1. Sunrise/Sunset Library. These functions have been converted from Javascript to C code by Bret Whissel. The original Javascript code was taken from the NOAA Solar Calculator webpage located at the URL `http://www.esrl.noaa.gov/gmd/grad/solcalc/`. We'll start by defining a few useful constants.

```
#define TRUE (1 ≡ 1)
#define FALSE (¬TRUE)
```

2. The original Javascript provided a few more functions which may be useful someday, but they are not needed to calculate sunrise/sunset times. Their definitions are preserved here, but they are not compiled unless `COMPLETE` evaluates to `TRUE`.

```
#define COMPLETE FALSE
  ⟨ Includes 3 ⟩
  ⟨ Functions 4 ⟩
```

3. ⟨ Includes 3 ⟩ ≡

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
```

This code is used in section 2.

4. ⟨ Functions 4 ⟩ ≡

```
  static double calcTimeJulianCent(double jd)
  {
    return (jd − 2451545.0)/36525.0;
  }
```

See also sections 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, and 33.

This code is used in section 2.

5. ⟨ Functions 4 ⟩ +≡

```
#if COMPLETE
  static double calcJDFromJulianCent(double t)
  {
    return t ∗ 36525.0 + 2451545.0;
  }
#endif
```

6. ⟨ Functions 4 ⟩ +≡

```
#if COMPLETE
  static int isLeapYear(int yr)
  {
    return ((yr % 4 ≡ 0 ∧ yr % 100 ≠ 0) ∨ yr % 400 ≡ 0);
  }
#endif
```

7. ⟨ Functions 4 ⟩ +≡

```
#if COMPLETE
  static double calcDoyFromJD(double jd)
  {
    double f, day;
    int A, B, C, D, E, alpha, doy, z;
    int month, year, k;
    z ← floor(jd + 0.5);
    f ← (jd + 0.5) − z;
    if (z < 2299161) A ← z;
    else {
      alpha ← floor((z − 1867216.25)/36524.25);
      A ← z + 1 + alpha − alpha/4;
    }
    B ← A + 1524;
    C ← floor((B − 122.1)/365.25);
    D ← floor(365.25 ∗ C);
    E ← floor((B − D)/30.6001);
    day ← B − D − floor(30.6001 ∗ E) + f;
    month ← (E < 14) ? E − 1 : E − 13;
    year ← (month > 2) ? C − 4716 : C − 4715;
    k ← (isLeapYear(year) ? 1 : 2);
    doy ← ((275 ∗ month)/9) − k ∗ ((month + 9)/12) − 30;
    return doy + day;
  }
#endif
```

8. ⟨ Functions 4 ⟩ +≡

```
  static double radToDeg(double angleRad)
  {
    return (180.0 ∗ angleRad/M_PI);
  }
```

9. ⟨ Functions 4 ⟩ +≡

```
  static double degToRad(double angleDeg)
  {
    return (M_PI ∗ angleDeg/180.0);
  }
```

10. ⟨ Functions 4 ⟩ +≡

```
  static double calcGeomMeanLongSun(double t)
  {
    double L0 ← 280.46646 + t ∗ (36000.76983 + t ∗ (0.0003032));
    while (L0 > 360.0) L0 −= 360.0;
    while (L0 < 0.0) L0 += 360.0;
    return L0;    /∗ in degrees ∗/
  }
```

11. ⟨ Functions 4 ⟩ +≡

```
static double calcGeomMeanAnomalySun(double t)
{
  double M ← 357.52911 + t * (35999.05029 − 0.0001537 * t);
  return M;    /* in degrees */
}
```

12. ⟨ Functions 4 ⟩ +≡

```
static double calcEccentricityEarthOrbit(double t)
{
  double e ← 0.016708634 − t * (0.000042037 + 0.0000001267 * t);
  return e;    /* unitless */
}
```

13. ⟨ Functions 4 ⟩ +≡

```
static double calcSunEqOfCenter(double t)
{
  double m, mrad, sinm, sin2m, sin3m, C;
  m ← calcGeomMeanAnomalySun(t);
  mrad ← degToRad(m);
  sinm ← sin(mrad);
  sin2m ← sin(mrad + mrad);
  sin3m ← sin(mrad + mrad + mrad);
  C ← sinm * (1.914602 − t * (0.004817 + 0.000014 * t)) + sin2m * (0.019993 − 0.000101 * t) + sin3m * 0.000289;
  return C;    /* in degrees */
}
```

14. ⟨ Functions 4 ⟩ +≡

