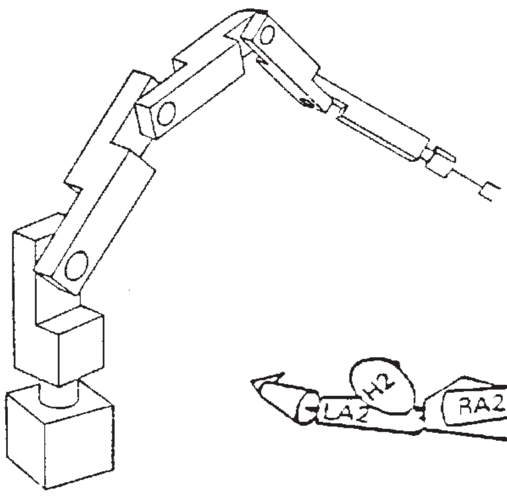
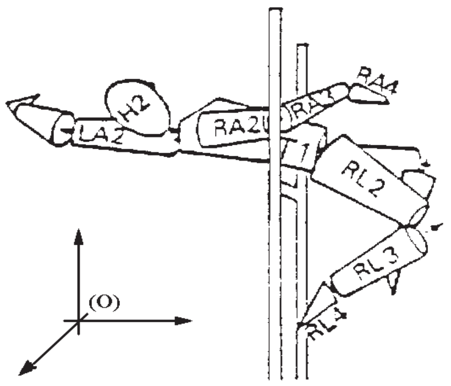
SpaceLib© in C

Version 2.2a - March 2006

A software library for
the kinematic and dynamic analysis
of systems of rigid bodies.
Includes general functions for vectors, matrices,
kinematics, dynamics, Euler angles and linear systems

USER'S MANUAL





By G. Legnani, R. Adamini and B. Zappa

Università di Brescia - Dip. Ing. Meccanica
Via Branze 38, 25123 BRESCIA, Italy
Tel. +39/030 3715425 Fax +39/030 3702448

e-mail: giovanni.legnani@ing.unibs.it
http://bsing.ing.unibs.it/~glegnani
http://robotics.ing.unibs.it

With the cooperation of D.Amadori, P.Ghislotti, G.Pugliese, R.Faglia and D.Manara

Typeset in LaTeX

SpaceLib© and its authors.

The first functions contained in SpaceLib© were written in C language by myself in 1988 when a friend of mine working with the *European Space Agency* asked for help. I wrote the program described in § 7.5 of this manual. I realized that a few friends needed software to deal with 3D rototranslations and I started writing a "private" version of SpaceLib©.

After a while other people asked for help and in 1990 I wrote the first "public" version of SpaceLib© with the support of *R. Faglia*. The mathematical bases of SpaceLib© grown and new functions were realized to deal with velocities, accelerations, forces, torques, momentum and angular momentum.

A second public version of SpaceLib© was then realized in 1993 with the help of *R. Adamini* and several dozens of copies have been distributed through the world. People have been using it both for *Robotic* and *Biomechanics* applications. I have used it for my research and lectures and the students have shown a great interest in it.

In 1997, a new version of the library in C language has been realized under my supervision by *D. Amadori*, *P. Ghislotti* and *G. Pugliese*. I made the final refinements with a strong support by *B. Zappa*; *R. Adamini* gave a good theoretical and technical support. This version contains additional functions and an extended documentation.

Many people asked for a new version of SpaceLib© in MATLAB ©[1]. The bases of the MATLAB © version of SpaceLib© have been realized by *C. Moiola* under my supervision. I made a final "strong" correction and I also performed some patches in 2001, 2003 and 2004. In this occasion a new version of the manual have been realized with the help of *D. Tosi* and *M. Camposaragna*.

Time passed and the need for a new version of SpaceLib© which made possible the writing of symbolic equations grown. This stimulated the realization of the SpaceLib© in Maple 9 ©[2]. The basis of this version were put by *F. Bignamini* and *N. Serana* under my supervision. The final version of the library were made by *D. Manara* under my supervision with the cooperation of *A. Rodenghi*.

Between the end of 2004 and the beginning of 2005, *D. Manara* put a great effort in revising all the manuals of three SpaceLib© versions (C, MATLAB ©, Maple 9 ©). This was an occasion to perform small revisions to the three versions. A great effort was put in maintaining aligned the three releases.

This version of SpaceLib© will be probably upgraded in future.

We look forward for comments, suggestions, bugs report and copies of papers related with the use of SpaceLib©. Send them to *G. Legnani* (address on cover page).

Brescia, Italy; March 2006

Giovanni Legnani

[1] MATLAB © is a registered trademark of MathWorks (`http://www.mathworks.com`) inc.

[2] Maple 9 © is a registered trademark of Maplesoft (`http://www.maplesoft.com`), a division of Waterloo Maple inc.

Contents

Chapter 1

Introduction

1.1 What is SpaceLib©

SpaceLib© is a software library useful for the realization of programs for the kinematic and dynamic analysis of systems of rigid bodies. This library is currently used in *Robotics* and *Biomechanics*. It has been developed at the Mechanical Engineering Department of the University of Brescia.

The library is intended as an aid in writing programs for the analysis of mechanical systems following a particular methodology based on 4×4 matrices show in [2], [3] and [4][1]. This approach can be considered a powerful generalization of the Transformation Matrix Approach proposed by *Denavit* and *Hartenberg* [1].

The main feature of this methodology is that it allows the development of the analysis of systems of rigid bodies in a systematic way simplifying the symbolic manipulation of equations as well as the realization of efficient numerical programs.

Three versions of the library are presently available, two for numerical simulations in C and MATLAB © languages, and one version for the symbolic computation in Maple 9©. The MATLAB © version is useful for a fast development of numeric programs. The C version is preferable to obtain fast high-efficient numeric simulations. The Maple 9 © version is useful when symbolic manipulation is essential, however it also make possible the development of numeric programs.

Particular effort has been posed in order to keep "aligned" the different versions. Functions with the same name in the three versions produces essentially the same results. However intrinsic differences between the languages result in few difformities between the different implementations (see § A, page 145).

All the distributions contain the software source code, and if one likes, he can analyzes it to better understand its use.

1.2 About this manual

This USER'S MANUAL has been written assuming that the reader knows both the C language and the theory on which SpaceLib© is based. The latter subject is widely described in the *references* (page 143). Since SpaceLib© has been developed by successive steps, some details contained in the references can differ from pieces of information here contained. In this case, please refer to this manual.

This manual contains:

- introduction;
- general information on the use of SpaceLib©;
- a commented directory of the library;
- sample programs;
- a Reference list of papers which describes the mathematical bases of SpaceLib©.

[1]copy of [3] and [4] is also included into the distribution file

1.3 Technical information

SpaceLib© has been initially written in ANSI C language. It has been developed under MS-DOS© operative system using the Microsoft© C compiler v 5.1. It has been also tested with Microsoft© Quick-C v 2.5, Microsoft© Visual C++ v 4.2 and Borland© Turbo C++ v 3.0 compilers. The library has also been successfully used on VAX© computers under VMS© operative system and many Unix distributions.

Warning: A common source of mistakes using SpaceLib© is the possibility to define the type `real` as synonymous with `float` or with `double`. This operation is performed modifying the first line of `spacelib.h`. If this equivalence is modified, it is generally necessary to correct also some statements of the user program (e.g. the strings defining the format of `scanf()` or `fscanf()` functions reading `real` data). To help writing more "safe" programs the `typedef` statement has now been posed inside an `#ifdef` structure which allows the use of conditional compilations. As default `real` is assumed equivalent to `double`; to assume `real` equivalent to `float` it is necessary to compile SpaceLib© defining the symbol `float` (see § 2.4.1).

1.4 Authors' notes and disclaimer warranties

The authors know that the present version of SpaceLib© should be possibly improved in the future. The code or the documentation could contain errors or the documentation could have lacks in some parts. The library and the related information is provided "as is" without warranty of any kind. The authors disclaim all warranties, either express or implied, including the warranties of merchantability and fitness for a particular purpose. In no event shall the authors or their institution or its suppliers be liable for any damages whatsoever including direct, indirect, incidental, consequential, loss of business profits or special damages, even if the authors or their suppliers have been advised of the possibility of such damages.
The authors stimulate any suggestion and error reporting by the users.

Chapter 2

Writing programs with `SpaceLib`©

2.1 General information - Read me first

`SpaceLib`© consists of six source files and two header files (`spacelib.c`, `spaceli3.c`, `spaceli4.c`, `spaceli5.c`, `linear.c`, `linear2.c`, `spacelib.h` and `linear.h`). `spacelib.c`, `spaceli3.c`, `spaceli4.c` and `spaceli5.c` include the main routines for the direct kinematic and inverse dynamic analysis while `linear.c` and `linear2.c` contains the routines for the solution of a linear system of equation and for the direct dynamic problem.

The word *routine* is here used to indicate either a *function* or a *macro.* The functions are contained in the source files, while the macros are defined in the headers (see § 6). Macros are utilized in order to call general-purpose functions with standard values of some parameters. In fact `SpaceLib`© was designed in order to handle both 3×3 rotation matrices and 4×4 transformation matrices. The macros often perform also some recast operations. For instance the function `molt(M A,M B,M C,i,j,k)` performs the matrix operation $C = A \cdot B$ where the dimension of the three matrices are `A[i,j]`, `B[j,k]`, `C[i,k]`. But two macros `molt3(A,B,C)` and `molt4(A,B,C)` are supplied to multiply 3×3 and 4×4 matrices.

In order to improve the performances (accuracy and efficiency) of the functions, their algorithms have been designed to take advantage of special properties of the matrices involved. For this reason many functions are "`specialized`". For instance function `invers` inverts very quickly a matrix, but it can be utilized only for 4×4 transformation matrices. Generally, in the routines calling list the input parameters precede the output ones. Exceptions are the dimensions of the matrices which are generally placed at the end.

Many types had been defined in the header files in order to simplify the declaration and the definition of matrices and other variables. For instance, the statement

```
MAT4 mat1, mat2;
POINT p1;
```

defines two 4×4 matrices and a point (points are represented by an array of 4 elements containing their homogeneous coordinates)

A special `typedef` statement has been included in `spacelib.h` in order to define a user type `real` equivalent to the type `double`. Users should use `real` instead of `double`. This trick allows the users to change their programs from double precision to single precision simply redefining `real` as equivalent to `float`. To modify it, follow instructions contained in § 2.4.1. In this case all the components of the library <u>must</u> be recompiled. See also "Warning" in § 1.3.

Both the source files and the header files contain useful comments about the routines, the types and the macro definitions. Before using `SpaceLib`©, users should have a look, at least, at the header files.

All the previous subjects are detailed in the following paragraphs.

2.1.1 Include

Any file containing calls to `SpaceLib`© functions must begin with the following statements

```
#include <stdio.h>
#include <float.h>
#include <math.h>
```

```
#include "spacelib.h"
```

If the file contains also calls to function `dyn_eq`, `linear` or other functions contained in `linear.c` or in `linear2.c` the following line must be also added:

```
#include "linear.h"
```

The header files must be placed in a proper directory so that the compiler can find them.

2.1.2 Compiling and linking

To obtain an executable file, any program using SpaceLib© routines must be compiled and linked with the required SpaceLib© components. Generally all the programs require `spacelib.c`; some would also require other components like `spaceli2.c`, `spaceli4.c`, `spaceli5.c`, `linear.c` and/or `linear2.c`. If you use a command-line environment you must individually compile each components of the user program and of SpaceLib© and then you must link all together. Details depend on the operative systems and on the compiler utilized. For example, if and old `microsoft` compiler is used, the compiling and linking session are those reported below. The examples concern the sample programs `ROB-MAT.C` and `TEXT.C` described in sections § 7.1 and § 7.2 of this manual. `ROB-MAT.C` uses some of the subroutines of `spacelib.c`. The compiling and linking phase is composed by the three following statements :

Command line	*result*
`c:\ >cl -c ROB_MAT.C`	`ROB_MAT.OBJ`
`c:\ >cl -c spacelib.c`	`spacelib.obj`
`c:\ >link ROB_MAT.OBJ + spacelib.obj`	`ROB_MAT.EXE`

`TEST.C` uses also some of the subroutines of `linear.c`. The compiling and linking phase is composed by the four following statements:

Command line	*result*
`c:\ >cl -c TEST.C`	`TEST.OBJ`
`c:\ >cl -c spacelib.c`	`spacelib.obj`
`c:\ >cl -c linear.c`	`linear.obj`
`c:\ >link ROB_MAT.OBJ + spacelib.obj + linear.obj`	`TEST.EXE`

Warning: The way to compile the programs differs from compiler to compiler. For example in an integrated development environment (IDE) or a visual environment, such as `Visual c++`, `lccwin` (or other), the user must create a configuration file, a project file or a make file (file `*.mak`) which contains the names of the source files to be compiled and linked. For further details the user should examine the specific compiler manual. Here is an example of the make file created with an old compiler `Microsoft©` `Quick-C v 2.5` necessary to compile the source code of the example concerning a satellite described in § 7.5. This example uses the files: `sat.c`, `spacelib.c`, `spaceli3.c`.

Sat.mak example of make file for Microsoft© `C`

```
PROJ    =SAT
DEBUG   =1
CC  =qcl
CFLAGS_G     = /AM /W1 /Ze /DFLOAT
CFLAGS_D     = /Zi /Zr /Gi$(PROJ).mdt /Od
CFLAGS_R     = /O /Ot /DNDEBUG
CFLAGS       =$(CFLAGS_G) $(CFLAGS_D)
LFLAGS_G     = /CP:0xfff/NOI /SE:0x80 /ST:0x2710
LFLAGS_D     = /CO /INCR
LFLAGS_R     =
LFLAGS       =$(LFLAGS_G) $(LFLAGS_D)
RUNFLAGS     =
OBJS_EXT =
LIBS_EXT =
.asm.obj: ; $(AS) $(AFLAGS) -c $*.as
all:    $(PROJ).EXE
```

```
spacelib.obj:   spacelib.c $(H)
spaceli4.obj:   spaceli4.c $(H)
spaceli3.obj:   spaceli3.c $(H)
sat.obj:    sat.c $(H)
$(PROJ).EXE:    spacelib.obj spaceli4.obj
                spaceli3.obj sat.obj
$(OBJS_EXT)
echo >NUL @<<$(PROJ).crf
spacelib.obj + spaceli4.obj + spaceli3.obj +
sat.obj +
$(OBJS_EXT)$(PROJ).EXE $(LIBS_EXT); <<
ilink -a -e "qlink $(LFLAGS) @$(PROJ).crf"
$(PROJ)
run: $(PROJ).EXE
$(PROJ) $(RUNFLAGS)
```

2.2 Notation

In this section are briefly described the notation used in the `SpaceLib`©. More information can be found in [2], [3] and [4].

2.2.1 Subscript conventions

The relative motions between bodies are represented by matrices which usually appear with some subscripts:

$$M_{i,j} \qquad W_{i,j(k)} \qquad H_{i,j(k)} \qquad L_{i,j(k)}$$

Subscripts i and j specifies the bodies involved, the subscript k, which is in round brackets, denotes the frame onto which the quantities are projected. For instance the velocity of the body (5) with respect to body (3) projected on frame (2) is:

$$W_{3,5(2)}$$

In special cases when subscripts assume "standard" or "obvious values" some of them can be omitted to simplify the notation. This happens for example where the meaning of each matrix is presented. For the same reason the third subscript k is often omitted when $k = i$. The dynamics quantities J, Γ and Φ require just two subscripts:

$$J_{i(k)} \qquad \Gamma_{i(k)} \qquad \Phi_{i(k)}$$

The subscript i denotes the body involved, and (k) has the previous meaning (frame on which the quantities are projected). Frame (0) is the absolute reference frame; in dynamics it is assumed to be also the inertial frame.

2.2.2 Naming convection for parameters

In describing `SpaceLib`© functions and macros the authors will make use of *matrices*, *vectors*, *axes*, *frames*, *planes*, *line*, *points* and *constants*. *Matrices* and *points* are generally denoted by upper case characters while *vectors*, *axes* and scalar parameters (i.e. "phi", "alpha", "dim") are denoted by lowercase letters. However there are a few exceptions.
In the user manual the names of the parameters are generally given according to the convention described in table 2.1.

NOTE (1) Refereing to table 2.1, the '`x`' character following a variable name is generally substituted by a digit. It is useful in order to specify two or more variables of the same type in a function prototype (i.e. `m1` and `m2` or `R1` and `R2`).

NOTE (2) The `M` character mustn't be used to indicate a position matrix because it represents a recast operator (see § 2.6).

2.2.3 Units

Although sometimes different set of congruent units can be used, users are suggested to utilize always the *International Units System* (see table 2.2). Angles must be expressed in radians.

2.3 Math functions

For the users convenience, the math macros listed in table 2.3 have been defined in the header file `spacelib.h` (see § 6.1). They work with parameters of any type (`float`, `double` or `integer`).

2.4 Variables declarations and types

2.4.1 Data types

For all the floating point operations the user should utilize variables of type `real`. As already specified, this type was defined for the user's convenience and it's equivalent to `double`. The type `real` has been

Special Matrices Names	***Use***
Φ, PHI	Action matrix
G	3×3 upper-left submatrix of H matrix
H, Hx, Wp	Acceleration matrix(1)
Hg	Gravity acceleration matrix
J	Inertia matrix
L, Lx	L matrix(1)
m, mx	Position or transformation matrix (1)
R, Rx	Rotation matrix(1)
W, Wx	Velocity matrix(1)
Special Points names	
O	Frame origin
Axes names	
X, Y, Z, U, a	Rototranslations axis, Axis of rotation (a=X, Y or Z)
Scalar Parameters names	
i, j, k	Parameters related to operation dealing with the x, y, z axes
q, qx, qp, qpp	joint variables, first derivative, second derivative(1)
Generic elements names	
A, B, C, Ax, Mx, mx	Matrix name(1)
Pl	plane name
v, vx	Vector name(1)
L, Lx	line name(1)
dim, n	Matrix dimension
P, Px	Point name(1)

Table 2.1: Naming convention for SpaceLib© parameters

SpaceLib© units Table		
Length	m	meter
Time	s	second
Force	N	newton
Torque	N m	newton · meter
Mass	kg	kilogram
Angle	rad	radian

Table 2.2: International Units System

used in order to define all the further floating-point types. The usage of `float` or `double` should be avoided when not strictly required.

The type `real` can be easily redefined as equal to `float` whenever a minor precision is required. To change this definition it is necessary to add the following line at the very beginning of `spacelib.h` (see § 6.1)

```
#define FLOAT
```

Alternatively it is possible to recompile the library with the following switch:

`/DFLOAT` for Microsoft© `Visual C++`

`-DFLOAT` for Borland© `Turbo C++`

macro name	returning value
`aabs(x)`	absolute value of x
`max(a,b)`	the maximum between a and b
`min(a,b)`	the minimum between a and b
`sign(x)`	the sign of x which is defined as $\begin{cases} -1 & & x<0 \\ 0 & if & x=0 \\ 1 & & x>0 \end{cases}$
`rad(x)`	degrees to radians conversion
`deg(x)`	radians to degrees conversion

Table 2.3: Macro for math function

Users using IDE visual compiler tools must see their documentation.

The further following data type have been defined in the header file `spacelib.h` for the user's convenience (see § 6.1):

`MAT3` declaration of 3×3 matrices (matrix of 3×3 real elements).

`MAT4` declaration of 4×4 matrices (matrix of 4×4 real elements).

`POINT` declaration of a point (array of 4 real values containing the homogeneous coordinates of a point).

`LINE` declaration of a line (struct containing a point on the line and a unit vector which defines the direction of the line itself. See also § 2.4.2).

`PLANE` declaration of a plane (array of 4 real values containing the three components of the unit vector orthogonal to the plane and the distance of the plane from the origin of reference frame. See also § 2.4.2).

`AXIS` declaration of an axis (array of 3 real containing the three components of a unit vector).

`VECTOR` equivalent to `AXIS` but used for vectors.

The following types have also been defined in the header file `spacelib.h` (see § 6.1) for internal use and they will be probably used rarely by the user.

`MAT` Pointer to real (used in the prototype of functions operating on matrices of non prefixed dimensions. See § 2.6).

`MAT3P` Pointer to an array of 3 real elements. Used to point a 3×3 matrix.

`MAT4P` Pointer to an array of 4 real elements. Used to point a 4×4 matrix.

`LINEP` Pointer to a line type structure. Used to point a line struct.

2.4.2 Geometric elements

As better described in the references, in `SpaceLib`© some geometric entities are used: *points*, *lines*, *vector*, *planes* and *frames*. A *point* is represented by its homogeneous coordinates x, y, z and u:

$$P = [x,\ y,\ z,\ u]^t$$

In `C` this is obtained by storing these coordinates into one array with four elements:

```
P={x, y, z, u};
```

A *line* is represented by one point and by its unit vector:

$$\begin{cases} x = x_p + \alpha \cdot t \\ y = y_p + \beta \cdot t \\ z = z_p + \gamma \cdot t \end{cases} \tag{2.1}$$

where $P = [x_p,\ y_p,\ z_p]^t$ is a point that lies on the line. The vector $[\alpha,\ \beta,\ \gamma]^t$ contains the director cosines which express the direction of the line in a reference frame ($\alpha^2 + \beta^2 + \gamma^2 = 1$); t is the abscissa. In C this is obtained by the definition of a new type in the file spacelib.h (see § 6.1)

```
typedef struct {
          POINT P;
          VECTOR dir;
                } LINE, * LINEP;
```

The type LINEP is used to point a line structure. It allows to get back values from functions dealing with line structs.
A *vector* is represented by its components u_x, u_y and u_z:

$$V = [u_x,\ u_y,\ u_z]^t$$

In C this is obtained by storing these coordinates into one array with three elements:

```
V = {ux, uy, uz}
```

A *plane* is implicity defined by the following equation:

$$a \cdot x + b \cdot y + c \cdot z + d = 0 \tag{2.2}$$

where a, b, c are the components in a reference frame of the unit vector orthogonal to the plane itself ($a^2 + b^2 + c^2 = 1$). The fourth element d expresses the distance with sign of the origin of the reference frame from the plane. In C language the plane type is simply defined as a four real element array:

```
pl={ a, b, c, d }
```

These types and some functions dealing with them are applied in example 3.28 and in example 3.29. A *frame* is represented by a 4×4 matrix containing the homogeneous coordinates of the frame axes and of the origin of the frame:

$$\left[\begin{array}{ccc|c} X_x & Y_x & Z_x & x \\ X_y & Y_y & Z_y & y \\ X_z & Y_z & Z_z & z \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

See also table 2.4.

SpaceLib©	C typedef
point	real POINT[4];
line	struct {POINT P; VECTOR dir;} LINE, * LINEP;
vector	real VECTOR[3];
plane	real PLANE[4];
frame	real MAT4[4][4];

Table 2.4: Correspondence between the SpaceLib© and the C Data Type

2.4.3 Useful constants

The following constants has been defined in the header file spacelib.h (see § 6.1)

OK = 1
NOTOK = 0
	Values returned by some SpaceLib© functions in order to specify the success or the failure of their operations. If a function returns NOTOK, it means that it could not perform the requested operation. In general, it happens if the function was called with non valid values for the input parameters.

SYMM = 1
SKEW =-1
	Utilized by some functions in order to specify if a matrix is symmetric or skew-symmetric.

`Rev` = 0 `Pri` = 1	Utilized to denote revolute or prismatic (sliding) pairs.
`Tor` = 0 `For` = 1	Utilized to denote torques or forces.
`Row` = 0 `Col` = 0	Utilized to denote rows and columns.

X=0
Y=1
Z=2
U=3
	Utilized to denote the four homogeneous coordinates of a point or the three components of a vector or axis. To remember the differences in the constants definition to identify the Cartesian axes between `SpaceLib`© in `C`, `MATLAB`© and `Maple 9`©, refer to the table 2.5.

	`C`	`MATLAB`©	`Maple 9`©
X	0	1	1
Y	1	2	2
Z	2	3	3
U	3	4	4

Table 2.5: Rotation axes naming convention

The following constants have been also defined in order to initialize, when applicable, matrices, points and vectors. They can be generally used only to initialize global or static arrays or matrices.

`Xaxis` = $\{1., 0., 0.\}$ `Yaxis` = $\{0., 1., 0.\}$ `Zaxis` = $\{0., 0., 1.\}$	Applicable to variables of the type `AXIS` in order to set them equal to an axis coordinate.
`Xaxis_n` = $\{-1., 0., 0.\}$ `Yaxis_n` = $\{0., -1., 0.\}$ `Zaxis_n` = $\{0., 0., -1.\}$	Applicable to variables of the type `AXIS` in order to set them equal to a negative axis coordinate.
`ORIGIN =` $\{0., 0., 0., 1.\}$	Applicable to a variable of the type `POINT` in order to set it equal to the origin of a frame.

`NULL3`
`UNIT3`
`NULL4`
`UNIT4`
	Applicable to 3×3 or 4×4 matrices in order to set them equal to the null or the identity matrix.

`PIG_2 = 1.57...`
`PIG = 3.14...`
`PIG2 = 6.28...`
	Useful trigonometric constants $\pi/2$, π, 2π.

2.5 Arrays of scalar data and of matrices.

Array of scalar data are defined with the standard `C` syntax. For example if `a` is an array of `n` real elements, for `n`=25 the declaration reads

```
define n 25
real a[n];
```

and the individual elements of a are addressed as `a[0]`, `a[1]`, `a[2]`, ... `a[24]`. In fact in C the subscripts for an array of `n` elements start with 0 and terminate to `n-1`. Array of matrices are defined in a similar way. For example if `A` is an array of six 4×4 matrix, the declaration statement is

```
MAT4 A[6];
```

The individual matrices are denoted as `A[0]`, `A[1]`, ... `A[5]`, while the element of row `i` and column `j` of the third matrix is addressed as `A[2][i][j]`

2.6 Functions working on matrices with non predefined dimensions

Some SpaceLib© functions have been realized in order to handle matrices of not predefined dimensions. These dimensions must be declared in the calling list of the functions. For technical reasons, these matrices have been defined in the function prototypes as pointers to real. For this reason the matrices must be recasted to that type. In the prototype of functions they appear as `MAT` type parameters. For example the sum of two 3×4 matrices can be performed by means of the following statement ($C = A+B$)

```
sum((real *) A, (real *) B, (real *) C, 3, 4);
```

where the prototype of function sum is

```
void sum (MAT A, MAT B, MAT C, int d1, int d2);
```

For the user's convenience a proper define statement has been included in the header `spacelib.h` (see § 6.1) and so the previous line can be more shortly written as

```
sum(M A, M B, M C, 3, 4);
```

The uppercase `M`, defined in `spacelib.h` as equivalent to the string `(real *)`, performs the recast. This rule can be applied to all the parameters of any function whose parameters are of type `MAT` (see § 3). As already mentioned, in order to simplify the use of the functions in the more usual situations, a number of macros have been defined. These macros perform a proper recast and assign the value of some parameters with standard constant values (i.e. the matrix dimensions). For instance the sum of two 3×3 matrices can be obtained by the following statement ($C = A + B$)

```
sum3(A,B,C);
```

Whenever it's possible, in order to avoid programming problems, users should use macros instead of functions.

2.7 Application examples

Two groups of examples are supplied with SpaceLib©:

- Short examples
- Big examples

Short examples are described throughout the § 3 of the manual while big examples are described in depth in the § 7. To compile the examples it is suggested to copy the sample files in the same directory as the library.

Chapter 3

General functions, Kinematics, Dynamics, Euler-Cardanic angles

In this section, the routines (functions and macros) of the library are briefly described. The routines are divided in few groups. A mnemonic description of the functions is sometimes added. When in doubt, the type of the parameters of the procedures can be verified looking at the function prototypes contained in the header files `spacelib.h` and `linear.h`. These files are listed in § 6.

3.1 Position, rotation and rototranslation matrices

`dhtom`

Denavit and Hartenberg parameters to Matrix (extended version). Contained in `spacelib.c`

Calling sequence:	`rv = dhtom (jtype, theta, d, b, a, alpha, q, m);`
Prototype:	`int dhtom (int jtype, real theta, real d, real b, real a, real alpha, real q, MAT4 m);`
Return value:	`int rv.`
Input parameters:	`int jtype; real theta, d, b, a, alpha, q.`
Output parameters:	`MAT4 m.`

Builds the position matrix **m** of a link from the extended *Denavit* and *Hartenberg's* parameters [3], [4] **theta**, **d**, **b**, **a**, **alpha**, the value of the joint coordinate **q** and the type of the joint **jtype**. **jtype** is an integer whose value must be either `Rev` or `Pri` (see § 2.4.3). `Rev` and `Pri` are constants defined in the header file `spacelib.h` (see § 6.1). If the joint type is prismatic, the value of **q** is added to **d**, while for revolute joint **q** is added to **theta**. The function returns the value **rv** that can be either `OK` or `NOTOK` (see § 2.4.3) in order to specify the correct outcome. The matrix computed by the function is equivalent to the following rototranslation combination:

$$[ROT(z;\mathbf{theta})]\,[TRAS(z;\mathbf{d})]\,[TRAS(x;\mathbf{a})]\,[TRAS(y;\mathbf{b})]\,[ROT(x;\mathbf{alpha})]$$

If **b** is equal to 0 the extended *Denavit* and *Hartenberg's* parameters coincide with the canonical ones [1].
Example: The position matrix `m` of frame (i) referred to frame (i-1) (see figure 3.1) is obtained by the following statement:

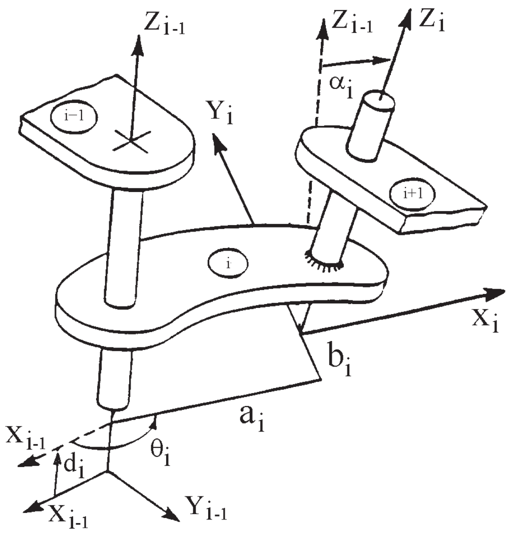
```
dhtom (Rev, theta, d, b, a, alpha, q, m);
```

The resulting matrix m is evaluated as:

$$m = \left[\begin{array}{ccc|c} \cos(\theta+q) & -\sin(\theta+q)\cos(\alpha) & \sin(\theta+q)\sin(\alpha) & a\cos(\theta+q) - b\sin(\theta+q) \\ \sin(\theta+q) & \cos(\theta+q)\cos(\alpha) & -\cos(\theta+q)\sin(\alpha) & a\sin(\theta+q) + b\cos(\theta+q) \\ 0 & \sin(\alpha) & \cos(\alpha) & d \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{3.1}$$

See also example 3.1.

See also: `DHtoMstd`, `rotat`, `screwtom`.



Extended Denavit and Hartenberg parameters	
θ	link rotation
d	link offset
b	shift (b=0 for standard definition)
a	link length
α	link twist

Figure 3.1: Definition of the *Denavit* and *Hartenberg*'s parameters.

DHtoMstd

Denavit and Hartenberg parameters to Matrix (standard version). Macro contained in `spacelib.h`

Calling sequence:	`rv = DHtoMstd (theta, d, a, alpha, m);`
Prototype:	`int DHtoMstd (real theta, real d, real a, real alpha, MAT4 m);`
Return value:	`int rv.`
Input parameters:	`real theta, d, a, alpha.`
Output parameters:	`MAT4 m.`

Builds the position matrix **m** of a link from the standard *Denavit* and *Hartenberg's* parameters [3], [4] **theta**, **d**, **a** and **alpha**. The function returns the value **rv** that can be either `OK` or `NOTOK` (see § 2.4.3) in order to specify the correct outcome. The matrix computed by the function is equivalent to the following rototranslation combination:

$$[ROT(z;\mathbf{theta})]\,[TRAS(z;\mathbf{d})]\,[TRAS(x;\mathbf{a})]\,[ROT(x;\mathbf{alpha})]$$

Equivalent to `dhtom (0, theta, d, 0., a, alpha, 0., m)`.

Example: The position matrix `m` of frame (i) referred to frame (i-1) (see figure 3.1) is obtained by the following statement:

```
DHtoMstd (theta, d, a, alpha, m);
```

The resulting matrix m is evaluated as:

$$m = \left[\begin{array}{ccc|c} \cos(\theta) & -\sin(\theta)\cos(\alpha) & \sin(\theta)\sin(\alpha) & a\cos(\theta) \\ \sin(\theta) & \cos(\theta)\cos(\alpha) & -\cos(\theta)\sin(\alpha) & a\sin(\theta) \\ 0 & \sin(\alpha) & \cos(\alpha) & d \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{3.2}$$

See also example 3.1.

See also: `dhtom`, `rotat`, `screwtom`.

Example 3.1. *See sample program* `E_DHTOM.C`.

The following example shows the use of `dhtom` and `DHtoMstd` for the direct kinematics of a serial manipulator. Numerical data refers to the *Stanford Arm* (see fig. 3.2 and table 3.1) which has one prismatic joint and five revolute ones. It is possible to see how the adoption of `dhtom` simplify the writing of the code.

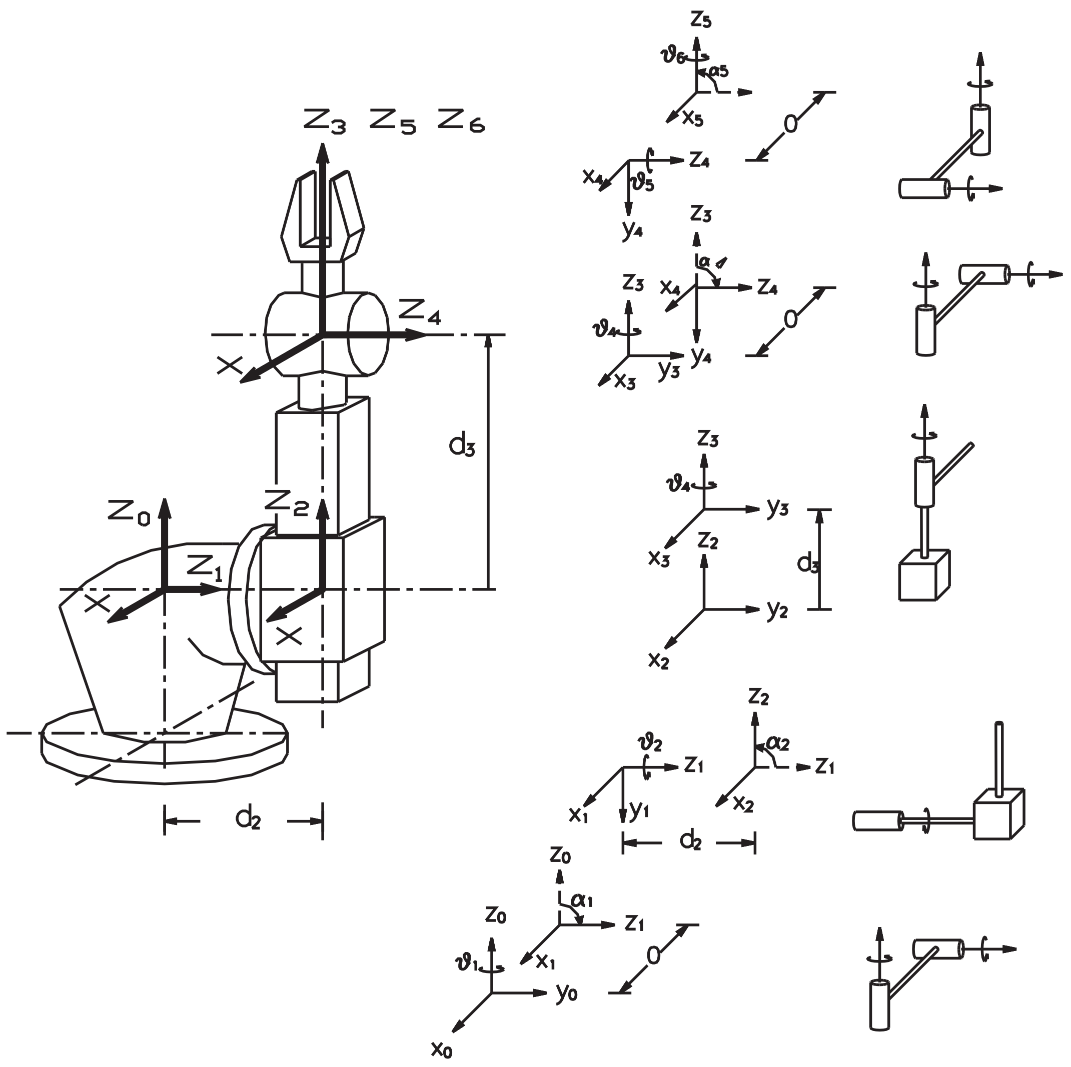


Figure 3.2: The *Stanford arm* with the *Denavit e Hantenberg* frames.

n.link	j.type	θ	d	a	α
1	R	q_1	0	0	$-\pi/2$
2	R	q_2	0.2	0	$\pi/2$
3	P	0	q_3	0	0
4	R	q_4	0	0	$-\pi/2$
5	R	q_5	0	0	$\pi/2$
6	R	q_6	0	0	0

Table 3.1: *Denavit e Hantenberg*'s parameters of *The Stanford arm* used in example 3.1.

```
/* E_dhtom.c   sample program to test the functions 'dhtom' and 'DHtoMstd' they must
               perform the same result
               direct kinematic of the Stanford Arm           */

#include <stdio.h> #include <conio.h> #include <math.h>
#include <stdlib.h> #include <time.h> #include "spacelib.h"

#define MAXLINK 6

int jtype[]={Rev, Rev, Pri, Rev, Rev, Rev};   /* joint type */
real alpha[]={-PIG_2, PIG_2, 0., -PIG_2, PIG_2, 0.};
real a[]={0., 0., 0., 0., 0., 0};           /* Denavit & Hartemberg's parameters */
real d[]={0., 0.2, 0., 0., 0., 0.};
real theta[]={0., 0., 0., 0., 0., 0.};
real q[]={0.11,0.22, 0.33, 0.44, 0.55, 0.66};   /* array of joint pos. variables */

#define RandData 0                            /* 1 = random data, 0 = fixed data */

void main(int argc,char *argv[]) {
    int i;

    MAT4 Ma, Mb, MM, dM;
    MAT4 tmp;

    if (RandData==1) {  // choose fixed or random data
       srand( (unsigned)time( NULL ) );
       for (i=0;i<MAXLINK;i++) {
            q[i]=rand()/(1.*RAND_MAX);
       }
    }

    idmat4(Ma);                                /* direct kinematics with 'dhtom' */
    for (i=0;i<MAXLINK;i++) {
        dhtom(jtype[i],theta[i],d[i],0.,a[i],alpha[i],q[i],MM);
        molt4(Ma,MM,tmp);
        mcopy4(tmp,Ma);
    }
    printm4("Ma: result with 'dhtom'",Ma);

    idmat4(Mb);                             /* direct kinematics with 'DHtoMstd' */
    for (i=0;i<MAXLINK;i++) {
        if (jtype[i]==Rev){
            DHtoMstd(q[i],d[i],a[i],alpha[i],MM);
        } else {
            DHtoMstd(theta[i],q[i],a[i],alpha[i],MM);
        }
        molt4(Mb,MM,tmp);
        mcopy4(tmp,Mb);
    }
    printm4("Mb: result with 'DHtoMstd",Mb);

    sub4(Ma,Mb,dM);
    printm4("the results must be identical and so dM=Ma-Mb=[0]",dM);

printf("hit any key"); getch(); }
```

extract

Extracts unit vector of screw axis and rotation angle from rotation matrix. Contained in `spacelib.c`

Calling sequence:	`extract(M A, u, &phi, dim);`
Prototype:	`void extract (MAT A, AXIS u, real *phi, int dim);`
Input parameters:	`MAT A; int dim.`
Output parameters:	`AXIS u; real phi.`

Extracts the unit vector **u** of the screw axis and the rotation angle **phi** from a rotation matrix stored in the upper-left 3×3 submatrix of a **dim**×**dim** position matrix **A**. **phi** is assumed in $[0..\pi]$. Matrix **A** must be recasted (see also §2.6). **dim** must be greater or equal to **3** (**dim** ≥ **3**). Matrix **A** must be recasted using the `M` operator (see § 2.6). `extract` performs the inverse operation than `rotat`.

Example: (see also example 3.2)

`extract (M R, u, &phi, 3)`	extracts `u` and `phi` from a rotation matrix `R`.
`extract (M m, u, &phi, 4)`	extracts `u` and `phi` from the rotation sub-matrix of the position matrix `m`.

Note: `M` is the recasting operator (see § 2.6).

See also: `mtoscrew`, `screwtom`, `rotat`.

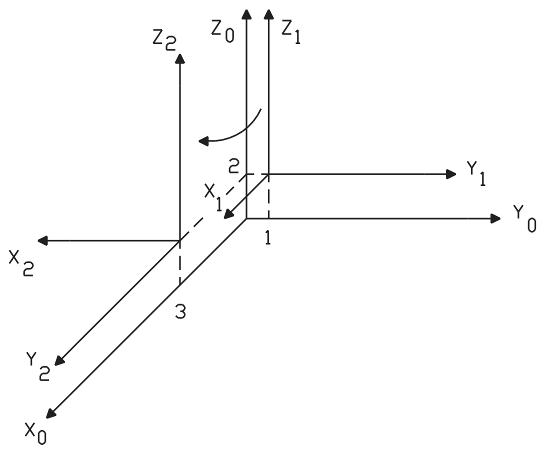


Figure 3.3: Rototranslation frame of example 3.2

Example 3.2. *See sample program* `EXTRACT.C`.

This example shows the extraction of the screw parameters of the rototranslation, which superimposes frame (1) onto (2) (see figure 3.3). The results are computed in frame (0). The rototranslation is contained in matrix $Q_{1,2(0)}{=}M_{0,2}\ M_{1,0}$. The position matrix of frame (0) with respect to reference frame (1) and the position matrix of frame (2) with respect to reference frame (0) are respectively

$$M_{1,0} = \left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -2 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \qquad M_{0,2} = \left[\begin{array}{ccc|c} 0 & 1 & 0 & 3 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

and the rototranslation matrix is

$$Q_{1,2(0)} = M_{0,2}\ M_{1,0} = \left[\begin{array}{ccc|c} 0 & 1 & 0 & 2 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] = \left[\begin{array}{ccc|c} & R & & T \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

The following statements

```
MAT4 Q= { {0., 1., 0., 0.},
         {-1., 0., 0., 2.},
          {0., 0., 1., 0.},
          {0., 0., 0., 1.} };
AXIS u;
real phi;
extract(M Q,u,&phi,4);
```

give the following result

$$phi = \pi/2 = 1.57079 \qquad u = [0, 0, -1]^t$$

mtoscrew

Matrix to screw. Contained in `spacelib.c`

Calling sequence:	`rv = mtoscrew (Q, u, &phi, &h, P);`
Prototype:	`int mtoscrew (MAT4 Q, AXIS u, real *phi, real *h, POINT P);`
Return value:	`int rv.`
Input parameters:	`MAT4 Q.`
Output parameters:	`AXIS u; real phi, h, POINT P.`

Extracts from a rototranslation matrix **Q** the parameters of the screw displacement (axis **u**, rotation angle **phi**, displacement **h** along **u**, a point of the axis **P**). Point **P** is the point of the screw axis nearest to the origin of the reference frame. The function returns the value **rv** that can be either `OK` or `NOTOK` (see § 2.4.3) in order to specify the correct outcome. `mtoscrew` performs the inverse operation than `screwtom`.

See also example 3.3.

See also: `screwtom`, `rotat`, `extract`.

Example 3.3. *See sample program* `MTOSCREW.C`.

Referring to example 3.2, the following statements

```
MAT4 Q= { {0., 1., 0., 0.},
          {-1., 0., 0., 2.},
          {0., 0., 1., 0.},
          {0., 0., 0., 1.}  };
AXIS u;
real phi,h;
POINT P;
mtoscrew(Q,u,&phi,&h,P);
```

gives the following result

$$phi = \pi/2 = 1.57079 \qquad h = 0$$

$$P = [1, -1, 0, 1]^t \qquad u = [0, 0, -1]^t$$

screwtom

Screw to Matrix. Contained in `spacelib.c`

Calling sequence:	`screwtom (u, phi, h, P, Q);`
Prototype:	`void screwtom (AXIS u, real phi, real h, POINT P, MAT4 Q);`
Input parameters:	`AXIS u; real phi, h; POINT P.`
Output parameters:	`MAT4 Q.`

Builds the rototranslation matrix **Q** from the axis of the screw displacement **u**, the rotation angle **phi**, the translation **h** along **u** and the coordinates of a point **P** of the axis. `screwtom` performs the inverse operation than `mtoscrew`.

See also example 3.4.

See also: `mtoscrew`, `extract`, `rotat`.

Example 3.4. *See sample program* `SCREWTOM.C`.

Referring to example 3.2, with the given values

$$phi = \pi/2 = 1.57079 \qquad h = 0 \qquad P = [1, -1, 0, 1]^t \qquad u = [0, 0, -1]^t$$

the following statements

```
MAT4 Q;
real phi=PIG_2;
real h=0;
AXIS u=Zaxis_n;
POINT P={1., 1., 0., 1.};
screwtom(u,phi,h,P,Q);
```

gives the resulting matrix

$$Q_{1,2(0)} = \left[\begin{array}{ccc|c} 0 & 1 & 0 & 2 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

rotat

Builds the rotation matrix R. Contained in `spacelib.c`

Calling sequence:	`rotat (u, phi, M A, dim);`
Prototype:	`void rotat (AXIS u, real phi, MAT A, int dim);`
Input parameters:	`AXIS u; real phi; int dim.`
Output parameters:	`MAT A.`

Builds the rotation matrix `R` from the unit vector **u** and the rotation angle **phi** of the angular displacement; it stores the matrix in the 3×3 upper left submatrix of a **dim**×**dim** matrix **A**. **dim** must be greater or equal to **3** (**dim** ≥ **3**). Matrix **A** must be recasted using the `M` operator (see § 2.6). `rotat` performs the inverse operation than `extract`.

Example:

`rotat(U, phi, M R, 3)`	builds a 3×3 rotation matrix `R`.
`rotat(U, phi, M A, 4)`	builds a rotation matrix storing it in the 3×3 upper left part of a 4×4 matrix `A`.

Note: `M` is the recasting operator (see §2.6).

See also example 3.5.

See also: `rotat2`, `rotat24`, `extract`.

Example 3.5. *See sample program* `ROTAT.C`.

Referring to example 3.2, with the given values

$$phi = \pi/2 = 1.57079 \qquad u = [0, 0, -1]^t$$

the following statements

```
MAT4 A;
AXIS u=Zaxis_n;
real phi=PIG_2;
rotat(u, phi,M A,4);
```

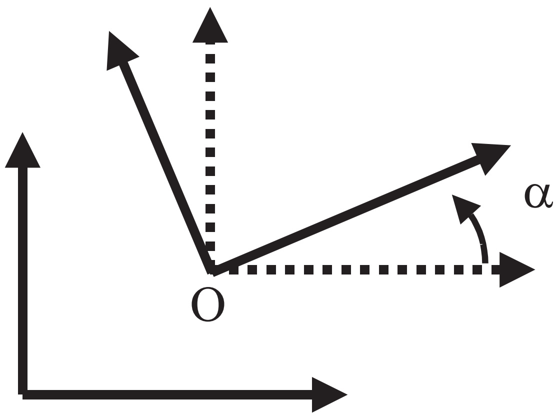


Figure 3.4: `rotat24` example of use: `rv=rotat24(a,alpha,O,M01)` with `a=X`, `Y`, or `Z`

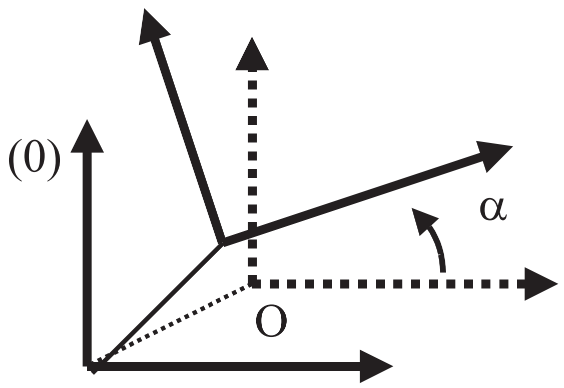


Figure 3.5: `rotat34` example of use: `rv=rotat34(a,alpha,O,M01)` with `a=X`, `Y`, or `Z`

gives the resulting matrix

$$A = \left[\begin{array}{ccc|c} 0 & 1 & 0 & ? \\ -1 & 0 & 0 & ? \\ 0 & 0 & 1 & ? \\ \hline ? & ? & ? & ? \end{array}\right]$$

where the character '?' means that the value of these elements is not affected by the function.

rotat2

Rotation around a frame axis. Contained in `spaceli3.c`

Calling sequence:	`rv = rotat2 (a, q, M R, dim);`
Prototype:	`int rotat2 (int a, real q, MAT R, int dim);`
Return value:	`int rv.`
Input parameters:	`int a, dim; real q.`
Output parameters:	`MAT R.`

This function builds a 3×3 rotation matrix **R** describing a rotation of angle **q** about axis **a**. **a** must be one of the constants `X`, `Y`, `Z`, `U` (see § 2.4.3). The rotation matrix is stored in the 3×3 upper-left submatrix of a **dim**×**dim** matrix **R**. **dim** must be greater or equal to **3** (**dim** ≥ **3**). If **a**==`U`, the rotation is assumed null (3×3 identity matrix generated). The **R** matrix must be recasted using the `M` operator (see § 2.6). The function returns the value **rv** that can be either `OK` or `NOTOK` (see § 2.4.3) in order to specify the correct outcome.

