JCAAC (2025), **2**, 87–103

SECTION: ASTRONOMICAL COMPUTING

Cálculo de ángulos de orientación de la Tierra y transformaciones de coordenadas

Tomás Alonso Albi[1]

[1]Starion/European Space Agency, 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 esta entrega de la sección discutimos el cálculo de la diferencia TT−UT1, que resulta básico para cualquier cálculo de fenómenos, ya que relaciona el tiempo en el que se expresan las efemérides planetarias con el tiempo asociado a la rotación de la Tierra y, por tanto, a la posición de un observador sobre la misma. El Tiempo Sidéreo Aparente, que es esencialmente el ángulo de rotación terrestre, se relaciona de manera directa con UT1 y con la ecuación de los equinoccios. Además, se discute el cálculo de los parámetros de precesión y nutación, y de la oblicuidad de la eclíptica, necesarios para diversas y necesarias transformaciones entre sistemas de coordenadas astronómicas, que se tratan al final del artículo.

Abstract

In this article we discuss the calculation of the TT−UT1 difference, which is fundamental for any calculation of astronomical phenomena, as it relates the time in which planetary ephemerides are expressed to the time associated with the Earth's rotation and, therefore, to the position of an observer on its surface. Apparent Sidereal Time, which is essentially the Earth's rotation angle, is directly related to UT1 and the equation of the equinoxes. Additionally, the calculation of precession and nutation parameters, as well as the obliquity of the ecliptic, is discussed, as these are necessary for various essential transformations between astronomical coordinate systems, which are addressed at the end of the article.

1. Introducción

En el número anterior [1] vimos el cálculo del día juliano, un primer paso básico para la inmensa mayoría de los cálculos astronómicos, y vimos que en general utilizaríamos el día juliano en Tiempo Universal (UT). En esta segunda entrega seguiremos con el mismo criterio de representar el día juliano en UT, y avanzaremos hacia el objetivo de obtener las efemérides con el cálculo de diferentes parámetros que están ligados a la rotación y orientación del eje de la Tierra:

- Diferencia entre el Tiempo Terrestre (TT) menos el UT. Es otro parámetro esencial para el cálculo de efemérides, dado que éstas se calculan a partir del TT. Este valor está asociado a pequeñas variaciones en el ritmo de rotación de la Tierra, y no es posible prever su evolución.
- Ángulos de precesión, nutación, y oblicuidad, que definen la orientación del eje terrestre en el espacio. Como sabemos el eje terrestre no es estable en el tiempo, pues la Tierra gira como una peonza a punto a detenerse, con un periodo de casi 26000 años. Además, la Luna y los planetas generan otro movimiento menor de nutación. Esto afecta a las coordenadas aparentes de los objetos.

- Tiempo sidéreo local, que puede definirse con la coordenada de ascensión recta o longitud celeste que en un instante dado esta cruzando el meridiano local del observador, lo cual depende del instante y la longitud geográfica del observador.

El primero de los códigos presentado en esta ocasión puede también encontrarse en el repositorio de GitHub [2] correspondiente a esta sección, cuyo uso (y posible adaptación a otros lenguajes) ayudará a evitar los frecuentes despistes y errores de codificación del pasado.

Posteriormente, en esta misma, entrega utilizaremos estos ángulos para hacer todo tipo de transformaciones entre coordenadas ecuatoriales (ascensión recta y declinación), eclípticas (longitud y latitud eclípticas), u horizontales (acimut y elevación), además de corregir las coordenadas por precesión y nutación durante el proceso posterior que veremos para el cálculo de efemérides. El correspondiente código también se puede encontrar en el repositorio.

2. Cálculo de los ángulos de orientación de la Tierra

2.1. Cálculo de la diferencia TT−UT

En el primero de los listados que se muestra al final del artćiulo, este cálculo tiene lugar en la subrutina contenida entre las líneas 14 y 40. Inicialmente se hace uso del código de la entrega anterior para operar con el día juliano y obtener el año, mes, y día. Luego se utilizan diferentes polinomios para efectuar el cálculo en diferentes intervalos de años: un ajuste parabólico que sigue el trabajo de Stephenson & Morrison [3] para años anteriores al -500, y posteriores al 2200, y otros ajustes para los intervalos -500 al 1600, y 1600 al 2200. Los valores entre los años 2020 y 2200 son extrapolaciones tentativas y con el tiempo habrá que refinarlas. De hecho, como se puede comprobar al final del listado el cálculo propuesto devuelve casi 71 segundos, cuando en otras referencias de Intenet (como las páginas web mostradas al final del código) es posible comprobar que el valor correcto ha dejado de crecer con la tendencia que seguía en años anteriores (el ajuste utilizado se hizo hace un par de años), manteniéndose por debajo de 70 segundos. Algunos trabajos de investigación sugieren que el motivo es la desaparición acelerada de los glaciares por el cambio climático, que está compensando, al menos temporalmente, la tendencia observada por Stephenson y Morrison de que el día solar medio se alarga unos 2 milisegundos por siglo, por el efecto de la influencia de la Luna.

En otros programas esta corrección se hace con más precisión mediante interpolación en tablas como la disponible en la página web de USNO [4], o del IERS (Servicio Internacional de Rotación de la Tierra y Sistemas de Referencia) [5], pero por razones de espacio aquí proporcionamos un ajuste polinómico aproximado, válido para muchos siglos. En todo caso, el efecto de este error de pocos segundos en las coordenadas de los objetos es mínimo.

En el listado se observa una corrección en las líneas 37 y 38, que dependen de valores calculados en otras líneas. Esta corrección sólo se aplica al cálculo de Morrison para épocas remotas en el pasado o futuro. En dicho trabajo se utilizó la posición de la Luna para estimar la diferencia TT−UT, en base a registros de eclipses pasados, pero se utilizó una teoría antigua para estimar la posición de la Luna, que ahora sabemos que tiene un cierto error. Para corregir este efecto se utiliza un valor más moderno de la aceleración secular de la Luna en longitud eclíptica: -25.858 segundos de arco por siglo al cuadrado, en lugar del valor antiguo de -26.

