

Sunrise and Sunset Calculation

Source: <http://sci.tech--archive.net/Archive/sci.astro.amateur/2009-01/msg00425.html>

- *From:* "TR Oltrogge" <troltrogge@xxxxxxxxxxxx>
 - *Date:* Thu, 15 Jan 2009 00:21:41 GMT
-

Last spring/summer I posted my 'C' language program for calculating the time of sunrise and sunset for any (i.e. reasonable, not above arctic circle, etc) location on the earth. My logic was based on (1) solving elliptical orbits per Kepler, (2) determining earth's axis tilt by noting what happens at summer solstice, and (3) synchronizing GMT (for the earth's exact rotational position) with the sun's physical position by **assuming** they were correct at summer solstice.

Upon entering the new year I was curious if any constants (like orbital period, axis tilt, etc) might need to be updated for 2009. Well, I haven't actually investigated those "constants", but I **did** correct the error in my assumption for item (3). GMT and the sun's actual position are synchronized according to the Equation of Time (as we all know) and I have now used the date June 14th as one of those times when there is zero error. This has eliminated the six minute error I was getting relative to the published sunrise/sunset times on the U.S. government's Navy site. I'm always within one minute now! I now post my new code if anyone is interested. The comments have been changed to, hopefully, better explain what I'm doing in the program. The Equation of Time, BTW, is not needed in my program since vector mathematics handles the orientation of the sun and earth as a normal consequence, but I **did** have to refer to its basis to know how our illustrious scientists have pinned GMT down to its relationship to the prime meridian.

You can also download the source file (and an associated vector support header file) from...

<http://mysite.verizon.net/troltrogge/sunset.c>

and

<http://mysite.verizon.net/troltrogge/VCTRSUPPORT.h>

by right-clicking and using Save Target As

Tim

/*

SUNSET, a program to calculate the time of sunrise and sunset for any

Sunrise and Sunset Calculation

latitude, longitude, and date.

THE GEOMETRIC VECTOR SOLUTION

Geometrical observation leads us to conclude that sunrise and sunset occur at a time when a line from the sun (considered a point source) runs tangent to the earth and touches earth exactly at the point of the observer's latitude and longitude. We can test for this by checking if a vector from the sun to the observer's point on the surface of the earth is perpendicular to a vector from the center of the earth to the observer's point on the surface. The calculation must be done iteratively since the vector from the sun to the observer's point on the surface of the earth is not directly solvable by algebra. Therefore, the solution to the time of sunset is to compute these two vectors for various times during the day in question and make successively closer guesses until the perpendicular condition is reached.

The computation of these two vectors requires analysis of two types of motion.

(1) The position of the center of the earth relative to the sun at any time 't' in the earth's yearly elliptical orbit of the sun. From this position we construct the 'sun_earth' vector.

(2) The position of any point on the surface of the earth relative to the earth's center at any time 't' due to its daily rotation on its own axis. From this we construct the 'earth_observer' vector.

Both positions must be defined in the same coordinate system, which is a 3-dimensional Cartesian coordinate system with the sun at its origin. Vector algebra is then used to assess the orientation of these two positions to determine sunrise and sunset. To do this a third vector, 'sun_observer', which is a line from the sun to the observer's position on the surface of the earth, is formed from the sum of the 'sun_earth' and 'earth_observer' vectors. The 'sun_observer' and 'earth_observer' vectors are the two that are checked for perpendicularity.

THE PRIMARY COORDINATE SYSTEM

The primary 3-axis coordinate system has the sun at the origin. The earth's orbit is an ellipse in the X-Y plane with the sun at one of the foci. When the earth is at perihelion, or closest to the sun, it is on the positive X-axis with Y=0. From a point on the positive Z axis and looking down at

Sunrise and Sunset Calculation

the X–Y plane the earth's orbit is counter-clockwise. At perihelion the earth has an angle of zero radians relative to the sun. As the earth orbits counter-clockwise its angle from the sun increases until it is π radians at aphelion and 2π radians after one complete orbit.

CALCULATING THE sun_earth VECTOR AS A FUNCTION OF TIME t

The 'sun_earth' vector is determined by solving the problem of elliptical orbits as found by Kepler.

See http://en.wikipedia.org/wiki/Kepler's_laws_of_planetary_motion

To determine, using polar coordinates, the position of a planet on its elliptical orbit as a function of time Kepler devised the following procedure. Kepler's solution yields the earth's position from the sun as an angle measured from perihelion and a distance from the sun. The sun is at the origin of the polar coordinate system.

(Step 1) Compute the "mean anomaly" M from the equation

$$M = 2\pi * t / T$$

where ' M ' is what Kepler called the "mean" anomaly (really, just an angle), ' t ' is the time since perihelion, and ' T ' is the time for a full orbital period (perihelion to perihelion). ' M ' applies to a perfectly circular orbit.

(Step 2) Compute the "eccentric anomaly" E from the equation

$$M = E - e * \sin(E)$$

where ' E ' is what Kepler called the "eccentric" anomaly (again, just an angle), and ' e ' is the eccentricity of the ellipse. ' M ' is the result from Step 1. This cannot be done algebraically but is determined accurately enough with the first three terms of an infinite series.

(Step 3) Compute the "true anomaly" Θ from the equation

$$\tan(\Theta / 2) = \sqrt{(1 + e) / (1 - e)} * \tan(E / 2)$$

where ' Θ ' is what Kepler called the "true anomaly" (the actual angle, in polar coordinates, of the earth from the sun with the sun being the center of the polar coordinate system), ' e ' is the eccentricity of the ellipse, and ' E ' is the result from Step 2.

(Step 4) Compute the heliocentric distance r from the equation

Sunrise and Sunset Calculation

$$r = p / (1 + e * \cos(\text{Theta}))$$

where 'p' is the semi-latus rectum of the ellipse, 'e' is the eccentricity of the ellipse, and 'Theta' is the result from Step 3. The semi-latus rectum 'p' of an ellipse is derived from its relationship to the semi-major axis 'a' of an ellipse due to its eccentricity 'e' as defined in the equation

$$a = p / (1 - e^2)$$

and solved for 'p' as

$$p = a * (1 - e^2)$$

From the internet I determined the orbital period 'T' of the earth (anomalous) is 365 days 6 hours 13 minutes and 53 seconds, This time is from perihelion to perihelion. This is 31558433 seconds. This allows us to complete Step 1. The eccentricity 'e' of the earth's orbit is 0.0167. This allows us to complete Steps 2 and 3. The semi-major axis of the earth's orbit is 92955807.0 miles. This allows us to complete Step 4. These four steps are performed by the 'calc_sun_earth_vector' function in my code. The last action of this function is to convert the polar coordinates to the Cartesian ones I need. Thus, at the start of the 'main' function the first of the two vectors needed to solve our problem of the time of sunset is already available as a call to function 'calc_sun_earth_vector'.

CALCULATING THE earth_observer VECTOR AS A FUNCTION OF TIME t

The 'earth_observer' vector goes from the center of the earth to the observer's point on the surface of the earth. All we need to determine it are the relative X, Y, and Z displacements in the primary coordinate system at a given time 't'. We will do this in two steps. First, we will define a secondary coordinate system based on the axis of rotation of the earth and determine the 'earth_observer' component vector displacements in this secondary system for any time 't'. Second, we will transform the coordinates in this secondary system to the primary system using standard mathematics for coordinate transformation between two systems.