See also: `rotat`, `rotat24`, `rotat34`, `rotat23`.

rotat23

Rotation around an axis for 3×3 rotation matrices. Macro contained in `spacelib.h`

Calling sequence:	`rv = rotat23 (a, q, R);`
Prototype:	`int rotat23 (int a, real q, MAT3 R);`
Return value:	`int rv.`
Input parameters:	`int a; real q.`
Output parameters:	`MAT3 R`

This macro builds a 3×3 rotation matrix **R** describing a rotation of the angle **q** about axis **a**. Equivalent to `rotat2 (a, q, M R, 3)`.

See also: `rotat2`.

rotat24

Rotation matrix around an axis with origin in a given point. Contained in `spaceli3.c`

Calling sequence:	`rv = rotat24 (a, q, O, m);`
Prototype:	`int rotat24 (int a, real q, POINT O, MAT4 m);`
Return value:	`int rv.`
Input parameters:	`int a; real q; POINT O.`
Output parameters:	`MAT4 m.`

This function builds a position matrix **m** of a frame whose origin is stored in point **O** and rotated of angle **q** about axis **a** (see fig. 3.4). **a** must be one of the constants `X`, `Y`, `Z`, `U` (see § 2.4.3). If **a**==`U` the rotation is assumed null. The function returns the value **rv** that can be either `OK` or `NOTOK` (see § 2.4.3) in order to specify the correct outcome.

See also: `rotat`, `rotat2`, `rotat34`.

rotat34

Rotation matrix around an axis with origin in a given point. Contained in `spaceli4.c`

Calling sequence:	`rotat34 (a, q, O, m);`
Prototype:	`void rotat34 (int a, real q, POINT O, MAT4 m);`
Input parameters:	`int a; real q; POINT O.`
Output parameters:	`MAT4 m.`

This function builds a position matrix **m** of a frame whose origin is initially in point **O**. First of all the origin is placed in point **O**, then the frame (origin included) is rotated of angle **phi** about axis **a** of the absolute frame (see fig. 3.5). **a** must be one of the constants `X`, `Y`, `Z`, `U` (see § 2.4.3). If a==U the rotation is assumed null.

See also: `rotat`, `rotat2`, `rotat24`.

traslat

Builds the matrix m of a translation along a vector. Contained in `spaceli5.c`

Calling sequence:	`traslat (u, h, m);`
Prototype:	`void traslat (VECTOR u, real h, MAT4 m);`
Input parameters:	`VECTOR u; real h.`
Output parameters:	`MAT4 m.`

Builds the translation matrix **m** from the unit vector **u** and the translation distance **h** of the prismatic displacement.

See also: `traslat2`, `traslat24`, `mtoscrew`.

traslat2

Builds the matrix m of a translation along a frame axis. Contained in `spaceli5.c`

Calling sequence:	`traslat2 (a, h, m);`
Prototype:	`void traslat2 (int a, real h, MAT4 m);`
Input parameters:	`int a; real h.`
Output parameters:	`MAT4 m.`

Builds the matrix **m** of the translation along axis **a** and with translation distance **q**. **a** must be one of the constants `X`, `Y`, `Z`, `U` (see § 2.4.3).

See also: `traslat`, `traslat24`, `mtoscrew`.

traslat24

Builds the matrix m of a translation along a frame axis with origin in a given point. Contained in `spaceli5.c`

Calling sequence:	`traslat24 (a, h, p, m);`
Prototype:	`void traslat24 (int a, real h, POINT p, MAT4 m);`
Input parameters:	`int a; real h, POINT p.`
Output parameters:	`MAT4 m.`

Builds the matrix **m** of the translation along axis **a**, translation distance **q** and origin in point **P**. **a** must be one of the constants `X`, `Y`, `Z`, `U` (see § 2.4.3).

See also: `traslat`, `traslat2`, `mtoscrew`.

3.2 Speed and acceleration matrices

gtom

Gravity acceleration to Matrix. Contained in `spacelib.c`

Calling sequence:	`gtom (gx, gy, gz, Hg);`
Prototype:	`void gtom (real gx, real gy, real gz, MAT4 Hg);`
Input parameters:	`real gx, gy, gz.`
Output parameters:	`MAT4 Hg.`

Builds the gravity matrix **Hg** starting from the components **gx**, **gy**, **gz** of the gravity acceleration. Usually the `z` axis is vertical and points upwards and so the acceleration vector components are **gx** = 0 m/s^2, **gy** = 0 m/s^2, **gz** = -9.81 m/s^2.

See also example 3.6

Gtomegapto

G to omega dot. Contained in `spaceli4.c`

Calling sequence:	`Gtomegapto (G, omegapto);`
Prototype:	`void Gtomegapto (MAT3 G, VECTOR omegapto);`
Input parameters:	`MAT3 G.`
Output parameters:	`VECTOR omegapto.`

Extracts the angular acceleration vector **omegapto** from the 3×3 upper-left submatrix **G** of the acceleration matrix. It uses the relation

$$\dot{\Omega} = \frac{(G - G^t)}{2} \tag{3.3}$$

See also: `Wtovel`.

Example 3.6. *See sample program* `GTOM.C`.

This example shows how to create the acceleration matrix H_g. This is the most common situation where the body falls down along the z direction (see figure 3.6). The following statements

```
MAT4 Hg;
real gx=0.;
real gy=0.;
real gz=-9.81;
gtom(gx,gy,gz,Hg);
```

build the acceleration matrix H_g of the falling body in figure figure 3.6. H_g is:

$$H_g = \left[\begin{array}{ccc|c} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -9.81 \\ \hline 0 & 0 & 0 & 0 \end{array}\right]$$

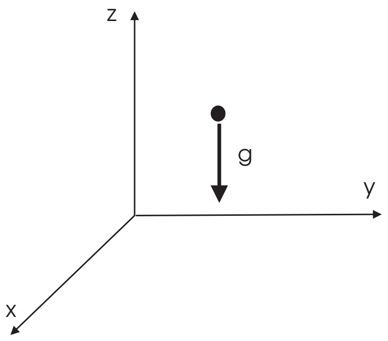


Figure 3.6: Frame definition for example 3.6

makeL

Builds a L matrix. Contained in `spaceli3.c`

Calling sequence:	`rv = makeL(jtype, u, pitch, P, L);`
Prototype:	`int makeL (int jtype, AXIS u, real pitch, POINT P, MAT4 L);`
Return value:	`int rv.`
Input parameters:	`int jtype; AXIS u; real pitch; POINT P.`
Output parameters:	`MAT4 L.`

This function builds a *ISA*'s (*Instantaneous Screw Axis*) matrix **L** describing screw motion (including simple rotations or a translation) about an axis which passes through the point **P** and whose unit vector is **u**. **pitch** is the pitch of the screw. **jtype** specifies the type of the motion. It must be either the constant `Pri` for prismatic joints or `Rev` for revolute or screw joints (see § 2.4.3). `Pri` and `Rev` are constants defined in `spacelib.h` (see § 6.1). If **jtype**==`Pri`, `pitch` is ignored. The function returns the value **rv** that can be either `OK` or `NOTOK` (see § 2.4.3) in order to specify the correct outcome.

See also example 3.7 and *example* 3.8.

See also: `makeL2`

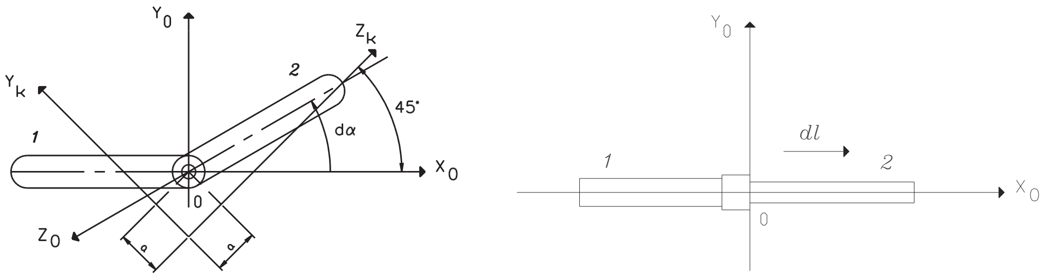


Figure 3.7: Frames definition for examples 3.7 and 3.8

Example 3.7. *See sample program* `MAKEL.C` *and* `MAKEL0.C`.

This example shows how to create the *ISA*'s (*Instantaneous Screw Axis*) matrix L of the body n. 2 rotating about an axis coincident with z_0 in two different reference frame (0) and (k) (see figure 3.7). The 2^{nd} element of this revolute joint rotates about axis z of reference frame (0) and has $a = 1.2$ m. The L matrix referred to the reference frame (0) is

$$L_{1,2(0)} = \left[\begin{array}{ccc|c} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \end{array}\right]$$

This matrix can be built by the following statements

```
POINT O=ORIGIN;
real pitch=0.;
AXIS u=Zaxis;
MAT4 LO;
makeL(Rev,u,pitch,O,LO);
```

The L matrix referred to frame (k) is

$$L_{1,2(k)} = \left[\begin{array}{ccc|c} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1.2 \\ 0 & -1 & 0 & 1.2 \\ \hline 0 & 0 & 0 & 0 \end{array}\right]$$

It is built by the following statements

```
POINT P={0.,1.2,1.2,1.};
real pitch=0.;
AXIS u=Xaxis_n;
MAT4 Lk;
makeL(Rev,u,pitch,P,Lk);
```

Example 3.8. *See sample program* `MAKEL_ P.C.`

In this example body n. 2 moves in the direction of x_0. Frame (0) is embedded on body n. 1 (see figure 3.7). The *ISA*'s (*Instantaneous Screw Axis*) matrix L of this prismatic joint referred to frame (0) is:

$$L_{1,2(0)} = \left[\begin{array}{ccc|c} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \end{array}\right]$$

This matrix is built by the following statements:

```
POINT O=ORIGIN;
AXIS u=Xaxis;
MAT4 LO;
real pitch;
makeL(Pri,u,pitch,O,LO);
```

makeL2

Builds a L matrix - version 2. Contained in `spaceli3.c`

Calling sequence:	`rv = makeL2 (jtype, a, pitch, P, L);`
Prototype:	`int makeL2 (int jtype, int a, real pitch, POINT P, MAT4 L);`
Return value:	`int rv.`
Input parameters:	`int jtype, a; real pitch, POINT P.`
Output parameters:	`MAT4 L.`

This function builds a *ISA*'s (*Instantaneous Screw Axis*) matrix **L** describing a rotation or a translation about an axis parallel to the frame axis **a** and passing through the point **P**. **a** must be one of the constants `X`, `Y`, `Z`, `U` (see § 2.4.3). **pitch** is the pitch of the screw. **jtype** specifies the type of the motion. It must be either the constant `Pri` for prismatic joints or `Rev` for revolute or screw joints (see § 2.4.3). `Pri` and `Rev` are constants defined in `spacelib.h` (see § 6.1). If jtype=Pri, pitch is ignored. The function returns the value **rv** that can be either `OK` or `NOTOK` (see § 2.4.3) in order to specify the correct outcome.

See also: `makeL`.

WtoL

Extracts L matrix from the corresponding W matrix. Contained in `spacelib.c`

Calling sequence:	`WtoL (W, L);`
Prototype:	`void WtoL (MAT4 W, MAT4 L);`
Input parameters:	`MAT4 W;`
Output parameters:	`MAT4 L;`

Extracts *ISA*'s (*Instantaneous Screw Axis*) matrix **L** from the corresponding **W** matrix. If **W** is the null matrix, the function returns a **L** matrix filled with zeros (null matrix).

See also example 3.9 and *example* 3.10.

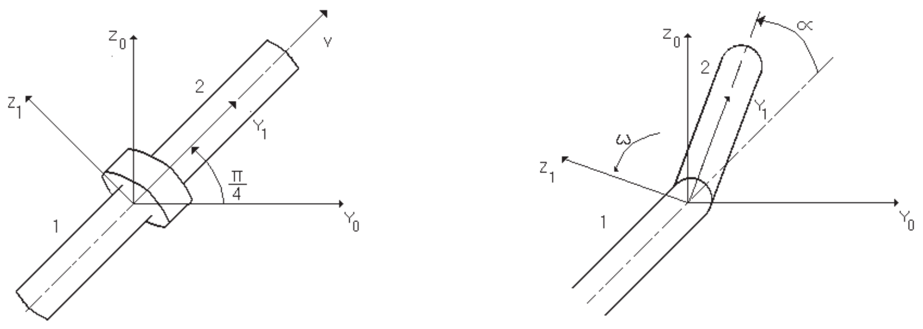


Figure 3.8: Frames definition for examples 3.9 and 3.10

Example 3.9. *See sample program* `WTOL_P.C`.

This example shows how to extract the *ISA*'s (*Instantaneous Screw Axis*) matrix L knowing the velocity one. In this case the considered joint is prismatic and it lies in the $y_0 - z_0$ plane forming an angle of $\pi/4$ with y_0. The element 2 is connected by a prismatic joint to body 1 which does not move with respect to frame (0) (see figure 3.8). Body 2 has the following velocity matrix W referred to frame (0) embedded on body 1:

$$W_{1,2(0)} = \left[\begin{array}{ccc|c} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1.4142 \\ 0 & 0 & 0 & 1.4142 \\ \hline 0 & 0 & 0 & 0 \end{array}\right]$$

The L matrix is built by the statements:

```
MAT4 W = { {0.,0.,0.,0.},
           {0.,0.,0.,1.4142},
           {0.,0.,0.,1.4142},
           {0.,0.,0.,0.} };
MAT4 L;
WtoL(W,L);
```

and the result is:

$$L_{1,2(0)} = \left[\begin{array}{ccc|c} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.7071 \\ 0 & 0 & 0 & 0.7071 \\ \hline 0 & 0 & 0 & 0 \end{array}\right]$$

Example 3.10. *See sample program* WTOL_R.C.

This example shows how to extract the *ISA*'s (*Instantaneous Screw Axis*) matrix L knowing the velocity one. In this case the considered joint is revolute and it rotates about axes x_0 orthogonal to plane $x_0 - y_0$ passing through the center of the joint. The element 2 is connected by a revolute joint to body 1 which does not move with respect to frame (0) (see figure 3.8). Body 2 has the following velocity matrix referred to frame (0):

$$W_{1,2(0)} = \left[\begin{array}{ccc|c} 0 & 0 & 0 & 0 \\ 0 & 0 & -2 & 0 \\ 0 & 2 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \end{array}\right]$$

The L matrix is built by the statements:

```
MAT4 W = { {0.,0.,0.,0.},
           {0.,0.,-2.,0.},
           {0.,2.,0.,0.},
           {0.,0., 0., 0.} };
MAT4 L;
WtoL(W,L);
```

and the result is:

$$L_{1,2(0)} = \left[\begin{array}{ccc|c} 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \end{array}\right]$$

Wtovel

Velocity matrix to velocity parameters. Contained in spaceli3.c

Calling sequence:	`Wtovel (W, u, &omega, &vel, P);`
Prototype:	`void Wtovel (MAT4 W, AXIS u, real *omega, real *vel, POINT P);`
Input parameters:	`MAT4 W.`
Output parameters:	`AXIS u; real omega, vel; POINT P.`

Extracts the screw parameters from a velocity matrix **W**. The parameters are: **u** axis of rotation (unit vector), **omega** angular speed around **u** (scalar), **vel** linear velocity along **u**, **P** a point of the axis (the closest to the origin). If **omega** == 0 (pure translation) the origin is assumed as **P**. If **omega** == 0 and **vel** == 0, **u** is undefined.

See also example 3.11.

Example 3.11. *See sample program* WTOVEL.C *and* WTOVEL_P.C.

Considering a velocity matrix W

$$W = \left[\begin{array}{ccc|c} 0 & 2 & 2.5 & 2.5 \\ 2 & 0 & -4.5 & 1.7 \\ -2.5 & 4.5 & 0 & 3.2 \\ \hline 0 & 0 & 0 & 0 \end{array}\right]$$

The following statements:

```
MAT4 W={{0.,-2.,2.5,2.5},
        {2.,0.,-4.5,1.7},
        {-2.5,4.5,0.,3.2},
        {0.,0.,0.,0.}  };
AXIS u;
real omega;
real vel;
POINT P;
Wtovel(W,u,&omega,&vel,P);
```

evaluates the angular velocity *omega*, scalar velocity *vel* and a point P of the screw axis u. P is the point of the axis nearest to the origin of the reference frame. The result is:

$$u = [0.815, 0.453, 0.362]^t \quad omega = 5.52 \text{ rad/s}$$

$$P = [0.151, -0.308, 0.046, 1.]^t \quad vel = 3.965 \text{ m/s}$$

Considering a velocity matrix W

$$W = \left[\begin{array}{ccc|c} 0 & 0 & 0 & 2.5 \\ 0 & 0 & 0 & 1.7 \\ 0 & 0 & 0 & 3.2 \\ \hline 0 & 0 & 0 & 0 \end{array}\right]$$

The statements are the same as in the previous case except that the matrix W is filled with different values. The result in this case is:

$$u = [0.568, 0.386, 0.727]^t \quad omega = 0 \text{ rad/s}$$

$$P = [0, 0, 0, 1]^t \qquad vel = 4.40227 \text{ m/s}$$

velactoWH

Velocity and Acceleration to W and H matrices. Contained in `spacelib.c`

Calling sequence:	`rv = int velacctoWH (jtype, qp, qpp, W, H);`
Prototype:	`int velacctoWH (int jtype, real qp, real qpp, MAT4 W, MAT4 H);`
Return value:	`int rv.`
Input parameters:	`int jtype; real qp, qpp.`
Output parameters:	`MAT4 W, H.`

Builds both velocity and acceleration matrices in local frame (**W** and **H**) from the values of the joint velocity and acceleration (**qp** and **qpp**) and the type of the joint **jtype**. The axis of the movement is the z axis of the local reference frame. **jtype** is an integer whose value must be either `Rev` or `Pri`. `Rev` and `Pri` are constant (see §2.4.3) defined in the the header file `spacelib.h` (see §6.1). Frames are assumed to be positioned using the *Denavit* and *Hartenberg's* convention [3], [4]. The function returns the value **rv** that can be either `OK` or `NOTOK` (see § 2.4.3) in order to specify the correct outcome.

See also: `velacctoWH2`, `velacctoWH3`.

velacctoWH2

Velocity and Acceleration to W and H matrices - version 2. Contained in `spacelib.c`

Calling sequence:	`rv = int velacctoWH2 (jtype, a, qp, qpp, W, H);`
Prototype:	`int velacctoWH2 (int jtype, int a, real qp, real qpp, MAT4 W, MAT4 H);`
Return value:	`int rv.`
Input parameters:	`int jtype, a; real qp, qpp.`
Output parameters:	`MAT4 W, H.`

Builds both local speed and acceleration matrices (**W** and **H**) from the values of the velocity and acceleration **qp** and **qpp** of the link around the axis **a**. The motion axis is coincident with x, y, or z of the local reference frame. This function is equivalent to `velactoWH` except that the movement axis can be specified. **a** must be one of the constants `X`, `Y`, `Z`, `U` (see § 2.4.3). The function returns the value **rv** that can be either `OK` or `NOTOK` (see § 2.4.3) in order to specify the correct outcome.
The following statement

```
velacctoWH2(jtype,Z,qp,qpp,W,H);
```

is equivalent to

```
velacctoWH(jtype,qp,qpp,W,H);
```

See also example 3.12.

See also: `velactoWH`.

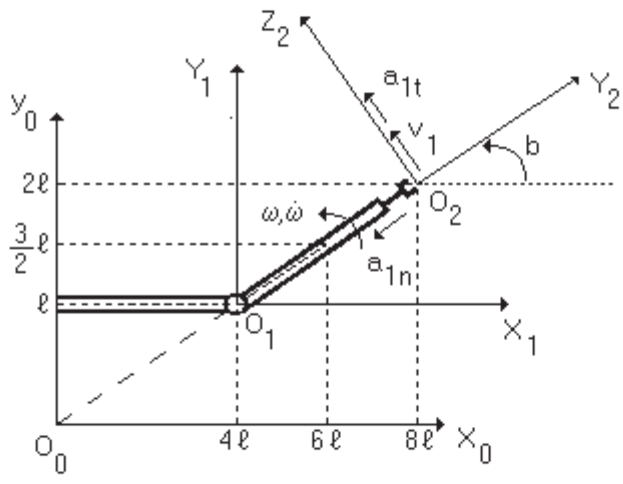


Figure 3.9: Frames definition for example 3.12

Example 3.12. *See sample program* `VELWH2.C`.

The fixed reference frames are defined. The two bodies are connected by a revolute joint. One body is fixed while the other rotates about the origin of frame (1) (see figure 3.9). With the given values

$$l = 0.1 \text{ m} \qquad \omega = 1.5 \text{ rad/s} \qquad \dot{\omega} = 0.9 \text{ rad/s}^2$$

the velocity and acceleration matrices referred to frame (1) are

$$W_{12(1)} = \left[\begin{array}{ccc|c} 0 & -1.5 & 0 & 0 \\ 1.5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \end{array}\right] \qquad H_{12(1)} = \left[\begin{array}{ccc|c} -2.25 & -0.9 & 0 & 0 \\ 0.9 & 2.25 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \end{array}\right]$$

These matrices are built by the statements

```
MAT4 W1,H1;
real qp=1.5;
real qpp=0.9;
velacctoWH2(Rev,Z,qp,qpp,W1,H1);
```

velacctoWH3

Velocity and Acceleration to W and H matrices - version 3. Contained in `spaceli4.c`

Calling sequence:	`velacctoWH3 (jtype, a, qp, qpp, O, W, H);`
Prototype:	`void velacctoWH3 (int jtype, int a, real qp, real qpp, POINT O, MAT4 W, MAT4 H);`
Input parameters:	`int jtype, a; real qp, qpp.`
Output parameters:	`POINT O; MAT4 W, H.`

Builds both local speed and acceleration matrices (**W** and **H**) from the values of the velocity and acceleration **qp** and **qpp** of the link around the axis **a**. The motion axis is parallel to x, y, or z of the local reference frame. This function is similar to `velacctoWH2`: the movement axis is parallel to one of the coordinate axes and can pass through a point different from the origin of the reference frame. **a** must be one of the constants `X`, `Y`, `Z`, `U` (see § 2.4.3). **O** is a point of the rotation axis.
The following statement

```
velacctoWH3(jtype,a,qp,qpp,O,W,H);
```

is equivalent to

```
velacctoWH2(jtype,a,qp,qpp,W,H);
```

when the point **O** coincides with the origin of the reference frame (and therefore **a** is one of the axes of the reference frame).

Example: Referring to example 3.12, the matrix $W_{1,2(0)}$ is obtained by the following statement

```
MAT4 W0, H0;
real qp=1.5;
real qpp=0.9;
POINT O1={0.4,0.1,0,1};
velacctoWH3(Rev,Z,qp,qpp,O1,W0,H0);
```

See also: `velacctoWH2`.

coriolis

Coriolis theorem.	Contained in `spacelib.c`
Calling sequence:	`coriolis (H0, H1, W0, W1, H);`
Prototype:	`void coriolis (MAT4 H0, MAT4 H1, MAT4 W0, MAT4 W1, MAT4 H);`
Input parameters:	`MAT4 H0, H1, W0, W1.`
Output parameters:	`MAT4 H.`

Performs the Coriolis' theorem:

$$H = H_0 + H_1 + 2W_0 \cdot W_1 \tag{3.4}$$

3.3 Inertial and Actions Matrices

dyn_eq

Solve Direct Dynamics system.	Contained in `linear.c`
Calling sequence:	`rv = int dyn_eq (J, Wp, F, var);`
Prototype:	`int dyn_eq (MAT4 J, MAT4 Wp, MAT4 F, int var[2][6]);`
Return value:	`int rv.`
Input parameters:	`MAT4 J (I); int var[2][6].`
In-out parameters:	`MAT4 Wp, F.`

Evaluates the acceleration term $\dot{W}$ of a rigid body free in space solving the matrix equation

$$\Phi = skew\left(\dot{W} \cdot J\right) \tag{3.5}$$

where `Wp is` $\dot{W}$ `and F is` Φ. It can also extract the velocity matrix W of a body from the angular momentum matrix Γ solving the equation

$$\Gamma = skew\,(W \cdot J) \tag{3.6}$$

where `Wp is` W, and `F is` Γ.
var specifies which elements of **Wp** and **F** are unknown (for more details on this function see also § 5). The function returns the value **rv** that can be either `OK` or `NOTOK` (see § 2.4.3) in order to specify the correct outcome.

actom

Actions to Matrix. Contained in `spacelib.c`

Calling sequence:	`actom (fx, fy, fz, cx, cy, cz, PHI);`
Prototype:	`void actom (real fx, real fy, real fz, real cx, real cy, real cz, MAT4 PHI);`
Input parameters:	`real fx, fy, fz, cx, cy, cz;`
Output parameters:	`MAT4 PHI.`

Builds the action matrix **PHI** from the components of the forces **fx**, **fy**, **fz** and the torque (or couples) **cx**, **cy**, **cz**.

Example: With the given values

$$fx = a, \quad fy = b, \quad fz = c, \quad cx = d, \quad cy = e, \quad cz = f$$

the statement

```
actom(fx, fy, fz, cx, cy, cz, PHI);
```

fills the matrix PHI in the following way

$$PHI = \left[\begin{array}{rrr|r} 0 & -f & e & a \\ f & 0 & -d & b \\ -e & d & 0 & c \\ \hline -a & -b & -c & 0 \end{array}\right]$$

jtoJ

Inertia moment and mass to inertia matrix. Contained in `spacelib.c`

Calling sequence:	`jtoJ (mass, jxx, jyy, jzz, jxy, jyz, jxz, xg, yg, zg, J);`
Prototype:	`void jtoJ (real mass, real jxx, real jyy, real jzz, real jxy, real jyz, real jxz, real xg, real yg, real zg, MAT4 J);`
Input parameters:	`real mass, jxx, jyy, jzz, jxy, jyz, jxz, xg, yg, zg.`
Output parameters:	`MAT4 J.`

Builds the inertia matrix **J** of a body from the values of its mass **m**, its barycentral moments of inertia **jxx**, **jyy**, **jzz**, **jxy**, **jyz**, **jxz** and the position of its center of mass **xg**, **yg**, **zg**. The barycentral frame <u>must</u> be parallel to the reference frame. The resulting matrix is:

$$J = \left[\begin{array}{ccc|c} Ixx & Iyx & Izx & m\,xg \\ Ixy & Iyy & Izy & m\,yg \\ Ixz & Iyz & Izz & m\,zg \\ \hline m\,xg & m\,yg & m\,zg & m \end{array}\right]$$

The elements I of the **J** matrix <u>are not</u> the usual barycentral moments. These elements are related to the usual barycentral moments as follows:

$$Ixx = \frac{-\,Jxx + Jyy + Jzz}{2} \qquad Iyy = \frac{-\,Jyy + Jxx + Jzz}{2} \qquad Izz = \frac{-\,Jzz + Jxx + Jyy}{2}$$

$$Ixy = -Jxy \qquad Iyz = -Jyz \qquad Izx = -Jzx$$

where the value of `Ixx`, `Iyy`, `Izz` must be positive, therefore `Jxx`, `Jyy`, `Jzz` cannot be assigned random values. The **J** elements are defined as follows

$$Jxx = \int y^2 + z^2 \,\mathrm{d}m \qquad Jyy = \int x^2 + z^2 \,\mathrm{d}m \qquad Jxx = \int x^2 + y^2 \,\mathrm{d}m$$

$$Jxy = \int -xy \,\mathrm{d}m \qquad Jyz = \int -yz \,\mathrm{d}m \qquad Jxz = \int -xz \,\mathrm{d}m$$

The I elements are defined as follows

$$Ixx = \int x^2\,\mathrm{dm} \qquad Iyy = \int y^2\,\mathrm{dm} \qquad Ixx = \int z^2\,\mathrm{dm}$$

$$Ixy = \int xy\,\mathrm{dm} \qquad Iyz = \int yz\,\mathrm{dm} \qquad Ixz = \int xz\,\mathrm{dm}$$

See also example 3.13.

psedot

Pseudo scalar product. Contained in `spaceli5.c`

Calling sequence:	`ps = psedot (L, F);`
Prototype:	`real psedot (MAT4 L, MAT4 F);`
Return value:	`real ps.`
Input parameters:	`MAT4 L, F.`

Performs the pseudo-scalar product **ps** between matrices **L** and **F**.

Example:

If `W` is the velocity matrix of a body and `F` is the matrix of the actions (forces and torques) applied to it,

$$W = \left[\begin{array}{ccc|c} 0 & -\omega_z & \omega_y & v_x \\ \omega_z & 0 & -\omega_x & v_y \\ -\omega_y & \omega_x & 0 & v_z \\ \hline 0 & 0 & 0 & 0 \end{array}\right] \qquad \Phi = \left[\begin{array}{ccc|c} 0 & -c_z & c_y & f_x \\ c_z & 0 & -c_x & f_y \\ -c_y & c_x & 0 & f_z \\ \hline -f_x & -f_y & -f_z & 0 \end{array}\right]$$

where f is the force applied to the body and c is the torque, then the power

$$w = W \odot \Phi = \omega_x c_x + \omega_y c_y + \omega_z c_z + v_x f_x + v_y f_y + v_z f_z$$

is evaluated as

```
w=PseDot(W,F)
```

If `L` represents the screw axis of a joint and `F` is the total action transmitted by it, the actuator force (or torque) is evaluated as

```
Fl=PseDot(L,F)
```

Example 3.13. *See sample program* `JTOJ.C`.

This example shows how to create the inertial matrix of a cylinder in two different frames (0) and (1). The first one is centered in the center of mass G. The cylinder in figure 3.10 has $r = 1$ m, a density of 1 kg/m^3 and $h = 5$ m. G is the center of mass. Its position in frame (0) is $[0, 0, 0]$ and in frame (1) $[0, 3, 4]$. The following statement

```
MAT4 M10;
MAT4 J0, J1;
real m=15.71;
real jxx=36.633;
real jyy=36.633;
real jzz=7.850;
real jxy=0., jyz=0., jxz=0.;
real xg=0.,  yg=0.,  zg=0.;
jtoJ(m, jxx, jyy, jzz, jxy, jyz, jxz, xg, yg, zg, J0);
```

where the known values jxx, jyy, jzz, jxy, jyz, jxz are the usual baricentral inertia moments while xg, yg, zg are the baricentral coordinates in frame (0), builds the inertia matrix $J_{(0)}$ of the cylinder in the baricentral reference frame:

$$J_{(0)} = \left[\begin{array}{ccc|c} I_{xx} & 0 & 0 & 0 \\ 0 & I_{yy} & 0 & 0 \\ 0 & 0 & I_{zz} & 0 \\ \hline 0 & 0 & 0 & mass \end{array}\right] = \left[\begin{array}{ccc|c} 3.925 & 0 & 0 & 0 \\ 0 & 3.925 & 0 & 0 \\ 0 & 0 & 32.708 & 0 \\ \hline 0 & 0 & 0 & 15.71 \end{array}\right]$$

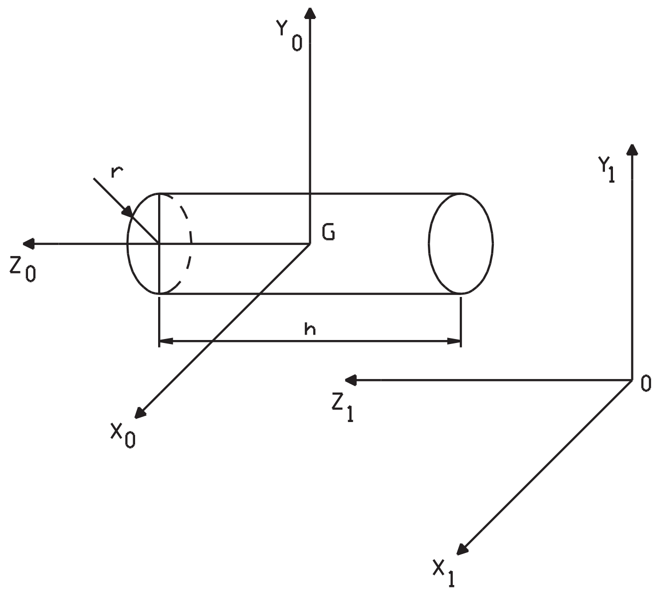


Figure 3.10: Frames definition for example 3.13

The following statement

```
MAT4 M10;
MAT4 J0, J1;
real m=15.71;
real jxx=36.633;
real jyy=36.633;
real jzz=7.850;
real jxy=0., jyz=0., jxz=0.;
xg=0.;  yg=3.;  zg=4.;
jtoJ(m,jxx,jyy,jzz,jxy,jyz,jxz,xg,yg,zg, J1);
```

where jxx, jyy, jzz, jxy, jyz, jxz are the usual inertia moments in frame (0) and xg, yg, zg are the baricentral coordinates in frame (1), builds the inertia matrix $J_{(1)}$ of the cylinder in frame (1)

$$J_{(1)} = \left[\begin{array}{ccc|c} I_{xx} & I_{yx} & I_{zx} & mass \cdot x_g \\ I_{xy} & I_{yy} & I_{zy} & mass \cdot y_g \\ I_{xz} & I_{yz} & I_{zz} & mass \cdot z_g \\ \hline -mass \cdot x_g & -mass \cdot y_g & -mass \cdot z_g & mass \end{array}\right] = \left[\begin{array}{ccc|c} 3.925 & 0 & 0 & 0 \\ 0 & 145.315 & 188.52 & 47.13 \\ 0 & 188.52 & 284.068 & 62.84 \\ \hline 0 & 47.13 & 62.84 & 15.71 \end{array}\right]$$

It is easy to verify that inertia matrix $J_{(1)}$ can also be obtained using the function `trasf_mamt4` (see § 3.4) with the statement

```
trasf_mamt4(J0,M10,J1)
```

where `M10` is the position matrix $M_{1,0}$ of frame (0) with respect to frame (1).

3.4 Matrix transformations

3.4.1 Matrix normalization

normal

Normalizes (orthogonalises) a 3×3 rotation matrix or the 3×3 upper-left submatrix of a position matrix. Contained in `spacelib.c`

Calling sequence:	`rv = normal (M R, n);`
Prototype:	`int normal (MAT R, int n);`
Return value:	`int rv.`
Input parameters:	`int n.`
In-out parameters:	`MAT R.`

Normalizes the rotation matrix **R** (if **n**==3) or the rotation part (if **n**==4), that is **R**, of a position matrix. (When **n**==4 the 3×3 upper-left submatrix is made orthogonal). **R** is iteratively put equal to (see [2])

$$R_{i+1} = \frac{1}{2} \cdot \left(\frac{1}{\sqrt[3]{det(R_i)}} \cdot R_i + \sqrt[3]{det(R_i)} \cdot (R_i^t)^{-1} \right) \tag{3.7}$$

until R_{i+1} does not vary in one iteration. This function is used when the rotation matrix is evaluated by numerical procedure and could contain errors. The function returns the value **rv** that can be either `OK` or `NOTOK` (see § 2.4.3) in order to specify the correct outcome. Matrix **R** must be recasted using the `M` operator (see § 2.6).

See also: `normal3`, `normal4`.

normal3

Normalizes (orthogonalises) a 3×3 rotation matrix. Macro contained in `spacelib.h`

Calling sequence:	`rv = normal3 (R);`
Prototype:	`int normal3 (MAT3 R);`
Return value:	`int rv.`
In-out parameters:	`MAT3 R.`

This macro normalizes a 3×3 rotation matrix **R**.
Equivalent to `normal(M R,3);`.

See also: `normal`.

normal4

Normalizes the rotation part of the 4×4 transformation matrix. Macro contained in `spacelib.h`

Calling sequence:	`rv = normal4 (R);`
Prototype:	`int normal4 (MAT4 R);`
Return value:	`int rv.`
In-out parameters:	`MAT4 R.`

This macro normalizes the rotation part of the 4×4 transformation matrix **R**.
Equivalent to `normal(M R,4);`.

See also: `normal`.

norm_simm_skew

Normalizes symmetric or skew-symmetric matrices. Contained in `spacelib.c`

Calling sequence:	`rv = int norm_simm_skew (M A, n, dim, sign);`
Prototype:	`int norm_simm_skew (MAT A, int n, int dim, int sign);`
Return value:	`int rv.`
Input parameters:	`int n, dim, sign.`
In-out parameters:	`MAT A.`

Normalizes the upper-left square **n**×**n** submatrix of a square **dim**×**dim** matrix **A** (extracts the symmetric or skew-symmetric part of **A**). **n** must be greater or equal to **dim** (**n** ≥ **dim**). This function can be used when matrix **A** is evaluated by numerical procedure and could contain errors. **sign** is an integer whose value can be either `SYMM` or `SKEW`. `SYMM` and `SKEW` are constants defined in the header file `spacelib.h` (see § 6.1 and § 2.4.3). The function returns the value **rv** that can be either `OK` or `NOTOK` (see § 2.4.3) in order to specify the correct outcome. Matrix **A** must be recasted using the `M` operator (see § 2.6). If **sign**==`SYMM` then (normalizes symmetric matrices)

$$\mathbf{A} = \frac{A + A^t}{2} \tag{3.8}$$

If **sign**==`SKEW` then (normalizes skew-symmetric matrices)

$$\mathbf{A} = \frac{A - A^t}{2} \tag{3.9}$$

See also: `n_simm3`, `n_simm34`, `n_simm4`, `n_skew3`, `n_skew34`, `n_skew4`.

n_simm3

Normalize 3×3 symmetric matrices. Macro contained in `spacelib.h`

Calling sequence:	`rv = n_simm3 (A);`
Prototype:	`int n_simm3 (MAT3 A);`
Return value:	`int rv.`
In-out parameters:	`MAT3 A.`

Macro useful to reduce numerical computation errors on the 3×3 matrix **A**. It uses the equation (3.8). Equivalent to `norm_simm_skew(M A, 3, 3, SYMM);`.

See also: `norm_simm_skew`.

n_simm34

Normalizes the 3×3 upper-left symmetric part of a 4×4 matrix. Macro contained in `spacelib.h`

Calling sequence:	`rv = n_simm34 (A);`
Prototype:	`int n_simm34 (MAT4 A);`
Return value:	`int rv.`
In-out parameters:	`MAT4 A.`

Macro useful to reduce numerical computation errors on the 3×3 submatrix of a 4×4 matrix **A**. It uses the equation (3.8). Equivalent to `norm_simm_skew(M A, 3, 4, SYMM)`.

See also: `norm_simm_skew`.

n_simm4

Normalizes 4×4 symmetric matrices. Macro contained in `spacelib.h`

Calling sequence:	`rv = n_simm4 (A);`
Prototype:	`int n_simm4 (MAT4 A);`
Return value:	`int rv.`
In-out parameters:	`MAT4 A.`

Macro useful to reduce numerical computation errors on the 4×4 matrix **A**. It uses the equation (3.8). Equivalent to `norm_simm_skew(M A, 4, 4, SYMM)`.

See also: `norm_simm_skew`.

n_skew3

Normalizes 3×3 skew-symmetric matrices. Macro contained in `spacelib.h`

Calling sequence:	`rv = n_skew3 (A);`
Prototype:	`int n_skew3 (MAT3 A);`
Return value:	`int rv.`
In-out parameters:	`MAT3 A.`

Macro useful to reduce numerical computation errors on the 3×3 matrix **A**. It uses the equation (3.9).
Equivalent to `norm_simm_skew(M A, 3, 3, SKEW)`.

See also: `norm_simm_skew`.

`n_skew34`

Normalizes 3×3 skew-symmetric part of a 4×4 matrix. Macro contained in `spacelib.h`

Calling sequence:	`rv = n_skew34 (A);`
Prototype:	`int n_skew34 (MAT4 A);`
Return value:	`int rv.`
In-out parameters:	`MAT4 A.`

Macro useful to reduce numerical computation errors on the 3×3 submatrix of a 4×4 matrix **A**. It uses the equation (3.9).
Equivalent to `norm_simm_skew(M A, 3, 4, SKEW)`.

See also: `norm_simm_skew`.

`n_skew4`

Normalizes 4×4 skew-symmetric matrix. Macro contained in `spacelib.h`

Calling sequence:	`rv = n_skew4 (A);`
Prototype:	`int n_skew4 (MAT4 A);`
Return value:	`int rv.`
In-out parameters:	`MAT4 A.`

Macro useful to reduce numerical computation errors on the 4×4 matrix **A**. It uses the equation (3.9).
Equivalent to `norm_simm_skew(M A, 4, 4, SKEW)`.

See also: `norm_simm_skew`.

3.4.2 Change of reference

`trasf_mami`

Transforms a matrix by the rule of $M \cdot A \cdot M^{-1}$ (mami=$M \cdot A \cdot$ Minverse). Contained in `spacelib.c`

Calling sequence:	`trasf_mami (A1, m, A2);`
Prototype:	`void trasf_mami (MAT4 A1, MAT4 m, MAT4 A2);`
Input parameters:	`MAT4 A1, m.`
Output parameters:	`MAT4 A2.`

Performs the matrix operation

$$A_{(r)} = M_{r,s} \cdot A_{(s)} \cdot M_{s,r} = M_{r,s} \cdot A_{(s)} \cdot M_{r,s}^{-1} \tag{3.10}$$

useful in the change of reference of Q, L, W and H matrices. **A1** and **A2** are square 4×4 matrices. **m** is a transformation matrix. `trasf_mami` performs the inverse operation than `trasf_miam`.

See also example 3.14.

See also: `trasf_miam`, `trasf_miamit`, `trasf_mamt`.

Example 3.14. *See sample program* `TRASF_MA.C` *and* `TRASF_MH.C`.

Referring to example 3.12, let's consider the velocity matrix $W_{1,2(1)}$ and the acceleration matrix $H_{1,2(1)}$ both referred to reference frame (1) defined as follows

$$W_{1,2(1)} = \left[\begin{array}{ccc|c} 0 & -1.5 & 0 & 0 \\ 1.5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \end{array}\right]$$

$$H_{1,2(1)} = \left[\begin{array}{ccc|c} -2.25 & -0.9 & 0 & 0 \\ 0.9 & -2.25 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \end{array}\right]$$

Moreover the position of frame (1) referred to frame (0) is expressed by the matrix

$$M_{0,1} = \left[\begin{array}{ccc|c} 1 & 0 & 0 & 0.4 \\ 0 & 1 & 0 & 0.1 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

The velocity and acceleration matrices referred to reference frame (0) are $W_{1,2(0)}$ and $H_{1,2(0)}$ defined as

$$W_{1,2(0)} = \left[\begin{array}{ccc|c} 0 & -1.5 & 0 & 0.150 \\ 1.5 & 0 & 0 & -0.60 \\ 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \end{array}\right]$$

$$H_{1,2(0)} = \left[\begin{array}{ccc|c} -2.25 & -0.9 & 0 & 0.900 \\ 0.9 & -2.25 & 0 & -0.135 \\ 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \end{array}\right]$$

$W_{1,2(0)}$ and $H_{1,2(0)}$ matrices can be built by means of the following statements

```
MAT4 W1={ { 0.  , -1.5, 0., 0.},
          { 1.5,  0.  , 0., 0.},
          { 0.,   0.,   0., 0.},
          { 0.,   0.,   0., 0.} };
MAT4 m01={{ 1., 0., 0., 0.4},
          { 0., 1., 0., 0.1},
          { 0., 0., 1., 0. },
          { 0., 0., 0., 1. }  };
MAT4 W0;
MAT4 H0;
trasf_mami( W1, m01, W0);
trasf_mami( H1, m01, H0);
```

trasf_miam

Transforms a matrix by the rule of $M^{-1} \cdot A \cdot M$ (mami=Minverse $\cdot A \cdot M$). Contained in `spacelib.c`

Calling sequence:	`trasf_miam (A1, m, A2);`
Prototype:	`void trasf_miam (MAT4 A1, MAT4 m, MAT4 A2);`
Input parameters:	`MAT4 A1, m.`
Output parameters:	`MAT4 A2.`

Performs the matrix operation

$$A_{(s)} = M_{s,r} \cdot A_{(r)} \cdot M_{r,s} = M_{r,s}^{-1} \cdot A_{(r)} \cdot M_{r,s} \tag{3.11}$$

for 4×4 matrices which are contra-variant with respect to the row index and co-variant with respect to the column index. `trasf_miam` performs the inverse operation than `trasf_mami`.

See also: `trasf_mami`, `trasf_miamit`, `trasf_mamt`.

trasf_mamt

Transforms a matrix by the rule of $M \cdot A \cdot M^t$ (mamt=$M \cdot A \cdot$ Mtransposed). Contained in `spacelib.c`

Calling sequence:	`void trasf_mamt (M A1, M m, M A2, dim);`
Prototype:	`void trasf_mamt (MAT A1, MAT m, MAT A2, int dim);`
Input parameters:	`MAT A1, m; int dim.`
Output parameters:	`MAT A2.`

Performs the matrix operation

$$A_{k(r)} = M_{r,s} \cdot A_{k(s)} \cdot M^t_{r,s} \tag{3.12}$$

useful in the change of reference of J, Γ and Φ matrices. **A1** and **A2** are square **dim**×**dim** matrices. **m** is a transformation matrix. Matrices **A1**, **A2** and **m** must be recasted using the `M` operator (see § 2.6). `trasf_mamt` performs the inverse operation than `trasf_miamit`.

See also example 3.15.

See also: `trasf_mami`, `trasf_miam`, `trasf_miamit`, `trasf_mamt4`.

trasf_miamit

Transforms a matrix by the rule of $M^{-1} \cdot A \cdot M^{-t}$ (miamit=Minverse $\cdot A \cdot$ Minverse transposed). Contained in `spacelib.c`

Calling sequence:	`void trasf_miamit (A1, m, A2);`
Prototype:	`void trasf_miamit (MAT4 A1, MAT4 m, MAT4 A2);`
Input parameters:	`MAT4 A1, m.`
Output parameters:	`MAT4 A2.`

Performs the matrix operation

$$A_{k(s)} = M_{s,r} \cdot A_{k(r)} \cdot M^t_{s,r} = M^{-1}_{r,s} \cdot A_{k(r)} \cdot M^{-t}_{r,s} \tag{3.13}$$

for 4×4 contra-variant matrices. `trasf_miamit` performs the inverse operation than `trasf_mamt`.

See also example 3.15.

See also: `trasf_mami`, `trasf_miam`, `trasf_mamt`.

Example 3.15. *See sample program* `TR_MAMT.C`.

A system is made up of 3 point bodies whose masses are 5 kg, 1 kg and 2.5 kg. Their position and velocity referred to a reference frame (1) are respectively

$$P_{1(1)} = [2,3,4,1]^t \qquad P_{2(1)} = [0,1,0,1]^t$$

$$P_{3(1)} = [1,3,0,1]^t \qquad \dot{P}_{1(1)} = [1,0,2,0]^t$$

$$\dot{P}_{2(1)} = [4,0.5,1,0]^t \qquad \dot{P}_{3(1)} = [0,0,1,0]^t$$

The angular momentum matrix $\Gamma_{(1)}$ referred to this reference frame can be written as

$$\Gamma_{(1)} = \sum_{i=1}^{3} \left(\dot{P}_i P_i^t - P_i \dot{P}_i^t \right) m_i = \left[\begin{array}{ccc|c} 0 & 19 & -2.5 & 9 \\ -19 & 0 & -38.5 & 0.5 \\ 2.5 & 19 & 0 & 13.5 \\ \hline -9 & -0.5 & -13.5 & 0 \end{array} \right]$$

The position of reference frame (1) respect to another frame, frame (0), is expressed by the following matrix

$$M_{0,1} = \left[\begin{array}{ccc|c} 0 & 1 & 0 & -1.2 \\ -1 & 0 & 0 & 0.5 \\ 0 & 0 & 1 & 4 \\ \hline 0 & 0 & 0 & 1 \end{array} \right]$$

The angular momentum matrix $\Gamma_{(0)}$ in reference frame (0) can be calculated

$$\Gamma_{(0)} = M_{0,1}\,\Gamma_{(1)}\,M_{0,1}^t = \left[\begin{array}{ccc|c} 0 & 29.55 & -52.7 & 9 \\ -29.55 & 0 & -26.75 & 0.5 \\ 52.7 & 26.75 & 0 & 13.5 \\ \hline -9 & -0.5 & -13.5 & 0 \end{array}\right]$$

The previous operations can be executed by means of the following statements

```
MAT4 GAMMA1={{  0. ,19. , -2.5, 9. },
             {-19. , 0. ,-38.5, 0.5},
             {  2.5,38.5,  0. ,13.5},
             { -9. ,-0.5,-13.5, 0. }};
MAT4 m={ { 0.,1.,0., 1.2},
         {-1.,0.,0.,-0.5},
         { 0.,0.,1., 4. },
         { 0.,0.,0., 1.  }};
MAT4 GAMMA0;
trasf_mamt(M GAMMA1, M m, M GAMMA0,4);
```

or using the macro

```
trasf_mamt4(GAMMA1, m, GAMMA0);
```

where m is $M_{0,1}$, GAMMA0 is $\Gamma_{(0)}$, GAMMA1 is $\Gamma_{(1)}$. Opposite, $\Gamma_{(1)}$ can be evaluated from $\Gamma_{(0)}$ by the following statement:

```
trasf_miamit(M GAMMA0, M m, M GAMMA01);
```

trasf_mamt4

Transforms a 4×4 matrix by the rule of $M\cdot A\cdot M^t$ (mamt=$M\cdot A\cdot$ Mtransposed). Macro contained in `spacelib.h`

Calling sequence:	`trasf_mamt4 (A1, m, A2);`
Prototype:	`void trasf_mamt4 (MAT4 A1, MAT4 m, MAT4 A2);`
Input parameters:	`MAT4 A1, m.`
Output parameters:	`MAT4 A2.`

This macro transforms by the equation (3.12). **A1** and **A2** are 4×4 matrices. Equivalent to `trasf_mamt(M A1, M m, M A2, 4);`

See also: `trasf_mamt`.

