Revised January 2008 • NREL/TP-560-34302

Technical Report

Solar Position Algorithm for Solar Radiation Applications

Ibrahim Reda and Afshin Andreas

National Renewable Energy Laboratory

1617 Cole Boulevard
Golden, Colorado 80401-3393

NREL is a U.S. Department of Energy Laboratory
Operated by Midwest Research Institute • Battelle • Bechtel

Contract No. DE-AC36-99-GO10337

Revised January 2008 • NREL/TP-560-34302

Solar Position Algorithm for Solar Radiation Applications

Ibrahim Reda and Afshin Andreas

Prepared under Task No. WU1D5600

National Renewable Energy Laboratory

1617 Cole Boulevard
Golden, Colorado 80401-3393

NREL is a U.S. Department of Energy Laboratory
Operated by Midwest Research Institute • Battelle • Bechtel

Contract No. DE-AC36-99-GO10337

Acknowledgment

We thank Bev Kay for all her support by manually typing all the data tables in the report into text files, which made it easy and timely to transport to the report text and all of our software code. We also thank Daryl Myers for all his technical expertise in solar radiation applications.

Table of Contents

List of Figures

Appendix

List of Appendix Figures

List of Appendix Tables

Abstract

There have been many published articles describing solar position algorithms for solar radiation applications. The best uncertainty achieved in most of these articles is greater than ±0.01˚ in calculating the solar zenith and azimuth angles. For some, the algorithm is valid for a limited number of years varying from 15 years to a hundred years. This report is a step by step procedure for implementing an algorithm to calculate the solar zenith and azimuth angles in the period from the year -2000 to 6000, with uncertainties of ±0.0003˚. The algorithm is described by Jean Meeus [3]. This report is written in a step by step format to simplify the complicated steps described in the book, with a focus on the sun instead of the planets and stars in general. It also introduces some changes to accommodate for solar radiation applications. The changes include changing the direction of measuring azimuth angles to be measured from north and eastward instead of being measured from south and eastward, and the direction of measuring the observer's geographical longitude to be measured as positive eastward from Greenwich meridian instead of negative. This report also includes the calculation of incidence angle for a surface that is tilted to any horizontal and vertical angle, as described by Iqbal [4].

1. Introduction

With the continuous technological advancements in solar radiation applications, there will always be a demand for smaller uncertainty in calculating the solar position. Many methods to calculate the solar position have been published in the solar radiation literature, nevertheless, their uncertainties have been greater than ± 0.01˚ in solar zenith and azimuth angle calculations, and some are only valid for a specific number of years[1]. For example, Michalsky's calculations are limited to the period from 1950 to 2050 with uncertainty of greater than ± 0.01˚ [2], and the calculations of Blanco-Muriel et al.'s are limited to the period from 1999 to 2015 with uncertainty greater than > ± 0.01˚ [1].

An example emphasizing the importance of reducing the uncertainty of calculating the solar position to lower than ± 0.01˚, is the calibration of pyranometers that measure the global solar irradiance. During the calibration, the responsivity of the pyranometer is calculated at zenith angles from 0˚ to 90˚ by dividing its output voltage by the reference global solar irradiance (G), which is a function of the cosine of the zenith angle (cos θ). Figure 1 shows the magnitude of errors that the 0.01˚ uncertainty in θ can contribute to the calculation of cos θ, and consequently G that is used to calculate the responsivity. Figure 1 shows that the uncertainty in cos θ exponentially increases as θ reaches 90˚(e.g. at θ equal to 87˚, the uncertainty in cos θ is 0.7%, which can result in an uncertainty of 0.35% in calculating G; because at such large zenith angles the normal incidence irradiance is approximately equal to half the value of G). From this arises the need to use a solar position algorithm with lower uncertainty for users that are interested in measuring the global solar irradiance with smaller uncertainties in the full zenith angle range from 0˚ to 90˚.

In this report we describe a procedure for a Solar Position Algorithm (SPA) to calculate the solar zenith and azimuth angle with uncertainties equal to ±0.0003˚ in the period from the year -2000 to 6000. Figure 1 shows that the uncertainty of the reference global solar irradiance, resulting from ±0.0003˚ in calculating the solar zenith angle in the range from 0˚ to 90˚ is negligible. The procedure is adopted from *The Astronomical Algorithms* [3], which is based on the Variations Sèculaires des Orbites Planètaires Theory (VSOP87) that was developed by P. Bretagnon in 1982 then modified in 1987 by Bretagnon and Francou [3]. In this report, we summarize the complex algorithm elements scattered throughout the book to calculate the solar position, and introduce some modification to the algorithm to accommodate solar radiation applications. For example, in *The Astronomical Algorithms* [3], the azimuth angle is measured westward from south, but for solar radiation applications, it is measured eastward from north. Also, the observer's geographical longitude is considered positive west, or negative east from Greenwich, while for solar radiation applications, it is considered negative west, or positive east from Greenwich.

We start this report by:

- Describing the time scales because of the importance of using the correct time in the SPA
- Providing a step by step procedure to calculate the solar position and the solar incidence angle for an arbitrary surface orientation using the methods described in *An Introduction*

to Solar Radiation [4]

- Evaluating the SPA against the *Astronomical Almanac* (AA) data for the years 1994, 1995, 1996, and 2004.

Because of the complexity of the algorithm we included some examples, in the Appendix, to give the users confidence in their step by step calculations. We also included in the Appendix an explanation of how to calculate the equation of time, sun transit (solar noon), sunrise, sunset, and how to change the Julian Day to a Calendar Date. We also included a C source code with header file, for all the calculations in this report (except for the Julian Day to Calendar Date conversion). The users can incorporate this module into their own code by including the header file, declaring the SPA structure, filling in the required input parameters into the structure, and then call the SPA calculation function. This function will calculate all the output values and fill in the SPA structure for the user.

The users should note that this report is used to calculate the solar position for solar radiation applications only, and that it is purely mathematical and not meant to teach astronomy or to describe the Earth rotation. For more description about the astronomical nomenclature that is used through out the report, the user is encouraged to review the definitions in the *Astronomical Almanacs*, or other astronomical reference.

2. Time Scale

The following are the internationally recognized time scales:

- The Universal Time (UT), or Greenwich civil time, is based on the Earth's rotation and counted from 0-hour at midnight; the unit is mean solar day [3]. UT is the time used to calculate the solar position in the described algorithm. It is sometimes referred to as UT1.

- The International Atomic Time (TAI) is the duration of the System International Second (SI-second) and based on a large number of atomic clocks [5].

- The Coordinated Universal Time (UTC) is the bases of most radio time signals and the legal time systems. It is kept to within 0.9 seconds of UT1 (UT) by introducing one second steps to its value (leap second); to date the steps are always positive.

- The Terrestrial Dynamical or Terrestrial Time (TDT or TT) is the time scale of ephemerides for observations from the Earth surface.

The following equations describe the relationship between the above time scales (in seconds):

$$TT = TAI + 32.184 \quad , \tag{1}$$

$$UT = TT - \Delta T \quad , \tag{2}$$

where ΔT is the difference between the Earth rotation time and the Terrestrial Time (TT). It is derived from observation only and reported yearly in the *Astronomical Almanac* [5].

$$UT = UT1 = UTC + \Delta UT1 \quad , \tag{3}$$

where ΔUT1 is a fraction of a second, positive or negative value, that is added to the UTC to adjust for the Earth irregular rotational rate. It is derived from observation, but predicted values are transmitted in code in some time signals, e.g. weekly by the U.S. Naval Observatory (USNO) [6].

3. Procedure

3.1. Calculate the Julian and Julian Ephemeris Day, Century, and Millennium:

The Julian date starts on January 1, in the year - 4712 at 12:00:00 UT. The Julian Day (*JD*) is calculated using UT and the Julian Ephemeris Day (JDE) is calculated using TT. In the following steps, note that there is a 10-day gap between the Julian and Gregorian calendar where the Julian calendar ends on October 4, 1582 (*JD* = 2299160), and after 10-days the Gregorian calendar starts on October 15, 1582.

3.1.1 Calculate the Julian Day (*JD*),

$$JD = INT(365.25*(Y + 4716)) + INT(30.6001*(M + 1)) + D + B - 1524.5 \quad , \tag{4}$$

where,

- INT is the Integer of the calculated terms (e.g. 8.7 = 8, 8.2 = 8, and -8.7 = -8..etc.).
- *Y* is the year (e.g. 2001, 2002, ..etc.).
- *M* is the month of the year (e.g. 1 for January, ..etc.). Note that if $M > 2$, then *Y* and *M* are not changed, but if $M = 1$ or 2, then $Y = Y-1$ and $M = M + 12$.
- *D* is the day of the month with decimal time (e.g. for the second day of the month at 12:30:30 UT, *D* = 2.521180556).
- *B* is equal to 0, for the Julian calendar {i.e. by using $B = 0$ in Equation 4, $JD < 2299160$}, and equal to (2 - *A* + INT (*A*/4)) for the Gregorian calendar {i.e. by using $B = 0$ in Equation 4, $JD > 2299160$}, where $A = INT(Y/100)$.

For users who wish to use their local time instead of UT, change the time zone to a fraction of a day (by dividing it by 24), then subtract the result from *JD*. Note that the fraction is subtracted from *JD* calculated before the test for $B<2299160$ to maintain the Julian and Gregorian periods.

Table A4.1 shows examples to test any implemented program used to calculate the *JD*.

3.1.2. Calculate the Julian Ephemeris Day (*JDE*),

$$JDE = JD + \frac{\Delta T}{86400} \quad . \tag{5}$$

3.1.3. Calculate the Julian century (*JC*) and the Julian Ephemeris Century (*JCE*) for the 2000 standard epoch,

$$JC = \frac{JD - 2451545}{36525} \quad , \tag{6}$$

$$JCE = \frac{JDE - 2451545}{36525} \quad . \tag{7}$$

3.1.4. Calculate the Julian Ephemeris Millennium (*JME*) for the 2000 standard epoch,

$$JME = \frac{JCE}{10} \quad . \tag{8}$$

3.2. Calculate the Earth heliocentric longitude, latitude, and radius vector (*L*, *B*, and *R*):

"Heliocentric" means that the Earth position is calculated with respect to the center of the sun.

3.2.1. For each row of Table A4.2, calculate the term $L0_i$ (in radians),

$$L0_i = A_i * \cos(B_i + C_i * JME) \quad , \tag{9}$$

where,

- i is the ith row for the term *L0* in Table A4.2.
- A_i , B_i , and C_i are the values in the ith row and *A*, *B*, and *C* columns in Table A4.2, for the term *L0* (in radians).

3.2.2. Calculate the term *L0* (in radians),

$$L0 = \sum_{i=0}^{n} L0_i \quad , \tag{10}$$

where n is the number of rows for the term *L0* in Table A4.2.

3.2.3. Calculate the terms *L1*, *L2*, *L3*, *L4*, and *L5* by using Equations 9 and 10 and changing the 0 to 1, 2, 3, 4, and 5, and by using their corresponding values in

columns A, B, and C in Table A4.2 (in radians).

3.2.4. Calculate the Earth heliocentric longitude, *L* (in radians),

$$L = \frac{L0 + L1 * JME + L2 * JME^2 + L3 * JME^3 + L4 * JME^4 + L5 * JME^5}{10^8} \quad . \qquad (11)$$

3.2.5. Calculate *L* in degrees,

$$L\,(in\ Degrees) = \frac{L\,(in\ Radians) * 180}{\pi} \quad , \qquad (12)$$

where π is approximately equal to 3.1415926535898.

3.2.6. Limit *L* to the range from 0° to 360°. That can be accomplished by dividing *L* by 360 and recording the decimal fraction of the division as *F*. If *L* is positive, then the limited *L* = 360 * *F*. If L is negative, then the limited *L* = 360 - 360 * *F*.

3.2.7. Calculate the Earth heliocentric latitude, *B* (in degrees), by using Table A4.2 and steps 3.2.1 through 3.2.5 and by replacing all the *L*s by *B*s in all equations. Note that there are no *B2* through *B5*, consequently, replace them by zero in steps 3.2.3 and 3.2.4.

3.2.8. Calculate the Earth radius vector, *R* (in Astronomical Units, AU), by repeating step 3.2.7 and by replacing all *L*s by *R*s in all equations. Note that there is no *R5*, consequently, replace it by zero in steps 3.2.3 and 3.2.4.

3.3. Calculate the geocentric longitude and latitude (Θ and β):

"Geocentric" means that the sun position is calculated with respect to the Earth center.

3.3.1. Calculate the geocentric longitude, Θ (in degrees),

$$\Theta = L + 180 \quad . \qquad (13)$$

3.3.2. Limit Θ to the range from 0° to 360° as described in step 3.2.6.

3.3.3. Calculate the geocentric latitude, β (in degrees),

$$\beta = -B \quad . \qquad (14)$$

3.4. Calculate the nutation in longitude and obliquity ($\Delta\psi$ and $\Delta\varepsilon$):

3.4.1. Calculate the mean elongation of the moon from the sun, X_0 (in degrees),

$$X_0 = 297.85036 + 445267.111480 * JCE - 0.0019142 * JCE^2 + \frac{JCE^3}{189474} \quad . \tag{15}$$

3.4.2. Calculate the mean anomaly of the sun (Earth), X_1 (in degrees),

$$X_1 = 357.52772 + 35999.050340 * JCE - 0.0001603 * JCE^2 - \frac{JCE^3}{300000} \quad . \tag{16}$$

3.4.3. Calculate the mean anomaly of the moon, X_2 (in degrees),

$$X_2 = 134.96298 + 477198.867398 * JCE + 0.0086972 * JCE^2 + \frac{JCE^3}{56250} \quad . \tag{17}$$

3.4.4. Calculate the moon's argument of latitude, X_3 (in degrees),

$$X_3 = 93.27191 + 483202.017538 * JCE - 0.0036825 * JCE^2 + \frac{JCE^3}{327270} \quad . \tag{18}$$

3.4.5. Calculate the longitude of the ascending node of the moon's mean orbit on the ecliptic, measured from the mean equinox of the date, X_4 (in degrees),

$$X_4 = 125.04452 - 1934.136261 * JCE + 0.0020708 * JCE^2 + \frac{JCE^3}{450000} \quad . \tag{19}$$

3.4.6. For each row in Table A4.3, calculate the terms $\Delta\psi_i$ and $\Delta\varepsilon_i$ (in 0.0001of arc seconds),

$$\Delta\psi_i = (a_i + b_i * JCE) * \sin\left(\sum_{j=0}^{4} X_j * Y_{i,j}\right) \quad , \tag{20}$$

$$\Delta\varepsilon_i = (c_i + d_i * JCE) * \cos\left(\sum_{j=0}^{4} X_j * Y_{i,j}\right) \quad , \tag{21}$$

where,

- a_i , b_i , c_i , and d_i are the values listed in the i[th] row and columns a, b, c, and d in Table A4.3.
- X_j is the j[th] X calculated by using Equations 15 through 19.
- $Y_{i,j}$ is the value listed in i[th] row and j[th] Y column in Table A4.3.

3.4.7. Calculate the nutation in longitude, $\Delta\psi$ (in degrees),

$$\Delta\psi = \frac{\sum_{i=0}^{n} \Delta\psi_i}{36000000} \quad , \tag{22}$$

where n is the number of rows in Table A4.3 (n equals 63 rows in the table).

3.4.8. Calculate the nutation in obliquity, $\Delta\varepsilon$ (in degrees),

$$\Delta\varepsilon = \frac{\sum_{i=0}^{n} \Delta\varepsilon_i}{36000000} \quad . \tag{23}$$

3.5. Calculate the true obliquity of the ecliptic, ε (in degrees):

3.5.1. Calculate the mean obliquity of the ecliptic, ε_0 (in arc seconds),

$$\begin{aligned} \varepsilon_0 = {} & 84381.448 - 4680.93U - 1.55U^2 + 1999.25U^3 - \\ & 51.38U^4 - 249.67U^5 - 39.05U^6 + 7.12U^7 + \\ & 27.87U^8 + 5.79U^9 + 2.45U^{10} \quad , \end{aligned} \tag{24}$$

where $U = JME/10$.

3.5.2. Calculate the true obliquity of the ecliptic, ε (in degrees),

$$\varepsilon = \frac{\varepsilon_0}{3600} + \Delta\varepsilon \quad . \tag{25}$$

3.6. Calculate the aberration correction, $\Delta\tau$ (in degrees):

$$\Delta\tau = -\frac{20.4898}{3600 * R} \quad . \tag{26}$$

3.7. Calculate the apparent sun longitude, λ (in degrees):

$$\lambda = \Theta + \Delta\psi + \Delta\tau \quad . \tag{27}$$

3.8. Calculate the apparent sidereal time at Greenwich at any given time, ν (in degrees):

3.8.1. Calculate the mean sidereal time at Greenwich, ν_0 (in degrees),

$$\nu_0 = 280.46061837 + 360.98564736629 * (JD - 2451545) + 0.000387933 * JC^2 - \frac{JC^3}{38710000} \quad . \tag{28}$$

3.8.2. Limit ν_0 to the range from 0° to 360° as described in step 3.2.6.

3.8.3. Calculate the apparent sidereal time at Greenwich, ν (in degrees),

$$\nu = \nu_0 + \Delta\psi * \cos(\varepsilon) \quad . \tag{29}$$

3.9. Calculate the geocentric sun right ascension, α (in degrees):

3.9.1. Calculate the sun right ascension, α (in radians),

$$\alpha = Arc\tan 2\left(\frac{\sin\lambda * \cos\varepsilon - \tan\beta * \sin\varepsilon}{\cos\lambda}\right) \quad , \tag{30}$$

where Arc*tan*2 is an arctangent function that is applied to the numerator and the denominator (instead of the actual division) to maintain the correct quadrant of the α where α is in the rage from -π to π.

3.9.2. Calculate α in degrees using Equation 12, then limit it to the range from 0° to 360° using the technique described in step 3.2.6.

3.10. Calculate the geocentric sun declination, δ (in degrees):

$$\delta = Arc\sin(\sin\beta * \cos\varepsilon + \cos\beta * \sin\varepsilon * \sin\lambda) \quad , \tag{31}$$

where δ is positive or negative if the sun is north or south of the celestial equator, respectively. Then change δ to degrees using Equation 12.

3.11. Calculate the observer local hour angle, *H* (in degrees):

$$H = \nu + \sigma - \alpha \quad , \tag{32}$$

Where σ is the observer geographical longitude, positive or negative for east or west of Greenwich, respectively.

Limit *H* to the range from 0° to 360° using step 3.2.6 and note that it is measured westward from south in this algorithm.

3.12. Calculate the topocentric sun right ascension α' (in degrees):

"Topocentric" means that the sun position is calculated with respect to the observer local position at the Earth surface.

3.12.1. Calculate the equatorial horizontal parallax of the sun, ξ (in degrees),

$$\xi = \frac{8.794}{3600 * R} \quad , \tag{33}$$

where *R* is calculated in step 3.2.8.

3.12.2. Calculate the term *u* (in radians),

$$u = Arc\tan(0.99664719 * \tan\varphi) \quad , \tag{34}$$

where φ is the observer geographical latitude, positive or negative if north or south of the equator, respectively. Note that the 0.99664719 number equals (1 - *f*), where *f* is the Earth's flattening.

3.12.3. Calculate the term *x*,

$$x = \cos u + \frac{E}{6378140} * \cos\varphi \quad , \tag{35}$$

where *E* is the observer elevation (in meters). Note that x equals ρ * cos φ' where ρ is the observer's distance to the center of Earth, and φ' is the observer's geocentric latitude.

