Arduino Uno and Solar Position Calculations

© David Brooks, February 2015
Institute for Earth Science Research and Education www.instesre.org

With the advent of Arduino microcontrollers and associated hardware for controlling motors, including stepper motors, there has been considerable interest in using this inexpensive open-source device to power sun trackers. There are two basic approaches to this problem. The first is to use light sensors (typically four of them) to find the sun by looking for the brightest source of light and using two stepper motors to drive a two-axis system to point toward that source.

This approach is hardware-intensive because of managing the light sensors. It is also prone to errors if there are stray sources of light and on cloudy days. A second approach takes advantage of the fact that, in principle, the position of the sun in the sky – specifically its elevation and azimuth angles – can be calculated based on well-known astronomical equations. The potential problem for the Arduino computing environment is that these equations require a lot of high-precision math. This document addresses the question of whether the Arduino Uno is up to this challenge. As it turns out, Arduino Uno math will calculate solar positions to an accuracy which is beyond that of the pointing capabilities of a reasonably priced stepper motor-driven two-axis positioning system. Implementing such a system requires an accurate longitude and latitude and correct positioning of the system – horizontal and aligned with geographic north (not magnetic north). That is, north-south is along a meridian from north pole to south pole.[1]

All the equations in this document are from Meeus' 1991 classic book, *Astronomical Algorithms*,[2] and they are numbered according to that book.

The algorithm for calculating Julian date for a specified year, month, day, hour, minute, and second is:

if month ≤ 2, $y' = year - 1$, $m' = month + 12$
otherwise $y' = year$, $m' = month$
$A = \langle floor \rangle (y'/100)$
$B = 2 + A + \langle floor \rangle (A/4)$
$Julian\ Date = \langle floor \rangle [365.25 \bullet (y' + 4716)] + \langle floor \rangle [30.6001 \bullet (m' + 1)] + day + B - 1524.5 + (hour + minute/60 + second/3600)/24$ (7.1)

where <floor> means to take the integer part of the calculation without rounding. That is, <floor>(17.7) is 17, not 18. In Arduino programming, an `(int)` type cast will do this. Since the numbers are all positive, there is no concern about what a <floor> or <int> operation will do with negative numbers.

The Julian Date is a decimal number which in principle should be expressed to the nearest second (1/86400 of a day). But it is a large number, with a value of 2457024.0 on January 1, 2015, 12:00:00 UT. The precision of floating point numbers on the Arduino Uno is no better than 7 digits, so the fractional part of a Julian Date is essentially lost in Arduino Uno floating point math.[3]

The solution is to separate the integer and fractional parts, because type `long` integers will handle values up to 2,147,483,647:

Julian Date (whole, use type `long`) = <floor>[365.25•(y′ + 4716)] + <floor>[30.6001•(m′ + 1)] + day + B - 1524
Julian Date (fraction, use type `float`) = (hour + minute/60 + second/3600)/24 – 0.5

The fact that the fractional Julian Date may be negative is OK.

---

[1] By definition, the sun's shadow falls along a north-south meridian at local solar noon (not local clock noon), which can be found using online solar position calculators. See, for example, http://www.instesre.org/Solar/insolation.htm or http://www.esrl.noaa.gov/gmd/grad/solcalc/azel.html.

[2] Jean Meeus, *Astronomical Algorithms, 1st English Edition*. Willmann-Bell, Inc. Richmond, VA, 1991. A 2nd edition is now available.

[3] The Arduino Due supports "double precision" floating point numbers, which could solve this problem, but using that hardware seems justified only if the Arduino Uno is simply not up to this task.

Now for the solar position calculations. These define the sun's position in heliocentric space relative to a pre-determined starting point and then transform the position to local Earth equatorial coordinates relative to the Greenwich meridian. For additional discussion of these quantities, see Meeus' book. Angles calculated in degrees must be converted to radians (multiply value in degrees by $\pi/180$) when they are used in trigonometric functions.

