NAME-PPM
MD3/1(1)

Investigation of Latitude-Longitude Coordinate Transformations

Matthew Hort

12 January 2001

1 Summary

A new coordinate transformation routine has been written by David Thomson for NAME-PPM. The purpose of this work is to test and certify this code. The use of simple idealised i.e., analytically solvable, cases showed that the code was performing latitude longitude coordinate transformations correctly. However, during testing significant errors were found in the accuracy of the coordinate conversions. These were found to originate from two sources, namely; the inability of single precision to offer accuracy $> \mathcal{O}(10^0)$ m in a global $lat - long$ system and; inaccuracies in the Fortran inverse trigonometric functions when operated on values approaching ± 1, that under single and double precision resulted in errors of up to $\mathcal{O}(10^3)$ m and $\mathcal{O}(10^0)$ m respectively. These errors were eliminated by converting the code to double precision and deriving alternate conversion formula that avoided inverse trigonometric operations on values approaching ± 1.

2 Introduction

This note briefly reports on the evaluation of, and the resulting work on, 'coord' the coordinate transformation routine written by David Thomson for NAME-PPM. The purpose of the routine is to transform between two coordinate systems. The attention of this note is on the transformation between two latitude longitude systems. The code modules used in this study are included in the Appendix at the end of this note.

Within this note we shall firstly illustrate the standard and our example rotated coordinate system in Section 3. Then, very briefly, Section 4 will present the trigonometric formulations used within the code. Section 5 will present some results and in doing so highlight certain problems that were discovered with the original code. These will then be briefly discussed. Section 6 will present our proposed solution before the results from this are discussed in Section 7. Finally we shall conclude in Section 8.

3 Coordinate Systems

Within this note we are concerned with coordinate transformations between differing latitude longitude ($lat - long$) systems, the systems being distinguished through rotation in either or both latitude λ and longitude ϕ. Figure 1 shows a globe marked out as for the standard $lat - long$ coordinate system. Latitude is positive east and longitude positive north of (0, 0). For the purposes of this note we use only one rotated coordinate system, that being a rotation of (-90, 0) which is illustrated in Figure 2.

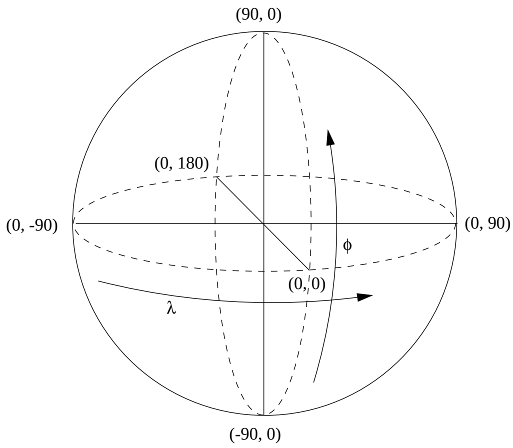

Figure 1: Standard latitude λ longitude ϕ coordinate system.

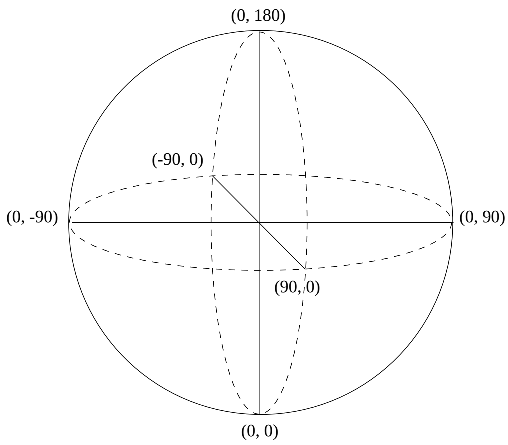

Figure 2: Rotated latitude λ longitude ϕ coordinate system.

4 Model Formulation

Within this section the formulation of the original trigonometric transformations will be outlined. This is adapted from notes by David Thomson.

From Abramowitz and Stegun (1972) (page 79) we have the following relationships for the solution of spherical triangles as shown in Figure 3.

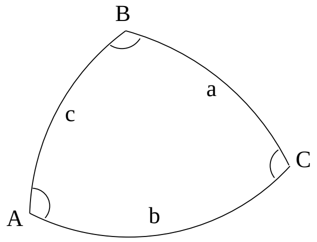

Figure 3: Spherical triangle.

$$\frac{A}{a} = \frac{B}{b} = \frac{C}{c} \tag{1}$$

$$\cos a = \cos b \cos c + \sin b \sin c \cos A \tag{2}$$

$$\cos A = -\cos B \cos C + \sin B \sin C \cos a \tag{3}$$

From equation (3) we can then derive:

$$\cos A = \frac{-\cos B \sin a \cos c + \sin c \cos a}{\sin b} \tag{4}$$

If we now consider two polar coordinate systems; the first standard and the second rotated in an arbitrary fashion, whose points are represented by the addition of $'$ to the notation. Figure 4 illustrates such a transformation.

Using the coordinate notation from Figure 4 and the spherical trigonometric relationships (1) to (3) and Figure 3 we are able to determine the transformed coordinates of a given point P in the following manner:

4.1 $(lat, long) \rightarrow (lat', long')$

From equation (2) we can calculate lat'

$$\begin{aligned} \sin(lat') = &\sin(lat)\sin(pole'lat) \\ &+ \cos(lat)\cos(pole'lat)\cos(long - pole'long). \end{aligned} \tag{5}$$

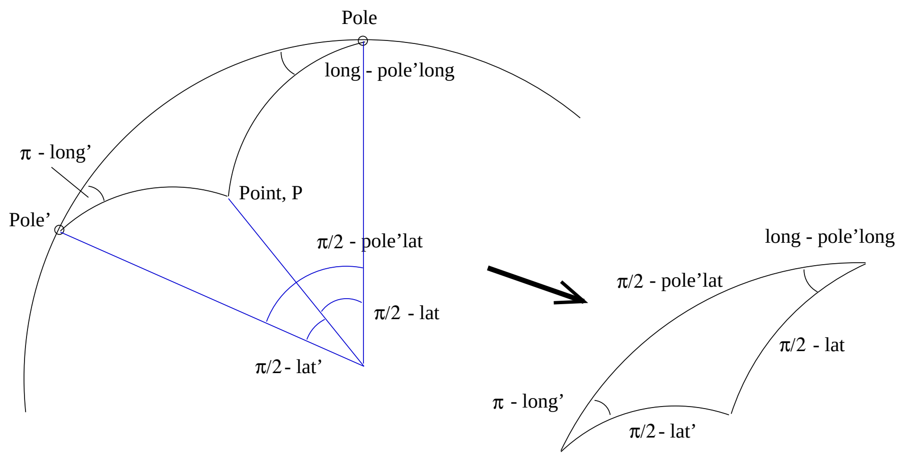

Figure 4: Illustration of relationship between original and transformed coordinate systems.

