Improved Algorithm for Calculation of UTM and Geodetic Coordinates

Jeff Dozier
Washington, D.C.
September 1980
Contents

Abstract .................................................. 1

1. Description of UTM projection ......................... 1

2. UTM equations in closed form ......................... 1

   2.1 Notation .......................................... 1

   2.2 Gauss-Krüger equations .......................... 2

   2.3 Some useful approximations and identities ......... 2

      2.3.1 q-series approximations .................... 2

      2.3.2 Gauss’ arithmetic-geometric mean methods ... 3

      2.3.3 Identities .................................. 3

      2.3.4 Derivatives .................................. 3

3. Solution of UTM equations ............................ 4

   3.1 Infinite series method ........................... 4

   3.2 Iterative method .................................. 4

      3.2.1 The forward transform ....................... 4

      3.2.2 The inverse transform ....................... 5

4. Subprograms ........................................... 5

Acknowledgements ......................................... 6

References .............................................. 6

Appendix .................................................. 7
IMPROVED ALGORITHM FOR CALCULATION OF UTM
AND GEODETIC COORDINATES

JEFF DOZIER

National Environmental Satellite Service
Environmental Sciences Group
(on leave from University of California, Santa Barbara)

ABSTRACT

Expression of the equations for a UTM (Gauss-Krüger) projection in
terms of Jacobian elliptic functions, rather than their series expansions, allows
the projection to be used over wider zones than the standard 6 deg strips, and
thus makes it applicable to satellite data from the NOAA A-G series. An
efficient iterative solution method for either UTM or geodetic coordinates is
developed using a complex-arithmetic version of Newton's method. The
method can be used for longitudes up to 90 deg from the central meridian.

1. Description of UTM projection

The UTM (Universal Transverse Mercator) projection is commonly used world-wide for
the production of large-scale (i.e. $\geq 1:1,000,000$) maps. The coordinate system is described in
detail in many references (e.g. Richardus and Adler, 1972; Maling, 1973, U.S. Army, 1973).
It is a Gauss-Krüger projection with the following standards and modifications:

(1) The world is divided, with minor exceptions, into 60 zones with principal meridians every
6 deg of longitude. The meridian for zone 1 is 177 deg W.

(2) Along the principal meridian the scale is set to 0.9996 instead of 1.0, in order to minimize
the average error in the zone.

(3) To avoid the use of negative numbers, the easting coordinate is incremented by $5 \times 10^5$.
Similarly, in the southern hemisphere, the northing coordinate is incremented by $10^7$.

2. UTM equations in closed form

Description of the UTM projection coordinates requires elliptic functions and integrals.
Normally these have been expressed as series expansions, and the existent calculations methods
were designed for hand computation with extensive use of tables. The equivalent computer
programs (e.g. Quinones, 1970a, 1970b) typically have code which is cumbersome and whose
accuracy decreases markedly outside the 6 deg longitude zone.

The more formal descriptions of the Gauss-Krüger equations which have appeared in the
literature, have presented the results in terms of series expansions (e.g. Redfearn, 1948; Lee,
1962, 1976). Unfortunately these converge slowly and are often not sufficiently accurate. In
this paper I adapt Lee's (1976) closed form expressions for these equations to a reasonably
efficient computer solution which is accurate to machine precision. The method can be used
for all longitudes up to and including 90 deg from the central meridian.

2.1. Notation

- $a$: semi-major axis of the ellipsoid
- $b$: semi-minor axis of the ellipsoid
- $k$: eccentricity of the ellipsoid [$k^2 = (a^2-c^2)/a^2$]
- $\phi$: geodetic latitude
- $\psi$: isometric latitude [$\psi = \arctanh(\sin\phi) - k \arctanh(k \sin\phi)$]
- $\lambda$: longitude, from principal meridian of projection
northing and easting coordinates of Gauss-Krüger projection
northing and easting coordinates of intermediate projection
\( \psi + i \lambda \)
\( x + iy \)
\( u + iv \)
parameter of elliptic functions \([m = k^2]\)
Jacobian elliptic functions
elliptic integrals of first and second kinds (complete when no argument is specified)
parameters used in calculation of complete elliptic integrals

### 2.2. Gauss-Krüger equations

The general equation for a transverse mercator projection of a ellipsoid, expressed in terms of elliptic functions, is:

\[
\zeta = \text{arctanh}(\text{sn}w) - k \text{arctanh}(k \text{sn}w)
\]

where \( \text{sn}w \) is the Jacobian elliptic function \( \text{sn}(w|m) \) (see Abramowitz and Stegun, 1964, ch. 16; Byrd and Friedman, 1971). The Gauss-Krüger projection (one of a general class of transverse mercator projections) is defined in terms of \( w \) as:

\[
\frac{x}{a} = (1 - m) \int_0^w \text{dn}^{-2} t \, dt
\]

or

\[
\frac{x}{a} = E(w|m) - m \frac{\text{sn} w \cdot \text{cn} w}{\text{dn} w}
\]

where \( E(w|m) \) is an elliptic integral of the second kind.