En el servidor de efemérides del JPL [6] el cálculo de este parámetro no sigue exactamente el método de Stephenson y Morrison, sino que el valor se deriva de la integración numérica de las últimas efemérides del JPL. El siguiente código puede utilizarse en lugar del sugerido en el caso de que el lector prefiera obtener un valor más parecido al de Horizons:

```
1 TTminusUT = 32.498952 * Math.pow((jd - 2377032.2) / 36525.0, 2.0) - 120;
2 double t = (jd - J2000) / 36525.0;
```

```
3 if (year > -720 && year < 1320) TTminusUT -= (((((((((2.6208E-8 * t + 3.12608E-6) * t
      + 1.29912E-4) * t + 1.04337E-3) * t - 0.0939895) * t - 3.74281165) * t -
      64.8913308) * t - 598.32335) * t - 2832.8992) * t - 5500.079525) * t - 715.276688;
4 if (year > 1320 && year < 2020) TTminusUT -= (((((((((-0.0144076 * t - 0.459961) * t
      - 6.1225996) * t - 43.893463) * t - 182.345180) * t - 437.489625) * t -
      561.221579) * t - 301.145054) * t + 14.885739) * t + 91.5968394) * t - 45.3217576;
5 if (year >= 2020) TTminusUT = 69.185;
```

En todo caso, el efecto en las efemérides es muy limitado, y como puede comprobarse en la Fig. 1, el algoritmo propuesto arriba aún muestra una diferencia de algunos segundos en torno al año 2000 y en siglos anteriores, diferencia que se incrementa notablemente en épocas remotas.

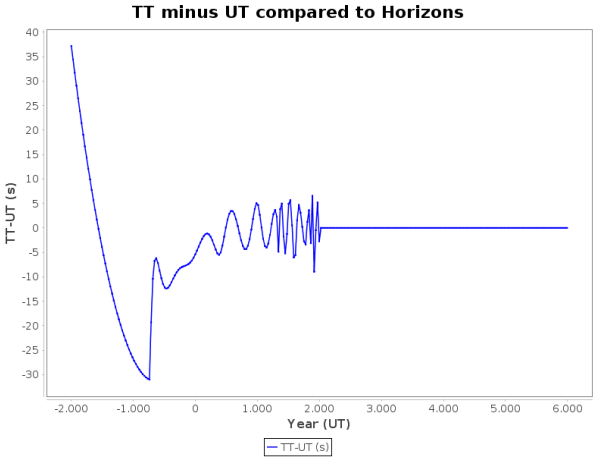

Figura 1. *Diferencia entre los valores de TT−UT obtenidos por el código alternativo superior y el servidor Horizons.*

2.2. Cálculo de los ángulos de nutación

La nutación es un pequeño bamboleo del eje terrestre con un periodo principal de 18.6 años, provocado por la influencia gravitatoria del Sol, la Luna y los planetas. El código de la subrutina contenida entre las líneas 60 y 75 es muy sencillo en comparación con códigos más evolucionados que pueden alcanzar precisiones mejores que el milisegundo de arco, como los algoritmos aprobados por las últimas resoluciones de la Unión Astronómica Internacional (UAI), pero para el propósito que tenemos aquí en cuanto a requerimientos de precisión, y por motivos de espacio, el algoritmo propuesto es suficiente, pues el error que produce es a lo sumo de pocas décimas de segundo de arco. El código sigue la obra de Jean Meeus [7].

En el cálculo se utiliza un método previo que permite obtener el lapso de tiempo en siglos julianos. También se hace uso de constantes que aparecen en la clase `Constant`, que puede consultarse en el repositorio. El nombre de las constantes, si bien se indica en inglés, debería clarificar la operación. Aparecen constantes como SECONDS_PER_DAY, que obviamente es 86400, que quizá el lector no esté habituado a utilizar, sustituyéndolas directamente por su valor. Pero es buena práctica utilizarlas para dejar más claro en el código que el objetivo es transformar las unidades de un valor de días a segundos, o al revés si se divide por ese valor, evitando así posibles errores en la operación si se introduce un número erróneo, los cuales podrían resultar muy difíciles de corregir.

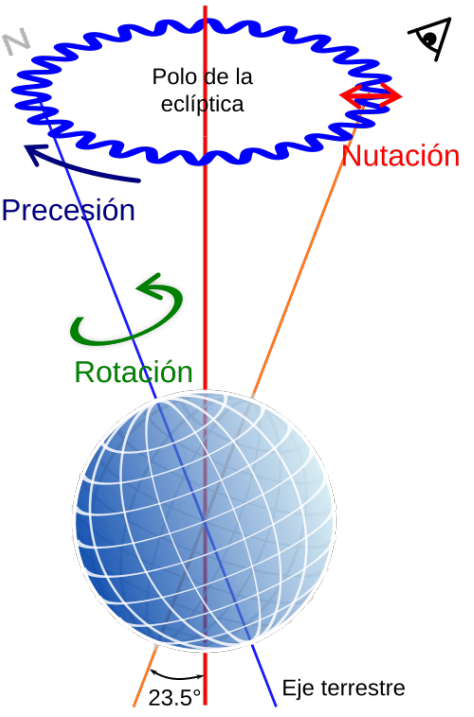

Figura 2. *Movimientos de precesión y nutación del eje terrestre respecto al polo eclíptico (Wikipedia)..*

2.3. Cálculo de la oblicuidad media de la eclíptica

La oblicuidad de la eclíptica es la inclinación media (sin tener en cuenta la nutación) del eje terrestre respecto a la perpendicular al plano de la eclíptica. Su cálculo se hace en la práctica con la evaluación de un polinomio que ajusta la evolución de este valor a lo largo de los siglos. Existen múltiples modos de hacer esto, y por claridad en el código se muestran tres expansiones. La más sencilla es la adoptada en las resoluciones de la UAI de 1976 [8], utilizada aún por el servidor Horizons. En la práctica nosotros también usaremos esta expansión para tratar de acercarnos lo más posible a este servidor de efemérides. La expansión de Laskar es compatible con la anterior, da esencialmente el mismo resultado, pero puede utilizarse más lejos del año 2000. La expansión de Capitaine et al. [9] es la utilizada en las resoluciones de la UAI del año 2000, y en principio no la utilizaremos, dado que implica también el uso de otros algoritmos por consistencia. Las referencias se muestran en el código.

Existen expansiones más modernas como la correspondiente al trabajo de Vondrak et al. [10], pero son algo más laboriosas de implementar y no añaden ningún valor especial al propósito de nuestros cálculos. Aunque su uso sería interesante en programas que pretendan proporcionar una representación realista del cielo nocturno muchos miles de años en el pasado.

El método presente a continuación, *trueObliquity*, calcula la oblicuidad verdadera de la eclíptica, como resultado de sumarle a la oblicuidad media la nutación en oblicuidad. Será útil en el futuro para la transformación de coordenadas eclípticas.

