Calc_Sun code

Project home: https://github.com/DouglasAllen/calc_sun

C extension

```
#include <ruby.h>

#ifndef DBL2NUM
# define DBL2NUM(dbl) rb_float_new(dbl)
#endif

#define R2D 57.295779513082320876798154814105
#define D2R 0.017453292519943295769236907684886
#define M2PI M_PI * 2.0
#define INV24 1.0 / 24.0
#define INV360 1.0 / 360.0
#define J2000 2451545.0

// static ID id_status;

static VALUE
t_init(VALUE self)
{
  return self;
}

static VALUE
func_rev12(VALUE self, VALUE vx)
{
  double x = NUM2DBL(vx);
  return DBL2NUM(x - 24.0 * floor(x * INV24 + 0.5));
}

static VALUE
func_mean_anomally(VALUE self, VALUE vd)
{
  double d = NUM2DBL(vd);
  double vma =
  fmod(
    (357.5291  +
     0.98560028 * d
    ) * D2R, M2PI);
```

```
  return DBL2NUM(vma);
}

static VALUE
func_eccentricity(VALUE self, VALUE vd)
{
  double d = NUM2DBL(vd);
  double ve =
  0.016709 -
  1.151e-9 * d;
  return DBL2NUM(ve);
}

static VALUE
func_equation_of_center(VALUE self, VALUE vd)
{
  double vma =
  NUM2DBL(func_mean_anomally(self, vd));
  double ve =
  NUM2DBL(func_eccentricity(self, vd));
  double ve2 = ve * 2.0;
  double vesqr = ve * ve;
  double vesqr54 = 5.0 / 4.0 * vesqr;
  double vecube12 = (vesqr * ve) / 12.0;
  double veoc =
  ve2 * sin(vma) +
  vesqr54 * sin(2 * vma) +
  vecube12 * (13.0 * sin(3 * vma) - 3.0 * sin(vma));
  return DBL2NUM(veoc);
}

static VALUE
func_true_anomally(VALUE self, VALUE vd)
{
  double vma =
  NUM2DBL(func_mean_anomally(self, vd));
  double veoc =
  NUM2DBL(func_equation_of_center(self, vd));
  double vta = vma + veoc;
  return DBL2NUM(vta);
}

static VALUE
func_mean_longitude(VALUE self, VALUE vd)
{
```

```
  double d = NUM2DBL(vd);
  double vml =
  fmod(
    (280.4664567 +
     0.9856473601037645 * d
    ) * D2R, M2PI);
  return DBL2NUM(vml);
}

static VALUE
func_eccentric_anomally(VALUE self, VALUE vd)
{
  double ve =
  NUM2DBL(func_eccentricity(self, vd));
  double vml =
  NUM2DBL(func_mean_longitude(self, vd));
  double vea =
  vml + ve * sin(vml) * (1.0 + ve * cos(vml));
  return DBL2NUM(vea);
}

static VALUE
func_obliquity_of_ecliptic(VALUE self, VALUE vd)
{
  double d = NUM2DBL(vd);
  double vooe =
  (23.439291 - 3.563E-7 * d) * D2R;
  return DBL2NUM(vooe);
}

static VALUE
func_longitude_of_perihelion(VALUE self, VALUE vd)
{
  double d = NUM2DBL(vd);
  double vlop =
  fmod(
    (282.9404 +
     4.70935e-05 * d
    ) * D2R, M2PI);
  return DBL2NUM(vlop);
}

static VALUE
func_xv(VALUE self, VALUE vd)
{
```

```
  double vea =
  NUM2DBL(func_eccentric_anomally(self, vd));
  double ve =
  NUM2DBL(func_eccentricity(self, vd));
  double vxv = cos(vea) - ve;
  return DBL2NUM(vxv);
}

static VALUE
func_yv(VALUE self, VALUE vd)
{
  double vea =
  NUM2DBL(func_eccentric_anomally(self, vd));
  double ve =
  NUM2DBL(func_eccentricity(self, vd));
  double vyv =
  sqrt(1.0 - ve * ve) * sin(vea);
  return DBL2NUM(vyv);
}

static VALUE
func_true_longitude(VALUE self, VALUE vd)
{
  double vta =
  NUM2DBL(func_true_anomally(self, vd));
  double vlop =
  NUM2DBL(func_longitude_of_perihelion(self, vd));
  double vtl =
  fmod(vta + vlop, M2PI);
  return DBL2NUM(vtl);
}

static VALUE
func_rv(VALUE self, VALUE vd)