3.12.4. Calculate the term *y*,

$$y = 0.99664719 * \sin u + \frac{E}{6378140} * \sin \varphi \quad , \tag{36}$$

note that *y* equals $\rho * \sin \varphi'$,

3.12.5. Calculate the parallax in the sun right ascension, $\Delta\alpha$ (in degrees),

$$\Delta \alpha = Arc\tan 2\,(\frac{-x * \sin \xi * \sin H}{\cos \delta - x * \sin \xi * \cos H}) \quad . \tag{37}$$

Then change $\Delta\alpha$ to degrees using Equation 12.

3.12.6. Calculate the topocentric sun right ascension α' (in degrees),

$$\alpha' = \alpha + \Delta \alpha \quad . \tag{38}$$

3.12.7. Calculate the topocentric sun declination, δ' (in degrees),

$$\delta' = Arc\tan 2\,(\frac{(\sin \delta - y * \sin \xi) * \cos \Delta \alpha}{\cos \delta - x * \sin \xi * \cos H}) \quad . \tag{39}$$

3.13. Calculate the topocentric local hour angle, *H'* (in degrees),

$$H' = H - \Delta \alpha \quad . \tag{40}$$

3.14. Calculate the topocentric zenith angle, θ (in degrees):

3.14.1. Calculate the topocentric elevation angle without atmospheric refraction correction, e_0 (in degrees),

$$e_0 = Arc\sin\,(\sin \varphi * \sin \delta' + \cos \varphi * \cos \delta' * \cos H') \quad . \tag{41}$$

Then change e_0 to degrees using Equation 12.

3.14.2. Calculate the atmospheric refraction correction, Δe (in degrees),

$$\Delta e = \frac{P}{1010} * \frac{283}{273 + T} * \frac{1.02}{60 * \tan\,(e_0 + \frac{10.3}{e_0 + 5.11})} \quad , \tag{42}$$

Note that $\Delta e = 0$ when the sun is below the horizon.

where,

- P is the annual average local pressure (in millibars).
- T is the annual average local temperature (in ˚C).
- e_0 is in degrees. Calculate the tangent argument in degrees, then convert to radians if required by calculator or computer.

3.14.3. Calculate the topocentric elevation angle, e (in degrees),

$$e = e_0 + \Delta e \quad . \tag{43}$$

3.14.4. Calculate the topocentric zenith angle, θ (in degrees),

$$\theta = 90 - e \quad . \tag{44}$$

3.15. Calculate the topocentric azimuth angle, Φ (in degrees):

3.15.1. Calculate the topocentric astronomers azimuth angle, Γ (in degrees),

$$\Gamma = Arc\tan 2\left(\frac{\sin H'}{\cos H' * \sin\varphi - \tan\delta' * \cos\varphi}\right) \quad , \tag{45}$$

Change Γ to degrees using Equation 12, then limit it to the range from 0˚ to 360˚ using step 3.2.6. Note that Γ is measured <u>westward</u> from <u>south</u>.

3.15.2. Calculate the topocentric azimuth angle, Φ for navigators and solar radiation users (in degrees),

$$\Phi = \Gamma + 180 \quad , \tag{46}$$

Limit Φ to the range from 0˚ to 360˚ using step 3.2.6. Note that Φ is measured <u>eastward</u> from <u>north</u>.

3.16. Calculate the incidence angle for a surface oriented in any direction, *I* (in degrees):

$$I = Arc\cos(\cos\theta * \cos\omega + \sin\omega * \sin\theta * \cos(\Gamma - \gamma)) \quad , \tag{47}$$

where,

- ω is the slope of the surface measured from the horizontal plane.
- γ is the surface azimuth rotation angle, measured from south to the projection of the surface normal on the horizontal plane, positive or negative if oriented west or east from south, respectively.

4. SPA Evaluation and Conclusion

Because the solar zenith, azimuth, and incidence angles are not reported in the *Astronomical Almanac* (AA), the following sun parameters are used for the evaluation: The main parameters (ecliptic longitude and latitude for the mean Equinox of date, apparent right ascension, apparent declination), and the correcting parameters (nutation in longitude, nutation in obliquity, obliquity of ecliptic, and true geometric distance). Exact trigonometric functions are used with the AA reported sun parameters to calculate the solar zenith and azimuth angles, therefore it is adequate to evaluate the SPA uncertainty using these parameters. To evaluate the uncertainty of the SPA, we chose the second day of each month, for each of the years 1994, 1995, 1996, and 2004, at 0-hour Terrestrial Time (TT). Figures 2 shows that the maximum difference between the AA and SPA main parameters is -0.00015˚. Figure 3 shows that the maximum difference between the AA and SPA for calculating the zenith or azimuth angle is 0.00003˚ and 0.00008˚, respectively. This implies that the SPA is well within the stated uncertainty of ± 0.0003˚.

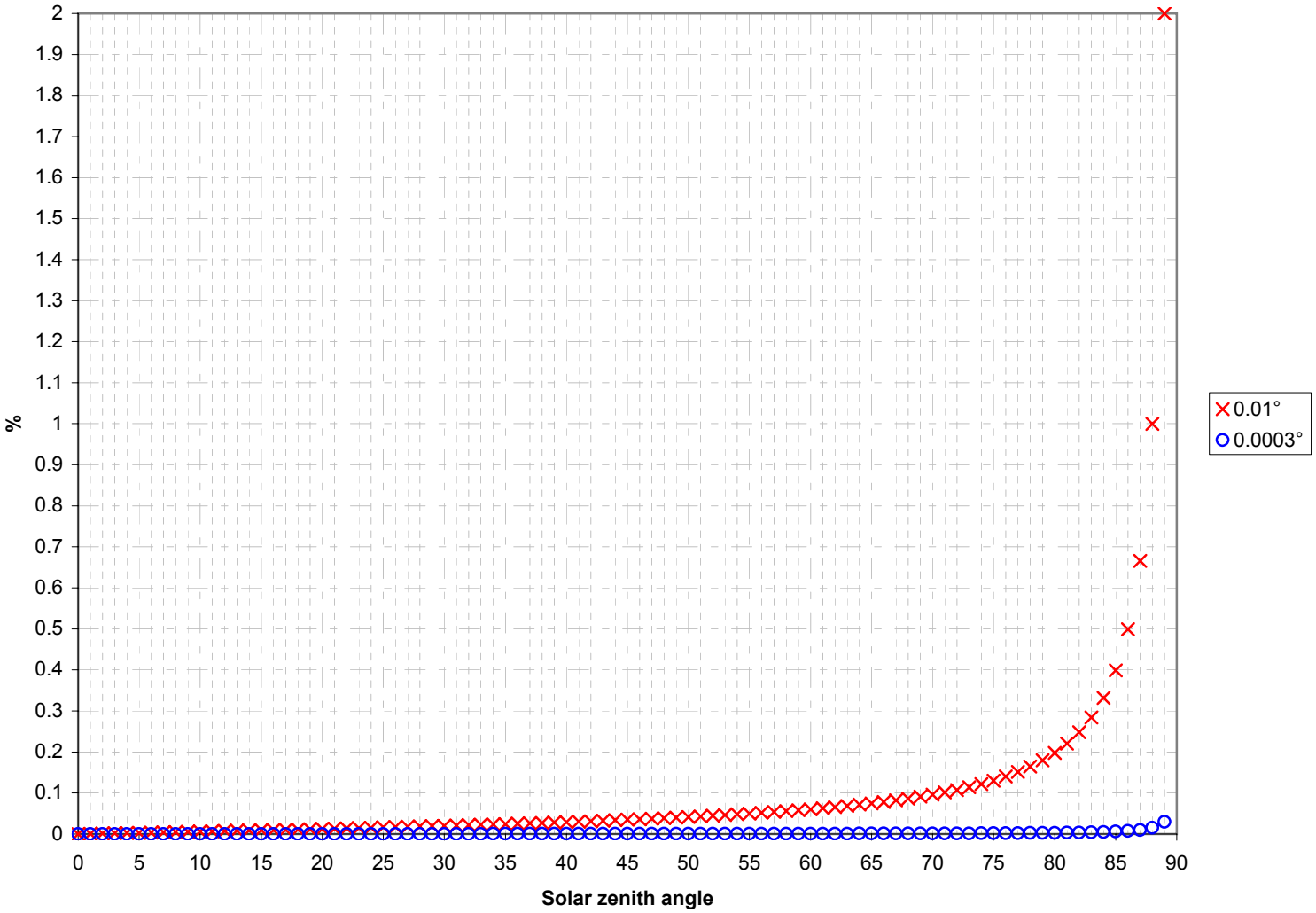

Figure 1. Uncertainty of cosine the solar zenith angle resulting from 0.01° and 0.0003° uncertainty in the angle calculation

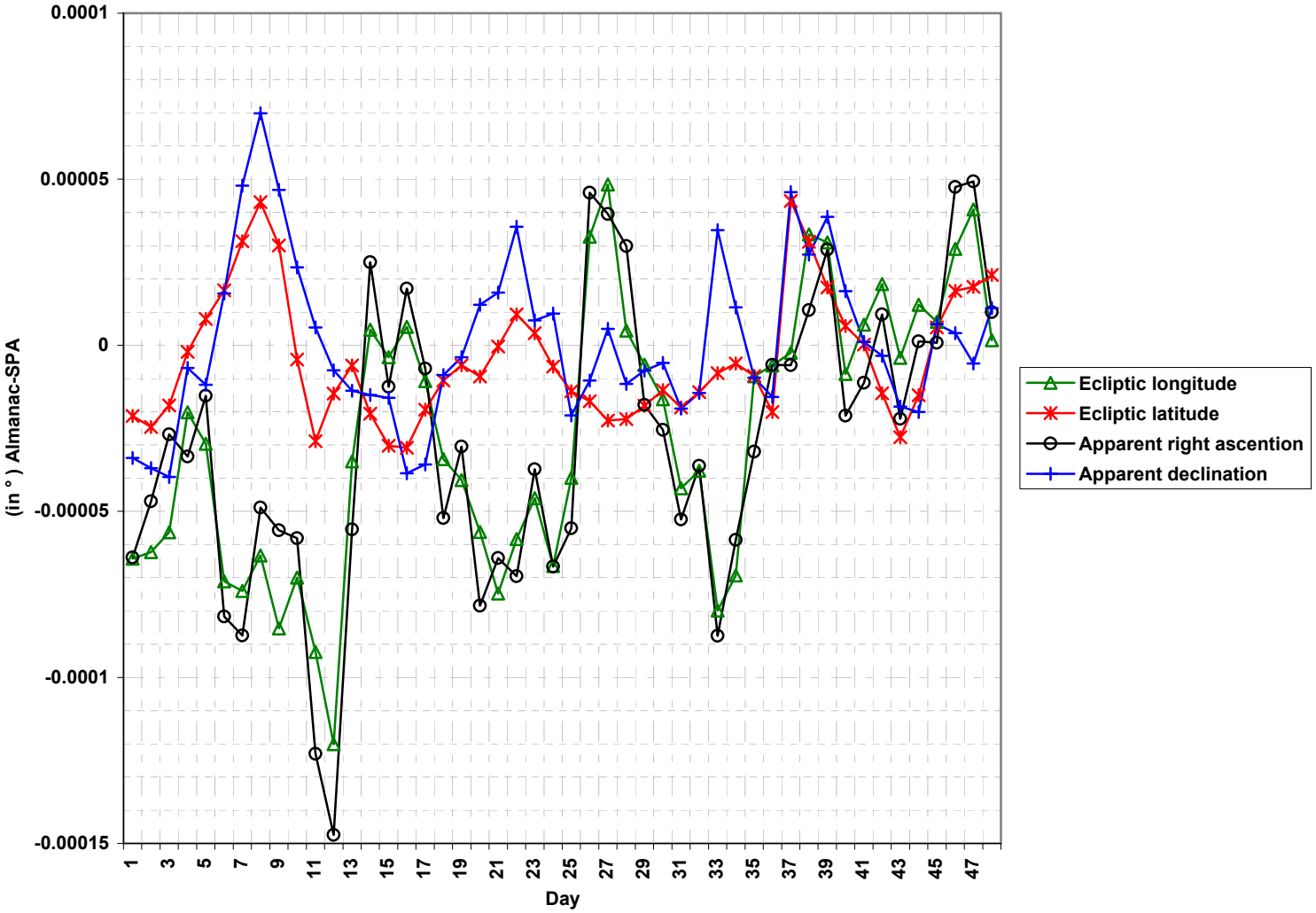

Figure 2. Difference between the Almanac and SPA for the ecliptic longitude, ecliptic latitude, apparent right ascension, and apparent declination on the second day of each month at 0-TT for the years 1994, 1995, 1996, and 2004

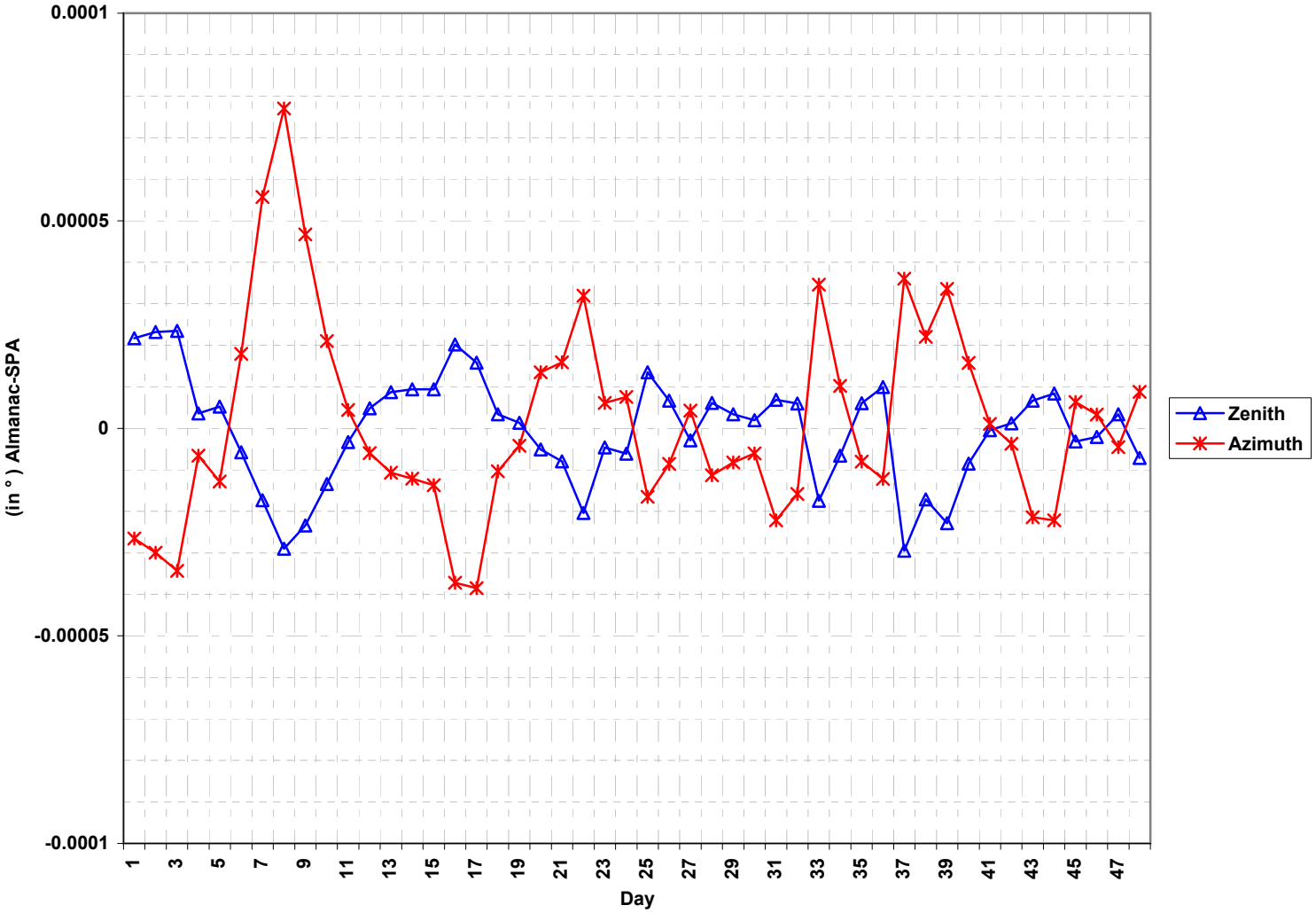

Figure 3. Difference between the Almanac and SPA for the solar zenith and azimuth angles on the second day of each month at 0-TT for the years 1994, 1995, 1996, 2004

References

1. Blanco-Muriel, M., et al. “Computing the Solar Vector”. *Solar Energy*. Vol. 70, No. 5, 2001; pp. 431-441, 2001, Great Britain.

2. Michalsky, J. J. “The Astronomical Almanac’s Algorithm for Approximate Solar Position (1950-2050)”. *Solar Energy*. Vol. 40, No. 3, 1988; pp. 227-235, USA.

3. Meeus, J. “Astronomical Algorithms”. Second edition 1998, Willmann-Bell, Inc., Richmond, Virginia, USA.

4. Iqbal, M. “An Introduction to Solar Radiation”. New York: 1983; pp. 23-25.

5. The Astronomical Almanac. Norwich:2004.

6. The U.S. Naval Observatory. Washington, DC, http://www.usno.navy.mil/.

Appendix

Note that some of the symbols used in the appendix are independent from the symbols used in the main report.

A.1. Equation of Time

The Equation of Time, E, is the difference between solar apparent and mean time. Use the following equation to calculate E (in degrees),

$$E = M - 0.0057183 - \alpha + \Delta\psi * \cos\varepsilon \quad , \tag{A1}$$

where,

- M is the sun's mean longitude (in degrees),

$$M = 280.4664567 + 360007.6982779 * JME + 0.03032028 * JME^2 + \frac{JME^3}{49931} - \frac{JME^4}{15300} - \frac{JME^5}{2000000} \quad , \tag{A2}$$

where JME is the Julian Ephemeris Millennium calculated from Equation 8, and M is limited to the range from 0° to 360° using step 3.2.6.

- α is the geocentric right ascention, from Equation 30 (in degrees).
- $\Delta\psi$ is the nutation in longitude, from Equation 22 (in degrees).
- ε is the obliquity of the ecliptic, from Equation 25 (in degrees).

Multiply E by 4 to change its unit from degrees to minutes of time. Limit E if its absolute value is greater than 20 minutes, by adding or subtracting 1440.

A.2. Sunrise, Sun Transit, and Sunset

The value of 0.5667° is typically adopted for the atmospheric refraction at sunrise and sunset times. Thus for the sun radius of 0.26667°, the value -0.8333° of sun elevation (h'_0) is chosen to calculate the times of sunrise and sunset. On the other hand, the sun transit is the time when the center of the sun reaches the local meridian.

A.2.1. Calculate the apparent sidereal time at Greenwich at 0 UT, ν (in degrees), using Equation 29.

A.2.2. Calculate the geocentric right ascension and declination at 0 TT, using Equations 30 and 31, for the day before the day of interest (D_{-1}), the day of interest (D_0),

then the day after (D_{+1}). Denote the values as $\begin{matrix}\alpha_{-1} & \delta_{-1}\\ \alpha_0 & \delta_0\\ \alpha_{+1} & \delta_{+1}\end{matrix}$, in degrees.

A.2.3. Calculate the approximate sun transit time, m_0 , in fraction of day,

$$m_0 = \frac{\alpha_0 - \sigma - \nu}{360} \quad , \tag{A3}$$

where σ is the observer geographical longitude, in degrees, positive east of Greenwich..

A.2.4. Calculate the local hour angle corresponding to the sun elevation equals -0.8333°, H_0 ,

$$H_0 = Arc\cos\left(\frac{\sin h'_0 - \sin\varphi * \sin\delta_0}{\cos\varphi * \cos\delta_0}\right) \quad , \tag{A4}$$

where,

- h'_0 equals -0.8333°.
- φ is the observer geographical latitude, in degrees, positive north of the equator.