Having done this, equation (4) allows us to calculate $long'$

$$\cos(long') = \frac{\cos(long - pole'long)\cos(lat)\sin(pole'lat)}{\cos(lat')} - \frac{\sin(lat)\cos(pole'lat)}{\cos(lat')} \tag{6}$$

4.2 $(lat', long') \rightarrow (lat, long)$

Again, using equation (2) and (4), we can calculate lat and $long$;

$$\sin(lat) = \sin(lat')\sin(pole'lat) - \cos(lat')\cos(pole'lat)\cos(long') \tag{7}$$

$$\cos(long - pole'long) = \frac{\cos(long')\cos(lat')\sin(pole'lat)}{\cos(lat)} + \frac{\sin(lat')\cos(pole'lat)}{\cos(lat)} \tag{8}$$

5 Results

Testing of the $(lat, long)$ coordinate transformation was conducted using the program `main.f90`. For these the new coordinate system was obtained by rotating, as described in Section 3, the standard $(lat, long)$ system through 90° latitude, placing the pole of the new coordinate system on the equator of the old.

Initial tests consisting of single point transformations between $(lat, long)$ and $(lat', long')$, that had analytically simply determined transformed positions, demonstrated that this transformation worked. At this stage high precision i.e., $> 10^{-2}$ degrees output was not requested so the errors latter discovered were not evident at this point.

5.1 Is Single Precision Enough?

Derrick Ryall commented on the fact that within current NAME all coordinates are held in double precision as large errors were found to exist when only using single precision. To investigate this more extensively the test program 'main.f90' was adapted to perform a double transformation i.e., transforming the particles position into the rotated frame and then back into the original. This would enable the simple evaluation of an error in the representation of the particles position. In order to conduct a more extensive check the program was adapted to convert multiple particle positions along any arbitrary $(lat, long)$ line.

Under single precision the position accuracy after the double conversion was limited to machine single precision accuracy i.e., $10^{-6} \rightarrow 10^{-5}$ degrees. This is unfortunately not sufficient when the desired applications of NAME-PPM at short ranges are considered. Errors of order $(\mathcal{O})$ 10^{-5} degrees translate, based on an equatorial diameter of 12742458 m, into positional errors of $\mathcal{O}(1)$ meter. Conversion of the code to double precision solved this general problem. Conversion errors were now reduced to $< \mathcal{O}(10^{-13})$ degrees $\equiv < \mathcal{O}(10^{-8})$ m.

5.2 Errors in Fortran Trigonometric Functions

During our initial tests, under single precision, errors at specific coordinates of several orders of magnitude greater then the single precision limit had also been found. Conversion to double precision, while solving the general accuracy problems, had only reduced these specific errors to $\mathcal{O}(10^{-6})$ degrees. These errors, outlined next, were eventually attributed to the inaccuracy of the trigonometric functions for certain values.

Coordinate transformations in three regions resulted in large additional errors:

- Approaching and at 0 and ±180 degrees longitude.

- Approaching and at ±90 degrees latitude.

At *long* = 0 and 180 degrees, errors in longitude typically of $\mathcal{O}(10^{-6})$ in the converted and the returned 'original', although not necessarily both, coordinate positions were present. For coordinate transformation at longitudes significantly removed from 0 or 180 degrees errors were found to be $\leq \mathcal{O}(10^{-14})$ i.e., reduced to machine accuracy. Investigation of the extent of the region of increased error found that it was very limited. Table 1 shows how the error is negligible even at only one degree removed from the O and 180 degree longitude lines and has reached machine level accuracy by $5^\circ > long < 175^\circ$. These values are typical over almost the entire latitude range. However, larger errors, as reported next, were discovered when positions very near ($lat > 89^\circ$) the poles were investigated.

Longitude	Typical Error[1](degrees)
45	$\mathcal{O}(10^{-14})$
0	$\mathcal{O}(10^{-6})$
1	$\mathcal{O}(10^{-11})$
5	$\mathcal{O}(10^{-14})$
180	$\mathcal{O}(10^{-6})$
179	$\mathcal{O}(10^{-11})$
175	$\mathcal{O}(10^{-14})$

Table 1: Errors in rotated and returned longitudes.

This is the average error found over the entire range of latitude. Data taken at one degree latitude steps.

Investigation of near pole positions revealed that such locations introduced positional errors in their own right. In this case *long* and *lat* errors increased, although errors in *lat* remained $\leq \mathcal{O}(10^{-11})$. The magnitude of these errors was further affected by the longitude of the point. Points at *long* = 0 or ±180 resulted in larger errors than positions at intermediate longitudes. This is consistent with the findings presented previously within this Section. Table 2 presents the errors in degrees for a range of points in close proximity to the pole. For points not close to *long* = 0 or 180° the errors are negligible. At *long* = 0 or 180° the errors in degrees of longitude are quite large. Of course at such high latitudes such errors translate into short distances, indeed in the cases listed in Table 2 the positional error in meters is of $\mathcal{O}(10^{-2})$ for these positions.

Within this section we have shown evidence of systematic errors occurring during the conversion of $(lat, long)$ coordinates, when these coordinates are within certain regions. We shall now discuss the source of these errors.

Long	*Lat*	Error (degrees)
0 or 180	85.000	$\mathcal{O}(10^{-6})$
	89.000	$\mathcal{O}(10^{-6})$
	89.900	$\mathcal{O}(10^{-4})$
	89.950	$\mathcal{O}(10^{-4})$
	89.990	$\mathcal{O}(10^{-5})$
	89.999	$\mathcal{O}(10^{-2})$
$5 < long < 175$	85.000	$\mathcal{O}(10^{-15})$
	89.000	$\mathcal{O}(10^{-12})$
	89.900	$\mathcal{O}(10^{-10})$
	89.950	$\mathcal{O}(10^{-9})$
	89.990	$\mathcal{O}(10^{-7})$
	89.999	$\mathcal{O}(10^{-5})$

Table 2: Errors in returned longitudes for near pole latitudes.

Investigation of the above errors revealed that they occurred when trigonometric calculations were performed on certain coordinate positions. Specifically all the previously mentioned errors occurred when calculations of $\sin^{-1}$ and $\cos^{-1}$ of values very close to ± 1 occurred during the coordinate conversion. The error seemed to result from errors in the functions within the Fortran used to describe the inverse trigonometric functions. The reason for their significance for only such a limited range of values would appear to be due to the form of the functions near ± 1 where the returned angle varies like the square of the value. Therefore rounding errors of $\mathcal{O}(10^{-15})$ result in errors of $\mathcal{O}(10^{-7})$ degrees in the converted coordinate. This is consistent with the error values previously reported.

