EusLisp

EusLisp version 9.29/ irteus version 1.2.5

Reference Manual

-Extended Robot Modelling-

ETL-TR-95-19 + JSK-TR-10-03

2023 年 3 月 3 日

(irteus 1.2.5)
The University of Tokyo
Graduate School of Information Science and Technology　Department of Mechano Informatics
野沢 峻一 , 植田 亮平, 岡田 慧
ueda@jsk.t.u-tokyo.ac.jp nozawa@jsk.t.u-tokyo.ac.jp k-okada@jsk.t.u-tokyo.ac.jp
〒 113-8656 東京都文京区本郷 7-3-1 工学部 2 号館 7 階 73B2
(EusLisp 9.29)
通商産業省　工業技術院
電子技術総合研究所 知能システム部
松井 俊浩, 原 功, 中垣 博文 (九州電力)
matsui@etl.go.jp, hara@etl.go.jp, nakagaki@etl.go.jp
〒 305 茨城県つくば市梅園 1-1-4

目 次

第I部
EusLisp Basics

1 Introduction

EusLisp is an integrated programming system for the research on intelligent robots based on Common Lisp and Object-Oriented programming. The principal subjects in the field of robotics research are sensory data processing, visual environment recognition, collision avoiding motion planning, and task planning. In either problem, three dimensional shape models of robots and environment play crucial roles. A motivation to the development of EusLisp was a demand for an extensible solid modeler that can easily be made use of from higher level symbolic processing system. Investigations into traditional solid modelers proved that the vital requirement for their implementation language was the list processing capability to represent and manage topology among model components. Numerical computation power was also important, but locality of geometric computation suggested the provision of vector/matrix functions as built-ins would greatly ease programming.

Thus the primary decision to build a solid modeler in a Lisp equipped with a geometric computation package was obtained. Although a solid modeler provides facilities to define shapes of 3D objects, to simulate their behaviors, and to display them graphically, its applications are limited until it is incorporated in robot modules mentioned above. These modules also need to be tightly interconnected to achieve fully integrated robot systems. EusLisp sought for the framework of this integration in object-oriented programming (OOP). While OOP promotes modular programming, it facilitates incremental extension of existing functions by using inheritance of classes. In fact, components in the solid modeler, such as bodies, faces, and edges, can orderly be inplemented by extending one of the most basic class *coordinates*. These components may have further subclasses to provide individual functions for particular robot applications.

Based upon these considerations, EusLisp has been developped as an object-oriented Lisp which implements an extensible solid modeler[**?**]. Other features include intertask communication needed for the cooperative task coordination, graphics facilities on X-window for visual user interface, and foreign language interface to support mixed language programming.

In the implementation of the language, two performance-effective techniques were invented in type discrimination and memory management [**?**, **?**, **?**]. The new type discrimination method guarantees constant-time discrimination between types in tree structured hiearchy without regard to the depth of trees. Heap memory is managed in Fibonacci buddy method, which improves memory efficiency without sacrificing runtime or garbage-collection performance.

This reference manual describes EusLisp version 7.27 in two parts, *EusLisp Basics and EusLisp Extensions.* The first part describes Common Lisp features and object-oriented programming. Since a number of literatures are available on both topics, the first part is rather indifferent except EusLisp's specific features as described in *Interprocess Communication and Network*, *Toplevel Interaction*, *Disk Save*, etc. Beginners of EusLisp are advised to get familiar with Common Lisp and object oriented programming in other ways [**?**, **?**]. The second part deals with features more related with robot applications, such as *Geometric Modelling*, *Image Processing*, *Manipulator Model* and so on. Unfortunately, the descriptions in this part may become incomplete or inaccurate because of EusLisp's rapid evolution. The update information is available via euslisp mailing list as mentioned in section **??**.

1.1 EusLisp's Object-Oriented Programming

Unlike other Lisp-based object-oriented programming languages like CLOS [**?**], EusLisp is a Lisp system built on the basis of object-orientation. In the former approach, Lisp is used as an implementation language for the object-oriented programming, and there is apparent distinction between system defined objects and user defined objects, since system data types do not have corresponding classes. On the other hand, every data structure in EusLisp except number is represented by an object, and there is no inherent difference between built-in data types, such as `cons` and `symbols`, and user defined classes. This implies that even the system built-in data types can be extended (inherited) by user-defined classes. Also, when a user defines his own class as a subclass of a built-in class, he can use built-in methods and functions for the new class, and the amount of description for a new program can be reduced. For example, you may extend the `cons` class to have extra field other than `car` and `cdr` to define queues, trees, stacks, etc. Even for these instances, built-in functions for built-in cons are also applicable without any loss of efficiency, since those functions recognize type hierarchy in a constant time. Thus, EusLisp makes all the system built-in facilities open to programmers in the form of extensible data types. This uniformity is also beneficial to the implementation of EusLisp, because, after defining a few kernel functions such as **defclass**, **send**, and **instantiate**, in the implementation language, most of house-keeping functions to access the internal structure of built-in data types can be coded in EusLisp itself. This has much improved the reliability and maintainability of EusLisp.

1.2 Features

object-oriented programming EusLisp provides single-inheritance Object-Oriented programming. All data types except numbers are represented by objects whose behaviors are defined in their classes.

Common Lisp EusLisp follows the specifications of Common Lisp described in [**?**] and [**?**] as long as they are consistent with EusLisp's goal and object-orientation. See next subsection for incompatibilities.

compiler EusLisp's compiler can boost the execution 5 to 30 times as fast as the interpreted execution. The compiler keeps the same semantics as the interpreter.

memory management Fibonacci buddy method, which is memory efficient, GC efficient, and robust, is used for the memory management. EusLisp can run on machines with relatively modest amount of memory. Users are free from the optimization of page allocation for each type of data.

geometric primitives Since numbers are always represented as immediate data, no garbage is generated by numeric computation. A number of geometric functions for arbitrary-sized vectors and matrices are provided as built-in functions.

geometric modeler Solid models can be defined from primitive bodies using CSG set operations. Mass properties, interference checking, contact detection, and so on, are available.

graphics Hidden-line eliminated drawing and hidden-surface eliminated rendering are available. Postscript output to idraw can be generated.

image processing Edge based image processing facility is provided.

manipulator model 6 D.O.F.s robot manipulator can easily be modeled.

Xwindow interface Three levels of Xwindow interface, the Xlib foreign functions, the Xlib classes and the original XToolKit classes are provided.

foreign-language interface Functions written in C or other languages can be linked into EusLisp. Bidirectional call between EusLisp and other language are supported. Functions in libraries like LINPACK become available through this interface. Call-back functions in X toolkits can be defined in Lisp.

unix binding Most of unix system calls and unix library functions are assorted as Lisp functions. Signal handling and asynchronous I/O are also possible.

multithread multithread programming, which enables multiple contexts sharing global data, is available on Solaris 2 operating system. Multithread facilitates asynchronous programming and improves real-time response[**?**, **?**]. If EusLisp runs on multi-processor machines, it can utilize parallel processors' higher computating power.

1.3 Compatibility with Common Lisp

Common Lisp has become the well-documented and widely-available standard Lisp [**?**, **?**]. Although EusLisp has introduced lots of Common Lisp features such as variable scoping rules, packages, sequences, generalized variables, blocks, structures, keyword parameters, etc., incompatibilities still remain. Here is a list of missing features:

1. multiple values: multiple-value-call,multiple-value-prog1, etc., are present only in a limited way;
2. some of data types: bignum, character, deftype, complex number and ratio (the last two are present only in a limited way);
3. some of special forms: progv, compiler-let,macrolet

Following features are incomplete:

4. closure – only valid for dynamic extent
5. declare,proclaim – inline and ignore are unrecognized

1.4 Revision History

1986 The first version of EusLisp ran on Unix-System5/Ustation-E20. Fibonacci buddy memory management, simple compiler generating M68020 assembly code, and vector/matrix functions were tested.

1987 The new fast type checking method is implemented. The foreign language interface and the SunView interface were incorporated.

1988 The compiler was changed to generate C programs as intermediate code. Since the compiler became processor independent, EusLisp was ported on Ultrix/VAX8800 and on SunOS3.5/Sun3 and /Sun4 . IPC facility using socket streams was added. The solid modeler was implemented. Lots of Common Lisp features such as keyword parameters, labeled print format to handle recursive data objects, generic sequence functions, readtables, tagbody, go, flet, and labels special forms, etc., were added.

1989 The Xlib interface was introduced. % read macro to read C-like mathematical expressions was made. manipulator class is defined.

1990 The XView interface was written by M.Inaba. Ray tracer was written. Solid modeler was modified to keep CSG operation history. Asynchronous I/O was added.

1991 The motion constraint program was written by H.Hirukawa. Ported to DEC station. Coordinates class changed to handle both 2D and 3D coordinate systems. Body composition functions were enhanced to handle contacting objects. CSG operation for contacting objects. The package system became compatible with Common Lisp.

1992 **Face+** and **face*** for union and intersection of two coplanar faces were added. Image processing facility was added. The first completed reference manual was printed and delivered.

1993 EusLisp was stable.

1994 Ported to Solaris 2. Multi-context implementation using Solaris's multithread facility. XToolKit is built. Multi robot simulator, MARS was written by Dr. Kuniyoshi. EusLisp organized session at RSJ 94, in Fukuoka.

1995 The second version of the reference manual is published.

2010 Version 9.00 is releaced, The licence is changed to BSD.

2011 Add Darwin OS Support, Add model files.

2013 Add Cygwin 64 Bit support, expand MXSTACK from 65536 to 8388608, KEYWORDPARAMETER-LIMIT from 32 to 128.

2014 Use UTF-8 for documents, Version 9.10 is releaced.

2015 more error check on min/max, support arbitrary length for vplus, more quiet for non-ttyp mode, Version 9.11 is releaced.

2015 Version 9.12 is released, support ARM Version 9.13 is released, support class documentation Version 9.14 is released, fix assert API. Now message is optional (defmacro assert (pred &optional message) Version 9.15 is released, fix char comparison function (previous version retuns opossite result), support multiple argument at function /=, add url encode feature (escape-url function), support microsecond add/subtract in interval-time class Version 9.16 is released, added make-random-state, fixed bug in lib/llib/unittest.l

2016 Version 9.17 is released, add trace option in (init-unit-test), enable to read #f(nan inf). fix models/doc. Version 9.18 is released, support gcc-5. Version 9.20 is released, support OSX (gluTessCallback, glGenTexturesEXT), add GL_COLOR_ATTACHMENT constants, fix color-image class, (it uses RGB not BGR). Version 9.21 is released, fix :trim of hashtab class, enable to compile filename containing -, do not raise error when not found cygpq.dll (Cygwin) Version 9.22 is released, add :color option to :draw-box, :draw-polyline, :draw-star, with-output-to-string returns color instead of nil, print call stack on error, check if classof is called with pointer, pass symbol pointer to funcall in apply, add error check of butlast and append.

2017 Version 9.23 is released, support ARM64, udpate models.

2018 Version 9.24 is released, change trans.l to put .h file on same directory, fix potential segmentation error in READLINE, increase max count of pushsequence for 64bit machine, remove size limitation for READLINE, enable to compile filename containing '-', add pattern option in :methods, check norm is nan for ROTANGLE, force normalize norm vector in optional argument of vector-angle, fix error on :distance when point is on the same plane, fix compiler when argument is not integer wit (1+) / (1-), fix abs for 64bit machine, fix read-binary, use cfree instead of free, extend defun function for documentation, support 18.04. Version 9.25 is released, C defun() function now takes 5 arguments

FILES	this document
README	a brief guide to lisence, installation and sample run
VERSION	EUSLisp version number
bin	executables (eus, euscomp and eusx)
c/	EusLisp kernel written in C
l/	kernel functions written in EusLisp
comp/	EusLisp compiler written in EusLisp
clib/	library functions written in C
doc/	documentation (latex and jlatex sources and memos)
geo/	geometric and graphic programs
lib/	shared libraries (.so) and start-up files
llib/	Lisp library
llib2/	secondary Lisp library developed at UTYO
xwindow/	X11 interface
makefile@	symbolic link to one of makefile.sun[34]os[34],.vax, etc.
pprolog/	tiny prolog interpreter
xview/	xview tool kit interface
tool/	
vxworks/	interface with VxWorks real-time OS
robot/	robot models and simulators
vision/	image processing programs
contact/	motion constraint solver by H.Hirukawa [?, ?, ?]
demo/	demonstrative programs
bench/	benchmark programs

表 1: Directories in `*eusdir*`

includeing doc string. Version 9.26 is released, fix typo in manuals, move test code from jskeus repository, clean compile warnings, use minmemory instead of _end in all architecture for some compiler (aarch64/gcc-6), fix problem on call :draw-on after :draw-arrow, generate euslisp.hlp when compiled, enable to run :halve and :double in color-image.

2019 Version 9.27 is released. Fix documentation. Print E_USER within default error handler. Add :init method into ration class. Update Mesa version of GL constant files. Add :word-size=64 to *features* and refer this information to execute on 64bit machine.

2021 Version 9.28 is released. Fix bugs on foreign function call ARM. Add glpixmapsurface class for offline drawing. Fix compiled function name. Close file handler after reading help file. Set :primitive to set :csg on make-gdome and make-body-from-vertices. Use gcc as linker on i386.

2022 Version 9.29 is released. Introduce euspointer_t in the source code. call malloc()/cfree() within mainthread(ctx).

1.5 Installation

The installation procedure is described in `README`. The installation directory, which is assumed to be `"/usr/local/eus/"`, should be set to the global variable `*eusdir*`, since this location is referenced by **load** and the compiler.

Subdirectories in `*eusdir*` are described in table **??**. Among these, `c/, l/, comp/, geo/, clib/,` and `xwindow` contain essential files to make eus and eusx. Others are optional libraries, demonstration programs and contributions from users.

1.6 License

EusLisp is distributed under the following BSD License.

Until version 8.25, Euslisp is distributed under following licence.

EusLisp can be obtained with its source code via ftp from etlport.etl.go.jp (192.31.197.99). Those who use EusLisp must observe following articles and submit a copy of license agreement (doc/LICENCE) to the author.

Toshihiro MATSUI
Intelligent Systems Division,
Electrotechnical Laboratory
1-1-4 Umezono, Tsukuba, Ibaraki 3058568, JAPAN. email: matsui@etl.go.jp

Users are registered in the euslisp mailing list (euslisp@etl.go.jp), where information for Q&A, bug fix, and upgrade information is circulated. This information has been accumulated in `*eusdir*/doc/mails`.

1. The copyright of EusLisp belongs to the author (Toshihiro Matsui) and Electrotechnical Laboratory. The user must get agreement of use from the author.

2. Licensee may use EusLisp for any purpose other than military purpose.

3. EusLisp can be obtained freely from Elecrotechnical Laboratory via ftp.

4. EusLisp may be copied or sold as long as articles described here are observed. When it is sold, the seller must inform the customers that the original EusLisp is free.

5. When licensees publicize their researches or studies which used EusLisp, the use of EusLisp must be cited with appropriate bibliography.

6. Licensees may add changes to the source code of EusLisp. The resulted program is still EusLisp as long as the change does not exceed 50% of codes, and these articles must be observed for unchanged part.

7. The copyright of programs developped in EusLisp belongs to the developper. However, he cannot extend his copyright over the main body of EusLisp.

8. Neither the author nor ETL provides warranty.

1.7 Demonstrations

Demonstration programs are found in `demo` subdirectory. `cd` to `*eusdir*` and run eusx.

Robot Animation Load `demo/animdemo.l` from eusx. Smooth animation of eta3 manipulator will be shown after a precomputation of approximately 20 minutes.

Ray-Tracing If you have 8-bit pseudo color display, a ray-tracing image can be generated by loading `demo/renderdemo.l`. Make sure `geo/render.l` has already been compiled.

Edge Vision Loading `demo/edgedemo.l`, a sample gray-scale image is displayed. You give parameters for choosing the gradient operator and edge thresholds. Edges are found in a few second and overlayed on the original image.

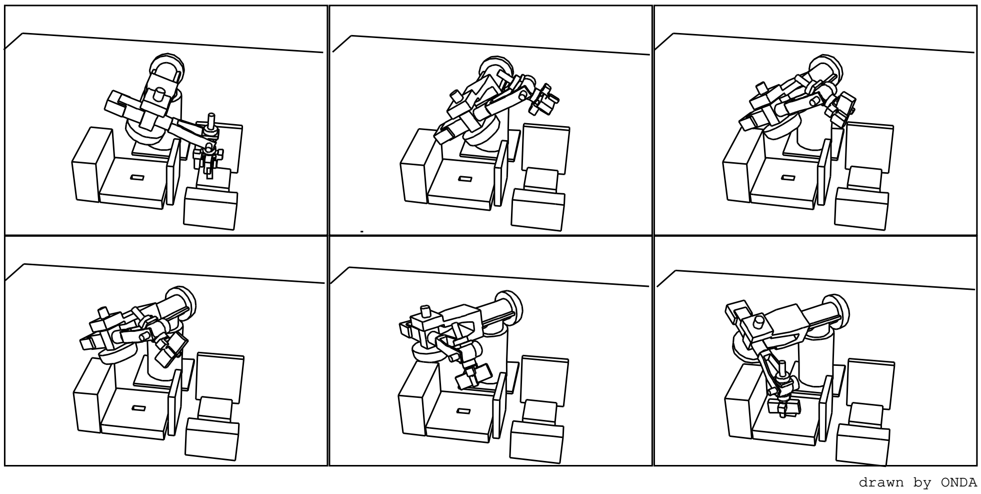


図 1: Animation of Collision Avoidance Path Planning

2 Data Types

Like other Lisps, it is data objects that are typed, not variables. Any variable can have any object as its value. Although it is possible to declare the type of object which is bound to a variable, but usually it is only advisory information to the compiler to generate faster code. Numbers are represented as immediate values in pointers and all the others are represented by objects referenced by pointers.

In the implementation of Sun4, a pointer or a number is represented by a long word as depicted in fig.**??**. Two bits at LSB of a pointer are used as tag bits to discriminate between a pointer, an integer, and a float. Since a pointer's tags are all zero and it can use all 32 bits for addressing an object, EusLisp can utilize up to 4GB of process address space.

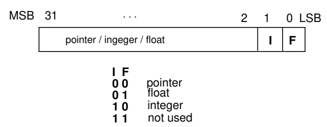


図 2: Pointer and Immediate Value

2.1 Numbers

There are two kinds of numbers, integer and float (floating-point number), both are represented with 29 bits value and 1 bit sign. Thus, integers range from -536,870,912 to 536,870,911. Floats can represent plus/minus from 4.8E-38 to 3.8E38 with the approximate accuracy of 6 digits in decimal, i.e., floating-point epsilon is approximately 1/1,000,000.

Numbers are always represented by immediate data, and not by objects. This is the only exception of EusLisp's object orientation. However, since numbers never waste heap memory, number crunching applications run efficiently without causing garbage collection.

EusLisp does not have the character type, and characters are represented by integers. In order to write a program independent of character code sets, `#\` reader dispatch macro is used. However, when the character is read, it is converted to numerical representation, and the printer does not know how to reconvert it to `#\` notation.

A number has two tag bits in a long word Figure **??**, which must be stripped off by shifting or masking when used in arithmetic computation. Note that an integer should ignore two MSB bits by arithmetic shifting, while a float should ignore two LSB bits by masking. Byte swap is also necessary for an architecture like VAX which does not use the rightmost byte as the least-significant mantissa byte.

2.2 Objects

Every data other than number is represented by an object which is allocated in heap. Each memory cell of an object has the object header and fixed number of slots for object variables. Since vectors may consist of arbitrary number of elements, they have 'size' slot immediately after the header. Fig. **??** depicts the structures of object and vector, and their header word. Only the words indicated as *slot* and *element* are accessible from users.

A header is composed of six fields. Two MSB bits, *m* and *b*, are used to indicate the side of the neighbor cell in Fibonacci-buddy memory management. There are three mark bits in the *mark* field, each of which is used by the garbage collector to identify accessible cells, by the printer to recognize circular objects in printing in `#n=` and `#n#` notations, and by **copy-object** to copy shared objects. The *elmt* field discriminates one of seven possible data types of vector elements, *pointer, bit, character, byte, integer, float* and *foreign-string*. Although *elmt* can be available in the class, it is provided in the header to make the memory manager independent of the structure of a class and to make the element accessing faster. The *bid* field represents the physical size of a memory cell. 31 different sizes up to 16 MB are represented by the five bits in this field. The lower short word (16 bits) is used for the class id. This is used to retrieve the class of an object via the system's class table. This class id can be regarded as the type tag of traditional Lisps. Currently only the lower 8 bits of the cid are used and the upper 8 bits are ignored. Therefore, the maximum number of classes is limited to 256, though this limit can be raised up to 65536 by reconfiguring the EusLisp to allocate more memory to the system's class table.

2.3 Class Hierarchy

The data structure of objects are defined by classes, and their behaviors are defined by methods in the classes. In EusLisp, a few dozens of classes have already been defined in tree structured hierarchy as depicted in fig. **??**. You can browse the real inheritance structure by the **class-hierarchy** function. The class 'object' at the leftmost is the ultimate super-class of all the classes in EusLisp. User-defined classes can inherit any of these built-in classes.

A class is defined the **defclass** macro or by the **defstruct** macro.

```
(defclass class-name &key :super          class
```

```
                              :slots          ()
                              :metaclass      metaclass
                              :element-type   t
                              :size -1
)
(defstruct struct-name slots...)
(defstruct (struct-name [struct-options ...])
              (slot-name1 [slot-option...])
              (slot-name2 [slot-option...])
              ...)
```

Methods are defined by the **defmethod** special form. **Defmethod** can appear any times for a particular class.

```
(defmethod class-name
(:method-name1 (parameter...)  .  body1)
(:method-name2 (parameter...)  .  body2)
...)
```

Field definitions for most of built-in classes are found in `*eusdir*/c/eus.h` header file. `(describe)` *class*) gives the description of all the slots in *class*, namely, super class, slot names, slot types, method list, and so on. Definitions of built-in classes follow. Note that the superclass of class **object** is NIL since it has no super class.

```
(defclass object :super NIL :slots ())

(defclass cons :super object :slots (car cdr))

(defclass propertied-object :super object
            :slots (plist))       ;property list


(defclass symbol :super propertied-object
            :slots (value          ;specially bound value
                    vtype          ;const(0),var(1),special(2)
                    function       ;global func def
                    pname          ;print name string
                    homepkg))      ;home package


(defclass foreign-pod :super symbol
            :slots (podcode          ;entry code
                    paramtypes       ;type of arguments
                    resulttype))


(defclass package :super propertied-object
```

```
            :slots (names         ;list of package name and nicknames
                    uses          ;spread use-package list
                    symvector     ;hashed obvector
                    symcount      ;number of interned symbols
                    intsymvector  ;hashed obvector of internal symbols
                    intsymcount   ;number of interned internal symbols
                    shadows       ;shadowed symbols
                    used-by))     ;packages using this package
```

(defclass **stream** :super **propertied-object**

```
            :slots (direction   ;:input or :output, nil if closed
                    buffer      ;buffer string
                    count       ;current character index
                    tail))      ;last character index
```

(defclass **file-stream** :super **stream**

```
            :slots (fd       ;file descriptor (integer)
                    fname))  ;file name str; qid for msgq
```

(defclass **broadcast-stream** :super **stream**

```
            :slots (destinations))       ;streams to which output is delivered
```

(defclass **io-stream** :super **propertied-object**

```
                    :slots (instream outstream))
```

(defclass **socket-stream** :super **io-stream**

```
                        :slots (address))            ; socket address
```

(defclass **read-table** :super **propertied-object**

```
            :slots (syntax      ; byte vector representing character types
                                ; 0:illegal, 1:white, 2:comment, 3:macro
                                ; 4:constituent, 5:single_escape
                                ; 6:multi_escape, 7:term_macro, 8:nonterm_macro
                    macro       ;character macro expansion function
                    dispatch-macro))
```

(defclass **array** :super **propertied-object**

```
            :slots (entity      ;simple vector storing array entity
                    rank        ;number of dimensions:  0-7
                    fillpointer ;pointer to push next element
                    offset      ;offset for displaced array
                    dim0,dim1,dim2,dim3,dim4,dim5,dim6)) ;dimensions
```

```
(defclass metaclass :super propertied-object
          :slots (name      ;class name symbol
                 super     ;super class
                 cix       ;class id
                 vars      ;var name vector including inherited vars
                 types     ;type vector of object variables
                 forwards  ;components to which messages are forwarded
                 methods)) ;method list


(defclass vectorclass :super metaclass
          :slots (element-type    ;vector element type 0-7
                 size))          ;vector size; 0 if unspecified


(defclass cstructclass :super vectorclass
          :slots (slotlist))    ;cstruct slot descriptors


(defclass vector :super object :slots (size))

(defclass float-vector :super vector :element-type :float)

(defclass string :super vector :element-type :char)

(defclass hash-table :super propertied-object
          :slots (lisp::key           ;hashed key vector
                 value               ; value vector
                 size                ; the size of the hash table
                 count               ; number of elements entered in the table
                 lisp::hash-function
                 lisp::test-function
                 lisp::rehash-size
                 lisp::empty lisp::deleted)


(defclass queue :super cons)

(defclass pathname :super propertied-object
          :slots (lisp::host device           ; not used
                 directory                  ; list of directories
                 name                       ; file name before the last "."
                 type                       ; type field after the last "."
                 lisp::version)             ; not used


(defclass label-reference      ;for reading #n=, #n# objects
          :super object
          :slots (label value unsolved next))
```

```
(defclass compiled-code :super object
            :slots (codevector
                    quotevector
                    type          ;0=func, 1=macro, 2=special
                    entry))       ;entry offset

(defclass closure :super compiled-code
                  :slots (env1 env2));environment

(defclass foreign-code :super compiled-code
            :slots (paramtypes       ;list of parameter types
                    resulttype))     ;function result type

(defclass load-module :super compiled-code
         :slots (symbol-table    ;hashtable of symbols defined
                 object-file     ;name of the object file loaded, needed for unloading
                 handle          ;file handle returned by ''dlopen''
```

2.4 Type Specifier

Though EusLisp does not have the **deftype** special form, type names are used in declarations and functions requesting to specify the type of results or contents, as in **coerce, map, concatenate, make-array**, etc. Usually, class names can be used as type specifiers, as in `(concatenate cons "ab" "cd") = (97 98 99 100)`, where Common Lisp uses `(quote list)` instead of `cons`.

As EusLisp does not have classes to represent numbers, types for numbers need to be given by keywords. **:integer**, **integer**, **:int**, **fixnum**, or **:fixnum** is used to represent the integer type, **:float** or **float**, the floating point number type. As the *element-type* argument of **make-array**, **:character**, **character**, **:byte**, and **byte** are recognized to make strings. Low level functions such as **defcstruct**, **sys:peek**, and **sys:poke**, also recognize **:character**, **character**, **:byte**, or **byte** for the byte access, and **:short** or **short** for short word access. In any cases, keywords are preferable to lisp package symbols with the same pname.

3 Forms and Evaluation

3.1 Atoms

A data object other than a cons is always an atom, no matter what complex structure it may have. Note that NIL, which is sometimes noted as () to represent an empty list, is also an atom. Every atom except a symbol is always evaluated to itself, although quoting is required in some other Common Lisp implementations.

3.2 Scoping

Every symbol may have associated value. A symbol is evaluated to its value determined in the current binding context. There are two kinds of variable bindings; the lexical or static binding and the special or dynamic binding. Lexically bound variables are introduced by **lambda** form or **let** and **let*** special forms unless they are declared special. Lexical binding can be nested and the only one binding which is introduced innermost level is visible, hiding outer lexical bindings and the special binding. Special variables are used in two ways: one is for global variables, and the other is for dynamically scoped local variables which are visible even at the outside of the lexical scope as long as the binding is in effect. In the latter case, special variables are needed to be declared special. The declaration is recognized not only by the compiler, but also by the interpreter. According to the Common Lisp's terms, special variables are said to have indefinite scope and dynamic extent.

Even if there exists a lexical variable in a certain scope, the same variable name can be redeclared to be special in inner scope. Function **symbol-value** can be used to retrieve the special values regardless to the lexical scopes. Note that **set** function works only for special variable, i.e. it cannot be used to change the value of lambda or let variables unless they are declared special.

```
(let ((x 1))
   (declare (special x))
   (let* ((x (+ x x)) (y x))
      (let* ((y (+ y y)) (z (+ x x)))
         (declare (special x))
         (format t "x=~S y=~s z=~s~%" x y z) ) ) )
--> x=1 y=4 z=2
```

A symbol can be declared to be a constant by **defconstant** macro. Once declared, an attempt to change the value signals an error thereafter. Moreover, such a constant symbol is inhibited to be used as the name of a variable even for a local variable. NIL and T are examples of such constants. Symbols in the keyword package are always declared to be constants when they are created. In contrast, **defvar** and **defparameter** macro declare symbols to be special variables. **defvar** initializes the value only if the symbol is unbound, and does nothing when it already has a value assigned, while **defparameter** always resets the value.

When a symbol is referenced and there is no lexical binding for the symbol, its special value is retrieved. However, if no value has been assigned to its special value yet, unbound variable error is signaled.

3.3 Generalized Variables

Generally, any values or attributes are represented in slots of objects (or in stack frames). To retrieve and alter the value of a slot, two primitive operations, *access* and *update*, must be provided. Instead of defining two distinct primitives for every slot of objects, EusLisp, like Common Lisp, provides uniform update operations based on the generalized variable concept. In this concept, a common form is recognized either as a value access form or as a slot location specifier. Thus, you only need to remember accessing form for each slot and update is achieved by **setf** macro used in conjunction with the access form. For example, `(car x)` can be used to replace the value in the car slot of `x` when used with **setf** as in `(setf (car '(a b) 'c)`, as well as to take the car value out of the list.

This method is also applicable to all the user defined objects. When a class or a structure is defined, the access and update forms for each slot are automatically defined. Each of those forms is defined as a macro whose name is the concatenation of the class name and slot name. For example, car of a cons can be addressed by `(cons-car '(a b c))`.

```
(defclass person :super object :slots (name age))
(defclass programmer :super person :slots (language machine))
(setq x (instantiate programmer))
(setf (programmer-name x) "MATSUI"
      (person-age x) 30)
(incf (programmer-age x))
(programmer-age x)   --> 31
(setf (programmer-language x) 'EUSLISP
      (programmer-machine x) 'SUN4)
```

Array elements can be accessed in the same manner.

```
(setq a (make-array '(3 3) :element-type :float))
(setf (aref a 0 0) 1.0 (aref a 1 1) 1.0 (aref a 2 2) 1.0)
a --> #2f((1.0 0.0 0.0) (0.0 1.0 0.0) (0.0 0.0 1.0))

(setq b (instantiate bit-vector 10))  --> #*0000000000
(setf (bit b 5) 1)
b --> #*0000010000
```

In order to define special setf methods for particular objects, **defsetf** macro is provided.

```
(defsetf symbol-value set)
(defsetf get (sym prop) (val) `(putprop ,sym ,val ,prop))
```

3.4 Special Forms

All the special forms are listed in Table **??**. **macrolet, compiler-let,** and **progv** have not been implemented. Special forms are essential language constructs for the management of evaluation contexts and control flows. The interpreter and compiler have special knowledge to process each of these constructs properly, while the application method is uniform for all functions. Users cannot add their own special form definition.

3.5 Macros

Macro is a convenient method to expand language constructs. When a macro is called, arguments are passed to the macro body, which is a macro expansion function, without being evaluated. Then, the macro

and	flet	quote
block	function	return-from
catch	go	setq
cond	if	tagbody
declare	labels	the
defmacro	let	throw
defmethod	let*	unwind-protect
defun	progn	while
eval-when	or	

表 2: EusLisp's special forms

expansion function expands the arguments, and returns the new form. The resulted form is then evaluated again outside the macro. It is an error to apply a macro or special form to a list of arguments. **Macroexpand** function can be used for the explicit macro expansion.

Though macro runs slowly when interpreted, it speeds up compiled code execution, because macro expansion is taken at compile-time only once and no overhead is left to run-time. Note that explicit call to eval or apply in the macro function may produce different results between interpreted execution and the compiled execution.

3.6 Functions

A function is expressed by a lambda form which is merely a list whose first element is **lambda**. If a lambda form is defined for a symbol using **defun**, it can be referred as a global function name. Lambda form takes following syntax.

```
(lambda ({var}*
          [&optional {var | (var [initform])}*]
          [&rest form]
          [&key {var | (var [initform]) | ((:keyword var) [initform])}*
                [&allow-other-keys]]
          [&aux {var | (var [initform])}*])
      {declaration}*
      {form}*)
```

There is no function type such as EXPR, LEXPR, FEXPR, etc.: arguments to a function are always evaluated before its application, and the number of acceptable arguments is determined by lambda-list. Lambda-list specifies the sequence of parameters to the lambda form. Each of **&optional, &rest, &key** and **&aux** has special meaning in lambda-lists, and these symbols cannot be used as variable names. Supplied-p variables for &optional or &key parameters are not supported.