3.4.3 General operations

invers

Inverse of a position matrix. Contained in `spacelib.c`

Calling sequence:	`invers (m, mi);`
Prototype:	`void invers (MAT4 m, MAT4 mi);`
Input parameters:	`MAT4 m.`
Output parameters:	`MAT4 mi.`

Evaluates the inverse **mi** of a 4×4 position (transformation) matrix **m** using the equation:

$$M_{0,1} = \left[\begin{array}{ccc|c} & R_{0,1} & & T_{0,1} \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \qquad M_{0,1}^{-1} = M_{1,0} = \left[\begin{array}{ccc|c} & R_{0,1}^t & & -R_{0,1}^t T_{0,1} \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{3.14}$$

This function <u>works only</u> for 4×4 transformation matrices.

mtov

matrix to vector. Contained in `spacelib.c`

Calling sequence:	`mtov (M A, dim, v);`
Prototype:	`void mtov (MAT A, int dim, VECTOR v);`
Input parameters:	`MAT A; int dim.`
Output parameters:	`VECTOR v.`

Extracts vector $\mathbf{v}$[1] 3×1 from the upper-left 3×3 skew-symmetric submatrix of a **dim**×**dim** matrix **A**. **dim** must be greater or equal to **3** (**dim** ≥ **3**). Matrix **A** must be recasted using the `M` operator (see § 2.6). `mtov` performs the inverse operation than `vtom`.

Example:

The function `mtov(M A, 4, v)` applied to the skew-symmetric matrix A defined as

$$A = \left[\begin{array}{ccc|c} 0 & -c & b & d \\ c & 0 & -a & e \\ -b & a & 0 & f \\ \hline g & h & i & j \end{array}\right]$$

give the resulting vector v defined as $v = [a, b, c,]^t$

See also: `mtov3`, `mtov4`.

mtov3

matrix 3×3 to vector. Macro contained in `spacelib.h`

Calling sequence:	`mtov3 (A, v);`
Prototype:	`void mtov3 (MAT3 A, VECTOR v);`
Input parameters:	`MAT3 A.`
Output parameters:	`VECTOR v.`

Transforms a 3×3 skew-symmetric matrix into a vector with 3 components.
Equivalent to `mtov(M A, v, 3);`

See also: `mtov`.

mtov4

matrix 4×4 to vector. Macro contained in `spacelib.h`

Calling sequence:	`mtov4 (A, v);`
Prototype:	`void mtov4 (MAT4 A, VECTOR v)`
Input parameters:	`MAT4 A.`
Output parameters:	`VECTOR v.`

Transforms the 3×3 upper left skew-symmetric of a 4×4 matrix **A** into a vector **v** with 3 components.
Equivalent to `mtov(M A, v, 4);`

See also: `mtov`.

vtom

vector to matrix. Contained in `spacelib.c`

Calling sequence:	`void vtom (v, M A, dim);`
Prototype:	`void vtom (VECTOR v, MAT A, int dim);`
Input parameters:	`VECTOR v; int dim.`
Output parameters:	`MAT A.`

[1] A vector $\overrightarrow{v}$ could be represented in the following forms [3], [4], [2]:

$$v = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} \qquad \underline{v} = \begin{bmatrix} 0 & -v_z & v_y \\ v_z & 0 & -v_x \\ -v_y & v_x & 0 \end{bmatrix}$$

Creates a 3×3 skew-symmetric submatrix from vector **v**[2] and stores it in the upper-left part of the **dim**×**dim** matrix **A**. **dim** must be greater or equal to **3** (**dim** ≥ **3**). Matrix **A** must be recasted using the `M` operator (see § 2.6). `vtom` performs the inverse operation than `mtov`.

Example: The statement

```
mtov(M A, 4, v);
```

applied to the vector v defined as $v = [a, b, c,]^t$ gives the resulting skew-symmetric matrix A defined as follows

$$A = \left[\begin{array}{ccc|c} 0 & -c & b & ? \\ c & 0 & -a & ? \\ -b & a & 0 & ? \\ \hline ? & ? & ? & ? \end{array}\right]$$

where the character '?' means that the value of this element is not affected by the function.

See also: `vtom3`, `vtom4`, `vtom`.

vtom3

vector to matrix 3×3. Macro contained in `spacelib.h`

Calling sequence:	`vtom3 (v, A);`
Prototype:	`void vtom3 (VECTOR v, MAT3 A);`
Input parameters:	`VECTOR v.`
Output parameters:	`MAT3 A.`

Transforms a vector with 3 components into a 3×3 skew-symmetric matrix.
Equivalent to `vtom(v, M A, 3);`

See also: `vtom`.

vtom4

vector to matrix 4×4. Macro contained in `spacelib.h`

Calling sequence:	`vtom4 (v, A);`
Prototype:	`void vtom4 (VECTOR v, MAT4 A);`
Input parameters:	`VECTOR v.`
Output parameters:	`MAT4 A.`

Transforms a vector with 3 components into a 3×3 skew-symmetric submatrix and stores it in the upper-left part of a 4×4 matrix.
Equivalent to `vtom(v, M A, 4);`

See also: `vtom`.

skew

skew operator. Contained in `spacelib.c`

Calling sequence:	`skew (M A, M B, M C, dim);`
Prototype:	`void skew (MAT A, MAT B, MAT C, int dim);`
Input parameters:	`MAT A, B; int dim.`
Output parameters:	`MAT C.`

Performs the matrix operation

$$C = skew\{A \cdot B\} = A \cdot B - B^t \cdot A^t \tag{3.15}$$

[2]A vector $\overrightarrow{v}$ could be represented in the following forms [3], [4], [2]:

$$v = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} \qquad \underline{v} = \begin{bmatrix} 0 & -v_z & v_y \\ v_z & 0 & -v_x \\ -v_y & v_x & 0 \end{bmatrix}$$

applicable to square **dim**×**dim** matrices. Matrices **A**, **B**, **C** must be recasted using the `M` operator (see § 2.6).

See also: `skew4`.

`skew4`

skew operator for 4×4 matrices. Macro contained in `spacelib.h`

Calling sequence:	`void skew4 (A, B, C);`
Prototype:	`skew4 (MAT4 A, MAT4 B, MAT4 C);`
Input parameters:	`MAT4 A, B.`
Output parameters:	`MAT4 C.`

Performs the operation (3.15) for 4×4 matrices.
Equivalent to `skew(M A, M B, M C, 4);`

See also: `skew`.

`trac_ljlt4`

Trace of $L_1 \cdot J \cdot L_2^t$. Contained in `spacelib.c`

Calling sequence:	`tr = trac_ljlt4 (L1, J, L2);`
Prototype:	`real trac_ljlt4 (MAT4 L1, MAT4 J, MAT4 L2);`
Return value:	`real tr.`
Input parameters:	`MAT4 L1, J, L2.`

Returns the trace[3] **tr** of the matrix product $\mathbf{L1} \cdot \mathbf{J} \cdot \mathbf{L2}^t$. Applicable to 4×4 matrices.

3.5 Conversion between Cardan (or Euler) angles and matrices

There are several types of coordinates which can express the relative angular position of two moving bodies in a 3D space; generally two frames are fixed on the two bodies. In the "*neutral*" position, these frames are parallel to each other. One of the two reference frames is called *absolute*, while the other *moving*. Their relative angular position is expressed by the position of the moving frame with respect to the fixed one. The relative orientation of two frames can be imagined as obtained from the neutral position by three subsequent rotations α, β, γ of the moving frame around three axes: i, j, k. Rotations are generally performed about the axes of the fixed or of the moving frame.

It is possible to show that a group of three rotations α, β, γ around axes i, j, k of the fixed frame are equivalent to a sequence of rotations γ, β, α, around axes k, j, i (reverse order) of the moving frame. As the rotation axes are given, the angular position can be expressed by the three rotation angle values. Rotations about fixed axes multiply to left, while about moving axes to right.

Value of i, j, k for rotations around axes of fixed frame				
Systems	***Cardan (Tait-Brian)*** $i \neq j \neq k \neq i$		***Euler*** $i = k \neq j$	
Cyclic	x, y, z	*Cardan angles*	x, y, x	
	y, z, x		y, z, y	
	z, x, y		z, x, z	*Euler angles*
Anti-cyclic	z, y, x	*Nautic angles*	z, y, z	
	x, z, y		x, z, x	
	y, x, z		y, x, y	

Table 3.2: Cardan angles convention

[3]The trace of a square matrix is the sum of its diagonal elements. If X is a column matrix it yields $Trace(XX^t) = X^tX$.

3.5.1 Angular position

`cardantor`

Cardan (or Euler) angles to rotation matrix. Contained in `spaceli3.c`

Calling sequence:	`rv = int cardantor (q, i, j, k, M A, dim);`
Prototype:	`int cardantor (real *q, int i, int j, int k, MAT A, int dim);`
Return value:	`int rv.`
Input parameters:	`real q[3]; int i, j, k, dim.`
Output parameters:	`MAT A.`

Builds a rotation matrix starting from the *Cardan* or *Euler* angles and stores it in the 3×3 upper left submatrix of a **dim**×**dim** matrix **A**. **dim** must be greater or equal to **3** (**dim** ≥ **3**). The parameters **i**, **j**, **k** specify the sequence of the rotation axes (their value must be the constant `X`, `Y` or `Z`. See § 2.4.3). **j** must be different from **i** and **k**, **k** could be equal to **i** (see also table 3.2). **q** is a 3 element vector containing the 1st, 2nd and 3rd angle. The function returns the value **rv** that can be either `OK` or `NOTOK` (see § 2.4.3) in order to specify the correct outcome. Matrix **A** must be recasted using the `M` operator (see § 2.6). `cardantor` performs the inverse operation than `rtocardan`.

See also: `cardantoM`, `cardantor3`, `cardantor4`.

`cardantor3`

Cardan (or Euler) angles to rotation matrix 3×3. Macro contained in `spacelib.h`

Calling sequence:	`rv = cardantor3 (q, i, j, k, R);`
Prototype:	`cardantor3 (real *q, int i, int j, int k, MAT3 R);`
Return value:	`int rv.`
Input parameters:	`real q[3]; int i, j, k.`
Output parameters:	`MAT3 R.`

Builds a rotation matrix starting from the *Cardan* or *Euler* angles and stores it in a 3×3 rotation matrix **R**.
Equivalent to `cardantor(q, i, j, k, M R, 3);`

See also: `cardantor`.

`cardantor4`

Cardan (or Euler) angles to rotation matrix 4×4. Macro contained in `spacelib.h`

Calling sequence:	`rv = cardantor4 (q, i, j, k, m);`
Prototype:	`cardantor4 (real *q, int i, int j, int k, MAT4 m);`
Return value:	`int rv.`
Input parameters:	`real q[3]; int i, j, k.`
Output parameters:	`MAT4 m.`

Builds a rotation matrix starting from the *Cardan* or *Euler* angles and stores it in a 3×3 upper-left corner of a 4×4 position matrix **m**.
Equivalent to `cardantor(q, i, j, k, M m, 4);`

See also: `cardantor`.

`rtocardan`

Rotation matrix to Cardan (or Euler) angles. Contained in `spaceli3.c`

Calling sequence:	`rv = int rtocardan (M R, dim, i, j, k, q1, q2);`
Prototype:	`int rtocardan (MAT R, int dim, int i, int j, int k, real *q1, real *q2);`
Return value:	`int rv.`
Input parameters:	`MAT R; int dim, i, j, k.`
Output parameters:	`real q1[3], q2[3].`

Extracts the *Cardan* (or the *Euler*) angles from a rotation matrix. The rotation matrix must be stored in the the 3×3 upper-left submatrix of a **dim×dim** matrix. **dim** must be greater or equal to **3** (**dim** $\geq$ **3**). The parameters **i**, **j**, **k** specify the sequence of the rotation axes (their value must be the constant `X`, `Y` or `Z`. See § 2.4.3). **j** must be different from **i** and **k**, **k** could be equal to **i** (see also table 3.2). The two solutions are stored in the three-element vectors **q1** and **q2**. The function returns the value **rv** that can be either `OK` or `NOTOK` (see § 2.4.3) in order to specify the correct outcome. Matrix **R** must be recasted using the `M` operator (see § 2.6). `rtocardan` performs the inverse operation than `cardantor`.

See also example 3.16.

See also: `Mtocardan`, `rtocardan3`, `rtocardan4`.

Example 3.16. *See sample program* `RTOCARDA.C`.

The angular position of a generic reference frame (i) referred to frame $(i-1)$ is expressed by the matrix

$$M_{i-1,i} = \begin{bmatrix} 0.840 & -0.395 & -0.371 \\ -0.415 & -0.029 & -0.909 \\ 0.348 & 0.918 & -0.189 \end{bmatrix}$$

The rotation sequence is made up of a rotation about axis y, a rotation about axis x and a rotation about axis z (anti-cyclic *cardanic* convention. See table 3.2). The Cardan angles which perform the rotation from frame $(i-1)$ to frame (i) are calculated by the statements

```
MAT3 R={ { 0.840, -0.395, -0.371},
         {-0.415, -0.029, -0.909},
         { 0.348,  0.918, -0.189} };
VECTOR q1,q2;
rtocardan(M R, 3, Y, X, Z, q1, q2);
```

The two solutions are $q1 = [-2.042, 1.141, -1.641]^t$ and $q2 = [1.100, 2.001, 1.501]^t$

rtocardan3

Rotation matrix to Cardan (or Euler) angles for 3×3 matrices. Macro contained in `spacelib.h`

Calling sequence:	`rv = rtocardan3 (R, i, j, k, q1, q2);`
Prototype:	`rtocardan3 (MAT3 R, int i, int j, int k, real *q1, real *q2);`
Return value:	`int rv.`
Input parameters:	`MAT3 R; int i, j, k.`
Output parameters:	`real q1[3], q2[3].`

Extracts the *Cardan* (or the *Euler*) angles from a 3×3 rotation matrix.
Equivalent to `rtocardan(M R, 3, i, j, k, q1, q2);`

See also: `rtocardan`.

rtocardan4

Rotation matrix to Cardan (or Euler) angles for 4×4 matrices. Macro contained in `spacelib.h`

Calling sequence:	`rv = rtocardan4 (m, i, j, k, q1, q2);`
Prototype:	`rtocardan4 (MAT4 m, int i, int j, int k, real *q1, real *q2);`
Return value:	`int rv.`
Input parameters:	`MAT4 m; int i, j, k.`
Output parameters:	`real q1[3], q2[3].`

Extracts the *Cardan* (or the *Euler*) angles from a 3×3 rotation matrix stored in the upper-left 3×3 corner of a 4×4 position matrix **m**.
Equivalent to `rtocardan(M m, 4, i, j, k, q1, q2);`

See also: `rtocardan`.

cardantoM

Cardan angles to position matrix. Contained in `spaceli3.c`

Calling sequence:	`void cardantoM (q, i, j, k, O, m);`
Prototype:	`void cardantoM (real *q, int i, int j, int k, POINT O, MAT4 m);`
Input parameters:	`real q[3]; int i, j, k.`
Output parameters:	`POINT O; MAT4 m.`

Builds the position matrix **m** of a frame whose origin is in point **O** and whose orientation is specified by an *Euler/Cardanic* convention. The parameters **i**, **j**, **k** specify the sequence of the rotation axes (their value must be the constant `X`, `Y` or `Z`. See § 2.4.3). **j** must be different from **i** and **k**, **k** could be equal to **i** (see also table 3.2). **q** : 3 element vector containing the 1st, 2nd and 3rd rotation angle.

See also example 3.17.

See also: `cardantor`, `Mtocardan`, `cardtoM`, `eultoM`, `nauttoM`.

Example 3.17. *See sample program* `CARDAM.C`.

The position matrix m of a frame whose origin is in the point $O = [100, 200, 300]^t$ which has q defined by a rotation of 1 rad about axis x, a second rotation of 2 rad about axis z and a third rotation of 1.5 rad about axis y is built by the following statements

```
POINT O={100.,200.,300.,1.};
MAT4 m;
VECTOR q={1.,2.,1.5};
cardantoM(q,X,Z,Y,O,m);
```

The resulting matrix is

$$m = \left[\begin{array}{ccc|c} -0.029 & -0.909 & -0.415 & 100. \\ 0.874 & -0.225 & 0.431 & 200. \\ -0.485 & -0.350 & 0.801 & 300. \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

cardtoM

Cardan angles to position matrix. Macro contained in `spacelib.h`

Calling sequence:	`cardtoM (q, O, m);`
Prototype:	`void cardtoM (real *q, POINT O, MAT4 m);`
Input parameters:	`real q[3]; POINT O.`
Output parameters:	`MAT4 m.`

Builds the position matrix **m** whose origin is in point **O** and whose angular orientation is specified by the *Tait-Brian* angles. The rotation sequence is X, Y, Z (see also § 2.4.3 and table 3.2).
Equivalent to `cardantoM(q, X, Y, Z, O, m);`

See also: `cardantoM`.

eultoM

Cardan angles to position matrix. Macro contained in `spacelib.h`

Calling sequence:	`eultoM (q, O, m);`
Prototype:	`void eultoM (real *q, POINT O, MAT4 m);`
Input parameters:	`real q[3]; POINT O.`
Output parameters:	`MAT4 m.`

Builds the position matrix **m** whose origin is in point **O** and whose angular orientation is specified by the *Euler* angles. The rotation sequence is Z, X, Z (see also § 2.4.3 and table 3.2).
Equivalent to `cardantoM(q, Z, X, Z, O, m);`

See also: `cardantoM`.

nauttoM

Cardan angles to position matrix. Macro contained in `spacelib.h`

Calling sequence:	`nauttoM (q, O, m);`
Prototype:	`void nauttoM (real *q, POINT O, MAT4 m);`
Input parameters:	`real q [3]; POINT O.`
Output parameters:	`MAT4 m.`

Builds the position matrix **m** whose origin is in point **O** and whose angular orientation is specified by the *Nautical* angles. The rotation sequence is Z, Y, X (see also § 2.4.3 and table 3.2).
Equivalent to `cardantoM(q, Z, Y, X, O, m);`

See also: `cardantoM`.

Mtocardan

Position matrix to Cardan angles. Contained in `spaceli3.c`

Calling sequence:	`rv = Mtocardan (m, i, j, k, q1, q2);`
Prototype:	`int Mtocardan (MAT4 m, int i, int j, int k, real *q1, real *q2);`
Return value:	`int rv.`
Input parameters:	`MAT4 m; int i, j, k.`
Output parameters:	`real q1[3], q2[3].`

Builds the *Euler/Cardan* angles which specify the position of a frame whose position matrix is **m**. Both solutions are evaluated. The parameters **i**, **j**, **k** specify the sequence of the rotation axes (their value must be the constant `X`, `Y` or `Z`. See § 2.4.3). **j** must be different from **i** and **k**, **k** could be equal to **i** (see also table 3.2). **q1** and **q2** are a 3 element vectors containing the 1^{st}, 2^{nd} and 3^{rd} rotation angle of the two solutions. The function returns the value **rv** that can be either `OK` or `NOTOK` (see § 2.4.3) in order to specify the correct outcome.

See also: `rtocardan`, `cardantoM`, `Mtocard`, `Mtoeul`, `Mtonaut`.

Mtocard

Position matrix to Cardan angles. Macro contained in `spacelib.h`

Calling sequence:	`rv = Mtocard (m, q1, q2);`
Prototype:	`int Mtocard (MAT4 m, real *q1, real *q2);`
Return value:	`int rv.`
Input parameters:	`MAT4 m.`
Output parameters:	`real q1[3], q2[3].`

Builds the *Tait-Brian* angles of a frame whose position matrix is **m**. The rotation sequence is X, Y, Z (see also § 2.4.3 and table 3.2).
Equivalent to `Mtocardan(m, X, Y, Z, q1, q2);`

See also: `Mtocardan`.

Mtoeul

Position matrix to Cardan angles. Macro contained in `spacelib.h`

Calling sequence:	`rv = Mtoeul (m, q1, q2);`
Prototype:	`Mtoeul (MAT4 m, real *q1, real *q2);`
Return value:	`int rv.`
Input parameters:	`MAT4 m.`
Output parameters:	`real q1[3], q2[3].`

Builds the *Euler* angles of a frame whose position matrix is **m**. The rotation sequence is Z, X, Z (see also § 2.4.3 and table 3.2).
Equivalent to `Mtocardan(m, Z, X, Z, q1, q2);`

See also: `Mtocardan`.

Mtonaut

Position matrix to Cardan angles. Macro contained in `spacelib.h`

Calling sequence:	`rv = Mtonaut (m, q1, q2);`
Prototype:	`Mtonaut (MAT4 m, real *q1, real *q2);`
Return value:	`int rv.`
Input parameters:	`MAT4 m.`
Output parameters:	`real q1[3], q2[3].`

Builds the *Nautical* angles of a frame whose position matrix is **m**. The rotation sequence is Z, Y, X (see also § 2.4.3 and table 3.2).
Equivalent to `Mtocardan(m, Z, Y, X, q1, q2);`

See also: `Mtocardan`.

3.5.2 Angular velocity and acceleration

cardantoW

Cardan angles to velocity matrix. Contained in `spaceli3.c`

Calling sequence:	`cardantoW (q, qp, i, j, k, O, W);`
Prototype:	`void cardantoW (real *q, real *qp, int i, int j, int k, POINT O, MAT4 W);`
Input parameters:	`real q[3], qp[3]; int i, j, k; POINT O.`
Output parameters:	`MAT4 W.`

Builds the velocity matrix **W** of a frame whose origin is **O** and whose orientation is specified by an *Euler/Cardan* convention. The parameters **i**, **j**, **k** specify the sequence of the rotation axes (their value must be the constant `X`, `Y` or `Z`. See § 2.4.3). **j** must be different from **i** and **k**, **k** could be equal to **i** (see also table 3.2). **q** is a 3 element vector containing the 1st, 2nd and 3rd angle. **qp** is a 3 element vector containing the time derivative of **q**. `cardantoW` performs the inverse operation than `Wtocardan`.

See also example 3.18.

See also: `cardantoH`, `Wtocardan`, `cardtoW`, `eultoW`, `nauttoW`.

Example 3.18. *See sample program* `CARDAW.C`.

Let's consider a moving frame whose origin is in the point $O = [\,50,\ 10,\ 100\,]^t$. The rotation sequence is made up by a rotation of 1 rad around axis x, one of 2.5 rad about axis z and a rotation of 0.9 rad about axis y. The time derivative qp of q is filled with the values $\{\,0.2,\ 4.,\ 1.\,\}$ rad/s. The 4×4 velocity matrix W is built by the following statements

```
MAT4 W;
POINT O={50.,10.,100.,1.};
VECTOR q={0.1,0.5,0.9};
VECTOR qp={0.2,0.4,0.1};
cardantoW(q,qp,X,Z,Y,O,W);
```

The resulting matrix W is

$$W = \left[\begin{array}{ccc|c} 0 & -0.407 & 0.047 & -0.671 \\ 0.407 & 0 & -0.152 & -5.132 \\ -0.047 & 0.0152 & 0 & 0.849 \\ \hline 0 & 0 & 0 & 0 \end{array}\right]$$

cardtoW

Cardan angles to velocity matrix. Macro contained in `spacelib.h`

Calling sequence:	`cardtoW (q, qp, O, W);`
Prototype:	`void cardtoW (real *q, real *qp, POINT O, MAT4 W);`
Input parameters:	`real q[3], qp[3]; POINT O.`
Output parameters:	`MAT4 W;`

Builds the velocity matrix **W** of a frame whose origin is **O** and whose angular orientation is specified by the *Tait-Brian* angles and their first time derivative. The rotation sequence is X, Y, Z (see also § 2.4.3 and table 3.2).
Equivalent to `cardantoW(q, qp, X, Y, Z, O, W);`

See also: `cardantoW`

eultoW

Cardan angles to velocity matrix. Macro contained in `spacelib.h`

Calling sequence:	`eultoW (q, qp, O, W);`
Prototype:	`void eultoW (real *q, real *qp, POINT O, MAT4 W);`
Input parameters:	`real q[3], qp[3]; POINT O.`
Output parameters:	`MAT4 W;`

Builds the velocity matrix **W** of a frame whose origin is **O** and whose angular orientation is specified by the *Euler* angles and their first time derivative. The rotation sequence is Z, X, Z (see also § 2.4.3 and table 3.2).
Equivalent to `cardantoW(q, qp, Z, X, Z, O, W);`

See also: `cardantoW`

nauttoW

Cardan angles to velocity matrix. Macro contained in `spacelib.h`

Calling sequence:	`nauttoW (q, qp, O, W);`
Prototype:	`void nauttoW (real *q, real *qp, POINT O, MAT4 W);`
Input parameters:	`real q[3], qp[3]; POINT O.`
Output parameters:	`MAT4 W;`

Builds the velocity matrix **W** of a frame whose origin is **O** and whose angular orientation is specified by the *Nautical* angles and their first time derivative. The rotation sequence is Z, Y, X (see also § 2.4.3 and table 3.2).
Equivalent to `cardantoW(q, qp, Z, Y, X, O, W);`

See also: `cardantoW`

Wtocardan

Velocity matrix to Cardan angles. Contained in `spaceli4.c`

Calling sequence:	`rv = Wtocardan (m, W, i, j, k, q1, q2, qp1, qp2);`
Prototype:	`int Wtocardan (MAT4 m, MAT4 W, int i, int j, int k, real *q1, real *q2, real *qp1, real *qp2);`
Return value:	`int rv.`
Input parameters:	`MAT4 m, W; int i, j, k.`
Output parameters:	`real q1[3], q2[3], qp1[3], qp2[3].`

Builds the *Euler*/*Cardan* angles, first and second time derivative of a frame. It uses the velocity matrix **W** and the position matrix **m**. Both solutions are evaluated. The parameters **i**, **j**, **k** specify the sequence of the rotation axes (their value must be the constant `X`, `Y` or `Z`. See § 2.4.3). **j** must be different from **i** and **k**, **k** could be equal to **i** (see also table 3.2). **q1** and **q2** are 3 element vectors containing the two angles set. **qp1** and **qp2** are 3 element vectors containing the time derivative of **q1** and **q2** respectively. The function returns the value **rv** that can be either `OK` or `NOTOK` (see § 2.4.3) in order to specify the correct outcome.

See also: `Htocardan`, `cardantoW`, `Wtocard`, `Wtoeul`, `Wtonaut`.

`Wtocard`

Velocity matrix to Cardan angles. Macro contained in `spacelib.h`

Calling sequence:	`rv = Wtocard (m, W, q1, q2, qp1, qp2);`
Prototype:	`int Wtocard (MAT4 m, MAT4 W, real *q1, real *q2, real *qp1, real *qp2);`
Return value:	`int rv.`
Input parameters:	`MAT4 m, W.`
Output parameters:	`real q1[3], q2[3], qp1[3], qp2[3].`

Builds the *Tait-Brian* angles of a frame whose position matrix is **m** and their first time derivative. The rotation sequence is X, Y, Z (see also § 2.4.3 and table 3.2).
Equivalent to `Wtocardan(m, W, X, Y, Z, q1, q2, qp1, qp2);`

See also: `Wtocardan`.

`Wtoeul`

Velocity matrix to Cardan angles. Macro contained in `spacelib.h`

Calling sequence:	`rv = Wtoeul (m, W, q1, q2, qp1, qp2);`
Prototype:	`int Wtoeul (MAT4 m, MAT4 W, real *q1, real *q2, real *qp1, real *qp2);`
Return value:	`int rv.`
Input parameters:	`MAT4 m, W.`
Output parameters:	`real q1[3], q2[3], qp1[3], qp2[3].`

Builds the *Euler* angles of a frame whose position matrix is **m** and their first time derivative. The rotation sequence is Z, X, Z (see also § 2.4.3 and table 3.2).
Equivalent to `Wtocardan(m, W, Z, X, Z, q1, q2, qp1, qp2);`

See also: `Wtocardan`.

`Wtonaut`

Velocity matrix to Cardan angles. Macro contained in `spacelib.h`

Calling sequence:	`rv = Wtonaut (m, W, q1, q2, qp1, qp2);`
Prototype:	`int Wtonaut (MAT4 m, MAT4 W, real *q1, real *q2, real *qp1, real *qp2);`
Return value:	`int rv.`
Input parameters:	`MAT4 m, W.`
Output parameters:	`real q1[3], q2[3], qp1[3], qp2[3].`

Builds the *Nautical* angles of a frame whose position matrix is **m** and their first time derivative. The rotation sequence is Z, Y, X (see also § 2.4.3 and table 3.2).
Equivalent to `Wtocardan(m, W, Z, Y, X, q1, q2, qp1, qp2);`

See also: `Wtocardan`.

`cardanto_OMEGA`

Cardan angles to angular velocity. Contained in `spaceli3.c`

Calling sequence:	`cardanto_OMEGA (q, qp, i, j, k, &omega);`
Prototype:	`void cardanto_OMEGA (real *q, real *qp, int i, int j, int k, real *omega);`
Input parameters:	`real q[3], qp[3]; int i, j, k.`
Output parameters:	`real omega.`

Evaluates the angular velocity of a moving frame from the three *Cardan* (or *Euler*) angles **q** and their time derivative **qp**. The parameters **i**, **j**, **k** specify the sequence of the rotation axes (their value must be the constant `X`, `Y` or `Z`. See § 2.4.3). **j** must be different from **i** and **k**, **k** could be equal to **i** (see also table 3.2). **q** is a 3 element vector containing 1st, 2nd and 3rd angle. **qp** is the time derivative of **q**. **omega** is a 3 element vector containing the angular velocity.

See also example 3.19.

See also: cardanto_omega, cardanto_OMEGAPTO.

Example 3.19. *See sample program* CARDA-OM.C.

Consider a moving frame whose origin is in the point $O = [\,200,\,40,\,300\,]^t$. It has variable q defined by a rotation of 10 rad around axis z, a rotation of 5 rad around axis y and a rotation of 12 rad around axis x. The first time derivative of q is filled with the values $[\,0, 2, 1\,]$ rad/s. The angular velocity ω is evaluated by the following statements

```
VECTOR q={10.,5.,12.};
VECTOR qp={0.,2.,1.};
VECTOR omega;
cardanto_OMEGA(q,qp,Z,Y,X,omega);
```

The resulting vector is

$$omega = [\,0.850,\,-1.832,\,0.959\,]^t$$

cardanto_omega

Cardan angles to angular velocity matrix. Contained in spaceli3.c

Calling sequence:	`cardanto_omega (q, qp, i, j, k, M A, dim);`
Prototype:	`void cardanto_omega (real *q, *qp, int i, int j, int k, MAT A, int dim);`
Input parameters:	`real q[3], qp[3]; int i, j, k, dim.`
Output parameters:	`MAT A.`

Evaluates the angular velocity matrix Ω. Equivalent to `cardanto_OMEGA(q, qp, i, j, k, A);` but stores the angular velocity in the upper-left 3×3 submatrix of a **dim**×**dim** matrix **A**. The parameters **i**, **j**, **k** specify the sequence of the rotation axes (their value must be the constant X, Y or Z. See § 2.4.3). **j** must be different from **i** and **k**, **k** could be equal to **i** (see also table 3.2). Matrix **A** must be recasted using the M operator (see § 2.6).

See also example 3.20.

See also: cardanto_OMEGAPTO, cardanto_omega3, cardanto_omega4.

Example 3.20. *See sample program* CARDA_OM.C.

There are only a few differences between this example and example 3.21. The angular velocity omega is no more stored in a `real *` type variable because now it is stored in the 3×3 skew-symmetric upper-left submatrix of a MAT type variable A. So, we have the statements

```
MAT4 A;
POINT O={200.,40.,300.,1.};
VECTOR q={10.,5.,12.};
VECTOR qp={0.,2.,1.};
clear4(A);
cardanto_omega(q,qp,Z,Y,X,M A,4);
```

The resulting matrix is

$$A = \left[\begin{array}{ccc|c} 0 & -0.959 & -1.832 & 0 \\ 0.959 & 0 & -0.850 & 0 \\ 1.832 & 0.850 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \end{array}\right]$$

cardanto_omega3

Cardan angles to angular velocity for 3×3 matrix. Macro contained in `spacelib.h`

Calling sequence:	`cardanto_omega3 (real q, qp, i, j, k, OMEGA);`
Prototype:	`void cardanto_omega3 (real *q, real *qp, int i, int j, int k, MAT3 OMEGA);`
Input parameters:	`real q[3], qp[3]; int i, j, k.`
Output parameters:	`MAT3 OMEGA.`

Evaluates the angular velocity matrix `OMEGA`. Similar to `cardanto_OMEGA` but stores the angular velocity in a 3×3 matrix.
Equivalent to `cardanto_omega(q, qp, i, j, k, M OMEGA, 3);`

See also: `cardanto_omega`.

cardanto_omega4

Cardan angles to angular velocity for 4×4 matrix. Macro contained in `spacelib.h`

Calling sequence:	`cardanto_omega4 (real q, qp, i, j, k, OMEGA);`
Prototype:	`void cardanto_omega3 (real *q, real *qp, int i, int j, int k, MAT4 OMEGA);`
Input parameters:	`real q[3], qp[3]; int i, j, k.`
Output parameters:	`MAT4 OMEGA.`

Evaluates the angular velocity matrix `OMEGA`. Similar to `cardanto_OMEGA` but stores the angular velocity in the 3×3 upper-left submatrix of a 4×4 matrix **A**.
Equivalent to `cardanto_omega(q, qp, i, j, k, M OMEGA, 4);`

See also: `cardanto_omega`.

cardantoH

Cardan angles to acceleration matrix. Contained in `spaceli3.c`

Calling sequence:	`cardantoH (q, qp, qpp, i, j, k, O, H);`
Prototype:	`void cardantoH (real *q, real *qp, real *qpp, int i, int j, int k, POINT O, MAT4 H);`
Input parameters:	`real q[3], qp[3], qpp[3]; int i, j, k; POINT O.`
Output parameters:	`MAT4 H.`

Builds the acceleration matrix **H** of a frame whose origin is **O** and whose orientation is specified by a *Euler/Cardanic* convention. The parameters **i**, **j**, **k** specify the sequence of the rotation axes (their value must be the constant `X`, `Y` or `Z`. See § 2.4.3). **j** must be different from **i** and **k**, **k** could be equal to **i** (see also table 3.2). **q** is a 3 element vector containing the 1st, 2nd and 3rd angle. **qp** is a 3 element vector containing the time derivative of **q**. **qpp** is a 3 element vector containing the 2*nd* time derivative of **q**.

See also example 3.21.

See also: `cardantoW`, `Htocardan`, `cardtoH`, `eultoH`, `nauttoH`.

Example 3.21. *See sample program* `CARDAH.C`.

Let's consider example 3.18. The second time derivative of `q`, which is `qpp`, is filled with the values $[0.5, 1.2, 0.3]$ rad/s^2. The acceleration matrix H of the frame is built by the following statements

```
MAT4 H;
POINT O={50.,10.,100.,1.};
VECTOR q={0.1,0.5,0.9};
VECTOR qp={0.2,0.4,0.1};
VECTOR qpp={0.5,1.2,0.3};
cardantoH(q,qp,qpp,X,Z,Y,O,H);
```

The resulting matrix H is

$$H = \left[\begin{array}{ccc|c} -0.168 & -1.221 & 0.104 & 10.234 \\ 1.235 & -0.189 & -0.302 & -29.688 \\ 0.020 & 0.340 & -0.025 & -1.873 \\ \hline 0 & 0 & 0 & 0 \end{array}\right]$$

When O is put equal to $[\,0, 0, 0, 1\,]$ this example gives the same resulting matrix of function cardanto_G (see example 3.22).

cardtoH

Cardan angles to acceleration matrix.	Macro contained in `spacelib.h`
Calling sequence:	`cardtoH (q1, q2, qp1, qp2, m, W, H);`
Prototype:	`void cardtoH (real *q1, real *q2, real *qp1, real *qp2, MAT4 m, MAT4 W, MAT4 H);`
Input parameters:	`real q1[3], q2[3], qp1[3], qp2[3].`
Output parameters:	`MAT4 m, W, H.`

Builds the acceleration matrix **H** of a frame whose origin is **O** and whose angular orientation is specified by the *Tait-Brian* angles, their first and second time derivative. The rotation sequence is X, Y, Z (see also § 2.4.3 and table 3.2).
Equivalent to `cardantoH(q, qp, qpp, X, Y, Z, O, H);`

See also: `cardantoH`.

eultoH

Cardan angles to acceleration matrix.	Macro contained in `spacelib.h`
Calling sequence:	`eultoH (q1, q2, qp1, qp, m, W, H);`
Prototype:	`void eultoH (real *q1, real *q2, real *qp1, real *qp2, MAT4 m, MAT4 W, MAT4 H);`
Input parameters:	`real q1[3], q2[3], qp1[3], qp2[3].`
Output parameters:	`MAT4 m, W, H.`

Builds the acceleration matrix **H** of a frame whose origin is **O** and whose angular orientation is specified by the *Euler* angles, their first and second time derivative. The rotation sequence is Z, X, Z (see also § 2.4.3 and table 3.2).
Equivalent to `cardantoH(q, qp, qpp, Z, X, Z, O, H);`

See also: `cardantoH`.

nauttoH

Cardan angles to acceleration matrix.	Macro contained in `spacelib.h`
Calling sequence:	`nauttoH (q1, q2, qp1, qp2, m, W, H);`
Prototype:	`void nauttoH (real *q1, real *q2, real *qp1, real *qp2, MAT4 m, MAT4 W, MAT4 H);`
Input parameters:	`real q1[3], q2[3], qp1[3], qp2[3].`
Output parameters:	`MAT4 m, W, H.`

Builds the acceleration matrix **H** of a frame whose origin is **O** and whose angular orientation is specified by the *Nautical* angles, their first and second time derivative. The rotation sequence is Z, Y, X (see also § 2.4.3 and table 3.2).
Equivalent to `cardantoH(q, qp, qpp, Z, Y, X, O, H);`

See also: `cardantoH`.

Htocardan

Acceleration matrix to Cardan angles. Contained in `spaceli4.c`

Calling sequence:	`rv = Htocardan (m, W, H, i, j, k, q1, q2, qp1, qp2, qpp1, qpp2);`
Prototype:	`int Htocardan (MAT4 m, MAT4 W, MAT4 H, int i, int j, int k, real *q1, real *q2, real *qp1, real *qp2, real *qpp1, real *qpp2);`
Return value:	`int rv.`
Input parameters:	`MAT4 m, W, H; int i, j, k.`
Output parameters:	`real q1[3], q2[3], qp1[3], qp2[3], qpp1[3], qpp2[3].`

Builds the *Euler/Cardan* angles, first and second time derivative of a frame. It uses the acceleration matrix **H**, the velocity matrix **W** and the position matrix **m**. Both solutions are evaluated. The parameters **i**, **j**, **k** specify the sequence of the rotation axes (their value must be the constant `X`, `Y` or `Z`. See § 2.4.3). **j** must be different from **i** and **k**, **k** could be equal to **i** (see also table 3.2). **q1** and **q2** are 3 element vectors containing the two angles set. **qp1** and **qp2** are a 3 element vectors containing the 1^{st} time derivative of **q1** and **q2** respectively. **qpp1** and **qpp2** are a 3 element vectors containing the 2^{nd} time derivative of **q1** and **q2**. The function returns the value **rv** that can be either `OK` or `NOTOK` (see § 2.4.3) in order to specify the correct outcome.

See also example 3.22

See also: `Mtocardan`, `Wtocardan`, `cardantoH`, `Htocard`, `Htoeul`, `Htonaut`.

Htocard

Acceleration matrix to Cardan angles. Macro contained in `spacelib.h`

Calling sequence:	`rv = Htocard (m, W, H, q1, q2, qp1, qp2, qpp1, qpp2);`
Prototype:	`int Htocard (MAT4 m, MAT4 W, MAT4 H, real *q1, real *q2, real *qp1, real *qp2, real *qpp1, real *qpp2);`
Return value:	`int rv.`
Input parameters:	`MAT4 m, W, H.`
Output parameters:	`real q1[3], q2[3], qp1[3], qp2[3], qpp1[3], qpp2[3].`

Builds the *Tait-Brian* angles of a frame whose position matrix is **m**, their first and second time derivative. The rotation sequence is X, Y, Z (see also § 2.4.3 and table 3.2).
Equivalent to `Htocardan(m, W, H, X, Y, Z, q1, q2, qp1, qp2, qpp1, qpp2);`

See also: `Htocardan`

Htoeul

Acceleration matrix to Cardan angles. Macro contained in `spacelib.h`

Calling sequence:	`rv = Htoeul (m, W, H, q1, q2, qp1, qp2, qpp1, qpp2);`
Prototype:	`int Htoeul (MAT4 m, MAT4 W, MAT4 H, real *q1, real *q2, real *qp1, real *qp2, real *qpp1, real *qpp2);`
Return value:	`int rv.`
Input parameters:	`MAT4 m, W, H.`
Output parameters:	`real q1[3], q2[3], qp1[3], qp2[3], qpp1[3], qpp2[3].`

Builds the *Euler* angles of a frame whose position matrix is **m**, their first and second time derivative. The rotation sequence is Z, X, Z (see also § 2.4.3 and table 3.2).
Equivalent to `Htocardan(m, W, H, Z, X, Z, q1, q2, qp1, qp2, qpp1, qpp2);`

See also: `Htocardan`

Htonaut

Acceleration matrix to Cardan angles. Macro contained in `spacelib.h`

Calling sequence:	`rv = Htonaut (m, W, H, q1, q2, qp1, qp2, qpp1, qpp2);`
Prototype:	`int Htonaut (MAT4 m, MAT4 W, MAT4 H, real *q1, real *q2, real *qp1, real *qp2, real *qpp1, real *qpp2);`
Return value:	`int rv.`
Input parameters:	`MAT4 m, W, H.`
Output parameters:	`real q1[3], q2[3], qp1[3], qp2[3], qpp1[3], qpp2[3].`

Builds the *Nautical* angles of a frame whose position matrix is **m**, their first and second time derivative. The rotation sequence is Z, Y, X (see also § 2.4.3 and table 3.2).
Equivalent to `Htocardan(m, W, H, Z, Y, X, q1, q2, qp1, qp2, qpp1, qpp2);`

See also: `Htocardan`

cardanto_G

Cardan angles to angular acceleration matrix. Contained in `spaceli3.c`

Calling sequence:	`cardanto_G (q, qp, qpp, i, j, k, M A, dim);`
Prototype:	`void cardanto_G (real *q, real *qp, real *qpp, int i, int j, int k, MAT A, int dim);`
Input parameters:	`real q[3], qp[3], qpp[3]; int i, j, k, dim.`
Output parameters:	`MAT A.`

Evaluates the angular acceleration matrix of a moving frame from the three *Cardan* (or *Euler*) angles **q** and their first and second time derivatives **qp** and **qpp**. The parameters **i**, **j**, **k** specify the sequence of the rotation axes (their value must be the constant `X`, `Y` or `Z`. See § 2.4.3). **j** must be different from **i** and **k**, **k** could be equal to **i** (see also table 3.2). Matrix **A** must be recasted using the `M` operator (see § 2.6). **q** is a 3 element vector containing the 1^{st}, 2^{nd} and 3^{rd} angle. **qp** is the first time derivative of **q**. **qpp** is the second time derivative of **q**. **A** is a matrix where the result must be stored. The result is stored in the 3×3 upper-left submatrix **G** of the **dim**×**dim** matrix **A**.

$$G = \underline{\dot{\omega}} + \underline{\omega}^2 \tag{3.16}$$

See also example 3.22.

See also: `cardanto_G3`, `cardanto_G4`.

Example 3.22. *See sample program* `CARDG.C`.

This example is quite similar to example 3.21 (see also example 3.18). The angular acceleration is now stored in the 3×3 upper-left submatrix of a matrix A, so we have the following statements

```
MAT4 A;
VECTOR q={0.1,0.5,0.9};
VECTOR qp={0.2,0.4,0.1};
VECTOR qpp={0.5,1.2,0.3};
clear4(A);
cardanto_G(q,qp,qpp,Y,X,Z,M A,4);
```

The resulting matrix is

$$A = \left[\begin{array}{ccc|c} -0.025 & 0.020 & 0.340 & ? \\ 0.104 & -0.168 & -1.221 & ? \\ -0.302 & 1.235 & -0.189 & ? \\ \hline ? & ? & ? & ? \end{array}\right]$$

where the character '?' means that the value of this element is not affected by the function.

cardanto_G3

Cardan angles to angular acceleration matrix for 3×3 matrices. Macro contained in `spacelib.h`

Calling sequence:	`cardanto_G3 (q, qp , qpp, i, j, k, OMEGA);`
Prototype:	`void cardanto_G3 (real *q, real *qp, real *qpp, int i, int j, int k, MAT3 OMEGA);`
Input parameters:	`real q[3], qp[3], qpp[3]; int i , j, k.`
Output parameters:	`MAT3 OMEGA.`

Cardan angles to angular acceleration matrix for 3×3 matrices. Stores the result in a 3×3 matrix. Equivalent to `cardanto_G(q, qp, qpp, i, j, k, OMEGA, 3);`

See also: `cardanto_G`.

cardanto_G4

Cardan angles to angular acceleration matrix for 4×4 matrices. Macro contained in `spacelib.h`

Calling sequence:	`cardanto_G4 (q, qp , qpp, i, j, k, OMEGA);`
Prototype:	`void cardanto_G4 (real *q, real *qp, real *qpp, int i, int j, int k, MAT4 OMEGA);`
Input parameters:	`real q[3], qp[3], qpp[3]; int i , j, k.`
Output parameters:	`MAT4 OMEGA.`

Cardan angles to angular acceleration matrix for 3×3 matrices. Stores the result in a 3×3 upper-left submatrix of a 4×4 matrix.
Equivalent to `cardanto_G(q, qp, qpp, i, j, k, OMEGA, 4);`

See also: `cardanto_G`.

cardanto_OMEGAPTO

Cardan angles to angular acceleration. Contained in `spaceli3.c`

Calling sequence:	`cardanto_OMEGAPTO (q, qp, qpp, i, j, k, omegapto);`
Prototype:	`void cardanto_OMEGAPTO (real *q, real *qp, real *qpp, int i, int j, int k, real *omegapto);`
Input parameters:	`real q[3], qp[3], qpp[3]; int i, j, k.`
Output parameters:	`real omegapto[3].`

Evaluates the angular acceleration of a moving frame from the three *Cardan* (or *Euler*) angles **q** and their first and second time derivatives **qp** and **qpp**.The parameters **i**, **j**, **k** specify the sequence of the rotation axes (their value must be the constant `X`, `Y` or `Z`. See § 2.4.3). **j** must be different from **i** and **k**, **k** could be equal to **i** (see also table 3.2). **q** is a 3 element vector containing 1st, 2nd and 3rd angle. **qp** is the first time derivative of **q**. **qpp** is the second time derivative of **q**. **omegapto** is a 3 element vector containing the angular acceleration.

See also example 3.23.

Example 3.23. *See sample program* `CARDOPTO.C`.

