Calculation of the Sun Position

This document is a technical note on the sun position calculation, which is applied in EA weather data navigation program EA DataNavi 8, and graohic tools, SolMap and SkyMap, provided in EA Graphic Tools 2022. Calculation method of the sun position (solar altitude and azimuth) is quite general one but the calculation parameters for it, the solar declination and the equation of time are calculated by "Matsumoto method (Rev. 2022)" that will be explained in detail in this document.

Before the explanation of "Matsumoto method (Rev. 2022)", althogh it is a little bit verbose for well educated users, we describe the calculation method of the sun position at first.

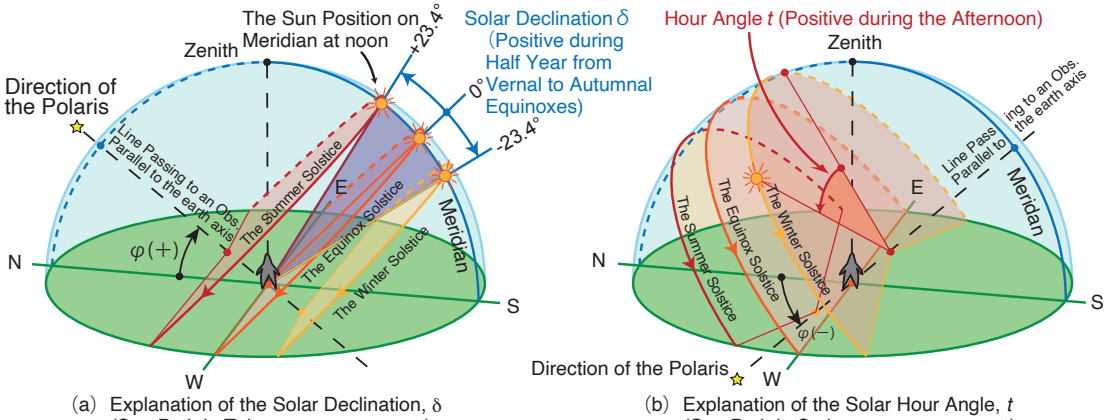
1 Calculation of the Sun Position

In astronomical use, the sun position is enough to be described with two angle parameters: the solar declination δ [°] and the hour angle t [°]. The hour angle can be calculated by the following equation but to find E_t [°] in the equation "the equation of time" is important. Calculation of δ and E_t is main theme in this document. However, we start the explanation on time difference ΔT, which is the difference between astronomical (theoretical system) time and our citizen time in the next Chapter 3.

$$t = 15(T_m - 12) + (L - L_0) + E_t \tag{1}$$

where, T_m: Destination time in ZST [h], L_0: Reference longitude for the ZST [°],
L: Destination longitude [°], E_t: Destination equation of time [°].

East longitude must be considered as positive in L_0 and L. For instance, in Japan, ZST(=JST) location is at east longitude of 135° (Akashi City) then $L_0 = 135$.



(a) Explanation of the Solar Declination, δ (Sun Path in Tokyo, φ = approx.+35°)

(b) Explanation of the Solar Hour Angle, t (Sun Path in Sydney, φ = approx.−34°)

Fig. 1 Explanation of the Solar Declination δ and Hour Angle t (The Four Seasons Are Based on the Northern Hemisphere)

In building environmental engineering field, the sun position is usually assigned with two angles for the destinated location's latitude φ [°]: the solar altitue h [°] and the solar azimuth A [°]. The relationship among δ, t, h, and A is expressed in the next equation, following the spherical trigonometry formulae (See [11], [12]).

$$\sin h = \sin\varphi \sin\delta + \cos\phi \cos\delta \cos t \tag{2}$$

$$\sin A = \frac{\cos\delta \sin t}{\cos h} \tag{3}$$

$$\cos A = \frac{\sin h \sin\varphi - \sin\delta}{\cos h \cos\varphi} \tag{4}$$

When $\sin h > 0$, then it is day time but when $\sin h < 0$, then it is night time. Of course, $\sin h = 0$ means the sunrise or sunset. The solar azimuth A has positive value when the sun located in west hemisphere, that has origin at the south direction[*1]. Thus A has a range: -180°$\leqq A \leqq$ 180°. The sign of A is determined by refering t. When $t > 0$, then $A > 0$ and when $t < 0$, then $A < 0$.

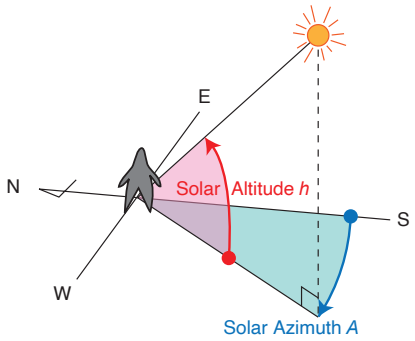


Fig. 2 Definition of Solar Altitude h and Solar Azimuth A

Namely, when $\sin A > 0$ $(t > 0)$; $\quad A = 90 - \tan^{-1}\left(\dfrac{\sin A}{\cos A}\right)$ (5)

when $\sin A < 0$ $(t < 0)$; $\quad A = -90 - \tan^{-1}\left(\dfrac{\sin A}{\cos A}\right)$ (6)

Now, we start the explanation of calculation procedure by "Matsumoto method (Rev. 2022)" [1], [2]. Main theme is to determine two parameters: the solar declination δ and the equation of time E_t[*2]

2 Calculation of Time Difference ΔT and Julian Century T

2.1 Reason for Developing "Matsumoto Method (Rev. 2022)"[*3]

Matsumoto developed a calculation method of the solar declination δ [°] and the equation of time E_t [°] (T_e [min.]) on the basis of the method of Hydrographic and Oceanographic Department in Japan Coast Guard [3]– [6]. The method is named "Matsumoto method" here. The

[*1] International rule is different: the origin is north direction 0° and clockwise direction (east side) is positive. If you need to convert A to the international one A', then use the following equation.

$A' = 180 - A \quad (0 \leqq A \leqq 180), \quad A' = -180 - A \quad (-180 \leqq A \leqq 0)$

[*2] In Apppedix, two famous methods: Yamazaki method [12], [13] and Akasaka method [14]–[17] are mentioned with program source codes.

[*3] This subsection can be omitted if you just want to understand how-to-use of "Matusmoto method (Rev. 2022)".

"Matsumoto method" adopted the new time system TT [h] and TCG [h] that was replaced to TD defined in IAU till year 2000. The time system is one of common basis of astronimical science. "Matsumoto method" was implemented in previous EA weather data navigation program (–EA DataNavi 7) to calculate the sun position data.

In 2022, facing to publish plan of new EA weather data DVDs, Matsumolto decide to revise and renewal the method due to the following reasons and/or circumstances. The revised method explained latter is named "Matsumoto method (Rev. 2022)" here.

(a) There are some theories on expression of the planetary position or orbit. However, VSOP 87 (Variations Séculaires des Orbites Planétairs, 1987) [20] may be the most popular one, that is very complicated and long calculation method using functions each having over 400 cosine terms with a parameter of Julian century T. Although it may be difficult to apply the formulae, due to the open and free-fee source code of VSOP 87, that method is accepted worldwide as a de facto standard calculation method. We understand now the situation of studies in astronomical science so.

(b) NREL (National Renewable Energy Laboratory, US DOE) developed a method named "NREL method" here [21]. This method is famous in American and European nations for building environmental engineering practices and studies. It may become popular method due to the EPW data, we think so. The EPW (EnergyPlus™ Weather) data includes the solar altitude and the solar azimuth data calculated by NREL method. Additionally, open and free source code of NREL method is a reason for its popularity. NREL method is applied a simplified formulae of VSOP 87 proposed by Meeus, a famous Belgian astronomic scientist [22]. We have a doubt for this method because of lack of information on the time difference ΔT that is an important parameter for precise calculation. It may cause a wrong usage with bonehead input data by users.

(c) The reference studies by Hydrographic and Oceanographic Department of Japan Coast Guard [19], [23] for "Matsumoto method" must be one of simplified calculation method based on VSOP 87. The method by Japan Coast Guard released every year must be simplified one to fit for small time span with closely estimated value of ΔT.

(d) The reasons and/or circumstances mentioned as items (a)–(c) support Matsumoto's research activities [6] on updating ΔT with validation with recent reliable data to be proper, although the studied opportunities were affected to release schedules of the EA weather data navigation programs (EA DataNavi 6 and 7).

(e) In spring of 2022, new EA weather data DVDs were released and new operation program (EA DataNavi 8) was provided. The one of hottest features in this release turn is that a future standard weather data of 2086 (`FRY2086.wea2`), based on estimated 20 years from 2076 to 2095 is released. This gives great impact for calculation program code of the sun position, because we never check the long future data of the sun position. Thus, this is a good opportunity to refine or renewal the implemented method.

(f) In "Matsumoto method", there is a dissatisfied point by the developer Matsumoto himself: the time difference between UT1 and UTC is ignored, on the other hand, the time difference between TT and TCG is considered inconsistently. Now, for consideration of time differences, it is clear from the literature by Meeus [22]: He says that old time system

ET, previous time system TD, and present time system TT are considerable to be fundamentally same and one constant time system practically. Thus, "Matsumoto method (Rev. 2022)" is not strict to tune up the time system.

(g) A good issue on estimation of the time difference ΔT for long time is published by NASA based on the research by Espenak and Meeus [24]. It covers till year 2150 and suitable to implement in the procedure of "Matsumoto method (Rev. 2022)".

Explanation in the next section is mainly related on items (f) and (g) mentioned above.
Now on item (f), we use UT without discrimination between UT1 and UTC.

2.2 Definition of Time Difference ΔT

The time difference term ΔT is defined as the value difference between the one of three astronimical times (TT, TD, or ET) and our citizen time, universal Time UT, expressed in second. Here TT is a abbreviation of Terrestrial Time; TD for Dynamical Time, and ET for Ephemeris Time[*4].

$$\begin{aligned} \Delta T &= 3,600^{\mathrm{s}} \times (\mathrm{TT} - \mathrm{UT}) \\ &= 3,600^{\mathrm{s}} \times (\mathrm{TD} - \mathrm{UT}) = 3,600 \times (\mathrm{ET} - \mathrm{UT}) \end{aligned} \tag{7}$$

ΔT is a time depended continuous function but its fluctuation during a whole year is not so wide. Thus "Matsumoto method (Rev. 2022)" adopts one value for each year, that is a datum at $0^{\mathrm{h}} UT$ on July 1.

2.3 Calculation Formulae of ΔT (Renewal)

As mentioned as itemized sentences (e)–(g) in Section refse2:2.1, by refering NASA's report [24], we use the following equations to estimate ΔT. We assume that `YYYY` is a target year to calculate. `YYYY` can be chosen for years from 1600 and 2150[*5].

At first, we calculate time parameter y (positive real value) by using Eq. (8). In the equation decimal value 0.5 means July 1.0.

$$y = \texttt{YYYY} + 0.5 \tag{8}$$

Usually, y should be changed to an elapsed year from a specified era t. Then we calculate ΔT by using the polynomial formula in Eq. (9).

$$\begin{aligned} \Delta T &= a_0 + a_1\, t + a_2\, t^2 + \cdots + a_n\, t^n \\ &= a_0 + t\,(a_1 + t\,(a_2 + \cdots t\,(a_{n-1} + t\, a_n) \cdots)) \end{aligned} \tag{9}$$

[*4] In Matsumoto's research works [3]–[6], symbol ΔT_1 is used. This is the same as ΔT here.
By the way, ET was used before 1984; TD was used during 1984–1990; and TT is used since 1991 in the astronomical science field.
And we know the relationship between UT and the Japan Standard Time JST: $\mathrm{UT} = \mathrm{JST} - 9^{\mathrm{h}}$.

[*5] Correctly writing, in the report [24], equations for years from -1999 to 3000 are proposed. However, we adopted a practical range of years for engineering purpose.

The equation for t and data of coefficients $a_i\,(i = 1, 2, \ldots, n)$ in Eq. (9) are given by one case of following 11 cases separated by year ranges[*6].

- year: 1600–1700: $t = y - 1600,\ n = 3$

  $a_0 = 120 \quad a_1 = -0.980\,8 \quad a_2 = -0.015\,32 \quad a_3 = 1/7,129$

- year: 1701–1800: $t = y - 1700,\ n = 4$

  $a_0 = 8.83 \qquad a_1 = 0.160\,3 \quad a_2 = -0.005\,928\,5 \quad a_3 = 0.000\,133\,36$
  $a_4 = -1/1,174,000$

- year: 1801–1860: $t = y - 1800,\ n = 7$

  $a_0 = 13.72 \qquad a_1 = -0.332\,447 \qquad a_2 = 0.006\,861\,2$
  $a_3 = 0.004\,111\,6 \qquad a_4 = -0.000\,374\,36 \qquad a_5 = 0.000\,012\,127\,2$
  $a_6 = -0.000\,000\,169\,9 \quad a_7 = 0.000\,000\,000\,875$

- year: 1861–1900: $t = y - 1860,\ n = 5$

  $a_0 = 7.62 \qquad a_1 = 0.573\,7 \qquad a_2 = -0.251\,754 \quad a_3 = 0.016\,806\,68$
  $a_4 = -0.000\,447\,362\,4 \quad a_5 = 1/233\,174$

- year: 1901–1920: $t = y - 1900,\ n = 4$

  $a_0 = -2.79 \qquad a_1 = 1.494\,119 \quad a_2 = -0.059\,893\,9 \quad a_3 = 0.006\,196\,6$
  $a_4 = -0.000\,197\,362\,4$

- year: 1921–1940: $t = y - 1920,\ n = 3$

  $a_0 = 21.20 \quad a_1 = 0.844\,93 \quad a_2 = -0.076\,100 \quad a_3 = 0.002\,093\,6$

- year: 1941–1960: $t = y - 1950,\ n = 3$

  $a_0 = 29.07 \quad a_1 = 0.407 \quad a_2 = -1/233 \quad a_3 = 1/2,547$

- year: 1961–1985: $t = y - 1975,\ n = 3$

  $a_0 = 45.45 \quad a_1 = 1.067 \quad a_2 = -1/260 \quad a_3 = -1/718$

- year: 1986–2005: $t = y - 2000,\ n = 5$

  $a_0 = 63.86 \qquad a_1 = 0.334\,5 \qquad a_2 = -0.060\,374 \quad a_3 = 0.001\,727\,5$
  $a_4 = 0.000\,651\,814 \quad a_5 = 0.000\,023\,735\,99$

- year: 2006–2050: $t = y - 2000,\ n = 2$

  $a_0 = 62.92 \quad a_1 = 0.322\,17 \quad a_2 = 0.005\,589$

[*6] In the second line equation of Eq. (9), the Müller's polynomial expression is displayed. This is a good technique to avoid information loss caused by bad source code.

We assume that dT must be calculated with the third decimal place, *i.e.* order of 1 ms.

- year: 2051–2150: $t = y - 1820,\ n = 2$

$$a_0 = -205.724 \quad a_1 = 0.562\,8 \quad a_2 = 0.003\,2$$

Long term fluctuation of dT is shown in Fig. 3 and Fig. 4 . In these figure, we put two marks obtained from table data of almanac citect:AA2020: • means observed (and corrected) value; ◦ means extrapolated value. These refered values are shown for every years' January 1.0. Thus just for drawing these figures, we use $t = \mathrm{int}(\ t\ )$. In addition, Matsumoto's previous poposed values of ΔT are shown in a dashed line for 220 years from 1800 to 2020.

As shown in Fig. 3 , we find that there is a big difference between observed data and calculated data before 1640. However, the observed data themselves may be not so precise because lack of measurement instrument and techniques. Thus the difference must be accepted from viewpoint of practical and engineering purposes. In the other figure (Fig. 4), we should note that the time difference ΔT will grow to about 180 s (3 min.) in 2090!

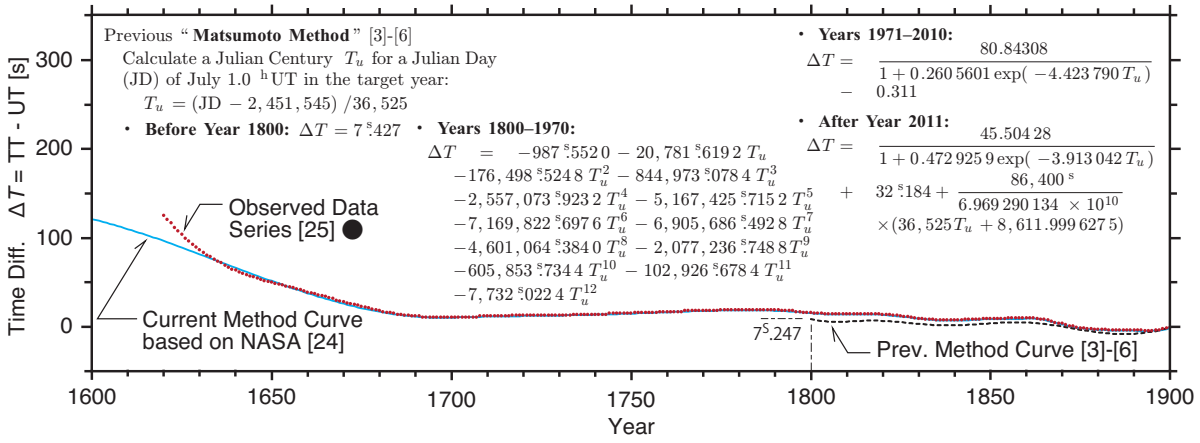


Fig. 3 Fluctuation of Renewal Time Difference ΔT (1/2) (1600–1900)

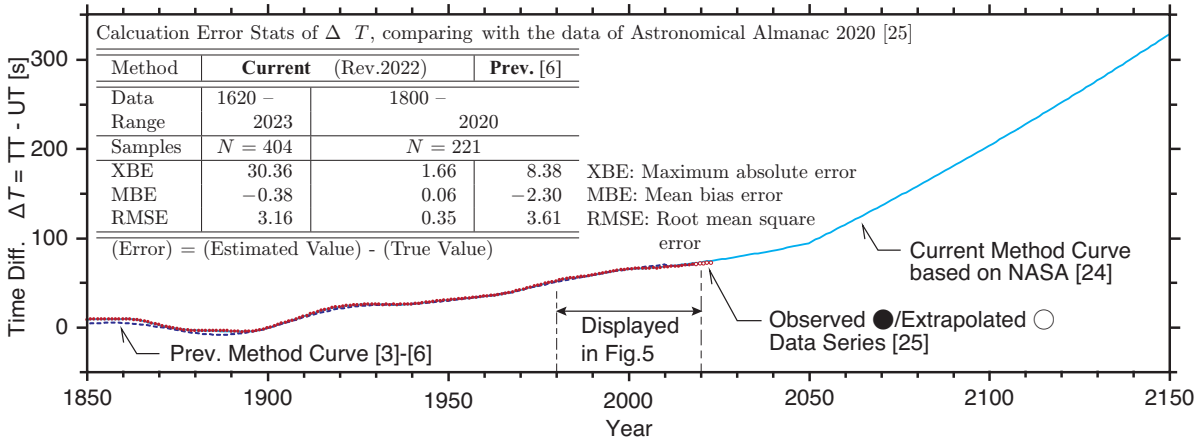


Fig. 4 Fluctuation of Renewal Time Difference ΔT (2/2) (1850–2150)

2.4 Error Evaluation of Time Diffence ΔT

In blank space of Fig. 4 , the error evaluation results are tabulated. It seems that "Matsumoto method (Rev. 2022)" has more improved precision than previous "Matsumoto method".

