Oliver Montenbruck Thomas Pfleger

Astronomy on the Personal Computer

Translated by Storm Dunlop
With a Foreword by Richard M. West

Fourth, Completely Revised Edition
With 46 Figures and CD-Rom

Springer

ISBN 3-540-67221-4

a high level of abstraction, and gives optimal encapsulation of the necessary data. Tasks such as the preparation of workspace are no longer the user's responsibility, but are independently carried out by the constructor of the corresponding objects.

2. Coordinate Systems

Astronomy uses a whole series of different coordinate systems to specify the positions of stars and planets. Here we must distinguish between heliocentric coordinates, which are based on the Sun, and geocentric coordinates, which are centred on the Earth. Ecliptic coordinates refer the position of a point to the plane of the Earth's orbit, while equatorial coordinates are measured relative to the position of the Earth's or celestial equators. The slow shift of this reference plane because of precession means that the equinox of the coordinate system that is being used also has to be taken into account. Although it would obviously be possible to decide, once and for all, to use a single, fixed coordinate system, every system has specific advantages that make it particularly suitable for certain purposes.

Because there are various coordinate systems, it is often necessary to convert coordinates given in one system to those of another. The `Coco` ('Coordinate Conversion') program is designed to do just this. It provides accurate conversions between ecliptic and equatorial coordinates, and between geocentric and heliocentric coordinates, for various epochs. The individual transformations, which will be described shortly, are written as separate functions. They may therefore be used elsewhere, and are an important basis for later programs.

2.1 Making a Start

The C++ programming language has at its disposal powerful standard libraries that already contain many mathematical functions. Nevertheless, it proves very useful for subsequent work if we start by defining our own basic library modules. The first of these contains the definitions of important constants that will be repeatedly required subsequently. The corresponding module `APC_Const` consists solely of a header file `APC_Const.h`, which may be passed to other modules by the use of an `#include` instruction. Apart from mathematical constants such as the quantity π and the factors required for the conversion of degrees and angular measure, this contains astronomical and physical constants, such as the length of the astronomical unit, and the speed of light.

```
const double pi      = 3.14159265358979324;
const double pi2     = 2.0*pi;
const double Rad     = pi / 180.0;
const double Deg     = 180.0 / pi;
const double Arcs    = 3600.0*180.0/pi;
...
```

```
const double AU      = 149597870.0;    // [km]
const double c_light = 173.14;         // [AU/d]
```

The module APC_Math provides (among others) the two functions

```
//-------------------------------------------------------------------------
// Frac: Gives the fractional part of a number
//-------------------------------------------------------------------------
double Frac ( double x )
{
  return x-floor(x);
}
```

and

```
//-------------------------------------------------------------------------
// Modulo: calculates x mod y
//-------------------------------------------------------------------------
double Modulo ( double x, double y )
{
  return y*Frac(x/y);
}
```

which enable us to calculate the fractional and integral parts of a number. These functions are, for example, repeatedly required, when one wants to express the monotonically increasing longitude of a celestial body as lying between 0° bis 360°.

In addition, APC_Math contains two functions Ddd and DMS, which support the sexagesimal expression of times and angles. In specifying angles in degrees, fractions of a degree are normally given in minutes and seconds of arc. For time, this corresponds to the normal use of minutes and seconds. Conversion of the two forms is normally required once (at most) when a value is input or ouput. Provided all values are positive, this, in principle, causes no difficulty. It is, however, important to take precautions to ensure that the sign is correctly handled with negative angles. The action of the functions Ddd and DMS is best explained by a few examples:

```
      Dd        D  M  S

 15.50000    15 30 00.0
 -8.15278    -8 09 10.0
  0.01667     0  1  0.0
 -0.08334     0 -5  0.0
```

In converting a negative number Dd into degrees, minutes and seconds, only the leading (non-zero) figure of the three numbers D, M and S is negative. It should also be noted that in both functions D and M, which by the very nature of things take integer values only, are defined as int variables, whereas S is a double variable.

```
//-------------------------------------------------------------------------
// Ddd: Conversion of degrees, minutes and seconds of arc to decimal
//      representation of an angle
//   D,M,S     Degrees, minutes and seconds of arc
//   <return>  Angle in decimal representation
//-------------------------------------------------------------------------
double Ddd( int D, int M, double S )
{
  double sign;
  if ( (D<0) || (M<0) || (S<0) ) sign = -1.0; else sign = 1.0;
  return  sign * ( fabs((double)D)+fabs((double)M)/60.0+fabs(S)/3600.0 );
}

//-------------------------------------------------------------------------
// DMS: Finds degrees, minutes and seconds of arc for a given angle
//   Dd        Angle in degrees in decimal representation
//   D,M,S     Degrees, minutes and seconds of arc
//-------------------------------------------------------------------------
void DMS ( double Dd, int& D, int& M, double& S )
{
  double x;
  x = fabs(Dd);   D = int(x);
  x = (x-D)*60.0; M = int(x);  S = (x-M)*60.0;
  if (Dd<0.0) { if (D!=0) D*=-1; else if (M!=0) M*=-1; else S*=-1.0; }
}
```

Because the conversion of angles into degrees, minutes and seconds is almost exclusively required for output, it is worthwhile combining both functions in a suitable manner. To do so, we first make use of the possibility offered in C++ of defining a suitable class:

```
// Format tag for Angle output (used by Angle class output operator)
enum AngleFormat {
  Dd,    // decimal representation
  DMM,   // degrees and whole minutes of arc
  DMMm,  // degrees and minutes of arc in decimal representation
  DMMSS, // degrees, minutes of arc and whole seconds of arc
  DMMSSs // degrees, minutes, and seconds of arc in decimal representation
};

//
// Auxiliary class for sexagesimal angle output
//
class Angle
{
  public:
    // Constructor
    Angle (double alpha, AngleFormat Format=Dd);
    // Modifiers
    void Set (AngleFormat Format=Dd);
    // Angle output
    friend ostream& operator << (ostream& os, const Angle& alpha);
  private:
    double       m_angle;
    AngleFormat  m_Format;
};
```

Objects of the Angle class may be created by the equally named constructor by specifying the angle and a format descriptor. The corresponding data are stored in the class attributes and used for later output by employing the shift operator <<. Before doing so the angle is converted into degrees, minutes and seconds as required. Finally the resulting values are output, individually formatted. The definition of the shift operator implements the common manipulators for the number of places, sign, etc., so that the output may be easily laid out within broad limits. Because of the differences between the cases for the individual formats, the implementation of the class is quite comprehensive and will not, therefore, be discussed in detail here. Its application is, however, extremely simple, as the following short program shows:

```
#include <iostream>
#include <iomanip>
#include "APC_Math.h"

void main() {

  using namespace std;
  double x=12.3456;

  cout << setprecision(2) << setw(12) << Angle(x,Dd) << endl;
  cout << setprecision(2) << setw(12) << Angle(x,DMM) << endl;
  cout << setprecision(2) << setw(12) << Angle(x,DMMm) << endl;
  cout << setprecision(2) << setw(12) << Angle(x,DMMSS) << endl;
  cout << setprecision(2) << setw(12) << Angle(x,DMMSSs) << endl;

}
```

The output for the angle 12°3456 is represented in five different forms:

```
       12.35
       12 21
    12 20.74
    12 20 44
 12 20 44.16
```

Any sign would be printed left justified.

We should also introduce a third module here, the APC_VecMat3D library, which provides two classes for three-dimensional vectors and matrices as well as numerous related operators and functions. Three-dimensional vectors serve to describe the position of a point $\boldsymbol{r}$ in space, where, in addition to the Cartesian coordinates $\boldsymbol{r} = (x, y, z)$, the polar coordinates r, ϑ and φ may also be employed (Fig. 2.1). The relationship between the two forms is given by

$$\begin{aligned} x &= r\cos\vartheta\cos\varphi & r &= \sqrt{x^2+y^2+z^2} \\ y &= r\cos\vartheta\sin\varphi & \tan\varphi &= y/x \\ z &= r\sin\vartheta & \tan\vartheta &= z/\sqrt{x^2+y^2} \end{aligned} \quad . \tag{2.1}$$

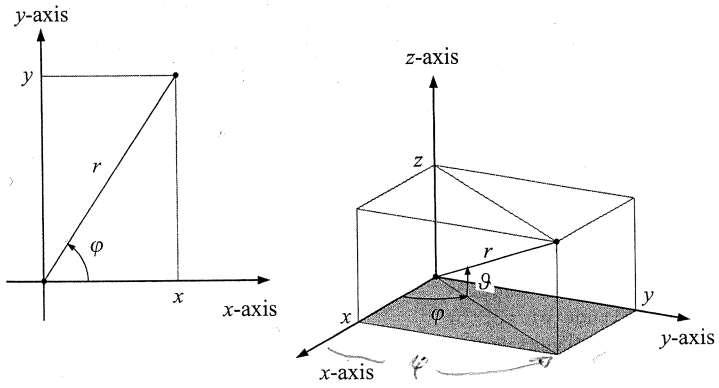

Fig. 2.1. Plane and spatial polar coordinates

The class Vec3D allows both representations to be employed alongside one another in a simple fashion. To this end, the corresponding constructors Vec3D and the access operators [] are provided:

```
//
// Vec3D: three dimensional vectors
//
class Vec3D
{
  public:
    // Constructors
    Vec3D ();  // default constructor initializes to zero vector
    Vec3D (double X, double Y, double Z);
    Vec3D (const Polar& polar);
    // component access (read only)
    double operator [] (xyz_index Index) const;
    // retrieves polar angles or norm of vector
    double operator [] (pol_index Index);
    // simple vector output
    friend ostream& operator << (ostream& os, const Vec3D& Vec);
    // further operators and functions
    ...
  private:
    // Members
    double m_Vec[3];        // components of vector
    double m_phi;           // polar angle (azimuth)
    double m_theta;         // polar angle (altitude)
    double m_r;             // norm of vector
    bool   m_bPolarValid;   // flag for validity of polar coordinates
    // On-demand calculation of polar components
    void CalcPolarAngles ();
};
```

Here the data types index, pol_index and Polar are defined as

```
enum xyz_index { x=0, y=1, z=2 };
enum pol_index { phi=0, theta=1, r=2 };  // azimuth, altitude, radius
```

and

```
struct Polar {
  // Constructors
  Polar();
  Polar(double Az, double Elev, double R = 1.0);
  // Members
  double phi;    // azimuth of vector
  double theta;  // altitude of vector
  double r;      // norm of vector
};
```

Conversion of Cartesian coordinates is carried out using the private function

```
//
// Calculate polar components
//
void Vec3D::CalcPolarAngles ()
{
  // Length of projection in x-y-plane:
  const double rhoSqr = m_Vec[0] * m_Vec[0] + m_Vec[1] * m_Vec[1];
  // Norm of vector
  m_r = sqrt ( rhoSqr + m_Vec[2] * m_Vec[2] );
  // Azimuth of vector
  if ( (m_Vec[0]==0.0) && (m_Vec[1]==0.0) )
    m_phi = 0.0;
  else
    m_phi = atan2 (m_Vec[1], m_Vec[0]);
  if ( m_phi < 0.0 )  m_phi += 2.0*pi;
  // Altitude of vector
  const double rho = sqrt ( rhoSqr );
  if ( (m_Vec[2]==0.0) && (rho==0.0) )
    m_theta = 0.0;
  else
    m_theta = atan2(m_Vec[2], rho);
}
```

as soon as polar coordinates are accessed for the first time with the `[]` operator. Finally the status variable `m_bPolarValid` is set to true, so that the conversion is at most carried out once. It should be noted that angles are required, and will be calculated, in radians and are to be converted on input and output. `APC_Const` provides the constants `Deg` $= 180°/\pi$ and `Rad` $= \pi/180°$ for this purpose.

As well as the basic constructors and access operators, the `Vec3D` class includes numerous operators and functions, that allow vector calculations to be carried out easily. For example, vectors may be added by using the + operator, while the * operator allows the multiplication of a vector with a scalar (see Table 2.1).

The class `Vec3D` is complimented by the three-dimensional matrix class `Mat3D`, which is defined as follows:

Table 2.1. Vector- und matrix operations in `APC_VecMat3D`

Name	Arg$_1$ (a)	Arg$_2$ (b)	Value (c)	Notation	Explanation
-	Vec3D		Vec3D	$\boldsymbol{c} = -\boldsymbol{a}$	Unary minus
	Mat3D		Mat3D	$\boldsymbol{C} = -\boldsymbol{A}$	
+	Vec3D	Vec3D	Vec3D	$\boldsymbol{c} = \boldsymbol{a} + \boldsymbol{b}$	Vector addition
	Mat3D	Mat3D	Mat3D	$\boldsymbol{C} = \boldsymbol{A} + \boldsymbol{B}$	Matrix addition
*	double	Vec3D	Vec3D	$\boldsymbol{c} = a\boldsymbol{b}$	Scalar multiplication
	Vec3D	double	Vec3D	$\boldsymbol{c} = \boldsymbol{a}b$	
	double	Mat3D	Mat3D	$\boldsymbol{C} = a\boldsymbol{B}$	
	Mat3D	double	Mat3D	$\boldsymbol{C} = \boldsymbol{A}b$	
/	Vec3D	double	Vec3D	$\boldsymbol{c} = \boldsymbol{a}/b$	Scalar division
	Mat3D	double	Mat3D	$\boldsymbol{C} = \boldsymbol{A}/b$	
*	Mat3D	Vec3D	Vec3D	$\boldsymbol{c} = \boldsymbol{A}\boldsymbol{b}$	Matrix/vector product
	Vec3D	Mat3D	Vec3D	$\boldsymbol{c} = \boldsymbol{a}\boldsymbol{B}$	
Norm	Vec3D		double	$c = \lvert\boldsymbol{a}\rvert$	Euclidean norm
Dot	Vec3D	Vec3D	double	$c = \boldsymbol{a}^T\boldsymbol{b}$	Dot product
Cross	Vec3D	Vec3D	Vec3D	$\boldsymbol{c} = \boldsymbol{a} \times \boldsymbol{b}$	Cross product
Id3D			Mat3D	$\mathbf{1}$	Identity matrix
R_x	double		Mat3D	$\boldsymbol{R}_x(a)$	Elementary rotation
R_y	double		Mat3D	$\boldsymbol{R}_y(a)$	matrix
R_z	double		Mat3D	$\boldsymbol{R}_z(a)$	

```
//
// Mat3D: 3 dimensional transformation matrices
//

class Mat3D
{
  public:
    // Constructors (empty matrix; matrix from column vectors)
    Mat3D ();
    Mat3D ( const Vec3D& e_1, const Vec3D& e_2, const Vec3D& e_3 );
    // component access
    friend Vec3D Col(const Mat3D& Mat, xyz_index Index);
    friend Vec3D Row(const Mat3D& Mat, xyz_index Index);
    // simple matrix output
    friend ostream& operator << (ostream& os, const Mat3D& Mat);
    // further operators and functions
    ...

  private:
    double m_Mat[3][3];   // Elements of the matrix
};
```

Whereas – as mentioned – objects in the vector class describe spatial coordinates, in what follows, objects in the matrix class primarily serve to describe the rotation of coordinate systems in space. Simple rotations around the x-, y-, or z-axes, may be obtained using the `R_x`, `R_y` and `R_z` functions, which, for a given rotation angle

ϕ, calculate the following matrices:

$$\boldsymbol{R}_x(\phi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & +\cos\phi & +\sin\phi \\ 0 & -\sin\phi & +\cos\phi \end{pmatrix} \tag{2.2}$$

$$\boldsymbol{R}_y(\phi) = \begin{pmatrix} +\cos\phi & 0 & -\sin\phi \\ 0 & 1 & 0 \\ +\sin\phi & 0 & +\cos\phi \end{pmatrix} \tag{2.3}$$

$$\boldsymbol{R}_z(\phi) = \begin{pmatrix} +\cos\phi & +\sin\phi & 0 \\ -\sin\phi & +\cos\phi & 0 \\ 0 & 0 & 1 \end{pmatrix} . \tag{2.4}$$

The signs here are chosen so that a positive angle corresponds to a positive rotation (i.e., an anticlockwise rotation) of the reference frame around the rotational axis.