Consider a moving frame whose origin is in the point $O = [25,\ 23,\ 30]^t$. It has variable q defined by a rotation of 1 rad around axis y, a rotation of 1.2 rad around axis x and a rotation of 3 rad around axis z. The first time derivative of q is filled with the values $[0, 1, 0]$ rad/s, while the second time derivative is filled with the values [3,2.5,4.01] rad/s^2. The angular acceleration *omegapto* is evaluated by the following statements

```
VECTOR q={1.,1.2,3.};
VECTOR qp={0.,1.,0.};
VECTOR qpp={3.,2.5,4.01};
VECTOR omegapto;
cardanto_OMEGAPTO(q,qp,qpp,Y,X,Z,
                  omegapto);
```

The resulting vector is

$$omegapto = [\,2.573,\ -0.737,\ -1.319\,]^t$$

cardantol

Cardan angles to L matrix. Contained in `spaceli3.c`

Calling sequence:	`rv = cardantol (q, i, j, k, M R, dim);`
Prototype:	`int cardantol (real *q, int i, int j, int k, MAT R, int dim);`
Return value:	`int rv.`
Input parameters:	`real q[3]; int i, j, k, dim.`
Output parameters:	`MAT R.`

Builds the *ISA*'s (*Instantaneous Screw Axis*) matrix **L** of a frame whose orientation is specified by an *Euler/Cardan* convention. The parameters **i**, **j**, **k** specify the sequence of the rotation axes (their value must be the constant X, Y or Z. See § 2.4.3). **j** must be different from **i** and **k**, **k** could be equal to **i** (see also table 3.2). **q** is a 3 element vector containing the 1st, 2nd and 3rd angle. `cardantol` is internally called by `cardanto_omega` and `cardanto_OMEGAPTO`. The function returns the value **rv** that can be either OK or NOTOK (see § 2.4.3) in order to specify the correct outcome.

cardantoWPROD

Contained in `spaceli3.c`

Calling sequence:	`rv = int cardantoWPROD (q, i, j, k, M R, dim);`
Prototype:	`int cardantoWPROD (real *q, int i, int j, int k, R, dim);`
Return value:	`int rv.`
Input parameters:	`real q[3]; int i, j, k, dim.`
Output parameters:	`MAT R.`

Internally called by `cardanto_omega` and `cardanto_OMEGAPTO`. The parameters **i**, **j**, **k** specify the sequence of the rotation axes (their value must be the constant X, Y or Z. See § 2.4.3). **j** must be different from **i** and **k**, **k** could be equal to **i** (see also table 3.2). The function returns the value **rv** that can be either OK or NOTOK (see § 2.4.3) in order to specify the correct outcome.

invA

Contained in `spaceli4.c`

Calling sequence:	`rv = invA (alpha, beta, sig, i, j, k, Ai);`
Prototype:	`int invA (real alpha, real beta, int sig, int i, int j, int k, MAT3 Ai);`
Input parameters:	`real alpha, beta; int sig, i, j, k.`
Output parameters:	`MAT3 Ai.`

Internally called by `Wtocardan` and `Htocardan`. The parameters **i**, **j**, **k** specify the sequence of the rotation axes (their value must be the constant X, Y or Z. See § 2.4.3). **j** must be different from **i** and **k**, **k** could be equal to **i** (see also table 3.2). The function returns the value **rv** that can be either OK or NOTOK (see § 2.4.3) in order to specify the correct outcome.

WPRODtocardan

Contained in `spaceli4.c`

Calling sequence:	`WPRODtocardan (alpha, beta, sig, i, j, k, Atil);`
Prototype:	`void WPRODtocardan (real alpha, real beta, int sig, int i, int j, int k, MAT3 Atil);`
Input parameters:	`real alpha, beta; int sig, i, j, k.`
Output parameters:	`MAT3 Atil;`

Internally called by `Htocardan`. This function builds the transpose of the matrix find by `cardantoWPROD`. The parameters **i**, **j**, **k** specify the sequence of the rotation axes (their value must be the constant X, Y or

Z. See § 2.4.3). **j** must be different from **i** and **k**, **k** could be equal to **i** (see also table 3.2). The function returns the value **rv** that can be either OK or NOTOK (see § 2.4.3) in order to specify the correct outcome.

3.6 Construction of frames attached to points or vectors

frameP

Frames from points. Contained in `spaceli3.c`

Calling sequence:	`rv = frameP (P1, P2, P3, a1, a2, M A, dim);`
Prototype:	`int frameP (POINT P1, POINT P2, POINT P3, int a1, int a2, MAT A, int dim);`
Return value:	`int rv.`
Input parameters:	`POINT P1, P2, P3; int a1, a2, dim.`
Output parameters:	`MAT A.`

Builds a rotation matrix describing the angular position of a frame attached to three points. The origin is in **P1**, axis **a1** points from **P1** toward point **P2**, axis **a2** from **P1** toward point **P3** (if possible). Axis **a1** has priority on **a2**. Axis **a1** is directed as (**P2-P1**). Axis **a3** is directed as (**P2-P1**)×(**P3-P1**). Axis **a2** is directed as axis **a3**×axis **a1**. **a1** and **a2** (the axes) must be either the constant X, Y or Z defined in spacelib.h (see also §6.1). **a1** must be different from **a2**. The rotation matrix is stored in the 3×3 upper-left part of the **dim**×**dim** matrix **A**. Matrix **A** must be recasted using the M operator (see § 2.6). The function returns the value **rv** that can be either OK or NOTOK (see § 2.4.3) in order to specify the correct outcome.

See also example 3.24.

See also: frame4P, frameP3, frameP4.

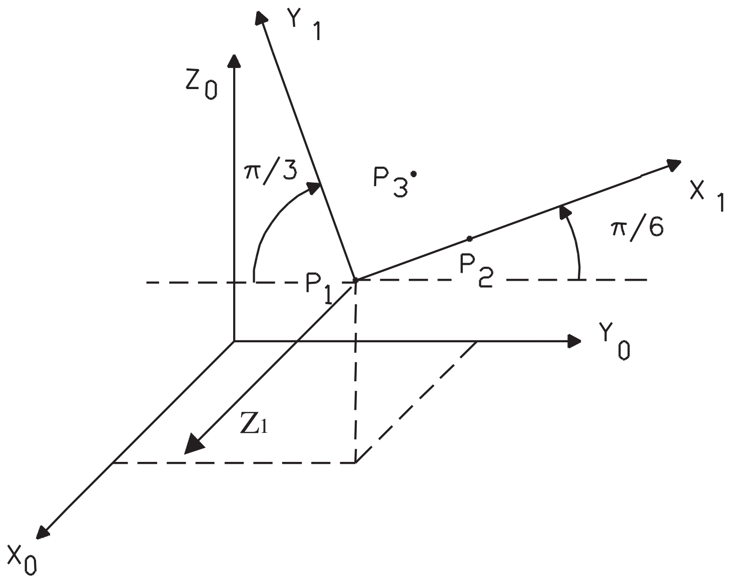


Figure 3.11: Frames definition for example 3.24

Example 3.24. *See sample program* FRAMEP.C.

In this example the three given points $P1$, $P2$ and $P3$ in the absolute frame (0) are used to build the frame (1) (see figure 3.11). With the given values

$$P1 = [\,5,\ 4,\ 3\,]^t, P2 = [\,5,\ 6,\ 4\,]^t, P3 = [\,5,\ 5,\ 5\,]^t$$

the angular position of reference frame (1) in figure, referred to frame (0), is expressed by the matrices

$$m_{0,1} = \left[\begin{array}{ccc|c} 0 & 0 & 1 & ? \\ 0.894 & -0.447 & 0 & ? \\ 0.447 & 0.894 & 0 & ? \\ \hline ? & ? & ? & ? \end{array}\right] \qquad R_{0,1} = \begin{bmatrix} 0 & 0 & 1 \\ 0.894 & -0.447 & 0 \\ 0.447 & 0.894 & 0 \end{bmatrix}$$

where the character '?' means that the value of this element is not affected by the function. These matrices can be built by the statements

```
MAT3 R01;
MAT4 m01;
POINT P1={5.,4.,3.,1.};
POINT P2={5.,6.,4.,1.};
POINT P3={5.,5.,5.,1.};
clear4(m01);
frameP(P1,P2,P3,X,Y,M m01,4);
frameP(P1,P2,P3,X,Y,M R01,3);
```

frameP3

Frames from points for 3×3 rotation matrices. Macro contained in `spacelib.h`

Calling sequence:	`rv = frameP3 (P1, P2, P3, a1, a2, R);`
Prototype:	`int frameP3 (POINT P1, POINT P2, POINT P3, int a1, int a2, MAT3 R);`
Return value:	`int rv.`
Input parameters:	`POINT P1, P2, P3; int a1, a2.`
Output parameters:	`MAT3 R.`

Builds a rotation matrix describing the angular position of a frame attached to three points.
Equivalent to `frameP(P1, P2, P3, a1, a2, M R, 3);`

See also: `frameP`.

frameP4

Frames from points for 4×4 rotation matrices. Macro contained in `spacelib.h`

Calling sequence:	`rv = frameP4 (P1, P2, P3, a1, a2, m);`
Prototype:	`int frameP4 (POINT P1, POINT P2, POINT P3, int a1, int a2, MAT4 m);`
Return value:	`int rv.`
Input parameters:	`POINT P1, P2, P3; int a1, a2.`
Output parameters:	`MAT4 m.`

Builds a rotation matrix describing the angular position of a frame attached to three points and stores it in the 3×3 upper-left part of a 4×4 position matrix **m**.
Equivalent to `frameP(P1, P2, P3, a1, a2, M m, 4);`

See also: `frameP`.

frame4P

Frame from three points. Contained in `spaceli3.c`

Calling sequence:	`frame4P (P1, P2, P3, a1, a2, m);`
Prototype:	`void frame4P (POINT P1, POINT P2, POINT P3; int a1, int a2, MAT4 m);`
Input parameters:	`POINT P1, P2, P3; int a1, a2.`
Output parameters:	`MAT4 m.`

Builds a 4×4 position matrix **m** describing the position and orientation of a frame attached to three points. The origin is put in point **P1**, axis **a1** points toward point **P2**, axis **a2** points toward point

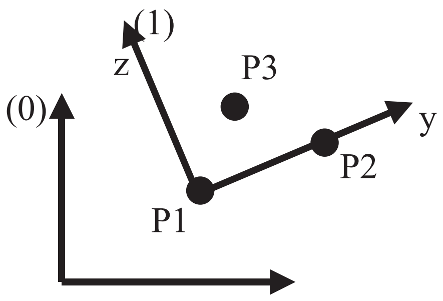


Figure 3.12: `frame4P` example of use: `frame4P(P1,P2,P3,Y,Z,M01)`

P3 (if possible). Axis **a1** has priority on **a2**. Axis **a1** is directed as (**P2**-**P1**). Axis **a3** is directed as (**P2**-**P1**)×(**P3**-**P1**). Axis **a2** is directed as axis **a3**×axis **a1**. The second axis is directed as axis **a3**×axis **a1** (see fig. 3.12). The axes **a1** and **a2** must be either the constant `X`, `Y` or `Z` defined in `spacelib.h` (see also §6.1). **a1** must be different from **a2**.

See also example 3.25.

See also: `frameP`.

Example 3.25. *See sample program* `FRAME4P.C`.

Referring to example 3.24, the origin of frame (1) is in the point $P1 = [5,\ 4,\ 3]^t$. The position matrix $m_{0,1}$ of frame (1) is then

$$m_{0,1} = \left[\begin{array}{ccc|c} 0 & 0 & 1 & 5 \\ 0.894 & -0.447 & 0 & 4 \\ 0.447 & 0.894 & 0 & 3 \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

This matrix is built by the following statements

```
MAT4 m01;
POINT P1={5.,4.,3.,1.};
POINT P2={5.,6.,4.,1.};
POINT P3={5.,5.,5.,1.};
clear4(m01);
frame4P(P1,P2,P3,X,Y,M m01,4);
```

frameV

Frame from vectors. Contained in `spaceli3.c`

Calling sequence:	`rv = frameV (v1, v2, a1, a2, M A, dim);`
Prototype:	`int frameV (VECTOR v1, VECTOR v2, int a1, int a2, MAT A, int dim);`
Return value:	`int rv.`
Input parameters:	`VECTOR v1, v2; int a1, a2, dim.`
Output parameters:	`MAT A.`

Builds a rotation matrix describing the angular position of a frame attached to two vectors. Axis **a1** is directed as **v1**, axis **a2** is directed as **v2** (if possible). Axis **a1** has priority. The third axis is directed as **v1**×**v2**. The second axis is directed as axis **a3**×axis **a1**. Axes **a1** and **a2** must be either the constant `X`, `Y` or `Z` defined in `spacelib.h` (see also 6.1). **a1** must be different from **a2**. The rotation matrix is stored in the 3×3 upper-left part of the **dim**×**dim** matrix **A**. Matrix **A** must be recasted using the `M` operator (see § 2.6). The function returns the value **rv** that can be either `OK` or `NOTOK` (see § 2.4.3) in order to specify the correct outcome.

See also example 3.26.

See also: `frame4V`, `frameV3`, `frameV4`.

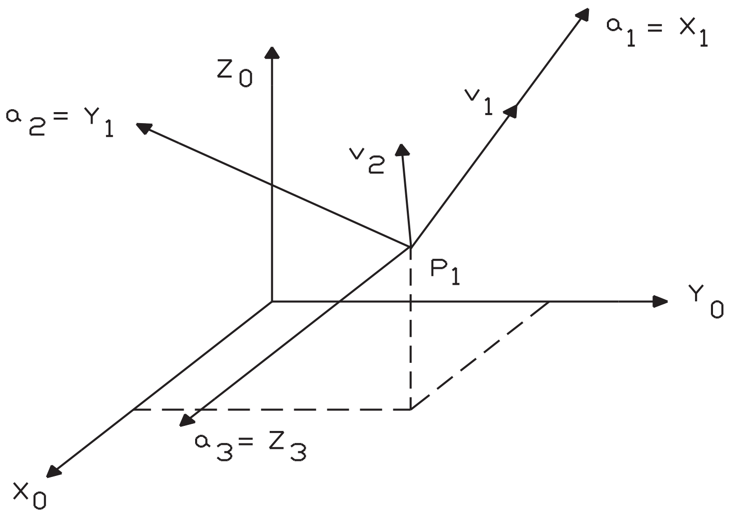


Figure 3.13: Frames definition for example 3.26.

Example 3.26. *See sample program* `FRAMEV.C`.

In this example a point $P1$ and two given vectors $v1$ and $v2$ in the absolute frame (0) are used to build the frame (1) (see figure 3.13). Referring to example 3.25 these vectors are:

$$v1 = P2 - P1 = [\,0,\ 2,\ 1\,]^t$$

$$v2 = P3 - P1 = [\,0,\ 1,\ 2\,]^t$$

The angular position of reference frame (1) in figure, referred to frame (0), is expressed by the matrix

$$m_{0,1} = \left[\begin{array}{ccc|c} 0 & 0 & 1 & ? \\ 0.894 & -0.447 & 0 & ? \\ 0.447 & 0.894 & 0 & ? \\ \hline ? & ? & ? & ? \end{array}\right]$$

where the character '?' means that the value of this element is not affected by the function.

This matrix can be built by the statements

```
MAT4 m01;
VECTOR v1={0.,2.,1.};
VECTOR v2={0.,1.,2.};
clear4(m01);
frameV(v1,v2,X,Y,M m01,4);
```

frameV3

Frame from vectors for 3×3 rotation matrices. Macro contained in `spacelib.h`

Calling sequence:	`rv = frameV3 (v1, v2, a1, a2, R);`
Prototype:	`int frameV3 (VECTOR v1, VECTOR v2, int a1, int a2, MAT3 R);`
Return value:	`int rv.`
Input parameters:	`VECTOR v1, v2; int a1, a2.`
Output parameters:	`MAT3 R.`

Builds a rotation matrix describing the angular position of a frame attached to two vectors. Equivalent to `frameV(v1, v2, a1, a2, M R, 3);`

See also: `frameV`.

frameV4

Frame from vectors for 4×4 position matrices. Macro contained in `spacelib.h`

Calling sequence:	`rv = frameV4 (v1, v2, a1, a2, m);`
Prototype:	`int frameV4 (VECTOR v1, VECTOR v2, int a1, int a2, MAT4 m);`
Return value:	`int rv.`
Input parameters:	`VECTOR v1, v2; int a1, a2.`
Output parameters:	`MAT4 m.`

Builds a rotation matrix describing the angular position of a frame attached to two vectors and stores it in the upper-left 3×3 submatrix of a 4×4 position matrix **m**.
Equivalent to `frameV(v1, v2, a1, a2, M R, 3);`

See also: `frameV`.

frame4V

Frame from a point and two vectors. Contained in `spaceli3.c`

Calling sequence:	`frame4V (P1, v1, v2, a1, a2, m);`
Prototype:	`void frame4V (POINT P1, VECTOR v1, VECTOR v2, int a1, int a2, MAT4 m);`
Input parameters:	`POINT P1, v1, v2, a1, a2.`
Output parameters:	`MAT4 m;`

Builds a 4×4 position matrix **m** describing the position and orientation of a frame attached to two vectors and one point. The origin is put in point **P1**, axis **a1** is directed as vector **v1**, axis **a2** is directed as **v2** (if possible). Axis **a1** has priority. The third axis is directed as **v1**×**v2**. The second axis is directed as axis **a3**×axis **a1**. Axes **a1** and **a2** must be either the constant `X`, `Y` or `Z` defined in `spacelib.h` (see also § 2.4.3). **a1** must be different from **a2**.

See also example 3.27.

See also: `frameV`.

Example 3.27. *See sample program* `FRAME4V.C`.

Referring to example 3.26, the origin of frame (1) is in the point $P1 = [5, 4, 3]^t$. The position matrix $m_{0,1}$ of frame (1) is then

$$m_{0,1} = \left[\begin{array}{ccc|c} 0 & 0 & 1 & 5 \\ 0.894 & -0.447 & 0 & 4 \\ 0.447 & 0.894 & 0 & 3 \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

This matrix is built by the statements

```
MAT4 m01;
POINT P1={5.,4.,3.,1.};
VECTOR v1={0.,2.,1.};
VECTOR v2={0.,1.,2.};
frame4V(P1, v1,v2,X,Y,M m01);
```

axis

Axis of Frame. Contained in `spaceli5.c`

Calling sequence:	`axis (n, A);`
Prototype:	`void axis (int n, AXIS A);`
Input parameters:	`int n.`
Output parameters:	`AXIS A.`

Returns a 3 elements unit vector **A** parallel to a frame axis **n**. **n** must be either the constant `X`, `Y` or `Z` defined in `spacelib.h` (see also § 2.4.3). For example `axis(Y,a)` returns $a = \{0., 1., 0.\}$.

3.7 Working with points, lines and planes

3.7.1 Operations on points

angle

Angle between points. Contained in `spaceli3.c`

Calling sequence:	`alpha = real angle (P1, P2, P3);`
Prototype:	`real angle (POINT P1, POINT P2, POINT P3);`
Return value:	`real alpha.`
Input parameters:	`POINT P1, P2, P3.`

Function returning the angle **alpha** between three points which is the angle between vectors `(P1-P2)` and `(P3-P2)`.

dist

Distance between two points Contained in `spaceli3.c`

Calling sequence:	`d = dist (P1, P2);`
Prototype:	`real dist (POINT P1, POINT P2);`
Return value:	`real d.`
Input parameters:	`POINT P1, P2.`

Function returning the distance **d** between two points.

intermediate

Intermediate point. Contained in `spaceli4.c`

Calling sequence:	`intermediate (P1, a, P2, b, P);`
Prototype:	`void intermediate (POINT P1, real a, POINT P2, real b, POINT P);`
Input parameters:	`POINT P1, P2; real a, b.`
Output parameters:	`POINT P.`

Evaluates point **P** as the middle point between points **P1** and **P2** using two weights **a** and **b**. It uses the equation:

$$P = \frac{P_1 a + P_2 b}{a + b} \tag{3.17}$$

When **a**==**b**==1 the function `intermediate` is equivalent to function `middle`. If $a + b$ is equal to zero, it is assumed that $a + b = 1$.

See also: `middle`.

middle

Middle point. Contained in `spaceli3.c`

Calling sequence:	`middle (P1, P2, P);`
Prototype:	`void middle (POINT P1, POINT P2, POINT P);`
Input parameters:	`POINT P1, P2.`
Output parameters:	`POINT P.`

Evaluates point **P** as the middle point between points **P1** and **P2**. It uses the relation (see also (3.17)):

$$P = \frac{1}{2}\left(P_1 + P_2\right) \tag{3.18}$$

See also: `intermediate`.

`vect`

Vector between points. Contained in `spaceli3.c`

Calling sequence:	`vect (P1, P2, v);`
Prototype:	`void vect (POINT P1, POINT P2, VECTOR v);`
Input parameters:	`POINT P1, P2.`
Output parameters:	`VECTOR v.`

Function evaluating the vector **v**=**P1-P2** (from **P2** toward **P1**).

3.7.2 Operations on lines and planes

`line2p`

Line from two points. Contained in `spaceli4.c`

Calling sequence:	`line2p (P1, P2, l);`
Prototype:	`void line2p (POINT P1, POINT P2, LINEP l);`
Input parameters:	`POINT P1, P2.`
Output parameters:	`LINEP l.`

Builds a line passing from points **P1** and **P2**.

See also: `linepvect`.

`linepvect`

Line from point and vector. Contained in `spaceli4.c`

Calling sequence:	`void linepvect (P1, v, l);`
Prototype:	`void linepvect (POINT P1, VECTOR v, LINEP l);`
Input parameters:	`POINT P1; VECTOR v.`
Output parameters:	`LINEP l.`

Builds a line which passes through point **P1** and has the same direction as vector **v**.

See also: `line2p`.

`intersection`

Intersection of two lines. Contained in `spaceli4.c`

Calling sequence:	`intersection (l1, l2, lmindist, &mindist, pl, I, &inttype);`
Prototype:	`void intersection (LINE l1, LINE l2, LINEP lmindist, real *mindist, PLANE pl, POINT I, int *inttype);`
Input parameters:	`LINE l1, l2.`
Output parameters:	`LINEP lmindist; real mindist; PLANE pl; POINT I; int inttype;`

Function evaluating the intersection point **I** between lines **l1** and **l2**. This function builds also, when possible, the minimum distance line **lmindist** and a plane **pl** containing **l1** and **l2**. The parameter **inttype** defines whether an intersection point was found or not. **inttype** may have the following values:

1 - **l1** and **l2** are oblique lines. **I** is the middle point on the minimum distance line. **pl** does not really contain the two lines.

0 - **l1** and **l2** have exactly one intersection point **I**. The minimum distance **mindist** is zero. **pl** contains both **l1** and **l2**.

-1 - **l1** and **l2** are the same line. The intersection of the two is the line itself. **pl** can't be built.

2 - **l1** is parallel to **l2** (no intersection). **pl** contains both **l1** and **l2**.

See also: `interslpl`, `inters2pl`.

projponl

Projection of point on line. Contained in `spaceli4.c`

Calling sequence:	`d = projponl (P1, l, I);`
Prototype:	`real projponl (POINT P1, LINE l, POINT I);`
Return value:	`real d.`
Input parameters:	`POINT P1; LINE l.`
Output parameters:	`POINT I.`

It finds the projection **I** of point **P** on line **l**. This function returns also the distance **d** of **P** from **l**.

See also example 3.28.

See also: `project`.

Example 3.28. *See sample program* `PROJPONL.C`.

The given line l has the origin in point $O = [\,3,\ 7.2,\ 2.05,\ 1\,]$ and its direction is $[\,0.7,\ 4,\ 9\,]$. If the line m which is orthogonal to l and passes through the point $P = [\,5,\ 1,\ 3,\ 1\,]$ has to be found, this is performed by means of the following statements

```
LINE l={{3.,7.2,2.05,1.},
        {0.7,4.,9.} };
LINE m;
POINT P={5.,1.,3.,1.};
POINT P1;
projponl(P,l,P1);
line2p(P,P1,&m);
```

which produces the following result

$$m = \begin{bmatrix} 5 & -0.32867 \\ 1 & 0.87226 \\ 3 & -0.36211 \\ 1 & 0 \end{bmatrix}$$

The origin of the new line is: $\{\,5,\ 1,\ 3,\ 1\,\}$, the direction of the new line is: $\{-0.3287,\ 0.8723,\ -0.3621\,\}$.

distpp

Distance of point from a plane. Contained in `spaceli4.c`

Calling sequence:	`d = distpp (pl, P);`
Prototype:	`real distpp (PLANE pl, POINT P);`
Return value:	`real d.`
Input parameters:	`PLANE pl; POINT P.`

Evaluates the distance **d** of point **P** from plane **pl**.

project

Project a point on a plane. Contained in `spaceli4.c`

Calling sequence:	`d = project (P, pl, I);`
Prototype:	`real project (POINT P, PLANE pl, POINT I);`
Return value:	`real d.`
Input parameters:	`POINT P; PLANE pl.`
Output parameters:	`POINT I.`

Finds the projection **I** of point **P** on plane **pl**. This function returns also the distance **d** of **P** from **pl**.

See also: `projponl`.

plane

Plane from three points. Contained in `spaceli4.c`

Calling sequence:	`plane (P1, P2, P3, Pl);`
Prototype:	`void plane (POINT P1, POINT P2, POINT P3, PLANE pl);`
Input parameters:	`POINT P1, P2, P3.`
Output parameters:	`PLANE pl.`

Builds a plane **pl** which contains the three given points **P1**, **P2** and **P3**. **pl** is defined by four elements: the three components in the reference frame of the unit vector orthogonal to the plane itself and the distance of **pl** from the origin of reference frame (considered with the sign)(see also § 2.4.2).

See also: `plane2`.

plane2

Plane from point and vector. Contained in `spaceli4.c`

Calling sequence:	`plane2 (P1, v, pl);`
Prototype:	`void plane2 (POINT P1, VECTOR v, PLANE pl);`
Input parameters:	`POINT P1; VECTOR v.`
Output parameters:	`PLANE pl.`

Builds a plane **pl** which contains point **P1** and is directed as vector **v**.

See also: `plane`.

inters2pl

Intersecton of two planes. Contained in `spaceli4.c`

Calling sequence:	`rv = inters2pl (pl1, pl2, l);`
Prototype:	`int inters2pl (PLANE pl1, PLANE pl2, LINEP l);`
Return value:	`int rv.`
Input parameters:	`PLANE pl1, pl2.`
Output parameters:	`LINEP l.`

Function evaluating the intersection between two planes **pl1** and **pl2**. The line type structure pointed by **l** is filled with the resulting value. The function returns the value **rv** that can be either `OK` or `NOTOK` (see § 2.4.3) in order to specify the correct outcome. Returns `NOTOK` when **pl1** is parallel to **pl2** (no intersection found).

See also: `intersection`.

interslpl

Intersecton of line and plane. Contained in `spaceli4.c`

Calling sequence:	`interslpl (l, pl, I, &inttype);`
Prototype:	`void interslpl (LINE l, PLANE pl, POINT I, int *inttype);`
Input parameters:	`LINE l; PLANE pl.`
Output parameters:	`POINT I; int inttype.`

Function evaluating the intersection point **I** between line **l** and plane **pl**. The parameter **inttype** defines whether an intersection point was found or not. **inttype** has the following values

1 : line **l** lies on plane **pl** and the intersection of the two is the line itself.

-1 : line **l** is parallel to plane **pl** (no intersection).

0 : line **l** and plane **pl** have exactly one intersection point **I**.

See also example 3.29.

See also: `intersection`.

Example 3.29. *See sample program* INTERSLP.C.

Let' s consider a plane $pl = [0, 0, 1, -5]$ and a direction $dir = [0, 0, 1]$. If the symmetric point of $P = [0, 6, 10]$ with respect to plane pl in the direction dir has to be found, this is performed by means of the following statements

```
PLANE pl={0.,0.,1.,-5.};
VECTOR dir={0.,0.,1.},
VECTOR v;
POINT P={0.,6.,10.,1.},
POINT P1;
POINT Ps;
LINE l;
int type;
real d;
pcopy (P,l.P);
vcopy(dir,l.dir);
interslpl(l,pl,P1,&type);
d=dist(P,P1);
vector(dir,d,v);
subv(P1,v,Ps);
Ps[U]=1.;
```

the resulting value are

$$l = \begin{bmatrix} 0 & 0 \\ 6 & 0 \\ 10 & 1 \\ 1 & 0 \end{bmatrix} \quad P1 = \begin{bmatrix} 0 \\ 6 \\ 5 \\ 1 \end{bmatrix}$$

$$type = 0 \quad d = 5 \quad v = \begin{bmatrix} 0 \\ 0 \\ 5 \\ 0 \end{bmatrix} \quad Ps = \begin{bmatrix} 0 \\ 6 \\ 0 \\ 1 \end{bmatrix}$$

3.8 Operations on matrices and vectors

3.8.1 Matrices and vectors algebra

molt

Matrix multiplication. Contained in `spacelib.c`

Calling sequence:	`molt (M A, M B, M C, d1, d2, d3);`
Prototype:	`void molt (MAT A, MAT B, MAT C, int d1, int d2, int d3);`
Input parameters:	`MAT A, B; int d1, d2, d3.`
Output parameters:	`MAT C.`

Evaluates the matrix product **C**=**A** · **B** of generic matrices with the following numbers of rows and columns **A**[**d1**,**d2**], **B**[**d2**,**d3**] and **C**[**d1**,**d3**]. **A** can be equal to **B** but **C** must be another matrix. For example the following statement is legal:

```
molt(M A, M A, M C, d1, d1, d1);
```

and the result is C=A · A (if A==B then of course d1==d2==d3). Matrices **A**, **B**, **C** must be recasted using the M operator (see § 2.6).

See also: `molt3`, `molt4`, `moltp`, `moltmv3`.

molt3

3×3 matrix multiplication. Macro contained in `spacelib.h`

Calling sequence:	`molt3 (A, B, C);`
Prototype:	`void molt3 (MAT3 A, MAT3 B, MAT3 C);`
Input parameters:	`MAT3 A, B.`
Output parameters:	`MAT3 C.`

Multiplies a 3×3 matrix **A** by a 3×3 matrix **B** and puts the result in **C**.
Equivalent to `molt(M A, M B, M C, 3, 3, 3);`

See also: `molt`.

molt4

4×4 matrix multiplication. Macro contained in `spacelib.h`

Calling sequence:	`molt3 (A, B, C);`
Prototype:	`void molt4 (MAT4 A, MAT4 B, MAT4 C);`
Input parameters:	`MAT4 A, B.`
Output parameters:	`MAT4 C.`

Multiplies a 4×4 matrix **A** by a 4×4 matrix **B** and puts the result in **C**.
Equivalent to `molt(M A, M B, M C, 4, 4, 4);`

See also: `molt`.

moltp

Multiply a matrix by a point. Macro contained in `spacelib.h`

Calling sequence:	`moltp (A, P1, P2);`
Prototype:	`void molt4 (MAT4 A, POINT P1, POINT P2);`
Input parameters:	`MAT4 A; POINT P1.`
Output parameters:	`POINT P2.`

Multiplies a 4×4 matrix **A** by point **P1** and puts the resulting 4 element vector into **P2**. **P2**=**A**·**P1**.
Equivalent to `molt(M A, M P1, M P2, 4, 4, 1);`

See also: `molt`.

moltmv3

Multiply a matrix by a point. Macro contained in `spacelib.h`

Calling sequence:	`moltmv3 (A, v1, v2);`
Prototype:	`void moltmv3 (MAT4 A, VECTOR v1, VECTOR v2);`
Input parameters:	`MAT4 A; VECTOR v1.`
Output parameters:	`VECTOR v2.`

Multiplies a 3×3 matrix **A** by 3×1 vector **v1** and puts the resulting vector into **v2**. **v2**=**A**·**v1**.
Equivalent to `molt(M A, M v1, M v2, 3, 3, 1);`

See also: `molt`.

rmolt

Multiply a scalar by a vector. Contained in `spacelib.c`

Calling sequence:	`rmolt (M A, r, M B, d1, d2);`
Prototype:	`void rmolt (MAT A, real r, MAT B, int d1, int d2);`
Input parameters:	`MAT A; real r; int d1, d2.`
Output parameters:	`MAT B.`

Evaluates the **d1**×**d2** matrix **B** elements as a product of a matrix **A** and a scalar **r** (**B**=**r**·**A**). **A** and **B** can be the same matrix, so the following call is legal

```
rmolt(M A, r, M A, d1, d2);
```

and the result is **A**=**r**·**A**. Matrices **A** and **B** must be recasted using the `M` operator (see § 2.6).

See also: `rmolt3`, `rmolt4`, `rmoltv`.

rmolt3

Multiply a scalar by a 3×3 matrix. Macro contained in `spacelib.h`

Calling sequence:	`rmolt3 (A, r, B);`
Prototype:	`void rmolt3 (MAT3 A, real r, MAT3 B);`
Input parameters:	`MAT3 A; real r.`
Output parameters:	`MAT3 B.`

Multiplies a 3×3 square matrix **A** and scalar **r**, puts the result in **B**. $\mathbf{B}=\mathbf{r}\cdot\mathbf{A}$.
Equivalent to `rmolt(M A, r, M B, 3, 3);`

See also: `rmolt`.

rmolt4

Multiply a scalar by a 4×4 matrix. Macro contained in `spacelib.h`

Calling sequence:	`rmolt3 (A, r, B);`
Prototype:	`void rmolt4 (MAT4 A, real r, MAT4 B);`
Input parameters:	`MAT4 A; real r.`
Output parameters:	`MAT4 B.`

Multiplies a 4×4 square matrix **A** and scalar **r**, puts the result in **B**. $\mathbf{B}=\mathbf{r}\cdot\mathbf{A}$.
Equivalent to `rmolt(M A, r, M B, 4, 4);`

See also: `rmolt`.

rmoltv

Multiply a scalar by a vector. Macro contained in `spacelib.h`

Calling sequence:	`rmoltv (v1, r, v2);`
Prototype:	`void rmoltv (VECTOR v1, real r, VECTOR v2);`
Input parameters:	`VECTOR v1; real r.`
Output parameters:	`VECTOR v2.`

Multiplies a vector **v1** and a scalar **r**, puts the result in **v2**. $\mathbf{v2}=\mathbf{r}\cdot\mathbf{v1}$.
Equivalent to `rmolt(M v1, r, M v2, 3, 1);`

See also: `rmolt`.

sum

Sum of matrices Contained in `spacelib.c`

Calling sequence:	`sum (M A, M B, M C, d1, d2);`
Prototype:	`void sum (MAT A, MAT B, MAT C, int d1, int d2);`
Input parameters:	`MAT A, B; int d1, d2.`
Output parameters:	`MAT C.`

Evaluates the matrix sum of matrices having **d1** rows and **d2** columns. `C=A+B`. `A` can be equal to `B` and `C`. Matrices `A`, `B`, `C` must be recasted using the `M` operator (see § 2.6).
For instance the following calls are legal

```
sum(M A,M B,M B,d1,d2);      the result is B=B+A.
sum(M A,M A,M B,d1,d2);      the result is B=2*A.
```

See also: `sum3`, `sum4`, `sumv`.

sum3

Sum of 3×3 matrices Macro contained in `spacelib.h`

Calling sequence:	`sum3 (A, B, C);`
Prototype:	`void sum3 (MAT3 A, MAT3 B, MAT3 C);`
Input parameters:	`MAT3 A, B`
Output parameters:	`MAT3 C.`

Adds 3×3 matrices **A** and **B** and puts the result in **C**.
Equivalent to `sum(M A, M B, M C, 3, 3);`

See also: `sum`.

sum4

Sum of 4×4 matrices Macro contained in `spacelib.h`

Calling sequence:	`sum4 (A, B, C);`
Prototype:	`void sum4 (MAT4 A, MAT4 B, MAT4 C);`
Input parameters:	`MAT4 A, B`
Output parameters:	`MAT4 C.`

Adds 4×4 matrices **A** and **B** and puts the result in **C**.
Equivalent to `sum(M A, M B, M C, 4, 4);`

See also: `sum`.

sumv

Sum of vectors Macro contained in `spacelib.h`

Calling sequence:	`sum (a, b, c);`
Prototype:	`void sum (VECTOR a, VECTOR b, VECTOR c);`
Input parameters:	`VECTOR a, b`
Output parameters:	`VECTOR c.`

Adds vectors **A** and **B** and puts the result in **C**.
Equivalent to `sum(M a, M b, M c, 3, 1);`

See also: `sum`.

sub

Subtraction for matrices Contained in `spacelib.c`

Calling sequence:	`sub (M A, M B, M C, d1, d2);`
Prototype:	`void sub (MAT A, MAT B, MAT C, int d1, int d2);`
Input parameters:	`MAT A, B; int d1, d2.`
Output parameters:	`MAT C.`

Evaluates the matrix difference of matrices having **d1** rows and **d2** columns `C=A-B`. **A** can be equal to **B** and/or **C**. Matrices **A**, **B**, **C** must be recasted using the `M` operator (see § 2.6). For instance the following call is legal

`sub(M A,M B,M B,d1,d2);` the result is `B=B-A`.

See also: `sub3`, `sub4`, `subv`.

sub3

Subtraction for 3×3 matrices Macro contained in `spacelib.h`

Calling sequence:	`sub3 (A, B, C);`
Prototype:	`void sub3 (MAT3 A, MAT3 B, MAT3 C);`
Input parameters:	`MAT3 A, B.`
Output parameters:	`MAT3 C.`

Subtracts a 3×3 matrix **B** from **A** and puts the result in **C**.
Equivalent to `sub(M A, M B, M C, 3, 3);`

See also: `sub`

`sub4`

Subtraction for 4×4 matrices — Macro contained in `spacelib.h`

Calling sequence:	`sub4 (A, B, C);`
Prototype:	`void sub4 (MAT4 A, MAT4 B, MAT4 C);`
Input parameters:	`MAT4 A, B.`
Output parameters:	`MAT4 C.`

Subtracts a 4×4 matrix **B** from **A** and puts the result in **C**.
Equivalent to `sub(M A, M B, M C, 4, 4);`

See also: `sub`

`subv`

Subtraction for vectors — Macro contained in `spacelib.h`

Calling sequence:	`subv (a, b, c);`
Prototype:	`void subv (VECTOR a, VECTOR b, VECTOR c);`
Input parameters:	`VECTOR a, b.`
Output parameters:	`VECTOR c.`

Subtracts a vector **b** from **a** and puts the result in **c**. Equivalent to sub(M a, M b, M c, 3, 1);

See also: `sub`

3.8.2 General operations on matrices

`crossmtom`

Cross product to matrices. — Contained in `spaceli4.c`

Calling sequence:	`crossmtom (a, b, M c, dim);`
Prototype:	`void crossmtom (VECTOR a, VECTOR b, MAT c, int dim);`
Input parameters:	`VECTOR a, b; int dim.`
Output parameters:	`MAT c.`

Obsolete version. Please use function `crossvtom`.

This function performs the operation cross product between two vectors **a** and **b**

$$\vec{c} = \vec{a} \times \vec{b} \tag{3.19}$$

and put the result in a matrix **c**[4]. It uses the relation

$$\underline{c}_{(k)} = -a_{(k)} b^t_{(k)} + b_{(k)} a^t_{(k)} \tag{3.20}$$

Matrices **c** must be recasted using the `M` operator (see § 2.6).

[4]A vector $\vec{v}$ could be represented in the following forms [3], [4], [2]:

$$v = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} \qquad \underline{v} = \begin{bmatrix} 0 & -v_z & v_y \\ v_z & 0 & -v_x \\ -v_y & v_x & 0 \end{bmatrix}$$

crossvtom

Cross product to matrices (Vector version). Contained in `spaceli5.c`

Calling sequence:	`crossmtom (a, b, M c, dim);`
Prototype:	`void crossmtom (VECTOR a, VECTOR b, MAT c, int dim);`
Input parameters:	`VECTOR a, b; int dim.`
Output parameters:	`MAT c.`

This function performs the operation cross product between two vectors **a** and **b**

$$\vec{c} = \vec{a} \times \vec{b} \tag{3.21}$$

and put the result in a matrix **c**[5]. It uses the relation

$$\underline{c}_{(k)} = -a_{(k)} b^t_{(k)} + b_{(k)} a^t_{(k)} \tag{3.22}$$

Matrices **c** must be recasted using the `M` operator (see § 2.6).

crossMtoM

Cross product to matrices (Matricial version). Contained in `spaceli5.c`

Calling sequence:	`crossMtoM (M a, M b, M c, dim);`
Prototype:	`void crossmtom (MAT a, MAT b, MAT c, int dim);`
Input parameters:	`MAT a, b; int dim.`
Output parameters:	`MAT c.`

This function performs the operation cross product between two vectors in matricial form **a** and **b**

$$\vec{c} = \vec{a} \times \vec{b} \tag{3.23}$$

and put the result in a matrix **c**[5]. It uses the relation

$$c = a \cdot b - b \cdot a \tag{3.24}$$

Matrices **a**, **b** and **c** must be recasted using the `M` operator (see § 2.6).

clear

Clear a matrix (fill it with zeros). Contained in `spaceli3.c`

Calling sequence:	`clear (M A, id, jd);`
Prototype:	`void clear (MAT A, int id, int jd);`
Input parameters:	`int id, jd.`
Output parameters:	`MAT A.`

Fills with zeros a **id**×**jd** matrix **A**. Matrix **A** must be recasted using the `M` operator (see § 2.6).
See also: `clear3`, `clear4`, `clearv`.

clear3

Clear a 3×3 matrix (fill it with zeros). Macro contained in `spacelib.h`

Calling sequence:	`clear3 (A);`
Prototype:	`void clear3 (MAT3 A);`
Output parameters:	`MAT3 A.`

Fills with zeros a 3×3 matrix **A**.
Equivalent to `clear(M A, 3, 3);`
See also: `clear`

[5] A vector $\vec{v}$ could be represented in the following forms [3], [4], [2]:

$$v = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} \qquad \underline{v} = \begin{bmatrix} 0 & -v_z & v_y \\ v_z & 0 & -v_x \\ -v_y & v_x & 0 \end{bmatrix}$$

clear4

Clear a 4×4 matrix (fill it with zeros). Macro contained in `spacelib.h`

Calling sequence:	`clear4 (A);`
Prototype:	`void clear4 (MAT4 A);`
Output parameters:	`MAT4 A.`

Fills with zeros a 4×4 matrix **A**.
Equivalent to `clear(M A, 4, 4);`

See also: `clear`

idmat

Identity matrix. Contained in `spaceli3.c`

Calling sequence:	`idmat (M A, id, jd);`
Prototype:	`void idmat (M A, id, jd);`
Input parameters:	`int id, jd.`
Output parameters:	`MAT A.`

Makes unitary matrix **A** having **id**×**jd** dimension. Matrix **A** must be recasted using the `M` operator (see § 2.6).

See also: `idmat3`, `idmat4`.

idmat3

Identity 3×3 matrix. Macro contained in `spacelib.h`

Calling sequence:	`idmat3 (A);`
Prototype:	`void idmat (Mat3 A);`
Output parameters:	`MAT3 A.`

Makes unitary 3×3 matrix **A**.
Equivalent to `idmat(M A,3);`

See also: `idmat`

idmat4

Identity 4×4 matrix. Macro contained in `spacelib.h`

Calling sequence:	`idmat4 (A);`
Prototype:	`void idmat (Mat4 A);`
Output parameters:	`MAT4 A.`

Makes unitary 4×4 matrix **A**.
Equivalent to `idmat(M A,4);`

See also: `idmat`

norm

Norm of a matrix. Contained in `spacelib.c`

Calling sequence:	`n = norm (M A, d1, d2);`
Prototype:	`real norm (MAT A , int d1, int d2);`
Return value:	`real n.`
Input parameters:	`MAT A; int d1, d2;`

Returns the norm **n** (maximum absolute value of its elements) of a **d1**×**d2** matrix **A**.

See also: `norm3`, `norm4`.

norm3

Norm of a 3×3 matrix. Macro contained in `spacelib.h`

Calling sequence:	`n = norm3 (A);`
Prototype:	`real norm (MAT3 A);`
Return value:	`real n.`
Input parameters:	`MAT3 A.`

Returns the norm **n** of a 3×3 matrix **A**. Equivalent to `norm(M A, 3, 3);`

See also: `norm`.

norm4

Norm of a 4×4 matrix. Macro contained in `spacelib.h`

Calling sequence:	`n = norm4 (A);`
Prototype:	`real norm (MAT4 A);`
Return value:	`real n.`
Input parameters:	`MAT4 A.`

Returns the norm **n** of a 4×4 matrix **A**. Equivalent to `norm(M A, 4, 4);`

See also: `norm`.

transp

Transpose of a matrix. Contained in `spacelib.c`

Calling sequence:	`transp (M A, M At, d1, d2);`
Prototype:	`void transp (MAT A, MAT At, int d1, int d2);`
Input parameters:	`MAT A; int d1, d2.`
Output parameters:	`MAT At.`

Builds the transpose **At** of a matrix **A** (**d1**×**d2**). Matrix **A** must have **d1** rows and **d2** columns, **At** must have **d1** columns and **d2** rows. Matrices **A** and **At** must be recasted using the `M` operator (see § 2.6).

See also: `transp3`, `transp4`.

transp3

Transpose of a 3×3 matrix. Macro contained in `spacelib.h`

Calling sequence:	`transp3 (A, At);`
Prototype:	`void transp3 (MAT3 A, MAT3 At);`
Input parameters:	`MAT3 A.`
Output parameters:	`MAT3 At.`

Puts in **At** the transpose of **A** (for 3×3 matrices).
Equivalent to `transp(M A, M At, 3, 3);`

See also: `transp`

transp4

Transpose of a 4×4 matrix. Macro contained in `spacelib.h`

Calling sequence:	`transp4 (A, At);`
Prototype:	`void transp4 (MAT4 A, MAT4 At);`
Input parameters:	`MAT4 A.`
Output parameters:	`MAT4 At.`

Puts in **At** the transpose of **A** (for 4×4 matrices).
Equivalent to `transp(M A, M At, 4, 4);`

See also: `transp`

pseudo_inv

Pseudo inverse of a matrix. Contained in `spaceli4.c`

Calling sequence:	`rank = int pseudo_inv (M A, M Api, rows, cols);`
Prototype:	`int pseudo_inv (MAT A, MAT Api, int rows, int cols);`
Return value:	`int rank.`
Input parameters:	`MAT A; int rows, cols.`
Output parameters:	`MAT Api.`

Builds the pseudo-inverse matrix **Api** of a given **rows**×**cols** elements matrix **A** using the *Greville*'s algorithm performed by rows. The output matrix **Api** must be a **cols**×**rows** matrix. The matrices **A** and **Api** must be recasted using the `M` operator (see § 2.6). Returns the **rank** of the matrix.

3.8.3 General operations on vectors

clearv

Clear a vector (fill it with zeros). Macro contained in `spacelib.h`

Calling sequence:	`clearv (v);`
Prototype:	`void clearv (VECTOR v);`
Output parameters:	`VECTOR v.`

Fills with zeros the vector **v**. Equivalent to `clear(M v,3,1);`

cross

Cross product. Contained in `spacelib.c`

Calling sequence:	`cross (a, b, c);`
Prototype:	`void cross (VECTOR a, VECTOR b, VECTOR c);`
Input parameters:	`VECTOR a, b.`
Output parameters:	`VECTOR c.`

Evaluates the cross product of two vectors (`c=a`×`b`).

See also: `crossmtom`.

dot

3 elements vector dot (scalar) product. Contained in `spacelib.c`

Calling sequence:	`d = dot (a, b);`
Prototype:	`real dot (VECTOR a, VECTOR b);`
Return value:	`real d.`
Input parameters:	`VECTOR a, b.`

Returns the value of the dot product **d** of two 3 element vectors **a** and **b** (`d=a·b`).

dot2

any elements vector dot (scalar) product. Contained in `spaceli4.c`

Calling sequence:	`d = real dot2 (a, b, dim);`
Prototype:	`real dot2 (real *a, real *b, int dim);`
Return value:	`real d.`
Input parameters:	`real a[], b[]; int dim.`

Returns the value of the dot product **d** of two **dim** element vectors **a** and **b** (**d**=**a**·**b**).

mod

Module of a vector. Contained in `spacelib.c`

Calling sequence:	`m = mod (a);`
Prototype:	`real mod (VECTOR a)`
Return value:	`real m.`
Input parameters:	`VECTOR a.`

Returns the module of vector **a** ($m = |a|$).

unitv

Unit vector. Contained in `spacelib.c`

Calling sequence:	`m = unitv (v, u);`
Prototype:	`real unitv (VECTOR v, AXIS u);`
Return value:	`real m.`
Input parameters:	`VECTOR v.`
Output parameters:	`AXIS u.`

Extracts the unit vector **u** of a vector **v** ($u = v/|v|$) and returns the module **m** of the vector. If $v = [\,0,\,0,\,0\,]$ is $u = [\,0,\,0,\,0\,]$.

vector

Evaluate a vector (from module and direction). Contained in `spacelib.c`

Calling sequence:	`vector (u, mdl, v);`
Prototype:	`void vector (AXIS u, real mdl, VECTOR v);`
Input parameters:	`AXIS u; real mdl.`
Output parameters:	`VECTOR v.`

Evaluates a vector **v** which has **mdl** as module and **u** as unit vector ($v = mdl \cdot u$).

3.9 Copy functions

mcopy

Matrix copy. Contained in `spacelib.c`

Calling sequence:	`mcopy (M A1, M A2, d1, d2);`
Prototype:	`void mcopy (MAT A1, MAT A2, int d1, int d2);`
Input parameters:	`MAT A1; int d1, d2.`
Output parameters:	`MAT A2.`

Copies a **d1**×**d2** matrix **A1** into **A2** ($A2 = A1$). **A1** and **A2** must be recasted using the `M` operator (see § 2.6).