Fig. 5 shows the focused image for 40 years from 1981 to 2020, that are included as `EADyyyy.Wea2` datafiles. It is found that the "Matsumoto method (Rev. 20222)" is also improved for these 40 years.

Now we get one conclusion that new adoption of calculation method of ΔT based on NASA report [24] is suitable for EA weather data operation.

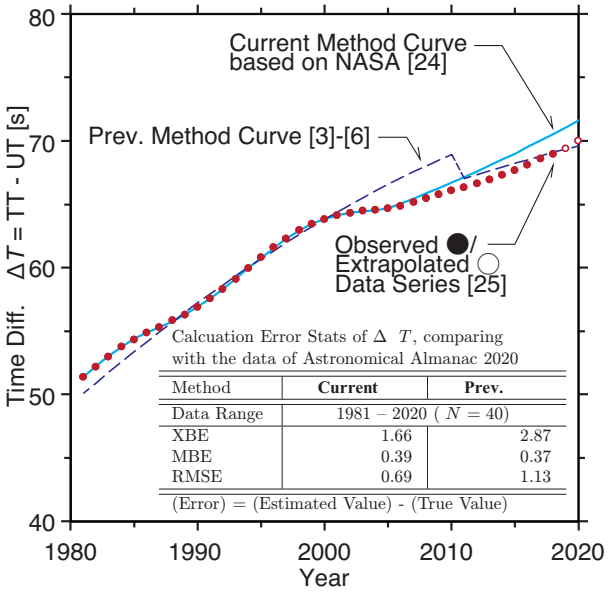


Fig. 5 Observed Values and Estimation Curve of Time Difference ΔT (1981–2020)

By the way, on difference of Matsumoto's methods, ΔT values calculated by previous method are different from the values by renewal method. It is true but when the sumn position data are calculated, in the solar altitude and azimuth data, the difference is less than 0.1°, that is output precision in EA DataNavi 7 and EA DataNavi 8.

You may wonder that the previous estimation curve in Fig. 5 has strange step. The reason was discussed in previous paper [6].

2.5 Calculation of Julian Century T

We define sympol $\overline{\text{JD}}$ to Julian day in integer for given year, moth, day. And define JD to Julian day including hours in real. At 0^{h}UT of a given date, JD $= \overline{\text{JD}} - 0.5$ because noon is the origin of the day in the astronomical science. Thus when we want to know JD at H^{h}JST, Julian day at that time is: JD $= \overline{\text{JD}} + (H - 9^{\text{h}})/24 - 0.5$.

However, we use the another Julian day, called "Julian Ephemeris Day" JDE for calculation. The time difference ΔT is taken into account for JD to get JDE.

Now, we use ΔT [s] obtained by Eq. (9) in the next equation.

$$\begin{aligned} \text{JDE} &= \text{JD} + \Delta T/3,600 \\ &= \overline{\text{JD}} + (H - 9^{\text{h}})/24 - 0.5 + \Delta T/3,600 \end{aligned} \tag{10}$$

And we convert this JDE to "Julian century" T to use as calculation parameter for many equations explained latter.

$$T = (\text{JDE} - 2\,451\,545.0)\,/36\,525 \tag{11}$$

By the way, there are many information in social media on a matter how to calculate Julian day $\overline{\mathrm{JD}}$, especially with Microsoft® Excel. If you need, refer them. And this document has an appendix on the calculation of $\overline{\mathrm{JD}}$ in Appendix B.

3 Calculation Procedure of the Solar Declintion δ and the Equation of TimeE_t

Now we assume the calculation at the specified location on the specified date and time: Latitude φ [°], Longitude L [°], Date `YYYY/MM/DD`, Time `hh:mm:ss` JST. Calculation of the solar altitude and azimuth, h and A [°] is exercised.

By the way, in following description in this chapter, arguments for trigonometric functions are defined in degree unit. The same manner is also applied to inverse trigonometric functions.

At first, JST time `hh:mm:ss` should be expressed in [h] unit and convert to UT [h]. Of course, negative value of UT is acceptable.

$$\mathrm{UT} = \mathtt{hh} + \mathtt{mm}/60 + \mathtt{ss}/3,600 - 9.0000000 \tag{12}$$

3.1 Each Step of the Calculation

Step 1: Calculation of the Time Difference ΔT

We calculate ΔT with the third decimal order. We should access correct year range in Section 2.3 to calculate with Eq. (8) and Eq. (9). As mentioned already, we assume ΔT value is representative for the year `YYYY`.

Step 2: Calculation of Julian Century T and T_u

We calculate Julian day $\overline{\mathrm{JD}}$ for year `YYYY`, month `MM`, and day `DD` [*7], then calculate T by Eq. (10) and Eq. (11).

Subsequently, we calculate JD: $\mathrm{JD} = \overline{\mathrm{JD}} + \mathrm{UT}/24 - 0.5$. Then T_u must be calculated by Eq. (13) for preparation for latter procedure.

$$T_u = (\mathrm{JD} - 2\,451\,545.0)\,/36\,525 \tag{13}$$

Step 3: Calculation of the Right Ascension α_M for Mean Sun

Using T_u, calculated in the previous **Step: 2**, we calculate the mean right ascension α_M by Eq. (14) and Eq. (15).

$$\begin{aligned}\alpha_m = \Big(&18^{\mathrm{h}}41^{\mathrm{m}}50^{\mathrm{s}}.548\,41 + 8,640,184^{\mathrm{s}}.812\,866\,T_u \\ &+ 0^{\mathrm{s}}.093\,104\,T_u^2 - 0^{\mathrm{s}}.000\,006\,2\,T_u^3\Big) \bmod 24 \end{aligned} \tag{14}$$

$$\alpha_M = 15\,\alpha_m \tag{15}$$

[*7] If the time is 0^{h}UT, you know that JDE must have decimal of 0.5. When destination date `YYYY/MM/DD` is after `1900/01/01`, consider that date has serial day number 1, calculate the serial day number of the destination date and add 2 415 019.5. That is classical calculation algorithm of JDE.

Eq. (14) is the standard equation of IAU with [h] unit. Eq. (15) is simply unit conversion to degree.

Step 4: Calculation of the Zodiac Tilt Angle ε

We calcuate $ve\,[^\circ]$ by Eq. (16) with time parameter T. Coefficients V_i $(i = 0, \cdots, 3)$ are summaried in Table 1.

$$\varepsilon = \left(V_0 + \sum_{i=1}^{3} V_i T^i \right) / 3,600 S + \sum_{j=1}^{2} X_j \cos(Y_j T + Z_j) \tag{16}$$

Cosine function's argument may calculate in degree unit in Eq. (16). The argument may be calculated as modulo of 360°.

Table 1 Coefficients for Calculation of the Zodiac Tilt Angle ε

i	$V_i\,['']$	j	$X_j\,[^\circ]$	$Y_j\,[^\circ]$	$Z_j\,[^\circ]$
0	−84,381.448	–			
1	+46.815	1	−0.002 56	1,934	235
2	+0.000 590	2	−0.000 15	72,002	201
3	−0.001 813	–			

Step 5: Calculation of the Apparent Ecliptic Longitude Ψ

Using coefficient P_i, Q_i, R_i $(i = 1, \cdots, 18)$ listed in Table 2, we calculate $\Psi\,[^\circ]$ by Eq. (17). Parameter of the functions is T.

$$\Psi = \left[\sum_{i=1}^{17} P_i \cos(Q_i T + R_i) + P_{18} T \cos(Q_{18} T + R_{18}) + 36,000.769\,5\, T + 280.460\,2 \right] \bmod 360 \tag{17}$$

Table 2 Calculation Coefficients for Ψ and T_e

i	$P_i\,[^\circ]$	$Q_i\,[^\circ]$	$R_i\,[^\circ]$	i	$P_i\,[^\circ]$	$Q_i\,[^\circ]$	$R_i\,[^\circ]$
1	+1.914 7	35,999.05	267.52	10	+0.000 7	9,038	64
2	+0.020 0	71,998.1	265.1	11	+0.000 6	33,718	316
3	+0.020 0	32,964	158	12	+0.000 5	155	118
4	+0.001 8	19	159	13	+0.000 5	2,281	221
5	+0.001 8	445,267	208	14	+0.000 4	29,930	48
6	+0.001 5	45,038	254	15	+0.000 4	31,557	161
7	+0.001 3	22,519	352	16	+0.004 8	1,934	145
8	+0.000 7	65,929	45	17	−0.000 4	72,002	111
9	+0.000 7	3,035	110	18	−0.004 8	35,999	268

By the way, mathematical expression of coefficients are correct as shown here but operation od summation in computer may be changed the adding order. In order to avoid information loss, addition should be done from small value to large value order.

Step 6: Calculation of the Solar Declination δ and the Equation of Time E_t

Now, we assemble the calculated mean right ascension α_M, the zodiac tilt angle ε, and the apparent ecliptic longitude Ψ. And tune by coefficients P_i, Q_i, R_i $(i = 16, 17)$ in Table 2. The calculation is expressed as follows:

$$\delta = \arctan\left(\frac{\sin\Psi\sin\varepsilon}{\sqrt{1-\sin^2\Psi\sin^2\varepsilon}}\right) \tag{18}$$

$$E_t = \left[\sum_{i=16}^{17} P_i\cos(Q_iT + R_i) - 0.005\,7\right]\cos\varepsilon + \arctan\left(\frac{\tan\alpha_M - \tan\Psi\cos\varepsilon}{1+\tan\alpha_M\tan\Psi\cos\varepsilon}\right) \tag{19}$$

$$T_e = \frac{1}{15}E_t \tag{20}$$

The equation of time E_t is calculated by using Eq. (19) in degree unit. And as you can see term of 1/15 in Eq. (20), the another equation of time T_e is calculated in hour. The expression of these two equations is based on orinal work by Matsumoto.

3.2 Final Step: Calculation of the Solar Altitude h and the Azimuth A

As usual procedure which is learned from general text book for building environmental engineering, using the spherical trigonometry formulae with hour angle $t\,[^\circ]$ calculated from obtained equation of time E_t or T_e and calculated solar declination δ, we can calculate from Eq. (1)–Eq. (6)[*8].

3.3 Another Calculation — Geocentric Distance

The "geocentric distance", r, in this subsection means the true distance in AU unit from the Earth to the Sun. This parameter is important when you should set up the extra-atmospheric normal beam solar irradiance, $I_0\,[\mathrm{W/m^2}]$ based on the solar constant $\overline{I_0} = 1367\,[\mathrm{W/m^2}]$[*9]

$$I_0 = \overline{I_0}/r^2 \tag{21}$$

[*8] The followings are just tiny technical chips.

To improve the accuracy of calculated solar azimuth, you may try the another equation:

$\sin A = \cos\delta\sin t/\cos h$

Using $\tan A$ is also good idea. And as you know, δ and E_t do not fluctuate hour by hour and to use daily values for them is also acceptable. However, as mentioned latter with a calculation example, from viewpoint to keep high accuracy, hourly calculation for δ and E_t is recommended.

[*9] It may be calculated day by day practically.

The geocentric distance r can be calculated as a function of the Julian century T as shown in the equation below [19]:

$$r = \sum_{i=1}^{8} S_i \cos(U_i T + W_i) + S_9 T \cos(U_9 T + W_9) \tag{22}$$

The coefficients S_i, U_i, and W_i have data described in the table below.

Table 3 Calculation Coefficients for r

i	S_i [–]	U_i [°]	W_i [°]
1	1.000 140	0.0	0.0
2	0.016 706	35, 999.05	177.53
3	0.000 139	71, 998	175
4	0.000 031	445, 267	298
5	0.000 016	32, 964	68
6	0.000 016	45, 038	164
7	0.000 005	22, 519	233
8	0.000 005	33, 718	226
9	−0.000 042	35, 999	178

Program source codes (FORTRAN, C/C++) for Eq. (22) and the calculated example are described in Appendix D.

4 Examples

4.1 Simple Example

We calculate the solar declination δ [°], the equation of time T_e [s], and solar altitude h [°] and azimuth A [°] for Tokyo (Latitude $\varphi = +35°41'06''$, Longitude $L = 139°45'36''$) at the time 15^{h}JST (6^{h}UT) on July 24, 2020.

The calculated results are summarized in Table 4. In the table, there are two lines with 0^{h}UT and 6^{h}UT. These difference is the timing difference to calculate δ and T_e.

4.2 Example for Programming Test Bed

AT first, find the time difference ΔT for year 2020.

Then calculate the solar declination δ (`Sol.Decl.`) [°] and the eauation of time E_t (`Eq.Time`) [°] hour by hour for following three days: March 30, 2020 (90th day), June 28, 2020 (180th day), and Sept. 26, 2020 (270th day).

Table 4 Calculation Example for Tokyo at 15 O'clock (JST) on July 24, 2020

Latitude $\varphi = +35°41'06''$, Longitude $L = +139°45'36''$

Method	δ [°]	T_e [s]	h [°]	A [°]
Rika Nenpyo 0^{h}UT	19.808 9	−392.1	45.053 0	82.593 0
Rika Nenpyo 6^{h}UT*	19.755 9	−392.4	45.026 6	82.528 0
Matsumoto (Rev. 2022) 0^{h}UT	19.808 7	-392.0	45.052 7	82.592 9
Matsumoto (Rev. 2022) 6^{h}UT	19.755 6	-392.3	45.026 1	82.528 0
Akasaka method	19.807 8	-392.0	45.052 0	82.592 1

∗ interpolated by quadratic function, Rika Nenpyo [18], Akasaka method [14]

In addtition, assuming two locations: Tokyo ($\varphi = 35.692, L = 139.750$) and Kagoshima ($\varphi = 31.555, L = 130.541$), calculate the hour angle t (`Hr.Angle`), solar altitude h (`Altitude`), and solar azimuth A (`Azimuth`) for each location.

The similar calculation will be done for the different time: March 31, 2086 (90th day), June 29, 2086 (180th day), and Sept. 27, 2086 (270th day).

Calculated ΔT are as follows:

- July 10^{h}UT, 2020 $\Delta T = 71.873$ [s]
- July 10^{h}UT, 2086 $\Delta T = 171.532$ [s]

The other calculated items are displayed in [°] unit. The solar declination (`Sol.Decl.`) and the equation of time (`Eq.Time`) are expressed in five decimal places (rounded). The others are expressed in four decimal places (rounded).

Table 5 and Table 6 are the listed results, which may be useful for a programmer to check his/her own codes.