### 2.3. Some useful approximations and identities

From the wealth of information about elliptic functions and integrals, I have made use of the following:

#### 2.3.1. q-series approximations

Define the nome \( q \) as:

\[
q = \exp \left( -\frac{\pi K'}{K} \right)
\]

The quarter periods \( K, K' \) are complete elliptic integrals of the first kind with parameters \( m, 1-m \) respectively.

\[
K = K(m) = \int_0^{\pi/2} (1 - m \sin^2 \theta)^{-1/2} \, d\theta
\]

\[
K' = K(1-m)
\]

From these, a rapidly converging expression for \( \text{sn} w \) is:

\[
\text{sn} w = \frac{2\pi}{m^{1/2} K} \sum_{n=0}^{\infty} \frac{q^{n+1/2}}{1-q^{2n+1}} \sin \left( (2n+1) \frac{\pi w}{2K} \right)
\]

Similar approximations exist for \( \text{cn} w, \text{dn} w \) etc., as well as for an elliptic integral of the second kind:
\[ E(w|m) = Z(w|m) + \frac{wE}{K} \]

E is the complete elliptic integral of the second kind, and \( Z(w|m) \) is Jacobi's zeta function:

\[ E = E(m) = \int_0^{\frac{\pi}{2}} (1 - m \sin^2 \theta)^{\frac{1}{2}} \, d\theta \]

\[ Z(w|m) = \frac{2\pi}{K} \sum_{n=1}^{\infty} q^n \sin \frac{n\pi w}{K} \]

2.3.2. Gauss' arithmetic-geometric mean methods

Start with the triple \((\alpha_0, \beta_0, \gamma_0)\). Then

\[ \alpha_n = \frac{\alpha_{n-1} + \beta_{n-1}}{2} \]

\[ \beta_n = \sqrt{\alpha_{n-1} \beta_{n-1}} \]

\[ \gamma_n = \alpha_n^2 - \beta_n^2 \]

until \( \alpha_n = \beta_n \) to the desired accuracy.

To evaluate complete elliptic integrals, we start with (Bulirsch, 1965; Hart, et al., 1968):

\[ \alpha_0 = 1 \quad \beta_0 = \sqrt{1 - m} \quad \gamma_0 = m \]

Then

\[ K = K(m) = \frac{\pi}{2\alpha_n} \]

\[ E = E(m) = K \left[ 1 - \sum_{j=0}^{n} 2^{-j} \gamma_j \right] \]

The algorithm can be checked by Legendre's relation:

\[ EK' + E'K - KK' = \frac{\pi}{2} \]

2.3.3. Identities

\[ sn^2 w + cn^2 w = 1 \]

\[ m \ sn^2 w + dn^2 w = 1 \]

2.3.4. Derivatives

\[ \frac{d}{dw} \ sn w = cn w \ dn w \]

\[ \frac{d}{dw} \ cn w = -sn w \ dn w \]

\[ \frac{d}{dw} \ dn w = -m \ sn w \ cn w \]

\[ \frac{d}{dw} \ E(w|m) = dn^2 w \]
3. Solution of UTM equations

We define the forward transform as solving for \((x,y)\) given \((\phi, \lambda)\) and the inverse transform as the reverse process. The principal difficulty in the forward transform is solving equation (1) for \(w\), given \(\zeta\). Likewise, the principal difficulty in the inverse transform is solving equation (2) for \(w\), given \(z\). Once \(w\) is known, neither the solution of (1) for \(\zeta\) nor (2) for \(z\) presents any formidable difficulty. The q-series approximations in 2.3.1 typically converge to \(10^{-18}\) accuracy in 10 or 11 iterations.

The numerical computation for the elliptic integrals, as described in section 2.3.2, need be performed only once for any ellipsoid.

There are two reasonable choices for solving (1) or (2) when \(w\) is unknown.

3.1. Infinite series method

The traditional method is to express the appropriate equation in its Taylor series expansion, and then to invert the power series (Van Orstrand, 1910). For example, Lee (1962) combines (1) and (2) to express \(z = f(\zeta)\) as:

\[
\frac{z}{a} = \zeta - \frac{1}{3! (1-m)^{\frac{1}{2}}} \zeta^3 + \frac{(5-m)}{5! (1-m)^{\frac{1}{2}}} \zeta^5 - \frac{(61+26m+m^2)}{7! (1-m)^{\frac{1}{2}}} \zeta^7 + \ldots
\]

The difficulties with this approach are that:

(1) No general equations for the coefficients in the series expansions of the Jacobian elliptic functions are known, so one must work them out from the Taylor series. The algebra can get difficult.

(2) Similarly, the formulas for inversion of a series become more and more cumbersome as the order increases.

(3) The series converges slowly; for example the 7th order term in (3) still has a magnitude exceeding 5 km at 45 deg N.

(4) It is therefore difficult to use the series method where our accuracy specifications are arbitrary, or where we might wish to use the Gauss-Krüger projection over a zone wider than 6 deg (Snyder, 1979). Therefore the traditional methods of calculating UTM coordinates are inappropriate for many kinds of satellite data, including those from the NOAA A-G series.

3.2. Iterative method

For these reasons I resort instead to an iterative scheme. This has the disadvantage of increasing the computation time, but gives us the capability to solve either the forward or inverse transform to the limit of our computing machine’s precision. The method is accurate for all longitudes up to 90 deg away from the central meridian of the projection, although the projection itself is generally not suitable at more than 10 or 15 deg away. The choice of numerical methods is influenced by two factors:

(1) The equations are well-behaved and have analytic derivatives. Furthermore, the identities between the Jacobian elliptic functions allow the first derivatives to be easily evaluated as a by-product of the evaluations of the functions.

(2) For all cases we can select a very good initial guess by letting \(k = 0\) and solving the equation for a spherical rather than ellipsoidal earth.

Under these circumstances Newton’s method will give us very rapid convergence.