See also: `mcopy3`, `mcopy4`, `vcopy`, `pcopy`, `mmcopy`, `mvcopy`.

mcopy3

3×3 matrix copy. Macro contained in `spacelib.h`

Calling sequence:	`mcopy3 (A, B);`
Prototype:	`mcopy3 (MAT3 A, MAT3 B);`
Input parameters:	`MAT3 A.`
Output parameters:	`MAT3 B.`

Copies the 3×3 matrix **A** into the 3×3 matrix **B**.
Equivalent to `mcopy(M A, M B, 3, 3);`
See also: `mcopy`

mcopy4

4×4 matrix copy. Macro contained in `spacelib.h`

Calling sequence:	`mcopy4 (A, B);`
Prototype:	`mcopy4 (MAT4 A, MAT4 B);`
Input parameters:	`MAT4 A.`
Output parameters:	`MAT4 B.`

Copies the 4×4 matrix **A** into the 4×4 matrix **B**.
Equivalent to `mcopy(M A, M B, 4, 4);`

See also: `mcopy`

vcopy

3 elements vector copy. Macro contained in `spacelib.h`

Calling sequence:	`vcopy (v1, v2);`
Prototype:	`vcopy (VECTOR v1, VECTOR v2);`
Input parameters:	`VECTOR v1.`
Output parameters:	`VECTOR v2.`

Copies the 3 element vector **v1** into the 3 element vector **v2**.
Equivalent to `mcopy(M v1, M v2, 3, 1);`

See also: `mcopy`

pcopy

Point copy. Macro contained in `spacelib.h`

Calling sequence:	`pcopy (P1, P2);`
Prototype:	`pcopy (POINT P1, POINT P2);`
Input parameters:	`POINT P1.`
Output parameters:	`POINT P2.`

Copies the point **P1** into the point **P2**.
Equivalent to `mcopy(M P1, M P2, 4, 1);`

See also: `mcopy`

mmcopy

Copy a part of a matrix. Contained in `spaceli3.c`

Calling sequence:	`mmcopy (M A, M B, d1, d2, im, jm);`
Prototype:	`void mmcopy (MAT A, MAT B, int d1, int d2, int im, int jm);`
Input parameters:	`MAT A; int d1, d2, im, jm.`
Output parameters:	`MAT B.`

Copies the **im**×**jm** upper-left part of matrix **A** into matrix **B**. **d1** is number of columns of **A** (dimension of **A**), **d2** is number of columns of **B** (dimension of **B**). Matrices **A**, **B** must be recasted using the `M` operator (see § 2.6).

See also: `mcopy`, `mvcopy`, `mcopy34`, `mcopy43`.

mcopy34

Copy a 3×3 matrix into a 4×4 matrix. Macro contained in `spacelib.h`

Calling sequence:	`mcopy34 (A, B);`
Prototype:	`void mcopy34 (MAT3 A, MAT4 B);`
Input parameters:	`MAT3 A.`
Output parameters:	`MAT4 B.`

Copies the 3×3 matrix **A** into the 3×3 upper-left submatrix of a 4×4 matrix **B**.
Equivalent to `mmcopy(M A, M B, 3, 4, 3, 3);`

See also: mmcopy

mcopy43

Copy a 3×3 submatrix into a 3×3 matrix. Macro contained in `spacelib.h`

Calling sequence:	`mcopy43 (A, B);`
Prototype:	`void mcopy43 (MAT4 A, MAT3 B);`
Input parameters:	`MAT4 A.`
Output parameters:	`MAT3 B.`

Copies the 3×3 upper-left submatrix of a 4×4 matrix **A** into a 3×3 matrix **B**. Equivalent to `mmcopy(M A, M B, 4, 3, 3, 3);`

See also: mmcopy

mvcopy

Copy a row or a column from a matrix. Contained in `spaceli4.c`

Calling sequence:	`rv = mvcopy (M A, rows, cols, val, type, v);`
Prototype:	`int mvcopy (MAT A, int rows, int cols, int val, int type, real *v);`
Return value:	`int rv.`
Input parameters:	`MAT A; int rows, cols, type, val.`
Output parameters:	`real v[ ].`

Copies a row or a column from a **rows**×**cols** matrix **A** into a vector **v**. **val** defines which row or column of the matrix must be extracted (range from 1 to rows/cols) while **type** identifies whether a row or a column must be copied into **v**. The value of **type** must be either `Row` or `Col` (see also § 2.4.3). The function returns the value **rv** that can be either `OK` or `NOTOK` (see § 2.4.3) in order to specify the correct outcome. Not all the problems can be detected by the function.The matrix **A** must be recasted using the `M` operator (see § 2.6). `mvcopy` performs the inverse operation than `vmcopy`.

See also: mcopy, mmcopy.

vmcopy

Copy a vector into a row or a column of a matrix. Contained in `spaceli4.c`

Calling sequence:	`rv = vmcopy (v, dim, val, type, M A, rows, cols);`
Prototype:	`int vmcopy (real *v, int dim, int val, int type, MAT A, int rows, int cols);`
Return value:	`int rv.`
Input parameters:	`real v[ ], int dim, type, val, rows, cols.`
Output parameters:	`MAT A.`

Copies a vector **v** into a row or a column of a **rows**×**cols** matrix **A**. **val** defines which row or column of the matrix must be copied to (range from 1 to rows/cols) while **type** identifies whether a row or a column must be substituted with **v**. The value of **type** must be either `Row` or `Col` (see also § 2.4.3). Matrix **A** must be recasted using the `M` operator (see § 2.6). The function returns the value **rv** that can be either `OK` or `NOTOK` (see § 2.4.3) in order to specify the correct outcome. `vmcopy` performs the inverse operation than `mvcopy`.

See also: mcopy, mmcopy.

3.10 Print functions

fprintm3

Print a 3×3 matrix (with a comment) on a file. Contained in `spacelib.c`

Calling sequence:	`fprintm3 (out, s, A)`
Prototype:	`void fprintm3 (FILE *out, char *s, MAT3 A);`
Input parameters:	`char s[ ]; MAT3 A.`
Output parameters:	`FILE out.`

Prints in a file a 3×3 matrix **A** preceded by the comment contained in **s**.

See also: `printm`, `fprintm4`.

fprintm4

Print a 4×4 matrix (with a comment) on a file. Contained in `spacelib.c`

Calling sequence:	`fprintm4 (out, s, A);`
Prototype:	`void fprintm4 (FILE *out, char *s, MAT4 A);`
Input parameters:	`char s[ ]; MAT4 A.`
Output parameters:	`FILE out.`

Prints in a file a 4×4 matrix **A** preceded by the comment contained in **s**.

See also: `printm`, `fprintm3`.

fprintv

Print vector (with a comment) on a file. Contained in `spacelib.c`

Calling sequence:	`fprintv (out, s, v, n);`
Prototype:	`void fprintmv (FILE *out, char *s, real *v, int n);`
Input parameters:	`char s[ ]; real v[ ]; int n.`
Output parameters:	`FILE out.`

Prints in a file a vector **v** whose dimension is **n** preceded by the comment contained in **s**.

See also: `printv`.

prmat

Prints position matrix for graphics post processors. Contained in `spaceli3.c`

Calling sequence:	`prmat (grpout, str, m);`
Prototype:	`void prmat (FILE *grpout, char *str, MAT4 m);`
Input parameters:	`char str[ ]; MAT4 m.`
Output parameters:	`FILE grpout.`

Writes to a file a position matrix **m** with the convention of `GRP_MAN` graphics post processor. Matrix **m** is written into file **grpout** preceded by string **str**.

printm

Prints 3×3 matrices. - obsolete Macro contained in `spacelib.h`

Calling sequence:	`printm (str, A);`
Prototype:	`void printm (char *str, MAT3 A);`
Input parameters:	`char str[ ]; MAT3 A.`

Obsolete. Provide just for compatibility. Please use `printm3`.

printm3

Prints 3×3 matrices. Macro contained in `spacelib.h`

Calling sequence:	`printm3 (str, A);`
Prototype:	`void printm3 (char *str, MAT3 A);`
Input parameters:	`char str[ ]; MAT3 A.`

Prints in the standard output a 3×3 matrix preceded by the comment contained in **str**. Equivalent to `fprintm3(stdout, str, A);`

printm4

Prints 4×4 matrices. Macro contained in `spacelib.h`

Calling sequence:	`printm4 (str, A);`
Prototype:	`void printm (char *str, MAT4 A);`
Input parameters:	`char str[ ]; MAT4 A.`

Prints in the standard output a 4×4 matrix preceded by the comment contained in **str**. Equivalent to `fprintm4(stdout, str, A);`

printv

Prints a vector. Macro contained in `spacelib.h`

Calling sequence:	`printv (str, v, dim);`
Prototype:	`void printv (char *str, real *v, int dim);`
Input parameters:	`char str[ ]; real v[ ]; int dim.`

Prints in the standard output a vector **v** of **dim** elements preceded by the comment contained in **str**. Equivalent to `fprintv(stdout, str, v, dim);`

printmat

Prints a real elements matrix. Contained in `spaceli4.c`

Calling sequence:	`printmat (s, M A, idim, jdim, imax, jmax);`
Prototype:	`void printmat (char *s, MAT A, int idim, int jdim, int imax, int jmax);`
Input parameters:	`char s[ ]; MAT A; int idim, jdim, imax, jmax.`

Prints in the standard output a submatrix of **idim**×**jdim** elements of a **imax**×**jmax** matrix **A** preceded by the comment contained in **s**. The matrix **A** must be recasted using the `M` operator (see § 2.6). The matrix **A** contains real elements.

iprintmat

Prints an integer elements matrix. Contained in `spaceli4.c`

Calling sequence:	`iprintmat (s, A, idim, jdim, imax, jmax);`
Prototype:	`void iprintmat (char *s, int *A, int idim, int jdim, int imax, int jmax);`
Input parameters:	`char s; int A[ ][ ]; int idim, jdim, imax, jmax.`

Prints in the standard output a submatrix of **idim**×**jdim** elements of a **imax**×**jmax** matrix **A** preceded by the comment contained in **s**. The matrix **A** contains integer elements.

3.11 Machine precision

dzerom

Double Machine's zero. Contained in `spacelib.c`

Calling sequence:	`e = dzerom ();`
Prototype:	`double dzerom (void);`
Return value:	`double e.`

Returns the double machine precision **e**[6]. This is the smallest double value e for which

$$1 + \epsilon \neq 1$$

See also: `fzerom`, `zerom`.

fzerom

Float Machine's zero. Contained in `spacelib.c`

Calling sequence:	`e = fzerom ();`
Prototype:	`float fzerom (void)`
Return value:	`float e.`

Returns the `float` machine precision **e**[7]. This is the smallest float value e for which

$$1 + \epsilon \neq 1$$

See also: `dzerom`, `zerom`.

zerom

Machine's zero. Contained in `spacelib.c`

Calling sequence:	`e = zerom ()`
Prototype:	`real zerom (void)`
Return value:	`real e`

Returns the value of the smallest `real` (see § 2.1) number **e**[8] for which

$$1 + \epsilon \neq 1$$

(also called the "*machine precision*"). Function `zerom()` is equivalent to `fzerom()` or to `dzerom()` depending of the definition of the type real.
See also: `fzerom`, `dzerom`.

[6]Function `dzerom` could fail if the compiler performs unwanted optimizations. In this case, recompile the function disabling the code optimization.
[7]Function `fzerom` could fail if the compiler performs unwanted optimizations. In this case, recompile the function disabling the code optimization.
[8]Function `zerom` could fail if the compiler performs unwanted optimizations. In this case, recompile the function disabling the code optimization.

Chapter 4

Linear Systems and inverses of matrices

Three functions are supplied for the resolution of linear systems: `solve`, `minvers` and `linear`.

- `solve` is useful in standard situations: square non singular coefficient matrix and a single right-hand vector.
- `minvers` evaluates the inverse of a matrix solving a particular system.
- `linear` is much more general, it deals also with rectangular coefficient matrices and more than one right-hand vectors to be handled at once.

To evaluate the pseudo-inverse of a matrix please use the SpaceLib© function `pseudo_inv` (see § 3.8.2).

4.1 Function solve

4.1.1 General description

This function can be considered a particularization of the more general function linear. It is useful in order to evaluate the solution of a linear system in the form

$$A \cdot x = b \tag{4.1}$$

where A is a square full rank matrix and b is the right-hand side vector.

`solve` uses function `linear` in order to obtain the system solution, but it also performs a reorder on the result it gets back from this function. This means that the user will be able to solve the most common system class elements using the more complex function linear in a very simple way.

4.1.2 Calling list

solve

solution of a linear system. Contained in `linear2.c`

Calling sequence:	`r = solve (M A, b, x, dim);`
Prototype:	`int solve (MAT A, real *b, real *x, int dim);`
Return value:	`int r.`
Input parameters:	`MAT A; real b[ ]; int dim.`
Output parameters:	`real x[ ].`

A: **dim**×**dim** matrix of the coefficients.

b: right-hand side vector.

dim: matrix and system dimension.

x: vector solution of the system.

r: the rank of the matrix

Matrix **A** must be recasted using the `M` operator (see § 2.6).

4.2 Function `minvers`

4.2.1 General description

Like `solve`, also this function can be considered a particularization of the more general function `linear`. So, `minvers` finds the inverse of a square matrix A solving a particular system. It uses the general property that, whenever the rank of A is equal to its dimensions, the equation

$$A \cdot x = I \tag{4.2}$$

where I is the identity matrix, has one single solution x. This solution is the inverse of A

$$x = A^{-1} \tag{4.3}$$

4.2.2 Calling list

`minvers`

Finds the inverse of a square matrix. Contained in `linear2.c`

Calling sequence:	`r = minvers (M A, M Ai, dim);`
Prototype:	`int minvers (MAT A, MAT Ai, int dim);`
Return value:	`int r.`
Input parameters:	`MAT A; int dim.);`
Output parameters:	`MAT Ai.`

A: **dim**×**dim** initial matrix.

dim: matrix dimension.

Ai: inverse matrix.

r: the rank of the matrix

Matrices **A** and **Ai** must be recasted using the `M` operator (see § 2.6).

4.3 Function `linear`

4.3.1 General description

This section contains information about function `linear`. This function allows the solution of a linear system by using a double pivoting elimination method. The linear system must be in the form

$$A \cdot x = b \tag{4.4}$$

where A is the matrix of coefficients and b is the right-hand side. A is generally a square `n`×`n` matrix, while b is an `n` element column vector. To use function `linear` in order to evaluate vector x, both A and b must be memorized in the same matrix (i.e. matrix H). More than one system with the same matrix A can be solved at the same time; for instance, the following systems can be handled at once:

$$A \cdot x_1 = b_1 \qquad A \cdot x_2 = b_2 \qquad A \cdot x_3 = b_3 \qquad \ldots \qquad A \cdot x_h = b_h \qquad \ldots \qquad A \cdot x_k = b_k \tag{4.5}$$

To solve the systems, matrix A and vectors b_i (at least one must be present) must be stored in the same `r`×`c` matrix H according to the scheme of figure 4.1. During the solution of the system, matrix A is replaced by an identity matrix and vectors b_i are replaced by the solutions of the correspondent system (i.e. x_i replaces b_i). However the order of the elements of each x_i is changed by the double pivoting algorithm and so the solution vectors must be reordered (see § 4.3.2 and § 4.3.4).

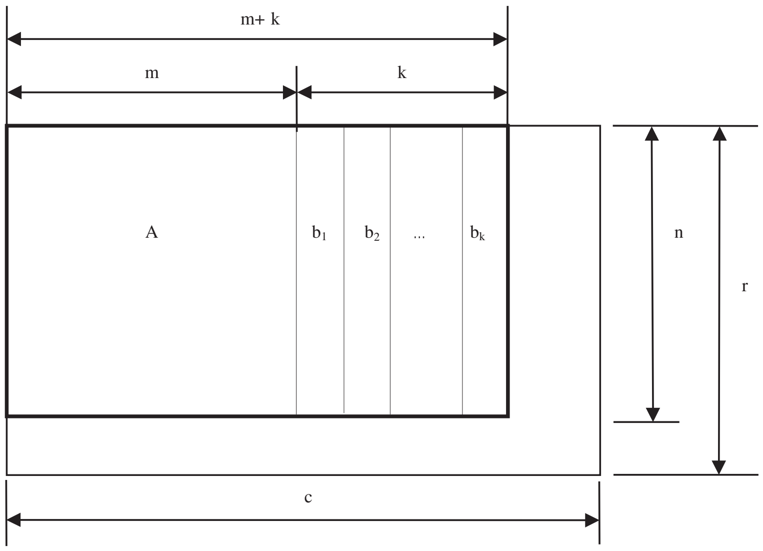


Figure 4.1: Definitions for parameters of function `linear`.
Generally `m`=`n`, `k`=1; always `r`≥`n`, `c`≥`m`+`k`.

4.3.2 Calling list

`linear`

Solve a linear system. Contained in `linear.c`

Calling sequence:	`linear (M H, idim, jdim, imax, jmax, nsol, ivet, &irank, &arm, vpr);`
Prototype:	`void linear (MAT H, int idim, int jdim, int imax, int jmax, int nsol, int *ivet, int *irank, real *arm, int *vpr);`
Input parameters:	`MAT H; int idim, jdim, imax, jmax, nsol; int vpr[ ]);`
Output parameters:	`int ivet[ ]; int *irank; real *arm.`

H: matrix containing matrix A and vectors b_i.

idim, **jdim**: the physical dimensions of H (`r` and **c** in figure 4.1).

imax, **jmax**: the logical dimensions of A (`n` and **m** in figure 4.1).

nsol: the number of the right-hand vectors (`k` in figure 4.1).

ivet: vectors of `m` integers that gives information necessary to reorder the elements of x. `ivet[i]==k` means that the value of the `k`th element of x is stored in position `i` of b. A standard way to reorder the solution putting the solution of the `k`th system ($k = 0, 1, \ldots$) in a vector x is as follows:

```
for(i=0;i<n;i++) x[ivet[i]]=H[i][n+k];
```

irank: an estimation of the rank of matrix A.

arm: the absolute value of the greater element of A during the last pass of elimination.

vpr: vector specifying which variables must be considered as main variables. That means that they will be forced in the first positions of vector **ivet** and so of vectors x. The list of the variables must be terminated by a value -1. So, for instance if `m`≥8, then a valid value for **vpr** is {3;5;0;7;-1}. In the usual cases, **vpr** is a vector containing only one element whose value is -1:

```
static int vpr[1] = {-1}
```

This allows the algorithm to perform full pivoting on all the variables.

Matrix **H** must be recasted using the `M` operator (see § 2.6).

NOTE: function `linear` can be also used to detect if a general linear system has or has not solution. In this case matrix A can have a number of columns which is different from the number of the rows (`n`$\neq$`m`). In this case function `linear` transforms the system into an equivalent system (i.e. which has the same solutions). After the execution of `linear` matrix A and vectors b_i will be in the block form of figure 4.2.

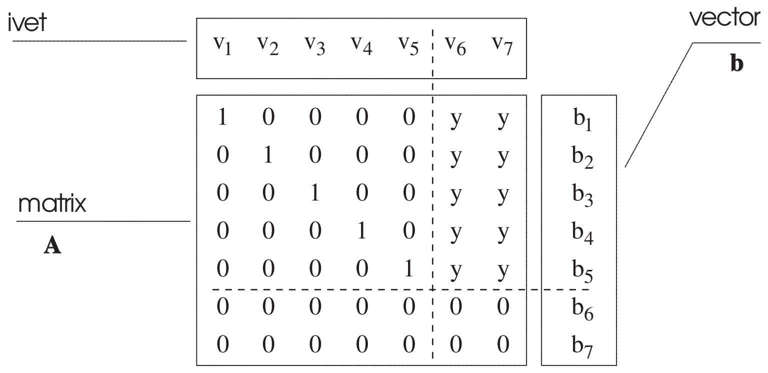


Figure 4.2: Final form of a linear system after a call to function `linear` with a 7×7 matrix.

0, 1	are elements whose value is 0 or 1.
y	are elements whose value depends on the system.
$\mathbf{b}_i$	are elements of vector **b**.
$\mathbf{v}_i$	are elements of vector **ivet**.

Table 4.1: Notation used for figure 4.2

All the blocks can have any dimension and may not be present depending on the coefficients stored in matrix A. If in the last rows of the matrix the block of zeros is present, generally the system has not solution (over determined system) unless the last elements of b after the transformation are null (the elements correspondent to the block of zeros, that is b_6 and b_7 in Figure 4.2). If the block of y is present but the block of zeros is not present, the system has an infinite number of solutions that can be found assigning an arbitrary value at the last elements of x (the elements corresponding to the block of y, that is v_6 and v_7 in Figure 4.2).

As an example if the system is as follows:

$$A = \begin{bmatrix} 1 & 1 & 1 & 2 & 1 \\ 0 & 1 & 1 & 2 & 1 \\ 0 & 0 & 2 & 2 & 1 \\ 0 & 0 & 2 & 2 & 1 \end{bmatrix}; \quad X = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix}; \quad b = \begin{bmatrix} 6 \\ 5 \\ 3 \\ 3 \end{bmatrix} \tag{4.6}$$

Function linear detects that matrix A has a rank of 3 and transforms it in the form of figure 4.3. That means that the rank of A is 3, the system has an infinitive number of solutions and that you can assign any value to elements 1 and 4 of x (a_1 and a_4) and then evaluate the corresponding value of the others (a_0, a_2 and a_3). If you put a_1=a_4=0, then the value of the other elements are directly stored in b ($a_3 = 3.5, a_0 = 1, a_2 = -2$). If a_1=7 and a_4=4 then a_3=-5.5, a_0=1 and a_2=5.

4.3.3 Sample program to solve a linear system `TEST-LIN`

As an example of the usage of function `linear`, let us consider the sample program `TEST-LIN` which reads and solves a linear system whose matrix of coefficient and whose right-hand side vector are memorized in file `INP.DAT` (See § 4.3.4). The program reads the dimension of the matrix, the matrix itself and the vector and prints the solution if it exists. If the matrix is not a square full rank matrix, the program prints the resulting matrix and vector after the transformation.

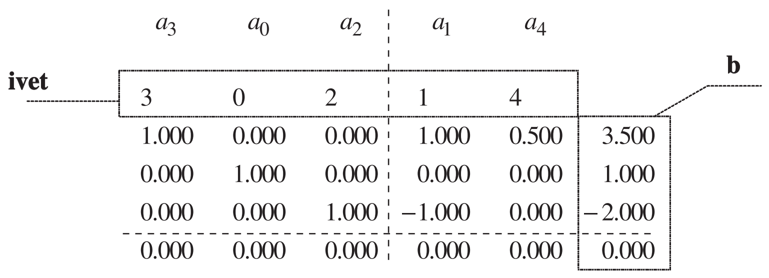


Figure 4.3: Numerical example of the output of function `linear`.

The input matrix A and vector b for the example just presented are

$$A = \begin{bmatrix} 1 & 1 & 1 & 2 & 1 \\ 0 & 1 & 1 & 2 & 1 \\ 0 & 0 & 2 & 2 & 1 \\ 0 & 0 & 2 & 2 & 1 \end{bmatrix} \qquad b = \begin{bmatrix} 6 \\ 5 \\ 3 \\ 3 \end{bmatrix} \tag{4.7}$$

while file `INP.DAT` is filled with the values listed in table 4.2.

FILE `INP.DAT` (*DATA_FILE*)						*MEANING*
4	5					Matrix dimensions
1 0 0 0	1 1 0 0	1 1 2 2	2 2 2 2	1 1 1 1	6 5 3 3	value of A and b

Table 4.2: Content of the file `INP.DAT`

4.3.4 The program (`TEST-LIN`)

NOTE: To run this program the type `real` must be set equivalent to the type `float` (see § 2.1)

```
/*
    TEST-LIN: Sample program which reads and solves a linear system
              whose matrix of coefficient and whose right-hand side vector
              are memorized in file INP.DAT
              A*x=b
*/
/* Note: To compile this program the type real must be set equivalent to the type
   float (see also User's Manual). This is necessary because the formatting string
   of the fscanf function have been written using %f as descriptor. */
#include <stdio.h>
#include <stdlib.h>
#include "spacelib.h"
#include "linear.h"
#define Nmax 6
#define Mmax 7
void main(void)
{     real H[Nmax][Mmax+1], A[Nmax][Mmax], x[Mmax], b[Nmax];
      int ivet[Mmax],vpr[1];
      int i,j,irank,n,m;
      real arm;
      double t;
      FILE *f;
      vpr[0]=-1;
```

```
        /* read matrix of coefficients and right-hand side vector from file */
    f=fopen("inp.dat","r");
    if(f==NULL)
    {     printf("Error on input file\a");
          exit(1);
    }
    fscanf(f,"%d %d",&n,&m);                /* read dimension of matrix */
    if(n>Nmax || m>Mmax)
    {     printf("ERROR: matrix too big");
          exit(1);
    }
    for(i=0;i<n;i++)                        /* read matrix and vector */
    {     for(j=0;j<m;j++)
          {     fscanf(f,"%lf",&t);
                A[i][j]=(real)t;
          }
          fscanf(f,"%lf",&t);
          b[i]=(real)t;
    }
    for(i=0;i<n;i++)
          for(j=0;j<m;j++)                  /* copy matrix of coefficients */
                H[i][j]=A[i][j];
    for(j=0;j<n;j++)                        /* copy right-hand side vector */
          H[j][m]=b[j];
    printf("\n Matrix and right-hand side vector\n\n");
    for(i=0;i<n;i++)
    {     for(j=0;j<m+1;j++)
                printf("%7.3f ",H[i][j]);
          printf("\n");
    }
    printf("\n");
    linear(M H,Nmax,Mmax+1,n,m,1,ivet,&irank,&arm,vpr); /* solve system */
    if(n==m && irank==n)
    {     printf("The solution is\n");
          for(i=0;i<n;i++)                  /* reorder solution */
                x[ivet[i]]=H[i][n];
          for(i=0;i<n;i++)                  /* output results */
                printf("%f ",x[i]);
    }
    else
    {     if(n!=m)
                printf("The number of rows and columns are different\n");
          printf("The rank of the matrix is %d\n\n",irank);
          for(i=0;i<m;i++)
                printf("%7d ",ivet[i]);
          printf("\n\n");
          for(i=0;i<n;i++)
          {     for(j=0;j<m+1;j++)
                      printf("%7.3f ",H[i][j]);
                printf("\n");
          }
          printf("\n");
    }
    fcloseall();
}
```

Chapter 5

Direct Dynamics: function dyn_eq

5.1 General discussion

This paragraph discusses function dyn_eq (see also § 3.3).
This function solves the equations

$$\Phi = skew\{\dot{W} \cdot J\} \tag{5.1}$$
$$\Gamma = skew\{W \cdot J\} \tag{5.2}$$

since they have the same form, we will discuss only the first one.

Generally in the first equation the unknown is $\dot{W}$, while in the second one is W. Both matrix Φ and $\dot{W}$ have six independent values; so in total they have 12 elements. If J is known and a total of 6 elements out of the 12 of Φ and $\dot{W}$ are known, it is possible to evaluate the others. To perform this operation function dyn_eq needs informations about which values are known and which values must be evaluated. This is performed by the 2×6 integer matrix `var`.
`var` must be filled by six "1" to indicate the elements to be evaluated and by six "0" to indicate the elements which are known. The first line is relative to the acceleration (angular and linear) and the second one to the actions (torques and forces). The first three columns are relative to angular terms (angular acceleration and torques) and the second to linear terms (acceleration and forces) according to this scheme:

$$var = \left[\begin{array}{ccc|ccc} \dot{w}_x & \dot{w}_y & \dot{w}_z & a_x & a_y & a_z \\ \hline t_x & t_y & t_z & f_x & f_y & f_z \end{array}\right] \tag{5.3}$$

so in the usual cases in which Φ is known and it's necessary to evaluate $\dot{W}$, matrix `var` must be filled in the following way:

$$var = \left[\begin{array}{ccc|ccc} 1 & 1 & 1 & 1 & 1 & 1 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \tag{5.4}$$

Opposite, if $\dot{W}$ is known and you want to evaluate Φ, the correct values are:

$$var = \left[\begin{array}{ccc|ccc} 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 1 & 1 & 1 & 1 & 1 & 1 \end{array}\right] \tag{5.5}$$

in this last special case, dyn_eq gives the same results as function `skew(Wp,J)`. In every case, you must have exactly one "1" and one "0" in every column of `var`.

5.2 The Calling List.

The calling list (see § 3.3) for function dyn_eq to solve equation (5.1) is:

```
dyn_eq (J, Wp, F, var)
```

where:

J	4×4 inertia matrix of the body
Wp	4×4 acceleration matrix containing $\dot{W}$
F	4×4 action matrix containing Φ
var	2×6 matrix which specifies which elements of $\dot{W}$ and Φ are unknown

The calling list for function dyn_eq to solve equation (5.2) is:

```
dyn_eq(J, W, G, var);
```

where:

J	4×4 inertia matrix of the body
W	4×4 velocity matrix containing W
G	4×4 momentum matrix containing Γ
var	2×6 matrix which specifies which elements of W and Γ are unknown

dyn_eq returns value OK if the operation could be performed properly or NOTOK if an error had been detected. OK and NOTOK are constants defined in spacelib.h (see also § 6.1). An error occurs if dyn_eq is called with parameters that have not physical meanings (e.g. J is not positive defined or both $\dot{W}_i$ and t_i or a_i and f_i are unknown for any i).

Chapter 6

Header files

In this section are listed the header files `spacelib.h` and `linear.h`.

6.1 The header spacelib.h

```
        /* ============= MATH MACRO, CONSTANTS AND TYPE DEFINITION ================ */

#ifndef _SPACELIB_                                           /* revision february 2006*/
#define _SPACELIB_
#ifdef __TURBOC__
#define _BORLAND_
#endif
#ifdef FLOAT
typedef float real;
#else
typedef double real;
#endif
#define aabs(a)   ((double)(a)>0.  ? (a) : -(a))
                  /* conditional compilation to avoid conflict with standard definitions of min
                     and max. SPACELIB (C) definitions coincide with ANSI definitions */
#ifndef max
#define max(a,b) ((a)>(b) ? (a) : (b))
#endif
#ifndef min
#define min(a,b) ((a)<(b) ? (a) : (b))
#endif
#define sign(x) ((x)<0 ? -1 : ((x)>0 ? 1 :0))
#define  PIG_2 1.570796326794897
#define  PIG   3.141592653589793
#define  PIG2  6.283185307179586
#define rad(x)  ((x)/180.*PIG)          /* degrees to radiants */
#define deg(x)  ((x)*180./PIG)          /* radiants to degrees */
#define M (real *)                      /* to recast pointers when calling functions
                                          (used for adjustable dimension matrices) */
typedef real *MAT;                      /* generic matrix                 */
typedef real MAT3[3][3];                /* rotation submatrices           */
typedef real MAT4[4][4];                /* transform matrices (& others)  */
typedef real POINT[4];                  /* homogeneous points coordinates */
typedef real AXIS[3];                   /* unit vectors (axes)            */
typedef real VECTOR[3];                 /* vectors                        */
typedef real (*MAT4P)[4];               /* used to point to 4*4 matrices  */
typedef real (*MAT3P)[3];               /* used to point to 3*3 matrices  */
typedef struct {            /* Definition of the LINE type following the parametric definition*/
                POINT P;                /*  X = Xp+t*alpha ** Y = Yp+t*beta ** Z = Zp+t*gamma*/
                VECTOR dir;             /* where P=(Xp,Yp,Zp) and dir=[alpha,beta,gamma]*/
               } LINE, * LINEP;         /* LINEP is a pointer to LINE structure type */
```

```
typedef real PLANE[4];   /* PLANE type is defined as a four real element array. The first three
                            define a unit vector orthogonal to the plane itself. The fourth real
                            is the distance of plane  from the origin of reference frame.
                            The distance is positive when the vector directed from plane to
                            origin has the same direction of the plane unit vector */
#define OK      1                       /*    successful completation of a function */
#define NOTOK   0                       /* un-successful completation of a function */
#define SYMM    1                       /* denote symmetrical or skew-symmetrical matrices */
#define SKEW   -1
#define Row     0                       /* denote a matrix row */
#define Col     1                       /* denote a matrix column */
#define Rev     0                       /* Revolute (and screw) pairs */
#define Pri     1                       /* Prismatic (Sliding) pairs */
#define Tor     0                       /* Torque */
#define For     1                       /* Force  */
#define X  0                            /* homogeneous coordinates */
#define Y  1
#define Z  2
#define U  3
#define Xaxis  {1.,0.,0.}               /* positive axes unit vectors */
#define Yaxis  {0.,1.,0.}
#define Zaxis  {0.,0.,1.}
#define Xaxis_n  {-1.,0.,0.}            /* negative axes unit vectors */
#define Yaxis_n  {0.,-1.,0.}
#define Zaxis_n  {0.,0.,-1.}
#define ORIGIN {0,0,0,1}
                                        /* identity and null matrices 3*3 and 4*4 */
#define NULL3 {{0.,0.,0.}, {0.,0.,0.}, {0.,0.,0.}}
#define UNIT3 {{1.,0.,0.}, {0.,1.,0.}, {0.,0.,1.}}
#define NULL4 {{0.,0.,0.,0.}, {0.,0.,0.,0.}, {0.,0.,0.,0.}, {0.,0.,0.,0.}}
#define UNIT4 {{1.,0.,0.,0.}, {0.,1.,0.,0.}, {0.,0.,1.,0.}, {0.,0.,0.,1.}}

/* ============= MACRO FOR ADJUSTABLE-DIMENSIONS MATRICES =============== */
            /* Note : 3/4 suffixes refer to 3*3 or 4*4 matrices */
                                                    /* --- norm of a matrix --- */
#define norm3(a)           norm(M a,3,3)
#define norm4(a)           norm(M a,4,4)
                                                    /* --- Rotation and Position Matrices --- */
#define rotat23(a,q,R)     rotat2(a,q,M R,3);       /* rotation matrix */
                                             /* --- Matrix Transformations, Normalization --- */
#define normal3(a)         normal(M a,3)            /* normalizes rotat. matrices */
#define normal4(a)         normal(M a,4)  /* normalizes rot. part. of transformation matrices */
                                                  /* normalizes symm. and skew symm. matrices */
#define n_simm3(m)         norm_simm_skew(M m,3,3, SYMM);
#define n_simm34(m)        norm_simm_skew(M m,3,4, SYMM);
#define n_simm4(m)         norm_simm_skew(M m,4,4, SYMM);
#define n_skew3(m)         norm_simm_skew(M m,3,3, SKEW);
#define n_skew34(m)        norm_simm_skew(M m,3,4, SKEW);
#define n_skew4(m)         norm_simm_skew(M m,4,4, SKEW);
                                                                  /* standard D&H matrix */
#define DHtoMstd(theta,d,a,alpha,MM)      dhtom(0,theta,d,0.,a,alpha,0.,MM)

                                      /* --- Matrix Transformations, Change of Reference --- */
#define trasf_mamt4(a,b,c) trasf_mamt(M a,M b,M c,4) /* change of reference for 4*4
                                                        contra-variant matrices */
                                        /* --- Matrix transformations, General Operations --- */
#define mtov3(v,m)         mtov(M m,3,v)             /* skew matrix 3*3 to vector */
#define mtov4(v,m)         mtov(M m,4,v)             /* skew matrix 4*4 to vector */
#define vtom3(v,m)         vtom(v,M m,3)             /* vector[3] to 3*3 skew mat. */
#define vtom4(v,m)         vtom(v,M m,4)             /* vector[3] to 4*4 skew mat. */
#define skew4(a,b,c)       skew(M a,M b,M c,4)       /* performs c=skew(a,b) for 4*4 matrices */
                               /* --- Conversion from Cardan Angles to Matrices, Position --- */
```

```
#define cardantor3(a,i,j,k,m)   cardantor(a,i,j,k,M m,3)
#define cardantor4(a,i,j,k,m)   cardantor(a,i,j,k,M m,4)
#define rtocardan3(m,i,j,k,a,b) rtocardan(M m,3,i,j,k,a,b)
#define rtocardan4(m,i,j,k,a,b) rtocardan(M m,4,i,j,k,a,b)
#define cardtoM(q,O,m)          cardantoM(q,X,Y,Z,O,m);    /* pos. mat. from T-B angles */
#define eultoM(q,O,m)           cardantoM(q,Z,X,Z,O,m);    /* pos. mat. from Euler angles */
#define nauttoM(q,O,m)          cardantoM(q,Z,Y,X,O,m);    /* pos. mat. from nautical angles */
#define Mtocard(m,q1,q2)        Mtocardan(m,X,Y,Z,q1,q2);  /* T-B angles from pos. mat. */
#define Mtoeul(m,q1,q2)         Mtocardan(m,Z,X,Z,q1,q2);  /* Euler angles from pos. mat. */
#define Mtonaut(m,q1,q2)        Mtocardan(m,Z,Y,X,q1,q2);  /* nautical angles from pos. mat. */
                        /* --- Conversion from Cardan Angles to Matrices, Vel. and Acc. --- */
                                                          /* W from T-B angles and derivative */
#define cardtoW(q,qp,O,W)          cardantoW(q,qp,X,Y,Z,O,W);
                                                        /* W from Euler angles and derivative */
#define eultoW(q,qp,O,W)           cardantoW(q,qp,Z,X,Z,O,W);
                                                     /* W from nautical angles and derivative */
#define nauttoW(q,qp,O,W)          cardantoW(q,qp,Z,Y,X,O,W);
                                                       /* T-B angles and derivative from m, W */
#define Wtocard(m,W,q1,q2,qp1,qp2)    Wtocardan(m,W,X,Y,Z,q1,q2,qp1,qp2);
                                                     /* Euler angles and derivative from m, W */
#define Wtoeul(m,W,q1,q2,qp1,qp2)     Wtocardan(m,W,Z,X,Z,q1,q2,qp1,qp2);
                                                  /* nautical angles and derivative from m, W */
#define Wtonaut(m,W,q1,q2,qp1,qp2)    Wtocardan(m,W,Z,Y,X,q1,q2,qp1,qp2);
#define cardanto_omega3(a,ap,i,j,k,m) cardanto_omega(a,ap,i,j,k,M m,3)
#define cardanto_omega4(a,ap,i,j,k,m) cardanto_omega(a,ap,i,j,k,M m,4)
                                              /* H from T-B angles, first and second derivative */
#define cardtoH(q,qp,qpp,O,H)         cardantoH(q,qp,qpp,X,Y,Z,O,H);
                                            /* H from Euler angles, first and second derivative */
#define eultoH(q,qp,qpp,O,H)          cardantoH(q,qp,qpp,Z,X,Z,O,H);
                                         /* H from nautical angles, first and second derivative */
#define nauttoH(q,qp,qpp,O,H)         cardantoH(q,qp,qpp,Z,Y,X,O,H);
                                      /* T-B angles, first and second derivative from m, W, H */
#define Htocard(m,W,H,q1,q2,qp1,qp2,qpp1,qpp2) Htocardan(m,W,H,X,Y,Z,q1,q2,qp1,qp2,qpp1,qpp2);
                                    /* Euler angles, first and second derivative from m, W, H */
#define Htoeul(m,W,H,q1,q2,qp1,qp2,qpp1,qpp2)  Htocardan(m,W,H,Z,X,Z,q1,q2,qp1,qp2,qpp1,qpp2);
                                 /* nautical angles, first and second derivative from m, W, H */
#define Htonaut(m,W,H,q1,q2,qp1,qp2,qpp1,qpp2) Htocardan(m,W,H,Z,Y,X,q1,q2,qp1,qp2,qpp1,qpp2);
#define cardanto_G3(a,ap,app,i,j,k,m) cardanto_G(a,ap,app,i,j,k,M m,3)
#define cardanto_G4(a,ap,app,i,j,k,m) cardanto_G(a,ap,app,i,j,k,M m,4)
                            /* --- Construction of Frames Attached to Points or Vectors --- */
#define frameP3(p1,p2,p3,a1,a2,r)     frameP(p1,p2,p3,a1,a2, M r,3)
#define frameP4(p1,p2,p3,a1,a2,r)     frameP(p1,p2,p3,a1,a2, M r,4)
#define frameV3(v1,v2,a1,a2,r)        frameV(v1,v2,a1,a2, M r,3)
#define frameV4(v1,v2,a1,a2,r)        frameV(v1,v2,a1,a2, M r,4)
                  /* --- Operations on Matrices and Vectors, Matrices and Vectors Algebra --- */
#define molt3(a,b,c)       molt(M a,M b,M c,3,3,3)  /* mult. square matrices */
#define molt4(a,b,c)       molt(M a,M b,M c,4,4,4)
#define moltp(a,b,c)       molt(M a,M b,M c,4,4,1)  /* mult. pos. matrix and point */
#define moltmv3(A,v1,v2)   molt(M A,M v1,M v2,3,3,1); /* mult. 3x3 matrix and vector[3] */
#define rmolt3(a,r,b)      rmolt(M a,r,M b,3,3)     /* mult. scalar and sq. mat. */
#define rmolt4(a,r,b)      rmolt(M a,r,M b,4,4)
#define rmoltv(v1,r,v2)    rmolt(M v1,r,M v2,3,1);  /* mult. vector[3] by a scalar r */
#define sum3(a,b,c)        sum(M a,M b,M c,3,3)     /* sum of square matrices */
#define sum4(a,b,c)        sum(M a,M b,M c,4,4)
#define sumv(a,b,c)        sum(M a,M b,M c,3,1);    /* sum of vectors */
#define sub3(a,b,c)        sub(M a,M b,M c,3,3)     /* diff. of square matrices */
#define sub4(a,b,c)        sub(M a,M b,M c,4,4)
#define subv(a,b,c)        sub(M a,M b,M c,3,1);    /* diff. of vectors */
                  /* --- Operations on Matrices and Vectors, Gen. Operations on Matrices --- */
#define clear3(m)          clear(M m,3,3)           /* clears a 3*3 matrix */
#define clear4(m)          clear(M m,4,4)           /* clears a 4*4 matrix */
#define idmat3(m)          idmat(M m,3)             /* makes unitary 3*3 matrices */
```

```
#define idmat4(m)          idmat(M m,4)             /* makes unitary 4*4 matrices */
#define transp3(a,b)       transp(M a,M b,3,3)      /* transposte of a matrix */
#define transp4(a,b)       transp(M a,M b,4,4)
                   /* --- Operations on Matrices and Vectors, Gen. Operations on Vectors --- */
#define clearv(v)          clear(M v,3,1);          /* clears vector */
                                                                /* --- Copy Functions --- */
#define mcopy3(a,b)        mcopy(M a,M b,3,3)       /* copy of square mat. */
#define mcopy4(a,b)        mcopy(M a,M b,4,4)
#define mcopy34(a,b)       mmcopy(M a,M b,3,4,3,3)  /* copy 3*3 submatrix */
#define mcopy43(a,b)       mmcopy(M a,M b,4,3,3,3)
#define vcopy(v1,v2)       mcopy(M v1,M v2,3,1);    /* copy vectors */
#define pcopy(P1,P2)       mcopy(M P1,M P2,4,1);    /* copy points */
                                                               /* --- Print Functions --- */
#define printm(str,m)   fprintm3(stdout,str,m)      /* for compatibility */
#define printm3(str,m)  fprintm3(stdout,str,m)      /* print 3*3 matrices to stdout */
#define printm4(str,m)  fprintm4(stdout,str,m)      /* print 4*4 matrices to stdout */
#define printv(str,m,n) fprintv(stdout,str,m,n)     /* print vector to stdout */

        /* =========================== FUNCTIONS PROTOTYPES ===================== */

                                                  /* --- Rotation and Position Matrices --- */
int    dhtom(int jtype, real theta, real d, real b, real a, real alpha,real q, MAT4 m);
void   extract(MAT A, AXIS u, real *fi, int dim);
int    mtoscrew(MAT4 Q, AXIS u, real *fi, real *h, POINT P);
void   screwtom(AXIS u, real fi, real h, POINT P, MAT4 Q);
void   rotat(AXIS u, real fi, MAT A, int dim);                 /* build rotation matrix */
int    rotat2(int a, real q, MAT R, int dim);
int    rotat24(int a, real q, POINT O, MAT4 m);
void   rotat34(int a, real q, POINT P, MAT4 m);
                                                 /* --- Speed and Acceleration Matrices --- */
void   gtom(real gx, real gy, real gz, MAT4 Hg);
void   Gtomegapto (MAT3 G, VECTOR omegapto);/* extracts the angular velocity vector from the 3x3
                                        upper left submatrix G of the acceleration matrix H */
int    makeL(int jtype, AXIS u, real pitch, POINT P, MAT4 L);
int    makeL2(int jtype, int a, real pitch, POINT P, MAT4 L);
void   WtoL(MAT4 W, MAT4 L);                       /* W matrix to L matrix */
void   Wtovel(MAT4 W, AXIS u, real *omega, real *vel, POINT P);
int    velacctoWH(int jtype, real qp, real qpp, MAT4 W, MAT4 H);
int    velacctoWH2(int jtype, int a, real qp, real qpp, MAT4 W, MAT4 H);
                                   /* W and H matrices from axis, point and joint variable */
void   velacctoWH3(int jtype, int a, real qp, real qpp, POINT O, MAT4 W, MAT4 H);
                                                   /* --- Inertial and Actions Matrices --- */
void   actom(real fx, real fy, real fz, real cx, real cy, real cz, MAT4 FI);
void   jtoJ(real mass, real jxx, real jyy, real jzz, real jxy, real jyz,
            real jxz, real xg, real yg, real zg, MAT4 J);
                                            /* --- Matrix Transformations, Normalization --- */
int    normal(MAT R, int n);
int    norm_simm_skew(MAT A, int n, int dim, int sign);
                                     /* --- Matrix Transformations, Change of Reference --- */
void   trasf_mami(MAT4 A1, MAT4 m, MAT4 A2);
void   trasf_miam(MAT4 A1, MAT4 m, MAT4 A2);
void   trasf_mamt(MAT A1, MAT m, MAT A2, int dim);
void   trasf_miamit(MAT4 A1, MAT4 m, MAT4 A2);
                                      /* --- Matrix transformations, General Operations --- */
void   coriolis(MAT4 H0, MAT4 H1, MAT4 W0, MAT4 W1, MAT4 H);
void   invers(MAT4 m, MAT4 mi);
void   mtov(MAT A, int dim, VECTOR v);      /* 3*3 skew matrix to vector[3] */
void   vtom(VECTOR v, MAT A, int dim);      /* vector[3] to 3*3 skew matrix */
void   skew(MAT A, MAT B, MAT C, int dim);
real   trac_ljlt4(MAT4 L1, MAT4 J, MAT4 L2);
                              /* --- Conversion from Cardan Angles to Matrices, Position --- */
int    cardantor(real *q, int i, int j, int k, MAT A, int dim);
int     rtocardan(MAT R, int dim, int i, int j, int k, real q1[3], real q2[3]);
void    cardantoM(real *q, int i, int j, int k, POINT O, MAT4 m);
                                             /* Euler/Cardan angles from a position matrix m */
int     Mtocardan(MAT4 m, int i, int j, int k, real q1[3], real q2[3]);
                          /* --- Conversion from Cardan Angles to Matrices, Vel. and Acc. --- */
void    cardantoW(real *q, real *qp, int i, int j, int k, POINT O, MAT4 W);
    /* extracts the Euler/Cardan angles and their first time derivative from matrices m and W */
int     Wtocardan(MAT4 m, MAT4 W, int i, int j, int k, real q1[3], real q2[3], real qp1[3],
                  real qp2[3]);
void    cardanto_OMEGA(real *q, real *qp, int i, int j, int k, real *omega);
void    cardanto_omega(real *q, real *qp, int i, int j, int k, MAT A, int dim);
void    cardantoH(real *q, real *qp, real *qpp, int i, int j, int k, POINT O, MAT4 H);
                  /* extracts the Euler/Cardan angles and their first and second time derivative
                     from matrices m, W and H */
int     Htocardan(MAT4 m, MAT4 W, MAT4 H, int i, int j, int k, real q1[3], real q2[3],
                  real qp1[3], real qp2[3], real qpp1[3], real qpp2[3]);
void    cardanto_G(real *q, real *qp, real *qpp, int i, int j, int k, MAT A, int dim);
void    cardanto_OMEGAPTO(real *q, real *qp, real *qpp, int i, int j, int k, real *omegapto);
int     cardantol(real *q, int i, int j, int k, MAT R, int dim);
int     cardantoWPROD(real *q, int i, int j, int k, MAT R, int dim);
int     invA(real alpha, real beta, int sig, int i, int j, int k, MAT3 Ai);
void    WPRODtocardan(real alpha, real beta, int sig, int i, int j, int k, MAT3 Atil);
                              /* --- Construction of Frames Attached to Points or Vectors --- */
int     frameP(POINT P1, POINT P2, POINT P3, int a1, int a2, MAT A, int dim);
void    frame4P(POINT P1, POINT P2, POINT P3, int a1, int a2, MAT4 m);
int     frameV(VECTOR v1, VECTOR v2, int a1, int a2, MAT A, int dim);
void    frame4V(POINT P1, VECTOR v1, VECTOR v2, int a1, int a2, MAT4 m);
                        /* --- Working with Points Lines and Planes, Operations on Points --- */
real    angle(POINT P1, POINT P2, POINT P3);
real    dist(POINT P1, POINT P2);
void    intermediate(POINT P1, real a, POINT P2, real b, POINT P3);
void    middle(POINT P1, POINT P2, POINT P);                /* middle point with weights */
void    vect(POINT P1, POINT P2, VECTOR v);
                   /* --- Working with Points Lines and Planes, Op. on Lines and Planes --- */
void    line2p(POINT P1, POINT P2, LINEP l);                /* Builds line through P1,P2 */
void    linepvect(POINT P1, VECTOR v, LINEP l);    /* Builds line through P1, with direction v */
real    projponl(POINT P1, LINE l, POINT I);                /* projection of P on line l */
real    distpp(PLANE pl, POINT P);                          /* distance of P from plane */
real    project(POINT P, PLANE pl, POINT I);                /* projection of P on plane */
void    plane(POINT P1, POINT P2, POINT P3, PLANE pl);      /* plane through P1,P2,P3 */
void    plane2(POINT P1, VECTOR v, PLANE pl);        /* plane pl through P1 and unit vector v */
int     inters2pl(PLANE pl1, PLANE pl2, LINEP l);           /* intersection of two planes */
void    interslpl(LINE l, PLANE pl, POINT I, int *inttype); /*intersection of a line and plane */
void    intersection(LINE l1, LINE l2, LINEP lmindist, real * mindist,
                     PLANE pl, POINT I,int *inttype);       /* intersection of two lines */
                  /* --- Operations on Matrices and Vectors, Matrices and Vectors Algebra --- */
void    molt(MAT A, MAT B, MAT C, int d1, int d2, int d3);
void    rmolt(MAT A, real r, MAT B, int d1, int d2);
void    sum(MAT A, MAT B, MAT C, int d1, int d2);
void    sub(MAT A, MAT B, MAT C, int d1, int d2);
real    norm(MAT A, int d1, int d2);
                  /* --- Operations on Matrices and Vectors, Gen. Operations on Matrices --- */
void    clear(MAT A, int id, int jd);
void    idmat(MAT A, int ijd);
void    transp(MAT A, MAT At, int d1, int d2);
int     pseudo_inv(MAT A, MAT Api, int rows, int cols);
                    /* --- Operations on Matrices and Vectors, Gen. Operations on Vectors --- */
void    cross(VECTOR a, VECTOR b, VECTOR  c);   /* cross product c = a x b */
real    dot(VECTOR a, VECTOR b);                /* dot product of 3 element vector */
real    dot2(real *v1, real *v2, int dim);      /* dot product of dim element vector */
real    mod(VECTOR a);                          /* module of a vector */
real    unitv(VECTOR v, AXIS u);                /* evaluate the unit vector u of a vector v
```

```
                                                 and it also returns its module */
void   vector(AXIS u, real m, VECTOR v);       /* evaluate a vector v whose module is
                                                  m and whose unit vector is u */
                                          /* --- Copy Functions --- */
void   mcopy(MAT A1, MAT A2, int d1, int d2);
void   mmcopy(MAT A, MAT B, int d1, int d2, int im, int jm);
int    mvcopy(MAT A, int rows, int cols, int val, int type, real *v); /* from row/col of A
                                                                         to vector */
int    vmcopy(real *v, int dim, int val, int type, MAT A, int rows,
              int cols);                       /* from vect. to row/col of A */
                                               /* --- Print Functions --- */
void   fprintm3(FILE *out, char *s, MAT3 A);
void   fprintm4(FILE *out, char *s, MAT4 A);
void   fprintv(FILE *out, char *s, real *v, int n);
void   prmat(FILE *grpout,char str[], MAT4 m);
void   printmat(char *s, MAT A, int idim, int jdim, int imax, int jmax);
void   iprintmat(char *s, int *A, int idim, int jdim, int imax, int jmax);
                                          /* --- Machine Precision Functions --- */
real   zerom(void);                            /* real machine precision */
float  fzerom(void);                           /* float machine precision */
double dzerom(void);                           /* double machine precision */

void   crossVtoM(real *a, real *b, MAT C, int dim);
void   crossMtoM(MAT A, MAT B, MAT C, int dima, int dimb, int dimc);
void   axis(int a, AXIS A);
void   traslat(VECTOR u, real h, MAT4 m);
void   traslat2(int a, real h, MAT4 m);
void   traslat24(int a, real h, POINT p, MAT4 m);
real   psedot(MAT4 A, MAT4 B);

#endif
```

6.2 The header linear.h

```
#ifndef _SPACE_LIN_
#define _SPACE_LIN_
                                          /* --- Speed and Acceleration Matrices --- */
int    dyn_eq(MAT4 J, MAT4 Wp, MAT4 F, int var[2][6]);
                                          /* --- Functions solve minvers linear, solve --- */
int    solve(MAT A, real *b, real *x, int dim); /* solves linear syst. A*x=b */
                                          /* --- Functions solve minvers linear, minvers --- */
int    minvers(MAT A, MAT Ai, int dim);   /* invers of A using linear */
                                          /* --- Functions solve minvers linear, linear --- */
void   linear(MAT H, int idim,int jdim, int imax, int jmax, int nsol,
              int ivet[],int *irank, real *arm, int vpr[]);
void   swabr(MAT H, int idim, int jdim, int j2max,int k, int im);
void   swabc(MAT H, int idim, int jdim, int imax, int k, int jm, int vet[]);
void   normalr(MAT H, int idim, int jdim, int j2max, int k, real rm);
void   elimin(MAT H, int idim, int jdim, int imax, int j2max, int k);
real   rmax(MAT H, int idim, int jdim, int imin, int imax, int jmin, int jmax,
            int *im, int *jm);
#endif
```

Chapter 7

Sample programs

To compile the examples refer to § 2.1.2.

