JCAAC (2025), **4**, 45–60

JCAAC

SECTION: ASTRONOMICAL COMPUTING

Efemérides precisas del Sol y la Luna

Tomás Alonso Albi[1]

[1]Astrónomo del Observatorio Astronómico Nacional - Instituto Geográfico Nacional, Spain. E-mail: talonsoalbi@gmail.com.

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

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

Resumen

En este número presentamos algoritmos para el cálculo preciso de la posición del Sol y la Luna para un observador en la Tierra, utilizando el procedimiento de reducción explicado en el número anterior. Para alcanzar la mejor consistencia posible, respecto de la última integración numérica disponible del JPL, se han modificado algoritmos clásicos, de manera notable en el caso de la Luna. Las discrepancias máximas que se obtienen son de 2" para la posición geocéntrica del Sol, y en torno a 15" para la de la Luna, durante al menos 4000 años alrededor del año 2000. También se presentan cambios que se han introducido en ficheros publicados con anterioridad, para corregir algunos problemas.

Abstract

In this article we will see how to compute the accurate positions of the Sun and the Moon for any observer on Earth, using the reduction procedures presented in a previous article in this Section. To achieve the maximum consistency, with respect to the latest JPL numerical integration, some classic algorithms has been modified, especially for the Moon. The maximum discrepancies found are of 2" for the geocentric posicion of the Sun, and around 15" for that of the Moon, during at least 4000 years around the year 2000. Some corrections to files published previously are also presented, solving different problems identified.

1. Introducción

En este número nos centraremos en obtener posiciones precisas del Sol y la Luna para un observador en la Tierra, necesarias para múltiples propósitos como el cálculos de eclipses, entre otros, que iremos viendo más adelante. Estos cálculos no son triviales, puesto que la posición del Sol se ve afectada por el propio movimiento de la Tierra alrededor del centro de masas del sistema que forma conjuntamente con la Luna, mientras que el movimiento de la Luna se ve afectado por la Tierra y otros cuerpos. La Luna ejerce un efecto sobre las mareas terrestres, que también afecta a su movimiento.

Trataremos entonces sobre las posiciones del Sol y la Luna, y veremos que con algoritmos no excesivamente extensos es posible lograr resultados que difieren en pocos segundos de arco respecto de la última integración numérica del JPL. Esto puede conseguirse mediante una ligera modificación o actualización de algoritmos clásicos. Posteriormente veremos cómo calcular también las posiciones precisas de los planetas y otros cuerpos menores, con una precisión similar, y seguiremos con el cálculo de eclipses, justo antes de que lleguen los eclipses de Sol visibles desde España, a partir de agosto de 2026.

Como siempre, el código presentado en esta ocasión puede también encontrarse en el repositorio de GitHub [1] correspondiente a esta sección, el cual será útil para los lectores que quieran utilizarlo tal

cual, o bien reescribirlo en su lenguaje de programación favorito. Es importante comentar que recientemente se han introducido correcciones en el código de cuatro ficheros publicados en números anteriores: *CoordinateSystem*, *EarthAngles*, *EphemReduction*, y *Constant*. En el primero se ha corregido la función *rotate*, que devolvía valores incorrectos al rotar las componentes de velocidad, si se incluían en la entrada. En el segundo se ha corregido un error en la función que calcula la nutación, se ha reemplazado la formulación de la IAU (1976) de la oblicuidad media por la de Laskar (1986), que ya estaba presente pero comentada en el código, y se ha forzado el valor TT-UT1 devuelto en la función *TTminusUT1* al valor 69.2s, cuando la fecha de entrada corresponde a un año entre el 2018 y 2030. Como se explicó en su momento, la diferencia TT-UT1 ha dejado en los últimos años de aumentar al ritmo previsto a largo plazo por el efecto de las mareas. Podría deberse a un cambio en la rotación del núcleo terrestre, o al deshielo de los polos y glaciares, pero el hecho es que la diferencia respecto al valor esperado alcanza ya los 7s, motivo por el que se ha actualizado el código presente en el repositorio. En el tercer fichero se ha establecido un valor mínimo para la distancia entre el objeto calculado y el centro de la Tierra, para el caso en que el objetivo sea el propio centro de la Tierra. En el cuarto fichero se ha retocado ligeramente el valor de la Unidad Astronómica, para hacerla consistente con la integración DE440, sin que haya consecuencias prácticas en los cálculos. Se recomienda revisar estos cambios e introducirlos en cualquier código que los lectores estén desarrollando a partir de ellos.

2. Efemérides precisas del Sol

El algoritmo que se propone en esta sección para calcular la posición del Sol procede de la publicación *Planetary Programs and Tables* [2], de 1986. Se trata de un ajuste a la integración DE200 durante miles de años, entre el 4000 a.C. y el 6000 d.C, que permite obtener la posición eclíptica geocéntrica del Sol para el equinoccio medio de la fecha, que es justo lo que necesitamos para aplicar el procedimiento de reducción de coordenadas explicado en el número anterior [3]. El error máximo de este código es de unos dos segundos de arco en longitud eclíptica. Además, al considerar que la latitud eclíptica del Sol es siempre cero, se introduce un error en latitud que puede llegar en casos extremos al segundo de arco.

En el código propuesto aquí se ha introducido el principal término correctivo en latitud eclíptica, debido al movimiento conjunto de la Tierra y la Luna. Este término es muy fácil de entender: dado que la Tierra gira alrededor del centro de masas que conforma con la Luna, y que ésta gira alrededor de la Tierra con una inclinación de 5 grados respecto de la eclíptica, cuando la Luna está en su máxima latitud eclíptica, la Tierra debe estar en la posición simétrica, con mínima latitud eclíptica. En esta situación, la posición geocéntrica del Sol tendrá su latitud eclíptica máxima, correspondiente por este término a unos 0.6", equivalente a un movimiento de 450 km en la posición de la Tierra, pequeño dado que es mucho más masiva que la Luna, pero considerable. Este término se ha introducido en un método aparte (*getSolarGeocenticEclipticLatitude*) para clarificar que se trata de una corrección al algoritmo original. Con este término no se reduce el error máximo del algoritmo, pero sí se reduce el error habitual que se puede esperar, a valores típicos por debajo del segundo de arco.

Además de esta corrección, la línea 66 presenta una pequeña corrección adicional en longitud eclíptica, descrita con detalle por Meeus [6] e implementada aquí de manera simplificada, para transformar las coordenadas del sistema de referencia utilizado por los autores en [2], denominado equinoccio dinámico J2000 (el mismo que en la teoría analítica VSOP87, desarrollada por los mismos autores para representar las posiciones de los planetas), al sistema de referencia J2000 estándard. Es poco relevante y se incluye por consistencia. Los tests sugieren que reduce las discrepancias con integraciones posteriores.

Pero la corrección más relevante se encuentra en la línea siguiente, la 67, en la que se introduce una corrección totalmente fuera del algoritmo original, con el objetivo de reproducir mejor los resultados devueltos por el servidor Horizons. Esta corrección se comenta con más detalle en la última sección. Se trata de usar el día Juliano en Tiempo Terrestre para aplicar una corrección empírica obtenida mediante

la comparación directa con Horizons. Con esto el error máximo de 2" ocurre en muy raras ocasiones.