2.4. Cálculo de los ángulos de precesión

Los ángulos de precesión permiten corregir las coordenadas celestes por el lento cambio a lo largo de los siglos de la dirección hacia donde apunta el eje terrestre en el cielo, que hace un movimiento

circular completo cada 26000 años, manteniendo una inclinación igual a la oblicuidad (ver Fig. 2). Como sabemos esto hace que miles de años atrás (o en un futuro lejano), las estrellas que marcaban la dirección del polo norte celeste fueran otras, como Vega o Thuban. El algoritmo propuesto para calcular este efecto es el clásico de Lieske [11], que de nuevo es el utilizado en el servidor Horizons. El código está simplificado asumiendo que la época de partida es siempre el año 2000, lo que nos permitirá en el futuro transformar coordenadas por precesión tanto desde el año 2000 hasta otra fecha, como hacia el año 2000. Al igual que sucede con la oblicuidad y el resto de algoritmos, la UAI ha adoptado oficialmente algoritmos más evolucionados para corregir por precesión, pero en la mayoría de los servidores de efemérides se siguen utilizando algoritmos antiguos.

2.5. Cálculo del tiempo sidéreo local

La subrutina propuesta para obtener el tiempo sidéreo local es bastante completa y compatible con Horizons. En primer lugar hay que utilizar una formulación que proporciona el valor medio para Greenwich, utilizando el día juliano de la medianoche anterior en UT. El tiempo restado se añade posteriormente en otra variable, lo que aumenta la precisión del resultado respecto de usar sólo el día juliano con todos sus decimales en la formulación. El valor final se obtiene añadiendo la longitud del observador, la proyección sobre el eje de ascensión recta de la nutación en longitud eclíptica, y un par de términos de poca amplitud que dependen del nodo ascendente de la órbita de la Luna. Hay una diferencia mínima con Horizons, como puede comprobarse en su documentación [12], pero esencialmente es el mismo cálculo.

El valor de entrada es el día juliano en UT, normalmente UTC (Tiempo Universal Coordinado), pero si es posible se puede introducir el valor en UT1, corregido por el movimiento del polo terrestre, que es un parámetro imposible de prever, y que es monitorizado por el IERS y publicado en sus boletines [5]. La corrección para un instante dado es habitualmente de una fracción de segundo de tiempo, y por los requerimientos de precisión que tenemos ignoraremos este efecto. Aunque es importante tener presente que la corrección podría llegar a ser mayor, dado que la práctica de añadir segundos intercalares al UTC desaparecerá en unos pocos años.

2.6. Código

El código completo se presenta a continuación. Las últimas líneas muestran los resultados del ejemplo de partida, para el día juliano 2457407.5 y la longitud geográfica correspondiente a Madrid. El resultado es TT-UT1 = 66.6s, frente a los 68.2s que muestra Horizons. La discrepancia en el valor del tiempo sidéreo es de apenas 0.2s. En los boletines del IERS es posible comprobar que el DUT1 (UT1-UTC) tiene un valor de 0.0447225s para el instante del ejemplo, valor que puede sumarse al día juliano para reducir la discrepancia a 0.15s. En todo caso, sin esta corrección la calculadora del USNO [13] proporciona un valor coincidente con el de este programa, con una diferencia del orden del milisegundo de tiempo. El error máximo de este programa debería ser de pocas centésimas de segundo, debido a la aproximación utilizada para la nutación.

```
1  package journal;
2
3  /**
4   * Computes multiple parameters related to the orientation of the Earth: TT minus UT1,
5   * nutation angles, mean obliquity, precession angles, and local apparent sidereal
        time
6   */
7  public class EarthAngles {
8
9      /**
10      * Computes the difference between Terrestrial Time and Universal Time UT1
```

```
     * @param jd The Julian day
     * @return TT-UT1 in seconds
     */
    public static double TTminusUT1(double jd) {
        JulianDay julDay = new JulianDay(jd);
        int year = julDay.year;
        int month = julDay.month;
        double day = julDay.day + julDay.getDayFraction();

        double TTminusUT1 = 0;
        double ndot = -25.858, c0 = 0.91072 * (ndot + 26.0) ;
        if (year < -500 || year >= 2200) {
               double u = (jd - 2385800.5) / 36525.0; // centuries since J1820
               TTminusUT1 = -20 + 32.0 * u * u;
        } else {
               double x = year + (month - 1 + (day - 1) / 30.0) / 12.0;
               double x2 = x * x, x3 = x2 * x, x4 = x3 * x, x8 = x4 * x4;
               if (year < 1600) {
                      TTminusUT1 = 10535.328003 - 9.9952386275 * x + 0.00306730763 * x2
                          - 7.7634069836E-6 * x3 + 3.1331045394E-9 * x4 +
                             8.2255308544E-12 * x2 * x3 - 7.4861647156E-15 * x4 * x2 +
                                 1.936246155E-18 * x4 * x3 - 8.4892249378E-23 * x8;
               } else {
                      TTminusUT1 = -1027175.34776 + 2523.2566254 * x - 1.8856868491 *
                          x2 + 5.8692462279E-5 * x3 + 3.3379295816E-7 * x4 +
                             1.7758961671E-10 * x2 * x3 - 2.7889902806E-13 * x2 * x4 +
                                 1.0224295822E-16 * x3 * x4 - 1.2528102371E-20 * x8;
               }
               c0 = 0.91072 * (ndot + 25.858) ;
        }
        double c = -c0 * Math.pow((jd - 2435109.0) / 36525.0, 2);
        if (year < 1955 || year > 2005) TTminusUT1 += c;
        return TTminusUT1;
    }

    /**
     * Transforms the input Julian day into centuries from J2000 in Terrestrial Time,
         to compute ephemerides
     * @param jd Julian day
     * @param UT True if input Julian day is in UT, otherwise TT is assumed
     * @return Centuries from J2000 in TT
     */
    public static double toCenturiesRespectJ2000(double jd, boolean UT) {
        if (UT) jd = jd + TTminusUT1(jd) / Constant.SECONDS_PER_DAY;
        return (jd - Constant.J2000) / Constant.JULIAN_DAYS_PER_CENTURY;

    }

    /**
     * Computes nutation in longitude and obliquity
     * @param jd Julian day in UT
     * @return Nutation angles in radians
     */
    public static double[] nutation(double jd) {
        double t = toCenturiesRespectJ2000(jd, true);

        // Compute approximate nutation
        // Mean longitude of the ascending node of the Moon
        double om = (125.04452 - 1934.136261 * t + 0.0020708 * t * t + t * t * t /
            450000.0) * Constant.DEG_TO_RAD;
        // 2 * Mean longitude of Sun
        double l2 = 2 * (280.4665 + 36000.7698 * t) * Constant.DEG_TO_RAD;
        // 2 * Mean longitude of Moon
        double l2p = 2 * (218.3165 + 481267.8813 * t) * Constant.DEG_TO_RAD;

