1. Skew-T. Create a blank "skew T–$\ln p$" diagram. Written by Bret D. Whissel, Tallahassee, Florida, November 2011. The ordinate coordinates are logarithmic units of pressure in millibars/hPa, with the surface pressure (maximum value) at the bottom. On the abcissa, the quantity is temperature, but skewed by the pressure level, so plotting a temperature point in the x-dimension also requires a pressure component.

Using this coordinate system, isotherms will be parallel straight lines at 45° angles. We'll also draw curved grid lines for dry adiabats, saturated (moist) adiabats, and mixing ratio lines. To draw these various grid lines, we need to calculate temperature and/or pressure quantities that define the curves for particular fixed values.

Once we've generated sets of points which define the curves at sufficient resolution, we apply a curve-fitting algorithm to the points to retrieve a new set of control points which we can then pass to the PostScript `curveto` operator. Not only is this a huge data-reduction tactic, but the resulting plot will be able to be printed at extremely high resolutions without revealing any underlying point-to-point straight line segments. This means that our blank chart can be pretty even at poster-size enlargements.

⟨ Includes 2 ⟩
⟨ Global Vars 3 ⟩
⟨ Function Defs 5 ⟩
⟨ Main Program 52 ⟩

2. We need math and a few standard libs. In addition, we'll need some definitions from *Graphics Gems* which are used by the curve-fitting functions.

⟨ Includes 2 ⟩ ≡
```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <getopt.h>
#include "graphics_gems.h"
```
This code is used in section 1.

3. Virtual grid definitions. We start by defining a grid on which we'll generate a plot. The number of pixels in each dimension here should be the same, creating a square grid of square pixels, i.e., $dy/dx = 1$. This is necessary in order to define a grid where the isotherms can be generated at exact 45° angles.

The grid is merely a virtual construct: we don't actually allocate space for it. And since the grid is just a mathematical convenience, it doesn't theoretically matter what its scale is: a mapping from 0 to 1 is as valid as a mapping from −999 to 10,000. In practice, however, it is useful to have a scale that includes a substantial component to the left of the decimal point, since we'll be spitting these numbers out to the PostScript engine.

```
#define GRID_SCALE 1000.0
```

⟨ Global Vars 3 ⟩ ≡
```
double xmax ← GRID_SCALE, ymax ← GRID_SCALE;
```
See also sections 4, 7, 11, 12, 13, 14, 15, 16, 17, 19, 20, 25, 31, 33, 35, 37, 39, 41, 43, and 46.
This code is used in section 1.

4. Now we allocate some variables which we'll use to convert from pressure units to y coordinates.

⟨ Global Vars 3 ⟩ +≡
```
double yscale, ybias;
```

5. Calculate the scale and bias in the y dimension. The pressure scale is logarithmic, so we must account for that in setting the scale. The pressure values are given in hPa.

```
⟨ Function Defs 5 ⟩ ≡
  void calc_ydims(double minp, double maxp)
  {
    ybias ← log(minp);
    yscale ← ymax/(log(maxp) − log(minp));
  }
```

See also sections 6, 8, 9, 10, 22, 23, 26, 28, 30, 32, 45, 47, 48, 50, 51, 66, 68, 71, 72, and 89.

This code is used in section 1.

6. This function converts a pressure value (in hPA) to a y-value given the previously established values of *ybias* and *yscale*. With increasing pressure, the y-value decreases. Here we create a mapping where the minimum pressure will be transformed into the value *ymax*, and the highest pressure will be mapped to a y-value of 0.0. We will force a return value of 0.0 if the calculated value is "close enough" (as defined by *eps*).

```
⟨ Function Defs 5 ⟩ +≡
  double p2y(double p)
  {
    static double eps ← 1.0 · 10^−5;
    double val ← ymax − yscale ∗ (log(p) − ybias);
    return fabs(val) ≤ eps ? 0.0 : val;
  }
```

7. Now we allocate space for variables which we'll use to convert from temperature and pressure units to x coordinates.

```
⟨ Global Vars 3 ⟩ +≡
  double xscale, xoffset, xbias;
```

8. Calculate the scale, offset, and bias in the x dimension. The $T_{\min}$ and $T_{\max}$ temperatures are passed in as °C and used to determine x scaling parameters. The x scale is linear with respect to temperature, but since it is skewed, the $T_{\min}$ and $T_{\max}$ values define the x range only when $y = 0$. We will offset the x value so that the center of the temperature scale will be transformed to an x value of 0.0.

```
⟨ Function Defs 5 ⟩ +≡
  void calc_xdims(double mint, double maxt)
  {
    xoffset ← −xmax/2.0;
    xbias ← mint;
    xscale ← xmax/(maxt − mint);
  }
```

9. This function converts a temperature in °C and pressure in hPa into an x value. Position along the x-axis is dependent on *both* temperature and pressure. Since the "pixels" are square, we calculate the pressure-related x-offset by calculating what the corresponding y-offset for the pressure should be, and then we add that value to the temperature term. An input value of *mint* with a pressure level of *maxp* (where $y = 0$) should produce an x value of $-xmax/2$. Likewise, an input value of *maxt* and *maxp* should be transformed to an x value of $xmax/2$.

```
⟨ Function Defs 5 ⟩ +≡
  double Tp2x(double T, double p)
  {
    return xscale * (T − xbias) + p2y(p) + xoffset;
  }
```

10. It will be convenient to provide a reverse mapping from (x, y) coordinates to (T, p) coordinates, so we provide that conversion here. The calculations are just inversions of the *p2y*( ) and *Tp2x*( ) functions. The calculation of $p \Rightarrow y$ and $y \Rightarrow p$ are particularly prone to loss-of-precision errors, i.e., $\neg\forall p : p - p\big(y(p)\big) = 0$, so we must be careful about how the resulting values are used.

```
⟨ Function Defs 5 ⟩ +≡
  void xy2Tp(double x, double y, double *T, double *p)
  {
    if (T) *T ← (x − y − xoffset)/xscale + xbias;
    if (p) *p ← exp(ybias − (y − ymax)/yscale);
  }
```

11. Plot dimensions. We have now defined the transformations $(T, p) \Rightarrow (x, y)$. Eventually these coordinates need to be mapped to positions on a page, and that's what we'll tackle next. We'll define the plot's dimensions with *plotx* and *ploty*. If the ratio $y/x \neq 1$, then the plot will be a window on the virtual grid, centered on $x = 0$ and anchored at $y = 0$.

If $plotx > ploty$, then the whole temperature range from *mint* to *maxt* will be visible, but the pressure values at the top of the virtual grid will be out of the window. If $ploty > plotx$, then the whole pressure range from *maxp* up to *minp* will be visible, but temperatures will be windowed out at both extremes near *mint* and *maxt*. The amount of data cut out of the window depends on the ratio of $ploty/plotx$. All of the data on the virtual grid will be visible if $plotx = ploty$.

