

SECTION

Astronomical computing

Tomás Alonso Albi¹¹Starion/European Space Agency, Spain. E-mail: talonsoalbi@gmail.com.**Keywords:** programación, programming, efemérides, ephemerides, cálculo astronómico, astronomical computing

© Este artículo está protegido bajo una licencia Creative Commons Attribution 4.0 License
Este artículo adjunta un *software* accesible en <https://github.com/JCAAC-FAAE>

Presentación

El objetivo de esta sección es desarrollar los fundamentos del cálculo de efemérides astronómicas desde una perspectiva más moderna en comparación con otras fuentes disponibles. Este tipo de cálculos ha sido muy habitual en el pasado en otras publicaciones conocidas como *Sky & Telescope*, pero con el tiempo esta tradición se ha perdido, a pesar de que hoy día las aplicaciones y posibilidades de la programación se han multiplicado. Existen nuevos lenguajes de programación que facilitan el trabajo, y la potencia de los ordenadores permite obtener resultados más precisos y rápidos que nunca. Por otro lado, la mayoría de las fuentes de información disponibles en este campo están escritas en inglés, lo que puede resultar un obstáculo para algunos aficionados.

Por tanto, en esta sección se presentará y explicará en castellano piezas de código que permitirán hacer multitud de cálculos en astronomía. Para no excluir al público anglosajón se incluyen los comentarios del código en inglés, además del resumen al principio del artículo. El lenguaje de programación elegido para presentar el código es Java, que resulta muy verboso y fácil de leer e interpretar, si bien esto depende mucho de cómo se escriba y documente el código. No es necesario que el lector lo escriba por sí mismo, sino que puede ser copiado desde el repositorio de software disponible, para evitar errores de transcripción en las operaciones matemáticas. A lo largo de múltiples artículos veremos desde los fundamentos más básicos como escalas de tiempo y cambios de coordenadas, hasta aplicaciones más interesantes que incluirán obtener la posición precisa del Sol, los planetas, la Luna y otros satélites, y la predicción de eclipses y otros eventos astronómicos. En esta primera ocasión veremos cómo transformar fechas entre nuestro calendario y el día Juliano, que es la base para calcular intervalos de tiempo, y así poder posteriormente obtener las posiciones de los astros.

El objetivo de esta sección no es servir como curso de programación, de manera que partimos de la idea de que el lector ya tiene suficientes conocimientos de programación en un determinado lenguaje como para portar el código a su lenguaje favorito, en el caso de que no sea Java. Históricamente estuvo muy extendido el uso de *Basic*, lo que sigue siendo posible hoy día gracias a las múltiples herramientas de programación gratuitas que existen para tan entrañable lenguaje (algunas como *FreeBasic* muy evolucionadas en comparación con el lenguaje original). Habitualmente los ficheros de código Java se integran en un proyecto utilizando un *Entorno de Desarrollo Integrado*, o IDE en inglés, que facilita mucho el trabajo, permitiendo también el uso del código Java para la programación en dispositivos móviles como Android. El lector es libre de decidir cómo organizar su código y establecer los objetivos que pretenda conseguir con él, pues el código presentado aquí puede utilizarse libremente respetando las normas de distribución del repositorio.

Presentation

In this section we will develop algorithms to cover from the most basic calculations, like obtaining the Julian day in this first chapter, to more complex ones that will allow us to calculate the positions of the celestial bodies and astronomical events like eclipses and many others. The programming language chosen to present the code is

Java, but the readers are expected to write this code in their preferred language, taking advantage of the software repository to avoid errors in the transcription of the mathematical operations.

The Julian day is basically a continuous count of days, with the day 1 set to January 1, 4713 B.C, at noon (UTC). It was developed by Joseph Scaliger and became soon a very useful system to count the time interval between events. In astronomy is essential to compute how much the celestial bodies have moved in a given interval of time, so the Julian day is used in most calculations. In the code presented, the relevant calculations are located around lines 143 (to obtain the Julian day from a civil date in the Gregorian calendar we use today, or even the old civil Julian calendar if the input boolean is set to true), and from line 53, to do the opposite calculation from the Julian day to the civil calendar. The output in the latter is set automatically to the Julian civil calendar for dates corresponding officially to this calendar, before October, 5, 1582. Inexistent dates in 1582 between October, 5 and October, 14 cannot be used and will throw an error. The code follows the astronomical convention for negative years, so year zero in the program is year one B.C. in the calendar, and the first Julian day belongs to the year number -4712 in the program. The comments included, the results presented in the main program at the end of the Java code, and the many tools available online to obtain the Julian day should be enough for the reader to test the calculations presented here.