Note that if the argument of the Arccosine is not in the range from -1 to 1, it means that the sun is always above or below the horizon for that day.

Change H_0 to degrees using Equation 12, then limit it to the range from 0° to 180° using step 3.2.6 and replacing 360 by 180.

A.2.5. Calculate the approximate sunrise time, m_1 , in fraction of day,

$$m_1 = m_0 - \frac{H_0}{360} \quad . \tag{A5}$$

A.2.6. Calculate the approximate sunset time, m_2 , in fraction of day,

$$m_2 = m_0 + \frac{H_0}{360} \quad . \tag{A6}$$

A.2.7. Limit the values of m_0 , m_1 , and m_2 to a value between 0 and 1 fraction of day using step 3.2.6 and replacing 360 by 1.

A.2.8. Calculate the sidereal time at Greenwich, in degrees, for the sun transit, sunrise, and sunset, v_i ,

$$v_i = v + 360.985647 * m_i \quad , \tag{A7}$$

where i equals 0, 1, and 2 for sun transit, sunrise, and sunset, respectively.

A.2.9. Calculate the terms n_i ,

$$n_i = m_i + \frac{\Delta T}{86400} \quad , \tag{A8}$$

where ΔT = TT-UT.

A.2.10. Calculate the values α'_i and δ'_i , in degrees, where i equals 0, 1, and 2,

where,

$$\alpha'_i = \alpha_0 + \frac{n_i(a + b + c * n_i)}{2} \quad , \tag{A9}$$

and,

$$\delta'_i = \delta_0 + \frac{n_i(a' + b' + c' * n_i)}{2} \quad , \tag{A10}$$

where,

- a and a' equal (α_0 - α_{-1}) and (δ_0 - δ_{-1}), respectively.
- b and b' equal (α_{+1} - α_0) and (δ_{+1} - δ_0), respectively.
- c and c' equal (b - a) and (b' - a'), respectively.

If the absolute value of a, a', b, or b' is greater than 2, then limit its value between 0 and 1 as shown in step A.2.7.

A.2.11. Calculate the local hour angle for the sun transit, sunrise, and sunset, H'_i (in degrees),

$$H'_i = v_i + \sigma - \alpha'_i \quad . \tag{A11}$$

H'_i in this case is measured as positive westward from the meridian, and negative eastward from the meridian. Thus limit H'_i between -180˚ and 180˚. To preserve the quadrant sign of H'_i limit it to ± 360˚ first, then if H'_i is less than or equal -180˚, then add 360˚ to force it's value to be between 0˚ and 180˚. And if H'_i is greater than or equal 180˚, then add -360˚ to force it's value to be between 0˚ and -180˚.

A.2.12. Calculate the sun altitude for the sun transit, sunrise, and sunset, h_i (in degrees),

$$h_i = Arc\sin(\sin\varphi * \sin\delta'_i + \cos\varphi * \cos\delta'_i * \cos H'_i) \quad . \tag{A12}$$

A.2.13. Calculate the sun transit, *T* (in fraction of day),

$$T = m_0 - \frac{H'_0}{360} \quad . \tag{A13}$$

A.2.14. Calculate the sunrise, *R* (in fraction of day),

$$R = m_1 + \frac{h_1 - h'_0}{360 * \cos\delta'_1 * \cos\varphi * \sin H'_1} \quad . \tag{A14}$$

A.2.15. Calculate the sunset, *S* (in fraction of day), by using Equation A14 and replacing *R* by *S*, and replacing the suffix number 1 by 2.

The fraction of day value is changed to UT by multiplying the value by 24.

To evaluate the uncertainty of the SPA, we chose the second day of each month, for each of the years 1994, 1995, 1996, and 2004, at 0-hour Terrestrial Time (TT). Figure A2.1 shows that the maximum difference between the AA and SPA sun transit time is -0.23 seconds.

Because the sunrise and sunset are recorded in the AA to a one minute resolution, we compared the SPA calculations at only three data points at Greenwich meridian at 0-UT. The comparison result in Table A2.1 shows that the maximum difference between AA and SPA is 15.4 seconds (0.26 minute), which is well within the AA resolution of one minute.

Note that UT can be changed to local time by adding the time zone as a fraction of a day (time zone is divided by 24), and limiting the result to the range from 0 to 1.

Table A2.1. The AA and SPA Results for Sunrise and Sunset at Greenwich Meridian at 0-UT

Date	Observer Latitude	Sunrise		Sunset	
		AA	SPA	AA	SPA
January 2, 1994	35°	7:08	7:08:12.8	17:00	16:59:55.9
July 5, 1996	-35°	7:08	7:08:15.4	17:00	17:01:04.5
December 4, 2004	-35°	4:39	4:38:57.1	19:02	19:02:2.5

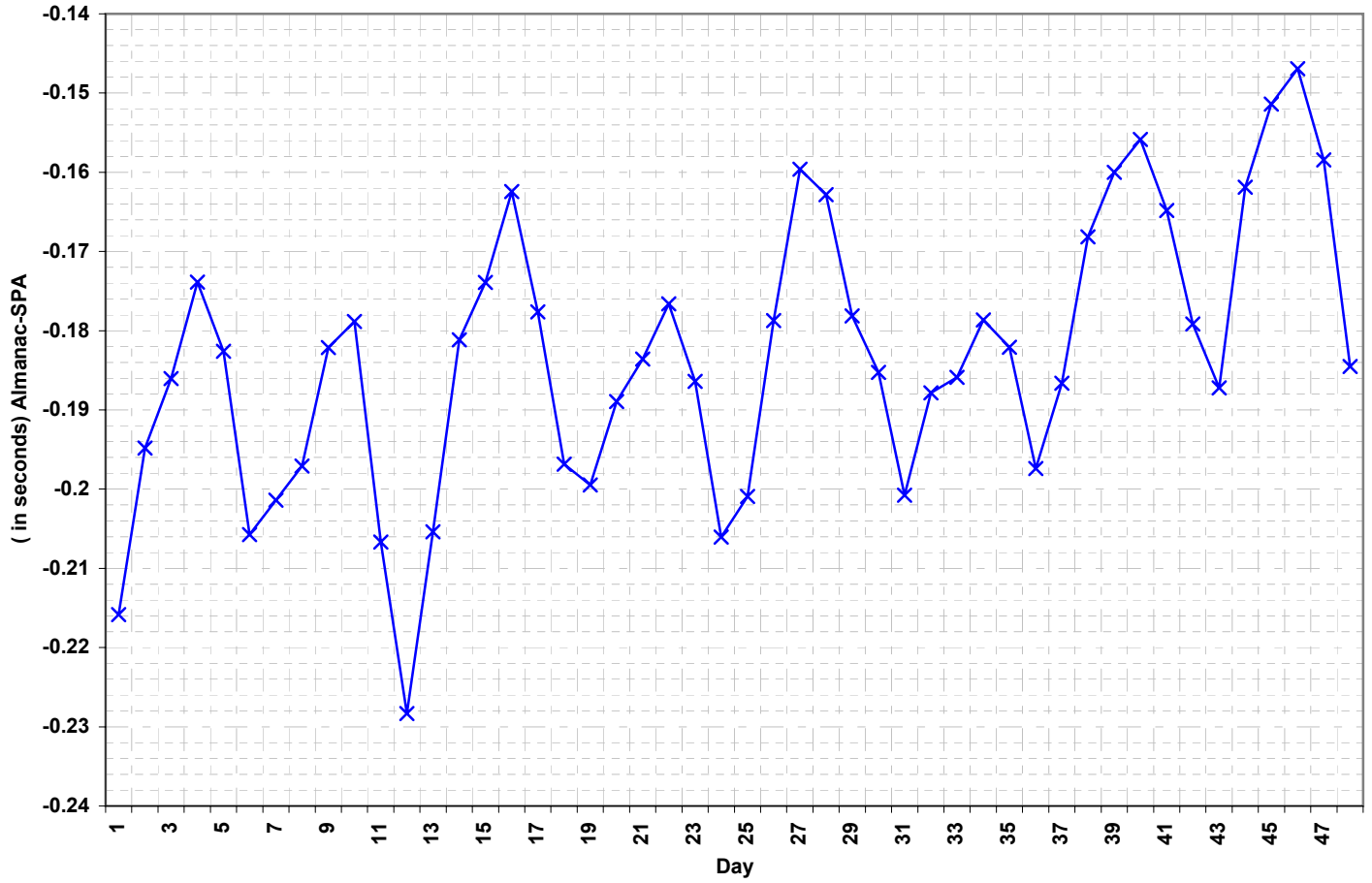

Figure A2.1. Difference between the Almanac and SPA for the Ephemeris Transit on the second day of each month at 0-TT for the years 1994, 1995, 1996, 2004

A.3. Calculation of Calendar Date from Julian Day

A.3.1. Add 0.5 to the Julian Day (*JD*), then record the integer of the result as *Z*, and the fraction decimal as *F*.

A.3.2. If *Z* is less than 2299161, then record *A* equals *Z*. Else, calculate the term *B*,

$$B = INT\left(\frac{Z - 1867216.25}{36524.25}\right) \quad , \tag{A15}$$

Then calculate the term *A*,

$$A = Z + 1 + B - INT\left(\frac{B}{4}\right) \quad . \tag{A16}$$

A.3.3. Calculate the term *C*,

$$C = A + 1524 \quad . \tag{A17}$$

A.3.4. Calculate the term *D*,

$$D = INT\left(\frac{C - 122.1}{365.25}\right) \quad . \tag{A18}$$

A.3.5. Calculate the term *G*,

$$G = INT\left(365.25 * D\right) \quad . \tag{A19}$$

A.3.6. Calculate the term *I*,

$$I = INT\left(\frac{C - G}{30.6001}\right) \quad . \tag{A20}$$

A.3.7. Calculate the day number of the month with decimals, *d*,

$$d = C - G - INT\left(30.6001 * I\right) + F \quad . \tag{A21}$$

A.3.8. Calculate the month number, *m*,

$$\begin{aligned} m &= I - 1 \,, \quad IF\ I < 14 \quad , \\ m &= I - 13 \,, \quad IF\ I \geq 14 \quad . \end{aligned} \tag{A22}$$

A.3.9. Calculate the year, *y*,

$$\begin{aligned} y &= D - 4716, \quad IF\ m > 2 \quad , \\ y &= D - 4715, \quad IF\ m \le 2 \quad . \end{aligned} \tag{A23}$$

Note that if local time is used to calculate the *JD*, then the local time zone is added to the *JD* in step A.3.1 to calculate the local Calendar Date.

A.4. Tables

Table A4.1. Examples for Testing any Program to Calculate the Julian Day

Date	UT	JD	Date	UT	JD
January 1, 2000	12:00:00	2451545.0	December 31, 1600	00:00:00	2305812.5
January 1, 1999	00:00:00	2451179.5	April 10, 837	07:12:00	2026871.8
January 27, 1987	00:00:00	2446822.5	December 31, -123	00:00:00	1676496.5
June 19, 1987	12:00:00	2446966.0	January 1, -122	00:00:00	1676497.5
January 27, 1988	00:00:00	2447187.5	July 12, -1000	12:00:00	1356001.0
June 19, 1988	12:00:00	2447332.0	February 29, -1000	00:00:00	1355866.5
January 1, 1900	00:00:00	2415020.5	August 17, -1001	21:36:00	1355671.4
January 1, 1600	00:00:00	2305447.5	January 1, -4712	12:00:00	0.0

Table A4.2. Earth Periodic Terms

Term	Row Number	A	B	C
L0	0	175347046	0	0
	1	3341656	4.6692568	6283.07585
	2	34894	4.6261	12566.1517
	3	3497	2.7441	5753.3849
	4	3418	2.8289	3.5231
	5	3136	3.6277	77713.7715
	6	2676	4.4181	7860.4194
	7	2343	6.1352	3930.2097
	8	1324	0.7425	11506.7698
	9	1273	2.0371	529.691
	10	1199	1.1096	1577.3435

	11	990	5.233	5884.927
	12	902	2.045	26.298
	13	857	3.508	398.149
	14	780	1.179	5223.694
	15	753	2.533	5507.553
	16	505	4.583	18849.228
	17	492	4.205	775.523
	18	357	2.92	0.067
	19	317	5.849	11790.629
	20	284	1.899	796.298
	21	271	0.315	10977.079
	22	243	0.345	5486.778
	23	206	4.806	2544.314
	24	205	1.869	5573.143
	25	202	2.458	6069.777
	26	156	0.833	213.299
	27	132	3.411	2942.463
	28	126	1.083	20.775
	29	115	0.645	0.98
	30	103	0.636	4694.003
	31	102	0.976	15720.839
	32	102	4.267	7.114
	33	99	6.21	2146.17
	34	98	0.68	155.42
	35	86	5.98	161000.69
	36	85	1.3	6275.96
	37	85	3.67	71430.7
	38	80	1.81	17260.15
	39	79	3.04	12036.46
	40	75	1.76	5088.63
	41	74	3.5	3154.69
	42	74	4.68	801.82
	43	70	0.83	9437.76
	44	62	3.98	8827.39

	45	61	1.82	7084.9
	46	57	2.78	6286.6
	47	56	4.39	14143.5
	48	56	3.47	6279.55
	49	52	0.19	12139.55
	50	52	1.33	1748.02
	51	51	0.28	5856.48
	52	49	0.49	1194.45
	53	41	5.37	8429.24
	54	41	2.4	19651.05
	55	39	6.17	10447.39
	56	37	6.04	10213.29
	57	37	2.57	1059.38
	58	36	1.71	2352.87
	59	36	1.78	6812.77
	60	33	0.59	17789.85
	61	30	0.44	83996.85
	62	30	2.74	1349.87
	63	25	3.16	4690.48
L1	0	628331966747	0	0
	1	206059	2.678235	6283.07585
	2	4303	2.6351	12566.1517
	3	425	1.59	3.523
	4	119	5.796	26.298
	5	109	2.966	1577.344
	6	93	2.59	18849.23
	7	72	1.14	529.69
	8	68	1.87	398.15
	9	67	4.41	5507.55
	10	59	2.89	5223.69
	11	56	2.17	155.42
	12	45	0.4	796.3
	13	36	0.47	775.52
	14	29	2.65	7.11

	15	21	5.34	0.98
	16	19	1.85	5486.78
	17	19	4.97	213.3
	18	17	2.99	6275.96
	19	16	0.03	2544.31
	20	16	1.43	2146.17
	21	15	1.21	10977.08
	22	12	2.83	1748.02
	23	12	3.26	5088.63
	24	12	5.27	1194.45
	25	12	2.08	4694
	26	11	0.77	553.57
	27	10	1.3	6286.6
	28	10	4.24	1349.87
	29	9	2.7	242.73
	30	9	5.64	951.72
	31	8	5.3	2352.87
	32	6	2.65	9437.76
	33	6	4.67	4690.48
L2	0	52919	0	0
	1	8720	1.0721	6283.0758
	2	309	0.867	12566.152
	3	27	0.05	3.52
	4	16	5.19	26.3
	5	16	3.68	155.42
	6	10	0.76	18849.23
	7	9	2.06	77713.77
	8	7	0.83	775.52
	9	5	4.66	1577.34
	10	4	1.03	7.11
	11	4	3.44	5573.14
	12	3	5.14	796.3
	13	3	6.05	5507.55
	14	3	1.19	242.73

	15	3	6.12	529.69
	16	3	0.31	398.15
	17	3	2.28	553.57
	18	2	4.38	5223.69
	19	2	3.75	0.98
L3	0	289	5.844	6283.076
	1	35	0	0
	2	17	5.49	12566.15
	3	3	5.2	155.42
	4	1	4.72	3.52
	5	1	5.3	18849.23
	6	1	5.97	242.73
L4	0	114	3.142	0
	1	8	4.13	6283.08
	2	1	3.84	12566.15
L5	0	1	3.14	0
B0	0	280	3.199	84334.662
	1	102	5.422	5507.553
	2	80	3.88	5223.69
	3	44	3.7	2352.87
	4	32	4	1577.34
B1	0	9	3.9	5507.55
	1	6	1.73	5223.69
R0	0	100013989	0	0
	1	1670700	3.0984635	6283.07585
	2	13956	3.05525	12566.1517
	3	3084	5.1985	77713.7715
	4	1628	1.1739	5753.3849
	5	1576	2.8469	7860.4194
	6	925	5.453	11506.77
	7	542	4.564	3930.21
	8	472	3.661	5884.927
	9	346	0.964	5507.553
	10	329	5.9	5223.694

	11	307	0.299	5573.143
	12	243	4.273	11790.629
	13	212	5.847	1577.344
	14	186	5.022	10977.079
	15	175	3.012	18849.228
	16	110	5.055	5486.778
	17	98	0.89	6069.78
	18	86	5.69	15720.84
	19	86	1.27	161000.69
	20	65	0.27	17260.15
	21	63	0.92	529.69
	22	57	2.01	83996.85
	23	56	5.24	71430.7
	24	49	3.25	2544.31
	25	47	2.58	775.52
	26	45	5.54	9437.76
	27	43	6.01	6275.96
	28	39	5.36	4694
	29	38	2.39	8827.39
	30	37	0.83	19651.05
	31	37	4.9	12139.55
	32	36	1.67	12036.46
	33	35	1.84	2942.46
	34	33	0.24	7084.9
	35	32	0.18	5088.63
	36	32	1.78	398.15
	37	28	1.21	6286.6
	38	28	1.9	6279.55
	39	26	4.59	10447.39
R1	0	103019	1.10749	6283.07585
	1	1721	1.0644	12566.1517
	2	702	3.142	0
	3	32	1.02	18849.23
	4	31	2.84	5507.55

	5	25	1.32	5223.69
	6	18	1.42	1577.34
	7	10	5.91	10977.08
	8	9	1.42	6275.96
	9	9	0.27	5486.78
R2	0	4359	5.7846	6283.0758
	1	124	5.579	12566.152
	2	12	3.14	0
	3	9	3.63	77713.77
	4	6	1.87	5573.14
	5	3	5.47	18849.23
R3	0	145	4.273	6283.076
	1	7	3.92	12566.15
R4	0	4	2.56	6283.08

Table A4.3. Periodic Terms for the Nutation in Longitude and Obliquity

Coefficients for Sin terms					**Coefficients for $\Delta\psi$**		**Coefficients for $\Delta\epsilon$**	
Y0	**Y1**	**Y2**	**Y3**	**Y4**	**a**	**b**	**c**	**d**
0	0	0	0	1	-171996	-174.2	92025	8.9
-2	0	0	2	2	-13187	-1.6	5736	-3.1
0	0	0	2	2	-2274	-0.2	977	-0.5
0	0	0	0	2	2062	0.2	-895	0.5
0	1	0	0	0	1426	-3.4	54	-0.1
0	0	1	0	0	712	0.1	-7	
-2	1	0	2	2	-517	1.2	224	-0.6
0	0	0	2	1	-386	-0.4	200	
0	0	1	2	2	-301		129	-0.1
-2	-1	0	2	2	217	-0.5	-95	0.3
-2	0	1	0	0	-158			
-2	0	0	2	1	129	0.1	-70	
0	0	-1	2	2	123		-53	
2	0	0	0	0	63			
0	0	1	0	1	63	0.1	-33	
2	0	-1	2	2	-59		26	

0	0	-1	0	1	-58	-0.1	32	
0	0	1	2	1	-51		27	
-2	0	2	0	0	48			
0	0	-2	2	1	46		-24	
2	0	0	2	2	-38		16	
0	0	2	2	2	-31		13	
0	0	2	0	0	29			
-2	0	1	2	2	29		-12	
0	0	0	2	0	26			
-2	0	0	2	0	-22			
0	0	-1	2	1	21		-10	
0	2	0	0	0	17	-0.1		
2	0	-1	0	1	16		-8	
-2	2	0	2	2	-16	0.1	7	
0	1	0	0	1	-15		9	
-2	0	1	0	1	-13		7	
0	-1	0	0	1	-12		6	
0	0	2	-2	0	11			
2	0	-1	2	1	-10		5	
2	0	1	2	2	-8		3	
0	1	0	2	2	7		-3	
-2	1	1	0	0	-7			
0	-1	0	2	2	-7		3	
2	0	0	2	1	-7		3	
2	0	1	0	0	6			
-2	0	2	2	2	6		-3	
-2	0	1	2	1	6		-3	
2	0	-2	0	1	-6		3	
2	0	0	0	1	-6		3	
0	-1	1	0	0	5			
-2	-1	0	2	1	-5		3	
-2	0	0	0	1	-5		3	
0	0	2	2	1	-5		3	
-2	0	2	0	1	4			

-2	1	0	2	1	4			
0	0	1	-2	0	4			
-1	0	1	0	0	-4			
-2	1	0	0	0	-4			
1	0	0	0	0	-4			
0	0	1	2	0	3			
0	0	-2	2	2	-3			
-1	-1	1	0	0	-3			
0	1	1	0	0	-3			
0	-1	1	2	2	-3			
2	-1	-1	2	2	-3			
0	0	3	2	2	-3			
2	-1	0	2	2	-3			

A.5. Example

The results for the following site parameters are listed in Table A5.1:

- Date = October 17, 2003.
- Time = 12:30:30 Local Standard Time (LST).
- Time zone(TZ) = -7 hours.
- Longitude = -105.1786˚.
- Latitude = 39.742476˚.
- Pressure = 820 mbar.
- Elevation = 1830.14 m.
- Temperature = 11˚C.
- Surface slope = 30˚.
- Surface azimuth rotation = -10˚.
- ΔT = 67 Seconds.