{
  double vxv =
  NUM2DBL(func_xv(self, vd));
  double vyv =
  NUM2DBL(func_yv(self, vd));
  double vrv =
  sqrt(vxv * vxv + vyv * vyv);
  return DBL2NUM(vrv);
}

static VALUE
```

```
func_ecliptic_x(VALUE self, VALUE vd)
{
  double vrv =
  NUM2DBL(func_rv(self, vd));
  double vtl =
  NUM2DBL(func_true_longitude(self, vd));
  double vex = vrv * cos(vtl);
  return DBL2NUM(vex);
}

static VALUE
func_ecliptic_y(VALUE self, VALUE vd)
{
  double vrv =
  NUM2DBL(func_rv(self, vd));
  double vtl =
  NUM2DBL(func_true_longitude(self, vd));
  double vey = vrv * sin(vtl);
  return DBL2NUM(vey);
}

static VALUE
func_right_ascension(VALUE self, VALUE vd)
{
  double vey =
  NUM2DBL(func_ecliptic_y(self, vd));
  double vooe =
  NUM2DBL(func_obliquity_of_ecliptic(self, vd));
  double vex =
  NUM2DBL(func_ecliptic_x(self, vd));
  double vra =
  fmod(atan2(vey * cos(vooe), vex) + M2PI, M2PI);
  return DBL2NUM(vra * R2D / 15.0);
}

static VALUE
func_declination(VALUE self, VALUE vd)
{
  double vex =
  NUM2DBL(func_ecliptic_x(self, vd));
  double vey =
  NUM2DBL(func_ecliptic_y(self, vd));
  double vooe =
  NUM2DBL(func_obliquity_of_ecliptic(self, vd));
  double ver = sqrt(vex * vex + vey * vey);
```

```
  double vz = vey * sin(vooe);
  double vdec = atan2(vz, ver);
  return DBL2NUM(vdec);
}

static VALUE
func_sidetime(VALUE self, VALUE vjd)
{
  double vd = NUM2DBL(vjd);
  double vst =
  fmod(
    (180 + 357.52911 + 282.9404) +
    (0.985600281725 + 4.70935E-5) * vd, 360.0);
  // printf("%f \n", vd);
  return DBL2NUM(vst / 15.0);
}

static VALUE
func_local_sidetime(VALUE self, VALUE vjd, VALUE vlon)
{
  double vst = NUM2DBL(func_sidetime(self, vjd));
  double vlst = NUM2DBL(vlon) / 15.0 + 12.0 + vst;
  return DBL2NUM(fmod(vlst, 24.0));
}

static VALUE
func_dlt(VALUE self, VALUE vd, VALUE vlat)
{
  double vsin_alt = sin(-0.8333 * D2R);
  double vlat_r = NUM2DBL(vlat) * D2R;
  double vcos_lat = cos(vlat_r);
  double vsin_lat = sin(vlat_r);
  double vooe =
  NUM2DBL(func_obliquity_of_ecliptic(self, vd));
  double vtl =
  NUM2DBL(func_true_longitude(self, vd));
  double vsin_dec = sin(vooe) * sin(vtl);
  double vcos_dec =
  sqrt( 1.0 - vsin_dec * vsin_dec );
  double vdl =
  acos(
    (vsin_alt - vsin_dec * vsin_lat) /
    (vcos_dec * vcos_lat));
  double vdla = vdl * R2D;
  double vdlt = vdla / 15.0 * 2.0;
```

```
  return DBL2NUM(vdlt);
}

static VALUE
func_diurnal_arc(VALUE self, VALUE vjd, VALUE vlat)
{
  double dlt = NUM2DBL(func_dlt(self, vjd, vlat));
  double da = dlt / 2.0;
  return DBL2NUM(da);
}

static VALUE
func_t_south(VALUE self, VALUE vjd, VALUE vlon)
{
  double lst = NUM2DBL(func_local_sidetime(self, vjd, vlon));
  double ra = NUM2DBL(func_right_ascension(self, vjd));
  double vx = lst - ra;
  double vt = vx - 24.0 * floor(vx * INV24 + 0.5);
  return DBL2NUM(12 - vt);
}

static VALUE
func_t_rise(VALUE self, VALUE vjd, VALUE vlon, VALUE vlat)
{
  double ts = NUM2DBL(func_t_south(self, vjd, vlon));
  double da = NUM2DBL(func_diurnal_arc(self, vjd, vlat));
  return DBL2NUM(ts - da);
}

static VALUE
func_t_set(VALUE self, VALUE vjd, VALUE vlon, VALUE vlat)
{
  double ts = NUM2DBL(func_t_south(self, vjd, vlon));