```

```
70          double nutLon = -17.20 * Math.sin(om) - 1.32 * Math.sin(l2) - 0.23 *
                Math.sin(l2p) + 0.21 * Math.sin(2 * om);
71          double nutObl = 9.20 * Math.cos(om) + 0.57 * Math.cos(l2) + 0.1 *
                Math.cos(l2p) - 0.09 * Math.cos(2 * om);
72
73          return new double[] {nutLon * Constant.ARCSEC_TO_RAD, nutObl *
                Constant.ARCSEC_TO_RAD};
74      }
75
76      /**
77       * Returns the mean obliquity in radians
78       * @param jd Julian day in UT
79       * @return Mean obliquity
80       */
81      public static double meanObliquity(double jd) {
82          double t = toCenturiesRespectJ2000(jd, true);
83
84          // IAU 1976 formulation, still used by Horizons
85          double eps0 = 84381.448;
86          double[] pol = {-468150., -590., 181320.};
87          // J. Laskar's expansion comes from "Secular terms of classical planetary
                theories using the results of general theory," Astronomy and Astrophysics
                157, 59070 (1986)
88          //double eps0 = 84381.448;
89          //double[] pol = {-468093., -155., 199925., -5138., -24967., -3905., 712.,
                2787., 579., 245. };
90          // Capitaine et al. Astronomy and Astrophysics 412, 567-586, (2003), Hilton et
                al. 2006,
91          //double eps0 = 84381.406;
92          //double[] pol = {-468367.69, -183.1, 200340., -5760., -43400.};
93
94          double meanObliquity = 0;
95          for (int i=0; i<pol.length; i++) {
96              meanObliquity += pol[i] * 0.01 * Math.pow(t * 0.01, i + 1);
97          }
98          return (meanObliquity + eps0) * Constant.ARCSEC_TO_RAD;
99      }
100
101     /**
102      * True obliquity
103      * @param jd Julian day
104      * @return True obliquity in radians
105      */
106     public static double trueObliquity(double jd) {
107         return meanObliquity(jd) + nutation(jd)[1];
108     }
109
110     /**
111      * Computes the angles to correct for precession between J2000 and another date,
112      * using the method by Lieske (IAU 1976)
113      * @param jd Julian day in UT
114      * @return The output precession angles
115      */
116     public static double[] precessionAnglesFromJ2000(double jd) {
117         double t = toCenturiesRespectJ2000(jd, true);
118
119         double zeta = 2306.2181 * t + 0.30188 * t * t + 0.017998 * t * t * t;
120         double z = 2306.2181 * t + 1.09468 * t * t + 0.018203 * t * t * t;
121         double theta = 2004.3109 * t - 0.42665 * t * t - 0.041833 * t * t * t;
122
123         zeta = Util.normalizeRadians(zeta * Constant.ARCSEC_TO_RAD);
124         z = Util.normalizeRadians(z * Constant.ARCSEC_TO_RAD);
125         theta = Util.normalizeRadians(theta * Constant.ARCSEC_TO_RAD);
126
127         return new double[] {zeta, z, theta};
```

```
128     }
129
130     /**
131      * Returns the local apparent sidereal time
132      * @param jd Julian day of calculations, in UTC (UT1 if possible)
133      * @param obsLon Longitude of the observer in radians
134      * @return Apparent sidereal time in radians
135      */
136     public static double localApparentSiderealTime(double jd, double obsLon) {
137         // Obtain local apparent sidereal time
138         double jd0 = Math.floor(jd - 0.5) + 0.5; // previous midnight
139         double t0 = toCenturiesRespectJ2000(jd0, false); // centuries from previous
                midnight
140         double secs = (jd - jd0) * Constant.SECONDS_PER_DAY;
141         double gmst = (((((-6.2e-6 * t0) + 9.3104e-2) * t0) + 8640184.812866) * t0) +
                24110.54841;
142         double msday = 1.0 + (((((-1.86e-5 * t0) + 0.186208) * t0) + 8640184.812866) /
                (Constant.SECONDS_PER_DAY *
143                    Constant.JULIAN_DAYS_PER_CENTURY));
144         gmst = (gmst + msday * secs) * 15.0 * Constant.ARCSEC_TO_RAD;
145
146         // IAU 1994 resolution C7 added two terms (dependent on the mean ascending
                node of the lunar orbit omega)
147         // to the equation of equinoxes, taking effect since 1997-02-27
148         double dt = toCenturiesRespectJ2000(jd, true);
149         double omega = 125.04452 - 1934.136261 * dt + 0.0020708 * dt * dt + (dt * dt *
                dt) / 450000;
150         omega = Util.normalizeDegrees(omega);
151
152         double nutLon = nutation(jd)[0]; // First element of the array returned by
                nutation
153         double last = Util.normalizeRadians(
154                gmst + obsLon + nutLon * Math.cos(meanObliquity(jd))
155                + 0.00264 * Math.sin(omega) * Constant.ARCSEC_TO_RAD
156                + 0.000063 * Math.sin(2 * omega) * Constant.ARCSEC_TO_RAD
157                );
158
159         return last;
160     }
161
162     /**
163      * Test program
164      * @param args Not used
165      */
166     public static void main(String[] args) {
167         try {
168             // Input Julian day
169             JulianDay julDay = new JulianDay(2016, 1, 20);
170             double jd = julDay.getJulianDay(); // 2457407.5
171             // Input longitude for the observer in radians. 0 = Greenwich
172             double lon = -(3 + 41 / 60.0 + 18.1 / 3600.0) * Constant.DEG_TO_RAD; //
                    -3.68836 deg
173
174             // Compute values
175             double ttMinusUt1 = TTminusUT1(jd);
176             double[] nut = nutation(jd);
177             double obl = meanObliquity(jd);
178             double last = localApparentSiderealTime(jd, lon);
179
180             // Report values
181             // To check TT-UT1:
182             // https://maia.usno.navy.mil/ser7/deltat.data
183             //
                    https://web.archive.org/web/20220918033245/http://asa.hmnao.com/SecK/DeltaT.html
```

```
184             System.out.println("TT-UT1: "+(float) ttMinusUt1+" s");// 66.6s, 68.18 in
                    Horizons
185             System.out.println("Nutation: "+Util.formatDEC(nut[0], 2)+", "+
186                    Util.formatDEC(nut[1], 2));                // -00° 00' 00.72", -00°
                           00' 09.62"
187             System.out.println("Obliquity: "+Util.formatDEC(obl, 2)); // 23° 26' 13.93"
188             System.out.println("LAST:  "+Util.formatRA(last, 3)); // 07h 40m 31.144s,
                    07:40:31.1426 in USNO, 07 40 31.3197 in Horizons
189             System.out.println();
190
191             // Check current sidereal time with the web page (for longitude = 0):
                    https://www.localsiderealtime.com/
192             jd = new JulianDay(System.currentTimeMillis()).getJulianDay(); // Julian
                    day now in UTC from the computer
193             double lastNow = localApparentSiderealTime(jd, 0); // For Greenwich
194
195             System.out.println("GAST now: "+Util.formatRA(lastNow, 3));
196         } catch (Exception exc) {
197             exc.printStackTrace();
198         }
199     }
200 }
```

3. Transformaciones de coordenadas

En la sección anterior hemos visto el cálculo de ciertos parámetros ligados a la orientación y rotación del eje de la Tierra, los cuales vamos a utilizar ahora para desarrollar un código que permite rotar las coordenadas entre distintos sistemas, como los de coordenadas eclípticas, horizontales, y también galácticas y supergalácticas. Como siempre el código puede encontrarse en el repositorio de GitHub [2] correspondiente a esta sección. En próximas entregas, cuando tengamos todas la herramientas previas implementadas, empezaremos a utilizarlas para el cálculo de efemérides.