Since a lambda form is indistinguishable from normal list data, **function** special form must be used to inform the interpreter and compiler the form is intended to be a function. [1] **Function** is also important to freeze the environment onto the function, so that all the lexical variables can be accessible in the function

[1] In CLtL-2 a quoted lambda form is no longer a function. Application of such a form is an error.

even the function is passed to another function of different lexical scope. The following program does not work either interpretedly nor after compiled, since `sum` from the `let` is invisible inside lambda form.

```
(let ((x '(1 2 3)) (sum 0))
  (mapc '(lambda (x) (setq sum (+ sum x))) x))
```

To get the expected result, it should be written as follows:

```
(let ((x '(1 2 3)) (sum 0))
   (mapc #'(lambda (x) (setq sum (+ sum x))) x ))
```

#' is the abbreviated notation of **function**, i.e. `#'(lambda (x) x)` is equivalent to `(function (lambda (x) x))`. Here is another example of what is called a funarg problem:

```
(defun mapvector (f v)
    (do ((i 0 (1+ i)))
       ((>= i (length v)))
       (funcall f (aref v i))))
(defun vector-sum (v)
    (let ((i 0))
       (mapvector #'(lambda (x) (setq i (+ i x))) v)
       i))
(vector-sum #(1 2 3 4)) --> 10
```

EusLisp's closure cannot have indefinite extent: i.e. a closure can only survive as long as its outer extent is in effect. This means that a closure cannot be used for programming of "generators". The following program does not work.

```
(proclaim '(special gen))
(let ((index 0))
   (setq gen #'(lambda () (setq index (1+ index)))))
(funcall gen)
```

However, the same purpose is accomplished by object oriented programming, because an object can hold its own static variables:

```
(defclass generator object (index))
(defmethod generator
 (:next () (setq index (1+ index)))
 (:init (&optional (start 0)) (setq index start) self))
(defvar gen (instance generator :init 0))
(send gen :next)
```

4 Control Structures

4.1 Conditionals

Although **and, or** and **cond** are advised to be macros by Common Lisp, they are implemented as special forms in EusLisp to improve the interpreting performance.

and *&rest forms* [special]

*Form*s are evaluated from left to right until NIL appears. If all forms are evaluated to non-NIL, the last value is returned.

or *&rest forms* [special]

*Form*s are evaluated from left to right until non-NIL appears, and the value is returned. If all forms are evaluated to NIL, NIL is returned.

if *test then &optional else* [special]

if can only have single *then* and *else* forms. To allow multiple *then* or *else* forms, they must be grouped by **progn**.

when *test &rest forms* [macro]

Unlike **if**, **when** and **unless** allow you to write multiple *forms* which are executed when *test* holds (**when**) or does not *unless*. On the other hand, these macros cannot have the *else* forms.

unless *test &rest forms* [macro]

is equivalent to `(when (not test) . forms)`.

cond *&rest (test &rest forms)* [special]

Arbitrary number of cond-clauses can follow **cond**. In each clause, the first form, that is *test*, is evaluated. If it is non-nil, the rest of the forms in that clause are evaluated sequentially, and the last value is returned. If no forms are given after the *test*, the value of the *test* is returned. When the *test* fails, next clause is tried until a *test* which is evaluated to non-nil is found or all clauses are exhausted. In the latter case, **cond** returns NIL.

case *key &rest (label &rest forms)* [macro]

For the clause whose *label* matches with *key*, *form*s are evaluated and the last value is returned. Equality between *key* and *label* is tested with **eq** or **memq**, not with **equal**.

4.2 Sequencing and Lets

prog1 *form1 &rest forms* [function]

form1 and *forms* are evaluated sequentially, and the value returned by *form1* is returned as the value of **prog1**.

progn *&rest forms* [special]

*Form*s are evaluated sequentially, and the value of the rightmost form is returned. **Progn** is a special form because it has a special meaning when it appeared at top level in a file. When such a form is compiled, all inner forms are regarded as they appear at top level. This is useful for a macro which expands to a series of **defun**s or **defmethod**s, which must appear at top level.

setf *&rest forms* [macro]

for each form in *forms*, assigns the second element to the generalized-variable signilized by the first element.

let *(&rest (var &optional value)) &rest forms* [special]

introduces local variables. All *value*s are evaluated and assigned to *var*s in parallel, i.e., `(let ((a 1)) (let ((a (1+ a)) (b a)) (list a b)))` produces (2 1). The first statements of *forms* can be declarations.

let* *(&rest (var &optional value)) &rest forms* [special]

introduces local variables. All *value*s are evaluated sequentially, and assigned to *var*s i.e., `(let ((a 1)) (let* ((a (1+ a)) (b a)) (list a b)))` produces (2 2).

4.3 Local Functions

flet *(&rest (fname lambda-list &rest body)) &rest forms* [special]

defines local functions.

labels *(&rest (fname lambda-list &rest body)) &rest forms* [special]

defines locally scoped functions. The difference between *flet* and *labels* is, the local functions defined by *flet* cannot reference each other or recursively, whereas *labels* allows such mutual references.

4.4 Blocks and Exits

block *tag &rest forms* [special]

makes a lexical block from which you can exit by **return-from**. *Tag* is lexically scoped and is not evaluated.

return-from *tag &optional value* [special]

exits the block labeled by *tag*. **return-from** can be used to exit from a function or a method which automatically establishes block labeled by its function or method name surrounding the entire body.

return *&optional value* [macro]

`(return x)` is equivalent to `(return-from nil x)`. This is convenient to use in conjunction with **loop, while, do, dolist,** and **dotimes** which implicitly establish blocks labeled NIL.

catch *tag &rest forms* [special]

establishes a dynamic block from which you can exit and return a value by **throw**. *Tag* is evaluated. The list of all visible catch tags can be obtained by `sys:list-all-catchers`.

throw *tag value* [special]

exits and returns *value* from a catch block. *tag* and *value* are evaluated.

unwind-protect *protected-form &rest cleanup-forms* [special]

After the evaluation of *protected-form* finishes, *cleanup-form* is evaluated. You may make a block or a catch block outside the `unwind-protect`. Even **return-from** or **throw** is executed in *protected-form* to escape from such blocks, *cleanup-forms* are assured to be evaluated. Also, if you had an error while executing *protected-form*, *cleanup-form* would always be executed by *reset*.

4.5 Iteration

while *test &rest forms* [special]

While *test* is evaluated to non-nil, *form*s are evaluated repeatedly. **While** special form automatically establishes a block by name of nil around *form*s, and **return** can be used to exit from the loop. To jump to next iteration, you can use following syntax with **tagbody** and **go** described below:

```
(setq cnt 0)
(while
  (< cnt 10)
  (tagbody while-top
    (incf cnt)
    (when (eq (mod cnt 3) 0)
      (go while-top))  ;; jump to next iteraction
    (print cnt)
    )) ;; 1, 2, 4, 5, 7, 8, 10
```

tagbody *&rest tag-or-statement* [special]

tags can be used as labels for **go**. You can use **go** only in **tagbody**.

go *tag* [special]

transfers control to the form just after *tag* which appears in a lexically scoped **tagbody**. **Go** to the tag in a different **tagbody** across the lexical scope is inhibited.

prog *varlist &rest tag-or-statement* [macro]

prog is a macro, which expands as follows:

```
(block nil (let varlist (tagbody tag-or-statement)))
```

do *(&rest (var &optional optional init next)) (endtest &optional result) &rest forms* [macro]

*var*s are local variables. To each *var*, *init* is evaluated in parallel and assigned. Next, *endtest* is evaluated and if it is true, **do** returns *result* (defaulted to NIL). If *endtest* returns NIL, each *form* is evaluated sequentially. After the evaluation of forms, *next* is evaluated and the value is reassigned to each *var*, and the next iteration starts.

do* *(&rest (var &optional optional init next)) (endtest &optional result) &rest forms* [macro]

do* is same as **do** except that the evaluation of *init* and *next*, and their assignment to *var* occur sequentially.

dotimes *(var count &optional result) &rest forms* [macro]

evaluates *forms count* times. *count* is evaluated only once. In each evaluation, *var* increments from integer zero to *count* minus one.

dolist *(var list &optional result) &rest forms* [macro]

Each element of *list* is sequentially bound to *var*, and *forms* are evaluated for each binding. **Dolist** runs faster than other iteration constructs such as **mapcar** and recursive functions, since **dolist** does not have to create a function closure or to apply it, and no new parameter binding is needed.

until *condition &rest forms* [macro]

evaluates forms until *condition* holds.

loop *&rest forms* [macro]

evaluates *forms* forever. To terminate execution, **return-from, throw** or **go** needed to be evaluated in *forms*.

4.6 Predicates

Typep and **subtypep** of Common Lisp are not provided, and should be simulated by **subclassp** and **derivedp**.

eq *obj1 obj2* [function]

returns T if *obj1* and *obj2* are pointers to the same object or the same numbers. Examples: `(eq 'a 'a)` is T, `(eq 1 1)` is T, `(eq 1. 1.0)` is nil, `(eq "a" "a")` is nil.

eql *obj1 obj2* [function]

Eq and **eql** are identical since all the numbers in EusLisp are represented as immediate values.

equal *obj1 obj2* [function]

Checks the equality of any structured objects, such as strings, vectors or matrices, as long as they do not have recursive references. If there is recursive reference in *obj1* or *obj2*, **equal** loops infinitely.

superequal *obj1 obj2* [function]

Slow but robust **equal**, since **superequal** checks circular reference.

null *object* [function]

T if *object* is nil. Equivalent to (eq *object* nil).

not *object* [function]

not is identical to **null**.

atom *object* [function]

returns NIL only if object is a cons. `(atom nil) = (atom '()) = T)`. Note that **atom** returns T for vectors, strings, read-table, hash-table, etc., no matter what complex objects they are.

every *pred &rest args* [function]

returns T if all *args* return T for *pred*. **Every** is used to test whether *pred* holds for every *args*.

some *pred &rest args* [function]

returns T if at least one of *args* return T for *pred*. **Some** is used to test whether *pred* holds for any of *args*.

functionp *object* [function]

T if *object* is a function object that can be given to **apply** and **funcall**. Note that macros cannot be *apply*'ed or *funcall*'ed. **Functionp** returns T, if *object* is either a compiled-code with type=0, a symbol that has function definition, a lambda-form, or a lambda-closure. Examples: (functionp 'car) = T, (functionp 'do) = NIL

compiled-function-p *object* [function]

T if *object* is an instance of compiled-code. In order to know the compiled-code is a function or a macro, send `:type` message to the object, and `function` or `macro` is returned.

5 Object Oriented Programming

The structures and behaviors of objects are described in classes, which are defined by **defclass** macro and **defmethod** special form. **defclass** defines the name of the class, its super class, and slot variable names, optionally with their types and message forwarding. **defmethod** defines methods which will invoked when corresponding messages are sent. Class definition is assigned to the symbol's special value. You may think of **class** as the counter part of Common Lisp's **structure**. Slot accessing functions and **setf** methods are automatically defined for each slot by **defclass**.

Most classes are instantiated from the built-in class **metaclass**. Class **vector-class**, which is a subclass of **metaclass**, is a metaclass for vectors. If you need to use class-variables and class-methods, you may make your own metaclass by subclassing **metaclass**, and the metaclass name should be given to **defclass** with `:metaclass` keyword.

Vectors are different from other record-like objects because an instance of the vector can have arbitrary number of elements, while record-like objects have fixed number of slots. EusLisp's object is either a record-like object or a vector, not both at the same time.

Vectors whose elements are typed or the number of elements are unchangeable can also be defined by **defclass**. In the following example, class `intvec5` which has five integer elements is defined. Automatic type check and conversion are performed when the elements are accessed by the interpreter. When compiled with proper declaration, faster accessing code is produced.

```
(defclass intvec5 :super vector :element-type :integer :size 5)
(setq x (instantiate intvec5))  --> #i(0 0 0 0 0)
```

When a message is sent to an object, the corresponding method is searched for, first in its class, and next in its superclasses toward **object**, until all superclasses are exhausted. If the method is undefined, forward list is searched. This forwarding mechanism is introduced to simulate multiple inheritance. If the search fails again, a method named `:nomethod` is searched, and the method is invoked with a list of all the arguments. In the following example, the messages `:telephone` and `:mail` are sent to `secretary` slot object which is typed `person`, and `:go-home` message is sent to `chauffeur` slot.

```
(defclass president :super object
                    :slots ((name :type string)
                            (age  :type :integer)
                            (secretary  :type person
                                        :forward (:telephone :mail))
                            (chauffeur  :forward (:go-home))))
```

In a method, two more local variables, **class** and **self**, become accessible. You should not change either of these variables. If you do that, the ones supplied by the system are hidden, and **send-super** and **send self** are confused.

5.1 Classes and Methods

defclass [macro]

classname &key (super object)
slots ; (var &optional type &rest forward selectors)*

(metaclass metaclass)
(element-type t)
(size -1)

creates or redefine a class. When a class is redefined to have different superclass or slot variables, old objects instantiated from the previous class definition will behave unexpectedly, since method definitions assume the new slots disposition.

defmethod *classname &rest (selector lambda-list &rest body)* [special]

defines one or more methods of *classname*. Each *selector* must be a keyword symbol.

defclassmethod *classname &rest (selector lambda-list &rest body)* [macro]

classp *object* [function]

T if *object* is a class object, that is, an instance of class **metaclass** or its subclasses.

subclassp *class super* [function]

Checks *class* is a subclass of *super*.

vector-class-p *x* [function]

T if *x* is an instance of **vector-class**.

delete-method *class method-name* [function]

The method definition is removed from the specified class.

find-method *object selector* [function]

tries to find a method specified by *selector* in the class of *object* and in its superclass. This is used to know whether *object* can respond to *selector*.

class-hierarchy *class* [function]

prints inheritance hierarchy below *class*.

system:list-all-classes [function]

lists up all the classes defined so far.

system::method-cache *&optional flag* [function]

Interrogates the hit ratio of the method cache, and returns a list of two numbers, hit and miss. If *flag* is NIL, method caching is disabled. If non-nil flag is given, method cache is purged and caching is enabled.

5.2 Message Sending

send *object selector &rest args* [function]

send a message consisting of *selector* and *args* to *object*. *object* can be anything but number. *selector* must be evaluated to be a keyword.

send-message *target search selector &rest args* [function]

Low level primitive to implement **send-super**.

send* *object selector &rest msg-list* [macro]

send* applies **send-message** to a list of arguments. The relation between **send** and **send*** is like the one between **funcall** and **apply**, or **list** and **list***.

send-all *receivers selector &rest mesg* [function]

sends the same message to all the receivers, and collects the result in a list.

send-super *selector &rest msgs* [macro]

sends *msgs* to self, but begins method searching at the superclass of the class where the method currently being executed is defined. It is an error to *send-super* outside a method (i.e. in a function).

send-super* *selector &rest msg-list* [macro]

send-super* is apply version of send-super.

5.3 Instance Management

instantiate *class &optional size* [function]

the lowest primitive to create a new object from a class. If the class is a vector-class, *size* should be supplied.

instance *class &rest message* [macro]

An instance is created, and the message is sent to it.

make-instance *class &rest var-val-pairs* [function]

creates an instance of *class* and sets its slot variables according to *var-val-pairs*. For example, `(make-instance cons :car 1 :cdr 2)` is equivalent to `(cons 1 2)`.

copy-object *object* [function]

copy-object function is used to copy objects keeping the referencing topologies even they have recursive references. **Copy-object** copies any objects accessible from *object* except symbols and packages, which are untouched to keep the uniqueness of symbols. **copy-object** traverses all the references in an object twice: once to create new objects and to mark original objects that they have already copied, and again to remove marks. This two-step process makes copy-object work slower than copy-seq. If what you wish to copy is definitely a sequence, use of **copy-seq** or **copy-tree** is recommended.

become *object class* [function]

changes the class of *object* to *class*. The slot structure of both the old class and the new class must be consistent. Usually, this can be safely used only for changing class between binary vectors, for example from an integer-vector to a bit-vector.

replace-object *dest src* [function]

dest must be an instance of the subclass of src.

class *object* [function]

returns the class object of *object*. To get the name of the class, send :name message to the class object.

derivedp *object class* [function]

derivedp checks if an object is instantiated from *class* or *class*'s subclasses. **subclassp** and **derivedp** functions do not search in class hierarchy: type check always finishes within a constant time.

slot *object class idex-or-name* [function]

Returns the named or indexed slot value.

setslot *object class index-or-name value* [function]

Setslot is a internal function and users should not use it. Use, instead, combination of **setf** and **slot**.

5.4 Basic Classes

object [Class]

:super
:slots

Object is the most basic class that is located at the top of class hierarchy. Since it defines no slot variables, it is no use to make an instance of **object**.

:prin1 *&optional stream &rest mesg* [method]

prints the object in the standard re-readable object format, that is, the class name and the address, enclosed by angle brackets and preceded by a pound sign. Any subclasses of **object** can use this method to print itself with more comprehensive information by using **send-super** macro specifying *mesg* string. An object is re-readable if it begins with #<, followed by its class name, correct address, any lisp-readable information, and >. Since every data object except numbers inherits **object**, you can get print forms in this notation, even for symbols or strings. Specifying this notation, you can catch data objects that you forgot to **setq** to a symbol, as long as there happened no garbage collection after it is printed.

:slots [method]

returns the list of variable-name and value pair of all the slots of the object. You can get the value of a specific slot by applying **assoc** to this list, although you cannot alter them.

:methods *&optional subname* [method]

returns a list of all methods callable by this object. If *subname* is given, returns only methods with names that include *subname*.

:get-val *variable-name* [method]

returns the value of the slot designated by *variable-name*. If the object does not have the *variable-name* slot, an error is reported.

:set-val *variable-name value* [method]

sets *value* in the *variable-name* slot of this object. If the object does not have the *variable-name* slot, an error is reported.

propertied-object [Class]

:super **object**
:slots plist

defines objects that have property list. Unlike other Common Lisp, EusLisp allows any objects that inherit propertied-object to have property lists, even if they are not symbols.

:plist *&optional plist* [method]

if *plist* is specified, it is set to the plist slot of this object. Previous plist, if there had been one, is lost. Legal plist should be of the form of `((indicator1 . value1) (indicator2 . value2) ...)`. Each `indicator` can be any lisp form that are tested its equality with the **eq** function. When a symbol is used for an indicator, use of keyword is recommended to ensure the equality check will be performed interpackage-widely. **:plist** returns the current plist.

:get *indicator* [method]

returns the value associated with *indicator* in the property list. `(send x :get :y) == (cdr (assoc :y (send x :plist)))`.

:put *indicator value* [method]

associates *value* to *indicator* in the plist.

:remprop *indicator* [method]

removes *indicator* and value pair from the plist. Further attempt to *:get* the value returns nil.

:name *&optional name* [method]

defines and retrieves the `:name` property in the plist. This property is used for printing.

:prin1 *&optional stream &rest mesg* [method]

prints the object in the re-readable form. If the object has **:name** property, it is printed after the address of the object.

metaclass [Class]

:super	**propertied-object**
:slots	name super cix vars types forwards methods

Metaclass defines classes. Classes that have own class variables should be defined with **metaclass** as their superclass.

:new [method]

creates an instance of this class and returns it after filling all the slots with NIL.

:super [method]

returns the super class object of this class. You cannot alter superclass once defclassed.

:methods [method]

returns a list of all the methods defined in this class. The list is composed of lists each of which describes the name of the method, parameters, and body.

:method *name* [method]

returns the method definition associated with *name*. If not found, NIL is returned.

:method-names *subname* [method]

returns a list of all the method names each of which contains *subname* in its method name. Methods are searched only in this class.

:all-methods [method]

returns a list of all methods that are defined in this class and its all the super classes. In other words, an instance of this class can execute each of these methods.

:all-method-names *subname* [method] returns a list of all the method names each of which matches with *subname*. The search is made from this class up to **object**.

:slots [method]

returns the slot-name vector.

:name [method]

returns the name symbol of this class.

:cid [method]

returns an integer that is assigned to every instance of this class to identify its class. This is an index to the system-internal class table, and is changed when a new subclass is defined under this class.

:subclasses [method]

returns a list of the direct subclass of this class.

:hierarchy [method]

returns a list of all the subclasses defined under this class. You can also call the **class-hierarchy** function to get a comprehensive listing of all the class hierarchy.

find-method *object selector* [function]

searches for the method identified by *selector* in *object*'s class and its super classes. This function is useful when object's class is uncertain and you want to know whether the object can handle the message without causing nomethod error.

6 Arithmetic Functions

6.1 Arithmetic Constants

most-positive-fixnum [constant]

#x1fffffff=536,870,911

most-negative-fixnum [constant]

-#x20000000= -536,870,912

short-float-epsilon [constant]

A floating point number on machines with IEEE floating-point format is represented by 21 bit mantissa with 1 bit sign and 7 bit exponent with 1 bit sign. Therefore, floating point epsilon is $2^{-21} = 4.768368 \times 10^{-7}$.

single-float-epsilon [constant]

same as **short-float-epsilon**, 2^{-21}.

long-float-epsilon [constant]

same as **short-float-epsilon** since there is no double or long float. 2^{-21}.

pi [constant]

π, actually 3.14159203, not 3.14159265.

2pi [constant]

$2 \times \pi$

pi/2 [constant]

$\pi/2$

-pi [constant]

-3.14159203

-2pi [constant]

$-2 \times \pi$

-pi/2 [constant]

$\pi/2$

6.2 Arithmetic Predicates

numberp *object* [function]

T if *object* is number, namely integer or float. Note that characters are also represented by numbers.

integerp *object* [function]

T if *object* is an integer number. A float can be converted to an integer by **round, trunc** and **ceiling** functions.

floatp *object* [function]

T if *object* is a floating-point number. An integer can be converted to a float by the **float** function.

zerop *number* [function]

T if the number is integer zero or float 0.0.

plusp *number* [function]

equivalent to (> number 0).

minusp *number* [function]

equivalent to (< number 0).

oddp *integer* [function]

The argument must be an integer. T if *integer* is odd.

evenp *integer* [function]

The argument must be an integer. T if *integer* is an even number.

/= *num1 num2 &rest more-numbers* [function]

Both *num1*, *num2* and all elements of *more-numbers* must be numbers. T if no two of its arguments are numerically equal, NIL otherwise.

= *num1 num2 &rest more-numbers* [function]

Both *n1* and *n2* and all elements of *more-numbers* must be numbers. T if *n1*, *n2* and all elements of *more-numbers* are the same in value, NIL otherwise.

> *num1 num2 &rest more-numbers* [function]

Both *n1* and *n2* and all elements of *more-numbers* must be numbers. T if *n1*, *n2* and all elements of *more-numbers* are in monotonically decreasing order, NIL otherwise. For numerical comparisons with tolerance, use functions prefixed by **eps** as described in the section **??**.

< *num1 num2 &rest more-numbers* [function]

Both *n1* and *n2* and all elements of *more-numbers* must be numbers. T if *n1*, *n2* and all elements of *more-numbers* are in monotonically increasing order, NIL otherwise. For numerical comparisons with tolerance, use functions prefixed by **eps** as described in the section **??**.

>= *num1 num2 &rest more-numbers* [function]

Both *n1* and *n2* and all elements of *more-numbers* must be numbers. T if *n1*, *n2* and all elements of *more-numbers* are in monotonically nonincreasing order, NIL otherwise. For numerical comparisons with tolerance, use functions prefixed by **eps** as described in the section **??**.

<= *num1 num2 &rest more-numbers* [function]

Both *n1* and *n2* and all elements of *more-numbers* must be numbers. T if *n1*, *n2* and all elements of *more-numbers* are in monotonically nondecreasing order, NIL otherwise. For numerical comparisons with tolerance, use functions prefixed by **eps** as described in the section **??**.

6.3 Integer and Bit-Wise Operations

Following functions request arguments to be integers.

mod *dividend divisor* [function]

returns remainder when *dividend* is divided by *divisor*. `(mod 6 5)=1`, `(mod -6 5)=-1`, `(mod 6 -5)=1`, `(mod -6 -5)=-1`.

1- *number* [function]

$number - 1$ is returned.

1+ *number* [function]

$number + 1$ is returned.

logand *&rest integers* [function]

bitwise-and of *integers*.

logior *&rest integers* [function]

bitwise-inclusive-or of *integers*.

logxor *&rest integers* [function]

bitwise-exclusive-or of *integers*.

logeqv *&rest integers* [function]

logeqv is equivalent to (lognot (logxor ...)).

lognand *&rest integers* [function]

bitwise-nand of *integers*.

lognor *&rest integers* [function]

bitwise-nor of *integers*.

lognot *integer* [function]

bit reverse of *integer*.

logtest *integer1 integer2* [function]

T if (logand *integer1 integer2*) is not zero.

logbitp *index integer* [function]

T if *index*th bit of *integer* (counted from the LSB) is 1, otherwise NIL.

ash *integer count* [function]

Arithmetic Shift Left. If *count* is positive, shift direction is left, and if *count* is negative, *integer* is shifted to right by abs(*count*) bits.

ldb *target position &optional (width 8)* [function]

LoaD Byte. Byte specifier for **ldb** and **dpb** does not exist in EusLisp. Use a pair of integers instead. The field of *width* bits at *position* within *target* is extracted. For example, `(ldb #x1234 4 4)` is 3.

dpb *value target position &optional (width 8)* [function]

DePosit byte. *Width* bits of *value* is put in *target* at *position*th bits from LSB.

6.4 Generic Number Functions

+ *&rest numbers* [function]

returns the sum of *numbers*.

- *num &rest more-numbers* [function]

If *more-numbers* are given, they are subtracted from *num*. Otherwise, *num* is negated.

***** *&rest numbers* [function]

returns the product of *numbers*.

/ *num &rest more-numbers* [function]

num is divided by *more-numbers*. If only one argument is given, 1.0 is divided by *num*. The result is an integer if all the arguments are integers, and a float if at least one of the arguments is a float.

abs *number* [function]

returns absolute number.

round *number* [function]

rounds to the nearest integer. `(round 1.5)=2, (round -1.5)=-2`.

floor *number* [function]

rounds to the nearest smaller integer. `(floor 1.5)=1, (floor -1.5)=-2`.

ceiling *number* [function]

rounds to the nearest larger integer. `(ceiling 1.5)=2, (ceiling -1.5)=-1`.

truncate *number* [function]

rounds to the absolutely smaller and nearest integer. `(truncate 1.5)=1, (truncate -1.5)=-1`.

float *number* [function]

returns floating-point representation of *number*.

max *num &rest more-numbers* [function]

finds the maximum value among *num* and *more-numbers*.

min *num &rest more-numbers* [function]

finds the minimum value among *num* and *more-numbers*.

make-random-state *&optional (state *random-state*)* [function]

creates a fresh object of type random-state suitable for use as the value of *random-state*. If *state* is a random state object, the new-state is a copy of that object. If state is NIL, the new-state is a copy of the current **random-state**. If state is T, the new-state is a fresh random state object that has been randomly initialized.

random *range &optional (state *random-state*)* [function]

Returns a random number between 0 or 0.0 and *range*. If *range* is an integer, the result is truncated to an integer. Otherwise, a floating value is returned. Optional *state* can be specified to get predictable random number sequence. There is no special data type for random-state, and it is represented with an integer vector of two elements.

incf *variable &optional (increment 1)* [macro]

variable is a generalized variable. The value of *variable* is incremented by *increment*, and it is set back to *variable*.

decf *variable &optional (decrement 1)* [macro]

variable is a generalized variable. The value of *variable* is decremented by *decrement*, and it is set back to *variable*.

reduce *func seq* [function]

combines all the elements in *seq* using the binary operator *func*. For an example, `(reduce #'expt`

'(2 3 4)) = (expt (expt 2 3) 4)=4096.

rad2deg *radian* [function]

Radian value is converted to degree notation. #R does the same thing at read time. Note that the official representation of angle in EusLisp is radian and every EusLisp function that accepts angle argument requests it to be represented by radian.

deg2rad *degree* [function]

Conversion from degree to radian. Also accomplished by #D reader's dispatch macros.

6.5 Trigonometric and Related Functions

sin *theta* [function]

theta is a float representing angle by radian. returns $sin(theta)$.

cos *theta* [function]

theta is a float representing angle by radian. returns $cos(theta)$.

tan *theta* [function]

theta is a float representing angle by radian. returns $tan(theta)$.

sinh *x* [function]

hyperbolic sine, that is, $\frac{e^x - e^{-x}}{2}$.

cosh *x* [function]

hyperbolic cosine, that is, $\frac{e^x + e^{-x}}{2}$.

tanh *x* [function]

hyperbolic tangent, that is, $\frac{e^x - e^{-x}}{e^x + e^{-x}}$.

asin *x* [function]

arc sine.

acos *x* [function]

arc cosine.

atan *y &optional x* [function]

When **atan** is called with one argument, its arctangent is calculated. When called with two arguments, $atan(y/x)$ is returned.

asinh *x* [function]

hyperbolic arc sine.

acosh *x* [function]

hyperbolic arc cosine.

atanh *x* [function]

hyperbolic arc tangent.

sqrt *number* [function]

returns square root of *number*.

log *number &optional base* [function]

returns natural logarithm of *number*. If *base* is provided, return the logarithm of *number* in the given base instead.

exp *x* [function]

returns exponential, e^x.

expt *a x* [function]

returns a^x.

6.6 Extended Numbers

ratio [Class]

:super	**extended-number**
:slots	(numerator denominator)

Describes rational numbers.

:init *num denom* [method]

initializes a rational number instance with numerator *num* and denominator *denom*.

complex [Class]

:super	**extended-number**
:slots	(real imaginary)

Describes complex numbers.

:init *re im* [method]

initializes a complex number instance with real part *re* and imaginary part *im*.

7 Symbols and Packages

7.1 Symbols

A symbols is assured to be unique if it is interned in a package. The uniqueness is tested by symbol's print-names. There are no duplicated symbols in a package which have the same print-name as other symbols in the package. When EusLisp is running, there always is a special package called the current package, which is referred by **lisp:*package***. When a symbol without a package name is read by the reader, the current package is searched for to locate the symbol with the same print-name. If no such symbol is found, search is continued in the packages listed in the package use list of the current package. If still no such symbol is found, a new symbol object with the designated print-name is created and is interned in the current package. The package can be specified by prefixing the package name followd by a colon(:). If a symbol name is preceeded by a package name, the search begins in the designated package.

Every symbol may have at most one home package. If a symbol has no such home package, it is said to be an uninterned symbol. Uninterned symbols can be created by the **gensym** or **make-symbol** function, and they are prefixed by "#:" when printed. Since these symbols are not interned, two such symbols with the same print-name are not guaranteed to be equal.