LST must be changed to UT by subtracting TZ from LST, and changing the date if necessary.

Table A5.1. Results for Example

JD	2452930.312847		
L0	172067561.526586	***L1***	628332010650.051147
L2	61368.682493	***L3***	-26.902819
L4	-121.279536	***L5***	-0.999999
L	24.0182616917˚		
B0	-176.502688	***B1***	3.067582
B	-0.0001011219˚		
R0	99653849.037796	***R1***	100378.567146

R2	-1140.953507	***R3***	-141.115419
R4	1.232361		
R	0.9965422974 AU		
Θ	204.0182616917˚	β	0.0001011219˚
$\Delta\psi$	-0.00399840˚	$\Delta\varepsilon$	0.00166657˚
ε	23.440465˚	λ	204.0085519281˚
α	202.22741˚	δ	-9.31434˚
H	11.105900˚	***H'***	11.10629˚
α'	202.22704˚	δ'	-9.316179˚
θ	50.11162˚	ϕ	194.34024˚
I	25.18700˚	***M***	205.8971722516˚
E	14.641503 minutes	***Transit***	18:46:04.97 UT
Sunrise	13:12:43.46 UT	***Sunset***	00:20:19.19 UT

A.6. C Code: SPA header file (SPA.h)

```
//////////////////////////////////////////
//          HEADER FILE for SPA.C       //
//                                      //
//      Solar Position Algorithm (SPA)  //
//                  for                 //
//        Solar Radiation Application   //
//                                      //
//               May 12, 2003           //
//                                      //
//   Filename: SPA.H                    //
//                                      //
//   Afshin Michael Andreas             //
//   afshin_andreas@nrel.gov (303)384-6383 //
//                                      //
//   Measurement & Instrumentation Team //
//   Solar Radiation Research Laboratory //
//   National Renewable Energy Laboratory //
//   1617 Cole Blvd, Golden, CO 80401   //
//////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
//                                                                      //
// Usage:                                                               //
//                                                                      //
//   1) In calling program, include this header file,                   //
//      by adding this line to the top of file:                         //
//           #include "spa.h"                                          //
//                                                                      //
//   2) In calling program, declare the SPA structure:                  //
//           spa_data spa;                                              //
//                                                                      //
//   3) Enter the required input values into SPA structure              //
//      (input values listed in comments below)                         //
//                                                                      //
//   4) Call the SPA calculate function and pass the SPA structure      //
//      (prototype is declared at the end of this header file):         //
//           spa_calculate(&spa);                                       //
//                                                                      //
//   Selected output values (listed in comments below) will be          //
//   computed and returned in the passed SPA structure.  Output         //
//   will based on function code selected from enumeration below.       //
//                                                                      //
//   Note: A non-zero return code from spa_calculate() indicates that   //
//         one of the input values did not pass simple bounds tests.    //
//         The valid input ranges and return error codes are also       //
//         listed below.                                                //
//////////////////////////////////////////////////////////////////////////

#ifndef __solar_position_algorithm_header
#define __solar_position_algorithm_header

//enumeration for function codes to select desired final outputs from SPA
enum {
    SPA_ZA            //calculate zenith and azimuth
    SPA_ZA_INC        //calculate zenith, azimuth, and incidence
    SPA_ZA_RTS        //calculate zenith, azimuth, and sun rise/transit/set values
    SPA_ALL           //calculate all SPA output values
};

typedef struct
{
    //----------------------INPUT VALUES------------------------

    int year;            // 4-digit year,    valid range: -2000 to 6000, error code: 1
    int month;           // 2-digit month,         valid range: 1 to 12, error code: 2
    int day;             // 2-digit day,           valid range: 1 to 31, error code: 3
    int hour;            // Observer local hour,   valid range: 0 to 24, error code: 4
    int minute;          // Observer local minute, valid range: 0 to 59, error code: 5
    int second           // Observer local second, valid range: 0 to 59, error code: 6
    double delta_t;      // Difference between earth rotation time and terrestrial time
                         // It is derived from observation only and is reported in this
                         // bulletin: http://maia.usno.navy.mil/ser7/ser7.dat,
                         // where delta_t = 32.184 + (TAI-UTC) + DUT1
                         // valid range: -8000 to 8000 seconds, error code: 7
    double timezone;     // Observer time zone (negative west of Greenwich)
```

```
                        // valid range: -12   to   12 hours,   error code: 8
    double longitude;   // Observer longitude (negative west of Greenwich)
                        // valid range: -180  to  180 degrees, error code: 9
    double latitude;    // Observer latitude (negative south of equator)
                        // valid range: -90   to   90 degrees, error code: 10
    double elevation;   // Observer elevation [meters]
                        // valid range: -6500000 or higher meters,    error code: 11
    double pressure;    // Annual average local pressure [millibars]
                        // valid range:    0 to 5000 millibars,       error code: 12
    double temperature; // Annual average local temperature [degrees Celsius]
                        // valid range: -273 to 6000 degrees Celsius, error code; 13
    double slope;       // Surface slope (measured from the horizontal plane)
                        // valid range: -360 to 360 degrees, error code: 14
    double azm_rotation; // Surface azimuth rotation (measured from south to projection of
                        //     surface normal on horizontal plane, negative west)
                        // valid range: -360 to 360 degrees, error code: 15
    double atmos_refract;// Atmospheric refraction at sunrise and sunset (0.5667 deg is typical)
                        // valid range: -5   to   5 degrees, error code: 16
    int function;       // Switch to choose functions for desired output (from enumeration)

    //-----------------Intermediate OUTPUT VALUES--------------------

    double jd;          //Julian day
    double jc;          //Julian century
    double jde;         //Julian ephemeris day
    double jce;         //Julian ephemeris century
    double jme;         //Julian ephemeris millennium
    double l;           //earth heliocentric longitude [degrees]
    double b;           //earth heliocentric latitude [degrees]
    double r;           //earth radius vector [Astronomical Units, AU]
    double theta;       //geocentric longitude [degrees]
    double beta;        //geocentric latitude [degrees]
    double x0;          //mean elongation (moon-sun) [degrees]
    double x1;          //mean anomaly (sun) [degrees]
    double x2;          //mean anomaly (moon) [degrees]
    double x3;          //argument latitude (moon) [degrees]
    double x4;          //ascending longitude (moon) [degrees]
    double del_psi;     //nutation longitude [degrees]
    double del_epsilon; //nutation obliquity [degrees]
    double epsilon0;    //ecliptic mean obliquity [arc seconds]
    double epsilon;     //ecliptic true obliquity  [degrees]
    double del_tau;     //aberration correction [degrees]
    double lamda;       //apparent sun longitude [degrees]
    double nu0;         //Greenwich mean sidereal time [degrees]
    double nu;          //Greenwich sidereal time [degrees]
    double alpha;       //geocentric sun right ascension [degrees]
    double delta;       //geocentric sun declination [degrees]
    double h;           //observer hour angle [degrees]
    double xi;          //sun equatorial horizontal parallax [degrees]
    double del_alpha;   //sun right ascension parallax [degrees]
    double delta prime; //topocentric sun declination [degrees]
    double alpha prime; //topocentric sun right ascension [degrees]
    double h_prime;     //topocentric local hour angle [degrees]
    double e0           //topocentric elevation angle (uncorrected) [degrees]
    double del_e;       //atmospheric refraction correction [degrees]
    double e;           //topocentric elevation angle (corrected) [degrees]
    double eot;         //equation of time [minutes]
    double srha;        //sunrise hour angle [degrees]
    double ssha;        //sunset hour angle [degrees]
    double sta;         //sun transit altitude [degrees]

    //----------------------Final OUTPUT VALUES------------------------

    double zenith;      //topocentric zenith angle [degrees]
    double azimuth180;  //topocentric azimuth angle (westward from south) [-180 to 180 degrees]
    double azimuth;     //topocentric azimuth angle (eastward from north) [   0 to 360 degrees]
    double incidence;   //surface incidence angle [degrees]
    double suntransit;  //local sun transit time (or solar noon) [fractional hour]
    double sunrise;     //local sunrise time (+/- 30 seconds) [fractional hour]
    double sunset;      //local sunset time (+/- 30 seconds) [fractional hour]
} spa_data;

//Calculate SPA output values (in structure) based on input values passed in structure
int spa_calculate(spa_data *spa);

#endif
```