2.2 Calendar and Julian Dates

When calculating ephemerides, the difference in time between two given dates is generally required. A running day number, as has been commonly used in astronomy for a very long time, is therefore extremely convenient. The Julian Date gives the total number of days that have elapsed since 4713 B.C. January 1. It is named after Julius Scaliger, the father of Joseph Justus Scaliger, who was the first to use it for chronological purposes. As the count begins in biblical times, the Julian Date has now reached a very high value. At noon on 1980 July 23 it amounted to 2 444 444.0 days, for example. When we recall that a second is approximately equal to 0.00001 day, then we find that we require a twelve-figure Julian Date to express an accurate time. The first two numbers hardly alter over a period of three centuries, however, and therefore the Modified Julian Date

$$\text{MJD} = \text{JD} - 2400000.5$$

is employed alongside the true Julian Date. The MJD is the total number of days that has elapsed since 1858 November 17 00:00 hours. Like the ordinary civil date, the MJD therefore changes at midnight, and not at noon like the Julian Date.

```
//-----------------------------------------------------------------------
//
// Mjd: Modified Julian Date from calendar date and time
//   Year,Month,Day  Calendar date components
//   Hour,Min,Sec    Time components (optional)
//   <return>        Modified Julian Date
//
//-----------------------------------------------------------------------
double Mjd ( int Year, int Month, int Day,
             int Hour=0, int Min=0, double Sec=0.0 )
{
  // Variables
  long    MjdMidnight;
  double  FracOfDay;
  int     b;

  if (Month<=2) { Month+=12; --Year;}
  if ( (10000L*Year+100L*Month+Day) <= 15821004L )
    b = -2 + ((Year+4716)/4) - 1179;      // Julian calendar
  else
    b = (Year/400)-(Year/100)+(Year/4);  // Gregorian calendar

  MjdMidnight = 365L*Year - 679004L + b + int(30.6001*(Month+1)) + Day;
  FracOfDay   = Ddd(Hour,Min,Sec) / 24.0;

  return MjdMidnight + FracOfDay;
}
```

`Mjd` takes into account the fact that, because of the Gregorian calendar reform, 1582 October 4 (JD 2299159.5) was followed by 1582 October 15 (JD 2299160.5).[1] Until that date the Julian calendar is used, in which every fourth year included February 29. After that date, leap days are omitted in every century year that is not evenly divisible by 400. As a result, the average length of the year according to the Gregorian calendar is

$$365 + 1/4 - 1/100 + 1/400 = 365.2425 \text{ days},$$

slightly shorter than the 365.25 days of the Julian year.

`Caldat` is a function to calculate the normal calendar date from the Modified Julian Date. Because it is closely linked to the subject of this section it will be given here.

```
//-----------------------------------------------------------------------
// CalDat: Calendar date and time from Modified Julian Date
//   Mjd             Modified Julian Date
//   Year,Month,Day  Calendar date components
//   Hour            Decimal hours
//-----------------------------------------------------------------------
void CalDat ( double Mjd,
              int& Year, int& Month, int& Day, double & Hour )
{
  long    a,b,c,d,e,f;
```

[1] Caution needs to be exercised, because only Italy, Spain and Portugal adopted the new calendar on this date. Most other Catholic states in Continental Europe changed at various dates in 1583 and 1584, with Protestant states and cities following in the following century, most by 1700–1701. Great Britain, its colonies and dominions changed on 1752 September 3 (which was followed by September 14). A full listing is given in the *Explanatory Supplement to the Astronomical Ephemeris and the American Ephemeris and Nautical Almanac*, published jointly by the U. S. Government Printing Office, Washington, and Her Majesty's Stationery Office, London, in 1974, pp.412–416.

```
  double  FracOfDay;
  // Convert Julian day number to calendar date
  a = long(Mjd+2400001.0);
  if ( a < 2299161 ) {  // Julian calendar
    b = 0;  c = a + 1524;
  }
  else {                // Gregorian calendar
    b = long((a-1867216.25)/36524.25);
    c = a +  b - (b/4) + 1525;
  }
  d     = long ( (c-122.1)/365.25 );
  e     = 365*d + d/4;
  f     = long ( (c-e)/30.6001 );
  Day   = c - e - int(30.6001*f);
  Month = f - 1 - 12*(f/14);
  Year  = d - 4715 - ((7+Month)/10);
  FracOfDay = Mjd - floor(Mjd);
  Hour = 24.0*FracOfDay;
}
//---------------------------------------------------------------------------
// CalDat: Calendar date and time from Modified Julian Date
//   Mjd             Modified Julian Date
//   Year,Month,Day  Calendar date components
//   Hour,Min,Sec    Time components
//---------------------------------------------------------------------------
void CalDat ( double Mjd,
              int& Year, int& Month, int& Day,
              int& Hour, int& Min, double& Sec ) {
  double Hours;
  CalDat (Mjd, Year, Month, Day, Hours);
  DMS (Hours, Hour, Min, Sec);
}
```

The two overloaded versions of `CalDat` differ in the way they handle the fraction of a day since midnight, which is given either in decimal hours or in hours, minutes, and seconds. As with the sexagesimal expression of angles, the latter form is generally required only for the output of times. The `APC_Time` module therefore provides, alongside `Mjd` and `CalDat`, two special classes `Time` and `DateTime`, with the corresponding constructors

```
  Time ( double Hour, TimeFormat Format=HHMMSS );
  DateTime ( double Mjd, TimeFormat Format=None );
```

which provide an even better means of carrying out this task. The enumeration type

```
enum TimeFormat {
  None,   // don't output time (date only)
  DDd,    // output time as fractional part of a day
  HHh,    // output time as hours with one decimal place
  HHMM,   // output time as hours and minutes (rounded to the next minute)
  HHMMSS  // output time as hours, minutes and seconds (rounded to next s)
};
```

here serves to establish the format and the required rounding for the output of `Time` and `DateTime` objects, using the shift operator (<<). Here is an example:

```
double MJD = Mjd ( 1961,01,14, 03,30,10.0 ); // 14 Jan. 1961, 3:30:10
double JD  = MJD + 2400000.5;                 // Julian Date
cout << setprecision(1)
     << "Date:" << DateTime(MJD,HHMMSS)       // Output
     << setprecision(3)
     << "MJD: " << setw(12) << MJD
     << "JD:  " << setw(12) << JD;
```

2.3 Ecliptic and Equatorial Coordinates

Ecliptic and equatorial coordinates differ in the reference plane from which they are measured. In the former case, the (x, y) plane is the Earth's orbital plane (the *ecliptic*), and in the latter, the plane perpendicular to the Earth's axis, i.e., parallel to the plane of the Earth's equator (Fig. 2.2). The x-x'-axis is common to both systems, and is defined as being the direction of the vernal equinox or *First Point of Aries*, designated by (Υ). This direction is perpendicular to the north pole of the ecliptic (the z-axis) and to the North Celestial Pole (the z'-axis). The angle ε between the ecliptic and the equator amounts to approximately 23°.5.

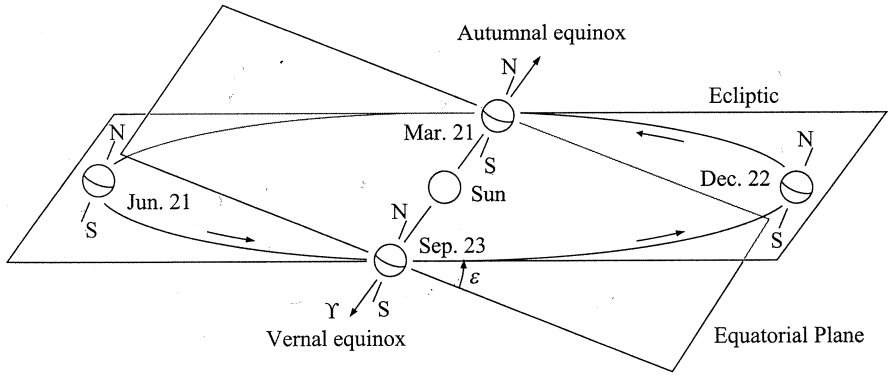

Fig. 2.2. Ecliptic and equator

What are the relationships between the ecliptic coordinates (x, y, z) and the equatorial coordinates (x', y', z') of a point? As $x = x'$, only the transformations $(y, z) \leftrightarrow (y', z')$ have to be considered. A point on the y-axis ($z = 0$) has equatorial coordinates $y' = +y\cos\varepsilon$ and $z' = +y\sin\varepsilon$. If the point lies on the z-axis, then its equatorial coordinates are $y' = -z\sin(\varepsilon)$ and $z' = +z\cos\varepsilon$. In the equatorial system, therefore, a given point (x, y, z) has the components

$$\begin{aligned} x' &= +x \\ y' &= +y\cos\varepsilon - z\sin\varepsilon \\ z' &= +y\sin\varepsilon + z\cos\varepsilon \quad . \end{aligned} \tag{2.5}$$

The corresponding inverse relationships are

$$\begin{aligned} x &= +x' \\ y &= +y'\cos\varepsilon + z'\sin\varepsilon \\ z &= -y'\sin\varepsilon + z'\cos\varepsilon \quad . \end{aligned} \tag{2.6}$$

The polar coordinates corresponding to (x, y, z) are ecliptic longitude l, ecliptic latitude b and distance r (see Fig. 2.3).

$$\begin{aligned} x &= r\cos b\cos l & \qquad x' &= r\cos\delta\cos\alpha \\ y &= r\cos b\sin l & \qquad y' &= r\cos\delta\sin\alpha \\ z &= r\sin b & \qquad z' &= r\sin\delta \quad . \end{aligned} \tag{2.7}$$

The equatorial coordinates corresponding to these are right ascension α, declination δ, and again distance r, which has the same value in both systems.

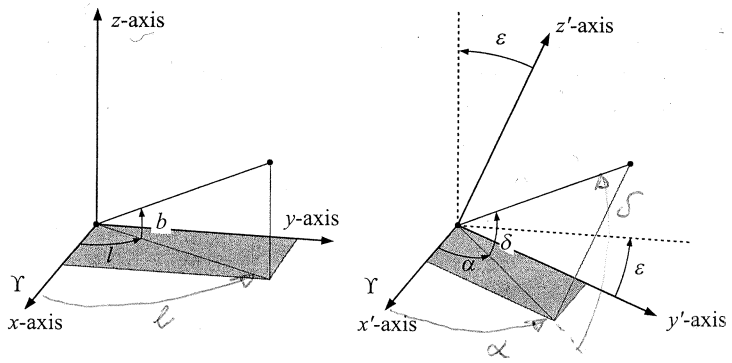

Fig. 2.3. Ecliptic and equatorial coordinates

The transformations between ecliptic and equatorial coordinates may be described in a particularly compact and clear manner, when expressed in vector and matrix notation. Using the elementary rotational matrices introduced earlier, we have

$$\boldsymbol{r} = \boldsymbol{R}_x(+\varepsilon)\,\boldsymbol{r}' \qquad \text{and} \qquad \boldsymbol{r}' = \boldsymbol{R}_x^T(+\varepsilon)\,\boldsymbol{r} = \boldsymbol{R}_x(-\varepsilon)\,\boldsymbol{r} \quad , \tag{2.8}$$

where $\boldsymbol{R}_x^T$ indicates the transposition (reflection) of the matrix $\boldsymbol{R}_x$. Similarly, we obtain the transformed coordinates of a vector of type `Vec3D` by multiplying the corresponding rotational matrix. Here is another short sample program:

```
double eps = Rad*23.5;                  // Obliquity of ecliptic in [rad]
Vec3D  e   = Vec3D(1.0,2.0,3.0);        // Ecliptic coordinates
Vec3D  a   = R_x(-eps)*e;               // Equatorial coordinates
cout << "l   " << Deg*e[phi]      << " b   " << Deg*e[theta] << endl
     << "RA  " << Deg*a[phi]/15.0 << " Dec " << Deg*a[theta] << endl;
```

The value of right ascension will be given – as usual – in hours ($1^h \equiv 15°$), instead of in degrees.

So far, nothing has been said about the exact value of the obliquity of the ecliptic ε. ε is not constant, but is decreasing by about 47″ per century. This primarily arises from slow alterations in the Earth's orbit as a result of perturbations caused by the other planets. If we wish to avoid errors in coordinate transformations, it is particularly important whether the ecliptic for 1950 or for 2000 is used. An expression of the form 'equinox 2000' is therefore used, to specify that coordinates refer to the equinox and the ecliptic (or the equator) of the year 2000, for example. The exact value for the obliquity of the ecliptic as a function of time is

$$\varepsilon = 23°43929111 - 46\overset{''}{.}8150\,T - 0\overset{''}{.}00059\,T^2 + 0\overset{''}{.}001813\,T^3 \quad , \tag{2.9}$$

where T is the number of Julian centuries between the epoch and 2000 January 1 (12^h). In contrast to a century in the Gregorian calendar, which averages 36 524.25 days, a Julian century is always exactly 36 525 days. Important Julian epochs are J1900 (1900 January 0.5, JD 2415020.0) and J2000 (2000 January 1.5, JD 2451545.0). There is exactly one Julian century between these dates. T may be calculated for a given epoch with the Julian Date JD from

$$T = (\mathrm{JD} - 2451545)/36525 \quad .$$

To a close approximation, the value of T may be obtained from the year y by $T \approx (y - 2000)/100$. For a given epoch T, the following function provides directly the matrix for the transformation of equatorial into ecliptic coordinates.

```
//-----------------------------------------------------------------------------
// Equ2EclMatrix: Transformation of equatorial to ecliptical coordinates
//   T          Time in Julian centuries since J2000
//   <return>   Transformation matrix
//-----------------------------------------------------------------------------
Mat3D Equ2EclMatrix (double T) {
  const double
    eps = ( 23.43929111-(46.8150+(0.00059-0.001813*T)*T)*T/3600.0 ) * Rad;
  return R_x(eps);
}
```

The opposite transformation is correspondingly described by the transformation matrix that follows.

```
//-----------------------------------------------------------------------------
// Ecl2EquMatrix: Transformation of ecliptical to equatorial coordinates
//   T          Time in Julian centuries since J2000
//   <return>   Transformation matrix
//-----------------------------------------------------------------------------
Mat3D Ecl2EquMatrix (double T)
{
  return Transp(Equ2EclMatrix(T));
}
```

Using `Ecl2EquMatrix`, we may write the above example as follows (let the equinox now be J1950 = JD2433282.50):

```
double MJD = 33282.0;                    // Epoch
double T   = (MJD-MJD_J2000)/36525.0;
Mat3D  U   = Ecl2EquMatrix(T);           // Transformation matrix
Vec3D  e   = Vec3D(1.0,2.0,3.0);         // Ecliptic coordinates
Vec3D  a   = U*e;                        // Equatorial coordinates
cout << "l   " << Deg*e[phi]      << " b   " << Deg*e[theta] << endl
     << "RA  " << Deg*a[phi]/15.0 << " Dec " << Deg*a[theta] << endl;
```

2.4 Precession

The shift in the position of the Earth's axis and of the ecliptic caused by forces exerted by the Sun, Moon and planets not only causes a slight change in the angle ε between the equator and the ecliptic, but also a shift of the vernal equinox by about $1\overset{\circ}{.}5$ per century ($1'$ per year). For precise calculations, therefore, the equinox of the coordinate system used must be stated. The equinoxes most frequently used are

- the equinox of date,
- equinox J2000 and
- equinox B1950.