Usually, when the lisp reader encounters a symbol, the reader converts the print-name string of the symbol to uppper case. Thus, for example, if you input `(symbol-name 'car)`, EusLisp responds `"CAR"` instead of `"car"`. Note that `(make-symbol "car")` returns |`car`| instead of `car` or `CAR`. If you want the reader to make symbols constituted by lower case letters, use reader's escapes, \ and |...|.

symbolp *object* [function]

returns T if *object* is an instance of CLASS symbol or its subclasses.

symbol-value *symbol* [function]

gets *symbol*'s special value. Lexical (local) variables' values cannot be retrieved by this function.

symbol-function *symbol* [function]

gets *symbol*'s global function definition. Lexical (local) function cannot be taken by this function.

symbol-package *sym* [function]

returns the package where *sym* is interned.

symbol-name *sym* [function]

returns *sym*'s print-name. Note that **symbol-name** does not copy the pname string, whereas **string** does. Thus, if you change the string returned by **symbol-name**, the symbol becomes inaccessible through normal intern procedure.

symbol-plist *sym* [function]

Returns *sym*'s property list (plist). EusLisp's plist takes the same form as an association list, which consists of dotted pairs of an attribute name and its value. This is incompatible with Common Lisp definition which requests a plist to have linear lists of attribute name and value. In EusLisp, plist is not the unique facility of symbols. Any objects instantiated from a class that inherits **propertied-object** can have property lists. To set and retrieve these plists in propertied-objects, **propertied-object-plist** macro should be used instead of **symbol-plist**. However, **get** and **putprop** work for either object.

boundp *symbol* [function]

Checks if *symbol* has a globally bound value. Note that symbols used for local and object variables always have bound value and **boundp** cannot test the bound state of these local variables.

fboundp *symbol* [function]

Checks if *symbol* has a globally bound function definition.

makunbound *symbol* [function]

symbol is forced to be unbound (to have no special value). Note that lexical (local) variables always have values assigned and cannot be *makunbound*ed.

get *sym attribute* [function]

retrieves *sym*'s value associated with *attribute* in its plist. `= (cdr (assoc` *`attribute`* `(symbol-plist` *`sym`*`)))`

putprop *sym val attribute* [function]

Putprop should be replaced with the combination of **setf** and **get**.

remprop *sym attr* [function]

removes attribute-value pair from *sym*'s property list.

setq *&rest forms* [special]

for each form in *forms*, assigns the second element to the first element, which is either a symbol or a dotted-pair. The first element is searched for in the name spaces of local variables, object variables, and special variables in this order unless explicitly declared special.

set *sym val* [function]

assigns *val* to the special value of *sym*. **Set** cannot assign values to local or object variables.

defun *symbol lambda-list &rest body* [special]

defines a global function to *symbol*. First element in *body* can be a documentation string. Use *flet* or *labels* for defining local functions.

defmacro *symbol lambda-list &rest body* [special]

defines a global macro. EusLisp does not have facilities for defining locally scoped macros.

defvar *var &optional (init nil) doc* [macro]

If *var* symbol has any special value, **defvar** does nothing. If *var* is unbound, it is declared to be special and *init* is set to its value.

defparameter *var init &optional doc* [macro]

defparameter declares *var* to be special and *init* is set to its value, even if *var* already has value.

defconstant *sym val &optional doc* [macro]

defconstant sets *val* as *sym*'s special value. Unlike *defvar, defparameter* and *setq*, the value set by *defconstant* cannot be altered by these forms. If the value of a constant symbol is tried to be changed, an error is reported. However, another *defconstant* can override the previous constant value, issuing a warning message.

keywordp *obj* [function]

T if *obj* is a symbol and its home package is **KEYWORD**.

constantp *symbol* [function]

T if the symbol is declared to be constant with defconstant macro.

documentation *sym &optional type* [function]

retrieves documentation string of *sym*.

gensym *&optional x* [function]

creates a new uninterned symbol composed of a prefix string and a suffix number like `g001`. Uninterned symbols are denoted by the #: package prefix indicating no package is associated with the symbols. Symbols with #: prefix are unreadable symbols and the reader cannot create references to these uninterned symbols. *X* can either be a string or an integer, which is used as the prefix or the suffix.

gentemp *&optional (prefix "T") (pkg *package*)* [function]

creates a new symbol interned in *pkg*. In most applications, **gensym** is preferable to **gentemp**, because creation of uninterned symbols is faster and uninterned symbols are garbage collect-able.

7.2 Packages

Packages provide separate name spaces for groups of symbols. Common Lisp introduced the package system in order to reduce the symbol (function and variable name) conflict problems in the course of developing huge software systems which require more than one programmer to work together. Each package may have internal symbols and external symbols. When a symbol is created in a package, it is always internal, and it becomes external by **export**. External symbols in different packages are referenced by prefixing the package name and a single colon, as `x:*display*`, while referencing internal symbols in other packages requires double colons, as `sys::free-threads`. In order to omit this package prefixing, a package may **import** symbols from other packages. Moreover, **use-package** allows importing all external symbols from another package at once. When symbols are exported or imported, symbol name conflicts can be detected, since every symbol in any packages must have the unique print name. **Shadow** allows creating a symbol with the same print name as the existing symbol in a package by virtually removing the old symbol from the package.

EusLisp defines following eight packages;

lisp: All the lisp functions, macros, constants, etc.

keyword: keyword symbols

unix: unix system calls and library functions

system: system management or dangerous functions; nicknames=sys,si

compiler: EusLisp compiler; nicknames=comp

user: User's work space

geometry: geometric classes and functions

xwindow: X-window interface; nickname=x

These packages and user-defined packages are linked in the system's package list, which can be obtained by **list-all-packages**. Each package manages two hash tables to find and locate internal and external symbols. Also, a package records its name (string or symbol) and a list of nick names, and a list of other packages that the package is using. ***Package*** is a special variable that holds the current package for read and print. If ***package*** is not `user:`, top-level prompt changes to indicate the current package, like `mypkg:eus`$.

lisp-package [constant]

Lisp package.

user-package [constant]

User package.

unix-package [constant]

Unix package.

system-package [constant]

System Package.

keyword-package [constant]

Keyword Package.

find-symbol *string &optional (package *package*)* [function]

finds and locates the symbol which has *string* as its print name in *package*. If found, the symbol is returned, NIL otherwise.

make-symbol *string* [function]

makes a new uninterned symbol by the print name of *string*.

intern *string &optional (package *package*) (klass symbol)* [function]

tries to find a symbol whose print-name is same with *string*. If the search succeeds, the symbol is returned. If fails, a symbol whose print-name is *string* is newly made, and is located in *package*.

list-all-packages [function]

returns the list of all packages ever made.

find-package *name* [function]

find the package whose name or nickname is equal to the *name* string.

make-package *name &key nicknames (use '(lisp))* [function]

makes a new package by the name of *name*. *Name* can either be a string or a symbol. If the package already exists, error is reported.

in-package *pkg &key nicknames (uses '(lisp))* [function]

changes the current package (the value of ***package***) to *pkg*.

package-name *pkg* [function]

returns the string name of the *pkg* package.

package-nicknames *pkg* [function]

returns a list of nicknames of *pkg*.

rename-package *pkg new-name &optional new-nicknames* [function]

changes the name of *pkg* to *new-name* and its nicknames to *new-nicknames*, which can either be a symbol, a string, or a list of symbols or strings.

package-use-list *pkg* [function]

returns the list of packages which are used by *pkg*.

packagep *pkg* [function]

T if *pkg* is a package.

use-package *pkg &optional (curpkg *package*)* [function]

adds *pkg* to *curpkg*'s use-list. Once added, symbols in *pkg* become visible from *curpkg* without package prefix.

unuse-package *pkg &optional (curpkg *package*)* [function]

removes *pkg* from *curpkg*'s use-list.

shadow *sym &optional(pkg *package*)* [function]

makes a symbol interned in *pkg*, by hiding existing *sym*.

export *sym &optional (pkg *package*)* [function]

sym is a symbol or a list of symbols. **export** makes *sym* accessible from other packages as external symbol(s). Actually, *sym* is registered as an external symbol in *pkg*. If a symbol is exported, it becomes accessible using a single colon ":" as package marker, whereas unexported symbols require double colons. In addition, exported symbols do not need colons when they are used by **use-package** or they are imported into the package. Whether a symbol is exported or not is attributed to packages where it is interned, not to each symbol. So, a symbol can be internal in a package and external in another. **Export** checks *sym* to have name conflict with symbols in other packages using *pkg*. If there is a symbol having the same print name with *sym*, "symbol conflict" error is reported.

unexport *sym &optional pkg* [function]

If *sym* is an external symbol in *pkg*, it is unexported and becomes an internal symbol.

import *sym &optional (pkg *package*)* [function]

sym is a symbol or a list of symbols. **import** makes symbols defined in other packages visible in *pkg* as an internal symbol without package prefix. If there is already a symbol that has the same print name as *sym*, then an "name conflict" error is reported.

do-symbols *(var pkg &optional result) &rest forms* [macro]

repeats evaluatiing forms for each binding of *var* to symbols (internal or external) in *pkg*.

do-external-symbols *(var pkg &optional result) &rest forms* [macro]

repeats evaluating forms for each binding of *var* to external symbols in *pkg*.

do-all-symbols *(var &optional result) &rest forms* [macro]

repeats evaluating forms for each binding of *var* to symbols in all packages. Note that forms may be evaluated more than once to a symbol if it appears more than one package.

8 Sequences, Arrays and Tables

8.1 General Sequences

Vectors (one dimensional arrays) and lists are generic sequences. A string is a sequence, since it is a vector of characters.

For the specification of result type in **map, concatenate** and **coerce**, use class name symbol, such as `cons, string, integer-vector, float-vector`, etc. without quotes, since the class object is bound to the symbol.

elt *sequence pos* [function]

elt is the most general function to get and put (in conjunction with **setf**) value at the specific position *pos* in *sequence*. *Sequence* may be a list, or a vector of arbitrary object, bit, char, integer, or float. **Elt** cannot be applied to a multi-dimensional array.

length *sequence* [function]

returns the length of *sequence*. For vectors, **length** finishes in constant time, but time proportional to the length is required for a list. **Length** never terminates if *sequence* is a circular list. Use **list-length**, instead. If *sequence* is an array with a fill-pointer, **length** returns the fill-pointer, not the entire size of the array entity. Use **array-total-size** to know the entire size of those arrays.

subseq *sequence start &optional end* [function]

makes a copy of the subsequence from *start*th through (*end*-1)th inclusively out of *sequence*. *end* is defaulted to the length of *sequence*.

copy-seq *sequence* [function]

does shallow-copying of *sequence*, that is, only the top-level references in *sequence* are copied. Use **copy-tree** to copy a nested list, or **copy-object** for deep-copying of a sequence containing recursive references.

reverse *sequence* [function]

reverse the order of *sequence* and returns a new sequence of the same type as *sequence*.

nreverse *sequence* [function]

Nreverse is the destructive version of **reverse**. **Nreverse** does not allocate memory, while **reverse** does.

concatenate *result-type &rest sequences* [function]

concatenates all *sequences*. Each *sequence* may be of any sequence type. Unlike **append**, all the sequences including the last one are copied. *Result-type* should be a class such as cons, string, vector, float-vector etc.

coerce *sequence result-type* [function]

changes the type of *sequence*. For examples, `(coerce '(a b c) vector) = #(a b c)` and `(coerce "ABC" cons) = (a b c)`. A new sequence of type *result-type* is created, and each element of *sequence* is copied to it. *result-type* should be one of vector, integer-vector, float-vector, bit-vector, string, cons or other user-defined classes inheriting one of these. Note that *sequence* is copied even if its type equals to *result-type*.

map *result-type function seq &rest more-seqs* [function]

function is applied to a list of arguments taken from *seq* and *more-seqs* orderly, and the result is accumulated in a sequence of type *result-type*. For example, you can write as follows: `(map float-vector #'(lambda (x) (* x x)) (float-vector 1 2 3))`

fill *sequence item &key (start 0) (end (length sequence))* [function]

fills *item* from *start*th through (*end*-1)th in *sequence*.

replace *dest source &key start1 end1 start2 end2* [function]

elements in *dest* sequence indexed between *start1* and *end1* are replaced with elements in *source* indexed between *start2* and *end2*. *start1* and *start2* are defaulted to zero, and *end1* and *end2* to the length of each sequence. If the one of subsequences is longer than the other, its end is truncated to match with the shorter subsequence.

sort *sequence compare &optional key* [function]

sequence is destructively sorted using Unix's quick-sort subroutine. *key* is not a keyword parameter. Be careful with the sorting of a sequence which have same elements. For example, `(sort '(1 1) #'>)` fails because comparisons between 1 and 1 in both direction fail. To avoid this problem, use functions like #'>= or #'<= for comparison.

merge *result-type seq1 seq2 pred &key (key #'identity)* [function]

two sequences *seq1* and *seq2* are merged to form a single sequence of *result-type* whose elements satisfy the comparison specified by *pred*.

merge-list *list1 list2 pred key* [function]

merges two lists. Unlike **merge** no general sequences are allowed for the arguments, but **merge-list** runs faster than **merge**.

Following functions consist of one basic function and its variants suffixed by -if and -if-not. The basic form takes at least the item and sequence arguments, and compares item with each element in the sequence, and do some processing, such as finding the index, counting the number of appearances, removing the item, etc. Variant forms take predicate and sequence arguments, applies the predicate to each element of sequence, and do something if the predicate returns non-nil (-if version), or nil (-if-not version).

position *item seq &key start end test test-not key (count 1)* [function]

finds *count*th appearance of *item* in *seq* and returns its index. The search begins from the *start*th element, ignoring elements before it. By default, the search is performed by **eql**, which can be altered by the *test* or *test-not* parameter.

position-if *predicate seq &key start end key* [function]

position-if-not *predicate seq &key start end key* [function]

find *item seq &key start end test test-not key (count 1)* [function]

finds *count*th element between the *start*th element and the *end*th element in *seq*. The element found, which is eql to *item* if no *test* or *test-not* other than #'eql is specified, is returned.

find-if *predicate seq &key start end key (count 1)* [function]

finds *count*th element in *seq* for which *pred* returns non nil.

find-if-not *predicate seq &key start end key* [function]

count *item seq &key start end test test-not key* [function]

counts the number of *item*s which appear between the *start*th element and the *end*th element in *seq*.

count-if *predicate seq &key start end key* [function]

count the number of elements in *seq* for which *pred* returns non nil.

count-if-not *predicate seq &key start end key* [function]

remove *item seq &key start end test test-not key count* [function]

creates a new sequence which has eliminated *count* (defaulted to infinity) occurrences of of *item*(s) between the *start*th element and the *end*th element in *seq*. If you are sure that there is only one occurrence of *item*, *count=1* should be specified to avoid meaningless scan over the whole sequence.

remove-if *predicate seq &key start end key count* [function]

remove-if-not *predicate seq &key start end key count* [function]

remove-duplicates *seq &key start end key test test-not count* [function]

removes duplicated items in *seq* and creates a new sequence.

delete *item seq &key start end test test-not key count* [function]

is same with **remove** except that **delete** modifies *seq* destructively and does not create a new sequence. If you are sure that there is only one occurrence of *item*, *count=1* should be specified to avoid meaningless scan over the whole sequence.

delete-if *predicate seq &key start end key count* [function]

delete-if-not *predicate seq &key start end key count* [function]

count for *remove*s and *delete*s is defaulted to 1,000,000. If you have a long sequence and you want to delete an element which appears only once, :count should be specified as 1.

substitute *newitem olditem seq &key start end test test-not key count* [function]

returns a new sequence which has substituted the *count* occurrence(s) of *olditem* in *seq* with *newitem*. By default, all the *olditems* are substituted.

```
(substitute #\Space #\_ "Euslisp_euslisp") ;; => "Euslisp euslisp"
```

substitute-if *newitem predicate seq &key start end key count* [function]

substitute-if-not *newitem predicate seq &key start end key count* [function]

nsubstitute *newitem olditem seq &key start end test test-not key count* [function]

substitute the *count* occurrences of *olditem* in *seq* with *newitem* destructively. By default, all the *olditem*s are substituted.

nsubstitute-if *newitem predicate seq &key start end key count* [function]

nsubstitute-if-not *newitem predicate seq &key start end key count* [function]

8.2 Lists

listp *object* [function]

returns T if object is an instance of cons or NIL.

consp *object* [function]

equivalent to (not (atom object)). (consp '()) is nil.

car *list* [function]

returns the first element in *list.* **car** of NIL is NIL. **car** of atom is error.

cdr *list* [function]

returns the list which removed the first element of *list.* **cdr** of NIL is NIL. **cdr** of atom is error.

cadr *list* [function]

`(cadr list) = (car (cdr list))`

cddr *list* [function]

`(cddr list) = (cdr (cdr list))`

cdar *list* [function]

`(cdar list) = (cdr (car list))`

caar *list* [function]

`(caar list) = (car (car list))`

caddr *list* [function]

`(caddr list) = (car (cdr (cdr list)))`

caadr *list* [function]

`(caadr list) = (car (car (cdr list)))`

cadar *list* [function]

`(cadar list) = (car (cdr (car list)))`

caaar *list* [function]

`(caaar list) = (car (car (car list)))`

cdadr *list* [function]

`(cdadr list) = (cdr (car (cdr list)))`

cdaar *list* [function]

`(cdaar list) = (cdr (car (car list)))`

cdddr *list* [function]

`(cdddr list) = (cdr (cdr (cdr list)))`

cddar *list* [function]

`(cddar list) = (cdr (cdr (car list)))`

first *list* [function]

retrieves the first element in *list.* **second, third, fourth, fifth, sixth, seventh, eighth** are als available.

nth *count list* [function]

returns the *count*-th element in *list*. Note that `(nth 1 list)` is equivalent to `(second list)`, and to `(elt list 1)`.

nthcdr *count list* [function]

applies **cdr** *count* times to *list*.

last *list* [function]

the last cons is returned, not the last element.

butlast *list &optional (n 1)* [function]

returns a list which does not contain the last *n* elements.

cons *car cdr* [function]

makes a new cons whose car is *car* and cdr is *cdr*.

list *&rest elements* [function]

makes a list of *elements*.

list* *&rest elements* [function]

makes a list of *elements*, but the last element is consed in cdr: for example, `(list* 1 2 3 '(4 5)) = (1 2 3 4 5)`.

list-length *list* [function]

returns the length of the *list*. *List* can be circular.

make-list *size &key initial-element* [function]

makes a list whose length is *size* and elements are *initial-element*.

rplaca *cons a* [function]

replace the car of *cons* with a. Use of **setf** to **car** is recommended.

rplacd *cons d* [function]

replace the cdr of *cons* with d. Use of **setf** to **cdr** is recommended.

memq *item list* [function]

resembles **member**, but test is always done by **eq**.

member *item list &key key (test #'eq) test-not* [function]

the *list* is searched for an element that satisfies the *test*. If none is found, NIL is returned; otherwise, the tail of *list* beginning with the first element that satisfied the test is returned. The *list* is searched on the top level only.

assq *item alist* [function]

assoc *item alist &key key (test #'eq) test-not* [function]

searches the association list *alist*. The value returned is the first pair in the *alist* such that the *car* of the pair satisfies the *test*, or NIL if there is no such pair in the *alist*.

rassoc *item alist* [function]

returns the first pair in *alist* whose cdr is equal to *item*.

pairlis *l1 l2 &optional alist* [function] makes a list of pairs consing corresponding elements in *l1* and *l2*. If *alist* is given, it is concatenated at the tail of the pair list made from *l1* and *l2*.

acons *key val alist* [function]

add the *key val* pair to *alist*, that is, `(cons (cons key val) alist)`.

append *&rest list* [function]

appends *list* to form a new list. All the elements in *list*, except the last list, are copied.

nconc *&rest list* [function]

concatenates *list* destructively by replacing the last cdr of each *list*.

subst *new old tree* [function]

substitutes every *old* in *tree* with *new*.

flatten *complex-list* [function]

Complex-list composed of atoms and lists of any depth is transformed into a single level linear list which have all the elements in *complex-list* at the top level. For example, `(flatten '(a (b (c d) e))) = (a b c d e)`

push *item place* [macro]

pushes item into a stack (list) bound to *place*.

pop *stack* [macro]

removes the first item from *stack* and returns it. If *stack* is empty (nil), nil is returned.

pushnew *item place &key test test-not key* [macro]

pushes *item* in the *place* list if *item* is not a member of *place*. The *test*, *test-not* and *key* arguments are passed to the member function.

adjoin *item list* [function]

The item is added at the head of the list if it is not included in the list.

union *list1 list2 &key (test #'eq) test-not (key #'identity)* [function]

returns union set of two lists.

subsetp *list1 list2 &key (test #'eq) test-not (key #'identity)* [function]

tests if *list1* is a subset of *list2*, i.e. if each element of *list1* is a member of *list2*.

intersection *list1 list2 &key (test #'eq) test-not (key #'identity)* [function]

returns the intersection of two sets, *list1* and *list2*.

set-difference *list1 list2 &key (test #'eq) test-not (key #'identity)* [function]

returns the list whose elements are only contained in *list1* and not in *list2*.

set-exclusive-or *list1 list2 &key (test #'eq) test-not (key #'identity)* [function]

returns the list of elements that appear only either in *list1* or *list2*.

list-insert *item pos list* [function]

insert *item* as the *pos*'th element in *list* destructively. If *pos* is bigger than the length of *list*, *item* is nconc'ed at the tail. For example, `(list-insert 'x 2 '(a b c d)) = (a b x c d)`

copy-tree *tree* [function]

returns the copy of *tree* which may be a nested list but cannot have circular reference. Circular lists can be copied by **copy-object**. Actually, **copy-tree** is simply coded as `(subst t t tree)`.

mapc *func arg-list &rest more-arg-lists* [function]

applies *func* to a list of N-th elements in *arg-list* and each of *more-arg-lists*. The results of application are ignored and *arg-list* is returned.

mapcar *func &rest arg-list* [function]

maps *func* to each element of *arg-list*, and makes a list from all the results. For example, you can write as follows: `(mapcar #'(lambda (x) (* x x)) '(1 2 3))`. Before using **mapcar**, try **dolist**.

mapcan *func arg-list &rest more-arg-lists* [function]

maps *func* to each element of *arg-list*, and makes a list from all the results by **nconc**. **Mapcan** is suitable for filtering (selecting) elements in *arg-list*, since nconc does nothing with NIL.

8.3 Vectors and Arrays

Up to seven dimensional arrays are allowed. A one-dimensional array is called vector. Vectors and lists are grouped as sequence. If the elements of an array is of any type, the array is said to be general. If an array does not have fill-pointer, is not displaced to another array, or is adjustable, the array is said to be simple.

Every array element can be recalled by **aref** and set by **setf** in conjunction with aref. But for simple vectors, there are simpler and faster access functions: **svref** for simple general vectors, **char** and **schar** for simple character vectors (string), **bit** and **sbit** for simple bit vectors. When these functions are compiled, the access is expanded in-line and no type check and boundary check are performed.

Since a vector is also an object, it can be made by instantiating some vector-class. There are five kinds of built-in vector-classes; vector, string, float-vector, integer-vector and bit-vector. In order to ease instantiation of vectors, the function make-array is provided. Element-type should be one of **:integer, :bit, :character, :float, :foreign** or user-defined vector class. **:initial-element** and **:initial-contents** key word arguments are available to set initial values of the array you make.

array-rank-limit [constant]

7. Is the maximum array rank supported.

array-dimension-limit [constant]

#x1fffffff, logically, but stricter limit is imposed by the physical or virtual memory size of the system.

vectorp *object* [function]

An array is not a vector even if it is one dimensional. T is returned for vectors, integer-vectors, float-vectors, strings, bit-vectors or other user-defined vectors.

vector *&rest elements* [function]

makes a simple vector from *elements*.

make-array [function]

dims &key (element-type vector)
initial-contents
initial-element
fill-pointer
displaced-to
(displaced-index-offset 0)
adjustable

makes a vector or array. *dims* is either an integer or a list. If *dims* is an integer, a simple-vector is created.

svref *vector pos* [function]

returns *pos*th element of *vector*. *Vector* must be a simple general vector.

aref *vector &rest indices* [function]

returns the element indexed by *indices*. **Aref** is not very efficient because it needs to dispatch according to the type of *vector*. Type declarations should be given to improve the speed of compiled code whenever possible.

vector-push *val array* [function]

store *val* at the *fill-pointer*th slot in *array*. *array* must have a *fill-pointer*. After val is stored, the fill-pointer is advanced by one to point to the next location. If it exceeds the array boundary, an error is reported.

vector-push-extend *val array* [function]

Similar to **vector-push** except that the size of the array is automatically extended when *array*'s fill-pointer reaches the end.

arrayp *obj* [function]

T if *obj* is an instance of array or vector.

array-total-size *array* [function]

returns the total number of elements of *array*.

fill-pointer *array* [function]

returns the fill-pointer of *array*. Returns NIL if *array* does not have any fill-pointer.

array-rank *array* [function]

returns the rank of *array*.

array-dimensions *array* [function]

returns a list of array-dimensions.

array-dimension *array axis* [function]

Axis starts from 0. **array-dimension** returns the *axis*th dimension of *array*.

bit *bitvec index* [function]

returns the *index*th element of *bitvec*. Use **setf** and **bit** to change an element of a bit-vector.

bit-and *bits1 bits2 &optional result* [function]

bit-ior *bits1 bits2 &optional result* [function]

bit-xor *bits1 bits2 &optional result* [function]

bit-eqv *bits1 bits2 &optional result* [function]

bit-nand *bits1 bits2 &optional result* [function]

bit-nor *bits1 bits2 &optional result* [function]

bit-not *bits1 &optional result* [function]

For bit vectors *bits1* and *bits2* of the same length, their boolean and, inclusive-or, exclusive-or, equivalence, not-and, not-or and not are returned, respectively.

8.4 Characters and Strings

There is no character type in EusLisp; a character is represented by an integer. In order to handle strings representing file names, use **pathname**s described in **??**.

digit-char-p *ch* [function]

T if *ch* is #\0 through #\9.

alpha-char-p *ch* [function]

T if *ch* is #\A through #\Z or #\a through #\z.

upper-case-p *ch* [function]

T if *ch* is #\A through #\Z.

lower-case-p *ch* [function]

T if *ch* is #\a through #\z.

alphanumericp *ch* [function]

T if *ch* is #\0 through #\9, #\A through #\Z or #\a through #\z.

char-upcase *ch* [function]

convert the case of *ch* to upper.

char-downcase *ch* [function]

convert the case of *ch* to lower.

char *string index* [function]

returns *index*th character in *string*.

schar *string index* [function]

extracts a character from *string*. Use **schar** only if the type of *string* is definitely known and no type check is required.

stringp *object* [function]

returns T if *object* is a vector of bytes (integers less than 256).

string-upcase *str &key start end* [function]

converts *str* to upper case string and returns a new string.

string-downcase *str &key start end* [function]

converts *str* to lower case string and returns a new string.

nstring-upcase *str* [function]

converts *str* to upper case string destructively.

nstring-downcase *str &key start end* [function]

converts *str* to lower case string destructively.

string= *str1 str2 &key start1 end1 start2 end2* [function]

T if *str1* is equal to *str2*. *string=* is case sensitive.

string-equal *str1 str2 &key start1 end1 start2 end2* [function]

tests equality of *str1* and *str2*. **string-equal** is not case sensitive.

string *object* [function]

gets string notation of *object*. If *object* is a string, the *object* is returned. If *object* is a symbol, its pname is copied and returned. Note that (equal (string 'a) (symbol-pname 'a))==T, but (eq (string 'a) (symbol-pname 'a))==NIL. If object is number its string representation is returned (this is incompatible with Common Lisp). In order to get string representation for more complex objects, use **format** with NIL in the first argument.

string< *str1 str2* [function]

string<= *str1 str2* [function]

string> *str1 str2* [function]

string>= *str1 str2* [function]

string-left-trim *bag str* [function]

string-right-trim *bag str* [function]

str is scanned from the left(or right), and its elements are removed if it is included in the *bag* list. Once a character other than the ones in the *bag* is found, further scan is aborted and the rest of *str* is returned.

string-trim *bag str* [function]

Bag is a sequence of character codes. A new copy of *str* which does not contain characters specified in *bag* in its both end is made and returned.

substringp *sub string* [function]

T if string *sub* is contained in *string* as a substring. Not case sensitive.

8.5 Foreign Strings

A foreign-string is a kind of byte-vector whose entity is held somewhere outside EusLisp's heap. While a normal string is represented by a sequence of bytes and its length, a foreign-string holds the length and the address of the string entity. Although foreign-string is a string, some string and sequence functions cannot be applicable. Only **length**, **aref**, **replace**, **subseq** and **copy-seq** recognize the foreign-string, and application of other functions may cause a crash.

A foreign-string may refer to a part of I/O space usually taken in /dev/a??d?? special file where ?? is either 32 or 16. In case the device attached in one of these I/O space only responds to byte access, **replace** always copies element byte by byte, which is relatively slow when a large chunk of memory is accessed consecutively.

make-foreign-string *address length* [function]

makes an instance of foreign-string located at *address* and spanning for *length* bytes. For example, `(make-foreign-string (unix:malloc 32) 32)` makes a reference to a 32-byte memory located outside EusLisp's heap.

8.6 Hash Tables

Hash-table is a class to search for the value associated with a key, as accomplished by **assoc**. For a relatively large problem, hash-table performs better than assoc, since time required for searching remains constant even the number of key-value pairs increases. Roughly speaking, hash-table should be used in search spaces with more than 100 elements, and assoc in smaller spaces.

Hash-tables automatically expands if the number of elements in the table exceeds rehash-size. By default, expansion occurs when a half of the table is filled. **sxhash** function returns a hash value which is independent of memory address of an object, and hash values for **equal** objects are always the same. So, hash tables can be re-loadable since they use sxhash as their default hashing functions. While sxhash is robust and safe, it is relatively slow because it scans all the elements in a sequence or a tree. For faster hashing, you may choose another hash function appropriate for your application. To change the hash function, send `:hash-function` message to the hash-table. In simple cases, it is useful to change hash function from #'**sxhash** to #'**sys:address**. This is possible because the addresses of any objects never change in a EusLisp process.

sxhash *obj* [function]

calculates the hash value for *obj*. Two objects which are **equal** are guaranteed to yield the same hash value. For a symbol, hash value for its pname is returned. For numbers, their integer representations are returned. For a list, sum of hash values for all its elements is returned. For a string, shifted sum of each character code is returned. For any other objects, **sxhash** is recursively called to calculate the hash value of each slot, and the sum of them is returned.

make-hash-table *&key (size 30) (test #'eq) (rehash-size 2.0)* [function]

creates a hash table and returns it.

gethash *key htab* [function]

gets the value that corresponds to *key* in *htab*. **Gethash** is also used to set a value to key by combining with **setf**. When a new entry is entered in a hash table, and the number of filled slots in the table exceeds 1/rehash-size, then the hash table is automatically expanded to twice larger size.

remhash *key htab* [function]

removes a hash entry designated by *key* in *htab*.

maphash *function htab* [function]

maps *function* to all the elements of *htab*.

hash-table-p *x* [function]

T if *x* is an instance of class hash-table.

hash-table [Class]

:super	**object**
:slots	(key value count hash-function test-function rehash-size empty deleted)

defines hash table. *Key* and *value* are simple-vectors of the same *size*. *Count* is the number of filled slots in *key* and *value*. *Hash-function* is defaulted to **sxhash** and *test-function* to **eq**. *Empty* and *deleted* are uninterned symbols to indicate slots are empty or deleted in the *key* vector.

:hash-function *newhash* [method]

changes the hash function of this hash table to *newhash*. *Newhash* must be a function with one argument and returns an integer. One of candidates for *newhash* is **system:address**.

8.7 Queue

A queue is a data structure that allows insertion and retrieval of data in the FIFO manner, i.e. the first-in first-out order. Since the queue class is defined by extending the cons class, ordinary list functions can be applied to a queue. For example, caar retrieves the next element to be dequeued, and cadr gets the element that is queued most recently.

queue [Class]

:super	**cons**
:slots	(car cdr)

defines queue (FIFO) objects.

:init [method]

initializes the queue to have no elements.

:enqueue *val* [method]

puts val in the queue as the most recent element.

:dequeue *&optional (error-p nil)* [method]

retrieves the oldest value in the queue, and removes it of the queue. If the queue is empty, it reports an error when error-p is non-nil, or returns NIL otherwise.

:empty? [method]

returns T if the queue is empty.

:length [method]

returns the length of the queue.

:trim *s* [method]

discard old entries to keep the size of this queue to s.

:search *item &optional (test #'equal)* [method]

find element which is equal to item. the search is performed by equal, which can be altered by test

:delete *item &optional (test #'equal) (count 1)* [method]

eliminate count occurrences of item in this queue.

:first [method]

returns the first entry (oldest value) of this queue.

:last [method]

returns tha last entry (newest value) of this queue.

9 Streams and Input/Output

9.1 Streams

Echo-streams and concatenated-streams are not available. Predefined streams are following:

standard-input stdin fd=0

standard-output stdout fd=1

error-output stderr fd=2 bufsize=1

terminal-io two-way stream made of ***standard-input*** and ***standard-output***

streamp *object* [function]

Any object created from **stream**, **io-stream**, or their subclasses returns T.

input-stream-p *object* [function]

T if *object* is a stream and capable of reading.

output-stream-p *object* [function]

T if *object* is a stream and capable of writing.

io-stream-p *object* [function]

T if *object* is a two-way stream.

open [function]

path &key (direction :input)
(if-exists :new-version)
(if-does-not-exist 'default)
(permission #o644)
(buffer-size 512)

Open makes a stream associated with a file designated by *path*. *path* may either be a string or a pathname. Direction should be one of **:input, :output** or **:io**. Several open options, **:append, :new-version, :overwrite, :error** and nil are allowed for **:if-exists** parameter. However, this parameter is ignored when *direction* is **:input**. Alternatives for **:if-does-not-exist** are **:error, :create** and nil. **:new-version**, **:rename** and **:supersede** are not recognized. By default, the file is overwritten if *direction* is either **:output** or **:io** when the file exists. For :input files, an error is reported when the file does not exist. To know the existence of a file, **probe-file** can be used. Default value for **buffer-size** is 512 bytes, and #O644 for **:permission**. SunOS4 allows to open as many as sixty files at the same time.

with-open-file *(svar path &rest open-options) &rest forms* [macro]

A file named *path* is opened with *open-options* and the stream is bound to *svar*. Then *forms* are evaluated. The stream is automatically closed when evaluation of *forms* finishes or exits with **throw**, **return-from** or error. **With-open-file** is a macro defined by **unwind-protect** with **close** in its clean-up forms.

close *stream* [function]

closes the *stream*, and returns T if successful. The stream may have already been closed, in which case nil is returned. Streams are automatically closed by GC if there is no reference to that stream object.

make-string-input-stream *string* [function]

makes an input stream from a string.

make-string-output-stream *&optional size* [function]

makes an output stream to a string of *size* length. Actually, the length is automatically expanded, so *size* is only advisory information to allocate string at initialization.

get-output-stream-string *string-stream* [function]

gets a string out of a *string-stream*.

make-broadcast-stream *&rest output-streams* [function]

makes a broad-cast stream which forwards all the messages written to this stream to each of *output-streams*.