En diferentes obras se pueden encontrar algoritmos parecidos, con los cuales sería en principio posible lograr precisiones ligeramente mejores a base de introducir más términos, tanto en longitud eclíptica como en latitud. Las pruebas realizadas dieron como resultado que el código propuesto es posiblemente de los más eficientes que se han desarrollado, pues proporciona las posiciones que necesitamos para la reducción, y contiene un número de términos suficientemente bajo como para presentarlo aquí, con unos 51 en total para las tres coordenadas de longitud, latitud, y distancia. Además, está actualizado para reproducir los resultados de Horizons durante milenios.

El código utiliza el mecanismo de inheritancia que se explicó en el número anterior [3], con el objetivo de reducir la repetición de código. El cálculo de la posición tiene lugar en el método *getBodyPosition*, en el que se usan los términos introducidos en *sunElements* y se llama al método para obtener la latitud eclíptica y corregir la longitud. Las coordenadas devueltas son luego utilizadas en la reducción. Existe otro método llamado *getGeometricEclipticPositionEquinoxOfDate*, el cual devuelve posiciones y velocidades cartesianas geométricas, en vez de coordenadas esféricas corregidas por aberración. Lo utilizaremos en el próximo número para obtener las posiciones de los planetas y poder hacer correcciones de aberración.