```
static double calcSunTrueLong(double t)
{
  double L0, c, Ob;
  L0 ← calcGeomMeanLongSun(t);
  c ← calcSunEqOfCenter(t);
  Ob ← L0 + c;
  return Ob;    /* in degrees */
}
```

15. ⟨ Functions 4 ⟩ +≡

```
#if COMPLETE
  static double calcSunTrueAnomaly(double t)
  {
    double m, c, v;
    m ← calcGeomMeanAnomalySun(t);
    c ← calcSunEqOfCenter(t);
    v ← m + c;
    return v;    /* in degrees */
  }
#endif
```

16. ⟨ Functions 4 ⟩ +≡

```
#if COMPLETE
  static double calcSunRadVector(double t)
  {
    double v, e, R;
    v ← calcSunTrueAnomaly(t);
    e ← calcEccentricityEarthOrbit(t);
    R ← (1.000001018 * (1 − e * e))/(1 + e * cos(degToRad(v)));
    return R;    /* in AUs */
  }
#endif
```

17. ⟨ Functions 4 ⟩ +≡

```
  static double calcSunApparentLong(double t)
  {
    double o, omega, lambda;
    o ← calcSunTrueLong(t);
    omega ← 125.04 − 1934.136 * t;
    lambda ← o − 0.00569 − 0.00478 * sin(degToRad(omega));
    return lambda;    /* in degrees */
  }
```

18. ⟨ Functions 4 ⟩ +≡

```
  static double calcMeanObliquityOfEcliptic(double t)
  {
    double seconds, e0;
    seconds ← 21.448 − t * (46.8150 + t * (0.00059 − t * (0.001813)));
    e0 ← 23.0 + (26.0 + (seconds/60.0))/60.0;
    return e0;    /* in degrees */
  }
```

19. ⟨ Functions 4 ⟩ +≡

```
  static double calcObliquityCorrection(double t)
  {
    double e0, omega, e;
    e0 ← calcMeanObliquityOfEcliptic(t);
    omega ← 125.04 − 1934.136 * t;
    e ← e0 + 0.00256 * cos(degToRad(omega));
    return e;    /* in degrees */
  }
```

20. ⟨ Functions 4 ⟩ +≡

```
#if COMPLETE
  static double calcSunRtAscension(double t)
  {
    double e, lambda, tana_num, tana_denom, alpha;
    e ← calcObliquityCorrection(t);
    lambda ← calcSunApparentLong(t);
    tana_num ← (cos(degToRad(e)) * sin(degToRad(lambda)));
    tana_denom ← (cos(degToRad(lambda)));
    alpha ← radToDeg(atan2(tana_num, tana_denom));
    return alpha;    /* in degrees */
  }
#endif
```

21. ⟨ Functions 4 ⟩ +≡

```
  static double calcSunDeclination(double t)
  {
    double e, lambda, sint, theta;
    e ← calcObliquityCorrection(t);
    lambda ← calcSunApparentLong(t);
    sint ← sin(degToRad(e)) * sin(degToRad(lambda));
    theta ← radToDeg(asin(sint));
    return theta;    /* in degrees */
  }
```

22. ⟨ Functions 4 ⟩ +≡

```
  static double calcEquationOfTime(double t)
  {
    double epsilon, l0, e, m, y, sin2l0, sinm, cos2l0, sin4l0, sin2m, Etime;
    epsilon ← calcObliquityCorrection(t);
    l0 ← calcGeomMeanLongSun(t);
    e ← calcEccentricityEarthOrbit(t);
    m ← calcGeomMeanAnomalySun(t);
    y ← tan(degToRad(epsilon)/2.0);
    y *= y;
    sin2l0 ← sin(2.0 * degToRad(l0));
    sinm ← sin(degToRad(m));
    cos2l0 ← cos(2.0 * degToRad(l0));
    sin4l0 ← sin(4.0 * degToRad(l0));
    sin2m ← sin(2.0 * degToRad(m));
    Etime ← y * sin2l0 − 2.0 * e * sinm + 4.0 * e * y * sinm * cos2l0 − 0.5 * y * y * sin4l0 − 1.25 * e * e * sin2m;
    return radToDeg(Etime) * 4.0;    /* in minutes of time */
  }
```

23. ⟨ Functions 4 ⟩ +≡