The secondary coordinate system is also a 3-axis Cartesian system. It uses the center of the earth as its origin. A line from the south pole to the north pole is its Z axis. A line from the center of the earth to the equator at a point along the prime meridian is the X-axis. Finally, a line from the center of the earth to the equator at a point along

Sunrise and Sunset Calculation

the 90th east longitude meridian is the Y-axis.

Using this secondary coordinate system we can compute the observer's location vector with the three equations...

$$Z = 4000 * \sin(\text{latitude})$$

$$Y = (4000 * \cos(\text{latitude})) * \sin(\text{longitude})$$

$$X = (4000 * \cos(\text{latitude})) * \cos(\text{longitude})$$

Notice that the familiar notation of east and west longitude designation must be replaced with the mathematical requirement that east longitude is used as-is (a positive number from 0 to 180) but west longitude is negated in sign (so 0 to 180 west longitude becomes 0 to -180 degrees).

COORDINATING GMT WITH THE PRIME MERIDIAN'S ORIENTATION IN SPACE

The time scale used throughout this program is GMT. This is the time for someone standing exactly on the prime meridian that runs through Greenwich, UK. Though there are 24 time zones on the earth's surface, each with added or subtracted hourly offsets from GMT, these serve only to give humans "sensible" values for the hours of the day so that it's basically 12 o'clock noon local time everywhere when the sun is highest in the sky. No matter what local time is in use, though, at any instant there is only one GMT value that applies to the entire earth. This same GMT value would be valid on the planet Mars, too, if people there were concerned about synchronizing their activities with those on earth! When you think about it, someone 100 miles east of Greenwich would see the sun rise a few minutes earlier and, therefore, will be directly south a few minutes *before* noon if his clock is set to GMT (as it normally would be). If that person insisted on having a clock correctly synchronized with the motion of the sun he would use a clock set forward by those few minutes from GMT. He would then have a clock set to local mean time (LMT). Since everyone setting their own clocks to their LMT invites confusion as people travel the time zone system has been defined. It could have been created with 7 or 10 or 50 or 100 zones depending on the amount of sun-position error relative to your clock one is willing to live with and how often one wants to cross zone boundaries when traveling. Less error means many more zones with the subsequent clock changing adjustments. More error gives the relative simplicity of fewer zones. The present system of 24 zones matches the number of (arbitrary) hours that have been defined in a day so it's a natural fit to our numbering system.

Now that we understand our local mean time (LMT) is *supposed* to be synchronized with the movement of the sun and that GMT is that specific local mean time for persons on the prime meridian it is

Sunrise and Sunset Calculation

a little disconcerting to find that GMT varies by about plus or minus a dozen minutes from true sun-position time during the year according to what is known as the Equation of Time. The variation is due to the elliptical orbit of the earth and its axis tilt. The Equation of Time is a function that tells us the difference between clock time and sundial time throughout the year. We do not need to include an adjustment for the Equation of Time since our geometric analysis of the time of sunset accounts for these effects. But since astronomers have set our clocks to keep synchronized with the sun for a *specific* point on the Equation of Time we must know where that point is so GMT can be made to correlate correctly with the sun's position. We need to know what its value is on specific dates so we can determine the exact orientation of our secondary coordinate system to the primary coordinate system. Specifically, we can use a particular GMT during the year known to have exact agreement of clock time with sundial time. That is, the particular GMT where the Equation of Time is zero. There are four such times during the year. They are in April, June, October, and December.

We will use the June date, for no particular reason. We will use noon GMT as the time of calculation on that date. At noon on the prime meridian we would see the sun exactly due south. This means the 'sun_earth' vector at that time goes from the sun to the center of the earth and pierces through the surface of the earth on a point that lies on the prime meridian. If we take the cross product of the 'sun_earth' vector with the already-computed 'tilt_z' vector we will have a vector perpendicular to both 'tilt_z' and 'sun_earth'. It will be a vector starting at the earth's center and leaving the surface at a point of 90 degrees east longitude. This is our 'tilt_y' vector. Taking the cross product of 'tilt_z' with 'tilt_y' will give us the 'tilt_x' that goes from the earth's center out through the prime meridian exactly at the equator. These 3 'tilt' vectors are defined in the primary coordinate system. The direction cosines they have with the basis vectors for the primary coordinate system allow us to convert any vector described using secondary coordinate system coordinates into one described in the primary coordinate system. We do this transformation in only one place: the fourth step of the 'calc_angle' function.

The earth rotates on its own axis approximately every 23 hours 56 minutes 4 seconds (a sidereal day). So even though the observer is fixed in geographic latitude and longitude his effective longitude increases by 360 degrees every 23h56m4s. Therefore, an incremented or decremented longitude is computed from the time of interest based on how much rotation the earth has gone through (positive or negative) since our arbitrary time when the Equation of Time is zero.

Sunrise and Sunset Calculation

The 'tilt' vectors are calculated by logic described at the beginning of the 'main' function.

CONVERSION OF COORDINATES BETWEEN TWO COORDINATE SYSTEMS

The coordinate conversion from the secondary system to the original system is accomplished via the general-purpose linear transformation algebraic solution found on the top of page 534 of my Van Nostrand Reinhold "Concise Encyclopedia of Mathematics" book. This solution handles the general case of both a translation of the origin and a rotation of the axes for two distinct coordinate systems. It requires the calculation of "direction cosines" for each of nine angles. Given any vectors aligned with the three axes for the two coordinate systems these nine direction cosines can be computed by using the dot-product relationship of two vectors. Because the dot-product of two vectors is the product of the magnitudes of both vectors multiplied by the cosine of the angle between them the cosine itself is the dot-product divided by the product of the magnitudes of both vectors. The calculation of the nine direction cosines follows:

$$\begin{aligned}a_{11} &= \text{dot-product}(\text{unit_x}, \text{tilt_x}) / (|\text{unit_x}| * |\text{tilt_x}|) \\a_{12} &= \text{dot-product}(\text{unit_x}, \text{tilt_y}) / (|\text{unit_x}| * |\text{tilt_y}|) \\a_{13} &= \text{dot-product}(\text{unit_x}, \text{tilt_z}) / (|\text{unit_x}| * |\text{tilt_z}|) \\a_{21} &= \text{dot-product}(\text{unit_y}, \text{tilt_x}) / (|\text{unit_y}| * |\text{tilt_x}|) \\a_{22} &= \text{dot-product}(\text{unit_y}, \text{tilt_y}) / (|\text{unit_y}| * |\text{tilt_y}|) \\a_{23} &= \text{dot-product}(\text{unit_y}, \text{tilt_z}) / (|\text{unit_y}| * |\text{tilt_z}|) \\a_{31} &= \text{dot-product}(\text{unit_z}, \text{tilt_x}) / (|\text{unit_z}| * |\text{tilt_x}|) \\a_{32} &= \text{dot-product}(\text{unit_z}, \text{tilt_y}) / (|\text{unit_z}| * |\text{tilt_y}|) \\a_{33} &= \text{dot-product}(\text{unit_z}, \text{tilt_z}) / (|\text{unit_z}| * |\text{tilt_z}|)\end{aligned}$$

The direction cosines are the 'Amn' coefficients in the following equations. The translational distance between the origins of the two coordinate systems are the 'An' constants.