El código se completa con el método *getEquinoxesAndSolstices*, el cual es una demostración del uso de un algoritmo que podríamos llamar de fuerza bruta, para obtener una configuración específica, en este caso el instante en que la ascensión recta del Sol (o bien, opcionalmente, siendo algo más rigurosos, su longitud eclíptica, usando la línea comentada) alcanza valores de 0, 90, 180, y 270 grados, correspondientes a los instantes de los equinoccios y solsticios. Dado que la Tierra gira alrededor del Sol aproximadamente un grado por día, o 3600" por cada 86400s de tiempo, con el error del algoritmo los instantes devueltos estarán mal por unos 24s por cada segundo de arco de error, que es el error más habitual. La gran velocidad de los procesadores actuales hace conveniente el uso de este tipo de algoritmos, mucho más sencillos de comprender y limitados por la precisión del algoritmo principal, que otros más complejos desarrollados en el pasado por las limitaciones de velocidad. En un futuro veremos algoritmos de este tipo, aunque más elaborados, para el cálculo de eclipses.

```
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 EphemSun extends EphemReduction {
7
8      public EphemSun(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     // Sun data from "Planetary Programs and Tables" by Pierre Bretagnon and
           Jean-Louis Simon, Willman-Bell, 1986
13     private static double[][] sunElements = {
14         new double[] { 403406.0, 0.0, 4.721964, 1.621043 }, new double[] { 195207.0,
               -97597.0, 5.937458, 62830.348067 },
15         new double[] { 119433.0, -59715.0, 1.115589, 62830.821524 }, new double[] {
               112392.0, -56188.0, 5.781616, 62829.634302 },
16         new double[] { 3891.0, -1556.0, 5.5474, 125660.5691 }, new double[] { 2819.0,
               -1126.0, 1.512, 125660.9845 },
17         new double[] { 1721.0, -861.0, 4.1897, 62832.4766 }, new double[] { 0.0,
               941.0, 1.163, .813 },
18         new double[] { 660.0, -264.0, 5.415, 125659.31 }, new double[] { 350.0,
               -163.0, 4.315, 57533.85 },
19         new double[] { 334.0, 0.0, 4.553, -33.931 }, new double[] { 314.0, 309.0,
               5.198, 777137.715 },
```

```
        new double[] { 268.0, -158.0, 5.989, 78604.191 }, new double[] { 242.0, 0.0,
            2.911, 5.412 },
        new double[] { 234.0, -54.0, 1.423, 39302.098 }, new double[] { 158.0, 0.0,
            .061, -34.861 },
        new double[] { 132.0, -93.0, 2.317, 115067.698 }, new double[] { 129.0, -20.0,
            3.193, 15774.337 },
        new double[] { 114.0, 0.0, 2.828, 5296.67 }, new double[] { 99.0, -47.0, .52,
            58849.27 },
        new double[] { 93.0, 0.0, 4.65, 5296.11 }, new double[] { 86.0, 0.0, 4.35,
            -3980.7 },
        new double[] { 78.0, -33.0, 2.75, 52237.69 }, new double[] { 72.0, -32.0, 4.5,
            55076.47 },
        new double[] { 68.0, 0.0, 3.23, 261.08 }, new double[] { 64.0, -10.0, 1.22,
            15773.85 },
        new double[] { 46.0, -16.0, .14, 188491.03 }, new double[] { 38.0, 0.0, 3.44,
            -7756.55 },
        new double[] { 37.0, 0.0, 4.37, 264.89 }, new double[] { 32.0, -24.0, 1.14,
            117906.27 },
        new double[] { 29.0, -13.0, 2.84, 55075.75 }, new double[] { 28.0, 0.0, 5.96,
            -7961.39 },
        new double[] { 27.0, -9.0, 5.09, 188489.81 }, new double[] { 27.0, 0.0, 1.72,
            2132.19 },
        new double[] { 25.0, -17.0, 2.56, 109771.03 }, new double[] { 24.0, -11.0,
            1.92, 54868.56 },
        new double[] { 21.0, 0.0, .09, 25443.93 }, new double[] { 21.0, 31.0, 5.98,
            -55731.43 },
        new double[] { 20.0, -10.0, 4.03, 60697.74 }, new double[] { 18.0, 0.0, 4.27,
            2132.79 },
        new double[] { 17.0, -12.0, .79, 109771.63 }, new double[] { 14.0, 0.0, 4.24,
            -7752.82 },
        new double[] { 13.0, -5.0, 2.01, 188491.91 }, new double[] { 13.0, 0.0, 2.65,
            207.81 },
        new double[] { 13.0, 0.0, 4.98, 29424.63 }, new double[] { 12.0, 0.0, .93,
            -7.99 },
        new double[] { 10.0, 0.0, 2.21, 46941.14 }, new double[] { 10.0, 0.0, 3.59,
            -68.29 },
        new double[] { 10.0, 0.0, 1.5, 21463.25 }, new double[] { 10.0, -9.0, 2.55,
            157208.4 }
    };

    /**
     * Geocentric position of the Sun. Mean equinox and ecliptic of date. Expansion is
        from "Planetary Programs and Tables",
     * by Pierre Bretagnon and Jean-Louis Simon, Willman-Bell, 1986. The expansion is
        valid from 4000 B.C. to 8000 A.D, but
     * has been modified to reproduce the DE440 integration. The ecliptic latitude is
        not supposed to be 0 as in the previous
     * reference, but computed approximately from the first term of the VSOP solution
        in Jean Meeus's Astronomical Algorithms.
     * Peak errors are below 2".
     * @return Array with ecliptic longitude, latitude (0), distance, and angular
        radius of the Sun.
     */
    @Override
    public double[] getBodyPosition() {
        double t = EarthAngles.toCenturiesRespectJ2000(jd_UT, true) / 100.0;
        double L = 0.0, R = 0.0;

        for (int i = 0; i < sunElements.length; i++) {
            double variable = sunElements[i][2] + sunElements[i][3] * t;
            double u = Util.normalizeRadians(variable);
            double sU = Math.sin(u);
            double cU = Math.cos(u);
            L = L + sunElements[i][0] * sU;
            R = R + sunElements[i][1] * cU;
```

```
61          }
62
63          double tmp = Util.normalizeRadians(62833.196168 * t);
64          L = Util.normalizeRadians(4.9353929 + tmp + L * 1E-7);
65          R = 1.0001026 + R * 1E-7;
66          L -= 0.09 * Constant.ARCSEC_TO_RAD; // Basic transformation from mean
                dynamical equinox to FK5 J2000, as explained by Meeus
67          L += getCorrectionToEclipticLongitude(jd_UT + EarthAngles.TTminusUT1(jd_UT) /
                Constant.SECONDS_PER_DAY);
68
69                // Compute aberration, substracted later
70          double aberration = (993 - 17 * Math.cos(3.10 + 62830.14 * t)) * 1E-7;
71
72          return new double[] { L - aberration, getSolarGeocenticEclipticLatitude(t *
                10), R, Math.atan(695700.0 / (R * Constant.AU)) };
73      }
74
75      /**
76       * Geometric rectangular position of the Sun, mean ecliptic of date.
77       * @param jd Julian day (TT).
78       * @return x, y, z, vx, vy, vz, in AU and AU/d.
79       */
80      public static double[] getGeometricEclipticPositionEquinoxOfDate(double jd) {
81          double t = (jd - Constant.J2000) / 3652500.0;
82          double L = 0.0, R = 0.0, DL = 0.0, DR = 0.0;
83
84          for (int i = 0; i < sunElements.length; i++) {
85              double variable = sunElements[i][2] + sunElements[i][3] * t;
86              double u = Util.normalizeRadians(variable);
87              double sU = Math.sin(u);
88              double cU = Math.cos(u);
89              L = L + sunElements[i][0] * sU;
90              R = R + sunElements[i][1] * cU;
91              DL = DL + sunElements[i][0] * sunElements[i][3] * cU;
92              DR = DR - sunElements[i][1] * sunElements[i][3] * sU;
93          }
94
95          double tmp = Util.normalizeRadians(62833.196168 * t);
96          L = Util.normalizeRadians(4.9353929 + tmp + L * 1E-7);
97          R = 1.0001026 + R * 1E-7;
98          DL = (62833.196168 + DL * 1E-7) / 3652500.0;
99          DR = (DR * 1E-7) / 3652500.0;
100         L -= 0.09 * Constant.ARCSEC_TO_RAD; // Basic transformation from mean
                dynamical equinox to FK5 J2000, as explained by Meeus
101         double b = getSolarGeocenticEclipticLatitude(t * 10);
102         L += getCorrectionToEclipticLongitude(jd);
103
104         // For rectangular coordinates
105         double x = R * Math.cos(L) * Math.cos(b);
106         double y = R * Math.sin(L) * Math.cos(b);
107         double z = R * Math.sin(b);
108         double vx = DR * Math.cos(L) - DL * y;
109         double vy = DR * Math.sin(L) + DL * x;
110         double vz = 0.0;
111
112         return new double[] { x, y, z, vx, vy, vz};
113     }
114
115     private static double getSolarGeocenticEclipticLatitude(double t) {
116         // First 0-order B term from Meeus's Astronomical Algorithms, Appendix II, due
                to the Earth position respect E-M barycenter when the Moon has
117         // a high geocentric ecliptic latitude. Computed with the mean distance of
                moon from its ascending node. At most 0.6"
118         return -280E-8 * Math.cos(3.199 + 84334.662 * t); // t in Julian millenia
119     }
```

```
120
121     private static double getCorrectionToEclipticLongitude(double jd) {
122         return -(9.40 - 0.0000090 * jd + 291E-14 * jd * jd - 321E-21 * jd * jd * jd) *
                Constant.ARCSEC_TO_RAD; // To fit DE441 (Horizons)
123     }
124
125     /**
126      * Returns the dates of the official (geocentric) equinoxes and solstices.
127      * @return Dates of equinoxes and solstices, error always below 50s.
128      */
129     public double[] getEquinoxesAndSolstices() {
130         double jdOld = jd_UT;
131         double[] out = new double[4];
132
133         double prec = 0.1 / Constant.SECONDS_PER_DAY; // Output precision 0.1s,
                accuracy around 1 minute
134         JulianDay julDay = new JulianDay(jd_UT);
135         int year = julDay.year;
136
137         int[] months = new int[] {3, 9, 6, 12};
138         double[] refs = new double[] {0, Math.PI, Constant.PI_OVER_TWO, 3 *
                Constant.PI_OVER_TWO}; // Will search for these RAs
139         for (int i=0; i<4; i++) {
140             julDay = new JulianDay(year, months[i], 18);
141             double jd = julDay.getJulianDay();
142             setUTDate(jd);
143             double min = -1, minT = -1;
144             double stepDays = 0.25, lastRaDif = -1;
145             while (true) {
146                 EphemData data = doCalc(getBodyPosition(), true);
147                 double lon = data.rightAscension; // Using RA, replace with next lines
                        to use ecliptic longitude
148 //double lon = Util.normalizeRadians(CoordinateSystem.cartesianToSpherical
149 //(CoordinateSystem.equatorialToEcliptic(CoordinateSystem.sphericalToCartesian
150 //(data.rightAscension, data.declination), jd))[0]);
151                 double raDif = Math.abs(refs[i] - lon);
152                 if (raDif > Math.PI) raDif = Constant.TWO_PI - raDif;
153                 if (raDif < min || min == -1) {
154                     min = raDif;
155                     minT = jd;
156                 }
157                 if (raDif > lastRaDif && lastRaDif >= 0) {
158                     if (Math.abs(stepDays) < prec) {
159                         out[i] = minT;
160                         break;
161                     }
162                     stepDays = -stepDays / 2;
163                 }
164                 lastRaDif = raDif;
165                 jd += stepDays;
166                 setUTDate(jd);
167             }
168         }
169         setUTDate(jdOld);
170         return out;
171     }
172
173     /**
174      * Test program
175      * @param args Not used
176      */
177     public static void main(String[] args) {
178         // Prepare input data
179         int year = 2020, month = 6, day = 9, h = 18, m = 0, s = 0;
180         JulianDay jday = new JulianDay(year, month, day);
```

```
        jday.setDayFraction((h + m / 60.0 + s / 3600.0) / 24.0);

        double jd_utc = jday.getJulianDay();
        double lon = -4; // degrees
        double lat = 40;
        double alt = 0; // m
        int tz = 3; // h
        TWILIGHT tw = TWILIGHT.HORIZON_34arcmin;
        TWILIGHT_MODE twm = TWILIGHT_MODE.TODAY_UT;

        // Compute the ephemerides data
        EphemSun sunEph = new EphemSun(jd_utc, lon, lat, alt, tw, twm, tz);
        EphemData sunData = EphemReduction.getEphemeris(sunEph);

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

        double[] eqxSols = sunEph.getEquinoxesAndSolstices();
        System.out.println("Sprint equinox: " + EphemData.getDateAsString(eqxSols[0]));
        System.out.println("Autumn equinox: " + EphemData.getDateAsString(eqxSols[1]));
        System.out.println("Summer solstice: " +
            EphemData.getDateAsString(eqxSols[2]));
        System.out.println("Winter solstice: " +
            EphemData.getDateAsString(eqxSols[3]));

/*
Sun
 Az:      285.78955°
 El:      17.4235°
 Dist:    1.0152142 au
 RA:      78.40913°
 DEC:     23.0075°
 Ill:     100.0%
 ang.R:   0.26245394°
 Rise:    2020/06/09 04:46:56 UT
 Set:     2020/06/09 19:43:59 UT
 Transit: 2020/06/09 12:15:21 UT (elev. 72.99474°)

Sprint equinox: 2020/03/20 03:49:45 UT
Autumn equinox: 2020/09/22 13:30:43 UT
Summer solstice: 2020/06/20 21:43:33 UT
Winter solstice: 2020/12/21 10:02:36 UT

Horizons: 2020-Jun-09 18:01:09.200 * 78.40923 23.00749 285.788988 17.372201
    1.01521568029101 69.184687
 */
    }
}
```