'Equinox of date' means that the values used are those for the equator, ecliptic, and vernal equinox for the actual date under consideration. Such daily alteration of the coordinate system is sensible if one requires the coordinates of a planet, for example, for use in conjunction with the setting circles on an equatorially mounted telescope, or on a transit circle. Because of the shift in the Earth's axis, the orientation of the polar axis of a telescope will also alter. On the other hand, if one wants to study the actual spatial motion of a planet, then it is better to use a fixed equinox, such as that for Julian Epoch J2000 (2000 January 1.5 = JD 2451545.0), which was generally introduced in 1984. Before that, the older equinox B1950 had been used for a long time, and was employed for many stellar catalogues and atlases (such as the *SAO Star Catalog* and *Atlas Coeli*). The prefix B indicates that it is not half-way between the epochs J1900 and J2000 (which would, of course, be J1950 = JD 2433282.5), but the beginning of the Besselian year 1950 (1950 Jan. 0.923 = JD 2433282.423).

Transformation of coordinates from one equinox to another requires similar steps to those required in converting from ecliptic to equatorial coordinates. Whereas those two coordinate systems could be brought into coincidence by a simple rotation about the x-axis, we now have to deal with a total of three rotations: first about the z-axis, second about the x-axis, and finally about the z-axis again.

If we first consider the ecliptic at time T_0 and time $T_0 + T$, then the angle between the two planes is

$$\begin{aligned}\pi &= (47''.0029 - 0''.06603 \cdot T_0 + 0''.000598 \cdot T_0^2) \cdot T \\ &\quad +(-0''.03302 + 0''.000598 \cdot T_0) \cdot T^2 + 0''.000060 \cdot T^3\end{aligned} \tag{2.10}$$

(see Fig. 2.4). If two coordinate systems (x', y', z') and (x'', y'', z'') are established, such that the x'-y'-plane coincides with the ecliptic for time T_0, and the x''-y''-plane

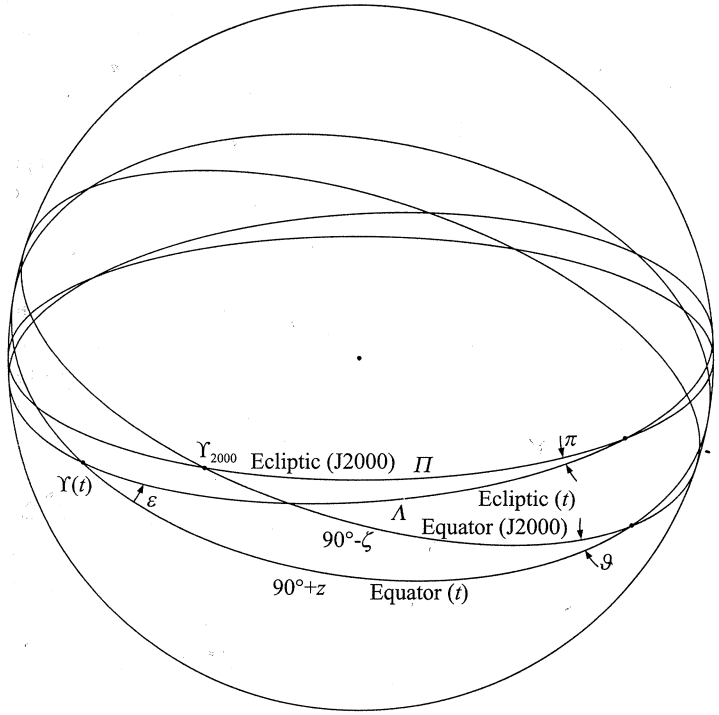

Fig. 2.4. The effects of precession on the ecliptic, equator, and vernal equinox

with that for time $T_0 + T$, then we have:

$$\begin{aligned} x'' &= x' \\ y'' &= +\cos\pi \cdot y' + \sin\pi \cdot z' \\ z'' &= -\sin\pi \cdot y' + \cos\pi \cdot z' \quad . \end{aligned}$$

This assumes that the x'-axis and the x''-axis coincide with the line of intersection of the two planes. Let (x_0, y_0, z_0) be the ecliptic coordinates corresponding to the epoch T_0 and (x, y, z) the coordinates at epoch $T_0 + T$. We then have

$$\begin{aligned} x' &= +\cos\Pi \cdot x_0 + \sin\Pi \cdot y_0 \\ y' &= -\sin\Pi \cdot x_0 + \cos\Pi \cdot y_0 \\ z' &= z_0 \quad , \end{aligned}$$

and

$$\begin{aligned} x &= +\cos\Lambda \cdot x'' - \sin\Lambda \cdot y'' \\ y &= +\sin\Lambda \cdot x'' + \cos\Lambda \cdot y'' \\ z &= z'' \quad , \end{aligned}$$

where

$$\begin{aligned}
\Pi &= (174^\circ.876383889 + 3289''.4789T_0 + 0''.60622T_0^2) \\
&\quad +(-869''.8089 - 0''.50491T_0)T + 0''.03536T^2 \\
p &= (5029''.0966 + 2''.22226T_0 - 0''.000042T_0^2)T \\
&\quad +(1''.11113 - 0''.000042T_0)T^2 - 0''.000006T^3 \\
\Lambda &= \Pi + p \quad .
\end{aligned} \tag{2.11}$$

Π and Λ are the angles between the x'/x''-axis and the vernal equinox Υ_0 at epoch T_0 (x_0-axis), and the vernal equinox Υ at epoch $T_0 + T$ (x-axis), respectively. p denotes precession in longitude, because the change in the vernal equinox mainly takes place in ecliptic longitude. Substituting for the various relationships, we obtain the equation

$$\boldsymbol{r} = \boldsymbol{P}\,\boldsymbol{r}_0 \quad \text{with} \quad \boldsymbol{P} = \boldsymbol{R}_z(-\Pi - p)\,\boldsymbol{R}_x(\pi)\,\boldsymbol{R}_z(\Pi) \quad . \tag{2.12}$$

Explicitly, the elements of $\boldsymbol{P}$ are given by

$$\begin{aligned}
p_{11} &= +\cos\Lambda\cos\Pi + \sin\Lambda\cos\pi\sin\Pi \\
p_{21} &= +\sin\Lambda\cos\Pi - \cos\Lambda\cos\pi\sin\Pi \\
p_{31} &= +\sin\pi\sin\Pi \\
p_{12} &= +\cos\Lambda\sin\Pi - \sin\Lambda\cos\pi\cos\Pi \\
p_{22} &= +\sin\Lambda\sin\Pi + \cos\Lambda\cos\pi\cos\Pi \\
p_{32} &= -\sin\pi\cos\Pi \\
p_{13} &= -\sin\Lambda\sin\pi \\
p_{23} &= +\cos\Lambda\sin\pi \\
p_{33} &= +\cos\pi \quad .
\end{aligned} \tag{2.13}$$

The angles in equatorial coordinates corresponding to the three angles π, Π and Λ are $90^\circ - \zeta$, ϑ and $90^\circ + z$:

$$\begin{aligned}
\zeta &= (2306''.2181 + 1''.39656T_0 - 0''.000139T_0^2)T \\
&\quad +(0''.30188 - 0''.000345T_0)T^2 + 0''.017998T^3 \\
\vartheta &= (2004''.3109 - 0''.85330T_0 - 0''.000217T_0^2)T \\
&\quad +(-0''.42665 - 0''.000217T_0)T^2 - 0''.041833T^3 \\
z &= \zeta + (0''.79280 + 0''.000411T_0)T^2 + 0''.000205T^3 \quad .
\end{aligned} \tag{2.14}$$

This gives corresponding formulae where

$$\boldsymbol{P} = \boldsymbol{R}_z(-z)\,\boldsymbol{R}_y(\vartheta)\,\boldsymbol{R}_z(-\zeta) \tag{2.15}$$

or, by component,

$$\begin{aligned}
p_{11} &= -\sin z\sin\zeta + \cos z\cos\vartheta\cos\zeta \\
p_{21} &= +\cos z\sin\zeta + \sin z\cos\vartheta\cos\zeta \\
p_{31} &= +\sin\vartheta\cos\zeta \\
p_{12} &= -\sin z\cos\zeta - \cos z\cos\vartheta\sin\zeta \\
p_{22} &= +\cos z\cos\zeta - \sin z\cos\vartheta\sin\zeta \\
p_{32} &= -\sin\vartheta\sin\zeta \\
p_{13} &= -\cos z\sin\vartheta \\
p_{23} &= -\sin z\sin\vartheta \\
p_{33} &= +\cos\vartheta \quad .
\end{aligned} \tag{2.16}$$

The most onerous part of calculating precession is determining the values for the various angles and for p_{ij}. However, the transformation matrices depend only on the two epochs T_0 and $T_0 + T$, and may be reused if a whole series of different positions are to be converted from one fixed epoch to another.

```
//----------------------------------------------------------------------------
// PrecMatrix_Ecl: Precession of ecliptic coordinates
//   T1          Epoch given
//   T2          Epoch to precess to
//   <return>    Precession transformation matrix
// Note: T1 and T2 in Julian centuries since J2000
//----------------------------------------------------------------------------
Mat3D PrecMatrix_Ecl (double T1, double T2)
{
  const double dT = T2-T1;
  double Pi, pi, p_a;

  Pi  = 174.876383889*Rad +
        ( ((3289.4789+0.60622*T1)*T1) +
              ((-869.8089-0.50491*T1) + 0.03536*dT)*dT )/Arcs;
  pi  = ( (47.0029-(0.06603-0.000598*T1)*T1)+
              ((-0.03302+0.000598*T1)+0.000060*dT)*dT )*dT/Arcs;
  p_a = ( (5029.0966+(2.22226-0.000042*T1)*T1)+
              ((1.11113-0.000042*T1)-0.000006*dT)*dT )*dT/Arcs;
  return R_z(-(Pi+p_a)) * R_x(pi) * R_z(Pi);
}

//----------------------------------------------------------------------------
// PrecMatrix_Equ: Precession of equatorial coordinates
//   T1          Epoch given
//   T2          Epoch to precess to
//   <return>    Precession transformation matrix
// Note: T1 and T2 in Julian centuries since J2000
//----------------------------------------------------------------------------
Mat3D PrecMatrix_Equ (double T1, double T2)
{
```

```
  const double dT = T2-T1;
  double zeta,z,theta;
  zeta  =  ( (2306.2181+(1.39656-0.000139*T1)*T1)+
             ((0.30188-0.000344*T1)+0.017998*dT)*dT )*dT/Arcs;
  z     =  zeta + ( (0.79280+0.000411*T1)+0.000205*dT)*dT*dT/Arcs;
  theta =  ( (2004.3109-(0.85330+0.000217*T1)*T1)-
             ((0.42665+0.000217*T1)+0.041833*dT)*dT )*dT/Arcs;
  return R_z(-z) * R_y(theta) * R_z(-zeta);
}
```

To give an idea of the effects of precession, and also to allow some practice in using the various routines, a somewhat longer example will now be given. A set of coordinates for equinox B1950 is to be converted to a corresponding set for equinox J2000. This may be carried out in two ways, according to whether precession or the conversion between ecliptic and equatorial coordinates is applied first.

```
// Declarations
double T_B1950 = -0.500002108;          // Epoch B1950 (JD2433282.423)
double l   = 200.0*Rad;                 // Longitude
double b   =  10.0*Rad;                 // Latitude
Vec3D  r_0 = Vec3D(Polar(l,b));         // Ecliptic coordinates B1950
Vec3D  r;

// Method 1
r = PrecMatrixEcl(T_B1950,T_J2000) * r_0;  // Ecliptic J2000
cout << "l,b (J2000)     " << Deg*r[phi]      << Deg*r[theta] << endl;
r = Ecl2EquMatrix(T_J2000) * r;            // Equator J2000
cout << "RA,Dec (J2000)  " << Deg*r[phi]/15.0 << Deg*r[theta] << endl;

// Method 2
r = Ecl2EquMatrix(T_B1950) * r_0;          // Equator B1950
cout << "RA,Dec (B1950)  " << Deg*r[phi]/15.0 << Deg*r[theta] << endl;
r = PrecMatrixEqu(T_B1950,T_J2000) * r;    // Equator J2000
cout << "RA,Dec (J2000)  " << Deg*r[phi]/15.0 << Deg*r[theta] << endl;
```

2.5 Geocentric Coordinates and the Orbit of the Sun

All that is now required to complete the program is a method for converting heliocentric coordinates (referred to the centre of the Sun) into geocentric ones (referred to the centre of the Earth), and the converse. Conversion of the origin of the coordinates is described by the equations

$$\boldsymbol{r}_{\mathrm{p}} = \boldsymbol{r}_{\odot\mathrm{p}} + \boldsymbol{r}_{\odot} \quad \text{and} \quad \boldsymbol{r}_{\odot\mathrm{p}} = \boldsymbol{r}_{\mathrm{p}} - \boldsymbol{r}_{\odot} \tag{2.17}$$

which are derived from the Sun-Earth-Planet triangle illustrated in Fig. 2.5. Here $\boldsymbol{r}_{\odot\mathrm{p}}$ and $\boldsymbol{r}_{\mathrm{p}}$ are the heliocentric and geocentric position vectors of a point P, and $\boldsymbol{r}_{\odot}$ is the geocentric position of the Sun. Written in the form of individual components, the transformation equations are

$$\begin{aligned} x_{\mathrm{p}} &= x_{\odot\mathrm{p}} + x_{\odot} \\ y_{\mathrm{p}} &= y_{\odot\mathrm{p}} + y_{\odot} \\ z_{\mathrm{p}} &= z_{\odot\mathrm{p}} + z_{\odot} \end{aligned} \quad \text{and} \quad \begin{aligned} x_{\odot\mathrm{p}} &= x_{\mathrm{p}} - x_{\odot} \\ y_{\odot\mathrm{p}} &= y_{\mathrm{p}} - y_{\odot} \\ z_{\odot\mathrm{p}} &= z_{\mathrm{p}} - z_{\odot} \end{aligned} ,$$

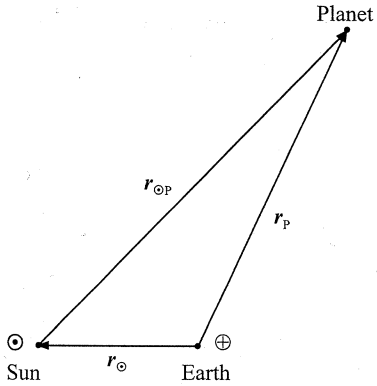

Fig. 2.5. The Earth-Sun-Planet triangle

where it makes no difference whether ecliptic or equatorial coordinates are employed to express the various vectors.