The alternative trigonometric expressions used to circumvent this issue and the rewriting of the code are dealt with in the following section.

6 Alternative Model Formulation

In light of the errors present, as outlined in the preceding section, within certain coordinate transformations, even under double precision, it was decided that an alternate formulation of the coordinate transformation be developed that avoided the noted problems.

The errors had been traced to the inverse trigonometric calculations contained within the functions H11S and H1S1. These calculations correspond to the calculation of $(lat, long)$ and $(lat', long')$ in equations (5)to (8). The solution adopted here was to continue using the current equations except where this would involve their operation on a value approaching ± 1. In this region alternate trigonometric derivations would be used that resulted in the application of different inverse trigonometric function to values not approaching ± 1. While this is computationally more expensive it is hopped that the limited area of effect and the unlikely nature of repeated coordinate transformations will mean that the impact on computational cost will be minimal while the increased robustness will be of considerable benefit.

The following formulation is adapted from notes by David Thomson, as in Section 4, and from Chapter 2 of the New Dynamics documentation (dated July 28, 2000) available on line at `http://www-nwp/~frax/public_NewDyDoc-UM5p1/goveqstrans.ps` (as of November 2000). The modified code is listed in Appendix A.

6.1 $(lat, long) \rightarrow (lat', long')$

Equations (5) and (6) are rewritten in the form

$$\sin(lat') = \underbrace{\sin(lat)\sin(pole'lat) + \cos(lat)\cos(pole'lat)\cos(long - pole'long)}_{\mathbf{Z}}. \qquad (9)$$

$$\cos(long')\cos(lat') = \underbrace{\cos(long - pole'long)\cos(lat) \times \sin(pole'lat) - \sin(lat)\cos(pole'lat)}_{\mathbf{X}} \qquad (10)$$

In addition, from equation (1) we can write

$$\sin(long')\cos(lat') = \underbrace{\sin(long - pole'long)\cos(lat)}_{\mathbf{Y}}. \qquad (11)$$

By taking the square of equations (10) and (11) and adding them we obtain

$$\cos^2(lat') = \mathbf{X}^2 + \mathbf{Y}^2. \qquad (12)$$

We first calculate $\mathbf{X}$, $\mathbf{Y}$ and $\mathbf{Z}$. Latitude can then be calculated from equation (10), if however $\mathbf{Z}$ is close to ±1 then an error is possible in the calculated lat'. In this case equation (12) can be used. $long'$ can then be determined from equation (9). However, if the subject of $\cos^{-1}$ is close to ±1 then an error is possible in the calculated $long'$. In this case equation (11) which in this region will operate on a values the is not approaching or equal to ±1 can be used.

6.2 $(lat', long') \rightarrow (lat, long)$

Similarly to the previous section, equations (7) and (8) are rewritten in the form

$$\sin(lat) = \underbrace{\sin(lat')\sin(pole'lat) - \cos(lat')\cos(pole'lat)\cos(long')}_{\mathbf{Z}} \tag{13}$$

$$\cos(long - pole'long)\cos(lat) = \underbrace{\cos(long')\cos(lat')\sin(pole'lat) + \sin(lat')\cos(pole'lat)}_{\mathbf{X}} \tag{14}$$

In addition, from equation (1) we can write

$$\sin(long - pole'long)\cos(lat) = \underbrace{\sin(long')\cos(lat)}_{\mathbf{Y}} \tag{15}$$

and by taking the square of equations (14) and (15) and adding them we obtain

$$\cos^2(lat') = \mathbf{X}^2 + \mathbf{Y}^2. \tag{16}$$

We first calculate $\mathbf{X}$, $\mathbf{Y}$ and $\mathbf{Z}$. Latitude can then be calculated from equation (14), if however $\mathbf{Z}$ is close to ±1 then an error is possible in the calculated lat'. In this case equation (16) will avoid this and can be used. $long$ can then be determined from equation (13). Again, if the subject of cos^{-1} is close to ±1 an error is possible in the returned position. In this case equation (15) can be used to avoid this.

6.3 Further Simplification

The formulation just outlined, while over coming the limitations of the initial code is not very elegant and IS computationally more expensive. A further development upon this approach that makes use of a certain Fortran function has been found to greatly simplify the coding and will also hopefully reduce the computational effort. This section shall briefly outline this.

If $S = \sin x$ and $C = \cos x$ then, based on the trigonometric identity $\tan x = \sin x / \cos x$, x can be expressed as $x = \tan^{-1}(S/C)$. This could be implemented to determine the latitude and longitude but the angle given is not unique and therefore the solution is open to error or requires additional code which will reduce transparency and efficiency. However, the Fortran function $\tan^{-1} 2$ (ATAN2) resolves this ambiguity. Using this function we can rewrite the previous expression for x as $x = \tan^{-1} 2(S, C)$, the result of which is the principle value of the argument, expressed in radians, in the range $-\pi < \tan^{-1} 2(S, C) \leq \pi$. The only proviso is that S and C must not both be zero.

6.3.1 (lat, $long$) $\rightarrow$ (lat', $long'$)

From equations (9) and (12) lat' can be determined in the following manner

$$\sin(lat') = \mathbf{Z} \tag{17}$$

$$\cos(lat') = \left(\mathbf{X}^2 + \mathbf{Y}^2\right)^{0.5} \tag{18}$$

which gives

$$lat' = \tan^{-1} 2\left(\mathbf{Z}, \left(\mathbf{X}^2 + \mathbf{Y}^2\right)^{0.5}\right) \tag{19}$$

Similarly from equations (10) and (11) $long'$ can be expressed as

$$long' = \tan^{-1} 2\left(\mathbf{Y}, \mathbf{X}\right) \tag{20}$$

6.3.2 (lat', $long'$) $\rightarrow$ (lat, $long$)

Following the approach of Section 6.3.1 lat and $long$ for the reverse conversion can be expressed as

$$lat = \tan^{-1} 2\left(\mathbf{Z}, \left(\mathbf{X}^2 + \mathbf{Y}^2\right)^{0.5}\right) \tag{21}$$

$$long = \tan^{-1} 2\left(\mathbf{Y}, \mathbf{X}\right) + pole'long \tag{22}$$

7 Alternative Model Results

Extensive tests of the new coordinate transformation routines, as given in Appendix A, were carried out. The routines were evaluated as for the original code i.e., after each transformation during a conversion to a rotated and back to a standard ($lat, long$) system. In all cases the errors were limited to $\leq \mathcal{O}(10^{-14})$; indeed in the majority of cases exact transformations were returned. For all positions, including previously identified 'sensitive' areas the new code achieved far greater accuracy than the original code. Transformations for positions defined outside of the bounds of normal *lat*, *long* bounds were also tested and found work as expected.