Since PostScript is our target language to generate the plot, these dimensions should be given in PostScript units (72 points per inch). This defines the size of the plot on the page. The default plot will be portrait-oriented, filling a US letter-sized page allowing for a quarter-inch border at each edge.

```
⟨ Global Vars 3 ⟩ +≡
  double plotx ← 8.0 * 72.0, ploty ← 10.5 * 72.0;
```

12. Now we define the page size, again in PostScript units. The width and height of the paper are always defined in portrait mode (longest edge along the y-axis). We're setting up default US Letter (8.5×11 in).

```
⟨ Global Vars 3 ⟩ +≡
  float pgdim_x ← 8.5 * 72.0, pgdim_y ← 11.0 * 72.0;
```

13. These variables give the offset of the plot on the page, again given in PostScript units. This defines where the lower left corner of the plot will start in relation to the page's lower-left corner (after landscape rotation, if requested).

```
⟨ Global Vars 3 ⟩ +≡
  double transx ← 0.25 * 72.0, transy ← 0.25 * 72.0;
```

14. Set *landscape* ← `TRUE` for landscape orientation on the page. By default we start with portrait orientation.

⟨ Global Vars 3 ⟩ +≡
```
int landscape ← FALSE;
```

15. Set *colormode* ← `TRUE` for color output. Otherwise, the plot will be created in black and white.

⟨ Global Vars 3 ⟩ +≡
```
int colormode ← TRUE;
```

16. Other configurable parameters. The spacing of the isotherm and dry adiabat gridlines is specified by *tinc*. Reasonable values are 1° or 2°. Black and white mode is more readable using the `LORES_T` setting.

```
#define HIRES_T 1.0;
#define LORES_T 2.0;
```

⟨ Global Vars 3 ⟩ +≡
```
double tinc ← HIRES_T;
```

17. To do the transformation from data values to grid coordinates, we define the minimum and maximum temperatures and pressures. We give them their default values here.

⟨ Global Vars 3 ⟩ +≡
```
double mint ← −25.0, maxt ← 55.0, minp ← 100.0, maxp ← 1050.0;
```

18. Meteorological constants. Let's start with a few simple conversions. `C2K`(x) converts Celsius degrees to Kelvin, and `K2C`(x) converts Kelvin to Celsius.

```
#define C2K(x) ((x) + 273.15)
#define K2C(x) ((x) − 273.15)
```

19. We need a few constants in our calculations. c_{pd} is the specific heat of dry air at constant pressure, and c_{pv} is the specific heat of water vapor at constant pressure. R_d is the Ideal Gas Law constant for dry air, while R_v is the constant for moist air. We initialize the standard pressure *std_p* ← 1000. I tried to find values with the most precision for these constants using several sources (textbooks, journal articles, web).

⟨ Global Vars 3 ⟩ +≡
```
double cpd ← 1005.7; /* J kg^−1 K^−1 */
double cpv ← 1952.0; /* J kg^−1 K^−1 */
double Rd ← 287.05307; /* J kg^−1 K^−1 */
double Rv ← 461.5; /* J kg^−1 K^−1 */
double std_p ← 1000.0; /* 1000 hPa */
```

20. There are a few values which we will pre-calculate from the constants we've already defined. However, we can't initialize global variables using operations on other variables in `C`, so we'll allocate space for them here, and we'll have to initialize them later.

⟨ Global Vars 3 ⟩ +≡
```
double epsilon, Rd_cpd;
```

21. The ratio of the dry air gas constant to the water vapor constant (R_d/R_v) is often called ϵ in meteorological equations. Another ratio that pops up is R_d/c_{pd}, and we pre-calculate that here to save operations later.

⟨ Pre-calculate variables 21 ⟩ ≡
```
epsilon ← Rd/Rv;
Rd_cpd ← Rd/cpd;
```
This code is used in section 52.

22. Curve Calculations. We'll generate five different kinds of curves: isobars, isotherms, dry adiabats, moist adiabats, and mixing ratios. The following functions will convert temperature and pressure pairs into (x, y) coordinates. We won't bother with an isobar function since we really only need the *p2y*( ) function to get the corresponding y-coordinate. We'll start with the trivial isotherm function:

```
⟨ Function Defs 5 ⟩ +≡
  void isotherm(double T, double p, double *x, double *y)
  {
    *x ← Tp2x(T, p); *y ← p2y(p);
  }
```

23. This function calculates dry adiabats. The potential temperature θ for a particular temperature T may be calculated for a new pressure p using Poisson's equation:

$$\theta = T \left(\frac{p_0}{p} \right)^{R_d/c_{pd}},$$

where p_0 is our starting point (standard pressure, 1000 hPa), c_{pd} is the specific heat of dry air at constant pressure, and R_d is the gas constant for dry air. If we wish to draw a curve with constant θ, we can rearrange the equation to find T as a function of pressure and θ:

$$T = \theta \Big/ \left(\frac{p_0}{p} \right)^{R_d/c_{pd}},$$

with θ and T being expressed in Kelvin. We'll design the function to accept and return the temperature in °C, and we'll do the conversion from Celsius to Kelvin (and back again) on the fly.

```
⟨ Function Defs 5 ⟩ +≡
  void dry_adiabat(double theta, double p, double *x, double *y)
  {
    double T;
    T ← C2K(theta)/pow(std_p/p, Rd_cpd);
    *x ← Tp2x(K2C(T), p); *y ← p2y(p);
  }
```

24. To draw mixing ratio lines, we use an expression for the saturation mixing ratio w_s as follows:

$$w_s(T,p) = \frac{\epsilon\, e_s(T)}{p - e_s(T)},$$

where $\epsilon = R_d/R_v$. The symbol $e_s(T)$ is the saturation vapor pressure which is approximated empirically according to the August-Roche-Magnus formula. (See Lawrence (2005) for historical background.)

$$e_s(T) \approx C\, \exp\left(\frac{A\,T_c}{T_c + B}\right),$$

The temperature T_c is expressed in Celsius, not Kelvin, and the result is given in hPa (with suitable setting of constants). (The standard reference formula for $e_s(T)$ is the Goff–Gratch equation, but it is much harder to rearrange to solve for T, as we'll be doing next.) If we expand $w_s(T,p)$ using $e_s(T)$, we can then set $w_s(T,p) \equiv W$ (for some constant W) and rearrange to solve for temperature as follows:

$$W = \frac{\epsilon\, C\, \exp\left(\frac{A\,T_c}{T_c + B}\right)}{p - C\, \exp\left(\frac{A\,T_c}{T_c + B}\right)},$$

rearranging to solve for T_c:

$$T_c = \frac{B\, \ln\left(\frac{W \cdot p}{C(W+\epsilon)}\right)}{A - \ln\left(\frac{W \cdot p}{C(W+\epsilon)}\right)}.$$

25. Here are the constant values for calculating the saturation vapor pressure using the August-Roche-Magnus formula. These constants are recommended by Alduchov and Eskridge (1996).

```
⟨ Global Vars 3 ⟩ +≡
  double Aes ← 17.625;      /* dimensionless */
  double Bes ← 243.04;      /* °C */
  double Ces ← 6.1094;      /* hPa */
```

26. For the mixing ratio curves, we'll accept particular values of W and p, and then we'll calculate the proper value of T_c for those parameters. Since the mixing ratio is often expressed as $\mathrm{g\,kg^{-1}}$, we design the function to accept values using those units. However, the equations are designed for $\mathrm{kg\,kg^{-1}}$, so we must first make a conversion to the proper units.

```
⟨ Function Defs 5 ⟩ +≡
  void mixing_ratio(double g_kg, double hpa, double *x, double *y)
  {
    double T, logval, kg_kg ← g_kg/1000.0;
    logval ← log((kg_kg * hpa)/(Ces * (kg_kg + epsilon)));
    T ← (Bes * logval)/(Aes − logval);
    *x ← Tp2x(T, hpa); *y ← p2y(hpa);
  }
```

27. And now for the toughest part: drawing saturated adiabatic lines. The AMS *Glossary of Meteorology* provides this formula for the lapse rate of the pseudoadiabatic process (not reversible):

$$\Gamma_{ps}(T,p) = g\frac{\big(1+w_s(T,p)\big)\left(1+\dfrac{L_v(T)w_s(T,p)}{R_dT}\right)}{c_{pd}+w_s(T,p)c_{pv}+\dfrac{L_v(T)^2w_s(T,p)\cdot(\epsilon+w_s(T,p))}{R_dT^2}}.$$

$w_s(T,p)$ is the mixing ratio of water vapor, c_{pd} is the specific heat of dry air at constant pressure, and c_{pv} is the specific heat of water vapor at constant pressure. $L_v(T)$ is the latent heat of vaporization/condensation, R_d is the dry air gas constant. ϵ is the ratio of the gas constants of dry air and water vapor. The lapse rate Γ_{ps} provides the change in temperature with height, not pressure, so we'll make this adjustment next.