  double da = NUM2DBL(func_diurnal_arc(self, vjd, vlat));
  return DBL2NUM(ts + da);
}

void Init_calc_sun(void)
{
  VALUE cCalcSun =
  rb_define_class("CalcSun", rb_cObject);
  rb_define_method(cCalcSun, "initialize", t_init, 0);
  rb_define_method(cCalcSun,
  "reverse_12", func_rev12, 1);
  rb_define_method(cCalcSun,
```

```
"mean_anomally", func_mean_anomally, 1);
rb_define_method(cCalcSun,
"eccentricity", func_eccentricity, 1);
rb_define_method(cCalcSun,
"equation_of_center", func_equation_of_center, 1);
rb_define_method(cCalcSun,
"true_anomally", func_true_anomally, 1);
rb_define_method(cCalcSun,
"mean_longitude", func_mean_longitude, 1);
rb_define_method(cCalcSun,
"eccentric_anomally", func_eccentric_anomally, 1);
rb_define_method(cCalcSun,
"obliquity_of_ecliptic", func_obliquity_of_ecliptic, 1);
rb_define_method(cCalcSun,
"longitude_of_perihelion", func_longitude_of_perihelion, 1);
rb_define_method(cCalcSun,
"xv", func_xv, 1);
rb_define_method(cCalcSun,
"yv", func_yv, 1);
rb_define_method(cCalcSun,
"true_longitude", func_true_longitude, 1);
rb_define_method(cCalcSun,
"rv", func_rv, 1);
rb_define_method(cCalcSun,
"ecliptic_x", func_ecliptic_x, 1);
rb_define_method(cCalcSun,
"ecliptic_y", func_ecliptic_y, 1);
rb_define_method(cCalcSun,
"right_ascension", func_right_ascension, 1);
rb_define_method(cCalcSun,
"declination", func_declination, 1);
rb_define_method(cCalcSun,
"sidereal_time", func_sidetime, 1);
rb_define_method(cCalcSun,
"local_sidereal_time", func_local_sidetime, 2);
rb_define_method(cCalcSun,
"dlt", func_dlt, 2);
rb_define_method(cCalcSun,
"diurnal_arc", func_diurnal_arc, 2);
rb_define_method(cCalcSun,
"t_south", func_t_south, 2);
rb_define_method(cCalcSun,
"t_rise", func_t_rise, 3);
rb_define_method(cCalcSun,
"t_set", func_t_set, 3);
```

```
}
```

Example usage code

```
lib = File.expand_path('../lib', __FILE__)
$LOAD_PATH.unshift(lib) unless $LOAD_PATH.include?(lib)
require 'calc_sun'
cs = CalcSun.new

require 'date'
J2000 = DateTime.parse('2000-01-01T12:00:00').jd

lat = 41.95
lon = -88.75
day = Date.parse('2016-12-25')
jd = day.jd - J2000 - lon / 360.0
rise = cs.t_rise(jd, lon, lat)
set = cs.t_set(jd, lon, lat)

printf("\n")

printf("\tSun rises \t\t\t : %2.0f:%02.0f UTC\n",
       rise.floor, (rise % 1 * 60.0).floor)

printf("\tSun mid day \t\t\t : %2.0f:%02.0f UTC\n",
       ((rise + set) / 2.0).floor,
       (((rise + set) / 2.0 % 1.0) * 60).floor)

printf("\tSun sets \t\t\t : %2.0f:%02.0f UTC\n",
       set.floor, (set % 1 * 60.0).floor)
```

Test code

```
require 'rubygems'
gem 'minitest'
require 'minitest/autorun'
# require 'test/unit'
lib = File.expand_path('../../lib', __FILE__)
$LOAD_PATH.unshift(lib) unless $LOAD_PATH.include?(lib)
require 'calc_sun'
require 'date'
#
class TestCalcSun < MiniTest::Test
  # class TestCalcSun < Test::Unit::TestCase
```

```
def setup
  @t = CalcSun.new
  @t_ajd = 0.0
  @t_lat = 0.0
  @t_lon = 0.0
end

def test_mean_anomally
  assert_equal(
    6.240059966692,
    @t.mean_anomally(@t_ajd).round(12)
  )
end

def test_eccentricity
  assert_equal(
    0.016709,
    @t.eccentricity(@t_ajd)
  )
end

def test_equation_of_center
  assert_equal(
    -0.001471380867,