8 Conclusion

While testing the new coordinate transformation routines for NAME-PPM significant errors were found. These originated from two sources namely;

- the inability of single precision to offer accuracy $> \mathcal{O}(10^0)$ m in a global $(lat, long)$ system and;
- inaccuracies in the Fortran inverse trigonometric functions when operated on values approaching ± 1, that under single and double precision resulted in errors of up to $\mathcal{O}(10^3)$ m and $\mathcal{O}(10^0)$ m respectively.

These errors were reduced to $\leq \mathcal{O}(10^{-11})$ degrees by converting the code to double precision and through the implementation of alternate coordinate transformation formula that avoided inverse trigonometric operations on values approaching ± 1. The new code, listed in Appendix A was extensively tested over the entire globe.

A Program code

A.1 coord.f90

```
! Module: Coord Module
! Date:   4/7/00
! Author: Dave Thomson
! Files:  coord.f90

!Include 'Define.txt'

Module CoordModule

  Type :: HCoord_ ! Information defining horizontal coord system.
    Integer(4) CoordType ! 1 = latitude-longitude coord system with arbitrary
                         !     position for the coord system's north pole,
                         ! 2 = Cartesian coord system in a tangent plane.
    Double Precision    Pole(2)   ! For coord systems of type 1: position of the coord
                         !     system's north pole in a standard latitude-longitude
                         !     coord system, but with units defined by PoleScale and
                         !     latitude replaced by angle from the true north pole.
                         ! For coord systems of type 2: position of the tangent point
                         !     in standard latitude-longitude coords, but with units
                         !     defined by PoleScale and latitude replaced by angle
                         !     from the true north pole.
    Double Precision    Angle     ! For coord systems of type 1: rotation of coord
                         !     system from that with the same north pole location
                         !     and with the zero longitude line passing through
                         !     the true south  pole, in units defined by PoleScale.
                         ! For coord systems of type 2: angle between negative y axis
                         !     and the zero longitude line of a type 1 coord system
                         !     with the same Pole and Angle, in units defined by
                         !     PoleScale.
                         ! In each case a positive value means that, standing at the
                         ! origin or north pole and looking down, the system is
                         ! is rotated anticlockwise relative the its orientation for
                         ! a zero value.
    Double Precision    PoleScale ! Scaling of values of Pole and Angle relative to
                         ! radians. PoleScale > 1 means the values are bigger and
                         ! the units smaller.
    Double Precision    Origin(2) ! Offset of the origin relative to the tangent point
                         ! in units defined by Scale (defined for type 2 coord systems
                         ! only).
    Double Precision    Scale     ! Scaling of coords and Origin relative to radians
                         ! (for type 1 coord systems) or metres (for type 2 coord
                         ! systems). Scale > 1 means the values are bigger and the
                         ! units smaller.

    ! Note easterly longitudes and northerly latitudes are positive.

    ! For type 1 coord systems with PoleScale = 1, Pole(2) + Pi/2, Pole(1), Angle -
    ! Pi/2 can be identified with the three Euler angles, where the second rotation
    ! takes the pole away from the north pole down the zero longitude line.

    ! Points on tangent plane are identified with those on the sphere by polar
    ! stereographic projection.

  End Type HCoord_
  ! $$ Note need to add national grid system

  Type :: VCoord_ ! Information defining vertical coord system.
```

```
    Integer(4) CoordType         ! 1 = height above ground (metres),
                                 ! 2 = height above sea (metres),
                                 ! 3 = pressure (hPa),
                                 ! 4 = flight level - i.e. pressure, converted
                                 !     to height above sea level using the ICAO
                                 !     standard atmosphere (hundreds of feet),
                                 ! 5 = a coordinate system specific to a
                                 !     particular flow module instance.
    Double Precision    Scale    ! Scaling of coord relative to above units (for
                                 ! type 1-4 coord systems). Scale > 1 means the
                                 ! values are bigger and the units smaller.
    Integer(4) FlowIndex         ! Index of flow module defining a type 5 coord
                                 ! system.
    Integer(4) FlowInstanceIndex ! Index of flow module instance defining a type
                                 ! 5 coord system.
  End Type VCoord_
  ! $$ check flight level definition.

  Type :: TCoord_                 ! Information defining temporal coord system.
    Integer(4)         CoordType ! 1 = relative to midnight on 31/12/1999,
                                 ! 2 = relative to start of simulation,
                                 ! 3 = relative to release time.
    Double Precision    Origin    ! Offset of the origin in units defined by Scale.
    Double Precision    Scale     ! Scaling of coord units relative to seconds.
                                  ! Values > 1 mean the coord values are bigger
                                  ! and the units smaller.
  End Type TCoord_

  Type :: Time_                        ! Information defining a time.
    Type (TCoord_)        TCoord       ! Temporal coord system used.
    Double Precision      T            ! Numerical value of time.
    Logical(4)            TInfinite    ! Indicates an infinite time (or a
                                       ! negative infinite time if T is negative).
  End Type Time_

  Type :: Domain_ ! A space-time region.
    Type (HCoord_) HCoord              ! Horizontal coord system used to define the
                                       ! domain.
!    Double Precision      VertexCoords(2,5) ! Vertices of defining the horizontal
                                       ! extent, going round the domain's boundary
                                       ! anti-clockwise with the first point
                                       ! stored twice. The horizontal extent of
                                       ! the domain is the convex hull of the
                                       ! vertices in the coord system HCoord.
    Double Precision      XMin
    Double Precision      XMax
    Double Precision      YMin
    Double Precision      YMax
    Logical(4)            HUnbounded     ! Indicates the domain is unbounded
                                         ! horizontally.
    Type (VCoord_) VCoord                ! Vertical coord system used to define
                                         ! the domain.
    Double Precision      DomainTop      ! Top of domain.
    Logical(4)            VUnbounded     ! Indicates the domain is unbounded vertically.
    Type (Time_)          StartTime      ! Start of temporal extent of domain.
    Type (Time_)          EndTime        ! End of temporal extent of domain.
  End Type Domain_

Contains

  Function InitHCoord(CoordType, Pole, Angle, PoleScale, Origin, Scale)
  ! Initialises a horizontal coord system.
```

```
    Use MathsModule
    Implicit None
    Integer(4),            Intent(In) :: CoordType
    Double Precision,      Intent(In) :: Pole(2)
    Double Precision,      Intent(In) :: Angle
    Double Precision,      Intent(In) :: PoleScale
    Double Precision,      Intent(In) :: Origin(2)
    Double Precision,      Intent(In) :: Scale
    Type (HCoord_)                       InitHCoord

    !DEC$ IF DEFINED(ExtraChecks)
      If (CoordType <= 0   .or. &
          CoordType >= 3   .or. &
          PoleScale  <= 0.0 .or. &
          Scale      <= 0.0) Then
        Write (6,*) 'Error in InitHCoord'
        Stop
      End If
    !DEC$ ENDIF
    InitHCoord = HCoord_(CoordType, Pole, Angle, PoleScale, Origin, Scale)

  End Function InitHCoord

  Function StandardLatLongCoordRadians()
  ! Returns the standard latitude-longitude coord system (with units in radians).
    Implicit None
    Type (HCoord_) StandardLatLongCoordRadians

    StandardLatLongCoordRadians = HCoord_(1, (/ 0.0, 0.0 /), 0.0, 1.0, &
                                          (/ 0.0, 0.0 /), 1.0)

  End Function StandardLatLongCoordRadians

  Function StandardLatLongCoordDegrees()
  ! Returns the standard latitude-longitude coord system (with units in degrees).
    Use MathsModule
    Implicit None
    Type (HCoord_) StandardLatLongCoordDegrees

    StandardLatLongCoordDegrees = HCoord_(1, (/ 0.0, 0.0 /), 0.0, 1.0, &
                                          (/ 0.0, 0.0 /), 180.0/Pi)