As any error in the position of the Sun inevitably affects the accuracy of the conversions, it is worthwhile taking some trouble in calculating the coordinates of the Sun. The SunPos function, introduced here, attains an accuracy of about 1″, which should be more than sufficient for most purposes. For a time

$$T = (\mathrm{JD} - 2451545)/36525$$

it provides the Cartesian ecliptic coordinates

$$\boldsymbol{r}_{\odot} = \begin{pmatrix} x_{\odot} \\ y_{\odot} \\ z_{\odot} \end{pmatrix} = \begin{pmatrix} R\cos B\cos L \\ R\cos B\sin L \\ R\sin B \end{pmatrix} . \tag{2.18}$$

Like the associated ecliptic longitude L and latitude B these coordinates initially refer to the equinox of date, but may be converted to another equinox by means of the functions already discussed.

```
//-----------------------------------------------------------------------
//
// SunPos: Computes the Sun's ecliptical position using analytical series
//   T         Time in Julian centuries since J2000
//   <return>  Geocentric position of the Sun (in [AU]), referred to the
//             ecliptic and equinox of date
//
//-----------------------------------------------------------------------
Vec3D SunPos (double T) {

  // Variables
  double  M2,M3,M4,M5,M6;          // Mean anomalies
  double  D, A, U;                 // Mean arguments of lunar orbit
  Pert    Ven, Mar, Jup, Sat;      // Perturbations
  double  dl, dr, db;              // Corrections in longitude ["],
                                   // radius [AU] and latitude ["]
  double  l,b,r;                   // Ecliptic coordinates
```

```
// Mean anomalies of planets and mean arguments of lunar orbit [rad]
M2 = pi2 * Frac ( 0.1387306 + 162.5485917*T );
M3 = pi2 * Frac ( 0.9931266 +  99.9973604*T );
M4 = pi2 * Frac ( 0.0543250 +  53.1666028*T );
M5 = pi2 * Frac ( 0.0551750 +   8.4293972*T );
M6 = pi2 * Frac ( 0.8816500 +   3.3938722*T );
D  = pi2 * Frac ( 0.8274 + 1236.8531*T );
A  = pi2 * Frac ( 0.3749 + 1325.5524*T );
U  = pi2 * Frac ( 0.2591 + 1342.2278*T );

// Keplerian terms and perturbations by Venus
Ven.Init ( T, M3,0,7, M2,-6,0 );
Ven.Term ( 1, 0,0,-0.22,6892.76,-16707.37, -0.54, 0.00, 0.00);
Ven.Term ( 1, 0,1,-0.06, -17.35,    42.04, -0.15, 0.00, 0.00);
Ven.Term ( 1, 0,2,-0.01,  -0.05,     0.13, -0.02, 0.00, 0.00);
Ven.Term ( 2, 0,0, 0.00,  71.98,  -139.57,  0.00, 0.00, 0.00);
Ven.Term ( 2, 0,1, 0.00,  -0.36,     0.70,  0.00, 0.00, 0.00);
Ven.Term ( 3, 0,0, 0.00,   1.04,    -1.75,  0.00, 0.00, 0.00);
Ven.Term ( 0,-1,0, 0.03,  -0.07,    -0.16, -0.07, 0.02,-0.02);
Ven.Term ( 1,-1,0, 2.35,  -4.23,    -4.75, -2.64, 0.00, 0.00);
Ven.Term ( 1,-2,0,-0.10,   0.06,     0.12,  0.20, 0.02, 0.00);
Ven.Term ( 2,-1,0,-0.06,  -0.03,     0.20, -0.01, 0.01,-0.09);
Ven.Term ( 2,-2,0,-4.70,   2.90,     8.28, 13.42, 0.01,-0.01);
Ven.Term ( 3,-2,0, 1.80,  -1.74,    -1.44, -1.57, 0.04,-0.06);
Ven.Term ( 3,-3,0,-0.67,   0.03,     0.11,  2.43, 0.01, 0.00);
Ven.Term ( 4,-2,0, 0.03,  -0.03,     0.10,  0.09, 0.01,-0.01);
Ven.Term ( 4,-3,0, 1.51,  -0.40,    -0.88, -3.36, 0.18,-0.10);
Ven.Term ( 4,-4,0,-0.19,  -0.09,    -0.38,  0.77, 0.00, 0.00);
Ven.Term ( 5,-3,0, 0.76,  -0.68,     0.30,  0.37, 0.01, 0.00);
Ven.Term ( 5,-4,0,-0.14,  -0.04,    -0.11,  0.43,-0.03, 0.00);
Ven.Term ( 5,-5,0,-0.05,  -0.07,    -0.31,  0.21, 0.00, 0.00);
Ven.Term ( 6,-4,0, 0.15,  -0.04,    -0.06, -0.21, 0.01, 0.00);
Ven.Term ( 6,-5,0,-0.03,  -0.03,    -0.09,  0.09,-0.01, 0.00);
Ven.Term ( 6,-6,0, 0.00,  -0.04,    -0.18,  0.02, 0.00, 0.00);
Ven.Term ( 7,-5,0,-0.12,  -0.03,    -0.08,  0.31,-0.02,-0.01);
dl = Ven.dl(); dr = Ven.dr();  db = Ven.db();

// Perturbations by Mars
Mar.Init ( T, M3,1,5, M4,-8,-1 );
Mar.Term ( 1,-1,0,-0.22,   0.17,    -0.21, -0.27, 0.00, 0.00);
Mar.Term ( 1,-2,0,-1.66,   0.62,     0.16,  0.28, 0.00, 0.00);
Mar.Term ( 2,-2,0, 1.96,   0.57,    -1.32,  4.55, 0.00, 0.01);
Mar.Term ( 2,-3,0, 0.40,   0.15,    -0.17,  0.46, 0.00, 0.00);
Mar.Term ( 2,-4,0, 0.53,   0.26,     0.09, -0.22, 0.00, 0.00);
Mar.Term ( 3,-3,0, 0.05,   0.12,    -0.35,  0.15, 0.00, 0.00);
Mar.Term ( 3,-4,0,-0.13,  -0.48,     1.06, -0.29, 0.01, 0.00);
Mar.Term ( 3,-5,0,-0.04,  -0.20,     0.20, -0.04, 0.00, 0.00);
Mar.Term ( 4,-4,0, 0.00,  -0.03,     0.10,  0.04, 0.00, 0.00);
Mar.Term ( 4,-5,0, 0.05,  -0.07,     0.20,  0.14, 0.00, 0.00);
Mar.Term ( 4,-6,0,-0.10,   0.11,    -0.23, -0.22, 0.00, 0.00);
Mar.Term ( 5,-7,0,-0.05,   0.00,     0.01, -0.14, 0.00, 0.00);
Mar.Term ( 5,-8,0, 0.05,   0.01,    -0.02,  0.10, 0.00, 0.00);
dl += Mar.dl(); dr += Mar.dr();  db += Mar.db();

// Perturbations by Jupiter
Jup.Init ( T, M3,-1,3, M5,-4,-1 );
Jup.Term (-1,-1,0, 0.01,   0.07,     0.18, -0.02, 0.00,-0.02);
Jup.Term ( 0,-1,0,-0.31,   2.58,     0.52,  0.34, 0.02, 0.00);
Jup.Term ( 1,-1,0,-7.21,  -0.06,     0.13,-16.27, 0.00,-0.02);
Jup.Term ( 1,-2,0,-0.54,  -1.52,     3.09, -1.12, 0.01,-0.17);
Jup.Term ( 1,-3,0,-0.03,  -0.21,     0.38, -0.06, 0.00,-0.02);
Jup.Term ( 2,-1,0,-0.16,   0.05,    -0.18, -0.31, 0.01, 0.00);
Jup.Term ( 2,-2,0, 0.14,  -2.73,     9.23,  0.48, 0.00, 0.00);
Jup.Term ( 2,-3,0, 0.07,  -0.55,     1.83,  0.25, 0.01, 0.00);
Jup.Term ( 2,-4,0, 0.02,  -0.08,     0.25,  0.06, 0.00, 0.00);
Jup.Term ( 3,-2,0, 0.01,  -0.07,     0.16,  0.04, 0.00, 0.00);
Jup.Term ( 3,-3,0,-0.16,  -0.03,     0.08, -0.64, 0.00, 0.00);
Jup.Term ( 3,-4,0,-0.04,  -0.01,     0.03, -0.17, 0.00, 0.00);
dl += Jup.dl(); dr += Jup.dr();  db += Jup.db();

// Perturbations by Saturn
Sat.Init ( T, M3,0,2, M6,-2,-1 );
Sat.Term ( 0,-1,0, 0.00,   0.32,     0.01,  0.00, 0.00, 0.00);
Sat.Term ( 1,-1,0,-0.08,  -0.41,     0.97, -0.18, 0.00,-0.01);
Sat.Term ( 1,-2,0, 0.04,   0.10,    -0.23,  0.10, 0.00, 0.00);
Sat.Term ( 2,-2,0, 0.04,   0.10,    -0.35,  0.13, 0.00, 0.00);
dl += Sat.dl(); dr += Sat.dr();  db += Sat.db();

// Difference of Earth-Moon-barycentre and centre of the Earth
dl += +  6.45*sin(D) - 0.42*sin(D-A) + 0.18*sin(D+A)
      +  0.17*sin(D-M3) - 0.06*sin(D+M3);
dr += + 30.76*cos(D) - 3.06*cos(D-A) + 0.85*cos(D+A)
      -  0.58*cos(D+M3) + 0.57*cos(D-M3);
db += + 0.576*sin(U);

// Long-periodic perturbations
dl += + 6.40 * sin ( pi2*(0.6983 + 0.0561*T) )
      + 1.87 * sin ( pi2*(0.5764 + 0.4174*T) )
      + 0.27 * sin ( pi2*(0.4189 + 0.3306*T) )
      + 0.20 * sin ( pi2*(0.3581 + 2.4814*T) );

// Ecliptic coordinates ([rad],[AU])
l = pi2 * Frac ( 0.7859453 + M3/pi2 +
                 ( (6191.2+1.1*T)*T + dl ) / 1296.0e3 );
r = 1.0001398 - 0.0000007 * T + dr * 1.0e-6;
b = db / Arcs;

return Vec3D ( Polar(l,b,r) );  // Position vector
}
```

The extent of the SunPos function is explained by the fact that the relative motion of the Sun and the Earth cannot be described to the required accuracy by simple elliptical orbits. Apart from the perturbations of the other planets – particularly those of Venus and Jupiter – there is also the monthly oscillation of the centre of the Earth around the barycentre of the Earth-Moon system. Strictly speaking, not the Earth itself, but rather the position of the barycentre, is moving in the plane

of the ecliptic. The Moon's orbit around the Earth is therefore reflected in a small periodic perturbation of the geocentric orbit of the Sun. The details of the function and its basis will not be discussed here. Both will be treated thoroughly in Chap. 6, where appropriate routines for all the planets will be given.

2.6 The COCO Program

The functions described so far will now be combined into a complete program. In the following, all the relevant routines as well as the main program are given. The functions that are already familiar are combined into the corresponding library modules, whose matching header files are incorporated at the beginning of the program.

The core functions of the program are incorporated into a class `Position`, which, as well as the input and output of positions, has three methods for selecting the origin (heliocentric or geocentric), the equinox, and the reference frame (ecliptic or equatorial). The main program is therefore restricted to the declaration of an object of this class and the user-selected call to its methods. Although the class `Position` has no immediate application outside the `Coco` program, the object-oriented approach here, fosters a compact and clean programming.

```
//--------------------------------------------------------------------------
// File:    Coco.cpp
// Purpose: Coordinate transformations
// (c) 1999 Oliver Montenbruck, Thomas Pfleger
//--------------------------------------------------------------------------

#include <iomanip>
#include <iostream>

#include "APC_Const.h"
#include "APC_Math.h"
#include "APC_PrecNut.h"
#include "APC_Spheric.h"
#include "APC_Sun.h"
#include "APC_Time.h"
#include "APC_VecMat3D.h"

using namespace std;

// Definition of class "Position"
enum enOrigin   { Heliocentric, Geocentric };  // Origin of coordinates
enum enRefSys   { Ecliptic, Equator };         // Reference system
class Position {
  public:
    void Input(); // Query user for parameters
    void SetOrigin(enOrigin Origin);
    void SetRefSys(enRefSys RefSys);
    void SetEquinox(double T_Equinox);
    void Print();
  private:
    Vec3D      m_R;          // Coordinate vector
    enOrigin   m_Origin;     // Origin of coordinate system
    enRefSys   m_RefSys;     // Reference system
    double     m_TEquinox;   // Equinox (centuries since J2000)
    double     m_MjdEpoch;   // Epoch (Modif. Julian Date)
};

// Data input and output
void Position::Input() { ... }
void Position::Print() { ... }

// Change of origin
void Position::SetOrigin(enOrigin Origin)
{
  double T_Epoch;
  Vec3D  R_Sun;
  if (Origin!=m_Origin) {
    // Geocentric coordinates of the Sun at epoch w.r.t.
    // given reference system and equinox
    T_Epoch = (m_MjdEpoch-MJD_J2000)/36525.0;
    if (m_RefSys==Ecliptic)
      R_Sun = PrecMatrix_Ecl(T_Epoch,m_TEquinox) * SunPos(T_Epoch);
    else
      R_Sun = Ecl2EquMatrix(m_TEquinox) *
              PrecMatrix_Ecl(T_Epoch,m_TEquinox) * SunPos(T_Epoch);
    // Change origin
    if (m_Origin==Heliocentric) {
      m_R += R_Sun; m_Origin = Geocentric;
    }
    else {
      m_R -= R_Sun; m_Origin = Heliocentric;
    };
  };
}
// Change of reference system
void Position::SetRefSys(enRefSys RefSys)
{
  if (RefSys!=m_RefSys) {
    if (m_RefSys==Equator) {
      m_R = Equ2EclMatrix(m_TEquinox) * m_R;  m_RefSys = Ecliptic;
    }
    else {
      m_R = Ecl2EquMatrix(m_TEquinox) * m_R;  m_RefSys = Equator;
    }
  };
}
// Change of equinox
void Position::SetEquinox(double T_Equinox)
{
  if (T_Equinox!=m_TEquinox) {
    if (m_RefSys==Equator)
      m_R = PrecMatrix_Equ(m_TEquinox,T_Equinox) * m_R;
```

```
    else
      m_R = PrecMatrix_Ecl(m_TEquinox,T_Equinox) * m_R;
    m_TEquinox = T_Equinox;
  };
}

//--------------------------------------------------------------------------
//
// Main program
//
//--------------------------------------------------------------------------
void main() {
  // Variables
  Position  Pos;
  char      c;
  bool      End = false;
  double    Year;

  // Header
  cout << endl
       << "                COCO: coordinate conversions       " << endl
       << "        (c) 1999 Oliver Montenbruck, Thomas Pfleger" << endl
       << endl;

  // Initialization
  Pos.Input();
  Pos.Print();
  // Command loop

  do {
    // Command input
    cout << "Enter command (?=Help) ... ";
    cin >> c; cin.ignore(81,'\n'); c=tolower(c);
    // Actions
    switch (c) {
      case 'x':
        End = true;  break;
      case 'a':
        Pos.SetRefSys(Equator); Pos.Print();  break;
      case 'e':
        Pos.SetRefSys(Ecliptic); Pos.Print();  break;
      case 'g':
        Pos.SetOrigin(Geocentric); Pos.Print();  break;
      case 'h':
        Pos.SetOrigin(Heliocentric); Pos.Print();  break;
      case 'n':
        Pos.Input(); Pos.Print();  break;
      case 'p':
        cout << "New equinox (yyyy.y)   ...  ";
        cin >> Year; cin.ignore(81,'\n');
        Pos.SetEquinox( (Year-2000.0)/100.0 );
        Pos.Print();
        break;
```



```
      default:
        // Display help text
        cout << endl
             << "Available commands" << endl
             << "  e=ecliptic,   a=equatorial,   p=precession, " << endl
             << "  g=geocentric, h=heliocentric, n=new input,  " << endl
             << "  x=exit                                      " << endl
             << endl;
        break;
    }
  }
  while (!End);
  cout << endl;
}
```