7.1 Program ROB-MAT.C

7.1.1 General information

This section presents the bases of a computer program for the automatic solution of the *direct kinematic problem* and the *inverse dynamic problem* for an industrial robot. To solve the direct kinematic problem means to find the motion of the end-effector of a *given* robot when the motions of its joint actuators are known. To solve the inverse dynamic problem means to find the actuators and the constraint actions (torques and forces) between the contiguous links of a given robot when the *external actions* and the *motion* of the manipulator are known.

In this example the robot has been regarded as an open chain of rigid bodies (links) which are jointed to each other by revolute or prismatic pairs (see figure 7.1). The program reads from a *"description file"* (`*.DAT`) the structure of the robot (link, lengths, masses, joint types etc.) and from a *"motion file"* (`*.MOT`) the motors movement and, as output, prints the motion (position, velocity and acceleration) of each link, as well as the internal actions between each couple of contiguous links.

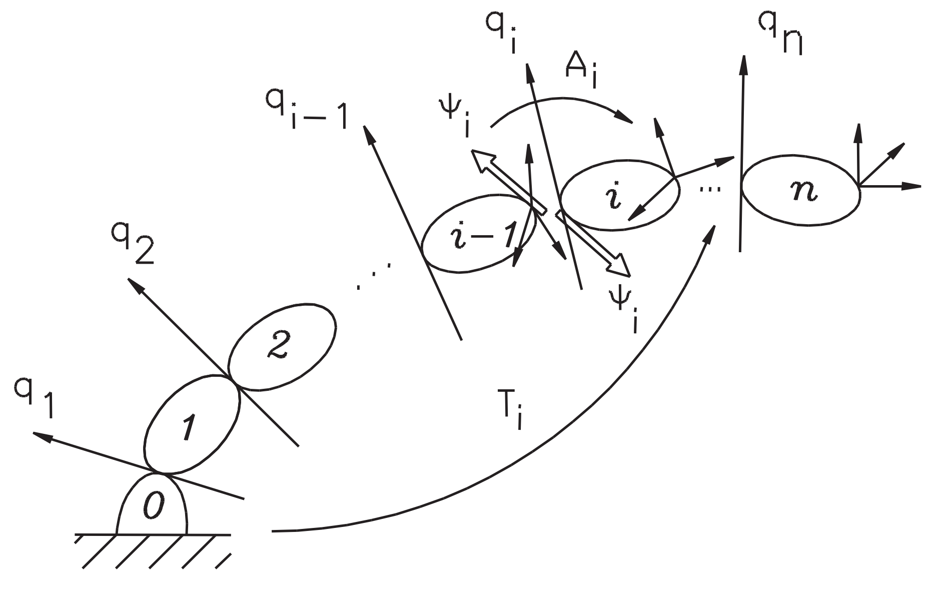


Figure 7.1: The scheme of a general serial manipulator.

7.1.2 The solution algorithm

Initially, the sample program `ROB-MAT` calculates the absolute position, speed and acceleration of all the links of the robot. This task is iteratively executed to evaluate the kinematic quantities of the links, starting from the base of the robot and proceeding to the end-effector. Conversely, the dynamic analysis is iteratively executed from the end to the base. The table 7.1 explains the meaning of the symbols used in `ROB-MAT`.

The program is structured in three main parts[1]: `DATA INPUT`, `CALCULATIONS` and `DATA OUTPUT`.

`DATA INPUT` can be divided into four steps:

1) Input data describing the geometrical structure of the manipulator:
   - *the number of links* constituting the robot;
   - for each link "i":
     * *the joint type*;
     * *five parameters* to describe the position of the frame (i), fixed on link "i", with respect to the frame (i-1), fixed on link "i-1", according to an extension to Denavit and Hartenberg approach (see [3], [4]);
2) Input data describing the dynamic parameters of the manipulator:
   - for each link "i":
     * *the six barycentral inertial moments*;
     * *the mass*;
     * *the coordinates of the center of mass* referred to the local frame (i);
3) Input data describing the external actions on manipulator:
   - *the three components of gravity acceleration* referred to the base frame;
   - *the actions* (the components of force and the components of torque) *applied on the end-effector* of the robot referred to the local frame of the gripper;
4) Input data describing the motions of the actuators:
   - for each link "i" and for each instant:
     * *the relative position*, *speed* and *acceleration* of frame (i) with respect to frame (i-1);

The `CALCULATION` part, deeply using the subroutines of the library, can be briefly described by means of the following statements (see § 7.1.4):

- Step (1) relates with sorts of *initialization procedures* to change from the scalar to the matrix environment.
- Steps (2) to (9) are included in a `for` cycle to repeat the *kinematic calculations* (absolute position, speed and acceleration) for all the links forming the robot.
- Step (10) initializes the dynamic calculations reading the external actions on the end-effector from file, building the external actions matrix and transforming it from local to base frame.
- Steps (11) to (13) performs the *dynamic calculations* (i.e. evaluates the internal actions), and are included in a `for` cycle where the counter i decreases from the total number of the robot's links to 1.
- While the kinematic calculations iteratively develop from the base of the robot to the end-effector, the dynamic calculations begin from the hand of the manipulator ending at the base.
- Note that all the matrices have been brought back to the base frame (steps (6), (7), (10), (11)) before executing the main operations (steps (8), (9), (12), (13)): this is not a set choice (one can assume as reference any frame), but it seems the easiest approach.

The program has a very simple `DATA OUTPUT` just intended for its debug, therefore only the most significant matrices are printed.

[1]It is important to remark that the only purpose of the above program is to give a simple example of the library use, so that any programmer can find better programming solutions.

7.1.3 Using ROB-MAT

As wider described above, `ROB-MAT` program requires as input:

- `DATA_FILE`: file describing the geometry of the robot, its inertial parameters and the external actions;
- `MOTION_FILE`: file containing, for every link, joint position, velocity and acceleration.

and as output:

- `OUT_FILE`: file where `ROB-MAT` will print all the matrices describing the movements of the links and the joint internal actions.

The program reads from the `DATA_FILE` the description of the robot and from the `MOTION_FILE` the motion of its motors and it prints in `OUT_FILE` file all the matrices describing the movements of the links and the joint internal actions.
To run `ROB-MAT` on a `MS-DOS`© system just type this command line:

```
a:\>ROB-MAT DATA_FILE MOTION_FILE
```

The program will read from the `DATA_FILE` the description of the robot and from the `MOTION_FILE` the motion of its motors and it will print on the screen all the matrices describing the movements of the links and the joint internal actions. One can obtain the output matrices on a file by means of the `MS-DOS`© operator ">" which allows to redirect the output. The following command line allows to store the output results in `OUTPUT_FILE`:

```
a:\>ROB-MAT DATA_FILE MOTION_FILE > OUTPUT_FILE
```

Program Symbols	*Meaning*
`A[i]`	Relative location of the frame (i) with respect to frame $(i-1)$
`T[i]`	Absolute location of the frame (i) with respect to frame (0)
`IT[i]`	Inverse of T[i]
`W[i]`	Relative velocity matrix of the frame (i) with respect to frame $(i-1)$ seen in frame $(i-1)$
`WO[i]`	Relative velocity matrix of the frame (i) with respect to frame $(i-1)$ seen in frame (0)
`WA[i]`	Absolute velocity matrix of the frame (i) with respect to frame (0) seen in frame (0)
`H[i]`	Relative acceleration of the frame (i) with respect to frame $(i-1)$ seen in frame $(i-1)$
`HO[i]`	Relative acceleration of the frame (i) with respect to frame $(i-1)$ seen in frame (0)
`HA[i]`	Absolute acceleration of the frame (i) with respect to frame (0) seen in frame (0)
`Hg`	Matrix including the three components of gravity acceleration seen in the absolute frame (0)
`Ht`	Sum of `HA[i]` and `Hg`
`J[i]`	Mass distribution of the link (i) with respect to the origin of the frame (i) seen in frame (i)
`JO[i]`	Mass distribution of the link (i) with respect to the origin of the frame (0) seen in frame (0)
`FI[i]`	Actions (forces and torques) due to inertia and weight applied on link (i) seen in (0)
`ACTO[i]`	Matrix embodies the constraint actions on joint (i) seen in absolute frame (0)

Table 7.1: Meaning of the symbols used in the program `ROB-MAT.C`

7.1.4 Listing of the program ROB_MAT

```
/* ROB-MAT: program for direct kinematics and inverse dynamics of ANY serial robot v.2
                       Developed on MS-DOS operative system with Microsoft C compiler V. 5.10 */
/* Note: To compile this program the type real must be set equivalent to the type float  (see
   also User's Manual). This is necessary because the formatting string of the fscanf function
   have been written using %f as descriptor. */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "spacelib.h"
void main(int argc,char *argv[])
{
      #define MAXLINK 10                        /* max number of links */
      int nlink,jtype[MAXLINK];                 /* n.links;  joint type */
      real theta[MAXLINK],d[MAXLINK];           /* Extended D.&H. parameters */
      real b[MAXLINK],a[MAXLINK],alpha[MAXLINK];
      real m,jxx,jxy,jxz,jyy,jyz,jzz,xg,yg,zg; /* dynamics parameters */
      real q,qp,qpp;                            /* joint variables */
      real gx,gy,gz;                            /* gravity acceleration */
      real fx,fy,fz,cx,cy,cz;                   /* external forces and torques on end-effector */
      MAT4 *mrel,*T,*W,*WO,*WA,*H,*HO,*HA,*J,*JO,*FI,*ACTO;   /* matrices */
      MAT4 EXT,Hg,Ht;
      MAT4 TMP;                                 /* temporary matrix */
      FILE *data;                               /* file including robot parametres */
      FILE *motion;                             /* file including actuator motions */
      int i;                                    /* counter */
      int ierr;                                 /* error code */
      int size;                                 /* for use of calloc function */
      float t,dt;
      if(argc!=3){
            printf("Usage: Rob-Mat  data_file  motion_file\n");
            exit(1);
      }
      data=fopen(argv[1],"r");                  /* check of input data */
      if(data==NULL)
            exit(2);
      motion=fopen(argv[2],"r");
      if(motion==NULL)
            exit(3);
      size=sizeof(MAT4);                        /* dynamic allocation of memory */
      mrel =(MAT4 *) calloc(MAXLINK,size);
      T =(MAT4 *) calloc(MAXLINK,size);
      W =(MAT4 *) calloc(MAXLINK,size);
      WO=(MAT4 *) calloc(MAXLINK,size);
      WA=(MAT4 *) calloc(MAXLINK,size);
      H =(MAT4 *) calloc(MAXLINK,size);
      HO=(MAT4 *) calloc(MAXLINK,size);
      HA=(MAT4 *) calloc(MAXLINK,size);
      J =(MAT4 *) calloc(MAXLINK,size);
      JO=(MAT4 *) calloc(MAXLINK,size);
      FI=(MAT4 *) calloc(MAXLINK,size);
      ACTO=(MAT4 *) calloc((MAXLINK+1),size);                                  /* step (1) */
      idmat4(T[0]);                             /* INITIALIZATION of matrices */
      clear4(WA[0]);
      clear4(HA[0]);                                                 /* read robot description */
      fscanf(data,"%d",&nlink);                 /* n. of links */
      for (i=1;i<=nlink;i++)                    /* for each link */
      {                                         /* D.&H. parameters */
            fscanf(data,"%d %f %f %f %f %f",&jtype[i],&theta[i],&d[i],&b[i],&a[i],&alpha[i]);
            fscanf(data,"%f %f %f %f %f %f %f",&m,&jxx,&jxy,&jxz,&jyy,&jyz,&jzz);
            fscanf(data,"%f %f %f",&xg,&yg,&zg);            /* dynamic data */
            jtoJ(m,jxx,jyy,jzz,jxy,jyz,jxz,xg,yg,zg,J[i]);  /* build inertia mat. */
      }
      fscanf(data,"%f %f %f",&gx,&gy,&gz);     /* read gravity acceleration vector */
      gtom(gx,gy,gz,Hg);                       /* build gravity acceleration matrix */
      fscanf(motion,"%f",&dt);                 /* read the range of time */
                                                                 /* ***** KINEMATICS *****  */
      for(t=0;;t+=dt)                          /* for each instant of time */
            for (i=1;i<=nlink;i++)             /* for each link */
            {
                  ierr=fscanf(motion,"%f %f %f",&q,&qp,&qpp);   /* read motions (2) */
                  if (ierr!=3)                 /* end of data in file MOTION */
                  {       fcloseall();
                        exit(0);
                  }
                  dhtom(jtype[i],theta[i],d[i],b[i],a[i],alpha[i],q,mrel[i]); /* build relative
                                                                       position matrix (3) */
                  velacctoWH(jtype[i],qp,qpp,W[i],H[i]);/* build relative velocity and
                                                         acceleration matrix in local frame (4) */
                  molt4(T[i-1],mrel[i],T[i]);   /* evaluate absolute position matrix (5) */
                  trasf_mami(W[i],T[i-1],W0[i]);/* transform relative velocity matrix from local
                                                   frame to base frame  (6) */
                  trasf_mami(H[i],T[i-1],H0[i]);/* transform relative acceleration matrix from
                                                   local frame  to  base frame  (7) */
                  sum4(WA[i-1],W0[i],WA[i]);    /* evaluate absolute velocity matrix  (8) */
                  coriolis(HA[i-1],H0[i],WA[i-1],W0[i],HA[i]);/* evaluate absolute acceleration
                                                   matrix (9) */
            }                                   /* ***** DYNAMICS ***** initializations  (10) */
            fscanf(data,"%f %f %f %f %f %f",&fx,&fy,&fz,&cx,&cy,&cz);/* read external actions on
                                                                       end-effector */
            actom(fx,fy,fz,cx,cy,cz,EXT);       /* build external action matrix */
            trasf_mamt4(EXT,T[nlink],ACT0[nlink+1]);/* transforms external actions from local to
                                                   base frame */
            for(i=nlink;i>0;i--)                /* for each link */
            {
                  trasf_mamt4(J[i],T[i],J0[i]); /* transform inertia matrix from local to base
                                                   frame  (11) */
                  rmolt4(HA[i],-1.,TMP);        /* change sign to find inertia action */
                  sum4(TMP,Hg,Ht);              /* evaluate total acceleration matrix */
                  skew4(Ht,J0[i],FI[i]);        /* evaluate the action matrix due to inertia
                                                   and weight (12) */
                  sum4(FI[i],ACT0[i+1],ACT0[i]);/* evaluate total action matrix (13) */
            }                                                /* ***** OUTPUT RESULTS ***** */
            for(i=1;i<=nlink;i++)         /* for each link */
            {
                  printf("\n\n Link %d \n\n",i);
                  printm4("Rel. pos. matrix",mrel[i]);
                  printm4("Abs. pos. matrix",T[i]);
                  printm4("Rel. vel. matrix in frame (i)",W[i]);
                  printm4("Rel. vel. matrix in frame (0)",W0[i]);
                  printm4("Absolute velocity matrix in frame (0)",
                        WA[i]);
                  printm4("Rel. acc. matrix in frame (i)",H[i]);
                  printm4("Rel. acc. matrix in frame (0)",H0[i]);
                  printm4("Absolute acceleration matrix in frame (0)",
                        HA[i]);
                  printm4("Inertia matrix in frame (i)",J[i]);
                  printm4("Inertia matrix in frame (0)",J0[i]);
                  printm4("Total actions",FI[i]);
                  printm4("Action on joint i",ACT0[i]);
            }
      }
}                                        /*  end main  */
```

7.1.5 Use of Rob-Mat

Example n.1: `SCARA ICOMATIC03`© `ROBOT`

Here is an example of the simulation of a 3 d.o.f. `SCARA ICOMATIC03`© `ROBOT` (see figure 7.2), described in file `SCARA.DAT`, acting a trajectory of two points included in file `SCARA. MOT`. File `SCARA.DAT`

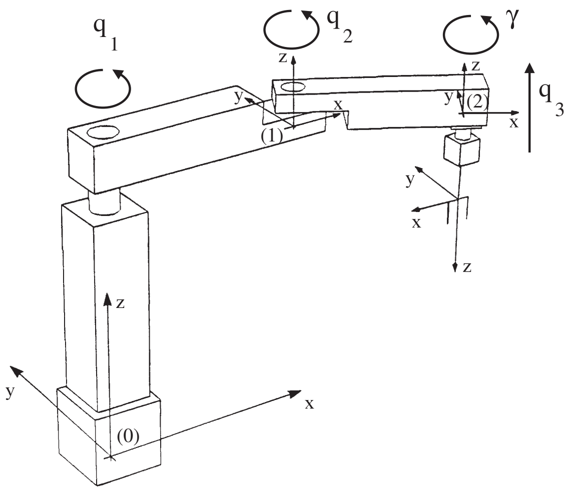


Figure 7.2: Kinematic structure of the `SCARA` robot.

contains the geometry of the robot `SCARA ICOMATIC03`©, its inertial parameters and the external actions on the gripper (see table 7.2). In the example the angle γ is considered constant ($\gamma = 0$) and so the x axes of the two last frames are parallel to each other.
File `SCARA.MOT` includes the motion of the actuators of the robot, given in terms of displacement, speed and acceleration (see table 7.3). In this example the law of motion is formed by two points only; obviously the program is able to elaborate laws of motions composed by a larger number of points!
File `SCARA.OUT` includes the output matrices. They have not been printed here for the file is very long; however they can be found in the `BIGEXA` directory of `SpaceLib`©.
To print the result matrices on the screen run the program as follows[2]:

```
a:\>ROB-MAT SCARA.DAT SCARA.MOT
```

To obtain the same file `SCARA.OUT` contained in the directory `BIGEXA` of `SpaceLib`©, type:

```
a:\>ROB-MAT SCARA.DAT SCARA.MOT > SCARA.OUT
```

Example n.2: `SMART`© `ROBOT`

Here is an example of the simulation of a `SMART`© `ROBOT` (6 degrees of freedom) described in file `SMART.DAT` acting a trajectory of only one point included in file `SMART.MOT` (see table 7.4 and figure 7.3). To print the result matrices on the screen run the program as follows[3]:

```
a:\>ROB-MAT SMART.DAT SMART.MOT
```

To obtain the same file `SMART.OUT` contained in the directory `BIGEXA` of `SpaceLib`©, type:

```
a:\>ROB-MAT SMART.DAT SMART.MOT > SMART.OUT
```

File `SMART.OUT` includes the output matrices. It have not been printed here for the file is very long; however they can be found in the `BIGEXA` directory of `SpaceLib`©.

[2]To run this program the type `real` must be set equivalent to the type `float` (see also § 2.1).
[3]To run this program the type `real` must be set equivalent to the type `float` (see also § 2.1).

DATA_FILE `SCARA.DAT`	MEANING
3	number of link
	FIRST LINK
0	Joint type
0 0 0 0.33 0	Denavit and Hartenberg parameters
10	Mass of the first link
0.03 0 0	Inertia moments jxx, jxy, jxz
0.03 0	Inertia moments jyy, jyz
0.05	Inertia moments jzz
-0.05 0.0 0.0	Center of Mass coordinates Xg, Yg, Zg
	SECOND LINK
0	
0 0 0 0.33 0	
10	
0.03 0 0	
0.03 0	
0.05	
-0.05 0 0	
	THIRD LINK (END-EFFECTOR)
1	
0 -0.1 0 0 3.1415	
3	
0.0008 0 0	
0.0008 0	
0.0015	
0 0 -0.10	
	EXTERNAL ACTIONS
0 0 -9.8	Gravity components in base frame (0)
0 0 0 0 0 0	External forces and torques applied on the end effector

Table 7.2: Content of the file `SCARA.DAT`

MOTION_FILE `SCARA.MOT`	MEANING
0.05	dt
	FIRST POINT
0 1 10	displacement, speed and acceleration of the first motor
0 2 20	displacement, speed and acceleration of the second motor
0 0 0	displacement, speed and acceleration of the third motor
	SECOND POINT
0.1 1.1 11	displacement, speed and acceleration of the first motor
0.1 2.1 21	displacement, speed and acceleration of the second motor
0 0 0	displacement, speed and acceleration of the third motor
	OTHERS POINT
...	

Table 7.3: Content of the file `SCARA.MOT`

NOTE: The geometrical data of the robot links correspond to the actual robot, while the values of the dynamical parameters have been estimated very approximatively.

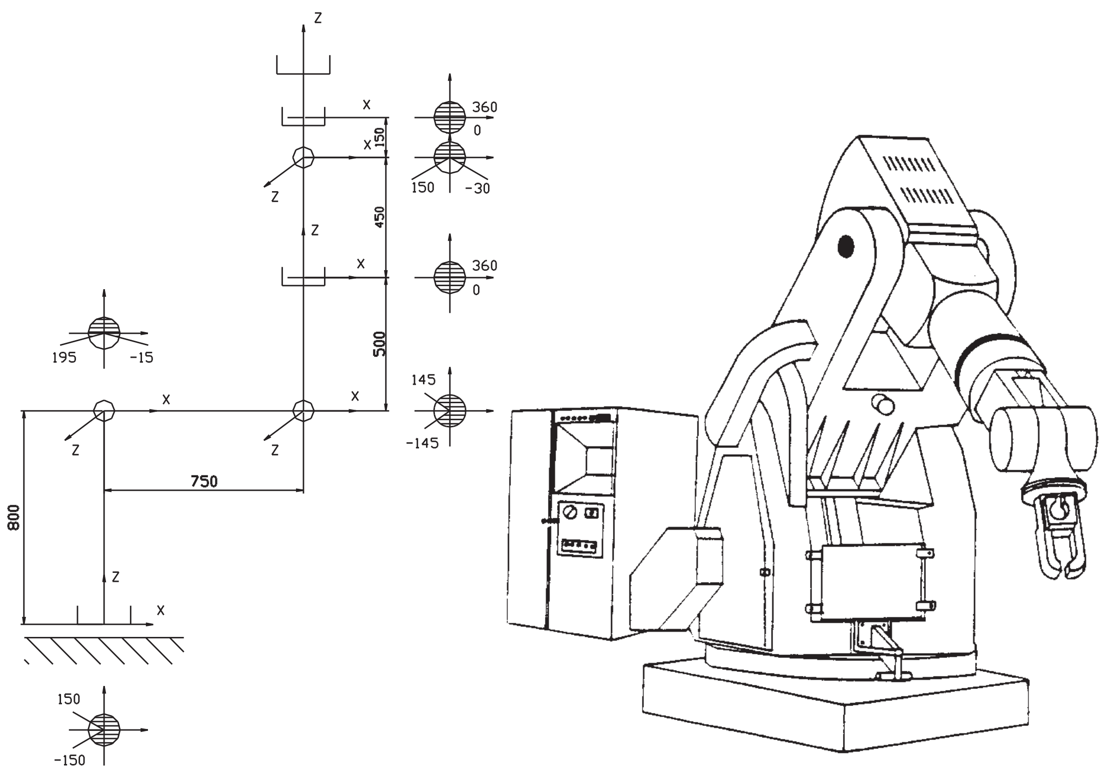


Figure 7.3: Frames definition for `SMART ROBOT`.

DATA_FILE
`SMART.DAT`

```
6

0
0 0.8 0 0 1.57079
600
100 0 0
120 0
100
0 -0.40 0

0
0 0 0 0.75 0
300
18 0 0
10 0
10
-0.30 0 0
```

```
0
0 0 0.5 0 -1.57079
200
10 0 0
10 0
18
0 0 -0.2

0
0 0.45 0 0 1.57079
100
3 0 0
5 0
3
0 -0.20 0
```

```
0
0 0 0.15 0 -1.57079
100
3 0 0
3 0
5
0 0 -0.07

0
0 0.0 0 0 0
200
4 0 0
4 0
6
0 0 -0.5

0 0 -9.8
0 0 0 0 0 0
```

MOTION_FILE
`SMART.MOT`

```
0.05

0 0.5 2
1.57 0.5 2
-1.57 0.5 2

0 0.5 2
0.26 0.5 2
0 0.5 2
```

Table 7.4: Content of the files `SMART.DAT` and `SMART.MOT`

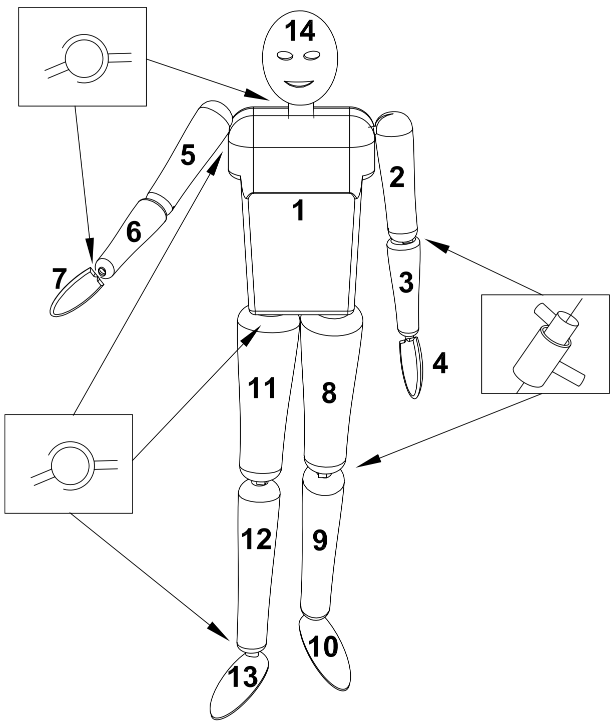


Figure 7.4: The model of the human body considered in the references [8], [9], and [14]:: the human joints are approximated by spherical or revolute hinges.

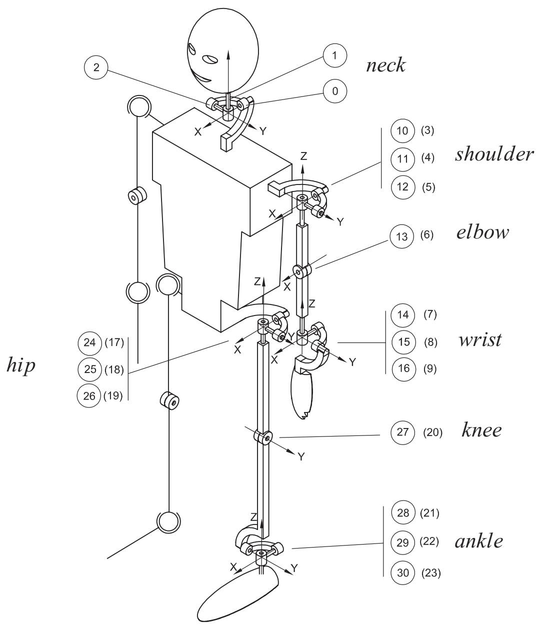


Figure 7.5: Schematization of the spherical joints by revolute hinges and enumeration of the degrees of freedom. Number in parentheses refer to right side. See section 7.2 for a simplified version of the model.

7.2 Program Test

7.2.1 General information

This sample program[4] demonstrates the use of `dyn_eq` function for the solution of the direct dynamic problem of a two-link system floating in the 3D space (see figure 7.6). In practice, the program predicts the trajectory of the system. This is an educational simplification of the problem of finding the trajectory of a man during a jump (or dive) widely described in [8], [9] and [14] to which one can refer for more details. The program reads from a file (`TEST.DAT`) the links description (masses, inertias, ...) and from another file (`TEST.MOT`) the motion of the motor and prints on the screen the trajectory of the two links (position, velocity and acceleration matrices).

7.2.2 Theory in brief

The system (see figure 7.6) is free in the space and the relative position of the two links is forced by a motor which imposes a motion law $q(t)$, $\dot{q}(t)$, $\ddot{q}(t)$. If the following data are known:

- inertia of the two links (J_1 and J_2)
- the initial position and velocity of body 1 ($M_{0,1}$ and $W_{0,1}$)
- the relative motion between the links (q, $\dot{q}$ and $\ddot{q}$, and so $M_{1,2}$, $W_{1,2}$, $H_{1,2}$)

then it is possible to evaluate the acceleration of link 1. This is what is necessary in order to obtain the trajectory of the system by a numerical integration. More in detail the acceleration of body 1 is the sum of two terms one of which is known while the other is the unknown.
The dynamic equation[5] of the system is

$$\Phi_g = skew\left\{H_{0,1} \cdot J_{1(0)}\right\} + skew\left\{H_{0,2} \cdot J_{2(0)}\right\} = [0] \tag{7.1}$$

[4]To run this program the type `real` must be set equivalent to the type `float` (see also § 2.1).
[5]For the subscript convection see § 2.2.1.

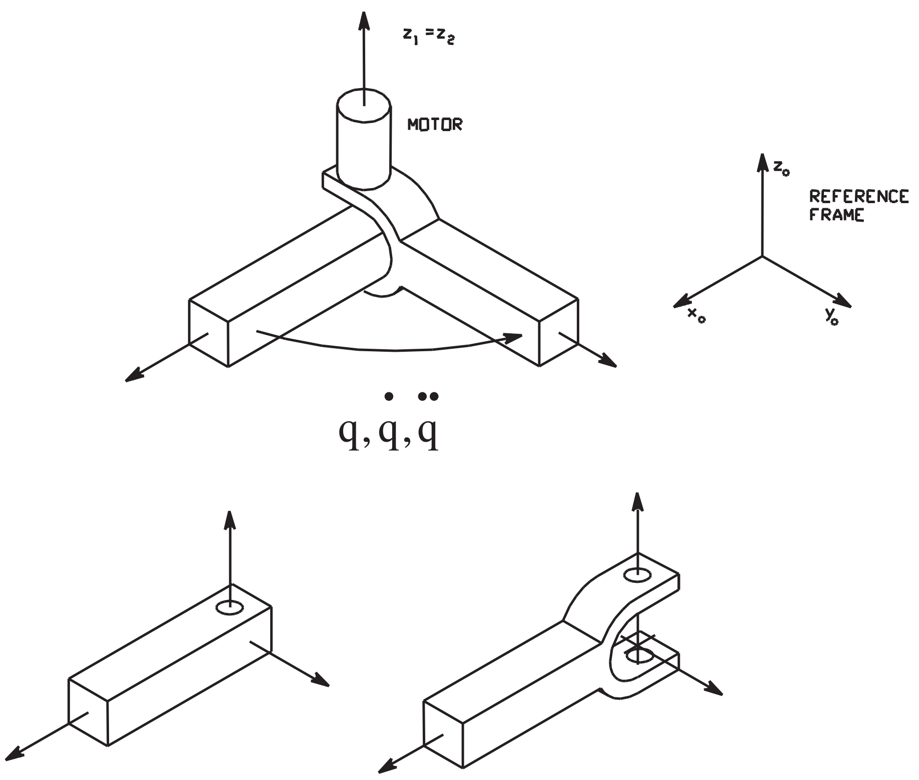


Figure 7.6: The system of the example `test`.

since the system is free in the space and it is not subjected to the gravity force, then Φ_g is the null matrix [0]. The matrices describing the motion of the two bodies are related by the following relations:

- the acceleration of body 1 is the sum of two terms. The first is known, while the second is the unknown

$$H_{0,1} = W_{0,1}^2 + \dot{W}_{0,1} \tag{7.2}$$

- the position of body 2 is

$$M_{0,2} = M_{0,1} \cdot M_{1,2} \qquad M_{1,2} = M\,(q) \tag{7.3}$$

- the velocity of body 2 is

$$W_{0,2} = W_{0,1} + W_{1,2(0)} \qquad W_{1,2(0)} = M_{0,1} \cdot W_{1,2} \cdot M_{0,1}^{-1} \qquad W_{1,2} = W(\dot{q}) \tag{7.4}$$

- the acceleration of body 2 is

$$H_{0,2} = H_{0,1} + H_{1,2(0)} + 2 \cdot W_{0,1} \cdot W_{1,2(0)} \qquad H_{1,2(0)} = M_{0,1} \cdot H_{1,2} \cdot M_{0,1}^{-1} \tag{7.5}$$

$$H_{1,2} = W_{1,2}^2 + \dot{W}_{1,2} \qquad \dot{W}_{1,2} = \dot{W}(\ddot{q}) \tag{7.6}$$

- the inertia of the two links can be expressed in base frame by the following relations

$$J_{1(0)} = M_{0,1} \cdot J_1 \cdot M_{0,1}^t \qquad J_{2(0)} = M_{0,2} \cdot J_2 \cdot M_{0,2}^t \tag{7.7}$$

J_1 and J_2 are constant and their value is known, while $M_{1,2}$, $W_{1,2}$ and $\dot{W}_{1,2}$ can be easily evaluated by knowing $q(t)$, $\dot{q}(t)$ and $\ddot{q}(t)$. At last $M_{0,1}$ and $W_{0,1}$ are known at the initial time t=0 and $\dot{W}_{0,1}$ will be the result of the following calculation.

The dynamic equation (7.1) can be "exploded" by means of three successive steps:

- union of the two terms

$$[0] = skew\left\{H_{0,1} \cdot J_{1(0)} + H_{0,2} \cdot J_{2(0)}\right\} \tag{7.8}$$

- explosion of H terms

$$[0] = skew\left\{\left(W_{0,1}^2 + \dot{W}_{0,1}\right) \cdot J_{1(0)} + \left(H_{0,1} + H_{1,2(0)} + 2 \cdot W_{0,1} \cdot W_{1,2(0)}\right) \cdot J_{2(0)}\right\} \tag{7.9}$$

- new explosion of H terms

$$[0] = skew\left\{\left(W_{0,1}^2 + \dot{W}_{0,1}\right) \cdot J_{1(0)} + \left(\left(W_{0,1}^2 + \dot{W}_{0,1}\right) + H_{1,2(0)} + 2 \cdot W_{0,1} \cdot W_{1,2(0)}\right) \cdot J_{2(0)}\right\} \tag{7.10}$$

and then the terms contained in the skew operator are divided in order to separate the terms containing the unknown $\dot{W}_{0,1}$ from the others.

$$skew\left\{W_{0,1}^2 \cdot J_{1(0)} + \left(W_{0,1}^2 + H_{1,2(0)} + 2 \cdot W_{0,1} \cdot W_{1,2(0)}\right) \cdot J_{2(0)}\right\} = skew\left\{-\dot{W}_{0,1} \cdot \left(J_{1(0)} + J_{2(0)}\right)\right\} \tag{7.11}$$

or shortly

$$= skew\left\{-\dot{W}_{0,1} \cdot J_{tot}\right\} \tag{7.12}$$

with the positions

$$\begin{gathered} J_{tot} = J_{1(0)} + J_{2(0)} \\ \Phi = skew\left\{H_{0,1}^* \cdot J_{1(0)} + H_{0,2}^* \cdot J_{2(0)}\right\} \end{gathered} \tag{7.13}$$

where $H_{0,1}^*$ and $H_{0,2}^*$ are the “partial” acceleration of body 1 and 2 (i.e. their absolute acceleration evaluated considering $\dot{W}_{0,1}$=[0])

$$H_{0,1}^* = W_{0,1}^2 \qquad H_{0,2}^* = H_{0,1}^* + H_{1,2(0)} + 2 \cdot W_{0,1} \cdot W_{1,2(0)} \tag{7.14}$$

equation (7.12) can be solved by using the `dyn_eq` function of `SpaceLib`©.

Then the total absolute acceleration of bodies 1 and 2 can be evaluated. It yields:

$$H_{0,1} = H_{0,1}^* + \dot{W}_{0,1} \qquad H_{0,2(0)} = H_{0,2}^* + \dot{W}_{0,1} \tag{7.15}$$

Although more raffinate integration methods can be set up, the new position and speed of link 1 at the time (`t`+Δ`t`) can be approximatively evaluated, for instance, by the simple following integration method

$$\begin{gathered} \left\{\begin{array}{l} M_{0,1<t+\Delta t>} \cong M_{0,1} + \dot{M}_{0,1}\Delta t + \frac{1}{2}\ddot{M}_{0,1}\Delta t^2 = \left([1] + W_{0,1}\Delta t + \frac{1}{2}H_{0,1}\Delta t^2\right) M_{0,1} = \Delta M \cdot M_{0,1} \\ W_{0,1<t+\Delta t>} \cong W_{0,1} + \dot{W}_{0,1}\Delta t \end{array}\right. \\ \textit{with} \\ \left\{\begin{array}{l} \dot{M}_{0,1} = W_{0,1} \cdot M_{0,1} \\ \ddot{M}_{0,1} = H_{0,1} \cdot M_{0,1} \\ \Delta M = \left([1] + W_{0,1} \cdot \Delta t + \frac{1}{2} \cdot H_{0,1} \cdot \Delta t^2\right) \end{array}\right. \qquad [1] = \text{identity matrix} \end{gathered} \tag{7.16}$$

All of these operations must be repeated iteratively in order to evaluate the trajectory of the system.

7.2.3 The program (cross reference)

The variables of the program[6] have the meaning listed in table 7.5. The initial position and speed of body 1 are assigned by initializing the matrices `m1`, `W1`.
The inertia matrices of the bodies are assigned by initializing the matrices `J1` and `J2`.
The relative motion between the links is described by three variables `q`, `qp`, `qpp` (position, speed and acceleration).

[6]To run this program the type `real` must be set equivalent to the type `float` (see also § 2.1).

Program Symbols	*Equation Symbols*	*Meaning*
q, qp and qpp	q, $\dot{q}$ and $\ddot{q}$	position, speed and acceleration of the motor
m1	$M_{0,1}$	abs. position of body 1
W1	$W_{0,1}$	abs. velocity of body 1 (in frame 0)
H1	$\begin{cases} H^*_{0,1} \\ H_{0,1} \end{cases}$	partial acceleration of body 1 (in frame 0) absolute acceleration of body 1 (in frame 0)
Wp	$\dot{W}_{0,1}$	Unknown part of acceleration of body 1
m2	$M_{0,2}$	abs. position of body 2
W2	$W_{0,2(0)}$	abs. velocity of body 2 (in frame 0)
H2	$\begin{cases} H^*_{0,2} \\ H_{0,2} \end{cases}$	partial acceleration of body (in frame 0) absolute acceleration of body 2 (in frame 0)
m12	$M_{1,2}$	relative position of body 1 and 2
W12	$W_{1,2}$	rel. velocity between body 1 and 2 (in frame 1)
H12	$H_{1,2}$	rel. acceleration between body 1 and 2 (in frame 1)
W120	$W_{1,2(0)}$	rel. velocity between body 1 and 2 (in frame 0)
H120	$H_{1,2(0)}$	rel. acceleration between body 1 and 2 (in frame 0)
J1, J2, J10, J20, Jtot	J_1, J_2, $J_{1(0)}$, $J_{2(0)}$, J_{tot}	Inerzia
F1		$skew\left\{H^*_{0,1} \cdot J_{1(0)}\right\} + skew\left\{H^*_{0,2} \cdot J_{2(0)}\right\}$
F2		$skew\left\{H^*_{0,2} \cdot J_{2(0)}\right\}$

Table 7.5: Cross reference for the program `TEST`

7.2.4 Scheme of the program

The program consists of the following steps (letters and digits refer to the program source code listed in the following pages).

1. Reads the description of the links and the initial condition (position and velocity) of link 1 from file `TEST.DAT`.
2. For each instant
   a) reads from file `TEST.MOT` the motion (q, $\dot{q}$, $\ddot{q}$) of the motor.
   b) evaluates `m12`, the relative position matrix of body one and two.
   c) evaluates `m2`, the absolute position of body 2.
   d) evaluates partial acceleration of link1: `H1` = `W1` · `W1`.
   e) evaluates `W12` and `H12`, the relative velocity and acceleration between the bodies.
   f) evaluates `W120` and `H120` referring `W12` and `H12` to the reference frame.
   g) evaluates absolute velocity `W02` and partial acceleration of link 2 `H02`.
   h) evaluates `J10` and `J20` referring `J1` and `J2` to the base frame.
   i) evaluates `Jtot` = `J10` + `J20`.
   j) evaluates `F2` = *skew*(`H2`, `J20`) and `F1` = `F2` + skew(`H1`, `J10`).
   k) finds the unknown `Wp` by using the `dyn_eq` function.
   l) evaluates the total acceleration of links 1 & 2 `H1` = `H1` +`Wp` and `H2` = `H2` +`Wp`.
   m) evaluate matrix `dm`: `dm` = [1] + `W01 dt` + 0.5 `H01 dt`^{2}.
   n) evaluates the new absolute position of link 1 (`t = t+dt`).
   o) evaluates the new absolute velocity of link 1 (`t = t+dt`).

DATA_FILE TEST.DAT	MEANING
	LINK 1
10 1 1 1	mass, Jx, Jy, Jz inertia moments
0 0 0	Jxy, Jyz, Jxz
1 0 0	Xg, Yg, Zg centre of mass position
	LINK 2
10 1 1 1	mass, Jx, Jy, Jz inertia moments
0 0 0	Jxy, Jyz, Jxz
1 0 0	Xg, Yg, Zg centre of mass position

0 0 0 1
0 0 0 0
0 0 0 0
0 0 0 0
	velocity matrix of link 1

1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
	position matrix of link 1

Table 7.6: Content of the file TEST.DAT

3. Repeats steps a÷o until the motion file is completely scanned.

Note: An improved version of the program (TEST_NEW) is also contained; it is based on the following considerations.

- *The angular moment of the system should be constant, but inaccuracy in the integration method corrupts it. In this new version of the program, some statements have been added to preserve the total angular momentum obtaining an improved final accuracy.*
- *At each integration step, the linear and angular momentum are evaluated and a velocity dW added to each link of the system in order to set the value of the linear and angular momentum equal to their initial value (G=G0 for t=0.)*

7.2.5 The format of the inputfiles

The BIGEXA directory of SpaceLib© contains an example of input files (TEST.DAT and TEST.MOT). They are here listed in order to show their format. The input file TEST.DAT has the format listed in table 7.6 while the law of motion file TEST.MOT has the format show in table 7.7. The first line of the file TEST.MOT contains the time step dt, while the other lines contain the value of the motor position, speed and acceleration ad each time.

7.2.6 Source code of TEST

```
/*                             ****    TEST    ****
       Program for the trajectory prediction of a two-link system floating in the space
                                                                          October 1990*/
/* Note: To compile this program the type real must be set equivalent to the type float (see
         also User's Manual). This is necessary because the formatting string of the fscanf
         function have been written using %f as descriptor. */
#include <stdio.h>
#include <float.h>
#include <math.h>
#include "spacelib.h"
#include "linear.h"
```

MOTION_FILE TEST.MOT			MEANING
0.002			dt (step of time)
0.000000E+00	0.000000E+00	0.000000E+00	$q\ \dot{q}\ \ddot{q}$ (t=0)
5.556963E-07	8.334976E-04	8.333988E-01	.. .. .. (t=dt)
4.444593E-06	3.334976E-03	1.665613	.. .. .. (t=2 · dt)
1.499523E-05	7.494372E-03	2.495461	.. .. ..
3.552638E-05	1.331228E-02	3.321762	
6.934328E-05	2.077827E-02	4.143342	
...	...	...	