!    StandardLatLongCoordDegrees = HCoord_(1, (/ 0.0, 0.0 /), 0.0, 180.0/Pi, &
!                                          (/ 0.0, 0.0 /), 180.0/Pi)

  End Function StandardLatLongCoordDegrees



  Function ConvertH11S(HCoordIn, HCoordOut, PointIn)
  ! Converts between two type 1 coord systems, the second one being the standard
  ! latitude-longitude coord system (apart from a possible scale factor).
    Use MathsModule
    Implicit None
    Type (HCoord_), Intent(In) :: HCoordIn
    Type (HCoord_), Intent(In) :: HCoordOut
    Double Precision,       Intent(In) :: PointIn(2)
    Double Precision                      ConvertH11S(2)
    Double Precision, Parameter :: Small = 1.0E-15 ! Local: for cos(lat) less than
                                                   ! this, treat as if at pole
                                                   ! and set longitude = 0
                                          ! $$ test value
    Double Precision LatIn                ! Local: input latitude in radians
```

```
    Double Precision LongIn               ! Local: input longitude in radians
    Double Precision LatPole              ! Local: latitude of input pole, radians
    Double Precision LongPole             ! Local: longitude of input pole, radians
    Double Precision PointOut(2)          ! Local
    Double Precision Temp2          ! Local
    Double Precision XTemp, YTemp, ZTemp  ! Local

    !DEC$ IF DEFINED(ExtraChecks)
      If (HCoordIn%CoordType   /= 1 .or. &
          HCoordOut%CoordType  /= 1 .or. &
          HCoordOut%Pole(1)    /= 0 .or. &
          HCoordOut%Pole(2)    /= 0 .or. &
          HCoordOut%Angle      /= 0) Then
        Write (6,*) 'Error in ConvertH11S'
        Stop
      End If
    !DEC$ ENDIF

    LatIn         = PointIn(1)
    LatIn         = HCoordIn%Scale
    LatIn         = PointIn(1)/HCoordIn%Scale
    LongIn        = PointIn(2)/HCoordIn%Scale + HCoordIn%Angle/HCoordIn%PoleScale
    LatPole       = Pi/2.0d00 - HCoordIn%Pole(1)/HCoordIn%PoleScale
    LongPole      = HCoordIn%Pole(2)/HCoordIn%PoleScale

    XTemp         = Cos(LongIn)*Cos(LatIn)*Sin(LatPole) &
                  + Sin(LatIn) * Cos(LatPole)
    YTemp         = Sin(Pi-LongIn) * Cos(LatIn)
    ZTemp         = Sin(LatIn) * Sin(LatPole) &
                  - Cos(LatIn) * Cos(LatPole) * Cos(LongIn)
! Latitude calculation
    PointOut(1)   = Atan2(ZTemp,SQRT(XTemp**2+YTemp**2))
! Longitude calculation
    If ((Pi/2.0d00 - ABS(PointOut(1))) < Small) Then
      PointOut(2) = 0.0d00
    Else
      PointOut(2) = Atan2(YTemp,XTemp) + LongPole
      If (PointOut(2) > Pi) PointOut(2) = PointOut(2) - Pi
    End If
! Final conversion for any scaling involved
    PointOut(1)   = PointOut(1)*HCoordOut%Scale
    PointOut(2)   = PointOut(2)*HCoordOut%Scale
    ConvertH11S   = PointOut

  End Function ConvertH11S


  Function ConvertH1S1(HCoordIn, HCoordOut, PointIn)
  ! Converts between two type 1 coord systems, the first one being the standard
  ! latitude-longitude coord system (apart from a possible scale factor).
    Use MathsModule
    Implicit None
    Type (HCoord_),    Intent(In) :: HCoordIn
    Type (HCoord_),    Intent(In) :: HCoordOut
    Double Precision,  Intent(In) :: PointIn(2)
    Double Precision              :: ConvertH1S1(2)
    Double Precision,   Parameter :: Small = 1.0E-15
                                        ! Local: for cos(lat) less than this,
                                        ! treat as if at pole and set
                                        ! longitude = 0 $$ test value
    Double Precision LatIn              ! Local: input latitude in radians
    Double Precision LongIn             ! Local: input longitude in radians
```

```
    Double Precision LatPole               ! Local: latitude of output pole, radians
    Double Precision LongPole              ! Local: longitude of output pole, radians
    Double Precision LongInMPole           ! Local: LongIn - LongPole
    Double Precision PointOut(2)           ! Local
    Double Precision Temp2                 ! Local
    Double Precision XTemp, YTemp, ZTemp   ! Local

    !DEC$ IF DEFINED(ExtraChecks)
      If (HCoordOut%CoordType /= 1 .or. &
          HCoordIn%CoordType  /= 1 .or. &
          HCoordIn%Pole(1)    /= 0 .or. &
          HCoordIn%Pole(2)    /= 0 .or. &
          HCoordIn%Angle      /= 0) Then
        Write (6,*) 'Error in ConvertH1S1'
        Stop
      End If
    !DEC$ ENDIF

    LatIn         = PointIn(1)/HCoordIn%Scale
    LongIn        = PointIn(2)/HCoordIn%Scale
    LatPole       = Pi/2.0 - HCoordOut%Pole(1)/HCoordOut%PoleScale
    LongPole      = HCoordOut%Pole(2)/HCoordOut%PoleScale

    XTemp         = Cos(LongIn - LongPole) * Cos(LatIn) * Sin(LatPole) &
                  - Sin(LatIn) * Cos(LatPole)
    YTemp         = Sin(LongIn - LongPole) * Cos(LatIn)
    ZTemp         = Sin(LatIn)*Sin(LatPole) &
                  + Cos(LatIn) * Cos(LatPole) * Cos(LongIn - LongPole)
! Latitude calculation
    PointOut(1)   = Atan2(ZTemp,SQRT(XTemp**2+YTemp**2))
! Longitude calculation
    If (ABS(YTemp-XTemp) < Small) Then
      PointOut(2) = 0.0d00
    Else
      PointOut(2) = Atan2(YTemp,XTemp)
    End If
! Final conversion for any scaling involved
    PointOut(1)   = PointOut(1)*HCoordOut%Scale
    PointOut(2)   = (PointOut(2) - HCoordOut%Angle/HCoordOut%PoleScale)* &
                    HCoordOut%Scale
    ConvertH1S1   = PointOut

  End Function ConvertH1S1


  Function ConvertH12(HCoordIn, HCoordOut, PointIn)
  ! Converts from a type 1 to a type 2 coord system with the tangent point at the
  ! north pole and the same values of Angle and PoleScale
    Use GlobalParametersModule
    Implicit None
    Type (HCoord_),          Intent(In) :: HCoordIn
    Type (HCoord_),          Intent(In) :: HCoordOut
    Double Precision,        Intent(In) :: PointIn(2)
    Double Precision                       ConvertH12(2)
    Double Precision                    :: PointOut(2)  ! Local
    Double Precision                    :: U, V, UPrime ! Local

    !DEC$ IF DEFINED(ExtraChecks)
      If (HCoordIn%CoordType  /= 1                  .or. &
          HCoordOut%CoordType /= 2                  .or. &
          HCoordIn%Pole(1)    /= HCoordOut%Pole(1)  .or. &
          HCoordIn%Pole(2)    /= HCoordOut%Pole(2)  .or. &
```

```
        HCoordIn%Angle      /= HCoordOut%Angle   .or. &
        HCoordIn%PoleScale  /= HCoordOut%PoleScale) Then
      Write (6,*) 'Error in ConvertH12'
      Stop
    End If
  !DEC$ ENDIF
  U           = DCos(PointIn(1)/HCoordIn%Scale)
  V           = DSin(PointIn(1)/HCoordIn%Scale)
  UPrime      = 2.0*EarthRadius*U/(1.0 + V)
  PointOut(1) = UPrime*DSin(PointIn(2)/HCoordIn%Scale)
  PointOut(2) = - UPrime*DCos(PointIn(2)/HCoordIn%Scale)
  PointOut(1) = PointOut(1)*HCoordOut%Scale - HCoordOut%Origin(1)
  PointOut(2) = PointOut(2)*HCoordOut%Scale - HCoordOut%Origin(2)
  ConvertH12  = PointOut

End Function ConvertH12

Function ConvertH21(HCoordIn, HCoordOut, PointIn)
! Converts from a type 2 to a type 1 coord system with the tangent point at the
! north pole and the same value of Angle and PoleScale
  Use GlobalParametersModule
  Implicit None
  Type (HCoord_),          Intent(In) :: HCoordIn
  Type (HCoord_),          Intent(In) :: HCoordOut
  Double Precision,        Intent(In) :: PointIn(2)
  Double Precision                    :: ConvertH21(2)
  Double Precision                    :: PointOut(2) ! Local
  Double Precision                    :: U, V        ! Local

  !DEC$ IF DEFINED(ExtraChecks)
    If (HCoordOut%CoordType /= 1                 .or. &
        HCoordIn%CoordType  /= 2                 .or. &
        HCoordIn%Pole(1)    /= HCoordOut%Pole(1) .or. &
        HCoordIn%Pole(2)    /= HCoordOut%Pole(2) .or. &
        HCoordIn%Angle      /= HCoordOut%Angle   .or. &
        HCoordIn%PoleScale  /= HCoordOut%PoleScale) Then
      Write (6,*) 'Error in ConvertH21'
      Stop
    End If
  !DEC$ ENDIF
  V           = (PointIn(1) + HCoordIn%Origin(1))**2 + &
                (PointIn(2) + HCoordIn%Origin(2))**2
  V           = V/(2.0*HCoordIn%Scale*EarthRadius)**2
  V           = (1.0 - V)/(1.0 + V)
  U           = Sqrt(1.0 - V**2)
  PointOut(1) = HCoordOut%Scale*ATan(V/U)
  PointOut(2) = HCoordOut%Scale*ATan2(PointIn(1), - PointIn(2))
  ConvertH21  = PointOut

End Function ConvertH21

Function ConvertH22(HCoordIn, HCoordOut, PointIn)
! Converts between two type 2 coord systems with the same tangent plane
  Implicit None
  Type (HCoord_), Intent(In) :: HCoordIn
  Type (HCoord_), Intent(In) :: HCoordOut
  Double Precision,        Intent(In) :: PointIn(2)
  Double Precision                       ConvertH22(2)
  Double Precision PointOut(2)     ! Local
  Double Precision Point(2), Angle ! Local

  !DEC$ IF DEFINED(ExtraChecks)
```

```
    If (HCoordOut%CoordType /= 2                 .or. &
        HCoordIn%CoordType  /= 2                 .or. &
        HCoordIn%Pole(1)    /= HCoordOut%Pole(1) .or. &
        HCoordIn%Pole(2)    /= HCoordOut%Pole(2) .or. &
        HCoordIn%PoleScale  /= HCoordOut%PoleScale) Then
      Write (6,*) 'Error in ConvertH22'
      Stop
    End If
  !DEC$ ENDIF
  Point(1)    = (PointIn(1) + HCoordIn%Origin(1))/HCoordIn%Scale
  Point(2)    = (PointIn(2) + HCoordIn%Origin(2))/HCoordIn%Scale
