JCAAC (2025), **3**, 65–78

JCAAC

SECTION: ASTRONOMICAL COMPUTING

Cálculo de efemérides

Tomás Alonso Albi[1]

[1]Astrofísico en el Observatorio Astronómico Nacional, Spain. E-mail: talonsoalbi@gmail.com.

Keywords: programación, programming, efemérides, ephemerides, cálculo astronómico, astronomical computing

Este artículo adjunta un *software* accesible en `https://github.com/JCAAC-FAAE`

Resumen

En este número veremos cómo reducir las coordenadas de un objeto para obtener la posición del mismo en el cielo para un observador situado sobre la superficie terrestre. Veremos el caso más sencillo en el que suponemos que ya tenemos la posición eclíptica geocéntrica del objeto referida al equinoccio medio de la fecha, de manera que no es necesario aplicar la corrección de precesión, vista en entregas anteriores. También se introducirá cierta cantidad de código adicional para tener bien organizado los resultados de los cálculos, incluyendo los resultados de operaciones potencialmente iterativas, como obtener los instantes de salida y puesta del objeto. En el caso de cuerpos estáticos, como las estrellas, esta iteración no es necesaria.

Abstract

In this article we see how to reduce the coordinates of an object to obtain its position in the sky for an observer located on the Earth's surface. We will examine the simplest case, in which we assume that we already have the geocentric ecliptic position of the object referred to the mean equinox of the date, so it is not necessary to apply the correction for precession, as discussed in previous articles in this Section. We also introduce some additional code to keep the calculation results well organized, including the results of potentially iterative operations, such as obtaining the rise and set times of the object. In the case of static bodies, such as stars, this iteration is not necessary.

1. Introducción

En esta entrega sólo se mostrará el cálculo práctico de la posición del Sol, con un algoritmo de baja precisión con un error máximo en torno a 10 segundos de arco. También veremos el cálculo de la posición aparente de una estrella estática a partir de las coordenadas J2000 de los catálogos. En próximas entregas veremos cómo aplicar el código presentado aquí para obtener posiciones muy precisas del Sol y la Luna.

Como siempre, el código presentado en esta ocasión puede también encontrarse en el repositorio de GitHub [1] correspondiente a esta sección, cuyo uso (y posible adaptación a otros lenguajes) ayudará a evitar los frecuentes errores de codificación del pasado.

2. Organización de los resultados del cálculo de efemérides

En el código presentado en esta sección se pretende dar una sugerencia de cómo organizar la información de salida del cálculo de efemérides. Se trata de una clase Java que debe utilizarse para instanciar o crear objetos que contendrán los resultados de las efemérides de los distintos cuerpos. Cada objeto contendrá valores particulares de campos como el acimut, la elevación, la ascensión recta, declinación, o distancia de un objeto particular para unas condiciones concretas del cálculo, que incluyen, entre otras opciones,

el instante de cálculo y la posición del observador. En otros lenguajes de programación populares como Python también es posible crear este tipo de objetos.

El código que se presenta contiene además algunas funciones útiles. El método *toString* está destinado a mostrar los resultados de los valores contenidos en el objeto, para reportar los resultados de las efemérides calculadas. El método *setIlluminationPhase* permite rellenar el valor del campo *illuminationPhase* (fase de iluminación) siempre que se proporcione como entrada la posición del Sol. Este método debe utilizarse tras obtener las efemérides, y sólo para objetos distintos del Sol. El método *getDateAsString* es un método auxiliar del método *toString* para formatear fechas. En él se utilizan los métodos estáticos *getHMS* y *fmt02* de la clase *Util*, los cuales pueden consultarse en el repositorio. En el primero se proporciona la hora, minuto, y segundo de una fracción de día expresado en radianes, y en el segundo se formatea con dos dígitos un valor numérico, añadiendo un cero delante si el valor es menor a 10.