The equations for converting a coordinate in the 'prime' coordinate system to a coordinate in the 'non-prime' coordinate system are on the left-hand side. The equations for converting a coordinate in the 'non-prime' coordinate system to a coordinate in the 'prime' coordinate system are on the right-hand side.

$$\begin{aligned}x &= a_1 + a_{11} * x' + a_{12} * y' + a_{13} * z' \quad | \quad x' = a_{11} * (x - a_1) + a_{21} \\& * (y - a_2) + a_{31} * (z - a_3) \\y &= a_2 + a_{21} * x' + a_{22} * y' + a_{23} * z' \quad | \quad y' = a_{12} * (x - a_1) + a_{22} \\& * (y - a_2) + a_{32} * (z - a_3) \\z &= a_3 + a_{31} * x' + a_{32} * y' + a_{33} * z' \quad | \quad z' = a_{13} * (x - a_1) + a_{23} \\& * (y - a_2) + a_{33} * (z - a_3)\end{aligned}$$

In both sets of equations a_1 , a_2 , and a_3 are the coordinates (x , y , and z) of the origin of the 'prime' system with respect to the

Sunrise and Sunset Calculation

'non-prime' system. That is, the origin point {0, 0, 0} of the 'prime' system is at location {a1, a2, a3} in the 'non-prime' system.

In both sets of equations the 'Amn' direction cosines refer to the cosine of the angle between axis 'm' of the 'non-prime' system and axis 'n' of the 'prime' system (where m=1 is the X-axis of the 'non-prime' system, m=2 is the Y-axis of the 'non-prime' system, and m=3 is the Z-axis of the 'non-prime' system; where n=1 is the X-axis of the 'prime' system, n=2 is the Y-axis of the 'prime' system, and n=3 is the Z-axis of the 'prime' system). Notice that because of the symmetry of the cosine function about the zero angle it makes no difference whether the angle is rotated through according to a right-handed orientation or a left-handed orientation since $\cos(\theta) = \cos(360 - \theta)$. However, each of the nine angles rotated through must all be determined by choosing representative vectors of the axes of each coordinate system that have the same orientation to those axes with regard to positive (or negative) direction.

Since we are dealing with vectors and not actual coordinates there is no translation of the origin of our two systems to be considered. a1, a2, and a3 are all zero. We will have coordinates in the secondary system that need to be converted to the primary system and the equations on the left will be used.

*/

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "VCTRSUPPORT.h"
//SOME USEFUL CONSTANTS
#define PI 3.141592653589793
#define SIDEREAL_DAY (23*3600.0+56*60.0+4) //Number of seconds
(86164) in a sidereal day (one earth rotation)
#define SUN_RISE_SET_ANGLE 89.167 //Angle between
'sun_earth' & 'earth_observer' at sunrise/sunset
#define SEMI_MAJOR_AXIS 92955807.0 //Semi-major axis of
earth's orbit ellipse
#define SECONDS_PER_ORBIT (365*86400+6*3600+13*60+53) //Number of seconds in
a true orbit (anomalous) of the sun
#define ECCENTRICITY 0.0167 //Earth's elliptical
orbit eccentricity
#define AXIS_TILT 23.45 //Number of degrees of
tilt in the earth's axis
double obs_lat=40.267; //Observer's latitude (positive is northern
hemisphere, negative is southern hemisphere)
double obs_lon=-80.117; //Observer's longitude (positive is east of
Greenwich, negative is west of Greenwich)
//double obs_lat=0.0; //Observer's latitude (positive is northern
hemisphere, negative is southern hemisphere)
//double obs_lon=0.0; //Observer's longitude (positive is east of
Greenwich, negative is west of Greenwich)
typedef struct {
```

Sunrise and Sunset Calculation

```
int yr; //2008, 2009, 2010, etc
int mo; //1=JAN, 2=FEB, etc
int da; //1-31
int hh; //0-23
int mm; //0-59
int ss; //0-59
int tzone; //0=GMT -5=EST, etc
} datim;
// YYYY MM DD HH MM SS Z
datim perihelion={2008, 1, 3, 0, 0, 0,0}; //Date/time (GMT) of perihelion
(earth closest to the sun, our t=0)
datim s_solstice={2008, 6,20,23,59, 0,0}; //Date/time (GMT) of summer
solstice (tilt_z leans directly towards the sun)
datim zero_eot={2008, 6,14,12, 0, 0,0}; //Date/time (GMT) of one of four
times when Equation of Time equals 0
//Other interesting, but unneeded, dates
//datim sp_equinox={2008, 3,20, 5,48, 0,0}; //Date/time (GMT) of spring
equinox (sun_earth perpendicular to tilt_z)
//datim aphelion ={2008, 7, 4, 8, 0, 0,0}; //Date/time (GMT) of aphelion
(earth farthest from sun)
//datim fa_equinox={2008, 9,22,15,44, 0,0}; //Date/time (GMT) of fall
equinox (sun_earth perpendicular to tilt_z again)
//datim w_solstice={2008,12,21,12, 4, 0,0}; //Date/time (GMT) of winter
solstice (tilt_z leans directly away from sun)
//The three vectors for the secondary coord system (described in primary
coord system components)
VECTOR tilt_z; //Z-axis of secondary coord syst def'd in *primary
coords* (earth's rotation axis)
VECTOR tilt_x; //X-axis of secondary coord syst def'd in *primary
coords* (prime meridian at t=0)
VECTOR tilt_y; //Y-axis of secondary coord syst def'd in *primary
coords* (90 degrees east longitude)
//Three unit vectors described in the primary coord system used for
computing direction cosines
VECTOR unit_x={1,0,0};
VECTOR unit_y={0,1,0};
VECTOR unit_z={0,0,1};
//The nine direction cosines
double a11,a12,a13,a21,a22,a23,a31,a32,a33;
int days_in_month[12]={31,28,31,30, 31, 30, 31, 31, 30, 31, 30, 31};
int cum_days[12] ={ 0,31,59,90,120,151,181,212,242,273,303,334};

//-----
-----

//Function prototypes
int date_to_time (datim *d,long *t);
int adjust_date (datim *dt,long time);
int calc_angle (datim *dt,double *angle);
int calc_sun_earth_vector (datim *dt,VECTOR *se);
int print_vec (VECTOR *v); //Debug support
int adjust_dst (datim *dt);
```

Sunrise and Sunset Calculation

```
//-----  
-----  
int main() {  
int it; //Time 't' as an integer  
double t; //Time in seconds from initial position of earth  
VECTOR sun_earth; //Vector from sun to earth's center  
VECTOR normal; //Computed vector normal to orbital plane through the  
sun  
datim current_date; //Work area for constructing dates to be analyzed  
double angle;  
long delta_t;  
double theta;  
int dst_flag; //DST return flag used to annotate printed output  
/*  
OK, let's get started!  
  