T is the number of Julian centuries (36525 days) since 12h:00m:00s Universal Time,[4] Jan 1, 2000:

$$T = (\text{Julian Date} - 2451545)/36525 \quad (24.1)$$

The suggested implementation in Arduino code is:

```
long JD_whole;
float T, JD_frac;
T = JD_whole - 2451545; T = (T + JD_frac)/36525.;
```

Solar longitude L_o (deg):

$$280.46645+36000.76983 \cdot T+0.0003032 \cdot T^2 \quad (24.2)$$

Solar mean anomaly M (deg):

$$357.5291 + 35999.0503 \cdot T - 0.0001559 \cdot T^2 - 0.00000048 \cdot T^3 \quad (24.3)$$

Eccentricity of Earth's orbit e:

$$0.016708617-0.000042037 \cdot T -0.0000001236 \cdot T^2 \quad (24.4)$$

Sun's equation of center C (deg):

$$(1.9146 - 0.004847 \cdot T - 0.000014 \cdot T^2) \cdot \sin(M) + (0.019993 - 0.000101 \cdot T) \cdot \sin(2 \cdot M) + 0.00029 \cdot \sin(3 \cdot M)\backslash \quad (p152)$$

Solar true longitude L_{true} (deg):

$$C + L_o \quad (p152)$$

Solar true anomaly f (deg):

$$M + C \quad (p152)$$

Earth distance from sun R (AU):

$$1.000001018 \cdot (1 - e^2)/[1 + e \cdot \cos(f)] \quad (24.5)$$

Greenwich hour angle (deg):

$$280.46061837 + 360.98564736629 \cdot (JD - 2451545) + 0.000387933 \cdot T^2 - T^3/38710000 \quad (11.4)$$

Obliquity of the equator (deg):

$$23 + 26/60 + 21.448/3600 - 46.815/3600 \cdot T - (0.00059/3600) \cdot T^2 + (0.001813/3600) \cdot T^3 \quad (21.2)$$

Right ascension:

$$\tan^{-1}[(\sin(L_{true}) \cdot \cos(\text{Obliquity})/\cos(L_{true})] \quad (12.3)$$

[4] Universal time is standard time at the Greenwich meridian (longitude = 0°).

Declination:

$\sin^{-1}[\sin(\text{Obliquity}) \bullet \sin(L_{true})]$ (12.4)

Hour angle:

Greenwich hour angle + longitude – right ascension

Azimuth of sun (measured eastward from north, add π)

$\tan^{-1}\{\sin(\text{Hour angle})/[\cos(\text{Hour angle}) \bullet \sin(\text{Latitude}) - \tan(\text{Declination}) \bullet \cos(\text{Latitude})]\}$ (12.5)

Elevation of sun

$\sin^{-1}[\sin(\text{Lat}) \bullet \sin(\text{Declination}) + \cos(\text{Latitude}) \bullet (\cos(\text{Declination}) \bullet \cos(\text{Hour angle})]$ (12.6)

These equations include many terms which give very small numbers, such as $0.001813/3600 \bullet T^3$ in (21.2) Can the terms with higher orders of T, which is currently a number roughly equal to 0.15, be ignored? Are there cumulative errors associated with these small terms that can cause loss of accuracy over time?

Not unexpectedly, at least for dates not in the distant future (certainly as long as T is less than 1), the results calculated without these higher-order T terms are identical in Arduino math with those that include the higher-order terms because the value of those higher order terms is essentially ignored. However, this fact does not mean that there won't be errors due to the limited precision of Arduino Uno real number math. The Greenwich hour angle calculation is a problem. For example, the Julian Date for 12:00:00 UT on June 21, 2015 is 2457195. The term $360.98564736629 \bullet (JD - 2451545) = 2039568.91$. Just the whole part of this number requires 7-digit precision, but because of Earth's rotation relative to the sun, it is the fractional part that is critical to tracking solar position during a day. Thus, this calculation will be *not* be accurate. It follows, then, that the calculation of the local hour angle will be similarly inaccurate and, following that, the calculation for azimuth angle will be inaccurate. For Arduino math, this rewrite of the equation for Greenwich hour angle will alleviate the problem:

```
long JD_whole,JDx;
float JD_frac, GrHrAngle;
…
JDx=JD_whole-2451545;
GrHrAngle=280.46061837+(360*JDx)%360+0.98564736629*(JDx)+360.98564736629*JD_frac;
```