3.2.1. The forward transform

Rewrite (1) as:

\[
f(w) = \arctanh(sn w) - k \arctanh(k sn w) - \zeta = 0
\]

Then, using sections 2.3.3 and 2.3.4:
\[ f'(w) = \frac{1-m}{\text{cn}w \, \text{dn}w} \]

By Newton’s method (see Lanczos, 1957):

\[ w_{n+1} = w_n - \frac{f(w_n)}{f'(w_n)} \]

The starting guess \( w_0 \) is found by letting \( k = 0 \). Then \( m = 0 \) and \( \text{sn}w = \text{sin}w \), and:

\[ w_0 = \arcsin(\tanh\xi) \]

3.2.2. The inverse transform

Rewrite (2) as:

\[ f(w) = E(w|m) - m \frac{\text{sn}w \, \text{cn}w}{\text{dn}w} - \frac{z}{a} = 0 \]

Then

\[ f'(w) = \text{dn}^2w - m \left[ \text{cn}^2w - (1-m) \frac{\text{sn}^2w}{\text{dn}^2w} \right] \]

When \( k = 0 \), solution of (2) leads to:

\[ \frac{z}{a} = \frac{w}{2} + \frac{\sin(2w)}{4} \]

To a first order approximation \( \sin(2w) = 2w \), so our initial guess \( w_0 \) is:

\[ w_0 = \frac{z}{a} \]

4. Subprograms

There are only two subprograms with which the user need interface. They are written in the C programming language (Kernighan and Ritchie, 1978) but would be easily translated into other languages, as the C code is relatively algorithmic. The routines are utmf and utminv, to calculate the forward and inverse transforms respectively. They are listed in the Appendix, along with the necessary supporting routines.

Arguments for utmf are:

\[ \text{utmf (lc, zone, lat, lon, north, east)} \]
\[ \text{int lc, *zone;} \]
\[ \text{double lat, lon, *north, *east;} \]

The input variables are:

\[ \text{lc} \quad \text{ellipsoid code (see Table 1)} \]
\[ \text{zone} \quad \text{UTM reference zone (if zone = 0, the routine will calculate the appropriate zone)} \]
\[ \text{lat, lon} \quad \text{geodetic coordinates in radians (positive in northern, eastern hemispheres)} \]

Output variables are:

\[ \text{north} \quad \text{northing} \]
\[ \text{east} \quad \text{easting} \]

For utminv, the arguments are in a different order:

\[ \text{utminv (hflag, lc, zone, north, east, lat, lon)} \]
\[ \text{int hflag, lc, zone;} \]
\[ \text{double north, east, *lat, *lon;} \]
In this case zone must be non-zero, as it is part of the UTM coordinate specification; north and east are input variables and lat and lon are output variables. The variable hflag, when negative, specifies that the northing coordinate has been incremented by $10^7$.

### Table 1

<table>
<thead>
<tr>
<th>code</th>
<th>ellipsoid</th>
<th>major axis (m)</th>
<th>minor axis (m)</th>
<th>eccentricity</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>sphere of Int'l volume</td>
<td>6370949</td>
<td>6370949</td>
<td>.000000</td>
</tr>
<tr>
<td>1</td>
<td>International</td>
<td>6378388</td>
<td>6356912</td>
<td>.081992</td>
</tr>
<tr>
<td>2</td>
<td>Clarke 1866</td>
<td>6378206</td>
<td>6356584</td>
<td>.082272</td>
</tr>
<tr>
<td>3</td>
<td>Clarke 1880</td>
<td>6378249</td>
<td>6356515</td>
<td>.082483</td>
</tr>
<tr>
<td>4</td>
<td>Everest</td>
<td>6377276</td>
<td>6356075</td>
<td>.081473</td>
</tr>
<tr>
<td>5</td>
<td>Bessel</td>
<td>6377397</td>
<td>6356079</td>
<td>.081697</td>
</tr>
<tr>
<td>6</td>
<td>Modified Everest</td>
<td>6377304</td>
<td>6356103</td>
<td>.081473</td>
</tr>
<tr>
<td>7</td>
<td>Australian National</td>
<td>6378160</td>
<td>6356775</td>
<td>.081820</td>
</tr>
<tr>
<td>8</td>
<td>Airy</td>
<td>6377563</td>
<td>6356257</td>
<td>.081673</td>
</tr>
<tr>
<td>9</td>
<td>Modified Airy</td>
<td>6377342</td>
<td>6356036</td>
<td>.081673</td>
</tr>
<tr>
<td>10</td>
<td>Walbeck</td>
<td>6376896</td>
<td>6355835</td>
<td>.081207</td>
</tr>
<tr>
<td>11</td>
<td>Southeast Asia</td>
<td>6378155</td>
<td>6356773</td>
<td>.081813</td>
</tr>
<tr>
<td>12</td>
<td>Krasovskiy</td>
<td>6378245</td>
<td>6356863</td>
<td>.081813</td>
</tr>
<tr>
<td>13</td>
<td>GTDS Model</td>
<td>6378140</td>
<td>6356755</td>
<td>.081820</td>
</tr>
</tbody>
</table>

### Acknowledgements

This work was initiated and completed while I was supported by a Senior Postdoctoral Research Associateship from the National Research Council, National Academy of Sciences. John P. Snyder of the U.S. Geological Survey provided several useful comments on the manuscript.

### References


**APPENDIX**

Here are listed the source codes for the routines in the UTM programs. The codes are in alphabetical order, but the structure of the routines is as follows:

1. The major subprograms
   - `utm`: calculates UTM coordinates from geodetic
   - `utminv`: calculates geodetic coordinates from UTM

2. Miscellaneous routines related to the UTM projection
   - `utmorg`: longitude of principal meridian
   - `utmzone`: zone associated with a given longitude

3. Miscellaneous routines related to earth coordinates
   - `geodet`: geodetic latitude (ζ) from isometric (φ)
   - `isomet`: isometric latitude (φ) from geodetic (ζ)
   - `sphcon`: axes (a, c) and eccentricity (k) for a given ellipsoid

4. Some transcendental and complex functions
   - `complex`: elementary operations (place ahead of main)
   - `atanh`: inverse of hyperbolic tangent (real)
   - `casin`: complex arc sine
   - `catan`: complex arc tangent
   - `ctanh`: complex arc tanh
   - `cpow`: complex number to real power
   - `csin`: complex sine
   - `ctanh`: complex hyperbolic tangent