First, calculate the constant 'tilt_z' vector (the earth's axis of  
rotation) in the primary coordinate system  
based on the fact that at summer solstice the axis of the earth is, at  
that moment, tilting *directly* towards  
the sun. Another way to put it is that a normal vector to the orbital  
plane passing through the sun (i.e. the  
Z axis of our primary coordinate system) will intersect the earth's  
extended rotation axis at some point on that  
Z axis in the positive Z direction. Thus, a right triangle is formed by  
the earth, the sun, and that point of  
intersection on the positive Z axis. (This condition also happens at  
winter solstice, but then the intersection  
is on the negative Z axis.) The 'sun_earth' vector at time of  
summer_solstice is the short leg of this right  
triangle. The normal vector to the earth's orbital plane and passing  
through the sun (our Z axis) is the long  
leg of this right triangle. The hypotenuse is the 'tilt_z' vector for the  
earth. The tangent of 23.45 degrees  
(0.4337751161) is the ratio of the 'sun_earth' vector's length to the  
length of the normal vector. So the length  
of the normal vector is computed by dividing the length of the  
'sun_earth' vector by the tangent of 23.45 degrees.  
*/  
if (calc_sun_earth_vector(&s_solstice,&sun_earth)) { //Get vector  
which is the short leg of the right triangle  
printf("sunset: cannot use summer solstice date/time\n");  
goto bad_exit;  
}  
//OK, now have 'sun_earth' vector at summer solstice. Solve for normal  
vector's length from right triangle relationship.  
normal.x=0.0; //Normal vector  
has no X component  
normal.y=0.0; //Normal vector  
has no Y component  
normal.z=vector_length(&sun_earth)/tan(Axis_TILT*PI/180); //Z length is
```

Sunrise and Sunset Calculation

```
'sun_earth' length divided by tan(23.45)
vector_sub(&normal,&sun_earth,&tilt_z); //'tilt_z'
vector is the 'normal' vector minus the 'sun_earth' vector
/*
```

OK, now have 'tilt_z' vector described in primary coordinates. Note that it is of an arbitrary length that goes from the center of the earth to that point of intersection of the extended earth rotation axis with the Z axis. But its length is not important. Its direction is what we wanted to ascertain. This direction is constant regardless of where the earth is in its orbit.

We now need the 'tilt_x' and 'tilt_y' vectors to complete our definition of the secondary coordinate system. These will be defined for one specific time in the earth's orbit. Choosing a time when the Equation of Time is zero will allow us to synchronize clock time (GMT) with the sun's true position. We will chose June 14th as this time.

```
*/
if (calc_sun_earth_vector(&zero_eot,&sun_earth)) { //Get
'sun_earth' vector for a time when Equation of Time is zero
printf("sunset: cannot use Equation of Time=0 date/time\n");
goto bad_exit;
}
cross_product(&sun_earth,&tilt_z,&tilt_y); //Cross product
gives us the 'tilt_y' secondary coordinate system axis
cross_product(&tilt_y,&tilt_z,&tilt_x); //Cross product
gives us the 'tilt_x' secondary coordinate system axis
//Compute the nine direction cosines we'll need later for the secondary to
primary coordinate system conversion
a11=dot_product(&unit_x,&tilt_x)/(vector_length(&unit_x)*vector_length(&tilt_x));
a12=dot_product(&unit_x,&tilt_y)/(vector_length(&unit_x)*vector_length(&tilt_y));
a13=dot_product(&unit_x,&tilt_z)/(vector_length(&unit_x)*vector_length(&tilt_z));
a21=dot_product(&unit_y,&tilt_x)/(vector_length(&unit_y)*vector_length(&tilt_x));
a22=dot_product(&unit_y,&tilt_y)/(vector_length(&unit_y)*vector_length(&tilt_y));
a23=dot_product(&unit_y,&tilt_z)/(vector_length(&unit_y)*vector_length(&tilt_z));
a31=dot_product(&unit_z,&tilt_x)/(vector_length(&unit_z)*vector_length(&tilt_x));
a32=dot_product(&unit_z,&tilt_y)/(vector_length(&unit_z)*vector_length(&tilt_y));
a33=dot_product(&unit_z,&tilt_z)/(vector_length(&unit_z)*vector_length(&tilt_z));
/*
```

OK, now test all our logic by choosing some date/time values and then calculating the 'sun_earth' vector, the 'earth_observer' vector, the 'sun_observer' vector, and the angle between the 'sun_observer' and 'earth_observer' vectors. When the angle is close to 90 degrees then we have a sunrise or sunset.

Given a date/time determine the time of sunrise and sunset for that date. Do this by iterating times during the date and check that the angle of the sun is close enough to SUN_RISE_SET_ANGLE.

Sunrise and Sunset Calculation

```
*/
printf("For observer at latitude %7.3f, longitude %8.3f, time_zone
%d\n\n",obs_lat,obs_lon,-5);
for (current_date.yr=2008;current_date.yr<=2010;current_date.yr++)
{ //Do several years, if wanted
printf("YEAR %d\n\n",current_date.yr);
printf(" JAN FEB MAR APR MAY
JUN JUL AUG SEP OCT NOV DEC\n");
printf(" -----
-----\n");

for (current_date.da=1;current_date.da<=31;current_date.da++)
{ //Consider 31 days in each month
printf("%2d",current_date.da);
//Print line starts with the day #
for (current_date.mo=1;current_date.mo<=12;current_date.mo++)
{ //Consider 12 months in the year
if (current_date.da>(days_in_month[current_date.mo-1]
//If day-of-month beyond max for the month
+((!(current_date.yr%4) && current_date.mo==2) ? 1 : 0)))
{ //Account for Feb in leap year
printf(" ");
//Issue blanks for non-existent date
continue;
//But keep going
}
current_date.hh=12; //Start iterative
time search at noon – sun should be definitely up!
current_date.mm=0;
current_date.ss=0;
current_date.tzone=-5; //I'm interested in
Eastern Standard Time (GMT minus 5 hrs)
//First, work earlier in the day looking for sunrise
if (calc_angle(&current_date,&angle)) { //Get the angle for
noon
printf("bad return from 'calc_angle'\n");
goto bad_exit;
}
if (angle<SUN_RISE_SET_ANGLE) { //If sun is below
the horizon at noon (!)
printf("error: sun below horizon at noon!\n");
goto bad_exit;
}
for (delta_t=-3600;abs(delta_t)>0;) { //Iterate to earlier
time starting with one hour movement
adjust_date(&current_date,delta_t); //Use 'delta_t' to
get another date/time to look at
if (calc_angle(&current_date,&angle)) { //Get the angle for
this new time
printf("bad return from 'calc_angle'\n");
goto bad_exit;
}
}
```

Sunrise and Sunset Calculation