Al final del código aparecen los resultados esperados para el ejemplo propuesto en el método *main*, junto con los resultados devueltos por el servidor Horizons. La diferencia es de unos 0.3" en ascensión recta. Recordar que la diferencia TT-UT1 se estableció en nuestros programas en 69.2s, fija durante algunos años, mientras que Horizons utiliza para esta fecha un valor muy similar de 69.18s.

3. La posición precisa de la Luna

El algoritmo propuesto está basado en la obra de Peter Duffet-Smith [4], aunque con modificaciones muy notables para actualizarlo debidamente, y así obtener posiciones consistentes con la última integración numérica del JPL. Este algoritmo utiliza una serie de términos trigonométricos, que conforman una teoría analítica del movimiento de la Luna, que pretende describir interacciones que de otra manera sólo

podrían calcularse mediante la integración numérica de los cuerpos del Sistema Solar. Si es debidamente implementada, una teoría analítica puede describir el movimiento de los astros durante milenios, pero para ello podría requerir de miles de términos trigonométricos para describir las interacciones de menor importancia. En la bibliografía se pueden encontrar otros algoritmos que hemos descartado por ser más laboriosos, como la expansión ELP2000 [5], que es utilizada por Jean Meeus [6] en su libro de manera simplificada. En lugar de utilizar métodos más elaborados, aquí se propone la implementación de [4], pero notablemente modificada para, de algún modo, ajustarla a la integración DE440, dentro de un margen de pocos segundos de arco, durante bastantes milenios. De esta manera obtenemos posiciones más precisas en el pasado que con los otros algoritmos, que están desarrollados para reproducir los resultados de integraciones más antiguas en un arco de tiempo mucho más limitado. Esto también es aplicable a los algoritmos *Standards of Fundamental Astronomy*, o SOFA [7]. El programa sugerido en SOFA para la posición de la Luna no está modificado para mantener la precisión en tiempos remotos, mientras que el del Sol, aunque es más preciso durante 1000 años alrededor de la época actual, también es mucho más extenso. Las modificaciones que veremos no aumentan significativamente la longitud del código.

Para esta adaptación, los valores de diferentes parámetros del Sol y la Luna de la teoría original han sido reemplazados por los valores calculados por S. L. Moshier [8], cuyas expansiones para obtener los valores medios de la longitud o latitud eclípticas de la Luna, entre otras, son válidas durante milenios, y fueron obtenidas de forma consistente con la integración DE405, bastante más evolucionada que la DE200, y relativamente cercana y consistente con DE440. Con eso mejoramos ya la teoría original cuando nos movemos a fechas lejanas. Pero también es necesario introducir términos trigonométricos adicionales en cada coordenada (ver líneas 106-117, 139-141, 176-180), para lograr afinar los resultados, como se describirá con más detalle en la sección siguiente. Estos términos incluyen una corrección secular o parabólica, debido a una estimación imprecisa de la aceleración secular de la Luna en las teorías implementadas décadas atrás, y unos términos trigonométricos adicionales, que logran ajustar los resultados a las integraciones numéricas del JPL. Las correcciones respecto de la integración DE431 fueron originalmente presentadas en un blog del autor algunos años atrás, y se han actualizado aquí para el DE440. Hasta donde sabemos, ni éste ni ningún otro código parecido ha sido nunca publicado en una revista.

El código se completa con métodos para obtener la edad de la Luna, a partir de la diferencia entre las longitudes eclípticas del Sol y la Luna, y las fases lunares, en donde se recurre de nuevo a un algoritmo de fuerza bruta. La precisión en las fases lunares es mejor que el minuto también, dado que, aunque la posición de la Luna pueda alcanzar picos de error 10 veces mayores que en el caso del Sol, se compensa porque la Luna se mueve en el cielo 10 veces más rápido. El repositorio [1] incluye un método adicional que permite obtener las libraciones y el resto de ángulos que definen la orientación del disco lunar para una fecha dada. Estos valores son útiles para dibujar el aspecto aparente del disco para un observador. En el futuro es posible que veamos cómo hacer esto, tanto para la Luna como para el resto de cuerpos.

