International Astronomical Union

Standards Of Fundamental Astronomy

SOFA Vector-Matrix Library

Software version 15
Document revision 1.00
Version for C programming language

http://www.iausofa.org

2020 October 25

MEMBERS OF THE IAU SOFA BOARD (2020)

John Bangert	United States Naval Observatory (retired)
Steven Bell	Her Majesty's Nautical Almanac Office
Nicole Capitaine	Paris Observatory
Mickaël Gastineau	Paris Observatory, IMCCE
Catherine Hohenkerk	Her Majesty's Nautical Almanac Office (chair, retired)
Li Jinling	Shanghai Astronomical Observatory
Brian Luzum	United States Naval Observatory (IERS)
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
Dennis McCarthy	United States Naval Observatory
Skip Newhall	Jet Propulsion Laboratory
Jin Wen-Jing	Shanghai Observatory

Contents

1 Preliminaries **1**
1.1 Introduction . . . 1
1.2 General principles . . . 1

2 Guide to the VML functions **3**
2.1 Spherical trigonometry . . . 3
2.1.1 Formatting angles . . . 4
2.2 P-vectors and R-matrices . . . 6
2.2.1 SOFA functions for vectors and matrices . . . 7
2.3 R-vectors . . . 10
2.4 PV-vectors . . . 12

3 Reference material **13**
3.1 SOFA vector-matrix conventions . . . 13
3.1.1 p-vectors . . . 13
3.1.2 pv-vectors . . . 14
3.1.3 r-matrices . . . 14
3.2 The twelve r-matrices . . . 16
3.3 Function specifications . . . 19
iauA2af . . . 19
iauA2tf . . . 20
iauAf2a . . . 21
iauAnp . . . 22
iauAnpm . . . 23
iauC2s . . . 24
iauCp . . . 25
iauCpv . . . 26
iauCr . . . 27
iauD2tf . . . 28
iauIr . . . 29
iauP2pv . . . 30
iauP2s . . . 31
iauPap . . . 32
iauPas . . . 33
iauPdp . . . 34
iauPm . . . 35
iauPmp . . . 36
iauPn . . . 37
iauPpp . . . 38
iauPpsp . . . 39
iauPv2p . . . 40
iauPv2s . . . 41
iauPvdpv . . . 42
iauPvm . . . 43
iauPvmpv . . . 44
iauPvppv . . . 45

iauPvu . . . 46
iauPvup . . . 47
iauPvxpv . . . 48
iauPxp . . . 49
iauRm2v . . . 50
iauRv2m . . . 51
iauRx . . . 52
iauRxp . . . 53
iauRxpv . . . 54
iauRxr . . . 55
iauRy . . . 56
iauRz . . . 57
iauS2c . . . 58
iauS2p . . . 59
iauS2pv . . . 60
iauS2xpv . . . 61
iauSepp . . . 62
iauSeps . . . 63
iauSxp . . . 64
iauSxpv . . . 65
iauTf2a . . . 66
iauTf2d . . . 67
iauTr . . . 68
iauTrxp . . . 69
iauTrxpv . . . 70
iauZp . . . 71
iauZpv . . . 72
iauZr . . . 73
3.4 Classified list of functions . . . 74
Operations involving p-vectors and r-matrices . . . 74
Operations involving pv-vectors . . . 76
Operations on angles . . . 77
3.5 Calls: quick reference . . . 78

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 189 astronomy functions supported by 55 utility functions 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 functions.

The present document concerns the VML angle/vector/matrix tools, that were either implemented in the course of writing the astronomical functions 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 functions 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 functions 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 function can be found in Section 3.3, which is arranged alphabetically.

1.2 General principles

The SOFA VML consists mostly of functions 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[3]`.
- "Position/velocity vectors" or "pv-vectors" are `double[2][3]`. 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 `iauCp` function (copy a p-vector) can be used to copy the velocity component of a pv-vector (indeed, this is how the `iauCpv` function 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[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 functions 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[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 functions 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 functions that are required by SOFA's astronomical software.

2 Guide to the VML functions

This section outlines a number of positional-astronomy topics, for background and to provide a context in which the various functions 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 functions that operate directly on generic spherical coordinates are provided: `iauSeps` computes the angular separation between two points (*i.e.* the distance along a great circle) and `iauPas` computes the bearing or *position angle* of one point seen from the other. As a simple demonstration, we will use these two functions (and a spherical-Earth approximation) to estimate the distance from London to Sydney and the initial compass heading:

```
#include <sofa.h>
#include <stdio.h>
#define R2D 57.2957795     /* radians to degrees */
#define RKM 6375.0         /* Earth radius in km */
int main ( )
{
```

```
/* Longitudes and latitudes (radians) for London and Sydney. */
   double al =  -0.2/R2D, bl =  51.5/R2D,
          as = 151.2/R2D, bs = -33.9/R2D;

/* Great-circle distance and initial heading. */
   printf ( "%5.0f km,%4.0f deg\n", iauSeps(al,bl,as,bs)*RKM,
                                    iauPas(al,bl,as,bs)*R2D );
   return 0;
}
```

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

The functions `iauSepp` (p62) and `iauPap` (p32) are equivalents of `iauSeps` (p63) and `iauPas` (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 functions 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 C's low-level `scanf` 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 functions for expressing an angle in radians in a preferred range:

