International Astronomical Union

Standards Of Fundamental Astronomy

SOFA Vector-Matrix Library

Software version 18
Document revision 1.01
Version for Fortran programming language

http://www.iausofa.org

2021 April 18

MEMBERS OF THE IAU SOFA BOARD (2021)

John Bangert	United States Naval Observatory (retired)
Steven Bell	Her Majesty's Nautical Almanac Office
Nicole Capitaine	Paris Observatory
Maria Davis	United States Naval Observatory (IERS)
Mickaël Gastineau	Paris Observatory, IMCCE
Catherine Hohenkerk	Her Majesty's Nautical Almanac Office (chair, retired)
Li Jinling	Shanghai Astronomical Observatory
Zinovy Malkin	Pulkovo Observatory, St Petersburg
Jeffrey Percival	University of Wisconsin
Wendy Puatua	United States Naval Observatory
Scott Ransom	National Radio Astronomy Observatory
Nick Stamatakos	United States Naval Observatory
Patrick Wallace	RAL Space (retired)
Toni Wilmot	Her Majesty's Nautical Almanac Office (trainee)

Past Members

Wim Brouw	University of Groningen
Mark Calabretta	Australia Telescope National Facility
William Folkner	Jet Propulsion Laboratory
Anne-Marie Gontier	Paris Observatory
George Hobbs	Australia Telescope National Facility
George Kaplan	United States Naval Observatory
Brian Luzum	United States Naval Observatory
Dennis McCarthy	United States Naval Observatory
Skip Newhall	Jet Propulsion Laboratory
Jin Wen-Jing	Shanghai Observatory

Contents

1 Preliminaries

1.1 Introduction

SOFA stands for *Standards Of Fundamental Astronomy.* The SOFA software is a collection of Fortran 77 and ANSI C subprograms that implement official IAU algorithms for fundamental-astronomy computations. At the present time the SOFA software comprises 192 astronomy routines supported by 55 utility routines that deal with angles, vectors and matrices and called the SOFA vector-matrix library (VML). The core documentation for the SOFA collection consists of classified and alphabetic lists of subroutine calls plus detailed preamble comments in the source code of individual routines.

The present document concerns the VML angle/vector/matrix tools, that were either implemented in the course of writing the astronomical routines or that were thought likely to be useful in writing astronomical applications. Although in the wider context they are far from exhaustive (there is for example no treatment of quaternions) they are at least a good starting point. And as most are very short and simple, they could act as models for implementing similar facilities in other programming languages.

Using the VML routines requires knowledge of vector/matrix methods, simple spherical trigonometry, and methods of attitude representation. These topics are covered in many textbooks[1], and the present document does not pretend to be an *ab initio* tutorial. Its main objective is simply to set out the SOFA routines in context and allow their names and calls to be quickly discovered. Experienced users will seldom need to refer to anything more than the quick reference material at the end, namely Sections 3.4 and 3.5. More complete information about a given routine can be found in Section 3.3, which is arranged alphabetically.

1.2 General principles

The SOFA VML consists mostly of routines which operate on ordinary Cartesian vectors (x, y, z) and 3×3 rotation matrices, plus a few related to spherical angles. There is also support for vectors that represent velocity as well as position and vectors that represent rotation instead of position. Thus the array-based entities that SOFA uses are the following:

- "Position vectors" or "p-vectors" (which are just ordinary 3-vectors) are `DOUBLE PRECISION (3)`.
- "Position/velocity vectors" or "pv-vectors" are `DOUBLE PRECISION (3,2)`. In terms of memory address, the velocity components of a pv-vector follow the position components. Application code is permitted to exploit this and all other knowledge of the internal layouts: that x, y and z appear in that order and are in a right-handed Cartesian coordinate system *etc.* For example, the `iau_CP` routine (copy a p-vector) can be used to copy the velocity component of a pv-vector (indeed, this is how the `iau_CPV` routine is coded).

[1] For instance *Spacecraft Attitude Determination and Control*, James R. Wertz (ed.), Astrophysics and Space Science Library, Vol. 73, D. Reidel Publishing Company, 1986.

- "Rotation matrices" or "r-matrices" are `DOUBLE PRECISION (3,3)`. When used for rotation, they are *orthogonal*; each row or column is a unit vector, and the inverse is equal to the transpose. Most of the matrix routines do not assume that r-matrices are necessarily orthogonal and in fact work on any 3×3 matrix.

- "Rotation vectors" or "r-vectors" (or *Euler vectors*) are `DOUBLE PRECISION (3)`. Such vectors are a combination of the Euler axis and angle and are convertible to and from r-matrices. The direction is the axis of rotation and the magnitude is the angle of rotation, in radians. Because the amount of rotation can be scaled up and down simply by multiplying the vector by a scalar, r-vectors are useful for representing spins about an axis which is fixed.

The set of routines provided do not completely fill the range of operations that link all the various vector and matrix options, but are with some exceptions limited to routines that are required by SOFA's astronomical software.

2 Guide to the VML routines

This section outlines a number of positional-astronomy topics, for background and to provide a context in which the various routines can be introduced.

2.1 Spherical trigonometry

Celestial phenomena occur at such vast distances from the observer that for most practical purposes there is no need to work in 3D; only the direction of a source matters, not how far away it is. Things can therefore be viewed as if they were happening on the inside of sphere with the observer at the centre – the *celestial sphere*. Problems involving positions and orientations in the sky can then be solved by using the formulas of *spherical trigonometry*, which apply to *spherical triangles*, the sides of which are *great circles*.

Positions on the celestial sphere may be specified by using a spherical polar coordinate system, defined in terms of some fundamental plane and a direction in that plane chosen to represent zero longitude. Mathematicians usually work with the co-latitude, with zero at the principal pole, whereas most astronomical coordinate systems use latitude, reckoned plus and minus from the equator. Astronomical coordinate systems may be either right-handed (*e.g.* right ascension and declination $[\alpha, \delta]$, galactic longitude and latitude $[l^{II}, b^{II}]$) or left-handed (*e.g.* hour angle and declination $[h, \delta]$). In some cases different conventions have been used in the past, a fruitful source of mistakes. Azimuth and geographical longitude are examples; azimuth is now generally reckoned north through east (making a left-handed system); geographical longitude is now usually taken to increase eastwards (a right-handed system) but astronomers at one time employed a west-positive convention. In reports and program comments it is wise to spell out what convention is being used, if there is any possibility of confusion.

When applying spherical trigonometry formulas, attention must be paid to rounding errors (for example it is a bad idea to find a small angle through its cosine, as we shall see later) and to the possibility of problems close to poles. Formulas that involve tangents and cotangents need to be treated with particular care. Also, if a formulation relies on inspection to establish the quadrant of the result, it is a sure sign that a vector-related method will be preferable.

Although SOFA includes many functions that work in terms of specific spherical coordinates such as $[\alpha, \delta]$, only two routines that operate directly on generic spherical coordinates are provided: `iau_SEPS` computes the angular separation between two points (*i.e.* the distance along a great circle) and `iau_PAS` computes the bearing or *position angle* of one point seen from the other. As a simple demonstration, we will use these two routines (and a spherical-Earth approximation) to estimate the distance from London to Sydney and the initial compass heading:

```
      IMPLICIT NONE
      DOUBLE PRECISION R2D, RKM
      PARAMETER ( R2D = 57.2957795D0, RKM = 6375D0 )
      DOUBLE PRECISION AL, BL, AS, BS, S, B

*  Longitudes and latitudes (radians) for London and Sydney.
```

```
      AL = -0.2D0 / R2D
      BL = 51.5D0 / R2D
      AS = 151.2D0 / R2D
      BS = -33.9 / R2D

*  Great-circle distance and initial heading.
      CALL iau_SEPS ( AL,BL, AS,BS, S )
      CALL iau_PAS ( AL,BL, AS,BS , B )
      WRITE ( *, '(F7.1,'' km,'',F6.1,'' deg'')' ) S*RKM, B*R2D

      END
```

The result is range 17011 km, bearing 61° (towards Moscow).

The routines `iau_SEPP` (p62) and `iau_PAP` (p32) are equivalents of `iau_SEPS` (p63) and `iau_PAS` (p33) but starting from p-vectors instead of spherical coordinates.

In view of what will be said later about the superiority of vector techniques, it should be noted that the use of spherical trigonometry formulas in the SOFA collection is essentially nil.

2.1.1 Formatting angles

SOFA has routines for converting angles to and from sexagesimal form (hours, minutes, seconds or degrees, arcminutes, arcseconds). These apparently straightforward operations contain hidden traps which the SOFA functions avoid.

In that connection, an aspect that application developers need to address is how to go about decoding numbers from a character string, such as might be entered using a keyboard. SOFA at present offers no help with this, leaving the user to rely either on locally-developed libraries or Fortran's low-level formatted `READ` facilities. A particular difficulty arises with sexagesimal formats, where it is tempting simply to decode three numbers and apply the sign of the first to the final answer. This is the notorious "minus zero" bug, where the string `'-0'` is received as (plus) zero and the minus sign lost; consequently declinations *etc.* in the range 0° to −1° mysteriously migrate to the range 0° to +1°.[2] The only solution is to eschew the number decoding facilities of the programming language and resort instead to low-level character based techniques.

SOFA provides two routines for expressing an angle in radians in a preferred range:

- normalize radians to range 0 to 2π, `iau_ANPM` (p23)
- normalize radians to range −π to +π, `iau_ANP` (p22)

The first is suitable for hour angles and the second right ascension, for example. Six routines...

[2] For instance source ICRF J001611.0-001512 in Table 5 of Fey, A.L. *et al.*, *The Second Realization of the International Reference Frame by Very Long Baseline Interferometry*, Astronomical Journal, 150:58, 2015.