```
static double calcHourAngleSunrise(double lat, double solarDec)
{
  double latRad, sdRad, HAarg, HA;
  latRad ← degToRad(lat);
  sdRad ← degToRad(solarDec);
  HAarg ← (cos(degToRad(90.833))/(cos(latRad) * cos(sdRad)) − tan(latRad) * tan(sdRad));
  HA ← acos(HAarg);
  return HA;    /* in radians (for sunset, use -HA) */
}
```

24. This function has not yet been converted from Javascript to C.

```
⟨ Functions 4 ⟩ +≡
#if COMPLETE
  double calcAzEl(output, T, localtime, latitude, longitude,
        zone){ var eqTime ← calcEquationOfTime(T) var theta ← calcSunDeclination(T)
    if (output) {document.getElementById("eqtbox").value ← Math.floor(eqTime * 100 +
        0.5)/100.0 document.getElementById("sdbox").value ← Math.floor(theta * 100 + 0.5)/100.0}
    var solarTimeFix ← eqTime + 4.0 * longitude − 60.0 * zone var earthRadVec ←
        calcSunRadVector(T) var trueSolarTime ← localtime + solarTimeFix
    while (trueSolarTime > 1440) {trueSolarTime −= 1440}
    var hourAngle ← trueSolarTime/4.0 − 180.0;
    if (hourAngle < −180) {hourAngle += 360.0}
    var haRad ← degToRad(hourAngle) var csz ← Math.sin(degToRad(latitude)) *
        Math.sin(degToRad(theta)) + Math.cos(degToRad(latitude)) * Math.cos(degToRad(theta)) *
        Math.cos(haRad)
    if (csz > 1.0) {csz ← 1.0}
    else if (csz < −1.0) {csz ← −1.0}
    var zenith ← radToDeg(Math.acos(csz)) var azDenom ← (Math.cos(degToRad(latitude)) *
        Math.sin(degToRad(zenith))) if (Math.abs(azDenom) > 0.001) {
        azRad ← ((Math.sin(degToRad(latitude)) * Math.cos(degToRad(zenith))) −
        Math.sin(degToRad(theta)))/azDenom
    if (Math.abs(azRad) > 1.0) {
      if (azRad < 0) {azRad ← −1.0}
      else {azRad ← 1.0}
    }
    var azimuth ← 180.0 − radToDeg(Math.acos(azRad))
    if (hourAngle > 0.0) {azimuth ← −azimuth}
    }
    else {
      if (latitude > 0.0) {azimuth ← 180.0}
      else {azimuth ← 0.0}
    }
    if (azimuth < 0.0) {azimuth += 360.0}
    var exoatmElevation ← 90.0 − zenith      /* Atmospheric Refraction correction */
    if (exoatmElevation > 85.0) {
      var refractionCorrection ← 0.0;
    }
    else {
      var te ← Math.tan(degToRad(exoatmElevation));
      if (exoatmElevation > 5.0) {
        var refractionCorrection ← 58.1/te − 0.07/(te * te * te) + 0.000086/(te * te * te * te * te);
      }
      else if (exoatmElevation > −0.575) {
        var refractionCorrection ← 1735.0 + exoatmElevation * (−518.2 + exoatmElevation * (103.4 +
            exoatmElevation * (−12.79 + exoatmElevation * 0.711)));
      }
      else {
        var refractionCorrection ← −20.774/te;
      }
      refractionCorrection ← refractionCorrection/3600.0;
    }
    var solarZen ← zenith − refractionCorrection;
```

```
    if ((output) ∧ (solarZen > 108.0))
    {document.getElementById("azbox").value ← "dark" document.getElementById("elbox").value ←
        "dark"}
    else if (output) { document.getElementById("azbox").value ←
        Math.floor(azimuth * 100 + 0.5)/100.0 document.getElementById("elbox").value ←
        Math.floor((90.0 − solarZen) * 100 + 0.5)/100.0
    if (document.getElementById("showae").checked) {showLineGeodesic("#ffff00", azimuth)}
    } return (azimuth) }
#endif
```

25. This function has not yet been converted from Javascript to C.