The pressure lapse rate is given by $dT/dp = \Gamma_{ps}(T,p)/\rho g$, so we can integrate this expression from a starting temperature and pressure to get the new temperature at the new pressure using something like the following:

$$T(p_1) = T(p_0) + \int_{p_0}^{p_1} \frac{\Gamma_{ps}(T,p)}{\rho g}dp.$$

This expression isn't completely accurate since $\Gamma_{ps}(T,p)$ is a function of both temperature and pressure, and the temperature will also be changing during the integration over pressure. The gravity acceleration constant g cancels. To determine the density ρ, we can make use of the following definition from the Ideal Gas Law and Dalton's Law of Partial Pressures:

$$\rho = \frac{p-e_s(T)}{R_dT} + \frac{e_s(T)}{R_vT}.$$

28. With the following function definition, Tc is given in °C, and p in hPa. Output units are °K hPa^{-1}. The calculations for $e_s(T)$ and $w_s(T,p)$ have already been discussed in reference to drawing the mixing ratio curves.

```
⟨ Function Defs 5 ⟩ +≡
  ⟨ Latent Heat Def 29 ⟩
  double dt_dp(double Tc, double p)
  {
    double es, ws, rho, Tk ← C2K(Tc), L ← latent_heat(Tc);
    double num, denom;
    es ← Ces * exp((Aes * Tc)/(Tc + Bes));
    ws ← (epsilon * es)/(p − es);
    rho ← (p − es)/(Rd * Tk) + es/(Rv * Tk);
    num ← (1.0 + ws) * (1.0 + (L * ws/(Rd * Tk)));
    denom ← cpd + ws * cpv + (L * L * ws * (epsilon + ws)/(Rd * Tk * Tk));
    return num/(denom * rho);
  }
```

29. The latent heat of vaporization/condensation varies slightly with temperature. The following is a cubic fit to the data in Table 2.1 in the textbook *A Short Course in Cloud Physics*, 3rd ed. (1989), by R. R. Rogers and M. K. Yau. The temperature Tc is specified in °C. The returned units are $\mathrm{J\,kg^{-1}}$.

```
⟨ Latent Heat Def 29 ⟩ ≡
  double latent_heat(double Tc)
  {
    static double A ← −6.14342 · 10^−5, B ← 1.58927 · 10^−3, C ← −2.36418, D ← 2500.79;
    double Tc2 ← Tc ∗ Tc, Tc3 ← Tc2 ∗ Tc;
    return (A ∗ Tc3 + B ∗ Tc2 + C ∗ Tc + D) ∗ 1000.0;
  }
```

This code is used in section 28.

30. So we know what the rate of change will be for any given position along the temperature/pressure continuum. To draw a moist adiabat, we start with a specified temperature and pressure, and then we integrate up to a new pressure to find what the new temperature should be. We can't use some of the fancier integration methods (i.e., Romberg or Gauss quadrature) since we need to integrate two variables. So we just pick a small step size and hope it will be sufficient for our purposes here. We also make double use of this function: we'll return value of the new temperature as well as calculate (x, y) coordinates, if asked.

```
⟨ Function Defs 5 ⟩ +≡
  double moist_adiabat(double initT, double p, double ∗x, double ∗y)
  {
    static double dp ← 0.25;
    double newT, pi;
    newT ← initT;
    if (p ≠ std_p) {
      if (p > std_p)
        for (pi ← std_p; pi ≤ p; pi += dp) newT += dp ∗ dt_dp(newT, pi);
      else
        for (pi ← std_p; pi ≥ p; pi −= dp) newT −= dp ∗ dt_dp(newT, pi);
    }
    if (x) ∗x ← Tp2x(newT, p);
    if (y) ∗y ← p2y(p);
    return newT;
  }
```

31. Curve fitting. We will convert some of the data points into fitted Beziér curves. To do this, we need a place to store the data points to be fitted.

```
⟨ Global Vars 3 ⟩ +≡
  Point2 points[200];
```

32. Draw a Beziér curve using the PostScript `curveto` operator. PostScript assumes that the first point of the curve is already part of the current path. The `curveto` operator pops two (x, y) control points and the terminating point from the stack, rendering the curve from the current point to the final point. Should the function be called several times, a new curve is considered a continuation of a former curve if the final point of the old curve is the same as the starting point of the new curve. To finish off a curve, call the function once setting $n \leftarrow -1$: this emits the PostScript `stroke` operator and forces the start of a new curve at the next invocation of the function.

DrawBezierCurve( ) is called from the function *FitCurve*( ) whenever that function has finished fitting a portion of the data points to a curve segment. *FitCurve*( ) is defined externally and needs to be linked into this program. We also call *DrawBezierCurve*( ) locally to finish off a curve as described in the previous paragraph.

```
#define UNSET_FLAG −9999.0
⟨ Function Defs 5 ⟩ +≡
  void DrawBezierCurve(int n, Point2 *curve)
  {
    static double lastx ← UNSET_FLAG, lasty ← UNSET_FLAG;
    if (n < 0) {
      printf("␣s\n");
      lastx ← lasty ← UNSET_FLAG;
      return;
    }
    if (curve[0].x ≠ lastx ∨ curve[0].y ≠ lasty) {
      if (lastx > UNSET_FLAG) printf("␣s\n");
      printf("␣n␣%g␣%g␣m\n", curve[0].x, curve[0].y);
    }
    curve++;
    n−−;
    while (n > 0) {
      printf("␣␣%g␣%g␣%g␣%g␣%g␣%g␣c\n",
          curve[0].x, curve[0].y, curve[1].x, curve[1].y, curve[2].x, curve[2].y);
      n −= 3;
      curve += 3;
    }
    lastx ← curve[−1].x;
    lasty ← curve[−1].y;
  }
```

33. Finding Limits. In order to keep the PostScript output as small as possible, we'll try to limit the curves to those that are actually visible within the windowed area. This isn't easily determined for some of the curves, but we'll give it a good shot. To start, we'll calculate the (x, y) coordinates of the visible boundaries.

```
⟨ Global Vars 3 ⟩ +≡
  double minvisx, maxvisx, minvisy, maxvisy;
```

34.
```
⟨ Calculate visible (x, y) boundaries 34 ⟩ ≡
  if (ploty > plotx) {
    maxvisx ← (xmax * plotx)/(2.0 * ploty);
    minvisx ← −maxvisx;
    minvisy ← 0.0;
    maxvisy ← ymax;
  } else {      /* plotx ≥ ploty */
    maxvisx ← xmax/2.0;
    minvisx ← −maxvisx;
    minvisy ← 0.0;
    maxvisy ← (ymax * ploty)/plotx;
  }
```
This code is used in section 53.

35. We can now use the minimum and maximum visible x and y values to calculate the maximum and minimum visible temperatures and pressures. There are four temperature values at each corner of the plot, but only two pressure values at top and bottom.

```
⟨ Global Vars 3 ⟩ +≡
  double minTminp, minTmaxp, maxTminp, maxTmaxp, minvisp, maxvisp;
```

36.
```
⟨ Calculate visible temperatures and pressures 36 ⟩ ≡
  xy2Tp(minvisx, maxvisy, &minTminp, &minvisp);
  xy2Tp(maxvisx, maxvisy, &maxTminp, Λ);
  xy2Tp(minvisx, minvisy, &minTmaxp, &maxvisp);
  xy2Tp(maxvisx, minvisy, &maxTmaxp, Λ);
```
This code is used in section 53.