Table 5 Calculated Exaple for Three Days in 2020 for Tokyo and Kagoshima

Matsumoto Rev. Method (2022) ... Year:2020 Delta T: 71.873 sec. [All Units: deg. of angle]

	Tokyo					Kagoshima			
Latitude, Longitude:	35.692	139.750				31.555	130.547		
YYYY/MM/DD HHh:	Sol.Decl.	Eq.Time	Hr.Angle	Altitude	Azimuth	Hr.Angle	Altitude	Azimuth
2020/03/30 01h:	3.73383	-1.14699	-161.3870	-46.8974	-152.2178	-170.6003	-53.5941	-164.0620
2020/03/30 02h:	3.75002	-1.14387	-146.3839	-39.5538	-134.2338	-155.5972	-47.7419	-142.1890
2020/03/30 03h:	3.76621	-1.14075	-131.3807	-29.8322	-120.3355	-140.5941	-38.5080	-125.9543
2020/03/30 04h:	3.78240	-1.13763	-116.3776	-18.7596	-109.2503	-125.5910	-27.4089	-113.9322
2020/03/30 05h:	3.79859	-1.13451	-101.3745	-6.9610	-99.7793	-110.5878	-15.3266	-104.4138
2020/03/30 06h:	3.81477	-1.13139	-86.3714	5.1694	-90.9897	-95.5847	-2.7471	-96.1789
2020/03/30 07h:	3.83095	-1.12827	-71.3683	17.3310	-82.0712	-80.5816	10.0262	-88.3408
2020/03/30 08h:	3.84713	-1.12515	-56.3652	29.2106	-72.1270	-65.5785	22.7451	-80.0910
2020/03/30 09h:	3.86331	-1.12203	-41.3620	40.3564	-59.9061	-50.5754	35.1130	-70.4176
2020/03/30 10h:	3.87949	-1.11891	-26.3589	49.9593	-43.5166	-35.5722	46.6299	-57.6918
2020/03/30 11h:	3.89566	-1.11580	-11.3558	56.5250	-20.8643	-20.5691	56.2556	-39.1256
2020/03/30 12h:	3.91183	-1.11268	3.6473	58.0487	6.8880	-5.5660	61.8658	-11.8421
2020/03/30 13h:	3.92800	-1.10956	18.6504	53.8753	32.7631	9.4371	60.9834	19.7085
2020/03/30 14h:	3.94417	-1.10645	33.6536	45.6130	52.2191	24.4402	54.0900	44.7293
2020/03/30 15h:	3.96033	-1.10333	48.6567	35.1378	66.3298	39.4433	43.8393	61.4867
2020/03/30 16h:	3.97650	-1.10022	63.6598	23.5766	77.2757	54.4464	32.0450	73.2373
2020/03/30 17h:	3.99266	-1.09710	78.6629	11.5311	86.6159	69.4496	19.5628	82.4429
2020/03/30 18h:	4.00882	-1.09399	93.6660	-0.6318	95.3935	84.4527	6.8204	90.5256
2020/03/30 19h:	4.02497	-1.09088	108.6691	-12.6158	104.4358	99.4558	-5.9072	98.4156
2020/03/30 20h:	4.04113	-1.08776	123.6722	-24.0861	114.5876	114.4589	-18.3646	106.9184
2020/03/30 21h:	4.05728	-1.08465	138.6753	-34.5533	126.8942	129.4620	-30.2124	116.9805
2020/03/30 22h:	4.07343	-1.08154	153.6785	-43.2158	142.6352	144.4651	-40.8827	129.9338
2020/03/30 23h:	4.08957	-1.07843	168.6816	-48.8345	162.6983	159.4682	-49.3458	147.5222
2020/03/30 24h:	4.10572	-1.07532	183.6847	-50.0596	-174.2697	174.4714	-53.9525	170.6014
2020/06/28 01h:	23.27937	-0.80563	-161.0456	-28.3668	-160.1787	-170.2590	-34.3784	-169.1456
2020/06/28 02h:	23.27743	-0.80777	-146.0478	-22.8543	-146.1688	-155.2611	-30.2736	-153.5688
2020/06/28 03h:	23.27548	-0.80990	-131.0499	-15.0400	-134.1651	-140.2632	-23.2763	-140.2629
2020/06/28 04h:	23.27351	-0.81203	-116.0520	-5.5786	-123.9813	-125.2654	-14.1929	-129.3159
2020/06/28 05h:	23.27154	-0.81417	-101.0542	5.0141	-115.1684	-110.2675	-3.6931	-120.2815
2020/06/28 06h:	23.26955	-0.81630	-86.0563	16.3657	-107.2186	-95.2696	7.7496	-112.6007
2020/06/28 07h:	23.26756	-0.81843	-71.0584	28.2059	-99.5959	-80.2718	19.8161	-105.7508
2020/06/28 08h:	23.26555	-0.82055	-56.0606	40.3173	-91.6255	-65.2739	32.2867	-99.2229
2020/06/28 09h:	23.26352	-0.82268	-41.0627	52.4687	-82.1442	-50.2760	44.9918	-92.3749
2020/06/28 10h:	23.26149	-0.82480	-26.0648	64.2493	-68.3009	-35.2781	57.7550	-83.9802
2020/06/28 11h:	23.25945	-0.82692	-11.0669	74.3016	-40.6760	-20.2803	70.2210	-70.2273
2020/06/28 12h:	23.25739	-0.82905	3.9310	77.1133	16.4040	-5.2824	80.4727	-30.7320
2020/06/28 13h:	23.25532	-0.83116	18.9288	69.4238	57.9967	9.7155	78.0415	48.4415
2020/06/28 14h:	23.25324	-0.83328	33.9267	58.1578	76.4026	24.7134	66.6086	75.3589
2020/06/28 15h:	23.25115	-0.83540	48.9246	46.1052	87.3953	39.7113	53.9860	86.7352
2020/06/28 16h:	23.24904	-0.83751	63.9225	33.9391	95.8814	54.7092	41.2157	94.4663
2020/06/28 17h:	23.24693	-0.83962	78.9204	21.9411	103.5667	69.7070	28.5611	101.1319
2020/06/28 18h:	23.24480	-0.84174	93.9183	10.3243	111.2883	84.7049	16.1879	107.6970
2020/06/28 19h:	23.24266	-0.84385	108.9162	-0.6731	119.6251	99.7028	4.2755	114.7387
2020/06/28 20h:	23.24051	-0.84595	123.9140	-10.7332	129.0938	114.7007	-6.9325	122.7613
2020/06/28 21h:	23.23835	-0.84806	138.9119	-19.4121	140.1859	129.6986	-17.0775	132.3023
2020/06/28 22h:	23.23617	-0.85016	153.9098	-26.1145	153.2527	144.6965	-25.6303	143.9150
2020/06/28 23h:	23.23399	-0.85227	168.9077	-30.1527	168.2026	159.6944	-31.8656	157.9465
2020/06/28 24h:	23.23179	-0.85437	183.9056	-30.9673	-175.8140	174.6923	-34.9781	174.0453
2020/09/26 01h:	-1.20818	2.13946	-158.1005	-49.9744	-144.5625	-167.3139	-57.3716	-155.9705
2020/09/26 02h:	-1.22440	2.14305	-143.0970	-41.4391	-126.7937	-152.3103	-49.9567	-133.7715
2020/09/26 03h:	-1.24062	2.14663	-128.0934	-30.9051	-113.5056	-137.3067	-39.6057	-118.3704
2020/09/26 04h:	-1.25685	2.15021	-113.0898	-19.3450	-102.9155	-122.3031	-27.8236	-107.1580
2020/09/26 05h:	-1.27307	2.15379	-98.0862	-7.3069	-93.6988	-107.2995	-15.3647	-98.1469
2020/09/26 06h:	-1.28929	2.15736	-83.0826	4.8573	-84.9036	-92.2960	-2.6310	-90.1024
2020/09/26 07h:	-1.30552	2.16094	-68.0791	16.8496	-75.7107	-77.2924	10.1065	-82.1476
2020/09/26 08h:	-1.32174	2.16451	-53.0755	28.3187	-65.2121	-62.2888	22.5867	-73.4540
2020/09/26 09h:	-1.33796	2.16809	-38.0719	38.7283	-52.2078	-47.2852	34.4488	-62.9675
2020/09/26 10h:	-1.35418	2.17166	-23.0683	47.1633	-35.1790	-32.2817	45.0608	-49.1039
2020/09/26 11h:	-1.37040	2.17523	-8.0648	52.1876	-13.2245	-17.2781	53.2196	-29.7297
2020/09/26 12h:	-1.38662	2.17880	6.9388	52.3667	11.4080	-2.2745	56.9877	-4.1762
2020/09/26 13h:	-1.40285	2.18237	21.9424	47.6371	33.6687	12.7290	54.8985	22.5239
2020/09/26 14h:	-1.41907	2.18594	36.9459	39.3827	51.0237	27.7326	47.8219	43.8558
2020/09/26 15h:	-1.43529	2.18950	51.9495	29.0687	64.2465	42.7362	37.7763	59.1266
2020/09/26 16h:	-1.45151	2.19307	66.9531	17.6439	74.8617	57.7397	26.1958	70.4117
2020/09/26 17h:	-1.46773	2.19663	81.9566	5.6627	84.0938	72.7433	13.8454	79.4953
2020/09/26 18h:	-1.48395	2.20019	96.9602	-6.5171	92.8632	87.7469	1.1424	87.5560
2020/09/26 19h:	-1.50017	2.20375	111.9638	-18.6000	101.9867	102.7504	-11.6368	95.4574
2020/09/26 20h:	-1.51638	2.20731	126.9673	-30.2458	112.3984	117.7540	-24.2381	104.0301
2020/09/26 21h:	-1.53260	2.21087	141.9709	-40.9325	125.3964	132.7575	-36.3207	114.3622
2020/09/26 22h:	-1.54882	2.21443	156.9744	-49.7311	142.7777	147.7611	-47.2773	128.1876
2020/09/26 23h:	-1.56504	2.21798	171.9780	-55.0768	165.8962	162.7647	-55.8798	148.1277
2020/09/26 24h:	-1.58126	2.22154	186.9815	-55.2859	-167.6808	177.7682	-59.9522	175.5412

Table 6 Calculated Exaple for Three Days in 2086 for Tokyo and Kagoshima

Matsumoto Rev. Method (2022) ... Year:2086 Delta T: 171.532 sec. [All Units: deg. of angle]

	Tokyo					Kagoshima			
Latitude, Longitude:	35.692	139.750				31.555	130.547		
YYYY/MM/DD HHh:	Sol.Decl.	Eq.Time	Hr.Angle	Altitude	Azimuth	Hr.Angle	Altitude	Azimuth	
2086/03/31 01h:	4.12483	-1.05896	-161.2990	-46.5023	-152.3153	-170.5123	-53.1934	-164.0726	
2086/03/31 02h:	4.14096	-1.05587	-146.2959	-39.1848	-134.4359	-155.5092	-47.3627	-142.3808	
2086/03/31 03h:	4.15710	-1.05277	-131.2928	-29.4921	-120.5792	-140.5061	-38.1646	-126.2174	
2086/03/31 04h:	4.17323	-1.04967	-116.2897	-18.4419	-109.5077	-125.5030	-27.0959	-114.2133	
2086/03/31 05h:	4.18936	-1.04657	-101.2866	-6.6572	-100.0422	-110.4999	-15.0343	-104.6974	
2086/03/31 06h:	4.20549	-1.04348	-86.2835	5.4677	-91.2580	-95.4968	-2.4662	-96.4646	
2086/03/31 07h:	4.22162	-1.04038	-71.2804	17.6325	-82.3473	-80.4937	10.3044	-88.6334	
2086/03/31 08h:	4.23774	-1.03728	-56.2773	29.5251	-72.4111	-65.4906	23.0296	-80.3966	
2086/03/31 09h:	4.25387	-1.03419	-41.2742	40.6950	-60.1869	-50.4875	35.4149	-70.7394	
2086/03/31 10h:	4.26999	-1.03110	-26.2711	50.3315	-43.7459	-35.4844	46.9629	-58.0160	
2086/03/31 11h:	4.28610	-1.02800	-11.2680	56.9241	-20.9182	-20.4813	56.6314	-39.3741	
2086/03/31 12h:	4.30222	-1.02491	3.7351	58.4284	7.1272	-5.4782	62.2655	-11.8040	
2086/03/31 13h:	4.31833	-1.02182	18.7382	54.1864	33.1912	9.5248	61.3314	20.1175	
2086/03/31 14h:	4.33444	-1.01873	33.7413	45.8545	52.6758	24.5279	54.3482	45.2518	
2086/03/31 15h:	4.35055	-1.01564	48.7444	35.3319	66.7579	39.5310	44.0305	61.9745	
2086/03/31 16h:	4.36666	-1.01255	63.7475	23.7437	77.6723	54.5341	32.1972	73.6739	
2086/03/31 17h:	4.38276	-1.00946	78.7505	11.6871	86.9917	69.5372	19.6958	82.8432	
2086/03/31 18h:	4.39886	-1.00637	93.7536	-0.4742	95.7609	84.5403	6.9485	90.9070	
2086/03/31 19h:	4.41496	-1.00329	108.7567	-12.4446	104.8057	99.5434	-5.7723	98.7937	
2086/03/31 20h:	4.43106	-1.00020	123.7598	-23.8882	114.9676	114.5465	-18.2109	107.3068	
2086/03/31 21h:	4.44716	-0.99711	138.7629	-34.3136	127.2818	129.5496	-30.0254	117.3887	
2086/03/31 22h:	4.46325	-0.99403	153.7660	-42.9194	143.0018	144.5526	-40.6445	130.3572	
2086/03/31 23h:	4.47934	-0.99095	168.7691	-48.4771	162.9683	159.5557	-49.0389	147.9131	
2086/03/31 24h:	4.49543	-0.98786	183.7721	-49.6640	-174.1844	174.5588	-53.5785	170.8384	
2086/06/29 01h:	23.21697	-0.89536	-161.1354	-28.4511	-160.2469	-170.3487	-34.4542	-169.2308	
2086/06/29 02h:	23.21473	-0.89745	-146.1374	-22.9494	-146.2132	-155.3508	-30.3647	-153.6258	
2086/06/29 03h:	23.21248	-0.89953	-131.1395	-15.1410	-134.1897	-140.3529	-23.3758	-140.2945	
2086/06/29 04h:	23.21022	-0.90162	-116.1416	-5.6820	-123.9910	-125.3550	-14.2960	-129.3283	
2086/06/29 05h:	23.20794	-0.90370	-101.1437	4.9102	-115.1668	-110.3570	-3.7970	-120.2798	
2086/06/29 06h:	23.20566	-0.90578	-86.1458	16.2620	-107.2083	-95.3591	7.6462	-112.5884	
2086/06/29 07h:	23.20336	-0.90786	-71.1479	28.1028	-99.5781	-80.3612	19.7138	-105.7293	
2086/06/29 08h:	23.20105	-0.90994	-56.1499	40.2145	-91.6013	-65.3633	32.1855	-99.1922	
2086/06/29 09h:	23.19873	-0.91202	-41.1520	52.3655	-82.1162	-50.3654	44.8914	-92.3338	
2086/06/29 10h:	23.19640	-0.91410	-26.1541	64.1447	-68.2857	-35.3674	57.6542	-83.9273	
2086/06/29 11h:	23.19406	-0.91617	-11.1562	74.2009	-40.7839	-20.3695	70.1175	-70.1773	
2086/06/29 12h:	23.19170	-0.91824	3.8418	77.0699	15.9759	-5.3716	80.3760	-30.9777	
2086/06/29 13h:	23.18933	-0.92031	18.8397	69.4413	57.7014	9.6264	78.0505	47.9355	
2086/06/29 14h:	23.18695	-0.92238	33.8376	58.1941	76.2151	24.6243	66.6527	75.1176	
2086/06/29 15h:	23.18456	-0.92445	48.8355	46.1461	87.2551	39.6222	54.0366	86.5776	
2086/06/29 16h:	23.18216	-0.92652	63.8335	33.9792	95.7635	54.6201	41.2658	94.3426	
2086/06/29 17h:	23.17974	-0.92858	78.8314	21.9770	103.4594	69.6181	28.6076	101.0236	
2086/06/29 18h:	23.17732	-0.93065	93.8294	10.3533	111.1847	84.6160	16.2285	107.5951	
2086/06/29 19h:	23.17488	-0.93271	108.8273	-0.6537	119.5207	99.6140	4.3078	114.6374	
2086/06/29 20h:	23.17243	-0.93477	123.8252	-10.7267	128.9858	114.6119	-6.9114	122.6564	
2086/06/29 21h:	23.16997	-0.93683	138.8232	-19.4223	140.0736	129.6098	-17.0713	132.1911	
2086/06/29 22h:	23.16750	-0.93888	153.8211	-26.1450	153.1390	144.6078	-25.6432	143.7976	
2086/06/29 23h:	23.16501	-0.94094	168.8191	-30.2058	168.0954	159.6057	-31.9018	157.8282	
2086/06/29 24h:	23.16251	-0.94299	183.8170	-31.0416	-175.9036	174.6037	-35.0391	173.9381	
2086/09/27 01h:	-1.61061	2.21709	-158.0229	-50.2921	-144.1586	-167.2362	-57.7214	-155.5727	
2086/09/27 02h:	-1.62681	2.22067	-143.0193	-41.6934	-126.3642	-152.2327	-50.2251	-133.2888	
2086/09/27 03h:	-1.64300	2.22424	-128.0158	-31.1150	-113.0950	-137.2291	-39.8127	-117.9107	
2086/09/27 04h:	-1.65920	2.22782	-113.0122	-19.5289	-102.5272	-122.2255	-27.9932	-106.7359	
2086/09/27 05h:	-1.67539	2.23139	-98.0086	-7.4793	-93.3240	-107.2219	-15.5149	-97.7514	
2086/09/27 06h:	-1.69159	2.23496	-83.0050	4.6836	-84.5312	-92.2184	-2.7753	-89.7185	
2086/09/27 07h:	-1.70779	2.23853	-68.0015	16.6623	-75.3302	-77.2148	9.9562	-81.7605	
2086/09/27 08h:	-1.72398	2.24210	-52.9979	28.1037	-64.8164	-62.2112	22.4176	-73.0501	
2086/09/27 09h:	-1.74018	2.24567	-37.9943	38.4690	-51.8020	-47.2077	34.2452	-62.5370	
2086/09/27 10h:	-1.75637	2.24923	-22.9908	46.8435	-34.8036	-32.2041	44.8023	-48.6551	
2086/09/27 11h:	-1.77257	2.25279	-7.9872	51.8066	-12.9804	-17.2005	52.8873	-29.3315	
2086/09/27 12h:	-1.78876	2.25636	7.0164	51.9574	11.4273	-2.1970	56.5910	-3.9904	
2086/09/27 13h:	-1.80495	2.25992	22.0199	47.2430	33.5039	12.8066	54.4930	22.4239	
2086/09/27 14h:	-1.82115	2.26348	37.0235	39.0217	50.7741	27.8101	47.4513	43.5972	
2086/09/27 15h:	-1.83734	2.26703	52.0270	28.7377	63.9712	42.8137	37.4451	58.8226	
2086/09/27 16h:	-1.85353	2.27059	67.0306	17.3335	74.5817	57.8173	25.8937	70.1042	
2086/09/27 17h:	-1.86972	2.27414	82.0341	5.3633	83.8133	72.8208	13.5610	79.1922	
2086/09/27 18h:	-1.88592	2.27769	97.0377	-6.8148	92.5803	87.8244	0.8658	87.2542	
2086/09/27 19h:	-1.90211	2.28124	112.0412	-18.9053	101.6987	102.8279	-11.9150	95.1506	
2086/09/27 20h:	-1.91830	2.28479	127.0448	-30.5690	112.1071	117.8315	-24.5278	103.7126	
2086/09/27 21h:	-1.93449	2.28834	142.0483	-41.2845	125.1194	132.8350	-36.6336	114.0341	
2086/09/27 22h:	-1.95068	2.29188	157.0519	-50.1189	142.5747	147.8385	-47.6271	127.8733	
2086/09/27 23h:	-1.96687	2.29543	172.0554	-55.4860	165.8896	162.8421	-56.2733	147.9265	
2086/09/27 24h:	-1.98306	2.29897	187.0590	-55.6679	-167.4222	177.8456	-60.3582	175.6434	

Appendix

A Critical Review of Four Calculation Methods

A.1 Outline of the Four Methods

In building environmental engineering field in Japan, there are two famous calculation methods for determining the sun position, that have been refered in text books used in educational courses for college students in Japan: one is (a) Yamazaki method [12], [13]; the another is (b) Akasaka method [14]–[17]. Akasaka Method has been revised recently [17]. Thus we adopt the newest formulae for computer programing here. Additionally, (c) NREL method is alse popular because it is used for constructing weather data for the building energy calculation program EnergyPlus (EPW). Thus, we compare these three methods with (d) Matsumoto method (Rev. 2022) here.