The following example should illustrate the use and possibilities offered by Coco. The coordinates to be converted may be entered in either the equatorial or the ecliptic form. One can also choose between Cartesian coordinates or polar coordinates. This choice is offered by Coco at the beginning. All entries are indicated in italic. We select equatorial coordinates in polar representation and enter the coordinates of the vernal equinox of 1950.0. The distance is arbitrarily chosen as 1 AU. It must always be positive.

```
                COCO: coordinate conversions
        (c) 1999 Oliver Montenbruck, Thomas Pfleger

New input:

  Reference system (e=ecliptic,a=equator) ... a
  Format (c=cartesian,p=polar)            ... p
  Coordinates (RA [h m s] Dec [o ' "] R)  ... 0 0 0.0 0 0 0.0 1.0
  Equinox (yyyy.y)                        ... 1950.0
  Origin (h=heliocentric,g=geocentric)    ... g
  Epoch (yyyy mm dd hh.h)                 ... 1989 1 1 0.0
```

Coco acknowledges the input by displaying these data in Cartesian and polar coordinates:

```
Geocentric equatorial coordinates
(Equinox J1950.0,  Epoch 1989/01/01 00.0)

  (x,y,z) =  (  1.00000000,  0.00000000,  0.00000000)

        h  m  s              o  '  "
  RA =  0 00 00.00    Dec = + 0 00 00.0    R =   1.00000000

Enter command (?=Help) ... ?

Available commands
  e=ecliptic,   a=equatorial,   p=precession,
  g=geocentric, h=heliocentric, n=new input,
  x=exit
```

```
Enter command (?=Help) ...
```

Subsequently, the program expects the input of a command for converting the coordinates. Possible inputs are:

- **a** Conversion to equatorial coordinates,
- **e** Conversion to ecliptic coordinates,
- **p** Precession (Choice of equinox),
- **g** Conversion to geocentric coordinates,
- **h** Conversion to heliocentric coordinates,
- **n** New input,
- **?** Help,
- **x** Exit (end of program) .

We first want to convert the given position to epoch 2000, and therefore choose option p to calculate the precession. After entering the year, we obtain the position of the vernal equinox for 1950 referred to the new epoch 2000. The precession amounts to slightly more than half a degree.

```
Enter command (?=Help) ... p
New equinox (yyyy.y)   ... 2000.0

Geocentric equatorial coordinates
(Equinox J2000.0,  Epoch 1989/01/01 00.0)

  (x,y,z) =  (  0.99992571,  0.01117889,  0.00485898)

        h  m  s                o  '  "
  RA =  0 02 33.73    Dec = + 0 16 42.2    R =   1.00000000
```

We now need the given equatorial coordinates to be converted into the ecliptic system. We therefore choose option e and obtain the converted coordinates – again in Cartesian and polar forms. Instead of right ascension RA and declination Dec we now have ecliptic longitude L and latitude B. As in the previous step, the distance remains unchanged.

```
Enter command (?=Help) ... e

Geocentric ecliptic coordinates
(Equinox J2000.0,  Epoch 1989/01/01 00.0)

  (x,y,z) =  (  0.99992571,  0.01218922,  0.00001132)

        o  '  "              o  '  "
  L =   0 41 54.27    B = + 0 00 02.3    R =   1.00000000
```

It is now possible to convert the given coordinates from geocentric to heliocentric coordinates for the initially specified epoch using option h. As in the previous step we chose ecliptic coordinates, we now obtain heliocentric ecliptic coordinates.


```
Enter command (?=Help) ... h

Heliocentric ecliptic coordinates
(Equinox J2000.0,  Epoch 1989/01/01 00.0)

  (x,y,z) =  (  0.81725247,  0.97838164,  0.00003597)

        o  '  "              o  '  "
  L =  50 07 39.50    B = + 0 00 05.8    R =   1.27480674
```

The conversion into ecliptic coordinates may be reversed by selecting option a, which converts the current (ecliptic) coordinates into equatorial coordinates.

```
Enter command (?=Help) ... a

Heliocentric equatorial coordinates
(Equinox J2000.0,  Epoch 1989/01/01 00.0)

  (x,y,z) =  (  0.81725247,  0.89763329,  0.38921086)

        h  m  s                o  '  "
  RA =  3 10 44.07    Dec = +17 46 36.5    R =   1.27480674
```

Let us now recalculate the coordinates for the original equinox of 1950.

```
Enter command (?=Help) ... p
New equinox (yyyy.y)   ... 1950.0

Heliocentric equatorial coordinates
(Equinox J1950.0,  Epoch 1989/01/01 00.0)

  (x,y,z) =  (  0.82911747,  0.88843066,  0.38521087)

        h  m  s                o  '  "
  RA =  3 07 54.68    Dec = +17 35 17.2    R =   1.27480674
```

Conversion of these coordinates into geocentric coordinates should now yield the original coordinates of the vernal equinox, because we have applied the inverse transformation to the three conversions that we invoked. Unavoidable rounding errors within the computer may, however, prevent exactly the same coordinates as those entered at the beginning of this example from being obtained.

```
Enter command (?=Help) ... g

Geocentric equatorial coordinates
(Equinox J1950.0,  Epoch 1989/01/01 00.0)

  (x,y,z) =  (  1.00000000, -0.00000000,  0.00000000)

        h  m  s                o  '  "
  RA = 24 00 00.00    Dec = + 0 00 00.0    R =   1.00000000
```

Entering x then terminates the program.

```
Enter command (?=Help) ... x
```

The values shown here may differ slightly depending on the particular C++ compiler being used.

3. Calculation of Rising and Setting Times

3.1 The Observer's Horizon System

The ecliptic and equatorial coordinates discussed so far are specified in terms of the mean plane of the Earth's orbit and of the position of the Earth's axis. Neither of these systems is particularly suitable, however, for an observer situated on the surface of the Earth. As such an observer (without being aware of it) takes part in the Earth's daily rotation, it appears as if the Sun, Moon, and stars follow large arcs across the sky from East to West during the course of a day, reaching their highest point above the horizon when they are on the meridian.

The stars' apparent paths depend on the geographical latitude of the point of observation. In the Southern Hemisphere most stars do not reach their greatest altitude in the South, but in the North. As a result, orienting themselves on the sky is more difficult for observers who are used to the appearance of the sky at moderate northern latitudes. All the well-known constellations appear to be standing on their heads.

Two points on the celestial sphere are particularly significant: the zenith (the point directly above the observer), and the North Celestial Pole (Fig. 3.1). The latter is taken to be the point on the celestial sphere towards which the Earth's axis is pointing, and about which, in consequence, all the stars appear to be moving in concentric circles. The altitude of this point above the horizon corresponds to the observer's geographic latitude φ. A great circle drawn through the North Celestial Pole and the zenith, known as the *meridian*, intersects the horizon at the exact North and South points.

The coordinates used in the horizontal system are azimuth (A) and altitude (h), which may, for instance, be determined by the use of a theodolite (see Fig. 3.1). The altitude is simply the angular height above the horizon. The azimuth, determined by rotating the theodolite around its vertical axis, is defined as the angle between the South point and the required position. On this basis, the azimuth of the West point is $A = 90°$ and, correspondingly, for the East point $A = 270°$ (or $A = -90°$). Unfortunately, alongside the definition just given, there is a second one that is used, in particular, in navigation. Here, azimuth is taken to originate at the North point. According to this method, the East point therefore has an azimuth of 90°. In case of any doubt, it is essential to check which of these two definitions of azimuth is being used!

Because of the rotation of the Earth, the altitude and azimuth of an object with a given right ascension and declination alter continuously. If a telescope is held at

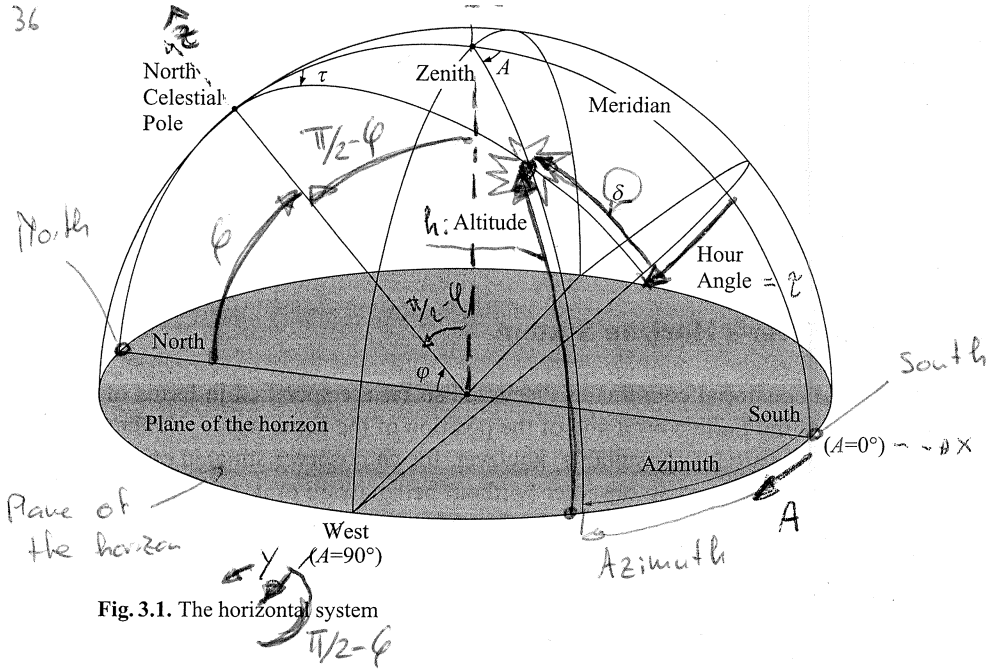

Fig. 3.1. The horizontal system

a fixed position in the horizontal system, with time, the stars that cross the field of view are those that have the same declination, and which differ only in their right ascension. The determination of the declination of a star from altitude and azimuth is therefore independent of time. For right ascension, on the other hand, only the difference between the object and stars on the meridian may be determined.

To obtain reciprocal conversions between horizontal and equatorial coordinates, the concept of *hour angle* τ is used. This is the difference between the right ascension of the star being observed and the right ascension of stars on the meridian. Hour angle τ, like right ascension α, is generally measured in units of time ($1^h \equiv 15°$), and therefore corresponds approximately[1] to the amount of time that has passed since the star crossed the meridian. In the system employed here, A and τ have the same mathematical sign. If we have the Cartesian coordinates

$$\boldsymbol{r} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \cos h \cos A \\ \cos h \sin A \\ \sin h \end{pmatrix} \qquad \hat{\boldsymbol{r}} = \begin{pmatrix} \hat{x} \\ \hat{y} \\ \hat{z} \end{pmatrix} = \begin{pmatrix} \cos\delta \cos\tau \\ \cos\delta \sin\tau \\ \sin\delta \end{pmatrix} \tag{3.1}$$

the transformations

$$\begin{aligned} x &= +\hat{x}\sin\varphi - \hat{z}\cos\varphi & \hat{x} &= +x\sin\varphi + z\cos\varphi \\ y &= +\hat{y} & \hat{y} &= +y \\ z &= +\hat{x}\cos\varphi + \hat{z}\sin\varphi & \hat{z} &= -x\cos\varphi + z\sin\varphi \end{aligned} \tag{3.2}$$

apply, which may in brief be written as $\boldsymbol{r} = \boldsymbol{R}_y(\pi/2 - \varphi)\hat{\boldsymbol{r}}$. Here φ is the geographical latitude of the observer, and $\hat{}$ indicates equatorial coordinates. Substitution

[1] One rotation of the Earth takes only 23^h56^m.

then yields a system of three equations

$$\begin{aligned} \cos h \cos A &= +\cos\delta\cos\tau\sin\varphi - \sin\delta\cos\varphi \\ \cos h \sin A &= +\cos\delta\sin\tau \\ \sin h &= +\cos\delta\cos\tau\cos\varphi + \sin\delta\sin\varphi \end{aligned} \tag{3.3}$$

for computing azimuth and altitude from hour angle and declination. Conversely, the hour angle and declination may be determined from

$$\begin{aligned} \cos\delta\cos\tau &= +\cos h\cos A\sin\varphi + \sin h\cos\varphi \\ \cos\delta\sin\tau &= +\cos h\sin A \\ \sin\delta &= -\cos h\cos A\cos\varphi + \sin h\sin\varphi \end{aligned} \tag{3.4}$$

for a given azimuth and altitude. The transformation between the equatorial system and the horizontal system may be accomplished by the functions `Equ2Hor` and `Hor2Equ`, which are given here for the sake of completeness.

```
//------------------------------------------------------------------------
// Equ2Hor: Transformation of equatorial coordinates to the horizon system
//   Dec       Declination [rad]
//   tau       Hour angle [rad]
//   lat       Geographical latitude of the observer [rad]
//   h,Az      Altitude and azimuth [rad]
//------------------------------------------------------------------------
void Equ2Hor ( double Dec, double tau, double lat,
               double& h, double& Az )
{
  Vec3D e_equ, e_hor;
  e_equ = Vec3D(Polar(tau,Dec));       // unit vector in horizontal system
  e_hor = R_y(pi/2.0-lat) * e_equ;     // unit vector in equatorial system
  Az = e_hor[phi];                     // polar angles
  h  = e_hor[theta];
}
//------------------------------------------------------------------------
// Hor2Equ: Transformation of horizon system coordinates
//          to equatorial coordinates
//   h,Az      Altitude and azimuth [rad]
//   lat       Geographical latitude of the observer [rad]
//   Dec       Declination [rad]
//   tau       Hour angle [rad]
//------------------------------------------------------------------------
void Hor2Equ ( double h, double Az, double lat,
               double& Dec, double& tau )
{
  Vec3D e_equ, e_hor;
  e_hor = Vec3D(Polar(Az,h));          // unit vector in horizontal system
  e_equ = R_y(-(pi/2.0-lat)) * e_hor;  // unit vector in equatorial system
  tau = e_equ[phi];                    // polar angles
  Dec = e_equ[theta];
}
```

For determining the times of rising and setting, which is the aim of this chapter, one equation only is required, namely

$$\sin h = \cos\varphi\cos\delta\cos\tau + \sin\varphi\sin\delta \quad . \tag{3.5}$$

It allows the altitude h to be calculated from given values of geographical latitude φ, declination δ, and hour angle τ.

Before we can carry out this conversion, however, some preparation is required. First, we lack the means of determining the coordinates (α, δ) of the Sun and the Moon. Then we need to clarify how we can calculate the hour angle from the known right ascension at a particular time. Finally, we have to take into account a whole series of corrections that affect the altitude of the observed horizon. Only after these steps will we return to the equation just mentioned.