37. For each of the curves, we'll figure out the low and high index values to calculate. The isotherm indices are pretty easy to calculate. The step size for plotting isotherms and dry adiabats has already been established by setting *tinc*. We add an additional variable for calculating alternating isotherm color stripes.

```
⟨ Global Vars 3 ⟩ +≡
  double isotherm_lo, isotherm_hi, stripe_width ← 10.0;
```

38.
```
⟨ Calculate isotherm indices 38 ⟩ ≡
  isotherm_lo ← round_right(minTminp, tinc);
  isotherm_hi ← round_left(maxTmaxp, tinc);
```
This code is used in section 53.

39. The isobar indices are also easy to calculate.

```
⟨ Global Vars 3 ⟩ +≡
  double isobar_lo, isobar_hi, isobar_inc ← 50.0;
```

40. Note that we're working with indices here, so "low" and "high" have to do with initialization and termination of the **for** loop.

⟨ Calculate isobar indices 40 ⟩ ≡
```
isobar_lo ← round_left(minvisp, isobar_inc);
isobar_hi ← round_left(maxvisp, isobar_inc);
```
This code is used in section 53.

41. We are able to calculate the dry adiabit limits directly.

⟨ Global Vars 3 ⟩ +≡
```
double dryadiabat_lo, dryadiabat_hi;
```

42. ⟨ Calculate dry adiabat indices 42 ⟩ ≡
```
dryadiabat_lo ← C2K(minTmaxp) * pow((std_p/maxvisp), Rd_cpd);
dryadiabat_lo ← round_right(K2C(dryadiabat_lo), tinc);
dryadiabat_hi ← C2K(maxTminp) * pow((std_p/minvisp), Rd_cpd);
dryadiabat_hi ← round_left(K2C(dryadiabat_hi), tinc);
```
This code is used in section 53.

43. Now we'll work on the moist adiabatic curve.

⟨ Global Vars 3 ⟩ +≡
```
double moistadiabat_lo, moistadiabat_hi, moistadiabat_inc ← 1.0;
```

44. Because the moist adiabatic curve is calculated numerically, it isn't possible to calculate the bounds directly, so we'll need to make some guesses. At lower temperatures, the shape of the moist adiabatic curve moves to the left quickly, so we can assume that a lower bound near *minTmaxp* will be close. In higher temperature ranges, the curve starts out moving right and then curves back to the left at higher elevations. Depending on where *maxt* is set, we may need to test the curve at both the maximum and minimum pressure levels to see if the curve is visible in either location, and we'll use the maximum of either of those two values.

⟨ Calculate moist adiabat indices 44 ⟩ ≡
```
moistadiabat_lo ← find_moist_limit(minTmaxp, maxvisp, minTmaxp − 10.0, minTmaxp + 5.0);
moistadiabat_lo ← round_right(moistadiabat_lo, moistadiabat_inc);
moistadiabat_hi ← find_moist_limit(maxTminp, minvisp, maxTmaxp − 20.0, maxTmaxp + 15.0);
moistadiabat_hi ← round_left(MAX(maxTmaxp, moistadiabat_hi), moistadiabat_inc);
```
This code is used in section 53.

45. Another root-finding function.

```
⟨ Function Defs 5 ⟩ +≡
  double find_moist_limit(double targT, double p, double lo, double hi)
  {
    double static tol ← 1.0 · 10^−2;      /* we don't need high precision here */
    double t, testT;
    int iter ← 0, maxiter ← 30;
    do {
      t ← (lo + hi)/2.0;
      testT ← moist_adiabat(t, p, Λ, Λ);
#ifdef DEBUG
      fprintf(stderr, "i=%d␣lo=%g␣hi=%g␣t=%g␣val=%g\n", iter, lo, hi, t, testT);
#endif
      if (testT < targT) lo ← t;
      else hi ← t;
    } while (fabs(testT − targT) > tol ∧ (hi − lo) > tol ∧ ++iter < maxiter);
    if (iter ≥ maxiter) fprintf(stderr, "find_moist_limit():␣max␣iterations␣(%d)␣reached"
        "without␣convergence\n", maxiter);
    return t;
  }
```

46. We have to do something special for indexing when it comes to mixing ratio lines. The spacing is somewhat logarithmic, and we want to choose the lines and spacing that look reasonably good at values that are somewhat "nice". We'll create an array that holds increments and boundary levels. The first element contains the initial value in its *limit*, and the last element should contain a *limit* < 0, indicating that we stick with the current increment level until reset.

```
⟨ Global Vars 3 ⟩ +≡
  double ws_limit;
  struct ws_idx {
    double inc, limit;
  } *wsp, wsidx[] ← {{0.0, 0.1},
      {0.1, 0.3}, {0.2, 0.5}, {0.5, 2.0}, {1.0, 4.0}, {2.0, 12.0},
      {3.0, 15.0}, {5.0, 60.0}, {10.0, 100.0}, {25.0, 200.0}, {50.0, −1.0}, };
```

47. Set things up so that we can use the index array. We'll use this same setup when we draw the mixing ratio labels later.

```
⟨ Function Defs 5 ⟩ +≡
  double ws_init(void)
  {
    wsp ← &(wsidx[1]);
    return wsidx[0].limit;
  }
```

48. This function calculates what the next value of *ws* should be given the current conditions. If $ws \geq wsp\neg limit$, then we bump up to the next increment level; otherwise, we remain at the current increment level. If the $wsp\neg limit < 0$, we've reached the end of the line, and we stay at the current level.

```
⟨ Function Defs 5 ⟩ +≡
  double ws_inc(double ws)
  {
    if (wsp¬limit > 0.0 ∧ ws ≥ wsp¬limit) wsp++;
    return ws + wsp¬inc;
  }
```

49. We can figure out where to stop plotting mixing ratio lines when the x value at the maximum visible pressure exceeds the maximum visible x value for a particular mixing ratio.

```
⟨ Calculate upper mixing ratio limit 49 ⟩ ≡
  {
    double lastws;
    lastws ← ws ← ws_init( );
    while (1) {
      mixing_ratio(ws, maxvisp, &x, &y);
      if (x < maxvisx) {
        lastws ← ws;
        ws ← ws_inc(ws);
      } else break;
    }
    ws_limit ← lastws;
  }
```

This code is used in section 53.

50. We need functions that will round v up or down to the nearest multiple of some value m, so here they are.

```
⟨ Function Defs 5 ⟩ +≡
  double round_left(double v, double m)
  {
    return floor(v/m) ∗ m;
  }
```

51.

```
⟨ Function Defs 5 ⟩ +≡
  double round_right(double v, double m)
  {
    return ceil(v/m) ∗ m;
  }
```

52. Main Program.

```
⟨ Main Program 52 ⟩ ≡
  int main(int argc, char *argv[])
  {
    double x, y, T, p, ws;
    int i;
    ⟨ Pre-calculate variables 21 ⟩
    ⟨ Parse command-line parameters 69 ⟩
    ⟨ Initialization 53 ⟩
    ⟨ Plot Background 55 ⟩
    ⟨ Plot Isobars 56 ⟩
    ⟨ Plot Moist Adiabats 60 ⟩
    ⟨ Plot Isotherms 57 ⟩
    ⟨ Plot Dry Adiabats 58 ⟩
    ⟨ Plot Mixing Ratios 59 ⟩
    ⟨ Plot Labels 61 ⟩
    FinalizeOutput( );
  }
```

This code is used in section 1.