El código se ha organizado para intentar que sea fácil de entender y leer. En primer lugar, se definen métodos que permiten obtener la matriz de rotación, para cierto ángulo de entrada (siempre en radianes, siguiendo el criterio de códigos anteriores), para cada uno de los ejes. Se han llamado *getRotX*, *getRotY*, y *getRotZ*. A continuación se define el método que permite multiplicar un vector, que representa las coordenadas cartesianas de un objeto (x, y, z, o bien x, y, z, vx, vy, vz, incluyendo las velocidades) por una matriz de rotación. Los otros *ladrillos* que necesitamos son los dos métodos siguientes para transformar coordenadas esféricas (como ascensión recta y declinación) en cartesianas, asumiendo que la norma del vector será la unidad, y el proceso inverso. Con estas herramientas implementadas estamos en condiciones de hacer cualquier tipo de conversión de coordenadas en muy pocas líneas de código.

3.1. Coordenadas eclípticas

La conversión de coordenadas ecuatoriales a eclípticas está definida por una rotación alrededor del eje X por un ángulo igual a la oblicuidad de la eclíptica. Esto deja la coordenada de longitud inalterada. Para seguir el criterio de los resultados que ofrece el servidor Horizons se ha incluido el día juliano en los parámetros de entrada, y el ángulo por el que se rota es la oblicuidad verdadera, que incluye la nutación en oblicuidad sumada a la oblicuidad media para el instante de cálculo. Todo ello llama al código de la entrega anterior. La conversión inversa es simplemente la misma rotación con el ángulo cambiado de signo. En algunos casos las coordenadas a convertir pueden ser astrométricas J2000, no aparentes, en cuyo caso utilizaríamos la oblicuidad media para el año 2000, conocida como la constante *eps0*.

3.2. Coordenadas horizontales

La conversión de ecuatoriales a horizontales requiere, además de la fecha, de la posición geográfica del observador. En primer lugar se hace una rotación sobre el eje Z igual al tiempo sidéreo, que permite cambiar la coordenada de ascensión recta y referirla al ángulo horario. Luego se hace una rotación sobre el eje Y por un ángulo igual a 90 grados menos la latitud, para referir la coordenada de declinación a la elevación respecto a la posición del observador. Un observador situado en el polo norte vería una elevación igual a la declinación, en cuyo caso esta rotación sería por un ángulo nulo. El último paso consiste en corregir el acimut obtenido por el criterio habitual de referirlo respecto a la dirección norte, en lugar de considerar que el acimut cero se refiere al sur (dirección del meridiano en el hemisferio norte), y al hecho de que su valor aumenta cuando la ascensión recta disminuye. Esto implica la transformación a esféricas, cambiar el signo y sumar 180° y volver a las coordenadas de tipo cartesianas para devolver el resultado. Es fácil comprobar que la conversión inversa del método siguiente sigue exactamente los mismos pasos en orden contrario, inviertiendo el orden de las rotaciones y los signos de los ángulos.

3.3. Coordenadas galácticas

La conversión de coordenadas ecuatoriales a galácticas requiere conocer la dirección del centro de la galaxia, que como sabemos coincide con la constelación de *Sagitario*. En la práctica se utiliza el polo norte galáctico para las rotaciones, que se localiza en la constelación de *La Cabellera de Berenice*. En el código se muestra en unas constantes definidas al inicio como privadas y de tipo final, que significa que su valor no puede ser obtenido desde otras clases Java ni alterado, funcionando a todos los efectos como constantes disponibles sólo en esta pieza de código. La definición de la dirección precisa del polo galáctico depende del criterio utilizado, y como puede verse nosotros adoptaremos la definición del trabajo de Jia-Cheng Liu et al. (2010). El lector interesado en conocer más detalles puede acudir al artículo de los autores, disponible libremente en el portal *arXiv*, siguiendo el enlace presente en los comentarios del código.

En cuanto a la rotación, se parece mucho a la anterior, reemplazando el tiempo sidéreo por la ascensión recta del polo norte galáctico, y la latitud por su declinación. Luego hay que hacer una transformación en la coordenada de longitud para obtener la longitud galáctica final referida al punto que se toma como origen de la longitud galáctica. Para ello se le suma la longitud del nodo más 180°, si bien en el código esto se hace mediante una rotación en el eje Z. La transformación inversa es totalmente simétrica. Es fácil comprobar que la dirección galáctica $(0, 0)$ correspondiente al centro de la galaxia corresponde a las coordenadas ecuatoriales J2000 (17h 45m 37.199s, -28° 56' 10.221"), las cuales no coinciden con la posición de la fuente Sagittarius A*, que marca la posición física del centro de la Vía Láctea (17h 45m 40.04s, -29° 00' 28.1"). Como se explica en el artículo esto es un problema inherente a la definición clásica de las coordenadas galácticas. En el programa hemos adoptado las constantes que menciona Jia-Cheng Liu et al. para el sistema FK5, para poder comparar los resultados con otros servidores como el NED [14]. Es posible cambiar estos valores para utilizar coordenadas ICRS, o bien cambiar los valores por los que sugiere el autor, para que además de utilizar el sistema de referencia ICRF, la posición galáctica (0, 0) coincida con el centro físico. El servidor Horizons utiliza otra definición del sistema de coordenadas galáctico, lo que incrementa la confusión.

Es importante mencionar que las coordenadas ecuatoriales de entrada y salida deben ser J2000.0, referidas al equinoccio 2000. Por tanto, puede ser necesario corregirlas por precesión antes de usar estos métodos. Lo mismo sucede con las coordenadas supergalácticas. El procedimiento para corregir por precesión se muestra más adelante.