Calculating the "big number" as a type `long` integer and applying the integer mod operator gets rid of the too-large result of performing the entire calculation using real number math; in context, the result of the integer mod operation will be converted to a floating point number less than 360.0 in the calculation.

The table below compares solar position elevation and azimuth angle calculations done in Excel and Arduino for June 21, 2015 and December 21, 2015. (The Excel calculations retain all the higher-order terms in T and the Arduino calculations do not.) Real number calculations in Excel (and other computer applications which follow the IEEE 754 specification for storing and calculating floating-point numbers, see http://support.microsoft.com/kb/78113) are done with 15-digit precision – twice that of Arduino math. But even with this reduced precision, the maximum difference between the two calculations for elevation or azimuth is less than 0.002º.

Comparison of solar position calculations from Excel and Arduino Uno.

			June 21, 2015				December 21, 2015			
			Excel		Arduino Uno		Excel		Arduino Uno	
local hour	min	sec	Elevation (deg)	Azimuth (deg)	Elevation (deg)	Azimuth (deg)	Elevation (deg)	Azimuth (deg)	Elevation (deg)	Azimuth (deg)
4	0	0	-5.7584	52.6659						
5	0	0	3.9347	62.4369	3.9350	62.4370				
6	0	0	14.4962	71.3769	14.4960	71.3770				
7	0	0	25.6184	80.0108	25.6190	80.0110	-3.8800	117.6107		
8	0	0	37.0417	89.0132	37.0420	89.0130	5.8057	127.3857	5.8060	127.3860
9	0	0	48.4861	99.4738	48.4860	99.4730	14.2220	138.4407	14.2220	138.4410
10	0	0	59.4964	113.6657	59.4960	113.6660	20.8584	151.0838	20.8580	151.0840
11	0	0	68.9263	137.1846	68.9260	137.1850	25.1369	165.2919	25.1370	165.2920
12	0	0	73.4329	178.5533	73.4330	178.5550	26.5621	180.4993	26.5620	180.4990
13	0	0	69.3859	220.8900	69.3860	220.8900	24.9412	195.6700	24.9410	195.6700
14	0	0	60.1246	245.2711	60.1240	245.2720	20.4923	209.7871	20.4920	209.7870
15	0	0	49.1647	259.8137	49.1650	259.8140	13.7227	222.3226	13.7220	222.3230
16	0	0	37.7304	270.4153	37.7300	270.4160	5.2096	233.2826	5.2090	233.2830
17	0	0	26.2972	279.4679	26.2970	279.4680	-4.5437	242.9887		
18	0	0	15.1498	288.1026	15.1500	288.1020				
19	0	0	4.5468	297.0085	4.5470	297.0090				
20	0	0	-5.2086	306.7163						