53.

```
⟨ Initialization 53 ⟩ ≡
  calc_ydims(minp, maxp);
  calc_xdims(mint, maxt);
  ⟨ Calculate visible (x, y) boundaries 34 ⟩
  ⟨ Calculate visible temperatures and pressures 36 ⟩
  ⟨ Calculate isotherm indices 38 ⟩
  ⟨ Calculate isobar indices 40 ⟩
  ⟨ Calculate dry adiabat indices 42 ⟩
  ⟨ Calculate moist adiabat indices 44 ⟩
  ⟨ Calculate upper mixing ratio limit 49 ⟩
  ⟨ Show indices 54 ⟩
  InitOutput( );
```

This code is used in section 52.

54. For debugging purposes, show the calculated boundaries and indices.

```
⟨ Show indices 54 ⟩ ≡
#if (TESTING ∨ DEBUG)
  fprintf(stderr, "minvx=%g␣maxvx=%g␣minvy=%g␣maxvy=%g␣minvp=%g␣maxvp=%g\n", minvisx, maxvisx,
      minvisy, maxvisy, minvisp, maxvisp);
  fprintf(stderr, "minTmaxp=%g␣maxTmaxp=%g␣minTminp=%g␣maxTminp=%g\n", minTmaxp, maxTmaxp,
      minTminp, maxTminp);
  fprintf(stderr, "maxp-maxvisp=%g␣minp-minvisp=%g\n", maxp − maxvisp, minp − minvisp);
  fprintf(stderr, "isotlo=%g␣isothi=%g\n", isotherm_lo, isotherm_hi);
  fprintf(stderr, "isoblo=%g␣isobhi=%g\n", isobar_lo, isobar_hi);
  fprintf(stderr, "dalo=%g␣dahi=%g\n", dryadiabat_lo, dryadiabat_hi);
  fprintf(stderr, "malo=%g␣mahi=%g\n", moistadiabat_lo, moistadiabat_hi);
  fprintf(stderr, "ws␣limit=%g\n", ws_limit);
#endif
```

This code is used in section 53.

55. Primary curve plotting. Generate a striped background along isotherms with a width given by *stripe_width*.

```
⟨ Plot Background 55 ⟩ ≡
  printf("%% Background:\ngs greenband\n");
  for (T ← round_left(isotherm_lo, stripe_width); T ≤ round_right(isotherm_hi, stripe_width);
       T += 2 * stripe_width) {
    isotherm(T, maxp, &x, &y);
    printf("  n %g %g m ", x, y);
    isotherm(T, minp, &x, &y);
    printf("%g %g l ", x, y);
    isotherm(T + stripe_width, minp, &x, &y);
    printf("%g %g l cp f\n", x, y);
  }
  printf("gr\n");
```

This code is used in section 52.

56.

```
⟨ Plot Isobars 56 ⟩ ≡
  printf("%% Isobars:\ngs darkgray\n");
  for (p ← isobar_lo; p ≤ isobar_hi; p += isobar_inc) {
    if (fmod(p, 100.0) ≡ 0.0) printf("  lmed ");
    else printf("  lthin ");
    printf("n %g %g m %g %g l s\n", minvisx, p2y(p), maxvisx, p2y(p));
  }
  printf("gr\n");
```

This code is used in section 52.

57.

```
⟨ Plot Isotherms 57 ⟩ ≡
  printf("%% Isotherms:\ngs orange\n");
  for (T ← isotherm_lo; T ≤ isotherm_hi; T += tinc) {
    if (fmod(T, 10.0) ≡ 0.0) printf("  lthick ");
    else if (fmod(T, 5.0) ≡ 0.0) printf("  lmed ");
    else printf("  lthin ");
    isotherm(T, maxp, &x, &y);
    printf("n %g %g m ", x, y);
    isotherm(T, minp, &x, &y);
    printf("%g %g l s\n", x, y);
  }
  printf("gr\n");
```

This code is used in section 52.

58. ⟨ Plot Dry Adiabats 58 ⟩ ≡

```
printf("%% Dry Adiabats:\ngs orange\n");
for (T ← dryadiabat_lo; T ≤ dryadiabat_hi; T += tinc) {
  for (p ← maxp, i ← 0; p ≥ minp; p −= 5.0) {
    dry_adiabat(T, p, &x, &y);
    points[i].x ← x; points[i].y ← y; i++;
    if (x < minvisx) break;
  }
#ifdef DEBUG
  fprintf(stderr, "last p=%g (x,y)=(%g,%g) p2y=%g\n", p, x, y, p2y(p));
#endif
  if (fmod(T, 10.0) ≡ 0.0) printf(" lthick ");
  else if (fmod(T, 5.0) ≡ 0.0) printf(" lmed ");
  else printf(" lthin ");
  FitCurve(points, i, 0.0125);
  DrawBezierCurve(−1, Λ);
}
printf("gr\n");
```

This code is used in section 52.

59. ⟨ Plot Mixing Ratios 59 ⟩ ≡

```
printf("%% Mixing Ratios:\ngs blue lmedth dashed\n");
ws ← ws_init( );
do {
  for (p ← maxp, i ← 0; p ≥ minp; p −= 10.0) {
    mixing_ratio(ws, p, &x, &y);
    points[i].x ← x; points[i].y ← y; i++;
    if (x > maxvisx) break;
  }
  FitCurve(points, i, 0.0125);
  DrawBezierCurve(−1, Λ);
  ws ← ws_inc(ws);
} while (ws ≤ ws_limit);
printf("gr\n");
```

This code is used in section 52.

60. ⟨ Plot Moist Adiabats 60 ⟩ ≡

```
printf("%% Moist Adiabats:\ngs green\n");
for (T ← moistadiabat_lo; T ≤ moistadiabat_hi; T += moistadiabat_inc) {
  for (p ← maxp, i ← 0; p ≥ minp; p −= 10.0) {
    moist_adiabat(T, p, &x, &y);
    points[i].x ← x; points[i].y ← y; i++;
    if (x < minvisx) break;
  }
  if (fmod(T, 10.0) ≡ 0.0) printf(" lthick ");
  else if (fmod(T, 5.0) ≡ 0.0) printf(" lmed ");
  else printf(" lthin ");
  FitCurve(points, i, 0.0125);
  DrawBezierCurve(−1, Λ);
}
printf("gr\n");
```

This code is used in section 52.

61. Plotting labels.

```
⟨ Plot Labels 61 ⟩ ≡
  {
    double axt, xl, xr, yl, yr;
    ⟨ Plot Isotherm Labels 63 ⟩
    ⟨ Plot Dry Adiabat Labels 64 ⟩
    ⟨ Plot Moist Adiabat Labels 65 ⟩
    ⟨ Plot Mixing Ratio Labels 67 ⟩
    ⟨ Plot Isobar Labels 62 ⟩
  }
```

This code is used in section 52.

62. We label the isobars at the left edge of the plot. Nothing too fancy here.

```
⟨ Plot Isobar Labels 62 ⟩ ≡
  printf("black␣10␣/Helvetica-Bold␣fsf\n");
  for (p ← 1000.0; p ≥ 200.0; p −= 100.0) {
    y ← p2y(p);
    printf("0␣1␣(%d)␣el␣10␣unsc␣add␣%g␣chws\n", (int) p, y);
  }
  if (ploty ≥ plotx) {
    y ← p2y(100.0);
    printf("0␣1␣(100)␣el␣10␣unsc␣add␣%g␣6␣unsc␣sub␣chws\n", y);
  }
```

This code is used in section 61.

63. To label the isotherms, we'll pick an axis along which we would like to draw the labels. If we draw an imaginary line from the upper left corner to the lower right corner of the plot, we'll have identified an axis that provides the largest number of visible intersections with the isotherms.