```
// printf("trying %02d:%02d:%02d gives %.3f
degrees\n",current_date.hh,current_date.mm,current_date.ss,angle);
if (angle<SUN_RISE_SET_ANGLE) { //If we're earlier
than sunrise
if (delta_t<0) //If we were moving
toward earlier times
delta_t/=-2; //Halve the time
increment and reverse its direction
continue; //Stay in the
iteration
}
if (angle>SUN_RISE_SET_ANGLE) { //If we're later
than sunrise
if (delta_t>0) //If we were moving
toward later times
delta_t/=-2; //Halve the time
increment and reverse its direction
continue; //Stay in the
iteration
}
break; //If we're right
(exactly) at sunrise, then stop the iteration
}
dst_flag=adjust_dst(&current_date); //Adjust hour for
Daylight Savings Time just before printout
printf(" %c%02d%02d ",dst_flag ? '!': '
',current_date.hh,current_date.mm);
//Second, work later in the day looking for sunset
current_date.hh=12; //Reset starting
date/time to noon again
current_date.mm=0;
current_date.ss=0;
for (delta_t=3600;abs(delta_t)>0;) { //Iterate to later
time starting with one hour movement
adjust_date(&current_date,delta_t); //Use 'delta_t' to
get another date/time
if (calc_angle(&current_date,&angle)) { //Get the angle for
this new time
printf("bad return from 'calc_angle\n");
goto bad_exit;
}
// printf("trying %02d:%02d:%02d gives %.3f
degrees\n",current_date.hh,current_date.mm,current_date.ss,angle);
if (angle>SUN_RISE_SET_ANGLE) { //If we're earlier
than sunset
if (delta_t<0) //If we were moving
toward earlier times
delta_t/=-2; //Halve the time
increment and reverse its direction
continue; //Stay in the
iteration
```

Sunrise and Sunset Calculation

```
}
if (angle<SUN_RISE_SET_ANGLE) { //If we're later
than sunset
if (delta_t>0) //If we were moving
toward later times
delta_t/=-2; //Halve the time
increment and reverse its direction
continue; //Stay in the
iteration
}
break; //If we're right
(exactly) at sunset, then stop the iteration
}
dst_flag=adjust_dst(&current_date); //Adjust hour for
Daylight Savings Time just before printout
printf("%02d%02d",current_date.hh,current_date.mm);
}
printf("\n");
}
printf("\n");
}
return(0);
bad_exit:
return(1);
}
//-----
-----

/*
Convert a date and time-of-day value to the time elapsed since perihelion
(t=0)
*/
int date_to_time(datim *d,long *t) {
int max_days;
long cum_d,cum_sec_perihelion,cum_sec_input,cum_sec_intervening,y;
if (d->yr < 1901 || d->yr > 2099) goto bad_datim;
//Century years not leap unless divisible by 400
if (d->mo < 1 || d->mo > 12) goto bad_datim;
//Month in range?
max_days=days_in_month[d->mo-1]+(!((d->yr%4) && d->mo==2) ? 1 : 0);
//Account for Feb in leap year
if (d->da < 1 || d->da > max_days) goto bad_datim; //Day
in range?
if (d->hh < 0 || d->hh > 23) goto bad_datim;
//Hour in range?
if (d->mm < 0 || d->mm > 59) goto bad_datim;
//Minute in range?
if (d->ss < 0 || d->ss > 59) goto bad_datim;
//Second in range?
if (d->tzone < -12 || d->tzone > 11) goto bad_datim;
//Time zone in range?
//Calc # seconds perihelion date/time is into its own year
```

Sunrise and Sunset Calculation

```

cum_d=cum_days[perihelion.mo-1]
//Start with # days in prior months
+((!(perihelion.yr%4) && perihelion.mo>2) ? 1 : 0); //Add
1 if beyond Feb in leap yr
cum_d+=perihelion.da-1; //Add
# days into month
cum_sec_perihelion=cum_d*86400 //Cum
secs total starts w/secs in prior days
+(perihelion.hh-perihelion.tzone)*3600
//Plus secs in prior hrs of day (adj for tzone)
+perihelion.mm*60
//Plus secs in prior mins
+perihelion.ss;
//Plus secs into current min
//Calc # seconds input date/time is into its own year
cum_d=cum_days[d->mo-1]
//Start with # days in prior months
+((!(d->yr%4) && d->mo>2) ? 1 : 0); //Add
1 if beyond Feb in leap yr
cum_d+=d->da-1; //Add
# days into month
cum_sec_input=cum_d*86400 //Cum
secs total starts w/secs in prior days
+(d->hh-d->tzone)*3600
//Plus secs in prior hrs of day (adj for tzone)
+d->mm*60
//Plus secs in prior mins
+d->ss;
//Plus secs into current min
//Calc # seconds in intervening years starting at beginning of perihelion
year
for (cum_sec_intervening=0,y=perihelion.yr;y<d->yr;y++) //For
all intervening years
cum_sec_intervening+=(365*86400)+(!(y%4) ? 86400 : 0); //Add
# secs in year, accounting for leap
*t=cum_sec_intervening-cum_sec_perihelion+cum_sec_input;
//Subt perihelion start fract, add input start fract
return(0);
bad_datim:
printf("date_to_time: invalid datim structure (%d/%d/%d %d:%d:%d
tZone:%d\n)\n",
d->yr,d->mo,d->da,d->hh,d->mm,d->ss,d->tzone);
return(1);
}
//-----
-----

int adjust_date(datim *dt,long t_sec) {
/*
Take a date/time in 'dt' and adjust it forward (t_sec positive) or
backward (t_sec negative) and
construct a new value for 'dt'. Do not move beyond midnight if moving

```

Sunrise and Sunset Calculation

backward or 23:59:59 if moving forward. Return 0 as a normal status. Return 1 if a limit for the movement was reached, but otherwise set the value to the limit.

```
*/
long adj_hh,adj_mm,adj_ss;
adj_hh=t_sec/3600;
adj_mm=(t_sec-adj_hh*3600)/60;
adj_ss=t_sec-adj_hh*3600-adj_mm*60;
dt->ss+=adj_ss;
dt->mm+=adj_mm;
dt->hh+=adj_hh;
if (dt->ss > 59) {
dt->ss-=60;
dt->mm++;
}
if (dt->ss < 0) {
dt->ss+=60;
dt->mm--;
}
if (dt->mm > 59) {
dt->mm-=60;
dt->hh++;
}
if (dt->mm < 0) {
dt->mm+=60;
dt->hh--;
}
if (dt->hh > 23) {
dt->hh=23;
dt->mm=59;
dt->ss=59;
goto bad_exit;
}
if (dt->hh < 0) {
dt->hh=0;
dt->mm=0;
dt->ss=0;
goto bad_exit;
}
return(0);
bad_exit:
return(1);
}
//-----
-----

int calc_angle(datim *dt,double *angle) {
/*
Given a date/time in 'dt' and using the observer's position ('obs_lat'
and 'obs_lon') compute the
angle between the 'earth_observer' vector and the 'sun_observer' vector.
```

Sunrise and Sunset Calculation

Return its value in degrees.

Return 0 if no error. Return 1 if an error is found in the input date/time 'dt'.

*/

long it;

double t1,t2,lat,lon,time_diff,dp;

VECTOR sun_earth; //Vector from sun to earth's center

VECTOR earth_observer; //Vector from earth's center to the observer at given latitude, longitude, and time

VECTOR obs; //Vector from earth's center to observer's point (after time rotation) in *secondary coords*

VECTOR sun_observer; //Vector from sun to the observer's tangent point

//First, calculate the 'sun_earth' vector for this particular time 'dt'

if (calc_sun_earth_vector(dt,&sun_earth)) { //Compute the 'sun_earth' vector for the date/time in question

printf("calc_angle: unable to calculate sun_earth vector\n");

goto bad_exit;

}

//Second, find the time difference between 'dt' and the time 'zero_eot' when the secondary coordinate system was established

if (date_to_time(dt,&it)) { //Convert 'dt' to a time 'it' in seconds since perihelion

printf("calc_angle: cannot use input date/time\n");

goto bad_exit;

}

t1=it; //Convert time to

float

if (date_to_time(&zero_eot,&it)) { //Convert

'zero_eot' to a time 'it' in seconds since perihelion

printf("calc_angle: cannot use input date/time\n");

goto bad_exit;

}

t2=it; //Convert time to

float

time_diff=t1-t2; //Calculate

difference

//Third, set up a "rotated" vector in the secondary coordinate system by adjusting longitude for earth rotation relative to 'zero_eot' time

lon=obs_lon+time_diff/SIDEREAL_DAY*360.0; //Compute longitude with rotation due to 'time_diff' done

lat=obs_lat; //Latitude always

stays the same

obs.z= 4000.0*sin(lat*PI/180.0); //Observer's vector from earth's center (secondary coords)

obs.y=(4000.0*cos(lat*PI/180.0))*sin(lon*PI/180.0); // "

obs.x=(4000.0*cos(lat*PI/180.0))*cos(lon*PI/180.0); // "

//Fourth, change vector in secondary coordinate system to a vector in the primary coordinate system

earth_observer.x=a11*obs.x+a12*obs.y+a13*obs.z; //Now transform to primary system

earth_observer.y=a21*obs.x+a22*obs.y+a23*obs.z; // "

Sunrise and Sunset Calculation