Table 7 Summary of Four Calculation Methods for the Sun Position

Method Name	**Yamazaki** [13]	**Akasaka** [17]
Developer(s) Published Year	H. Yamazaki (1980)	H. Akasaka (1992, Rev.2008)
Primary Theory Time Unit	S. Newcomb (1906) [10] Ephemeris Time, ET	
Modification	H. Yamazaki	H. Akasaka
Coord. System	Geocentric equatorial system	
Coord. Conv.	—	
Description of $\Delta\ T$	N/A	
Remark/Note	Long-term change of the obliquity of the ecliptic is considered.	Simplified method after Yamazaki (1980).
Method Name	**(DOE) NREL** [21]	**Matsumoto** [1]
Developer(s) Published Year	I. Reda & A. Andreas (2003, Rev.2008)	S. Matsumoto (Rev.2022)
Primary Theory Time Unit	VSOP87 by P. Bretagnon & G. Francou (1988) [20] Terrestrial Time, TT	
Modification	J. Meeus [22]	HG & OG Dept., JCG*[19]
Coord. System	Heliocentric ecliptic system	
Coord. Conv.	Proc. by J. Meeus	Proc. by S. Matsumoto
Description of $\Delta\ T$	data in AA [25]?	expressed in equations
Remark/Note	No data and no recommendation on ΔT.	ΔT is proposed in equation forms for the years 1600-2100 after NASA [24].

* Hydrographic & Oceangraphic Department, Japan Coast Guard

Four methods (a)–(d) are summarized in Table 7. Yamazaki method and Akasaka method are based on classical theory by Newcomb. As mentioned latter, the theory is out of date from global scientific standard. These may be acceptable from viewpoint of education.

A.1.1 Two Methods Based on Newcomb's Theory

The astronomical theory by Newcomb [10] is based on time system called "Ephemeric Time" (ET), that is out of standard in International Astronimical Union. Definition of the vernal equinox point for the equatorial coordinate system's origin is also too old. We can not recommend to use this theory now. However, the calculation is relatively simple and have enough precision for some practical purposes. Meeus says that ET is the similar with our current standard time system "Terrestial Time" (TT) [22]. However, now we don't have any information on the diffence between ET and UT, that is our ordinal citizen time system.

Anyway, TT is now common time parameter in positional astronomy field after ET and "Terestial Dynamical Time" TD discussed in last four decades. Thus we should watch carefully the the difference between TT and UT, *i.e.,* ΔT.

A.1.2 Two Methods Based on VSOP87

VSOP87 is high grade (precise) and comlicated (detailed) calculation theory for expressing planets positions in the ecliptic coordinate system [20]. The expression includes over 2000 cosine function terms with coefficients. Because the whole program source code is open to the public in fee-free, the theory may be de facto standard grobally although its complexity. However, its complexity consumes a lot of memory and time in conputation process. So some simplified calculation procedures have been proposed. We can say that Meeus method is a typical simplified method fo VSOP87, that is completely followed by NREL method [21]. Generally in building engineering field, it is known that the NREL method is precise. However, we can point out two weakness: (i) There is no explicit instruction how to apply dT value in NREL method. The program code is open but some users may not have any concern internal action. When user use NREL method blindly like a black box, the precision may be lost.; (ii) The conversion procedure from the ecliptic coordinate system to the equatorial coordinate system that we prefer to use is too complex to use plainly. In order to use it enough, you may understand the source code[*10]

The simplification of VSOP87 is studied in Japan, too. The methods by Hydrographic and Oceanographic Department of Japan Coast Guard [19], [19], [23] may be typical works. The works were applied in "Matsumoto method" and "Matsumoto Method (Rev. 2022)" partially.

A.2 Comparison of Calculation Results of the Solar Declination δ and the Equation of Time T_e

A.2.1 Method of Comparison

We compare calculation performance of the four methods reviewed in Section A.1 here. We assume that the book named "Chronological Scientific Tables" by National Astronomical Observatory, Japan [18] includes the most detailed data calculated by VSPO87, then the data of the solar declination δ [°] and the equation of time T_e [min.] are considered as the true data. And calculated data by each method are evaluated by three indices: XBE (maximum data of absolute error), MBE (mean bias error), and RMSE (roor mean square error).

The original data treated as the true data are daily data for 40 years from 1981 to 2020 ($N = 14,610$) [18]. The data were tuned by the authers[*11].

As mentioned above, NREL method's program source code is open to the public but we make source code originally with refering the published code. Of course, the validation was done by using example results reported in the paper [21].

[*10] We remark that even in VPOP87's sourcr code some basic computation techniques to avoid information loss are not taken into account, unbelievably. You know that the last line equation expressed in Müller's polynomial form is a clasical but essential technique to be applied in Eq. (9) for example.

[*11] There is one clear misprint data, that were replaced to the interpolated value. The solar declination data were prepared as integer with arcsec. unit. And the equation of time data were prepared as integer with ds(deci-sec.) unit. The original data for the first four years (1981–1984) are based on ET instead of UT. Of course, the difference was taken into account for each method's calculation.

A.2.2 Comparison and Evaluation of the Calculated Results

The evaluated indices are summarized in Table 8 and Fig. 6 . It is found that Matsumoto method (Rev. 2022) and NREL method, that are based on VSOP87 have the better accuracy than Yamazaki and Akasaka methods based on Newcomb's theory. Although Matsumoto method has smaller operation steps than one NREL mwethod has, the accuracy of Matsumoto method is not so worse than NREL method.

As explanation above, the precision of "Matsumoto method (Rev. 2022)" was evaluated and expected to be applied for the far future like year 2086.

Table 8 Accuracy analysis for four calculation methods

(a) Error of the solar declination δ [″(arcsec)]

Index∗	Yamazaki	Akasaka	NREL	Matsumoto
XBE	19.0	18.0	2.0	3.0
MBE	−0.2	−0.7	0.0	0.0
RMSE	7.0	6.5	0.7	0.9

(b) Error of the equation of time T_e [s]

Index∗	Yamazaki	Akasaka	NREL	Matsumoto
XBE	2.50	3.00	0.40	0.40
MBE	−0.35	−0.35	0.19	0.07
RMSE	0.74	0.83	0.19	0.15

∗: (Error)=(Estimated Value)−(True Value)

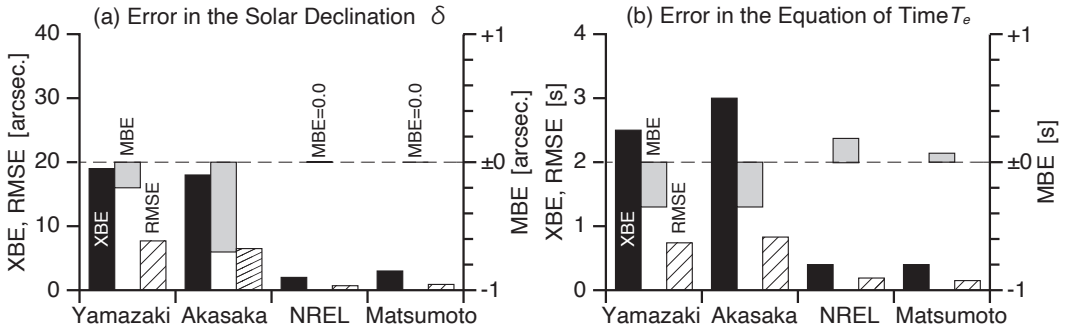


Fig. 6 Comaprison of Calculated Error of Four Methods

B Program Source Codes of Julian Day $\overline{\text{JD}}$

In following two sections, Julian day calculation codes in FORTRAN90 and C/C++ are described. The algorithm for $\overline{\text{JD}}$ calculation is based on Meeus metho [22].

B.1 FORTRAN

```
! ------------------------------------------------
!   Calculation of the Julian Day as long integer
!   "Meeus Method"  (ユリウス通日計算関数, Meeus)
!     YYYY/MM/DD (0 < YYYY, 1 <= MM <= 12,
!                 1 <= DD <= 28, 29, 30, 31)
!   Note:
!     This function does not check the ranges of
!     args.: YYYY, MM, DD
!   Ref.
!     J. Meeus: Astronomical Algorithms (2nd Ed.),
!       pp.59-66, Willmann-Bell, Richmond, 1998.
! --+----*----+----*----+----*----+----*----+----*
function JD(YYYY,MM,DD)
  implicit none
  integer   YYYY,MM,DD
  integer*8 JD, GREG0, jul, GREG1
  integer   ja, jm, jy
!
! Gregorian Calendar adopted in Oct. 15, 1582.
  GREG0 = GREG(1582,10,15)
!
```

```
  jy = YYYY
  GREG1 = GREG(YYYY,MM,DD)
  if ( 0 > jy ) then
    jy = jy + 1
  end if
  if ( 2 < MM ) then
    jm = MM + 1
  else
    jy = jy - 1
    jm = MM + 13
  end if
  jul = floor(365.25*jy) + floor(30.6001*jm) &
      + DD + 1720995
  if ( GREG0 <= GREG1 ) then
    ja  = int(0.01*jy)
    jul = jul + 2 - ja + int(0.25*ja);
  end if
  JD = jul
!
  contains
    function GREG(Y,M,D)
      integer   Y, M, D
      integer*8 GREG
      GREG = D+31*(M+12*Y)
    end function ! GREG
end function
```

B.2 C/C++

```
// ------------------------------------------------
//      Reference day number macro for comparison
#define LGREG(Y,M,D) ((D)+31L*((M)+12L*(Y)))
#define GREG0 LGREG(1582,10,15)
//      Gregorian Calendar's adopted date in 16C
typedef unsigned int   uint;
typedef unsigned long ulong;
// ------------------------------------------------
ulong JD0( uint YYYY, uint MM, uint DD )
//  Calculation of the Julian Day as long integer
//  "Meeus Method"
//   (ユリウス通日を求める関数, Meeus による)
//    YYYY/MM/DD (0 < YYYY, 1 <= MM <= 12,
//                1 <= DD <= 28, 29, 30, 31)
//  Note:
//    This function does not check the ranges of
//    args.: YYYY, MM, DD
//  Ref.
//    J. Meeus: Astronomical Algorithms (2nd Ed.),
//      pp.59-66, Willmann-Bell, Richmond, 1998.
{
  int   ja, jm, jy = int(YYYY);
  long  jul;
  long GREG1 =
    LGREG( long(YYYY), long(MM), long(DD) );
  if ( 0 > jy ) ++jy;
  if ( 2 < int(MM) )  jm = int(MM) + 1;
  else {
    --jy; jm = int(MM) + 13;
  }
  jul = long(floor(365.25*jy)
           + floor(30.6001*jm) + aDate.DD)
      + 1720995L;
  if ( GREG0 <= GREG1 ) {
    ja   = int(0.01*jy);
    jul += 2 - ja + int(0.25*ja);
  }
  return ( ulong(jul) );
}
```

B.3 Reference Table

The following table is a series of data to be helpful for validation works. The days with $*$ mark have not 12^{h}UT. Thus, Julian day for such day is not integer $\overline{\mathrm{JD}}$ but real JD including decimal value.

Table 9 Some Example Data of Julian Day

2000	Jan.	1	2 451 545	1600	Dec.	31.0$*$	2 305 812.5
1999	Jan.	1.0$*$	2 451 179.5	837	Apr.	10.3$*$	2 026 871.8
1987	Jan.	27.0$*$	2 446 822.5	-123	Dec.	31.0$*$	1 676 496.5
1987	June	19	2 446 966	-122	Jan.	1.0$*$	1 676 497.5
1988	Jan.	27.0$*$	2 447 187.5	-1000	July	12	1 356 001
1988	June	19	2 447 332	-1000	Feb.	29.0$*$	1 355 866.5
1900	Jan.	1.0$*$	2 415 020.5	-1001	Aug.	17.9$*$	1 355 671.4
1600	Jan.	1.0$*$	2 305 447.5	-4712	Jan.	1	0

$*$: With consideration of time UT or TT

C Classical Program Source Codes for the Sun Position Parameters

In this section, two classical calculation methods' program source codes of parameters to determine the sun position: Yamazaki method [12], [13] and Akasaka mehtod [**?**]–[16], are introduced.

The both codes were written in FORTRAN90. Similar functions were written in C/C++ in Appendix D.

By the way, the calculation examples in above (Appendix A) were calculated by the C/C++ program mentioned in Appendix D latter.

C.1 FORTRAN Code for Yamazaki Method

```
! ------------------------------------------------
!   Calculation of the Solar Declination
!   and the Equation of Time "Yamazaki Method"
!    (赤緯と均時差の計算サブルーチン，山崎による)
!   (output)
!    DELTA: Solar Declination [rad.]
!       ET: Equation of Time [rad.]
!   (input)
!       YY: Year, MM: Month, DD: Day
!       TT: Hour in ZST
!   LONGIT: Reference Longitude for ZST in [deg.]
!   Note:
!     This subroutine does not check the ranges
!     of the inputarguments.
!   Ref.
!     Hitoshi Yamazaki: Fundamental Formulae for
!       Daylighting   IV (Precise Program to
!       Calculate Solar Declination   and Equation
!       of Time) (in Japanese), Proc. of AIJ
!       Annual Meeting 1980, pp.407--408, 1980.9.
!     山崎　均：日照環境の基礎計算式 IV（対象
!       地域の太陽視赤緯及び均時差を正確に計算
!       するプログラム），日本建築学会大会学術
!       講演梗概集，計画系，pp.407--408，1980.9.
! --+----*----+----*----+----*----+----*----+----*
subroutine SUN(DELTA,ET,YY,MM,DD,TT,LONGIT)
  implicit none
  integer YY, MM, DD
  real DELTA, ET, LONGIT, TT
  real M, dM, A, D, D2, D3, ET1, ET2
  real t1, t2, t3, DELTA0, E, E0, EPS
  real SD, CD, V, VEPS2, YN
  real SinM, Sin2M, Sin3M
  integer M1,i
  integer :: MONTH(12) = &
    (/ 31,28,31,30,31,30,31,31,30,31,30,31 /)
  real :: RAD  = 1.7453292e-2
  real :: DEL0 = 23.4522
!
  YN = float(YY) - 1900.0 ! Elasped yr after 1900
  D2 = aint((YN - 1.)/4.)
  D3 = aint(YN/4.)
  MONTH(2) = 28 + ifix(D3 - D2)
  D = float(DD) + (YN - 30.) * 1.1574e-5
  D = D + (TT - 12.) / 24. - LONGIT / 15. / 24.
  M1 = MM - 1
  if ( M1 >= 1 ) then
    do i=1, M1
      D = D + float(MONTH(i))
    end do
  end if
!
! Calculation of Julian Century from JD1900.0
  t1 = (365. * YN + D2 + D) / 36525.
  t2 = t1 * t1
  t3 = t2 * t1
!
! Calculation of the Zodiac Tilt Angle DELTA0
!  黄道傾斜角
  DELTA0 = -9.44e-5 + 1.30125e-2 * t1 &
           + 1.64e-6 * t2 + 5.0e-7 * t3
  DELTA0 = DELTA0 - 23.4522
  DELTA0 = DELTA0 * RAD
! Altanative DELTA0 for year 2000
  DELTA0 = -DEL0 * RAD
!
! Calculation of Eccentricity (E) 離心率
  E = 1.04e-6 - 4.18e-5 * t1 - 1.26e-7 * t2 &
    + 0.01675
!
! Angle (rad.) bwetween Perihelion and
! Winter Solstice (EPS)
  EPS = 0.719175 * t1 + 0.000453 * t2
  EPS = EPS + 11.220833 + t1
  EPS = EPS * RAD
!
! Calculation of Mean Perigee Elongation (M)
! 平均近点離角
  M = 6.00267e-4 * (D2 + D) &
    - 9.02579e-4 * YN &
    - 0.00015 * t2 - 1.667e-4
  M = M - 1.524 - 0.255 * YN + 0.985 * D2
  M = M + 0.985 * D
  M = M * RAD
  SinM   = sin(M)
  Sin2M  = sin(2.0 * M)
  Sin3M  = sin(3.0 * M)
!
  dM = 9.93502e-5 * &
    (1.0 - E * (cos(M) - 2.0 * E * SinM * SinM))
  M = M - dM
!
  V = (2.0 - 0.25 * E * E) * E * SinM
  V = V + 1.25 * E * E * Sin2M
  V = V + 13.0 / 12.0 * E * E * E * Sin3M
  v = V + M
!
  SD = cos(EPS + V) * sin( DELTA0 )
  CD = sqrt(1.0 - SD * SD)
  VEPS2 = 2.0 * (EPS + V)
  A = (1.0 - cos(DELTA0)) / (1.0 + cos(DELTA0))
!
! Solar Declination 視赤緯
  DELTA = atan( SD / CD )
!
! Equation of Time 均時差
  ET1 = M - V;
  ET2 = -atan( (A * sin(VEPS2)) &
             / (1.0 - A * cos(VEPS2)) )
  ET = ET1 + ET2;
!
  return
end subroutine
```