Es técnicamente posible prescindir de este tipo de código en la organización de la información de salida, pero poco recomendable, pues a medida que avancemos esto supondría, muy probablemente, tener un código menos limpio, y con más funciones repetidas en distintos ficheros.

```
1  package journal;
2
3  /**
4   * A class to hold the results of the ephemerides calculations.
5   */
6  public class EphemData {
7
8      /** Values for azimuth, elevation, rise, set, and transit for the Sun. Angles in
           radians, rise ...
9       * as Julian days in UT. Distance in AU */
10     public double azimuth, elevation, rise, set, transit, transitElevation, distance,
           rightAscension,
11       declination, illuminationPhase, eclipticLongitude, eclipticLatitude,
             angularRadius;
12
13     /**
14      * Main constructor
15      * @param azi Azimuth
16      * @param ele Elevation
17      * @param rise2 Rise
18      * @param set2 Set
19      * @param transit2 Transit
20      * @param transit_alt Transit elevation
21      * @param ra Right ascension
22      * @param dec Declination
23      * @param dist Distance
24      * @param eclLon Ecliptic longitude
25      * @param eclLat Ecliptic latitude
26      * @param angR Angular radius
27      */
28     public EphemData(double azi, double ele, double rise2, double set2,
29           double transit2, double transit_alt, double ra, double dec,
30           double dist, double eclLon, double eclLat, double angR) {
31       azimuth = azi;
32       elevation = ele;
33       rise = rise2;
34       set = set2;
35       transit = transit2;
36       transitElevation = transit_alt;
37       rightAscension = ra;
38       declination = dec;
39       distance = dist;
40       illuminationPhase = 100;
41       eclipticLongitude = eclLon;
42       eclipticLatitude = eclLat;
```

```
43          angularRadius = angR;
44      }
45
46      @Override
47      public String toString() {
48          String degSymbol = "\u00b0";
49          String lsep = Util.getLineSeparator();
50          StringBuilder sb = new StringBuilder();
51          sb.append(" Az:    "+(float) (azimuth * Constant.RAD_TO_DEG)+degSymbol + lsep);
52          sb.append(" El:    "+(float) (elevation * Constant.RAD_TO_DEG)+degSymbol +
                lsep);
53          sb.append(" Dist:  "+(float) (distance)+" au" + lsep);
54          sb.append(" RA:    "+(float) (rightAscension * Constant.RAD_TO_DEG)+degSymbol +
                lsep);
55          sb.append(" DEC:   "+(float) (declination * Constant.RAD_TO_DEG)+degSymbol +
                lsep);
56          sb.append(" Ill:   "+(float) (illuminationPhase)+"%" + lsep);
57          sb.append(" ang.R: "+(float) (angularRadius * Constant.RAD_TO_DEG)+degSymbol +
                lsep);
58          sb.append(" Rise:  "+EphemData.getDateAsString(rise) + lsep);
59          sb.append(" Set:   "+EphemData.getDateAsString(set) + lsep);
60          sb.append(" Transit: "+EphemData.getDateAsString(transit)+" (elev. "+(float)
                (transitElevation * Constant.RAD_TO_DEG)+degSymbol+")" + lsep);
61          return sb.toString();
62      }
63
64      /**
65       * Sets the illumination phase field for the body
66       * @param sun The Ephem object for the Sun
67       */
68      public void setIlluminationPhase(EphemData sun) {
69          double dlon = rightAscension - sun.rightAscension;
70          double cosElong = (Math.sin(sun.declination) * Math.sin(declination) +
71                  Math.cos(sun.declination) * Math.cos(declination) * Math.cos(dlon));
72
73          double rsun = sun.distance;
74          double rbody = distance;
75          // Use elongation cosine as trick to solve the rectangle and get rp (distance
                body - sun)
76          double rp = Math.sqrt(-(cosElong * 2.0 * rsun * rbody - rsun * rsun - rbody *
                rbody));
77
78          double dph = ((rp * rp + rbody * rbody - rsun * rsun) / (2.0 * rp * rbody));
79          illuminationPhase = 100 * (1.0 + dph) * 0.5;
80      }
81
82      /**
83       * Returns a date as a string yyyy/mm/dd hh:mm:ss UT
84       * @param jd The Julian day
85       * @return The String
86       */
87      public static String getDateAsString(double jd) {
88          if (jd == -1) return "NO RISE/SET/TRANSIT FOR THIS OBSERVER/DATE";
89
90          JulianDay julDay = new JulianDay(jd);
91          double[] hms = Util.getHMS(julDay.getDayFraction() * Constant.TWO_PI);
92          String out = Util.fmt02(julDay.year, "/");
93          out += Util.fmt02(julDay.month, "/");
94          out += Util.fmt02(julDay.day, " ");
95          out += Util.fmt02((int) hms[0], ":");
96          out += Util.fmt02((int) hms[1], ":");
97          out += Util.fmt02((int) (hms[2] + 0.5), " UT");
98          return out;
99      }
100 }
```

3. Reducción de coordenadas

Esta es la parte principal del código, que permite la reducción de coordenadas para obtener la posición del objeto desde un observador en la Tierra, a partir de la posición media en coordenadas eclípticias, referida al equinoccio medio de la fecha. El proceso principal se encuentra en el método *doCalc*, en el cual se proporcionan esas coordenadas y se especifica si la reducción debe ser geocéntrica (para un observador hipotético situado en el centro de la Tierra), o referida a un observador real situado en la superficie de la Tierra (topocéntrica). En general, la reducción será siempre topocéntrica, salvo en cálculos específicos como obtener los instantes de los equinoccios y solsticios, o las fases lunares.

El método *doCalc* primero convierte las coordenadas medias en verdaderas sumando la nutación en longitud eclíptica. A continuación, las coordenadas son corregidas por la posición del observador en la Tierra, teniendo en cuenta tanto la forma de geoide de la Tierra como la altura del observador sobre la superficie (variables *geocLat*, *geocR*, y *eradius*). Las coordenadas son después corregidas por aberración diurna, un efecto menor, aunque relevante si queremos obtener posiciones precisas, con errores menores a 0.1". A continuación, se obtienen las coordenadas horizontales de acimut y altura, rotando directamente las coordenadas sin utilizar el código del fichero *CoordinateSystem*, donde la rotación requeriría de dos pasos y no se ahorrarían líneas de código. El código presente después permite corregir la altura por refracción, y obtener los instantes de salida y puesta del objeto, suponiendo que sea estático en el cielo. En estos cálculos se han introducido nuevas constantes en el fichero *Constant.java*: *SPEED_OF_LIGHT* (299792458.0 m/s), y *SIDEREAL_DAY_LENGTH* (1.00273781191135448 días). En el repositorio aparecen comentadas en la clase Java mencionada.