CÁLCULO DEL DÍA JULIANO

El día Juliano o fecha Juliana es básicamente un recuento sucesivo de días propuesto por Joseph Scaliger en 1583, un año después de la reforma del calendario Gregoriano. Está basado en tres ciclos que se utilizaban en el calendario Juliano, de ahí el nombre establecido. El primer día en este recuento se estableció en el 1 de enero del año 4713 a.C., a las 12^h de Tiempo Universal, por la coincidencia del inicio de los tres ciclos en los que se basa y para poder datar cualquier acontecimiento histórico conocido. El inicio del día se estableció al mediodía solar porque resultaba mucho más fácil determinar este momento que la medianoche, a partir de la elevación máxima del Sol, que coincide con su paso por el meridiano. El lector interesado en saber más puede consultar las páginas de *Wikipedia* [1].

Históricamente el día Juliano se ha utilizado para representar fechas en Tiempo Universal Coordinado (UTC). En nuestros cálculos también haremos esto la mayoría de las veces. La transformación a Tiempo Terrestre la veremos más adelante, si bien no usaremos, por lo general, directamente el día Juliano expresado en esta escala de tiempo. Uno de los problemas con el Tiempo Universal, al entrar en detalles, es la posibilidad de que un determinado año tenga un segundo intercalar, una práctica que dejó de aplicarse hace algunos años. Por simplificación ignoraremos esta posible complicación.

El código que se muestra más abajo puede guardarse en un fichero con el nombre *JulianDay.java* (una clase Java). Contiene tres constructores para crear múltiples objetos que representen fechas diferentes y poder trabajar con los métodos de esta clase u otras que lleguemos a usar en el futuro. Los constructores aceptan como entrada el año, mes, y día en un caso (para una fecha a las 0^h), en otro un valor numérico tipo *long* que representa el número de milisegundos que han transcurrido desde 1970 en UTC (valor que puede obtenerse fácilmente en múltiples lenguajes de programación), y en el tercer caso un día Juliano. Al establecer una fecha con el constructor se rellenan los valores de los campos de la clase, que incluyen desde el año (*year*) hasta el segundo (*second*). En el código que presentaremos en el futuro se intentará evitar el uso de objetos (la así llamada orientación a objetos) en lo posible, pero en algunos casos este estilo de programación resultará muy útil, motivo por el cual en este código se presenta la clase Java de esta manera, aprovechando la simplicidad del cálculo para intentar que el lector pueda entender este tipo de programación.

La operación más relevante tiene lugar a partir de la línea 143 (método *dateToJulianDay*), que retorna el día Juliano a partir del año, mes y día de entrada. Este método es de tipo estático, lo que significa que puede llamarse directamente sin la necesidad de crear un objeto. También es interesante observar que este método admite como entrada fechas expresadas tanto en el calendario Gregoriano (si se establece el campo *julian* a false), como en el calendario Juliano (utilizado antes de la reforma del calendario), por lo que puede utilizarse para obtener el día Juliano para cualquier fecha que se encuentre en registros

históricos. El valor devuelto se completa posteriormente a partir de la fracción de día transcurrido con la hora, minuto, y segundo, o bien llamando al método *setDayFraction* manualmente. La operación contraria se localiza a partir de la línea 53, que calcula los valores de año, mes, día, hora, minuto, y segundo para el día Juliano de entrada (*jd*). En este método el calendario civil Juliano se utiliza de manera automática cuando la fecha de salida es anterior al 5 de Octubre de 1582. Ambos métodos están basados en la obra *Astronomical Algorithms*, de Jean Meeus [2], que es una de las obras clásicas de referencia en el cálculo de efemérides.

A partir de la línea 158 encontramos un programa de ejemplo para comprobar los resultados, siguiendo la sintaxis de Java para implementar un programa que corre en la propia clase que hace los cálculos. Seguiremos esta misma dinámica en futuros códigos. El programa muestra el uso de los constructores y las llamadas a los métodos desde el lenguaje Java. La llamada al método *toString* está implementada a partir de la línea 130, y en él se hacen llamadas a los métodos *Util.fmt02(int, String)* y *Util.formatValue(double, int)*. Se trata de métodos auxiliares contenidos en el fichero *Util.java*, que el lector puede encontrar en el repositorio de código. Con ellos se formatea los campos enteros de la fecha con dos dígitos, añadiendo un cero antes del valor si es necesario, y también se formatea el segundo con un número limitado de cifras decimales. En el código que presentaremos en el futuro se harán llamadas a otros métodos comunes presentes en este fichero. En cuanto a las operaciones de ejemplo, en total hay tres cálculos de testeo: el primero obtiene el día Juliano para el instante actual (que puede comprobarse con diferentes herramientas en Internet, como la Ref. [3]), el segundo para una fecha concreta en el calendario Juliano, y el tercero es un ejemplo que debe intencionadamente devolver un error, utilizando como entrada una fecha inexistente. A raíz de la reforma Gregoriana del calendario, el día 4 de octubre de 1582 fue el último día del calendario Juliano. Tras éste, el siguiente día fue el 15 de octubre, ya en el calendario Gregoriano. Tal como se indica en los comentarios del encabezado de la clase Java, las fechas intermedias no existen oficialmente. En la práctica la adopción del calendario Gregoriano tuvo lugar en diferentes momentos después de 1582 en cada país, lo que puede complicar la interpretación de las fechas de los acontecimientos históricos, pero el programa propuesto aquí permite obtener la fecha Juliana para fechas civiles arbitrarias, incluidas aquéllas expresadas en el calendario Juliano después de 1582.