Una vez más, el método *main* presente al final del listado contiene un ejemplo con la salida que cabe esperar de la ejecución del programa, para los datos de entrada que aparecen.

```
1  package journal;
2
3  /**
4   * A class to compute the ephemerides of the Moon. This class uses the code inside
        {@linkplain EphemReduction} by inheritance.
5   */
6  public class EphemMoon extends EphemReduction {
7
8      /** The set of phases to compute the moon phases */
9      public enum MOONPHASE {
10         /** New Moon phase */
11         NEW_MOON ("New Moon:     ", 0),
```

```
12          /** Crescent quarter phase */
13          CRESCENT_QUARTER ("Crescent quarter:", 0.25),
14          /** Full Moon phase */
15          FULL_MOON ("Full Moon:   ", 0.5),
16          /** Descent quarter phase */
17          DESCENT_QUARTER ("Descent quarter: ", 0.75);
18
19          /** Phase name */
20          public String phaseName;
21          /** Phase value */
22          public double phase;
23
24          private MOONPHASE(String name, double ph) {
25              phaseName = name;
26              phase = ph;
27          }
28      }
29
30      public EphemMoon(double jd_utc, double lon, double lat, double alt, TWILIGHT tw,
            TWILIGHT_MODE twm, int tz) {
31          super(jd_utc, lon, lat, alt, tw, twm, tz);
32      }
33
34      /**
35       * Geocentric position of the Moon. Mean equinox and ecliptic of date.
36       * Based on the Petter Duffet program (and mean elements from S. L. Moshier),
37       * modified to fit DE441 during millenia.
38       * @return Array with ecliptic longitude, latitude (0), distance, and angular
             radius of the Sun
39       */
40      @Override
41      protected double[] getBodyPosition() {
42          double t = EarthAngles.toCenturiesRespectJ2000(jd_UT, true);
43
44          // These expansions up to t^7 for the mean elements are taken from S. L.
                Moshier
45          /* Mean elongation of moon = D */
46          double x = (1.6029616009939659e+09 * t + 1.0722612202445078e+06);
47          x += (((((-3.207663637426e-013 * t + 2.555243317839e-011) * t +
                2.560078201452e-009) * t - 3.702060118571e-005) * t +
                6.9492746836058421e-03) * t /* D, t^3 */
48                      - 6.7352202374457519e+00) * t * t; /* D, t^2 */
49          double phase = Util.normalizeRadians(Constant.ARCSEC_TO_RAD * x);
50
51          /* Mean distance of moon from its ascending node = F */
52          x = (1.7395272628437717e+09 * t + 3.3577951412884740e+05);
53          x += (((((4.474984866301e-013 * t + 4.189032191814e-011) * t -
                2.790392351314e-009) * t - 2.165750777942e-006) * t -
                7.5311878482337989e-04) * t /* F, t^3 */
54                      - 1.3117809789650071e+01) * t * t; /* F, t^2 */
55          double node = Util.normalizeRadians(Constant.ARCSEC_TO_RAD * x);
56
57          /* Mean anomaly of sun = l' (J. Laskar) */
58          x = (1.2959658102304320e+08 * t + 1.2871027407441526e+06);
59          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;
60          double sanomaly = Util.normalizeRadians(Constant.ARCSEC_TO_RAD * x);
61
62          /* Mean anomaly of moon = l */
63          x = (1.7179159228846793e+09 * t + 4.8586817465825332e+05);
64          x += (((((-1.755312760154e-012 * t + 3.452144225877e-011) * t -
                2.506365935364e-008) * t - 2.536291235258e-004) * t +
                5.2099641302735818e-02) * t /* l, t^3 */
65                      + 3.1501359071894147e+01) * t * t; /* l, t^2 */
```

```
66          double anomaly = Util.normalizeRadians(Constant.ARCSEC_TO_RAD * x);
67
68          /* Mean longitude of moon, re mean ecliptic and equinox of date = L */
69          x = (1.7325643720442266e+09 * t + 7.859398092105242 0e+05);
70          x += (((((7.200592540556e-014 * t + 2.235210987108e-010) * t -
                1.024222633731e-008) * t - 6.073960534117e-005) * t +
                6.9017248528380490e-03) * t /* L, t^3 */
71                      - 5.6550460027471399e+00) * t * t; /* L, t^2 */
72          double l = Util.normalizeRadians(Constant.ARCSEC_TO_RAD * x) *
                Constant.RAD_TO_DEG;
73
74          // Now longitude, with the three main correcting terms of evection,
75          // variation, and equation of year, plus other terms (error<0.01 deg)
76          // P. Duffet's Moon program taken as reference for the periodic terms
77          double E = 1.0 - (.002495 + 7.52E-06 * (t + 1.0)) * (t + 1.0), E2 = E * E;
78          double td = t + 1, td2 = t * t;
79          double M6 = td * Constant.JULIAN_DAYS_PER_CENTURY * 360.0 / 6.798363307E3;
80          double NA = Util.normalizeRadians((2.59183275E2 - M6 + (2.078E-3 + 2.2E-6 *
                td) * td2) * Constant.DEG_TO_RAD);
81          double C = NA + Constant.DEG_TO_RAD * (275.05 - 2.3 * td);
82
83          l += 6.28875 * Math.sin(anomaly) + 1.274018 * Math.sin(2 * phase - anomaly) +
                .658309 * Math.sin(2 * phase);
84          l += 0.213616 * Math.sin(2 * anomaly) - E * .185596 * Math.sin(sanomaly) -
                0.114336 * Math.sin(2 * node);
85          l += .058793 * Math.sin(2 * phase - 2 * anomaly) + .057212 * E * Math.sin(2 *
                phase - anomaly - sanomaly) + .05332 * Math.sin(2 * phase + anomaly);
86          l += .045874 * E * Math.sin(2 * phase - sanomaly) + .041024 * E *
                Math.sin(anomaly - sanomaly) - .034718 * Math.sin(phase) - E * .030465 *
                Math.sin(sanomaly + anomaly);
87          l += .015326 * Math.sin(2 * (phase - node)) - .012528 * Math.sin(2 * node +
                anomaly) - .01098 * Math.sin(2 * node - anomaly) + .010674 * Math.sin(4 *
                phase - anomaly);
88          l += .010034 * Math.sin(3 * anomaly) + .008548 * Math.sin(4 * phase - 2 *
                anomaly);
89          l += -E * .00791 * Math.sin(sanomaly - anomaly + 2 * phase) - E * .006783 *
                Math.sin(2 * phase + sanomaly) + .005162 * Math.sin(anomaly - phase) + E *
                .005 * Math.sin(sanomaly + phase);
90          l += .003862 * Math.sin(4 * phase) + E * .004049 * Math.sin(anomaly - sanomaly
                + 2 * phase) + .003996 * Math.sin(2 * (anomaly + phase)) + .003665 *
                Math.sin(2 * phase - 3 * anomaly);
91          l += E * 2.695E-3 * Math.sin(2 * anomaly - sanomaly) + 2.602E-3 *