En cuanto a la corrección por refracción, es interesante notar que las fórmulas clásicas como la de Bennet permiten obtener la altura geométrica a partir de la aparente, pues estas fórmulas se derivaron a partir de observaciones en las que lógicamente se medía la altura aparente. Dado que en las efemérides se obtiene la altura geométrica, es necesario invertir numéricamente la fórmula para aplicarla correctamente. Como detalle, en el método *getGeometricElevation* se muestra comentado un código que permitiría corregir por refracción en el caso de que las observaciones se hagan en longitudes de onda radio, en vez de en el óptico. En estos cálculos el efecto de depresión del horizonte no se considera, de manera que la altura proporcionada se refiere al horizonte astronómico.

El método *getBodyPosition* no contiene código en este caso y devuelve un array nulo. La idea es que en otras piezas de código que se deriven de este fichero ese método quede reemplazado por el código necesario para obtener la posición de un objeto particular. Para ello se utiliza la propiedad de inheritancia, como se describe en la sección siguiente.

El método *getEphemeris* es de tipo estático (no es específico de un objeto de esa clase, sino común a todos ellos), presente con la idea de ser llamado desde otros ficheros para hacer todos los cálculos, incluyendo las iteraciones necesarias para obtener la salida y puesta de objetos que se mueven en el cielo. Estas iteraciones requieren de cambiar la fecha de cálculo para seguir el movimiento del objeto, lo que se hace en el método *setUTDate*.

También es necesario mencionar las construcciones de tipo *enum* presentes en el fichero, que incluyen las distintas opciones para seleccionar el tipo de salida y puesta que se calcula (*TWILIGHT*, que selecciona si los resultados se refieren al orto y ocaso civil, naútico, o astronómico), o para qué fecha deben estar referidos estos valores (*TWILIGHT_MODE*, que selecciona si los resultados deben referirse al momento más cercano al instante de cálculo, o al día actual en hora local o UT). El *enum EVENT* sólo se utiliza internamente y no forma parte de las opciones que deben especificarse para el cálculo de efemérides. En el pasado, lenguajes como Fortran que no tienen este tipo de construcciones requerían de valores enteros o *flags* para especificar este tipo de opciones, de manera que podía introducirse cualquier valor, correcto o no, y el código requería de chequeos para lanzar errores en caso necesario. Además, estos valores numéricos carecían de sentido por sí mismos, y era necesario documentar los valores disponibles y su