- decompose radians into degrees, arcminutes, arcseconds, `iau_A2AF`, p19
- decompose radians into hours, minutes, seconds, `iau_A2TF`, p20
- decompose days into hours, minutes, seconds, `iau_D2TF`, p28
- degrees, arcminutes, arcseconds to radians, `iau_AF2A`, p21
- hours, minutes, seconds to radians, `iau_TF2A`, p66
- hours, minutes, seconds to days, `iau_TF2D`, p67

...are provided to convert angles to and from sexagesimal form. They avoid the common "inconsistent rounding" bug, which produces angles like $24^{\mathrm{h}}\,59^{\mathrm{m}}\,59^{\mathrm{s}}.999$; they also avoid the "minus zero" bug mentioned earlier. Here is code which displays an hour angle in radians, awkwardly placed on the boundary between 0 and −1 hours, using two different resolutions:

```
      IMPLICIT NONE
      DOUBLE PRECISION HA
      CHARACTER SIGN
      INTEGER IHMSF(4)

      HA = -0.261799315D0
      CALL iau_A2TF ( 3, HA, SIGN, IHMSF )
      WRITE ( *, '( A, I2.2, 2I3.2, ''.'', I3.3 )' ) SIGN, IHMSF
      CALL iau_A2TF ( 2, HA, SIGN, IHMSF )
      WRITE ( *, '( A, I2.2, 2I3.2, ''.'', I2.2 )' ) SIGN, IHMSF

      END
```

The output is:

```
-00 59 59.999
-01 00 00.00
```

Note, however, that cases where rounding has moved the angle beyond the desired range will need to be detected explicitly, by testing whether the first field has reached 24, 360, 180 *etc.* and reacting appropriately.

2.2 P-vectors and R-matrices

Readers will already be aware that the SOFA philosophy is to avoid spherical trigonometry and instead favor vector methods. Many find this offputting. Given a positional-astronomy problem to solve, they expect there to be a simple formula involving a few sines and cosines that they can punch into a calculator to produce the required answers. The equivalent vector

expressions seem terse and unfriendly, and do not lend themselves to calculator evaluation. Vector-based positional-astronomy texts are peppered with intimidating symbols set in heavy type, and diagrams are few—an array of matrix elements, for example, lacks the intuitive appeal of a picture showing the physical meaning of the various angles. However, in practice it turns out that the vector methods are more powerful and better behaved than using spherical trigonometry, and the terseness of expression is a compelling advantage as problems become more complex.

The starting point is to recognize that a spherical polar coordinate system is only one way to describe the direction of an astronomical target. A convenient alternative is the sum of three vectors at right angles, forming a system of *Cartesian coordinates*. The x- and y-axes lie in the fundamental plane (*e.g.* the equator in the case of $[\alpha, \delta]$), with the x-axis pointing to zero longitude. The z-axis is normal to the fundamental plane and points towards the positive (north) pole. The y-axis can lie in either of the two possible directions, depending on whether the coordinate system is right-handed or left-handed. The three axes are sometimes called a *triad*. For most applications involving arbitrarily distant objects such as stars, the vector which defines the direction concerned is constrained to have unit length, no different to omitting the distance for spherical coordinates. The x-, y- and z- components can be regarded as the scalar (dot) product of this vector onto the three axes of the triad in turn. Because the vector is a unit vector, each of the three dot-products is simply the cosine of the angle between the unit vector and the axis concerned, and the three components are sometimes called *direction cosines* for this reason.

For some applications, including those involving objects within the Solar System, unit vectors are inappropriate, and it is necessary to use vectors scaled in length-units such as au, km *etc.* In these cases the origin of the coordinate system might not be the observer, but instead be the Sun, the Solar-System barycenter, the center of the Earth *etc.* But whatever the application, the final direction in which the observer sees the object can always be expressed as a unit vector.

But what has all this achieved? Instead of two numbers—a longitude and a latitude—we now have three numbers to look after, namely the x-, y- and z- components, whose quadratic sum we have somehow to constrain to be unity. And, in addition to this apparent redundancy, most people find it harder to visualize problems in terms of $[x, y, z]$ than in $[\theta, \phi]$, as mentioned above. Despite these objections, the vector approach turns out to have significant advantages over the spherical trigonometry approach:

- Vector formulas tend to be much more succinct; one vector operation hides strings of sines and cosines.
- The formulas are as a rule rigorous, even at the poles.
- Precision is maintained all over the celestial sphere. When one Cartesian component is nearly unity and therefore insensitive to direction, the others automatically become small and therefore relatively more precise: the precision is shared out.
- Formulations usually deliver the quadrant of the result without the need for inspection, an aspect delegated to the library routine `ATAN2`).

A number of important transformations in positional astronomy turn out to be nothing more than changes of coordinate system, something which is especially convenient if the vector approach is used. A direction with respect to one triad can be expressed relative to another triad

simply by multiplying the $[x, y, z]$ column vector by the appropriate 3×3 orthogonal matrix (a tensor of Rank 2, or *dyadic*). The three rows of this *rotation matrix* are the vectors in the old coordinate system of the three new axes, and the transformation amounts to obtaining the dot-product of the direction-vector with each of the three new axes. Conversely, the three columns of the matrix are the vectors in the new coordinate system of the three original axes. Precession, nutation, $[h, \delta]$ to $[Az, El]$, $[\alpha, \delta]$ to $[l^{II}, b^{II}]$ and so on are typical examples of the technique. An especially convenient property of the rotation matrices is that they can be inverted simply by taking the transpose.

The elements of these "p-vectors" and "r-matrices" are assorted combinations of the sines and cosines of the various angles involved (right ascension, declination and so on, depending on which transformation is being applied). If you write out the matrix multiplications in full you get expressions which are essentially the same as the equivalent spherical trigonometry formulas. Indeed, many of the standard formulas of spherical trigonometry are most easily derived by expressing the problem initially in terms of vectors.

2.2.1 SOFA routines for vectors and matrices

Transformations between spherical and vector form, with support for both unit vectors (direction cosines) and ones of specified length, are provided for by these routines:

- spherical to unit vector, `iau_S2C`, p58
- unit vector to spherical, `iau_C2S`, p24
- spherical to p-vector, `iau_S2P`, p59
- p-vector to spherical, `iau_P2S`, p31

An assortment of standard 3-vector operations (dot and cross products, add and subtract *etc.*) are carried out by these routines:

- zero p-vector, `iau_ZP`, p71
- p-vector plus p-vector, `iau_PPP`, p38
- p-vector minus p-vector, `iau_PMP`, p35
- p-vector plus scaled p-vector, `iau_PPSP`, p39
- inner (=scalar=dot) product of two p-vectors, `iau_PDP`, p34
- outer (=vector=cross) product of two p-vectors, `iau_PXP`, p49
- modulus of p-vector, `iau_PM`, p35
- normalize p-vector returning modulus, `iau_PN`, p37
- multiply p-vector by scalar, `iau_SXP`, p64

These routines make copies of 3-vectors and 3×3 matrices:

- copy p-vector, `iau_CP`, p25
- copy r-matrix, `iau_CR`, p27

There are routines for 3×3 matrix product and transpose:

- r-matrix multiply, `iau_RXR`, p55
- transpose r-matrix, `iau_TR`, p68

. . . and two for matrix-vector products:

- product of r-matrix and p-vector, `iau_RXP` , p53
- product of transpose of r-matrix and p-vector, `iau_TRXP`, p69

Initializing an r-matrix to null (all elements zero) or the identity matrix (diagonal elements unity, otherwise zero) can be accomplished by calling either

- initialize r-matrix to null, `iau_ZR`, p73
- initialize r-matrix to identity, `iau_IR`, p29

respectively. The latter is the first step when creating an r-matrix from *Euler angles* (successive rotations about specified Cartesian axes—see Section. 3.2 for all the three-angle cases). Each rotation can be applied by one of the routines. . .

- rotate r-matrix about x, `iau_RX`, p52
- rotate r-matrix about y, `iau_RY`, p56
- rotate r-matrix about z, `iau_RZ`, p57

In some cases (the construction of a bias-precession-nutation matrix is a good example) more than three calls will be needed. Note that the order is all-important; it is a common blunder to code an expression like $\mathbf{R}_x(\psi)\mathbf{R}_y(\theta)\mathbf{R}_z(\phi)$ by starting with $\mathbf{R}_x(\psi)$ and ending with $\mathbf{R}_z(\phi)$ when it is the reverse.

As a simple example of using a vector approach, the following code demonstrates how far an International Celestial Reference Frame source has moved between successive issues, namely ICRF2 and ICRF3:

```
      IMPLICIT NONE
      INTEGER J
      DOUBLE PRECISION RA, DA, RB, DB, THETA, A(3), B(3), AXB(3), S, C


*  RA,Dec of source ICRF J044238.6-001743 in the ICRF2 catalog.
      CALL iau_TF2A ( '+', 04, 42, 38.66073910D0, RA, J )
      CALL iau_AF2A ( '-', 00, 17, 43.4203921D0, DA, J )

*  RA,Dec of the same source in the ICRF3 (S/X) catalog.
      CALL iau_TF2A ( '+', 04, 42, 38.66072366D0, RB, J )
      CALL iau_AF2A ( '-', 00, 17, 43.4209582D0, DB, J )

*  Method 1:  spherical trigonometry (cosine rule).
      THETA = ACOS ( SIN(DA)*SIN(DB) + COS(DA)*COS(DB)*COS(RB-RA) )
      WRITE ( *, '(''The two positions are'',' //
     :              'F10.6, '' arcsec apart.'')' )
     :              THETA*206264.80624709635515647335733D0

*  Method 2:  vectors (sine and cosine from cross and dot products).
      CALL iau_S2C ( RA, DA, A )
      CALL iau_S2C ( RB, DB, B )
      CALL iau_PXP ( A, B, AXB )
      CALL iau_PM ( AXB, S )
      CALL iau_PDP ( A, B, C )
      THETA = ATAN2 ( S, C )
      WRITE ( *, '(''The two positions are'',' //
     :              'F10.6, '' arcsec apart.'')' )
     :              THETA*206264.80624709635515647335733D0

      END
```

The output is:

```
The two positions are  0.000000 arcsec apart.
The two positions are  0.000612 arcsec apart.
```

The failure of the first method to deliver a useful answer is simply because $\cos\theta$ of a small angle is close to unity. The vector-based code ensures accurate performance at all ranges of angle by computing both sine and cosine. This is the method used by the routines `iau_SEPS` (p63) and `iau_SEPP` (p62), and of course the six statements of Method 2 could be replaced by a single call to `iau_SEPS` without affecting the result.

2.3 R-vectors

Rotation matrices are just one way of describing attitude, and have both advantages and disadvantages. The fact that they comprise nine numbers means there is clearly some redundancy, and this is manifested as the requirement for each row and column to be a unit vector, a condition that will be compromised as rounding errors accumulate (and messy to renormalize). On the other hand, once the nine numbers are available they can be used with complete efficiency to reorient multiple vectors, something often needed in astronomical applications, for example to apply precession to a list of star positions.

But other methods exist, each with their own set of pros and cons. Any rotation can be expressed as *Euler axis and angle*, the former being the pole of rotation as a unit vector and the latter the amount of rotation, a scalar; this representation is thus a total of four numbers. These elements can be combined, two examples being *Gibbs vectors* and *Euler symmetric parameters* or *quaternions*, neither of which SOFA uses. Gibbs vectors consist of only three numbers, namely the Euler axis vector but scaled by the tangent of half the angle. A unit quaternion is four numbers, one of which is the cosine of half the angle and the other three the Euler axis scaled by the sine of half the angle.

Despite the fact that SOFA does not use them, quaternions are important and it is worth listing some of their advantages:

- Quaternions are more compact (four numbers) than the r-matrix representation (nine numbers), and as rounding errors build up renormalization is straightforward and efficient.
- The quaternion elements vary continuously over the unit sphere as the orientation changes, avoiding discontinuous jumps and singularities.
- Translating a unit quaternion into the equivalent rotation matrix involves no trigonometric functions.
- It is simple to combine two individual rotations represented as quaternions using a quaternion product, and this requires about half the arithmetic operations that combining two rotation matrices does. (On the other hand rotating a vector takes about twice the arithmetic operations than if a rotation matrix is used.)

The quaternion approach comes into its own for applications where computational efficiency is paramount, many different rotations are in play at once, and smooth interpolation is required, for example in computer games. It has less to offer to SOFA, which instead supplements its use of r-matrices with a "rotation vector" scheme, which simply scales the Euler axis unit vector by the angle in radians. The two techniques (quaternions and r-vectors) have much in common, and while the r-vector approach sacrifices some computational efficiency compared with quaternions it has its own set of advantages:

- Intuitive appeal—very easy to understand.
- An r-vector is completely non-redundant, comprising just three numbers.
- The numbers are independent, and the question of normalization does not arise.

- Smooth interpolation at constant angular speed is trivial.
- Multiple rotations (*i.e.* where the angle is more than 2π) can be expressed.

To demonstrate the first point, consider the so-called "frame bias" between the International Celestial Reference System and the J2000.0 mean equator and equinox triad. If we would like to know (i) where on the celestial sphere the frame bias is zero and (ii) the maximum effect frame bias can have on a star position, this is easy using r-vectors:[3]

```
      IMPLICIT NONE

      DOUBLE PRECISION DAS2R
      PARAMETER ( DAS2R = 4.848136811095359935899141D-6 )

      CHARACTER SIGN
      INTEGER IHMSF(4), IDMSF(4), I
      DOUBLE PRECISION RB(3,3), VB(3), ANGLE, RA, DEC

*  Generate frame bias matrix.
      CALL iau_IR ( RB )
      CALL iau_RZ ( -0.0146D0*DAS2R, RB )
      CALL iau_RY ( -0.041775D0*DAS2R * SIN(84381.448D0*DAS2R), RB )
      CALL iau_RX ( 0.0068192D0*DAS2R, RB )

*  Convert into r-vector form.
      CALL iau_RM2V ( RB, VB )

*  Report.
      CALL iau_P2S ( VB, RA, DEC, ANGLE )
      CALL iau_A2TF ( 1, RA, SIGN, IHMSF )
      CALL iau_A2AF ( 0, DEC, SIGN, IDMSF )
      WRITE ( *, '(1X,''Frame bias is'',F5.1,'' mas around'','' //
     :           '3I3.2,''.'',I1,1X,A,I2.2,2I3.2,'' GCRS.'')' )
     :                     ANGLE*1D3/DAS2R,IHMSF, SIGN, (IDMSF(I),I=1,3)

      END
```

The resulting report is:

```
Frame bias is 23.1 mas around 19 29 14.8 -39 06 19 GCRS.
```

[3] For clarity, the code uses literal angles; a real application would either get them by calling `iau_BI00` or would generate the matrix by calling `iau_PFW06` followed by `iau_FW2M`.

SOFA provides just two routines for dealing with r-vectors, namely the conversions between r-matrix and r-vector:

- r-matrix to r-vector, `iau_RM2V`, p50
- r-vector to r-matrix, `iau_RV2M`, p51

2.4 PV-vectors

SOFA calls a 3-vector used to represent a direction (whether of unit length or not) a "p-vector", mainly to distinguish it from an r-vector, and the related routines (see Section 2.2.1) work on all sorts of 3-vector, including for example changes of attitude by computing r-matrix × p-vector. However, special additional support is provided for the common case where for a body in space both position and velocity are available. This is SOFA's "pv-vector", which consists of a pair of 3-vectors containing $[\,x, y, z\,]$ and $[\,\dot{x}, \dot{y}, \dot{z}\,]$ respectively.

The following routines are provided:

- zero a pv-vector, `iau_ZPV`, p72
- copy a pv-vector, `iau_CPV`, p26
- create a pv-vector by appending zero velocity to a p-vector, `iau_P2PV`, p30
- dispense with the velocity to leave a p-vector, `iau_PV2P`, p40
- create a pv-vector from spherical coordinates, `iau_S2PV`, p60
- transform a pv-vector into spherical coordinates, `iau_PV2S`, p41
- add two pv-vectors together, `iau_PVPPV`, p45
- subtract one pv-vector from another, `iau_PVMPV`, p44
- form the scalar product of two pv-vectors, `iau_PVDPV`, p42
- form the vector product of two pv-vectors, `iau_PVXPV`, p48
- find the modulus of a pv-vector *i.e.* extract distance and speed, `iau_PVM`, p43
- multiply position and velocity by a scalar, `iau_SXPV`, p65
- multiply position and velocity separately by different scalars, `iau_S2XPV`, p61
- update the position part of a pv-vector, `iau_PVU`, p46
- update the position part of a pv-vector returning only the position, `iau_PVUP`, p47
- product of r-matrix and pv-vector, `iau_RXPV`, p54
- product of transpose of r-matrix and pv-vector, `iau_TRXPV`, p70

3 Reference material

3.1 SOFA vector-matrix conventions

When setting out vector and matrix expressions in mathematical notation there are choices to be made about the relation of rows and columns and associated indices. In addition, computer programming languages add further complications in that the order in which items are stored in memory has to be decided.

Although the present document is tailored towards SOFA's Fortran implementation, it will be useful to compare and contrast the two supported languages, the other being C. This will not only help developers who need to use both languages, but may also cast light on any choices that may seem surprising in the context of one language or the other.

Setting out the conventions clearly will also provide an opportunity to present some of the basic formulas. However, there will be no attempt to provide a comprehensive treatment of the underlying algebra (what operations commute, how sums are formed, *etc.*), beyond stressing at every opportunity that the order in which successive rotations are applied is crucial.

3.1.1 p-vectors

The convention for p-vectors is that they are considered to be column vectors:

$$\mathbf{a} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

It goes without saying that the three elements x, y and z occupy successive memory locations in both C and Fortran.

The spatial length of the vector is called the *modulus*, given by the routine `iau_PM`:

$$|\mathbf{a}| = (x^2 + y^2 + z^2)^{1/2}$$

For a unit vector the modulus is 1, and the three components are *direction cosines*.

The formula for *scalar product* of two vectors **a** and **b** is:

$$\mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y + a_z b_z$$

The result is a scalar equal to $|\mathbf{a}||\mathbf{b}| \cos\theta$, where θ is the angle between the two vectors, and hence for two unit vectors it is simply $\cos\theta$. Scalar product can be calculated by calling the `iau_PDP` routine, p34.

The formula for *vector product* is:

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix} = \begin{pmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{pmatrix},$$

where **i**, **j** and **k** are the unit vectors forming the xyz triad. The result is a vector of magnitude $|\mathbf{a}||\mathbf{b}|\sin\theta$, where θ is the angle between the two vectors, and with the direction given by the "right-hand rule", where $\mathbf{a}\times\mathbf{b}$ is the thumb, **a** is the forefinger and **b** is the middle finger. Thus when **a** and **b** are both unit vectors, $|\mathbf{a}\times\mathbf{b}|$ is simply $\sin\theta$. Vector product can be calculated by calling the `iau_PXP` routine, p49.

3.1.2 pv-vectors

For pv-vectors, the convention is that the two 3-vector components occupy successive triples of memory locations, first position and then velocity. This is convenient because any of the p-vector routines can be used to process either the position part or the velocity part without the routine having to know that it is part of a pv-vector. The consequence is that whereas in C the dimensions are `[2][3]`, in Fortran they are `(3,2)`.

3.1.3 r-matrices

Writing the elements of a matrix **R** with indices i, j where i is row and j is column as follows:

$$\mathbf{R} = \begin{pmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{pmatrix},$$

in Fortran these correspond to array elements:

$$\begin{pmatrix} (1,1) & (1,2) & (1,3) \\ (2,1) & (2,2) & (2,3) \\ (3,1) & (3,2) & (3,3) \end{pmatrix},$$

and in C:

$$\begin{pmatrix} [0][0] & [0][1] & [0][2] \\ [1][0] & [1][1] & [1][2] \\ [2][0] & [2][1] & [2][2] \end{pmatrix}.$$

However, the memory storage order is different in the two languages. In Fortran the matrix elements are stored in this order:

```
(1,1), (2,1), (3,1), (1,2), (2,2), (3,2), (1,3), (2,3), (3,3).
```

but in C the order is:

```
[0][0], [0][1], [0][2], [1][0], [1][1], [1][2], [2][0], [2][1], [2][2],
```

In other words, in memory, successive triples are columns in Fortran and rows in C.

To refer a p-vector **a** to a different frame using rotation matrix **R** we evaluate the product $\mathbf{b} = \mathbf{R}\,\mathbf{a}$ thus:

$$\begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix} = \begin{pmatrix} R_{11}\,a_x + R_{12}\,a_y + R_{13}\,a_z \\ R_{21}\,a_x + R_{22}\,a_y + R_{23}\,a_z \\ R_{31}\,a_x + R_{32}\,a_y + R_{33}\,a_z \end{pmatrix},$$

which is what the routine `iau_RXP` (p53) does. The inverse transformation is $\mathbf{a} = \mathbf{R}^{-1}\,\mathbf{b}$:

$$\begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} = \begin{pmatrix} R_{11}\,b_x + R_{21}\,b_y + R_{31}\,b_z \\ R_{12}\,b_x + R_{22}\,b_y + R_{32}\,b_z \\ R_{13}\,b_x + R_{23}\,b_y + R_{33}\,b_z \end{pmatrix},$$

which is what `iau_TRXP` (p69) does.

The matrix product $\mathbf{C} = \mathbf{BA}$ takes matrix **A** and rotates it using matrix **B** to give matrix **C**; as always, note the order. In terms of matrix elements:

$$\mathbf{C} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31} & A_{11}B_{12} + A_{12}B_{22} + A_{13}B_{32} & A_{11}B_{13} + A_{12}B_{23} + A_{13}B_{33} \\ A_{21}B_{11} + A_{22}B_{21} + A_{23}B_{31} & A_{21}B_{12} + A_{22}B_{22} + A_{23}B_{32} & A_{21}B_{13} + A_{22}B_{23} + A_{23}B_{33} \\ A_{31}B_{11} + A_{32}B_{21} + A_{33}B_{31} & A_{31}B_{12} + A_{32}B_{22} + A_{33}B_{32} & A_{31}B_{13} + A_{32}B_{23} + A_{33}B_{33} \end{pmatrix},$$

This is what the routine `iau_RXR` (p55) does.

3.2 The twelve r-matrices

Any 3D reorientation can be broken into three successive "elemental" rotations about one or other of the coordinate axes. Discounting degenerate cases where successive rotations are about the same axis, there are twelve possible instances, six using all three axes and another six where the first and third rotations are about the same axis.[4]

In the matrices listed on the next two pages, successive rotations are ϕ then θ then ψ about coordinate axes i, j and k, and are formed by evaluating $\mathbf{R}_k(\psi)\mathbf{R}_j(\theta)\mathbf{R}_i(\phi)$ (note the order). The three axes 123 are synonymous with xyz. Thus the matrix labeled 1-3-2 corresponds to rotations ϕ about the x-axis, followed by θ about the z-axis followed by ψ about the y-axis, giving the expression $\mathbf{R}_y(\psi)\mathbf{R}_z(\theta)\mathbf{R}_x(\phi)$. The most common choice in fundamental astronomy applications is 3-1-3.

These explicit representations of the matrix elements are useful when solving an r-matrix for one or more of the angles used in its formation.

The sign convention for the angles that SOFA uses is that they have positive values when they represent a rotation that appears clockwise when looking in the positive direction of the axis. Moreover SOFA uses rotations only to reorient the coordinate system, as opposed to rotating the vector itself within a fixed coordinate system.

Fortran code to form the 3-1-3 matrix might look like this:

```
      DOUBLE PRECISION RM(3,3), PHI, THETA, PSI

      CALL sla_IR ( RM )
      CALL sla_RZ ( PHI, RM )
      CALL sla_RX ( THETA, RM )
      CALL sla_RZ ( PSI, RM )
```

[4] SOFA calls the angles for all twelve axis sequences simply "Euler angles", but various names are in use to distinguish the two sets of six axis sequences, such as "Tait-Bryan angles" for the three-axis case and "proper" or "classic" Euler angles when the same axis is used twice.

$$\underset{\text{1-2-3}}{\mathbf{R}_z(\psi)\mathbf{R}_y(\theta)\mathbf{R}_x(\phi)} = \begin{bmatrix} \cos\psi\cos\theta & \cos\psi\sin\theta\sin\phi+\sin\psi\cos\phi & -\cos\psi\sin\theta\cos\phi+\sin\psi\sin\phi \\ -\sin\psi\cos\theta & -\sin\psi\sin\theta\sin\phi+\cos\psi\cos\phi & \sin\psi\sin\theta\cos\phi+\cos\psi\sin\phi \\ \sin\theta & -\cos\theta\sin\phi & \cos\theta\cos\phi \end{bmatrix}$$

$$\underset{\text{1-3-2}}{\mathbf{R}_y(\psi)\mathbf{R}_z(\theta)\mathbf{R}_x(\phi)} = \begin{bmatrix} \cos\psi\cos\theta & \cos\psi\sin\theta\cos\phi+\sin\psi\sin\phi & \cos\psi\sin\theta\sin\phi-\sin\psi\cos\phi \\ -\sin\theta & \cos\theta\cos\phi & \cos\theta\sin\phi \\ \sin\psi\cos\theta & \sin\psi\sin\theta\cos\phi-\cos\psi\sin\phi & \sin\psi\sin\theta\sin\phi+\cos\psi\cos\phi \end{bmatrix}$$

$$\underset{\text{2-3-1}}{\mathbf{R}_x(\psi)\mathbf{R}_z(\theta)\mathbf{R}_y(\phi)} = \begin{bmatrix} \cos\theta\cos\phi & \sin\theta & -\cos\theta\sin\phi \\ -\cos\psi\sin\theta\cos\phi+\sin\psi\sin\phi & \cos\psi\cos\theta & \cos\psi\sin\theta\sin\phi+\sin\psi\cos\phi \\ \sin\psi\sin\theta\cos\phi+\cos\psi\sin\phi & -\sin\psi\cos\theta & -\sin\psi\sin\theta\sin\phi+\cos\psi\cos\phi \end{bmatrix}$$

$$\underset{\text{2-1-3}}{\mathbf{R}_z(\psi)\mathbf{R}_x(\theta)\mathbf{R}_y(\phi)} = \begin{bmatrix} \cos\psi\cos\phi+\sin\psi\sin\theta\sin\phi & \sin\psi\cos\theta & -\cos\psi\sin\phi+\sin\psi\sin\theta\cos\phi \\ -\sin\psi\cos\phi+\cos\psi\sin\theta\sin\phi & \cos\psi\cos\theta & \sin\psi\sin\phi+\cos\psi\sin\theta\cos\phi \\ \cos\theta\sin\phi & -\sin\theta & \cos\theta\cos\phi \end{bmatrix}$$

$$\underset{\text{3-1-2}}{\mathbf{R}_y(\psi)\mathbf{R}_x(\theta)\mathbf{R}_z(\phi)} = \begin{bmatrix} \cos\psi\cos\phi-\sin\psi\sin\theta\sin\phi & \cos\psi\sin\phi+\sin\psi\sin\theta\cos\phi & -\sin\psi\cos\theta \\ -\cos\theta\sin\phi & \cos\theta\cos\phi & \sin\theta \\ \sin\psi\cos\phi+\cos\psi\sin\theta\sin\phi & \sin\psi\sin\phi-\cos\psi\sin\theta\cos\phi & \cos\psi\cos\theta \end{bmatrix}$$

$$\underset{\text{3-2-1}}{\mathbf{R}_x(\psi)\mathbf{R}_y(\theta)\mathbf{R}_z(\phi)} = \begin{bmatrix} \cos\theta\cos\phi & \cos\theta\sin\phi & -\sin\theta \\ -\cos\psi\sin\phi+\sin\psi\sin\theta\cos\phi & \cos\psi\cos\phi+\sin\psi\sin\theta\sin\phi & \sin\psi\cos\theta \\ \sin\psi\sin\phi+\cos\psi\sin\theta\cos\phi & -\sin\psi\cos\phi+\cos\psi\sin\theta\sin\phi & \cos\psi\cos\theta \end{bmatrix}$$

$$\underset{\text{1-2-1}}{\mathbf{R}_x(\psi)\mathbf{R}_y(\theta)\mathbf{R}_x(\phi)} = \begin{bmatrix} \cos\theta & \sin\theta\sin\phi & -\sin\theta\cos\phi \\ \sin\psi\sin\theta & \cos\psi\cos\phi - \sin\psi\cos\theta\sin\phi & \cos\psi\sin\phi + \sin\psi\cos\theta\cos\phi \\ \cos\psi\sin\theta & -\sin\psi\cos\phi - \cos\psi\cos\theta\sin\phi & -\sin\psi\sin\phi + \cos\psi\cos\theta\cos\phi \end{bmatrix}$$

$$\underset{\text{1-3-1}}{\mathbf{R}_x(\psi)\mathbf{R}_z(\theta)\mathbf{R}_x(\phi)} = \begin{bmatrix} \cos\theta & \sin\theta\cos\phi & \sin\theta\sin\phi \\ -\cos\psi\sin\theta & \cos\psi\cos\theta\cos\phi - \sin\psi\sin\phi & \cos\psi\cos\theta\sin\phi + \sin\psi\cos\phi \\ \sin\psi\sin\theta & -\sin\psi\cos\theta\cos\phi - \cos\psi\sin\phi & -\sin\psi\cos\theta\sin\phi + \cos\psi\cos\phi \end{bmatrix}$$

$$\underset{\text{2-1-2}}{\mathbf{R}_y(\psi)\mathbf{R}_x(\theta)\mathbf{R}_y(\phi)} = \begin{bmatrix} \cos\psi\cos\phi - \sin\psi\cos\theta\sin\phi & \sin\psi\sin\theta & -\cos\psi\sin\phi - \sin\psi\cos\theta\cos\phi \\ \sin\theta\sin\phi & \cos\theta & \sin\theta\cos\phi \\ \sin\psi\cos\phi + \cos\psi\cos\theta\sin\phi & -\cos\psi\sin\theta & -\sin\psi\sin\phi + \cos\psi\cos\theta\cos\phi \end{bmatrix}$$

$$\underset{\text{2-3-2}}{\mathbf{R}_y(\psi)\mathbf{R}_z(\theta)\mathbf{R}_y(\phi)} = \begin{bmatrix} \cos\psi\cos\theta\cos\phi - \sin\psi\sin\phi & \cos\psi\sin\theta & -\cos\psi\cos\theta\sin\phi - \sin\psi\cos\phi \\ -\sin\theta\cos\phi & \cos\theta & \sin\theta\sin\phi \\ \sin\psi\cos\theta\cos\phi + \cos\psi\sin\phi & \sin\psi\sin\theta & -\sin\psi\cos\theta\sin\phi + \cos\psi\cos\phi \end{bmatrix}$$

$$\underset{\text{3-1-3}}{\mathbf{R}_z(\psi)\mathbf{R}_x(\theta)\mathbf{R}_z(\phi)} = \begin{bmatrix} \cos\psi\cos\phi - \sin\psi\cos\theta\sin\phi & \cos\psi\sin\phi + \sin\psi\cos\theta\cos\phi & \sin\psi\sin\theta \\ -\sin\psi\cos\phi - \cos\psi\cos\theta\sin\phi & -\sin\psi\sin\phi + \cos\psi\cos\theta\cos\phi & \cos\psi\sin\theta \\ \sin\theta\sin\phi & -\sin\theta\cos\phi & \cos\theta \end{bmatrix}$$

$$\underset{\text{3-2-3}}{\mathbf{R}_z(\psi)\mathbf{R}_y(\theta)\mathbf{R}_z(\phi)} = \begin{bmatrix} \cos\psi\cos\theta\cos\phi - \sin\psi\sin\phi & \cos\psi\cos\theta\sin\phi + \sin\psi\cos\phi & -\cos\psi\sin\theta \\ -\sin\psi\cos\theta\cos\phi - \cos\psi\sin\phi & -\sin\psi\cos\theta\sin\phi + \cos\psi\cos\phi & \sin\psi\sin\theta \\ \sin\theta\cos\phi & \sin\theta\sin\phi & \cos\theta \end{bmatrix}$$

3.3 Routine specifications

---

iau_A2AF *radians to deg, arcmin, arcsec* **iau_A2AF**

CALL :

```
CALL iau_A2AF ( NDP, ANGLE, SIGN, IDMSF )
```

ACTION :

Decompose radians into degrees, arcminutes, arcseconds, fraction.

GIVEN :

NDP	`i`	resolution (Note 1)
ANGLE	`d`	angle in radians

RETURNED :

SIGN	`c`	'+' or '-'
IDMSF	`i(4)`	degrees, arcminutes, arcseconds, fraction

NOTES :

1. `NDP` is interpreted as follows:

```
NDP         resolution
 :      ...0000 00 00
-7         1000 00 00
-6          100 00 00
-5           10 00 00
-4            1 00 00
-3            0 10 00
-2            0 01 00
-1            0 00 10
 0            0 00 01
 1            0 00 00.1
 2            0 00 00.01
 3            0 00 00.001
 :            0 00 00.000...
```

2. The largest positive useful value for `NDP` is determined by the size of `ANGLE`, the format of `DOUBLE PRECISION` numbers on the target platform, and the risk of overflowing `IDMSF(4)`. On a typical platform, for `ANGLE` up to 2π, the available floating-point precision might correspond to `NDP = 12`. However, the practical limit is typically `NDP = 9`, set by the capacity of a 32-bit `IDMSF(4)`.

3. The absolute value of `ANGLE` may exceed 2π. In cases where it does not, it is up to the caller to test for and handle the case where `ANGLE` is very nearly 2π and rounds up to 360°, by testing for `IDMSF(1) .EQ. 360` and setting `IDMSF(1-4)` to zero.

iau_A2TF *radians to hours, minutes, seconds* iau_A2TF

CALL :

```
CALL iau_A2TF ( NDP, ANGLE, SIGN, IHMSF )
```

ACTION :

Decompose radians into hours, minutes, seconds, fraction.

GIVEN :

NDP	`i`	resolution (Note 1)
ANGLE	`d`	angle in radians

RETURNED :

SIGN	`c`	'+' or '-'
IHMSF	`i(4)`	hours, minutes, seconds, fraction

NOTES :

1. `NDP` is interpreted as follows:

```
NDP         resolution
 :      ...0000 00 00
-7         1000 00 00
-6          100 00 00
-5           10 00 00
-4            1 00 00
-3            0 10 00
-2            0 01 00
-1            0 00 10
 0            0 00 01
 1            0 00 00.1
 2            0 00 00.01
 3            0 00 00.001
 :            0 00 00.000...
```

2. The largest positive useful value for `NDP` is determined by the size of `ANGLE`, the format of `DOUBLE PRECISION` numbers on the target platform, and the risk of overflowing `IHMSF(4)`. On a typical platform, for `ANGLE` up to 2π, the available floating-point precision might correspond to `NDP = 12`. However, the practical limit is typically `NDP = 9`, set by the capacity of a 32-bit `IHMSF(4)`.

3. The absolute value of `ANGLE` may exceed 2π. In cases where it does not, it is up to the caller to test for and handle the case where `ANGLE` is very nearly 2π and rounds up to 360°, by testing for `IHMSF(1) = 360` and setting `IHMSF(1-4)` to zero.

iau_AF2A *deg, arcmin, arcsec to radians* iau_AF2A

CALL :

```
CALL iau_AF2A ( S, IDEG, IAMIN, ASEC, RAD, J )
```

ACTION :

Convert degrees, arcminutes, arcseconds to radians.

GIVEN :

S	`c`	sign: `'-'` = negative, otherwise positive
IDEG	`i`	degrees
IAMIN	`i`	arcminutes
ASEC	`d`	arcseconds

RETURNED :

RAD	`d`	angle in radians
J	`i`	status: 0 = OK 1 = `IDEG` outside range 0-359 2 = `IAMIN` outside range 0-59 3 = `ASEC` outside range 0-59.999...

NOTES :

1. If the `S` argument is a string, only the leftmost character is used and no warning status is provided.
2. The result is computed even if any of the range checks fail.
3. Negative `IDEG`, `IAMIN` and/or `ASEC` produce a warning status, but the absolute value is used in the conversion.
4. If there are multiple errors, the status value reflects only the first, the smallest taking precedence.

iau_ANP *normalize radians to range 0 to 2π* iau_ANP

CALL :

```
D = iau_ANP ( A )
```

ACTION :

Normalize angle into the range $0 \leq \mathrm{A} < 2\pi$.

GIVEN :

A	`d`	angle (radians)

RETURNED (function value) :

	`d`	angle in range 0-2π

iau_ANPM *normalize radians to range $-\pi$ to $+\pi$* iau_ANPM

CALL :

```
D = iau_ANPM ( A )
```

ACTION :

Normalize angle into the range $-\pi \leq A < +\pi$.

GIVEN :

A	`d`	angle (radians)

RETURNED (function value) :

	`d`	angle in range $\pm\pi$

iau_C2S *unit vector to spherical* iau_C2S

CALL :

```
CALL iau_C2S ( P, THETA, PHI )
```

ACTION :

P-vector to spherical coordinates.

GIVEN :

P	`d(3)`	p-vector

RETURNED :

THETA	`d`	longitude angle (radians)
PHI	`d`	latitude angle (radians)

NOTES :

1. The vector **P** can have any magnitude; only its direction is used.
2. If **P** is null, zero `THETA` and `PHI` are returned.
3. At either pole, zero `THETA` is returned.

iau_CP *copy p-vector* iau_CP

CALL :

```
CALL iau_CP ( P, C )
```

ACTION :

Copy a p-vector.

GIVEN :

P	`d(3)`	p-vector to be copied

RETURNED :

C	`d(3)`	copy

iau_CPV *copy pv-vector* iau_CPV

CALL :

```
CALL iau_CPV ( PV, C )
```

ACTION :

Copy a position/velocity vector.

GIVEN :

PV	`d(3,2)`	position/velocity vector to be copied

RETURNED :

C	`d(3,2)`	copy

---

iau_CR *copy r-matrix* iau_CR

CALL :

```
CALL iau_CR ( R, C )
```

ACTION :

Copy an r-matrix.

GIVEN :

R	`d(3,3)`	r-matrix to be copied

RETURNED :

C	`d(3,3)`	copy

iau_D2TF *days to hours, minutes, seconds* iau_D2TF

CALL :

```
CALL iau_D2TF ( NDP, DAYS, SIGN, IHMSF )
```

ACTION :

Decompose days to hours, minutes, seconds, fraction.

GIVEN :

NDP	`i`	resolution (Note 1)
DAYS	`d`	interval in days

RETURNED :

SIGN	`c`	'+' or '-'
IHMSF	`i(4)`	hours, minutes, seconds, fraction

NOTES :

1. `NDP` is interpreted as follows:

```
NDP         resolution
 :      ...0000 00 00
-7         1000 00 00
-6          100 00 00
-5           10 00 00
-4            1 00 00
-3            0 10 00
-2            0 01 00
-1            0 00 10
 0            0 00 01
 1            0 00 00.1
 2            0 00 00.01
 3            0 00 00.001
 :            0 00 00.000...
```

2. The largest positive useful value for `NDP` is determined by the size of `DAYS`, the format of `DOUBLE PRECISION` numbers on the target platform, and the risk of overflowing `IHMSF(4)`. On a typical platform, for `DAYS` up to `1D0`, the available floating-point precision might correspond to `NDP = 12`. However, the practical limit is typically `NDP = 9`, set by the capacity of a 32-bit capacity of a 32-bit `IHMSF(4)`.

3. The absolute value of `DAYS` may exceed 1D0. In cases where it does not, it is up to the caller to test for and handle the case where `DAYS` is very nearly `1D0` and rounds up to 24 hours, by testing for `IHMSF(1) .EQ. 24` and setting `IHMSF(1-4)` to zero.

iau_IR *initialize r-matrix to identity* iau_IR

CALL :

```
CALL iau_IR ( R )
```

ACTION :

Initialize an r-matrix to the identity matrix.

RETURNED :

R	`d(3,3)`	r-matrix

iau_P2PV *append zero velocity to p-vector* iau_P2PV

CALL :

```
CALL iau_P2PV ( P, PV )
```

ACTION :

Extend a p-vector to a pv-vector by appending a zero velocity.

GIVEN :

P	`d(3)`	p-vector

RETURNED :

PV	`d(3,2)`	pv-vector

iau_P2S *p-vector to spherical* iau_P2S

CALL :

```
CALL iau_P2S ( P, THETA, PHI, R )
```

ACTION :

P-vector to spherical polar coordinates.

GIVEN :

P	`d(3)`	p-vector

RETURNED :

THETA	`d`	longitude angle (radians)
PHI	`d`	latitude angle (radians)
R	`d`	radial distance

NOTES :

1. If **P** is null, zero `THETA`, `PHI` and `R` are returned.
2. At either pole, zero `THETA` is returned.

iau_PAP *position-angle from p-vectors* iau_PAP

CALL :

```
CALL iau_PAP ( A, B, THETA )
```

ACTION :

Position-angle from two p-vectors.

GIVEN :

A	`d(3)`	direction of reference point
B	`d(3)`	direction of point whose PA is required

RETURNED :

THETA	`d`	position angle of **B** with respect to **A** (radians)

NOTES :

1. The result is the position angle, in radians, of direction **B** with respect to direction **A**. It is in the range $-\pi$ to $+\pi$. The sense is such that if **B** is a small distance "north" of **A** the position angle is approximately zero, and if **B** is a small distance "east" of **A** the position angle is approximately $+\pi/2$.

2. **A** and **B** need not be unit vectors.

3. Zero is returned if the two directions are the same or if either vector is null.

4. If **A** is at a pole, the result is ill-defined.

iau_PAS *position-angle from spherical coordinates* iau_PAS

CALL :

```
CALL iau_PAS ( AL, AP, BL, BP, THETA )
```

ACTION :

Position-angle from spherical coordinates.

GIVEN :

AL	`d`	longitude of point A (*e.g.* RA) in radians
AP	`d`	latitude of point A (*e.g.* Dec) in radians
BL	`d`	longitude of point B
BP	`d`	latitude of point B

RETURNED :

THETA	`d`	position angle of B with respect to A

NOTES :

1. The result is the bearing (position angle), in radians, of point B with respect to point A. It is in the range $-\pi$ to $+\pi$. The sense is such that if B is a small distance "east" of point A, the bearing is approximately $+\pi/2$.

2. Zero is returned if the two points are coincident.

iau_PDP *dot product of two p-vectors* iau_PDP

CALL :

```
CALL iau_PDP ( A, B, ADB )
```

ACTION :

p-vector inner (≡scalar≡dot) product.

GIVEN :

A	`d(3)`	first p-vector
B	`d(3)`	second p-vector

RETURNED :

ADB	`d`	scalar product **A**.**B**

---

iau_PM *modulus of p-vector* iau_PM

CALL :

```
CALL iau_PM ( P, R )
```

ACTION :

Modulus of p-vector.

GIVEN :

P	`d(3)`	p-vector

RETURNED :

R	`d`	modulus $\|\mathbf{P}\|$

iau_PMP *p-vector minus p-vector* iau_PMP

CALL :

```
CALL iau_PMP ( A, B, AMB )
```

ACTION :

P-vector subtraction.

GIVEN :

A	`d(3)`	first p-vector
B	`d(3)`	second p-vector

RETURNED :

AMB	`d(3)`	**A** − **B**

iau_PN *normalize p-vector returning modulus* iau_PN

CALL :

```
CALL iau_PN ( P, R, U )
```

ACTION :

Convert a p-vector into modulus and unit vector.

GIVEN :

P	`d(3)`	p-vector

RETURNED :

R	`d`	modulus $\|\mathbf{P}\|$
U	`d(3)`	unit vector $\hat{\mathbf{P}}$

NOTE :

If `P` is null, the result is null. Otherwise the result is a unit vector.

iau_PPP *p-vector plus p-vector* iau_PPP

CALL :

```
CALL iau_PPP ( A, B, APB )
```

ACTION :

P-vector addition.

GIVEN :

A	`d(3)`	first p-vector
B	`d(3)`	second p-vector

RETURNED :

APB	`d(3)`	$\mathbf{A} + \mathbf{B}$

iau_PPSP *p-vector plus scaled p-vector* iau_PPSP

CALL :

```
CALL iau_PPSP ( A, S, B, APSB )
```

ACTION :

P-vector plus scaled p-vector.

GIVEN :

A	`d(3)`	first p-vector
S	`d`	scalar (multiplier for **B**)
B	`d(3)`	second p-vector

RETURNED :

APSB	`d(3)`	$\mathbf{A} + S \times \mathbf{B}$

iau_PV2P *discard velocity component of pv-vector* iau_PV2P

CALL :

```
CALL iau_PV2P ( PV, P )
```

ACTION :

Discard velocity component of a pv-vector.

GIVEN :

PV	`d(3,2)`	pv-vector

RETURNED :

P	`d(3)`	p-vector

iau_PV2S *pv-vector to spherical* iau_PV2S

CALL :

```
CALL iau_PV2S ( PV, THETA, PHI, R, TD, PD, RD )
```

ACTION :

Convert position/velocity from Cartesian to spherical coordinates.

GIVEN :

PV	`d(3,2)`	pv-vector

RETURNED :

THETA	`d`	longitude angle (radians)
PHI	`d`	latitude angle (radians)
R	`d`	radial distance
TD	`d`	rate of change of THETA
PD	`d`	rate of change of PHI
RD	`d`	rate of change of R

NOTES :

1. If the position part of `PV` is null, `THETA`, `PHI`, `TD` and `PD` are indeterminate. This is handled by extrapolating the position through unit time by using the velocity part of `PV`. This moves the origin without changing the direction of the velocity component. If the position and velocity components of `PV` are both null, zeroes are returned for all six results.

2. If the position is a pole, `THETA`, `TD` and `PD` are indeterminate. In such cases zeroes are returned for all three.

iau_PVDPV *dot product of two pv-vectors* iau_PVDPV

CALL :

```
CALL iau_PVDPV ( A, B, ADB )
```

ACTION :

Inner ($\equiv$ scalar $\equiv$ dot) product of two pv-vectors.

GIVEN :

A	`d(3,2)`	first pv-vector
B	`d(3,2)`	second pv-vector

RETURNED :

ADB	`d(2)`	*A.B* (see note)

NOTE :

If the position and velocity components of the two pv-vectors are $(\mathbf{A}_p, \mathbf{A}_v)$ and $(\mathbf{B}_p, \mathbf{B}_v)$, the result, *A.B*, is the pair of numbers $(\mathbf{A}_p.\mathbf{B}_p, \mathbf{A}_p.\mathbf{B}_v + \mathbf{A}_v.\mathbf{B}_p)$. The two numbers are the dot-product of the two p-vectors and its derivative.

iau_PVM *modulus of pv-vector* iau_PVM

CALL :

```
CALL iau_PVM ( PV, R, S )
```

ACTION :

Modulus of pv-vector.

GIVEN :

PV	`d(3,2)`	pv-vector

RETURNED :

R	`d`	modulus of position component
S	`d`	modulus of velocity component

iau_PVMPV *pv-vector minus pv-vector* iau_PVMPV

CALL :

```
CALL iau_PVMPV ( A, B, AMB )
```

ACTION :

Subtract one pv-vector from another.

GIVEN :

A	`d(3,2)`	first pv-vector
B	`d(3,2)`	second pv-vector

RETURNED :

AMB	`d(3,2)`	$A - B$

iau_PVPPV *pv-vector plus pv-vector* iau_PVPPV

CALL :

```
CALL iau_PVPPV ( A, B, APB )
```

ACTION :

Add one pv-vector to another.

GIVEN :

A	`d(3,2)`	first pv-vector
B	`d(3,2)`	second pv-vector

RETURNED :

APB	`d(3,2)`	$A + B$

iau_PVU *update pv-vector* iau_PVU

CALL :

```
CALL iau_PVU ( DT, PV, UPV )
```

ACTION :

Update a pv-vector.

GIVEN :

DT	`d`	time interval
PV	`d(3,2)`	pv-vector

RETURNED :

UPV	`d(3,2)`	position part of `PV` updated, velocity part unchanged

NOTES :

1. "Update" means "refer the position component of the vector to a new date `DT` time units from the existing date".
2. The time units of `DT` must match those of the velocity.

iau_PVUP *update pv-vector discarding velocity* iau_PVUP

CALL :

```
CALL iau_PVUP ( DT, PV, P )
```

ACTION :

Update a pv-vector, discarding the velocity component.

GIVEN :

DT	`d`	time interval
PV	`d(3,2)`	pv-vector

RETURNED :

P	`d(3)`	p-vector

NOTES :

1. "Update" means "refer the position component of the vector to a new date `DT` time units from the existing date".

2. The time units of `DT` must match those of the velocity.

iau_PVXPV *cross product of two pv-vectors* iau_PVXPV

CALL :

```
CALL iau_PVXPV ( A, B, AXB )
```

ACTION :

Outer ($\equiv$ vector $\equiv$ cross) product of two pv-vectors.

GIVEN :

A	`d(3,2)`	first pv-vector
B	`d(3,2)`	second pv-vector

RETURNED :

AXB	`d(3,2)`	$A \wedge B$

NOTE :

If the position and velocity components of the two pv-vectors are ($\mathbf{A}_p$, $\mathbf{A}_v$) and ($\mathbf{B}_p$, $\mathbf{B}_v$), the result, $A \wedge B$, is the pair of vectors ($\mathbf{A}_p \wedge \mathbf{B}_p$, $\mathbf{A}_p \wedge \mathbf{B}_v + \mathbf{A}_v \wedge \mathbf{B}_p$). The two vectors are the cross-product of the two p-vectors and its derivative.

iau_PXP *cross product of two p-vectors* iau_PXP

CALL :

```
CALL iau_PXP ( A, B, AXB )
```

ACTION :

p-vector outer ($\equiv$ vector $\equiv$ cross) product.

GIVEN :

A	`d(3)`	first p-vector
B	`d(3)`	second p-vector

RETURNED :

AXB	`d(3)`	$A \wedge B$

iau_RM2V *r-matrix to r-vector* iau_RM2V

CALL :

```
CALL iau_RM2V ( R, W )
```

ACTION :

Express an r-matrix as an r-vector.

GIVEN :

R	`d(3,3)`	rotation matrix

RETURNED :

W	`d(3)`	rotation vector (Note 1)

NOTES :

1. A rotation matrix describes a rotation through some angle about some arbitrary axis called the Euler axis. The "rotation vector" returned by this routine has the same direction as the Euler axis, and its magnitude is the angle in radians. (The magnitude and direction can be separated by means of the routine `iau_PN`.)

2. If `R` is null, so is the result. If `R` is not a rotation matrix the result is undefined. `R` must be proper (*i.e.* have a positive determinant) and real orthogonal (inverse = transpose).

3. The reference frame rotates clockwise as seen looking along the rotation vector from the origin.

iau_RV2M *r-vector to r-matrix* iau_RV2M

CALL :

```
CALL iau_RV2M ( W, R )
```

ACTION :

Form the r-matrix corresponding to a given r-vector.

GIVEN :

W	`d(3)`	rotation vector (Note 1)

RETURNED :

R	`d(3,3)`	rotation matrix

NOTES :

1. A rotation matrix describes a rotation through some angle about some arbitrary axis called the Euler axis. The "rotation vector" supplied to this routine has the same direction as the Euler axis, and its magnitude is the angle in radians.

2. If **W** is null, the unit matrix is returned.

3. The reference frame rotates clockwise as seen looking along the rotation vector from the origin.

iau_RX *rotate r-matrix about x axis* iau_RX

CALL :

```
CALL iau_RX ( PHI, R )
```

ACTION :

Rotate an r-matrix about the x-axis.

GIVEN :

PHI	`d`	angle ϕ (radians)

GIVEN and **RETURNED** :

R	`d(3,3)`	r-matrix, rotated

NOTES :

1. Calling this routine with positive ϕ incorporates in the supplied r-matrix `R` an additional rotation, about the x-axis, anticlockwise as seen looking towards the origin from positive x.

2. The additional rotation can be represented by this matrix:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & +\cos\phi & +\sin\phi \\ 0 & -\sin\phi & +\cos\phi \end{pmatrix}$$

iau_RXP *product of r-matrix and p-vector* iau_RXP

CALL :

```
CALL iau_RXP ( R, P, RP )
```

ACTION :

Multiply a p-vector by an r-matrix.

GIVEN :

R	`d(3,3)`	r-matrix
P	`d(3)`	p-vector

RETURNED :

RP	`d(3)`	$R * P$

iau_RXPV *product of r-matrix and pv-vector* iau_RXPV

CALL :

```
CALL iau_RXPV ( R, PV, RPV )
```

ACTION :

Multiply a pv-vector by an r-matrix.

GIVEN :

R	`d(3,3)`	r-matrix
PV	`d(3,2)`	pv-vector

RETURNED :

RPV	`d(3,2)`	$R * PV$

NOTE :

The algorithm is for the simple case where the r-matrix R is not a function of time. The case where R is a function of time leads to an additional velocity component equal to the product of the derivative of R and the position part of PV.

iau_RXR *r-matrix multiply* iau_RXR

CALL :

```
CALL iau_RXR ( A, B, ATB )
```

ACTION :

Multiply two r-matrices.

GIVEN :

A	`d(3,3)`	first r-matrix
B	`d(3,3)`	second r-matrix

RETURNED :

ATB	`d(3,3)`	$A * B$

iau_RY *rotate r-matrix about y axis* iau_RY

CALL :

```
CALL iau_RY ( THETA, R )
```

ACTION :

Rotate an r-matrix about the y-axis.

GIVEN :

THETA	`d`	angle θ (radians)

GIVEN and **RETURNED** :

R	`d(3,3)`	r-matrix, rotated

NOTES :

1. Calling this routine with positive θ incorporates in the supplied r-matrix `R` an additional rotation, about the y-axis, anticlockwise as seen looking towards the origin from positive y.

2. The additional rotation can be represented by this matrix:

$$\begin{pmatrix} +\cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ +\sin\theta & 0 & +\cos\theta \end{pmatrix}$$

iau_RZ — *rotate r-matrix about z axis* — iau_RZ

CALL :

```
CALL iau_RZ ( PSI, R )
```

ACTION :

Rotate an r-matrix about the z-axis.

GIVEN :

PSI	`d`	angle ψ (radians)

GIVEN and **RETURNED** :

R	`d(3,3)`	r-matrix, rotated

NOTES :

1. Calling this routine with positive ψ incorporates in the supplied r-matrix `R` an additional rotation, about the z-axis, anticlockwise as seen looking towards the origin from positive z.

2. The additional rotation can be represented by this matrix:

$$\begin{pmatrix} +\cos\psi & +\sin\psi & 0 \\ -\sin\psi & +\cos\psi & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

iau_S2C *spherical to unit vector* iau_S2C

CALL :

```
CALL iau_S2C ( THETA, PHI, C )
```

ACTION :

Convert spherical coordinates to Cartesian.

GIVEN :

THETA	`d`	longitude angle (radians)
PHI	`d`	latitude angle (radians)

RETURNED :

C	`d(3)`	direction cosines

iau_S2P *spherical to p-vector* iau_S2P

CALL :

```
CALL iau_S2P ( THETA, PHI, R, P )
```

ACTION :

Convert spherical polar coordinates to p-vector.

GIVEN :

THETA	`d`	longitude angle (radians)
PHI	`d`	latitude angle (radians)
R	`d`	radial distance

RETURNED :

P	`d(3)`	Cartesian coordinates

iau_S2PV *spherical to pv-vector* iau_S2PV

CALL :

```
CALL iau_S2PV ( THETA, PHI, R, TD, PD, RD, PV )
```

ACTION :

Convert position/velocity from spherical to Cartesian coordinates.

GIVEN :

THETA	`d`	longitude angle (radians)
PHI	`d`	latitude angle (radians)
R	`d`	radial distance
TD	`d`	rate of change of `THETA`
PD	`d`	rate of change of `PHI`
RD	`d`	rate of change of `R`

RETURNED :

PV	`d(3,2)`	pv-vector

iau_S2XPV *multiply pv-vector by two scalars* iau_S2XPV

CALL :

```
CALL iau_S2XPV ( S1, S2, PV, SPV )
```

ACTION :

Multiply a pv-vector by two scalars.

GIVEN :

S1	`d`	scalar to multiply position component by
S2	`d`	scalar to multiply velocity component by
PV	`d(3,2)`	pv-vector

RETURNED :

SPV	`d(3,2)`	pv-vector: p scaled by `S1`, v scaled by `S2`

iau_SEPP *angular separation from p-vectors* iau_SEPP

CALL :

```
CALL iau_SEPP ( A, B, S )
```

ACTION :

Angular separation between two p-vectors.

GIVEN :

A	`d(3)`	first p-vector (not necessarily unit length)
B	`d(3)`	second p-vector (not necessarily unit length)

RETURNED :

S	`d`	angular separation (radians, always positive)

NOTES :

1. If either vector is null, a zero result is returned.
2. The angular separation is most simply formulated in terms of scalar product. However, this gives poor accuracy for angles near zero and π. The present algorithm uses both cross product and dot product, to deliver full accuracy whatever the size of the angle.

iau_SEPS *angular separation from spherical coordinates* iau_SEPS

CALL :

```
CALL iau_SEPS ( AL, AP, BL, BP, S )
```

ACTION :

Angular separation between two sets of spherical coordinates.

GIVEN :

AL	`d`	first longitude (radians)
AP	`d`	first latitude (radians)
BL	`d`	second longitude (radians)
BP	`d`	second latitude (radians)

RETURNED :

S	`d`	angular separation (radians)

iau_SXP *multiply p-vector by scalar* iau_SXP

CALL :

```
CALL iau_SXP ( S, P, SP )
```

ACTION :

Multiply a p-vector by a scalar.

GIVEN :

S	`d`	scalar
P	`d(3)`	p-vector

RETURNED :

SP	`d(3)`	$S * \mathbf{P}$

iau_SXPV *multiply pv-vector by scalar* iau_SXPV

CALL :

```
CALL iau_SXPV ( S, PV, SPV )
```

ACTION :

Multiply a pv-vector by a scalar.

GIVEN :

S	`d`	scalar
PV	`d(3,2)`	pv-vector

RETURNED :

SPV	`d(3,2)`	*S* * *PV*

iau_TF2A *hours, minutes, seconds to radians* iau_TF2A

CALL :

```
CALL iau_TF2A ( S, IHOUR, IMIN, SEC, RAD, J )
```

ACTION :

Convert hours, minutes, seconds to radians.

GIVEN :

S	`c`	sign: `'-'` = negative, otherwise positive
IHOUR	`i`	hours
IMIN	`i`	minutes
SEC	`d`	seconds

RETURNED :

RAD	`d`	angle in radians
J	`i`	status: 0 = OK 1 = `IHOUR` outside range 0-23 2 = `IMIN` outside range 0-59 3 = `SEC` outside range 0-59.999...

NOTES :

1. If the `S` argument is a string, only the leftmost character is used and no warning status is provided.
2. The result is computed even if any of the range checks fail.
3. Negative `IHOUR`, `IMIN` and/or `SEC` produce a warning status, but the absolute value is used in the conversion.
4. If there are multiple errors, the status value reflects only the first, the smallest taking precedence.

iau_TF2D *hours, minutes, seconds to days* iau_TF2D

CALL :

```
CALL iau_TF2D ( S, IHOUR, IMIN, SEC, DAYS, J )
```

ACTION :

Convert hours, minutes, seconds to days.

GIVEN :

S	`c`	sign: `'-'` = negative, otherwise positive
IHOUR	`i`	hours
IMIN	`i`	minutes
SEC	`d`	seconds

RETURNED :

DAYS	`d`	interval in days
J	`i`	status: 0 = OK 1 = `IHOUR` outside range 0-23 2 = `IMIN` outside range 0-59 3 = `SEC` outside range 0-59.999...

NOTES :

1. If the `S` argument is a string, only the leftmost character is used and no warning status is provided.
2. The result is computed even if any of the range checks fail.
3. Negative `IHOUR`, `IMIN` and/or `SEC` produce a warning status, but the absolute value is used in the conversion.
4. If there are multiple errors, the status value reflects only the first, the smallest taking precedence.

iau_TR *transpose r-matrix* iau_TR

CALL :

```
CALL iau_TR ( R, RT )
```

ACTION :

Transpose an r-matrix.

GIVEN :

R	`d(3,3)`	r-matrix

RETURNED :

RT	`d(3,3)`	transpose

iau_TRXP *product of r-matrix transpose and p-vector* iau_TRXP

CALL :

```
CALL iau_TRXP ( R, P, TRP )
```

ACTION :

Multiply a p-vector by the transpose of an r-matrix.

GIVEN :

R	`d(3,3)`	r-matrix
P	`d(3)`	p-vector

RETURNED :

TRP	`d(3)`	$R^T \times P$

iau_TRXPV *product of r-matrix transpose and pv-vector* iau_TRXPV

CALL :

```
CALL iau_TRXPV ( R, PV, TRPV )
```

ACTION :

Multiply a pv-vector by the transpose of an r-matrix.

GIVEN :

R	`d(3,3)`	r-matrix
PV	`d(3,2)`	pv-vector

RETURNED :

$TRPV$	`d(3,2)`	$R^T \times PV$

NOTE :

The algorithm is for the simple case where the r-matrix R is not a function of time. The case where R is a function of time leads to an additional velocity component equal to the product of the derivative of R^T and the position part of PV.

iau_ZP *zero p-vector* iau_ZP

CALL :

```
CALL iau_ZP ( P )
```

ACTION :

Zero a p-vector.

RETURNED :

P	`d(3)`	zero p-vector

iau_ZPV *zero pv-vector* iau_ZPV

CALL :

```
CALL iau_ZPV ( PV )
```

ACTION :

Zero a pv-vector.

RETURNED :

PV	`double(3,2)`	zero pv-vector

iau_ZR *initialize r-matrix to null* iau_ZR

CALL :

```
CALL iau_ZR ( R )
```

ACTION :

Initialize an r-matrix to the null matrix.

RETURNED :

R	`double(3,3)`	null r-matrix

3.4 Classified list of routines

Operations involving p-vectors and r-matrices

initialize

`CALL iau_ZP ( P )`
zero p-vector p71

`CALL iau_ZR ( R )`
initialize r-matrix to null p73

`CALL iau_IR ( R )`
initialize r-matrix to identity p29

copy

`CALL iau_CP ( P, C )`
copy p-vector p25

`CALL iau_CR ( R, C )`
copy r-matrix p27

build rotations

`CALL iau_RX ( PHI, R )`
rotate r-matrix about x p52

`CALL iau_RY ( THETA, R )`
rotate r-matrix about y p56

`CALL iau_RZ ( PSI, R )`
rotate r-matrix about z p57

spherical/Cartesian conversions

`CALL iau_S2C ( THETA, PHI, C )`
spherical to unit vector p58

`CALL iau_C2S ( P, THETA, PHI )`
unit vector to spherical p24

`CALL iau_S2P ( THETA, PHI, R, P )`
spherical to p-vector p59

`CALL iau_P2S ( P, THETA, PHI, R )`
p-vector to spherical p31

operations on p-vectors

`CALL iau_PPP ( A, B, APB )`
p-vector plus p-vector p38

`CALL iau_PMP ( A, B, AMB )`
p-vector minus p-vector p36

`CALL iau_PPSP ( A, S, B, APSB )`
p-vector plus scaled p-vector p39

`CALL iau_PDP ( A, B, ADB )`
inner (=scalar=dot) product of two p-vectors p34

`CALL iau_PXP ( A, B, AXB )`
outer (=vector=cross) product of two p-vectors p49

`CALL iau_PM ( P, R )`
modulus of p-vector p35

`CALL iau_PN ( P, R, U )`
normalize p-vector returning modulus p37

`CALL iau_SXP ( S, P, SP )`
multiply p-vector by scalar p64

operations on r-matrices

`CALL iau_RXR ( A, B, ATB )`
r-matrix multiply p55

`CALL iau_TR ( R, RT )`
transpose r-matrix p68

matrix-vector products

`CALL iau_RXP ( R, P, RP )`
product of r-matrix and p-vector p53

`CALL iau_TRXP ( R, P, TRP )`
product of transpose of r-matrix and p-vector p69

separation and position-angle

`CALL iau_SEPP ( A, B, S )`
angular separation from p-vectors p62

`CALL iau_SEPS ( AL, AP, BL, BP, S )`
angular separation from spherical coordinates p63

`CALL iau_PAP ( A, B, THETA )`
position-angle from p-vectors p32

`CALL iau_PAS ( AL, AP, BL, BP, THETA )`
position-angle from spherical coordinates p33

rotation vectors

`CALL iau_RV2M ( P, R )`
r-vector to r-matrix p51

`CALL iau_RM2V ( R, P )`
r-matrix to r-vector p50

Operations involving pv-vectors

initialize

`CALL iau_ZPV ( PV )`
zero pv-vector p72

copy/extend/extract

`CALL iau_CPV ( PV, C )`
copy pv-vector p26

`CALL iau_P2PV ( P, PV )`
append zero velocity to p-vector p30

`CALL iau_PV2P ( PV, P )`
discard velocity component of pv-vector p40

spherical/Cartesian conversions

`CALL iau_S2PV ( THETA, PHI, R, TD, PD, RD, PV )`
spherical to pv-vector p60

`CALL iau_PV2S ( PV, THETA, PHI, R, TD, PD, RD )`
pv-vector to spherical p41

operations on pv-vectors

`CALL iau_PVPPV ( A, B, APB )`
pv-vector plus pv-vector p45

`CALL iau_PVMPV ( A, B, AMB )`
pv-vector minus pv-vector p44

`CALL iau_PVDPV ( A, B, ADB )`
inner (=scalar=dot) product of two pv-vectors p42

`CALL iau_PVXPV ( A, B, AXB )`
outer (=vector=cross) product of two pv-vectors p48

`CALL iau_PVM ( PV, R, S )`
modulus of pv-vector p43

`CALL iau_SXPV ( S, PV, SPV )`
multiply pv-vector by scalar p65

`CALL iau_S2XPV ( S1, S2, PV )`
multiply pv-vector by two scalars p61

`CALL iau_PVU ( DT, PV, UPV )`
update pv-vector p46

`CALL iau_PVUP ( DT, PV, P )`
update pv-vector discarding velocity p47

matrix-vector products

`CALL iau_RXPV ( R, PV, RPV )`
product of r-matrix and pv-vector p54

`CALL iau_TRXPV ( R, PV, TRPV )`
product of transpose of r-matrix and pv-vector p70

Operations on angles

wrap

`D = iau_ANP ( A )`
normalize radians to range 0 to 2π p22

`D = iau_ANPM ( A )`
normalize radians to range $-\pi$ to $+\pi$ p23

to sexagesimal

`CALL iau_A2AF ( NDP, ANGLE, SIGN, IDMSF )`
decompose radians into degrees, arcminutes, arcseconds p19

`CALL iau_A2TF ( NDP, ANGLE, SIGN, IHMSF )`
decompose radians into hours, minutes, seconds p20

`CALL iau_D2TF ( NDP, DAYS, SIGN, IHMSF )`
decompose days into hours, minutes, seconds p28

from sexagesimal

`CALL iau_AF2A ( S, IDEG, IAMIN, ASEC, RAD, J )`
degrees, arcminutes, arcseconds to radians p21

`CALL iau_TF2A ( S, IHOUR, IMIN, SEC, RAD, J )`
hours, minutes, seconds to radians p66

`CALL iau_TF2D ( S, IHOUR, IMIN, SEC, DAYS, J )`
hours, minutes, seconds to days p67

3.5 Calls: quick reference