9.2 Reader

Reader's global variables:

read-base number base to be read; default is decimal ten

readtable current readtable which determines reader syntax

Reader's default macro characters:

(	read list
"	read string
'	read quoted expression
#	dispatch macro
;	comment until end of line
`	back-quote
,	list-time eval
@	append
%	read C-like mathematical forms

Escape characters:

\	single character escape
|...|	multiple character escape

When an unescaped symbol is read, all the constituent characters are converted to upcase by default, and upcase-character symbol is stored internally. For example, 'abc and 'ABC are regarded as the same symbol. Escape is necessary to distinguish between them. '|ABC|, 'ABC and 'abc are identical, while '|abc| and 'abc are different symbols. By default, even if you enter a symbol with upcase letters, When symbols are printed, EusLisp's printer converts them into lowercase from internal upcase representation. This conversion is suppressed by setting ***print-case*** to **:UPCASE**.

Note that 10. is read as integer 10, not floating 10.0. Since ':' is reserved for package marker, it must be escaped when used as a constituent of a symbol, like '$|g : pcube|$. This restriction is imposed not by the syntax of the character ':', but by the attribute which determines the alphabetical order and the meaning of the letter. The attributes of characters are hardwired in the reader. Thus, although you may change the syntax of a certain character by creating a new readtable by **copy-readtable** and resetting the syntactic meaning for the character by **set-syntax-from-char**, you cannot change its attribute anyway. In other words, digits are always digits, alphabets are alphabets, and we cannot use letters like '#$%@' to represent numbers.

String is denoted by two double quotes '"' at the beginning and at the end. No case conversion is taken inside the quotes. A back-slash 'ís used as an escape to include a double quote. Therefore, "He said, Ï like Lisp.ῗs read as a string including two double quotes. To enter a back-slash, two back-slashes are needed. Note that shift-JIS encoding of Japanese text is inadequate for this read-string convention, since some characters happen to have the code of a back-slash (`#x5c`) as their second byte. Use of EUC coding is preferrable.

% is an extended read-macro character specific to EusLisp. Preceding % to a mathematical formula written in infix notation, the formula is converted to lisp's prefix form. For an instance, `%(1 + 2 * 3 / 4.0)` is transformed to `(+ 1 (/ (* 2 3) 4.0))` and `2.5` is resulted. C-like function calls and array references are converted to lisp forms, too, thus, `%(sin(x) + a[1])` is evaluated to `(+ (sin x) (aref a 1))`. Functions having more than one arguments and arrays of more than two dimeisions are notated as `func(a b c ...)` and `ary[1 2 3 ...]`, not `func(a,b,c)` nor `ary[1][2][3]`. Relative expressions and assignments are also properly handled, so, `%(a < b)` is converted to `(< a b)`, and `%(a[0] = b[0] * c[0])` is to `(setf (aref b 0) (* (aref b 0) (aref c 0)))`. A simple optimization is performed to reduce duplicated function calls and array references. `%(sin(x) + cos(x) / sin(x))` is converted into `(let* ((temp (sin x))) (+ temp (/ (cos x) temp)))`.

Dispatch macros are preceeded by the `#` character. A number (integer) argument can be given between `#` and a dispatch macro character. This means that any digits (0 .. 9) cannot be defined as dispatch macro characters. Reader's standard dispatch macro characters follow:

`#nA(..)` array

`#B` binary number

`#D` degree to radian conversion; #D180 = 3.14

`#F(...)` floatvector

`#nF((..))` float array; #2F((..) (..)) is matrix

`#I(...)` integer-vector

`#nI((...))` integer array

`#J(...)` general object #J(myclass ....); obsolete

`#O` octal number

`#P` pathname

`#R` radian to degree conversion; #R3.14 = 180.0

`#S(classname slotname1 val1 slotname2 val2 ...)` structure (any object)

`#V(...)` vector #V(vectorclass ...)

`#X` hexadecimal number

`#(...)` vector

`#n#` label reference

`#n=` label definition

`#'` FUNCTION; compiled-code or lambda-closure

`#\` character

`#,` read-time evaluation

`#+` conditional read (positive)

`#-` conditional read (negative)

`#*` bit vector

`#:` uninterned symbol

`#|...|#` comment; can be nested

Some reader functions have *eof-error-p, eof-value* and *recursive-p* parameters. The first two parameters control the behavior when the reader encounters with end-of-file. The default of *eof-error-p* is t, which causes an error at eof. If you want to know the occurrence of eof and don't want the system's error-handler to snatch control, specify nil to *eof-error-p*. Thus, when an eof appears during reading, the reader returns the *eof-value* instead of entering an error loop. *Eof-value* is defaulted to nil. So, you cannot know if nil is actually read, or eof appears. To distinguish them, give a value which can never appear in the stream. Use **cons** or **gensym** to make such unique data object.

Recursive-p is often used in read-macro functions, which call reader recursively. Non-nil value of *recursive-p* tells the reader that the read operation has been started somewhere else and it should not reset the internal table for reading forms labeled by `#n=` and `#n#`.

read *&optional stream (eof-error-p t) (eof-value nil) recursive-p* [function]

reads one S-expression.

read-delimited-list *delim-char &optional stream recursive-p* [function]

reads s-expression which is delimited by *delim-char*. This is useful to read comma-separated list, or to read a sequence terminated by a special character like `#\]`.

read-line *&optional stream (eof-error-p t) (eof-value nil)* [function]

reads a line which is terminated by a `#\newline`. The string returned does not contain the last newline character.

read-char *&optional stream (eof-error-p t) (eof-value nil)* [function]

reads one character and returns its integer representation.

read-from-string *string &optional (eof-error-p t) (eof-value nil)* [function]

reads one s-expression from *string*. Only the first s-expression can be read. If successive read operations need to be performed on a string containing more than one expression, use string-stream made by **make-string-input-stream**.

unread-char *char &optional stream* [function]

puts the *char* back to the *stream*. More than one characters cannot be put back successively.

peek-char *&optional stream (eof-error-p t) (eof-value nil)* [function]

reads a character from the *stream* without removing it from the buffer of the *stream*. This is equivalent to a *read-char* followed by a *unread-char*.

y-or-n-p *&optional format-string &rest args* [function]

prints *format-string* and *args* on your terminal, and asks "y-or-n". Repeat query until your response begins with either of "y" or "n", and returns T or NIL. Case does not matter.

yes-or-no-p *&optional stream* [function]

prints *format-string* and *args* on your terminal, and asks "yes-or-no". Repeat query until your response is either of "yes" or "no", and returns T or NIL. Case does not matter.

In the readtable manipulating functions, the default value of readtable is the value of the global variable ***readtable***.

readtablep *x* [function]

T if *x* is an readtable.

copy-readtable *&optional from-readtable to-readtable* [function]

If no *to-readtable* is specified, a new one is created. All the information in *from-readtable* is transferd to *to-readtable*. The information included is, syntax table, read-macro table and dispatch-macro table, each of which has 256 elements.

set-syntax-from-char *from-char to-char &optional to-readtable from-readtable* [function]

copies syntax and read-macro definition of *from-char* in *from-readtable* to that of *to-char* in *to-readtable*.

set-macro-character *char func &optional non-teminating-p readtable* [function]

defines *func* as the read-macro function for *char*.

get-macro-character *char &optional readtable* [function]

returns the read-macro function for *char*.

set-dispatch-macro-character *dispchar char func &optional readtable* [function]

defines *func* as the dispatch read-macro function for the combination of *dispchar* and *char*.

get-dispatch-macro-character *dispchar char &optional readtable* [function]

returns the dispatch read-macro function for the combination of *dispchar* and *char*.

9.3 Printer

The followings are special variables controlling printer's behaviors.

print-case if this is `:downcase`, all symbols are printed in lowercase althought symbols are represented in uppercase internally unless they are escaped.

print-circle print objects preserving recursive refernce

print-object print the details of all objects

print-structure print objects using #s format.

print-level printable depth of a sequence

print-length printable length of a sequence

print-escape currently not used

print-pretty currently not used

print-base number base in printing; defaulted to decimal ten

In order to print objects containing recursive references so that they can be read back again, print the objects with both ***print-circle*** and ***print-structure*** set to T. Although most of the user defined objects can be printed in re-readable forms, classes, compiled-codes and packages cannot be dumped in that way, because classes and compiled-code include unrelocatable executable codes, and the rereading packages damages the consistency among symbols.

print *obj &optional stream* [function]

is **prin1** followed by **terpri**.

prin1 *obj &optional stream* [function]

outputs one s-expression in the format that they can be read back again by **read**. The format includes slashes (escapes) and quotation marks.

princ *obj &optional stream* [function]

same as **print** except that **princ** does not add escape or quote. Objects printed by **princ** cannot be read back. For example, the output of `(princ 'abc)` is identical with that of `(princ "abc")` and the reader cannot distinguish between them.

terpri *&optional stream* [function]

outputs #\newline and flush *stream*.

finish-output *&optional stream* [function]

flushes output stream.

princ-to-string *x &optional (l 16)* [function]

prin1-to-string *x &optional (l 16)* [function]

makes a string-output-stream, writes to it, and get-output-stream-string.

format *stream format-string &rest args* [function]

Format only recognizes ~A(ascii), ~S(S-expression), ~D(decimal), ~X(hexadicimal), ~O(octal), ~C(character), ~F(floating), ~E(exponential), ~G(general float), ~V(dynamic number parameter), ~T(tab) and ~%(newline) format specifiers.

```
(format t "~s ~s ~a ~a ~10,3f~%" "abc" 'a#b "abc" 'a#b 1.2)
--->  "abc" |A#B| abc a#b     1.200
```

pprint *obj &optional (stream *standard-output*) (tab 0) (platen 75)* [function]

pretty-prints *obj*.

print-functions *file &rest fns* [function]

write the "defun" forms of function definitions of *fns* out to *file*.

write-byte *integer stream* [function]

write-word *integer stream* [function]

write-long *integer stream* [function]

write *integer* as a one-, two- or four-byte binary.

spaces *n &optional stream* [function]

outputs spaces *n* times.

pf *func &optional stream *standard-output*)* [macro]

pretty-prints a function. Compiled function cannot be printed.

pp-method *class selector &optional (stream *standard-output*)* [function]

pretty-prints the method defined in *class* by the name of *selector*.

tprint *obj tab &optional (indent 0) (platen 79) (cpos 0)* [function]

print *obj* in tabular format.

print-size *obj* [function]

returns inexact length of obj when it is printed.

9.4 InterProcess Communication and Network

EusLisp provides four kinds of IPC facilities, *shared memory, message-queue, FIFO* and *socket.* [2] Normally, efficiency decreases in this order. If you are using multithread facility, synchronization functions described in the section **??** are also used for communications. Availability of these facilities depends on the configuration and the version of Unix.

9.4.1 Shared Memory

EusLisp supports the shared memory provided by SunOS's mmap, not by System5's shmem. Shared memory is allocated by the **map-file** function. **Map-file** maps a file into the EusLisp process memory space and an instance of **foreign-string** is returned. Data can be written and retrieved using string functions on this foreign-string. Since shared memory is allocated at system-dependent page boundary, you should not specify the map address. Mapping a file with the **:share** keyparameter set to NIL or **:private** set to T means the file should be accessed privately (exclusively). Since this is not useful for the purpose of memory sharing, the default value of **:share** key is T. When a file is shared between two users, the read/write permission must be properly set for both users. Unfortunately, SunOS does not support file sharing through networks between different workstations.

Example programs to share a file of 64 byte length between two euslisp are shown below.

```
;; Create a file of 64 bytes
(with-open-file (f "afile" :direction :output)  (princ (make-string 64) f))
;; Map it
(setq shared-string1 (map-file "afile" :direction :io))
;;
;; In another process
(setq shared-string2 (map-file "afile" :direction :io))
```

Then, data written to `shared-string1` immediately appears in `shared-string2`, and vice versa. Writing to a foreign string can be made by **replace** or **setf** in conjunction with **aref**.

map-file *filename &key (direction :input) length (offset 0) (share t) (address 0)* [function]

maps the file named *filename* to memory space. *Filename* can be either of a local file, an NFS-mounted remote file, or a memory device in `/dev`. A **foreign-string**, whose elements can be accessed by **aref**, is returned. Writing data into a foreign-string mapped by `map-file` with *direction=:input* will result a segmentation fault.

9.4.2 Message Queues and FIFOs

A message-queue is created by **make-msgq-input-stream** or **make-msgq-output-stream**. Each of these returns an instance of file-stream, which can then accept read and print operations like other streams connected to files. The `fname` slot of message-queue stream is set to the key when it is created.

To make a stream to FIFO, you first create a FIFO node with **unix:mknod** function by setting its second argument *mode*=#o10000, and you open it as a normal file. Message-queues and FIFOs are created locally on a machine and only provide communication channels within the machine.

[2] Since the pipe, the traditional process communication mechanism in Unix, is always used in conjunction with 'fork', EusLisp provides the **piped-fork** function explained in the section **??**.

Note that message-queues and FIFOs are not removed from the system even after the owner process terminates. Explicit use of **unix:msgctl** or **ipcrm** command is needed to delete them.

make-msgq-input-stream *key &optional (buffer-size 128)* [function]

returns an input file-stream which is connected to a message-queue identified by *key*.

make-msgq-output-stream *key &optional (buffer-size 128)* [function]

returns an output file-stream which is connected to a message-queue identified by *key*.

9.4.3 Sockets

The socket is more versatile than other communication mechanisms because it can operate either host-locally (in unix domain) or network-widely (in internet domain). Connection-oriented socket (SOCK_STREAM) and unconnected socket (SOCK_DGRAM) are supported. In both cases, you must first create a socket address object by **make-socket-address** function, which returns an instance of `socket-address`. In unix domain, a socket address is specified by a path-name in the unix file system. In internet domain, the address is specified by combining the host machine name, the port number, and optionally the protocol number. If the port number is defined in `/etc/services`, it can be referred through the symbol specified by the service name. The function **unix:getservbyname** can be used to retrieve the port number from the symbolic service name. Port numbers less than 1024 are reserved for root users, and non-priviledged users are advised to use port numbers greater than 1024 for their private sockets.

Although connected streams provide bidirectional communication channels, the connection establishment operation is asymmetric. One endpoint is refered to server and other to client. The endpoint on the behalf of the server (service access point) must be first established. It is created by **make-socket-port** function which returns an instance of `socket-port`. The socket-port object is then used to accept connections from one or more clients by **make-server-socket-stream**. A call to **make-server-socket-stream** may be blocked until a connection request from a client really happens. Clients can make socket streams by **make-client-socket-stream** specifying a socket-address.

```
;;; an example of IPC through a socket stream:
;;; server side
(setq saddr  (make-socket-address :domain af_inet :host "etlic2" :port 2000))
(setq sport  (make-socket-port saddr))
(setq sstream (make-server-socket-stream sport))
;;;
;;; client side
(setq caddr (make-socket-address :domain af_inet :host "etlic2" :port 2000))
(setq cstream (make-client-socket-stream caddr))
```

In applications like a database or an environment simulator for mobile robots, *multiple connection service* between one server and many clients is required. This type of server can be programmed by the **open-server** function. From the current host name and given port number, **open-server** creates a socket port (service access point) on which connection requests are listened for. Since this port is attributed to be asynchronous, **open-server** is not blocked and returns immediately. Thereafter, each connection request interrupts EusLisp's main loop, and an socket-stream is created asynchronously. This socket-stream also works in asynchronous mode: the asynchronous input processing function which is the second argument to

open-server is invoked whenever new data appear in this stream. Up to 30 connections can be established so that as many clients can access the server's data at the same time.

```
;; server side
(defun server-func (s)
   (case (read s) ...    ;do appropriate jobs according to inputs
(open-server 3000 #'server-func)
... do other jobs in parallel
;; client-1 through client-N
(setq s (connect-server "etlmmd" 3000))
(format s "..." ...) (finish-output s) ;issue a command to the server
(read s)                                ;receive response
```

In contrast to the *connection-oriented* streams which provide reliable communication channels, the *connectionless* sockets are unreliable: messages may be lost, duplicated, and may arrive out-of-order. The *connectionless* sockets, however, have advantages that they do not need to assign file descriptor to each connection, and sending process is never blocked even if the receiver is not reading data and the buffer overflows.

To make connectionless sockets, use the following procedures. Messages are transferred by the **unix:sendto** and **unix:recvfrom**.

```
;;; receiver side
(setq saddr  (make-socket-address :domain af_inet :host "etlic2" :port 2001))
(setq sock   (make-dgram-socket saddr))
(unix:recvfrom sock)
;;;
;;; client side
(setq caddr (make-socket-address :domain af_inet :host "etlic2" :port 2001))
(setq sock (unix:socket (send caddr :domain) 2 0))
(unix:sendto sock caddr "this is a message")
;;;
;;; how to use echo service which is registered in /etc/services.
(setq caddr (make-socket-address :domain af_inet :host "etlic2"
                                 :port (car (unix:getservbyname "echo"))))
(setq echosock (unix:socket (send caddr :domain) 2 0))
(unix:sendto echosock caddr "this is a message")
(unix:recvfrom echosock)  --> "this is a message"
```

make-socket-address *&key domain pathname host port proto service* [function]

makes a sockaddr structure.

make-socket-port *sockaddr* [function]

makes a server-side socket port which is used to establish a connection with a client.

make-server-socket-stream *sockport &optional (size 100)* [function]

accepts a connection from a client and returns a two-way stream.

make-client-socket-stream *sockaddr &optional (timeout 5) (size 512)* [function] connects to a server port and returns a two-way stream.

open-server *port remote-func* [function]

prepares a socket port designated by the host name and *port* in internetnet domain, and waits for the connection requests asynchronously. Each time a connection is requested, it is accepted and a new socket-stream is opened. When a message arrives at the socket-port, *remote-func* is invoked with the socket port as the argument.

connect-server *host port* [function]

This is a shorhand of successive calls to make-socket-address and make-client-socket-stream. A socket-stream for a client to communicate with the server specified by *host* and *port* is returned. The port is made in internet domain.

9.5 Asynchronous Input/Output

select-stream *stream-list timeout* [function]

finds a list of streams which are ready for input operation, in *stream-list*. NIL is returned if *timeout* seconds have expired before any streams become ready. **Select-stream** is useful when you choose active streams out of a list of input-streams on which input operation becomes possible asynchronously. *Timeout* specifies the time when the select operation is aborted. It can be a float number. If no timeout is specified, select-stream blocks until input arrives at least one stream. If *timeout* is specified and no input appears on any streams, select-stream aborts and returns NIL.

def-async *stream function* [macro]

defines *function* to be called when data arrives at *stream*. *stream* is either a file-stream or a socket-port. When data comes to the file-stream or a connection request appears on the socket-port, *function* is invoked with the stream as its argument. This macro installs a SIGIO handler that dispatches to user supplied *function* which is expected to perform actual input operation, and uses **unix:fcntl** on *stream* to issue SIGIO asynchronously when *stream* becomes ready to be read.

9.6 Pathnames

Pathnames give the way to analyze and compose file names OS-independently. A typical path name is assumed to be consisted of following components: host:device/directory1/.../directory-n/name.type.version. Since EusLisp only runs on Unix, host, device and version fields are ignored. The **pathname** function decomposes a string into a list of directory components, name and type, and returns a pathname object, which is printed as a string prefixed by `#P`.

pathnamep *name* [function]

returns T if *name* is a pathname.

pathname *name* [function]

name is pathname or string. *name* is converted to pathname. To indicate the last name is a directory name, don't forget to suffix with "/". The inverse conversion is performed by *namestring*.

pathname-directory *path* [function]

returns a list of directory names of *path*. Root directory (/) is represented by :ROOT. *path* can be either of string or pathname.

pathname-name *path* [function]

returns the file-name portion of *path*. *path* can be either of string or pathname.

pathname-type *path* [function]

extracts the file-type portion out of *path*. *path* can be either of string or pathname.

make-pathname *&key host device directory name type version defaults* [function]

makes a new pathname from *directory*, *name* and *type*. On unix, other parameters are ignored.

merge-pathnames *name &optional (defaults *default-pathname-defaults*)* [function]

namestring *path* [function]

returns string representation of *path*.

parse-namestring *name* [function]

truename *path* [function]

tries to find the absolute pathname for the file named *path*.

9.7 URL-Pathnames

URL-Pathname is an extension of pathname to have slots for a protocol and a port. A URL is composed of six components; protocol, server, port, directories, filename, and file-type, like `http://shock2.etl.go.jp/matsui/inc`

url-pathname *name* [function]

name is pathname or string. *name* is converted to pathname. To indicate the last name is a directory name, don't forget to suffix with "/". The inverse conversion is performed by *namestring*.

9.8 File-name generation

digits-string *n digits &optional (base 10))* [function]

generates a string representing the integer *n* in n columns of digits. Zeros are padded before the number if n is too small to represent in digits.

sequential-file-name *head num extension &optional (digits 4))* [function]

generates a filename string with an advancing number part. This is similar to gentemp, but differs in that an extension can be specified and the result is a string.

timed-file-name *head extension &optional (dt (unix:localtime)))* [function]

generates a filename string that consists of head, hour, minute, second, and extension. For example, (timed-file-name "img" "jpg") generates "img191015.jpg" at 7:10:15 pm.

dated-file-name *head extension &optional (dt (unix:localtime)))* [function]

generates a filename string formatted as "headyymmmdd.extension", where yy is the lower two digits of the year, mmm is the abbreviated month name, and dd is the date.

9.9 File System Interface

probe-file *path* [function]

checks if a file named *path* exists.

file-size *path* [function]

returns the size of the file named *path* in bytes.

directory-p *path* [function]

returns T if *path* is a directory, NIL otherwise even *path* does not exist.

find-executable *file* [function]

returns the full pathname for the Unix command named *file*. *Find-executable* provides almost the same functionality with Unix's 'which' command that searches the executable file in your path list.

file-write-date *file* [function]

returns the integer representation of the time when the *file* was last modified. String representation can be obtained by `(unix:asctime (unix:localtime (file-write-date file)))`

file-newer *new old* [function]

returns T if the *new* file is modified more recently than the *old* file.

object-file-p *file* [function]

returns T if the *file* is an object file by looking at the file's magic number in the header.

directory *&optional (path ".")* [function]

makes a list of all the files in the *path*.

dir *&optional (dir ".")* [function]

prints file names in the specified directory.

10 Evaluation

10.1 Evaluators

In order to specify the behaviors upon an error and an interrupt(signal), set an appropriate function to each of the special variables ***error-handler*** and ***signal-handler*** in advance. There is no correctable or continue-able error. After analyzing errors you must abort the current execution by **reset** or appropriate **throw** to upper level catchers. **reset** is equivalent to `(throw 0 NIL)`, since EusLisp's top-level creates catch frame named `0`.

Error handlers should be programmed as functions with three or four arguments: *code msg1 form &optional (msg2)*. *Code* is the error code which identifies system defined errors, such as 14 for 'mismatch argument' or 13 for 'undefined function'. These mappings are described in "c/eus.h". *msg1* and *msg1* are messages displayed to the user. *form* is the S-expression which caused the error.

Signal handlers should be programmed as functions receiving two arguments: *sig* and *code*. *Sig* is the signal number ranging from 1 to 31, and *code* is the minor signal code defined in signal-number dependent manners.

^D (*end-of-file*) at the top-level terminates eus session. This is useful when eus is programmed as a filter.

Eval-dynamic is the function to find the dynamic value bound to a symbol used as a let or lambda variable. This is useful for debugging.

identity *obj* [function]

returns obj itself. Note the difference between **identity** and **quote**. **identity** is a function whereas **quote** is a special form. Therefore, `(identity 'abc)` is evaluated to `abc` and `(quote 'abc) == (quote (quote abc))` is evaluated to `'abc`. **Identity** is often used as the default value for `:key` parameters of many generic sequence functions.

eval *form &optional environment* [function]

evaluates *form* and returns its value. Hook function can be called before entering the evaluation, if ***evalhook*** is set to some function that accept form and environment.

apply *func &rest args* [function]

func is applied to *args*. *Func* must be evaluated to be a function symbol (a symbol which has a function definition), a lambda form, or a closure. Macros and special forms cannot be applied. The last element of *args* must be a list of arguments while other args should be bare arguments. Thus, if the last *args* is NIL, then apply is almost equivalent to funcall, except that apply has one more arguments than funcall. `(apply #'max 2 5 3 '(8 2)) --> 8`.

funcall *func &rest args* [function]

applies *func* to *args*. The number of *args* must coincide to the number of arguments the *func* requests.

quote *obj* [special]

evaluates to *obj* itself.

function *func* [special]

makes a function closure. If *func* is a symbol, its function definition is retrieved.

evalhook *hookfunc form &optional env* [function]

evaluates *form* once after binding *hookfunc* to ***evalhook***.

eval-dynamic *variable* [function]

finds the value of *variable* (symbol) on the stack.

macroexpand *form* [function]

expands *form* if it is a macro call. If *form* is expanded to a macro call again, expansion is repeated until non macro call results.

eval-when *situation &rest forms* [special]

Situation is a list of `compile, load and eval`. Forms are evaluated when the current execution mode matches with situation. **eval-when** is important to control the behavior and environment of the compiler. If `compile` is specified, *forms* are evaluated by the compiler so that the result will affect the consequent compilation. For example, *defmacro* should be evaluated by the compiler in order to let the compiler expand macro calls at compile time. If `load` is given in the *situation* list, *forms* are compiled to be loaded (evaluated) at load time, i.e., compiled functions are defined at load time. This is the normal effect that we expect to the compiler. `load` situation is used to control the compiler's environment. If `eval` is included in situation list, *forms* are evaluated when their source code is loaded.

the *type form* [special]

Declares *form* is of *type*. *type* is either a class object, :integer, :fixnum, or :float.

declare *&rest declarations* [special]

Each *declaration* is a list of a declaration specifier and an integer or target symbols. Declarations are important to let the compiler produce faster code.

special declares special variables

type declares the type of variables; `(type integer count)`; valid type specifiers are *integer*, *:integer fixnum*, *:float* and *float*. The **type** keyword may be omitted if type specifier is either one listed here. So `(integer count)` is a correct declaration. Other types (classes) such as *float-vector*, *integer-vector*, etc. need to be preceded by **type**, as `(type float-vector vec1)`.

ftype declares the result type of functions

optimize set *optimize* parameter (0–3) of the compiler

safety set *safety* parameter (0–3) of the compiler

space set *space* parameter (0–3) of the compiler

inline not recognized

not-inline not recognized

proclaim *proclamation* [function]

globally declares the types of variables and compiler options. The same declarations are accepted as described for **declare** special form. However, **proclaim** is a function of one argument and proclamation is evaluated.

warn *format-string &rest args* [function]

prints warning-message given as *format-string* and *args* to *error-output*.

error *format-string &rest args* [function]

calls the current error-handler function bound to ***error-handler***. The default error-handler 'euserror' first prints arguments to ***error-output*** using **format**, then enters a new top level session. The prompt shows you the depth of your error session. **Throw**ing to the number, you can go back to the lower level error session.

In the multithread EusLisp, special variables are shared among threads and the same ***error-handler*** is referenced by different threads. To avoid this inconvenience, multithread EusLisp provides the **install-error-handler** function which installs different error handler for each thread.

lisp::install-error-handler *handler* [function]

installs the *handler* as the error handler of the current thread.

10.2 Top-level Interaction

EusLisp's standard top-level read-eval-print loop is controlled by **eustop**. When EusLisp is invoked, **eustop** tries to load the file named `".eusrc"` in your home directory or the file specified by the `EUSRC` environment variable. It also tries to load a file named `".eusrc"` in the current working directory. So, if you are in your home directory, note that .eusrc is loaded twice. Then EusLisp loads files specified in its argument list. After these loading, **eustop** enters normal interactive session.

When ***standard-input*** is connected to user's tty, **eustop** prints prompt generated by the **toplevel-prompt** function. The default toplevel-prompt prints `"eus$ "`. The effect of changing the definition of toplevel-prompt appears when eustop is invoked next time. One way to change the prompt from the first time is to define toplevel-prompt in your `.eusrc` file.

Inputs are read from ***terminal-io*** stream. If the input is parenthesized, it is taken as a lisp form and is evaluated by **eval**. Else if the first symbol of the input line has function definition, the line is automatically parenthesized and evaluated. If no function definition is found, then its special value is examined and the value is printed. If the symbol is unbound, the line is regarded as UNIX command and passed to sh (Bourn's shell). If sh cannot find corresponding unix command, "command unrecognized" message is printed. Thus, **eustop** works both as a lisp interpreter and as a unix shell. If you do not wish to execute the input as UNIX command, you may escape the form by preceeding a comma ',' at the begining of the line. This is also useful to see the dynamic value binding when an error occurred in the interpretive execution. Since EusLisp adopts lexical scope, we cannot examine the value of local variables outside of the scope unless they are declared special.

If the environment variable, **USE_TOP_SELECTOR**, is defined, the toplevel input is read in an asynchronous manner using the `select` library call. The input stream (`*standard-input*`) is registered to the `*top-selector*`, which is an instance of the **port-selector** class, together with the read-eval-print function (`repsel`) Therefore arrival of key inputs invokes the evaluation of the `repsel`. This feature is particularly useful when EusLisp is to handle multiple events, i.e., key inputs, X window events, and socket connection requests, at the same time. In order to exploit this asynchronous toplevel interaction, users should never write a code that blocks at the `read` operation. Instead, the input stream should be registered to the `*top-selector*` with its handler function by using the `:add-port` method. The handler function is expected to read from the stream, which is already known ready to return the input without blocking.

Note that Xwindow event handlers are defined to use the `*top-selector*` implicitly when `USE_TOP_SELECTOR` is defined, and user programs do not have to call `x:window-main-loop` at all to catch X events.

Using the time-out of the select call, users may define a timer handler. Each time the select call times out, the function bound to `*timer-job*` is invoked with no argument. The timer interval is defined by *top-selector-interval*, which is defaulted to 10.0 second. Note that the timer function invokation is not precisely periodic when there are inputs to the *top-selector*.

In the toplevel interaction, each line input is remembered in ***history*** vector with a sequence number. You can recall a specific input by **!** function as if you were in csh. The difference from csh's history is, you need at least one white space between the exclamation mark and the sequence number since **!** is a function.

^D (EOF) terminates EusLisp normally. To return abnormal termination code to upper level (usually a csh), use **exit** with an appropriate condition code.

eustop sets a signal handler only for SIGINT and SIGPIPE, and other signals are not caught. Thus, signals such as SIGTERM or SIGQUIT cause EusLisp to terminate. In order to catch these signals to avoid termination, use **unix:signal** function to set user-defined signal handlers.

- [variable]

current input.

+ [variable]

previous input.

++ [variable]

old input.

+++ [variable]

ancient input.

* [variable]

previous result.

** [variable]

old result.

*** [variable]

ancient result.

prompt-string [variable]

prompt string used by **eustop**.

program-name [variable]

the command that invoked this EusLisp, possibly eus, eusx, eusxview or user-saved euslisp.

eustop *&rest argv* [function]

is the default toplevel loop.

eussig *sig code* [function]

is the default signal hander for SIGPIPE. **eussig** prints signal number upon its arrival and enters another toplevel loop.

sigint-handler *sig code* [function]

is the default signal handler for SIGINT (control-C). It enters a new top level session.

euserror *code message &rest arg* [function]

the default error handler that prints *message* and enters a new error session.

reset [function]

quits error loop and goes back to the outermost eustop session.

exit *&optional termination-code* [function]

terminates EusLisp process and returns *termination-code* (0..255) as the process status code (0..255).

top-selector [variable]

The port-selector object to handle asynchronous function invocation according to inputs from multiple streams.

h [function]

prints all the inputs remembered in ***history*** vector with associated sequence numbers.