significado, y leer el código para saber dar el valor correcto. Los campos de tipo *enum* en Java evitan este problema, pues en ellos sólo hay un número limitado de valores aceptables, y todos ellos vienen expresados por un nombre que identifica de qué estamos hablando. Si se intenta poner un valor distinto, directamente el código no podrá ser compilado y ejecutado, de manera que no hacen falta chequeos. Otros lenguajes incluyen construcciones similares al enum de Java, por ejemplo Python a partir de la version 3.

```
1  package journal;
2
3  /**
4   * A class to reduce the position of a body to obtain the ephemerides as visible from
        a given observer, include rise/set times.
5   */
6  public class EphemReduction {
7
8      /** The set of twilights to calculate (types of rise/set events) */
9      public enum TWILIGHT {
10        /** Identifier to compute rise/set times for astronomical twilight (center of
              the body at -18 degrees of geometrical elevation) */
11        ASTRONOMICAL(-18),
12        /** Identifier to compute rise/set times for nautical twilight (center of the
              body at -12 degrees of geometrical elevation) */
13        NAUTICAL(-12),
14        /** Identifier to compute rise/set times for civil twilight (center of the body
              at -6 degrees of geometrical elevation) */
15        CIVIL(-6),
16        /** The standard value of 34' for the refraction at the local horizon */
17        HORIZON_34arcmin(-34.0 / 60.0);
18
19        /** Target elevation of the center of the body in degrees */
20        private double elevation;
21
22        TWILIGHT(double elev) {
23           elevation = elev;
24        }
25     }
26
27     /** Possible options to return the rise/set/transit times */
28     public enum TWILIGHT_MODE {
29        /** Closest events */
30        CLOSEST,
31        /** Compute events for the current date in UT */
32        TODAY_UT,
33        /** Compute events for the current date in LT */
34        TODAY_LT;
35     }
36
37     /** The set of events to calculate (rise/set/transit events) */
38     public enum EVENT { RISE, SET, TRANSIT }
39
40     protected double jd_UT;
41     private double nutLon;
42     protected double obsLon, obsLat, obsAlt;
43     protected double lst;
44     protected TWILIGHT twilight;
45     protected TWILIGHT_MODE twilightMode;
46     protected int timeZone = 0; /** Time zone for option {@linkplain
           TWILIGHT_MODE#TODAY_LT}, LT-UT, hours. */
47
48     /**
49      * The constructor with the data for the ephemerides reduction process
50      * @param jd_utc
51      * @param lon
52      * @param lat
```

```
53       * @param alt
54       * @param tw
55       * @param twm
56       * @param tz
57       */
58      public EphemReduction(double jd_utc, double lon, double lat, double alt, TWILIGHT
             tw, TWILIGHT_MODE twm, int tz) {
59         obsLon = lon * Constant.DEG_TO_RAD;
60         obsLat = lat * Constant.DEG_TO_RAD;
61         obsAlt = alt;
62         twilight = tw;
63         twilightMode = twm;
64         timeZone = tz;
65         setUTDate(jd_utc);
66      }
67
68      /**
69       * Sets the UT date from the provided Julian day and computes the nutation and
               sidereal time
70       * @param jd The new Julian day in UT
71       */
72      public void setUTDate(double jd) {
73         this.jd_UT = jd;
74         double[] nut = EarthAngles.nutation(jd);
75         nutLon = nut[0];
76         lst = EarthAngles.localApparentSiderealTime(jd, obsLon);
77      }
78
79      /**
80       * Compute the position of the body
81       * @param pos Values for the ecliptic longitude, latitude, distance and so on from
               previous methods for the specific body
82       * @param geocentric True to return geocentric position. Set this to false
               generally
83       * @return The Ephem object with the output position. The rise/set/transit times
               returned here are only valid for non-moving bodies
84       */
85      public EphemData doCalc(double[] pos, boolean geocentric) {
86         // Correct for nutation in longitude
87         pos[0] = pos[0] + nutLon;
88
89         // Ecliptic to equatorial coordinates using true obliquity
90         double[] xyz =
               CoordinateSystem.eclipticToEquatorial(CoordinateSystem.sphericalToCartesian(pos[0],
               pos[1]), jd_UT);
91
92         // Obtain topocentric rectangular coordinates
93         double geocLat = (obsLat - .1925 * Math.sin(2 * obsLat) * Constant.DEG_TO_RAD);
94         double geocR = 1.0 - Math.pow(Math.sin(obsLat), 2) / 298.257;
95         double eradius = (geocR * Constant.EARTH_RADIUS + obsAlt * 0.001);
96         double radiusAU = eradius / (pos[2] * Constant.AU);
97         if (!geocentric) {
98            double cosLat = Math.cos(geocLat);
99            double[] correction = new double[] {
100                 radiusAU * cosLat * Math.cos(lst),
101                 radiusAU * cosLat * Math.sin(lst),
102                 radiusAU * Math.sin(geocLat)};
103           xyz[0] -= correction[0];
104           xyz[1] -= correction[1];
105           xyz[2] -= correction[2];
106        }
107
108        // Obtain spherical topocentric equatorial coordinates
109        double[] sph = CoordinateSystem.cartesianToSpherical(xyz);
110        double ra = sph[0], dec = sph[1], dist = pos[2] * sph[2];
```

```
111
112         // Correct the equatorial position by diurnal aberration (< 0.3")
113         if (!geocentric) {
114             double rotRate = Constant.SIDEREAL_DAY_LENGTH * Constant.TWO_PI /
                    Constant.SECONDS_PER_DAY; // rad/s
115             double factor = rotRate * (eradius * 1000.0) / Constant.SPEED_OF_LIGHT; //
                    v/c
116             double ddec = factor * Math.cos(geocLat) * Math.sin(dec) * Math.sin(angh);
117             if (Math.cos(dec) != 0.0) ra += factor * Math.cos(angh) * Math.cos(geocLat)
                    / Math.cos(dec);
118             dec += ddec;
119         }
120
121         // Hour angle
122         double angh = lst - ra;
123
124         // Obtain azimuth and geometric alt
125         double sinLat = Math.sin(obsLat);
126         double cosLat = Math.cos(obsLat);
127         double sinDec = Math.sin(dec), cosDec = Math.cos(dec);
128         double h = sinLat * sinDec + cosLat * cosDec * Math.cos(angh);
129         double alt = Math.asin(h);
130         double azx = Math.cos(angh) * sinLat - sinDec * cosLat / cosDec;
131         double azi = Math.PI + Math.atan2(Math.sin(angh), azx); // 0 = north
132
133         if (geocentric) return new EphemData(azi, alt, -1, -1, -1, -1,
                Util.normalizeRadians(ra), dec, dist, pos[0], pos[1], pos[3]);
134
135         // Get apparent elevation
136         alt = getApparentElevation(alt);
137
138         double tmp = twilight.elevation * Constant.DEG_TO_RAD;
139         // Consider the angular radius (pos[3]) for rise, set, transit times when using
                the HORIZON_34arcmin twilight.
140         // Removing angular radius here would do calculations for the center of the
                disk instead of the lower/upper limb.
141         if (twilight == TWILIGHT.HORIZON_34arcmin) tmp = tmp - pos[3];
142
143         // Compute cosine of hour angle
144         tmp = (Math.sin(tmp) - sinLat * sinDec) / (cosLat * cosDec);
145
146         // Make calculations for the meridian
147         double transit_alt = Math.asin(sinDec * sinLat + cosDec * cosLat);
148         transit_alt = getApparentElevation(transit_alt);
149
150         // Obtain the current transit event in time
151         double transit = getTwilightEvent(ra, 0);