A.7. C Code: SPA source file (SPA.c)

```
////////////////////////////////////////////
//      Solar Position Algorithm (SPA)    //
//                for                     //
//        Solar Radiation Application     //
//                                        //
//              May 12, 2003              //
//                                        //
//   Filename: SPA.C                      //
//                                        //
//   Afshin Michael Andreas               //
//   afshin_andreas@nrel.gov (303)384-6383 //
//                                        //
//   Measurement & Instrumentation Team   //
//   Solar Radiation Research Laboratory  //
//   National Renewable Energy Laboratory //
//   1617 Cole Blvd, Golden, CO 80401     //
////////////////////////////////////////////

////////////////////////////////////////////
//   See the SPA.H header file for usage  //
//                                        //
//   This code is based on the NREL       //
//   technical report "Solar Position     //
//   Algorithm for Solar Radiation        //
//   Application" by I. Reda & A. Andreas //
////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////
//
//   NOTICE
//
//This solar position algorithm for solar radiation applications (the "data") was produced by
//the National Renewable Energy Laboratory ("NREL"), which is operated by the Midwest Research
//Institute ("MRI") under Contract No. DE-AC36-99-GO10337 with the U.S. Department of Energy
//(the "Government").
//
//Reference herein, directly or indirectly to any specific commercial product, process, or
//service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply
//its endorsement, recommendation, or favoring by the Government, MRI or NREL.
//
//THESE DATA ARE PROVIDED "AS IS" AND NEITHER THE GOVERNMENT, MRI, NREL NOR ANY OF THEIR
//EMPLOYEES, MAKES ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING THE WARRANTIES OF MERCHANTABILITY
//AND FITNESS FOR A PARTICULAR PURPOSE, OR ASSUMES ANY LEGAL LIABILITY OR RESPONSIBILITY FOR THE
//ACCURACY, COMPLETENESS, OR USEFULNESS OF ANY SUCH INFORMATION DISCLOSED IN THE ALGORITHM, OR
//OF ANY APPARATUS, PRODUCT, OR PROCESS DISCLOSED, OR REPRESENTS THAT ITS USE WOULD NOT INFRINGE
//PRIVATELY OWNED RIGHTS.
//


////////////////////////////////////////////////////////////////////////////////////////////////
// Revised 27-FEB-2004 Andreas
//         Added bounds check on inputs and return value for spa_calculate().
// Revised 10-MAY-2004 Andreas
//         Changed temperature bound check minimum from -273.15 to -273 degrees C.
// Revised 17-JUN-2004 Andreas
//         Corrected a problem that caused a bogus sunrise/set/transit on the equinox.
// Revised 18-JUN-2004 Andreas
//         Added a "function" input variable that allows the selecting of desired outputs.
// Revised 21-JUN-2004 Andreas
//         Added 3 new intermediate output values to SPA structure (srha, ssha, & sta).
// Revised 23-JUN-2004 Andreas
//         Enumerations for "function" were renamed and 2 were added.
//         Prevented bound checks on inputs that are not used (based on function).
// Revised 01-SEP-2004 Andreas
//         Changed a local variable from integer to double.
// Revised 12-JUL-2005 Andreas
//         Put a limit on the EOT calculation, so that the result is between -20 and 20.
// Revised 26-OCT-2005 Andreas
//         Set the atmos. refraction correction to zero, when sun is below horizon.
//         Made atmos_refract input a requirement for all "functions".
//         Changed atmos_refract bound check from +/- 10 to +/- 5 degrees.
// Revised 07-NOV-2006 Andreas
//         Corrected 3 earth periodic terms in the L_TERMS array.
//         Corrected 2 earth periodic terms in the R_TERMS array.
```

```
// Revised 10-NOV-2006 Andreas
//         Corrected a constant used to calculate topocentric sun declination.
//         Put a limit on observer hour angle, so result is between 0 and 360.
// Revised 13-NOV-2006 Andreas
//         Corrected calculation of topocentric sun declination.
//         Converted all floating point inputs in spa structure to doubles.
// Revised 27-FEB-2007 Andreas
//         Minor correction made as to when atmos. refraction correction is set to zero.
//////////////////////////////////////////////////////////////////////////////////////////

#include <math.h>
#include "spa.h"

#define PI         3.14159265358979323846264338327950288419 71
#define SUN_RADIUS 0.26667

#define L_COUNT 6
#define B_COUNT 2
#define R_COUNT 5
#define Y_COUNT 63

#define L_MAX_SUBCOUNT 64
#define B_MAX_SUBCOUNT 5
#define R_MAX_SUBCOUNT 40

enum {TERM_A, TERM_B, TERM_C, TERM_COUNT};
enum {TERM_X0, TERM_X1, TERM_X2, TERM_X3, TERM_X4, TERM_X_COUNT};
enum {TERM_PSI_A, TERM_PSI_B, TERM_EPS_C, TERM_EPS_D, TERM_PE_COUNT};
enum {JD_MINUS, JD_ZERO, JD_PLUS, JD_COUNT};
enum {SUN_TRANSIT, SUN_RISE, SUN_SET, SUN_COUNT};

#define TERM_Y_COUNT TERM_X_COUNT

const int l_subcount[L_COUNT] = {64,34,20,7,3,1};
const int b_subcount[B_COUNT] = {5,2};
const int r_subcount[R_COUNT] = {40,10,6,2,1};

/////////////////////////////////////////////////
///  Earth Periodic Terms
/////////////////////////////////////////////////
const double L_TERMS[L_COUNT][L_MAX_SUBCOUNT][TERM_COUNT]=
{
    {
        {175347046.0,0,0},
        {3341656.0,4.6692568,6283.07585},
        {34894.0,4.6261,12566.1517},
        {3497.0,2.7441,5753.3849},
        {3418.0,2.8289,3.5231},
        {3136.0,3.6277,77713.7715},
        {2676.0,4.4181,7860.4194},
        {2343.0,6.1352,3930.2097},
        {1324.0,0.7425,11506.7698},
        {1273.0,2.0371,529.691},
        {1199.0,1.1096,1577.3435},
        {990,5.233,5884.927},
        {902,2.045,26.298},
        {857,3.508,398.149},
        {780,1.179,5223.694},
        {753,2.533,5507.553},
        {505,4.583,18849.228},
        {492,4.205,775.523},
        {357,2.92,0.067},
        {317,5.849,11790.629},
        {284,1.899,796.298},
        {271,0.315,10977.079},
        {243,0.345,5486.778},
        {206,4.806,2544.314},
        {205,1.869,5573.143},
        {202,2.458,6069.777},
        {156,0.833,213.299},
        {132,3.411,2942.463},
        {126,1.083,20.775},
        {115,0.645,0.98},
        {103,0.636,4694.003},
        {102,0.976,15720.839},
         102,4.267,7.114},
```

```
    {99,6.21,2146.17},
    {98,0.68,155.42},
    {86,5.98,161000.69},
    {85,1.3,6275.96},
    {85,3.67,71430.7},
    {80,1.81,17260.15},
    {79,3.04,12036.46},
    {75,1.76,5088.63},
    {74,3.5,3154.69},
    {74,4.68,801.82},
    {70,0.83,9437.76},
    {62,3.98,8827.39},
    {61,1.82,7084.9},
    {57,2.78,6286.6},
    {56,4.39,14143.5},
    {56,3.47,6279.55},
    {52,0.19,12139.55},
    {52,1.33,1748.02},
    {51,0.28,5856.48},
    {49,0.49,1194.45},
    {41,5.37,8429.24},
    {41,2.4,19651.05},
    {39,6.17,10447.39},
    {37,6.04,10213.29},
    {37,2.57,1059.38},
    {36,1.71,2352.87},
    {36,1.78,6812.77},
    {33,0.59,17789.85},
    {30,0.44,83996.85},
    {30,2.74,1349.87},
    {25,3.16,4690.48}
},
{
    {628331966747.0,0,0},
    {206059.0,2.678235,6283.07585},
    {4303.0,2.6351,12566.1517},
    {425.0,1.59,3.523},
    {119.0,5.796,26.298},
    {109.0,2.966,1577.344},
    {93,2.59,18849.23},
    {72,1.14,529.69},
    {68,1.87,398.15},
    {67,4.41,5507.55},
    {59,2.89,5223.69},
    {56,2.17,155.42},
    {45,0.4,796.3},
    {36,0.47,775.52},
    {29,2.65,7.11},
    {21,5.34,0.98},
    {19,1.85,5486.78},
    {19,4.97,213.3},
    {17,2.99,6275.96},
    {16,0.03,2544.31},
    {16,1.43,2146.17},
    {15,1.21,10977.08},
    {12,2.83,1748.02},
    {12,3.26,5088.63},
    {12,5.27,1194.45},
    {12,2.08,4694},
    {11,0.77,553.57},
    {10,1.3,6286.6},
    {10,4.24,1349.87},
    {9,2.7,242.73},
    {9,5.64,951.72},
    {8,5.3,2352.87},
    {6,2.65,9437.76},
    {6,4.67,4690.48}
},
{
    {52919.0,0,0},
    {8720.0,1.0721,6283.0758},
    {309.0,0.867,12566.152},
    {27,0.05,3.52},
    {16,5.19,26.3},
    {16,3.68,155.42},
    {10,0.76,        },
```

```
        {9,2.06,77713.77},
        {7,0.83,775.52},
        {5,4.66,1577.34},
        {4,1.03,7.11},
        {4,3.44,5573.14},
        {3,5.14,796.3},
        {3,6.05,5507.55},
        {3,1.19,242.73},
        {3,6.12,529.69},
        {3,0.31,398.15},
        {3,2.28,553.57},
        {2,4.38,5223.69},
        {2,3.75,0.98}
    },
    {
        {289.0,5.844,6283.076},
        {35,0,0},
        {17,5.49,12566.15},
        {3,5.2,155.42},
        {1,4.72,3.52},
        {1,5.3,18849.23},
        {1,5.97,242.73}
    },
    {
        {114.0,3.142,0},
        {8,4.13,6283.08},
        {1,3.84,12566.15}
    },
    {
        {1,3.14,0}
    }
};

const double B_TERMS[B_COUNT][B_MAX_SUBCOUNT][TERM_COUNT]=
{
    {
        {280.0,3.199,84334.662},
        {102.0,5.422,5507.553},
        {80,3.88,5223.69},
        {44,3.7,2352.87},
        {32,4,1577.34}
    },
    {
        {9,3.9,5507.55},
        {6,1.73,5223.69}
    }
};

const double R_TERMS[R_COUNT][R_MAX_SUBCOUNT][TERM_COUNT]=
{
    {
        {100013989.0,0,0},
        {1670700.0,3.0984635,6283.07585},
        {13956.0,3.05525,12566.1517},
        {3084.0,5.1985,77713.7715},
        {1628.0,1.1739,5753.3849},
        {1576.0,2.8469,7860.4194},
        {925.0,5.453,11506.77},
        {542.0,4.564,3930.21},
        {472.0,3.661,5884.927},
        {346.0,0.964,5507.553},
        {329.0,5.9,5223.694},
        {307.0,0.299,5573.143},
        {243.0,4.273,11790.629},
        {212.0,5.847,1577.344},
        {186.0,5.022,10977.079},
        {175.0,3.012,18849.228},
        {110.0,5.055,5486.778},
        {98,0.89,6069.78},
        {86,5.69,15720.84},
        {86,1.27,161000.69},
        {65,0.27,17260.15},
        {63,0.92,529.69},
        {57,2.01,83996.85},
        {56,5.24,71430.7},
        {49,3.25,2544.31 ,
```

```
        {47,2.58,775.52},
        {45,5.54,9437.76},
        {43,6.01,6275.96},
        {39,5.36,4694},
        {38,2.39,8827.39},
        {37,0.83,19651.05},
        {37,4.9,12139.55},
        {36,1.67,12036.46},
        {35,1.84,2942.46},
        {33,0.24,7084.9},
        {32,0.18,5088.63},
        {32,1.78,398.15},
        {28,1.21,6286.6},
        {28,1.9,6279.55},
        {26,4.59,10447.39}
    },
    {
        {103019.0,1.10749,6283.07585},
        {1721.0,1.0644,12566.1517},
        {702.0,3.142,0},
        {32,1.02,18849.23},
        {31,2.84,5507.55},
        {25,1.32,5223.69},
        {18,1.42,1577.34},
        {10,5.91,10977.08},
        {9,1.42,6275.96},
        {9,0.27,5486.78}
    },
    {
        {4359.0,5.7846,6283.0758},
        {124.0,5.579,12566.152},
        {12,3.14,0},
        {9,3.63,77713.77},
        {6,1.87,5573.14},
        {3,5.47,18849.23}
    },
    {
        {145.0,4.273,6283.076},
        {7,3.92,12566.15}
    },
    {
        {4,2.56,6283.08}
    }
};

//////////////////////////////////////////////////////////////
///  Periodic Terms for the nutation in longitude and obliquity


const int Y_TERMS[Y_COUNT][TERM_Y_COUNT]=
{
    {0,0,0,0,1},
    {-2,0,0,2,2},
    {0,0,0,2,2},
    {0,0,0,0,2},
    {0,1,0,0,0},
    {0,0,1,0,0},
    {-2,1,0,2,2},
    {0,0,0,2,1},
    {0,0,1,2,2},
    {-2,-1,0,2,2},
    {-2,0,1,0,0},
    {-2,0,0,2,1},
    {0,0,-1,2,2},
    {2,0,0,0,0},
    {0,0,1,0,1},
    {2,0,-1,2,2},
    {0,0,-1,0,1},
    {0,0,1,2,1},
    {-2,0,2,0,0},
    {0,0,-2,2,1},
    {2,0,0,2,2},
    {0,0,2,2,2},
    {0,0,2,0,0},
    {-2,0,1,2,2},
     0,0,0,2,0},
```

```
    {-2,0,0,2,0},
    {0,0,-1,2,1},
    {0,2,0,0,0},
    {2,0,-1,0,1},
    {-2,2,0,2,2},
    {0,1,0,0,1},
    {-2,0,1,0,1},
    {0,-1,0,0,1},
    {0,0,2,-2,0},
    {2,0,-1,2,1},
    {2,0,1,2,2},
    {0,1,0,2,2},
    {-2,1,1,0,0},
    {0,-1,0,2,2},
    {2,0,0,2,1},
    {2,0,1,0,0},
    {-2,0,2,2,2},
    {-2,0,1,2,1},
    {2,0,-2,0,1},
    {2,0,0,0,1},
    {0,-1,1,0,0},
    {-2,-1,0,2,1},
    {-2,0,0,0,1},
    {0,0,2,2,1},
    {-2,0,2,0,1},
    {-2,1,0,2,1},
    {0,0,1,-2,0},
    {-1,0,1,0,0},
    {-2,1,0,0,0},
    {1,0,0,0,0},
    {0,0,1,2,0},
    {0,0,-2,2,2},
    {-1,-1,1,0,0},
    {0,1,1,0,0},
    {0,-1,1,2,2},
    {2,-1,-1,2,2},
    {0,0,3,2,2},
    {2,-1,0,2,2},
};

const double PE_TERMS[Y_COUNT][TERM_PE_COUNT]={
    {-171996,-174.2,92025,8.9},
    {-13187,-1.6,5736,-3.1},
    {-2274,-0.2,977,-0.5},
    {2062,0.2,-895,0.5},
    {1426,-3.4,54,-0.1},
    {712,0.1,-7,0},
    {-517,1.2,224,-0.6},
    {-386,-0.4,200,0},
    {-301,0,129,-0.1},
    {217,-0.5,-95,0.3},
    {-158,0,0,0},
    {129,0.1,-70,0},
    {123,0,-53,0},
    {63,0,0,0},
    {63,0.1,-33,0},
    {-59,0,26,0},
    {-58,-0.1,32,0},
    {-51,0,27,0},
    {48,0,0,0},
    {46,0,-24,0},
    {-38,0,16,0},
    {-31,0,13,0},
    {29,0,0,0},
    {29,0,-12,0},
    {26,0,0,0},
    {-22,0,0,0},
    {21,0,-10,0},
    {17,-0.1,0,0},
    {16,0,-8,0},
    {-16,0.1,7,0},
    {-15,0,9,0},
    {-13,0,7,0},
    {-12,0,6,0},
    {11,0,0,0},
    {-10,0,5, },
```

```
    {-8,0,3,0},
    {7,0,-3,0},
    {-7,0,0,0},
    {-7,0,3,0},
    {-7,0,3,0},
    {6,0,0,0},
    {6,0,-3,0},
    {6,0,-3,0},
    {-6,0,3,0},
    {-6,0,3,0},
    {5,0,0,0},
    {-5,0,3,0},
    {-5,0,3,0},
    {-5,0,3,0},
    {4,0,0,0},
    {4,0,0,0},
    {4,0,0,0},
    {-4,0,0,0},
    {-4,0,0,0},
    {-4,0,0,0},
    {3,0,0,0},
    {-3,0,0,0},
    {-3,0,0,0},
    {-3,0,0,0},
    {-3,0,0,0},
    {-3,0,0,0},
    {-3,0,0,0},
    {-3,0,0,0},
};

////////////////////////////////////////////

double rad2deg(double radians)
{
    return (180.0/PI)*radians;
}

double deg2rad(double degrees)
{
    return (PI/180.0)*degrees;
}

double limit_degrees(double degrees)
{
    double limited;

    degrees /= 360.0;
    limited = 360.0*(degrees-floor(degrees));
    if (limited < 0) limited += 360.0;

    return limited;
}

double limit_degrees180pm(double degrees)
{
    double limited;

    degrees /= 360.0;
    limited = 360.0*(degrees-floor(degrees));
    if      (limited < -180.0) limited += 360.0;
    else if (limited >  180.0) limited -= 360.0;

    return limited;
}

double limit_degrees180(double degrees)
{
    double limited;

    degrees /= 180.0;
    limited = 180.0*(degrees-floor(degrees));
    if (limited < 0)         +=      ;

    return limited;
}
```

```
double limit_zero2one(double value)
{
    double limited;

    limited = value - floor(value);
    if (limited < 0) limited += 1.0;

    return limited;
}

double limit_minutes(double minutes)
{
    double limited=minutes;

    if      (limited < -20.0) limited += 1440.0;
    else if (limited >  20.0) limited -= 1440.0;

    return limited;
}

double dayfrac_to_local_hr(double dayfrac, double timezone)
{
    return 24.0*limit_zero2one(dayfrac + timezone/24.0);
}

double third_order_polynomial(double a, double b, double c, double d, double x)
{
    return ((a*x + b)*x + c)*x + d;
}

////////////////////////////////////////////////////////////////////////////////////////
int validate_inputs(spa_data *spa)
{
    if ((spa->year        < -2000) || (spa->year        > 6000)) return 1;
    if ((spa->month       < 1    ) || (spa->month       > 12  )) return 2;
    if ((spa->day         < 1    ) || (spa->day         > 31  )) return 3;
    if ((spa->hour        < 0    ) || (spa->hour        > 24  )) return 4;
    if ((spa->minute      < 0    ) || (spa->minute      > 59  )) return 5;
    if ((spa->second      < 0    ) || (spa->second      > 59  )) return 6;
    if ((spa->pressure    < 0    ) || (spa->pressure    > 5000)) return 12;
    if ((spa->temperature <= -273) || (spa->temperature > 6000)) return 13;
    if ((spa->hour        == 24  ) && (spa->minute      > 0   )) return 5;
    if ((spa->hour        == 24  ) && (spa->second      > 0   )) return 6;

    if (fabs(spa->delta_t)       > 8000    ) return 7;
    if (fabs(spa->timezone)      > 12      ) return 8;
    if (fabs(spa->longitude)     > 180     ) return 9;
    if (fabs(spa->latitude)      > 90      ) return 10;
    if (fabs(spa->atmos_refract) > 5       ) return 16;
    if (     spa->elevation      < -6500000) return 11;

    if ((spa->function == SPA_ZA_INC) || (spa->function == SPA_ALL))
    {
        if (fabs(spa->slope)         > 360) return 14;
        if (fabs(spa->            )  > 360) return 15;
    }

    return 0;
}
////////////////////////////////////////////////////////////////////////////////////////
double julian_day (int year, int month, int day, int hour, int minute, int second, double tz)
{
    double day_decimal, julian_day, a;

    day_decimal = day + (hour - tz + (minute + second/60.0)/60.0)/24.0;

    if (month < 3) {
        month += 12;
        year--;
    }

    julian_day = floor(365.25*(year+4716.0)) + floor(30.6001*(month+1)) + day_decimal - 1524.5;

    if (julian_day > 2299160.0) {
        a = floor(year/100);
        julian_day    (2 - a + floor(a/4));
```

```
    }

    return julian_day;
}

double julian_century(double jd)
{
    return (jd-2451545.0)/36525.0;
}

double julian_ephemeris_day(double jd, double delta_t)
{
    return jd+delta_t/86400.0;
}

double julian_ephemeris_century(double jde)
{
    return (jde - 2451545.0)/36525.0;
}

double julian_ephemeris_millennium(double jce)
{
    return (jce/10.0);
}

double earth_periodic_term_summation(const double terms[][TERM_COUNT], int count, double jme)
{
    int i;
    double sum=0;

    for (i = 0; i < count; i++)
        sum += terms[i][TERM_A]*cos(terms[i][TERM_B]+terms[i][TERM_C]*jme);

    return sum;
}

double earth_values(double term_sum[], int count, double jme)
{
    int i;
    double sum=0;

    for (i = 0; i < count; i++)
        sum += term_sum[i]*pow(jme, i);

    sum /= 1.0e8;

    return sum;
}

double earth_heliocentric_longitude(double jme)
{
    double sum[L_COUNT];
    int i;

    for (i = 0; i < L_COUNT; i++)
        sum[i] = earth_periodic_term_summation(L_TERMS[i], l_subcount[i], jme);

    return limit_degrees(rad2deg(earth_values(sum, L_COUNT, jme)));

}

double earth_heliocentric_latitude(double jme)
{
    double sum[B_COUNT];
    int i;

    for (i = 0; i < B_COUNT; i++)
        sum[i] =      _        _    _          (B_TERMS[i], b_subcount[i], jme);

    return rad2deg(earth_values(sum, B_COUNT, jme));

}

double earth_radius_vector(double jme)
{
    double sum[R_COUNT];
```

```
    int i;

    for (i = 0; i < R_COUNT; i++)
        sum[i] = earth_periodic_term_summation(R_TERMS[i], r_subcount[i], jme);

    return earth_values(sum, R_COUNT, jme);

}

double geocentric_longitude(double l)
{
    double theta = l + 180.0;

    if (theta >= 360.0) theta -= 360.0;

    return theta;
}

double geocentric_latitude(double b)
{
    return -b;
}

double mean_elongation_moon_sun(double jce)
{
    return third_order_polynomial(1.0/189474.0, -0.0019142, 445267.11148, 297.85036, jce);
}

double mean_anomaly_sun(double jce)
{
    return third_order_polynomial(-1.0/300000.0, -0.0001603, 35999.05034, 357.52772, jce);
}

double mean_anomaly_moon(double jce)
{
    return third_order_polynomial(1.0/56250.0, 0.0086972, 477198.867398, 134.96298, jce);
}

double argument_latitude_moon(double jce)
{
    return third_order_polynomial(1.0/327270.0, -0.0036825, 483202.017538, 93.27191, jce);
}

double ascending_longitude_moon(double jce)
{
    return third_order_polynomial(1.0/450000.0, 0.0020708, -1934.136261, 125.04452, jce);
}

double xy_term_summation(int i, double x[TERM_X_COUNT])
{
    int j;
    double sum=0;

    for (j = 0; j < TERM_Y_COUNT; j++)
        sum += x[j]*Y_TERMS[i][j];

    return sum;
}

void nutation_longitude_and_obliquity(double jce, double x[TERM_X_COUNT], double *del_psi,
                                                                           double *del_epsilon)
{
    int i;
    double xy_term_sum, sum_psi=0, sum_epsilon=0;

    for (i = 0; i < Y_COUNT; i++) {
        xy_term_sum  = deg2rad(xy_term_summation(i, x));
        sum_psi     += (       [i][TERM_PSI_A] + jce*PE_TERMS[i][TERM_PSI_B])*sin(xy_term_sum);
        sum_epsilon += (PE_TERMS[i][TERM_EPS_C] + jce*PE_TERMS[i][TERM_EPS_D])*cos(xy_term_sum);
    }

    *del_psi     = sum_psi     / 36000000.0;
    *del_epsilon =             / 36000000.0;
}

double ecliptic_mean_obliquity(double jme)
```

```
{
    double u = jme/10.0;

    return 84381.448 + u*(-4680.96 + u*(-1.55 + u*(1999.25 + u*(-51.38 + u*(-249.67 +
                      u*(  -39.05 + u*( 7.12 + u*(  27.87 + u*(  5.79 + u*2.45)))))))));
}

double ecliptic_true_obliquity(double delta_epsilon, double epsilon0)
{
    return delta_epsilon + epsilon0/3600.0;
}

double aberration_correction(double r)
{
    return -20.4898 / (3600.0*r);
}

double apparent_sun_longitude(double theta, double delta_psi, double delta_tau)
{
    return theta + delta_psi + delta_tau;
}

double greenwich_mean_sidereal_time (double jd, double jc)
{
    return limit_degrees(280.46061837 + 360.98564736629 * (jd - 2451545.0) +
                                        jc*jc*(0.000387933 - jc/38710000.0));
}

double greenwich_sidereal_time (double nu0, double delta_psi, double epsilon)
{
    return nu0 + delta_psi*cos(deg2rad(epsilon));
}

double geocentric_sun_right_ascension(double lamda, double epsilon, double beta)
{
    double lamda_rad   = deg2rad(lamda);
    double epsilon_rad = deg2rad(epsilon);

    return limit_degrees(rad2deg(atan2(sin(lamda_rad)*cos(epsilon_rad) -
                                        tan(deg2rad(beta))*sin(epsilon_rad), cos(lamda_rad))));
}

double geocentric_sun_declination(double beta, double epsilon, double lamda)
{
    double beta_rad    = deg2rad(beta);
    double epsilon_rad = deg2rad(epsilon);

    return rad2deg(asin(sin(beta_rad)*cos(epsilon_rad) +
                        cos(beta_rad)*sin(epsilon_rad)*sin(deg2rad(lamda))));
}

double observer_hour_angle(double nu, double longitude, double alpha_deg)
{
    return limit_degrees(nu + longitude - alpha_deg);
}

double sun_equatorial_horizontal_parallax(double r)
{
    return 8.794 / (3600.0 * r);
}

void sun_right_ascension_parallax_and_topocentric_dec(double latitude, double elevation,
               double xi, double h, double delta,        *delta_alpha, double *delta_prime)
{
    double delta_alpha_rad;
    double lat_rad   = deg2rad(latitude);
    double xi_rad    = deg2rad(xi);
    double h_rad     = deg2rad(h);
    double delta_rad = deg2rad(delta);
    double u = atan(0.99664719 * tan(lat_rad));
    double y = 0.99664719 * sin(u) + elevation*sin(lat_rad)/6378140.0;
    double x =              cos(u) + elevation*cos(lat_rad)/6378140.0;

    delta_alpha_rad =      atan2(                - x*sin(xi_rad) *sin(h_rad),
                                 cos(delta_rad) - x*sin(xi_rad) *cos h_rad));
```

```
    *delta_prime = rad2deg(atan2((sin(delta_rad) - y*sin(xi_rad))*cos(delta_alpha_rad),
                                 cos(delta_rad) - x*sin(xi_rad) *cos(h_rad)));

    *delta_alpha = rad2deg(delta_alpha_rad);
}

double topocentric_sun_right_ascension(double alpha_deg, double delta_alpha)
{
    return alpha_deg + delta_alpha;
}

double topocentric_local_hour_angle(double h, double delta_alpha)
{
    return h - delta_alpha;
}

double topocentric_elevation_angle(double latitude, double delta_prime, double h_prime)
{
    double lat_rad         = deg2rad(latitude);
    double delta_prime_rad = deg2rad(delta_prime);

    return rad2deg(asin(sin(lat_rad)*sin(delta_prime_rad) +
                        cos(lat_rad)*cos(delta_prime_rad) * cos(deg2rad(h_prime))));
}

double atmospheric_refraction_correction(double pressure, double temperature,
                                         double atmos_refract, double e0)
{
    double del_e = 0;

    if (e0 >= -1*(SUN_RADIUS + atmos_refract))
        del_e = (pressure / 1010.0) * (283.0 / (273.0 + temperature)) *
                1.02 / (60.0 * tan(deg2rad(e0 + 10.3/(e0 + 5.11))));

    return del_e;
}

double topocentric_elevation_angle_corrected(double e0, double delta_e)
{
    return e0 + delta_e;
}

double topocentric_zenith_angle(double e)
{
    return 90.0 - e;
}

double topocentric_azimuth_angle_neg180_180(double h_prime, double latitude, double delta_prime)
{
    double h_prime_rad = deg2rad(h_prime);
    double lat_rad     = deg2rad(latitude);

    return rad2deg(atan2(sin(h_prime_rad),
                         cos(h_prime_rad)*sin(lat_rad) -
                         tan(deg2rad(           ))*cos(lat_rad)));
}

double topocentric_azimuth_angle_zero_360(double azimuth180)
{
    return azimuth180 + 180.0;
}

double surface_incidence_angle(double zenith, double azimuth180, double azm_rotation,
                                                                  double slope)
{
    double zenith_rad = deg2rad(zenith);
    double slope_rad  = deg2rad(slope);

    return rad2deg(acos(cos(zenith_rad)*cos(slope_rad)  +
                        sin(slope_rad  *sin(zenith_rad) * cos(deg2rad(azimuth180 -
                        azm_rotation)))));
}

double sun_mean_longitude(double jme)
{
    return limit_degrees(280.4664567 + jme*(360007.6982779 + jme*(0.03032028 +
```

```
                    jme*(1/49931.0    + jme*(-1/15300.0     + jme*(-1/2000000.0))))));
}

double eot(double m, double alpha, double del_psi, double epsilon)
{
    return limit_minutes(4.0*(m - 0.0057183 - alpha + del_psi*cos(deg2rad(epsilon))));
}

double approx_sun_transit_time(double alpha_zero, double longitude, double nu)
{
    return (alpha_zero - longitude - nu) / 360.0;
}

double sun_hour_angle_at_rise_set(double latitude, double delta_zero, double h0_prime)
{
    double h0             = -99999;
    double latitude_rad   = deg2rad(latitude);
    double delta_zero_rad = deg2rad(delta_zero);
    double argument       = (sin(deg2rad(h0_prime)) - sin(latitude_rad)*sin(delta_zero_rad)) /
                                                      (cos(latitude_rad)*cos(delta_zero_rad));

    if (fabs(argument) <= 1) h0 = limit_degrees180(rad2deg(acos(argument)));

    return h0;
}

void approx_sun_rise_and_set(double *m_rts, double h0)
{
    double h0_dfrac = h0/360.0;

    m_rts[SUN_RISE]    = limit_zero2one(m_rts[SUN_TRANSIT] - h0_dfrac);
    m_rts[SUN_SET]     = limit_zero2one(m_rts[SUN_TRANSIT] + h0_dfrac);
    m_rts[SUN_TRANSIT] = limit_zero2one(m_rts[SUN_TRANSIT]);
}

double rts_alpha_delta_prime(double *ad, double n)
{
    double a = ad[JD_ZERO] - ad[JD_MINUS];
    double b = ad[JD_PLUS] - ad[JD_ZERO];

    if (fabs(a) >= 2.0) a = limit_zero2one(a);
    if (fabs(b) >= 2.0) b = limit_zero2one(b);

    return ad[JD_ZERO] + n * (a + b + (b-a)*n)/2.0;
}

double rts_sun_altitude(double latitude, double delta_prime, double h_prime)
{
    double latitude_rad    = deg2rad(latitude);
    double delta_prime_rad = deg2rad(delta_prime);

    return rad2deg(asin(sin(latitude_rad)*sin(delta_prime_rad) +
                        cos(latitude_rad)*cos(delta_prime_rad)*cos(deg2rad(h_prime))));
}

double sun_rise_and_set(double *m_rts,   double *h_rts,   double *delta_prime, double latitude,
                        double *h_prime, double h0_prime, int sun)
{
    return m_rts[sun] + (h_rts[sun] - h0_prime) /

          (360.0*cos(deg2rad(delta_prime[sun]))*cos(deg2rad(latitude))
          *sin(deg2rad(h_prime[sun])));
}

////////////////////////////////////////////////////////////////////////////////////////////
// Calculate required SPA parameters to get the right ascension (alpha) and declination (delta)
// Note: JD must be already calculated and in structure
////////////////////////////////////////////////////////////////////////////////////////////
void calculate_geocentric_sun_right_ascension_and_declination(spa_data *spa)
{
    double x[TERM_X_COUNT];

    spa->jc = julian_century(spa->jd);

    spa->jde = julian_ephemeris_day(spa->jd, spa->delta_t);
    spa->jce =                          (   ->jde);
```

```
    spa->jme = julian_ephemeris_millennium(spa->jce);

    spa->l = earth_heliocentric_longitude(spa->jme);
    spa->b = earth_heliocentric_latitude(spa->jme);
    spa->r = earth_radius_vector(spa->jme);

    spa->theta = geocentric_longitude(spa->l);
    spa->beta  = geocentric_latitude(spa->b);

    x[TERM_X0] = spa->x0 = mean elongation moon sun(spa->jce);
    x[TERM_X1] = spa->x1 = mean anomaly sun(spa->jce);
    x[TERM_X2] = spa->x2 = mean_anomaly_moon(spa->jce);
    x[TERM_X3] = spa->x3 = argument_latitude_moon(spa->jce);
    x[TERM_X4] = spa->x4 = ascending_longitude_moon(spa->jce);

    nutation_longitude_and_obliquity(spa->jce, x, &(spa->del_psi), &(spa->del_epsilon));

    spa->epsilon0 = ecliptic_mean_obliquity(spa->jme);
    spa->epsilon  = ecliptic_true_obliquity(spa->del_epsilon, spa->epsilon0);

    spa->del_tau   = aberration_correction(spa->r);
    spa->lamda     = apparent_sun_longitude(spa->theta, spa->del_psi, spa->del_tau);
    spa->nu0       = greenwich_mean_sidereal_time (spa->jd, spa->jc);
    spa->nu        = greenwich_sidereal_time (spa->nu0, spa->del_psi, spa->epsilon);

    spa->alpha = geocentric_sun_right_ascension(spa->lamda, spa->epsilon, spa->beta);
    spa->delta = geocentric_sun_declination(spa->beta, spa->epsilon, spa->lamda);
}

/////////////////////////////////////////////////////////////////////////
// Calculate Equation of Time (EOT) and Sun Rise, Transit, & Set (RTS)
/////////////////////////////////////////////////////////////////////////

void calculate_eot_and_sun_rise_transit_set(spa_data *spa)
{
    spa_data sun_rts = *spa;
    double nu, m, h0, n;
    double alpha[JD_COUNT], delta[JD_COUNT];
    double m_rts[SUN_COUNT], nu_rts[SUN_COUNT], h_rts[SUN_COUNT];
    double alpha_prime[SUN_COUNT], delta_prime[SUN_COUNT], h_prime[SUN_COUNT];
    double h0_prime = -1*(SUN_RADIUS + spa->atmos_refract);
    int i;


    m        = sun_mean_longitude(spa->jme);
    spa->eot = eot(m, spa->alpha, spa->del_psi, spa->epsilon);

    sun_rts.hour = sun_rts.minute = sun_rts.second = sun_rts.timezone = 0;

    sun_rts.jd = julian_day (sun_rts.year, sun_rts.month,  sun_rts.day,
                             sun_rts.hour, sun_rts.minute, sun_rts.second, sun_rts.timezone);

    calculate_geocentric_sun_right_ascension_and_declination(&sun_rts);
    nu = sun_rts.nu;

    sun_rts.delta_t = 0;
    sun_rts.jd--;
    for (i = 0; i < JD_COUNT; i++) {
        calculate_geocentric_sun_right_ascension_and_declination(&sun_rts);
        alpha[i] = sun_rts.alpha;
        delta[i] = sun_rts.delta;
        sun_rts.jd++;
    }

    m_rts[SUN_TRANSIT] = approx_sun_transit_time(alpha[JD_ZERO], spa->longitude, nu);
    h0 =                            (spa->latitude, delta[JD_ZERO], h0_prime ;

    if (h0 >= 0) {

        approx_sun_rise_and_set(m_rts, h0);

        for (i = 0; i < SUN_COUNT; i++) {

            nu_rts[i]      = nu + 360.985647*m_rts[i];

            n              = m_rts[i] + spa->delta_t/86400.0;
```

```
            alpha_prime[i] = rts_alpha_delta_prime(alpha, n);
            delta_prime[i] = rts_alpha_delta_prime(delta, n);

            h_prime[i]     = limit_degrees180pm(nu_rts[i] + spa->longitude - alpha_prime[i]);

            h_rts[i]       = rts_sun_altitude(spa->latitude, delta_prime[i], h_prime[i]);
        }

        spa->srha = h_prime[SUN_RISE];
        spa->ssha = h_prime[SUN_SET];
        spa->sta  = h_rts[SUN_TRANSIT];

        spa->suntransit = dayfrac_to_local_hr(m_rts[SUN_TRANSIT] - h_prime[SUN_TRANSIT] / 360.0,
                                              spa->timezone);

        spa->sunrise = dayfrac_to_local_hr(sun_rise_and_set(m_rts, h_rts, delta_prime,
                        spa->latitude, h_prime, h0_prime, SUN_RISE), spa->timezone);

        spa->sunset  = dayfrac_to_local_hr(sun_rise_and_set(m_rts, h_rts, delta_prime,
                        spa->latitude, h_prime, h0_prime, SUN_SET),  spa->timezone);

    } else spa->srha= spa->ssha= spa->sta= spa->suntransit= spa->sunrise= spa->sunset= -99999;

}

//////////////////////////////////////////////////////////////////////////////////////////
// Calculate all SPA parameters and put into structure
// Note: All inputs values (listed in header file) must already be in structure
//////////////////////////////////////////////////////////////////////////////////////////
int spa_calculate(spa_data *spa)
{
    int result;

    result = validate_inputs(spa);

    if (result == 0)
    {
        spa->jd = julian_day (spa->year, spa->month,  spa->day,
                              spa->hour, spa->minute, spa->second, spa->timezone);

        calculate_geocentric_sun_right_ascension_and_declination(spa);

        spa->h  = observer hour angle(spa->nu, spa->longitude, spa->alpha);
        spa->xi = sun_equatorial_horizontal_parallax(spa->r);

        sun_right_ascension_parallax_and_topocentric_dec(spa->latitude, spa->elevation, spa->xi,
                                  spa->h, spa->delta, &(spa->del_alpha), &(spa->delta_prime));

        spa->alpha_prime = topocentric_sun_right_ascension(spa->alpha, spa->del_alpha);
        spa->h_prime     = topocentric_local_hour_angle(spa->h, spa->del_alpha);

        spa->e0      = topocentric_elevation_angle(spa->latitude, spa->delta_prime,
        spa->h_prime);
        spa->del_e   = atmospheric_refraction_correction(spa->pressure, spa->temperature,
                                                         spa->atmos_refract, spa->e0);
        spa->e       = topocentric_elevation_angle_corrected(spa->e0, spa->del_e);

        spa->zenith     = topocentric_zenith_angle(spa->e);
        spa->azimuth180 = topocentric_azimuth_angle_neg180_180(spa->h_prime, spa->latitude,
                                                               spa->delta_prime);
        spa->azimuth    = topocentric_azimuth_angle_zero_360(spa->azimuth180);

        if ((spa->function == SPA_ZA_INC) || (spa->function == SPA_ALL))
            spa->incidence  = surface_incidence_angle(spa->zenith, spa->azimuth180,
                                                      spa->            , spa->slope);

        if ((spa->function == SPA ZA RTS) || (spa->function == SPA_ALL))
                  _   _   _   _    _       _    (spa);
    }

    return result;
}
//////////////////////////////////////////////////////////////////////////////////////////
```