! *&optional (seq 0)* [function]

recalls the input line associated with the sequence number *seq*. When *seq* is 0, the most recent command is recalled, and if *seq* is negative, the line is specified relatively to the current input. The recalled line is printed and the cursor is located at the end of the line. You can go backward by control-H (backspace) or control-B, go forward by control-F or control-K, go to the beginning of line by control-A, to the end of line by control-L. control-C cancels the line editing. control-M (carriage-return) or control-J (line-feed) finishes editing the line and starts evaluation of the edited line. If *seq* is not a number and is a symbol or a string, the history list is searched toward old input, and a command line which include the symbol or a string as a substring is returned.

new-history *depth* [function]

initializes ***history*** vector to have *depth* length. *Depth* input lines are remembered. All the input lines recorded in the current ***history*** are discarded.

10.3 Compilation

EusLisp compiler is used to speed the execution of Lisp programs. You can expect 5 to 30 times faster execution and notable reduction of garbage collection time elapsed by macro expansion.

Euscomp does optimization for arithmetic operation and vector access. Sometimes proper type declarations are needed to inform the compiler applicability of optimization.

Compile-function compiles functions one by one. **Compile-file** compiles an entire source file. During the execution of **Compile-file**, each form in a file is read and evaluated. This may change the current EusLisp environment. For examples, **defparameter** may set a new value to a symbol and **defun** may substitute the existing compiled function with its non-compiled version. To avoid these unexpected effects, use the **eval-when** special form without compile time situation, or use **euscomp** command to run the compiler as a separate process.

Euscomp is a unix command, which is usually a symbolic link to **eus**. It recognizes several options. -O flag indicates optimization of the C compiler. Each of -O1,-O2, -O3 indicates optimization level of EusLisp compiler, which is equivalent to proclaiming (optimize 1 or 2 or 3). Each of -S0, -S1, -S2, -S3 set 0,1,2 and 3 to compiler:*safety*. If *safety* is less than 2, no code for checking interrupt is emitted, and you will lose control if the program enters an infinite loop. If *safety* is zero, the number of required arguments is not checked. -V flag is used to print function names when they are compiled (verbose). -c flag prevents from forking and exec'ing cc. -D pushes next argument to the ***features*** list, which can be used for conditional compilation in conjunction with #- and #+ read-macro.

The compiler translates EusLisp source program named as "xxx.l" into the intermediate C program file named "xxx.c" and the header file named "xxx.h". Then the C compiler is run and "xxx.o" is generated. Intermediate files "xxx.c" and "xxx.h" are left for the purpose of cross compilation: usually you only need to compile "xxx.c" files by cc unix command when you wish to use the code on machines of different architecture. Compiled code is loaded to EusLisp by '(load "xxx")'.

Each intermediate file refers to the "eus.h" header file, which is supposed to be located in the `*eusdir*/c` directory. `*eusdir*` is copied from the `EUSDIR` environment variable. If none is set, `/usr/local/eus/` is taken as the default directory.

When compiled, intermediate C programs are usually much bigger than the original source code. For example, 1,161 lines of "l/common.l" lisp source expands to 8,194 lines of "l/common.c" and 544 lines of "l/common.h". Compiling 1,000 lines of lisp source is not a hard task, but optimized compililation of nearly 10,000 lines of C program not only takes long time (several minutes), but also consumes much disk space. So if you are compiling relatively big programs, be sure your machine has sufficient /var/tmp disk, otherwise CC may die. Setting the `TEMPDIR` environment variable to a bigger disk slice may help.

As the linkage is performed at load-time or at run-time, no recompilation is required even the eus kernel is updated. On the other hand, run-time linkage may impose you another inconvenience. Suppose you have two functions A and B in a file "x.l" and A calls B. After compiling "x.l", you load "x.o" and try to call A which internally calles B. Then you find a bug in B, and probably you would redefine B. Here, you have compiled A and non-compiled B. You may call A again, but nothing will change, since A still calls old compiled B which is rigidly linked when A first called B. To avoid this problem, A must be redefined again, or B must be redefined just after "x.o" is loaded and before A is called.

When a compiled-code is loaded, its top level code, which is normally a series of defun, defmethod, etc., is executed. This top level code is defined as the entry function of the load module. The compiler names the entry function, and the loader has to know the exact name of this function. To make the situation simple, both the compiler and the loader assume the entry function name is identical to the basename of the object file. For example, if you compile and load "fib.l", the compiler produces "fib(...)" as the entry function of "fib.c", and the loader looks for "fib" in the "fib.o" object file. Since the final object file is produced by "cc" and "ld" of unix, this entry function name has to satisfy the naming rule of C functions. Therefore, you have to avoid C's reserved keywords such as "int", "struct", "union", "register", "extern", etc., or the private identifiers defined in "c/eus.h" such as "pointer", "cons", "makeint", etc., to be used as the name of the file. If you have to use one of these reserved words as the name of the source file, you specify it as *:entry* arguments of the compiler and the loader.

A restriction exists for the usage of closure: **return-from** special form in closures and clean-up forms in unwind-protect is not always correctly compiled.

Disassemble is not implemented. In order to analyze compiled code, see the intermediate C program or use `adb`.

euscomp *&rest filename* [unix-command]

Invokes EusLisp compiler.

compile-file [function]

srcfile &key (verbose nil)
(optimize 2) (c-optimize 1) (safety 1) ; optimization level
(pic t) ; position independent code
(cc t) ; run c compiler
(entry (pathname-name file))

compiles a file. ".l" is assumed for the suffix of the *srcfile*. If *:verbose* is T, names of functions and methods being compiled are printed to make it easy to find the expressions where errors occurred. *:Optimize, :c-optimize* and *:safety* specifies the optimization levels. *:Pic* should be set T, unless the module is hard-linked in the EusLisp core during the make stage.

compile *funcname* [function]

compiles a function. **Compile** first prints the function definition into a temporary file. The file is compiled by **compile-file** and then is loaded by **load**. Temporary files are deleted.

compile-file-if-src-newer *srcfile &key compiler-options* [function]

compiles the *srcfile* if it is newer (more recently modified) than its corresponding object file. The object file is supposed to have the ".o" suffix.

compiler:*optimize* [variable]

controls optimization level.

compiler:*verbose* [variable]

When set to non-nil, the name of a function or a method being compiled, and the time required for the compilation are displayed.

compiler:*safety* [variable]

controls safety level.

10.4 Program Loading

load [function]

fname &key (verbose *load-verbose*)
(package *package*)
(entry (pathname-name fname))
(symbol-input nil)
(symbol-output "")
(print nil)
(ld-option "")

Load is the function to read either a source file or a compiled object file into the EusLisp process. If the file specified by *fname* exists, it is loaded. Whether the file is source or binary is automatically checked by seeing its magic number. If the file does not exist but a file with the file type '.o' exists, the file is loaded as an object file. Else if a file with the '.l' suffix is found, it is loaded as a source program. Therefore, there is a case where you specified "foo.so" expecting "foo.l" is already compiled, but "foo.l" is actually loaded, since it has not yet been compiled in reality. In other words, if you just specify a base-name of a file, its compiled version is first tried to be loaded, and the source file suffixed by ".l" is tried later. If the file name is not specified in the absolute path by prefixing the name with a slash "/", **load** searches for the file in the directories specified by the ***load-path*** global variable.

For example, if `*load-path*` is `("/user/eus/" "/usr/lisp/")`, and `"llib/math"` is given as *fname*, **load** tries to find `"/user/eus/llib/math.o", "/usr/lisp/llib/math.o", "/user/eus/llib/math.l"` and `"/usr/lisp/llib/math.l"` in this order. If no appropriate file could be found, an error is reported.

:entry option specifies the entry address to initialize the load module. For example, `:entry "_myfunc"` option means that the execution begins at `_myfunc`. Default entry is the basename of the file loaded as described in the section **??**. Library module names can be specified in **:ld-option** option string. For example, in order to link a module which uses suncore libraries, **:ld-option** `"-lsuncore -lsunwindow -lpixrect -lm -lc"` should be given. On non Solaris systems, ld runs twice when libraries are included; once to determine the size of the linked module, and again to link them actually with a proper memory allocation.

:symbol-input and **:symbol-output** options are used to solve references from one object module to another or to avoid duplicated loading of libraries. Suppose you have two object modules A and B which has reference to symbols defined in A. You first load the module A specifying `:symbol-output = "a.out"`. Symbol information generated by this linking is written to `a.out`. In order to load the module B, you have to specify `:symbol-input = "a.out"` to solve the references from B to A.

On Solaris2 OS, the loading of compiled code is done by calling *dlopen* in the dynamic loader library. Application of *dlopen* is restricted to shared objects which are compiled position independently with "-K pic" option. Also, since *dlopen* cannot open the same file twice, load first does *dlclose* on the file already loaded.
:print option decides whether load should produce output to *standard-output* for each input expression. This option is provided to find which expression (usually defun, defmethod, etc.) results error in loading.

load-files *&rest files* [function]

loads *files* successively with setting `:verbose` to T.

modules [variable]

holds a list of names of the modules that have been loaded so far.

provide *module-name &rest version-info* [function]

adds the concatenation of *module-name* and *version-info* to `*modules*` as the name of the module being loaded. *module-name* should be a symbol or a string. Calls to **require** should appear at the beginning of files that compose a complete modules.

require *module-name &rest load-arg* [function]

loads file given by *module-name* or the first argument or *load-arg* unless *module-name* is found in `*modules*`. **provide** and **require** control dependency among modules and are used to avoid duplicated loading of basic modules. Suppose you have one basic module named "A" and two application modules named "B" and "C" which are independent from each other but rely on "A" module. At the beginning of each file, module name is declared by **provide**. Since "A" module does not depend on any other modules it does not **require** anything. `(require "A" "a.o")` follows calls to **provide** in "B" and "C". If you load "B" (more precisely, "b.o"), "a.o" is also loaded since it is found in `*modules*` and two module names "A" and "B" are added to `*modules*`. Then if you load "C", "A" module is not loaded and "C" is added to `*modules*`.

system:binload [function]

opath qpath &optional (entry (pathname-name opath))
(symfile "/usr/local/bin/eus")
(symout "a.out")
(ldopt "")

link-load a binary file.

system::txtload *fname* [function]

10.5 Debugging Aid

describe *obj &optional (stream *standard-output*)* [function]

Describe prints the contents of an object slot by slot.

describe-list *list &optional (stream *standard-output*)* [function]

describes each element in *list*.

inspect *obj* [macro]

Inspect is the interactive version of **describe**. It accepts subcommands to print each slot of an object, to go deeper into a slot, or set a new value to a slot, etc. Use '?' command to see the subcommand menu.

more *&rest forms* [function]

After evaluating forms with the binding of *standard-output* to a temporary file, the temporary file is output to *standard-output* with Unix's 'more' command. **More** is useful to see a long output generated by functions like **describe**.

break *&optional (prompt ":: ")* [function]

Enters a break loop. Since the current binding context is in effect, local variables can be seen by prefixing "," to an input. To end break, type control-D.

help *topic* [function]

Help prints the brief description on the topic which is usually a function symbol. The help description has been created from the reference manual (this document). The environment variable `LANG` is referrenced to determine one of two reference manuals, Japanese or English. If `LANG` is constituted either with `"ja", "JA", "jp"`, or `"JP"`, Japanese is selected. Otherwise, English. This determination is made when EusLisp start up. The actual reading of the help document is made at the first time when the 'help' is invoked to save memory if unnecessary.

apropos *strng &optional pack* [function]

Apropos is useful when you forget the exact name of a function or a variable and you only know its partial or ambiguous name. It prints all the symbols whose symbol-names include the *strng* as a substring. If *pack* is provided, only prints symbols that belong to this package instead. Case insensitive.

apropos-list *strng &optional pack* [function]

is similar to **apropos** but does no printing and returns the result as a list.

constants *&optional (string "") (pkg *package*)* [function]

lists every symbol in pkg which has defined constant and matches with *string*.

variables *&optional (string "") (pkg *package*)* [function]

lists every symbol in pkg which has global value assigned and matches with *string*.

functions *&optional (string "") (pkg *package*)* [function]

lists every symbol in pkg which has global function defined and matches with *string*.

btrace *&optional (depth 10)* [function]

prints call history of *depth* levels.

step-hook *form env* [function]

step *form* [function]

Step and **trace** work correctly only for functions, and not for macro or special forms.

trace *&rest functions* [function]

begins tracing of *functions*. Each time functions are called, their arguments and results are prited.

untrace *&rest functions* [function]

stops tracing.

timing *count &rest forms* [macro]

executes *forms* count times, and calculates time required for one execution of forms.

time *function* [macro]

begins measurement of time elapsed by *function*.

sys:list-all-catchers [function]

returns a list of all **catch** tags.

sys:list-all-instances *aclass &optional scan-sub* [function]

scans in the overall heap, and collects all the instances of the specified class. If *scan-sub* is NIL, then instances of exactly the *aclass* are listed, otherwise, instances of *aclass* or its subclasses are collected.

sys:list-all-bindings [function]

scans bind stack, and returns a list of all the accessible value bindings.

sys:list-all-special-bindings [function]

scans the stack and list up all value bindings.

10.6 Dump Objects

EusLisp's reader and printer are designed so that they can write any objects out to files in the forms that are rereadable. The objects may have mutual or recursive references. This feature is enabled when `*print-circle*` and `*print-object*` are set to T. Following functions set these variables to T, open a file, and print objects. The most important application of these functions is to dump the structures of 3D models that have mutual references.

dump-object *file &rest objects* [function]

dump-structure *file &rest objects* [function]

dumps objects to *file* in a format as they can be read back again.

dump-loadable-structure *file &rest symbols* [function]

dumps objects bound to symbols to *file*. The file can be read back again by simply loading it.

```
(setq a (make-cube 1 2 3))

;; sample for dump-object
(dump-object "a-cube.l" a)
(with-open-file
  (f "a-cube.l" :direction :input)
  (setq a (read f)))
(print a)

;; sample for dump-structure
(dump-structure "a-cube.l" a)
(with-open-file
  (f "a-cube.l" :direction :input)
  (setq a (read f)))
(print a)

;; sample for dump-loadable-structure
(dump-loadable-structure "a-cube.l" a)
(load "a-cube.l")
(print a)
```

10.7 Process Image Saving

This process image saving is no longer supported on Solaris2 based EusLisp, since it heavily depends on Solaris's dynamic loading facility which loads shared objects position-independently above the `sbrk` point.

sys:save *path &optional (symbol-file "") starter* [function]

Save dumps the current EusLisp process environment to a file which can be invoked as a Unix command later. If a function name is specified for *starter*, the function is evaluated when the command begins execution. Each command line argument is coerced to string in EusLisp and they are passed to *starter* as its arguments, so that it can parse the command line. Be sure that you have closed all the streams except ***standard-input*** and ***standard-output***. File open states cannot be saved. Also, be sure you have not attempted **mmap**, which unnoticeably happens when you make internetwork socket-stream. Sun's network library always memory-maps NIS information such as host-by-name table and locates them at the uppermost available location of a process that cannot be saved. When the saved image is run later, any access to the network library fails and causes core dump. Note that Xwindow also uses this library, thus you cannot save your process image once you opened connection to Xserver.

10.8 Customization of Toplevel

When EusLisp is invoked from Unix, execution is initiated by the toplevel function bound to ***toplevel***. This function is `eustop` in `eus` and `xtop` in `eusx`. You can change this toplevel function by specifying your own function to the third argument to **save**.

The toplevel function should be programmed to accept arbitrary number of arguments. Each argument on the command line is coerced to a string and transfered to the toplevel function. The program below repeatedly reads expressions from the file given by the first argument and pretty-prints them to the second argument file.

```
(defun pprint-copy (infile outfile)
  (with-open-file (in infile)
     (with-open-file (out outfile :direction :output)
        (let ((eof (cons nil nil)) (exp))
           (while (not (eq (setq exp (read in nil eof)) eof))
              (pprint exp out))))))
(defun pprint-copy-top (&rest argv)
   (when (= (length argv) 2)
      (pprint-copy (first argv) (second argv))))
```

Once you defined these functions in EusLisp, `(save "ppcopy" "" 'pprint-copy-top)` creates a unix executable command named `ppcopy`.

In Solaris based EusLisp, the toplevel evaluator cannot change in this manner, since **save** is not available. Instead, edit `lib/eusrt.l` to define the custom toplevel evaluator and set it to ***toplevel***. `lib/eusrt.l` defines initialization procedures evaluated at every invocation of the EusLisp.

10.9 Miscelaneous Functions

lisp-implementation-type [function]

returns `"EusLisp"`.

lisp-implementation-version [function]

returns the name, the version and the make-date of this EusLisp. This string is also printed at the opening of a session. `"MT-EusLisp 7.50 X 1.2 for Solaris Sat Jan 7 11:13:28 1995"`

第II部
EusLisp Extension

11 System Functions

11.1 Memory Management

The design of memory management scheme affects much to the flexibility and efficiency of object-oriented languages. EusLisp allocates memory to any sort of objects in a unified manner based on the *Fibonacci buddy method*. In this method, each of large memory pools called chunks is split into small cells which are unequally sized but aligned at Fibonacci numbers. A memory chunk is a homogeneous data container for any types of objects such as `symbol, cons, string, float-vector`, etc. as long as their sizes fit in the chunk. A chunk has no special attributes, like static, dynamic, relocatable, alternate, etc. EusLisp's heap memory is the collection of chunks, and the heap can extend dynamically by getting new chunks from UNIX. The expansion occurs either automatically on the fly or on user's explicit demand by calling **system:alloc** function. When it is managed automatically, free memory size is kept about 25% of total heap size. This ratio can be changed by setting a value between 0.1 and 0.9 to the **sys:*gc-margin*** parameter.

When all the heap memory is exhausted, mark-and-sweep type garbage collection runs. Cells accessible from the root (packages, classes and stacks) remain at the same place where they were. Other inaccessible cells are reclaimed and linked to the free-lists. No copying or compactification occurs during GC. When a garbage cell is reclaimed, its neighbor is examined whether it is also free, and they are merged together to form a larger cell if possible. This merging, however, is sometimes meaningless, since `cons`, which is the most frequently called memory allocator, requests the merged cell to be divided to the smallest cell. Therefore, EusLisp allows to leave a particular amount of heap unmerged to speed up cons. This ratio is determined by **sys:*gc-merge*** parameter, which is set to 0.3 by default. With the larger **sys:*gc-merge***, the greater portion of heap is left unmerged. This improves the performance of consing, since buddy-cell splitting rarely occurs when conses are requested. This is also true for every allocation of relatively small cells, like three dimensional float-vectors.

SYS:GC invokes garbage collector explicitly, returning a list of two integers, numbers of free words and total words (not bytes) allocated in the heap. **SYS:*GC-HOOK*** is a variable to hold a function that is called upon the completion of a GC. The hook function should receive two arguments representing the sizes of the free heap and the total heap.

If "fatal error: stack overflow" is reported during execution, and you are convinced that the error is not caused by a infinite loop or recursion, you can expand the size of the Lisp stack by **sys:newstack**. **reset** should be performed before **sys:newstack**, since it discards everything in the current stack such as special bindings and clean-up forms of *unwind-protect*. After a new stack is allocated, execution starts over from the point of printing the opening message. The default stack size is 65Kword. The Lisp stack is different from the system stack. The former is allocated in the heap, while the latter is allocated in the stack segment by the operating system. If you get "segmentation fault" error, it might be caused by the shortage of the system stack. You can increase the system stack size by the `limit` csh command.

Sys:reclaim and **sys:reclaim-tree** function put cells occupied by objects back to the memory manager, so that they can be reused later without invoking garbage collection. You must be assured that there remains no reference to the cell.

memory-report and **room** function display statistics on memory usage sorted by cell sizes and classes respectively.

address returns the byte address of the object and is useful as a hash function when used with hash-table, since this address is unique in the process.

Peek and **poke** are the functions to read/write data directly from/to a memory location. The type of access should be either of **:char, :byte, :short, :long, :integer, :float** and **:double**. For an instance, `(SYS:PEEK (+ 2 (SYS:ADDRESS '(a b))) :short)` returns class id of a cons cell, normally 1.

There are several functions prefixed with 'list-all-'. These functions returns the list of a system resource or environment, and are useful for dynamic debugging.

sys:gc [function]

starts garbage collection, and returns a list of the numbers of free words and total words allocated.

sys:*gc-hook* [variable]

Defines a function that is called upon the completion of a GC.

sys:gctime [function]

returns a list of three integers: the count of gc invoked, the time elapsed for marking cells (in 1/60 sec. unit), and the time elapsed for reclamation (unmarking and merging).

sys:alloc *size* [function]

allocates at least *size* words of memory in the heap, and returns the number of words really allocated.

sys:newstack *size* [function]

relinquishes the current stack, and allocates a new stack of *size* words.

sys:*gc-merge* [variable]

is a memory management parameter. ***gc-merge*** is the ratio the ratio of heap memory which is left unmerged at GC. This unmerged area will soon filled with smallest cells whose size is the same as a cons. The default value is 0.3. The larger values, like 0.4, which specifies 40% of free heap should be unmerged, favors for consing but do harm to instantiating bigger cells like float-vectors, edges, faces, etc.

sys:*gc-margin* [variable]

is a memory management parameter. ***gc-margin** determines the ratio of free heap size versus the total heap. Memory is acquired from UNIX so that the free space does not go below this ratio. The default value 0.25 means that 25% of free space is maintained at every GC.

sys:reclaim *object* [function]

relinquishes *object* as a garbage. It must be guaranteed that it is no longer referenced from any other objects.

sys:reclaim-tree *object* [function]

reclaims all the objects except symbols traversable from *object*.

sys::bktrace *num* [function]

prints the back-trace information of *num* depth on the Lisp stack.

sys:memory-report *&optional strm* [function]

prints a table of memory usage report sorted by cell sizes to the *strm* stream.

sys:room *output-stream* [function]

outputs memory allocation information ordered by classes.

sys:address *object* [function]

returns the address of *object* in the process memory space.

sys:peek *[vector] address type* [function]

reads data at the memory location specified by *address* and returns it as an integer. *type* is one of **:char, :byte, :short, :long, :integer, :float**, and **:double**. If no *vector* is given, the address is taken in the unix's process space. For example, since the a.out header is located at `#x2000` on SunOS4, `(sys:peek #x2000 :short)` returns the magic number (usually `#o403`). Solaris2 locates the ELF header at `#10000`, and `(sys:peek #x10000 :long)` returns `#xff454c46` whose string representation is "ELF".

If *vector*, which can be a foreign-string, is specified, address is recognized as an offset from the vector's origin. `(sys:peek "123456" 2 :short)` returns short word representation of "34", namely #x3334 (13108).

Be careful about the address alignment: reading short, integer, long, float, double word at odd address may cause bus error by most CPU architectures.

sys:poke *value [vector] address value-type* [function]

writes *value* at the location specified by *address*. Special care should be taken since you can write to anywhere in the process memory space. Writing to outside the process space surely causes segmentation fault. Writing short, integer, long, float, double word at odd address causes bus error.

sys:list-all-chunks [function]

list up all allocated heap chunks. Not useful for other than the implementor.

sys:object-size *obj* [function]

counts the number of cells and words accessible from *obj*. All the objects reference-able from obj are traversed, and a list of three numbers is returned: the number of cells, the number of words logically allocated to these objects (i.e. accessible from users), and the number of words physically allocated including headers and extra slots for memory management. Traversing stops at symbols, i.e. objects referenced from a symbol such as property-list or print-name string are not counted.

11.2 Unix System Calls

EusLisp assorts functions which directly correspond to the system calls and library functions of UNIX operating system. For further detail of these functions, consult UNIX system interface reference (2). These low-level functions defined in `*unix-package*` are sometimes dangerous. Use higher level functions defined in other packages if possible. For example, use IPC facilities described in the section ?? instead of `unix:socket, unix:bind, unix:connect`, and so on.

11.2.1 Times

unix:ptimes [function]

a list of five elements, elapsed time, system time, user time, subprocess's system time, subprocess's user time, is returned. Unit is always one sixtieth second. This function is obsolete and use of **unix:getrusage** is recommended.

unix:runtime [function]

Sum of the process's system and user time is returned. Unit is 1/60 second.

unix:localtime [function]

Current time and date is returned in an integer vector. Elements are second, minute, hour, day-of-a-month, month (zero-based), year (the number of years since 1900), weekday (the number of days since Sunday, in the range 0 to 6), day-in-the-year (the number of days since January 1, in the range 0 to 365), daylight-saving-time-is-set (a flag that indicates whether daylight saving time is in effect at the time described) and supported-time-zone.

```
ex.)  unix:localtime => #(10 27 12 8 10 116 2 312 nil (``JST'' ``JST''))
```

unix:asctime *tm_intvector* [function]

Converts localtime represented with an integer-vector into a string notation.
`(unix:asctime (unix:localtime))` returns a string representation of the current real time.

11.2.2 Process

unix:getpid [function]

returns the process id (16bit integer) of this process.

unix:getppid [function]

returns the process id of the parent process.

unix:getpgrp [function]

returns the process group id.

unix:setpgrp [function]

sets a new process group id.

unix:getuid [function]

gets user id of this process.

unix:geteuid [function]

returns the effective user id of this process.

unix:getgid [function]

returns the group id of this process.

unix:getegid [function]

returns the effective group id of this process.

unix:setuid *integer* [function]

sets effective user id of this process.

unix:setgid *integer* [function]

sets the effective group id of this process.

unix:fork [function]

creates another EusLisp. 0 is returned to the subprocess and the pid of the forked process is returned to the parent process. Use **system:piped-fork** described in section **??** to make a process connected via pipes.

unix:vfork [function]

forks another EusLisp, and suspends the parent process from execution until the new EusLisp process terminates.

unix:exec *path* [function]

replaces executing EusLisp with another program.

unix:wait [function]

waits for the completion of one of subprocesses.

unix::exit *code* [function]

terminates execution and returns *code* as its completion status. Zero means normal termination.

sys:*exit-hook* [variable]

Defines a function that is called just before the process is exited.

unix:getpriority *which who* [function]

returns the highest priority (nice value) enjoyed by this process. *Which* is one of 0(process), 1(process-group) or 2(user).

unix:setpriority *which who priority* [function]

sets priority of the resource determined by *which* and *who*. *which* is one of 0(process), 1(process-group) or 2(user). *who* is interpreted relative to *which* (a process identifier for *which* = 0, process group identifier for *which* = 1, and a user ID for *which* = 2. A zero value of who denotes the current process, process group, or user. To lower the priority (nice value) of your EusLisp process, `(unix:setpriority 0 0 10)` will sets the nice value to 10. Bigger nice value makes your process get less favored.

unix:getrusage *who* [function]

returns list of system resource usage information about *who* process. Elements are ordered as follows: More comprehensive display is obtained by **lisp:rusage**.

```
float ru_utime (sec.) /* user time used */
float ru_stime (sec.) /* system time used */
```

```
int  ru_maxrss;       /* maximum resident set size */
int  ru_ixrss;        /* currently 0 */
int  ru_idrss;        /* integral resident set size */
int  ru_isrss;        /* currently 0 */
int  ru_minflt;       /* page faults without physical I/O */
int  ru_majflt;       /* page faults with physical I/O */
int  ru_nswap;        /* number of swaps */
int  ru_inblock;      /* block input operations */
int  ru_oublock;      /* block output operations */
int  ru_msgsnd;       /* messages sent */
int  ru_msgrcv;       /* messages received */
int  ru_nsignals;     /* signals received */
int  ru_nvcsw;        /* voluntary context switches */
int  ru_nivcsw;       /* involuntary context switches */
```

unix:system *&optional command* [function]

executes *command* in a sub shell. *command* must be recognizable by Bourn-shell.

unix:getenv *env-var* [function]

gets the value for the environment variable *env-var*.

unix:putenv *env* [function]

adds *env* in the process's environment variable list. *env* is a string which equates var to value like `"VARIABLE=value"`.

unix:sleep *time* [function]

suspends execution of this process for *time* seconds.

unix:usleep *time* [function]

suspends execution of this process for *time* micro-seconds (**u** represents micro). **Usleep** is not available on Solaris2 or other Sys5 based systems.

11.2.3 File Systems and I/O

unix:uread *stream &optional buffer size* [function]

reads *size* bytes from *stream*. *stream* may either be a stream object or an integer representing fd. If *buffer* is given, the input is stored there. Otherwise, input goes to the buffer-string in *stream*. Therefore, if *stream* is fd, *buffer* must be given. **unix:uread** never allocates a new string buffer. **unix:uread** returns the byte count actually read.

unix:write *stream string &optional size* [function]

writes *size* bytes of *string* to *stream*. If *size* is omitted, the full length of *string* is output.

unix:fcntl *stream command argument* [function]

unix:ioctl *stream command buffer* [function]

unix:ioctl_ *stream command1 command2* [function]

unix:ioctl_R *stream command1 command2 buffer &optional size* [function]

unix:ioctl_W *stream command1 command2 buffer &optional size* [function]

unix:ioctl_WR *stream command1 command2 buffer &optional size* [function]

unix:uclose *fd* [function]

close a file specifying its file descriptor *fd*.

unix:dup *fd* [function]

returns the duplicated file descriptor for *fd*.

unix:pipe [function]

creates a pipe. An io-stream for this pipe is returned.

unix:lseek *stream position &optional (whence 0)* [function]

sets the file pointer for *stream* at *position* counted from *whence*.

unix:link *path1 path2* [function]

makes a hard link.

unix:unlink *path* [function]

removes a hard link to the file specified by *path*. If no reference to the file lefts, it is deleted.

unix:mknod *path mode* [function]

makes inode in a file system. *path* must be a string, not a pathname object.

unix:mkdir *path mode* [function]

makes directory in a file system. *path* must be a string, not a pathname object.

unix:access *path mode* [function]

checks the access rights to *path*.

unix:stat *path* [function]

gets inode information of *path* and returns a list of integers described below.

```
st_ctime ; file last status change time
st_mtime ; file last modify time
st_atime ; file last access time
st_size ; total size of file, in bytes
st_gid ; group ID of owne
st_uid ; user ID of owner
st_nlink ; number of hard links to the file
st_rdev ; the device identifier (special files only)
st_dev ; device file resides on
st_ino ; the file serial number
st_mode ; file mode
```

unix:chdir *path* [function]

changes the current working directory to *path*.

unix:getwd [function]

gets current working directory.

unix:chmod *path integer* [function] changes access mode (permission) for *path*.

unix:chown *path integer* [function]

changes the owner of the file *path*.

unix:isatty *stream-or-fd* [function]

returns T if *stream-or-fd* is connected to a tty-type device (a serial port or a pseudo tty) .

unix:msgget *key mode* [function]

creates or allocates a message queue which is addressed by *key*.

unix:msgsnd *qid buf &optional msize mtype flag* [function]

unix:msgrcv *qid buf &optional mtype flag* [function]

unix:socket *domain type &optional proto* [function]

creates a socket whose name is defined in *domain* and whose abstract type is *type*. *type* should be one of 1 (SOCK_STREAM), 2 (SOCK_DGRAM), 3 (SOCK_RAW), 4 (SOCK_RDM) and 5 (SOCK_SEQPACKET).

unix:bind *socket name* [function]

associates *name* to *socket*. *name* should be a unix path-name if the socket is defined in unix-domain.

unix:connect *socket addr* [function]

connects *socket* to another socket specified by *addr*.

unix:listen *socket &optional backlog* [function]

begins to accept connection request on *socket*. *backlog* specifies the length of the queue waiting for the establishment of connection.

unix:accept *socket* [function]

accepts the connection request on *socket* and returns a file-descriptor on which messages can be exchanged bidirectionally.

unix:recvfrom *socket &optional mesg from flag* [function]

receives a datagram message from *socket*. The socket must be assigned a name by **unix:bind**. *mesg* is a string in which the incoming message will be stored. If *mesg* is given, **recvfrom** returns the number of bytes received. If it is omitted, a new string is created for the storage of the message and returned.

unix:sendto *socket addr mesg &optional len flag* [function]

sends a datagram message to another socket specified by *addr*. *Socket* must be a datagram-type socket which has no name assigned. *Mesg* is a string to be sent and *len* is the length of the message counting from the beginning of the string. If omitted, whole string is sent.

unix:getservbyname *servicename* [function]

returns the service number (integer) for *servicename* registered in `/etc/services` or in NIS database.

unix:gethostbyname *hostname* [function]

returns the list of ip address of *hostname* and its address type (currently always AF_INET==2).

unix:syserrlist *errno* [function]

returns a string describing the error information for the error code *errno*.