C.2 FORTRAN Code for Akasaka Method

```
! ------------------------------------------------
!  Calculation of the Solar Declination
!  and the Equation of Time "Akasaka Method"
!   (赤緯と均時差を求めるサブルーチン，赤坂 (2022)
!    による)
!  (output)
!    SINDLT: Sine of Solar Declination [-]
!    COSDLT: Cosine of Solar Declination [-]
!        ET: Equation of Time [deg.]
!  (input)
!    YEAR: Year
!    NDAY: Serial day number of the YEAR
!  Note:
!    This subroutine does not check the ranges
!    of the input arguments.
!  Ref.
!    H. Akasaka: Simplified Calculation Method of
!     the Sun Position with Secular Changes,
!     General Technical Report on the EA Weather
!     Data. MetDS HP. 2022.8
!    赤坂　裕：年差を考慮した太陽位置の簡易計算，
!      EA 気象データ技術解説一般，MetDS HP，2022.8.
!
! --+----*----+----*----+----*----+----*----+----*
subroutine SUNLD(YEAR,NDAY,SINDLT,COSDLT,ET)
  implicit none
  integer YEAR, NDAY
  real SINDLT, COSDLT, ET
  real D0, M, M1, N
  real V, RAD, DLT0, EPS, VEPS, VE2
!
  RAD  = 3.141592 / 180.
!
  N = float(YEAR) - 1968.
!
  DLT0 = (-23.4393 + 0.00013 * &
         (FLOAT(YEAR) - 2000.)) * RAD
  D0 = 3.71 + 0.2596 * N - int((N + 3.) / 4.)
!
  M = 0.9856 * (NDAY - D0)
  EPS = 12.3901 + 0.0172 * (N + M / 360.)
  V = M + 1.914 * SIN(M * RAD) &
        + 0.02 * SIN(2.* M * RAD)
  VEPS = (V+EPS) * RAD
  VE2  = 2. * VEPS
!
  ET  = (M - V) &
      - ATAN(0.043 * SIN(VE2) &
      / (1.-0.043 * COS(VE2))) / RAD
!
      SINDLT = COS(VEPS) * SIN(DLT0)
      COSDLT = SQRT(ABS(1. - SINDLT * SINDLT))
!
  return
end subroutine
```

D Program Source Codes for the Sun Position Parameters by Matsumoto Method (Rev. 2022)

D.1 FORTRAN Code

The source code described below is a kind of translation from C/C++ code mentioned in Section D.2 latter. We understand that some variables are not need to have structure types in FORTRAN code. They are C/C++ oriented. In FORTRAN, they may be strange, inefficient, counterproductive, and risky. However, we hope that the source code is something helpful for you.

```
! ------------------------------------------------
!   SolPos.f90
!     Calculation Subroutines and functions
!     for Sun Position Parameters:
!       Solar Declination 太陽赤緯 [deg.] and
!       Equation of Time  均時差   [deg.]
!
!   Coded by Shin-ichi Matsumoto, 2022
!
!   Proofed by gfortran
!
! (c) 2022, Shin-ichi Matsumoto, Akita Pref. Univ.
! --+----*----+----*----+----*----+----*----+----*
module mGrobal
  implicit none
  integer, parameter :: YEAR_GREG = 1582
!              Gregorian calendar adopted in 1582
  integer*8, parameter ::  JD2000 = 2451545
!              Julian day of Jan. 1 12TT, 2000
  double precision, parameter :: pi=3.14159265d0
  integer, parameter :: DAYS_OF_MONTH(12) = &
    (/ 31,28,31,30,31,30,31,31,30,31,30,31 /)
!
  logical, parameter :: T_ = .true.
  logical, parameter :: F_ = .false.
!
  type TC
    logical :: T
    double precision :: A, B, C
  end type TC
!
  type TJday
    integer*8 :: Value
  end type TJday
!
  type TTday
    integer :: Value
  end type TTday
!
  type TIntDate
    integer :: Year, Month, Day
  end type TIntDate
!
  integer :: C_MAX = 18
  type(TC), save :: CLC(18) = (/ &
  TC(T_, 36000.7695d0, 0.0000000d0, 0.0000000d0),&
  TC(F_, 280.465900d0, 0.0000000d0, 0.0000000d0),&
  TC(F_, 1.91470000d0, 35999.050d0, 267.52000d0),&
  TC(F_, 0.02000000d0, 71998.100d0, 265.10000d0),&
```

```
  TC(T_,-0.00480000d0, 35999.000d0, 268.00000d0),&
  TC(F_, 0.00200000d0, 32964.000d0, 158.00000d0),&
  TC(F_, 0.00180000d0, 19.000000d0, 159.00000d0),&
  TC(F_, 0.00180000d0, 445267.00d0, 208.00000d0),&
  TC(F_, 0.00150000d0, 45038.000d0, 254.00000d0),&
  TC(F_, 0.00130000d0, 22519.000d0, 352.00000d0),&
  TC(F_, 0.00070000d0, 65929.000d0, 45.000000d0),&
  TC(F_, 0.00070000d0, 3035.0000d0, 110.00000d0),&
  TC(F_, 0.00070000d0, 9038.0000d0, 64.000000d0),&
  TC(F_, 0.00060000d0, 33718.000d0, 316.00000d0),&
  TC(F_, 0.00050000d0, 155.00000d0, 118.00000d0),&
  TC(F_, 0.00050000d0, 2281.0000d0, 221.00000d0),&
  TC(F_, 0.00040000d0, 29930.000d0, 48.000000d0),&
  TC(F_, 0.00040000d0, 31557.000d0, 161.00000d0) &
  /)
!
  integer, save :: D_MAX = 9
  type(TC), save :: SDC(9) = (/ &
  TC(F_, 1.00014000d0, 0.0000000d0, 0.0000000d0),&
  TC(F_, 0.01670600d0, 35999.050d0, 177.53000d0),&
  TC(F_, 0.00013900d0, 71998.000d0, 175.00000d0),&
  TC(T_,-0.00004200d0, 35999.000d0, 178.00000d0),&
  TC(F_, 0.00003100d0, 445267.00d0, 298.00000d0),&
  TC(F_, 0.00001600d0, 32964.000d0, 68.000000d0),&
  TC(F_, 0.00001600d0, 45038.000d0, 164.00000d0),&
  TC(F_, 0.00000500d0, 22519.000d0, 233.00000d0),&
  TC(F_, 0.00000500d0, 33718.000d0, 226.00000d0) &
  /)
!
contains
  function DegToRad(D)
    implicit none
    double precision D, DegToRad
    DegToRad = D * (pi / 180.d0)
  end function DegToRad
!
  function RadToDeg(R)
    implicit none
    double precision R, RadToDeg
    RadToDeg = R * (180.d0 / pi)
  end function RadToDeg
!
  function JC2000(JDE)
    implicit none
    integer*8 JDE
    double precision JC2000
    JC2000 = dble(JDE - JD2000) / 36525.d0
  end function JC2000
end module mGrobal
! ------------------------------------------------
! Calendar functions and subroutines
function Is_Leap(YY)
  use mGrobal
  implicit none
  logical Is_Leap, Ret
  integer YY
!
  if ( YY <= YEAR_GREG ) then
    Ret = .false.
  else
    if ( 0 == mod(YY,400) ) then
      Ret = .true.
    else
      if ( 0 == mod(YY,100) ) then
        Ret = .false.
      else
        if ( 0 == mod(YY,4) ) then
          Ret = .true.
        else
          Ret = .false.
        end if
      end if
    end if
  end if
!
  Is_Leap = Ret
!
end function
! ------------------------------------------------
subroutine CorrectDate(YY,MM,DD,Done)
  use mGrobal
  implicit none
  integer YY, MM, DD
  integer m, DOM(12)
  logical Done, chk, leap, Is_Leap
!
  chk = .true.
  if ( 0 == YY ) then
    YY =  1; chk = .false.
  end if
  if (  1 > MM ) then
    MM =  1; chk = .false.
  end if
  if ( 12 < MM ) then
    MM = 12; chk = .false.
  end if
  if (  1 > DD ) then
    DD =  1; chk = .false.
  end if
  leap = Is_Leap( YY );
  do m=1, 12
    DOM(m) = DAYS_OF_MONTH(m)
  end do
  if ( leap ) then
    DOM(2) = DOM(2) + 1
  end if
  if ( DD > DOM(MM) ) then
    DD = DOM(MM); chk = .false.
  end if
!
  Done = chk
  return
end subroutine
! ------------------------------------------------
subroutine TdayToIntDate(TDay,YY,IDate)
  use mGrobal
  implicit none
  type(TTday) :: TDay, t_day
  type(TIntDate) :: IDate
  integer YY
  integer m, day
  integer days(12)
  logical leap, Is_Leap
!
  leap = Is_Leap(YY);
  day = 0
  do m=1, 12
    day = day + DAYS_OF_MONTH(m);
    if ( leap .and. 2 == m ) then
      day = day + 1
    end if
    days(m) = day
  end do
  t_day%Value = TDay%Value
  if ( 1 > TDay%Value ) then
    t_day%Value = 1
  else
    if ( 365 < TDay%Value ) then
      if ( leap ) then
        t_day%Value = 366
      else
        t_day%Value = 365
      endif
    end if
  end if
!
  IDate%Year = YY
  do m=1, 12
    if ( t_day%Value <= days(m) ) then
      IDate%Month = m
      exit
    end if
  end do
  IDate%Day = t_day%Value
  if ( 1 /= IDate%Month ) then
    IDate%Day = IDate%Day &
              - days(IDate%Month - 1)
  end if
!
  return
end subroutine
! ------------------------------------------------
subroutine IntDateToTday(IDate, TDay)
  use mGrobal
  implicit none
  type(TIntDate) :: IDate
  type(TTday) ::    TDay, days
  integer m, DOM(12)
  logical leap, Is_Leap
!
  if ( 1 == IDate%month ) then
    TDay%Value = IDate%Day
```

```
  else
    leap = Is_Leap(IDate%year)
    do m=1, 12
      DOM(m) = DAYS_OF_MONTH(m)
    end do
    if ( leap ) then
      DOM(2) = DOM(2) + 1
    end if
    days%Value = 0
    do m=1, IDate%Month - 1
      days%Value = days%Value + DOM(m)
    end do
    TDay%Value = IDate%Day + days%Value
  end if
!
  return
end subroutine
! vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
!   This part is a little bit different
!   from code described above in the
!   technical report
! -------------------------------------------------
!   Calculation of the Julian Day as long integer
!   "Meeus Method" (ユリウス通日計算関数, Meeus)
!   using TIntDate argument
!   SUBROUTINE STYLE
!   Note:
!     This function does not check the ranges of
!     args.: YYYY, MM, DD
!   Ref.
!     J. Meeus: Astronomical Algorithms (2nd Ed.),
!       pp.59-66, Willmann-Bell, Richmond, 1998.
! --+----*----+----*----+----*----+----*----+----*
subroutine IntDateToJDay(IDate,JDay)
  use mGrobal
  implicit none
  type(TIntDate) :: IDate
  type(TJday) ::    JDay
  integer   YY,MM,DD
  integer*8 JD, GREG0, jul, GREG1
  integer   ja, jm, jy
!
! Gregorian Calendar adopted in Oct. 15, 1582.
  GREG0 = GREG(1582,10,15)
!
  jy = IDate%Year
  YY = IDate%Year
  MM = IDate%Month
  DD = IDate%Day
  GREG1 = GREG( YY, MM, DD )
  if ( 0 > jy ) then
    jy = jy + 1
  end if
  if ( 2 < MM ) then
    jm = MM + 1
  else
    jy = jy - 1
    jm = MM + 13
  end if
  jul = floor(365.25d0*jy) + floor(30.6001d0*jm) &
      + DD + 1720995
  if ( GREG0 <= GREG1 ) then
    ja  = int(0.01d0*jy)
    jul = jul + 2 - ja + int(0.25d0*ja)
  end if
  JDay%Value = jul
!
  contains
    function GREG(Y,M,D)
      integer   Y, M, D
      integer*8 GREG
      GREG = D+31*(M+12*Y)
    end function ! GREG
end subroutine
! -------------------------------------------------
!   Calculation of the Julian Day as long integer
!   "Meeus Method" (ユリウス通日計算関数, Meeus)
!   using TIntDate argument
!   FUNCTION STYLE
!   Note:
!     This function does not check the ranges of
!     args.: YYYY, MM, DD
!   Ref.
!     J. Meeus: Astronomical Algorithms (2nd Ed.),
!       pp.59-66, Willmann-Bell, Richmond, 1998.
! --+----*----+----*----+----*----+----*----+----*
function JD_of( IDate )
  use mGrobal
  implicit none
  type(TIntDate) :: IDate
  type(TJday) ::    JDay
  integer   ja, jm, jy
  integer*8 jul, GREG0, JD_of
!
  call IntDateToJDay( IDate, JDay )
!
  JD_of = JDay%Value
!
  return
end function
! ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
! MAIN REVISION 2022   !
! .................................................
! Time Difference dT in [msec.] integer
!  "NASA Method"
!   YY: year (1600-2150)
! Ref.
!   Fred~Espenak and Jean~Meeus: Five
!     Millennium Canon of Solar Eclipses:
!     -1999 to +3000, The NASA Technical
!     Publication (NASA/TP--2006--214141),
!     NASA, Greenbelt, 2006.10.
! --+----*----+----*----+----*----+----*----+----*
function DeltaTime(YY) ! in [msec.]
  use mGrobal
  implicit none
  type(TIntDate) :: UTDate
  integer  YY, DeltaTime, dt
  UTDate%Year  = YY
  UTDate%Month = 7
  UTDate%Day   = 1
!
  DeltaTime = DeltaTimeEx(UTDate)
!
  contains
    function RoundTo(V)
      implicit none
      double precision V
      integer IV, RoundTo
      logical :: minus = .false.
!
      if ( 0.d0 > V ) minus = .true.
      IV = nint(abs(V) * 1000.d0)
      if ( minus ) then
        IV = -IV
      end if
      RoundTo = IV
!
      return
    end function
!
    function DeltaTimeEx(IDate)
      use mGrobal
      implicit none
      type(TIntDate) :: IDate
      double precision y, t, S
      integer DeltaTimeEx
      integer YY, MM, DD
!
      YY = IDate%Year
      MM = IDate%Month
      DD = IDate%Day
      y = dble(YY) + (dble(MM) - 0.5) / 12.0
      if ( 7 == MM .and. 1 == DD ) then
        y = dble(YY) + 0.5d0
      end if
      if ( 1 == MM .and. 1 == DD ) then
        y = dble(YY)
      end if
!
      if ( 1600 > YY ) then
        DeltaTimeEx = 0
        return
      end if
      if ( 1701 > YY ) then
        t = y - 1600d0
        S =  ((1. / 7129d0 * t  - 0.01532d0) &
          * t - 0.9808d0) * t + 120d0
        DeltaTimeEx = RoundTo(S)
        return
      end if
      if ( 1801 > YY ) then
        t = y - 1700d0
```

```
        S = (((-1d0 / 1174000d0 * t &
          + 0.00013336d0) * t - 0.0059285d0) &
          * t  + 0.1603d0) * t + 8.83d0
        DeltaTimeEx = RoundTo(S)
        return
      end if
      if ( 1861 > YY ) then
        t = y - 1800d0
        S = ((((((0.000000000875d0 * t &
          - 0.0000001699d0) * t + 0.0000121272d0) &
          * t - 0.00037436d0) * t + 0.0041116d0) &
          * t + 0.0068612d0) * t - 0.332447d0) &
          * t + 13.72d0
        DeltaTimeEx = RoundTo(S)
        return
      end if
      if ( 1901 > YY ) then
        t = y - 1860d0
        S =  ((((1d0 / 233174d0 * t &
          - 0.0004473624d0) * t + 0.01680668d0) &
          * t - 0.251754d0) * t + 0.5737d0) &
          * t + 7.62d0
        DeltaTimeEx = RoundTo(S)
        return
      end if
      if ( 1921 > YY ) then
        t = y - 1900d0
        S = (((-0.000197d0 * t + 0.0061966d0) &
          * t - 0.0598939d0) * t + 1.494119d0) &
          * t - 2.79d0
        DeltaTimeEx = RoundTo(S)
        return
      end if
      if ( 1941 > YY ) then
        t = y - 1920d0
        S = ((0.0020936d0 * t - 0.076100d0) &
          * t + 0.84493d0) * t + 21.20d0
        DeltaTimeEx = RoundTo(S)
        return
      end if
      if ( 1961 > YY ) then
        t = y - 1950d0
        S = ((1d0 / 2547d0 * t  - 1d0/233d0) &
          * t + 0.407d0) * t + 29.07d0
        DeltaTimeEx = RoundTo(S)
        return
      end if
      if ( 1986 > YY ) then
        t = y - 1975d0
        S = ((-1d0/718d0 * t  - 1d0/260d0) &
          * t  + 1.067d0) * t + 45.45d0
        DeltaTimeEx = RoundTo(S)
        return
      end if
      if ( 2006 > YY ) then
        t = y - 2000d0
        S = ((((0.00002373599d0 * t &
          + 0.000651814d0) * t + 0.0017275d0) &
          * t - 0.060374d0) &
          * t + 0.3345d0) * t + 63.86d0
        DeltaTimeEx = RoundTo(S)
        return
      end if
      if ( 2051 > YY ) then
        t = y - 2000d0
        S = (0.005589d0 * t + 0.32217d0) &
          * t + 62.92d0
        DeltaTimeEx = RoundTo(S)
        return
      end if
      if ( 2151 > YY ) then
        t = y - 1820d0
        S = (0.0032d0 * t + 0.5628d0) &
          * t - 205.724d0
      else
        S = 328.48d0
      end if
      DeltaTimeEx = RoundTo(S)
      return
    end function DeltaTimeEx
end function DeltaTime
! ..............................................
! ----------------------------------------------
! Calculation of the obliquity of the ecliptic
! for a given Julian century JC based on J2000.0
! (黄道傾角)
!   Note:
!     On consideration of the nutation, IsPrec
!     is a switch.
!       = true (default): with consideration
!       = false: without consideration
!     Return value is in deg.
!     Hereafter, abbr. JC2000 for a Julian century
!     based on the J2000.0.
! --+----*----+----*----+----*----+----*----+----*
function Obliquity( JC, IsPrec )
  use mGrobal
  implicit none
  double precision JC, Obliquity, Eps, T1, T2
  logical IsPrec
!
  Eps = (( -5.0361111d-7  * JC &
          + 1.6388889d-7) * JC &
          + 1.3004167d-2) * JC - 2.343929d+1
  if ( IsPrec ) then
    T1 = DegToRad( &
         dmod( 1934d0 * JC + 235d0, 360d0) )
    T2 = DegToRad( &
         dmod(72002d0 * JC + 201d0, 360d0) )
    Eps = Eps - 0.00256d0 * cos( T1 ) &
              - 0.00015d0 * cos( T2 )
  end if
  Obliquity = Eps
!
  return
end function
! ----------------------------------------------
! Calculation of the solar celestial longitude
! for a given JC2000 (視黄経または平均黄経)
!   Note:
!    If IsApp = .false., then the mean value is
!    returned. Return value is in deg.
! --+----*----+----*----+----*----+----*----+----*
function CLongit( JC, IsApp )
  use mGrobal
  implicit none
  double precision    JC, CLongit
  logical IsApp
  double precision    cl, A, BT_C
  integer i
!
  cl = 0.0d0
  do i=C_MAX, 3, -1
    ! DON'T CHANGE the loop INDEX ORDER
    ! to avoid information loss!
    if ( CLC(i)%T ) then
      A = CLC(i)%A * JC
    else
      A = CLC(i)%A
    end if
    BT_C = CLC(i)%B * JC + CLC(i)%C
    BT_C = DegToRad( dmod(BT_C, 360d0) )
    cl = cl + A * cos( BT_C );
  end do
  cl = cl + CLC(2)%A + CLC(1)%A * JC
  if ( IsApp ) then
    A    = 0.0048d0
    BT_C = 1934d0 * JC + 145d0
    BT_C = DegToRad( dmod(BT_C, 360d0) )
    cl = cl + A * cos( BT_C )
    A    = -0.0004d0
    BT_C = 72002d0 * JC + 111d0
    BT_C = DegToRad( dmod(BT_C, 360d0) )
    cl = cl + A * cos( BT_C ) - 0.0057d0
  end if
  cl = dmod(cl, 360d0);
  if ( 0.0 <= cl ) then
    CLongit = cl
  else
    CLongit = cl + 360d0
  end if
!
  return
end function
! ----------------------------------------------
! Calculation of the mean solar right ascension
! for a given JC2000 (UT). (平均太陽の赤経)
!   Note:
!     Julian century UTJC should be calculated
!     with UTC. Return value is in [h].
! --+----*----+----*----+----*----+----*----+----*
function MAscens( UTJC )
  use mGrobal
  implicit none
  double precision UTJC, MAscens, Sec, Hour
!
```

```
  Sec = ((-0.0000062d0*UTJC + 0.093104d0)*UTJC + &
         8640184.812866d0)*UTJC + 67310.54841d0
  Hour = Sec / 3600d0
  Hour = dmod(Hour, 24d0)
  if ( 0.0 <= Hour ) then
    MAscens = Hour
  else
    MAscens = Hour + 24d0
  end if
!
  return
end function
! ------------------------------------------------
! Calculation of the equation of equinoxes for a
! for a given JC2000 (JC). (分点差)
!   Note:
!     Return value is in [h]. Based on the eq.:
!       dL * cos(eps) / 15.
!       where dL is the nutation of the
!       celestial longitude[deg.] and eps is the
!       apparent obliquity of the ecliptic.
! --+----*----+----*----+----*----+----*----+----*
function Eq( JC )
  use mGrobal
  implicit none
  double precision JC, Eq, X, Obliquity
!
  X = 0.0048 * sin( DegToRad( &
      dmod(1934d0 * JC + 235d0, 360d0) ) ) &
    - 0.0004 * sin( DegToRad( &
      dmod(72002d0 * JC + 201d0, 360d0) ) )
  X = X * cos( DegToRad(Obliquity(JC,.true.)) )
  Eq = X / 15d0
!
  return
end function
! ------------------------------------------------
! Calculation of the geocentric distance to
! the sun for a given JC2000 (JC). (地心距離)
!   Note:
!     Return value is in AU.
! --+----*----+----*----+----*----+----*----+----*
function SolarDistance( JC )
  use mGrobal
  implicit none
  double precision JC, SolarDistance, sd, A, BT_C
  integer i
!
  sd = 0.0d0
  do i=D_MAX, 2, -1
    ! DON'T CHANGE the loop INDEX ORDER
    ! to avoid information loss!
    if ( SDC(i)%T ) then
      A = SDC(i)%A * JC
    else
      A = SDC(i)%A
    end if
    BT_C = SDC(i)%B * JC + SDC(i)%C
    BT_C = DegToRad( dmod(BT_C, 360d0) )
    sd = sd + A * cos( BT_C );
  end do
  sd = sd + SDC(1)%A
  SolarDistance = sd
!
  return
end function
! ------------------------------------------------
! Calculation of the solar declination and the
! equation of time (solar parameters)
! "Matsumoto Method (Rev. 2022)" (視赤緯と均時差)
! SUBROUTINE STYLE
! (input)
!  YYYY: a given year (1600-2150)
!  SDAY: serial day number of YYYY (1-365, 366)
!    UT: a given time in UTC.
! (output)
!          D: the solar declination in deg.
! SinD, CosD: sin(D), cos(D)
!          R: the solar distance in AU.
!         ET: the equation of time in deg.
! --+----*----+----*----+----*----+----*----+----*
subroutine SolarParams( YYYY, SDAY, UT, &
                        SinD, CosD, D, R, ET )
  use mGrobal
  implicit none
  integer YYYY, SDAY, DeltaTime
  double precision    UT, SinD, CosD, D, R, ET
  double precision    Hr, T, Tu, JC, Del, L,
  double precision    Sd, Cd, Dec, Ta, Tam, ET0
  double precision, save :: dHr = 0.0d0
  integer, save :: pYear = 0
  type(TIntDate) theDate
  type(TTDay) TDay
  integer*8 JD_of, JD
  double precision Obliquity, Clongit
  double precision MAscens, Eq, SolarDistance
!
  TDay%Value = SDAY
  call TdayToIntDate( TDay, YYYY, theDate )
!
  Hr = UT
  if ( pYear /= YYYY ) then
    dHr = DeltaTime( YYYY ) * 0.001d0 / 3600d0
    pYear = YYYY
  end if
  JD = JD_of( theDate )
  JC = JC2000( JD )
  T  = JC &
    + (Hr + dHr - 12d0) / (24d0*36525d0)
  Tu = JC &
    + (Hr - 12d0) / (24d0*36525d0)
  Del = Obliquity( T, .true. )
  L   = CLongit( T, .true. )
  Del = DegToRad( Del )
  L   = DegToRad( L )
  Sd  = -sin( L ) * sin( Del )
  Cd  = sqrt( 1.0 - Sd * Sd )
  Dec = atan( Sd / Cd ) ! in RAD.
  Ta  = tan( L ) * cos( Del )
  Tam = tan( DegToRad( 15d0*MAscens( Tu )) )
  ET0 = RadToDeg( &
    atan( (Tam - Ta) / (1d0 + Tam * Ta) ) )
  ET  = ET0 + Eq( T ) * 15d0
  SinD = sin( Dec )
  CosD = cos( Dec )
  D = RadToDeg( Dec )
  R = SolarDistance( T )
!
  return
end subroutine
! ------------------------------------------------
! Calculation of the sun position
! "Matsumoto Method (Rev. 2022)" (太陽位置)
! SUBROUTINE STYLE (JAPAN ONLY)
! (input)
!   Phi: Latitude of a given location in deg.
!   Ell: Longitude of a given location in deg.
!   JST: Hour in JST
!    ET: Eauation of time calculated in deg.
!  SinD: Sin of the solar declination calculated
!  CosD: Cosine of the solar declination calculated
! (output)
!  SinH, CosH: Sin and Cosine of the solar altitude
!  SinA, CosA: Sin and Cosine of the solar azimuth
! --+----*----+----*----+----*----+----*----+----*
subroutine SolarPosition( Phi, Ell, JST, ET, &
             SinD, CosD, SinH, CosH, SinA, CosA )
  ! This function is just for JAPAN.
  ! You can change 135.0 in the following
  ! code to be a lontitude value for
  ! reference one in your time zone.
  use mGrobal
  implicit none
  double precision Phi, Ell, JST, ET, SinD, CosD
  double precision SinH, CosH, SinA, CosA, T, P
!
  T = DegToRad(15d0 * (JST - 12d0) &
    + (Ell - 135d0) + ET);
  P = DegToRad(Phi);
  SinH = sin( P ) * SinD &
    + cos( P ) * CosD * cos( T )
  CosH = sqrt( abs(1.0 - SinH*SinH) )
  SinA = CosD * sin( T ) / CosH;
  CosA = (SinH * sin( P ) - SinD) &
    / (CosH * cos( P ));
end subroutine
!
! ------------------------------------------------
! THE CODES DESCRIBED IN THE ABOVE CAN BE USE,
! COPIED, AND MODIFIED WITHOUT ANY ACCEPTANCE OF
! THE AUTHOR: Dr. Shin-ichi Matsumoto, Prof.,
```

```
! Akita Prefectural University, Japan.
!
! HOWEVER, PROMISE THAT YOU NEVER CHANGE THE
! COPYRIGHT DESCRITIONS, AND IMPORTANT COEFFICIENT
! VALUES IN THE CODES.
! IN ADDITION, PLEASE REFER THIS PDF, IF YOU MAKE
! YOUR CALCULATED RESULT(S) OPEN TO THE PUBLIC.
!
! THE AUTHOR NEVER ACCEPT ANY RESPONSIBILITY TO
! CALCULATED RESULT(S) YOU GOT VIA THESE CODES.
! ------------------------------------------------
program SolPos
  use mGrobal
  implicit none
  type(TIntDate) Date
  type(TTday)    TDay
  type(TJday)    JDay
  integer   YY, SDAY, dT, DeltaTime
  integer*8 JDE12, JD_of
  double precision UT, SD, CD, D, R, ET, JC
!
  YY = 2020

  Date%Year  = YY
  Date%Month = 7
  Date%Day   = 24

  call IntDateToTday( Date, TDay )
  call IntDateToJday( Date, JDay )

  SDAY = TDay%Value
  JDE12 = JD_of( Date )
  JC = JC2000( JDE12 )
  dT = DeltaTime( YY )
  print *, "2020/07/24 =", SDAY
  print *, "J.Day@12TT =", JDE12
  print *, "Jul. Cent. =", JC
  print *, "Time Diff. =", dT * 0.001

  UT = 6d0
  call SolarParams( YY,SDAY,UT, SD,CD,D,R,ET )
  print *, "Declination (deg)", D
  print *, "Eq. of time (s)", ET * 240d0

  stop
end program SolPos
```