- normalize radians to range 0 to 2π, `iauAnpm` (p23)
- normalize radians to range $-\pi$ to $+\pi$, `iauAnp` (p22)

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

- decompose radians into degrees, arcminutes, arcseconds, `iauA2af`, p19

[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 hours, minutes, seconds, `iauA2tf`, p20
- decompose days into hours, minutes, seconds, `iauD2tf`, p28
- degrees, arcminutes, arcseconds to radians, `iauAf2a`, p21
- hours, minutes, seconds to radians, `iauTf2a`, p66
- hours, minutes, seconds to days, `iauTf2d`, 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:

```
#include <sofa.h>
#include <stdio.h>
int main ( )
{
   double ha;
   char sign;
   int ihmsf[4];

   ha = -0.261799315;
   iauA2tf ( 3, ha, &sign, ihmsf );
   printf ( "%c%2.2d %2.2d %2.2d.%3.3d\n",
            sign, ihmsf[0], ihmsf[1], ihmsf[2], ihmsf[3] );
   iauA2tf ( 2, ha, &sign, ihmsf );
   printf ( "%c%2.2d %2.2d %2.2d.%2.2d\n",
            sign, ihmsf[0], ihmsf[1], ihmsf[2], ihmsf[3] );
   return 0;
}
```

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 function `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 functions 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 functions:

- spherical to unit vector, `iauS2c`, p58
- unit vector to spherical, `iauC2s`, p24
- spherical to p-vector, `iauS2p`, p59
- p-vector to spherical, `iauP2s`, p31

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

- zero p-vector, `iauZp`, p71
- p-vector plus p-vector, `iauPpp`, p38
- p-vector minus p-vector, `iauPmp`, p35
- p-vector plus scaled p-vector, `iauPpsp`, p39
- inner (=scalar=dot) product of two p-vectors, `iauPdp`, p34

- outer (=vector=cross) product of two p-vectors, `iauPxp`, p49
- modulus of p-vector, `iauPm`, p35
- normalize p-vector returning modulus, `iauPn`, p37
- multiply p-vector by scalar, `iauSxp`, p64

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

- copy p-vector, `iauCp`, p25
- copy r-matrix, `iauCr`, p27

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

- r-matrix multiply, `iauRxr`, p55
- transpose r-matrix, `iauTr`, p68

. . . and two for matrix-vector products:

- product of r-matrix and p-vector, `iauRxp`, p53
- product of transpose of r-matrix and p-vector, `iauTrxp`, 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, `iauZr`, p73
- initialize r-matrix to identity, `iauIr`, 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 functions. . .

- rotate r-matrix about x, `iauRx`, p52
- rotate r-matrix about y, `iauRy`, p56
- rotate r-matrix about z, `iauRz`, 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:

```
#include <sofa.h>
#include <stdio.h>
int main ()
{
   double 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. */
   (void) iauTf2a ( '+', 04, 42, 38.66073910, &ra );
   (void) iauAf2a ( '-', 00, 17, 43.4203921, &da );

/* RA,Dec of the same source in the ICRF3 (S/X) catalog. */
   (void) iauTf2a ( '+', 04, 42, 38.66072366, &rb );
   (void) iauAf2a ( '-', 00, 17, 43.4209582, &db );

/* Method 1:  spherical trigonometry (cosine rule). */
   theta = acos(sin(da)*sin(db) + cos(da)*cos(db)*cos(rb-ra));
   printf ( "The two positions are %9.6f arcsec apart.\n",
            theta*206264.8062470963551564733573 );

/* Method 2:  vectors (sine and cosine from cross and dot product). */
   iauS2c ( ra, da, a );
   iauS2c ( rb, db, b );
   iauPxp ( a, b, axb );
   s = iauPm ( axb );
   c = iauPdp ( a, b );
   theta = atan2 ( s, c );
   printf ( "The two positions are %9.6f arcsec apart.\n",
            theta*206264.8062470963551564733573 );

   return 0;
}
```

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 functions `iauSeps` (p63) and `iauSepp` (p62), and of course the six statements of Method 2 could be replaced by a single call to `iauSeps` 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]

```
#include <sofa.h>
#include <stdio.h>

int main ()
{
   char sign;
   int i4[4];
   double rb[3][3], vb[3], angle, ra, dec;

/* Generate frame bias matrix. */
   iauIr ( rb );
   iauRz ( -0.0146*DAS2R, rb );
   iauRy ( -0.041775*DAS2R * sin(84381.448*DAS2R), rb );
   iauRx ( 0.0068192*DAS2R, rb );

/* Convert into r-vector form. */
   iauRm2v ( rb, vb );

/* Report. */
   iauP2s ( vb, &ra, &dec, &angle );
   printf ( "Frame bias is %4.1f mas around", angle*1e3/DAS2R );
   iauA2tf ( 1, iauAnp(ra), &sign, i4 );
   printf ( "%3.2d %2.2d %2.2d.%d", i4[0], i4[1], i4[2], i4[3] );
   iauA2af ( 0, dec, &sign, i4 );
   printf ( " %c%2.2d %2.2d %2.2d GCRS.\n", sign, i4[0], i4[1], i4[2] );

   return 0;
}
```

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 `iauBi00` or would generate the matrix by calling `iauPfw06` followed by `iauFw2m`.