3.2 Sun and Moon

Calculation of rising and setting does not make demands for a high degree of accuracy in the coordinates of the Sun and the Moon. The `MiniSun` and `MiniMoon` functions therefore contain only the most important terms describing the respective orbits. They are greatly reduced versions of `SunPos` and `MoonPos`, which will be described in detail in Chap. 6 and Chap. 8. Conversion of ecliptic longitude and latitude into equatorial coordinates is also included, therefore both functions may be used without further operations to obtain the right ascension and declination of the object in question.

```
//-----------------------------------------------------------------------
// MiniMoon: Computes the Moon's RA and declination using a low precision
//           analytical series
//   T         Time in Julian centuries since J2000
//   RA        Right Ascension of the Moon in [rad]
//   Dec       Declination of the Moon in [rad]
//-----------------------------------------------------------------------
void MiniMoon (double T, double& RA, double& Dec)
{
  const double eps = 23.43929111*Rad;
  double L_0, l,ls, F, D, dL, S, h, N, l_Moon, b_Moon;
  Vec3D  e_Moon;
  // Mean elements of lunar orbit
  L_0 = Frac (0.606433 + 1336.855225*T);       // mean longitude [rev]
  l   = pi2*Frac ( 0.374897 + 1325.552410*T );  // Moon's mean anomaly
  ls  = pi2*Frac ( 0.993133 +   99.997361*T );  // Sun's mean anomaly
  D   = pi2*Frac ( 0.827361 + 1236.853086*T );  // Diff. long. Moon-Sun
  F   = pi2*Frac ( 0.259086 + 1342.227825*T );  // Dist. from ascending node
  // Perturbations in longitude and latitude
  dL = +22640*sin(l) - 4586*sin(l-2*D) + 2370*sin(2*D) +  769*sin(2*l)
       -668*sin(ls) - 412*sin(2*F) - 212*sin(2*l-2*D) - 206*sin(l+ls-2*D)
       +192*sin(l+2*D) - 165*sin(ls-2*D) - 125*sin(D) - 110*sin(l+ls)
       +148*sin(l-ls) - 55*sin(2*F-2*D);
  S  = F + (dL+412*sin(2*F)+541*sin(ls)) / Arcs;
  h  = F-2*D;
  N  = -526*sin(h) + 44*sin(l+h) - 31*sin(-l+h) - 23*sin(ls+h)
       + 11*sin(-ls+h) - 25*sin(-2*l+F) + 21*sin(-l+F);
  // Ecliptic longitude and latitude
  l_Moon = pi2 * Frac( L_0 + dL/1296.0e3 ); // [rad]
  b_Moon = ( 18520.0*sin(S) + N ) / Arcs;   // [rad]
  // Equatorial coordinates
  e_Moon = R_x(-eps) * Vec3D(Polar(l_Moon,b_Moon));
  RA  = e_Moon[phi];
  Dec = e_Moon[theta];
}

//-----------------------------------------------------------------------
// MiniSun: Computes the Sun's RA and declination using a low precision
//          analytical series
//   T         Time in Julian centuries since J2000
//   RA        Right Ascension of the Sun in [rad]
//   Dec       Declination of the Sun in [rad]
//-----------------------------------------------------------------------
void MiniSun (double T, double& RA, double& Dec)
{
  const double eps = 23.43929111*Rad;
  double L,M;
  Vec3D  e_Sun;
  // Mean anomaly and ecliptic longitude
  M  = pi2 * Frac ( 0.993133 + 99.997361*T);
  L  = pi2 * Frac ( 0.7859453 + M/pi2 +
                    (6893.0*sin(M)+72.0*sin(2.0*M)+6191.2*T) / 1296.0e3);
  // Equatorial coordinates
  e_Sun = R_x(-eps) * Vec3D(Polar(L,0.0));
  RA  = e_Sun[phi];
  Dec = e_Sun[theta];
}
```


3.3 Sidereal Time and Hour Angle

The right ascensions of the Sun and Moon are not sufficient to calculate their altitudes above the horizon. We also need to know the right ascension of stars on the meridian and, from that, determine the hour angle. Precisely which stars these are depends on the observer's location and the exact time. As the Earth rotates around its axis once a day, an observer sees all right ascensions between 0^h and 24^h transit the meridian. Over an hour, the right ascension of the stars that culminate alters by about 1^h.

The regular motion of the stars is very suitable for forming the basis for another system of time reckoning, which is known as *sidereal time*. For any given place, it is defined as the right ascension of the stars that are on the meridian at that particular instant. Because of this definition, sidereal time may be determined directly by observation of the sky.

The need for sidereal time, in addition to solar time, arises from the slight difference between the length of a solar day and one rotation of the Earth. Clocks, which we use continually to tell the time, are arranged to divide the day into 24 hours. Here, one day is the alternation of light and dark, which is determined by the Sun, and 24 hours are the amount of time, on average, between one meridian

passage of the Sun and the next. If, however, we measure the corresponding interval for a star, we find that this takes only $23^h56^m4^s.091$. This shorter period, known as a sidereal day in distinction to a solar day, is exactly the duration of one rotation of the Earth. The cause of the difference of about 4^m arises from the Earth's year-long orbit around the Sun. Because of this motion, the Sun's right ascension changes by $360° \equiv 24^h$ per year, which is about 4^m per day. So the time between two meridian transits of the Sun is greater than that for a star by this amount.

For any given Universal Time UT, the sidereal time at Greenwich may be determined from

$$\begin{aligned}\Theta_0 \;=\; & 24110^s.54841 + 8640184^s.812866 \cdot T_0 + 1.0027379093 \cdot \mathrm{UT} \\ & +0^s.093104 \cdot T^2 - 0^s.0000062 \cdot T^3 \end{aligned} \tag{3.6}$$

where

$$T_0 = \frac{\mathrm{JD}_0 - 2451545}{36525} \quad \text{and} \quad T = \frac{\mathrm{JD} - 2451545}{36525} \quad .$$

JD and JD_0 are the Julian Date of the time of observation, and the Julian Date at 0^h UT on the date of observation.

```
//-------------------------------------------------------------------------
// GMST: Greenwich mean sidereal time
//   MJD       Time as Modified Julian Date
//   <return>  GMST in [rad]
//-------------------------------------------------------------------------
double GMST (double MJD)
{
  const double Secs = 86400.0;        // Seconds per day
  double MJD_0, UT, T_0, T, gmst;
  MJD_0 = floor ( MJD );
  UT    = Secs*(MJD-MJD_0);     // [s]
  T_0   = (MJD_0-51544.5)/36525.0;
  T     = (MJD  -51544.5)/36525.0;
  gmst  = 24110.54841 + 8640184.812866*T_0 + 1.0027379093*UT
          + (0.093104-6.2e-6*T)*T*T;      // [sec]
  return  (pi2/Secs)*Modulo(gmst,Secs);   // [Rad]
}
```

For a position with a geographical longitude λ, the local sidereal time differs by $\lambda/15°$ hours from the sidereal time at Greenwich:

$$\Theta = \Theta_0 + \lambda \cdot 1^h/15° \quad . \tag{3.7}$$

Here, longitude λ is reckoned *positive towards the east* (for Munich in Germany, for example, $\lambda = +11^\circ.6$). The hour angle τ of a star of right ascension α is therefore given by

$$\tau = \Theta - \alpha \quad . \tag{3.8}$$

As an example for the use of sidereal time in the conversion of right ascension and hour angle, the following sample program computes the azimuth and altitude of the star Deneb (α Cyg) for a given place and time:

```
double lambda = +11.6*Rad;                      // Observing site Munich
double phi    = +48.1*Rad;
double RA     = 15.0*Ddd(20,41,12.8)*Rad        // Equ. coord. Deneb [rad]
double Dec    =      Ddd(45,15,25.8)*Rad
double ModJD  = Mjd ( 1993,8,1, 21,0,0.0 );    // Aug. 1 1993, 21:00 UT

double tau,h,Az;

tau = GMST(ModJD) + lambda - RA;                // Hour angle [rad]
Equ2Hor ( Dec,tau,phi, h,Az );                  // Azimuth and altitude

cout << "Deneb:" << endl                        // Output
     << setprecision(2)
     << "Altitude   =" << setw(7) << Deg*h  << " deg"
     << "Azimuth    =" << setw(7) << Deg*Az << " deg";
```

3.4 Universal Time and Ephemeris Time

We have already frequently used the concept of 'Time', without considering it in more detail. In astronomy, however, we encounter a whole series of ways of reckoning time that are employed alongside one another. In general, they may be divided into two classes, the most important members of which are *dynamical time* (TT, TDB) and *Universal Time* (UT). The different objectives that are the reasons for these different times need to be explained in moderate detail.

The concept behind dynamical time is that it is a time that allows astronomical processes to be described in terms of physics. One second of dynamical time is defined by the period of a specific transition occurring in the caesium atom, and is correspondingly measured nowadays by atomic clocks. Since 1984, dynamical time has replaced *Ephemeris Time* (ET). The basis on which the latter had previously been established was that of tables of the motion of the Sun, the Moon, and the planets, which had been calculated according to classical mechanics. Ephemeris Time could be determined by comparing the observed positions of these celestial bodies with the ephemerides calculated earlier. The grounds for the introduction of dynamical time lay in relativity theory, which established that there is no truly universally applicable time, but that its rate also depends on the position and motion of the reference system in which time is being measured. This necessitates, in particular, differentiating between Terrestrial (Dynamical) Time (TT or TDT), which refers to the centre of the Earth, and Barycentric Dynamical Time (TDB), which relates to the centre of gravity of the Solar System. For our purposes, however, the three forms of time ET, TT, and TDB may be regarded as equivalent.

In contrast to Ephemeris Time and dynamical time, *Universal Time* (UT) is a *non-uniform* timescale. UT is the best current realization of a solar time. It was

established to try to ensure that over several thousand years one day has an average length of 24 hours. But the result of this is that the length of one second of Universal Time is not constant, because the actual mean length of a day depends on the rotation of the Earth and the apparent motion of the Sun (i.e., the length of the year). Unfortunately, it is not possible to determine Universal Time by a suitable conversion from dynamical time, because the rotation of the Earth cannot be predicted accurately. Every change in the Earth's rotation alters the length of the day, and must therefore be taken into account in UT. Universal Time is therefore defined as a function of sidereal time, which directly reflects the rotation of the Earth. For any particular day, 0^h UT is defined as the instant at which Greenwich Mean Sidereal Time (GMST) has the value

$$\begin{aligned} \mathrm{GMST}(0^h\mathrm{UT}) &= 24110^s.54841 + 8640184^s.812866 \cdot T_0 \\ &\quad + 0^s.093104 \cdot T_0^2 - 0^s.0000062 \cdot T_0^3 \end{aligned}$$

where

$$T_0 = \frac{\mathrm{JD}(0^h\mathrm{UT}) - 2451545}{36525} \quad .$$

We have already encountered this equation in Sect. 3.3 where we used it to calculate sidereal time for a given instant of Universal Time. We have now, however, established that sidereal time is the only observable quantity from which we can derive Universal Time.

The difference between Universal Time and Ephemeris Time or Terrestrial Time may be determined only retrospectively. Table 3.1 summarizes the value of ΔT=ET-UT (ET=TT=TDB) over the course of the last century. At present ΔT is increasing by about 0.5 to 1.0 seconds per year.

Table 3.1. Value of ΔT=ET−UT (in s) for 1900-2000

Year	ET-UT	Year	ET-UT	Year	ET-UT	Year	ET-UT
1900	-2.72						
1905	3.86	1930	24.02	1955	31.07	1980	50.54
1910	10.46	1935	23.93	1960	33.15	1985	54.34
1915	17.20	1940	24.33	1965	35.73	1990	56.86
1920	21.16	1945	26.77	1970	40.18	1995	60.82
1925	23.62	1950	29.15	1975	45.48	2000	63.83

Clock time, which we use for everyday purposes, is derived from *Coordinated Universal Time* (UTC). UTC is obtained from atomic clocks, and therefore has the same rate as Terrestrial Time. By the use of leap seconds, which may be inserted once or twice a year, care is taken to ensure that UTC never deviates more than 0.9 seconds from Universal Time, UT. UTC is the basis for time measurement over the whole of the Earth. Every point is part of a specific time zone, within which the official time differs by whole hours (or half hours) from UTC. In practice, the time zones have been arranged to agree with geographical features and, in particular, with national borders, to avoid having several time zones across a single country. The time zone employed in a particular area is established by international agreement and may be determined from a chart of the time zones.

At this point we may well enquire which time we should use when we want to calculate the rising and setting times of the Sun and the Moon. The preceding definitions give us the following general rule of thumb for the use of Universal Time and Ephemeris Time:

- Universal Time (UT) is used to calculate sidereal time.
- Ephemeris Time (ET) or dynamical time (TT, TDB) are used to calculate solar, lunar, or planetary ephemerides.

Let us take as an example the calculation of the altitude of the Moon, and consider the individual steps in the process. Let us take the time as being 1982 January 1, 0^h Central European Time, and the observing site as Munich ($\varphi = 48.^\circ1$, $\lambda = -11.^\circ6$). We can ignore the slight difference between UTC and UT, so we obtain the Universal Time from UT=CET-1^h. As discussed earlier, Universal Time has to be used in calculating sidereal time, while Ephemeris Time is used in calculating the coordinates of the Moon. We obtain the conversion as shown in this portion of a program:

```
double lambda = +11.6*Rad;                    // Geogr. coord. Munich
double phi    = +48.1*Rad;
double MjdCET = Mjd ( 1982,1,1, 0,0,0.0 );    // Jan. 1, 1982, 00:00 CET
double CET_UT = 1.0                           // Zone time difference [h]
double ET_UT  = 52.17;                        // in sec; for 1982

double MjdUT,MjdET, RA,Dec, sinH;

MjdUT  = MjdCET - CET_UT/24.0;                // MJD Universal Time
MjdET  = MjdUT  + ET_UT/86400.0;              // MJD Ephemeris Time

MiniMoon ( MjdET,RA,Dec);                     // Equatorial coordinates

tau  = GMST(MjdUT) + lambda - RA;             // Hour angle [rad]
sinH = sin(phi)*sin(Dec)                      // Sine of altitude
       + cos(phi)*cos(Dec)*cos(tau);          //   above the horizon
```

We can also ask what errors occur if we ignore the difference between Universal Time and Ephemeris Time. If the coordinates of the Moon are calculated with the simplification that we use UT instead of ET, the example just given becomes:

```
MjdUT  = MjdCET - CET_UT/24.0;                // MJD Universal Time

MiniMoon ( MjdUT,RA,Dec);                     // Equatorial coordinates

tau  = GMST(MjdUT) + lambda - RA;             // Hour angle [rad]
sinH = sin(phi)*sin(Dec)                      // Sine of altitude
       + cos(phi)*cos(Dec)*cos(tau);          //   above the horizon
```

Although the sidereal time is calculated correctly, the coordinates of the Moon are now calculated for a time that is ΔT=ET−UT too *early*. At present, ΔT is about one minute, an interval of time in which the Moon moves by about 30 arc-seconds. This error is even smaller than the error obtained by using the simplified `MiniMoon` function. The times of moonrise and moonset will be in error by only about 3^s. For the Sun, these values are even smaller. In the `Sunset` program, we can therefore ignore ΔT with a clear conscience. In later applications, such as the calculation of exact planetary positions or in predicting stellar occultations, it is, however, essential to distinguish carefully between the various forms of time. This is why we have discussed in detail the various methods of reckoning astronomical time.