5. Elliptic integrals and elliptic functions
   - `ecomp`: complete elliptic integral of 1st or 2nd kind (K or E)
   - `e2el`: complex elliptic integral of 2nd kind (with real parameter)
   - `jsn`: complex Jacobian elliptic function sn
   - `agm1`: Gauss' arithmetic-geometric mean from initial triple
   - `jzeta`: Jacobi's zeta function
   - `jseries`: general routine for q-series approximations
   - `qper`: quarter periods K, K' for Jacobian elliptic functions and other q-series approximations

6. Gauss-Krüger and transverse Mercator coordinates
   - `gkfor`: Gauss-Krüger coordinates z from isometric coordinates ξ
   - `gkinv`: ξ from z
   - `gkf`: Gauss-Krüger coordinates from intermediate coordinates w
   - `gk`: Gauss-Krüger coordinates from w, snw
   - `tmw`: intermediate coordinates w from isometric coordinates ξ
   - `tmwi`: intermediate coordinates w from Gauss-Krüger coordinates z
   - `tmerc`: isometric coordinates ξ from snw

7. Miscellaneous routines
   - `abend`: terminates program with error message
   - `cnewton`: complex version of Newton's method
   - `getpi`: returns double precision value of π
   - `macheps`: returns machine epsilon
   - `zerobr`: finds a zero of an arbitrary real function (from Brent, 1973)

All of the routines are coded in the C programming language (Kernighan and Ritchie, 1978). This language is close enough to Algol to make translation into other languages relatively simple. The following conventions and notations may be somewhat peculiar:
1. C makes explicit the distinction between the value of a variable and its address. Only a variable whose _address_ is passed to a subprogram may be changed by the subprogram. These have an asterisk (*), meaning "pointer to", prepended to the variable name in the subprogram. Variables passed by value may be changed in a subprogram, but these changes will not be passed back to the calling program. The prefix & to a variable in a calling program means "address of".

2. The notation
   
   ```
   i op = j;
   ```
   
   is the same as
   
   ```
   i = i op j;
   ```
   
3. The notation
   
   ```
   i ++;
   ```
   
   applies to integers only and means
   
   ```
   i = i + 1;
   ```
   
   The `++` after the variable indicates that the incrementation takes place after the use of the variable in an operation.

1. **abend**

   ```
   #define BELL 07
   abend (msg, dump)
   char *msg;
   int dump;
   /*
   * function: abend
   * purpose: terminate a program if an irrecoverable
   * error occurs, and print explanatory message
   * if dump = 1, cause a core dump also
   * usage: abend ("message to be printed", dump);
   */
   
   putchar (BELL);
   printf ("%sn\n", msg);
   if (dump == 1) {
     printf ("(core dumped)\n\n");
     fclose (stdout);
     abort (); /* dump core for debugging */
   } else
     exit (0);
   }
   
2. **agm1**

   /*
    * returns arithmetic-geometric mean of a, b
    * also calculates sum 2^k(k-1) c sub k
    */
   double agm1 (a, b, c, c2sum)
   double a, b, c, *c2sum;
   
   double fabs0, macheps0, sqrt0();
   double sum, h, tk, eps;
   char str[80];
   
   if (a*b < 0.) {
     sprintf (str, "agm1: a %e b %e", a, b);
     abend (str, 0);
   }
   
   sum = 5.e-1 * c;
   tk = 1;
   eps = macheps();
   
   while (fabs (a-b) > a*eps) {
     h = a;
     a = (a + b) / 2;
     b = sqrt (h * b);
     c = (a-b) * (a+b);
     sum += tk * c;
     tk *= 2;
   }
   
   *c2sum = sum;
   return (a);
   }

3. **atanh**

   /*
    * inverse hyperbolic tangent
    */
   
   double atanh (x)
   double x;
   
   double log0, hold;
   char str[80];
   
   /* check range */
   hold = x * x;
   if (hold < 0. || hold > 1.) {
     sprintf (str, "atanh: argument = %e", x);
     abend (str, 1);
   }
   
   return (5.e-1 * log ((1. + x) / (1. - x)));

4. **casin**

   /*
    * complex arcsin
    */
   
   complex casin (z)
   complex z;
   
   extern int errno;
   
   complex result;
double    alpha, beta, xp, xm, y2, l, r;
double    sqrt0(), asin0(), log0();
char       str[80];

xp = z.re * z.re + 2 * z.re + 1;
xm = z.re * z.re - 2 * z.re + 1;
y2 = z.im * z.im;
l = sqrt0(xp + y2);
if (errno == EDOM)
    abend("casin: bad sqrt for l", 1);
result.re = asin0(beta);
if (errno == EDOM) {
    sprintf(str, "casin: bad asin() arg \%.18e", beta);
    abend(str, 0);
}
result.im = log(alpha + sqrt((alpha + 1) * (alpha - 1)));
if (errno == EDOM)
    abend("bad log or sqrt in casin", 1);
if (z.im < 0)
    result.im = -result.im;

return (result);
}

5.  catan

/*
complex arctan
*/
complex     catan (z)
complex     z;
{
    complex result;
    double    x, y, xxsqsq, tway;
    double    log0(), atan20();

    extern int   errno;

    x = z.re;
y = z.im;
if (x == 0.0 && y == 1.0)
    abend("catan: z'2 = -1', 0);
xxxsqsq = x * x + y * y;
tway = 2 * y;
result.re = 5.e-1 * atan2(2 * x, 1 - xxsqsq);
result.im = 2.5e-1 * log((xxxsqsq + tway + 1) / (xxxsqsq - tway + 1));
if (errno == EDOM)
    abend("catan: bad arg to log()");

return (result);
}

6.  catanh

/*
complex arctanh
uses relation arctanh z = -i arctan iz
*/
complex     catanh (z)
complex     z;
{
    complex     i, hold;
    complex     cmult0(), catan();
    i.re = 0;
    i.im = 1;
    hold = cmult(i, z);
    i.im = -1;

    return (cmult(i, catan(hold)));
 }

7.  cnewton

/*
complex version of Newton's method
solves for complex zero of f(z)
given initial guess z

the function f (z, ofzp) returns f(z)
and 1/f'(z) in ofzp
*/
complex     cnewton (z, f, tol)
complex     z, (*p)();
double      tol;
{
    complex     fz, ofzp, h, zold, cr, cmult0(), csub0();
    double      macheps(), cabs();
    double      eps, me;
    int         maxit;
    char        str[80];

    me = 2.0 * macheps();
    maxit = 50;
    zold = z;
fz = (*f)(z, &ofzp);

    while (maxit --) {
        /* check if z is a root */
        if (cabs(fz) <= tol) return (z);

        /* calculate new approximation */
        h = cmult(fz, ofzp);
        z = csub(z, h);

        /* see how close new approx. is */
        eps = tol + me * cabs(z);
        cr = csub(z, zold);
        if (5.e-1 * cabs(cr) < eps) return (z);

        zold = z;
fz = (*f)(z, &ofzp);
    }

    /* if we get to here, we didn't converge */
8. complex

typedef struct {
  double re, im;
} complex;

complex cadd (z1, z2) {
  complex result;
  result.re = z1.re + z2.re;
  result.im = z1.im + z2.im;
  return (result);
}

complex csub (z1, z2) {
  complex result;
  result.re = z1.re - z2.re;
  result.im = z1.im - z2.im;
  return (result);
}

complex cmult (z1, z2) {
  complex result;
  result.re = z1.re * z2.re - z1.im * z2.im;
  result.im = z1.re * z2.im + z1.im * z2.re;
  return (result);
}

complex cdiv (z1, z2) {
  double denom;
  complex result;
  denom = z2.re * z2.re + z2.im * z2.im;
  result.re = (z1.re * z2.re + z1.im * z2.im) / denom;
  result.im = (z2.re * z1.im - z1.re * z2.im) / denom;
  return (result);
}

double carg (z) {
  complex result;
  if (result.re <= 0.0) {
    result.re = -result.re;
    result.im = -result.im;
  }
  return (result);
}

9. cpowr

/*
   complex number to a real power
*/

complex cpowr (z, p) {
  complex result;
  double p;
  extern int errno;
  complex result;
  double pow(), carg(), cabs(), r, nt, cos(), sin(), rn;
  r = cabs(z);
  rn = pow(r, p);
  if (errno == ERANGE)
    abend("overflow in cpowr", 0);
  if (errno == EDOM)
    abend("bad arg to pow()", 1);
  nt = carg(z);
  nt *= p;
  result.re = rn * cos(nt);
  result.im = rn * sin(nt);
  return (result);
}

10. csin

/*
   complex sin
*/

complex csin (z) {
  complex result;
  double x, y, sin(), cos(), sinh(), cosh();
  x = z.re;
  y = z.im;
  result.re = sin(x) * cos(y);
  result.im = cos(x) * sin(y);
  return (result);
}

11. ctanh

/*
   complex tanh
*/

complex ctanh (z) {
  complex result;
  double sin(), cos(), sinh(), cosh();
  double denom, twox, twoy;
  twox = 2.0 * z.re;

```c
w = 2. * z.im;
den = cosh (2*w) + cos (2*w);
result.re = sinh (2*w) / denom;
result.im = sin (2*w) / denom;

return (result);
```

12. e2ci

```c
/*
  complex elliptic integral of 2nd kind
*/
complex e2ci (m, w)
double m;
complex w;
{
  complex jzeta(), cadd(), z;
double k, kp, ek, e_comp(), q;

  static double mlast, c;
  qiper (m, &k, &kp, &q);
  z = jzeta (k, q, w);

  if (m != mlast || m == 0.)
    c = e_comp (m, 2);
    mlast = m;
  
  ek = e / k;
  w.re *= ek;
  w.im *= ek;

  return (cadd (z, w));
}
```

13. e_comp

```c
/*
  complete elliptic integral
  parameter m
  kind 1 or 2
*/
double e_comp (m, kind)
double m;
int kind;
{
  /* uses a.g.m. method*/
  double sqrt(), agm1();
  double b, c, pi;

  static double mlast, a, sum;
  getpi (&pi);

  if (kind != 1 && kind != 2)
    abend ("e_comp: kind must be 1 or 2", 0);

  if (m == 1.)
    if (kind == 1)
      abend ("e_comp: 1st kind, m = 1", 0);
      return (1.);
```

14. geodet

```c
/*
  finds latitude as function of isometric latitude
  and eccentricity
*/
double psi_trn, k_tran;

double geodet (psi, k)
double psi, k;
{
  double zerobr(), isomd(), tanh(), asint();
  double phi, pi2, tol;
  char str[80];

  extern double psi_trn, k_tran;
  extern int errno;

  /* first check the special case k = 0 */
  if (k == 0.)
    phi = asin (tanh (psi));
    if (errno == EDOM)
      sprintf (str, "geodet: asin arg > 1, psi ", psi);
      abend (str, 1);
    
    return (phi);
  }

  psi_trn = psi;
  k_tran = k;
  getpi (&pi2);
  pi2 /= 2;
  tol = 0;

  if (psi == 0.) return (0.);

  else if (psi > 0.)
    phi = zerobr (0., pi2, tol, isomd);
  else
    phi = zerobr (-pi2, 0., tol, isomd);

  return (phi);
```
double isomet();
return (psi_tran - isomet (phi, k_tran));
}

15. getpi

/* sets double precision value of pi */
Usage: getpi (&pi)

*/
getpi(x)
double *x;
{
static int first;
static double holdpi;
double atan();

if (first != 1) {
    holdpi = 4.0 * atan(1.0);
    first = 1;
}

*x = holdpi;
}

16. gk

/*
calculates G-K coordinates from w, snw
*/
gk (m, w, snw)
double m;
complex snw, w, snw;
{
    complex cdw, cadd(), cmult(), csubb(), e2ci(), cpowr(),
        t1, t2, one, num, denom, ms;
    t1 = e2ci (m, w);
    one.re = 1;
    one.im = 0;
    num = cmult (csubb (one, snw), cadd (one, snw));
    ms.re = m * snw.re;
    ms.im = m * snw.im;
    denom = cmult (csubb (one, ms), cadd (one, ms));
    cdw = cpowr (cdiv (num, denom), 5.0 - 1);
    t2 = cmult (snw, cdw);
    t2.re *= m;
    t2.im *= m;
    return (csubb (t1, t2));
}
20. isomet
/*
finds isometric latitude as a function of latitude and eccentricity
*/

double isomet (lat, eccen)
double lat, eccen;
{
    double x, atanh0, sin0, pi2, fabs0;
    char str[80];

    /* check range of arguments */
    if (eccen < 0.0 || eccen > = 1.)
    {
        sprintf (str, "isomet: eccentricity = %e", eccen);
        abort (str, 1);
    }

    getpi (&pi2);
    pi2 /= 2;
    if (fabs (lat) > pi2) {
        sprintf (str, "isomet: latitude %.16f", lat);
        abort (str, 1);
    }

    x = sin (lat);
    return (atanh (x) - eccen * atanh (eccen * x));
}

21. jseries
/*
expands an arbitrary complex series (of the type found in Jacobian elliptic functions) until convergence
*/

complex jseries (start, q, qf, vf)
int start;
double q, (*qf)();
complex v, (*vf)();

/*
start = 0 or 1
q = real parameter for real qf()
v = complex parameter for complex vf()
*/
{
    int n;
    double h, aold, anew;
    complex z, x, cadd0;

    n = start;
    anew = z.real = z.imag = 0;
    aold = 9999;

    while (aold != anew) {
        h = (*qf) (n, q);
        x = (*vf) (n, v);
        n += 1;
        x.real *= h;
        x.imag *= h;
        aold = anew;
        z = cadd (z, x);
        anew = z.real * z.real + z.imag * z.imag;
    }
    return (z);
}

22. jsn
/*
complex Jacobian elliptic function sn u
*/

complex jsn (m, u)
double m;
complex u;
{
    double pi, k, kp, q, fact, qsnf0, sqrt0;
    complex v, sn;
    complex vsnf0, jsnser0, csinf0, ctanh0;

    if (m == 0.)
        return (csin (u));
    if (m == 1.0)
        return (ctanh (u));

    qper (m, &k, &kp, &q);
    getpi (&pi);
    fact = pi / (2.*k);
    v.real = fact * u.real;
    v.imag = fact * u.imag;
    sn = jsnser (0, q, v, qsnf0, vsnf0);
    fact = 2.*pi / (sqrt(m) * k);

    sn.real *= fact;
    sn.imag *= fact;
    return (sn);
}

/*
coefficients for q-series expansion of sn
*/

double qsnf (n, q)
int n;
double q;
{
    double np, dp, pow0;

    np = n;
    np += 5.e-1;
    dp = 2*n + 1;

    return (pow(q,np) / (1. - pow(q,dp)));
}

/*
argument for q-series expansion of sn
*/

complex vsnf (n, v)
int n;
complex v;
{
    complex csinf0, arg;
    double d;
d = 2 * n + 1;
arg.re = v.re * d;
arg.im = v.im * d;
return (csin (arg));
}

23. jzeta

/*
 * Jacobi's zeta function
 */
complex jzeta (k, q, w)
double k, q;
complex w;
{
double pi, exp(), qfzeta(),
complex v, h, vffzeta(), jsseries();
if (k == 0.0)
    h.re = h.im = 0;
else{
    getpi (&pi);
    v.re = w.re * pi / k;
    v.im = w.im * pi / k;
    h = jsseries (1, q, v, qfzeta, vffzeta);
    h.re *= 2 * pi / k;
    h.im *= 2 * pi / k;
}
return (h);
}

/*
 * coefficients for sin series for jzeta()
 */
double qfzeta (n, q)
int n;
double q;
{
double qp, pow();
    qp = pow (q, (double) n);
return (qp / (1.0 - q * qp));
}

/*
 * argument for sin-series for jzeta()
 */
complex vffzeta (n, v)
int n;
complex v;
{
    complex csin();
    v.re *= n;
    v.im *= n;
return (csin (v));
}

24. macheps

double macheps()
/**
 * returns machine epsilon
 */
{
double s, one;
static double x;
static int first;
if (first != -1) {
    first = 1;
    one = 1;
    x = 1;
s = 2;
while (s > one) {
    x *= 5.0e-1;
    s = one + x;
}
}
return (x);
}

25. qiper

/*
 * calculates quarter periods K & K' and nome q
 * for Jacobian elliptic functions
 */
qiper (m, k, kp, q)
double m, "k", "kp", "q";
{
    static double mlast, klast, kplast, qlast;
    double sqrt(), e_comp(), exp();
    double pi, ms;
    getpi (&pi);
    if (m == 0.0) {
        "k" = pi / 2.0;
        "q" = 0;
        "kp" = 1.0e33;
    } else {
        if (mlast == m) {
            "k" = klast;
            "kp" = kplast;
            "q" = qlast;
        } else {
            mlast = m;
            "k" = klast = e_comp (m, 1);
            ms = sqrt (m);
            "kp" = kplast = e_comp ((1.0 - ms) * (1.0 + ms), 1);
            "q" = qlast = exp (-pi * kplast / klast);
        }
    }
}
26. sphcon
See Table 1.

/*
  information about specific ellipsoids
*/
sphcon (l, major, minor, eccen)
int l;
double *major, *minor, *eccen;
/*
code for ellipsoid choices (l -- input)
0  Sphere of equal volume
1  International
2  Clarke 1866
3  Clarke 1880
4  Everest
5  Bessel
6  Modified Everest
7  Australian National
8  Airy
9  Modified Airy
10  Wallbeck
11  Southeast Asia
12  Krassovskiy
13  GTDS Model
(output)
major  semi-major axis
minor  semi-minor axis
eccen [defined as sqrt ((a*a - b*b) / (a*a))]
*/
char str[80];
double eccensq, sqrt0;
/*
in some of the values that follow, the minor axis is computed
from the flattening
*/
switch (l) {
  case 0: /*major = *minor = 6.370949;
    break;
  case 1: /*major = 6.378388;
    *minor = 6.3569119461;
    break;
  case 2: /*major = 6.3782064;
    *minor = 6.35658383;
    break;
  case 3: /*major = 6.378249145;
    *minor = *major * (1. - 1./2.93465e2);
    break;
  case 4: /*major = 6.3772763452;
    *minor = 6.3560754133;
    break;
  case 5: /*major = 6.377397155;
    *minor = 6.3560789628;
    break;
  case 6: /*major = 6.377304063;
    *minor = *major * (1. - 1./3.008017e2);
    break;
  case 7: /*major = 6.37816;
    *minor = *major * (1. - 1./2.9825e2);
    break;
  case 8: /*major = 6.377563396;
    *minor = *major * (1. - 1./2.9932496e2);
    break;
  case 9: /*major = 6.37734189;
    *minor = *major * (1. - 1./2.99324959e2);
    break;
  case 10: /*major = 6.376896;
    *minor = 6.3558348467;
    break;
  case 11: /*major = 6.378155;
    *minor = 6.3567732025;
    break;
  case 12: /*major = 6.378245;
    *minor = 6.3568630188;
    break;
  case 13: /*major = 6.37814;
    *minor = 6.356754786253143;
    break;
  default: 
    printf (str, "sphcon: ellipsoid code %d unimplemented", l);
    abend (str, 0);
  }
  eccensq = (*major - *minor) * (*major + *minor) / (*major * *major);
  if (eccensq < 0) {
    printf (str, "sphcon: eccen %f, maj %f, min %f",
      eccensq, *major, *minor);
    abend (str, 1);
  }
  *major *= 1.06;
  *minor *= 1.06;
  *eccen = sqrt (eccensq);
27. tmerc

/*@ returns transverse mercator coordinates as function of sn w */
complex tmerc(k, snw)
double k;
complex snw;
{
    complex ksnw, catanh(), csub(), t1, t2;
    ksnw.re = k * snw.re;
    ksnw.im = k * snw.im;

    t1 = catanh(snw);
    t2 = catanh(ksnw);
    t2.re *= k;
    t2.im *= k;

    return (csub(t1, t2));
}

28. tmw

/*@ iteratively finds w as function of k, zeta */
complex z_tran;
double k_tran;
complex tmw(k, zeta)
double k;
complex zeta;
{
    extern complex z_tran;
    extern double k_tran;

    complex cnewton(), tmfd(), w, casin(), catanh();
double tol, pi2;
double sqrt(), macheps();
double fabs();

tol = sqrt(macheps());
getpi(&pi2);
pi2 /= 2;

/* initial guess */
w = casin(catanh(zeta));

/* check for quadrant compatibility */
if(fabs(zeta.im) > pi2)
    abend("tmw: long. diff > pi/2", 0);

k_tran = k;
z_tran = zeta;

w = cnewton(w, tmfd, tol);

return (w);
}

returns (in ofpw) 1/f
*/

complex tmfd(w, ofpw)
complex w, *ofpw;
{
    extern complex z_tran;
    extern double k_tran;

    complex snw, fw, crn, dnn, ms, num, denom,
        one;

    complex jsn(), cdv(), cmult(), cadd(), cpowr(),
        tmerc();

double m;

    one.re = 1;
    one.im = 0;
    m = k_tran * k_tran;
    snw = jsn(m, w);
    fw = csub(tmerc(k_tran, snw), z_tran);

    crn = cpowr(cmult(csub(one, snw), cadd(one, snw)), 5.e-1);
    ms = snw;
    ms.re *= m;
    ms.im *= m;
    dnn = cpowr(cmult(csub(one, ms), cadd(one, ms)), 5.e-1);

    num = cmult(crn, dnn);
    denom.re = 1. - m;
    denom.im = 0;

    *ofpw = cdv(num, denom);

    return (fw);
}

29. tmwi

/*@ iteratively finds w as function of k, z */

complex z_tran;
double k_tran;
complex tmwi(k, z)
double k;
complex z;
{
    extern complex z_tran;
    extern double k_tran;

    complex cnewton(), casin(), tmfd();
    complex w;
double pi2, tol;
double sqrt(), macheps();

tol = sqrt(macheps());
getpi(&pi2);
pi2 /= 2;

k_tran = k;
z_tran = z;

/* initial guess */
w = z;
w = cnewton (w, tmid, tol);
return (w);
/* returns difference between transverse mercator intermediate coordinates and z */
returns (in opfw) reciprocal of first derivative
/* complex tmid (w, opfw)
complex w, *opfw;
{
extern complex z_tran;
extern double k_tran;
complex snw, cn2w, dn2w, hold, ms, fw, one;
complex jsn(), csb(), gk(), cmult();
double m, mp;
m = k_tran * k_tran;
mp = (1. + k_tran) * (1. - k_tran);
snw = jsn (m, w);
fw = csb (gk (m, w, snw), z_tran);
/* calculate derivative */
snw = cmult (snw, snw);
one.re = 1;
one.im = 0;
 cn2w = csb (one, snw);
ms = snw;
ms.re *= m;
ms.im *= m;
dn2w = csb (one, ms);
hold = cdiv (snw, dn2w);
hold.re *= mp;
hold.im *= mp;
hold = csb (cn2w, hold);
hold.re *= m;
hold.im *= m;
hold = csb (dn2w, hold);
*opfw = cdiv (one, hold);
return (fw);
} } 30. utmf
/* computes the UTM coordinates (major zone) for a point whose lat-long coordinates are known */
#define N_FALSE  1.e7
#define E_FALSE  5.e5
#define SCALE_F  9.996e-1
utmf (l, lat, lon, *north, *east,
variables:
(input)
l ellipsoid code [see sphcon() for tabulation]
zone (if != 0) major zone
lat latitude in radians (+ in north hem)
lon longitude in radians (+ in east hem)
(output)
zone major zone (if = 0 on input)
northnorthing
east casting
*/
double eccen, /* eccentricity of ellipsoid */
major, /* semi-major axis (m) */
orig, /* longitude origin of zone */
minor, /* minor axis, not used */
isomet(),
utmorig();
int utmzone(), abs();
sphcon (l, &major, &minor, &eccen);
if (*zone = 0) *zone = utmzone (lon);
orig = utmorig (*zone);
gkfor (eccen, major, isomet (lat, eccen), lon-orig, north, east);
*north = SCALE_F;
*east *= SCALE_F;
if (lat < 0.) *north += N_FALSE;
*east += E_FALSE;
} 31. utm2v
/* computes geodetic coordinates for a point whose UTM coordinates are known */
#define N_FALSE  1.e7
#define E_FALSE  5.e5
#define SCALE_F  9.996e-1
utm2v (hflag, l, zone, north, east, lat, lon)
int hflag, l, zone;
double north, east, *lat, *lon;
/* variables:
(input)
hflag (if negative) in so. hem., add N_FALSE to northing
l ellipsoid code [see sphcon() for tabulation]
zone (if != 0) major zone
lat latitude in radians (+ in north hem)
lon longitude in radians (+ in east hem)
(output)
zone major zone (if = 0 on input)
northnorthing
east casting
*/
double eccen,  /**< eccentricity of ellipsoid */
    major,  /**< semi-major axis (m) */
    orig,  /**< longitude of origin of zone */
    minor,  /**< minor axis, not used */
    psi,  /**< isometric latitude */
    dlon,  /**< longitude difference */
    geodet(),
    utmorig();

int utmzone(), abs();

sphcon (l, &major, &minor, &eccen);
orig = utmorig (zone);
    east -= E_FALSE;
if (hflag < 0)  
    north -= N_FALSE;
    east /= SCALE_F;
north /= SCALE_F;

gkinv (eccen, major, north, east, &psi, &dlon);

*lon = dlon + orig;
*lat = geodet (psi, eccen);

32. utmorig

/**< returns origin (in radians) of designated UTM zone */

double utmorig (zone)
int zone;
{
    double d, pi;
    /* check to see if zone reasonable */
    if (zone < 0 || zone > 60)  
        abend ("utmorig - zone value unreasonable", 1);

getpi (&pi);
    d = zone * 6 - 183;
return (d * pi / 1.892);  
}

33. utmzone

/**< returned value is UTM zone corresponding to lon (radians) */

int utmzone (lon)
    double lon;
{
    int i, j;
    double p, fabs();
    getpi (&pi);
    /* check range */
    if (fabs (lon) > pi)
        abend ("utmzone -- abs (lon) > pi", 0);
    rdms (lon, geod);
    z = geod[0] / 6;
    /* set i = 0 if on zone boundary */
    i = geod[0]%6 + geod[1] + geod[2] == 0 ? 0 : 1;
    if (lon < 0)  
        z += 31 - i;
else z += 30 + i;
return (2);
}

34. zeroBr

/**< finds zero of a function by Brent’s algorithm */
(from Brent, 1973)

double zeroBr (a, b, t, f)
    double a, b, t, (*f)();
{
    double c, d, e, fa, fb, fc, tol, m, p, q, r, s, eps;
    double machepee();
    double fabs(), log();
    int maxfun;
    char str[80];

eps = 2 * machepee();
    /* compute max number of function evaluations */
if (a == b)
    abend ("zeroBr(): a = b", 0);
    fb = fabs (b) >= fabs (a) ? fabs (b) : fabs (a);
tol = 5.e-1 * t + 2 * eps * fb;
s = log (fabs (b - a) / tol) / log (2);
maxfun = s * s + 1;

fa = (*f)(a);
f = fb = (*f)(b);
    if (fb == 0) return (b);
    if (fa == 0) return (a);
if (fb*fa > 0)  
    abend ("zeroBr(): root not spanned", 1);
while (maxfun --) |
    if ((fb > 0 && fc > 0) || (fb <= 0 && fc <= 0)) |
        c = a;
        fc = fa;
        d = e = b - a;
    }
    if (fabs(fc) < fabs(fb)) |
        a = b;
        b = c;
        c = a;
        fa = fb;
        fb = fc;
        fc = fa;
    }
    tol = eps * fabs(b) + t;
    m = (c - b) / 2;
if (fabs(m) < tol || fb == 0) return (b);
/* see if bisection is forced */
if (fabs(c) < tol \| fabs(fa) <= fabs(fb)) d = e = m;
else {
    s = fb/fa;

    if (a == c) { /* linear interpolation */
        p = 2*m*s;
        q = 1 - s;
    }

    else { /* inverse quadratic interpolation */
        q = fa/fc;
        r = fb/fc;
        p = s * (2*m*q*(q-r) - (b-a)*(r-1));
        q = 1;
        q = (r-1)*(s-1);
    }

    if (p > 0) q = -q;
    else p = -p;

    s = e;
    e = d;

    if (2*p < 3*m*s - fabs(tol*q) \&\& p < fabs(s*q/2))
        d = p/q;
    else d = e = m;
}

a = b;
fa = fb;

if (fabs(d) > tol) b += d;
else if (m > 0) b += tol;
else b -= tol;

fb = (*f)(b);
}

abend("zerobr(): did not converge",1);