```
earth_observer.z=a31*obs.x+a32*obs.y+a33*obs.z; // "  
//Fifth, compute vector from sun to observer as sum of vector from sun to  
earth plus observer's vector from earth's center  
vector_add(&sun_earth,&earth_observer,&sun_observer); //sun_observer' =  
'sun_earth' + 'earth_observer'  
//Sixth, compute the angle between 'earth_observer' and 'sun_observer'  
vectors by utilizing dot product  
dp=dot_product(&sun_observer,&earth_observer); //Get the dot  
product of the two vectors  
*angle=acos(dp/(vector_length(&sun_observer)*vector_length(&earth_observer)));  
//Angle is arccos of dot product divided by product of vector lengths  
*angle*=180.0/PI; //Convert radians  
to degrees  
return(0);  
bad_exit:  
return(1);  
}  
//-----  
-----  
int calc_sun_earth_vector(datim *dt,VECTOR *se) {  
/*  
Perform the 4-step procedure devised by Kepler to determine the earth's  
position, in XYZ Cartesian coordinates,  
at a specific time since perihelion.  
*/  
long it; //Time in seconds past perihelion (integer)  
double t; //Time in seconds past perihelion (floating  
point)  
double M; //The "mean" anomaly angle  
double E; //The "eccentric" anomaly angle  
double Theta; //Angle, in radians, of earth in its orbit from  
its initial position at perihelion  
double r; //Polar coordinate 'radius' of earth's position  
int i;  
if (date_to_time(dt,&it)) { //Convert date/time structure to a time  
't' in seconds since perihelion  
printf("calc_sun_earth_vector: cannot use input date/time\n");  
goto bad_exit;  
}  
t=it; //Convert time  
to float  
//Step 1  
M=2*PI*t/SECONDS_PER_ORBIT; //Calculate  
Kepler's "Mean" anomaly 'M'  
//Step 2  
/*  
Calculate 'E' since we now know 'M'. The equation for Step 2 cannot be  
solved via algebra, but a formula involving  
an infinite series will get it accurately enough with about three  
terms...
```

Sunrise and Sunset Calculation

```
E = M + (e - 1/8 * e^3) * sin (M) + 1/2 * e^2 * sin (2 * M) + 3/8 * e^3 *
sin(3 * M) + ...
*/
```

```
E=M //Compute
```

```
Kepler's "Eccentric" anomaly 'E' (approximately)
```

```
+(ECCENTRICITY-1.0/8.0*pow(ECCENTRICITY,3.0))*sin(1.0*M) //using the
first three terms of an infinite series
```

```
+ 1.0/2.0*pow(ECCENTRICITY,2.0) *sin(2.0*M) // "
```

```
+ 3.0/8.0*pow(ECCENTRICITY,3.0) *sin(3.0*M); // "
```

```
//Step 3
```

```
/*
```

Calculate 'Theta' since we now know 'E'. The equation for Step 3 is rearranged to solve for 'Theta'...

```
Theta = arctan ( sqrt ((1 + e) / (1 - e)) * tan ( E / 2 ) ) * 2
```

```
*/
```

```
Theta=atan(sqrt((1.0+ECCENTRICITY)/(1.0-ECCENTRICITY))*tan(E/2.0))*2.0;
```

```
//Compute polar coord angle 'Theta'
```

```
//Step 4
```

```
/*
```

Calculate 'r' since we now know 'Theta'. The equation for Step 4 has been changed to allow use of the semi-major axis instead of the semi-latus rectum...

```
r = a * (1 - e^2) / (1 + e * cos (Theta) )
```

```
since p = a * (1 - e^2)
```

```
*/
```

```
r=SEMI_MAJOR_AXIS*(1.0-pow(ECCENTRICITY,2.0))/(1.0+ECCENTRICITY*cos(Theta));
```

```
//Compute polar coord radius 'r'
```

```
se->x=r*cos(Theta); //Convert polar coordinates to Cartesian
```

```
se->y=r*sin(Theta); // "
```

```
se->z=0.0; //There's never any Z component for earth's center
```

```
return(0);
```

```
bad_exit:
```

```
return(1);
```

```
}
```

```
//-----
```

```
int print_vec(VECTOR *v) {
```

```
printf("%15.1fx %15.1fy %15.1fz",v->x,v->y,v->z);
```

```
return(0);
```

```
}
```

```
//-----
```

```
int adjust_dst(datim *dt) {
```

```
if (dt->yr*10000 + dt->mo*100 + dt->da >=20080309 && //If date is on or
after March 9, 2008, and
```

```
dt->yr*10000 + dt->mo*100 + dt->da < 20081102) { //before November 2,
2009
```

```
dt->hh+=1; //Increment the hour
```

Sunrise and Sunset Calculation

```
by one since on Daylight Savings Time
return(1); //Set return flag to
indicate DST in effect
}
if (dt->yr*10000 + dt->mo*100 + dt->da >=20090308 && //If date is on or
after March 8, 2009, and
dt->yr*10000 + dt->mo*100 + dt->da < 20091101) { //before November 1,
2009
dt->hh+=1; //Increment the hour
by one since on Daylight Savings Time
return(1); //Set return flag to
indicate DST in effect
}
return(0); //Normal return
doesn't adjust for DST and leaves flag unset
}
//-----
-----

// VCTRSUPPORT.h
//=====
=====
//===== 3D VECTOR AND GEOMETRIC PROCESSING ROUTINES
=====
//=====
=====
/*
Minimum angle to call lines parallel or perpendicular is arbitrarily set
to 0.1 degrees. The sine of 0.1 degrees is 0.001745.
*/
#define MIN_SINE 0.001745
#define MIN_COSINE 0.001745
//----- DEFINE GEOMETRIC TYPES (POINT, VECTOR, LINE, PLANE,
TRIANGLE, etc) -----

typedef struct { //A POINT has a single definition and is
defined by...
double x,y,z; //Its 3 orthogonal (x,y,z) coordinates relative
to the origin
} POINT;
/*
The definition of a VECTOR is the same as a POINT since it consists of 3
displacements in the 3 axes. So use the
already established POINT type to allow efficient type casting between
the two when it's appropriate.
*/
typedef POINT VECTOR; //A VECTOR is simply composed of 3 orthogonal
(x,y,z) composite vector lengths

typedef struct { //A LINE's preferred definition is...
POINT lp; //Any arbitrary POINT on the line
```

Sunrise and Sunset Calculation

```
VECTOR lv; //And a non-null direction VECTOR
} LINE;
/*
```

Note: In geometry a line has no inherit "direction" but using our preferred "vector" description will allow us to consider the LINE to be "flowing" in one of two directions and this can be useful for ordering points on the line.