```
⟨Functions 4⟩ +≡
#if COMPLETE
  double calcSolNoon(jd,
        longitude, timezone, dst){ var tnoon ← calcTimeJulianCent(jd − longitude/360.0) var eqTime ←
        calcEquationOfTime(tnoon) var solNoonOffset ← 720.0 − (longitude * 4) − eqTime
      /* in minutes */
    var newt ← calcTimeJulianCent(jd + solNoonOffset/1440.0) eqTime ←
        calcEquationOfTime(newt) solNoonLocal ← 720 − (longitude * 4) − eqTime + (timezone * 60.0)
      /* in minutes */
    if (dst) solNoonLocal += 60.0
    while (solNoonLocal < 0.0) {
      solNoonLocal += 1440.0;
    }
    while (solNoonLocal ≥ 1440.0) {
      solNoonLocal −= 1440.0;
    }
    document.getElementById("noonbox").value ← timeString(solNoonLocal, 3) }
#endif
```

26. ⟨Functions 4⟩ +≡

```
  static double calcSunriseSetUTC(int rise, double JD, double latitude, double longitude)
  {
    double t, eqTime, solarDec, hourAngle, delta, timeUTC;
    t ← calcTimeJulianCent(JD);
    eqTime ← calcEquationOfTime(t);
    solarDec ← calcSunDeclination(t);
    hourAngle ← calcHourAngleSunrise(latitude, solarDec);
    if (¬rise) hourAngle ← −hourAngle;
    delta ← longitude + radToDeg(hourAngle);
    timeUTC ← 720 − (4.0 * delta) − eqTime;      /* in minutes */
    return timeUTC;
  }
```

27. Convert a date at midnight UTC (start of day) *month*, *day*, *year* to a Julian date.

```
⟨ Functions 4 ⟩ +≡
  double mdyToJulianDate(int month, int day, int year)
  {
    int A, B;
    double jd;
    if (month ≤ 2) {
      year −= 1;
      month += 12;
    }
    A ← year/100;
    B ← 2 − A + A/4;
    jd ← floor(365.25 ∗ (year + 4716)) + floor(30.6001 ∗ (month + 1)) + day + B − 1524.5;
    return jd;
  }
```

28. Convert a UNIX timestamp to a Julian date. A Julian day is exactly 86400 seconds long, and the start of the UNIX epoch (1 Jan 1970 00:00 UTC) has the Julian date 2440587.5.

```
#define DAYSECS ((double) 86400.0)
#define UNIX_EPOCH ((double) 2440587.5)
⟨ Functions 4 ⟩ +≡
  double unixToJulianDate(time_t ∗t)
  {
    return ((double)(∗t)/DAYSECS) + UNIX_EPOCH;
  }
```

29. Given a day specified by UNIX timestamp *thetime*, determine the calendar date for the timezone specified by *tz*. Then return the UNIX time of midnight UTC for the given date.

```
⟨ Functions 4 ⟩ +≡
  static time_t daystart(time_t ∗thetime, char ∗tz)
  {
    char ∗oldzone;
    time_t daystartUTC;
    struct tm theday;
    ⟨ Get date from the current UNIX time 30 ⟩
    ⟨ Get the GMT UNIX time for the start of today 31 ⟩
    ⟨ Restore the original timezone 32 ⟩
    return daystartUTC;
  }
```

30. To convert the UNIX time to the current date for the given timezone, we must reset the `TZ` environment variable to the specified timezone. We save the current local timezone first.