3.5 Parallax and Refraction

So far, we have always given the right ascension and declination of a celestial body in geocentric equatorial coordinates, i.e., in a system with its origin at the centre of the Earth. We are not at the centre of the Earth, however, and observe from its surface. What differences does this cause between the coordinates that are calculated and those actually observed?

In observing fixed stars, in practice no difference is found between geocentric coordinates and the *topocentric* coordinates that are obtained by parallel translation of the zero point of the coordinates from the centre of the Earth to the position of the observer on the surface. The distance of the fixed stars is enormous in comparison with the distance between the observer and the centre of the Earth. But when we are dealing with nearer celestial bodies, such as the planets, the Sun, or the Moon, there are marked differences.

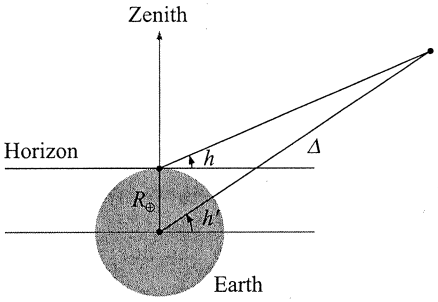

Fig. 3.2. Parallax

This difference is known as *parallax* and causes an object observed from the surface of the Earth to appear slightly lower than it would if it were observable from the centre of the Earth. The topocentric altitude h is less than the geocentric altitude h' (see Fig. 3.2). If the object is at the zenith, the parallax disappears. As the altitude decreases, however, the parallax becomes increasingly important, and becomes a maximum when the altitude $h = 0°$. This is known as the *horizontal parallax* and is calculated from

$$\pi = \arcsin(R_\oplus/\Delta) \quad , \tag{3.9}$$

where $R_\oplus \approx 6378$ km, the distance of the observer from the centre of the Earth, and Δ the *geo*centric distance of the object. If we substitute the distance of the Sun in (3.9) then we obtain a value of $\pi_\odot = 8''.8$. For the Moon, our nearest neighbour, on the other hand, the horizontal parallax amounts to the no longer insignificant amount of

$$\pi_{\text{Moon}} = \arcsin\left(\frac{6378\text{ km}}{384400\text{ km}}\right) \approx 57' \quad .$$

We can already see that for calculating rising and setting times we must use the topocentric altitude rather than the geocentric one. Before that, however, yet another effect must be described.

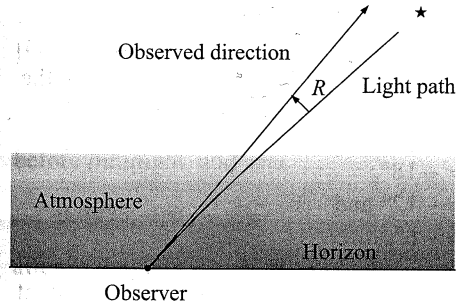

Fig. 3.3. Refraction

In passing from the vacuum of space into the optically denser atmosphere of the Earth, rays of light are deviated, in accordance with the law of refraction, towards the vertical (see Fig. 3.3). This is known as *atmospheric refraction*. To an observer on the ground, stars will appear to have been 'raised' slightly. A ray of light entering the atmosphere at a low angle has a longer path through layers of air of differing densities (and refractive indexes) than a ray entering vertically. Atmospheric refraction is therefore greater, the lower the altitude of the object. The maximum value of about 34 arc-minutes is attained at the horizon. Some further data are given in Table 3.2. To obtain the observed altitude, R should be added to the calculated topocentric altitude. Atmospheric refraction also depends on the density and temperature of the air. The figures quoted should therefore be taken as average values. Comprehensive tables of normal refraction and numerical approximation formulae may be found in appropriate reference works.

Table 3.2. Values of refraction near the horizon

h	10°	5°	2°	1°	0°
R	5′31″	10′15″	19′7″	25′36″	34′

We have seen that when dealing with rising and setting times, we must use topocentric altitudes. In addition, we also have to take refraction near the horizon into account. Because rising and setting times are always referred to the upper limb

of the Sun or the Moon, we also need to make allowance for the apparent radius $s_\odot$ or s_{Moon}. At the instant the body rises or sets, it therefore has the *geo*centric altitude

$$h_{\text{R/S}} = 0° + \pi - R_{h=0} - s \quad . \tag{3.10}$$

Our task now consists of finding, for a given date, the time at which the body being observed reaches an altitude of $h = h_{\text{R/S}}$. Because there is no sense in trying to obtain the times to an accuracy better than a few minutes, we can forgo accurate calculation of π and s and use average values. The values normally taken in calculating rising and setting times are therefore:

- Sunrise or sunset : $h_{\text{R/S}} = -0°50'$,
- Moonrise or moonset : $h_{\text{R/S}} = +0°08'$,
- for Stars or Planets : $h_{\text{R/S}} = -0°34'$.

As our program also calculates the times of the beginning and end of twilight, the necessary definitions should also be discussed here. The types of twilight are

- *astronomical* twilight when $h_\odot = -18°$,
- *nautical* twilight when $h_\odot = -12°$ and
- *civil* twilight when $h_\odot = -6°$.

No further corrections are applied to these values for parallax, refraction, or apparent semi-diameter. Twilight therefore begins or ends, by definition, when the *geocentric* altitude of the Sun attains one of the values quoted. What remains to be solved now, is the general problem of finding the instant of time at which a celestial body reaches a specified altitude.

3.6 Rising and Setting Times

Equation (3.5) allows us to determine the altitude h, given the geographical latitude φ, the declination δ, and the hour angle τ. The hour angle is obtained for a given local sidereal time from (3.8).

If, on the other hand, we have the altitude h, we can rearrange (3.5), and obtain a relationship for the hour angle:

$$\cos\tau = \frac{\sin h - \sin\varphi \sin\delta}{\cos\varphi \cos\delta} \quad . \tag{3.11}$$

We can therefore calculate the hour angle of a star when it is observed at an altitude h above the horizon. The equation is not, however, valid for every possible value of h, but only for the altitudes that the star may actually attain. At a geographical latitude of $\varphi = 50°$, a star of declination $\delta = 60°$, for example, is circumpolar, and may be observed only between $h = 20°$ and $h = 80°$. If, for a predetermined altitude h, the value of $|\cos\tau|$ is consistently greater than 1, then the star will always be found either above or below that altitude.

For the rising and setting of stars, we substitute the altitude $h_{\text{R/S}} = -0°34'$ in (3.11), the hour angle thus obtained being expressed in units of time ($15° \equiv 1^{\text{h}}$). It gives the number of hours in sidereal time that a star that is just rising requires to reach culmination. A sidereal day is equal to $23^{\text{h}}56^{\text{m}}4^{\text{s}}091$ solar time, giving a factor of $23^{\text{h}}56^{\text{m}}4^{\text{s}}091/24^{\text{h}} = 0.9972696$ by which an interval stated in sidereal time may be converted into solar time. If we multiply the hour angle by this factor, we obtain what is known as the *semi-diurnal arc*. This is half the overall time that the star is visible above the horizon.

For a given right ascension $\alpha_\star$ of a star, the sidereal time at the instant of rising or setting is given by

$$\Theta_{\text{R/S}} = \begin{cases} \alpha_\star - \tau & \text{for rise} \\ \alpha_\star + \tau & \text{for set} \end{cases} \quad .$$

If the Local Sidereal Time at 0^{h} is given by Θ_0, then the star rises

$$0.9973 \cdot (\Theta_0 - \Theta_{\text{R}})$$

before midnight, and sets

$$0.9973 \cdot (\Theta_{\text{S}} - \Theta_0)$$

after midnight.

In contrast to the coordinates of the stars, those of the Sun and the Moon alter noticeably during the course of a day. To calculate the hour angle and the times of rising and setting from the equations just given, however, we require the right ascension and declination at the moment the bodies cross the horizon. This requires an iterative procedure. The rising and setting times are first calculated for the coordinates at an arbitrary time for that particular day (in general, $t = 12^{\text{h}}$ is chosen). With the approximation thus obtained, improved coordinates may be determined, and these then again used to obtain more accurate rising and setting times. In the case of the Moon, these steps are repeated until the times obtained differ by less than one minute. For the Sun and the planets, the first iteration gives a sufficiently accurate result.

Unfortunately, there are a few problem cases, particularly in determining the times of moonrise and moonset, where the simple iteration method is difficult to apply. The following points need to be borne in mind:

- As the Moon orbits the Earth from West to East, it remains longer in the sky than stars with the same declination. As a result, moonrise is, on average, about 50^{m} later each day. This means that there is one day in every month on which the Moon does not rise, and also another day on which the Moon does not set. To give an example: On 1988 February 8 at Munich, the Moon rose at $23^{\text{h}}30^{\text{m}}$ and set at $9^{\text{h}}41^{\text{m}}$ on the following day. Because of its motion relative to the stars, it then remained below the horizon until after midnight. So the next moonrise did not take place on February 9, but on February 10, at $0^{\text{h}}43^{\text{m}}$.

- At high geographical latitudes, the Sun and the Moon frequently remain below the horizon for days. The rising and setting times are then primarily determined by the daily change in declination. Possible grazing contacts with the horizon are thus difficult to establish using the iteration method.

Because of this, the Sunset program is based on another principle. The rising and setting times are determined by inverse interpolation of a series of solar and lunar altitudes. This does require a greater computational effort, but leads to a significant simplification of the program's structure.

3.7 Quadratic Interpolation

If we calculate a table of altitudes at hourly intervals, it is then possible to represent the behaviour of the altitudes as a function of time through a simple interpolation function. Quadratic interpolation is suitable for our purposes: from three values we may calculate the coefficients of a parabola, which approximates the behaviour of the function between the given points. We work from three values of the function:

$$y_- = f(x=-1), \quad y_0 = f(x=0) \quad \text{and} \quad y_+ = f(x=+1) \quad .$$

We require the coefficients a, b and c of a parabola

$$y = a \cdot x^2 + b \cdot x + c \quad , \tag{3.12}$$

which runs through the points $(-1, y_-)$, $(0, y_0)$ and $(1, y_+)$. Substituting into the formula for the parabola, we have the three equations

$$\begin{aligned} y_- &= a - b + c \\ y_0 &= c \\ y_+ &= a + b + c \quad , \end{aligned}$$

from which the required coefficients of the parabola may be obtained:

$$\begin{aligned} a &= (y_+ + y_-)/2 - y_0 \\ b &= (y_+ - y_-)/2 \\ c &= y_0 \quad . \end{aligned} \tag{3.13}$$

For $b^2 \geq 4ac$, this parabola is zero at the points:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{3.14}$$

and has an extreme value

$$\begin{aligned} x_E &= -b/2a \\ y_E &= a \cdot x_E^2 + b \cdot x_E + c \quad . \end{aligned} \tag{3.15}$$

The Quad function solves these equations and also chooses just the roots that lie between $x = -1$ and $x = +1$.


```
//-----------------------------------------------------------------------------
// Quad: Quadratic interpolation
//       Performs root finding and search for extreme values based on three
//       equidistant values of a function.
//   y_minus   Value of function at x = -1
//   y_0       Value of function at x =  0
//   y_plus    Value of function at x =  1
//   xe        Abscissa of extremum (may be outside [-1, 1])
//   ye        Value of function at xe
//   root1     First root found
//   root2     Second root found
//   n_root    Number of roots found in [-1, 1]
//-----------------------------------------------------------------------------
void Quad ( double y_minus, double y_0, double y_plus,
            double& xe, double& ye, double& root1,
            double& root2, int& n_root )
{
  double a,b,c, dis, dx; n_root = 0;
  // Coefficients of interpolating parabola y=a*x^2+b*x+c
  a  = 0.5*(y_plus+y_minus) - y_0;
  b  = 0.5*(y_plus-y_minus);
  c  = y_0;
  // Find extreme value
  xe = -b/(2.0*a);
  ye = (a*xe+b) * xe + c;
  dis = b*b - 4.0*a*c; // Discriminant of y=a*x^2+b*x+c

  if (dis >= 0) // Parabola has roots  {
    dx = 0.5 * sqrt (dis) / fabs (a);
    root1 = xe - dx;
    root2 = xe + dx;
    if (fabs(root1) <= 1.0) ++n_root;
    if (fabs(root2) <= 1.0) ++n_root;
    if (root1       < -1.0) root1 = root2;
  }
}
```

3.8 The SUNSET Program

The program Sunset calculates the rising and setting times of the Sun and the Moon, as well as the beginning and ending of nautical twilight over a period of 10 days. Data to be entered are the starting date, the geographical coordinates of the observing site, and the difference between Local Time and Universal Time.

The search for the times of the various events is made by the method shown diagrammatically in Fig. 3.4. The SinAlt function calculates the sine of the solar or lunar altitude at hourly intervals. These values are interpolated in Quad and examined for zero points. If a root is found, and the Sun or Moon was below the horizon at the beginning of the day, then the event is a sunrise or moonrise. Otherwise, the Sun or Moon is setting. Finally, it may happen that two zero points

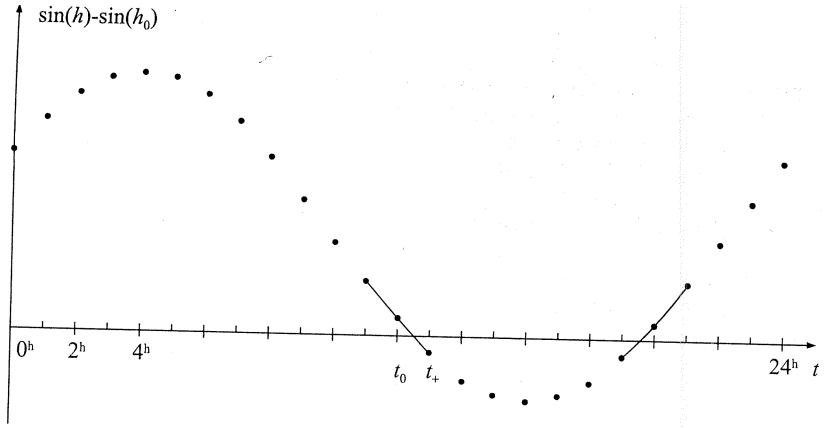

Fig. 3.4. Determining rising and setting times

are discovered in the interval being examined. To determine which of these two points is a rising and which a setting, we check to see if the altitude at the vertex is above or below the horizon. The process continues until the end of the day. It is then possible to determine whether the celestial body being considered is circumpolar, or whether it remains below the horizon for the entire day.