152
153         // Make calculations for rise and set
154         double rise = -1, set = -1;
155         if (Math.abs(tmp) <= 1.0) {
156             double ang_hor = Math.abs(Math.acos(tmp));
157             rise = getTwilightEvent(ra, -ang_hor);
158             set = getTwilightEvent(ra, ang_hor);
159         }
160
161         EphemData out = new EphemData(azi, alt, rise, set, transit, transit_alt,
162               Util.normalizeRadians(ra), dec, dist, pos[0], pos[1], pos[3]);
163         return out;
164     }
165
166     private double getTwilightEvent(double ra, double angh) {
167         double celestialHoursToEarthTime = 1.0 / (Constant.SIDEREAL_DAY_LENGTH *
                Constant.TWO_PI);
168         double jdToday_UT = Math.floor(jd_UT - 0.5) + 0.5;
```

```
169
170         double eventTime = celestialHoursToEarthTime * Util.normalizeRadians(ra + angh
                - lst);
171         double eventTimePrev = celestialHoursToEarthTime * (Util.normalizeRadians(ra +
                angh - lst) - Constant.TWO_PI);
172         double eventDatePrev_UT = Math.floor(jd_UT + eventTimePrev - 0.5) + 0.5;
173
174         if (Math.abs(eventTimePrev) < Math.abs(eventTime) && twilightMode ==
                TWILIGHT_MODE.CLOSEST) eventTime = eventTimePrev;
175         if (twilightMode == TWILIGHT_MODE.TODAY_UT) {
176             double eventDate_UT = Math.floor(jd_UT + eventTime - 0.5) + 0.5;
177             if (jdToday_UT != eventDate_UT) eventTime = -jd_UT - 1;
178             if (jdToday_UT == eventDatePrev_UT) eventTime = eventTimePrev;
179         }
180         if (twilightMode == TWILIGHT_MODE.TODAY_LT) {
181             double tz = timeZone / 24.0, jdToday_LT = Math.floor(jd_UT + tz - 0.5) + 0.5;
182             double eventDate_LT = Math.floor(jd_UT + tz + eventTime - 0.5) + 0.5;
183             if (jdToday_LT != eventDate_LT) eventTime = -jd_UT - 1;
184
185             double eventDatePrev_LT = Math.floor(jd_UT + tz + eventTimePrev - 0.5) + 0.5;
186             if (jdToday_LT == eventDatePrev_LT) eventTime = eventTimePrev;
187
188             double eventTimeNext = celestialHoursToEarthTime * (Util.normalizeRadians(ra
                    + angh - lst) + Constant.TWO_PI);
189             double eventDateNext_LT = Math.floor(jd_UT + tz + eventTimeNext - 0.5) + 0.5;
190             if (jdToday_LT == eventDateNext_LT) eventTime = eventTimeNext;
191         }
192
193         return jd_UT + eventTime;
194     }
195
196     /** Corrects input geometric elevation for refraction if it is greater than -3
            degrees, returning the apparent elevation */
197     private double getApparentElevation(double alt) {
198         if (alt <= -3 * Constant.DEG_TO_RAD) return alt;
199
200         double altIn = alt, prevAlt = alt;
201         int niter = 0;
202         do {
203             double altOut = getGeometricElevation(alt);
204             alt = altIn - (altOut-alt);
205             niter ++;
206             if (Math.abs(prevAlt-alt) < 0.001 * Constant.DEG_TO_RAD) break;
207             prevAlt = alt;
208         } while (niter < 8);
209
210         return alt;
211     }
212
213     /** Compute geometric elevation from apparent elevation. Note ephemerides
            calculates geometric elevation, so an inversion is
214      * required, something achieved in method {@linkplain
            #getApparentElevation(double)} by iteration */
215     private double getGeometricElevation(double alt) {
216         double ps = 1010; // Pressure in mb
217         double ts = 10 + 273.15; // Temperature in K
218         double altDeg = alt * Constant.RAD_TO_DEG;
219
220         // Bennet 1982 formulae for optical wavelengths, do the job but not accurate
                close to horizon
221         double r = Constant.DEG_TO_RAD * Math.abs(Math.tan(Constant.PI_OVER_TWO -
                (altDeg + 7.31 / (altDeg + 4.4)) * Constant.DEG_TO_RAD)) / 60.0;
222         double refr = r * (0.28 * ps / ts);
223         return Math.min(alt - refr, Constant.PI_OVER_TWO);
224
```

```
225         /*
226         // Bennet formulae adapted to radio wavelenths. Use this for position in radio
                wavelengths
227         // Reference for some values:
                http://icts-yebes.oan.es/reports/doc/IT-OAN-2003-2.pdf (Yebes 40m
                radiotelescope)
228         double hs = 20; // Humidity %
229         // Water vapor saturation pressure following Crane (1976), as in the ALMA
                memorandum
230         double esat = 6.105 * Math.exp(25.22 * (ts - 273.15) / ts) + Math.pow(ts /
                273.15, -5.31);
231         double Pw = hs * esat / 100.0;
232
233         double R0 = (16.01 / ts) * (ps - 0.072 * Pw + 4831 * Pw / ts) *
                Constant.ARCSEC_TO_RAD;
234         double refr2 = R0 * Math.abs(Math.tan(Constant.PI_OVER_TWO - (altDeg + 5.9 /
                (altDeg + 2.5)) * Constant.DEG_TO_RAD));
235         return Math.min(alt - refr, Constant.PI_OVER_TWO);
236         */
237       }
238
239       /**
240        * Computes an accurate rise/set/transit time for a moving object
241        * @param riseSetJD Start date for the event
242        * @param index Event identifier
243        * @param niter Maximum number of iterations
244        * @return The Julian day in UT for the event, 1s accuracy
245        */
246       protected double obtainAccurateRiseSetTransit(double riseSetJD, EVENT index, int
               niter) {
247         double step = -1;
248         for (int i = 0; i< niter; i++) {
249           if (riseSetJD == -1) return riseSetJD; // -1 means no rise/set from that
                  location
250           setUTDate(riseSetJD);
251           EphemData out = doCalc(getBodyPosition(), false);
252
253           double val = out.rise;
254           if (index == EVENT.SET) val = out.set;
255           if (index == EVENT.TRANSIT) val = out.transit;
256           step = Math.abs(riseSetJD - val);
257           riseSetJD = val;
258           if (step <= 1.0 / Constant.SECONDS_PER_DAY) break; // convergency reached
259         }
260         if (step > 1.0 / Constant.SECONDS_PER_DAY) return -1; // did not converge =>
                without rise/set/transit in this date
261         return riseSetJD;
262       }
263
264       protected double[] getBodyPosition() {
265         return null;
266       }
267
268       /**
269        * Computes the ephemerides for a body, including accurate rise/set/transit times
270        * @param bodyRed The body reduction data
271        * @return The ephemerides data
272        */
273       public static EphemData getEphemeris(EphemReduction bodyRed) {
274         double jd_UT = bodyRed.jd_UT;
275         EphemData bodyData = bodyRed.doCalc(bodyRed.getBodyPosition(), false);
276         int niter = 15; // Number of iterations to get accurate rise/set/transit times
277         bodyData.rise = bodyRed.obtainAccurateRiseSetTransit(bodyData.rise, EVENT.RISE,
                niter);
```

```
278         bodyData.set = bodyRed.obtainAccurateRiseSetTransit(bodyData.set, EVENT.SET,
                niter);
279         bodyData.transit = bodyRed.obtainAccurateRiseSetTransit(bodyData.transit,
                EVENT.TRANSIT, niter);
280         if (bodyData.transit == -1) {
281             bodyData.transitElevation = 0;
282         } else {
283             // Update Sun's maximum elevation
284             bodyRed.setUTDate(bodyData.transit);
285             bodyData.transitElevation = bodyRed.doCalc(bodyRed.getBodyPosition(),
                    false).transitElevation;
286         }
287         bodyRed.setUTDate(jd_UT);
288         return bodyData;
289     }
290 }
```