```
⟨ Get date from the current UNIX time 30 ⟩ ≡
  oldzone ← getenv("TZ");
  setenv("TZ", tz, 1);
  tzset();
  localtime_r(thetime, &theday);
```

This code is used in section 29.

31. Now that we have the current date for the specified timezone, we want to retrieve the UNIX time for midnight UTC of the current date (start of the day). Again, we need to reset the timezone to UTC by setting the `TZ` environment variable to the empty string. So that we're completely unambiguous, we set the time at 1 second past midnight and remove the extra second after the conversion back to UNIX time.

⟨ Get the GMT UNIX time for the start of today 31 ⟩ ≡

```
theday.tm_hour ← 0;
theday.tm_min ← 0;
theday.tm_sec ← 1;
setenv("TZ", "", 1);
tzset();
daystartUTC ← mktime(&theday) − 1;
```

This code is used in section 29.

32. We're done mucking about with timezones, so we'll restore the original `TZ` environment value.

⟨ Restore the original timezone 32 ⟩ ≡

```
if (oldzone) setenv("TZ", oldzone, 1);
else unsetenv("TZ");
tzset();
```

This code is used in section 29.

33. Calculate the sunrise and sunset time for the current day (in the local timezone), latitude, and longitude. Return the UNIX time of these events in *rise* and *set*. The function *calcSunriseSetUTC*( ) returns the time of the event in minutes relative to the start of the current day at UTC.

We start by converting the current UNIX time to a date in the local timezone. Then we calculate the UNIX time of midnight UTC for the current date. Then we calculate the sunrise/sunset times for the given latitude and longitude. Once an initial estimate of an event is calculated, the original Javascript recalculates the event for the Julian date of the initial estimate. This function duplicates that functionality. When the final calculation is made, the output is rounded to the nearest minute, converted from minutes to seconds from midnight UTC, and then added to *daystartUTC* to form the UNIX time for the event.

⟨ Functions 4 ⟩ +≡

```
void calcSRSS(time_t *now, double lat, double lon, char *tz, time_t *rise, time_t *set)
{
  time_t daystartUTC;
  double julianday, juladjust;

  daystartUTC ← daystart(now, tz);
  julianday ← unixToJulianDate(&daystartUTC);
  juladjust ← calcSunriseSetUTC(TRUE, julianday, lat, lon)/1440.0;
  *rise ← daystartUTC + 60 * (int)(calcSunriseSetUTC(TRUE, julianday + juladjust, lat, lon) + 0.5);
  juladjust ← calcSunriseSetUTC(FALSE, julianday, lat, lon)/1440.0;
  *set ← daystartUTC + 60 * (int)(calcSunriseSetUTC(FALSE, julianday + juladjust, lat, lon) + 0.5);
}
```

34. Testing. A short routine to verify that conversions work as expected. The output should be compared with the original NOAA webpage for verification that the conversions have been performed properly.

⟨ srsstest.c 34 ⟩ ≡

```
#include <stdio.h>
#include <time.h>
#include <math.h>
#include "srss.h"
  int main(int argc, char **argv)
  {
    time_t rise, set, day, now ← time(0);
    double lat ← 30.397, lon ← −84.350;
    struct tm today;
    int i;
    for (i ← −3; i ≤ 3; i++) {
      day ← now + i * 86400;
      calcSRSS(&day, lat, lon, "America/New_York", &rise, &set);
      localtime_r(&rise, &today);
      printf("%02d/%02d/%d␣rise␣%02d:%02d,␣", today.tm_mon + 1, today.tm_mday,
          today.tm_year + 1900, today.tm_hour, today.tm_min);
      localtime_r(&set, &today);
      printf("set␣%02d:%02d\n", today.tm_hour, today.tm_min);
    }
    return 0;
  }
```

35. We'll need to share a little information with external files who wish to use our routines. The following three function prototypes are the only publicly-accessible functions.

⟨ srss.h 35 ⟩ ≡

```
#ifndef SRSS_H
  void calcSRSS(time_t *, double, double, char *, time_t *, time_t *);
  double unixToJulianDate(time_t *);
  double mdyToJulianDate(int, int, int);
#define SRSS_H 1
#endif
```

36. Index.

⟨ Functions 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 33 ⟩ Used in section 2.
⟨ Get date from the current UNIX time 30 ⟩ Used in section 29.
⟨ Get the GMT UNIX time for the start of today 31 ⟩ Used in section 29.
⟨ Includes 3 ⟩ Used in section 2.
⟨ Restore the original timezone 32 ⟩ Used in section 29.
⟨ `srss.h` 35 ⟩
⟨ `srsstest.c` 34 ⟩

SUNRISE-SUNSET