  Angle       = HCoordOut%Angle/HCoordOut%PoleScale - &
                HCoordIn%Angle/HCoordIn%PoleScale
  PointOut(1) = Point(1)*DCos(Angle) + Point(2)*DSin(Angle)
  PointOut(2) = Point(2)*DCos(Angle) - Point(1)*DSin(Angle)
  PointOut(1) = Point(1)*HCoordOut%Scale - HCoordOut%Origin(1)
  PointOut(2) = Point(2)*HCoordOut%Scale - HCoordOut%Origin(2)
  ConvertH22  = PointOut

End Function ConvertH22

Function ConvertH(HCoordIn, HCoordOut, PointIn)
! Converts coords between coord systems.
  Implicit None
  Type (HCoord_),   Intent(In) :: HCoordIn
  Type (HCoord_),   Intent(In) :: HCoordOut
  Double Precision, Intent(In) :: PointIn(2)
  Double Precision             :: ConvertH(2)
  Type (HCoord_)               :: HCoordIn1   ! Local: type 1 coord system
                                              !        based on HCoordIn%
  Type (HCoord_)               :: HSLLRCoord  ! Local: standard latitude-longitude
                                              !        coord system in radians.
  Type (HCoord_)               :: HCoordOut1  ! Local: type 1 coord system based
                                              !        on HCoordOut%
  Double Precision             :: OldPoint(2) ! Local
  Double Precision             :: NewPoint(2) ! Local

  HCoordIn1           = HCoordIn
  HCoordIn1%CoordType  = 1
  HCoordOut1          = HCoordOut
  HCoordOut1%CoordType = 1
  HSLLRCoord          = StandardLatLongCoordRadians()

  OldPoint = PointIn
  ! Here, OldPoint is in HCoordIn
  If (HCoordIn%CoordType == 2) Then
    NewPoint =  ConvertH21(HCoordIn, HCoordIn1, OldPoint)
    OldPoint = NewPoint
  End If
  ! Here, OldPoint is in a type 1 coord system, based on HCoordIn
  NewPoint = ConvertH11S(HCoordIn1, HSLLRCoord, OldPoint)
  OldPoint = NewPoint
  ! Here, OldPoint is in standard latitude-longitude coords
  NewPoint = ConvertH1S1(HSLLRCoord, HCoordOut1, OldPoint)
  OldPoint = NewPoint
  ! Here, OldPoint is in a type 1 coord system, based to HCoordOut
  If (HCoordOut%CoordType == 2) Then
    NewPoint = ConvertH12(HCoordOut1, HCoordOut, OldPoint)
    OldPoint = NewPoint
  End If
  ! Here, OldPoint is in HCoordOut
  ConvertH = OldPoint
```

```
  End Function ConvertH

  Function InitVCoord(CoordType, Scale, FlowIndex, FlowInstanceIndex)
  ! Initialises a vertical coord system.
    Implicit None
    Integer(4),         Intent(In) :: CoordType
    Double Precision,   Intent(In) :: Scale
    Integer(4),         Intent(In) :: FlowIndex
    Integer(4),         Intent(In) :: FlowInstanceIndex
    Type (VCoord_)                 :: InitVCoord

    !DEC$ IF DEFINED(ExtraChecks)
      If (CoordType <= 0 .or. &
          CoordType >= 5 .or. &
          Scale      <= 0.0) Then
        Write (6,*) 'Error in InitVCoord'
        Stop
      End If
    !DEC$ ENDIF
    InitVCoord = VCoord_(CoordType, Scale, FlowIndex, FlowInstanceIndex)

  End Function InitVCoord

! $$ V conversion routines (type 3 to/from 4 only)

  Function InitTCoord(CoordType, Origin, Scale)
  ! Initialises a temporal coord system.
    Implicit None
    Integer(4),          Intent(In) :: CoordType
    Double Precision,    Intent(In) :: Origin
    Double Precision,    Intent(In) :: Scale
    Type (TCoord_)                  ::InitTCoord

    !DEC$ IF DEFINED(ExtraChecks)
      If (CoordType <= 0 .or. &
          CoordType >= 4 .or. &
          Scale      <= 0.0) Then
        Write (6,*) 'Error in InitTCoord'
        Stop
      End If
    !DEC$ ENDIF
    InitTCoord = TCoord_(CoordType, Origin, Scale)

  End Function InitTCoord

! $$ T conversion routines

  Function InitTime(TCoord, T, TInfinite)
  ! Initialises a time.
    Implicit None
    Type (TCoord_),     Intent(In) :: TCoord
    Double Precision,   Intent(In) :: T
    Logical(4),         Intent(In) :: TInfinite
    Type (Time_)                   :: InitTime

    InitTime = Time_(TCoord, T, TInfinite)

  End Function InitTime

  Function InfiniteTime()
  ! Returns a time equal to infinity.
```

```
    Implicit None
    Type (Time_) InfiniteTime
    Type (TCoord_) Dummy
    Integer(4) I

    I = 1
!    Dummy = InitTCoord(1, 0.0, 1.0)
!    Dummy = InitTCoord(I, 0.0, 1.0)
!   InfiniteTime = Time_(Dummy, 1.0, .true.)
    InfiniteTime = Time_(InitTCoord(I, 0.0D00, 1.0D00), 1.0D00, .true.)

  End Function InfiniteTime

  Function MinusInfiniteTime()
  ! Returns a time equal to minus infinity.
    Implicit None
    Type (Time_) MinusInfiniteTime
    Integer(4) I

    I = 1
    MinusInfiniteTime = Time_(InitTCoord(I, 0.0D00, 1.0D00), -1.0D00, .true.)

  End Function MinusInfiniteTime

  Function TGeT(Time1, Time2)
  ! Tests for Time1 >= Time2.
    Implicit None
    Type (Time_),   Intent(In) :: Time1
    Type (Time_),   Intent(In) :: Time2
    Logical(4)                 :: TGeT

    If (Time1%TInfinite .and. Time1%T >= 0.0) Then
       TGeT = .True.
    Else If (Time1%TInfinite) Then
      TGeT = Time2%TInfinite .and. Time2%T < 0.0
    Else
      If (Time2%TInfinite) Then
        TGeT = Time2%T < 0.0
      Else
        !DEC$ IF DEFINED(ExtraChecks)
          If (Time1%TCoord%CoordType /= Time2%TCoord%CoordType) Then
            Write (6,*) 'Error in TGeT'
            Stop
          End If
        !DEC$ ENDIF
        TGeT = (Time1%T + Time1%TCoord%Origin)/Time1%TCoord%Scale >= &
               (Time2%T + Time2%TCoord%Origin)/Time2%TCoord%Scale
      End If
    End If

  End Function TGeT

  Function InitDomain(HCoord, XMin, XMax, YMin, YMax, HUnbounded, &
                      VCoord, DomainTop, VUnbounded,              &
                      TCoord, StartTime, EndTime,                 &
                      StartUnbounded, EndUnbounded)
  ! Initialises a domain%
    Implicit None
    Type (HCoord_),      Intent(In) :: HCoord
    Double Precision,    Intent(In) :: XMin
    Double Precision,    Intent(In) :: XMax
    Double Precision,    Intent(In) :: YMin
```

```
    Double Precision,    Intent(In) :: YMax
    Logical(4),          Intent(In) :: HUnbounded
    Type (VCoord_),      Intent(In) :: VCoord
    Double Precision,    Intent(In) :: DomainTop
    Logical(4),          Intent(In) :: VUnbounded
    Type (TCoord_),      Intent(In) :: TCoord
    Double Precision,    Intent(In) :: StartTime
    Double Precision,    Intent(In) :: EndTime
    Logical(4),          Intent(In) :: StartUnbounded
    Logical(4),          Intent(In) :: EndUnbounded
    Type (Domain_)                  :: InitDomain

    InitDomain = Domain_(HCoord, XMin, XMax, YMin, YMax, HUnbounded,  &
                         VCoord, DomainTop, VUnbounded,               &
                         InitTime(TCoord, StartTime, StartUnbounded), &
                         InitTime(TCoord, EndTime,  EndUnbounded))

  End Function InitDomain

  Function StartTimeOfDomain(Domain)
  ! Returns the lower time limit of a domain%
    Implicit None
    Type (Domain_), Intent(In) :: Domain
    Type (Time_)               :: StartTimeOfDomain

    StartTimeOfDomain = Domain%StartTime

  End Function StartTimeOfDomain

  Function EndTimeOfDomain(Domain)
  ! Returns the lower time limit of a domain%
    Implicit None
    Type (Domain_), Intent(In) :: Domain
    Type (Time_)               :: EndTimeOfDomain

    EndTimeOfDomain = Domain%EndTime

  End Function EndTimeOfDomain

  Function InDomain(Domain, HCoord, Point, VCoord, Z)
  ! Checks whether a space-time location lies in a domain%
    Implicit None
    Type (Domain_),      Intent(In) :: Domain
    Type (HCoord_),      Intent(In) :: HCoord
    Double Precision,    Intent(In) :: Point(2)
    Type (VCoord_),      Intent(In) :: VCoord
    Double Precision,    Intent(In) :: Z
    Logical(4)                      :: InDomain

    ! check for coord system agreement%$$
    InDomain = .True.

    If (.not.Domain%VUnbounded) Then
      If (Z > Domain%DomainTop) Then
        InDomain = .false.
        Return
      End If
    End If

    If (.not.Domain%HUnbounded) Then
      If (Point(1) < Domain%XMin .or. Point(1) > Domain%XMax) Then
        InDomain = .false.
```

```
        Return
      End If  ! check for cyclic longitude etc. $$
      If (Point(2) < Domain%YMin .or. Point(2) > Domain%YMax) Then
        InDomain = .false.
        Return
      End If
    End If

  End Function InDomain

  Function DistanceToHEdge(Point, HCoord, Domain)
  ! Computes distance to edge of domain% $$
    Implicit None
    Double Precision,    Intent(In) :: Point(2)
    Type(HCoord_),       Intent(In) :: HCoord
    Type (Domain_),      Intent(In) :: Domain
    Double Precision                :: DistanceToHEdge
    Integer(4)                      :: i                       ! Local
    Double Precision                :: d, x, y, x1, y1, x2, y2 ! Local

    x = Point(1)
    y = Point(2)
    DistanceToHEdge = 1.0E20
    If (HCoord%CoordType == 1) Then
    ! $$
!    ElseIf (HCoord%CoordType == 2) Then
!      Do i = 1,4
!        x1 = Domain%VertexCoords(1,i)
!        y1 = Domain%VertexCoords(2,i)
!        x2 = Domain%VertexCoords(1,i + 1)
!        y2 = Domain%VertexCoords(2,i + 1)
!        d  = ((x - x1)*(y1 - y2) - (y - y1)*(x1 - x2))/Sqrt((y1 - y2)**2 + (x1 - x2)**2)
!        DistanceToHEdge = Min(DistanceToHEdge, d)
!      EndDo
    End If

  End Function DistanceToHEdge

End Module CoordModule
```