D.2 C/C++ Code

Calculation program code is listed below. As you know, the sun position can get by using function `SolPos( ... )`.

```
// ================================================
//  main.cpp (example) メイン関数の例
// ================================================
#include <tchar.h>
#include <stdio.h>
#include <conio.h>
#include <System.Math.hpp>
// For following functions, Round(...),
// DegToRad(...), RadToDeg(...) and so on
// are local for a specified compiler.
// You may find/make equivalent ones.
#pragma hdrstop
#pragma argsused
// Declaration of important functions
#include "SolPos.h"
//
int _tmain( void )
{
  TIntDate  Date;
  double    Dlt, Et;
  double    SinD, CosD, D, R, CosV,
            SinH, CosH, SinA, CosA, h, A;
  //
  // Calculation for Tokyo in 15h JST,
  // July 24, 2020
  //
  int  Year = 2020 Date.year = Year;
  Date.month = 7; Date.day = 24;
  int  Jst = 15;
  double Phi =  35.0 + 41.1 / 60.0;
  double Ell = 139.0 + 45.6 / 60.0;
  //
  // By Rika Nempyo
  // (Chronological Scientific Tables)
  //
  Dlt = DegToRad( 19.7559 );
  SinD  = sin( Dlt ); CosD = cos( Dlt );
  Et = 15.0 / 3600.0 * (-392.4);
  SolarPosition( Phi, Ell, Jst, Et,
    SinD, CosD, SinH, CosH, SinA, CosA );
  h = RadToDeg( asin( SinH ) );
  A = RadToDeg( acos( CosA ) )
    * (SinA >= 0.0 ? 1.0 : -1.0);
  wprintf( L"Rika 6UT Intp.: h = %8.4lf, "
    "A =%8.4lf\n", h , A );
  //
  //  Matsumoto Method (Rev. 2022)
  //
  unsigned Sday =
    unsigned(IntDateToTday( Date ));
  Et = SolarParams( unsigned(Year), Sday,
    6.0, SinD, CosD, D, R );
  wprintf( L"Matsu Rev. 6UT: d = %8.4lf, "
    "Te=%8.1lf\n", D , Et * 240.0 );
  SolarPosition( Phi, Ell, Jst, Et,
    SinD, CosD, SinH, CosH, SinA, CosA );
  h = RadToDeg( asin( SinH ) );
  A = RadToDeg( acos( CosA ) )
    * (SinA >= 0.0 ? 1.0 : -1.0);
  wprintf( L"Rika 6UT Intp.: h = %8.4lf, "
    "A =%8.4lf\n", h , A );
  //
  //  Akasaka Method afer Yamazaki Method
  //
  Et = SolarParamsAk2( unsigned(Year),
    Sday, SinD, CosD, CosV );
  D = RadToDeg( asin( SinD ) );
  wprintf( L"Akasaka Method: d = %8.4lf, "
    "Te=%8.1lf\n", D , Et * 240.0 );
  SolarPosition( Phi, Ell, Jst, Et,
    SinD, CosD, SinH, CosH, SinA, CosA );
  h = RadToDeg( asin( SinH ) );
  A = RadToDeg( acos( CosA ) )
    * (SinA >= 0.0 ? 1.0 : -1.0);
  wprintf( L"Rika 6UT Intp.: h = %8.4lf, "
    "A =%8.4lf\n", h , A );

  printf( "Hit any key..." );
  getch();

  return ( 0 );
}
// ================================================
//  SolPos.h (header file)
// ================================================
#ifndef SolPosH
#define SolPosH
// ------------------------------------------------
// SolPos for the building env. eng'g.
// SolPos.h & SolPos.cpp
// - Functions related on calc. of
//   the solar position
// (c) 2014-2020, Shin-ichi Matsumoto
// ------------------------------------------------
typedef unsigned long ulong;
typedef long TJday; // Julian day
typedef int  TTday; // Serial day no.
extern const ulong JD2000;
```

```
#pragma pack(push,1)
typedef struct {
  int year; int month; int day;
} TIntDate;
#pragma pack(pop)
// ------------------------------------------------
extern void SolarPosition(
  const double Phi,  const double Ell,
  const double Jst,  const double Et,
  const double SinD, const double CosD,
  double& SinH, double& CosH,
  double& SinA, double& CosA );
  /* Solar position calculation func.
      specified for JAPAN
     INPUT
       Phi: Latit.(+deg.),
       Ell: Longit.(+deg.),
       Jst: Hour in JST(0-24),
        Et: Eq. of time (deg.),
       SinD, CosD: for the declination
     OUTPUT
       SinH, CosH: for the solar altit.
       SinA, CosA: for the solar azimuth
   NOTE
     The input parameters of Et, SinD,
     and CosD must be calculated by
     function SolarParams0(...), or
     SolarParams1(...).
  */
// ------------------------------------------------
extern bool Is_Leap( int aYYYY );
  /* Judge whether the given year
     (aYYYY) is leap or not.
    When it is a leap year, then returns
    true.
  */
// ------------------------------------------------
extern bool CorrectDate( int& aYear,
  int& aMonth, int& aDay );
  /* Corrects and Returns data of aYear,
     aMonth, and aDay on a date if it was
     wrong.
  */
// ------------------------------------------------
extern void TdayToIntDate( const TTday aTday,
  const int aYYYY, TIntDate& aIntDate );
  /* Serial day number (aTday) in a
     year (aYYYY) is converted into
     the TIntDate Struct (aIntDate).
  */
// ------------------------------------------------
extern TTday IntDateToTday(
  const TIntDate& aIntD );
  /* A given TIntDate Struct (aIntD)
     is converted into the serial day
     number as a return value.
  */
// ------------------------------------------------
extern TJday IntDateToJday(
  const TIntDate& aIntDate );
  /* Julian day for the given Struct
     TIntDate (aIntDate) is returned.
  */
// ------------------------------------------------
inline double JC2000( const double aJD )
{ /* Returns the Julian century based on
     J2000.0 for a given Julina day aJD.
    NOTE
     aJD is NOT INTEGER!
  */
  return ( (aJD - JD2000) / 36525.0 );
}
// ------------------------------------------------
extern ulong JD_of( const TIntDate& aDate );
  /* Returns a Julian day number after
     calculation based on the given
     Struct TIntDate arg.( aDate )
    NOTE
     Meeus Method was implemented.
     This function does not check the
     validation of the given arg.
     Execution of the correction func.,
     CorrectDate(...) is recommended
     before using this function.
     DON'T GIVE a date before Jan. 1,
     4713B.C.
     The day separation of the Julian
     day system is at noon, thus
     returned ulong value means
     the Julian day at noon.
     Ref:  W.H.Press et al.,
       Numerical Recipes in C,
       Cambridge Univ. Press, pp.28-29,
       1988.
  */
// ------------------------------------------------
extern long DeltaTime( const int aYear );
  /* Returns a time difference between TT
     and UT for OUT on July 1 in a given
     year.
     Matsumoto Method (Rev. 2022) after
     Espenak and Meeus (NASA TP, 2006)
    NOTE
     Return value is in MILI SECOND.
  */
// ------------------------------------------------
extern long DeltaTimeEx(
  const TIntDate& aUTDate );
  /* Returns a time difference between TT
     and UT1 for OUT on a given date.
     Matsumoto Method (Rev. 2022) after
     Espenak and Meeus (NASA TP, 2006)
    NOTE
     Return value is in MILI SECOND.
  */
// ------------------------------------------------
extern double Obliquity( const double aJC,
  const bool IsPrecise = true );
  /* (黄道傾角)
     Returns the obliquity of the
     ecliptic for a given Julian century
     based on the J2000.0 point (aJC).
    NOTE
     On consideration of the nutation,
     IsPrecise is switch.
     = true (default): with consideration
     = false: w/o consideration
     Return value is in DEG.
     Hwreafter, abbr. JC2000 for
     a Julian day based on the J2000.0.
  */
// ------------------------------------------------
extern double CLongit( const double aJC,
  const bool IsApparent = true );
  /* (視黄経または平均黄経)
     Returns the solar celestial
     longitude for a given JC2000.
    NOTE
     If IsApparent = false, then the
     mean value is retuerned.
     Return value is in DEG.
  */
// ------------------------------------------------
extern double MAscens( const double aUTJC );
  /* (平均太陽の赤経)
     Returns the mean solar right
     ascension for a given JC2000 (aUTJC).
    NOTE
     aUTJC should be calculated
     with UTC.
     Return value is in DEG.
  */
// ------------------------------------------------
extern double Eq( const double aJC );
  /* (分点差)
     Returns the equation of equinoxes
     for a given JC2000 (aJC).
    NOTE
     Return value is in HOUR.
     Based on the equation:
       dL * cos(eps) / 15.
     Where dL is the nutation of the
     celestial longitude[deg.] and
     eps is he apparent obliquity of
     the ecliptic.
  */
// ------------------------------------------------
extern double SolarDistance( const double aJC );
```

```
  /* (地心距離)
     Returns the geocentric distance to
     the sun for a given JC2000 (aJC).
    NOTE
     Return value is in AU.
  */
// ------------------------------------------------
// For calc. of the solar declination
// and the equation of time (solar
// parameters) (視赤緯と均時差)
//
// INPUT args.
//    aYear: a given year,
//  aSerDay: a given serial day of aYear,
//     aUTC: a given JC2000 value at
//           designated UTC.
// OUTPUT args.
//  D: the solar declination in DEG.
//     [SinD, CosD: sin(D), cos(D)],
//  R: the solar distance in AU.
// NOTE
//  Following all functions return
//  the equation of time in DEG.
//
// ------------------------------------------------
extern double SolarParams(
  const unsigned aYear,
  const unsigned aSerDay,
  const double aUTC,
  double& SinD, double& CosD,
  double&D, double& R );
  /* Matsumoto Method (Rev. 2020)
     (松本の方法 2020 年改)
  */
// ------------------------------------------------
extern double SolarParamsAk2(
  const unsigned aYear,
  double& SinD, double& CosD,
  double& CosV );
  /* Akasaka(2022) Method (赤坂 (2022) の方法)
    NOTE
     For output arg. CosV, see
     the following reference.
    H. Akasaka: Simplified Calculation Method of
      the Sun Position with Secular Changes,
      General Technical Report on the EA Weather
      Data. MetDS HP. 2022.8
     赤坂　裕：年差を考慮した太陽位置の簡易計算，
       EA 気象データ技術解説一般，MetDS HP，2022.8.
  */
// ------------------------------------------------
extern void Yamazaki(
  const SDate* aDate,
  const STime* aUT,
  double& Decl, double& Et );
  /* Yamazaki Method (山崎の方法)
    Ref. H. Yamazaki: Fundamental Formulae for
      Daylighting   IV (Precise Program to
      Calculate Solar Declination   and Equation of
      Time) (in Japanese), Proc. of AIJ Annual
      Meeting 1980, pp.407--408, 1980.9.
      山崎　均：日照環境の基礎計算式 IV (対象地域
      の太陽視赤緯及び均時差を正確に計算するプロ
      グラム)，日本建築学会大会学術講演梗概集，
      計画系，pp.407--408，1980.9.
  */
// ------------------------------------------------
#endif // End flag for "SolPos.h"
//
// ================================================
//  SolPos.cpp (source code file)
// ================================================
#include <math.h>
#include <System.Math.hpp>  // Local!
#pragma hdrstop
#include "SolPos.h"
#pragma package(smart_init) // Local!
// ------------------------------------------------
// SolPos for the building env. eng'g.
// SolPos.h & SolPos.cpp
// - Functions related on calc. of
//   the solar position
// (c) 2014-2022, Shin-ichi Matsumoto
// ------------------------------------------------
#define YEAR_GREG (1582) // Adoped in 1582
const ulong JD2000 = 2451545UL;
//          Julian day of Jan. 1 0UT, 2000
//
const
  int DAYS_OF_MONTH[12] = { 31, 28, 31,
  30, 31, 30, 31, 31, 30, 31, 30, 31, };
int Days_of_Month[12]; // for temp.
#pragma pack(push,2)
typedef struct {
  bool T; double A; double B; double C;
} TAstroChart;
#pragma pack(pop)
// ------------------------------------------------
// Calendar functions
// ------------------------------------------------
bool Is_Leap( int aYYYY )
{
  if ( aYYYY  <= YEAR_GREG )
    return ( false );
  if ( 0 == (aYYYY % 400) )
    return ( true  );
  if ( 0 == (aYYYY % 100) )
    return ( false );
  if ( 0 == (aYYYY % 4) )
    return ( true  );
  else
    return ( false );
}
// ------------------------------------------------
bool CorrectDate(
  int& aYear, int& aMonth, int& aDay )
{
  bool check = true;
  if ( 0 == aYear ) {
    aYear = 1; check = false; }
  if (  1 > aMonth ) {
    aMonth =  1; check = false; }
  if ( 12 < aMonth ) {
    aMonth = 12; check = false; }
  if (  1 > aDay ) {
    aDay = 1; check = false; }
  int leap = Is_Leap( aYear );
  for ( int m = 0; m < 12; m++ )
    Days_of_Month[m] = DAYS_OF_MONTH[m];
  if ( leap )
    Days_of_Month[1]++;
  if ( aDay > Days_of_Month[aMonth - 1]) {
    aDay = Days_of_Month[aMonth - 1 ];
    check = false;
  }
  return ( check );
}
// ------------------------------------------------
void TdayToIntDate( const TTday aTday,
  const int aYYYY, TIntDate& aIntDate )
{
  int m, day;
  int days[12];
  int leap = Is_Leap( aYYYY );
  for ( m = 0, day = 0; m < 12; m++ ) {
    day += DAYS_OF_MONTH[m];
    if ( leap && (1 == m) ) day++;
    days[m] = day;
  }
  TTday tday = aTday;
  if (   1 > aTday ) tday = 1;
  if ( 365 < aTday )
    tday = leap ? 366 : 365;
  aIntDate.year = aYYYY;
  for ( m = 1; m <= 12; m++ ) {
    if ( tday <= days[m - 1] ) {
      aIntDate.month = m; break; }
  }
  if ( 1 == aIntDate.month )
    aIntDate.day = tday;
  else
    aIntDate.day = tday
      - days[aIntDate.month - 2];
```

```
}
// ------------------------------------------------
TTday IntDateToTday(
  const TIntDate& aIntD )
{
  int m;
  if ( 1 == aIntD.month )
    return( TTday(aIntD.day) );
  bool leap = Is_Leap( aIntD.year );
  for ( m = 0; m < 12; m++ )
    Days_of_Month[m] = DAYS_OF_MONTH[m];
  if ( leap ) Days_of_Month[1]++;
  TTday days = (TTday)0;
  for ( m = 0; m < aIntD.month - 1; m++ )
    days += TTday(Days_of_Month[m]);
  return ( days + (TTday)aIntD.day );
}
// ------------------------------------------------
#define LGREG(Y,M,D) ((D)+31L*((M)+12L*(Y)))
#define GREG0 LGREG(1582,10,15)
// The Greg. calendar was adopted in...
TJday IntDateToJday(