The times of civil, nautical, or astronomical twilight are calculated in a similar manner, in that a constant value of $\sin(-6°)$, $\sin(-12°)$ or $\sin(-18°)$ is subtracted from the sine of the altitude of the horizon.

```
//--------------------------------------------------------------------------
// File:    Sunset.cpp
// Purpose: Rising and setting times of Sun and Moon and twilight times
// (c) 1999 Oliver Montenbruck, Thomas Pfleger
//--------------------------------------------------------------------------
#include <cmath>
#include <iomanip>
#include <iostream>
#include "APC_Const.h"
#include "APC_Math.h"
#include "APC_Moon.h"
#include "APC_Spheric.h"
#include "APC_Sun.h"
#include "APC_Time.h"

using namespace std;
// Events to search for
enum enEvent {
  Moon,      // indicates moonrise or moonset
  Sun,       // indicates sunrise or sunset
  CivilTwi,  // indicates Civil twilight
  NautiTwi,  // indicates nautical twilight
  AstroTwi   // indicates astronomical twilight
};
```



```
//--------------------------------------------------------------------------
// SinAlt: Sine of the altitude of Sun or Moon
// Input:
//   Event     Indicates event to find
//   MJD0      0h at date to investigate (as Modified Julian Date)
//   Hour      Hour
//   lambda    Geographic east longitude in [rad]
//   Cphi      Cosine of geographic latitude
//   Sphi      Sine of geographic latitude
//   <return>  Sine of the altitude of Sun or Moon at instant of Event
//--------------------------------------------------------------------------
double SinAlt ( enEvent Event, double MJD0, double Hour,
                double lambda, double Cphi, double Sphi )
{
  double  MJD, T, RA, Dec, tau;
  MJD = MJD0 + Hour/24.0;
  T   = (MJD-51544.5)/36525.0;
  if ( Event == Moon ) MiniMoon ( T, RA, Dec);
  else MiniSun ( T, RA, Dec );
  tau = GMST(MJD) + lambda - RA;
  return ( Sphi*sin(Dec)+Cphi*cos(Dec)*cos(tau) );
}

//--------------------------------------------------------------------------
// GetInput: Prompts user for input
//   MJD       Time as Modified Julian Date
//   lambda    Geographic east longitude of the observer in [rad]
//   phi       Geographic latitude of the observer in [rad]
//   zone      Difference local time - universal time in [d]
//   twilight  Indicates civil, nautical or astronomical twilight
//--------------------------------------------------------------------------
void GetInput ( double& MJD, double& lambda,
                double& phi, double& zone, enEvent& twilight )
{
  int  year, month, day;
  char cTwilight;
  cout << " First date (yyyy mm dd)                ... ";
  cin >> year >> month >> day; cin.ignore(81,'\n');
  cout << endl;
  cout << " Observing site:  east longitude [deg] ... ";
  cin  >> lambda; cin.ignore(81,'\n');
  cout << "                 latitude [deg]        ... ";
  cin  >> phi; cin.ignore(81,'\n');
  cout << "                 local time - UT [h]   ... ";
  cin  >> zone; cin.ignore(81,'\n');
  cout << " Twilight definition (c, n or a)       ... ";
  cin  >> cTwilight; cin.ignore(81,'\n');
  switch (tolower(cTwilight)) {
    case 'c': twilight = CivilTwi; break;
    case 'a': twilight = AstroTwi; break;
    case 'n':
    default:  twilight = NautiTwi;
  }
```

```
  lambda*=Rad;  phi*=Rad;
  zone/=24.0;
  MJD = Mjd(year,month,day) - zone;
}

//---------------------------------------------------------------------
// FindEvents: Search for rise/set/twilight events of Sun or Moon
//   Event     Indicates event to search for
//   MJD0h     0h at desired date as Modified Julian Date
//   lambda    Geographic east longitude of the observer in [rad]
//   phi       Geographic latitude of the observer in [rad]
//   LT_Rise   Local time of rising or beginning of twilight
//   LT_Set    Local time of setting or end of twilight
//   rises     Event takes place
//   sets      Event takes place
//   above     Sun or Moon is circumpolar
//---------------------------------------------------------------------
void FindEvents ( enEvent Event, double MJD0h, double lambda, double phi,
                  double& LT_Rise, double& LT_Set,
                  bool& rises, bool& sets, bool& above )
{
  static const double sinh0[5] = {
    sin(Rad*( +8.0/60.0)), // Moonrise              at h= +8'
    sin(Rad*(-50.0/60.0)), // Sunrise               at h=-50'
    sin(Rad*(   - 6.0  )), // Civil twilight        at h=-6 deg
    sin(Rad*(   -12.0  )), // Nautical twilight     at h=-12deg
    sin(Rad*(   -18.0  )), // Astronomical twilight at h=-18deg
  };
  const double Cphi = cos(phi);
  const double Sphi = sin(phi);
  double hour = 1.0;
  double y_minus, y_0, y_plus;
  double xe, ye, root1, root2;
  int nRoot;

  // Initialize for search
  y_minus = SinAlt(Event, MJD0h, hour-1.0, lambda,Cphi,Sphi )-sinh0[Event];
  above = (y_minus>0.0);  rises = false;  sets  = false;
  // loop over search intervals from [0h-2h] to [22h-24h]
  do {
    y_0    = SinAlt( Event,MJD0h,hour    , lambda,Cphi,Sphi )-sinh0[Event];
    y_plus = SinAlt( Event,MJD0h,hour+1.0, lambda,Cphi,Sphi )-sinh0[Event];
    // find parabola through three values y_minus,y_0,y_plus
    Quad ( y_minus, y_0, y_plus, xe, ye, root1, root2, nRoot );
    if ( nRoot==1 ) {
      if ( y_minus < 0.0 )
        { LT_Rise = hour+root1;  rises = true; }
      else
        { LT_Set  = hour+root1;  sets  = true; }
    }
    if ( nRoot == 2 ) {
      if ( ye < 0.0 )
        { LT_Rise = hour+root2;  LT_Set = hour+root1; }
      else
        { LT_Rise = hour+root1;  LT_Set = hour+root2; }
      rises = true; sets = true;
    }
    y_minus = y_plus;     // prepare for next interval
    hour += 2.0;
  }
  while ( !( ( hour == 25.0 ) || ( rises && sets ) ) );
}

//---------------------------------------------------------------------
//
// Main program
//
//---------------------------------------------------------------------
void main() {

  // Variables
  bool     above, rise, sett;
  int      iEvent, day;
  double   lambda, zone, phi;
  double   date, start_date, LT_Rise, LT_Set;
  enEvent  Event, Twilight;

  // Title
  cout << endl
       << "      SUNSET: solar and lunar rising and setting times  " << endl
       << "        (c) 1999 Oliver Montenbruck, Thomas Pfleger     " << endl
       << endl;
  // Prompt user for input
  GetInput (start_date, lambda, phi, zone, Twilight );
  // Header
  cout << endl
       << "    Date"
       << "              Moon              Sun             Twilight " << endl
       << "         "
       << "           rise/set          rise/set        beginning/end"
       << endl << endl;

  // loop over 10 subsequent days
  for (day=0; day<10; day++) {
    // current date
    date = start_date + day;
    cout << " " << DateTime(date+zone) << "  ";
    // loop over cases (Moon, Sun, Twilight)
    for (iEvent=0; iEvent<=2; iEvent++)  {
      // After events for Moon and Sun: specify kind of twilight event
      Event = (iEvent<2) ? (enEvent) iEvent : Twilight;
      // Now try to find times of events
      FindEvents ( Event, date, lambda, phi,
                   LT_Rise, LT_Set, rise, sett, above );
      // Output
      if ( rise || sett )  {
```

```
        if ( rise )
          cout << "    " << Time(LT_Rise,HHMM) << " ";
        else
          cout << "    ----- ";
        if ( sett )
          cout << "    " << Time(LT_Set,HHMM) << " ";
        else
          cout << "    ----- ";
      }
      else
        if ( above ) {
          if ( Event >= CivilTwi )
            cout << "     always bright ";
          else
            cout << "    always visible ";
        }
        else {
          if ( Event >= CivilTwi )
            cout << "      always dark  ";
          else
            cout << "  always invisible";
        }
    }
    cout << endl;
  }

  // Trailer
  cout << endl;
  cout << " all times in local time (=UT"
       << showpos << 24.0*zone << noshowpos << "h)"  << endl;
}
```

We will demonstrate the use of Sunset with some examples, which will show some of the factors that arise in calculating rising and setting times.

Sunset calculates the desired times for a period of ten days, beginning with the date entered. The geographical longitude is entered with positive values for those places east of the Greenwich meridian. Finally the difference between Local Time and UT is entered in hours. A value of 1, for example, would be used for Central European Time (CET), or 2 for Central European Summer Time (CEST). For time zones in North America, negative values are entered: -5 for Eastern Standard Time (EST), for example, or -8 for Pacific Standard Time (PST). Furthermore, the twilight definition may be specified to choose between civil twilight ('c'), nautical twilight ('n') and astronomical twilight ('a').

Let us, for example, calculate the rising and setting times of the Sun and Moon for Munich ($\lambda = 11.^{\circ}6$ East, $\varphi = 48.^{\circ}1$ North), beginning with 2000 March 23. We require the output for Central European Time (1^{h} different from Universal Time). First the starting date must be entered in the form: Year, Month, Day. Then Sunset asks for the geographical coordinates and the difference between LT and UT. (In the following, all data entered are shown in italic.) From the data that we have entered, Sunset determines the rising and setting times of the Sun and Moon, as well as the

beginning and end of twilight. As the value for the altitude of the Sun that is used in the program is set at -12°, the output here and in the following example is for *nautical* twilight.

```
     SUNSET: solar and lunar rising and setting times
       (c) 1999 Oliver Montenbruck, Thomas Pfleger

First date (yyyy mm dd)                ... 2000 03 23

Observing site:  east longitude [deg] ... +11.6
                 latitude [deg]       ... +48.1
                 local time - UT [h]  ... +1
Twilight definition (c, n or a)       ... n
```

Date	Moon rise/set		Sun rise/set		Twilight beginning/end	
2000/03/23	22:12	08:01	06:10	18:31	05:02	19:39
2000/03/24	23:17	08:28	06:08	18:32	05:00	19:41
2000/03/25	-----	08:58	06:06	18:34	04:58	19:42
2000/03/26	00:18	09:33	06:04	18:35	04:56	19:44
2000/03/27	01:16	10:13	06:02	18:37	04:53	19:46
2000/03/28	02:08	10:59	06:00	18:38	04:51	19:47
2000/03/29	02:55	11:51	05:58	18:40	04:49	19:49
2000/03/30	03:37	12:48	05:56	18:41	04:47	19:50
2000/03/31	04:13	13:51	05:54	18:43	04:45	19:52
2000/04/01	04:44	14:56	05:52	18:44	04:42	19:54

```
all times in local time (=UT+1h)
```

It will be seen that there is no entry in the column under 'Moon rise' for 2000 March 25. On that night the Moon rose after midnight, so it is recorded under March 26. In general, there is one day per month on which it neither rises or sets.

To illustrate another interesting effect, let us calculate the rising and setting times for a fictitious point in Europe at latitude 65°, choosing 1989 June 15 as the starting date and Central European Summer Time as the local time. SUNSET gives the following output:

```
     SUNSET: solar and lunar rising and setting times
       (c) 1999 Oliver Montenbruck, Thomas Pfleger

First date (yyyy mm dd)                ... 1989 06 15

Observing site:  East longitude [deg] ... +10.0
                 latitude [deg]       ... +65.0
                 local time - UT [h]  ... +2.0
Twilight definition (c, n or a)       ... n
```

Date	Moon rise/set		Sun rise/set		Twilight beginning/end
1989/06/15	19:58	01:00	02:24	00:16	always bright

```
1989/06/16     22:26     23:53     02:23     00:18     always bright
1989/06/17   always invisible     02:22     00:19     always bright
1989/06/18   always invisible     02:21     00:20     always bright
1989/06/19   always invisible     02:20     00:21     always bright
1989/06/20   always invisible     02:20     00:22     always bright
1989/06/21     02:39     03:24     02:20     00:23     always bright
1989/06/22     01:35     06:21     02:20     00:23     always bright
1989/06/23     01:15     08:29     02:21     00:23     always bright
1989/06/24     01:01     10:25     02:22     00:22     always bright

all times in local time (=UT+2h)
```

In this case, after setting on June 16 at 23^h53^m, the Moon remains below the horizon and is therefore invisible until June 21 at 2^h39^m. Correspondingly, at high latitudes the Moon may remain above the horizon for days.

Naturally, similar events may occur with the Sun, which we call the polar night or polar daylight ('midnight Sun'). Using `Sunset` we can estimate how long polar night or polar daylight last at a specific place. To do this we vary the period over which calculations are carried out, until the Sun is first shown as 'always invisible'. Polar night begins on the corresponding day. The end of the polar night may be found in a precisely analogous way.

The result 'always bright' in the 'Twilight' column indicates that it never becomes fully dark. This may even occur at a latitude of only 50°, when we determine astronomical instead of nautical or civil twilight.

3.9 The PLANRISE Program

In the previous sections we became acquainted with the necessary tools for computing rising and setting times by the method of quadratic interpolation. The latter may also be used to handle exceptional cases of rising and setting time calculations, which may otherwise present severe difficulties. Nevertheless, we also want to describe a program that illustrates the iterative method of obtaining planetary rising and setting times. The commented source code of `Planrise` is provided on the enclosed CD.

The function of `Planrise` is based on the equations given in Sect. 3.6. For the computation of planetary positions we make use of the `PertPosition` function, which is described in Sect. 5.2. To compute the rising and setting times of a planet we first determine its (geocentric) equatorial coordinates α and δ for an arbitrary time on the given day. Equation (3.11) then yields the value $\cos\tau(\varphi, \delta)$ for the hour angle at the instant of rising or setting. If the right-hand side results in a value whose magnitude is larger than one, the planet never rises or sets on that day. From the declination and the geographical latitude, `Planrise` then determines whether the planet is continuously above or below the horizon.

Because the horizontal parallax and the apparent diameter have virtually no impact on the rising and setting times, a value of $h_{R/S} = -0°34'$ for the geometric altitude at the time of rising or setting is employed in the computation, which thus

takes refraction only into account. The iterative equation for determining the rising and setting times is then given by

$$t_{i+1} = t_i - 0.9973(\Theta(t_i) - \alpha(t_i) \pm \tau(\varphi, \delta(t_i))) \tag{3.16}$$

where

$$\tau = \arccos \frac{\sin h_{A/U} - \sin\varphi \sin\delta}{\cos\varphi \cos\delta} \quad . \tag{3.17}$$

Here the positive sign applies for the computation of rising times. $\Theta(t_i)$ and $\alpha(t_i)$ denote the local sidereal time and the planet's right ascension at the time t_i of the ith approximation. The conversion factor 0.9973 is used to convert between universal time and sidereal time, as mentioned earlier.

As with the computation of rising and setting times, we may also employ the relationship

$$t_{i+1} = t_i - 0.9973(\Theta(t_i) - \alpha(t_i)) \tag{3.18}$$

to determine the time of culmination of a planet.

To reduce computing times, `Planrise` makes use of various simplifications. Linear interpolation, for example, is used to obtain the planetary coordinates α and δ, as well as the local sidereal time Θ, from precomputed values at 0^h and 24^h on the same day.

As an example we will compute the rising and setting times of the planets on 1999 December 31 in Munich ($\lambda = 11.6°$East, $\varphi = 48.1°$North). All input data for the `Planrise` program are shown in italic.

```
PLANRISE: planetary and solar rising and setting times
          (c) 1999 Oliver Montenbruck, Thomas Pfleger

Date (yyyy mm dd)                          ... 1999 12 31

Observing site:  east longitude [deg] ... +11.6
                 latitude [deg]       ... +48.1
                 local time - UT [h]  ... +1.0

              rise    culmination    set

Sun           08:04       12:16      16:29
Mercury       07:33       11:37      15:41
Venus         04:52       09:30      14:08
Mars          10:33       15:35      20:37
Jupiter       12:29       19:10      01:55
Saturn        13:09       20:10      03:14
Uranus        10:02       14:45      19:28
Neptune       09:25       13:57      18:29
Pluto         05:11       10:22      15:32

all times in local time (=UT+1h)
```

Subject to good weather a total of three planets (Mars, Jupiter, and Saturn) can thus be seen with the naked eye in the evening sky.