                Math.sin(anomaly - 2*(node+phase));
92          l += E * 2.396E-3 * Math.sin(2*(phase - anomaly) - sanomaly) - 2.349E-3 *
                Math.sin(anomaly+phase);
93          l += E * E * 2.249E-3 * Math.sin(2*(phase-sanomaly)) - E * 2.125E-3 *
                Math.sin(2*anomaly+sanomaly);
94          l += -E * E * 2.079E-3 * Math.sin(2*sanomaly) + E * E * 2.059E-3 *
                Math.sin(2*(phase-sanomaly)-anomaly);
95          l += -1.773E-3 * Math.sin(anomaly+2*(phase-node)) - 1.595E-3 *
                Math.sin(2*(node+phase));
96          l += E * 1.22E-3 * Math.sin(4*phase-sanomaly-anomaly) - 1.11E-3 *
                Math.sin(2*(anomaly+node));
97          l += 8.92E-4 * Math.sin(anomaly - 3 * phase) - E * 8.11E-4 * Math.sin(sanomaly
                + anomaly + 2 * phase);
98          l += E * 7.61E-4 * Math.sin(4 * phase - sanomaly - 2 * anomaly);
99          l += E2 * 7.04E-4 * Math.sin(anomaly - 2 * (sanomaly + phase));
100         l += E * 6.93E-4 * Math.sin(sanomaly - 2 * (anomaly - phase));
101         l += E * 5.98E-4 * Math.sin(2 * (phase - node) - sanomaly);
102         l += 5.5E-4 * Math.sin(anomaly + 4 * phase) + 5.38E-4 * Math.sin(4 * anomaly);
103         l += E * 5.21E-4 * Math.sin(4 * phase - sanomaly) + 4.86E-4 * Math.sin(2 *
                anomaly - phase);
104         l += E2 * 7.17E-4 * Math.sin(anomaly - 2 * sanomaly);
105
106         // Ecliptic longitude: additional terms to fit DE404
```

```
107         l += 14.1983 * Math.cos(2.3232 * t + 0.4964) / 3600.0;
108         double ph = 0.5582 + 0.11 * t;
109         l += 7.2167 * Math.cos(33.8624 * t - ph) / 3600.0;
110         // Additional terms to fit DE431
111         l += -0.0759777 * Math.pow(t - 1.541336, 2.0) / 3600.0;
112         // Previous term can be replaced by this one for a better fit between years
                -2000 to 6000 (only for DE431)
113         //l += (-0.0001617 * t * t * t - 0.075925 * t * t + 0.3932 * t - 0.4375) /
                3600.0;
114         l += -0.01 * t * t * Math.cos(anomaly) / 3600.0;
115         // Fit DE440
116         l -= (0.88 - 0.156 * t - 0.00584 * t * t - 0.003128 * t * t * t) / 3600.0;
117         l += 0.25 * t * Math.cos(anomaly) / 3600.0;
118
119         double longitude = l * Constant.DEG_TO_RAD;
120
121         // Now Moon parallax
122         double p = .950724 + .051818 * Math.cos(anomaly) + .009531 * Math.cos(2 *
                phase - anomaly);
123         p += .007843 * Math.cos(2 * phase) + .002824 * Math.cos(2 * anomaly);
124         p += 0.000857 * Math.cos(2 * phase + anomaly) + E * .000533 * Math.cos(2 *
                phase - sanomaly);
125         p += E * .000401 * Math.cos(2 * phase - anomaly - sanomaly) + E * .00032 *
                Math.cos(anomaly - sanomaly) - .000271 * Math.cos(phase);
126         p += -E * .000264 * Math.cos(sanomaly + anomaly) - .000198 * Math.cos(2 * node
                - anomaly);
127         p += 1.73E-4 * Math.cos(3 * anomaly) + 1.67E-4 * Math.cos(4*phase-anomaly);
128         p += -E * 1.11E-4 * Math.cos(sanomaly) + 1.03E-4 * Math.cos(4 * phase - 2 *
                anomaly);
129         p += -8.4E-5 * Math.cos(2 * anomaly - 2 * phase) - E * 8.3E-5 * Math.cos(2 *
                phase + sanomaly);
130         p += 7.9E-5 * Math.cos(2 * phase + 2 * anomaly) + 7.2E-5 * Math.cos(4 * phase);
131         p += E * 6.4E-5 * Math.cos(2 * phase - sanomaly + anomaly)
132                     - E * 6.3E-5 * Math.cos(2 * phase + sanomaly - anomaly);
133         p += E * 4.1E-5 * Math.cos(sanomaly + phase) + E * 3.5E-5 * Math.cos(2 *
                anomaly - sanomaly);
134         p += -3.3E-5 * Math.cos(3 * anomaly - 2 * phase) - 3E-5 * Math.cos(anomaly +
                phase);
135         p += -2.9E-5 * Math.cos(2 * (node - phase)) - E * 2.9E-5 * Math.cos(2 *
                anomaly + sanomaly);
136         p += E2 * 2.6E-5 * Math.cos(2 * (phase - sanomaly)) - 2.3E-5 * Math.cos(2 *
                (node - phase) + anomaly);
137         p += E * 1.9E-5 * Math.cos(4 * phase - sanomaly - anomaly);
138
139               // Parallax: additional terms to fit DE431+
140         double pc = (20 - 1799 * t) * Constant.DEG_TO_RAD;
141         p += 0.0000925 * Math.cos(pc);
142
143         // So Moon distance in Earth radii is, more or less,
144         double distance = 1.0 / Math.sin(p * Constant.DEG_TO_RAD);
145
146         // Ecliptic latitude with nodal phase (error<0.01 deg)
147         l = 5.128189 * Math.sin(node) + 0.280606 * Math.sin(node + anomaly) + 0.277693
                * Math.sin(anomaly - node);
148         l += .173238 * Math.sin(2 * phase - node) + .055413 * Math.sin(2 * phase +
                node - anomaly);
149         l += .046272 * Math.sin(2 * phase - node - anomaly) + .032573 * Math.sin(2 *
                phase + node);
150         l += .017198 * Math.sin(2 * anomaly + node) + .009267 * Math.sin(2 * phase +
                anomaly - node);
151         l += .008823 * Math.sin(2 * anomaly - node) + E * .008247 * Math.sin(2 * phase
                - sanomaly - node) + .004323 * Math.sin(2 * (phase - anomaly) - node);
152         l += .0042 * Math.sin(2 * phase + node + anomaly) + E * .003372 *
                Math.sin(node - sanomaly - 2 * phase);
153         l += E * 2.472E-3 * Math.sin(2 * phase + node - sanomaly - anomaly);
```

```
154         l += E * 2.222E-3 * Math.sin(2 * phase + node - sanomaly);
155         l += E * 2.072E-3 * Math.sin(2 * phase - node - sanomaly - anomaly);
156         l += E * 1.877E-3 * Math.sin(node - sanomaly + anomaly) + 1.828E-3 *
                Math.sin(4 * phase - node - anomaly);
157         l += -E * 1.803E-3 * Math.sin(node + sanomaly) - 1.75E-3 * Math.sin(3 * node);
158         l += E * 1.57E-3 * Math.sin(anomaly - sanomaly - node) - 1.487E-3 *
                Math.sin(node + phase);
159         l += -E * 1.481E-3 * Math.sin(node + sanomaly + anomaly) + E * 1.417E-3 *
                Math.sin(node - sanomaly - anomaly);
160         l += E * 1.35E-3 * Math.sin(node - sanomaly) + 1.33E-3 * Math.sin(node -
                phase);
161         l += 1.106E-3 * Math.sin(node + 3 * anomaly) + 1.02E-3 * Math.sin(4 * phase -
                node);
162         l += 8.33E-4 * Math.sin(node + 4 * phase - anomaly) + 7.81E-4 *
                Math.sin(anomaly - 3 * node);
163         l += 6.7E-4 * Math.sin(node + 4 * phase - 2 * anomaly) + 6.06E-4 * Math.sin(2
                * phase - 3 * node);
