 0.155, 0.070,
      IlluminantD65), HDTVsystem("HDTV", 0.670, 0.330, 0.210, 0.710, 0.150, 0.060, IlluminantD65),
      CIEsystem("CIE", 0.7355, 0.2645, 0.2658, 0.7243, 0.1669, 0.0085, 0.3894, 0.3324);
```

40. Test program.

```
⟨ colour_test.c 1 ⟩ +≡
  ⟨ Test program include files 43 ⟩;
  ⟨ Show how to call test program 42 ⟩;
  int main(int argc, char *argv[])
  {
    extern char *optarg;      /* Imported from getopt */
    extern int optind;
    int opt;
    ⟨ Process command-line options 41 ⟩;
    return 0;
  }
```

41. We use *getopt* to process command line options. This permits aggregation of options without arguments and both `-d`*arg* and `-d` *arg* syntax.

```
⟨ Process command-line options 41 ⟩ ≡
  while ((opt = getopt(argc, argv, "nu-:")) ≠ −1) {
    switch (opt) {
    case 'u':     /* -u Print how-to-call information */
      case '?': usage();
      return 0;
    case '-':     /* -- Extended options */
      switch (optarg[0]) {
      case 'c':     /* --copyright */
        cout ≪ "This␣program␣is␣in␣the␣public␣domain.\n";
        return 0;
      case 'h':     /* --help */
        usage();
        return 0;
      case 'v':     /* --version */
        cout ≪ PRODUCT ≪ "␣" ≪ VERSION ≪ "\n";
        cout ≪ "Last␣revised:␣" ≪ REVDATE ≪ "\n";
        cout ≪ "The␣latest␣version␣is␣always␣available\n";
        cout ≪ "at␣http://www.fourmilab.ch/eggtools/eggshell\n";
        return 0;
      }
    }
  }
```

This code is used in section 40.

42. Procedure *usage* prints how-to-call information.

```
⟨ Show how to call test program 42 ⟩ ≡
  static void usage(void)
  {
    cout ≪ PRODUCT ≪ "␣␣--␣␣Analyse␣eggsummary␣files.␣␣Call:\n";
    cout ≪ "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣" ≪ PRODUCT ≪ "␣[options]␣[infile]␣[outfile]\n";
    cout ≪ "\n";
    cout ≪ "Options:\n";
    cout ≪ "␣␣␣␣␣␣␣␣␣␣␣--copyright␣␣␣␣␣␣␣Print␣copyright␣information\n";
    cout ≪ "␣␣␣␣␣␣␣␣␣␣␣-u,␣--help␣␣␣␣␣␣␣␣Print␣this␣message\n";
    cout ≪ "␣␣␣␣␣␣␣␣␣␣␣--version␣␣␣␣␣␣␣␣␣Print␣version␣number\n";
    cout ≪ "\n";
    cout ≪ "by␣John␣Walker\n";
    cout ≪ "http://www.fourmilab.ch/\n";
  }
```

This code is used in section 40.

43. We need the following definitions to compile the test program.

```
⟨ Test program include files 43 ⟩ ≡
#include "config.h"      /* Our configuration */      /* C++ include files */
#include <iostream>
#include <exception>
#include <stdexcept>
#include <string>
  using namespace std;
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#ifdef HAVE_GETOPT
#ifdef HAVE_UNISTD_H
#include <unistd.h>
#endif
#else
#include "getopt.h"      /* No system getopt—use our own */
#endif
#include "colour.h"      /* Class definitions for this package */
```

This code is used in section 40.

44. Index. The following is a cross-reference table for `colour`. Single-character identifiers are not indexed, nor are reserved words. Underlined entries indicate where an identifier was declared.

⟨ Application include files 4 ⟩ Used in section 2.
⟨ Class definitions 6, 7, 8, 9, 10, 11, 12, 33 ⟩ Used in section 3.
⟨ Class implementations 5 ⟩ Used in section 2.
⟨ Colour system data tables 37, 38, 39 ⟩ Used in section 2.
⟨ Colour systems 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 34, 35, 36 ⟩ Used in section 5.
⟨ Device colour derived methods 14 ⟩ Used in section 12.
⟨ Device colour fundamental methods 13 ⟩ Used in section 12.
⟨ Device colour system conversion utilities 15 ⟩ Used in section 12.
⟨ Process command-line options 41 ⟩ Used in section 40.
⟨ Show how to call test program 42 ⟩ Used in section 40.
⟨ Test program include files 43 ⟩ Used in section 40.
⟨ `colour.h` 3 ⟩
⟨ `colour_test.c` 1, 40 ⟩

COLOUR