11.2.4 Signals

unix:signal *signal func &optional option* [function]

installs the signal handler *func* for *signal.* In BSD4.2 systems, signals caught during system call processing cause the system call to be retried. This means that if the process is issuing a read system call, signals are ignored. If *option=2* is specified, signals are handled in the system-5 manner, which causes the system call to fail.

unix:kill *pid signal* [function]

sends a signal to a process named by *pid.*

unix:pause [function]

suspends execution of this process until a signal arrives.

unix:alarm *time* [function]

sends an alarm clock signal (SIGALRM 14) after *time* seconds. Calling **unix:alarm** with *time*=0 resets the alarm clock.

unix:ualarm *time* [function]

same as **unix:alarm** except that the unit of *time* is micro seconds. **ualarm** is not available on Solaris2 or on other Sys5 based systems.

unix:getitimer *timer* [function]

One Unix process is attached with three interval timers, i.e., a real-time timer that decrements as the real time passes, a virtual-timer that decrements as the process executes in the user space, and a prof-timer that decrements as the kernel executes on behalf of the user process. *timer* is either 0 (`ITIMER_REAL`), 1 (`ITIMER_VIRTUAL`), or 2(`ITIMER_PROF`). A list of two elements is returned, the value of the timer in second and the interval. Both are floating-point numbers.

unix:setitimer *timer value interval* [function]

sets *value* and *interval* in *timer. timer* is eiterh 0 (`ITIMER_REAL`), 1 (`ITIMER_VIRTUAL`), or 2(`ITIMER_PROF`). `ITIMER_REAL` delivers SIGALRM when *value* expires. `ITIMER_VIRTUAL` delivers SIGVTALRM, and `ITIMER_PROF` delivers SIGPROF.

unix:select *inlist outlist exceptlist timeout* [function]

inlist, outlist and *exceptlist* are bitvectors indicating file descriptors whose I/O events should be tested. For example, if *inlist*=#b0110, *outlist*=#b100, and *exceptlist*=NIL, then whether it is possible to read on fd=1 or 2, or to write on fd=2 is tested. *Timeout* specifies seconds for which **select** is allowed to wait. Immediately after incoming data appear on one of the ports specified in *inlist*, or writing become available on one of the ports specified in *outlist*, or exceptional condition arises in one of the ports specified in *exceptlist*, **select** returns the number of ports that are available for I/O operation, setting ones for the possible port s in each of *inlist, outlist and exceptlist.*

unix:select-read-fd *read-fdset timeout* [function]

I/O selection is usually meaningful only for input operation. **unix:select-read-fd** is a short-hand for `select fdset nil nil timeout`. *Read-fdset* is not a bit-vector, but an integer that specifies the reading fd set.

11.2.5 Multithread

There is no way to create bound threads. Therefore only one signal stack and one interval timer are available in a EusLisp process. On Solaris2, the main top-level runs in a separated thread.

unix:thr-self [function]

returns the id (integer) of the thread currently running.

unix:thr-getprio *id* [function]

returns the execution priority of the thread specified by *id*.

unix:thr-setprio *id newprio* [function]

sets the execution priority of the thread specified by *id* to *newprio*. The smaller numerical value of *newprio* means the higher priority. In other words, a thread with a numerically greater *newprio* gets less access to CPU. Users cannot raise the execution priority higher than the process's nice value, which is usually 0.

unix:thr-getconcurrency [function]

returns the concurrency value (integer) which represents the number of threads that can run concurrently.

unix:thr-setconcurrency *concurrency* [function]

The concurrency value is the number of LWP in the process. If the concurrency is 1, which is the default, many threads you created are assigned to one LWP in turn even though all of them are runnable. If the program is running on a multi-processor machine and you want to utilize more than one CPU at the same time, you should set a value bigger than one to *concurrency*. Note that a big concurrency value let the operating system consume more resource. Usually *concurrency* should be smaller than or equal to the number of processors.

unix:thr-create *func arg-list &optional (size 64*1024)* [function]

creates a new thread with *size* words of Lisp stack and *size* bytes of C stack, and let it apply *func* to *arg-list*. The thread cannot return any results to the caller. Use of this function is discouraged.

11.2.6 Low-Level Memory Management

unix:malloc *integer* [function]

allocates memory outside EusLisp memory space.

unix:free *integer* [function]

deallocates a memory block allocated by *unix:malloc*.

unix:valloc *integer* [function]

unix:mmap *address length protection share stream offset* [function]

unix:munmap *address length* [function]

unix:vadvise *integer* [function]

11.2.7 IOCTL

Although Unix controls terminal device by a set of commands (second argument) to `ioctl`, EusLisp provides them in the forms of function to eliminate to reference the include files and or'ing argument with the command codes. For the detail, refer to the *termio* manual pages of Unix.

There are two sets of terminal io-controls: TIOC* and TC*. Be careful about the availability of these functions on your operating system. Basically, BSD supports TIOC* io-controls and Sys5 supports TC*.

SunOS 4.1 Both TIOC* and TC*

Solaris2 only TC*

mips, ultrix? only TIOC*

unix:tiocgetp *stream &optional sgttybuf* [function]
gets parameters.

unix:tiocsetp *stream sgttybuf* [function]
sets parameters.

unix:tiocsetn *stream &optional sgttybuf* [function]

unix:tiocgetd *stream &optional sgttybuf* [function]

unix:tiocflush *stream* [function]
flushes output buffer.

unix:tiocgpgrp *stream integer* [function]
gets process group id.

unix:tiocspgrp *stream integer* [function]
sets process group id.

unix:tiocoutq *stream integer* [function]

unix:fionread *stream integer* [function]

unix:tiocsetc *stream buf* [function]

unix:tioclbis *stream buf* [function]

unix:tioclbic *stream buf* [function]

unix:tioclset *stream buf* [function]

unix:tioclget *stream buf* [function]

unix:tcseta *stream buffer* [function]

sets terminal parameters immediately.

unix:tcsets *stream buffer* [function]

sets terminal parameters.

unix:tcsetsw *stream buffer* [function]

sets terminal parameters after all characters queued for output have been transmitted.

unix:tcsetsf *stream buffer* [function]

sets terminal parameters after all characters queued for output have been transmitted and all characters queued for input are discarded.

unix:tiocsetc *stream buffer* [function]

unix:tcsetaf *stream buffer* [function]

unix:tcsetaw *stream buffer* [function]

unix:tcgeta *stream buffer* [function]

unix:tcgets *stream buffer* [function]

unix:tcgetattr *stream buffer* [function]

unix:tcsetattr *stream buffer* [function]

11.2.8 Keyed Indexed Files

Recent Unix provides with the *dbm* or *ndbm* library for the management of keyed index files. Making use of this library, you can build a data base that is composed of many pairs of key and datum association. Following functions are defined in clib/ndbm.c. On Sun, it should be compiled by `cc -c -Dsun4 -Bstatic`, and loaded into EusLisp by `(load "clib/ndbm" :ld-option "-lc")`.

dbm-open *dbname mode flag* [function]

Dbm-open must be called first to create a data base file, and to begin read/write operations to the data base. *Dbname* is the name of the data base. Actually, ndbm manager creates two files which have suffixes `".pag"` and `".dir"`. *Mode* specifies file-open mode; 0 for read-only access, 1 for write-only, and 2 for read-write; also #x200 should be *or*ed when you create the file at the first time. *Flag* gives access permission that is changed by chmod. #o666 or #o664 is good for *flag*. **Dbm-open** returns an integer that identifies the data base in the process. This value is used by other dbm functions to identify the data base. In other words, you can open several data bases at the same time.

dbm-store *db key datum mode* [function]

stores *key-datum* association in *db*. *Db* is an integer to identify the data base. *Key and datum* are strings. *Mode* is 0 (insert) or 1 (replace).

dbm-fetch *db key* [function]

retrieves datum that is associated with *key* in *db*.

11.3 Unix Processes

In order to launch unix commands from EusLisp, use the **unix:system** function. **Piped-fork** creates a subprocess whose standard input and standard output are connected to EusLisp's bidirectional stream through pipes. **Piped-fork** returns the stream. Following is a function to count the number of lines contained in a file by using `"wc"`.

```
(defun count-lines (file) (read (piped-fork "wc" file)))
```

The next example creates eus process on another workstation identified by "etlic0" and provides a port for distributed computation.

```
(setq ic0eus (piped-fork "rsh" "etlic0" "eus"))
(format ic0eus "(list 1 2 3)~%")
(read ic0eus) --> (1 2 3)
```

For source code editing, you can call `ez` from the EusLisp. The screen editor ez communicates with EusLisp through message-queues. If you have an ez process already running in parallel with the EusLisp, **ez** restarts ez and it gains the terminal control. By issuing esc-P or esc-M commands in ez, texts are sent back and evaluated by EusLisp. This is useful for the debugging since entire file does not need to be loaded when you add a little modification to the file. Similar function is available on emacs by M-X run-lisp command.

cd *&optional (dir (unix:getenv "HOME"))* [function]

changes the current working directory.

ez *&optional key* [function]

enters display editor ez, and reads Lisp forms from it, and evaluates them.

piped-fork *&optional (exec) &rest args* [function]

forks a process, and makes a two-way stream between the current EusLisp and the subprocess. Exec is the file name of a unix command and args are arguments to the command. If exec (string) includes one or more space, it is assumed a shell command, and executed by /bin/sh calling the unix:system function. If no exec is given, another euslisp is created as the subprocess.

xfork *exec &key (stdin *standard-input*) (stdout *standard-output*) (stderr *error-output*) (args nil)* [function]

forks a process, replaces its stdin, stdout, and stderr streams to specified ones, and exec's "exec" with the args arguments. piped-fork is roughly equivalent to `(xfork exec :stdin (unix:pipe) :stdout (unix:pipe))` Though xfork returns an io-stream to stdin and stdout with their directions reversed, it is not always useful unless they are pipes. The name of this function, xfork (cross-fork), comes from this reversed io-stream, namely, the io-stream's input comes from the stdout of the subprocess and the output comes from the stdin.

rusage [function]

prints resource usage of this process.

11.4 Adding Lisp Functions Coded in C

Programs that heavily refer to C include files or frequently access arrays perform better or are more clearly described if written in C or other languages rather than in EusLisp. EusLisp provides the way to link programs coded in C.

If you want to define EusLisp function written in C, each EusLisp-callable C-function must be coded to accept three arguments: the context pointer, the number of arguments and the pointer to the Lisp argument block. These arguments must be named as `ctx, n` and `argv`, since the macros in `c/eus.h` assume these names. The C program must include `*eusdir*/c/eus.h`. The programmer should be familiar with the types and macros described there. The entry function should be named by the basename of the source file.

A sample code for C function AVERAGE which computes the arithmetic average of arbitrary number of floats is shown below. In this example, you can see how to get float values from arguments, how to make the pointer of a float, how to set a pointer in the special variable AVERAGE, and how to define a function and a symbol in the entry function `ave`. Compile this program by `'cc -c -Dsun4 -DSolaris2 -K pic'`. `-Dsun4` and `-DSolaris2` are needed to chose proper definitions in `c/eus.h`. `-K pic` is needed to let the c compiler generate position independent code necessary for the loadable shared object. Then the resulted '.o' file can be loaded into EusLisp. More complete examples can be found in `*eusdir*/clib/*.c`, which are defined and loaded in the same manner described here.

```
/* ave.c */
/* (average &rest numbers) */
#include "/usr/local/eus/c/eus.h"
static pointer AVESYM;
pointer AVERAGE(ctx,n,argv)
context *ctx;
int n;
pointer argv[];
{ register int i;
  float sum=0.0, a, av;
  pointer result;
  numunion nu;
  for (i=0; i<n; i++) {
    a=ckfltval(argv[i]);
    sum += a;} /*get floating value from args*/
  av=sum/n;
  result=makeflt(av);
  AVESYM->c.sym.speval=result;  /*kindly set the result in symbol*/
  return(result);}

ave(ctx,n,argv)
context *ctx;
int n;
pointer argv[];
{ char *p;
  p="AVERAGE";
  defun(ctx,p,argv[0],AVERAGE);
  AVESYM=intern(ctx,p,strlen(p),userpkg); /* make a new symbol*/
```

```
}
```

11.5 Foreign Language Interface

Functions written in C without concern about linking with EusLisp can be loaded onto EusLisp, too. These functions are called foreign functions. Such programs are loaded by **load-foreign** macro which returns an instance of **foreign-module**. External symbol definitions in the object file is registered in the module object. **Defforeign** is used to make entries to C functions to be called from EusLisp. **Defun-c-callable** defines lisp functions callable from C. C-callable functions have special code piece called *pod-code* for converting parameters and transferring control to the corresponding EusLisp function. **Pod-address** returns the address of this code piece which should be informed to C functions.

Here is an example of C program and its interface functions to EusLisp.

```
/* C program named cfunc.c*/

static int (*g)(); /* variable to store Lisp function entry */

double sync(x)
double x;
{ extern double sin();
  return(sin(x)/x);}

char *upperstring(s)
char *s;
{ char *ss=s;
  while (*s) { if (islower(*s)) *s=toupper(*s); s++;}
  return(ss);}

int setlfunc(f)      /* remember the argument in g just to see */
int (*f)();          /* how Lisp function can be called from C */
{ g=f;}

int callfunc(x)      /* apply the Lisp function saved in g to the arg.*/
int x;
{ return((*g)(x));}

;;;; Example program for EusLisp's foreign language interface
;;;; make foreign-module
(setq m (load-foreign "cfunc.o"))

;; define foreign functions so that they can be callable from lisp
(defforeign sync m "sync" (:float) :float)
(defforeign toupper m "upperstring" (:string) :string)
(defforeign setlfunc m "setlfunc" (:integer) :integer)
(defforeign callfunc m "callfunc" (:integer) :integer)
```

```
;; call them
(sync 1.0) --> 0.841471
(print (toupper "abc123"))  --> "ABC123"

;; define a test function which is callable from C.
(defun-c-callable TEST ((a :integer)) :integer
      (format t "TEST is called, arg=~s~%" a)
      (* a a))    ;; return the square of the arg
;;  call it from C
;;setlfunc remembers the entry address of Lisp TEST function.
(setlfunc (pod-address (intern "TEST")))
(callfunc 12)  --> TEST is called, arg=12  144
```

Data representations in EusLisp are converted to those of C in the following manners: EusLisp's 30-bits integer (including character) is sign-extended and passed to a C function via stack. 30-bit float is extended to double and passed via stack. As for string, integer-vector and float-vector, only the address of the first element is passed on the stack, and the entire array remains uncopied. The string can either be a normal string or a foreign-string. A string may contain null codes, though it is guaranteed that the string also has a null code at the end. EusLisp does not know how to pass arrays of more than one dimension. Every array of more than one dimension has correspoiding one dimensional vector that holds the entire elements linearly. This vector is obtained by the **array-entity** macro. Also, note that a two-dimensional matrix should be transposed if it is sent to the FORTRAN subroutines, since rows and columns are ordered oppositely in FORTRAN.

Since EusLisp's representation of floating-point numbers is always single precision, conversion is required when you pass a vector of double precision floating point numbers. For this purpose, the conversion functions, **double2float** and **float2double** are provided by `clib/double.c`. For an instance, if you have a 3x3 float-matrix and want to pass it to a C function named cfun as a matrix of double, use the following forms.

```
(setq mat (make-matrix 3 3))
(cfun (float2double (array-entity mat)))
```

Struct in C can be defined by the **defcstruct** macro. **Defcstruct** accepts struct-name followed by field definition forms.

```
(defcstruct <struct-name>
   {(<field> <type> [*] [size])}*)
```

For example, following struct definition is represented by the next defcstruct.

```
/* C definition */
struct example {
   char  a[2];
   short b;
   long  *c;
   float *d[2];};

/* equivalent EusLisp definition */
(defcstruct example
   (a :char 2)
```

```
(b :short)
(c :long *)
(d :float * 2))
```

load-foreign *objfile &key symbol-input symbol-output (symbol-file objfile) ld-option)* [macro]

loads an object module written in languages other than EusLisp. In Solaris2, **load-foreign** just calls **load** with a null string as its *:entry* parameter. A compiled-code object is returned. This result is necessary to make entries to the functions in the module by **defforeign** called later on. Libraries can be specified in *ld-option*. However, the symbols defined in the libraries cannot be captured in the default symbol-output file. In order to allow EusLisp to call functions defined in the libraries, *symbol-output* and *symbol-file* must be given explicitly. (These arguments are not needed if you are not going to call the library functions directly from EusLisp, i.e. if you are referring them only from functions in *objfile*). **Load-foreign** links *objfile* with libraries specified and global symbols in EusLisp which is in core, and writes the linked object in *symbol-output*. Then, symbols in *symbol-file* are searched and listed in the foreign-module. Since *symbol-file* is defaulted to be *objfile*, only the symbols defined in *objfile* are recognized if *symbol-file* is not given. To find all the global entries both in *objfile* and libraries, the linked (merged) symbol table resulted from the first link process of load-foreign must be examined. For this reason, an identical file name must be given both to *symbol-output* and to *symbol-file*.

As shown below, the intermediate symbol file can be removed by **unix:unlink**. However, if you are loading more than one foreign modules both of which refer to the same library, and if you want to avoid loading the library duplicatedly, you have to use *symbol-input* argument. Suppose you have loaded all the functions in "linpack.a" in the above example and you are going to load another file "linapp.o" that calls functions in "linpack.a". The following call of load-foreign should be issued before you unlink "euslinpack". `(load-foreign "linapp.o" :symbol-input "euslinpack")` See `*eusdir*/llib/linpack.l` for more complete examples of **load-foreign** and **defforeign**.

```
(setq linpack-module
    (load-foreign "/usr/local/eus/clib/linpackref.o"
        :ld-option  "-L/usr/local/lib -llinpack -lF77 -lm -lc"
        :symbol-output "euslinpack"
        :symbol-file "euslinpack"
        ))
(unix:unlink "euslinpack")
```

defforeign *funcname module cname paramspec resulttype* [macro]

makes a function entry in a foreign language module. *funcname* is a symbol to be created in EusLisp. *module* is a compiled-code object returned by **load-foreign**. *cname* is the name of the C-function defined in the foreign program. It is a string like "_myfunc". *paramspec* is a list of parameter type specifications which is used for the data type conversion and coercion when arguments are passed from EusLisp to the C function. *Paramspec* can be NIL if no data conversion or type check is required. One of **:integer, :float , :string**, or *(:string n)* must be given to *resulttype*. **:Integer** means that the c function returns either char, short or int (long). **:Float** should be specified both for float and double. **:String** means the C function returns a pointer to a string, and EusLisp should add a long-word header to the string to accomodate it as a EusLisp string. The length of the string is found by *strlen*. Note that the writing a header just before the string may cause a disastrous result. On the other hand, **(:string n)** is safer but slower because a EusLisp string of length **n** is newly created and the contents of C string is copied there. `(:string 4)` should be used for a C function that returns a pointer to an integer. The resulted integer value of the result can be obtained by `(sys:peek result :long)`, where result is a variable set to the result of the C function. You may also specify `(:foreign-string [n])` for C functions that return a string or a struct. The result is a foreign-string whose content is held somewhere outside EusLisp control. If the result string is null-terminated and the length of the string is known by strlen, you don't need to specify the length [n]. However, if the result contains null codes, which is usual for structs, the length of the foreign-string should be explicitly given. Whether you should use (:string n) or (:foreign-string n) is not only the matter of speed, but the matter of structure sharing. The difference is whether the result is copied or not.

Fortran users should note that every argument to a Fortran function or a subroutine is passed by call-by-reference. Therefore, even a simple integer or float type argument must be put in a integer-vector or a float-vector before it is passed to Fortran.

defun-c-callable *funcname paramspec resulttype &rest body* [macro]

defines a EusLisp function that can be called from foreign language code. *funcname* is a symbol for which a EusLisp function is defined. *paramspec* is a list of type specifiers as in **defforeign**. Unlike **defforeign**'s paramspec, **defun-c-callable**'s paramspec cannot be omitted unless the function does not receive any argument. *:integer* should be used for all of int, short and char types and *:float* for both of float and double. *resulttype* is the type of the Lisp function. *resulttype* can be omitted unless you need type check or type coercion from integer to float. *body* is lisp expressions that are executed when this function is called from C. The function defined by **defun-c-callable** can be called from Lisp expressions, too. **Defun-c-callable** returns *funcname*. It looks like a symbol, but it is not, but an instance of **foreign-pod** which is a subclass of symbol.

pod-address *funcname* [function]

returns the address of a foreign-to-EusLisp interface code of the c-callable Lisp function *funcname* defined by **defun-c-callable**. This is used to inform a foreign language program of the location of a Lisp function.

array-entity *array-of-more-than-one-dimension* [macro]

returns one-dimensional vector which holds all the elements of a multi-dimensional array. This is needed to pass a multi-dimensional or general array to a foreign function, although a simple vector can be passed directly.

float2double *float-vector &optional doublevector* [function]

converts *float-vector* to double precision representation. The result is of type float-vector but the length is twice as much as the first argument.

double2float *doublevector &optional float-vector* [function]

A vector of double precision numbers is converted to single precision float-vector.

11.6 VxWorks

ホストと VxWorks との通信機能が"vxworks/vxweus.l"ファイルで提供されている。VxWorks 上に **vxwserv** サーバを常駐させることにより、ホスト上の EusLisp から **vxwserv** にコネクションを張り、vxws プロトコルに従ったコマンドを送ることにより、VxWorks の関数を起動し、引数を送り、結果を受け取ることができる。

VxWorks のソフトは Sun の c コンパイラによって開発することができる上、データ表現が sun3, sun4, VME147 の間で共通であることを利用して、vxws プロトコルは、バイナリモードで動作することができる。

11.6.1 VxWorks 側の起動

VxWorks にログインし、"*eusdir*/vxworks/vxwserv.o"をロードする。その後、**vxwserv** タスクを spawn する。**vxwserv** は VxWorks 上の 2200 番ポートを listen する。2200 が塞がっている場合、2201, 2202, ... を試す。正しく bind されたポート番号が表示される。

```
% rlogin asvx0  (あるいは etlic2 上であれば、% tip asvx[01] も可能)
-> cd "atom:/usr/share/src/eus/vxworks"
-> ld <vxwserv.o
-> sp vxwserv
port 2200 is bound.
```

VxWorks の i コマンドで、**vxwserv** タスクが常駐したことを確かめる。同じ要領で、eus から呼び出したい VxWorks のプログラムを VxWorks 上にロードする。その後、Euslisp と VxWorks とのコネクションが張られると、**vxwserv** を走らせた TTY に、次のようなメッセージが出力される。

```
CLIENT accepted: sock=9 port = 1129: family = 2: addr = c01fcc10:
VxWserv started with 16394 byte buffer
```

11.6.2 ホスト側の起動

任意のマシンの上で eus を起動し、"vxworks/vxweus"をロードする。**connect-vxw** 関数を用いて **vxwserv** に接続する。接続後、ソケットストリームが ***vxw-stream*** にバインドされる。以下に、コネクトの例を示す。この例では、VxWorks 上の sin, vadd 関数を euslisp の関数 VSIN,VAD として定義している。

```
(load "vxworks/vxweus")
(setq s (connect-vxw :host "asvx0" :port 2200 :buffer-size 1024))
(defvxw VSIN "_sin" (theta) :float)
(defvxw VAD "_vadd" (v1 v2) (float-vector 3))
```

VxWorks 上に作成される関数が、vxws を通じて呼び出されるためには、次の条件を満たさなければならない。

1. 引数は、32 個以内であること、引数に受け取るベクタの容量の合計が connect-vxw の:buffer-size で指定した値を越えないこと
2. struct を引数にしないこと、必ず struct へのポインタを引数にすること
3. 結果は、int, float, double または、それらの配列のアドレスであること
4. 配列のアドレスを結果とする場合、その配列の実体は、関数の外部に取られていること

connect-vxw [関数]

&key (host "asvx0")
(port 2200)
(buffer-size 16384)
(priority 1280)
(option #x1c)

:host に対して vxws プロトコルによる通信のためのソケットストリームを作成し、そのストリームを返す。*:host* には、ネットワークにおける VxWorks のアクセス番号あるいはアクセス名を指定する。*:port* には、VxWorks 上の **vxwserv** がバインドしたポートを捜すための最初のポート番号を指定する。このポート番号から、増加方向に接続を試行する。*:option* のコードについては、VxWorks の、spawn 関数を参照のこと。コネクションは、同時に複数張ってよい。

vxw *vxw-stream entry result-type args* [function]

vxw は、*vxw-stream* に接続されている VxWorks の関数 *entry* を呼び出し、その関数に引き数 *args* を与えて *result-type* で指定された結果を得る。*vxw-stream* には、**connect-vxw** で作成したソケットストリームを与える。*entry* には、VxWorks の関数名をストリングで指定するか、あるいは関数のアドレスを整数で指定する。関数のアドレスを知るには、VxWorks の findsymbol を呼び出す。知りたいシンボルは、通常、"_"で始まることに注意。*entry* がストリングの場合、VxWorks 上でシンボルテーブルの逐次探索が行われる。*result-type* には、結果のデータ型 (`:integer` または`:float`)、あるいはデータを受け取るベクタ型を指定する。ベクタは、`float-vector`，`integer-vector`，`string` のインスタンスである。`general vector`(lisp の任意のオブジェクトを要素とするベクタ) は指定できない。結果型は、必ず、実際の VxWorks 関数の結果型と一致しなければならない。*args* には、*entry* に与える引き数を指定する。引数に許される EusLisp データは、`integer`，`float`，`string`，`integer-vector`，`float-vector`，`integer-matrix`，`float-matrix` である。ポインタを含んだ一般のオブジェクト、一般のベクトルは送れない。また、送られたベクトルデータは、一旦 **vxwserv** が獲得したバッファの中に蓄積される。例えば、VxWorks に定義された関数"sin"を呼び出すためには、次のように実行すればよい。`(vxw *vxw-stream* "sin" :float 1.0)`

defvxw *eus-func-name entry args &optional (result-type :integer)* [macro]

defvxw は、findsymbol を用いて **vxw** を呼び出して、VxWorks の関数の高速な呼び出しを実現するためのマクロである。VxWorks の関数 *entry* を呼び出すための Euslisp の関数 *eus-func-name* を定義する。このマクロを実行後は、*eus-func-name* を呼び出すことにより、VxWorks の関数を呼び出すことができる。このとき、呼び出しに使用されるソケットストリームは***vxw-stream***に固定されている。ただし、VxWorks 側で、関数をコンパイルし直して再ロードした場合、新しい関数定義が呼ばれるようにするためには、eus 側で、**defvxw** をもう一度実行し直して、最新のエントリアドレスが指定されるようにする必要がある。

12 Multithread

The multithread is the concurrent and asynchronous programming facility on the Solaris operating system. Asynchronous programming is required for programs to respond to external events via multiple sensors occurring independently of the program's state. Parallel programming is effective to improve performance of computation bound processing such as image processing and interference checking in path planning.

12.1 Design of Multithread EusLisp

12.1.1 Multithread in Solaris 2 operating system

Multithread EusLisp (MT-Eus) runs on the Solaris 2 operating system with one or more processors. Solaris's threads are units for allocating CPU in a traditional UNIX process, having shared memory and different contexts. The thread library provided by the Solaris OS allocates each thread to a single LWP (light weight process), which is a kernel resource. The Unix kernel schedules the allocation of LWPs to one or more physical CPUs based on thread priorities assigned to each thread. Fig.**??** depicts the relations between threads, LWPs, and CPUs. Two major changes in the design of the contexts and the memory management of EusLisp have been made to upgrade it to multithread capabilities.

12.1.2 Context Separation

MT-Eus allocates private stacks and contexts to each threads so that they can run independently of each other. Objects such as symbols and conses are allocated in the shared heap memory as in sequential EusLisp. Therefore, thread-private data such as block labels, catch tags, and local variables are protected from other threads, whereas values (objects) pointed by global variables are visible to all threads allowing information exchange among threads.

A context consists of a C-stack, a binding-stack and frame pointers that chain lexical blocks such as `lambda, block, catch, let, flet`, and so on, and is established when a new thread is created. Since more than one context can be active at the same time on a real multi-processor machine, we cannot hold a single pointer to the current context in a global variable. Rather we have to add one more argument to every internal function to transfer the context pointer from the topmost eval to the memory manager at the bottom.

12.1.3 Memory Management

EusLisp adopts a Fibonacci buddy memory management scheme in a single heap for every type of object. After running programs having different memory request characteristics, we have been convinced that Fibonacci buddy can allocate objects of various sizes equally fast, garbage-collects quickly without copying , and exhibits high memory utilization (the internal loss is 10 to 15% and the external loss is negligible). For multithreading, the second point, i.e., non-copying GC, is very important. If addresses of objects were changed by copying-GC, pointers in the stack and CPU registers of all thread contexts would have to be redirected to new locations, which is impossible or very difficult.

All memory allocation requests are handled by the `alloc` function at the lowest level. `Alloc` does mutex-locking because it manipulates the global database of free lists. Since we cannot predict when a garbage collection begins and which thread causes it, every thread must prepare for sporadic GCs. All pointers to living objects have to be arranged to be accessible by the GC anytime to prevent them from being reclaimed as garbage. This is done by storing the pointers to the most recently allocated objects in fixed slots of each context, instead of trusting they are maintained on the stacks.

Fig. ?? illustrates flow of threads requesting memory and forked inside GC to process marking and sweeping in parallel. Note that threads that do not request memory or manipulate pointers can run in parallel with the GC, improving real-time response of the low-level tasks such as signal processing and image acquisition.

12.2 Asynchronous and Parallel Programming Constructs

12.2.1 Thread Creation and Thread Pool

In order for Solaris to execute a program in parallel on many processors, the program needs to be written as a collection of functions, each of which is executed by a thread dynamically created in a process. Although the time required for thread creation is faster than process creation, it takes a few mili-seconds for EusLisp to start off a thread after allocating stacks and setting a page attribute for detecting stack-overflow. Since this delay, which should be compared to a function invocation, is intolerable, sufficient number of threads are created by the `make-thread` function beforehand and put in the system's thread pool, eliminating the need for system calls at evaluation time. Each thread in the thread pool is represented by a thread object, as depicted in Fig.??, consisted of thread-id, several semaphores for synchronization, and slots for argument and evaluation result transfer.

12.2.2 Parallel Execution of Threads

For the allocation of parallel computation to threads, the thread function is used. Thread takes one free thread out of the thread pool, transfers arguments via shared memory, wakes up the thread by signaling the semaphore as indicated in fig. ??, and returns a thread object to the caller without blocking. The woken-up thread begins evaluation of the argument running in parallel to the calling thread. The caller uses `wait-thread` to receive the evaluation result from the forked thread. The `plist` macro is a more convenient form to describe parallel evaluation of arguments. `Plist` attaches threads to evaluate each argument and lists up results after waiting for all threads to finish evaluation.

12.2.3 Synchronization primitives

MT-Eus has three kinds of synchronization primitives, namely *mutex locks, condition variables*, and *semaphores*. Mutex locks are used to serialize accesses to shared variables between threads. Condition variables allow a thread to wait for a condition to become true in a mutex-locked section by temporarily releasing and re-acquiring the lock. Semaphores are used to inform occurrences of events, or to control sharing of finite resources. These synchronization primitives cause voluntary context switching, while the Solaris kernel generates involuntary task switching on a time-sliced scheduling basis.

12.2.4 Barrier synchronization

Barrier-synch is a mechanism for more than two threads to synchronize at the same time (Fig. ??). For this purpose, an instance of the barrier class is created and threads that participate in the synchronization register themselves in the object. Then, each thread sends the `:wait` message to the barrier object, and the thread is blocked. When the last thread registered in the object sends its `:wait` message, the waits are released and all waiting threads get a return value of T. Barrier-sync plays an important role of global clocking in a multi-robot simulation.

12.2.5 Synchronized memory port

Synchronized memory port is a kind of stream to exchange data between threads (Fig. ??). Since all threads in a process share the heap memory, if one thread binds an object to a global variable, it instantly becomes visible to other threads. However, shared memory lacks capability to send events that the global data is updated. Synchronized memory port ensures this synchronization for accessing a shared object. A synchronized memory port object consists of one buffer slot and two semaphores used for synchronizing read and write.

12.2.6 Timers

Real-time programs often require functions to execute at predetermined timing or to repeat in particular intervals. Sequential EusLisp could run user' functions triggered by signals generated periodically by Unix's interval timers. This preemption can cause deadlock in MT-Eus, because interruption may occur within a mutex-ed block. Therefore, control must be transferred at secured points such as at the beginning of `eval`. To avoid delays caused by the above synchronization, MT-Eus also provides signal-notification via semaphores. In other words, the signal function takes either a function or a semaphore that is called or posted upon the signal arrival. Since the semaphore is posted at the lowest level, latency for synchronization is minimal.