164         l += 5.97E-4 * Math.sin(2 * (phase + anomaly) - node);
165         l += E * 4.92E-4 * Math.sin(2 * phase + anomaly - sanomaly - node)
166                         + 4.5E-4 * Math.sin(2 * (anomaly - phase) - node);
167         l += 4.39E-4 * Math.sin(3 * anomaly - node) + 4.23E-4 * Math.sin(node + 2 *
                (phase + anomaly));
168         l += 4.22E-4 * Math.sin(2 * phase - node - 3 * anomaly)
169                         - E * 3.67E-4 * Math.sin(sanomaly + node + 2 * phase - anomaly);
170         l += -E * 3.53E-4 * Math.sin(sanomaly + node + 2 * phase) + 3.31E-4 *
                Math.sin(node + 4 * phase);
171         l += E * 3.17E-4 * Math.sin(2 * phase + node - sanomaly + anomaly);
172         l += E2 * 3.06E-4 * Math.sin(2 * (phase - sanomaly) - node) - 2.83E-4 *
                Math.sin(anomaly + 3 * node);
173         double W1 = 4.664E-4 * Math.cos(NA);
174         double W2 = 7.54E-5 * Math.cos(C);
175
176         // Ecliptic latitude: additional terms to fit DE431
177         double M1 = (124.90 - 1934.134 * t + 0.002063 * t * t) * Constant.DEG_TO_RAD;
178         l += -8 * Math.sin(M1) * Math.cos(node) / 3600.0; // For DE431
179         l += -0.007 * t * t * Math.cos(node) / 3600.0; // For DE431
180         l += 0.48 * t * Math.cos(node) / 3600.0; // For DE440
181
182         double latitude = l * Constant.DEG_TO_RAD * (1.0 - W1 - W2);
183
184         double earthRadius = 6378.1366;
185         double moonRadius = 1737.4;
186         return new double[] {longitude, latitude, distance * earthRadius / Constant.AU,
187                 Math.atan(moonRadius / (distance * earthRadius)), phase};
188     }
189
190     /**
191      * Returns the moon age
192      * @param sun Ephemerides of the Sun, or null for a moon age with less precision
193      * @return Age in days
194      */
195     public double getMoonAge(EphemData sun) {
196         double[] moon = getBodyPosition();
197         double Psin = 29.530588853;
198         if (sun != null) {
199             // Get Moon age, more accurate than 'phase' but we need the Sun position
200             return Util.normalizeRadians(moon[0] - sun.eclipticLongitude) * Psin /
                    Constant.TWO_PI;
201         } else {
202             // Use the phase variable as estimate, less accurate but this is used only
                    when we don't need an accurate value
203             return moon[4] * Psin / Constant.TWO_PI;
204         }
205     }
206
```

```
207         /**
208      * Returns the instant of a given moon phase.
209      * @param phase The phase.
210      * @return The instant of that phase, accuracy around 1 minute or better.
211      */
212     public double getMoonPhaseTime(MOONPHASE phase) {
213         double accuracy = 10 / (30 * Constant.SECONDS_PER_DAY); // 10s / lunar cycle
                length in s => 10s accuracy
214         double refPhase = phase.phase;
215         double oldJD = jd_UT;
216         EphemSun sunEph = new EphemSun(jd_UT, obsLon * Constant.RAD_TO_DEG, obsLat *
                Constant.RAD_TO_DEG, obsAlt,
217                 twilight, twilightMode, timeZone);
218         while (true) {
219             double[] sun = sunEph.getBodyPosition();
220             double[] moon = getBodyPosition();
221             double age = Util.normalizeRadians((moon[0] - sun[0])) / Constant.TWO_PI -
                    refPhase;
222             if (age < 0) age += 1;
223             if (age < accuracy || age > 1 - accuracy) break;
224             if (age < 0.5) {
225                 jd_UT -= age;
226             } else {
227                 jd_UT += 1-age;
228             }
229             sunEph.setUTDate(jd_UT);
230             setUTDate(jd_UT);
231         }
232         double out = jd_UT;
233         setUTDate(oldJD);
234         return out;
235     }
236
237     /**
238      * Test program
239      * @param args Not used
240      */
241     public static void main(String[] args) {
242         // Prepare input data
243         int year = 2020, month = 6, day = 9, h = 18, m = 0, s = 0;
244         JulianDay jday = new JulianDay(year, month, day);
245         jday.setDayFraction((h + m / 60.0 + s / 3600.0) / 24.0);
246
247         double jd_utc = jday.getJulianDay();
248         double lon = -4; // degrees
249         double lat = 40;
250         double alt = 0; // m
251         int tz = 3; // h
252         TWILIGHT tw = TWILIGHT.HORIZON_34arcmin;
253         TWILIGHT_MODE twm = TWILIGHT_MODE.TODAY_UT;
254
255         // Compute the ephemerides data
256         EphemMoon moonEph = new EphemMoon(jd_utc, lon, lat, alt, tw, twm, tz);
257         EphemData moonData = EphemReduction.getEphemeris(moonEph);
258
259         EphemSun sunEph = new EphemSun(jd_utc, lon, lat, alt, tw, twm, tz);
260         EphemData sunData = EphemReduction.getEphemeris(sunEph);
261         moonData.setIlluminationPhase(sunData);
262
263         // Report
264         System.out.println("Moon");
265         System.out.println(moonData.toString());
266         System.out.println(" Moon age: " + (float) moonEph.getMoonAge(sunData) + "
                days");
267         System.out.println();
```

```
268
269         System.out.println("Closest Moon phases:");
270         System.out.println(" New moon:    " +
                EphemData.getDateAsString(moonEph.getMoonPhaseTime(MOONPHASE.NEW_MOON)));
271         System.out.println(" Crescent quarter: " +
                EphemData.getDateAsString(moonEph.getMoonPhaseTime(MOONPHASE.CRESCENT_QUARTER)));
272         System.out.println(" Full moon:   " +
                EphemData.getDateAsString(moonEph.getMoonPhaseTime(MOONPHASE.FULL_MOON)));
273         System.out.println(" Descent quarter: " +
                EphemData.getDateAsString(moonEph.getMoonPhaseTime(MOONPHASE.DESCENT_QUARTER)));
274 /*
275 Moon
276  Az:      64.830414°
277  El:      -57.52784°
278  Dist:    0.0026317919 au
279  RA:      313.06357°
280  DEC:     -21.55362°
281  Ill:     82.01859%
282  ang.R:   0.25632995°
283  Rise:    2020/06/09 23:17:41 UT
284  Set:     2020/06/09 08:10:47 UT
285  Transit: 2020/06/09 03:21:08 UT (elev. 26.579206°)
286
287  Moon age: 18.860058 days
288
289 Closest Moon phases:
290  New moon:       2020/06/21 06:41:23 UT
291  Crescent quarter: 2020/05/30 03:30:03 UT
292  Full moon:      2020/06/05 19:12:33 UT
293  Descent quarter: 2020/06/13 06:23:07 UT
294  */
295     }
296 }
```