4. Efemérides aproximadas del Sol

El código presentado en esta sección permite obtener la posición aproximada del Sol para un observador. Para que el lector pueda comprobar los resultados, el método *main* presente al final del listado contiene un ejemplo de cálculo junto con el resultado esperado.

El código utiliza inheritancia para poder utilizar el código contenido en la clase *EphemReduction*, descrita en la sección anterior, y poder así acceder a los algoritmos de reducción. Esta propiedad también puede utilizarse en otros lenguajes como Python. En el caso de que el lector utilice un lenguaje que no ofrezca esta característica, la opción más sencilla es tener las funciones de reducción como métodos externos que se puedan llamar para calcular las efemérides de cualquier cuerpo, o bien copiar y pegar el codigo del fichero *EphemReduction.java* dentro de cada uno de los ficheros en los que se calcularán las efemérides de los distintos objetos. La última opción conviene evitarla para no repetir código que debería ser compartido.

La parte esencial es el método *getBodyPosition*, que proporciona la longitud eclíptica, la latitud eclíptica (para el Sol se asume que es cero dado que es siempre muy pequeña), la distancia, y el radio angular del Sol, obtenidos para el equinoccio medio de la fecha. Algunos de los algoritmos utilizados para este cálculo provienen del trabajo de S. L. Moshier [2]. Los resultados están corregidos por aberración o tiempo-luz, de manera aproximada.