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

- r-matrix to r-vector, `iauRm2v`, p50
- r-vector to r-matrix, `iauRv2m`, 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 functions (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 functions are provided:

- zero a pv-vector, `iauZpv`, p72
- copy a pv-vector, `iauCpv`, p26
- create a pv-vector by appending zero velocity to a p-vector, `iauP2pv`, p30
- dispense with the velocity to leave a p-vector, `iauPv2p`, p40
- create a pv-vector from spherical coordinates, `iauS2pv`, p60
- transform a pv-vector into spherical coordinates, `iauPv2s`, p41
- add two pv-vectors together, `iauPvppv`, p45
- subtract one pv-vector from another, `iauPvmpv`, p44
- form the scalar product of two pv-vectors, `iauPvdpv`, p42
- form the vector product of two pv-vectors, `iauPvxpv`, p48
- find the modulus of a pv-vector *i.e.* extract distance and speed, `iauPvm`, p43
- multiply position and velocity by a scalar, `iauSxpv`, p65
- multiply position and velocity separately by different scalars, `iauS2xpv`, p61
- update the position part of a pv-vector, `iauPvu`, p46
- update the position part of a pv-vector returning only the position, `iauPvup`, p47
- product of r-matrix and pv-vector, `iauRxpv`, p54
- product of transpose of r-matrix and pv-vector, `iauTrxpv`, 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 C implementation, it will be useful to compare and contrast the two supported languages, the other being Fortran. 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 function `iauPm`:

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

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 `iauPdp` function, 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 `iauPxp` function, 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 functions can be used to process either the position part or the velocity part without the function 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 $\mathbf{R}$ 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 function `iauRxp` (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 `iauTrxp` (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 function `iauRxr` (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.

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

```
double rm[3][3], phi, theta, psi;

slaIr ( rm );
slaRz ( phi, rm );
slaRx ( theta, rm );
slaRz ( 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 Function specifications

---

iauA2af *radians to deg, arcmin, arcsec* **iauA2af**

CALL :

```
iauA2af ( ndp, angle, &sign, idmsf );
```

ACTION :

Decompose radians into degrees, arcminutes, arcseconds, fraction.

GIVEN :

ndp	`int`	resolution (Note 1)
angle	`double`	angle in radians

RETURNED :

sign	`char`	'+' or '-'
idmsf	`int[4]`	degrees, arcminutes, arcseconds, fraction

NOTES :

1. The argument `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`s numbers on the target platform, and the risk of overflowing `idmsf[3]`. 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[3]`.

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[0] == 360` and setting `idmsf[0-3]` to zero.

iauA2tf *radians to hours, minutes, seconds* iauA2tf

CALL :

```
iauA2tf ( ndp, angle, &sign, ihmsf );
```

ACTION :

Decompose radians into hours, minutes, seconds, fraction.

GIVEN :

ndp	`int`	resolution (Note 1)
angle	`double`	angle in radians

RETURNED :

sign	`char`	'+' or '-'
ihmsf	`int[4]`	hours, minutes, seconds, fraction

NOTES :

1. The argument `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`s numbers on the target platform, and the risk of overflowing `ihmsf[3]`. 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[3]`.

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[0] == 360` and setting `idmsf[0-3]` to zero.

iauAf2a *deg, arcmin, arcsec to radians* iauAf2a

CALL :

```
j = iauAf2a ( s, ideg, iamin, asec, &rad );
```

ACTION :

Convert degrees, arcminutes, arcseconds to radians.

GIVEN :

s	`char`	sign: `'-'` = negative, otherwise positive
ideg	`int`	degrees
iamin	`int`	arcminutes
asec	`double`	arcseconds

RETURNED :

rad	`double`	angle in radians

RETURNED (function value) :

`int`	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.

---

iauAnp *normalize radians to range 0 to 2π.* **iauAnp**

CALL :

```
d = iauAnp ( a );
```

ACTION :

Normalize angle into the range $0 \leq a < 2\pi$.

GIVEN :

a	`double`	angle (radians)

RETURNED (function value) :

	`double`	angle in range 0-2π

---

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

CALL :

```
d = iauAnpm ( a );
```

ACTION :

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

GIVEN :

a	`double`	angle (radians)

RETURNED (function value) :

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

iauC2s *unit vector to spherical* iauC2s

CALL :

```
iauC2s ( p, &theta, &phi );
```

ACTION :

P-vector to spherical coordinates.

GIVEN :

p	`double[3]`	p-vector

RETURNED :

theta	`double`	longitude angle (radians)
phi	`double`	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.

---

iauCp *copy p-vector* iauCp

CALL :

```
iauCp ( p, c );
```

ACTION :

Copy a p-vector.

GIVEN :

p	`double[3]`	p-vector to be copied

RETURNED :

c	`double[3]`	copy

---

iauCpv *copy pv-vector* iauCpv

CALL :

```
iauCpv ( pv, c );
```

ACTION :

Copy a position/velocity vector.

GIVEN :

pv	`double[2][3]`	position/velocity vector to be copied

RETURNED :

c	`double[2][3]`	copy

---

iauCr *copy r-matrix* iauCr

CALL :

```
iauCr ( r, c );
```

ACTION :

Copy an r-matrix.

GIVEN :

r	`double[3][3]`	r-matrix to be copied

RETURNED :

c	`double[3][3]`	copy

iauD2tf *days to hours, minutes, seconds* iauD2tf

CALL :

```
iauD2tf ( ndp, days, &sign, ihmsf );
```

ACTION :

Decompose days to hours, minutes, seconds, fraction.

GIVEN :

ndp	`int`	resolution (Note 1)
days	`double`	interval in days

RETURNED :

sign	`char`	'+' or '-'
ihmsf	`int[4]`	hours, minutes, seconds, fraction

NOTES :

1. The argument `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`s on the target platform, and the risk of overflowing `ihmsf[3]`. On a typical platform, for `days` up to `1.0`, 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 `days` may exceed `1.0`. In cases where it does not, it is up to the caller to test for and handle the case where `days` is very nearly `1.0` and rounds up to 24 hours, by testing for `ihmsf[0] == 24` and setting `ihmsf[0-3]` to zero.

---

iauIr — *initialize r-matrix to identity* — iauIr

CALL :

```
iauIr ( r );
```

ACTION :

Initialize an r-matrix to the identity matrix.

RETURNED :

r	`double[3][3]`	r-matrix

---

iauP2pv *append zero velocity to p-vector* iauP2pv

CALL :

```
iauP2pv ( p, pv );
```

ACTION :

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

GIVEN :

p	`double[3]`	p-vector

RETURNED :

pv	`double[2][3]`	pv-vector

iauP2s — *p-vector to spherical* — iauP2s

CALL :

```
iauP2s ( p, &theta, &phi, &r );
```

ACTION :

P-vector to spherical polar coordinates.

GIVEN :

p	`double[3]`	p-vector

RETURNED :

theta	`double`	longitude angle (radians)
phi	`double`	latitude angle (radians)
r	`double`	radial distance

NOTES :

1. If **p** is null, zero `theta`, `phi` and `r` are returned.
2. At either pole, zero `theta` is returned.

iauPap *position-angle from p-vectors* iauPap

CALL :

```
d = iauPap ( a, b );
```

ACTION :

Position-angle from two p-vectors.

GIVEN :

a	`double[3]`	direction of reference point
b	`double[3]`	direction of point whose PA is required

RETURNED (function value) :

`double`	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.

---

iauPas *position-angle from spherical coordinates* iauPas

CALL :

```
d = iauPas ( al, ap, bl, bp );
```

ACTION :

Position-angle from spherical coordinates.

GIVEN :

al	`double`	longitude of point A (*e.g.* RA) in radians
ap	`double`	latitude of point A (*e.g.* Dec) in radians
bl	`double`	longitude of point B
bp	`double`	latitude of point B

RETURNED (function value) :

`double`	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.

---

iauPdp *dot product of two p-vectors* iauPdp

CALL :

```
d = iauPdp ( a, b );
```

ACTION :

p-vector inner (≡ scalar ≡ dot) product.

GIVEN :

a	`double[3]`	first p-vector
b	`double[3]`	second p-vector

RETURNED (function value) :

`double`	scalar product **a**.**b**

---

iauPm *modulus of p-vector* iauPm

CALL :

```
d = iauPm ( p );
```

ACTION :

Modulus of p-vector.

GIVEN :

p	`double[3]`	p-vector

RETURNED (function value) :

	`double`	modulus $\|\mathbf{p}\|$

iauPmp *p-vector minus p-vector* iauPmp

CALL :

```
iauPmp ( a, b, amb );
```

ACTION :

P-vector subtraction.

GIVEN :

a	`double[3]`	first p-vector
b	`double[3]`	second p-vector

RETURNED :

amb	`double[3]`	$\mathbf{a} - \mathbf{b}$

NOTE :

It is permissible to re-use the same array for any of the arguments.

---

iauPn *normalize p-vector returning modulus* iauPn

CALL :

```
iauPn ( p, &r, u );
```

ACTION :

Convert a p-vector into modulus and unit vector.

GIVEN :

p	`double[3]`	p-vector

RETURNED :

r	`double`	modulus $\|\mathbf{p}\|$
u	`double[3]`	unit vector $\hat{\mathbf{p}}$

NOTE :

If **p** is null, the result is null. Otherwise the result is a unit vector.

iauPpp *p-vector plus p-vector* iauPpp

CALL :

```
iauPpp ( a, b, apb );
```

ACTION :

P-vector addition.

GIVEN :

a	`double[3]`	first p-vector
b	`double[3]`	second p-vector

RETURNED :

apb	`double[3]`	$\mathbf{a} + \mathbf{b}$

NOTE :

It is permissible to re-use the same array for any of the arguments.

iauPpsp — *p-vector plus scaled p-vector* — iauPpsp

CALL :

```
iauPpsp ( a, s, b, apsb );
```

ACTION :

P-vector plus scaled p-vector.

GIVEN :

a	`double[3]`	first p-vector
s	`double`	scalar (multiplier for **b**)
b	`double[3]`	second p-vector

RETURNED :

apsb	`double[3]`	$\mathbf{a} + s \times \mathbf{b}$

NOTE :

It is permissible for any of *a*, *b* and *apsb* to be the same array.

iauPv2p *discard velocity component of pv-vector* iauPv2p

CALL :

```
iauPv2p ( pv, p );
```

ACTION :

Discard velocity component of a pv-vector.

GIVEN :

pv	`double[2][3]`	pv-vector

RETURNED :

p	`double[3]`	p-vector

iauPv2s *pv-vector to spherical* iauPv2s

CALL :

```
iauPv2s ( pv, &theta, &phi, &r, &td, &pd, &rd );
```

ACTION :

Convert position/velocity from Cartesian to spherical coordinates.

GIVEN :

pv	`double[2][3]`	pv-vector

RETURNED :

theta	`double`	longitude angle (radians)
phi	`double`	latitude angle (radians)
r	`double`	radial distance
td	`double`	rate of change of `theta`
pd	`double`	rate of change of `phi`
rd	`double`	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.

iauPvdpv *dot product of two pv-vectors* iauPvdpv

CALL :

```
iauPvdpv ( a, b, adb );
```

ACTION :

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

GIVEN :

a	`double[2][3]`	first pv-vector
b	`double[2][3]`	second pv-vector

RETURNED :

adb	`double[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.

iauPvm *modulus of pv-vector* iauPvm

CALL :

```
iauPvm ( pv, &r, &s );
```

ACTION :

Modulus of pv-vector.

GIVEN :

pv	`double[2][3]`	pv-vector

RETURNED :

r	`double`	modulus of position component
s	`double`	modulus of velocity component

iauPvmpv *pv-vector minus pv-vector* iauPvmpv

CALL :

```
iauPvmpv ( a, b, amb );
```

ACTION :

Subtract one pv-vector from another.

GIVEN :

a	`double[2][3]`	first pv-vector
b	`double[2][3]`	second pv-vector

RETURNED :

amb	`double[2][3]`	$a - b$

NOTE :

It is permissible to re-use the same array for any of the arguments.

---

iauPvppv *pv-vector plus pv-vector* iauPvppv

CALL :

```
iauPvppv ( a, b, apb );
```

ACTION :

Add one pv-vector to another.

GIVEN :

a	`double[2][3]`	first pv-vector
b	`double[2][3]`	second pv-vector

RETURNED :

apb	`double[2][3]`	$a + b$

NOTE :

It is permissible to re-use the same array for any of the arguments.

iauPvu *update pv-vector* iauPvu

CALL :

```
iauPvu ( dt, pv, upv );
```

ACTION :

Update a pv-vector.

GIVEN :

dt	`double`	time interval
pv	`double[2][3]`	pv-vector

RETURNED :

upv	`double[2][3]`	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.
3. It is permissible for `pv` and `upv` to be the same array.

---

iauPvup *update pv-vector discarding velocity* iauPvup

CALL :

```
iauPvup ( dt, pv, p );
```

ACTION :

Update a pv-vector, discarding the velocity component.

GIVEN :

dt	`double`	time interval
pv	`double[2][3]`	pv-vector

RETURNED :

p	`double[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.

iauPvxpv *cross product of two pv-vectors* iauPvxpv

CALL :

```
iauPvxpv ( a, b, axb );
```

ACTION :

Outer (≡ vector ≡ cross) product of two pv-vectors.

GIVEN :

a	`double[2][3]`	first pv-vector
b	`double[2][3]`	second pv-vector

RETURNED :

axb	`double[2][3]`	$a \wedge b$

NOTES :

1. 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.

2. It is permissible to re-use the same array for any of the arguments.

---

iauPxp *cross product of two p-vectors* **iauPxp**

CALL :

```
iauPxp ( a, b, axb );
```

ACTION :

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

GIVEN :

a	`double[3]`	first p-vector
b	`double[3]`	second p-vector

RETURNED :

axb	`double[3]`	$\mathbf{a} \wedge \mathbf{b}$

NOTE :

It is permissible to re-use the same array for any of the arguments.

iauRm2v *r-matrix to r-vector* iauRm2v

CALL :

```
iauRm2v ( r, w );
```

ACTION :

Express an r-matrix as an r-vector.

GIVEN :

r	`double[3][3]`	rotation matrix

RETURNED :

w	`double[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 function 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 function `iauPn`.)

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.

iauRv2m *r-vector to r-matrix* iauRv2m

CALL :

```
iauRv2m ( w, r );
```

ACTION :

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

GIVEN :

w	`double[3]`	rotation vector (Note 1)

RETURNED :

r	`double[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.

---

iauRx — *rotate r-matrix about x axis* — iauRx

CALL :

```
iauRx ( phi, r );
```

ACTION :

Rotate an r-matrix about the x-axis.

GIVEN :

phi	`double`	angle ϕ (radians)

GIVEN and **RETURNED** :

r	`double[3][3]`	r-matrix, rotated

NOTES :

1. Calling this function 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:

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

iauRxp — *product of r-matrix and p-vector* — iauRxp

CALL :

```
iauRxp ( r, p, rp );
```

ACTION :

Multiply a p-vector by an r-matrix.

GIVEN :

r	`double[3][3]`	r-matrix
p	`double[3]`	p-vector

RETURNED :

rp	`double[3]`	$r * p$

NOTE :

It is permissible for *p* and *rp* to be the same array.

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

CALL :

```
iauRxpv ( r, pv, rpv );
```

ACTION :

Multiply a pv-vector by an r-matrix.

GIVEN :

r	`double[3][3]`	r-matrix
pv	`double[2][3]`	pv-vector

RETURNED :

rpv	`double[2][3]`	$r * pv$

NOTES :

1. 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.

2. It is permissible for pv and rpv to be the same array.

iauRxr — *r-matrix multiply* — iauRxr

CALL :

```
iauRxr ( a, b, atb );
```

ACTION :

Multiply two r-matrices.

GIVEN :

a	`double[3][3]`	first r-matrix
b	`double[3][3]`	second r-matrix

RETURNED :

atb	`double[3][3]`	$a * b$

NOTE :

It is permissible to re-use the same array for any of the arguments.

iauRy — *rotate r-matrix about y axis* — iauRy

CALL :

```
iauRy ( theta, r );
```

ACTION :

Rotate an r-matrix about the y-axis.

GIVEN :

theta	`double`	angle θ (radians)

GIVEN and **RETURNED** :

r	`double[3][3]`	r-matrix, rotated

NOTES :

1. Calling this function 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:

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

---

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

CALL :

```
iauRz ( psi, r );
```

ACTION :

Rotate an r-matrix about the z-axis.

GIVEN :

psi	`d`	angle ψ (radians)

GIVEN and **RETURNED** :

r	`double[3][3]`	r-matrix, rotated

NOTES :

1. Calling this function 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:

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

iauS2c *spherical to unit vector* iauS2c

CALL :

```
iauS2c ( theta, phi, c );
```

ACTION :

Convert spherical coordinates to Cartesian.

GIVEN :

theta	`double`	longitude angle (radians)
phi	`double`	latitude angle (radians)

RETURNED :

c	`double[3]`	direction cosines

iauS2p — *spherical to p-vector* — iauS2p

CALL :

```
iauS2p ( theta, phi, r, p );
```

ACTION :

Convert spherical polar coordinates to p-vector.

GIVEN :

theta	`d`	longitude angle (radians)
phi	`double`	latitude angle (radians)
r	`double`	radial distance

RETURNED :

p	`double[3]`	Cartesian coordinates

iauS2pv *spherical to pv-vector* iauS2pv

CALL :

```
iauS2pv ( theta, phi, r, td, pd, rd, pv );
```

ACTION :

Convert position/velocity from spherical to Cartesian coordinates.

GIVEN :

theta	`double`	longitude angle (radians)
phi	`double`	latitude angle (radians)
r	`double`	radial distance
td	`double`	rate of change of `theta`
pd	`double`	rate of change of `phi`
rd	`double`	rate of change of `r`

RETURNED :

pv	`double[2][3]`	pv-vector

---

iauS2xpv — *multiply pv-vector by two scalars* — iauS2xpv

CALL :

```
iauS2xpv ( s1, s2, pv, spv );
```

ACTION :

Multiply a pv-vector by two scalars.

GIVEN :

s1	`double`	scalar to multiply position component by
s2	`double`	scalar to multiply velocity component by
pv	`double[2][3]`	pv-vector

RETURNED :

spv	`double[2][3]`	pv-vector: p scaled by `s1`, v scaled by `s2`

NOTE :

It is permissible for `pv` and `spv` to be the same array.

iauSepp *angular separation from p-vectors* iauSepp

CALL :

```
d = iauSepp ( a, b );
```

ACTION :

Angular separation between two p-vectors.

GIVEN :

a	`double[3]`	first p-vector (not necessarily unit length)
b	`double[3]`	second p-vector (not necessarily unit length)

RETURNED (function value) :

`double`	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.

---

iauSeps *angular separation from spherical coordinates* iauSeps

CALL :

```
d = iauSeps ( al, ap, bl, bp );
```

ACTION :

Angular separation between two sets of spherical coordinates.

GIVEN :

al	`double`	first longitude (radians)
ap	`double`	first latitude (radians)
bl	`double`	second longitude (radians)
bp	`double`	second latitude (radians)

RETURNED (function value) :

`double`	angular separation (radians)

iauSxp *multiply p-vector by scalar* iauSxp

CALL :

```
iauSxp ( s, p, sp );
```

ACTION :

Multiply a p-vector by a scalar.

GIVEN :

s	`double`	scalar
p	`double[3]`	p-vector

RETURNED :

sp	`double[3]`	$s * p$

NOTE :

It is permissible for *p* and *sp* to be the same array.

---

iauSxpv — *multiply pv-vector by scalar* — iauSxpv

CALL :

```
iauSxpv ( s, pv, spv );
```

ACTION :

Multiply a pv-vector by a scalar.

GIVEN :

s	`double`	scalar
pv	`double[2][3]`	pv-vector

RETURNED :

spv	`double[2][3]`	$s * pv$

NOTE :

It is permissible for `pv` and `spv` to be the same array.

iauTf2a *hours, minutes, seconds to radians* iauTf2a

CALL :

```
j = iauTf2a ( s, ihour, imin, sec, &rad );
```

ACTION :

Convert hours, minutes, seconds to radians.

GIVEN :

s	`c`	sign: `'-'` = negative, otherwise positive
ihour	`int`	hours
imin	`int`	minutes
sec	`double`	seconds

RETURNED :

rad	`double`	angle in radians

RETURNED (function value) :

`int`	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.

iauTf2d — *hours, minutes, seconds to days* — iauTf2d

CALL :

```
j = iauTf2d ( s, ihour, imin, sec, &days );
```

ACTION :

Convert hours, minutes, seconds to days.

GIVEN :

s	`c`	sign: `'-'` = negative, otherwise positive
ihour	`int`	hours
imin	`int`	minutes
sec	`double`	seconds

RETURNED :

days	`double`	interval in days

RETURNED (function value) :

`int`	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.

iauTr — *transpose r-matrix* — iauTr

CALL :

```
iauTr ( r, rt );
```

ACTION :

Transpose an r-matrix.

GIVEN :

r	`double[3][3]`	r-matrix

RETURNED :

rt	`double[3][3]`	transpose

NOTE :

It is permissible for `r` and `rt` to be the same array.

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

CALL :

```
iauTrxp ( r, p, trp );
```

ACTION :

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

GIVEN :

r	`double[3][3]`	r-matrix
p	`double[3]`	p-vector

RETURNED :

trp	`double[3]`	$r^T \times p$

NOTE :

It is permissible for `p` and `trp` to be the same array.

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

CALL :

```
iauTrxpv ( r, pv, trpv );
```

ACTION :

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

GIVEN :

r	`double[3][3]`	r-matrix
pv	`double[2][3]`	pv-vector

RETURNED :

trpv	`double[2][3]`	$r^T \times pv$

NOTES :

1. 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.

2. It is permissible for pv and $trpv$ to be the same array.

---

iauZp *zero p-vector* iauZp

CALL :

```
iauZp ( p );
```

ACTION :

Zero a p-vector.

RETURNED :

p	`double[3]`	zero p-vector

---

iauZpv *zero pv-vector* iauZpv

CALL :

```
iauZpv ( pv );
```

ACTION :

Zero a pv-vector.

RETURNED :

pv	`double[2][3]`	zero pv-vector

---

iauZr — *initialize r-matrix to null* — iauZr

CALL :

```
iauZr ( r );
```

ACTION :

Initialize an r-matrix to the null matrix.

RETURNED :

r	`double[3][3]`	null r-matrix

3.4 Classified list of functions

Operations involving p-vectors and r-matrices

initialize

`iauZp ( p );`
zero p-vector p71

`iauZr ( r );`
initialize r-matrix to null p73

`iauIr ( r );`
initialize r-matrix to identity p29

copy

`iauCp ( p, c );`
copy p-vector p25

`iauCr ( r, c );`
copy r-matrix p27

build rotations

`iauRx ( phi, r );)`
rotate r-matrix about x p52

`iauRy ( theta, r );`
rotate r-matrix about y p56

`iauRz ( psi, r );`
rotate r-matrix about z p57

spherical/Cartesian conversions

`iauS2c ( theta, phi, c );`
spherical to unit vector p58

`iauC2s ( p, &theta, &phi );`
unit vector to spherical p24

`iauS2p ( theta, phi, r, p );`
spherical to p-vector p59

`iauP2s ( p, &theta, &phi, &r );`
p-vector to spherical p31

operations on p-vectors

`iauPpp ( a, b, apb );`
p-vector plus p-vector p38

`iauPmp ( a, b, amb );`
p-vector minus p-vector p36

`iauPpsp ( a, s, b, apsb );`
p-vector plus scaled p-vector p39

`d = iauPdp ( a, b );`
inner (=scalar=dot) product of two p-vectors p34

`iauPxp ( a, b, axb );`
outer (=vector=cross) product of two p-vectors p49

`d = iauPm ( p );`
modulus of p-vector p35

`iauPn ( p, &r, u );`
normalize p-vector returning modulus p37

`iauSxp ( s, p, sp );`
multiply p-vector by scalar p64

operations on r-matrices

`iauRxr ( a, b, atb );`
r-matrix multiply p55

`iauTr ( r, rt );`
transpose r-matrix p68

matrix-vector products

`iauRxp ( r, p, rp );`
product of r-matrix and p-vector p53

`iauTrxp ( r, p, trp );`
product of transpose of r-matrix and p-vector p69

separation and position-angle

`d = iauSepp ( a, b );`
angular separation from p-vectors p62

`d = iauSeps ( al, ap, bl, bp );`
angular separation from spherical coordinates p63

`d = iauPap ( a, b );`
position-angle from p-vectors p32

`d = iauPas ( al, ap, bl, bp );`
position-angle from spherical coordinates p33

rotation vectors

`iauRv2m ( p, r );`
r-vector to r-matrix p51

`iauRm2v ( r, p );`
r-matrix to r-vector p50

Operations involving pv-vectors

initialize

`iauZpv ( pv );`
zero pv-vector p72

copy/extend/extract

`iauCpv ( pv, c );`
copy pv-vector p26

`iauP2pv ( p, pv );`
append zero velocity to p-vector p30

`iauPv2p ( pv, p );`
discard velocity component of pv-vector p40

spherical/Cartesian conversions

`iauS2pv ( theta, phi, r, td, pd, rd, pv );`
spherical to pv-vector p60

`iauPv2s ( pv, &theta, &phi, &r, &td, &pd, &rd );`
pv-vector to spherical p41

operations on pv-vectors

`iauPvppv ( a, b, apb );`
pv-vector plus pv-vector p45

`iauPvmpv ( a, b, amb );`
pv-vector minus pv-vector p44

`iauPvdpv ( a, b, adb );`
inner (=scalar=dot) product of two pv-vectors p42

`iauPvxpv ( a, b, axb );`
outer (=vector=cross) product of two pv-vectors p48

`iauPvm ( pv, &r, &s );`
modulus of pv-vector p43

`iauSxpv ( s, pv, spv );`
multiply pv-vector by scalar p65

`iauS2xpv ( s1, s2, pv );`
multiply pv-vector by two scalars p61

`iauPvu ( dt, pv, upv );`
update pv-vector p46

`iauPvup ( dt, pv, p );`
update pv-vector discarding velocity p47

matrix-vector products

`iauRxpv ( r, pv, rpv );`
product of r-matrix and pv-vector p54

`iauTrxpv ( r, pv, trpv );`
product of transpose of r-matrix and pv-vector p70

Operations on angles

wrap

`d = iauAnp ( a );`
normalize radians to range 0 to 2π p22

`d = iauAnpm ( a );`
normalize radians to range $-\pi$ to $+\pi$ p23

to sexagesimal

`iauA2af ( ndp, angle, &sign, idmsf );`
decompose radians into degrees, arcminutes, arcseconds p19

`iauA2tf ( ndp, angle, &sign, ihmsf );`
decompose radians into hours, minutes, seconds p20

`iauD2tf ( ndp, days, &sign, ihmsf );`
decompose days into hours, minutes, seconds p28

from sexagesimal

`i = iauAf2a ( s, ideg, iamin, asec, &rad );`
degrees, arcminutes, arcseconds to radians p21

`i = iauTf2a ( s, ihour, imin, sec, &rad );`
hours, minutes, seconds to radians p66

`i = iauTf2d ( s, ihour, imin, sec, &days );`
hours, minutes, seconds to days p67

3.5 Calls: quick reference


```
    iauA2af ( ndp, angle, &sign, idmsf );              p19
    iauA2tf ( ndp, angle, &sign, ihmsf );              p20
i = iauAf2a ( s, ideg, iamin, asec, &rad );            p21
d = iauAnp ( a );                                      p22
d = iauAnpm ( a );                                     p23
    iauC2s ( p, &theta, &phi );                        p24
    iauCp ( p, c );                                    p25
    iauCpv ( pv, c );                                  p26
    iauCr ( r, c );                                    p27
    iauD2tf ( ndp, days, &sign, ihmsf );               p28
    iauIr ( r );                                       p29
    iauP2pv ( p, pv );                                 p30
    iauP2s ( p, &theta, &phi, &r );                    p31
d = iauPap ( a, b );                                   p32
d = iauPas ( al, ap, bl, bp );                         p33
d = iauPdp ( a, b );                                   p34
d = iauPm ( p );                                       p35
    iauPmp ( a, b, amb );                              p36
    iauPn ( p, &r, u );                                p37
    iauPpp ( a, b, apb );                              p38
    iauPpsp ( a, s, b, apsb );                         p39
    iauPv2p ( pv, p );                                 p40
    iauPv2s ( pv, &theta, &phi, &r, &td, &pd, &rd );   p41
    iauPvdpv ( a, b, adb );                            p42
    iauPvm ( pv, &r, &s );                             p43
    iauPvmpv ( a, b, amb );                            p44
    iauPvppv ( a, b, apb );                            p45
    iauPvu ( dt, pv, upv );                            p46
    iauPvup ( dt, pv, p );                             p47
    iauPvxpv ( a, b, axb );                            p48
    iauPxp ( a, b, axb );                              p49
    iauRm2v ( r, p );                                  p50
    iauRv2m ( p, r );                                  p51
    iauRx ( phi, r );                                  p52
    iauRxp ( r, p, rp );                               p53
    iauRxpv ( r, pv, rpv );                            p54
    iauRxr ( a, b, atb );                              p55
    iauRy ( theta, r );                                p56
    iauRz ( psi, r );                                  p57
    iauS2c ( theta, phi, c );                          p58
    iauS2p ( theta, phi, r, p );                       p59
    iauS2pv ( theta, phi, r, td, pd, rd, pv );         p60
    iauS2xpv ( s1, s2, pv );                           p61
d = iauSepp ( a, b );                                  p62
```

```
d = iauSeps ( al, ap, bl, bp );                    p63
    iauSxp ( s, p, sp );                           p64
    iauSxpv ( s, pv, spv );                        p65
i = iauTf2a ( s, ihour, imin, sec, &rad );         p66
i = iauTf2d ( s, ihour, imin, sec, &days );        p67
    iauTr ( r, rt );                               p68
    iauTrxp ( r, p, trp );                         p69
    iauTrxpv ( r, pv, trpv );                      p70
    iauZp ( p );                                   p71
    iauZpv ( pv );                                 p72
    iauZr ( r );                                   p73
```