4. Comparación con la integración numérica DE440 (Horizons)

La Fig. 1 muestra las discrepancias entre la posición del Sol devuelta por el programa y la proporcionada por el servidor Horizons. Se han considerado tres mil puntos espaciados mil días entre sí, aproximadamente entre los años -2100 al 6100. A la izquierda, las posiciones corresponden al algoritmo original, sin hacer correcciones en la longitud, y manteniendo la latitud en el valor cero. A la derecha aparece el resultado de aplicar las correcciones propuestas en ambas coordenadas, con lo que la discrepancia se mantiene por debajo de los 2" durante al menos cuatro milenios alrededor del año 2000. El valor medio del error en longitud se mantiene en torno a 0.7", mientras que en latitud desaparecen los picos superiores a 1", y los errores pasan a valores habituales de 0.6" o menos. Estas correcciones propuestas representan la diferencia en la Tierra entre ajustar las efemérides DE200 del JPL, ya muy obsoletas, y las DE440. El algoritmo propuesto puede seguir utilizándose para fechas fuera del rango mostrado. Por ejemplo, para el año 4000 a.C. o el 8000 d.C. las discrepancias pueden superar los 3", pero sólo en casos excepcionales.

El mismo procedimiento se muestra para la Luna en la Fig. 2. En el código aparecen las correcciones necesarias para cada coordenada y para diferentes integraciones del JPL, como la DE404, DE431, y DE440, de manera incremental. Los errores se reducen drásticamente al aplicar las correcciones necesarias para cada una, y con todas ellas los picos de discrepancia respecto de la DE440 se reducen por debajo de los 10" (equivalente a 20 km), salvo para más de dos milenios de distancia respecto del año 2000. Se trata de un resultado excelente para un algoritmo compacto, lo que nos permitirá obtener resultados precisos incluso para cálculos de eclipses en épocas remotas. Los residuos aún muestran cierta periodicidad en épocas remotas, lo que indica que con un poco más de trabajo los residuos podrían

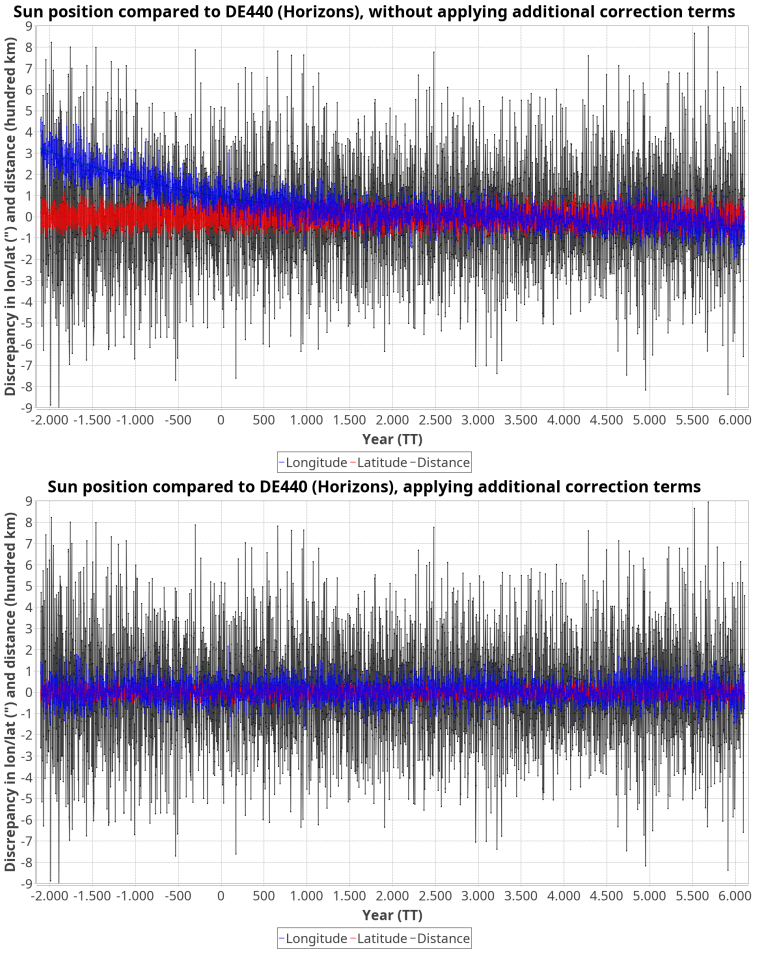

Figura 1. *Diferencia entre las posiciones del Sol calculadas con el programa propuesto y el servidor Horizons. Arriba: sin incluir las correcciones adicionales propuestas. Abajo: tras incluir las correcciones propuestas para la longitud y latitud eclípticas.*

reducirse a la mitad en los extremos del gráfico, pero el resultado ya es muy bueno. Para el año 3000 a.C., o el 7000 d.C., las discrepancias pueden superar los 25", y 1000 años más lejos los 50", especialmente en latitud eclíptica.

References

[1] Repositorio de código en GitHub: https://github.com/JCAAC-FAAE
[2] Planetary Programs and Tables, Pierre Bretagnon and Jean-Louis Simon, Willman-Bell, 1986.
[3] Astronomical computing: Cálculo de efemérides, T. Alonso Albi, JCAAC **3**, 65 (2025).
[4] Astronomy with your Personal Computer, Peter Duffet-Smith, Cambridge University Press, 1985.
[5] ELP 2000-85: a semi-analytical lunar ephemeris adequate for historical times, M. Chapront-Touze y J. Chapront, 1988. https://articles.adsabs.harvard.edu/pdf/1988A%26A...190..342C
[6] Astronomical Algorithms, Jean Meeus, editorial Atlantic Books.

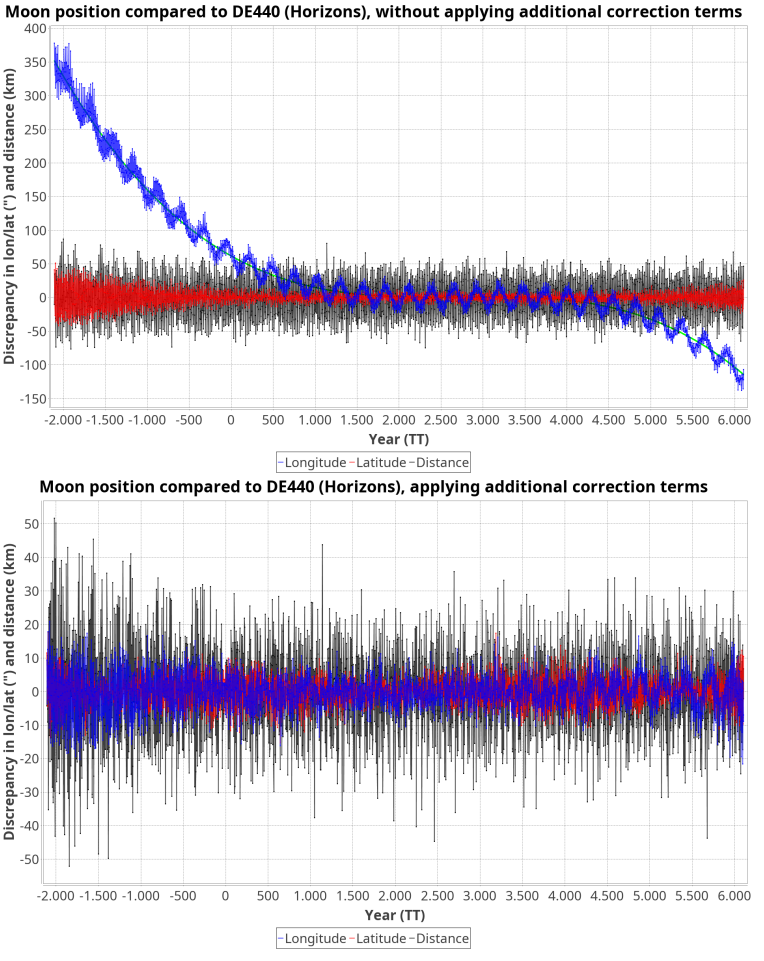

Figura 2. *Diferencia entre las posiciones de la Luna calculadas con el programa propuesto y el servidor Horizons. Arriba: sin incluir las correcciones adicionales propuestas. Abajo: tras incluir las correcciones propuestas para la distancia, y la longitud y latitud eclípticas.*

[7] Algoritmos de SOFA para el cálculo de la posición de la Luna:
http://iausofa.org/2021_0512_C/sofa/moon98.c
y para el Sol:
http://www.iausofa.org/2023_1011_F/sofa/epv00.for
[8] Algoritmos de S. L. Moshier: https://www.moshier.net/