Como ejercicio dejamos al lector calcular la fecha civil correspondiente al primer día Juliano. El resultado será el año -4712, que representa el año 4713 a.C. en nuestro calendario. Esto es debido a que matemáticamente pasamos en cierto momento por el año cero, que en el calendario no existe, pero en nuestro programa sí. Otro ejercicio puede ser calcular el periodo de variabilidad de una estrella sabiendo las fechas en que su brillo se encontraba en el máximo o mínimo.

```
1 package journal;
2
3 /**
4  * A class to store date/time values with a precision of one second or better.
5  * Support to
6  * retrieve the Julian day number is given for the Gregorian and Julian calendars.
7  * <P>
8  * Selectable dates are not limited in years, but there are some invalid dates (non
9  * existent) in the civil calendar, between October 5, 1582 and October 14, 1582.
10 * <P>
11 * In the constructors and fields the year is entered without considering that year 0
12 * does
13 * not exist (astronomical year). Year 0 is 1 B.C.
14 * @version September, 2024 (first version)
15 */
16 public class JulianDay {
17     /** Instance field */
18     public int year, month, day, hour, minute;
19     /** Instance field */
```

```

19 double second;
20
21 /**
22  * Constructor for a Julian day at 0h
23  * @param y Year
24  * @param m Month
25  * @param d Day
26  */
27 public JulianDay(int y, int m, int d) {
28     year = y;
29     month = m;
30     day = d;
31     hour = 0;
32     minute = 0;
33     second = 0;
34     if (isInvalid()) throw new IllegalArgumentException("Date is invalid");
35 }
36
37 /**
38  * Constructor for a date given as a long value, representing the number of
39  * milliseconds elapsed from 1970, January 1, at 0h UTC
40  * @param t Time from 1970-1-1 in milliseconds
41  */
42 public JulianDay(long t) {
43     setFromJd(JulianDay.dateToJulianDay(1970, 1, 1, false) + t / 86400000.0);
44 }
45
46 /**
47  * Constructor for a Julian day
48  * @param jd Julian day number
49  */
50 public JulianDay(double jd) {
51     setFromJd(jd);
52 }
53
54 private void setFromJd(double jd) {
55     // The conversion formulas are from Meeus, Chapter 7
56     double z = (int) (Math.abs(jd) + 0.5);
57     if (jd < 0) z = -z;
58     double a = z;
59     if (z >= 2299161.0) { // Gregorian
60         int a2 = (int) ((z - 1867216.25) / 36524.25);
61         a += 1 + a2 - (int) (a2 / 4.0);
62     }
63     double b = a + 1524;
64     int c = (int) ((b - 122.1) / 365.25);
65     int d = (int) (c * 365.25);
66     int e = (int) ((b - d) / 30.6001);
67
68     double f = jd + 0.5 - z;
69     double exactDay = f + b - d - (int) (30.6001 * e);
70     day = (int) exactDay;
71     month = e - 1;
72     if (month > 12) month = month - 12;
73     year = c - 4715;
74     if (month > 2) year = year - 1;
75     setDayFraction(exactDay - day);
76 }
77
78 /**
79  * Returns the day fraction, from 0 to 1
80  * @return
81  */
82 public double getDayFraction() {
83     return (hour + minute / 60.0 + second / 3600.0) / 24.0;