To calculate the intersections, we need the formulae of the axis line and each isotherm line. Using the point/slope form, the equation of the axis line will be $y = m_a(x - A_x) + A_y$ where $m_a = -ploty/plotx$, and a convenient point $A_{x,y}$ is fixed at the lower right-hand corner where $A_y = 0$. We've already calculated the (x, y) boundaries, so we can just set $A_{x,y} = (maxvisx, 0)$.

The point/slope form of an isotherm is $y = (x - T_x) + T_y$, where the slope is 1 by definition, and (T_x, T_y) is some point along the isotherm for temperature T. Again it will be convenient for us to choose a point where $T_y = 0$, which is the at the highest pressure reading in our plot. Now we can set these two equations equal and solve for x:

$$x - T_x = m_a(x - A_x),$$
$$x = \frac{T_x - m_a A_x}{1 - m_a}.$$

We get the corresponding y value by plugging the x value back into either equation, so we'll use $y = x - T_x$ since it's slightly simpler.

⟨ Plot Isotherm Labels 63 ⟩ ≡

```
{
  double ma ← −ploty/plotx, tx;
  printf("orange 14 /Helvetica-Bold fsf\n");
  for (T ← round_right(isotherm_lo, 10.0); T ≤ round_left(isotherm_hi, 10.0); T += 10.0) {
    tx ← Tp2x(T, maxvisp);
    x ← (tx − (ma ∗ maxvisx))/(1.0 − ma);
    y ← x − tx;
    printf("1 1 (%d) %g %g chws\n", (int) T, x, y);
  }
}
```

This code is used in section 61.

64. Plotting dry adiabat labels gets to be a little trickier. Let's say that we want to place dry adiabat labels along an axis that follows a particular isotherm line. Wherever this imaginary axis crosses a dry adiabat, that's where the label will be centered. Recall the equation which produced the dry adiabatic lines:

$$\theta = T\left(\frac{p_0}{p}\right)^{R_d/c_{pd}}.$$

Assuming that we've selected an isotherm line for our axis at temperature T and we know for which adiabat θ we'll be drawing a label, then we need to rearrange this equation to find the pressure p so that we can convert these values into (x, y) coordinates. Fortunately, it just takes a bit of algebra:

$$\begin{aligned}
\ln\frac{\theta}{T} &= \frac{R_d}{c_{pd}}\ln\frac{p_0}{p} \\
\frac{c_{pd}}{R_d}\ln\frac{\theta}{T} &= \ln p_0 - \ln p \\
p &= \exp\left[\ln p_0 - \ln\left(\frac{\theta}{T}\right)^{c_{pd}/R_d}\right] \\
p &= p_0 \Big/ \left(\frac{\theta}{T}\right)^{c_{pd}/R_d}
\end{aligned}$$

Next, we need to determine the angle of rotation on the curve at the selected point. This isn't too hard: we just find two adjacent points on the curve to calculate our "tangent." It will be close enough. PostScript has an appropriate arctangent function which converts dy/dx into a rotation angle. We'll plot labels along two isotherm axes.

⟨ Plot Dry Adiabat Labels 64 ⟩ ≡

```
printf("orange 11 /Helvetica-Bold fsf\n");
axt ← −45.0;
while (axt ≤ 5.0) {
  for (T ← round_right(dryadiabat_lo, 10.0); T ≤ round_left(dryadiabat_hi, 10.0); T += 10.0) {
    p ← std_p/pow(C2K(T)/C2K(axt), cpd/Rd);
    x ← Tp2x(axt, p);
    y ← p2y(p);
    dry_adiabat(T, p − 4, &xl, &yl);
    dry_adiabat(T, p + 4, &xr, &yr);
    printf("%g %g (%d) %g %g chws\n", yr − yl, xr − xl, (int) T, x, y);
  }
  axt += 50.0;
}
```

This code is used in section 61.

65. Once again, the moist adiabats provide the most challenging issues. We can't invert the function to solve it for pressure, so we just have to search for the right temperature and pressure values along a given label axis which intersects the curve. We'll also use the same trick for finding the curve tangent that we used for the dry adiabats.

We start labeling along one isotherm axis, but if we run off the plot in the y dimension before we've finished labeling, we try again on a new isotherm axis 10° to the right. (This may happen if $ploty < plotx$).

```
⟨ Plot Moist Adiabat Labels 65 ⟩ ≡
  printf("green␣11␣/Helvetica-Bold␣fsf\n");
  axt ← −25.0;
  for (T ← round_right(moistadiabat_lo, 5.0); T ≤ round_left(moistadiabat_hi, 5.0); T += 5.0) {
    p ← moist_find(T, axt);
    if (p ≤ minvisp) {
      axt += 10.0;
      p ← moist_find(T, axt);
    }
    x ← Tp2x(axt, p);
    y ← p2y(p);
    moist_adiabat(T, p − 4, &xl, &yl);
    moist_adiabat(T, p + 4, &xr, &yr);
    printf("%g␣%g␣(%d)␣%g␣%g␣chws\n", yr − yl, xr − xl, (int) T, x, y);
  }
```

This code is used in section 61.

66. Use a bracketing method to find a pressure p along a moist adiabatic curve for an initial temperature T which matches the target isotherm specified by a. We initialize the search setting hi_p and lo_p outside the bounds of the lowest and highest pressures in the plot. Should the search end at either extreme, it should theoretically place a label outside of the plot area. This method does not converge particularly quickly, but it's simple.

```
⟨ Function Defs 5 ⟩ +≡
  double moist_find(double T, double a)
  {
    double static tol ← 1.0 · 10^−3;
    double hi_p ← 1200.0, lo_p ← 25.0, p, t;
    int iter ← 0, maxiter ← 30;
    do {
      p ← (hi_p + lo_p)/2.0;
      t ← moist_adiabat(T, p, Λ, Λ);
      if (t < a) lo_p ← p;
      else hi_p ← p;
    } while (fabs(t − a) > tol ∧ fabs(hi_p − lo_p) > tol ∧ ++iter < maxiter);
    if (iter ≥ maxiter) fprintf(stderr, "find_moist():␣max␣iterations␣(%d)␣reached␣without␣"
        "convergence\n", maxiter);
    return p;
  }
```

67. Each label-plotting element seems to have its own challenges. To plot the mixing ratio labels, we'll need to use two different axes. If the mixing ratio curve terminates at with the top boundary of the plot, we'll use one labeling axis. We'll use a different labeling axis if the mixing ratio curve terminates at the right boundary of the plot instead.

When $minvisp \equiv 100$, I found that plotting labels along the $p \equiv 104$ pressure line to be about right. So we try to keep that spacing no matter what the current minimum visible pressure may be. To do that, we calculate the relative y-positions of the 100 and 104 mb levels, and then we subtract that from the current maximum visible y value. We then transform that y value back into a pressure.

```
⟨ Plot Mixing Ratio Labels 67 ⟩ ≡
  {
    double y1 ← p2y(100.0), y2 ← p2y(104.0);
    xy2Tp(0.0, maxvisy − (y1 − y2), Λ, &p);
    if (colormode) printf("blue 10 /Helvetica fsf\n");
    else printf("blue 10 /Helvetica-Bold fsf\n");
    ws ← ws_init( );
    do {
      mixing_ratio(ws, p, &x, &y);
      if (x ≥ maxvisx) {      /* switch to right border */
        x ← maxvisx − 13.0;
        y ← find_mr_y(ws, x);
      }
      mixing_ratio(ws, p + 25, &xl, &yl);
      mixing_ratio(ws, p − 25, &xr, &yr);
      printf("%g %g (%g) %g %g chws\n", yr − yl, xr − xl, ws, x, y);
      ws ← ws_inc(ws);
    } while (ws ≤ ws_limit);
  }
```

This code is used in section 61.