```
*/
typedef struct { //A line may also be defined by...
POINT lp1; //Any two distinct points
POINT lp2; //On the line
} line_2pt;
/*
```

The next two types are described nicely by the same structure that holds a LINE, so use the already established LINE type. This allows efficient type casting to change between any two of the three.

```
*/
typedef LINE RAY; //A RAY is simply a LINE that starts at a POINT
and extends infinitely in direction VECTOR
typedef LINE LINE_SEG; //A LINE SEGMENT is a LINE that starts at a
POINT and extends only as long as a VECTOR
```

```
typedef struct { //A PLANE's preferred definition, in "modified"
Hessian form, is...
```

```
VECTOR pn; //A non-null (but not necessarily unit) VECTOR
normal to the plane
double d; //And d, the distance of the plane (may be zero
or negative) from the origin
} PLANE;
```

```
typedef struct { //A plane may also be defined by...
POINT pp; //Any POINT on the plane
VECTOR pn; //And a non-null (but not necessarily unit)
VECTOR normal to the plane
} plane_pt_1v;
```

```
typedef struct { //A plane may also be defined by...
POINT pp; //Any POINT on the plane
VECTOR pv1,pv2; //And 2 non-null, non-parallel direction
VECTORS
} plane_pt_2v;
```

```
typedef struct { //A plane may also be defined in pure scalar
terms, as...
double A,B,C,D; //The general equation  $Ax+By+Cz+D=0$ 
} plane_coord;
/*
```

Note: This pure scalar definition is not currently in use and violates my desire to use 'vector' logic.

```
*/
```

```
typedef struct { //A TRIANGLE's preferred definition is...
```

Sunrise and Sunset Calculation

```
POINT tp; //A POINT defining one of the corners
VECTOR tv1,tv2; //And 2 non-null, non-parallel VECTORS which
define the other two corners
} TRIANGLE;
typedef struct { //A triangle may also be defined by...
POINT tp1,tp2,tp3; //Three distinct non-colinear POINTs
} triangle_3pt;

//----- FUNCTION PROTOTYPES OF GEOMETRIC OPERATIONS PROVIDED BY
THIS HEADER -----
//
int null_vector (VECTOR *v);
int vector_2_points (POINT *p1, POINT *p2, VECTOR *v);
double distance_2_points (POINT *p1, POINT *p2);
double vector_length (VECTOR *v);
int vector_add (VECTOR *v1, VECTOR *v2, VECTOR *v3);
int vector_sub (VECTOR *v1, VECTOR *v2, VECTOR *v3);
int vector_mpy (VECTOR *v, double scale);
int vector_div (VECTOR *v, double scale);
int vector_resize (VECTOR *v, double vector_len);
double dot_product (VECTOR *v1, VECTOR *v2);
int cross_product (VECTOR *v1, VECTOR *v2, VECTOR *v3);
int plane_3_points (POINT *a, POINT *b, POINT *c, PLANE
*p1);
int intersect_line_plane (LINE *l, PLANE *pl, POINT *p);
int plane_from_pt_1v (POINT *p, VECTOR *v, PLANE *pl);
int plane_from_pt_2v (POINT *p, VECTOR *v1, VECTOR *v2, PLANE
*p1);
int lines_2_closest_points (LINE *l1, LINE *l2, POINT *pa, POINT
*pb);
double distance_2_lines (LINE *l1, LINE *l2, LINE *l3);
int intersect_2_planes (PLANE *p1, PLANE *p2, LINE *l);
int triangle_from_3pt (POINT *p1, POINT *p2, POINT *p3, TRIANGLE
*t);
int line_from_2_points (POINT *p1, POINT *p2, LINE *l);
int vector_rotate (VECTOR *X, VECTOR *U, double theta);
//
//----- FUNCTION CODE FOR GEOMETRIC OPERATIONS PROVIDED BY
THIS HEADER -----
//
int null_vector(VECTOR *v) {
/*
Examine a VECTOR to see if it's null. This is true if all component
vectors are of zero length.
Zero is returned if the given VECTOR is *not* null. One is returned if it
*is* null.
*/
if (v->x==0.0 && v->y==0.0 && v->z==0.0) //If all components are zero
return(1); //Return TRUE
return(0); //Otherwise, return FALSE
}
}
```

Sunrise and Sunset Calculation

```
//-----  
-----  
//int point_from_vector(VECTOR *v,POINT *p) {  
/*  
Apply a vector at the origin to define a point. Supplying the null vector  
is not a problem.  
In that case the origin point is computed. Zero status is always  
returned.  
*/  
/*  
p->x=v->x;  
p->y=v->y;  
p->z=v->z;  
return(0);  
}  
*/  
//-----  
-----  
//int vector_1_point(POINT *p,VECTOR *v) {  
/*  
A VECTOR is constructed using the displacement of a POINT from the  
origin. If the point is the origin  
itself a null vector will result, but this is not a problem. Zero is  
always returned.  
*/  
/*  
v->x=p->x;  
v->y=p->y;  
v->z=p->z;  
return(0);  
}  
*/  
//-----  
-----  
int vector_2_points(POINT *p1,POINT *p2,VECTOR *v) {  
/*  
A VECTOR is constructed from two POINTs. If the two points are the same  
then a null vector  
will result, but this is not a problem. Zero is always returned.  
*/  
/*  
v->x=p2->x-p1->x;  
v->y=p2->y-p1->y;  
v->z=p2->z-p1->z;  
return(0);  
}  
//-----  
-----  
double distance_2_points(POINT *p1,POINT *p2) {  
/*  
The distance between two points is calculated via the Pythagorean  
theorem. If the points are
```

Sunrise and Sunset Calculation

identical the value 0.0 will be returned so no check is needed.