3.4. Coordenadas supergalácticas

El sistema de coordenadas supergaláctico se refiere a un plano identificado a mitad del siglo pasado en el que se concentran muchos cúmulos de galaxias según se observan desde la Vía Láctea. Su origen se eligió para cortar al plano galáctico en la constelación de *Casiopea*, mientras su polo norte se dirige hacia *Hércules*. La definición que hemos adoptado sigue el trabajo clásico de Vaucoleurs et al. (1991), con la posición del polo referida a las coordenadas galácticas, aunque en el código se referencia el trabajo posterior de Lahav et al. (2000) [15], cuya introducción puede ser interesante de leer. Como puede verse, el plano galáctico y supergaláctico son casi perpendiculares, lo que significa que todos estos cúmulos y supercúmulos de galaxias son fáciles de observar desde la Tierra, especialmente en invierno. Por ejemplo, el centro del cúmulo de Virgo se localiza en la posición supergaláctica (104°, −2°), muy cerca del plano.

La transformación de coordenadas ecuatoriales a supergalácticas exige el paso previo de transformar las coordenadas a galácticas, debido a que el polo está referido a la posición galáctica. Luego se hace una transformación similar a las anteriores, restando al final 90° a la longitud para corregir el origen. La transformación inversa es de nuevo totalmente simétrica.

En general, todas las rotaciones son conceptualmente equivalentes y pueden definirise con tres ángulos a lo sumo, por lo que resulta muy útil y limpio disponer de un código que las implemente a partir de sus elementos más basicos.

3.5. Corrección de coordenadas por precesión

Como nota al margen, no incluido en el código listado más abajo, es posible corregir las coordenadas por precesión utilizando los métodos de rotación de esta entrega junto con el método que vimos para calcular los ángulos de precesión en la entrega anterior. Siguiendo la expresión (5) del artículo de Lieske [11]. Sea *pa* los tres ángulos de precesión del método *EarthAngles.precessionAnglesFromJ2000*, calculados para la fecha final deseada (recordar que el índice cero se refiere al primer elemento de la lista en el lenguaje Java), y *p* el vector de entrada representando las coordenadas ecuatoriales cartesianas de un objeto para la época J2000, con sólo una línea de código podemos corregir por precesión.

```
1        return  rotate ( rotate ( rotate (p, getRotZ(−pa[1])), getRotY(pa[2])), getRotZ(−pa[0]));
```

Utilizaremos esto en un futuro durante el cálculo de efemérides, así como una formulación equivalente para la nutación. La transformación simétrica con los mismos ángulos (cambiados de signo y con las rotaciones en orden inverso) nos devolvería las coordenadas J2000.