A.6. C Code: SPA header file (SPA.h)

```
//////////////////////////////////////////
//          HEADER FILE for SPA.C       //
//                                      //
//      Solar Position Algorithm (SPA)  //
//                  for                 //
//        Solar Radiation Application   //
//                                      //
//               May 12, 2003           //
//                                      //
//   Filename: SPA.H                    //
//                                      //
//   Afshin Michael Andreas             //
//   afshin_andreas@nrel.gov (303)384-6383 //
//                                      //
//   Measurement & Instrumentation Team //
//   Solar Radiation Research Laboratory //
//   National Renewable Energy Laboratory //
//   1617 Cole Blvd, Golden, CO 80401   //
//////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
//                                                                      //
// Usage:                                                               //
//                                                                      //
//   1) In calling program, include this header file,                   //
//      by adding this line to the top of file:                         //
//           #include "spa.h"                                          //
//                                                                      //
//   2) In calling program, declare the SPA structure:                  //
//           spa_data spa;                                              //
//                                                                      //
//   3) Enter the required input values into SPA structure              //
//      (input values listed in comments below)                         //
//                                                                      //
//   4) Call the SPA calculate function and pass the SPA structure      //
//      (prototype is declared at the end of this header file):         //
//           spa_calculate(&spa);                                       //
//                                                                      //
//   Selected output values (listed in comments below) will be          //
//   computed and returned in the passed SPA structure.  Output         //
//   will based on function code selected from enumeration below.       //
//                                                                      //
//   Note: A non-zero return code from spa_calculate() indicates that   //
//         one of the input values did not pass simple bounds tests.    //
//         The valid input ranges and return error codes are also       //
//         listed below.                                                //
//////////////////////////////////////////////////////////////////////////

#ifndef __solar_position_algorithm_header
#define __solar_position_algorithm_header

//enumeration for function codes to select desired final outputs from SPA
enum {
    SPA_ZA            //calculate zenith and azimuth
    SPA_ZA_INC        //calculate zenith, azimuth, and incidence
    SPA_ZA_RTS        //calculate zenith, azimuth, and sun rise/transit/set values
    SPA_ALL           //calculate all SPA output values
};

typedef struct
{
    //----------------------INPUT VALUES------------------------

    int year;            // 4-digit year,    valid range: -2000 to 6000, error code: 1
    int month;           // 2-digit month,         valid range: 1 to 12, error code: 2
    int day;             // 2-digit day,           valid range: 1 to 31, error code: 3
    int hour;            // Observer local hour,   valid range: 0 to 24, error code: 4
    int minute;          // Observer local minute, valid range: 0 to 59, error code: 5
    int second           // Observer local second, valid range: 0 to 59, error code: 6
    double delta_t;      // Difference between earth rotation time and terrestrial time
                         // It is derived from observation only and is reported in this
                         // bulletin: http://maia.usno.navy.mil/ser7/ser7.dat,
                         // where delta_t = 32.184 + (TAI-UTC) + DUT1
                         // valid range: -8000 to 8000 seconds, error code: 7
    double timezone;     // Observer time zone (negative west of Greenwich)
```

```
                        // valid range: -12   to   12 hours,   error code: 8
    double longitude;   // Observer longitude (negative west of Greenwich)
                        // valid range: -180  to  180 degrees, error code: 9
    double latitude;    // Observer latitude (negative south of equator)
                        // valid range: -90   to   90 degrees, error code: 10
    double elevation;   // Observer elevation [meters]
                        // valid range: -6500000 or higher meters,    error code: 11
    double pressure;    // Annual average local pressure [millibars]
                        // valid range:    0 to 5000 millibars,       error code: 12
    double temperature; // Annual average local temperature [degrees Celsius]
                        // valid range: -273 to 6000 degrees Celsius, error code; 13
    double slope;       // Surface slope (measured from the horizontal plane)
                        // valid range: -360 to 360 degrees, error code: 14
    double azm_rotation; // Surface azimuth rotation (measured from south to projection of
                        //     surface normal on horizontal plane, negative west)
                        // valid range: -360 to 360 degrees, error code: 15
    double atmos_refract;// Atmospheric refraction at sunrise and sunset (0.5667 deg is typical)
                        // valid range: -5   to   5 degrees, error code: 16
    int function;       // Switch to choose functions for desired output (from enumeration)

    //-----------------Intermediate OUTPUT VALUES--------------------

    double jd;          //Julian day
    double jc;          //Julian century
    double jde;         //Julian ephemeris day
    double jce;         //Julian ephemeris century
    double jme;         //Julian ephemeris millennium
    double l;           //earth heliocentric longitude [degrees]
    double b;           //earth heliocentric latitude [degrees]
    double r;           //earth radius vector [Astronomical Units, AU]
    double theta;       //geocentric longitude [degrees]
    double beta;        //geocentric latitude [degrees]
    double x0;          //mean elongation (moon-sun) [degrees]
    double x1;          //mean anomaly (sun) [degrees]
    double x2;          //mean anomaly (moon) [degrees]
    double x3;          //argument latitude (moon) [degrees]
    double x4;          //ascending longitude (moon) [degrees]
    double del_psi;     //nutation longitude [degrees]
    double del_epsilon; //nutation obliquity [degrees]
    double epsilon0;    //ecliptic mean obliquity [arc seconds]
    double epsilon;     //ecliptic true obliquity  [degrees]
    double del tau;     //aberration correction [degrees]
    double lamda;       //apparent sun longitude [degrees]
    double nu0;         //Greenwich mean sidereal time [degrees]
    double nu;          //Greenwich sidereal time [degrees]
    double alpha;       //geocentric sun right ascension [degrees]
    double delta;       //geocentric sun declination [degrees]
    double h;           //observer hour angle [degrees]
    double xi;          //sun equatorial horizontal parallax [degrees]
    double del_alpha;   //sun right ascension parallax [degrees]
    double delta prime; //topocentric sun declination [degrees]
    double alpha prime; //topocentric sun right ascension [degrees]
    double h_prime;     //topocentric local hour angle [degrees]
    double e0;          //topocentric elevation angle (uncorrected) [degrees]
    double del_e;       //atmospheric refraction correction [degrees]
    double e;           //topocentric elevation angle (corrected) [degrees]
    double eot;         //equation of time [minutes]
    double srha;        //sunrise hour angle [degrees]
    double ssha;        //sunset hour angle [degrees]
    double sta;         //sun transit altitude [degrees]

    //----------------------Final OUTPUT VALUES------------------------

    double zenith;      //topocentric zenith angle [degrees]
    double azimuth180;  //topocentric azimuth angle (westward from south) [-180 to 180 degrees]
    double azimuth;     //topocentric azimuth angle (eastward from north) [   0 to 360 degrees]
    double incidence;   //surface incidence angle [degrees]
    double suntransit;  //local sun transit time (or solar noon) [fractional hour]
    double sunrise;     //local sunrise time (+/- 30 seconds) [fractional hour]
    double sunset;      //local sunset time (+/- 30 seconds) [fractional hour]
} spa_data;

//Calculate SPA output values (in structure) based on input values passed in structure
int spa_calculate(spa_data *spa);

#endif
```