    @t.equation_of_center(@t_ajd).round(12)
  )
end

def test_true_anomally
  assert_equal(
    6.238588585825,
    @t.true_anomally(@t_ajd).round(12)
  )
end

def test_mean_longitude
  assert_equal(
    4.895063110817,
    @t.mean_longitude(@t_ajd).round(12)
  )
end

def test_eccentric_anomaly
  assert_equal(
```

```
      4.878582250862,
      @t.eccentric_anomally(@t_ajd).round(12)
    )
  end

  def test_obliquity_of_ecliptic
    assert_equal(
      0.409092802283,
      @t.obliquity_of_ecliptic(@t_ajd).round(12)
    )
  end

  def test_xv
    assert_equal(
      0.148720277673,
      @t.xv(@t_ajd).round(12)
    )
  end

  def test_yv
    assert_equal(
      -0.986083974099,
      @t.yv(@t_ajd).round(12)
    )
  end

  def test_longitude_of_perihelion
    assert_equal(
      4.93824156691,
      @t.longitude_of_perihelion(@t_ajd).round(12)
    )
  end

  def test_true_longitude
    assert_equal(
      4.893644845555,
      @t.true_longitude(@t_ajd).round(12)
    )
  end

  def test_rv
    assert_equal(
      0.997235842199,
      @t.rv(@t_ajd).round(12)
    )
```

```
  end

  def test_ecliptic_x
    assert_equal(
      0.17976672602,
      @t.ecliptic_x(@t_ajd).round(12)
    )
  end

  def test_ecliptic_y
    assert_equal(
      -0.980899204395,
      @t.ecliptic_y(@t_ajd).round(12)
    )
  end

  def test_right_ascension
    assert_equal(
      18.753078192426,
      @t.right_ascension(@t_ajd).round(12)
    )
  end

  def test_declination
    assert_equal(
      -0.372949956542,
      @t.declination(@t_ajd).round(12)
    )
  end

  def test_sidereal_time
    assert_equal(
      6.697967333333,
      @t.sidereal_time(@t_ajd).round(12)
    )
  end

  def test_local_sidereal_time
    assert_equal(
      18.697967333333,
      @t.local_sidereal_time(@t_ajd, @t_lon).round(12)
    )
  end

  def test_dlt
```

```
    assert_equal(
      12.120732161881,
      @t.dlt(@t_ajd, @t_lat).round(12)
    )
  end

  def test_diurnal_arc
    assert_equal(
      6.06036608094,
      @t.diurnal_arc(@t_ajd, @t_lat).round(12)
    )
  end

  def test_t_south
    assert_equal(
      12.055110859092,
      @t.t_south(@t_ajd, @t_lon).round(12)
    )
  end

  def test_t_rise
    assert_equal(
      5.994744778152,
      @t.t_rise(@t_ajd, @t_lon, @t_lat).round(12)
    )
  end

  def test_t_set
    assert_equal(
      18.115476940033,
      @t.t_set(@t_ajd, @t_lon, @t_lat).round(12)
    )
  end

  def test_rise_time
    rise = @t.t_rise(@t_ajd, @t_lon, @t_lat).round(12)
    assert_equal(
      'Sun rises \t\t\t : 5:59 UTC',
      "Sun rises \t\t\t : #{rise.floor}:#{(rise % 1 * 60.0).floor} UTC"
    )
  end

  def test_midday_time
    rise = @t.t_rise(@t_ajd, @t_lon, @t_lat).round(12)
    set = @t.t_set(@t_ajd, @t_lon, @t_lat).round(12)
```

```
    dlt = rise + set
    assert_equal(
      "Sun mid day \t\t\t : 12:3 UTC",
      "Sun mid day \t\t\t : #{(dlt / 2.0).floor}:#{((dlt /
      2.0 % 1.0) * 60).floor} UTC"
    )
  end

  def test_set_time
    set = @t.t_set(@t_ajd, @t_lon, @t_lat).round(12)
    assert_equal(
      'Sun sets \t\t\t : 18:6 UTC',
      "Sun sets \t\t\t : #{set.floor}:#{(set % 1 * 60.0).floor} UTC"
    )
  end
end
```