  const TIntDate& aIntDate )
{ // General ver. of JD_of(...)
  TJday jul;
  int ja, jm, jy = aIntDate.year;
  if ( jy == 0 ) return( TJday(-1) );
  if ( jy < 0 )  jy++;
  if ( aIntDate.month > 2 )
    jm = aIntDate.month + 1;
  else {
    jy--; jm = aIntDate.month + 13;
  }
  jul  = TJday(floor( 365.25 * jy ));
  jul += TJday(floor( 30.6001 * jm ));
  jul += TJday(aIntDate.day)
    + TJday(1720995L);
  if ( GREG0 <= LGREG(aIntDate.year,
    long(aIntDate.month),
    long(aIntDate.day)) ) {
    // Use Gregorian Calendar
    ja   = int(0.01 * aIntDate.year);
    jul += TJday(2 - ja + int(0.25*ja));
  }
  return ( jul );
}
// ------------------------------------------------
ulong JD_of( const TIntDate& aDate )
{
  int ja, jm, jy = aDate.year;
  long jul;
  if ( 0 > jy ) ++jy;
  if ( 2 < aDate.month )
    jm = aDate.month + 1;
  else {
    --jy; jm = aDate.month + 13;
  }
  jul = long(floor( 365.25 * jy )
    + floor( 30.6001 * jm ) + aDate.day)
    + 1720995L;
  if ( GREG0 <= LGREG(aDate.year,
    long(aDate.month), long(aDate.day))) {
    ja   = int( 0.01 * jy );
    jul += 2 - ja +int( 0.25 * ja );
  }
  return ( (ulong)jul );
}
#undef GREG0
#undef LGREG(Y,M,D)
//
// vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
//  MAIN REVISION 2022 !
// ------------------------------------------------
extern long DeltaTime( const int aYear )
{
  TIntDate UTDate;
  UTDate.year  = aYear;
  UTDate.month = 7;
  UTDate.day   = 1;

  return ( DeltaTimeEx( UTDate ) );
}
// ------------------------------------------------
extern long DeltaTimeEx(
  const TIntDate& aUTDate )
{
  double y = aUTDate.year
    + (aUTDate.month - 0.5) / 12.0;
  if ( 7 == aUTDate.month &&
    1 == aUTDate.day )
    y = aUTDate.year + 0.5;
  if ( 1 == aUTDate.month &&
    1 == aUTDate.day ) y = aUTDate.year;

  double  t, S;
  long    R;

  if ( 1600 > aUTDate.year )  return (0L);
  else if ( 1701 > aUTDate.year ) {
    t = y - 1600.0;
    S =  ((1.0 / 7129.0 * t  - 0.01532)
      * t - 0.9808) * t + 120.0;
  }
  else if ( 1801 > aUTDate.year ) {
    t = y - 1700.0;
    S = (((-1.0 / 1174000.0 * t +
       0.00013336) * t - 0.0059285)
      * t  + 0.1603) * t + 8.83;
  }
  else if ( 1861 > aUTDate.year ) {
    t = y - 1800.0;
    S = ((((((0.000000000875 * t
      - 0.0000001699) * t + 0.0000121272)
      * t - 0.00037436) * t + 0.0041116)
      * t + 0.0068612) * t - 0.332447)
      * t + 13.72;
  }
  else if ( 1901 > aUTDate.year ) {
    t = y - 1860.0;
    S =  ((((1.0 / 233174.0 * t
      - 0.0004473624) * t + 0.01680668)
      * t - 0.251754) * t + 0.5737)
      * t + 7.62;
  }
  else if ( 1921 > aUTDate.year ) {
    t = y - 1900.0;
    S = (((-0.000197 * t + 0.0061966)
      * t - 0.0598939) * t + 1.494119)
      * t - 2.79;
  }
  else if ( 1941 > aUTDate.year ) {
    t = y - 1920.0;
    S = ((0.0020936 * t - 0.076100)
      * t + 0.84493) * t + 21.20;
  }
  else if ( 1961 > aUTDate.year ) {
    t = y - 1950.0;
    S = ((1.0 / 2547.0 * t  - 1.0 / 233.0)
      * t + 0.407) * t + 29.07;
  }
  else if ( 1986 > aUTDate.year ) {
    t = y - 1975.0;
    S = ((-1.0/718.0 * t  - 1.0 / 260.0)
      * t  + 1.067) * t + 45.45;
  }
  else if ( 2006 > aUTDate.year ) {
    t = y - 2000.0;
    S = ((((0.00002373599 * t
      + 0.000651814) * t + 0.0017275)
      * t - 0.060374)
      * t + 0.3345) * t + 63.86;
  }
  else if ( 2051 > aUTDate.year ) {
    t = y - 2000.0;
    S = (0.005589 * t + 0.32217)
    * t + 62.92;
  }
  else if ( 2151 > aUTDate.year ) {
    t = y - 1820.0;
    S = (0.0032 * t + 0.5628)
      * t - 205.724;
  }
  else S = 328.48;

  R = long( RoundTo( S, -3 ) * 1000.0 );
```

```
  return ( R );
}
// ------------------------------------------------
// ------------------------------------------------
double Obliquity( const double aJC,
  const bool IsPrecise )
{
  double epsilon = ((-5.036111e-7 * aJC
    + 1.638889e-7) * aJC + 1.300417e-2)
    * aJC - 2.343929e+1;
  if ( IsPrecise ) {
    double T1 = DegToRad(fmod(
       1934.0 * aJC + 235.0, 360.0 ));
    double T2 = DegToRad(fmod(
      72002.0 * aJC + 201.0, 360.0 ));
    epsilon  -= 0.00256 * cos( T1 )
      + 0.00015 * cos( T2 );
  }
  return ( epsilon );
}
// ------------------------------------------------
#define _T (true)
#define _F (false)
double CLongit( const double aJC,
  const bool IsApparent )
{
  static const int C_MAX = 18;
  static const TAstroChart Coef[C_MAX] = {
  {_T, 36000.7695,0.0,0.0},
  {_F, 280.465900,0.0,0.0},
  {_F, 1.91470000,35999.050,267.52000},
  {_F, 0.02000000,71998.100,265.10000},
  {_T,-0.00480000,35999.000,268.00000},
  {_F, 0.00200000,32964.000,158.00000},
  {_F, 0.00180000,19.000000,159.00000},
  {_F, 0.00180000,445267.00,208.00000},
  {_F, 0.00150000,45038.000,254.00000},
  {_F, 0.00130000,22519.000,352.00000},
  {_F, 0.00070000,65929.000,45.000000},
  {_F, 0.00070000,3035.0000,110.00000},
  {_F, 0.00070000,9038.0000,64.000000},
  {_F, 0.00060000,33718.000,316.00000},
  {_F, 0.00050000,155.00000,118.00000},
  {_F, 0.00050000,2281.0000,221.00000},
  {_F, 0.00040000,29930.000,48.000000},
  {_F, 0.00040000,31557.000,161.00000},
  };
  double cl = 0.0, A, BT_C;
  for ( int i = C_MAX - 1; i >= 2; i-- ) {
    // DON'T CHANGE the loop INDEX ORDER
    // to avoid information loss!
    A    = (Coef[i].T ?
      Coef[i].A * aJC : Coef[i].A);
    BT_C = Coef[i].B * aJC + Coef[i].C;
    BT_C = DegToRad( fmod(BT_C, 360.0) );
    cl += A * cos( BT_C );
  }
  cl += Coef[1].A + Coef[0].A * aJC;
  if ( IsApparent ) {
    A    = 0.0048;
    BT_C = 1934.0 * aJC + 145.0;
    BT_C = DegToRad( fmod(BT_C, 360.0) );
    cl += A * cos( BT_C );
    A    = -0.0004;
    BT_C = 72002.0 * aJC + 111.0;
    BT_C = DegToRad( fmod(BT_C, 360.0) );
    cl += A * cos( BT_C ) - 0.0057;
  }
  cl = fmod(cl, 360.0);
  return ( cl >= 0.0 ? cl : cl + 360.0 );
}
// ------------------------------------------------
double MAscens( const double aUTJC )
{
  static const double SecConst
    = 67310.54841;
  double Sec = -0.0000062
    * aUTJC * aUTJC * aUTJC;
  Sec += 0.093104 * aUTJC * aUTJC;
  Sec += 8640184.812866 * aUTJC;
  Sec += SecConst;
  double Hour = Sec / 3600.0;
  Hour = fmod(Hour, 24.0);
  return ( Hour < 0 ? Hour+24.0 : Hour );
}
// ------------------------------------------------
double Eq( const double aJC )
{
  double eq = 0.0048 * sin(
    DegToRad(fmod(1934.0 * aJC + 235.0,
    360.0)) ) - 0.0004 * sin( DegToRad(
    fmod(72002.0 * aJC + 201.0, 360.0)) );
  eq *= cos( DegToRad(
    Obliquity( aJC, true )) );
  return ( eq / 15.0 );
}
// ------------------------------------------------
double SolarDistance( const double aJC )
{
  static const int C_MAX = 9;
  static const TAstroChart Coef[C_MAX] = {
  {_F, 1.00014000,0.00000000,0.00000000},
  {_F, 0.01670600,35999.0500,177.530000},
  {_F, 0.00013900,71998.0000,175.000000},
  {_T,-0.00004200,35999.0000,178.000000},
  {_F, 0.00003100,445267.000,298.000000},
  {_F, 0.00001600,32964.0000,68.0000000},
  {_F, 0.00001600,45038.0000,164.000000},
  {_F, 0.00000500,22519.0000,233.000000},
  {_F, 0.00000500,33718.0000,226.000000},
  };
  double sd = 0.0, A, BT_C;
  for ( int i = TAC_MAX-1; i >= 1; i-- ) {
    // DON'T CHANGE the loop INDEX ORDER
    // to avoid information loss!
    A    = (Coef[i].T ?
      Coef[i].A * aJC : Coef[i].A);
    BT_C = Coef[i].B * aJC + Coef[i].C;
    BT_C = DegToRad(fmod(BT_C, 360.0));
    sd  += A * cos( BT_C );
  }
  sd += Coef[0].A;
  return ( sd );
}
#undef _T
#undef _F
// ------------------------------------------------
double SolarParamsAk2(
  const unsigned aYear, const unsigned aSerDay,
  const unsigned aDay,  const unsigned aHour,
  const unsigned aMin,  const unsigned aSec,
  const int aLongit0,
  double& SinD, double& CosD, double& CosV )
{
  //
  // Akasaka(2022) Method
  //
  int     n = int(aYear - 1968u);
  double d0 =-(n+3) /4; d0 += 3.71 + 0.2596 * n;
  double Delta0 = double(DegToRad(-23.4393 +
                  0.00013 * (aYear - 2000)));
  double  M = 0.9856 * (double(aSerDay) - d0);
  double MRad = DegToRad(M);
  double Eps  = 12.39 + 0.0172 *
                (double(n) + M / 360.0);
  double V    = M + 1.914 * sin(MRad)
              + 0.02 * sin(2.0 * MRad);
  double VEps = DegToRad(V + Eps);
  double VE2  = 2.0 * VEps;
  double Ang  = atan( (0.043 * sin( VE2 ))
                / (1.0 - 0.043 * cos( VE2 )) );
  double Et   = (M - V) - RadToDeg(Ang);
  SinD = cos( VEps ) * sin( Delta0 );
  CosD = sqrt( fabs( 1.0 - SinD*SinD ) );
  CosV = cos( DegToRad(V) );

  return ( Et ); // in DEG.! NOT in MIN.
}
// ------------------------------------------------
void Yamazaki(
  const SDate* aDate,
  const STime* aUT,
  double& Decl, double& Et )
{
  //
```

```
  // Yamazaki Method
  //
  double Hour = anUT->HH + (anUT->MM * 60.0
              + anUT->SS) / 3600.0;
  Hour = anUT->PM == sgnPlus ? Hour : -Hour;
  SDate Date = *aDate;
  int yn = Date.YY - 1900;
  int d2 = static_cast<int>((yn - 1) / 4.0);
  // temporary.....................
  double d = SerialDay_of( Date )
           + (yn - 30) * 1.1574e-5;
  //..............................
  // double d = SerialDay_of( Date );
  if ( yn >= 85 ) d += (yn - 30) * 1.1574e-5;
  d += (Hour - 12.0) / 24.0;
  double t1 = (365.0 * yn + d2 + d) / 36525.0;
  double t2 = t1 * t1;
  double t3 = t2 * t1;
//  a little bit bonehead but as it is original
  double delta0 = -9.44e-5 + 1.30125e-2 * t1
                + 1.64e-6 * t2 + 5.0e-7 * t3;
  delta0 -= 23.4522;
  delta0 = DegToRad( delta0 );
  double e = 1.04e-6 - 4.18e-5 * t1
           - 1.26e-7 * t2 + 0.01675;
  double eps = 0.719175 * t1 + 0.000453 * t2;
  eps += 0.220833 + t1 + 11.0;
  eps = DegToRad( eps );
  double m = 6.00267e-4 * (d2 + d)
           - 9.02579e-4 * yn
           - 0.00015 * t2 - 1.667e-4;
  m += -1.524 - 0.255 * yn + 0.985 * d2;
  m += 0.985 * d;
  m = DegToRad( m );

  double SinM   = sin( m );
  double Sin2M  = sin( 2.0 * m );
  double Sin3M  = sin( 3.0 * m );

  double dm = 9.93502e-5 * (1.0 - e * (cos( m )
            - 2.0 * e * SinM * SinM));
  m -= dm;

  double v = (2.0 - 0.25 * e * e) * e * SinM;
  v += 1.25 * e * e * Sin2M;
  v += 13.0 / 12.0 * e * e * e * Sin3M;
  v += m;

  double sd = cos( eps + v ) * sin( delta0 );
  double cd = sqrt( 1.0 - sd * sd );
  double v_eps2 = 2.0 * (eps + v);
  double a = (1.0 - cos( delta0 ))
           / (1.0 + cos( delta0 ));
  // Solar Declination in DEG. 視赤緯
  Decl = RadToDeg( atan( sd / cd ) )
  // Equation of time in min.  均時差
  double et1 = RadToDeg( m - v );
  double et2 = -atan( (a * sin( v_eps2 ))
             / (1.0 - a * cos( v_eps2 )) )
  et2 = RadToDeg( et2 );
  Et = (et1 + et2) * 4.0;
}
// ------------------------------------------------
double SolarParams(
    const unsigned aYear,
    const unsigned aSerDay,
    const double aUTC,
    double& SinD, double& CosD,
    double&D, double& R )
{
  //
  // Matsumoto Method (Rev. 2022)
  // with DeltaTime and DeltaTimeEx
  //
  static double   dHr(0.0);
  static unsigned pYear(0);
  TIntDate        theDate;
  TdayToIntDate(aSerDay, aYear, theDate);
  double Hr = aUTC; // UT
  if ( pYear != aYear ) {
    dHr = DeltaTime( aYear )
      * 0.001 / 3600.0;  // in Hour;
    // LOL!
    // This is only difference with
    // function SolarParams0(...)
    pYear = aYear;
  }
  double JC = JC2000( JD_of( theDate ) );
  double T  = JC
    + (Hr + dHr - 12.0) / (24.0*36525.0);
  double Tu = JC
    + (Hr - 12.0) / (24.0 * 36525.0);
  double Del =
    DegToRad(Obliquity( T, true ));
  double L   =
    DegToRad(CLongit( T, true ));
  double Sd  = -sin( L ) * sin( Del );
  double Cd  = sqrt( 1.0 - Sd * Sd );
  double Dec = atan( Sd / Cd ); // in RAD.
  double Ta  = tan( L ) * cos( Del );
  double Tam =
    tan( DegToRad( 15.0*MAscens( Tu )) );
  double et  = RadToDeg(
    atan( (Tam - Ta) / (1.0 + Tam*Ta) ));
    // in DEG.
  double Et  = et + Eq( T ) * 15.0;
  SinD = sin( Dec );
  CosD = cos( Dec );
  D = RadToDeg( Dec );
  R = SolarDistance( T );
  return ( Et ); // in DEG.! NOT in MIN.
}
// ------------------------------------------------
void SolarPosition(
    const double Phi,  // Latit. in deg.
    const double Ell,  // Longit. in deg.
    const double Jst,  // Hour in JST
    const double Et,   // Eqt. in deg.
                       // D: Declination
    const double SinD, // H: Solar altit.
    const double CosD, // A: Solar azim.
    double& SinH, double& CosH, //
    double& SinA, double& CosA )
{
  // This function is just for JAPAN.
  // You can change 135.0 in the following
  // code to be a lontitude value for
  // reference one in your time zone.
  double T = DegToRad(15.0 * (Jst - 12.0)
    + (Ell - 135.0) + Et);
  double P = DegToRad(Phi);
  SinH = sin( P ) * SinD
    + cos( P ) * CosD * cos( T );
  CosH = sqrt( fabs( 1.0 - SinH*SinH ) );
  SinA = CosD * sin( T ) / CosH;
  CosA = (SinH * sin( P ) - SinD)
    / (CosH * cos( P ));
}
// ------------------------------------------------
// THE CODES DESCRIBED IN THE ABOVE CAN BE USE,
// COPIED, AND MODIFIED WITHOUT ANY ACCEPTANCE OF
// THE AUTHOR: Dr. Shin-ichi Matsumoto, Prof.,
// Akita Prefectural University, Japan.
//
// HOWEVER, PROMISE THAT YOU NEVER CHANGE THE
// COPYRIGHT DESCRITIONS, AND IMPORTANT COEFFICIENT
// VALUES IN THE CODES.
// IN ADDITION, PLEASE REFER THIS PDF, IF YOU MAKE
// YOUR CALCULATED RESULT(S) OPEN TO THE PUBLIC.
//
// THE AUTHOR NEVER ACCEPT ANY RESPONSIBILITY TO
// CALCULATED RESULT(S) YOU GOT VIA THESE CODES.
// ------------------------------------------------
```