A.2 maths.f90

```
! Module: Maths Module
! Date:   4/7/00
! Author: Dave Thomson
! Files:  maths.f90

Module MathsModule

 Double Precision, Parameter :: Pi = 3.14159265358979323846264 3d00 ! Pi.

! Alternate way to define Pi
! Pi = 4.0d00*atan(1.0d00)

Contains

 Function Gauss()
 ! Generates Gaussian random number.
```

```
   Implicit None
   Double Precision Gauss   ! Gaussian Random number
   Integer(4) i             ! Local
   Double Precision    Temp ! Local

   Gauss = - 6.0
   Do i = 1, 12
     Call Random_Number(Temp)
     Gauss = Gauss + Temp
   End Do

 End Function Gauss

 Function Erf(X)
 ! This function calculates the error function using an
 ! approximation given by Abramowitz and Stegun, Handbook
 ! of mathematical functions, Dover Publications
 ! (1965).
   Implicit None
   Real, Intent(In)  :: X   ! Argument of error function.
   Double Precision                 Erf ! Error function.
   Double Precision, Parameter :: P  =  0.3275911   ! ! Constants
   Double Precision, Parameter :: A1 =  0.254829592 ! ! used in
   Double Precision, Parameter :: A2 = -0.284496736 ! ! numerical
   Double Precision, Parameter :: A3 =  1.421413741 ! ! approximation
   Double Precision, Parameter :: A4 = -1.453152027 ! ! to error
   Double Precision, Parameter :: A5 =  1.061405429 ! ! function.
   Double Precision T ! Local

   T = 1.0/(1.0 + P*Abs(X))
   Erf = 1.0 - Exp(-X*X)*(A1*T + A2*T**2 + A3*T**3 + A4*T**4 + A5*T**5)
   If (X < 0.0) Erf = -Erf

 End Function Erf

End Module MathsModule
```