68. If the mixing ratio line intesects with the right border of the plot, we must find a new y coordinate which matches a defined x coordinate.

```
⟨ Function Defs 5 ⟩ +≡
  double find_mr_y(double ws, double x)
  {
    double static tol ← 1.0 · 10^−3;
    double hi_p ← 1200.0, lo_p ← 25.0, p, t, nx ← 0.0, ny ← 0.0;
    int iter ← 0, maxiter ← 50;
    do {
      p ← (hi_p + lo_p)/2.0;
      mixing_ratio(ws, p, &nx, &ny);
      if (nx < x) hi_p ← p;
      else lo_p ← p;
    } while (fabs(nx − x) > tol ∧ fabs(hi_p − lo_p) > tol ∧ ++iter < maxiter);
    if (iter ≥ maxiter) fprintf(stderr, "find_mr_y(): max iterations (%d) reached without "
        "convergence\n", maxiter);
    return ny;
  }
```

69. Command line parsing. Here we handle command-line options to change some default settings as required. Once we've set everything up, all the parsing magic happens in the *getopt*( ) function.

```
⟨ Parse command-line parameters 69 ⟩ ≡
  while (1) {
    int opt_i ← 0, c;
    ⟨ Define long options 70 ⟩
    c ← getopt_long(argc, argv, "bcdDlH:W:x:y:t:T:p:P:?", long_opts, &opt_i);
    if (c ≡ −1) break;
    switch (c) {
    case 0: break;      /* flags already set: no further action required */
    case 'b': colormode ← FALSE; break;
    case 'c': colormode ← TRUE; break;
    case 'l': landscape ← TRUE; break;
    case 'd': tinc ← LORES_T; break;
    case 'D': tinc ← HIRES_T; break;
    case 'H': pgdim_y ← atof(optarg); break;
    case 'W': pgdim_x ← atof(optarg); break;
    case 'x': plotx ← atof(optarg); break;
    case 'y': ploty ← atof(optarg); break;
    case 't': mint ← atof(optarg); break;
    case 'T': maxt ← atof(optarg); break;
    case 'p': minp ← atof(optarg); break;
    case 'P': maxp ← atof(optarg); break;
    case '?':      /* fall through */
      default: showhelp(argv); exit(0);
    }
  }
```

This code is used in section 52.

70. ⟨ Define long options 70 ⟩ ≡

```
  static struct option long_opts[] ← {
      {"help", no_argument, 0, '?'},
      {"color", no_argument, &colormode, TRUE},
      {"bw", no_argument, &colormode, FALSE},
      {"landscape", no_argument, &landscape, TRUE},
      {"portrait", no_argument, &landscape, FALSE},
      {"lowdensity", no_argument, 0, 'd'},
      {"highdensity", no_argument, 0, 'D'},
      {"paperheight", required_argument, 0, 'H'},
      {"paperwidth", required_argument, 0, 'W'},
      {"xsize", required_argument, 0, 'x'},
      {"ysize", required_argument, 0, 'y'},
      {"mint", required_argument, 0, 't'},
      {"maxt", required_argument, 0, 'T'},
      {"minp", required_argument, 0, 'p'},
      {"maxp", required_argument, 0, 'P'},
      {0, 0, 0, 0}};
```

This code is used in section 69.

71. Help message.

```
⟨ Function Defs 5 ⟩ +≡
  void showhelp(char **argv)
  {
    fprintf(stderr, "\nUsage: %s [options] > output.ps\n\n", *argv);
    fprintf(stderr,
        " The following options are recognized (*=default):\n"
        "           --help | -?    : print this message\n"
        "           --color | -c   : generate color plot*\n"
        "           --bw | -b      : generate B&W plot\n"
        "       --landscape | -l   : create plot in landscape mode\n"
        "       --portrait         : create plot in portrait mode*\n"
        "     --lowdensity | -d    : low density temperature spacing\n"
        "    --highdensity | -D    : high density temperature spacing*\n"
        "          --mint t | -t t : lowest temperature value at y=0\n"
        "          --maxt t | -T t : highest temperature value at y=0\n"
        "          --minp p | -p p : minimum pressure value (p > 0)\n"
        "          --maxp p | -P p : maximum pressure value\n"
        " Specified in PostScript units (1 in = 72.0):\n"
        "        --xsize x | -x x  : plot X-size\n"
        "        --ysize y | -y y  : plot Y-size\n"
        "  --paperheight h | -H h  : page height\n"
        "   --paperwidth w | -W w  : page width\n");
  }
```

72. PostScript Header Material. Here we initialize the PostScript output.

```
⟨ Function Defs 5 ⟩ +≡
  void InitOutput(void)
  {
    double outscale;
    char today[50];
    ⟨ Create date string 73 ⟩
    ⟨ Determine scaling factor 74 ⟩
    ⟨ Generate PostScript header 75 ⟩
    ⟨ Create PostScript prolog 77 ⟩
    ⟨ Perform document setup 81 ⟩
    ⟨ Perform page setup 82 ⟩
  }
```

73. Create a date and time string from the current system time. (This is so we can set the `%%CreationDate:` field in the PostScript header.)

```
⟨ Create date string 73 ⟩ ≡
  {
    time_t now;
    struct tm tm_now;
    time(&now);
    localtime_r(&now, &tm_now);
    strftime(today, sizeof (today), "%B␣%d,␣%Y␣%T", &tm_now);
  }
```

This code is used in section 72.

74. So that we know what we're dealing with later, we calculate PostScript scaling factor, which translates dimensions from our data grid coordinates to the output plot on the page. Our data grid remains square, even if the output plot is not, so we have only one scale applied to both the x and y axes.

```
⟨ Determine scaling factor 74 ⟩ ≡
  if (ploty > plotx) outscale ← ploty/ymax;
  else outscale ← plotx/xmax;
```

This code is used in section 72.

75. We'll do our best to adhere to Adobe's document structuring conventions.

```
⟨ Generate PostScript header 75 ⟩ ≡
  printf("%%!PS-Adobe-3.0␣EPSF-3.0\n");
  ⟨ Write out bounding box 76 ⟩
  printf("%%%%Pages:␣1\n%%%%Creator:␣(skew-t.w␣by␣Bret␣Whissel)\n"
      "%%%%Title:␣(Blank␣Skew-T/Log␣p␣Chart)\n"
      "%%%%CreationDate:␣(%s)\n", today);
  printf("%%%%DocumentNeededResources:␣font␣Helvetica␣Helvetica-Bold\n"
      "%%%%DocumentMedia:␣Plain␣%d␣%d␣0␣()␣()\n", (int)(pgdim_x + 0.5), (int)(pgdim_y + 0.5));
  printf("%%%%EndComments\n"
      "%%%%BeginDefaults\n"
      "%%%%PageResources:␣font␣Helvetica␣Helvetica-Bold\n"
      "%%%%EndDefaults\n");
```

This code is used in section 72.

76. The PostScript bounding box is given in terms of the PostScript's natural orientation, so if we are creating a page in landscape mode, we must exchange the x and y dimensions accordingly.

```
⟨ Write out bounding box 76 ⟩ ≡
  if (landscape) printf("%%%%BoundingBox:␣%d␣%d␣%d␣%d\n%%%%Orientation:␣Landscape\n",
        (int)(pgdim_x − transy − ploty + 0.5), (int)(transx + 0.5), (int)(pgdim_x − ploty),
        (int)(transx + plotx + 0.5));
  else printf("%%%%BoundingBox:␣%d␣%d␣%d␣%d\n%%%%Orientation:␣Portrait\n", (int)(transx + 0.5),
        (int)(transy + 0.5), (int)(transx + plotx + 0.5), (int)(transy + ploty + 0.5));
```

This code is used in section 75.