D.3 Calculation Example by C/C++ Code

Julian Ephemeris Day JDE(JDE@0TT), the solar declination δ (Decl.) [°], the equation of time T_e (Te) [s], and the geocentric distance r (r) [–, AU] were calculated by C/C++ code mentined above in every five days for full course of the year of 2051 from January 0 at 0^{h}TT, which means December 31 at 0^{h}TT, 2050.

The results are described in Table 10. You may know that the results by FORTRAN code are a litte bit different from these data due to no careful treatment to guarantee precisions of floating point real constants, built-in functions return values, and so on in our codes. However, the difference is not significant for practical engineering purpose[*12].

Table 10 Calculated Parameters for Sun Position in 2051 by C/C++ Program Code

Year: 2051, Delta T: 96.059sec. by Matsumoto Method (Rev. 2022)

MM/DD	JDE@0UT	Decl.	Te. (sec.)	r [AU]	MM/DD	JDE@0UT	Decl.	Te. (sec.)	r [AU]
01/00*	2470171.5	-23.09193	-165.31929	0.98334	07/04	2470356.5	22.88638	-264.64520	1.01670
01/05	2470176.5	-22.63395	-304.33107	0.98332	07/09	2470361.5	22.37526	-314.77421	1.01670
01/10	2470181.5	-21.98920	-433.22671	0.98343	07/14	2470366.5	21.70330	-354.61873	1.01655
01/15	2470186.5	-21.16544	-548.48732	0.98364	07/19	2470371.5	20.87733	-381.85381	1.01627
01/20	2470191.5	-20.17269	-646.98543	0.98394	07/24	2470376.5	19.90510	-395.06927	1.01589
01/25	2470196.5	-19.02258	-726.55231	0.98437	07/29	2470381.5	18.79491	-393.72087	1.01542
01/30	2470201.5	-17.72755	-786.20481	0.98495	08/03	2470386.5	17.55570	-377.67640	1.01486
02/04	2470206.5	-16.30053	-825.83925	0.98567	08/08	2470391.5	16.19736	-346.84386	1.01417
02/09	2470211.5	-14.75525	-845.71362	0.98649	08/13	2470396.5	14.73074	-301.24806	1.01335
02/14	2470216.5	-13.10632	-846.21164	0.98738	08/18	2470401.5	13.16702	-241.43677	1.01242
02/19	2470221.5	-11.36882	-828.05557	0.98835	08/23	2470406.5	11.51701	-168.75724	1.01142
02/24	2470226.5	-9.55748	-792.66398	0.98941	08/28	2470411.5	9.79110	-85.18456	1.01036
03/01	2470231.5	-7.68611	-742.21138	0.99057	09/02	2470416.5	7.99984	7.15827	1.00924
03/06	2470236.5	-5.76796	-679.26110	0.99183	09/07	2470421.5	6.15451	106.34449	1.00803
03/11	2470241.5	-3.81639	-606.27623	0.99315	09/12	2470426.5	4.26704	210.55636	1.00673
03/16	2470246.5	-1.84520	-525.43320	0.99448	09/17	2470431.5	2.34929	317.70756	1.00537
03/21	2470251.5	0.13186	-438.85737	0.99583	09/22	2470436.5	0.41250	425.20099	1.00399
03/26	2470256.5	2.10184	-348.98347	0.99722	09/27	2470441.5	-1.53242	530.10232	1.00261
03/31	2470261.5	4.05280	-258.62747	0.99865	10/02	2470446.5	-3.47408	629.58878	1.00122
04/05	2470266.5	5.97320	-170.65203	1.00012	10/07	2470451.5	-5.40006	721.25778	0.99979
04/10	2470271.5	7.85108	-87.51879	1.00157	10/12	2470456.5	-7.29715	803.02375	0.99833
04/15	2470276.5	9.67398	-11.13769	1.00298	10/17	2470461.5	-9.15216	872.74892	0.99687
04/20	2470281.5	11.42946	56.87268	1.00435	10/22	2470466.5	-10.95226	928.02983	0.99547
04/25	2470286.5	13.10585	114.81770	1.00569	10/27	2470471.5	-12.68466	966.41643	0.99412
04/30	2470291.5	14.69226	160.87469	1.00702	11/01	2470476.5	-14.33590	985.91275	0.99282
05/05	2470296.5	16.17804	193.46856	1.00832	11/06	2470481.5	-15.89164	985.32270	0.99154
05/10	2470301.5	17.55225	211.74972	1.00954	11/11	2470486.5	-17.33722	964.15641	0.99030
05/15	2470306.5	18.80392	215.72964	1.01066	11/16	2470491.5	-18.65853	922.26004	0.98913
05/20	2470311.5	19.92272	205.96466	1.01168	11/21	2470496.5	-19.84235	859.62444	0.98807
05/25	2470316.5	20.89953	183.13841	1.01263	11/26	2470501.5	-20.87619	776.65109	0.98712
05/30	2470321.5	21.72643	147.99991	1.01352	12/01	2470506.5	-21.74811	674.66748	0.98627
06/04	2470326.5	22.39640	101.73850	1.01433	12/06	2470511.5	-22.44716	556.18637	0.98550
06/09	2470331.5	22.90333	46.39453	1.01503	12/11	2470516.5	-22.96410	424.61963	0.98481
06/14	2470336.5	23.24239	-15.17115	1.01558	12/16	2470521.5	-23.29203	283.68523	0.98424
06/19	2470341.5	23.41058	-79.79708	1.01600	12/21	2470526.5	-23.42651	137.03591	0.98381
06/24	2470346.5	23.40676	-144.58355	1.01633	12/26	2470531.5	-23.36537	-11.62754	0.98354
06/29	2470351.5	23.23139	-207.01679	1.01657	12/31	2470536.5	-23.10885	-158.24134	0.98339

[*12] In Table 10, Te was calculated as:

$$T_e = E_t * 4\text{min.} * 60\text{sec./min.}$$

So you may find that the difference between C/C++ results and FORTRAN results are relatively big. However, it is just mili-second order difference in the equation of time, that is not so important.

References

[1] Shin-ichi Matsumoto: A Calculation Method of the Solar Declination and Equation of Time – Revision of Matsumoto's Method (Rev. 2022) (in *Japanese*), Proc. of AIJ Tohoku Chapter Architectural Resaerch Meeting, Vol.85, pp.27–34, Architectural Institute of Japan, 2022.6.

[2] Shin-ichi Matsumoto: Critical Review of Four Calculation Methods of the Solar Declination and the Equation of Time and Comparison of Their Accuracy (in *Japanese*), Papers of SHASEJ Annual Meeting 2022 (R4), Vol.5, pp.000-999, The Society of Heating, Air-Conditioning and Sanitary Engineers of Japan, 2022.9 (TBA).

[3] Shin-ichi Matsumoto: Calculation of the Solar Declination and the Equation of Time by Modified Method after Hydrographic and Oceanographic Department, Japan Coast Guard (in *Japanese*), Papers of SHASEJ Annual Meeting 2019 (R1), Vol.5, pp.97-100, The Society of Heating, Air-Conditioning and Sanitary Engineers of Japan, 2019.9.

[4] Shin-ichi Matsumoto: Validation of Matsumoto's Method to Calculate the Solar Declination and the Equation of Time (in *Japanese*), Proc. of AIJ Tohoku Chapter Architectural Resaerch Meeting, Vol.82, pp.7–10, Architectural Institute of Japan, 2019.6.

[5] Shin-ichi Matsumoto: Validation of Matsumoto's Calculation Method of the Solar Declination and the Equation of Time (in *Japanese*), Proc. of AIJ Annual Meeting 2014, Vol.D–2, pp.25–26, 2014.9.

[6] Shin-ichi Matsumoto: Supplementary Notes on Matsumoto's Calculation Method of the Solar Declination and the Equation of Time (in *Japanese*), Proc. of AIJ Tohoku Chapter Architectural Resaerch Meeting, Vol.77, pp.49–56, Architectural Institute of Japan, 2014.6.

[7] Shin-ichi Matsumoto: Precision Evaluation of Several Calulation Methods for Solar Declination and Equation of Time (in *Japanese*), Proc. of AIJ Annual Meeting 2006, Vol.D–2, pp.7–8, 2006.8.

[8] Shin-ichi Matsumoto: Notes on Calculation Methods of the Solar Declination and Equation of Time (in *Japanese*), Proc. of AIJ Tohoku Chapter Architectural Resaerch Meeting, Vol.68, pp.89–96, Architectural Institute of Japan, 2005.6.

[9] Kou Nagasawa: Pisition Calculation of Celestial Bodies (Revised Ed.) (in *Japanese*), Chijin Syokan, Tokyo, 1985.

[10] Simon Newcomb: A Compendium of Spherical Astronomy with its Applications to the Determination and Reduction of Positions of the Fixed Stars, Macmillan & Co., New York, 1906.

[11] For instance, Shunroku Tanaka, and Hitoshi Takeda et al.: Daylight and Solar Energy, Contemporary Architectural Environmental Engineering (Ed.: S. Tanaka), Chapter 3, (in *Japanese*), pp.80-86, Inoue Shoin, Tokyo, 2014.

[12] For instance, Toshiyuki Watanabe, and Tetsuo Hayashi et al.: Daylight and Solar Energy, Architectural Environmental Engineering (Ed.: Y. Urano and H. Nakamura), Chapter 4, (in *Japanese*), pp.134-139, Morikita Shuppan, Tokyo, 1996.

[13] Hitoshi Yamazaki: Fundamental Formulae for Daylighting IV (Precise Program to Calculate the Solar Declination and the Equation of Time), (in *Japanese*), Proc. of AIJ Annual Meeting 1980, pp.407–408, 1980.9.

[14] Hiroshi Akasaka et al.: Expanded AMeDAS Weather Data, (in *Japanese*), Architectural Institute of Japan (Maruzen, Tokyo), 2000.

[15] Hiroshi Akasaka et al.: Expanded AMeDAS Weather Data, Architectural Institute of Japan

(Maruzen, Tokyo), 2003.

[16] Hiroshi Akasaka et al.: Expanded AMeDAS Weather Data 1981–2000, (in *Japanese*), Architectural Institute of Japan (Kagoshima TLO, Kagoshima), 2005.

[17] Hiroshi Akasaka: Simplified Calculation Method of the Sun Position with Secular Changes, ("TE_Simplified_SP220804.pdf"in *Japanese*), General Technical Reports PDF on the Expanded AMeDAS Weather Data, MetDS Homepage, https://www.metds.co.jp/, 2022.8.

[18] National Astronomical Observatory, Japan (Ed.): Chronological Scientific Tables Premium (eBook, in *Japanese*), https://www.rikanenpyo.jp/, (recent access: May 31, 2022).

[19] Almanac Calculation Research Group, Japan (Ed.): New Guidebook for Calendar, (in *Japanese*), Koseisha–Koseikaku, Tokyo, 1991.

[20] Pierre Bretagnon and George Francou: Planetary theories in rectangular and spherical variables. VSOP87 solutions, Astronomy and Astrophysics, 202, 1988, pp.309–315.

[21] Ibrahim Reda and Afsin Andreas: Solar Position Algorithm for Solar Radiation Applications, NREL Report (No. TP–560–34302), NREL, Golden, 2003 (Rev.2008.1).

[22] Jean Meeus: Astronomical Algorithms (2nd Ed.), Willmann-Bells, Inc, Virginia, 1999.

[23] Hydrographic and Oceanographic Department, Japan Coast Guard (Homepage): https://www1.kaiho.mlit.go.jp/KOHO/index.html (recent access: May 31, 2022).

[24] Fred Espenak and Jean Meeus: Five Millennium Canon of Solar Eclipses: −1999 to +3000, The NASA Technical Publication (NASA/TP–2006–214141), NASA, Greenbelt, 2006.10.

[25] Nautical Almanac Offices of U.S. Naval Observatory and Her Majesty's Hydrographic Office U.K.: The Astronomical Almanac for the year 2020, U.S. Government Publishing Office, Washington D.C., pp. K8–K9, etc., 2019.