A.7. C Code: SPA source file (SPA.c)

```
//////////////////////////////////////////
//      Solar Position Algorithm (SPA)     //
//                   for                   //
//        Solar Radiation Application      //
//                                         //
//               May 12, 2003              //
//                                         //
//   Filename: SPA.C                       //
//                                         //
//   Afshin Michael Andreas                //
//   afshin_andreas@nrel.gov (303)384-6383 //
//                                         //
//   Measurement & Instrumentation Team    //
//   Solar Radiation Research Laboratory   //
//   National Renewable Energy Laboratory  //
//   1617 Cole Blvd, Golden, CO 80401      //
//////////////////////////////////////////

//////////////////////////////////////////
//   See the SPA.H header file for usage   //
//                                         //
//   This code is based on the NREL        //
//   technical report "Solar Position      //
//   Algorithm for Solar Radiation         //
//   Application" by I. Reda & A. Andreas  //
//////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////////////
//
//   NOTICE
//
//This solar position algorithm for solar radiation applications (the "data") was produced by
//the National Renewable Energy Laboratory ("NREL"), which is operated by the Midwest Research
//Institute ("MRI") under Contract No. DE-AC36-99-GO10337 with the U.S. Department of Energy
//(the "Government").
//
//Reference herein, directly or indirectly to any specific commercial product, process, or
//service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply
//its endorsement, recommendation, or favoring by the Government, MRI or NREL.
//
//THESE DATA ARE PROVIDED "AS IS" AND NEITHER THE GOVERNMENT, MRI, NREL NOR ANY OF THEIR
//EMPLOYEES, MAKES ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING THE WARRANTIES OF MERCHANTABILITY
//AND FITNESS FOR A PARTICULAR PURPOSE, OR ASSUMES ANY LEGAL LIABILITY OR RESPONSIBILITY FOR THE
//ACCURACY, COMPLETENESS, OR USEFULNESS OF ANY SUCH INFORMATION DISCLOSED IN THE ALGORITHM, OR
//OF ANY APPARATUS, PRODUCT, OR PROCESS DISCLOSED, OR REPRESENTS THAT ITS USE WOULD NOT INFRINGE
//PRIVATELY OWNED RIGHTS.
//
//////////////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////////////
// Revised 27-FEB-2004 Andreas
//         Added bounds check on inputs and return value for spa_calculate().
// Revised 10-MAY-2004 Andreas
//         Changed temperature bound check minimum from -273.15 to -273 degrees C.
// Revised 17-JUN-2004 Andreas
//         Corrected a problem that caused a bogus sunrise/set/transit on the equinox.
// Revised 18-JUN-2004 Andreas
//         Added a "function" input variable that allows the selecting of desired outputs.
// Revised 21-JUN-2004 Andreas
//         Added 3 new intermediate output values to SPA structure (srha, ssha, & sta).
// Revised 23-JUN-2004 Andreas
//         Enumerations for "function" were renamed and 2 were added.
//         Prevented bound checks on inputs that are not used (based on function).
// Revised 01-SEP-2004 Andreas
//         Changed a local variable from integer to double.
// Revised 12-JUL-2005 Andreas
//         Put a limit on the EOT calculation, so that the result is between -20 and 20.
// Revised 26-OCT-2005 Andreas
//         Set the atmos. refraction correction to zero, when sun is below horizon.
//         Made atmos_refract input a requirement for all "functions".
//         Changed atmos_refract bound check from +/- 10 to +/- 5 degrees.
// Revised 07-NOV-2006 Andreas
//         Corrected 3 earth periodic terms in the L_TERMS array.
//         Corrected 2 earth periodic terms in the R_TERMS array.
```

```
// Revised 10-NOV-2006 Andreas
//         Corrected a constant used to calculate topocentric sun declination.
//         Put a limit on observer hour angle, so result is between 0 and 360.
// Revised 13-NOV-2006 Andreas
//         Corrected calculation of topocentric sun declination.
//         Converted all floating point inputs in spa structure to doubles.
// Revised 27-FEB-2007 Andreas
//         Minor correction made as to when atmos. refraction correction is set to zero.
//////////////////////////////////////////////////////////////////////////////////////////

#include <math.h>
#include "spa.h"

#define PI         3.14159265358979323846264338327950288419 71
#define SUN_RADIUS 0.26667

#define L_COUNT 6
#define B_COUNT 2
#define R_COUNT 5
#define Y_COUNT 63

#define L_MAX_SUBCOUNT 64
#define B_MAX_SUBCOUNT 5
#define R_MAX_SUBCOUNT 40

enum {TERM_A, TERM_B, TERM_C, TERM_COUNT};
enum {TERM_X0, TERM_X1, TERM_X2, TERM_X3, TERM_X4, TERM_X_COUNT};
enum {TERM_PSI_A, TERM_PSI_B, TERM_EPS_C, TERM_EPS_D, TERM_PE_COUNT};
enum {JD_MINUS, JD_ZERO, JD_PLUS, JD_COUNT};
enum {SUN_TRANSIT, SUN_RISE, SUN_SET, SUN_COUNT};

#define TERM_Y_COUNT TERM_X_COUNT

const int l_subcount[L_COUNT] = {64,34,20,7,3,1};
const int b_subcount[B_COUNT] = {5,2};
const int r_subcount[R_COUNT] = {40,10,6,2,1};

/////////////////////////////////////////////////
///  Earth Periodic Terms
/////////////////////////////////////////////////
const double L_TERMS[L_COUNT][L_MAX_SUBCOUNT][TERM_COUNT]=
{
    {
        {175347046.0,0,0},
        {3341656.0,4.6692568,6283.07585},
        {34894.0,4.6261,12566.1517},
        {3497.0,2.7441,5753.3849},
        {3418.0,2.8289,3.5231},
        {3136.0,3.6277,77713.7715},
        {2676.0,4.4181,7860.4194},
        {2343.0,6.1352,3930.2097},
        {1324.0,0.7425,11506.7698},
        {1273.0,2.0371,529.691},
        {1199.0,1.1096,1577.3435},
        {990,5.233,5884.927},
        {902,2.045,26.298},
        {857,3.508,398.149},
        {780,1.179,5223.694},
        {753,2.533,5507.553},
        {505,4.583,18849.228},
        {492,4.205,775.523},
        {357,2.92,0.067},
        {317,5.849,11790.629},
        {284,1.899,796.298},
        {271,0.315,10977.079},
        {243,0.345,5486.778},
        {206,4.806,2544.314},
        {205,1.869,5573.143},
        {202,2.458,6069.777},
        {156,0.833,213.299},
        {132,3.411,2942.463},
        {126,1.083,20.775},
        {115,0.645,0.98},
        {103,0.636,4694.003},
        {102,0.976,15720.839},
        {102,4.267,7.114},
```

```
    {99,6.21,2146.17},
    {98,0.68,155.42},
    {86,5.98,161000.69},
    {85,1.3,6275.96},
    {85,3.67,71430.7},
    {80,1.81,17260.15},
    {79,3.04,12036.46},
    {75,1.76,5088.63},
    {74,3.5,3154.69},
    {74,4.68,801.82},
    {70,0.83,9437.76},
    {62,3.98,8827.39},
    {61,1.82,7084.9},
    {57,2.78,6286.6},
    {56,4.39,14143.5},
    {56,3.47,6279.55},
    {52,0.19,12139.55},
    {52,1.33,1748.02},
    {51,0.28,5856.48},
    {49,0.49,1194.45},
    {41,5.37,8429.24},
    {41,2.4,19651.05},
    {39,6.17,10447.39},
    {37,6.04,10213.29},
    {37,2.57,1059.38},
    {36,1.71,2352.87},
    {36,1.78,6812.77},
    {33,0.59,17789.85},
    {30,0.44,83996.85},
    {30,2.74,1349.87},
    {25,3.16,4690.48}
},
{
    {628331966747.0,0,0},
    {206059.0,2.678235,6283.07585},
    {4303.0,2.6351,12566.1517},
    {425.0,1.59,3.523},
    {119.0,5.796,26.298},
    {109.0,2.966,1577.344},
    {93,2.59,18849.23},
    {72,1.14,529.69},
    {68,1.87,398.15},
    {67,4.41,5507.55},
    {59,2.89,5223.69},
    {56,2.17,155.42},
    {45,0.4,796.3},
    {36,0.47,775.52},
    {29,2.65,7.11},
    {21,5.34,0.98},
    {19,1.85,5486.78},
    {19,4.97,213.3},
    {17,2.99,6275.96},
    {16,0.03,2544.31},
    {16,1.43,2146.17},
    {15,1.21,10977.08},
    {12,2.83,1748.02},
    {12,3.26,5088.63},
    {12,5.27,1194.45},
    {12,2.08,4694},
    {11,0.77,553.57},
    {10,1.3,6286.6},
    {10,4.24,1349.87},
    {9,2.7,242.73},
    {9,5.64,951.72},
    {8,5.3,2352.87},
    {6,2.65,9437.76},
    {6,4.67,4690.48}
},
{
    {52919.0,0,0},
    {8720.0,1.0721,6283.0758},
    {309.0,0.867,12566.152},
    {27,0.05,3.52},
    {16,5.19,26.3},
    {16,3.68,155.42},
    {10,0.76,18849.23},
```

```
        {9,2.06,77713.77},
        {7,0.83,775.52},
        {5,4.66,1577.34},
        {4,1.03,7.11},
        {4,3.44,5573.14},
        {3,5.14,796.3},
        {3,6.05,5507.55},
        {3,1.19,242.73},
        {3,6.12,529.69},
        {3,0.31,398.15},
        {3,2.28,553.57},
        {2,4.38,5223.69},
        {2,3.75,0.98}
    },
    {
        {289.0,5.844,6283.076},
        {35,0,0},
        {17,5.49,12566.15},
        {3,5.2,155.42},
        {1,4.72,3.52},
        {1,5.3,18849.23},
        {1,5.97,242.73}
    },
    {
        {114.0,3.142,0},
        {8,4.13,6283.08},
        {1,3.84,12566.15}
    },
    {
        {1,3.14,0}
    }
};

const double B_TERMS[B_COUNT][B_MAX_SUBCOUNT][TERM_COUNT]=
{
    {
        {280.0,3.199,84334.662},
        {102.0,5.422,5507.553},
        {80,3.88,5223.69},
        {44,3.7,2352.87},
        {32,4,1577.34}
    },
    {
        {9,3.9,5507.55},
        {6,1.73,5223.69}
    }
};

const double R_TERMS[R_COUNT][R_MAX_SUBCOUNT][TERM_COUNT]=
{
    {
        {100013989.0,0,0},
        {1670700.0,3.0984635,6283.07585},
        {13956.0,3.05525,12566.1517},
        {3084.0,5.1985,77713.7715},
        {1628.0,1.1739,5753.3849},
        {1576.0,2.8469,7860.4194},
        {925.0,5.453,11506.77},
        {542.0,4.564,3930.21},
        {472.0,3.661,5884.927},
        {346.0,0.964,5507.553},
        {329.0,5.9,5223.694},
        {307.0,0.299,5573.143},
        {243.0,4.273,11790.629},
        {212.0,5.847,1577.344},
        {186.0,5.022,10977.079},
        {175.0,3.012,18849.228},
        {110.0,5.055,5486.778},
        {98,0.89,6069.78},
        {86,5.69,15720.84},
        {86,1.27,161000.69},
        {65,0.27,17260.15},
        {63,0.92,529.69},
        {57,2.01,83996.85},
        {56,5.24,71430.7},
        {49,3.25,2544.31},
```

```
        {47,2.58,775.52},
        {45,5.54,9437.76},
        {43,6.01,6275.96},
        {39,5.36,4694},
        {38,2.39,8827.39},
        {37,0.83,19651.05},
        {37,4.9,12139.55},
        {36,1.67,12036.46},
        {35,1.84,2942.46},
        {33,0.24,7084.9},
        {32,0.18,5088.63},
        {32,1.78,398.15},
        {28,1.21,6286.6},
        {28,1.9,6279.55},
        {26,4.59,10447.39}
    },
    {
        {103019.0,1.10749,6283.07585},
        {1721.0,1.0644,12566.1517},
        {702.0,3.142,0},
        {32,1.02,18849.23},
        {31,2.84,5507.55},
        {25,1.32,5223.69},
        {18,1.42,1577.34},
        {10,5.91,10977.08},
        {9,1.42,6275.96},
        {9,0.27,5486.78}
    },
    {
        {4359.0,5.7846,6283.0758},
        {124.0,5.579,12566.152},
        {12,3.14,0},
        {9,3.63,77713.77},
        {6,1.87,5573.14},
        {3,5.47,18849.23}
    },
    {
        {145.0,4.273,6283.076},
        {7,3.92,12566.15}
    },
    {
        {4,2.56,6283.08}
    }
};

//////////////////////////////////////////////////////////////////
///  Periodic Terms for the nutation in longitude and obliquity
//////////////////////////////////////////////////////////////////

const int Y_TERMS[Y_COUNT][TERM_Y_COUNT]=
{
    {0,0,0,0,1},
    {-2,0,0,2,2},
    {0,0,0,2,2},
    {0,0,0,0,2},
    {0,1,0,0,0},
    {0,0,1,0,0},
    {-2,1,0,2,2},
    {0,0,0,2,1},
    {0,0,1,2,2},
    {-2,-1,0,2,2},
    {-2,0,1,0,0},
    {-2,0,0,2,1},
    {0,0,-1,2,2},
    {2,0,0,0,0},
    {0,0,1,0,1},
    {2,0,-1,2,2},
    {0,0,-1,0,1},
    {0,0,1,2,1},
    {-2,0,2,0,0},
    {0,0,-2,2,1},
    {2,0,0,2,2},
    {0,0,2,2,2},
    {0,0,2,0,0},
    {-2,0,1,2,2},
    {0,0,0,2,0},
```

```
    {-2,0,0,2,0},
    {0,0,-1,2,1},
    {0,2,0,0,0},
    {2,0,-1,0,1},
    {-2,2,0,2,2},
    {0,1,0,0,1},
    {-2,0,1,0,1},
    {0,-1,0,0,1},
    {0,0,2,-2,0},
    {2,0,-1,2,1},
    {2,0,1,2,2},
    {0,1,0,2,2},
    {-2,1,1,0,0},
    {0,-1,0,2,2},
    {2,0,0,2,1},
    {2,0,1,0,0},
    {-2,0,2,2,2},
    {-2,0,1,2,1},
    {2,0,-2,0,1},
    {2,0,0,0,1},
    {0,-1,1,0,0},
    {-2,-1,0,2,1},
    {-2,0,0,0,1},
    {0,0,2,2,1},
    {-2,0,2,0,1},
    {-2,1,0,2,1},
    {0,0,1,-2,0},
    {-1,0,1,0,0},
    {-2,1,0,0,0},
    {1,0,0,0,0},
    {0,0,1,2,0},
    {0,0,-2,2,2},
    {-1,-1,1,0,0},
    {0,1,1,0,0},
    {0,-1,1,2,2},
    {2,-1,-1,2,2},
    {0,0,3,2,2},
    {2,-1,0,2,2},
};

const double PE_TERMS[Y_COUNT][TERM_PE_COUNT]={
    {-171996,-174.2,92025,8.9},
    {-13187,-1.6,5736,-3.1},
    {-2274,-0.2,977,-0.5},
    {2062,0.2,-895,0.5},
    {1426,-3.4,54,-0.1},
    {712,0.1,-7,0},
    {-517,1.2,224,-0.6},
    {-386,-0.4,200,0},
    {-301,0,129,-0.1},
    {217,-0.5,-95,0.3},
    {-158,0,0,0},
    {129,0.1,-70,0},
    {123,0,-53,0},
    {63,0,0,0},
    {63,0.1,-33,0},
    {-59,0,26,0},
    {-58,-0.1,32,0},
    {-51,0,27,0},
    {48,0,0,0},
    {46,0,-24,0},
    {-38,0,16,0},
    {-31,0,13,0},
    {29,0,0,0},
    {29,0,-12,0},
    {26,0,0,0},
    {-22,0,0,0},
    {21,0,-10,0},
    {17,-0.1,0,0},
    {16,0,-8,0},
    {-16,0.1,7,0},
    {-15,0,9,0},
    {-13,0,7,0},
    {-12,0,6,0},
    {11,0,0,0},
    {-10,0,5,0},
```

```
    {-8,0,3,0},
    {7,0,-3,0},
    {-7,0,0,0},
    {-7,0,3,0},
    {-7,0,3,0},
    {6,0,0,0},
    {6,0,-3,0},
    {6,0,-3,0},
    {-6,0,3,0},
    {-6,0,3,0},
    {5,0,0,0},
    {-5,0,3,0},
    {-5,0,3,0},
    {-5,0,3,0},
    {4,0,0,0},
    {4,0,0,0},
    {4,0,0,0},
    {-4,0,0,0},
    {-4,0,0,0},
    {-4,0,0,0},
    {3,0,0,0},
    {-3,0,0,0},
    {-3,0,0,0},
    {-3,0,0,0},
    {-3,0,0,0},
    {-3,0,0,0},
    {-3,0,0,0},
    {-3,0,0,0},
};

//////////////////////////////////////////////

double rad2deg(double radians)
{
    return (180.0/PI)*radians;
}

double deg2rad(double degrees)
{
    return (PI/180.0)*degrees;
}

double limit_degrees(double degrees)
{
    double limited;

    degrees /= 360.0;
    limited = 360.0*(degrees-floor(degrees));
    if (limited < 0) limited += 360.0;

    return limited;
}

double limit_degrees180pm(double degrees)
{
    double limited;

    degrees /= 360.0;
    limited = 360.0*(degrees-floor(degrees));
    if      (limited < -180.0) limited += 360.0;
    else if (limited >  180.0) limited -= 360.0;

    return limited;
}

double limit_degrees180(double degrees)
{
    double limited;

    degrees /= 180.0;
    limited = 180.0*(degrees-floor(degrees));
    if (limited < 0) limited += 180.0;

    return limited;
}
```

```
double limit_zero2one(double value)
{
    double limited;

    limited = value - floor(value);
    if (limited < 0) limited += 1.0;

    return limited;
}

double limit_minutes(double minutes)
{
    double limited=minutes;

    if      (limited < -20.0) limited += 1440.0;
    else if (limited >  20.0) limited -= 1440.0;

    return limited;
}

double dayfrac_to_local_hr(double dayfrac, double timezone)
{
    return 24.0*limit_zero2one(dayfrac + timezone/24.0);
}

double third_order_polynomial(double a, double b, double c, double d, double x)
{
    return ((a*x + b)*x + c)*x + d;
}

////////////////////////////////////////////////////////////////////////////////////////
int validate_inputs(spa_data *spa)
{
    if ((spa->year        < -2000) || (spa->year        > 6000)) return 1;
    if ((spa->month       < 1    ) || (spa->month       > 12  )) return 2;
    if ((spa->day         < 1    ) || (spa->day         > 31  )) return 3;
    if ((spa->hour        < 0    ) || (spa->hour        > 24  )) return 4;
    if ((spa->minute      < 0    ) || (spa->minute      > 59  )) return 5;
    if ((spa->second      < 0    ) || (spa->second      > 59  )) return 6;
    if ((spa->pressure    < 0    ) || (spa->pressure    > 5000)) return 12;
    if ((spa->temperature <= -273) || (spa->temperature > 6000)) return 13;
    if ((spa->hour        == 24  ) && (spa->minute      > 0   )) return 5;
    if ((spa->hour        == 24  ) && (spa->second      > 0   )) return 6;

    if (fabs(spa->delta_t)       > 8000    ) return 7;
    if (fabs(spa->timezone)      > 12      ) return 8;
    if (fabs(spa->longitude)     > 180     ) return 9;
    if (fabs(spa->latitude)      > 90      ) return 10;
    if (fabs(spa->atmos_refract) > 5       ) return 16;
    if (     spa->elevation      < -6500000) return 11;

    if ((spa->function == SPA_ZA_INC) || (spa->function == SPA_ALL))
    {
        if (fabs(spa->slope)         > 360) return 14;
        if (fabs(spa->azm_rotation)  > 360) return 15;
    }

    return 0;
}
////////////////////////////////////////////////////////////////////////////////////////
double julian_day (int year, int month, int day, int hour, int minute, int second, double tz)
{
    double day_decimal, julian_day, a;

    day_decimal = day + (hour - tz + (minute + second/60.0)/60.0)/24.0;

    if (month < 3) {
        month += 12;
        year--;
    }

    julian_day = floor(365.25*(year+4716.0)) + floor(30.6001*(month+1)) + day_decimal - 1524.5;

    if (julian_day > 2299160.0) {
        a = floor(year/100);
        julian_day += (2 - a + floor(a/4));
```

```
    }

    return julian_day;
}

double julian_century(double jd)
{
    return (jd-2451545.0)/36525.0;
}

double julian_ephemeris_day(double jd, double delta_t)
{
    return jd+delta_t/86400.0;
}

double julian_ephemeris_century(double jde)
{
    return (jde - 2451545.0)/36525.0;
}

double julian_ephemeris_millennium(double jce)
{
    return (jce/10.0);
}

double earth_periodic_term_summation(const double terms[][TERM_COUNT], int count, double jme)
{
    int i;
    double sum=0;

    for (i = 0; i < count; i++)
        sum += terms[i][TERM_A]*cos(terms[i][TERM_B]+terms[i][TERM_C]*jme);

    return sum;
}

double earth_values(double term_sum[], int count, double jme)
{
    int i;
    double sum=0;

    for (i = 0; i < count; i++)
        sum += term_sum[i]*pow(jme, i);

    sum /= 1.0e8;

    return sum;
}

double earth_heliocentric_longitude(double jme)
{
    double sum[L_COUNT];
    int i;

    for (i = 0; i < L_COUNT; i++)