Table 7.7: Content of the file TEST.MOT

```
void delta_m(MAT4 W, MAT4 H, real dt, MAT4 dm);
      POINT O=ORIGIN;
      AXIS  Zax=Zaxis;
      MAT4 m1, m12, m2, W1, W12, W120, W2, H1, H12, H120, H2;
      MAT4 Wp, dm, TMP, UNIT=UNIT4;
      MAT4 J1, J2, J10, J20, Jtot;
      MAT4 F1,F2;
      int var[2][6] = {{1, 1, 1, 1, 1, 1}, {0, 0, 0, 0, 0, 0}};
void main(void)
{
      real q, qp, qpp;
      real t,dt;
      float mass,jx,jy,jz,jxy,jyz,jxz,xg,yg,zg;
      int i,j,n,excode;
      FILE *fil;
      fil=fopen("test.dat","r");                /* open description file */
      if (fil==NULL)
      {
            printf("Error on input file TEST.DAT");
            exit(1);
      }                                         /* read description of the links --- step (1) */
      fscanf(fil,"%f %f %f %f %f %f %f %f %f %f",&mass,&jx,&jy,&jz,&jxy,&jyz,&jxz,&xg,&yg,&zg);
                                                /* link 1 */
      jtoJ(mass,jx,jy,jz,jxy,jyz,jxz,xg,yg,zg,J1); /*build inertia matrix*/
      fscanf(fil,"%f %f %f %f %f %f %f %f %f %f",&mass,&jx,&jy,&jz,&jxy,&jyz,&jxz,&xg,&yg,&zg);
                                                /* link 2 */
      jtoJ(mass,jx,jy,jz,jxy,jyz,jxz,xg,yg,zg,J2); /*build inertia matrix*/
      for(i=0;i<4;i++)                         /* read initial condition of the system */
            for(j=0;j<4;j++)                   /* read velocity matrix of link 1 */
                  fscanf(fil,"%f",&W1[i][j]);
      for(i=0;i<4;i++)                         /* read position matrix of link 1 */
            for(j=0;j<4;j++)
                  fscanf(fil,"%f",&m1[i][j]);
      fclose(fil);
      fil=fopen("test.mot","r");               /* open motion file */
      if (fil==NULL)
      {
      printf("error on input file TEST.MOT");
            exit(1);
      }
      fscanf(fil,"%f",&dt);                    /* read integration step "dt" */
      for(t=0;;t+=dt)                          /* loop for each instant t --- step (2) */
      {
            n=fscanf(fil,"%f %f %f ",&q,&qp,&qpp);    /* read motion of motor (a) */
            printf("--- t: %f ---- q: %f   qp: %f   qpp: %f",t,q,qp,qpp);
            screwtom(Zax, q, 0., 0, m12);      /* relative position links 1&2 (b) */
            molt4(m1,m12,m2);                  /* absolute position of link 2 (c) */
            molt4(W1,W1,H1);                   /* partial acceleration of link 1 (d) */
            velacctoWH2(Rev,Z,qp,qpp,W12,H12); /* rel.vel. & acc. of link 1&2 (e) */
            trasf_mami(W12,m1,W120);
            trasf_mami(H12,m1,H120);           /* (f) */
            norm_simm_skew(M W120,3,4,SKEW);   /* normalization reducing num. error */
                                 /* absolute velocity and partial acceleration of link 2 (g) */
            sum4(W1,W120,W2);
            coriolis(H1,H120,W1,W120,H2);
            trasf_mamt4(J1,m1,J10);            /* refer inertia moment to absolute frame (h) */
            trasf_mamt4(J2,m2,J20);
            norm_simm_skew(M J10,4,4,SYMM);    /* normalization reducing num. errors */
            norm_simm_skew(M J20,4,4,SYMM);    /* normalization reducing num. errors */
            mcopy4(J10,Jtot);                  /* total inertia (i)*/
            sum4(Jtot,J20,Jtot);
            skew4(H1,J10,F1);                  /* evaluate inertia actions (j) */
            skew4(H2,J20,F2);
            sum4(F1,F2,F1);
            excode=dyn_eq(Jtot,Wp,F1,var);
            rmolt4(Wp,-1,Wp);                  /* evaluate Wp (k) */
            if(excode==NOTOK)
            {
                  printf(" excode=%d",excode);
                  exit(1);
            }
            sum4(Wp,H1,H1);                    /* absolute acc. Of links 1 & 2 (l) */
            sum4(Wp,H2,H2);
            printm4("Position matrix of link 1",m1);
            printm4("Abs. position matrix of link 2",m2);
            printm4("Velocity matrix of link 1",W1);
            printm4("Abs. velocity matrix of link 2",W2);
            printm4("Acceleration matrix of link 1",H1);
            printm4("Abs. acceleration matrix of link 2",H2);
            if(n!=3) break;                     /*if motion file empty -> end of loop */
            delta_m(W1,H1,dt,dm);               /* builds matrix dm (m) */
            molt4(dm,m1,TMP);                   /* new position of link 1 (n) */
            mcopy4(TMP,m1);
            rmolt4(Wp,dt,Wp);                   /* new velocity of link 1 (o) */
            sum4(Wp,W1,W1);
      }
      fcloseall();
}
                     /* --- Function delta_m: builts matrix dm where dm=[1]+Wdt+0.5Hdt^2 --- */
void delta_m(MAT4 W, MAT4 H, real dt, MAT4 dm)
{
      int i,j;
      real dt2;
      dt2=dt*dt;
      for (i=0; i<3; i++)
            for (j=0; j<4; j++)
                  dm[i][j]=UNIT[i][j]+W[i][j]*dt+.5*H[i][j]*dt2;
      dm[U][X]=dm[U][Y]=dm[U][Z]=0.;
      dm[U][U]=1;
      normal4(dm);
}
```

7.2.7 Source code of TEST_NEW

```
/*                                **** TEST_NEW.C ****
        Program for the trajectory prediction of a two-link system floating in the space
```

```
                                                          July 1998 update version of test.c
% This is an improved version of Test.c
% The angular moment of the system should be constant, but inaccuracy in the integration method
% corrupts it. In this version, some statments have been added to preserve the total angular
% momentum obtaining an improved final accuracy. At each integration step the angular momentum
% is evaluated and a velocity dW added to the system in order to set the value of the angular
% momentum equal to its initial value (G=Go for t==0.)
*/
/* Note: To compile this program the type real must be set equivalent to the type float
         (see also User's Manual). This is necessary because the formatting string of the
         fscanf function has been written using %f as descriptor (and NOT %lf). */
#include <stdlib.h>
#include <stdio.h>
#include <float.h>
#include <math.h>
#include "spacelib.h"
#include "linear.h"
void delta_m(MAT4 W, MAT4 H, real dt, MAT4 dm);
      POINT O=ORIGIN;
      AXIS  Zax=Zaxis;
      MAT4 m1, m12, m2, W1, W12, W120, W2, H1, H12, H120, H2;
      MAT4 Wp, dm, TMP, UNIT=UNIT4;
      MAT4 J1, J2, J10, J20, Jtot;
      MAT4 F1,F2;
      MAT4 G1,G2;
      MAT4 G,G0;                 /* linear and angular momentum (current and initial) */
      MAT4 dW;
      int var[2][6] = {{1, 1, 1, 1, 1, 1}, {0, 0, 0, 0, 0, 0}};
void main(void)
{
      real q, qp, qpp;
      real t,dt;
      float mass,jx,jy,jz,jxy,jyz,jxz,xg,yg,zg;
      int i,j,n,excode;
      FILE *fil;
      fil=fopen("test.dat","r"); /* open description file */
      if (fil==NULL)
      {
            printf("Error on input file TEST.DAT");
            exit(1);
      }                          /* read description of the links --- step (1) */
      fscanf(fil,"%f %f %f %f %f %f %f %f %f %f",&mass,&jx,&jy,&jz,&jxy,&jyz,&jxz,&xg,&yg,&zg);
                                 /* link 1 */
      jtoJ(mass,jx,jy,jz,jxy,jyz,jxz,xg,yg,zg,J1); /*build inertia matrix*/
      fscanf(fil,"%f %f %f %f %f %f %f %f %f %f",&mass,&jx,&jy,&jz,&jxy,&jyz,&jxz,&xg,&yg,&zg);
                                 /* link 2 */
      jtoJ(mass,jx,jy,jz,jxy,jyz,jxz,xg,yg,zg,J2); /*build inertia matrix*/
      for(i=0;i<4;i++)           /* read initial condition of the system */
            for(j=0;j<4;j++)     /* read velocity matrix of link 1 */
                  fscanf(fil,"%f",&W1[i][j]);
      for(i=0;i<4;i++)           /* read position matrix of link 1 */
           for(j=0;j<4;j++)
                  fscanf(fil,"%f",&m1[i][j]);
      fclose(fil);
      fil=fopen("test.mot","r"); /* open motion file */
      if (fil==NULL)
      {
      printf("error on input file TEST.MOT");
            exit(1);
      }
      fscanf(fil,"%f",&dt);      /* read integration step "dt" */
      for(t=0;;t+=dt)            /* loop for each instant t --- step (2) */
      {
            n=fscanf(fil,"%f %f %f ",&q,&qp,&qpp);  /* read motion of motor (a) */
            printf("--- t: %f ---- q: %f   qp: %f   qpp: %f",t,q,qp,qpp);
            screwtom(Zax, q, 0., 0, m12);      /* relative position links 1&2 (b) */
            molt4(m1,m12,m2);                  /* absolute positio of link 2 (c) */
                                               /* step (d) moved forward */
            velacctoWH2(Rev,Z,qp,qpp,W12,H12); /* rel.vel. & acc. of link 1&2 (e) */
            trasf_mami(W12,m1,W120);
            trasf_mami(H12,m1,H120);           /* (f) */
            norm_simm_skew(M W120,3,4,SKEW);   /* normalization reducing num. error */
                                 /* absolute velocity and partial acceleration of link 2 (g1) */
            sum4(W1,W120,W2);                  /* step (g2) moved forward */

                                                /* refer inertia moment to absolute frame (h) */
            trasf_mamt4(J1,m1,J10);
            trasf_mamt4(J2,m2,J20);
            norm_simm_skew(M J10,4,4,SYMM);    /* normalization reducing num. errors */
            norm_simm_skew(M J20,4,4,SYMM);    /* normalization reducing num. errors */
            mcopy4(J10,Jtot);                  /* total inertia (i)*/
            sum4(Jtot,J20,Jtot);
/**/                                           /* new statments */
            skew4(W1,J10,G1);
            skew4(W2,J20,G2);
            sum4(G1,G2,G);
            if (t==0.) mcopy4(G,G0);
            sub4(G,G0,G);
            excode=dyn_eq(Jtot,dW,G,var);
            sub4(W1,dW,W1);
            sub4(W2,dW,W2);
                                                /* moved statments */
            molt4(W1,W1,H1);                    /* partial acceleration of link 1 (d) */
            coriolis(H1,H120,W1,W120,H2);       /* (g2) */
/**/
            skew4(H1,J10,F1);                   /* evaluate inertia actions (j) */
            skew4(H2,J20,F2);
            sum4(F1,F2,F1);
            excode=dyn_eq(Jtot,Wp,F1,var);
            rmolt4(Wp,-1,Wp);                   /* evaluate Wp (k) */
            if(excode==NOTOK)
            {
                  printf(" excode=%d",excode);
                  exit(1);
            }
            sum4(Wp,H1,H1);                      /* absolute acc. Of links 1 & 2 (l) */
            sum4(Wp,H2,H2);
            printm4("Position matrix of link 1",m1);
            printm4("Abs. position matrix of link 2",m2);
            printm4("Velocity matrix of link 1",W1);
            printm4("Abs. velocity matrix of link 2",W2);
            printm4("Acceleration matrix of link 1",H1);
            printm4("Abs. acceleration matrix of link 2",H2);
            if(n!=3) break;                      /*if motion file empty -> end of loop */
            delta_m(W1,H1,dt,dm);                /* builds matrix dm (m) */
            molt4(dm,m1,TMP);                    /* new position of link 1 (n) */
            mcopy4(TMP,m1);
            rmolt4(Wp,dt,Wp);                    /* new velocity of link 1 (o) */
            sum4(Wp,W1,W1);
      }
      fcloseall();
}
                       /* --- Function delta_m: builts matrix dm where dm=[1]+Wdt+0.5Hdt^2 --- */
void delta_m(MAT4 W, MAT4 H, real dt, MAT4 dm)
```

```
{
      int i,j;
      real dt2;
      dt2=dt*dt;
      for (i=0; i<3; i++)
           for (j=0; j<4; j++)
                  dm[i][j]=UNIT[i][j]+W[i][j]*dt+.5*H[i][j]*dt2;
      dm[U][X]=dm[U][Y]=dm[U][Z]=0.;
      dm[U][U]=1;
      normal4(dm);
}
```

7.3 Rototranslation

In this paragraph it is shown how to calculate the rototranslation (i.e. the axis of rototranslation) of the triangle which passes from the position (1) to the position (2) as in figure 7.7: Point P_1 moves to

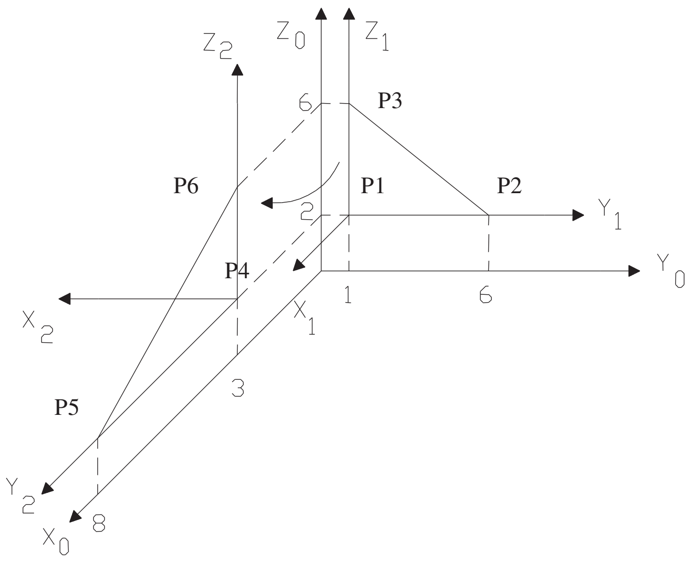


Figure 7.7: The frames definition for the example of rototranslation.

P_4, P_2 moves to P_5, P_3 moves to P_6. The frame attached to the triangle moves from $X_1Y_1Z_1$ to $X_2Y_2Z_2$. The position matrix of frame (2) with respect to reference frame (0) and of frame (0) with respect to (1) are

$$M_{1,0}=\left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -2 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \qquad M_{0,2}=\left[\begin{array}{ccc|c} 0 & 1 & 0 & 3 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{7.17}$$

and the desired rototranslation matrix is

$$Q_0 = M_{0,2} \cdot M_{1,0}=\left[\begin{array}{ccc|c} 0 & 1 & 0 & 2 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] = \left[\begin{array}{ccc|c} & R & & T \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{7.18}$$

The rotation is clearly a rotation of $\pi/2$ about an axis anti-parallel to Z_0. Using `SpaceLib`© this result can be obtained with the following statements

```
MAT4 Q;
AXIS u;
real phi;
real h;
POINT P1={0.,1.,2.,1.};
POINT P2={0.,6.,2.,1.};
POINT P3={0.,1.,6.,1.};
POINT P4={3.,0.,2.,1.};
POINT P5={8.,0.,2.,1.};
POINT P6={3.,0.,6.,1.};
MAT4 m01, m02, m10;
frame4P (P1,P2,P3,Y,Z,m01);
frame4P (P4,P5,P6,Y,Z,m02);
invers (m01,m10);
molt4 (m02,m10,Q);
mtoscrew (Q,u,&phi,&h,P);
```

It gives the result:

Rotation angle	`phi` $= \pi/2 = 1.57079$
Axis direction	`u` $= [0\ 0\ \text{-}1]^t$
Point	`P` $= [1\ \text{-}1\ 0\ 1]^t$
Translation	`h` $= 0$

7.4 Scara robot

7.4.1 Theory in brief

This example shows how to solve the direct kinematic problem for the position and velocity of the Scara robot.

There are five reference frames. Frame (0) is the fixed frame, frame (1) is attached to the base while frames (2), (3) and (4) are embedded in link 1, 2 and 3 respectively. The auxiliary frame (a) is a moving frame whose origin is in the center of the gripper and whose axes are parallel to the reference frame (0).
The program described in § 7.4.2 is based on the conventions of the figure 7.4.1. The 1st link of this Scara robot is 1.5 m long, while the 2nd and the 3rd link are 0.33 m long. Its joint variables Q and the first time derivative $\dot{Q}$ of Q is

$$Q = [\alpha, \beta, h*] = \left[\frac{\pi}{4}, \frac{\pi}{6}, \frac{1}{2}\right] \qquad \dot{Q} = \left[\frac{5\pi}{4}, \frac{5\pi}{4}, \frac{-1}{2}\right]$$

The position of frame (a) referred to frame (0) is expressed by the matrix

$$M_{0,a} = \left[\begin{array}{ccc|c} 1 & 0 & 0 & 0.319 \\ 0 & 1 & 0 & 0.552 \\ 0 & 0 & 1 & 1 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{7.19}$$

The velocity matrix of the center of the gripper in reference (0) is

$$W_{0,4(0)} = \left[\begin{array}{ccc|c} 0 & -7.854 & 0 & 0.916 \\ 7.854 & 0 & 0 & -0.916 \\ 0 & 0 & 0 & -0.5 \\ \hline 0 & 0 & 0 & 0 \end{array}\right] \tag{7.20}$$

The velocity matrix of the gripper in reference frame (a) is

$$W_{0,4(a)} = M_{a,0} \cdot W_{0,4(0)} \cdot M_{0,a} = \tag{7.21}$$

$$= \left[\begin{array}{ccc|c} 0 & -7.854 & 0 & -3.420 \\ 7.854 & 0 & 0 & 1.587 \\ 0 & 0 & 0 & -0.5 \\ \hline 0 & 0 & 0 & 0 \end{array}\right]$$

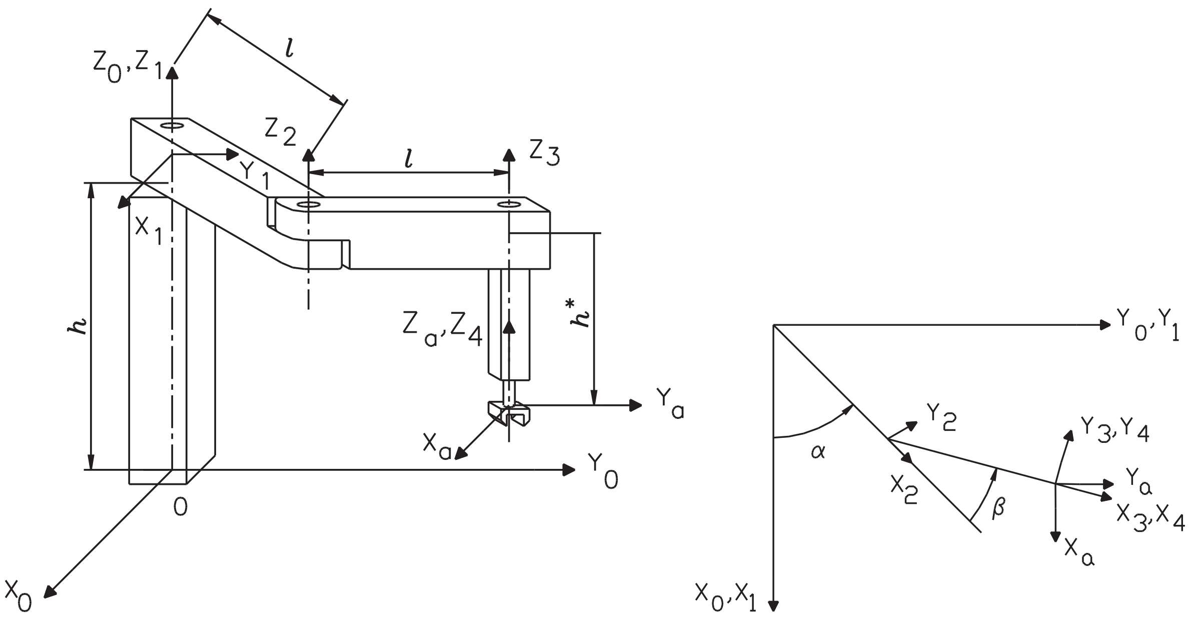


Figure 7.8: The frames definition for the example of robot Scara.

7.4.2 Listing of the program ROBSCARA

```
/*                              ****  ROBSCAR.C: ****
             Sample program for direct kinematics of Scara robot (See User's Manual)
*/
#include <stdio.h>
#include <math.h>
#include "spacelib.h"
void main(void)
{
      real q[3] ={PIG/4,PIG/6,-0.5};      /* joint variables array */
      real qp[3]={PIG*5/4,PIG*5/4,-0.5};  /* joint var. first time der. */
      POINT O1={0.,0.,1.5,1.};            /* origin of frame (1) in frame (0) */
      POINT O2={0.33,0.,0.,1.};           /* origin of frame (2) in frame (1) */
      POINT O3={0.33,0.,0.,1.};           /* origin of frame (3) in frame (2) */
      POINT O4={0.,0., 0., 1.};           /* origin of frame (4) in frame (3) */
      POINT Oa;                           /* origin of frame (a) in frame (0) */
      POINT O=ORIGIN;
      MAT4 m01,m12,m23,m34;               /* mi-1,i  : pos. of frame (i) in frame (i-1) */
      MAT4 m02,m03,m04,m0a;               /* m0,i    : pos. of frame (i) in frame (0) */
      MAT4 L12r,L23r,L34r;                /* Li,i+1r : L mat. of joint i in frame (i-1) */
      MAT4 L12f,L23f,L34f;                /* Li,i+1f : L mat. of joint i in frame (0) */
      MAT4 W01,W12,W23,W34;               /* Wi,i+1  : W mat. of frame (i+1) in frame (i) */
      MAT4 W04;                           /* velocity matrix of the gripper in frame (0) */
      MAT4 W04a;                          /* velocity matrix of the gripper in frame (a) */

      idmat4(m01);                        /* builds relative position matrices*/
      idmat4(m34);
      vmcopy(M O1,4,4,Col,M m01,4,4);
      rotat34(Z,q[0],O2,m12);
      rotat34(Z,q[1],O3,m23);
      vmcopy(M O4,4,4,Col,M m34,4,4); m34[Z][U]+=q[2];
      molt4(m01,m12,m02);                 /* builds absolute position matrices */
      molt4(m02,m23,m03);
      molt4(m03,m34,m04);
      idmat4(m0a);                        /* builds pos. matrix of frame (a) in frame (0) */
      mvcopy(M m04,4,4,4,Col,M Oa);
      vmcopy(M Oa,4,4,Col,M m0a,4,4);
      makeL2(Rev,Z,0.,O,L12r);            /* builds relative L matrices */
      makeL2(Rev,Z,0.,O,L23r);
      makeL2(Pri,Z,0.,O,L34r);
      trasf_mami(L12r,m01,L12f);          /* evaluates L matrices in frame (0) */
      trasf_mami(L23r,m02,L23f);
      trasf_mami(L34r,m03,L34f);
      clear4(W01);                        /* builds relative W matrices */
      rmolt4(L12f,qp[0],W12);
      rmolt4(L23f,qp[1],W23);
      rmolt4(L34f,qp[2],W34);
      sum4(W01,W12,W04);                  /* builds abs. W matrix of frame (4) in frame (0) */
      sum4(W04,W23,W04);
      sum4(W04,W34,W04);
      trasf_miam(W04,m0a,W04a);           /* evaluates W matrix of frame (4) in frame (a) */
                                          /* output results */
      printm4("The velocity matrix of the gripper in frame (0) is:",W04);
      printm4("The velocity matrix of the gripper in frame (a) is:",W04a);

}
```

7.5 Satellite

7.5.1 General information

This sample program demonstrates the use of several SpaceLib© functions described in this manual for the solution of a real problem. In practice, the program calculates how to obtain the best movement to spread out the antennas from a satellite.

7.5.2 Theory in brief

7.5.2.1 The problem For the installation of a satellite, launched with the rocket *Ariane*, it's necessary to spread out two antennas. During the launching phase, the antennas are bent and they are positioned as in fig.1 inside figure 7.13. When the service orbit has been reached, the antennas are spread out and they are oriented as in fig.2 inside figure 7.13. Each antenna reaches the service position through three subsequent rotations; considering the left antenna, these rototranslations are:

1) a rotation θ_1 around an axis passing through the point P_2 and orthogonal to the plane P_1-P_2-P_3: with this rotation the point P_1 of the antenna is aligned with the diagonal P_3-P_2 (fig.3 inside fig.7.13);

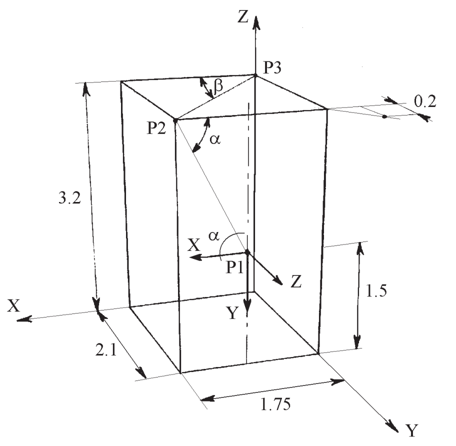


Figure 7.9: Dimensions of the rocket.

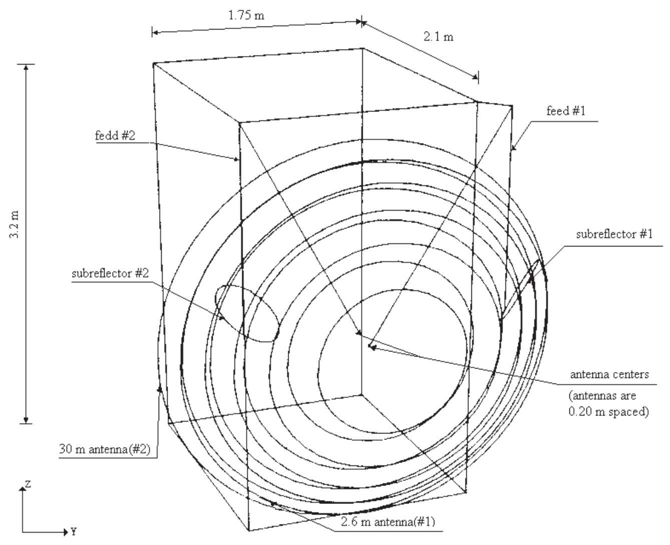


Figure 7.10: Antennas in the initial position.

2) a rotation θ_2 around an axis which coincides with the diagonal P_2-P_3 and which puts the concavity upwards (fig.4 and 5 inside figure 7.13);
3) a rotation θ_3 of 26 ° around an axis orthogonal to the diagonal; this rotation moves the antenna to its final configuration (fig.5 and 6 inside figure 7.13).

The problem was to work out if the antenna could be put in the correct position by means of just one single rototranslation.

7.5.2.2 The solution For geometrical properties is known that any combination of two or more rotations about the same point is equivalent to a "global" rotation. The following statements show how to evaluate the global rotation which allows to put in position each antenna with one single rotation movement. That's why just below are calculated the rototranslation axes of each antenna (direction cosines and a point of the axis), the rotation angles and the translations about these axes. The real dimensions of this satellite are presented in the figure 7.9, which also shows the reference frame of the rocket and the frame embedded on the antenna (initial position) (figures 7.9 and 7.10). Point P_1 is the center of left antenna, while point P_2 approximates the location of the hinge. Point P_2 can be ≅ 0.2 m higher or 0.3 m lower than the edge of the box. P_2 can also be ≅ 0.1 m outside the box. Angles α and β do not change during the whole movement, therefore they can be calculated by means of the following statements

```
α = atan2((3.2-1.5), (1.75/2)) = atan2 (1.7, 0.875) = 62.76° = 1.0953 rad
β = atan2 (2.1, 1.75) = 50.19° = 0.8759 rad
```

The sequence from the initial configuration to the final one is made up by the following steps:

- *Initial configuration* The initial position matrix of the frame of the left antenna is expressed by the matrix

$$M_i = \left[\begin{array}{ccc|c} 1 & 0 & 0 & 0.875 \\ 0 & 0 & 1 & 2.1 \\ 0 & -1 & 0 & 1.5 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \qquad (7.22)$$

- *Step 1*

  The left antenna turns about axis `Z2`, which is orthogonal to the plane containing points `P1`, `P2`, `P3`; point `P1` (the center of the antenna) get aligned with `P2` and `P3`. The right antenna turns about `Z1`, which is orthogonal to the `PA`, `PB`, `PC` plane.

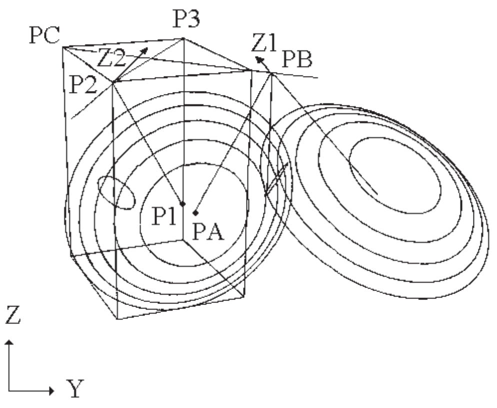


Figure 7.11: Step 1 of the antennas deployment.

- *Step 2*

  Both antennas have reached the right position relative to their feeds, which are locked onto antennas. The subreflectors still have to deploy and the antennas still have to complete their rotations.

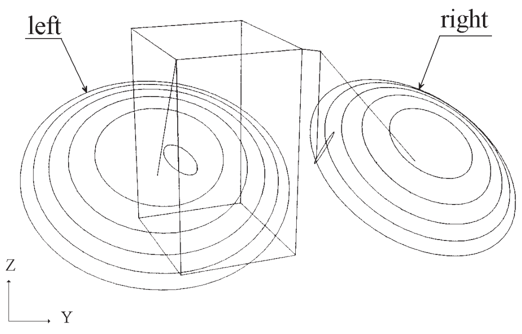


Figure 7.12: Step 2 of the antennas deployment.

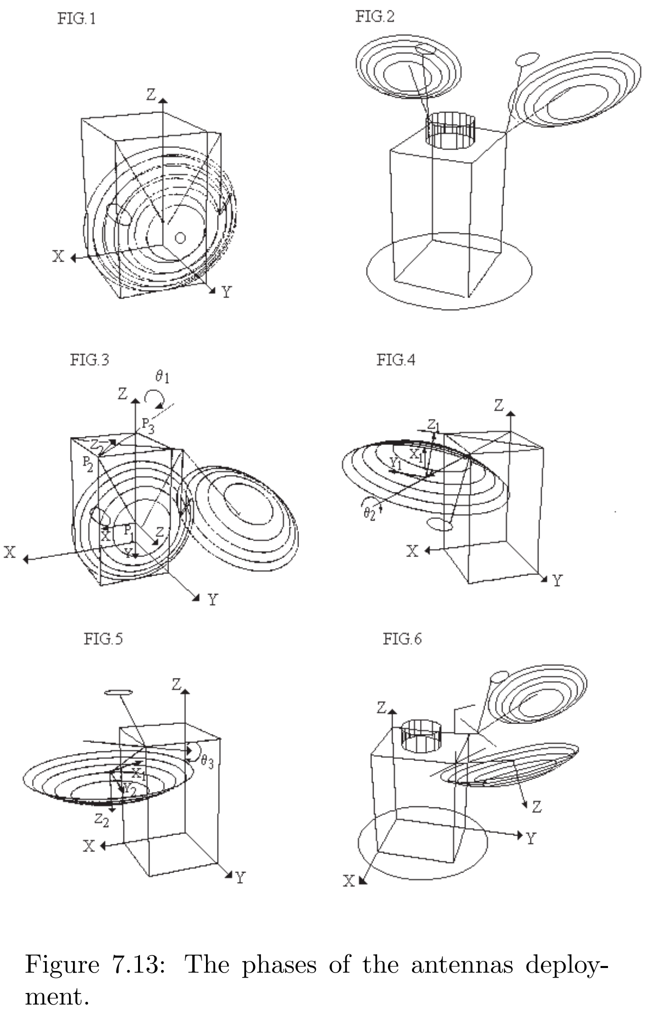


Figure 7.13: The phases of the antennas deployment.

- *Step 3*

  The antennas have completed their rotations (reference lines are on spacecraft top diagonal `d1` and `d2`). The subreflectors have deployed and reached right positions relative to antennas and feeds. They are locked onto feeds. The new position of point `P1` is now:

$$\begin{aligned} P1\,(x) &= P2\,(x) + d\cdot\cos\beta \\ P1\,(y) &= P2\,(y) + d\cdot\sin\beta \\ P1\,(z) &= P2\,(z) \end{aligned} \tag{7.23}$$

  while `d` is evaluated from figure 7.9 as the distance between `P2` and `P1`. The z axis of the moving frame is now pointing upwards (parallel to the rocket frame) while the direction of the others is presented in figure 6 inside figure 7.13. The position of the center of the left antenna at the end of step 3 is expressed by the following matrix

$$M_3 = \left[\begin{array}{ccc|c} -\cos(\alpha+\beta) & \sin(\alpha+\beta) & 0 & 2.97 \\ -\sin(\alpha+\beta) & -\cos(\alpha+\beta) & 0 & 3.57 \\ 0 & 0 & 1 & 3.20 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] = \left[\begin{array}{ccc|c} 0.3899 & 0.9208 & 0 & 2.97 \\ -0.9208 & 0.3899 & 0 & 3.57 \\ 0 & 0 & 1 & 3.20 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{7.24}$$

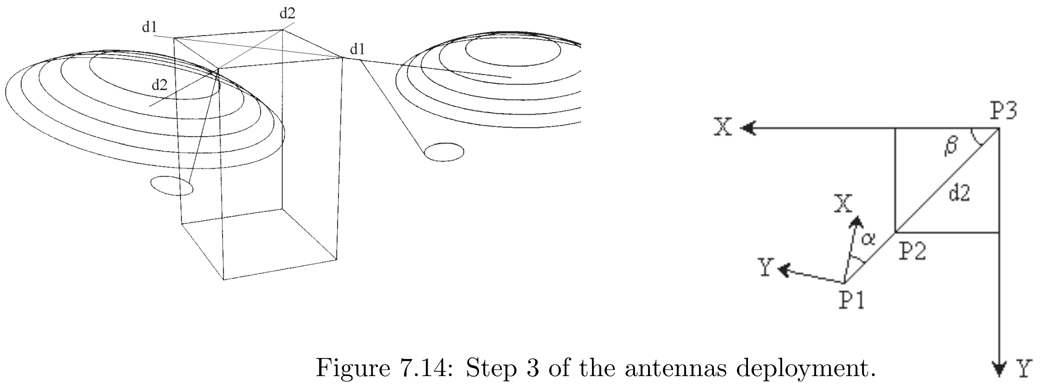
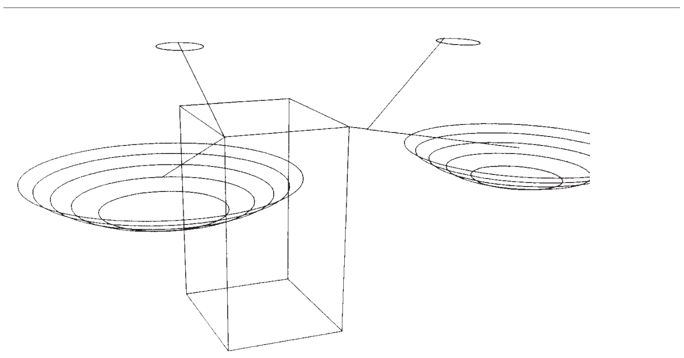


Figure 7.14: Step 3 of the antennas deployment.



- *Step 4*

  The antennas are turned face up through rotations of 180 ° about diagonals (`d1` and `d2`).

Figure 7.15: Step 4 of the antennas deployment.

The position of the center of the left antenna at the end of step 4 is expressed by the following matrix

$$M_4 = \left[\begin{array}{ccc|c} -\cos(\beta-\alpha) & -\sin(\beta-\alpha) & 0 & 2.97 \\ -\sin(\beta-\alpha) & \cos(\beta-\alpha) & 0 & 3.57 \\ 0 & 0 & -1 & 3.20 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] = \left[\begin{array}{ccc|c} -0.976 & 0.218 & 0 & 2.97 \\ 0.218 & 0.976 & 0 & 3.57 \\ 0 & 0 & -1 & 3.20 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{7.25}$$

- *Step 5*

  The antennas are filled 26 $^\circ$ up to reach the working configurations, through a rotation about axes `n1-n1` (axes `n1-n1` and `n2-n2` are normal to axes `d1-d1` and `d2-d2`). The unit vector of the rotation axis (`d1-d1`) has the following cosines

$$u_x = \sin(\beta) = 0.768$$
$$u_y = -\cos(\beta) = -0.640$$
$$u_z = 0$$

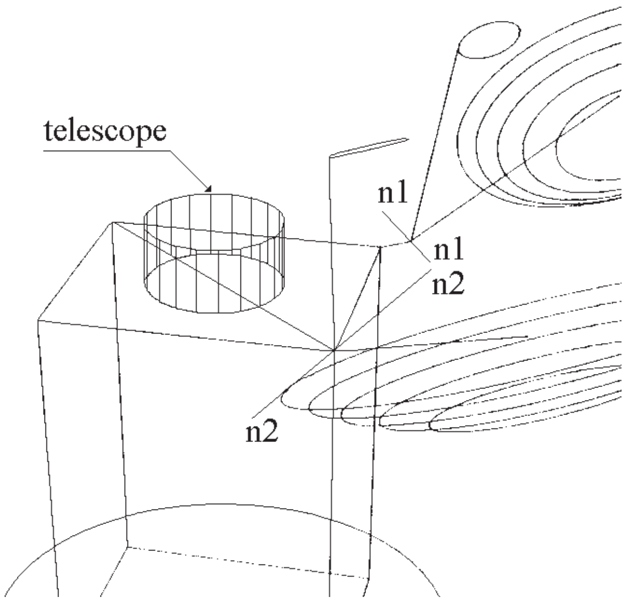


Figure 7.16: Step 5 of the antennas deployment.

The rotation axis passes through point `P2` and there is no translation along the axis.
So the Rototranslation matrix is

$$Q_5 = \left[\begin{array}{ccc|c} 0.959 & -0.050 & -0.281 & 1.075 \\ -0.050 & 0.940 & -0.337 & 1.290 \\ 0.281 & 0.337 & 0.899 & -0.874 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \qquad (7.26)$$

- *Final configuration*

The final position of the left antenna is expressed by the following matrix

$$M_f = Q_5 \cdot M_4 = \left[\begin{array}{ccc|c} -0.946 & 0.160 & 0.281 & 2.850 \\ 0.253 & 0.907 & 0.337 & 3.420 \\ -0.201 & 0.390 & -0.899 & 4.038 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \qquad (7.27)$$

The aim has now been reached:
the Rototranslation matrix which expresses the whole movement can be written as

$$Q_{tot} = M_f \cdot M_i^{-1} = \left[\begin{array}{ccc|c} -0.946 & 0.281 & -0.160 & 3.329 \\ 0.253 & 0.337 & -0.907 & 3.852 \\ -0.201 & -0.899 & -0.390 & 6.686 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \qquad (7.28)$$

From this matrix we can extract the axis unit vector whose cosines are

$$u_x = 0.1633 \qquad u_y = 0.8175 \qquad u_z = \text{-}0.5523$$

A point of the Rototranslation axis is `P2` = $[1.712,\ 1.908,\ 3.330,\ 1]^t$.

The rotation angle is 178.58 $^\circ$ and there is no translation along the axis.

A program which performs and prints on the screen the presented calculations is listed in § 7.5.3.

7.5.3 Source code of SAT

```
#include <stdio.h>
#include <math.h>
#include "spacelib.h"

void main (void)

#define sb      (real) sin(beta)
#define cb      (real) cos(beta)
#define sb_a    (real) sin(beta-alpha)
#define cb_a    (real) cos(beta-alpha)
{
        POINT P1={0.875, 2.100, 1.500, 1. };              /* values of initial configuration */
        POINT P2={1.750, 2.100, 3.200, 1. };
        MAT4 mi={ {1.,  0., 0., 0.875},
                  {0.,  0., 1., 2.100},
                  {0., -1., 0., 1.500},
                  {0.,  0., 0., 1.   } };
        real alpha=atan2( P2[Z]-P1[Z], P2[X]-P1[X] );
        real beta=atan2( P2[Y], P2[X] );
        real d=dist(P1,P2);
        MAT4 m4, Q5, mf, miinv;                              /* elements used in computation */
        AXIS u5;

        MAT4 Qtot;                                       /* global elements which are evaluated */
        AXIS utot;
        real fi,h;
        POINT P;
                                                         /* from initial configuration to step 4 */
        m4[X][X] = -cb_a; m4[X][Y] = -sb_a; m4[X][Z] =  0.; m4[X][U] = P2[X]+d*cb;
        m4[Y][X] = -sb_a; m4[Y][Y] =  cb_a; m4[Y][Z] =  0.; m4[Y][U] = P2[Y]+d*sb;
        m4[Z][X] =  0.;   m4[Z][Y] =  0.;   m4[Z][Z] = -1.; m4[Z][U] = P2[Z];
        m4[U][X] =  0.;   m4[U][Y] =  0.;   m4[U][Z] =  0.; m4[U][U] = 1.;
                                                                                   /* step 5 */
        u5[X]=sb; u5[Y]=-cb; u5[Z]=0.;
        screwtom(u5,rad(26.),0.,P2,Q5);
                                                                     /* final configuration */
        molt4(Q5,m4,mf);
                                                                /* evaluate global elements */
        invers(mi,miinv);
        molt4(mf,miinv,Qtot);
        mtoscrew(Qtot,utot,&fi,&h,P);
                                                                          /* output results */
        printf("\n*** Results ***\n");
        printv("The rototraslation axis is :",utot,3);
        printf("\nThe rotation angle about this axis is :%f [deg]\n",deg(fi));
        printf("\nThe traslation about this axis is :%f\n",h);
        printv("The point P of the axis is :",P,4);
#undef sb
#undef cb
#undef sb_a
#undef cb_a
}
```

7.6 Elbow robot

7.6.1 General information

Three different sample programs are described. They deal with the kinematics of a serial manipulator and present the use of many SpaceLib© functions. The robot under study is a 6 d.o.f. Elbow robot

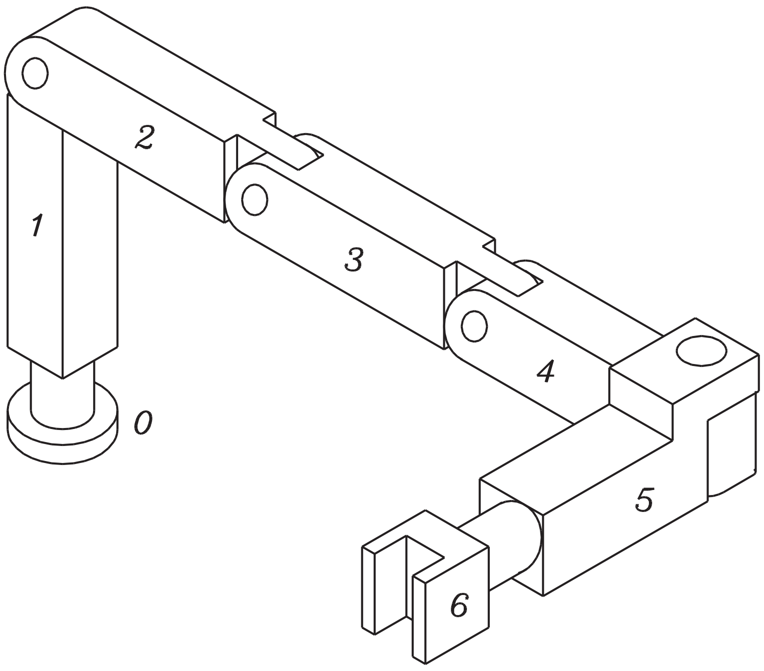


Figure 7.17: Kinematic structure of the Elbow robot.

(see [16], [3]). The robot has six revolute joints. This section contains two programs obtained with different approaches for the study of the direct kinematics of the robot (ELB_D_DH and ELB_D_PA) and one based on a numerical approach for the inverse kinematics (ELB_I_DH). The two programs for the direct kinematics accept the same input and produce identical output. The programs for the direct kinematics evaluates the gripper motion starting from the joint motions, while the program for the inverse kinematics is able to evaluate the joint motions necessary to produce an assigned gripper motion. The output file of the program for the inverse kinematics can be used as input for the programs for the direct kinematics and vice versa. These facts are summarized in the scheme of figure 7.18. In the previous

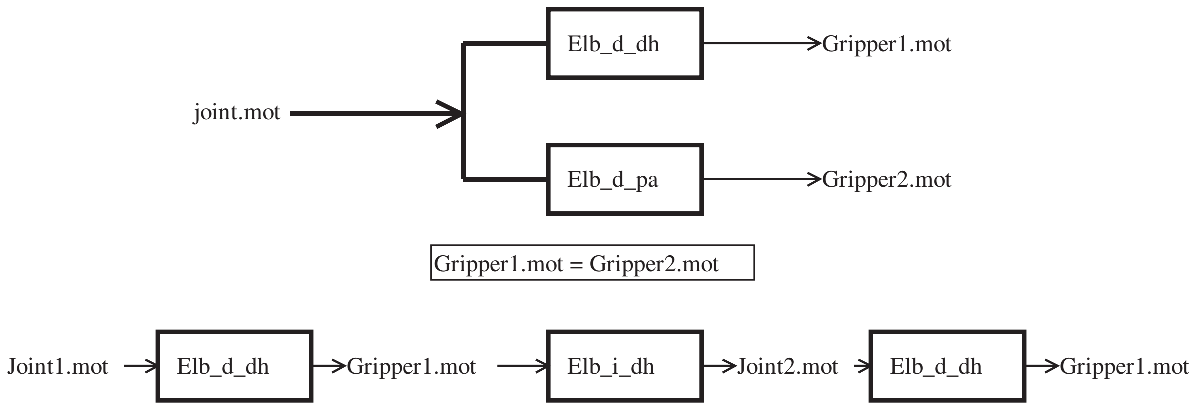


Figure 7.18: Input output files for Elbow robot

scheme gripper1.mot is identical to gripper2.mot. Since a robot can have many inverse solutions the program evaluates just one of them and in the scheme joint1.mot could be different from joint2.mot.

The two programs[7] for the direct kinematics (ELB_D_DH and ELB_D_PA) must be linked with `spacelib.c`, `spaceli3.c` and `spaceli4.c`. The program for the inverse kinematics (ELB_I_DH) must be linked with `spacelib.c`, `spaceli3.c`, `spaceli4.c`, `linear.c` and `linear2.c`.

7.6.2 Format of the data and motion files

The SpaceLib© contains an example of input/output files (ELBOW.DAT, JOINT.MOT, GRIPPER.MOT and GUESS.1ST) for the described programs. Their format is presented in table 7.8. The input file ELBOW.DAT

	ELB_D_DH ELB_D_PA	ELB_I_DH
ELBOW.DAT	Input	Input
JOINT.MOT	Input	Output
GRIPPER.MOT	Output	Input
GUESS.1ST	*Not used*	Input

Table 7.8: Input/Output files for the ELBOW ROBOT

which describes the link lengths has the format show in table 7.10 The file JOINT.MOT which describes the joint motions has the format described in table 7.9. The file GRIPPER.MOT which describes the gripper

JOINT.MOT			*Meaning*
0.01			dt
0.00114	0.25450	28.27440	Rotation, speed, acceleration of 1^{st} motor
0.00106	0.23560	26.17990	Rotation, speed, acceleration of 2^{nd} motor
0.00153	0.33930	37.69910	Rotation, speed, acceleration of 3^{rd} motor
-0.00153	-0.33930	-37.69910	Rotation, speed, acceleration of 4^{th} motor
0.00358	0.79520	88.35730	Rotation, speed, acceleration of 5^{th} motor
0.02863	6.36170	706.85828	Rotation, speed, acceleration of 6^{th} motor
0.00510	0.53720	28.27440	Rotation, speed, acceleration of 1^{st} motor
0.00473	0.49740	26.17990	Rotation, speed, acceleration of 2^{nd} motor
...	...	...	....

Table 7.9: Content of the file JOINT.MOT

ELBOW.DAT	*Meaning*
1.5	length of link 1
0.8	length of link 2
0.8	length of link 3
0.2	Length of link 4
0.0	Length of link 5
0.2	Length of link 6

Table 7.10: Content of the file ELBOW.DAT

motion has the format specified by the table 7.11 where α, β, γ denote the gripper orientation using an appropriate Cardan/Euler convention; X, Y, Z are the gripper position Single quote and double quote mark the time derivative. The file GUESS.1ST contains the value of the joint rotations for the first guess when solving the inverse kinematics problem. It has the simple format of table 7.12.

[7]To compile these programs the type real must be set equivalent to the type float (see also § 2.1); this is necessary because the formatting string of the fscanf function have been written using `%f` as descriptor.

GRIPPER.MOT			*Meaning*
0.01			dt
0	1	2	Cardan/Euler convention used X=0, Y=1, Z=2 for compatibility with the C-language version
0.029690	0.000136	0.004718	α, β, γ, t=0
6.597086	0.060601	1.048386	α', β', γ'
732.919800	20.197580	115.902000	α”, β”, γ”
0.197946	1.800940	1.503133	X, Y, Z
-0.459088	0.207974	0.695856	X’, Y’, Z’
-51.222855	22.670288	77.394531	X”, Y”, Z”
0.132292	0.002702	0.020876	α, β, γ, t=dt
13.918917	0.567294	2.161054	α', β', γ'
. . . ..	. . . ..	. . . ..	. . . ..

Table 7.11: Content of the file GRIPPER.MOT

GUESS.1ST	*Meaning*
.0 .1 .2 .0 .1 .0	q_1 q_2 q_3 q_4 q_5 q_6

Table 7.12: Content of the file GUESS.1ST

7.6.3 The file Joint.mot

This section describes the criteria under which the sample file JOINT.MOT has been created[8]. To study the robot movement, for each link a symmetrical law of motion at constant acceleration of the kind 1/3 -1/3 -1/3 has been considered (see. figure 7.19). The notation 1/3 -1/3 -1/3 indicates that the movements consist of three parts (acceleration, constant speed, deceleration) of identical duration. The joint motions are stored with a time step ΔT=0.01 seconds.