3.6. Código

El código completo se presenta a continuación, con dos ejemplos al final del código que incluyen los resultados del servidor Horizons, como comparación. En el caso de las coordenadas galácticas es mejor recurrir a la calculadora presente en el servidor NED, el cual utiliza utiliza la misma definición del sistema de coordenadas galáctico y supergaláctico que hacemos en este código. En todos los casos los resultados son coincidentes con los de Horizons o el servidor NED, hasta la tercera cifra decimal mostrada. En concreto, los resultados del primer ejemplo son:

```
1  Acimut:      118.369
2  Elevation :  −22.625
3  RA:  −154.261
4  DEC: −35.685
5   Ecliptic   longitude :  −142.786
```

```
Ecliptic latitude : −23.262
RA: −154.261
DEC: −35.685
Galactic longitude : −45.445
Galactic latitude : 26.021
RA: −154.261
DEC: −35.685
Supergalactic longitude : 153.839
Supergalactic latitude : 0.253
RA: −154.261
DEC: −35.685
```

```java
package journal;

/**
 * A class to perform coordinates transformations between equatorial, ecliptic, galactic, and supergalatic systems.
 * The orientation of the galactic plane follows Jia-Cheng Liu et al. 2010 (see http://arxiv.org/abs/1010.3773).
 * The orientation of the supergalactic pole follows Lahav et al 2000 (see https://arxiv.org/abs/astro-ph/9809343).
 */
public class CoordinateSystem {

    // J2000 position of the galactic pole in FK5 system. See Jia-Cheng Liu et al. 2010,
    // http://arxiv.org/abs/1010.3773. Distance to galactic center was set to 25830 +/- 500
    // light years (VERA collaboration, 2020)
    // FK5
    private static final double GALACTIC_POLE_RA_J2000 = 192.85948120833334 * Constant.DEG_TO_RAD;
    private static final double GALACTIC_POLE_DEC_J2000 = 27.128251194444445 * Constant.DEG_TO_RAD;
    private static final double GALACTIC_NODE_J2000 = 122.93191857 * Constant.DEG_TO_RAD;
    // Redefinition by Jia-Cheng Liu et al.
    //private static final double GALACTIC_POLE_RA_J2000 = 192.90297999208332 * Constant.DEG_TO_RAD;
    //private static final double GALACTIC_POLE_DEC_J2000 = 27.103109214444444 * Constant.DEG_TO_RAD;
    //private static final double GALACTIC_NODE_J2000 = 123.0075021536 * Constant.DEG_TO_RAD;
    // Galactic position of the supergalactic pole. See Lahav et al 2000, MNRAS 312 166-176,
    // https://arxiv.org/abs/astro-ph/9809343. Definition by G. de Vaucouleurs 1991.
    private static final double SUPER_GALACTIC_POLE_RA = 47.37 * Constant.DEG_TO_RAD;
    private static final double SUPER_GALACTIC_POLE_DEC = 6.32 * Constant.DEG_TO_RAD;

    /**
     * Returns a 3x3 pure rotation matrix along axis X.
     * @param angle The angle to rotate.
     * @return The matrix.
     */
    public static double[][] getRotX(double angle) {
        return new double[][] {
            new double[] {1.0, 0.0, 0.0},
            new double[] {0.0, Math.cos(angle), Math.sin(angle)},
            new double[] {0.0, -Math.sin(angle), Math.cos(angle)}};
    }

    /**
```

```
39      * Returns a 3x3 pure rotation matrix along axis Y.
40      * @param angle The angle to rotate.
41      * @return The matrix.
42      */
43     public static double[][] getRotY(double angle) {
44         return new double[][] {
45             new double[] {Math.cos(angle), 0.0, -Math.sin(angle)},
46             new double[] {0.0, 1.0, 0.0},
47             new double[] {Math.sin(angle), 0.0, Math.cos(angle)}};
48     }
49
50     /**
51      * Returns a 3x3 pure rotation matrix along axis Z.
52      * @param angle The angle to rotate.
53      * @return The matrix.
54      */
55     public static double[][] getRotZ(double angle) {
56         return new double[][] {
57             new double[] {Math.cos(angle), Math.sin(angle), 0.0},
58             new double[] {-Math.sin(angle), Math.cos(angle), 0.0},
59             new double[] {0.0, 0.0, 1.0}};
60     }
61
62     /**
63      * Multiplication of a vector with a matrix
64      * @param p The vector, with 3 (position) or 6 (position and velocity) components
65      * @param m The 3x3 rotation matrix
66      * @return The result of the rotation
67      */
68     public static double[] rotate(double[] p, double[][] m) {
69         double[] out = new double[p.length];
70         for (int i=0; i<p.length; i++) {
71             int ip = i;
72             if (i > 2) ip = i - 3;
73             out[i] = 0;
74             for (int j=0; j<3; j++) {
75                 out[i] += m[ip][j] * p[j];
76             }
77         }
78         return out;
79     }
80
81     /**
82      * Transform coordinates from cartesian, x y z, to spherical, lon lat r
83      * @param p x, y, z
84      * @return lon, lat (radians), r
85      */
86     public static double[] cartesianToSpherical(double[] p) {
87         double lon, lat;
88
89         double x = p[0], y = p[1], z = p[2];
90         if (y != 0.0 || x!= 0.0) {
91             double h = Math.hypot(x, y);
92             lon = Math.atan2(y, x);
93             lat = Math.atan2(z, h);
94         } else {
95             lon = 0.0;
96             lat = Constant.PI_OVER_TWO;
97             if (z < 0.0) lat = -lat;
98         }
99         double rad = Math.sqrt(x * x + y * y + z * z);
100
101        return new double[] {lon, lat, rad};
102    }
103
```

```
104     /**
105      * Transforms coordinates from spherical, lon, lat, to cartesian, x y z
106      * @param lon Longitude in radians
107      * @param lat Latitude in radians
108      * @return x y z coordinates, with unity as the norm of the vector
109      */
110     public static double[] sphericalToCartesian(double lon, double lat) {
111         double rad = 1;
112         double cl = Math.cos(lat);
113         double x = rad * Math.cos(lon) * cl;
114         double y = rad * Math.sin(lon) * cl;
115         double z = rad * Math.sin(lat);
116 
117         return new double[] { x, y, z };
118     }
119 
120     /**
121      * Transforms coordinates from equatorial to ecliptic
122      * @param p Rectangular coordinates
123      * @param jd Julian day
124      * @return Ecliptic rectangular coordinates
125      */
126     public static double[] equatorialToEcliptic(double[] p, double jd) {
127         return rotate(p, getRotX(EarthAngles.trueObliquity(jd)));
128     }
129 
130     /**
131      * Transforms coordinates from ecliptic to equatorial
132      * @param p Rectangular coordinates
133      * @param jd Julian day
134      * @return Equatorial rectangular coordinates
135      */
136     public static double[] eclipticToEquatorial(double[] p, double jd) {
137         return rotate(p, getRotX(-EarthAngles.trueObliquity(jd)));
138     }
139 
140     /**
141      * Transforms coordinates from equatorial to horizontal
142      * @param p Rectangular coordinates
143      * @param jd Julian day
144      * @param obsLon Observer's longitude in radians
145      * @param obsLat Observer's latitude in radians
146      * @return Horizontal rectangular coordinates
147      */
148     public static double[] equatorialToHorizontal(double[] p, double jd, double
          obsLon, double obsLat) {
149         double[] out = rotate(rotate(p,
              getRotZ(EarthAngles.localApparentSiderealTime(jd, obsLon))),
              getRotY((Constant.PI_OVER_TWO - obsLat)));
150         out = cartesianToSpherical(out);
151         return sphericalToCartesian(Math.PI - out[0], out[1]);
152     }
153 
154     /**
155      * Transforms coordinates from horizontal to equatorial
156      * @param p Rectangular coordinates
157      * @param jd Julian day
158      * @param obsLon Observer's longitude in radians
159      * @param obsLat Observer's latitude in radians
160      * @return Equatorial rectangular coordinates
161      */
162     public static double[] horizontalToEquatorial(double[] p, double jd, double
          obsLon, double obsLat) {
163         double[] in = cartesianToSpherical(p);
164         in = sphericalToCartesian(Math.PI - in[0], in[1]);
```

```
165         return rotate(rotate(in, getRotY(-(Constant.PI_OVER_TWO - obsLat))),
                getRotZ(-EarthAngles.localApparentSiderealTime(jd, obsLon)));
166     }
167
168     /**
169      * Transforms coordinates from J2000 equatorial to galactic
170      * @param p Rectangular coordinates
171      * @return Galactic rectangular coordinates
172      */
173     public static double[] equatorialJ2000ToGalactic(double[] p) {
174         return rotate(rotate(rotate(p, getRotZ(GALACTIC_POLE_RA_J2000)),
175                 getRotY(Constant.PI_OVER_TWO - GALACTIC_POLE_DEC_J2000)),
                    getRotZ(Math.PI-GALACTIC_NODE_J2000));
176     }
177
178     /**
179      * Transforms coordinates from galactic to J2000 equatorial
180      * @param p Rectangular coordinates
181      * @return Equatorial rectangular coordinates
182      */
183     public static double[] galacticToEquatorialJ2000(double[] p) {
184         return rotate(rotate(rotate(p, getRotZ(-(Math.PI - GALACTIC_NODE_J2000))),
185                 getRotY(-(Constant.PI_OVER_TWO - GALACTIC_POLE_DEC_J2000))),
                    getRotZ(-GALACTIC_POLE_RA_J2000));
186     }
187
188     /**
189      * Transforms coordinates from J2000 equatorial to supergalactic
190      * @param p Rectangular coordinates
191      * @return Supergalactic rectangular coordinates
192      */
193     public static double[] equatorialJ2000ToSupergalactic(double[] p) {
194         return rotate(rotate(rotate(equatorialJ2000ToGalactic(p),
                getRotZ(SUPER_GALACTIC_POLE_RA)),
195                 getRotY(Constant.PI_OVER_TWO - SUPER_GALACTIC_POLE_DEC)),
                    getRotZ(Constant.PI_OVER_TWO));
196     }
197
198     /**
199      * Transforms coordinates from supergalactic to J2000 equatorial
200      * @param p Rectangular coordinates
201      * @return J2000 equatorial rectangular coordinates
202      */
203     public static double[] supergalacticToEquatorialJ2000(double[] p) {
204         return galacticToEquatorialJ2000(rotate(rotate(rotate(p,
                getRotZ(-Constant.PI_OVER_TWO)),
205                 getRotY(-(Constant.PI_OVER_TWO - SUPER_GALACTIC_POLE_DEC))),
                    getRotZ(-SUPER_GALACTIC_POLE_RA)));
206     }
207
208     /**
209      * Test program
210      * @param s Not used
211      */
212     public static void main(String[] s) {
213         try {
214             /*
215             Example fata from Horizons, for galactic better test from
                    https://ned.ipac.caltech.edu/forms/calculator.html
216             Date__(UT)__HR:MN  R.A._(a-appar)_DEC. Azi____(a-app)___Elev L_Ap_Sid_Time
                    TDB-UT    ObsEcLon   ObsEcLat    GlxLon    GlxLat L_Ap_Hour_Ang
217             2016-Jan-20 00:00 m 205.73877 -35.68542 118.369371 -22.625219 7.6753665728
                    68.184474 217.2137137 -23.2621310 314.366606 26.140141 -6.040551182
218             2000-Jan-20 00:00 m 31.19459 -13.82278 255.493146 -4.310376 7.6670225044
                    64.184432 23.8036275 -24.7875597 178.724053 -68.320060 5.587383271
```

```
219             */
220             double jd = 2457407.5;
221             double obsLon = -3.6879 * Constant.DEG_TO_RAD;
222             double obsLat = 40.408414 * Constant.DEG_TO_RAD;
223             double[] eq = sphericalToCartesian(205.73877 * Constant.DEG_TO_RAD,
                    -35.68542 * Constant.DEG_TO_RAD);
224             // Uncomment for a second test
225             //jd = 2451563.5;
226             //eq = sphericalToCartesian(31.19459 * Constant.DEG_TO_RAD, -13.82278 *
                    Constant.DEG_TO_RAD);
227
228             // Equatorial to horizontal
229             double[] hor = equatorialToHorizontal(eq, jd, obsLon, obsLat);
230             double[] shor = cartesianToSpherical(hor);
231             System.out.println("Acimut: " + Util.formatValue(shor[0] *
                    Constant.RAD_TO_DEG, 3));
232             System.out.println("Elevation: " + Util.formatValue(shor[1] *
                    Constant.RAD_TO_DEG, 3));
233             // Invert back to equatorial
234             eq = horizontalToEquatorial(hor, jd, obsLon, obsLat);
235             double[] seq = cartesianToSpherical(eq);
236             System.out.println("RA: " + Util.formatValue(seq[0] * Constant.RAD_TO_DEG,
                    3));
237             System.out.println("DEC: " + Util.formatValue(seq[1] * Constant.RAD_TO_DEG,
                    3));
238             // To ecliptic, using the ecliptic of date (true obliquity), as in the
                    above results from Horizons
239             double[] ecl = equatorialToEcliptic(eq, jd);
240             double[] secl = cartesianToSpherical(ecl);
241             System.out.println("Ecliptic longitude: " + Util.formatValue(secl[0] *
                    Constant.RAD_TO_DEG, 3));
242             System.out.println("Ecliptic latitude: " + Util.formatValue(secl[1] *
                    Constant.RAD_TO_DEG, 3));
243             eq = horizontalToEquatorial(hor, jd, obsLon, obsLat);
244             seq = cartesianToSpherical(eq);
245             System.out.println("RA: " + Util.formatValue(seq[0] * Constant.RAD_TO_DEG,
                    3));
246             System.out.println("DEC: " + Util.formatValue(seq[1] * Constant.RAD_TO_DEG,
                    3));
247             // To galactic
248             //eq = sphericalToCartesian(192.85948120833334 * Constant.DEG_TO_RAD,
                    27.128251194444445 * Constant.DEG_TO_RAD);
249             double[] gal = equatorialJ2000ToGalactic(eq);
250             double[] sgal = cartesianToSpherical(gal);
251             System.out.println("Galactic longitude: " + Util.formatValue(sgal[0] *
                    Constant.RAD_TO_DEG, 3));
252             System.out.println("Galactic latitude: " + Util.formatValue(sgal[1] *
                    Constant.RAD_TO_DEG, 3));
253             eq = galacticToEquatorialJ2000(gal);
254             seq = cartesianToSpherical(eq);
255             System.out.println("RA: " + Util.formatValue(seq[0] * Constant.RAD_TO_DEG,
                    3));
256             System.out.println("DEC: " + Util.formatValue(seq[1] * Constant.RAD_TO_DEG,
                    3));
257             // To supergalactic
258             double[] supergal = equatorialJ2000ToSupergalactic(eq);
259             double[] ssupergal = cartesianToSpherical(supergal);
260             System.out.println("Supergalactic longitude: " +
                    Util.formatValue(ssupergal[0] * Constant.RAD_TO_DEG, 3));
261             System.out.println("Supergalactic latitude: " +
                    Util.formatValue(ssupergal[1] * Constant.RAD_TO_DEG, 3));
262             eq = supergalacticToEquatorialJ2000(supergal);
263             seq = cartesianToSpherical(eq);
264             System.out.println("RA: " + Util.formatValue(seq[0] * Constant.RAD_TO_DEG,
                    3));
```

```
265             System.out.println("DEC: " + Util.formatValue(seq[1] * Constant.RAD_TO_DEG,
                    3));
266         } catch (Exception exc) {
267             exc.printStackTrace();
268         }
269     }
270 }
```