El método *main* contiene el ejemplo práctico. El objeto de tipo *EphemSunSimple* contienen las condiciones del cálculo de efemérides, las cuales se utilizan en el método estático *EphemReduction.getEphemeris*. Este método hace internamente todas las operaciones necesarias para iterar a la hora de obtener los instantes precisos de salida y puesta del objeto. El objeto de salida es de tipo *EphemData*, descrito al principio, el cual se utiliza después para expresar los resultados mediante su método *toString*, mencionado anteriormente. De esta manera, el código queda bastante limpio y este fichero permite calcular la posición del Sol sin incluir ningún otro código común.

```
1  package journal;
2
3  /**
4   * A class to compute the ephemerides of the Sun. This class uses the code inside
        {@linkplain EphemReduction} by inheritance.
5   */
6  public class EphemSunSimple extends EphemReduction {
7
8      public EphemSunSimple(double jd_utc, double lon, double lat, double alt, TWILIGHT
           tw, TWILIGHT_MODE twm, int tz) {
9          super(jd_utc, lon, lat, alt, tw, twm, tz);
```

```
10      }
11
12      @Override
13      public double[] getBodyPosition() {
14         double t = EarthAngles.toCenturiesRespectJ2000(jd_UT, true);
15
16         // Mean longitude of Sun corrected for precession, taken from Moshier.
17         // Final accuracy up to +/- 27" or better over many millenia compared to VSOP87
18         double xe = (129597742.283429 * t + 361679.198) + (-5.23e-6 * t - 2.04411e-2) *
               t * t;
19         xe += (((((((((-8.66e-20 * t - 4.759e-17) * t + 2.424e-15) * t + 1.3095e-12) *
               t + 1.7451e-10) * t - 1.8055e-8) * t - 0.0000235316) * t + 0.000076) * t +
               1.105414) * t + 5028.791959) * t;
20         double lon = Util.normalizeRadians(Math.PI + Constant.ARCSEC_TO_RAD * xe) *
               Constant.RAD_TO_DEG;
21
22         /* Mean anomaly of sun = l' (J. Laskar) */
23         double x = (1.2959658102304320e+08 * t + 1.2871027407441526e+06);
24         x += (((((((1.62e-20 * t - 1.0390e-17) * t - 3.83508e-15) * t + 4.237343e-13)
               * t + 8.8555011e-11) * t - 4.77258489e-8) * t - 1.1297037031e-5) * t +
               8.7473717367324703e-05) * t - 5.5281306421783094e-01) * t * t;
25         double sanomaly = Util.normalizeRadians(Constant.ARCSEC_TO_RAD * x);
26
27         double c = (1.9146 - .004817 * t - .000014 * t * t) * Math.sin(sanomaly);
28         c = c + (.019993 - .000101 * t) * Math.sin(2 * sanomaly);
29         c = c + .00029 * Math.sin(3.0 * sanomaly); // Correction to the mean ecliptic
               longitude
30
31         // Now compute approximate aberration
32         double d = -.00569;
33
34         double slongitude = lon + c + d; // apparent longitude (error<0.003 deg)
35         double slatitude = 0; // Sun's ecliptic latitude is always negligible
36         double ecc = .016708617 - 4.2037E-05 * t - 1.236E-07 * t * t; // Eccentricity
37         double v = sanomaly + c * Constant.DEG_TO_RAD; // True anomaly
38         double sdistance = 1.000001018 * (1.0 - ecc * ecc) / (1.0 + ecc * Math.cos(v));
               // In UA
39
40         return new double[] {slongitude * Constant.DEG_TO_RAD, slatitude *
               Constant.DEG_TO_RAD, sdistance, Math.atan(696000.0 / (sdistance *
               Constant.AU))};
41      }
42
43      /**
44       * Test program
45       * @param args Not used
46       */
47      public static void main(String[] args) {
48         // Prepare input data
49         int year = 2020, month = 6, day = 9, h = 18, m = 0, s = 0;
50         JulianDay jday = new JulianDay(year, month, day);
51         jday.setDayFraction((h + m / 60.0 + s / 3600.0) / 24.0);
52
53         double jd_utc = jday.getJulianDay();
54         double lon = -4; // degrees
55         double lat = 40;
56         double alt = 0; // m
57         int tz = 3; // h
58         TWILIGHT tw = TWILIGHT.HORIZON_34arcmin;
59         TWILIGHT_MODE twm = TWILIGHT_MODE.TODAY_UT;
60
61         // Compute the ephemerides data
62         EphemSunSimple sunEph = new EphemSunSimple(jd_utc, lon, lat, alt, tw, twm, tz);
63         EphemData sunData = EphemReduction.getEphemeris(sunEph);
64
```

```
        // Report
        System.out.println("Sun");
        System.out.println(sunData.toString());

/*
Sun
 Az:      285.78873°
 El:      17.424834°
 Dist:    1.015206 au
 RA:      78.41086°
 DEC:     23.007696°
 Ill:     100.0%
 ang.R:   0.26256925°
 Rise:    2020/06/09 04:46:57 UT
 Set:     2020/06/09 19:43:59 UT
 Transit: 2020/06/09 12:15:22 UT (elev. 72.99493°)
 */
    }
}
```

5. Posición aparente de estrellas y otros objetos de catálogos

Las coordenadas de estrellas y objetos de cielo profundo presentes en los catálogos son fijas, al margen de posibles movimientos propios que puedan tener esos objetos a lo largo de los siglos. Estas coordenadas son también medias, como las calculadas en los métodos anteriores, pero están referidas a un equinoccio particular, habitualmente el equinoccio 2000. Para obtener las efemérides de una estrella primero debemos corregir la posición del catálogo por precesión, y después aplicar el método de reducción.

Para ello, una vez instanciado el objeto *EphemStar* en el ejemplo, equivalente al *EphemSunSimple* de la sección anterior, se utiliza el método *setJ2000Position* para establecer las coordenadas de catálogo del objeto, las cuales están incluídas como variables privadas en la clase Java. Por consistencia y claridad, el constructor no ha sido modificado respecto a códigos anteriores, aunque se podría incluir la introducción de esas coordenadas en el constructor y evitar la necesidad del método *setJ2000Position*. El método *getBodyPosition* devuelve las coordenadas de la estrella, corregiendo primero por precesión mediante el método *equatorialJ2000ToMeanEquatorialOfDate*, que contiene un código idéntico al mostrado en una entrega anterior de esta sección [3] para la clase *CoordinateSystem*, pero adaptado para la fecha introducida en el constructor. Este código debe estar lógicamente implementado previamente, y disponible con ese nombre de fichero en el mismo directorio o paquete, junto con el resto del código presentado en entregas anteriores.