```

```

84     }
85
86     /**
87     * Sets the day fraction (to set hour, minute, second)
88     * @param f Day fraction, from 0 to 1 (exclusive)
89     */
90     public void setDayFraction(double f) {
91         double frac = f * 24.0;
92         hour = (int) frac;
93         frac = (frac - hour) * 60.0;
94         minute = (int) frac;
95         second = (frac - minute) * 60.0;
96     }
97
98     /**
99     * Convert this instance of {@linkplain JulianDay} to a Julian day number. Dates
100    * for Gregorian or Julian calendars are handled automatically
101    * @return The Julian day that corresponds to this {@linkplain JulianDay} instance
102    */
103    public double getJulianDay() {
104        return dateToJulianDay(year, month, day, isJulian()) + getDayFraction();
105    }
106
107    /**
108    * Check if the {@linkplain JulianDay} instance contains an invalid date. A date is
109    * invalid between October 5, 1582 and October 14, 1582.
110    * @return true if the date is invalid, false otherwise.
111    */
112    public boolean isInvalid() {
113        return (year == 1582 && month == 10 && (day >= 5 && day < 15));
114    }
115
116    /**
117    * True if the instance represents a date in the Julian calendar, before October,
118    * 15, 1582
119    * @return
120    */
121    public boolean isJulian() {
122        if (year < 1582) return true;
123        if (year == 1582 && month < 10) return true;
124        if (year == 1582 && month == 10 && day < 15) return true;
125        return false;
126    }
127
128    /**
129    * Convert this {@linkplain JulianDay} instance to a String as YYYY-MM-DD
130    * hh:mm:ss.sss
131    * @return A date/time String as YYYY-MM-DD hh:mm:ss.sss
132    */
133    public String toString() {
134        return ""+year + "-" + Util.fmt02(month, "-") + Util.fmt02(day, " ") +
135            Util.fmt02(hour, ":") + Util.fmt02(minute, ":") +
136            Util.formatValue(second, 3);
137    }
138
139    /**
140    * Convert a date (0h) to a Julian Day. See Meeus, Astronomical Algorithms,
141    * chapter 7
142    * @param year The year
143    * @param month The month
144    * @param day The day
145    * @param julian true = Julian calendar, false for Gregorian. If not sure, enter
146    * false
147    * @return The Julian day for the date and calendar specified
148    */

```

```

144 public static double dateToJulianDay(int year, int month, int day, boolean julian)
145     {
146         if (month < 3) {
147             year = year - 1;
148             month = month + 12;
149         }
150         int a = year / 100;
151         int b = 0;
152         if (!julian) b = 2 - a + a / 4;
153         return (int) (365.25 * (year + 4716)) + (int) (30.6001 * (month + 1)) + day +
154             b - 1524.5;
155     }
156
157 /**
158  * Test program
159  * @param s Not used
160  */
161 public static void main(String[] s) {
162     try {
163         System.out.println("JD TEST 1: GREGORIAN (NOW)");
164         JulianDay jd = new JulianDay(System.currentTimeMillis()); // UTC
165         System.out.println("JD: " + jd.getJulianDay());
166         System.out.println("STR: " + jd.toString());
167         System.out.println("Julian? " + jd.isJulian());
168         System.out.println("Invalid? " + jd.isInvalid());
169         jd.setFromJd(jd.getJulianDay());
170         jd.setDayFraction(jd.getDayFraction());
171         System.out.println("NOW: " + jd.toString());
172
173         System.out.println();
174         System.out.println("JD TEST 2: JULIAN");
175         JulianDay jdJulian = new JulianDay(1582, 9, 15);
176         jdJulian.setDayFraction(0.25);
177         System.out.println("JD: " + jdJulian.getJulianDay());
178         System.out.println("STR: " + jdJulian.toString());
179         System.out.println("Julian? " + jdJulian.isJulian());
180         System.out.println("Invalid? " + jdJulian.isInvalid());
181         jdJulian.setFromJd(jdJulian.getJulianDay());
182         jdJulian.setDayFraction(jdJulian.getDayFraction());
183         System.out.println("NOW: " + jdJulian.toString());
184
185         System.out.println();
186         System.out.println("JD TEST 3: INVALID DATE");
187         new JulianDay(1582, 10, 10);
188
189         /*
190         JD TEST 1: GREGORIAN (NOW)
191         JD:      2460550.127662118
192         STR:    2024-08-27 15:03:50.007
193         Julian? false
194         Invalid? false
195         NOW:    2024-08-27 15:03:50.007
196
197         JD TEST 2: JULIAN
198         JD:      2299140.75
199         STR:    1582-09-15 06:00:0.000
200         Julian? true
201         Invalid? false
202         NOW:    1582-09-15 06:00:0.000
203
204         JD TEST 3: INVALID DATE
205         java.lang.IllegalArgumentException: Date is invalid
206         at journal.JulianDay.<init>(JulianDay.java:33)
207         at journal.JulianDay.main(JulianDay.java:181)
208         */
209     }
210 }

```

```
207     } catch (Exception exc) {  
208         exc.printStackTrace();  
209     }  
210 }  
211 }
```

References

[1] Páginas de Wikipedia:

https://es.wikipedia.org/wiki/Fecha_juliana

https://en.wikipedia.org/wiki/Julian_day

[2] Astronomical Algorithms, Jean Meeus (Atlantic Books, 1998).

[3] <https://apps.aavso.org/v2/tools/julian-date-converter/>