```
*/
return sqrt((p2->x - p1->x) * (p2->x - p1->x) +
(p2->y - p1->y) * (p2->y - p1->y) +
(p2->z - p1->z) * (p2->z - p1->z));
}
//-----
-----
double vector_length(VECTOR *v) {
/*
The length of a VECTOR is returned. No check for a null vector is needed.
*/
return sqrt(v->x * v->x + v->y * v->y + v->z * v->z);
}
//-----
-----
int vector_add(VECTOR *v1,VECTOR *v2,VECTOR *v3) {
/*
Add two VECTORS v1 and v2, giving a third VECTOR v3. No check for null
vectors is needed.
Zero is always returned.
*/
v3->x=v1->x+v2->x;
v3->y=v1->y+v2->y;
v3->z=v1->z+v2->z;
return(0);
}
//-----
-----
int vector_sub(VECTOR *v1,VECTOR *v2,VECTOR *v3) {
/*
Subtract VECTOR v2 from v1, giving a third VECTOR v3. No check for null
vectors is needed.
Zero is always returned.
*/
v3->x=v1->x-v2->x;
v3->y=v1->y-v2->y;
v3->z=v1->z-v2->z;
return(0);
}
//-----
-----
int vector_mpy(VECTOR *v,double scale) {
/*
Multiply the lengths of each of the three components of a VECTOR 'v' by
'scale'. The vector
'v' is modified as a result. 'scale' may be 0.0 resulting in 'v' becoming
the null vector.
This is not a problem. Also, the original vector 'v' may be the null
vector and this is also
not a problem. Zero is always returned.
*/
}
```

Sunrise and Sunset Calculation

```
*/
v->x*=scale;
v->y*=scale;
v->z*=scale;
return(0);
}
//-----
-----
int vector_div(VECTOR *v,double scale) {
/*
Divide the lengths of each of the three components of a VECTOR 'v' by
'scale'. The vector
'v' is modified as a result. Zero status is returned unless 'scale' is
0.0. In that case
1 is returned and 'v' is unchanged. The original vector 'v' may be the
null vector and this
is not a problem since it will stay the null vector regardless of the
value of 'scale'.
*/
if (scale==0.0) //Check for invalid divisor
return(1);
v->x/=scale;
v->y/=scale;
v->z/=scale;
return(0);
}
//-----
-----
int vector_resize(VECTOR *v,double vector_len) {
/*
Resize the VECTOR 'v' so its length is 'vector_len'. The vector 'v' is
modified as a result.
'vector_len' may be 0.0 resulting in 'v' becoming the null vector. This
is not a problem.
If 'vector_len' is less than zero then the direction of 'v' will be
reversed because the
sign of each of its component vectors will be reversed. In this case the
length of 'v' will
be set to the absolute value of 'vector_len'. Again, this is not a
problem.
*/
double factor;
/* if (vector_len<=0.0) return(0); Old logic forbidding 'vector_len' to be
negative */
factor=vector_length(v); //Get
original length of 'v'
if (factor==0.0)
{ //If it's zero
if (vector_len==0.0) { //But caller actually wants
final length zero
v->x=0.0; v->y=0.0; v->z=0.0; //Then make sure they really
```

Sunrise and Sunset Calculation

```
are absolute zero
}
else { //Otherwise, caller wants a non-zero final length, but
that's impossible
return(1);
//So return failure
}
}
else { //Otherwise, original
length was not zero
factor=vector_len/factor; //So calc scale factor (may be zero if
'vector_len' is zero)
v->x*=factor; v->y*=factor; v->z*=factor; //And apply it to
all the components
}
return(0);
//Return success
}
//-----
-----
double dot_product(VECTOR *v1,VECTOR *v2) {
/*
The dot product of the two vectors v1 and v2 is returned. If the two
vectors are perpendicular return 0.0.
*/
double temp,dp;
if (v1->x==0.0 && v1->y==0.0 && v1->z==0.0 || //If either
v2->x==0.0 && v2->y==0.0 && v2->z==0.0) { //vector is NULL
return(0.0); //Pretend they are
perpendicular and return 0.0
}
dp=v1->x * v2->x + v1->y * v2->y + v1->z * v2->z; //Compute dot product
if (dp==0.0) { //If dot product has
gone to zero
return(0.0); //Vectors must be
perpendicular so return 0.0
}
/*
THIS OPTIONAL CODE IMPLEMENTS THE "ESSENTIALLY PERPENDICULAR" LOGIC IF
YOU WANT IT
```

Since the magnitude of the dot product of two vectors is the product of their magnitudes times the cosine of the angle between them I can solve for the cosine of the angle between them by dividing the already computed dot product's magnitude by the product of their individual magnitudes. If this cosine is close to zero then the angle must be close to 90 degrees. If so, the vectors are 'essentially' perpendicular.

```
temp=vect_determinant is, so must still call the planes parallel. */
if (determinant==0.0)
return(1);
```

Sunrise and Sunset Calculation

```

c1=(d1*dot_product(v2,v2)-d2*dot_product(v1,v2))/determinant;
c2=(d2*dot_product(v1,v1)-d1*dot_product(v1,v2))/determinant;
l->lp.x=c1*v1->x+c2*v2->x;
l->lp.y=c1*v1->y+c2*v2->y;
l->lp.z=c1*v1->z+c2*v2->z;
return(0);
}
//-----
-----
int triangle_from_3pt(POINT *p1,POINT *p2,POINT *p3,TRIANGLE *t) {
VECTOR temp;
t->tp.x = p1->x;
t->tp.y = p1->y;
t->tp.z = p1->z;
t->tv1.x = p2->x - p1->x;
t->tv1.y = p2->y - p1->y;
t->tv1.z = p2->z - p1->z;
t->tv2.x = p3->x - p1->x;
t->tv2.y = p3->y - p1->y;
t->tv2.z = p3->z - p1->z;
if (cross_product(&t->tv1,&t->tv2,&temp)==1) //If vector from p1->p2
parallel to vector from p1->p3
return(1); //Return error, points are colinear
or two points are the same
else //Otherwise
return(0); //Normal return
}
//-----
-----
/*
Return the preferred 'point-vector' definition of a line given two points
on the line. This routine
always returns POINT p1 as the LINE's defining 'point' and the vector
going from POINT p1 to POINT p2
as the defining 'vector'. A check is made, however, to ensure that both
p1 and p2 are not the same POINT.
If they are an error status is returned but the LINE parameters are still
stored. Otherwise, zero is returned.
*/
int line_from_2_points(POINT *p1,POINT *p2,LINE *l) {
l->lp=*p1; //LINE 'l's point is
POINT 'p1'
vector_sub(p2,p1,&l->lv); //LINE 'l's vector is
VECTOR to POINT p2 minus VECTOR to POINT p1
// vector_sub(&((VECTOR)(*p2)),&((VECTOR)(*p1)),&l->lv); //Using type
casting to force "pointer to VECTOR" gives compile error
if (null_vector(&l->lv)) //If the vector is
null, points were the same
return(1); //So return bad status
else //Otherwise, points
were not the same

```

Sunrise and Sunset Calculation

```
return(0); //Return normal status
}
//-----
-----
/*
Rotate the VECTOR X about the VECTOR U through an angle of 'theta'
radians. Note that if X and U
are pointing in the same direction their cross product returns the null
vector. This is not a problem
because it simply causes X to be returned unchanged as it ought to.
```

This code follows the Rodrigues formula.

```
*/
int vector_rotate(VECTOR *X,VECTOR *U,double theta) {
VECTOR temp_u,temp_x,temp_v1,temp_v2;
double temp;
temp_u=*U; //Make a copy of U so we can
resize it
vector_resize(&temp_u,1.0); //Make it a unit vector for
the following calculations
temp_x=*X; //Make a copy of X to work
with
vector_mpy(&temp_x,cos(theta)); //Now have X*cos(theta)
[in temp_x]
temp=dot_product(&temp_u,&temp_x)*(1.0-cos(theta)); //Now have
(U.X)(1-cos(theta)) [in temp]
temp_v1=temp_u; //Prepare to multiply U by
result of previous line
vector_mpy(&temp_v1,temp); //Now have
U*(U.X)(1-cos(theta)) [in temp_v1]
```