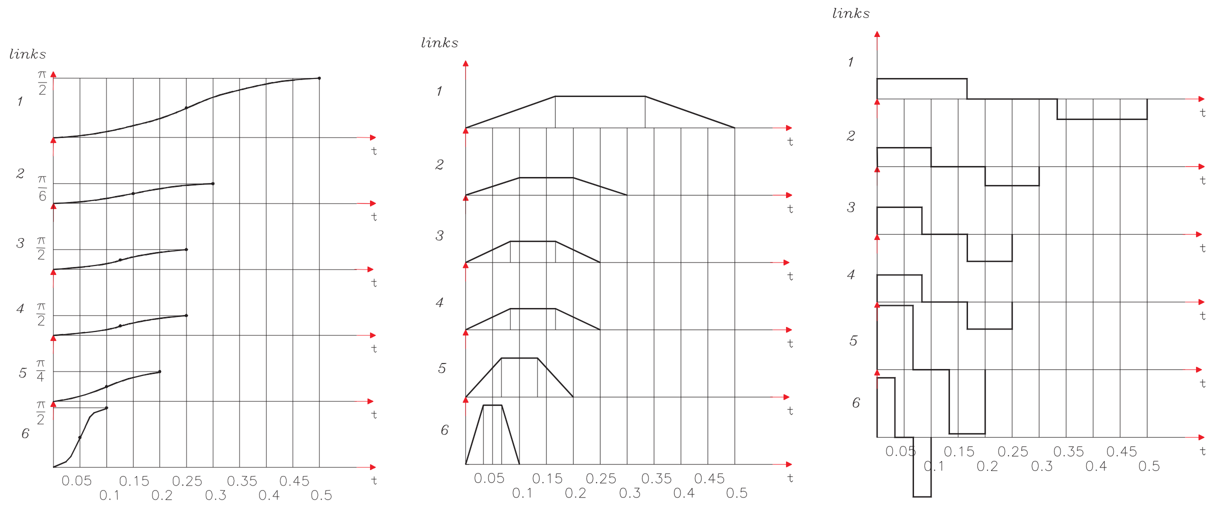


Figure 7.19: The law of motion contained in file JOINT.MOT.

7.6.4 Direct kinematics

The two sample programs here presented are: ELB_D_DH and ELB_D_PA; the first one is based on the *Denavit & Hartemberg* notation [1] and the relative frames are positioned according to figure 7.20 while

[8]The first point of the joint motions hasn’t been stored in JOINT.MOT because the robot is in a singular configuration at time t=0.

the second one is very similar but the relative frames are positioned as described in figure 7.21. The two programs accept the same input files and produce identical output.

Both programs display the gripper motion to the screen using the position, the velocity and the acceleration matrices. Velocity and acceleration are expressed in a *auxiliary frame* (parallel to the base frame) whose origin is in the `TCP` (*gripper center*). The auxiliary frame is not shown in the figures. The base frame Xa, Ya, Za and the gripper frame Xb, Yb, Zb are identical for the two programs, while the intermediate frames have been positioned using different approaches.

The gripper motion is also stored in a output file. The gripper position is represented by the `TCP` position, velocity and acceleration, while the orientation is stored as Cardan angles and their time derivatives; the chosen sequence of rotation is *rot* X, *rot* Y, *rot* Z (see also § 3.2).

Run the programs as follows:

```
a:\ >ELB_D_DH ELBOW.DAT JOINT.MOT GRIPPER.MOT
a:\ >ELB_D_PA ELBOW.DAT JOINT.MOT GRIPPER.MOT
```

7.6.5 The sample program ELB_D_DH

Frame positions

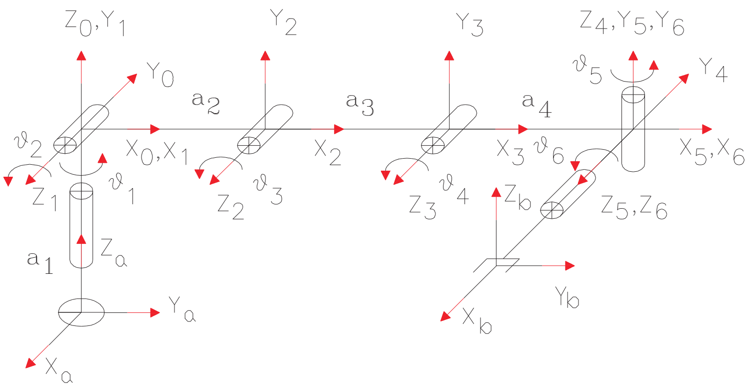


Figure 7.20: Frames definition for the example of *elbow* robot with the program `ELB_D_DH`.

The base frame is `Xa`, `Ya`, `Za` which does not move with respect to `X0`, `Y0`, `Z0`. The gripper frame `Xb`, `Yb`, `Zb` does not move with respect to `X6`, `Y6`, `Z6`. The reference frames from `X1`, `Y1`, `Z1` to `X6`, `Y6`, `Z6` attached to the links are positioned following the *Denavit* and *Hartenberg* convention (see also [1]) and so at the end of the links.

The program ELB_D_DH

```
/* elb_d_DH.c program for the DIRECT kinematics of ELBOW robot. Frames assigned according to
   Denavit and Hartenberg conventions. The output of this program is compatible with the input
   of elb_i_dh.c the input  of this program is compatible with tho output of elb_i_dh.c */

/* NOTE: To compile this program the type real must be set equivalent to the type float
      (see also User's Manual). This is necessary because the formatting string of the
       fscanf function have been written using %f as descriptor. */
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <math.h>
#include "spacelib.h"

#define MAXLINK 6
void main(int argc,char *argv[])
{
```

```
int i,j,k,ierr;
int ii=X;                 /* Euler/Cardan convention */
int jj=Y;                 /* for gripper             */
int kk=Z;                 /* angular position        */
                          /* Denavit & Hartemberg's parameters (D&H) */
real theta[MAXLINK+1]={0.,0.,0.,0.,0.,0.,0.};
real d[MAXLINK+1]={0.,0.,0.,0.,0.,0.,0.};
real b[MAXLINK+1]={0.,0.,0.,0.,0.,0.,0.};
real fi[MAXLINK+1]={0.,PIG_2,0.,0.,3*PIG_2,PIG_2,0.};
real a[MAXLINK+1];        /* to be read from data_file */
real q,qp,qpp;            /* joint variables : pos., vel., acc. */
real t,dt;
VECTOR q1,q2;             /* Euler/Cardan angle configurations */
VECTOR qp1,qp2;           /* Euler/Cardan angle first time der. */
VECTOR qpp1,qpp2;         /* Euler/Cardan angle sec. time der. */
MAT4 mreli_1[MAXLINK+1],  /* array containing pos. mat. of frame (i) seen in frame (i-1)*/
     mabs[MAXLINK+1],     /* array containing abs. pos. mat. Of frame (i) in base frame */
     Wreli_1[MAXLINK+1],/* array containing rel.vel.mat.of frame (i) seen in frame (i-1)*/
     Wrel0[MAXLINK+1],   /* array containing rel.vel.mat.of frame (i) seen in base frame*/
     Wabs[MAXLINK+1],     /* array containing abs. vel. mat. of frame (i) in base frame */
     Hreli_1[MAXLINK+1],/* array containing rel.acc.mat.of frame (i) seen in frame (i-1)*/
     Hrel0[MAXLINK+1],   /* array containing rel.acc.mat.of frame (i) seen in base frame*/
     Habs[MAXLINK+1];     /* array containing abs. vel. mat. of frame (i) in base frame */
MAT4 Last ={ {0.,1.,0.,0.},    /* transformation matrix from */
           {0.,0.,1.,0.},      /* frame (6) to gripper       */
           {1.,0.,0.,0.},      /* element Z-U is in a[6]     */
           {0.,0.,0.,1.} };    /*                            */
MAT4 gripper;             /* abs. frame of gripper */
POINT first=ORIGIN;       /* origin of frame 0 with respect to base, Z value is in a[1]*/
MAT4 Aus,                 /* Ausiliar frame, origin in gripper, parallel to base */
     Waus, Haus;          /* abs. gripper vel. and acc. in Aus frame */
FILE *data;               /* file containing robot description */
FILE *motion;             /* file containing motion description */
FILE *out;                /* output file */
if (argc!=4)              /* testing parameters */
{
      fprintf(stderr,"Usage: elb_d_dh data_file joint_motion_file gripper_output_file\n");
      exit(1);
}
data=fopen(argv[1],"r");
if(data==NULL)
{
      fprintf(stderr,"Error: Unable to open data_file\n");
      exit(2);
}
motion=fopen(argv[2],"r");
if(motion==NULL)
{
      fprintf(stderr,"Error: Unable to open motion_file\n");
      exit(3);
}
out=fopen(argv[3],"w");
if(out==NULL)
{
      fprintf(stderr,"Error: Unable to open output_file\n");
      exit(4);
}
a[0]=0;
  for(i=1;i<=MAXLINK;i++)          /* read link lengths */
{
      fscanf(data,"%f",&a[i]);
}
```

```
    first[Z]=a[1];                      /* matrices initialization */
    rotat24(Z,PIG_2,first,mabs[0]);     /* pos. mat. of frame 0 from base frame */
    Last[Z][U]=a[6];                    /* gripper position in frame 6 */
    idmat4(Aus);
    clear4(Wabs[0]);                    /* abs. velocity of base and of frame 0 */
    clear4(Habs[0]);                    /*     acceleration */
    a[1]=a[6]=0;                        /* D&H parameter 'a' of link 1 and link 6 are zero */
    fscanf(motion,"%f",&dt);            /* read time step */
    fprintf(out,"%f\n",dt);             /* write dt to out file */
    fprintf(out,"%d %d %d\n\n",ii,jj,kk);  /* write Cardan convention to out file */
    for(t=0;eof(motion);t+=dt)          /* main loop */
    {
        for(i=1;i<=MAXLINK;i++)
        {                               /* read joint motion */
            ierr=fscanf(motion,"%f %f %f\n",&q,&qp,&qpp);
            if(ierr!=3)
                      exit(0);
            dhtom(Rev,theta[i],d[i],b[i],a[i],fi[i],q,mreli_1[i]);/* builds rel.pos. matrix */
            velacctoWH(Rev,qp,qpp,Wreli_1[i],Hreli_1[i]);/* builds rel.vel.and acc.matrices */
            molt4(mabs[i-1],mreli_1[i],mabs[i]);          /* abs. pos. matrix of frame (i) */
            trasf_mami(Wreli_1[i],mabs[i-1],Wrel0[i]);     /*W and H matrices in base frame */
            trasf_mami(Hreli_1[i],mabs[i-1],Hrel0[i]);
            sum4(Wabs[i-1],Wrel0[i],Wabs[i]);               /* abs. vel. and acc. matrices */
            coriolis(Habs[i-1],Hrel0[i],Wabs[i-1],Wrel0[i],Habs[i]);
        }
        molt4(mabs[MAXLINK],Last,gripper);           /* gripper position */
        Aus[X][U]=gripper[X][U];
        Aus[Y][U]=gripper[Y][U];
        Aus[Z][U]=gripper[Z][U];
        trasf_miam(Wabs[MAXLINK],Aus,Waus);          /* transform velocity */
        trasf_miam(Habs[MAXLINK],Aus,Haus);          /* and acceleration in ausiliar frame */
                     /* extracts Cardan angles (and their time derivatives) of gripper  */
        Htocardan(gripper,Waus,Haus,ii,jj,kk,q1,q2,qp1,qp2,qpp1,qpp2);
                                                     /* output results */
        printf("Time=%f\n",t);
        printm4("The position matrix of the gripper is:",gripper);
        printm4("The velocity matrix of the gripper is:",Waus);
        printm4("The acceleration matrix of the gripper is:",Haus);
        printf("\nPress any key to continue\n");
        getch() ;
        fprintf(out,"%f %f %f\n",q1[0],q1[1],q1[2]);
        fprintf(out,"%f %f %f\n",qp1[0],qp1[1],qp1[2]);
        fprintf(out,"%f %f %f\n",qpp1[0],qpp1[1],qpp1[2]);
        fprintf(out,"%f %f %f\n",gripper[X][U],gripper[Y][U],gripper[Z][U]);
        fprintf(out,"%f %f %f\n",Waus[X][U],Waus[Y][U],Waus[Z][U]);
        fprintf(out,"%f %f %f\n",Haus[X][U],Haus[Y][U],Haus[Z][U]);
    }
    fcloseall();
}
```

7.6.6 The sample program ELB_D_PA

Frame positions

The reference frame are attached to the links in such a way that in the "home" position ($q_1 = q_2 = \ldots = q_6 = 0$) the frames are all parallel to each other. The frames are positioned at the beginning of the links. Different from the first case, the frames result to be seven. The frames #6 and #7 move together but they have different positions. The base frame Xa, Ya, Za coincides with X0, Y0, Z0. The gripper frame Xb, Yb, Zb coincides with X7, Y7, Z7.

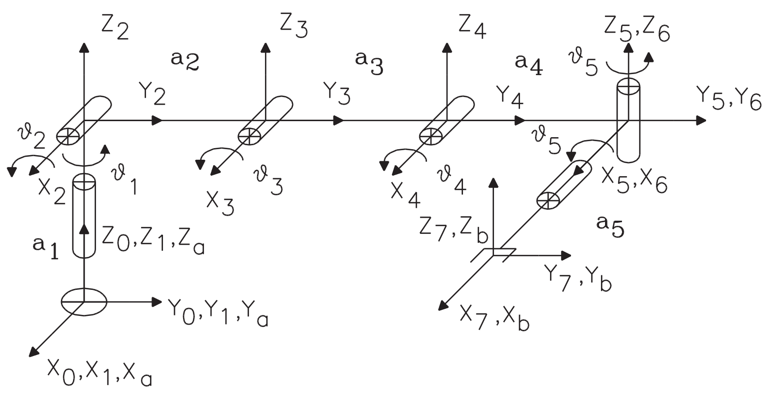


Figure 7.21: Frames definition for the example of *elbow* robot with the program ELB_D_DH.C.

The program ELB_D_PA

```
/* elb_d_PA.c program for the DIRECT kinematics of ELBOW robot. The output of this program is
   compatible with the input  of elb_i_dh.c, The input  of this program is compatible with tho
   output of elb_i_dh.c */
/* Note: To compile this program the type real must be set equivalent to the type float
      (see also User's Manual). This is necessary because the formatting string of the
       fscanf function have been written using %f as descriptor. */
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <math.h>
#include "spacelib.h"
#define MAXLINK 6
void main(int argc,char *argv[])
{
      int axis[MAXLINK+2]={U,Z,X,X,X,Z,X,U};
      int i,j,k,ierr;
        int ii=X;               /* Euler/Cardan convention */
        int jj=Y;               /* for gripper             */
        int kk=Z;               /* angular position        */
      real a[MAXLINK+1];        /* array of link lengths */
      POINT O[MAXLINK+2];       /* array containing orig. of ref. frames */
      real q,qp,qpp;            /* joint variables : pos., vel., acc. */
      real t,dt;
      VECTOR q1,q2,             /* Euler/Cardan angle configurations */
             qp1,qp2,           /* angles first time derivative */
             qpp1,qpp2;         /* angles second time derivative*/
      MAT4 mreli_1[MAXLINK+2],  /* array containing pos. mat. of frame (i) seen in frame (i-1)*/
           mabs[MAXLINK+2],     /* array containing abs. pos. mat. of frame (i) in frame (0) */
           Wreli_1[MAXLINK+2],/* array containing rel.vel.mat.of frame (i) seen in frame (i-1)*/
           Wrel0[MAXLINK+2],    /* array containing rel.vel.mat.of frame (i) seen in frame (0)*/
           Wabs[MAXLINK+2],     /* array containing abs. vel. mat. of frame (i) in frame (0) */
           Hreli_1[MAXLINK+2],/* array containing rel.acc.mat.of frame (i) seen in frame (i-1)*/
           Hrel0[MAXLINK+2],    /* array containing rel.acc.mat.of frame (i) seen in frame (0)*/
           Habs[MAXLINK+2];     /* array containing abs. vel. mat. of frame (i) in frame (0) */
     MAT4 Aus,                  /* Ausiliar frame, origin in gripper, parallel to base */
           Waus, Haus;          /* abs. gripper vel. and acc. in Aus frame */
      FILE *data;               /* file containing robot description */
      FILE *motion;             /* file containing motion description */
      FILE *out;                /* output file */
      if (argc!=4)              /* testing parameters */
      {
            fprintf(stderr,"Usage: Elb_d_pa data_file joint_motion_file griper_output_file\n");
            exit(1);
```

```
    }
    data=fopen(argv[1],"r");
    if(data==NULL)
    {
          fprintf(stderr,"Error: Unable to open data_file\n");
          exit(2);
    }
    motion=fopen(argv[2],"r");
    if(motion==NULL)
    {
          fprintf(stderr,"Error: Unable to open motion_file\n");
          exit(3);
    }
    out=fopen(argv[3],"w");
    if(out==NULL)
    {
          fprintf(stderr,"Error: Unable to open output_file\n");
          exit(4);
    }
    idmat4(mabs[0]); idmat4(Aus);     /* matrices initialization */
    clear4(Wabs[0]);
    clear4(Habs[0]);
    clearv(O[0]);
    O[0][U]=1.;
      for(i=1;i<=MAXLINK;i++)         /* read link lengths */
    {
      fscanf(data,"%f",&a[i]);
    }
                                      /* rel. origin of frame (i) in (i-1) */
    O[1][X]=0.;   O[1][Y]=0.;   O[1][Z]=0.;   O[1][U]=1.;
    O[2][X]=0.;   O[2][Y]=0.;   O[2][Z]=a[1]; O[2][U]=1.;
    O[3][X]=0.;   O[3][Y]=a[2]; O[3][Z]=0.;   O[3][U]=1.;
    O[4][X]=0.;   O[4][Y]=a[3]; O[4][Z]=0.;   O[4][U]=1.;
    O[5][X]=0.;   O[5][Y]=a[4]; O[5][Z]=0.;   O[5][U]=1.;
    O[6][X]=0.;   O[6][Y]=0.;   O[6][Z]=0.;   O[6][U]=1.;
    O[7][X]=a[6]; O[7][Y]=0.;   O[7][Z]=0.;   O[7][U]=1.;
    fscanf(motion,"%f",&dt);               /* read time step */
    fprintf(out,"%f\n",dt);                /* write dt to out file */
    fprintf(out,"%d %d %d\n\n",ii,jj,kk);  /* write gripper Cardan convention to out file */
 for(t=0;eof(motion);t+=dt)                /* main loop */
    {
          for(i=1;i<=MAXLINK;i++)
          {
                ierr=fscanf(motion,"%f %f %f",&q,&qp,&qpp);
                if(ierr!=3)
                     exit(0);

                rotat24(axis[i],q,O[i],mreli_1[i]);             /* builds rel. pos. matrix */
                                                   /* builds rel. vel. and acc. matrices */
                velacctoWH3(Rev,axis[i],qp,qpp,O[i],Wreli_1[i],Hreli_1[i]);
                molt4(mabs[i-1],mreli_1[i],mabs[i]);       /* abs. pos. matrix of frame (i) */

                trasf_mami(Wreli_1[i],mabs[i-1],Wrel0[i]);   /* W and H matrices in frame 0 */
                n_skew34(Wrel0[i]);
                trasf_mami(Hreli_1[i],mabs[i-1],Hrel0[i]);
                sum4(Wabs[i-1],Wrel0[i],Wabs[i]);            /* abs. vel. and acc. matrices */
                coriolis(Habs[i-1],Hrel0[i],Wabs[i-1],Wrel0[i],Habs[i]);
          }
          idmat4(mreli_1[MAXLINK+1]);                  /* rel. position matrix of frame (7) */
          mreli_1[MAXLINK+1][X][U]=O[7][X];
          clear4(Wreli_1[MAXLINK+1]);                  /* rel W and H matrices of frame (7) */
          clear4(Hreli_1[MAXLINK+1]);
```

```
            molt4(mabs[MAXLINK],mreli_1[MAXLINK+1],mabs[MAXLINK+1]);
                                                            /* W and H matrices in frame (0) */
            trasf_mami(Wreli_1[MAXLINK+1],mabs[MAXLINK],Wrel0[MAXLINK+1]);
            n_skew34(Wrel0[MAXLINK]);
            trasf_mami(Hreli_1[MAXLINK+1],mabs[MAXLINK],Hrel0[MAXLINK+1]);
                                                              /* abs. vel. and acc. matrices */
            sum4(Wabs[MAXLINK],Wrel0[MAXLINK+1],Wabs[MAXLINK+1]);
            coriolis(Habs[MAXLINK],Hrel0[MAXLINK+1],Wabs[MAXLINK], Wrel0[MAXLINK+1],
                     Habs[MAXLINK+1]);
                                                                 /* extracts Cardan angles */
            Htocardan(mabs[MAXLINK+1],Wabs[MAXLINK+1],Habs[MAXLINK+1], ii,jj,kk,
                      q1,q2,qp1,qp2,qpp1,qpp2);
            Aus[X][U]=mabs[MAXLINK+1][X][U];
            Aus[Y][U]=mabs[MAXLINK+1][Y][U];
            Aus[Z][U]=mabs[MAXLINK+1][Z][U];
            trasf_miam(Wabs[MAXLINK],Aus,Waus);                        /* transform velocity */
            trasf_miam(Habs[MAXLINK],Aus,Haus);         /* and acceleration in ausiliar frame */
                            /* extracts Cardan angles (and their time derivatives) of gripper */
            Htocardan(mabs[MAXLINK+1],Waus,Haus,ii,jj,kk,q1,q2,qp1,qp2,qpp1,qpp2);

            printf("Time=%f\n",t);                                        /* output results */
            printm4("The position matrix of the gripper is:",mabs[MAXLINK+1]);
            printm4("The velocity matrix of the gripper is:",Waus);
            printm4("The acceleration matrix of the gripper is:",Haus);
            printf("\nPress any key to continue\n");
            getch();
            fprintf(out,"%f %f %f\n",q1[0],q1[1],q1[2]);
            fprintf(out,"%f %f %f\n",qp1[0],qp1[1],qp1[2]);
            fprintf(out,"%f %f %f\n",qpp1[0],qpp1[1],qpp1[2]);
            fprintf(out,"%f %f %f\n",mabs[MAXLINK+1][X][U],
                    mabs[MAXLINK+1][Y][U], mabs[MAXLINK+1][Z][U]);
            fprintf(out,"%f %f %f\n",Waus[X][U],Waus[Y][U],Waus[Z][U]);
            fprintf(out,"%f %f %f\n",Haus[X][U],Haus[Y][U],Haus[Z][U]);
      }
      fcloseall();
}
```

7.6.7 Inverse kinematics

The program for the inverse kinematics reads the robot description and the requested motion for the gripper producing the correspondent joints motion.Run the program as follows:

```
a:\ >ELB_I_DH ELBOW.DAT GRIPPER.MOT GUESS.1st JOINT.MOT
```

where:

`GRIPPER.MOT` is a file with the same format as the output file obtained from the sample programs for the direct kinematics.

`JOINT.MOT` is a file with the same format as the input file for the sample programs for the direct kinematics. This file will contain the joint motion. This output file has a format compatible with the input files for the direct kinematics.

`GUESS.1ST` is a file containing 6 value to be used as first guess for the iterative process which evaluates the joint angles.

7.6.8 The sample program ELB_I_DH.

```
/* elb_I_DH.c program for the INVERSE kinematics of ELBOW robot. Frames assigned according to
   Denavit and Hartenberg conventions. The output of this program is compatible with the input
   of elb_d_dh.c the input  of this program is compatible with tho output of elb_d_dh.c */
/* Note: To compile this program the type real must be set equivalent to the type float
     (see also User's Manual). This is necessary because the formatting string of the
```

```
      fscanf function have been written using %f as descriptor. */
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include <stdlib.h>
#include "spacelib.h"
#include "linear.h"
#define MAXLINK 6
void main(int argc,char *argv[])
{                                      /* Denavit & Hartemberg's parameters */
      real theta[MAXLINK+1]={0.,0.,0.,0.,0.,0.,0.};
      real d[MAXLINK+1]={0.,0.,0.,0.,0.,0.,0.};
      real b[MAXLINK+1]={0.,0.,0.,0.,0.,0.,0.};
      real fi[MAXLINK+1]={0.,PIG_2,0.,0.,3*PIG_2,PIG_2,0.};
      real a[MAXLINK+1];
      VECTOR q1,                       /* Eul./Card. angle configurations */
             qp1,                      /* Eul./Card. angle first time der. */
             qpp1,vel,acc;             /* Eul./Card. angle sec. time der. */
                                       /* array of joint pos. variables */
      real q[MAXLINK];                 /* joint angles */
      real qp[MAXLINK];                /* array of joint vel. variables */
      real qpp[MAXLINK];               /* array of jint acc. variables */
      real ds[MAXLINK],                /* sol. of the eq. J*dq=ds */
           dq[MAXLINK],                /* sol. of Newton/Raphson alg. step */
           buf[MAXLINK];
      real t,dt;
      real toll=0.0005;                /* precision of the solution */
      int maxiter=15;                  /* max. num. of iter. in N-R alg. */
      int ierr,i,k,p;
        int ii,jj,kk;                  /* Eul./Card. convention (gripper orientation) */
      int rank;                        /* rank of linear system */
      real n;
      POINT O,orig=ORIGIN;
      real Jac[MAXLINK][MAXLINK];      /* Jacobian matrix */
      MAT4 mtar,                       /* target position matrix */
           mrelp_1[MAXLINK+1],    /* array containing pos.mat.of frame (p) seen in frame (p-1)*/
           mabs[MAXLINK+1],        /* array containing abs.pos.mat.of frame (p) in base frame */
           mabsinv,                    /* invers position matrix of the frame positioned in the
                                          centre of the gripper */
           Lrelp,                      /* L relative matrix of p-th joint seen in frame (p-1) */
           Lrel0,                      /* L relative matrix of p-th joint seen in base frame */
           dm,dS,
           Wrelp_1[MAXLINK+1],         /* array containing rel. vel. mat. of
                                          frame (p) seen in frame (p-1) */
           Wrel0[MAXLINK+1],           /* array containing rel. vel. mat. of
                                          frame (p) seen in base frame */
           Wabs[MAXLINK+1],            /* array containing abs. vel. mat. of
                                          frame (i) in base frame */
           Wtar,                       /* target velocity matrix */
           Hrelp_1[MAXLINK+1],         /* array containing rel. acc. mat. of
                                          frame (p) seen in frame (p-1) */
           Hrel0[MAXLINK+1],           /* array containing rel. acc. mat. of
                                          frame (p) seen in base frame */
           Habs[MAXLINK+1],            /* array containing abs. acc. mat. of
                                          frame (i) in base frame */
             Htar,                     /* target acceleration matrix */
             dH;                       /* Htar - H~    H~ is the acceleration
                                          evaluated with qpp=0 */
        MAT4 Last ={ {0.,1.,0.,0.},    /* transformation matrix from */
                     {0.,0.,1.,0.},    /* frame (6) to gripper       */
                     {1.,0.,0.,0.},    /* element Z-U is in a[6]     */
                     {0.,0.,0.,1.} };  /*                            */
```

```
MAT4 gripper;                   /* abs. frame of gripper */
POINT first=ORIGIN;             /* origin of frame 0 with respect to base,
                                   Z value is in a[1] */
MAT4 Aus,                       /* Ausiliar frame, origin in gripper, parallel to base */
     Waus, Haus;                /* abs. gripper vel. and acc. in Aus frame */
  FILE *data;                   /* file containing robot description */
  FILE *motion;                 /* file containing motion descr. */
  FILE *out;                    /* output file */
  FILE *guess;                  /* 1st guess for q */
  if (argc!=5)                  /* testing parameters */
{
      fprintf(stderr,"Usage: Elb_i_dh data_file motion_file q_guess.1st output_file\n");
      exit(1);
}
data=fopen(argv[1],"r");
if(data==NULL)
{
      fprintf(stderr,"Error: Unable to open data_file\n");
      exit(2);
}
motion=fopen(argv[2],"r");
if(motion==NULL)
{
      fprintf(stderr,"Error: Unable to open motion_file\n");
      exit(3);
}
guess=fopen(argv[3],"r");
if(guess==NULL)
{
      fprintf(stderr,"Error: Unable to open 1st_guess_file\n");
      exit(4);
}
out=fopen(argv[4],"w");
if(out==NULL)
{
      fprintf(stderr,"Error: Unable to open output_file\n");
      exit(5);
}
for(p=1;p<=MAXLINK;p++)
      fscanf(data,"%f",&a[p]);   /* read robot description */
for(p=0;p<MAXLINK;p++)
      fscanf(guess,"%f",&q[p]);  /* 1st guess for q */
                                /* matrices initialization */
clear(M Jac,MAXLINK,MAXLINK);
first[Z]=a[1];
rotat24(Z,PIG_2,first,mabs[0]);  /* pos. mat. of frame 0 from base frame */
Last[Z][U]=a[6];                /* gripper position in frame 6 */
idmat4(Aus);
clear4(Waus); clear4(Haus);
clear4(Wabs[0]);                /* abs velocity of base and of frame 0 */
clear4(Habs[0]);                /*     acceleration                    */
a[1]=a[6]=0;                    /* D&H parameter 'a' of link 1 and link 6 are zero */
fscanf(motion,"%f",&dt);                /* read time step */
fscanf(motion,"%d %d %d",&ii,&jj,&kk);  /* read Cardan convention */
fprintf(out,"%f\n\n",dt);
for(t=0;;t+=dt)                 /* main loop */
{
      ierr=fscanf(motion,"%f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f",
                 &q1[0],&q1[1],&q1[2],&qp1[0],&qp1[1],&qp1[2],
                 &qpp1[0],&qpp1[1],&qpp1[2],&O[X],&O[Y],&O[Z],
                 &vel[0],&vel[1],&vel[2],&acc[0],&acc[1],&acc[2]);
      if (ierr!=18)
```

```
          exit(0);
    O[U]=1.;
    cardantoM(q1,ii,jj,kk,O,mtar);                  /* builds target position matrix */
    vmcopy(M O,3,4,Col,M mtar,4,4);

    for (k=0;k<maxiter;k++)
    {
          for (p=1;p<=MAXLINK;p++)
          {                                                  /* builds rel. pos. matrix */
               dhtom(Rev,theta[p],d[p],b[p],a[p],fi[p],q[p-1],mrelp_1[p]);
                molt4(mabs[p-1],mrelp_1[p],mabs[p]);        /* builds abs. pos. matrix */
                makeL2(Rev,Z,0.,orig,Lrelp);    /* builds rel. L matrix in base frame */
                trasf_mami(Lrelp,mabs[p-1],Lrel0);/* builds rel L matrix in frame (p) */
                buf[0]=Lrel0[X][U];
                buf[1]=Lrel0[Y][U];
                buf[2]=Lrel0[Z][U];
                buf[3]=Lrel0[Z][Y];
                buf[4]=Lrel0[X][Z];
                buf[5]=Lrel0[Y][X];
                vmcopy(M buf,6,p,Col,M Jac,MAXLINK, MAXLINK);
          }
          molt4(mabs[MAXLINK],Last,gripper);
          sub4(mtar,gripper,dm);
          n=norm4(dm);
          if (n>toll)                              /* tests if sol. has been reached */
          {
                invers(gripper,mabsinv);
                molt4(dm,mabsinv,dS);
                ds[0]=dS[X][U];
                ds[1]=dS[Y][U];
                ds[2]=dS[Z][U];
                ds[3]=dS[Z][Y];
                ds[4]=dS[X][Z];
                ds[5]=dS[Y][X];
                rank=solve(M Jac,M ds,M dq,MAXLINK);
                                                /* builds the joint var. at next step */
                if(rank!=MAXLINK) printf("*** rank is %d: singular position!\a",rank);
                sum(M q,M dq,M q,MAXLINK,1);
          }
          else
                break;
    }
    if (k<maxiter)
    {
    Aus[X][U]=gripper[X][U];
    Aus[Y][U]=gripper[Y][U];
    Aus[Z][U]=gripper[Z][U];
                                                    /* builds target velocity matrix */
    cardantoW(q1,qp1,ii,jj,kk,O,Waus);
    vmcopy(M vel,3,4,Col,M Waus,4,4);
    trasf_mami(Waus,Aus,Wtar); /* transform velocity from ausiliar frame to base frame*/
    cardantoH(q1,qp1,qpp1,ii,jj,kk,O,Haus);      /* builds target acceleration matrix */
    vmcopy(M acc,3,4,Col,M Haus,4,4);
    trasf_mami(Haus,Aus,Htar);     /* transform acceleration from ausiliar frame to base
                                      frame */
                                                     /* builds joint volocity array */
    buf[0]=Wtar[X][U];
    buf[1]=Wtar[Y][U];
    buf[2]=Wtar[Z][U];
    buf[3]=Wtar[Z][Y];
    buf[4]=Wtar[X][Z];
    buf[5]=Wtar[Y][X];
```

```
        rank=solve(M Jac,M buf,M qp,MAXLINK);
        if(rank!=MAXLINK) printf("*** rank is %d: singular position!\a",rank);
        for(i=1;i<=MAXLINK;i++)                                       /* acceleration */
            { velacctoWH(Rev,qp[i-1],0.,Wrelp_1[i],Hrelp_1[i]);
              trasf_mami(Wrelp_1[i],mabs[i-1],Wrel0[i]);  /* W and H matrices in frame 0 */
              trasf_mami(Hrelp_1[i],mabs[i-1],Hrel0[i]);
              sum4(Wabs[i-1],Wrel0[i],Wabs[i]);           /* abs. vel. and acc. matrices */
              coriolis(Habs[i-1],Hrel0[i],Wabs[i-1],Wrel0[i],Habs[i]);
        }
        sub4(Htar,Habs[MAXLINK],dH);
        buf[0]=dH[X][U];                                /* builds joint acceleration array */
        buf[1]=dH[Y][U];
        buf[2]=dH[Z][U];
        buf[3]=dH[Z][Y];
        buf[4]=dH[X][Z];
        buf[5]=dH[Y][X];
        rank=solve(M Jac,M buf,M qpp,MAXLINK);
        if(rank!=MAXLINK) printf("*** rank is %d: singular position!\a",rank);
        }
        else
        {
              fprintf(stderr,"Newton-Raphson method doesn't converge\n");
              exit(1);
        }
                                                                    /* output results */
        printf("\nTime=%f",t);
            printv("The joint angles q are",q,6);
            printv("The joint velocity qp are",qp,6);
            printv("The joint ecceleration qpp are",qpp,6);
        getch();
        for (p=0;p<MAXLINK;p++)
        {
              fprintf(out,"%15.5f",q[p]);
              fprintf(out,"%15.5f",qp[p]);
              fprintf(out,"%15.5f\n",qpp[p]);
        }
        fprintf(out,"\n");
    }
    fcloseall();
}
```

Bibliography

[1] Denavit J., Hartenberg R. S., "A Kinematics Notation for Lower-Pair Mechanisms based on Matrices", *Trans. ASME J. Appl. Mech.* 22, 215-221, June 1955.

[2] G. Legnani "Robotica Industriale", CEA Casa Editrice Ambrosiana, 2003, isbn-88-408-1262-8.

[3] G. Legnani, F. Casolo, P. Righettini, B. Zappa "A homogeneus matrix approach to 3D kinematics and dynamics - I. Theory" Mech. Mach. Theory Vol. 31, No. 5, pp. 573-583, 1996.

[4] G. Legnani, F. Casolo, P. Righettini, B. Zappa "A homogeneus matrix approach to 3D kinematics and dynamics - II. Applications to Chains of Rigid Bodies and Serial Manipulators" Mech. Mach. Theory Vol. 31, No 5, pp. 589-605, 1996.

[5] G. Legnani, R. Riva "Un Modello per lo Studio di Robot Industriali" 7th AIMETA National Congress 1984. Trieste, Italy.

[6] G. Legnani, "Matrix Methods in Robot Kinematics" (in Italian), Doctoral Dissertation, Politecnico di Milano, Italy 1986.

[7] R. Faglia, G. Legnani "S.A.M. Robot: Simulazione ed Analisi Meccanica di Robot Industriali" Pixel n.11 1987 Ed. Il Rostro, Milano, Italy.

[8] F. Casolo, G. Legnani "A New Approach to Identify Kinematic Peculiarities in Human Motion" XII Int. Cong. of Biomechanics - UCLA 1989 - Los Angeles USA.

[9] F. Casolo, G. Legnani "A New Approach to the Dynamics of Human Motion" 2nd Int. Symposium on Computer Simulation in Biomechanics, 1989 Davis, California USA.

[10] F. Casolo, G. Legnani "A Consistent Matrix Approach for the Kinematics and Dynamics of Systems of Rigid Bodies" AIMETA nat.cong. 1988 Bari-Italy.

[11] G. Legnani, R. Riva "A New Matrix Approach to the Dynamic of Spatial Mechanisms" The 5th Int. Symposium on Linkages and Computer Aided Design Methods. Bucharest 1989.

[12] F. Casolo, R. Faglia, G. Legnani "Three dimensional analysis of systems of rigid bodies" X National Symposium DECUS, Baveno Italy 13-14 Aprile 1989.

[13] F. Casolo, R. Faglia, G. Legnani "Industrial Robots: Application of a Rational Solution for the Direct and Inverse Dynamic Problem." 2nd Int. Workshop on Advances in Robot Kinematics. Linz - Austria 10/12 Sept. 1990.

[14] F. Casolo, G. Legnani "Dynamic Simulation of Whole Body Motion: an Application to Horse Vaulting" 10th AIMETA national congress. Pisa Italy, 2-5 October 1990.

[15] R. P. Paul "Robot Manipulators: Mathematics, Programming and Controlling" The MT Press Cambridge, England, 1981.

[16] N. Doriot, L. Cheze, "A three-dimensional kinematic and dynamic study of the lower limb during the stance phase of gait using an homogeneous matrix approach." IEEE Trans Biomed Eng. 2004 Jan; 51(1): 21-7. Related Articles, Links.

Appendix A

Comparison between the versions of SpaceLib©.

A.1 Table of comparison

The following table lists all the SpaceLib functions comparing the three releases (C, MATLAB and Maple 9).

Table A.1: Table of comparison

C	MATLAB ©	Maple 9 ©	**description**
actom	actom	actoM	*Actions to Matrix.*
angle	angle	Angle	*Angle between points.*
axis	aaxis	Axis axis	*Axis of Frame.*
cardanto_G cardanto_G3 cardanto_G4	cardatog	cardantoG	*Cardan angles to angular acceleration matrix.*
cardanto_omega cardanto_omega3 cardanto_omega4	cardtome	cardanto_OMEGA	*Cardan angles to angular velocity matrix.*
cardanto_OMEGA	cardtoom	cardanto_omega	*Cardan angles to angular velocity.*
cardanto_OMEGAPTO	cardompt	cardanto_omegapto	*Cardan angles to angular acceleration.*
		cardanto_OMEGAPTO	*Cardan angles to angular acceleration matrix.*
cardantoH	cardatoh	cardantoH	*Cardan angles to acceleration matrix.*
cardantol	cardatol	cardantoL	*Cardan angles to L matrix.*
cardantoM	cardatom	cardantoM	*Cardan angles to position matrix.*
cardantor cardantor3 cardantor4	cardator	cardantoR	*Cardan (or Euler) angles to rotation matrix.*
cardantoW	cardatow	cardantoW	*Cardan angles to velocity matrix.*
cardantoWPROD WPRODtocardan	cardtowp	cardantoWPROD	
cardtoH		cardtoH	*Cardan angles to acceleration matrix.*
cardtoM		cardtoM	*Cardan angles to position matrix.*

C	MATLAB ©	Maple 9 ©	description
cardtoW		cardtoW	*Cardan angles to velocity matrix.*
clear clear3 clear4 clearv3	clearmat [1]	clear [2] nullM [2]	*Clear a matrix (fill it with zeros).*
coriolis	coriolis	coriolis	*Coriolis' theorem.*
cross	cross [3]	cross	*Vector cross product.*
crossMtoM	crossmto [3] crosstom [1]	crossMtoM [2]	*Cross product for matrices (Matricial form).*
crossvtom crossmtom [3]			*Cross product for matrices (Vector form).*
deg	deg	deg	*Conversion from radians to degrees.*
dhtom	dhtom	dhtom	*Denavit & Hartenberg's parameters to matrix (Extended version).*
DHtoMstd	dhtomstd	DHtoMstd	*Denavit & Hartenberg's parameters to matrix (Standard Version).*
dist	distp dist [3]	distp	*Distance between two points.*
distpp	distpp	distpp	*Distance of point from a plane.*
dot	dot [3] dot3	vdot3	*3 elements vector dot (scalar) product.*
dot2	dot2	vdot	*any elements vector dot (scalar) product.*
dyn_eq	dyn eq	dyn_eq	*Solve Direct Dynamics system.*
dzerom			*Double Machine's zero.*
eultoH		eultoH	*Cardan angles to acceleration matrix.*
eultoM		eultoM	*Cardan angles to position matrix.*
eultoW		eultoW	*Cardan angles to velocity matrix.*
extract	extract	extract	*Extracts unit vector of screw axis and rotation angle from rotation matrix.*
fprintm3 fprintm4 fprintv	fprintm	fprintm	*Print a matrix (with a comment) on a file.*
		fmod	*fractional part of x/y.*
frame4P	frame4p	frame4P	*Frame from three points.*
frame4V	frame4v	frame4V	*Frame from a point and two vectors.*
frameP frameP3 frameP4	framep	frameP	*Frames from points.*
frameV frameV3 frameV4	framev	frameV	*Frame from vectors.*
fzerom			*Float Machine's zero.*
	grad [3]		*Conversion from radians to degrees.*

[1] *Function not really necessary in MATLAB©: provided just for compatibility with the* C *version of* SpaceLib©.
[2] *Function not really necessary in Maple 9©: provided just for compatibility with the* C *version of* SpaceLib©.
[3] obsolete version.

C	MATLAB©	Maple 9©	description
`gtom`	`gtom`	`gtoM`	*Gravity acceleration to Matrix.*
`Gtomegapto`	`gtomgapt`	`Gtomegapto`	*G to omega dot.*
`Htocard`		`Htocard`	*Acceleration matrix to Cardan angles.*
`Htocardan`	`htocarda`	`Htocardan`	*Acceleration matrix to Cardan angles.*
`Htoeul`		`Htoeul`	*Acceleration matrix to Cardan angles.*
`Htonaut`		`Htonaut`	*Acceleration matrix to Cardan angles.*
`idmat` `idmat3` `idmat4`	`idmat` [1]	`idmat` [2] `eye` [2]	*Identity matrix.*
`intermediate`	`intermed`	`intermediate`	*Middle weight point.*
`inters2pl`	`inter2pl`	`inters2pl`	*Intersecton of two planes.*
`intersection`	`intersect`	`intersection`	*Intersection between two lines.*
`interslpl`	`interlpl`	`interslpl`	*Intersecton of line and plane.*
`invA`	`inva`	`invA`	
`invers`	`invers`	`invers`	*Inverse of a position matrix.*
	`jrand`	`jrand`	*Creates a random matrix with elements in the range min..max.*
`jtoJ`	`jtoj`	`jtoJ`	*Inertia moment and mass to inertia matrix.*
`line2p`	`line2p`	`line2p`	*Line from two points.*
`linear`	`linears`	`linear`	*Linear System.*
`linepvect`	`linpvect`	`linepvect`	*Line from point and vector.*
`makeL`	`makel`	`makeL`	*Builds a L matrix.*
`makeL2`	`makel2`	`makeL2`	*Builds a L matrix - version 2.*
		`Mcheck`	*Checks for Position/Rotation Matrix.*
		`Mcheck2`	*Checks for Position/Rotation Matrix version 2.*
`mcopy` `mcopy3` `mcopy4`	`mcopy` [1]		*Matrix copy.*
`middle`	`middle`	`middle`	*Middle point.*
`minvers`	`minvers` [1]	`minvers` [2]	*Matrix Inverse System.*
		`Miszero`	*Test Zero Matrix.*
`mmcopy` `mcopy34` `mcopy43`	`mmcopy` [1]		*Copy a part of a matrix.*
`mod`	`mod` [3] `modulus`	`modulus`	*Module of a vector.*
`molt` `molt3` `molt4`	`molt` [1]		*Matrix multiplication.*
`moltmv3`			*Multiply a matrix by a vector.*
`moltp`			*Multiply a matrix by a point.*
`Mtocard`		`Mtocard`	*Position matrix to Cardan angles.*
`Mtocardan`	`mtocarda`	`Mtocardan`	*Position matrix to Cardan angles.*

C	MATLAB ©	Maple 9 ©	**description**
Mtoeul		Mtoeul	*Position matrix to Cardan angles.*
Mtonaut		Mtonaut	*Position matrix to Cardan angles.*
mtoscrew	mtoscrew	Mtoscrew	*Matrix to screw.*
mtov mtov3 mtov4	mtov	Mtov	*Matrix to vector.*
mvcopy			*Copy a row or a column from a matrix.*
nauttoH		nauttoH	*Cardan angles to acceleration matrix.*
nauttoM		nauttoM	*Cardan angles to position matrix.*
nauttoW		nauttoW	*Cardan angles to velocity matrix.*
norm norm3 norm4			*Norm of a matrix.*
norm_simm_skew n_simm3 n_simm34 n_simm4 n_skew3 n_skew34 n_skew4	normskew	norm_simm_skew	*Normalizes symmetric or skew-symmetric matrices.*
normal normal3 normal4	normal normal3	normalR	*Normalizes (orthogonalises) a 3×3 rotation matrix or the 3×3 upper-left submatrix of a position matrix.*
	normal_g	normal_g	*Normalizes (orthogonalises) any square matrix.*
pcopy			*Point copy.*
plane	plane	plane3p	*Plane from three points.*
plane2	plane2	planepv	*Plane from point and vector.*
printm printm4 printv	printm	printm	*Print a matrix (with a comment) on the screen.*
printmat iprintmat			*Prints a real elements matrix.*
prmat	prmat	prmat	*Print a position matrix for GRP man graphics post-processor.*
project	project	project	*Project a point on a plane.*
projponl	projponl	projponl	*Projection of point on line.*
psedot	psedot	psedot	*Pseudo scalar product.*
pseudo_inv	pseudinv [1]	pseudo_inv [2]	*Pseudo inverse of a matrix.*
		Origin	*Origin Point.*
rmolt rmolt3 rmolt4	rmolt [2]		*Multiply a scalar r by a matrix.*
rmoltv			*Multiply a scalar r by a vector.*
rad	rad	rad	*Conversion from degrees to radians.*
rotat	rotat	rotat	*Builds the rotation matrix R.*

C	MATLAB ©	Maple 9 ©	**description**
rotat2 rotat23	rotat2	rotat2	*Rotation around a frame axis.*
rotat24	rotat24	rotat24	*Rotation matrix around an axis with origin in a given point.*
rotat34	rotat34	rotat34	*Rotation matrix around an axis with origin in a given point.*
rtocardan rtocardan3 rtocardan4	rtocarda	Rtocardan	*Rotation matrix to Cardan (or Euler) angles.*
screwtom	screwtom	screwtoM	*Screw to Matrix.*
skew skew4	skew	skew	*Skew operator.*
solve	solve_l [1]	solver [2]	*Solve linear system.*
sub sub3 sub4 subv	sub [1]		*Subtraction for matrices or vectors.*
sum sum3 sum4 sumv	ssum [1]		*Sum of matrices or vectors.*
trac_ljlt4	tracljlt	trac_ljlt	*Trace of* L_1 J ${L_2}^t$.
traslat	traslat	traslat	*Builds the matrix m of a traslation along a vector.*
traslat2	traslat2	traslat2	*Builds the matrix m of a traslation along a frame axis.*
traslat24	traslat24	traslat24	*Builds the matrix m of a traslation along a frame axis with origin in a given point.*
transp transp3 transp4	transp [1]		*Transpose of a matrix.*
trasf_mami	mami	trasf_mami	*Transforms a matrix by the rule of* M A M^{-1} *(mami = mAminverse).*
trasf_mamt trasf_mamt4	mamt	trasf_mamt	*Transforms a matrix by the rule of* M A M^t *(mami = mAmtranspose).*
trasf_miam	miam	trasf_miam	*Transforms a matrix by the rule of* M^{-1} A M *(miam = minverseAm).*
trasf_miamit	miamit	trasf_miamit	*Transforms a matrix by the rule of* M^{-1} A M^{-t} *(miamit = minverseAminverse transposed).*
unitv	unitv	unitv	*Unit vector.*
vcopy			*Vector copy.*
vect	vect	vect	*Vector between points.*
vector	vector		*Evaluate a vector (from module and direction).*
		vect3	*Vector(3) from vector.*

C	MATLAB ©	Maple 9 ©	**description**
velacctoWH	veactowh	velactoWH	*Velocity and Acceleration to W and H matrices.*
velacctoWH2	vactowh2	velactoWH2	*Velocity and Acceleration to W and H matrices - version 2.*
velacctoWH3	vactowh3	velactoWH3	*Velocity and Acceleration to W and H matrices - version 3.*
		viszero	*Test Zero vector.*
vmcopy			*Copy a vector into a row or a column of a matrix.*
vtom vtom3 vtom4	vtom	vtoM	*Vector to matrix.*
Wtocard		Wtocard	*Velocity matrix to Cardan angles.*
Wtocardan	wtocarda	Wtocardan	*Velocity matrix to Cardan angles.*
Wtoeul		Wtoeul	*Velocity matrix to Cardan angles.*
WtoL	wtol	WtoL	*Extracts L matrix from the corresponding W matrix.*
Wtonaut		Wtonaut	*Velocity matrix to Cardan angles.*
Wtovel	wtovel	Wtovel	*Velocity matrix to velocity parameters.*
zerom			*Machine's zero.*

Index