El ejemplo implementa el cálculo de la posición aparente de la estrella Vega, a partir de su posición J2000 18h 36m 56s, 38° 47' 00". La posición aparente se muestra al ejecutar el programa, y puede compararse con la proporcionada en el portal *theskylive.com*. No es posible cuantificar con claridad la diferencia que se obtiene en este programa con respecto al valor proporcionado en esa página web [4], sólo puede decirse que debe estar claramente por debajo del segundo de arco.

```
package journal;

/**
 * A class to compute the ephemerides of a star, or any other catalog body.
 * This class uses the code inside {@linkplain EphemReduction} by inheritance.
 */
public class EphemStar extends EphemReduction {

    private double ra;
    private double dec;

```

```
12      public EphemStar(double jd_utc, double lon, double lat, double alt, TWILIGHT tw,
            TWILIGHT_MODE twm, int tz) {
13         super(jd_utc, lon, lat, alt, tw, twm, tz);
14      }
15
16      /**
17       * Sets the J2000 position of the body from catalog coordinates
18       * @param ra Right ascension, hours
19       * @param dec Declination, degrees
20       */
21      public void setJ2000Position(double ra, double dec) {
22         this.ra = ra * 15.0 * Constant.DEG_TO_RAD;
23         this.dec = dec * Constant.DEG_TO_RAD;
24      }
25
26      /**
27       * Transforms coordinates from J2000 equatorial to mean equinox of date
28       * @param p Rectangular coordinates
29       * @return Mean equatorial rectangular coordinates
30       */
31      public double[] equatorialJ2000ToMeanEquatorialOfDate(double[] p) {
32         double t = EarthAngles.toCenturiesRespectJ2000(jd_UT, true);
33         double[] pa = EarthAngles.precessionAnglesFromJ2000(Constant.J2000 + t *
               Constant.JULIAN_DAYS_PER_CENTURY);
34         return
               CoordinateSystem.rotate(CoordinateSystem.rotate(CoordinateSystem.rotate(p,
               CoordinateSystem.getRotZ(-pa[1])), CoordinateSystem.getRotY(pa[2])),
               CoordinateSystem.getRotZ(-pa[0]));
35      }
36
37      @Override
38      public double[] getBodyPosition() {
39         double[] cartesianJ2000 = CoordinateSystem.sphericalToCartesian(ra, dec);
40         double[] meanEq = equatorialJ2000ToMeanEquatorialOfDate(cartesianJ2000);
41         double[] meanEcl = CoordinateSystem.equatorialToEcliptic(meanEq, jd_UT);
42         double[] meanEclSph = CoordinateSystem.cartesianToSpherical(meanEcl);
43         return new double[] {meanEclSph[0], meanEclSph[1], 1E100, 0}; // Assume
               infinite distance and angular size 0
44      }
45
46      /**
47       * Test program
48       * @param args Not used
49       */
50      public static void main(String[] args) {
51         // Julian day for current instant
52         JulianDay jday = new JulianDay(System.currentTimeMillis());
53
54         // Observer position (Madrid, as in theskylive.com)
55         double lon = -(3 + 42 / 60.0 + 9.2 / 3600.0); // degrees
56         double lat = 40 + 24 / 60.0 + 59.4 / 3600.0;
57         double alt = 0; // m
58         int tz = 2; // h
59
60         // Catalog body, J2000 coordinates. Check Vega with
               https://theskylive.com/sky/stars/vega-alpha-lyrae-star
61         String name = "Vega";
62         double ra = 18.0 + 36 / 60.0 + 56 / 3600.0;
63         double dec = 38 + 47 / 60.0 + 0 / 3600.0;
64
65         // Compute the ephemerides data
66         EphemStar bodyEph = new EphemStar(jday.getJulianDay(), lon, lat, alt,
               TWILIGHT.HORIZON_34arcmin, TWILIGHT_MODE.TODAY_UT, tz);
67         bodyEph.setJ2000Position(ra, dec);
68         EphemData bodyData = EphemReduction.getEphemeris(bodyEph);
```

```
69
70          // Report
71          System.out.println(name);
72          System.out.println(bodyData.toString());
73
74          System.out.println(" RA: " + Util.formatRA(bodyData.rightAscension, 1));
75          System.out.println(" DEC: " + Util.formatDEC(bodyData.declination, 0));
76      }
77  }
```

References

[1] Repositorio de código en GitHub, https://github.com/JCAAC-FAAE

[2] Algoritmos de S. L. Moshier: https://www.moshier.net/

[3] *Astronomical computing: Cálculo de ángulos de orientación de la Tierra y transformaciones de coordenadas*, T. Alonso Albi, JCAAC **2**, 87 (2025).

[4] Posición aparente de la estrella Vega: https://theskylive.com/sky/stars/vega-alpha-lyrae-star. Para comparar con el programa es necesario introducir antes la posición del observador (Madrid en el ejemplo del programa)