The following a example image processing program coded by using the multithread facilities. Image input thread and filtering threads are created. samp-image takes image data periodically by waiting for samp-sem to be posted every 33msec. Two threads synchronize via read-and-write of a thread-port. Filter-image employs two more threads for parallel computation of filtering.

```
(make-threads 8)
(defun samp-image (p)
   (let ((samp-sem (make-semaphore)))
        (periodic-sema-post 0.03 samp-sem)
        (loop (sema-wait samp-sem)
              (send p :write (read-image))))
(defun filter-image (p)
  (let (img)
       (loop (setf img (send p :read))
             (plist (filter-up-half img)
                    (filter-low-half img)))))
(setf port (make-thread-port))
(setf sampler (thread #'samp-image port))
(setf filter (thread #'filter-image port))
```

processors	1	2	4	8	GC (ratio)
(a) compiled Fibonacci	1.0	2.0	4.0	7.8	0
(b) interpreted Fibonacci	1.0	1.7	2.7	4.4	0
(c) copy-seq	1.0	1.3	0.76	0.71	0.15
(d) make-cube	1.0	0.91	0.40	0.39	0.15
(e) interference-check	1.0	0.88	0.55	0.34	0.21

表 3: Parallel gains of programs executed on multi-processors

12.3 Measured Parallel Gains

Table. **??** shows the parallel execution performance measured on a Cray Supserserver configured with 32 CPUs. Linear parallel gain was obtained for the compiled Fibonacci function, because there is no shared memory access and the program code is small enough to be fully loaded onto the cache memory of each processor. Contrally, when the same program was interpreted, linearly high performance could not be attained, since memory access scatters. Further, some programs that frequently refer to shared memory and request memory allocation cannot exhibit better performance than a single processor execution. This can be understood as the result of frequent cache memory purging.

12.4 Thread creation

A thread is a unit for assigning computation, usually evaluation of a lisp form. Threads in EusLisp are represented by instances of the **thread** class. This object is actually a control port of a thread to pass arguments and result, and let it start evaluation, rather than the thread's entity representing the context.

sys:make-thread *num &optional (lsize 32*1024) (csize lsize)* [function]

creates *num* threads with *lsize* words of Lisp stack and *csize* words of C stack, and put them in the system's thread pool. All the threads in the thread pool is bound to sys:*threads*, which is extended each time **make-thread** is called. By the **thread** function, a computation is assigned to one of free threads in the thread pool. Therefore it is not a good idea to change stack sizes from thread to thread, since you cannot control which thread is assigned to a specific computation.

sys:*threads* [variable]

returns the list of all the threads created by **make-threads**.

sys::free-threads [function]

returns the list of threads in the free thread pool. If the result is NIL, new commitment of a task to a thread is blocked until any currently running threads finish evaluation or new threads are created by **make-thread** in the free thread pool.

sys:thread *func &rest args* [function]

picks up one free thread from the thread pool, and assigns it for evaluation of *(func . args)*. **Sys:thread** can be regarded as asynchronous **funcall**, since **sys:thread** applies *func* to the spread list of *args* but it does not accept the result of the function application. Rather, **sys:thread** returns the thread object assigned to the funcall, so that the real result can be obtained later by **sys:wait-thread**.

```
(defun compute-pi (digits) ...)
```

```
(setq trd (sys:thread \#'compute-pi 1000)) ;assign compute-pi to a thread
...  ;; other computation
(sys:wait-thread trd) ;get the result of (compute-pi 1000)
```

sys:thread-no-wait *func &rest args* [function]

assigns computation to one of free threads. The thread is reclaimed in the free thread pool when it finishes evaluation without being **wait-thread**'ed.

sys:wait-thread *thread* [function]

waits for *thread* to finish evaluation of funcall given by the **sys:thread** function, and retrieves the result and returns it. **Sys:wait-thread** is mandatory if the thread is assigned evaluation by **sys:thread** because the thread is not returned to the free thread pool until it finishes transferring the result.

sys:plist *&rest forms* [macro]

evaluates *forms* by different threads in parallel and waits for the completion of all evaluation, and the list of results is returned. **Sys:plist** may be regarded as *parallel-list* except that each form listed must be a function call.

12.5 Synchronization

Among Solaris operating systems four synchronization primitives for multithread programs, EusLisp provides mutex locks, conditional variables, and semaphores. Reader-writer lock is not available now.

Based on these primitives, higher level synchronization mechanisms, such as synchronized memory port and barrier synchronization, are realized.

sys:make-mutex-lock [function]

makes a mutex-lock and returns it. A mutex-lock is represented by an integer-vector of six elements.

sys:mutex-lock *mlock* [function]

locks the mutex lock *mlock*. If the *mlock* is already locked by another thread, *mutex-lock* waits for the lock to be released.

sys:mutex-unlock *mlock* [function]

releases *mlock* and let one of other threads waiting for this lock resume running.

sys:mutex *mlock &rest forms* [macro]

Mutex-lock and mutex-unlock have to be used as a pair. **Mutex** is a macro that brackets a critical section. *Mlock* is locked before evaluating *forms* are evaluated, and the lock is released when the evaluation finishes. This macro expands to the following progn form. Note that **unwind-protect** is used to ensure unlocking even an error occurs during the evaluation of *forms*.

```
(progn
    (sys:mutex-lock mlock)
    (unwind-protect
        (progn . forms)
        (sys:mutex-unlock mlock)))
```

sys:make-cond [function]

makes a condition variable object which is an integer vector of four elements. The returned condition variable is in unlocked state.

sys:cond-wait *condvar mlock* [function]

waits for *condvar* to be signaled. If *condvar* has already been acquired by another thread, it releases *mlock* and waits for *condvar* to be signaled.

sys:cond-signal *condvar* [function]

signals the *condvar* condition variable.

sys:make-semaphore [function]

makes a semaphore object which is represented by an integer vector of twelve elements.

sys:sema-post *sem* [function]

signals *sem*.

sys:sema-wait *sem* [function]

waits for the *sem* semaphore to be posted.

sys:barrier-synch [Class]

:super	**propertied-object**
:slots	threads n-threads count barrier-cond threads-lock count-lock

represents a structure for barrier-synchronization. Threads waiting for the synchronization are put in *threads* which is mutually excluded by *threads-lock*. When a **barrier-synch** object is created, *count* is initialized to zero. Synchronizing threads are put in the *threads* list by sending `:add` message. Sending `:wait` to this barrier-sync object causes *count* to be incremented, and the sending thread is put in the wait state. When all the threads in *threads* send the `:wait` message, the waits are unblocked and all threads resume execution. The synchronization is implemented by the combination of the *count-lock* mutex-lock and the *barrier-cond* condition-variable.

:init [method]

initializes this barrier-synch object. Two mutex-lock and one condition-variable are created.

:add *thr* [method]

adds the *thr* thread in the *threads* list.

:remove *thr* [method]

removes the *thr* thread of the *threads* list.

:wait [method]

waits for all threads in the *threads* list to issue `:wait`.

sys:synch-memory-port [Class]

:super	**propertied-object**
:slots	sema-in sema-out buf empty lock

realizes the one-directional synchronized memory port, which synchronizes for two threads to transfer datum via this object. Control transfer is implemented by using semaphores.

:read [method]

reads datum buffered in this synch-memory-port. If it has not been written yet, the :read blocks.

:write *datum* [method]

writes *datum* in the buffer. Since only one word of buffer is available, if another datum has already been written and not yet read out, :write waits for the datum to be transferred by :read.

:init [method]

initializes this synch-memory-port where two semaphores are created and :write is made acceptable.

13 Geometric Functions

13.1 Float-vectors

A float-vector is a simple vector whose elements are specialized to floating point numbers. A float-vector can be of any size. When *result* is specified in an argument list, it should be a float-vector that holds the result.

float-vector *&rest numbers* [function]

makes a new float-vector whose elements are *numbers*. Note the difference between `(float-vector 1 2 3)` and `#F(1 2 3)`. While the former create a vector each time it is called, the latter does when it is read.

float-vector-p *obj* [function]

returns T if *obj* is a float-vector.

v+ *fltvec1 fltvec2 &optional result* [function]

adds two float-vectors.

v- *fltvec1 &optional fltvec2 result* [function]

subtract float-vectors. If fltvec2 is omitted, fltvec1 is negated.

v. *fltvec1 fltvec2* [function]

computes the inner product of two float-vectors.

v* *fltvec1 fltvec2 &optional result* [function]

computes the outer product of two float-vectors.

v.* *fltvec1 fltvec2 fltvec3* [function]

computes the scaler triple product [A,B,C]=(V. A (V* B C))=(V. (V* A B) C).

v< *fltvec1 fltvec2* [function]

returns T if every element of fltvec1 is smaller than the corresponding element of fltvec2.

v> *fltvec1 fltvec2* [function]

returns T if every element of fltvec1 is larger than the corresponding element of fltvec2.

vmin *&rest fltvec* [function]

finds the smallest values for each dimension in *fltvec*, and makes a float-vector from the values. **Vmin** and **vmax** are used to find the minimal bounding box from coordinates of vertices.

vmax *&rest fltvec* [function]

finds the greatest values for each dimension in *fltvec*, and makes a float-vector from the values.

minimal-box *v-list minvec maxvec &optional err* [function]

computes the minimal bounding box for a given vertex-list, and stores results in *minvec* and *maxvec*. If a floating number err is specified, the minimal box is grown by the ratio, i.e. if the err is 0.01, each element of minvec is decreased by 1% of the distance between minvec and maxvec, and each element of maxvec is increased by 1%. **Minimal-box** returns the distance between minvec and maxvec.

scale *number fltvec &optional result* [function]

the scaler *number* is multiplied to the every element of fltvec.

norm *fltvec* [function]

$|fltvec|$

norm2 *fltvec* [function]

$|fltvec|^2 = (\mathbf{v}.fltvec fltvec)$

normalize-vector *fltvec &optional result* [function]

normalizes *fltvec* to have the norm 1.0.

distance *fltvec1 fltvec2* [function]

returns the distance $|fltvec - fltvec2|$ between two float-vectors.

distance2 *fltvec1 fltvec2* [function]

$|fltvec - fltvec2|^2$

homo2normal *homovec &optional normalvec* [function]

A homogeneous vector *homovec* is converted to its normal representation.

homogenize *normalvec &optional homovec* [function]

A normal vector *normalvec* is converted to its homogenous representation.

midpoint *p p1 p2 &optional result* [function]

P is float, and *p1* and *p2* are float-vectors of the same dimension. A point $(1-p)p1 + pp2$, which is the point that breaks *p1-p2* by the ratio $p : (1-p)$, is returned.

rotate-vector *fltvec theta axis &optional result* [function]

rotates 2D or 3D *fltvec* by *theta* radian around *axis*. *Axis* can be one of *:x, :y, :z, 0, 1, 2* or NIL. When *axis* is NIL, *fltvec* is taken to be two dimensional. To rotate a vector around an arbitrary axis in 3D space, make a rotation matrix by the **rotation-matrix** function and multiply it to the vector.

13.2 Matrix and Transformation

A matrix is a two-dimensional array whose elements are all floats. In most functions a matrix can be of any size, but the **v*, v.***, **Euler-angle** and **rpy-angle** functions can only handle three dimensional matrices. **Transform, m*** and **transpose** do not restrict the matrices to be square, and they operate on general n*m size matrices.

Functions that can accept result parameter places the computed result there, and no heap is wasted. All matrix functions are intended for the transformation in the normal coordinate systems, and not in the homogeneous coordinates.

The **rpy-angle** function decomposes a rotation matrix into three components of rotation angles around z, y and x axes of the world coordinates. The **Euler-angle** function is similar to **rpy-angle** but decomposes into rotation angles around local z, y and again z axes. Both of these functions return two solutions since angles can be taken in the opposite directions.

```
; Mat is a 3X3 rotation matrix.
(setq rots (rpy-angle mat))
(setq r (unit-matrix 3))
(rotate-matrix r (car rots) :x t r)
(rotate-matrix r (cadr rots) :y t r)
```

```
(rotate-matrix r (caddr rots) :z t r)
;--> resulted r is equivalent to mat
```

To keep track of pairs of a position and a orientation in 3D space, use the **coordinates** and **cascaded-coords** classes detailed in the section **??**.

matrix *&rest elements* [function]

makes a new matrix from *elements*. Row x Col = (number of elements) x (length of the 1st element). Each of *elements* can be of any type of sequence. Each sequence is lined up as a row vector in the matrix.

make-matrix *rowsize columnsize &optional init* [function]

makes a matrix of $rowsize \times columnsize$.

matrixp *obj* [function]

T if *obj* is a matrix, i.e. *obj* is a two dimensional array and its elements are floats.

matrix-row *mat row-index* [function]

extracts a row vector out of matrix *mat*. **matrix-row** is also used to set a vector in a particular row of a matrix using in conjunction with **setf**.

matrix-column *mat column-index* [function]

extracts a column vector out of *mat*. **matrix-column** is also used to set a vector in a particular column of a matrix using in conjunction with **setf**.

m* *matrix1 matrix2 &optional result* [function]

concatenates *matrix1* and *matrix2*.

transpose *matrix &optional result* [function]

transposes *matrix*, i.e. columns of *matrix* are exchanged with *rows*.

unit-matrix *dim* [function]

makes an identity matrix of $dim \times dim$.

replace-matrix *dest src* [function]

replaces all the elements of dest matrix with ones of src matrix.

scale-matrix *scalar mat* [function]

multiplies *scaler* to all the elements of *mat*.

copy-matrix *matrix* [function]

makes a copy of *matrix*.

transform *matrix fltvector &optional result* [function]

multiplies *matrix* to *fltvector* from the left.

transform *fltvector matrix &optional result* [function]

multiplies *matrix* to *fltvector* from the right.

rotate-matrix *matrix theta axis &optional world-p result* [function]

multiplies a rotation matrix from the left (when world-p is non-nil) or from the right (when world-p is nil). When a matrix is rotated by **rotate-matrix**, the rotation axis **:x, :y, :z** or 0,1,2 may be taken either in the world coordinates or in the local coordinates. If *world-p* is specified nil, it means rotation along the axis in the local coordinate system and the rotation matrix is multiplied from the right. Else if worldp is non-nil, the rotation is made in the world coordinates and the rotation matrix is multiplied from the left. If NIL is given to *axis*, *matrix* should be two dimensional and the rotation is taken in 2D space where *world-p* does not make sense.

rotation-matrix *theta axis &optional result* [function]

makes a 2D or 3D rotation matrix around *axis* which can be any of :x, :y, :z, 0, 1, 2, a 3D float-vector or NIL. When you make a 2D rotation matrix, axis should be NIL.

rotation-angle *rotation-matrix* [function]

extracts a equivalent rotation axis and angle from *rotation-matrix* and a list of float and float-vector is returned. NIL is returned when *rotation-matrix* is a unit-matrix. Also if the rotation angle is too small, the result may have errors. When *rotation-matrix* is 2D, the single angle value is returned.

rpy-matrix *ang-z ang-y ang-x* [function]

makes a rotation matrix defined by roll-pitch-yaw angles. First, a unit-matrix is rotated by ang-x radian along X-axis. Next, ang-y around Y-axis and finally ang-z around Z-axis. All the rotation axes are taken in the world coordinates.

rpy-angle *matrix* [function]

extracts two triplets of roll-pitch-yaw angles of *matrix*.

Euler-matrix *ang-z ang-y ang2-z* [function]

makes a rotation matrix defined by three Euler angles. First, a unit-matrix is rotated *ang-z* around Z-axis, next, *ang-y* around Y-axis and finally *ang2-z* again around Z-axis. All the rotation axes are taken in the local coordinates.

Euler-angle *matrix* [function]

extracts two tuples of Euler angles.

13.3 LU decomposition

lu-decompose and **lu-solve** are provided to solve simultaneous linear equations. First, **lu-decompose** decomposes a matrix into a lower triangle matrix and an upper triable matrix. If the given matrix is singular, **LU-decompose** returns NIL, otherwise it returns the permutation vector which should be supplied to **LU-solve**. **Lu-solve** computes the solution for a LU matrix with a given constant vector. This method is efficient if solutions for many combinations of different constant vectors and the same factor matrix are required. **Simultaneous-equation** would be more handy when you wish to get only one solution. **Lu-determinant** computes a determinant of a lu-decomposed matrix. **Inverse-matrix** function computes an inverse matrix using **lu-decompose** once, and **lu-solve** n times. Computation time for a 3*3 matrix is estimated to be 4 milli-sec.

lu-decompose *matrix &optional result* [function]

performs lu-decomposition of *matrix*.

lu-solve *lu-mat perm-vector bvector &optional result* [function]

solves a linear simultaneous equations which has already been lu-decomposed. *perm-vector* should be the result returned by **lu-decompose**.

lu-determinant *lu-mat perm-vector* [function]

computes the determinant value for a matrix which has already been lu-decomposed.

simultaneous-equation *mat vec* [function]

solves a linear simultaneous equations whose coefficients are described in *mat* and constant values in *vec*.

inverse-matrix *mat* [function]

makes the inverse matrix of the square matrix, *mat*.

pseudo-inverse *mat* [function]

computes the pseudo inverse matrix using the singular value decomposition.

13.4 Coordinates

Coordinate systems and their transformations are represented by the **coordinates** class. Instead of 4*4 (homogeneous) matrix representation, coordinate system in EusLisp is represented by a combination of a 3*3 rotation matrix and a 3D position vector mainly for speed and generality.

coordinates [Class]

:super **propertied-object**
:slots (pos :type float-vector
rot :type array)

defines a coordinate system with a pair of a position vector and a 3x3 rotation matrix.

coordinates-p *obj* [function]

returns T when obj is an instance of coordinates class or its subclasses.

:rot [method]

returns the 3X3 rotation matrix of this coords.

:pos [method]

returns the 3-D position vector of this coords.

:newcoords *newrot &optional newpos* [method]

updates the coords with newrot and newpos. Whenever a condition that changes the state of this coords occurs, this method should be called with the new rotation matrix and the position vector. This message may invoke another :update method to propagate the event. If newpos is not given, newrot is given as a instance of coordinate class.

:replace-coords *newrot &optional newpos* [method]

changes the rot and pos slots to be updated without calling newcoords method. If newpos is not given, newrot is given as a instance of coordinate class.

:coords [method]

:copy-coords *&optional dest* [method]

If dest is not given, :copy-coords makes another coordinates object which has the same rot and pos slots. If dest is given, rot and pos of this coordinates is copied to the dest coordinates.

:reset-coords [method]

forces the rotation matrix of this coords to be identity matrix, and pos vector to be all zero.

:worldpos [method]

:worldrot [method]

:worldcoords [method]

Computes the position vector, the rotation matrix and the coordinates of this object represented in the world coordinates. The coordinates class is always assumed to be represented in world, these method can simply return pos, rot and self. These methods are provided for the compatibility with cascaded-coords class which cannot be assumed to be represented in world.

:copy-worldcoords *&optional dest* [method]

First, worldcoords is computed, and it is copied to dest. If no dest is specified, a coordinates object to store the result is newly created.

:rotate-vector *vec* [method]

A vector is rotated by the rotation of this coords, i.e., an orientation vector represented in this coords is converted to the representation in the world. The position of this coords does not affect rotation.

:transform-vector *vec* [method]

A vector in this local coords is transformed to the representation in the world.

:inverse-transform-vector *vec* [method]

A vector in the world is inversely transformed to the representation in this local coordinate system.

:transform *trans &optional (wrt :local)* [method]

Transform this coords by the trans represented in wrt coords. Trans must be of type coordinates, and wrt must be one of keywords `:local, :parent, :world` or an instance of coordinates. If wrt is `:local`, the trans is applied from the right to this coords, and if wrt is `:world` or `:parent`, the trans is multiplied from the left. Else, if wrt is of type coordinates, the trans represented in the wrt coords is first transformed to the representation in the world, and it is applied from the left.

:move-to *trans &optional (wrt :local)* [method]

Replaces the rot and pos of the coords with trans represented in wrt.

:translate *p &optional (wrt :local)* [method]

changes the position of this object relatively with respective to wrt coords.

:locate *p &optional (wrt :local)* [method]

Changes the location of this coords with the parameter represented in wrt. If wrt is `:local`, then the effect is identical to **:translate** with *wrt*=`:local`.

:rotate *theta axis &optional (wrt :local)* [method]

Rotates this coords relatively by *theta* radian around the *axis*. *Axis* is one of axis-keywords (`:x, :y` and `:z`) or an arbitrary float-vector. *Axis* is considered to be represented in the *wrt* coords. Thus, if *wrt*=`:local` and *axis*=`:z`, the coordinates is rotated around the z axis of this local coords, and *wrt*=`:world` or `:parent`, the coords is rotated around the z axis of world coords. In other words, if *wrt*=`:local`, a rotation matrix is multiplied from the right of this coords, and if *wrt*=`:world` or `:parent`, a rotation matrix is multiplied from the left. Note that even *wrt* is either `:world` or `:parent`, the *pos* vector of this coordinates does not change. For the true rotation around the world axis, an instance of coordinates class representing the rotation should be given to **:transform** method.

:orient *theta axis &optional (wrt :local)* [method]

forces setting rot. This is an absolute version of :**rotate** method.

:inverse-transformation [method]

makes a new coords that is inverse to self.

:transformation *coords (wrt :local)* [method]

makes the transformation (an instance of coordinates) between this coords and the coords given as the argument. If *wrt*=`:local`, the result is represented in local coords, i.e., if the resulted transformation is given as an argument to :transform with *wrt*=`:local`, this coords is transformed to be identical with the coords.

:Euler *az1 ay az2* [method]

sets rot with Euler angles, that are, rotation angles around z (*az1*, y (*ay*) and again z *az2* axis of this local coordinates system.

:roll-pitch-yaw *roll pitch yaw* [method]

sets rot with roll-pitch-yaw angles: rotation angles around x (*yaw*), y (*pitch*) and z (*roll*) axes of the world coordinate system.

:4x4 *&optional mat44* [method]

If a matrix of 4x4 is given as *mat44*, it is converted to coordinates representation with a 3x3 rotation matrix and a 3D position vector. If *mat44* is not given, this coordinates is converted to 4x4 matrix representation.

:init [method]

&key (pos #f(0 0 0))
(rot #2f((1 0 0) (0 1 0) (0 0 1)))
rpy ; roll pitch yaw
euler ; az ay az2
axis ; rotation-axis
angle ; rotation-angle
4X4 ; 4x4 matrix
coords ; another coordinates
properties ; list of (ind . value) pair
name ; name property

initializes this coordinates object and sets rot and pos. The meaning of each keyword follows:

:dimension 2 or 3 (default is 3)

:pos specifies a position vector (defaulted to #f(0 0 0))

:rot specifies a rotation matrix (defaulted to a unit-matrix)

:euler gives a sequence of three elements for Euler angles

:rpy gives a sequence of three elements for roll-pitch-yaw

:axis rotation axis (:x,:y,:z or an arbitrary float-vector)

:angle rotation angle (used with :axis)

:wrt where the rotation axis is taken (default is :local)

:4X4 4X4 matrix is used to specify both pos and rot

:coords copies pos and rot from coords

:name set :name property

`:Angle` can only be used in conjunction with the `:axis` that is determined in the `:wrt` coordinates. Without regard to `:wrt`, `:Euler` always specifies the Euler angles, *az1, ay* and *az2*, defined in the local coordinates, and `:rpy` specifies the angles around *z*, *y* and *x* axes of the world coordinates. Two or more of `:rot, :Euler, :rpy, :axis` and `:4X4` cannot be specified simultaneously, although no error is reported. Sequences can be supplied to the `:axis` and `:angle` parameters, which mean successive rotations around the given axes. List of pairs of an attribute and its value can be given as `:properties` argument. These pairs are copied in the plist of this coordinates.

13.5 CascadedCoords

cascaded-coords [Class]

:super **coordinates**

:slots (parent descendants worldcoords manager changed)

defines a linked coordinates. **Cascaded-coords** is often abbreviated as **cascoords**.

:inheritance [method]

returns the inheritance tree list describing all the descendants of the cascoords. If `a` and `b` are the direct descendants of this coords, and `c` is a descendant of `a`, `((a (c)) (b))` is returned.

:assoc *childcoords &optional relative-coords* [method]

childcoords is associated to this cascoords as a descendant. If childcoords has been already assoc'ed to some other cascoords, first childcoords is dissoc'ed since each cascoords can have only one parent. The orientation or location of childcoords in the world does not change.

:dissoc *childcoords* [method]

dissociates (removes) *childcoords* from the descendants list of this coords. The orientation or location of childcoords in the world does not change.

:changed [method]

informs this coords that the coordinates of parent has changed, and the re-computation of worldcoords is needed when it is requested later.

:update [method]

is called by the **:worldcoords** method to recompute the current worldcoord.

:worldcoords [method]

returns a coordinates object which represents this coord in the world by concatenating all the cascoords from the root to this coords. The result is held in this object and reused later. Thus, you should not modify the resulted coords.

:worldpos [method]

returns rot of this coordinates represented in the world.

:worldrot [method]

returns pos of this coordinates represented in the world.

:transform-vector *vec* [method]

Regarding *vec* represented in this local coords, transforms it to the representation in the world.

:inverse-transform-vector *vec* [method]

vec represented in the world is inversely transformed into the representation in this local coords.

:inverse-transformation [method]

makes an instance of coordinates which represents inverse transformation of this coord.

:transform *trans &optional (wrt :local)* [method]

:translate *fltvec &optional (wrt :local)* [method]

:locate *fltvec &optional (wrt :local)* [method]

:rotate *theta axis &optional (wrt :local)* [method]

:orient *theta axis &optional (wrt :local)* [method]

Refer to the descriptions in class **coordinates**.

make-coords *&key pos rot rpy Euler angle axis 4X4 coords name* [function]

make-cascoords *&key pos rot rpy Euler angle axis 4X4 coords name* [function]

coords *&key pos rot rpy Euler angle axis 4X4 coords name* [function]

cascoords *&key pos rot rpy Euler angle axis 4X4 coords name* [function]

All these functions make new coordinates or cascaded-coordinates. For the keyword parameter, see **:init** method of class coordinates.

transform-coords *coords1 coords2 &optional (coords3 (coords))* [function]

Coords1 is applied (multiplied) to the coords2 from the left. The product is stored in coords3.

transform-coords* *&rest coords* [function]

concatenates transformations listed in *coords*. An instance of coordinates that represents the concatenated transformation is returned.

wrt *coords vec* [function]

transforms *vec* into the representation in *coords*. The result is equivalent to `(send coords :transform-vector vec)`.

13.6 Relationship between transformation matrix and coordinates class

Relationship between transformation matrix and coordinates class is described, where a transformation matrix T represents a 4×4(homogeneous) matrix as below.

$$T = \begin{pmatrix} \mathbf{R}_T & \mathbf{p}_T \\ \mathbf{0} & 1 \end{pmatrix}$$

$\mathbf{R}_T$ is a 3×3 matrix, and $\mathbf{p}_T$ is a 3×1 matrix (a float-vector which has 3 elements in euslisp). Coordinates class has slot variables rot and pos. They are $\mathbf{R}_T$ and $\mathbf{p}_T$ respectively.

Getter method for rotation matrix and position

$\mathbf{R}$ and $\mathbf{p}$ can be obtained using methods of the coordinates class.

T is an instance of the coordinate class.

(send T :rot)
$\Rightarrow \mathbf{R}_T$

(send T :pos)
$\Rightarrow \mathbf{p}_T$

Methods for transforming vectors

$\mathbf{v}$ is 3-D position vector.

(send T :rotate-vector v)
$\Rightarrow \mathbf{R}_T\mathbf{v}$

(send T :inverse-rotate-vector v)
$\Rightarrow \mathbf{v}^T\mathbf{R}_T$

(send T :transform-vector v)
$\Rightarrow \mathbf{R}_T\mathbf{v} + \mathbf{p}_T$

Converts a vector represented in a local coordinate system T to a vector represented in the world coordinate system.

(send T :inverse-transform-vector v)
$\Rightarrow \mathbf{R}_T^{-1}(\mathbf{v} - \mathbf{p}_T)$

Converts a vector represented in the world coordinate system. to a vector represented in a local coordinate system T.

Methods returing coordinates without modifying itself

(send T :inverse-transformation)

$\Rightarrow T^{-1}$

Returns inverse matrix.

$$T^{-1} = \begin{pmatrix} \mathbf{R}_T^{-1} & -\mathbf{R}_T^{-1}\mathbf{p}_T \\ \mathbf{0} & 1 \end{pmatrix}$$

(send T :transformation A (&optional (wrt :local)))

when wrt == :local, $T^{-1}A$ is returned.

when wrt == :world, AT^{-1} is returned.

when wrt == W (coordinates class), $W^{-1}AT^{-1}W$ is returned.

Methods modifying itself

A is an instance of the coordinates class.

$\Leftrightarrow$ represents that slot variables (pos or rot) refer to a given instance (matrix or float vector). Please note that when one is changed, the other also reflects the change.

$\leftarrow$ represents substitution.

(send T :newcoords A)

$\mathbf{R}_T \Leftrightarrow \mathbf{R}_A$

$\mathbf{p}_T \Leftrightarrow \mathbf{p}_A$

(send T :newcoords R p)

$\mathbf{R}_T \Leftrightarrow \mathbf{R}$

$\mathbf{p}_T \Leftrightarrow \mathbf{p}$

(send T :move-to A (&optional (wrt :local)))

when wrt == :local,$T \leftarrow TA$

when wrt == :world,$T \Leftrightarrow A$

when wrt == W (coordinates class), $T \leftarrow WA$

(send T :translate v (&optional (wrt :local)))

when wrt == :local, $\mathbf{p}_T \leftarrow \mathbf{p}_T + \mathbf{R}_T\mathbf{v}$

when wrt == :world, $\mathbf{p}_T \leftarrow \mathbf{p}_T + \mathbf{v}$

when wrt == W (coordinates class), $\mathbf{p}_T \leftarrow \mathbf{p}_T + \mathbf{R}_W\mathbf{v}$

(send T :locate v (&optional (wrt :local)))

when wrt == :local, $\mathbf{p}_T \leftarrow \mathbf{p}_T + \mathbf{R}_T\mathbf{v}$

when wrt == :world, $\mathbf{p}_T \leftarrow \mathbf{v}$

when wrt == W (coordinates class), $\mathbf{p}_T \leftarrow \mathbf{p}_W + \mathbf{R}_W\mathbf{v}$

(send T :transform A (&optional (wrt :local)))

when wrt == :local, $T \leftarrow TA$

when wrt == :world, $T \leftarrow AT$

when wrt == W (coordinates class), $T \leftarrow (WAW)^{-1}T$

14 Geometric Modeling

EusLisp adopts *Brep* (Boundary Representation) as the internal representation of 3D geometric models. Components in Breps are represented by classes **edge, plane, polygon, face, hole,** and **body**. Primitive body creating functions and body composition functions create new instances of these classes. In order to use your private geometric classes having more attributes, set special variables ***edge-class*, *face-class*** and ***body-class*** to your class objects.