A.3 globalparameters.f90

```
! Module: Global Parameters Module
! Date:   4/7/00
! Author: Dave Thomson
! Files:  globalparameters.f90

Module GlobalParametersModule

 ! Maximum number of ...
 Integer(4), Parameter :: MaxFlowModules    = 3  ! ... flow modules.
 Integer(4), Parameter :: MaxFlow1s         = 10 ! ... flow 1 module
                                                 ! instances.
 Integer(4), Parameter :: MaxFlowsPerModule = 10 ! $$
 Integer(4), Parameter :: MaxFlows          = MaxFlow1s
                                                 ! ... flow module
                                                 ! instances.
                                                 ! Must be the sum of
                                                 ! the maximum number
                                                 ! of instances for the
                                                 ! various flow modules.
 Integer(4), Parameter :: MaxSetsOfFlows    = 10 ! ... sets of flows.
 Integer(4), Parameter :: MaxMetModules     = 3  ! ... met modules.
 Integer(4), Parameter :: MaxMet1s          = 10 ! ... met 1 module
```

```
                                                 ! instances.
 Integer(4), Parameter :: MaxMetsPerModule  = 10 ! $$
 Integer(4), Parameter :: MaxMets           = MaxMet1s
                                                 ! ... met module
                                                 ! instances.
                                                 ! Must be the sum of
                                                 ! the maximum number
                                                 ! of instances for the
                                                 ! various met modules.
 Integer(4), Parameter :: MaxHCoordsPerFlow = 1
 Integer(4), Parameter :: MaxVCoordsPerFlow = 2
 Integer(4), Parameter :: MaxTCoordsPerFlow = 1
 Integer(4), Parameter :: MaxSources        = 10

 Double Precision, Parameter :: EarthRadius =  6371229.0

End Module GlobalParametersModule
```

References

Abramowitz, M. and Stegun, I. A.: 1972, *Handbook of mathematical functions*, 9 edn, Dover Publications, New York, USA.