References

[1] *Astronomical computing: Cálculo del día juliano*, T. Alonso Albi, JCAAC **1**, 57 (2004).

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

[3] *Historical values of the Earth's clock error deltaT and the calculation of eclipses*, Stephenson and Morrison (2004), https://adsabs.harvard.edu/full/2004JHA....35..327M

[4] https://maia.usno.navy.mil/ser7/deltat.data

[5] https://www.iers.org/IERS/EN/DataProducts/EarthOrientationData/eop.html

[6] *Horizons*, disponible en https://ssd.jpl.nasa.gov/horizons/app.html#/

[7] *Astronomical Algorithms*, Jean Meeus (Editorial Atlantic Books, 1998)

[8] *Expressions for the precession quantities based upon the IAU (1976) system of astronomical constants*, J. H. Lieske et al., A&A **58**, 1 (1977). https://adsabs.harvard.edu/full/1977A%26A....58....1L

[9] *Expressions for IAU 2000 precession quantities*, N. Capitaine et al., A&A **412**, 567 (2003).

[10] *New precession expressions, valid for long time intervals*, J. Vondrák et al., A&A **534**, A22 (2011). https://www.aanda.org/articles/aa/pdf/2011/10/aa17274-11.pdf

[11] *Precession matrix based on IAU (1976) system of astronomical constants*, J. H. Lieske, A&A **73**, 282 (1979). https://adsabs.harvard.edu/full/1979A%26A....73..282L

[12] https://ssd.jpl.nasa.gov/horizons/manual.html

[13] https://aa.usno.navy.mil/data/siderealtime

[14] https://ned.ipac.caltech.edu/forms/calculator.html

[15] https://academic.oup.com/mnras/article/312/1/166/984983