77.

```
⟨ Create PostScript prolog 77 ⟩ ≡
  printf("%%%%BeginProlog\n"
      "50␣dict␣begin\n");
  ⟨ Create PostScript shortcuts 78 ⟩
  ⟨ Create color definitions 79 ⟩
  ⟨ Create linewidth definitions 80 ⟩
  ⟨ Create PostScript procedures 87 ⟩
  printf("%%%%EndProlog\n");
```

This code is used in section 72.

78. We use several PostScript operators over and over, and to reduce output size, we create brief shortcuts here.

```
⟨ Create PostScript shortcuts 78 ⟩ ≡
  printf("/bd{bind␣def}bind␣def␣/ed{exch␣def}bd\n"
      "/gs{gsave}bd␣/gr{grestore}bd\n"
      "/tr{translate}bd␣/sc{scale}bd␣/r{rotate}bd\n"
      "/n{newpath}bd␣/cp{closepath}bd\n"
      "/s{stroke}bd␣/f{fill}bd\n"
      "/m{moveto}bd␣/l{lineto}bd␣/c{curveto}bd\n"
      "/rm{rmoveto}bd␣/rl{rlineto}bd\n"
      "/slw{setlinewidth}bd\n");
```

This code is used in section 77.

79.

```
⟨ Create color definitions 79 ⟩ ≡
  if (colormode) printf("/orange{0.902␣0.404␣0.082␣setrgbcolor}def\n"
        "/green{0.439␣0.800␣0.255␣setrgbcolor}def\n"
        "/greenband{0.95␣1.0␣0.95␣setrgbcolor}def\n"
        "/blue{0␣0␣0.5␣setrgbcolor}def\n");
  else printf("/orange{0.4␣setgray}def\n"
        "/green{0.65␣setgray}def\n"
        "/greenband{0.96␣setgray}def\n"
        "/blue{0.3␣setgray}def\n");
  printf("/darkgray{0.25␣setgray}def\n"
      "/black{0␣setgray}def\n"
      "/white{1␣setgray}def\n");
```

This code is used in section 77.

80. Here we specify line thickness in terms of PostScript units. However, when these settings are invoked, we've already established the plot's scaling, so we must undo the scaling first. The scaling factor is saved in *sf*. The lines are drawn a little thicker in black-and-white mode.

```
⟨ Create linewidth definitions 80 ⟩ ≡
  printf("/sf{%0.6lf}def\n", outscale);
  printf("/unsc{sf␣div}def\n"
      "/dashed{[5␣unsc␣3␣unsc]0␣setdash}def\n");
  if (colormode) printf("/lthicker{1.4␣unsc␣slw}def␣"
        "/lthick{0.8␣unsc␣slw}def\n"
        "/lmed{0.55␣unsc␣slw}def␣"
        "/lmedth{0.35␣unsc␣slw}def\n"
        "/lthin{0.15␣unsc␣slw}def\n");
  else printf("/lthicker{1.7␣unsc␣slw}def␣"
        "/lthick{1.5␣unsc␣slw}def\n"
        "/lmed{0.9␣unsc␣slw}def␣"
        "/lmedth{0.75␣unsc␣slw}def\n"
        "/lthin{0.6␣unsc␣slw}def\n");
```

This code is used in section 77.

81. This segment is only to be Adobe DSC compliant.

```
⟨ Perform document setup 81 ⟩ ≡
  printf("%%%%BeginSetup\n"
      "%%%%IncludeResources:␣font␣Helvetica\n"
      "%%%%IncludeResources:␣font␣Helvetica-Bold\n"
      "%%%%EndSetup\n");
```

This code is used in section 72.

82. ⟨ Perform page setup 82 ⟩ ≡

```
  printf("%%%%Page:␣1␣1\n"
      "%%%%PageResources:␣font␣Helvetica␣Helvetica-Bold\n"
      "%%%%BeginPageSetup\n/pgsave␣save␣def\n");
  ⟨ If in landscape mode, reset origin and rotation 83 ⟩
  ⟨ Set up the clipping path 84 ⟩
  ⟨ Set up conversion from data coordinates to PostScript domain 85 ⟩
  ⟨ Calculate edges in data coordinates 86 ⟩
  printf("%%%%EndPageSetup\n");
```

This code is used in section 72.

83. If we are creating a landscape mode page, we must translate the origin to the right edge of the page and then rotate 90°.

```
⟨ If in landscape mode, reset origin and rotation 83 ⟩ ≡
  if (landscape) {
    printf("%g␣%g␣tr\n", pgdim_x, 0.0);
    printf("90␣r\n");
  }
```

This code is used in section 82.

84. ⟨ Set up the clipping path 84 ⟩ ≡

```
  printf("gs␣%g␣%g␣tr\n", transx, transy);
  printf("n␣0␣0␣m␣%g␣%g␣rl␣%g␣%g␣rl␣%g␣%g␣rl␣cp␣gs␣clip\n", plotx, 0.0, 0.0, ploty, -plotx, 0.0);
```

This code is used in section 82.

85. We've already calculated the scaling factor. However, before we invoke scaling, we shift the x axis so that $x = 0$ is located in the middle of the plot.

```
⟨ Set up conversion from data coordinates to PostScript domain 85 ⟩ ≡
  printf("%g␣0␣tr␣sf␣dup␣sc\n", plotx/2.0);
```

This code is used in section 82.

86. Some of the labels need to know where the edges of the plot are located. We've already calculated these, so now we just need to pass them along to the PostScript engine.

```
⟨ Calculate edges in data coordinates 86 ⟩ ≡
  printf("/el{%g}def␣/er{%g}def␣/eb{%g}def␣/et{%g}def\n",
      minvisx, maxvisx, minvisy, maxvisy);
```

This code is used in section 82.

87. Create a procedure that prints a character string centered horizontally and vertically at requested coordinates and rotation angle. An example invocation looks like the following:

```
dy dx (label) x y chws
```

First, we move to the label's location, then we acquire the string's bounding box. Then we rotate to be parallel to the curve's tangent at that point. Next we stroke the string's character outlines in white to clear space around the label characters, and then we fill the character outline.

```
⟨ Create PostScript procedures 87 ⟩ ≡
  printf("/chws{gs␣n␣m␣dup\n"
      "␣␣gs␣n␣0␣0␣m␣false␣charpath␣flattenpath␣pathbbox␣gr\n"
      "␣␣4␣dict␣begin␣/ury␣ed␣/urx␣ed␣/lly␣ed␣/llx␣ed\n"
      "␣␣3␣1␣roll␣atan␣r␣urx␣llx␣sub␣-2␣div␣ury␣lly␣sub␣-2␣div␣rm\n"
      "␣␣dup␣gs␣false␣charpath␣white␣ury␣lly␣sub␣0.18␣mul␣slw␣s␣gr\n"
      "␣␣true␣charpath␣fill␣end␣gr}bd\n");
```

See also section 88.

This code is used in section 77.

88. This procedure is merely a shortcut for setting up a font. It is invoked this way:

```
size /Fontname fsf
```

Since the font is drawn within the scaled domain, the fontsize is first unscaled before being applied.

```
⟨ Create PostScript procedures 87 ⟩ +≡
  printf("/fsf{findfont␣exch␣unsc␣scalefont␣setfont}bd\n");
```

89. Now we wrap up the plot.

```
⟨ Function Defs 5 ⟩ +≡
  void FinalizeOutput(void)
  {
    printf("gr␣1.4␣slw␣s␣gr\n"
        "pgsave␣restore\nshowpage\n"
        "%%%%PageTrailer\n%%%%Trailer\n"
        "end\n%%%%EOF\n");
  }
```

90. Index.

Skew-T

(Version 1.0)

Bret D. Whissel