        sum[i] = earth_periodic_term_summation(L_TERMS[i], l_subcount[i], jme);

    return limit_degrees(rad2deg(earth_values(sum, L_COUNT, jme)));

}

double earth_heliocentric_latitude(double jme)
{
    double sum[B_COUNT];
    int i;

    for (i = 0; i < B_COUNT; i++)
        sum[i] = earth_periodic_term_summation(B_TERMS[i], b_subcount[i], jme);

    return rad2deg(earth_values(sum, B_COUNT, jme));

}

double earth_radius_vector(double jme)
{
    double sum[R_COUNT];
```

```
    int i;

    for (i = 0; i < R_COUNT; i++)
        sum[i] = earth_periodic_term_summation(R_TERMS[i], r_subcount[i], jme);

    return earth_values(sum, R_COUNT, jme);

}

double geocentric_longitude(double l)
{
    double theta = l + 180.0;

    if (theta >= 360.0) theta -= 360.0;

    return theta;
}

double geocentric_latitude(double b)
{
    return -b;
}

double mean_elongation_moon_sun(double jce)
{
    return third_order_polynomial(1.0/189474.0, -0.0019142, 445267.11148, 297.85036, jce);
}

double mean_anomaly_sun(double jce)
{
    return third_order_polynomial(-1.0/300000.0, -0.0001603, 35999.05034, 357.52772, jce);
}

double mean_anomaly_moon(double jce)
{
    return third_order_polynomial(1.0/56250.0, 0.0086972, 477198.867398, 134.96298, jce);
}

double argument_latitude_moon(double jce)
{
    return third_order_polynomial(1.0/327270.0, -0.0036825, 483202.017538, 93.27191, jce);
}

double ascending_longitude_moon(double jce)
{
    return third_order_polynomial(1.0/450000.0, 0.0020708, -1934.136261, 125.04452, jce);
}

double xy_term_summation(int i, double x[TERM_X_COUNT])
{
    int j;
    double sum=0;

    for (j = 0; j < TERM_Y_COUNT; j++)
        sum += x[j]*Y_TERMS[i][j];

    return sum;
}

void nutation_longitude_and_obliquity(double jce, double x[TERM_X_COUNT], double *del psi,
                                                                           double *del_epsilon)
{
    int i;
    double xy_term_sum, sum_psi=0, sum_epsilon=0;

    for (i = 0; i < Y_COUNT; i++) {
        xy term_sum  = deg2rad(xy term summation(i, x));
        sum psi     += (PE TERMS[i][TERM PSI A] + jce*PE TERMS[i][TERM PSI B])*sin(xy term sum);
        sum_epsilon += (PE_TERMS[i][TERM_EPS_C] + jce*PE_TERMS[i][TERM_EPS_D])*cos(xy_term_sum);
    }

    *del_psi     = sum_psi     / 36000000.0;
    *del_epsilon = sum_epsilon / 36000000.0;
}

double ecliptic_mean_obliquity(double jme)
```

```
{
    double u = jme/10.0;

    return 84381.448 + u*(-4680.96 + u*(-1.55 + u*(1999.25 + u*(-51.38 + u*(-249.67 +
                      u*(  -39.05 + u*( 7.12 + u*(  27.87 + u*(  5.79 + u*2.45)))))))));
}

double ecliptic_true_obliquity(double delta_epsilon, double epsilon0)
{
    return delta_epsilon + epsilon0/3600.0;
}

double aberration_correction(double r)
{
    return -20.4898 / (3600.0*r);
}

double apparent_sun_longitude(double theta, double delta_psi, double delta_tau)
{
    return theta + delta_psi + delta_tau;
}

double greenwich_mean_sidereal_time (double jd, double jc)
{
    return limit_degrees(280.46061837 + 360.98564736629 * (jd - 2451545.0) +
                                        jc*jc*(0.000387933 - jc/38710000.0));
}

double greenwich_sidereal_time (double nu0, double delta_psi, double epsilon)
{
    return nu0 + delta_psi*cos(deg2rad(epsilon));
}

double geocentric_sun_right_ascension(double lamda, double epsilon, double beta)
{
    double lamda_rad   = deg2rad(lamda);
    double epsilon_rad = deg2rad(epsilon);

    return limit_degrees(rad2deg(atan2(sin(lamda_rad)*cos(epsilon_rad) -
                                        tan(deg2rad(beta))*sin(epsilon_rad), cos(lamda_rad))));
}

double geocentric_sun_declination(double beta, double epsilon, double lamda)
{
    double beta_rad    = deg2rad(beta);
    double epsilon_rad = deg2rad(epsilon);

    return rad2deg(asin(sin(beta_rad)*cos(epsilon_rad) +
                        cos(beta_rad)*sin(epsilon_rad)*sin(deg2rad(lamda))));
}

double observer_hour_angle(double nu, double longitude, double alpha_deg)
{
    return limit_degrees(nu + longitude - alpha_deg);
}

double sun_equatorial_horizontal_parallax(double r)
{
    return 8.794 / (3600.0 * r);
}

void sun_right_ascension_parallax_and_topocentric_dec(double latitude, double elevation,
               double xi, double h, double delta, double *delta_alpha, double *delta_prime)
{
    double delta_alpha_rad;
    double lat_rad   = deg2rad(latitude);
    double xi_rad    = deg2rad(xi);
    double h_rad     = deg2rad(h);
    double delta_rad = deg2rad(delta);
    double u = atan(0.99664719 * tan(lat_rad));
    double y = 0.99664719 * sin(u) + elevation*sin(lat_rad)/6378140.0;
    double x =              cos(u) + elevation*cos(lat_rad)/6378140.0;

    delta_alpha_rad =      atan2(               - x*sin(xi_rad) *sin(h_rad),
                                 cos(delta_rad) - x*sin(xi_rad) *cos(h_rad));
```

```
    *delta_prime = rad2deg(atan2((sin(delta_rad) - y*sin(xi_rad))*cos(delta_alpha_rad),
                                 cos(delta_rad) - x*sin(xi_rad) *cos(h_rad)));

    *delta_alpha = rad2deg(delta_alpha_rad);
}

double topocentric_sun_right_ascension(double alpha_deg, double delta_alpha)
{
    return alpha_deg + delta_alpha;
}

double topocentric_local_hour_angle(double h, double delta_alpha)
{
    return h - delta_alpha;
}

double topocentric_elevation_angle(double latitude, double delta_prime, double h_prime)
{
    double lat_rad         = deg2rad(latitude);
    double delta_prime_rad = deg2rad(delta_prime);

    return rad2deg(asin(sin(lat_rad)*sin(delta_prime_rad) +
                        cos(lat_rad)*cos(delta_prime_rad) * cos(deg2rad(h_prime))));
}

double atmospheric_refraction_correction(double pressure, double temperature,
                                         double atmos_refract, double e0)
{
    double del_e = 0;

    if (e0 >= -1*(SUN_RADIUS + atmos_refract))
        del_e = (pressure / 1010.0) * (283.0 / (273.0 + temperature)) *
                1.02 / (60.0 * tan(deg2rad(e0 + 10.3/(e0 + 5.11))));

    return del_e;
}

double topocentric_elevation_angle_corrected(double e0, double delta_e)
{
    return e0 + delta_e;
}

double topocentric_zenith_angle(double e)
{
    return 90.0 - e;
}

double topocentric_azimuth_angle_neg180_180(double h_prime, double latitude, double delta_prime)
{
    double h_prime_rad = deg2rad(h_prime);
    double lat_rad     = deg2rad(latitude);

    return rad2deg(atan2(sin(h_prime_rad),
                         cos(h_prime_rad)*sin(lat_rad) -
                         tan(deg2rad(delta_prime))*cos(lat_rad)));
}

double topocentric_azimuth_angle_zero_360(double azimuth180)
{
    return azimuth180 + 180.0;
}

double surface_incidence_angle(double zenith, double azimuth180, double azm_rotation,
                                                                  double slope)
{
    double zenith_rad = deg2rad(zenith);
    double slope_rad  = deg2rad(slope);

    return rad2deg(acos(cos(zenith_rad)*cos(slope_rad)  +
                        sin(slope_rad )*sin(zenith_rad) * cos(deg2rad(azimuth180 -
                        azm_rotation))));
}

double sun_mean_longitude(double jme)
{
    return limit_degrees(280.4664567 + jme*(360007.6982779 + jme*(0.03032028 +
```

```
                    jme*(1/49931.0    + jme*(-1/15300.0     + jme*(-1/2000000.0))))));
}

double eot(double m, double alpha, double del_psi, double epsilon)
{
    return limit_minutes(4.0*(m - 0.0057183 - alpha + del_psi*cos(deg2rad(epsilon))));
}

double approx_sun_transit_time(double alpha_zero, double longitude, double nu)
{
    return (alpha_zero - longitude - nu) / 360.0;
}

double sun_hour_angle_at_rise_set(double latitude, double delta_zero, double h0_prime)
{
    double h0             = -99999;
    double latitude_rad   = deg2rad(latitude);
    double delta_zero_rad = deg2rad(delta_zero);
    double argument       = (sin(deg2rad(h0_prime)) - sin(latitude_rad)*sin(delta_zero_rad)) /
                                                      (cos(latitude_rad)*cos(delta_zero_rad));

    if (fabs(argument) <= 1) h0 = limit_degrees180(rad2deg(acos(argument)));

    return h0;
}

void approx_sun_rise_and_set(double *m_rts, double h0)
{
    double h0_dfrac = h0/360.0;

    m_rts[SUN_RISE]    = limit_zero2one(m_rts[SUN_TRANSIT] - h0_dfrac);
    m_rts[SUN_SET]     = limit_zero2one(m_rts[SUN_TRANSIT] + h0_dfrac);
    m_rts[SUN_TRANSIT] = limit_zero2one(m_rts[SUN_TRANSIT]);
}

double rts_alpha_delta_prime(double *ad, double n)
{
    double a = ad[JD_ZERO] - ad[JD_MINUS];
    double b = ad[JD_PLUS] - ad[JD_ZERO];

    if (fabs(a) >= 2.0) a = limit_zero2one(a);
    if (fabs(b) >= 2.0) b = limit_zero2one(b);

    return ad[JD_ZERO] + n * (a + b + (b-a)*n)/2.0;
}

double rts_sun_altitude(double latitude, double delta_prime, double h_prime)
{
    double latitude_rad    = deg2rad(latitude);
    double delta_prime_rad = deg2rad(delta_prime);

    return rad2deg(asin(sin(latitude rad)*sin(delta prime rad) +
                        cos(latitude_rad)*cos(delta_prime_rad)*cos(deg2rad(h_prime))));
}

double sun_rise_and_set(double *m_rts,   double *h_rts,   double *delta_prime, double latitude,
                        double *h_prime, double h0_prime, int sun)
{
    return m_rts[sun] + (h_rts[sun] - h0_prime) /

          (360.0*cos(deg2rad(delta prime[sun]))*cos(deg2rad(latitude))
          *sin(deg2rad(h_prime[sun])));
}

//////////////////////////////////////////////////////////////////////////////////////////
// Calculate required SPA parameters to get the right ascension (alpha) and declination (delta)
// Note: JD must be already calculated and in structure
//////////////////////////////////////////////////////////////////////////////////////////
void calculate_geocentric_sun_right_ascension_and_declination(spa_data *spa)
{
    double x[TERM_X_COUNT];

    spa->jc = julian_century(spa->jd);

    spa->jde = julian_ephemeris_day(spa->jd, spa->delta_t);
    spa->jce = julian_ephemeris_century(spa->jde);
```

```
    spa->jme = julian_ephemeris_millennium(spa->jce);

    spa->l = earth_heliocentric_longitude(spa->jme);
    spa->b = earth_heliocentric_latitude(spa->jme);
    spa->r = earth_radius_vector(spa->jme);

    spa->theta = geocentric_longitude(spa->l);
    spa->beta  = geocentric_latitude(spa->b);

    x[TERM_X0] = spa->x0 = mean_elongation_moon_sun(spa->jce);
    x[TERM_X1] = spa->x1 = mean_anomaly_sun(spa->jce);
    x[TERM_X2] = spa->x2 = mean_anomaly_moon(spa->jce);
    x[TERM_X3] = spa->x3 = argument_latitude_moon(spa->jce);
    x[TERM_X4] = spa->x4 = ascending_longitude_moon(spa->jce);

    nutation_longitude_and_obliquity(spa->jce, x, &(spa->del_psi), &(spa->del_epsilon));

    spa->epsilon0 = ecliptic_mean_obliquity(spa->jme);
    spa->epsilon  = ecliptic_true_obliquity(spa->del_epsilon, spa->epsilon0);

    spa->del_tau   = aberration_correction(spa->r);
    spa->lamda     = apparent_sun_longitude(spa->theta, spa->del_psi, spa->del_tau);
    spa->nu0       = greenwich_mean_sidereal_time (spa->jd, spa->jc);
    spa->nu        = greenwich_sidereal_time (spa->nu0, spa->del_psi, spa->epsilon);

    spa->alpha = geocentric_sun_right_ascension(spa->lamda, spa->epsilon, spa->beta);
    spa->delta = geocentric_sun_declination(spa->beta, spa->epsilon, spa->lamda);
}

/////////////////////////////////////////////////////////////////////////
// Calculate Equation of Time (EOT) and Sun Rise, Transit, & Set (RTS)
/////////////////////////////////////////////////////////////////////////

void calculate_eot_and_sun_rise_transit_set(spa_data *spa)
{
    spa_data sun_rts = *spa;
    double nu, m, h0, n;
    double alpha[JD_COUNT], delta[JD_COUNT];
    double m_rts[SUN_COUNT], nu_rts[SUN_COUNT], h_rts[SUN_COUNT];
    double alpha_prime[SUN_COUNT], delta_prime[SUN_COUNT], h_prime[SUN_COUNT];
    double h0_prime = -1*(SUN_RADIUS + spa->atmos_refract);
    int i;


    m        = sun_mean_longitude(spa->jme);
    spa->eot = eot(m, spa->alpha, spa->del_psi, spa->epsilon);

    sun_rts.hour = sun_rts.minute = sun_rts.second = sun_rts.timezone = 0;

    sun_rts.jd = julian_day (sun_rts.year, sun_rts.month,  sun_rts.day,
                             sun_rts.hour, sun_rts.minute, sun_rts.second, sun_rts.timezone);

    calculate_geocentric_sun_right_ascension_and_declination(&sun_rts);
    nu = sun_rts.nu;

    sun_rts.delta_t = 0;
    sun_rts.jd--;
    for (i = 0; i < JD_COUNT; i++) {
        calculate geocentric sun right_ascension_and_declination(&sun_rts);
        alpha[i] = sun rts.alpha;
        delta[i] = sun_rts.delta;
        sun_rts.jd++;
    }

    m_rts[SUN_TRANSIT] = approx_sun_transit_time(alpha[JD_ZERO], spa->longitude, nu);
    h0 = sun_hour_angle_at_rise_set(spa->latitude, delta[JD_ZERO], h0_prime);

    if (h0 >= 0) {

        approx_sun_rise_and_set(m_rts, h0);

        for (i = 0; i < SUN_COUNT; i++) {

            nu_rts[i]      = nu + 360.985647*m_rts[i];

            n              = m_rts[i] + spa->delta_t/86400.0;
```

```
            alpha_prime[i] = rts_alpha_delta_prime(alpha, n);
            delta_prime[i] = rts_alpha_delta_prime(delta, n);

            h_prime[i]     = limit_degrees180pm(nu_rts[i] + spa->longitude - alpha_prime[i]);

            h_rts[i]       = rts_sun_altitude(spa->latitude, delta_prime[i], h_prime[i]);
        }

        spa->srha = h_prime[SUN_RISE];
        spa->ssha = h_prime[SUN_SET];
        spa->sta  = h_rts[SUN_TRANSIT];

        spa->suntransit = dayfrac_to_local_hr(m_rts[SUN_TRANSIT] - h_prime[SUN_TRANSIT] / 360.0,
                                              spa->timezone);

        spa->sunrise = dayfrac_to_local_hr(sun_rise_and_set(m_rts, h_rts, delta_prime,
                           spa->latitude, h_prime, h0_prime, SUN_RISE), spa->timezone);

        spa->sunset  = dayfrac_to_local_hr(sun_rise_and_set(m_rts, h_rts, delta_prime,
                           spa->latitude, h_prime, h0_prime, SUN_SET),  spa->timezone);

    } else spa->srha= spa->ssha= spa->sta= spa->suntransit= spa->sunrise= spa->sunset= -99999;

}

//////////////////////////////////////////////////////////////////////////////////////////
// Calculate all SPA parameters and put into structure
// Note: All inputs values (listed in header file) must already be in structure
//////////////////////////////////////////////////////////////////////////////////////////
int spa_calculate(spa_data *spa)
{
    int result;

    result = validate_inputs(spa);

    if (result == 0)
    {
        spa->jd = julian_day (spa->year, spa->month,  spa->day,
                              spa->hour, spa->minute, spa->second, spa->timezone);

        calculate_geocentric_sun_right_ascension_and_declination(spa);

        spa->h  = observer_hour_angle(spa->nu, spa->longitude, spa->alpha);
        spa->xi = sun_equatorial_horizontal_parallax(spa->r);

        sun_right_ascension_parallax_and_topocentric_dec(spa->latitude, spa->elevation, spa->xi,
                                  spa->h, spa->delta, &(spa->del_alpha), &(spa->delta_prime));

        spa->alpha_prime = topocentric_sun_right_ascension(spa->alpha, spa->del_alpha);
        spa->h_prime     = topocentric_local_hour_angle(spa->h, spa->del_alpha);

        spa->e0      = topocentric_elevation_angle(spa->latitude, spa->delta_prime,
        spa->h_prime);
        spa->del_e   = atmospheric_refraction_correction(spa->pressure, spa->temperature,
                                                         spa->atmos_refract, spa->e0);
        spa->e       = topocentric_elevation_angle_corrected(spa->e0, spa->del_e);

        spa->zenith     = topocentric_zenith_angle(spa->e);
        spa->azimuth180 = topocentric_azimuth_angle_neg180_180(spa->h_prime, spa->latitude,
                                                               spa->delta_prime);
        spa->azimuth    = topocentric_azimuth_angle_zero_360(spa->azimuth180);

        if ((spa->function == SPA_ZA_INC) || (spa->function == SPA_ALL))
            spa->incidence  = surface_incidence_angle(spa->zenith, spa->azimuth180,
                                                      spa->azm_rotation, spa->slope);

        if ((spa->function == SPA_ZA_RTS) || (spa->function == SPA_ALL))
            calculate_eot_and_sun_rise_transit_set(spa);
    }

    return result;
}
//////////////////////////////////////////////////////////////////////////////////////////
```

REPORT DOCUMENTATION PAGE

Form Approved
OMB NO. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Revised January 2008	Technical Report

4. TITLE AND SUBTITLE
Solar Position Algorithm for Solar Radiation Applications

5. FUNDING NUMBERS
WU1D5600

6. AUTHOR(S)
Ibrahim Reda & Afshin Andreas

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
National Renewable Energy Laboratory
1617 Cole Blvd.
Golden, CO 80401-3393

8. PERFORMING ORGANIZATION REPORT NUMBER
NREL/TP-560-34302

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
NREL Technical Monitor: Ibrahim Reda

12a. DISTRIBUTION/AVAILABILITY STATEMENT
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road
Springfield, VA 22161

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

1. This report is a step-by-step procedure for implementing an algorithm to calculate the solar zenith and azimuth angles in the period from the year –2000 to 6000, with uncertainties of ±0.0003/. It is written in a step-by-step format to simplify otherwise complicated steps, with a focus on the sun instead of the planets and stars in general. The algorithm is written in such a way to accommodate solar radiation applications.

14. SUBJECT TERMS
algorithm; solar position algorithm; solar radiation; solar radiation applications; solar zenith angles; solar azimuth angles

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL