```
#! /usr/bin/perl -w
# December 2010
use Math::Trig;
# Calculate values that should be global, to minimize repeated calculations
($sin60, $cos60, $yTranslate, @Prelims) = Preliminary();

$Skip = "YES";
unless ($Skip) {          # Option to skip printing output of sub Preliminary
  ($xA, $yA, $xB, $yB, $xC, $yC, $xD, $yD, $xE, $yE, $xF, $yF, $xG, $yG,
     $xM, $yM, $xT, $yT, $AG, $AB, $BD, $GF, $BDE, $GFE, $R, $DeltaMEq) = @Prelims;
  print "\n\nPoints";
  foreach $i qw(A B C D E F G M T) {printf "\t%s", $i; }
  print "\nx";
  foreach $i ($xA,$xB,$xC,$xD,$xE,$xF,$xG,$xM,$xT) {printf "\t%.4f", $i};
  print "\ny";
  foreach $i ($yA,$yB,$yC,$yD,$yE,$yF,$yG,$yM,$yT) {printf "\t%.4f", $i};
  print "\n\nLengths";
  foreach $i qw(AG AB BD GF BDE GFE R DeltaMEq) {printf "\t%s", $i; }
  print "\n";
  foreach $i ($AG,$AB,$BD,$GF,$BDE,$GFE,$R,$DeltaMEq) {printf "\t%.4f", $i};
  print "\n\n";
}  # End of skip printing output of sub Preliminary

$Skip = "YES";
unless ($Skip) {          # Option to skip calculating and printing out whole-numbered meridians
  # Meridians multiple of 5° are drawn from point A; other meridians are drawn from
  # parallel 85°. Array's second index is: 0 = polar start (point A or parallel 85°);
  # 1 = frigid joint; 2 = torrid joint; 3 = equator.
  ($xA, $yA, $xB, $yB, $xC, $yC, $xD, $yD, $xE, $yE, $xF, $yF, $xG, $yG,
     $xM, $yM, $xT, $yT, $AG, $AB, $BD, $GF, $BDE, $GFE, $R, $DeltaMEq) = @Prelims;
  # Meridian 0° has no joints; will make elements 1 and 2 equal to equatorial point, point G.
  ($x[0][0], $y[0][0]) = ($xA, $yA);
  ($x[0][1], $y[0][1]) = ($xG, $yG);
  ($x[0][2], $y[0][2]) = ($xG, $yG);
  ($x[0][3], $y[0][3]) = ($xG, $yG);
  foreach $m (1 .. 45) {
    # Meridians multiple of 5° are drawn from point A; other meridians are drawn from
    # parallel 85°.
    if ($m %5 == 0) {  # Every 5th meridian starts at point A
       ($x[$m][0], $y[$m][0]) = ($xA, $yA);
    } else {  # Minor meridians start at 85° of latitude
       ($x[$m][0], $y[$m][0]) = MPtoXY ($m, 85, @Prelims);
    }

    ($x[$m][3], $y[$m][3], $x[$m][2], $y[$m][2], $x[$m][1], $y[$m][1], $Lt, $Lm) =
       Joints ($m, $xA, $xE, $yE, $xF, $yF, $xG, $AB, $GF, $DeltaMEq);
  }
  print "\n\nJoints\nx";
  foreach $m (0 .. 45) { printf "\t%d", $m; } print "\n";
  foreach $i (0 .. 3) {
    print $i;
    foreach $m (0 .. 45) {
      if (defined ($x[$m][$i])) {printf "\t%.4f", $x[$m][$i];
        } else { printf "\t undef"; }
    }
    print "\n";
```

```
  }
  print "\ny";
  foreach $m (0 .. 45) { printf "\t%d", $m; } print "\n";
  foreach $i (0 .. 3) {
    print $i;
    foreach $m (0 .. 45) {
      if (defined ($y[$m][$i])) {printf "\t%.4f", $y[$m][$i];
        } else { printf "\t undef"; }
    }
    print "\n";
  }
  print "\n";
}  # End skipping calculation and printing out of whole-numbered meridians

$Skip = "YES";
unless ($Skip) {          # Option to skip calculating all whole-numbered meridian-parallel points
  foreach $p (0 .. 90) {
    foreach $m (0 .. 45) {
      ($x[$m][$p], $y[$m][$p]) = MPtoXY ($m, $p, @Prelims);
    }
  }
  print "\n\nPoints\nx";
  foreach $m (0 .. 45) { printf "\t%d", $m; } print "\n";
  foreach $p (0 .. 90) {
    print $p;
    foreach $m (0 .. 45) {
      if (defined ($x[$m][$p])) {printf "\t%.4f", $x[$m][$p];
        } else { printf "\t undef"; }
    }
    print "\n";
  }
  print "\ny";
  foreach $m (0 .. 45) { printf "\t%d", $m; } print "\n";
  foreach $p (0 .. 90) {
    print $p;
    foreach $m (0 .. 45) {
      if (defined ($y[$m][$p])) {printf "\t%.4f", $y[$m][$p];
        } else { printf "\t undef"; }
    }
    print "\n";
  }
}  # End skipping calculation of whole-numbered meridian-parallel points

$Skip = "";
unless ($Skip) {  # Option to skip making macro files of Coastal Data
  # Read some Coastal Data, convert to M-map coordinates, and output an OpenOffice.org
  # Draw macro. Note: macro assumes that the following functions already exist:
  # L() to start a polyline shape, P() to prepare x,y coordinates for a point, and
  # Collect() to group all shapes on the page.
  #
  # File read is of MAPGEN data format: two columns ASCII flat file with:
  # longitude tab latitude; at the start of each segment there is a line containing only
  # "# -b".
  # When downloading from the net, it was asked for data for scale 1:2,000,000, covering
  # longitude from -67° to -59° and latitude from 43° to 48°, that is, Nova Scotia's area.
  # File is NS-2M.dat.gz, is zipped, and has 13,493 lines.
```

```
print "Going to read land data.\n";
# Variable $Map can be set to "M" to output coordinates in M-map system, or to
# any other value, to output coordinates in Gene's one-octant system.
$Map = "G";
# Set up a few variables
$maxX=-99999;  $maxY=-99999;  $minX=999999;  $minY=999999;
$oldLong = 99999;  $oldLat = 99999;
# Start macro file
open (MACRO, ">NSmacro.txt");  # Name for output file with OOo macro
print MACRO "Sub NovaScotia\nD=ThisComponent\nG=D.DrawPages(0)\n";
print MACRO "C=RGB(0,0,0)\n";  # Line color will be black
# Open coastal data file
open (DATA, "zcat NS-2M.dat.gz | ");  # Name of input file with coastal data
$nData = 0;
while (<DATA>) {
  $Line = $_ ;
  chomp($Line);  # Remove carriage return from end of line of data read
  $nData ++;
  if ($nData % 1000 == 0) { print "Read $nData lines of land data so far.\n"; }
  if ($Line ne "\# -b") {
    ($Long,$Lat) = split(/\t/,$Line);
    if ($Long != $oldLong && $Lat != $oldLat) {
      # If this point is a repeat of the previous point on this segment, this section is
      # not run, and this point is neither converted nor included in the macro.
      $oldLong = $Long;  $oldLat = $Lat;
      # COORDINATE CONVERSION
      # Convert real longitude, latitude to template half-octant meridian and parallel
      ($m, $p, $Sign, $Octant) = LLtoMP ($Long, $Lat);
      # Convert template meridian, parallel to template half-octant x, y coordinates
      ($x, $y) = MPtoXY ($m, $p, @Prelims);
      # Convert template x, y coordinates to M-map or G's x and y coordinates.
      # Variable $Map was set previously in this "Skip" block.
      if ($Map eq "M") { # M-map coordinates
        ($xNew, $yNew) = MJtoG ($x, $Sign*$y, $Octant, $sin60, $cos60, $yTranslate);
      } else { # G's single octant coordinates
        ($xNew, $yNew) = MJtoG ($x, $Sign*$y, 0, $sin60, $cos60, $yTranslate);
      }
      # Keep track of maximum and minimum values
      if ($xNew < $minX) {$minX = $xNew;}
      if ($xNew > $maxX) {$maxX = $xNew;}
      if ($yNew < $minY) {$minY = $yNew;}
      if ($yNew > $maxY) {$maxY = $yNew;}
      # Make arrays of points for this segment to be used by LandMacro
      push (@Xs,$xNew);
      push (@Ys,$yNew);
    }  # End of skipping if the last point read was the same as the previous one
  } elsif (defined(@Xs) ) {  # Do only if data points have already been read
    # Write macro commands to draw the segment of boundary with arrays @Xs and @Ys
    $nPoints = @Xs - 2;
    print MACRO ("S=L(D,G,C)\nN=Array(" );
    foreach $i (0 .. $nPoints) {
      # Values are multiplied by 100 to convert to 100ths of mm, and y-value is made
      # negative because OOo Draw uses y positive downwards.
      printf MACRO ("P(%.0f,%.0f),_\n", $Xs[$i] * 100, -$Ys[$i] * 100);
    }
    # Last point does not end in line continuation
```

```
    printf MACRO ("P(%.0f,%.0f))\n", $Xs[$nPoints+1] * 100, -$Ys[$nPoints+1] * 100);
    print MACRO ("S.PolyPolygon = Array(N)\n");

    @Xs = ();  @Ys = ();  # Empty arrays, ready for next segment
    $oldLong = 99999;  $oldLat = 99999;  # Reset previous values to impossible values
   }
  }
  close (DATA);
  if ($Line ne "\# -b") { # Draw last segment, if it hasn't been drawn
   # Write macro commands to draw the segment of boundary with arrays @Xs and @Ys
   $nPoints = @Xs - 2;
   print MACRO ("S=L(D,G,C)\nN=Array(" );
   foreach $i (0 .. $nPoints) {
    # Values are multiplied by 100 to convert to 100ths of mm, and y-value is made
    # negative because OOo Draw uses y positive downwards.
    printf MACRO ("P(%.0f,%.0f),_\n", $Xs[$i] * 100, -$Ys[$i] * 100);
   }
   # Last point does not end in line continuation
   printf MACRO ("P(%.0f,%.0f))\n", $Xs[$nPoints+1] * 100, -$Ys[$nPoints+1] * 100);
   print MACRO ("S.PolyPolygon = Array(N)\n");
  }
  # Add command to group all the coastlines
  print MACRO ("S = Collect(\"All\")\nS.Name = NS\n");
  # Add a blue rectangular line, the size of the whole M-map, to aid in resizing and
  # positioning the coastline on the map.
  print MACRO ("C=RGB(0,0,255)\nS=L(D,G,C)\nN=Array(");
  if ($Map eq "M") { # M-map coordinates requiring area for 8 octants
   print MACRO ("P(-2000000,-1000000),P(2000000,-1000000),_\n");
   print MACRO ("P(2000000,1000000),P(-2000000,1000000),_\n");
   print MACRO ("P(-2000000,-1000000))\nS.PolyPolygon = Array(N)\n");
  } else { # G's coordinates, referring to area for a single octant
   print MACRO ("P(0,-900000),P(1000000,-900000),_\n");
   print MACRO ("P(1000000,300000),P(0,300000),_\n");
   print MACRO ("P(0,-900000))\nS.PolyPolygon = Array(N)\n");
  }
  # Add command to group the coastlines and the rectangle, and end the macro
  print MACRO ("S = Collect(\"All\")\nS.Name = NSrect\nEnd Sub\' Sub NovaScotia\n");
  close (MACRO);
  print "Read a total of $nData lines, and wrote file",' "NSmacro.txt".', "\n";
  print $minX, ' <= x <= ',$maxX," and ",$minY, ' <= y <= ', $maxY, "\n";

}  # End of skipping making macro for Coastal Data

print "All done.\n";


# - - - - - - - - - - S U B R O U T I N E S - - - - - - - - -
sub Preliminary{
  # Calculates and returns an array with 29 values (0 to 28), in this order:
  # (for use of subs MJtoG and Rotate) $sin60, $cos60, $yTranslate,
  # (x and y coordinates of points) $xA, $yA, $xB, $yB, $xC, $yC, $xD, $yD, $xE, $yE,
  # $xF, $yF, $xG, $yG, $xM, $yM, $xT, $yT, (lengths) $AG, $AB, $BD, $GF, $BDE, $GFE,
  # $R, $DeltaMEq.

  use Math::Trig;
  # Values that will be returned
```

```
my ($sin60, $cos60, $yTranslate);
my ($xA, $yA, $xB, $yB, $xC, $yC, $xD, $yD, $xE, $yE, $xF, $yF, $xG, $yG, $xM, $yM);
my ($xT, $yT, $AG, $AB, $BD, $GF, $BDE, $GFE, $R, $DeltaMEq);
# Variables temporary to this sub
my ($xN, $yN, $MB, $MN, $xU, $yU, $k, $xV, $yV);

# Some constants for use by subs MJtoG and Rotate, which do coordinate axis
# transformation. Angle of rotation is 60°.  Point G is (10000,0) in MJ, and (5000,0) in G
$sin60 = sin (deg2rad(60));
$cos60 = cos (deg2rad(60));
$yTranslate = 10000 * $sin60;

# Given input
$xM = 0;      $yM = 0;      # Point M is the origin of the axes
$xG = 10000;      $yG = 0;      # Point G, at center of base of octant
$xA = 940;      $yA = 0;      # Point A at apex of octant
# Other points and lengths of interest, relating to scaffold triangle and half-octant
$xN = $xG;      $yN = $xG * tan (deg2rad(30));      # Point N, point of triangle MNG
($xB, $yB) = LineIntersection ($xM, $yM, 30, $xA, $yA, 45);      # Point B
$AG = $xG - $xA;
$AB = Length ($xA, $yA, $xB, $yB);
$MB = Length ($xM, $yM, $xB, $yB);
$MN = Length ($xM, $yM, $xN, $yN);
# Calculate point D, considering that length DN = MB
$xD = Interpolate ($MB, $MN, $xN, $xM);      # D is away from N as B is away from M
$yD = Interpolate ($MB, $MN, $yN, $yM);
$xF = $xG;
$yF = $yN - $MB;
# Distance from point E to point N = distance from point A to point M = xA; calculate E
$xE = $xN - $xA * sin (deg2rad(30));
$yE = $yN - $xA * cos (deg2rad(30));
$GF = $yF;
$BD = Length ($xB, $yB, $xD, $yD);
$BDE = $BD + $AB;      # Length AB = length DE
$GFE = $AB + $GF;      # Length AB = length FE
$DeltaMEq = $GFE / 45;      # 45 meridian spacings along equator for half an octant
# Calculate Point T: First calculate point U = (30°,73°). Radius to circular arc of 73° =
# 15° x 104mm/° + 2° x 100 mm/° = 1760 mm.
$xU = $xA + 1760 * cos (deg2rad(30));
$yU = 1760 * sin (deg2rad(30));
# Point T is at intersection of line BD with line from point U perpendicular to BD.
# Since line BD is 30° from horizontal, perpendicular line is -60° from horizontal.
($xT, $yT) = LineIntersection ( $xU, $yU, -60, $xB, $yB, 30);

# To calculate point C, must first calculate point V = (29°, 15°).
# First calculate joints of meridian 29°
($xJe, $yJe, $xJt, $yJt, $xJf, $yJf, $Lt, $Lm) =
   Joints (29, $xA, $xE, $yE, $xF, $yF, $xG, $AB, $GF, $DeltaMEq);
# Next need point on parallel 73° for this meridian 29°; really, only $Lf is needed.
($xP73, $yP73, $Lf) = Parallel73 (29, $xA, $xT, $yT, $xJf, $yJf);
# Do something with $xP73 and $yP73, only so that the compiler doesn't complain
# that they were used only once; I really don't need them now.
$xP73 = 1 * $xP73; $yP73 = 1 * $yP73;
# Parallels are equally spaced between the equator and latitude 73° in this zone.
# Both torrid and frigid joints are at latitudes lower than 73° in this region.
# To find point V, calculate length from equator to parallel 15°, along meridian.
```

```
  # ($Lt + $Lm + $Lf) = length from equator to parallel 73° on meridian 29°.
  $L = 15 * ($Lt + $Lm + $Lf) / 73;
  if ($L <= $Lt) {
    # Measure length along the torrid segment, from the equator
    $xV = Interpolate ($L, $Lt, $xJe, $xJt);
    $yV = Interpolate ($L, $Lt, $yJe, $yJt);
  } else {
    # Measure length along the middle segment, from the torrid joint
    $L = $L - $Lt;
    $xV = Interpolate ($L, $Lm, $xJt, $xJf);
    $yV = Interpolate ($L, $Lm, $yJt, $yJf);
  }
  # Point C is the center of circular arc for parallel 15° with ends at points D and V, and,
  # therefore, it is equidistant from both. Radius, R = CD = CV. Thus:
  # $R^2 = ($xD - $xC)^2 + ($yD - $yC)^2 = ($xV - $xC)^2 + ($yV - $yC)^2
  # Point C is also on line MD, which has angle 30° with horizontal. M = (0 mm, 0 mm).
  # Thus, $yC / $xC = tan(deg2rad(30)) = 1 / sqrt(3) ; letting $k = sqrt(3), last equation is
  # equivalent to $xC = $k * $yC. Replacing this in the first equation and solving for $yC,
  # yields:
  $k = sqrt(3);
  $yC = ($xV * $xV + $yV  * $yV - $xD * $xD - $yD * $yD) /
        (2 * ($k * $xV + $yV  - $k * $xD - $yD ) );
  $xC = $k * $yC;
  $R = Length ($xC, $yC, $xD, $yD);

  # Return values needed by main program
  return ($sin60, $cos60, $yTranslate, $xA, $yA, $xB, $yB, $xC, $yC, $xD, $yD, $xE, $yE,
     $xF, $yF, $xG, $yG, $xM, $yM, $xT, $yT, $AG, $AB, $BD, $GF, $BDE, $GFE, $R,
     $DeltaMEq);
} # End of sub Preliminary

sub Equator {
  # Sub calculates equatorial point for a meridian, and returns ($xJe, $yJe).
  # Input is the wanted meridian, $m, and the following values calculated in sub Preliminary:
  # $xE, $yE, $xF, $yF, $xG, $AB, $GF, $DeltaMEq
  use Math::Trig;
  my ($m,  $xE, $yE, $xF, $yF, $xG, $AB, $GF, $DeltaMEq) = @_;  # Input arguments
  my ($xJe, $yJe);  # Values to be returned
  my ($L);  # Variable used just within this sub

  # Calculate point Je, the Intersection of meridian with equator, as in zone (d)
  $L = $DeltaMEq * $m;
  if ($L <= $GF) {
    $xJe = $xG;
    $yJe = $L
  } else {
    # Past point F; find point on line FE, a distance L from point G, along equator.
    # Length FE = length AB
   $L = $L - $GF;    # Part of length on segment FE
    $xJe = Interpolate ($L, $AB, $xF, $xE);
    $yJe = Interpolate ($L, $AB, $yF, $yE);
  }
  return ($xJe, $yJe);
}  # End sub Equator

sub Joints {
```

```
  # Sub calculates equatorial, torrid and frigid joints for given meridian, and lengths of
  # middle segments. Returns are array: ($xJe, $yJe, $xJt, $yJt, $xJf, $yJf, $Lt, $Lm).
  # $xJe and $yJe are calculated by calling sub Equator.
  # Input is the wanted meridian, $m, and the following values calculated in sub Preliminary:
  # $xA, $xE, $yE, $xF, $yF, $xG, $AB, $GF, $DeltaMEq
  use Math::Trig;
  my ($m, $xA, @Prelims) = @_;  # Input arguments
  # Parse the input arguments
  my ($xE, $yE, $xF, $yF, $xG, $AB, $GF, $DeltaMEq) = @Prelims ;

  my ($xJe, $yJe, $xJt, $yJt, $xJf, $yJf, $Lt, $Lm);  # Values to be returned
  my ($L);  # Variable just within this sub

  # Calculate point Je, the Intersection of meridian with equator
  ($xJe, $yJe) = Equator ($m, @Prelims);

  # Calculate torrid joint, Jt, the intersection of line of angle ($m/3) starting at point Je
  # with line of angle (2/3 * $m) starting at point M = (0 mm, 0 mm)
  ($xJt, $yJt) = LineIntersection (0, 0, 2*$m/3, $xJe, $yJe, $m/3);

  # Calculate frigid joing, Jf, the intersection of line of angle ($m) starting at point A
  # with line of angle (2/3 * $m) starting at point M. Point A = ($xA mm, 0 mm).
  ($xJf, $yJf) = LineIntersection ($xA, 0, $m, 0, 0, 2*$m/3);

  # Calculate lengths of torrid segment, $Lt = Je to Jt, and of middle segment, $Lm = Jt to Jf
  $Lt = Length ($xJe, $yJe, $xJt, $yJt);
  $Lm = Length ($xJt, $yJt, $xJf, $yJf);

    return ($xJe, $yJe, $xJt, $yJt, $xJf, $yJf, $Lt, $Lm);
} # End sub Joints

sub Parallel73 {
  # Sub calculates parallel 73° for a meridian, and length from that point to the frigid joint.
  # Note: if the point is on the middle segment, the length, Lf, to the frigid joint is given as
  # a negative number; this only happens for some of the meridians between 44° and 45°.
  # Returns are ($xP73, $yP73, $Lf).
  # Input is the wanted meridian, $m; $xA, $xT, and $yT, calculated in sub Preliminary;
  # $xJf, and $yJf, the frigid joint, calculated in sub Joints.
  use Math::Trig;
  my ($m, $xA, $xT, $yT, $xJf, $yJf) = @_;  # Input arguments
  my ($xP73, $yP73, $Lf);  # Values to be returned
  my ($x, $y);  # Values used only in this sub
  # Calculate point P73 = ($m, 73°) and length $Lf = distance from Jf to P73 (negative if
  # on middle segment).
  if ($m <= 30) {
    # Circular arc portion:
    # Radius to circular arc of 73° = 15° x 104mm/° + 2° x 100 mm/° = 1760 mm.
    $xP73 = $xA + 1760 * cos (deg2rad($m));
    $yP73 = 1760 * sin (deg2rad($m));
    # Calculate length $Lf = distance from point Jf to point P73
    $Lf = Length ($xJf, $yJf, $xP73, $yP73);
  } else {
    # Straight portion of parallel 73°. Calculate point P73, at the intersection of line UT
    # (angled -60° with the horizontal) with frigid segment of meridian $m, which is
    # angled +$m ° and passes through point . Point U = (30°, 73°) was
    # used to calculate point T, in sub Preliminary.
```

```
    ($xP73, $yP73) = LineIntersection($xT, $yT, -60, $xJf, $yJf, $m);

    # Calculate length $Lf, from point Jf to point P73
    $Lf = Length ($xJf, $yJf, $xP73, $yP73);
    if ($m > 44) {
      # Point P73 is on middle meridian segment for some of these meridians; check if it is.
      # Calculate intersection of line UT with middle segment, angled +(2/3*$m)°.
      ($x, $y) = LineIntersection ($xT, $yT, -60, $xJf, $yJf, (2/3*$m) );
      if ($x > $xP73) {
        # Correct intersection is on middle segment; correct point and length
        $xP73 = $x;
        $yP73 = $y;
        $Lf = - Length ($xJf, $yJf, $xP73, $yP73);  # Recalculating length and making it negative
      }  # End of correction
    }  # End of checking if it is on middle segment
  }
  return ($xP73, $yP73, $Lf);
}  # End sub Parallel73

sub MPtoXY {
  # Sub converts half-octant meridian,parallel to x,y coordinates.
  # Arguments are meridian, parallel, and array output by sub Preliminary not including
  # its first 3 values.
  # Sub returns (x,y).
  use Math::Trig;
  my ($m, $p, @Prelims) = @_;  # Input arguments
  # Parse the input arguments
  my ($xA, $yA, $xB, $yB, $xC, $yC, $xD, $yD, $xE, $yE, $xF, $yF, $xG, $yG,
      $xM, $yM, $xT, $yT, $AG, $AB, $BD, $GF, $BDE, $GFE, $R, $DeltaMEq) = @Prelims ;
  my ($x, $y);  # Variables to be returned

  # Extra variables used in this sub
  my ($L, $xP73, $yP73, $xP75, $yP75, $xJe, $yJe, $xJt, $yJt, $xJf, $yJf, $f73, $f75,
      $Lt, $Lm, $Lf, $L73, $xPm, $yPm, $flag);

  if ($m == 0) {  # Zones (a) and (b) on the center line of octant, on the base of
    # scaffold half-triangle
    $y = 0;
    if ($p >= 75) {  # Zone (a) – frigid center line
      # Parallel spacing is 104 mm/° from 75° to 90° of latitude, measured from point A (pole)
      $x = $xA + (90 - $p) * 104;

    } else {  # Zone(b) – torrid/temperate center line
      # Parallel spacing is 100 mm/° from 0° to 75° of latitude; measure from equator, that is
      # from point G.
      $x = $xG - $p * 100;
    }

  } elsif ($p >= 75) {  # Zone (c) – polar region with circular parallels spaced 104 mm/°
    # Meridian $m starts at point A and makes angle $m (in degrees) with line AG
      $L = 104 * (90 - $p);
      $x = $xA + $L * cos ( deg2rad($m) );
      $y = $L * sin( deg2rad($m) );

  } elsif ($p == 0) {  # Zone (d) – equator
    ($x, $y) = Equator ($m, $xE, $yE, $xF, $yF, $xG, $AB, $GF, $DeltaMEq);
```

```
} elsif ($p >= 73 && $m <= 30) {  # Zone (e) – frigid region with circular parallels
  # spaced 100 mm/° between 73° and 75° of latitude; meridian $m starts at point A
  # and makes angle $m with line AG.
  # Length from A to parallel 75° = 1560 mm = 104 mm/° x (90° - 75°).
  $L = 1560 + (75 - $p) * 100;
  $x = $xA + $L * cos ( deg2rad($m) );
  $y = $L * sin ( deg2rad($m) );

} elsif ($m == 45) {  # Outer boundary of octant, zones (f), (g), and (h)
  if ($p <= 15) {  # Zone (f) – torrid zone of outer boundary, that is, along line ED
    # E = 0° and D = 15° of latitude. Parallels are equally spaced within this zone.
    $x = Interpolate ($p, 15, $xE, $xD);
    $y = Interpolate ($p, 15, $yE, $yD);

  } elsif ($p <= 73) {  # Zone (g) – temperate zone of outer boundary, that is, along DT.
    # D = 15°; T = 73°. Parallels are equally spaced within this zone. 73° - 15° = 58°
    $L = $p - 15;
    $x = Interpolate ($L, 58, $xD, $xT);
    $y = Interpolate ($L, 58, $yD, $yT);

  } else {  # Zone (h) – frigid supple zone of outer boundary
    # Calculate point P75 = (45 °, 75°), point at parallel 75° on this meridian
    # Length from A to parallel 75° = 1560 mm = 104 mm/° x (90° - 75°).
    $xP75 = $xA + 1560 * cos (deg2rad(45));
    $yP75 = 1560 * sin (deg2rad(45));

    # Calculate length $Lf = parallel 73 (which is point T) to frigid joint (point B)
    $Lf = Length ($xT, $yT, $xB, $yB);
    # Calculate length from $Lf75 = distance from frigid joint (point B) to point P75
    $Lf75 = Length ($xB, $yB, $xP75, $yP75);
    # Length from P75 to P73 covers 2°
    $L = (75 - $p) * ($Lf75 + $Lf) / 2;  # Distance from parallel 75° to parallel p°
    if ($L <= $Lf75) {
      # Wanted latitude is on frigid segment, parallel 75° (P75) to B
      $x = Interpolate ($L, $Lf75, $xP75, $xB);
      $y = Interpolate ($L, $Lf75, $yP75, $yB);
    } else {
      # Wanted latitude is on segment B to T
      $L = $L - $Lf75;
      $x = Interpolate ($L, $Lf, $xB, $xP73);
      $y = Interpolate ($L, $Lf, $yB, $yP73);
    }
  }  # End of zones (f), (g), and (h); more specifically, end of zone (h)
} else {  # Zones (i), (j), (k), and (l) which require more complicated calculations
  # Need to calculate meridian joints and segment lengths for this meridian.
  ($xJe, $yJe, $xJt, $yJt, $xJf, $yJf, $Lt, $Lm) =
      Joints ($m, $xA, $xE, $yE, $xF, $yF, $xG, $AB, $GF, $DeltaMEq);

  # Calculate point P73 = ( $m, 73°), point at latitude 73° on this meridian and distance
  # from that point to frigid joint, $Lf. These may later be modified for zones (j), (k) and (l).
  ($xP73, $yP73, $Lf) = Parallel73 ($m, $xA, $xT, $yT, $xJf, $yJf);

  if ($m <= 29) {  # Zone (i) – torrid and temperate areas of central two-thirds of octant
    # Parallels are equally spaced between the equator and latitude 73° in this zone.
    # Both torrid and frigid joints are at latitudes lower than 73° in this region.
```

```
    # Calculate length from equator to point ($m, $p), along this meridian $m.
    $L = $p * ($Lt + $Lm + $Lf) / 73;
    if ($L <= $Lt) {
      # Measure length along the torrid segment, from the equator
      $x = Interpolate ($L, $Lt, $xJe, $xJt);
      $y = Interpolate ($L, $Lt, $yJe, $yJt);
    } elsif ($L <= ($Lt + $Lm)) {
      # Measure length along the middle segment, from the torrid joint
      $L = $L - $Lt;
      $x = Interpolate ($L, $Lm, $xJt, $xJf);
      $y = Interpolate ($L, $Lm, $yJt, $yJf);
    } else {
      # Measure length along the frigid segment
      $L = $L - $Lt - $Lm;
      $x = Interpolate ($L, $Lf, $xJf, $xP73);
      $y = Interpolate ($L, $Lf, $yJf, $yP73);
    }    # end of area (i)

  } else {  # Supple zones (j), (k), and (l): 29° < $m < 45° and 0° < $p < 73

    if ($p >= 73) {  # Zone (j) – frigid supple zone
      # Calculate point P75 = ($m °, 75°), point at parallel 75° on this meridian
      # Length from A to parallel 75° = 1560 mm = 104 mm/° x (90° - 75°).

      $xP75 = $xA + 1560 * cos (deg2rad($m));
      $yP75 = 1560 * sin (deg2rad($m));
      # Calculate length from $Lf75 = distance from frigid joint, Jf, to point P75
      $Lf75 = Length ($xJf, $yJf, $xP75, $yP75);
      # Length from P75 to P73 covers 2°; remember that Lf, from P73 to Jf is sometimes
      # negative for a few meridians between 44° and 45°.
      $L = (75 - $p) * ($Lf75 - $Lf) / 2;  # Distance from parallel 75° to parallel p°
      if ($L <= $Lf75) {
        # Wanted latitude is on frigid segment
        $x = Interpolate ($L, $Lf75, $xP75, $xJf);
        $y = Interpolate ($L, $Lf75, $yP75, $yJf);
      } else {
        # Wanted latitude is on middle segment
        $L = $L - $Lf75;
        $x = Interpolate ($L, -$Lf, $xJf, $xP73);
        $y = Interpolate ($L, -$Lf, $yJf, $yP73);
      }

    } else {  # Zones (k) and (l)
      # Calculate point P15 = (m, 15°), that is, point on this meridian at latitude 15°, which
      # is at intersection of meridian with circular arc of center C and radius R. Also
      # calculate length L15 = distance from equator (Je) to P15.
      # Try middle segment first, since most, if not all, parallel 15° points are in this segment
      ($flag, $xP15, $yP15) = CircleLineIntersection ($xC, $yC, $R, $xJt, $yJt, $xJf, $yJf);
      if ($flag == 1) {    # Found the intersection point in middle segment
        $L15 = $Lt + Length ($xJt, $yJt, $xP15, $yP15);
      } else {    # Intersection point is in torrid segment
        ($flag, $xP15, $yP15) = CircleLineIntersection ($xC, $yC, $R, $xJe, $yJe, $xJt, $yJt);
        if ($flag==0) {    # Hmmm... no intersection!
          print " no line-circular arc intersection for M $m, at parallel 15!";
          die;
        }
```

```
        $L15 = $Lt - Length ($xJt, $yJt, $xP15, $yP15);
      }

      if ($p <= 15) {  # Zone (k) – torrid supple zone
        # Parallels equally spaced between equator and 15°
        $L = $p * $L15 / 15;
        if ($L <= $Lt) {  # Point is in torrid segment
          $x = Interpolate ($L, $Lt, $xJe, $xJt);
          $y = Interpolate ($L, $Lt, $yJe, $yJt);
        } else {  # Point is in middle segment
          $L = $L - $Lt;
          $x = Interpolate ($L, $Lm, $xJt, $xJf);
          $y = Interpolate ($L, $Lm, $yJt, $yJf);
        }

      } else {  # Zone (l) – middle supple zone
        # Parallels equally spaced between 15° and 73°.
        # Will measure from the equator. ($Lt+$Lm+$Lf) = equator to P73. 58° = 73° - 15°
        $L = $L15 + ($p - 15) * (($Lt + $Lm + $Lf) - $L15) / 58;
        if ($L <= $Lt) {  # On torrid segment
          $x = Interpolate ($L, $Lt, $xJe, $xJt);
          $y = Interpolate ($L, $Lt, $yJe, $yJt);
        } elsif ($L <= $Lt + $Lm) {  # On middle segment
          $L = $L - $Lt;
          $x = Interpolate ($L, $Lm, $xJt, $xJf);
          $y = Interpolate ($L, $Lm, $yJt, $yJf);
        } elsif ($L <= $Lt + $Lm) {  # On middle segment
          $L = $L - $Lt - $Lm;
          $x = Interpolate ($L, $Lf, $xJf, $xP73);
          $y = Interpolate ($L, $Lf, $yJf, $yP73);
        }

      } # end zones (k) and (l)
    } # end zones (j), (k), and (l)
  } # end zones (i), (j), (k), and (l)
 } # end all zones
 return ($x, $y);
} # End of sub MPtoXY

sub LineIntersection {  # Written 2010-02-28; modified 2010-11-28
# Subroutine/function to calculate coordinates of point of intersection of two lines which
# are given by a point on the line and the line's slope angle in degrees.
#
# 2010-11-28 – Modified to assume that the lines do intersect, neither is either horizontal
#  or vertical, the arguments are the correct number and are all defined, and the
#  angles are within [-180,180].
#  Unlike on the previous version, this one has no checks and doesn't return a flag.
#
# Return is an array of two values, the x and y coordinates of the point of intersection.
#
# Arguments should be 6, in this order:
#  x and y coordinates of point of first line; slope of first line in degrees;
#  x and y coordinates of point of second line; slope of second line in degrees;
#
# Equations used are from: slope of line = tangent angle = delta-y / delta-x, and the fact
#  that intersection point x,y is on both lines.
```

```
  use Math::Trig;
  my ($nArguments,$xp,$yp,$m1,$m2);
  $nArguments=@_ ;
  my ($x1,$y1,$angle1,$x2,$y2,$angle2) = @_ ;

  $m1 = tan(deg2rad($angle1));
  $m2 = tan(deg2rad($angle2));
  $xp = ($m1 * $x1 - $m2 * $x2 - $y1 + $y2) / ($m1 - $m2);
  $yp = $m1 * $xp - $m1 * $x1 + $y1;
  return ($xp,$yp);
} # End of sub LineIntersection

sub Length {
  # Input are x1,y1,x2,y2
  my ($x1,$y1,$x2,$y2) = @_;
  return sqrt( ($x1-$x2)**2 + ($y1-$y2)**2);
} # End of sub Length

sub Interpolate {
# Inputs are 4: length wanted, of total-segment-length, start, end;
# Total-segment-length is different from (end - start); end and start may be x-coordinates,
# or y-coordinates, while length takes into account the other coordinates.
# (End - Start ) / Length = (Wanted - Start) / NewLength
# Returns single value: Wanted.
  my ($NewLength, $Length, $Start, $End) = @_;
  my ($Wanted);
  $Wanted = $Start + ($End - $Start) * $NewLength / $Length;
  $Skip = "YES";
  unless ($Skip) {  # Option to skip printing arguments
    foreach $i ($NewLength, $Length, $Start, $End, $Wanted) {printf "\t%.5f", $i;}
    print "\n";
  } # End of skipping printing arguments and result
  return $Wanted;
}  # End of sub Interpolate

sub CircleLineIntersection {
  # Subroutine to calculate intersection of circle with line segment. Equations from
  # http://local.wasp.uwa.edu.au/~pbourke/geometry/sphereline/
  # Arguments are 7, in the following order:
  # Circle given as x-center, y-center, radius; line segment given as (x1,y1), (x2,y2).
  # If line segment does not intersect circle, return is 0; else, return is 1,x,y of
  # point of intersection; it is assumed that circle only intersects line segment at one
  # point. If you want a subroutine for other purposes, read that website.
  my ($n);
  $n = @_;
  if ( $n != 7) {
    print "Sub CircleLineIntersection requires 7 arguments but got $n.\n";
    return 0;
  }
  my ($xc,$yc,$r,$x1,$y1,$x2,$y2) = @_;
  my ($u1,$u2,$a,$b,$c,$d,$x,$y);
  # Check if there is a point of intersection
  $a = ($x2-$x1)**2 + ($y2-$y1)**2;
  $b = 2 * ( ($x2-$x1) * ($x1-$xc) + ($y2-$y1) * ($y1-$yc) );
  $c = $xc**2 + $yc**2 + $x1**2 + $y1**2 - 2 * ($xc*$x1 + $yc*$y1) - $r**2;
  $d = $b**2 - 4*$a*$c;          # Determinant
```

```
  if ($a == 0) {
    # print "In sub CircleLineIntersection: line given is just one point!\n";
    return 0;
  }elsif ($d < 0) {
    # Determinant is negative: circle does not intersect the line, much less the
    # segment
    # print "In sub CircleLineIntersection: line doesn't intersect circle.\n";
    return 0;
  }
  # $u1 and $u2 are the roots to a quadratic equation
  $u1 = (-$b + sqrt($d) ) / (2*$a);      # + of +/- of the solution to the quadratic equation
  $u2 = (-$b - sqrt($d) ) / (2*$a);      # - of +/- of the solution to the quadratic equation

  # Check if there is an intersection and if it is within the line segment (not only the line)
  # If $u1=$u2, line is tangent to the circle; if $u1 != $u2, line intersects circle at two
  # points; however, point or points of intersection are within the line segment only if
  # the root is within interval [0,1].
  if (0 <= $u1 && $u1 <= 1) {  # This root is on the line segment; use it
    $x = $x1 + $u1 * ($x2 - $x1);
    $y = $y1 + $u1 * ($y2 - $y1);
    return 1,$x,$y;
  } elsif (0 <= $u2 && $u2 <= 1) {
    # 1st root was not on line segment but 2nd one is; use it
    $x = $x1 + $u2 * ($x2 - $x1);
    $y = $y1 + $u2 * ($y2 - $y1);
    return 1,$x,$y;
  } else {  # neither root is on line segment
    # print "In sub CircleLineIntersection: line segment doesn't intersect circle.\n";
    return 0;
  }
}  # End of sub CircleLineIntersection

# - - - - - - - - - -   SUBROUTINES For Coordinate conversion  - - - - - - - - - -

sub LLtoMP {
  # Arguments are real world longitude and latitude for one point, in decimal degrees.
  # West longitudes and south latitudes have negative values.
  # Returns corresponding meridian ($m) and parallel ($p) in MJ's template half-octant,
  # sign for meridian (-1 for western half octant and +1 for eastern one), and octant
  # number. Returned values of $m and $p are always positive.

  my ($Long, $Lat) = @_ ;  # Input values

  # $m and $p are the meridian and parallel numbers in template half-octant setting;
  # $Octant is the M-map octant of the real point; $Sign is for east or west side of
  # template octant;
  my ($m, $p, $Sign, $Octant);  # Values to be returned

  # @South are southern octants corresponding to northern octants 1, 2, 3 and 4; the 0
  # is just a place holder to facilitate correspondence.
  my (@South) = (0,6,7,8,5);  # Variables used only in this sub

  # Determine the correct octant; Octant 1 is +160° to -110°; octant 4 is 70° to 160°
  $Octant = int ( ($Long + 200) / 90 ) + 1;
  # Make longitude fit within template half-octant, and determine if y value should
  # be positive or negative.
```

```
  $m = ( ($Long + 200) - (90*($Octant - 1))) - 45;
  if ($m < 0) {
    $Sign = -1;
    $m = -$m;
  } else {
    $Sign = 1;
  }
  # Fix the octant number, if necessary
  if ($Octant == 5) { $Octant = 1; }
  if ($Lat < 0) {
    $Octant = $South [$Octant];
    $p = -$Lat;
  } else {
    $p = $Lat;
  }
  return ($m, $p, $Sign, $Octant);
}  # End sub LLtoMP

sub MJtoG {
# Subroutine to convert (that is, do coordinate transformation of) x and y coordinates
# from Mary Jo's half-octant on its side to Gene's leaning, single octant coordinates, or
# to Gene's M-map (eight-octants) coordinates.
#
# Subroutine returns converted x and y coordinates.
#
# Arguments are:
# - x and y coordinates of point to convert.
# - Third argument is the Octant to convert to:
#   - 0 – Gene's single-octant system, with y-axis on its left;
#   - 1, 2, 3 or 4 – convert to M-map coordinates, respectively to first, second, third or
#     fourth northern octant, from the left;
#   - 5, 6, 7 or 8 – convert to M-map coordinates, respectively to fourth, first, second or
#     third southern octant from the left.
# - $sin60, $cos60, $yTranslate – values calculated once, in sub Interpolate, to minimize
#     computations. ($sin60 = sin 60°, $cos60 = cos 60°, $yTranslate = 10,000 * sin 60°).
#
# - In MJ's coordinates, point M is the origin, at (0,0), points M, A and G are on the positive
# x-axis, and point G is at (10000, 0).
# - In G's system, point M, L, J and P are on the positive y-axis, and point G is on the
# positive x-axis; in this system, point G is at coordinates (5000, 0).
# - From MJ's system to G's, there is a +60° rotation, and also a translation.
# - The M-map coordinate system is like G's system, except that the y-axis is 10000mm
# to the right, that is, the x-coordinates for the start octant are 10000mm smaller.
#
# I got the equations for rotation and translation from my pocketbook "The Universal
# Encyclopedia of Mathematics, with a Foreword by James R. Newman", ©1964 by
# George Allen and Unwin, Ltd.; translated from original German language edition,
# pages 152, 153.

  my ($nArgs, $xnew, $ynew);
  my ($x, $y, $Octant, $sin60, $cos60, $yTranslate) = @_ ;
  if (not defined ($Octant) ) { $Octant = 0; }
  if ($Octant == 0) {
    ($xnew, $ynew) = Rotate ($x, $y, 60, $sin60, $cos60);
  } elsif ($Octant == 1) {
    ($xnew, $ynew) = Rotate ($x, $y, 120, $sin60, $cos60);
```

```
    $xnew = $xnew - 10000;
  } elsif ($Octant == 2) {
    ($xnew, $ynew) = Rotate ($x, $y, 60, $sin60, $cos60);
    $xnew = $xnew - 10000;
  } elsif ($Octant == 3) {
    ($xnew, $ynew) = Rotate ($x, $y, 120, $sin60, $cos60);
    $xnew = $xnew + 10000;
  } elsif ($Octant == 4) {
    ($xnew, $ynew) = Rotate ($x, $y, 60, $sin60, $cos60);
    $xnew = $xnew + 10000;
  } elsif ($Octant == 5) {
    ($xnew, $ynew) = Rotate ((20000-$x), $y, 60, $sin60, $cos60);
    $xnew = $xnew + 10000;
  } elsif ($Octant == 6) {
    ($xnew, $ynew) = Rotate ((20000-$x), $y, 120, $sin60, $cos60);
    $xnew = $xnew - 10000;
  } elsif ($Octant == 7) {
    ($xnew, $ynew) = Rotate ((20000-$x), $y, 60, $sin60, $cos60);
    $xnew = $xnew - 10000;
  } elsif ($Octant == 8) {
    ($xnew, $ynew) = Rotate ((20000-$x), $y, 120, $sin60, $cos60);
    $xnew = $xnew + 10000;
  } else {
    print "Error converting to M-map coordinates; there is no $Octant octant!\n";
    return ($x,$y);
  }
  $ynew = $ynew + $yTranslate;
  return ($xnew, $ynew);

}  # End of sub MJtoG, which converts coordinates to octants on M-map

sub Rotate {
  # Receives 5 arguments: x, y, angle by which to rotate the coordinate system, and
  # sin 60° and cos 60°. The last two are calculated once in sub Preliminary, to minimize
  # computations.
  # Expects that the axes will be rotated either 60° or 120°.
  # Returns new x and y values.
  my ($x, $y, $angle, $sin60, $cos60) = @_ ;
  my ($xnew, $ynew);
  if ( $angle == 60 ) {
    $xnew = $x * $cos60 + $y * $sin60;
    $ynew = -$x * $sin60 + $y * $cos60;
  } elsif ( $angle == 120 ) {
    $xnew = -$x * $cos60 + $y * $sin60;
    $ynew = -$x * $sin60 - $y * $cos60;
  } else {
    print "Sub Rotate expected angle = 60 or 120 but received $angle.!\n";
  }
  return $xnew, $ynew;
}  # End of sub Rotate
# - - - - - - - - - -      End SUBROUTINES       - - - - - - - - - - - - - -
```