Here is the Arduino Uno code for these calculations. This code can be used for any non-commercial purpose. Please acknowledge the source if you use the code. When compiled, this code occupies 7,010 bytes of the 32,256 byte maximum on the Arduino Uno.

```
/*
This program calculates solar positions as a function of location, date, and time.
The equations are from Jean Meeus, Astronomical Algorithms, Willmann-Bell, Inc., Richmond, VA
(C) 2015, David Brooks, Institute for Earth Science Research and Education.
*/
#define DEG_TO_RAD 0.01745329
#define PI 3.141592654
#define TWOPI 6.28318531
void setup() {
  int hour,minute=0,second=0,month=6,day=21,year,zone=5;
  float Lon=-75*DEG_TO_RAD, Lat=40*DEG_TO_RAD;
  float T,JD_frac,L0,M,e,C,L_true,f,R,GrHrAngle,Obl,RA,Decl,HrAngle,elev,azimuth;
  long JD_whole,JDx;
  Serial.begin(9600);
  Serial.print("Longitude and latitude "); Serial.print(Lon/DEG_TO_RAD,3);
  Serial.print("  "); Serial.println(Lat/DEG_TO_RAD,3);
  Serial.println("year,month,day,local hour,minute,second,elevation,azimuth");
  year=2015;
  // Changes may be required in for… loop to get complete
  // daylight coverage in time zones farther west.
  for (hour=10; hour<=24; hour++) {
      JD_whole=JulianDate(year,month,day);
      JD_frac=(hour+minute/60.+second/3600.)/24.-.5;
      T=JD_whole-2451545; T=(T+JD_frac)/36525.;
      L0=DEG_TO_RAD*fmod(280.46645+36000.76983*T,360);
      M=DEG_TO_RAD*fmod(357.5291+35999.0503*T,360);
      e=0.016708617-0.000042037*T;
      C=DEG_TO_RAD*((1.9146-0.004847*T)*sin(M)+(0.019993-0.000101*T)*sin(2*M)+0.00029*sin(3*M));
      f=M+C;
```

```
      Obl=DEG_TO_RAD*(23+26/60.+21.448/3600.-46.815/3600*T);
      JDx=JD_whole-2451545;
      GrHrAngle=280.46061837+(360*JDx)%360+.98564736629*JDx+360.98564736629*JD_frac;
      GrHrAngle=fmod(GrHrAngle,360.);
      L_true=fmod(C+L0,TWOPI);
      R=1.000001018*(1-e*e)/(1+e*cos(f));
      RA=atan2(sin(L_true)*cos(Obl),cos(L_true));
      Decl=asin(sin(Obl)*sin(L_true));
      HrAngle=DEG_TO_RAD*GrHrAngle+Lon-RA;
      elev=asin(sin(Lat)*sin(Decl)+cos(Lat)*(cos(Decl)*cos(HrAngle)));
      // Azimuth measured eastward from north.
      azimuth=PI+atan2(sin(HrAngle),cos(HrAngle)*sin(Lat)-tan(Decl)*cos(Lat));
      Serial.print(year); Serial.print(","); Serial.print(month);
      Serial.print(","); Serial.print(day); Serial.print(", ");
      Serial.print(hour-zone); Serial.print(",");
      Serial.print(minute); Serial.print(","); Serial.print(second);
      // (Optional) display results of intermediate calculations.
      //Serial.print(","); Serial.print(JD_whole);
      //Serial.print(","); Serial.print(JD_frac,7);
      //Serial.print(","); Serial.print(T,7);
      //Serial.print(","); Serial.print(L0,7);
      //Serial.print(","); Serial.print(M,7);
      //Serial.print(","); Serial.print(e,7);
      //Serial.print(","); Serial.print(C,7);
      //Serial.print(","); Serial.print(L_true,7);
      //Serial.print(","); Serial.print(f,7);
      //Serial.print(","); Serial.print(R,7);
      //Serial.print(","); Serial.print(GrHrAngle,7);
      //Serial.print(","); Serial.print(Obl,7);
      //Serial.print(","); Serial.print(RA,7);
      //Serial.print(","); Serial.print(Decl,7);
      //Serial.print(","); Serial.print(HrAngle,7);
      Serial.print(","); Serial.print(elev/DEG_TO_RAD,3);
      Serial.print(","); Serial.print(azimuth/DEG_TO_RAD,3); Serial.println();
  }
}
void loop() {}
long JulianDate(int year, int month, int day) {
  long JD_whole;
  int A,B;
  if (month<=2) {
    year--; month+=12;
  }
  A=year/100; B=2-A+A/4;
  JD_whole=(long)(365.25*(year+4716))+(int)(30.6001*(month+1))+day+B-1524;
  return JD_whole;
}
```