14.1 Miscellaneous Geometric Functions

vplus *vector-list* [function]

returns a newly created float-vector that is the sum of all the elements of *vector-list*. The difference from **v+** is that **vplus** computes the sum of more than two arguments and no result vector can be specified.

vector-mean *vector-list* [function]

returns the mean vector of *vector-list*.

triangle *a b c &optional (normal #f(0 0 1))* [function]

a, b, c are float-vectors representing 2 or 3 dimensional points. normal is the normal vector of the plane on which *a,b*, and *c* lie. **Triangle** returns 2*area of a triangle formed by *a,b,c*. **Triangle** is positive if *a,b*, and *c* turn clockwise when you are looking in the same direction as *normal*. In other words, if **triangle** is positive, *c* locates at the left hand side of line *a-b*, and *b* lies at the right side of *ac*.

triangle-normal *a b c* [function]

finds a normal vector which is vertical to the triangle defined by three points *a,b,*and *c*.

vector-angle *v1 v2 &optional (normal (v* v1 v2))* [function]

Computes an angle between two vectors, denoted by $atan(normal \cdot (v1 \times v2), v1 \cdot v2)$. *v1,v2* and *normal* must be normalized vectors. When *normal* is not given, a normalized vector commonly perpendicular to *v1* and *v2* is used, in which case the result is always a positive angle in the range between 0 and π. In order to obtain a signed angle, *normal* must be specified explicitly.

face-normal-vector *vertices* [function]

Computes surface normal vector from a list of float-vectors which lie on the same plane.

farthest *p points* [function]

finds the farthest point from *p* in the list of 3D float-vectors, *points*.

farthest-pair *points* [function]

finds the farthest point pair in the list of 3D float-vectors, *points*.

maxindex *3D-floatvec* [function]

Finds the index of the absolute maximum value of three elements.

random-vector *&optional (range 1.0)* [function]

Generates a random vector which is distributed homogeneously in 3D Cartesian space.

random-normalized-vector *&optional (range 1.0)* [function] returns a normalized-3D random vector.

random-vectors *count range* [function]

returns a list of random vectors.

line-intersection *p1 p2 p3 p4* [function]

p1, p2, p3 and *p4* are all float-vectors of more than two dimensions. *p1-p2* and *p3-p4* define two lines on a plane. **line-intersection** returns a list of two parameters of the intersection point for these two lines. When used in three dimension, *p1, p2, p3* and *p4* must be coplanar.

collinear-p *p1 p2 p3 &optional tolerance* [function]

p1, p2, p3 are all three-dimensional float-vectors representing three point locations. **Collinear-p** returns the parameter for *p2* on the line *p1-p3* if $norm((p2 - p1) \times (p3 - p1))$ is smaller than `*coplanar-threshold*`, otherwise NIL.

find-coplanar-vertices *p1 p2 p3 vlist* [function]

p1, p2, p3 are all three-dimensional float-vectors representing a plane. **Find-coplanar-vertices** looks for coplanar points in *vlist* that lie on the plane.

find-connecting-edge *vertex edgelist* [function]

finds an edge in *edgelist* that connects to *vertex*.

make-vertex-edge-htab *bodfacs* [function]

bodfacs is a body or a list of faces. **make-vertex-edge-htab** makes a hash-table which allows retrieving of edges connected to a vertex.

left-points *points p1 p2 normal* [function]

Assume *points, p1*, and *p2* lie on the plane whose normal vector is *normal*. **Left-points** searches in *points* and collects ones lying in the left hand side of the line passing on *p1, p2*.

right-points *points p1 p2 normal* [function]

Assume *points, p1*, and *p2* lie on the plane whose normal vector is *normal*. **Right-points** searches in *points* and collects ones lying in the right hand side of the line determined by *p1, p2*.

left-most-point *points p1 p2 normal* [function]

Assume *points, p1*, and *p2* lie on a plane whose normal vector is *normal*. **left-points** searches in *points* which lie in the left-hand side of the line determined by *p1, p2* and returns the farthest one.

right-most-point *points p1 p2 normal* [function]

Assume *points, p1*, and *p2* lie on a plane whose normal vector is *normal*. **right-most-point** searches in *points* which lie in the right-hand side of the line determined by *p1, p2* and returns the farthest one.

eps= *num1 num2 &optional (tolerance *epsilon*)* [function]

compares two float numbers *num1* and *num2* for equality with the tolerance of ***epsilon***.

eps< *num1 num2 &optional (tolerance *epsilon*)* [function]

returns T if $num1$ is apparently less than $num2$, i.e. $num1 < num2 - tolerance$.

eps<= *num1 num2 &optional (tolerance *epsilon*)* [function]

returns T if $num1$ is possibly less than or equal to $num2$, i.e. $num1 < num2 + tolerance$.

eps> *num1 num2 &optional (tolerance *epsilon*)* [function]

returns T if $num1$ is apparently greater than $num2$, i.e. $num1 > num2 + tolerance$.

eps>= *num1 num2 &optional (tolerance *epsilon*)* [function]

returns T if $num1$ is possibly greater than or equal to $num2$, i.e. $num1 > num2 - tolerance$.

bounding-box [Class]

:super	**object**
:slots	(minpoint maxpoint)

defines a minimal rectangular-parallel-piped which is bounded by the planes parallel to xy-, yz- and zx-planes. **Bounding-box** can be used in any dimension according to the dimension of vectors given at the initialization. Bounding-box had been defined by the name of surrounding-box.

:box [method]

returns this bounding-box object itself.

:volume [method]

returns the volume of this bounding box.

:grow *rate* [method]

increases or decreases the size of this box by the *rate*. When *rate* is 0.01, the box is enlarged by 1%.

:inner *point* [method]

returns T if *point* lies in this box, otherwise nil.

:intersection *box2 &optional tolerance* [method]

returns the intersectional bounding box of this box and *box2*. If *tolerance* is given, the box is enlarged by it. If there is no intersection, NIL is returned.

:union *box2* [method]

returns the union of bounding box of this box and *box2*.

:intersectionp *box2* [method]

returns T if this box has the intersection with the *box2*, NIL otherwise. This method is faster than **:intersection** because no new instance of bounding-box is created.

:extreme-point *direction* [method]

returns one of the eight corner points yielding the largest dot-product with *direction*.

:corners [method]

returns the list of all vertices of this box. If this box defines 2D bounding-box, then 4 points are returned, 3D, 8, and so on.

:below *box2 &optional (direction #(0 0 1)* [method]

returns T if this box is below *box2* in *direction*. This is used to check whether two box intersects when this box is moved toward *direction*.

:body [method]

returns a body object that represents a cube bounded by this box.

:init *vlist &optional tolerance* [method]

sets minpoint and maxpoint slots looking in *vlist*. If tolerance (float) is specified, the box is grown by the amount.

make-bounding-box *points &optional tolerance* [function] finds the minimum and maximum coordinates in the list of *points*, and make an instance of `bounding-box`.

bounding-box-union *boxes &optional (tolerance *contact-threshold*)* [function]

makes an instance of the surrounding-box representing the union of *boxes*. The resulted box is expanded by the *tolerance*.

bounding-box-intersection *boxes &optional (tolerance *contact-threshold*)* [function]

makes an instance of the surrounding-box representing the intersection of *boxes*. The resulted box is expanded by the *tolerance*.

14.2 Line and Edge

The direction of the vertex loop or the edge loop is defined so that the vertices or edges are arranged in the counter-clockwise order when the body is observed from outside. *Pvertex* and *nvertex*, and *pface* and *nface* are determined so that an edge is oriented from *pvertex* toward *nvertex* when *pface* is located at the left of the edge observing them from outside.

line [Class]

:super **propertied-object**
:slots ((pvert :type float-vector) (nvert :type float-vector))

defines a line passing on pvert and nvert. The line is directed from *pvert* to *nvert* in the parametric representation: $t \cdot pvert + (1-t)nvert$.

:vertices [method]

returns the list of *pvert* and *nvert*.

:point *p* [method]

returns a three dimensional float-vector that corresponds to the *p* parameter on this line. $parameter \cdot pvert + (1 - parameter)nvert$

:parameter *point* [method]

Computes the parameter for *point* on this line. This is the inverse method of `:point`.

:direction [method]

returns a normalized vector from **pvert** to **nvert**.

:end-point *v* [method]

returns the other end-point of this line, i.e. if *v* is *eq* to `pvert`, `nvert` is returned, if *v* is *eq* to `nvert`, `pvert` is returned, otherwise NIL.

:box [method]

creates and returns a **bounding-box** of this line.

:boxtest *box* [method]

checks intersection between *box* and the bounding-box of this line.

:length [method]

returns the length of this line.

:distance *point-or-line* [method]

returns the distance between the *point-or-line* and this line. If the foot of the vertical line from the *point* to this line does not lie between pvertex and nvertex, the distance to the closest end-point is returned. Using this method to calculate the distance between two lines, interference between two cylinders can be tested.

:foot *point* [method]

finds the parameter for the point which is the foot of the vertical line from *point* to this line.

:common-perpendicular *l* [method]

finds the line which is vertical both to this line and to *l* and returns a list of two 3D float-vectors.

:project *plane* [method]

returns a list of two points that are the projection of pvert of nvert onto *plane*. When two lines are in parallel and a common perpendicular line cannot be determined uniquely, `parallel` is returned.

:collinear-point *point &optional (tolerance *coplanar-threshold*)* [method]

checks whether *point* is collinear to this line with the tolerance of *tolerance* using **collinear-p**. If *point* is collinear to this line, the parameter for the point on the line is returned, otherwise NIL.

:on-line-point *point &optional (tolerance *coplanar-threshold*)* [method]

checks whether the *point* is collinear to this line, and the *point* lies on the part of the line between *pvert* and *nvert*.

:collinear-line *ln &optional (tolerance *coplanar-threshold*)* [method]

checks if *ln* is collinear to this line, i.e. if the two end-points of *ln* lie on this line. T or NIL is returned.

:coplanar *ln &optional (tolerance *coplanar-threshold*)* [method]

checks if this line and *ln* are coplanar. Two end-points of this line and one end-point of *ln* defines a plane. If another end-point of *ln* is on the plane, T is returned, otherwise NIL.

:intersection *ln* [method]

ln is a line coplanar with this line. **:Intersection** returns a list of two parameters for the intersection point of these two lines. A parameter may be any float number, but a parameter between 0 and 1 means an actual intersection on the line segmented by two end-points. NIL if they are in parallel.

:intersect-line *ln* [method]

ln is a line coplanar with this line. Two parameters of the intersecting point is returned along with symbolic information such as `:parallel, :collinear,` and `:intersect`.

edge [Class]

:super	**line**
:slots	(pface nface
	(angle :type float)
	(flags :type integer))

represents an edge defined as the intersection between two faces. Though *pface* and *nface* are statically defined in the slots, their interpretations are relative to the direction of this edge. For example, *pface* represents the correct pface when this edge is considered to goes from *pvert* toward *nvert*. So, pvert and nvert in your interpretation must be given to the **:pface** and **:nface** methods to select the appropriate face.

make-line *point1 point2* [function]

creates an instance of **line** whose pvert is *point1* and nvert is *point2*.

:pvertex *pf* [method]

returns pvertex when *face* is regarded as the pface of this edge.

:nvertex *face* [method]

returns nvertex regarding *face* as the pface of this edge.

:body [method]

returns the body object that defines this edge.

:pface *pv nv* [method]

returns pface when the *pv* and *nv* are interpreted as the virtual pface and nface of this edge, respectively.

:nface *pv nv* [method]

returns nface when the *pv* and *nv* are interpreted as the virtual pface and nface of this edge, respectively.

:binormal *aface* [method]

finds the direction vector which is perpendicular both to this line and to the normal of *aface*.

:angle [method]

returns the angle between two faces connected with this edge.

:set-angle [method]

computes the angle between two faces connected with this edge and stores it in the angle slot.

:invert [method]

:set-face *pv nv f* [method]

sets the *f* face as a pface regarding *pv* as the pvertex and *nv* as the nvertex. Note that this may change either pface or nface of this edge.

:contourp *viewpoint* [method]

T if this is a contour edge, i.e., either pface or nface of this edge is visible and the other is invisible from *viewpoint*.

:approximated-p [method]

T if this edge is an approximated edge representing curved surface like the side of a cylinder. Approximated edges are needed to represent curves by segmented straight lines.

:set-approximated-flag *&optional (threshold 0.7)* [method]

In EusLisp, every curved surface is approximated with many planar faces. The LSB of `flags` is used to indicate that the faces on the both sides of this edge are curved faces. **:set-approximated-flag** sets this flag to T, if the angle between two faces is greater than *threshold*.

:init *&key pface nface pvertex nvertex* [method]

14.3 Plane and Face

A plane object is represented by the normal vector on the plane and the distance from the coordinates origin to the plane. Two pairs of such normal vectors and distances are recorded in a plane object. One represents the current status after transformations, while the other represents the original normal and distance when the plane is defined.

plane [Class]

:super **propertied-object**
:slots ((normal :type float-vector)
(distance :float))

defines plane-equation. A plane is considered to have no boundaries and extend infinitely.

:normal [method]

returns this polygon's normal vector which is always normalized.

:distance *point* [method]

computes distance between this plane and *point*.

:coplanar-point *point* [method]

returns T if *point* lies on this plane.

:coplanar-line *line* [method]

returns T if *line* lies on this plane.

:intersection *point1 point2* [method]

computes the intersection point between this plane and the line determined by two end points, *point1* and *point2*, and returns the parameter for the intersection on the line. If the line and this plane are parallel, **:parallel** is returned.

:intersection-edge *edge* [method]

Returns the parameter of the intersection point for this plane and a line represented by point1 and point2, or edge.

:foot *point* [method]

Returns a 3D vector which is the orthogonally projection of *point* onto this plane.

:init *normal point* [method]

Defines a plane with the *point* on the plane and the *normal* vector. *Normal*must be a normalized vector, $|normal| = 1$.

polygon [Class]

:super **plane**
:slots (convexp edges vertices
(model-normal float-vector)
(model-distance :float))

Polygon represents a loop on a plane. *Convexp* is a boolean flag representing the convexity of the loop. *Edges* is a list of edges forming the contour of this loop, and *vertices* is a list of vertices.

:box *&optional tolerance* [method]

returns a bounding-box for this polygon.

:boxtest *box2 &optional tolerance* [method]

makes a bounding-box for this polygon, and returns the intersection of the bounding-box and *box2*. If there is no intersection, NIL is returned.

:edges [method]

returns the list of edges (circuit) of this polygon. The list is ordered clockwise when the polygon is viewed along the normal vector of this plane. If you think of the normal vector as a screw, the edges are ordered in the rotation direction for the screw to screw in. When polygon or face is used for the surface representation of a solid object, the normal vector is directed to its outside region. When a polygon is viewed from the outside of the object, edges are ordered counter-clockwise.

:edge *n* [method]

returns the *n*-th element of edges.

:vertices [method]

returns the vertices of this polygon ordered in the same manner as edges. Note that the first vertex is copied duplicatedly at the end of the list and the list is always longer by one than the actual number of vertices. This is for the ease of edge traversal by using the vertices list.

:vertex *n* [method]

returns the *n*-th element of vertices.

:insidep *point &optional (tolerance *epsilon*)* [method]

returns :inside, :outside or :border according to the location of *point* relative to this region.

:intersect-point-vector *point vnorm* [method]

Computes the intersection with the semi-line defined by the *point* and the normalized direction vector, *vnorm*.

:intersect-line *p1 p2* [method]

Computes intersection point with a line specified by *p1* and *p2*. The result is nil(no intersection) or list of the parameter and the intersection position.

:intersect-edge *edge* [method]

Computes intersection point with a line specified by the *edge*. The result is nil(no intersection) or list of the parameter and intersection position.

:intersect-face *aregion* [method]

Returns T if this region intersects with aregion.

:transform-normal [method]

:reset-normal [method]

recomputes the surface normal vector of this polygon from the current *vertices* list.

:invert [method]

:area [method]

returns the area of this polygon.

:init *&key vertices edges normal distance* [method]

face [Class]

:super **polygon**
:slots (holes mbody primitive-face id)

defines a face which may have holes. *Pbody* and *type* represent the primitive body and the type `(:top, :bottom, :side)` of the face in the body.

:all-edges [method]

:all-vertices [method]

Returns all the edges or vertices of the contour of this face and all the inner loops (holes). Note that **:edges** and **:vertices** methods only return edges and vertices composing the contour.

:insidep *point* [method]

decides whether the point is inside of this face or not. If the point is inside the outer contour of this face but also inside the loop of any holes, it is classified as outside.

:area [method]

returns the area of this face, that is the area surrounded by external edges subtracted by the areas of holes.

:centroid *&optional point* [method]

returns a list of the floating-point number and the float-vector representing the center-of-gravity of this face. If *point* is not given, the first number represents the area of this polygon, and the second float-vector the location of center-of-gravity of this polygon. If *point* is given, it is taken as the top vertex of the cone whose bottom face is formed by this polygon, and the volume of this cone and its center-of-gravity are returned.

:invert [method]

flips the direction of this face. The normal vector is inverted, and the order of edge loop is reversed.

:enter-hole *hole* [method]

adds a hole in this face.

:primitive-body [method]

returns the primitive-body which has defined this face.

:id [method]

returns one of `(:bottom), (:top)` and `(:side seq-no.)`.

:face-id [method]

returns a list of the type of primitive-body and the face type. For example, a side face of a cylinder returns `((:cylinder` *`radius height segments`*`) :side` *`id`*`)`.

:body-type [method]

returns primitive body which has defined this face.

:init *&key normal distance edges vertices holes* [method]

hole [Class]

:super	**polygon**
:slots	(myface)

hole is a polygon representing an inner loop of a face. A face may have a list of holes in its `holes` slot.

:face [method]

returns a face that contains this hole.

:enter-face *face* [method]

makes a link to a face which surrounds this hole. This method is only used in conjunction with the `:enter-hole` method of the face class.

:init *&key normal distance edges vertices face* [method]

14.4 Body

body [Class]

:super **cascaded-coords**

:slots (faces edges vertices model-vertices box convexp evertedp csg)

defines a three dimensional shape.

:magnify *rate* [method]

changes the size of this body by *rate*. Magnification is recorded in csg list.

:translate-vertices *vector* [method]

translates model-vertices. *Vector* should be given in the local coordinates. Translation is recorded in csg list.

:rotate-vertices *angle axis* [method]

rotates model-vertices *angle* radian around *axis*. Rotation is recorded in csg list.

:reset-model-vertices [method]

:newcoords *rot &optional pos* [method]

changes coordinates. If pos is not given, rot is given as a instance of coordinate class.

:vertices [method]

returns the list of all vertices of this body.

:edges [method]

returns the list of all edges of this body.

:faces [method]

returns the list of all the faces composing this body.

:box [method]

returns the bounding-box of this body.

:Euler [method]

calculates Euler number of this body, that is $faces + vertices - edges - 2 - holes$. This should equal to $-2rings$.

:perimeter [method]

returns the sum of length of all the edges.

:volume *&optional (reference-point #f(0 0 0))* [method]

returns the volume of this body.

:centroid *&optional (point #f(0 0 0)* [method]

returns the location of center-of-gravity assuming that this body is homogeneously solid.

:possibly-interfering-faces *box* [method]

:common-box *body* [method]

Returns common minimal box for this body and another body. If there is interference between two bodies, the intersection must exist in this common-box.

:insidep *point* [method]

returns **:inside** if *point* resides in this body, **:border** if *point* lies on a surface of this body, and **:outside** otherwise.

:intersect-face *face* [method]

returns T if there is an interference between the faces of this body and *face*.

:intersectp *body* [method]

Checks intersection with another body.

:evert [method]

reverse the directions of all the faces and edges so that the inside of this body becomes outside.

:faces-intersect-with-point-vector *point direction* [method]

collects all faces that intersect with a vector casted from *point* towards em direction.

:distance *target* [method]

target may either be a float-vector or a plane object. **:distance** finds the closest face from *target* and returns a list of the face and the distance.

:csg [method]

returns csg body construction history.

:primitive-body [method]

returns a list of primitive bodies which have constructed this body.

:primitive-body-p [method]

T if this body is a primitive body created by one of functions listed in **??**.

:creation-form [method]

returns a Lisp expression to create this body.

:body-type [method]

returns a list of creation parameters if this body is a primitive body, or an expression indicating this body is a complex (composed) body.

:primitive-groups [method]

returns a list of two elements. The first is a list of primitive bodies that is added (body+) to compose this body. The latter is a list of subtracted primitive-bodies.

:get-face *&optional body face id* [method]

body is an instance of body that has composed this body, one of primitive-body types such as :cube, :cylinder, :prism, :cone, :solid-of-revolution, etc., or nil. If neither *face* nor *id* is given, all the faces that matches *body* is returned. If *face* is given, further filtering is performed. *face* must be one of `:top`, `:bottom` and `:side`. `(send abody :get-face :cylinder :top)` returns all the top faces of cylinders that compose `abody`. If *face* is `:side`, you can pick up faces that are numbered as *id*. `(send abody nil :side 2)` returns all the third (id begins from zero) side faces for any primitive-type bodies.

:init *&key faces edges vertices* [method]

initializes this body from *:faces*. *:face* is a required argument. Since face, edge and vertex must maintain consistent relation to define a complete solid model, it is meaningless to call this method with inconsistent arguments. In order to create bodies, use the primitive body creating functions described in section **??** and the body composition functions in section **??**.

:constraint *b* [method]

returns self's constraint when self is in contact with *b*. Details of are given in section **??**.

14.5 Primitive Body Creation

make-plane *&key normal point distance* [function]

Makes a plane object which is oriented to *normal*, and passes *point*. Instead of giving *point*, *distance* can be specified.

xy-plane [variable]

yz-plane [variable]

zx-plane [variable]

make-cube *xsize ysize zsize &key name color* [function]

makes a cuboid whose sizes in x, y and z directions are *xsize, ysize* and *zsize*. The coordinates origin of this cuboid locates at the center of the body.

make-prism *bottom-points sweep-vector &key name color* [function]

Makes a prism by lifting the shape defined by *bottom-points* along *sweep-vector*. If the *sweep-vector* is a number, not a float-vector, it is taken as the height of the prism in the z direction. Bottom points must be ordered as they define the bottom face of the body. For example, (make-prism '(#f(1 1 0) #f(1 -1 0) #f(-1 -1 0) #f(-1 1 0)) 2.0) makes a cube of height 2.0.

make-cylinder *radius height &key (segments 12) name color* [function]

Makes a cylinder with specified *radius* and *height*. The bottom face is defined on xy-plane and the coordinates origin is located at the center of the bottom face.

make-cone *top bottom &key (segments 16) color name* [function]

makes a cone body whose summit is the *top* and bottom face is the *bottom*. *Top* is a 3D float-vector. *Bottom* is either a list of vertices of the bottom face or a radius (scalar). If it is the vertices list, it is order sensitive. `(make-cone #f(0 0 10) (list #f(10 0 0) #f(0 10 0) #f(-10 0 0) #f(0 -10 0)))` makes a cone of a square bottom.

make-solid-of-revolution *points &key (segments 16) name color* [function]

Points are revolted along z-axis in the clock wise direction. If two end points in the *points* list do not lie on z axis, those points make circular faces. Thus, `(make-solid-of-revolution '(#f(0 0 1) #f(1 0 0)))` makes a cone, and `(make-solid-of-revolution '(#f(1 0 1) #f(1 0 0)))` makes a cylinder. The *points* are order-sensitive, and are expected to be arranged from higher z coordinate to lower z.

make-torus *points &key (segments 16) name color* [function]

makes a torus, a donuts like object. *Points* is a list of vertices on a cross-section.

make-icosahedron *&optional (radius 1.0)* [function]

Makes a regular body of twenty faces. Each face is a regular triangle.

make-dodecahedron *&optional (radius 1.0)* [function]

Makes a regular body of twelve faces. Each face is a regular pentagon.

make-gdome *abody* [function]

By subdividing triangle faces of abody into four subfacets, makes a geodesic dome as a new body. Abody should be an icosahedron initially, and then the result of make-gdome can be given to make-gdome recursively. At each call, the number of faces of the Gdome increases by the factor of four, i.e. 20, 80, 320, 1280, 5120, etc.

```
(setq g0 (make-icosahedron 1.0)) ; 20 facets
(setq g1 (make-gdome g0)) ; 80 facets
(setq g2 (make-gdome g1)) ; 320 facets
...
```

grahamhull *vertices &optional (normal #f(0 0 1))* [function]

Computes convex-hull for 2D points by Graham's algorithm. Slower than quickhull.

quickhull *vertices &optional (normal #f(0 0 1))* [function]

Computes convex-hull for 2D points by the binary search method.

convex-hull-3d *vertices* [function]

Computes convex-hull for 3D points by gift-wrapping method.

make-body-from-vertices *vertices-list* [function]

creates a body from lists of vertices each of which define a loop of a face in the consistent order.

14.6 Body Composition

face+ *face1 face2* [function]

face* *face1 face2* [function]

face1 and *face2* are coplanar faces in 3D space. **face+** composes union of these faces and returns a face object. If there is no intersection, original two faces are returned. **face*** returns intersection of these faces. If there is no intersection, NIL is returned.

cut-body *body cutting-plane* [function]

Cuts a body by the *cutting-plane* and returns a list of faces made at the cross-section.

body+ *body1 body2 &rest more-bodies* [function]

body- *body1 body2* [function]

body* *body1 body2* [function]

Computes join, difference or intersection of two or more bodies. Each body is copied before each **body+, body-** and **body*** operation, and original bodies are unchanged. The new coordinates of the resulted body is located and oriented at the same location and orientation as the world coordinates. Even when two bodies are touching face by face, these functions are expected to work correctly if threshold parameters `*coplanar-threshold*`, `*contact-threshold*`, and `*parallel-threshold*` are properly set. However, if a vertex of a body is in contact with an edge or a face of the other body, any composition operation fails.

body/ *body plane* [function]

Cut the body by a plane which is an instance of class plane (made by **make-plane**). A newly created body is returned.

body-interference *&rest bodies* [function]

Checks interference between each one-to-one combination in *bodies*. Returns a list of two bodies that are intersecting.

14.7 Coordinates-axes

Class coordinates-axes defines 3D coordinates-axes drawable on a screen. Each axis and an arrow at the tip of z-axis are defined by line objects. Since the coordinates-axes class inherits cascaded-coords, a coordinates-axes object can be attached to another cascaded-coords originated object such as a body. This object is used to see the coordinates-axes of a body or a relative coordinates to another coordinates.

coordinates-axes [Class]

:super	**cascaded-coords**
:slots	(size model-points points lines)

Defines drawable 3-D coordinates-axes.

14.8 Bodies in Contact

The method and functions described in this subsection require **contact/model2const.l, contact/inequalities.l, contact/drawconst.l**.

constrained-motion *c* [function]

returns the possible motions which satisfy the constraint *c*.

constrained-force *m* [function]

returns the force which is applicable from the constrained body to the constraining body.

draw-constraint *c* [function]

draws the constraint *c*.

draw-motion *m a b* [function]

draws the possible motions of *a* in contact with *b*. Type the return key for drawing.

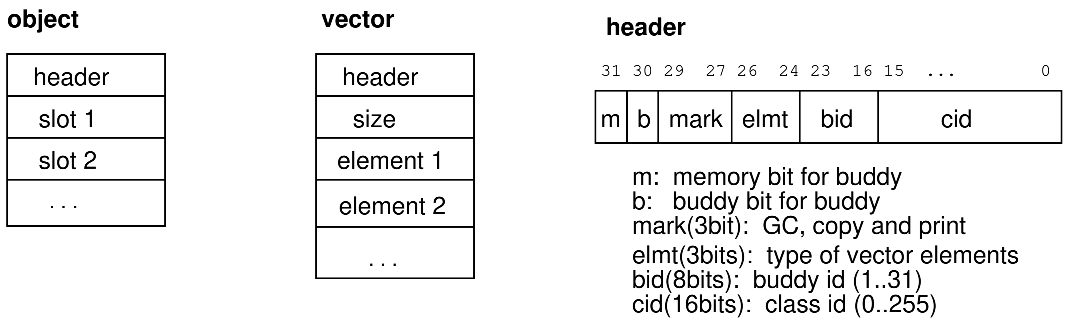
Example

```
;;
;;      peg in a hole with 6 contact points
;;
(in-package "GEOMETRY")
(load "view")
(load "../model2const.l" :package "GEOMETRY")
(load "../inequalities.l" :package "GEOMETRY")
(load "../drawconst.l" :package "GEOMETRY")

(setq x (make-prism '(#f(50 50 0) #f(50 -50 0) #f(-50 -50 0) #f(-50 50 0))
                    #f(0 0 200)))
(setq x1 (copy-object x))
(send x1 :translate #f(0 0 -100))
(send x1 :worldcoords)
(setq a1 (make-prism '(#f(100 100 -150) #f(100 -100 -150)
                       #f(-100 -100 -150) #f(-100 100 -150))
                     #f(0 0 150)))
(setq ana (body- a1 x1))
(send x :translate #f(0 -18.30127 -18.30127))
(send x :rotate -0.523599 :x)
(send x :worldcoords)

(setq c (list (send x :constraint ana)))
(setq m (constrained-motion c))
(setq f (constrained-force m))

(hidd x ana)
(draw-constraint c)
(draw-motion m)
```


object
header
slot 1
slot 2
. . .
vector
header
size
element 1
element 2
. . .
header
31 30 29 27 26 24 23 16 15 ... 0
m
b
mark
elmt
bid
cid
m: memory bit for buddy
b: buddy bit for buddy
mark(3bit): GC, copy and print
elmt(3bits): type of vector elements
bid(8bits): buddy id (1..31)
cid(16bits): class id (0..255)

```
object
     cons
          queue
     propertied-object
          symbol   -----  foreign-pod
          package
          stream
               file-stream
               broadcast-stream
          io-stream ---- socket-stream
          metaclass
               vectorclass
                    cstructclass
          read-table
          array
          thread
          barrier-synch
          synch-memory-port
          coordinates
               cascaded-coords
                    body
                    sphere
                    viewing
                         projection
                              viewing2d
                              parallel-viewing
                              perspective-viewing
                    coordinates-axes
               viewport
          line --- edge --- winged-edge
          plane
               polygon
                    face
                    hole
               semi-space
          viewer
          viewsurface  ----- tektro-viewsurface
     compiled-code
          foreign-code
          closure
          load-module
     label-reference
     vector
          float-vector
          integer-vector
          string
               socket-address
               cstruct
          bit-vector
          foreign-string
     socket-port
     pathname
     hash-table
     surrounding-box
     stereo-viewing
```

図 4: Hierarchy of Predefined Classes

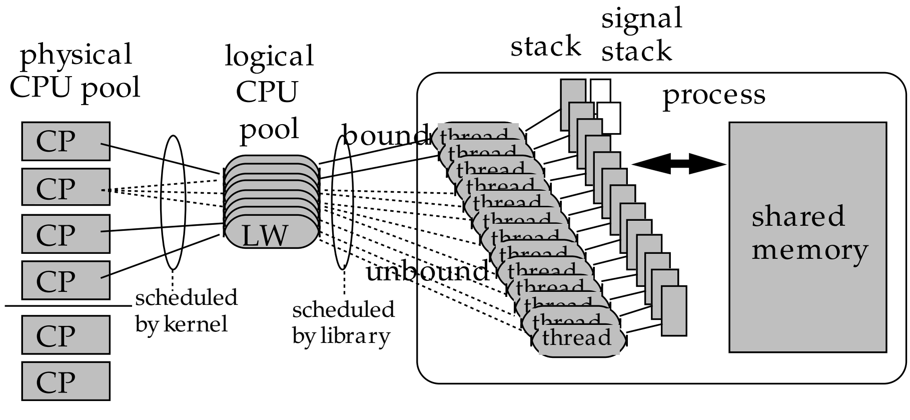


図 5: Solaris operating system's thread model

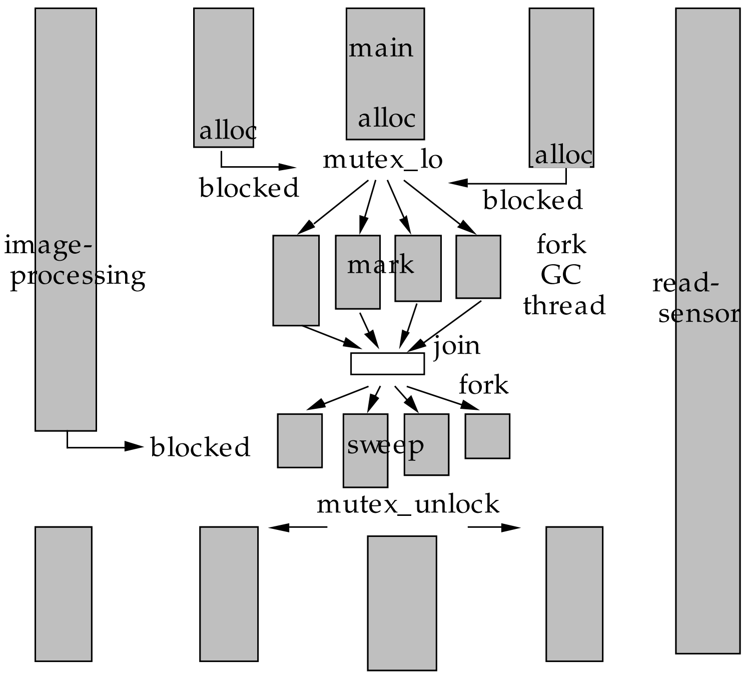


図 6: Parallel threads requesting memory and GC running in parallel

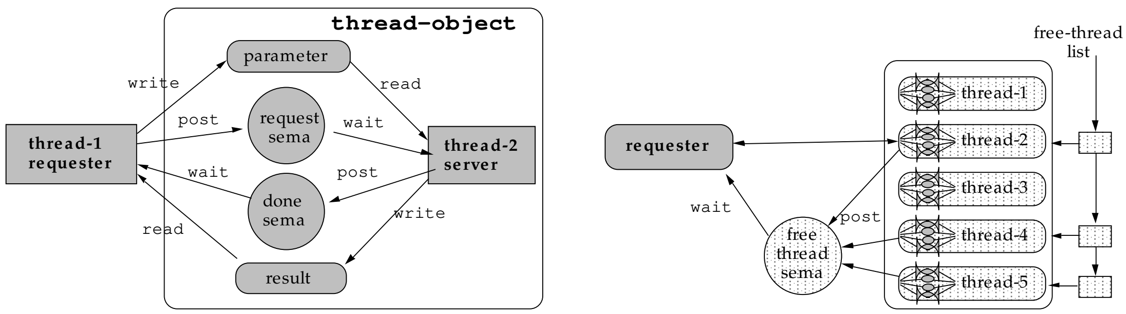


図 7: Thread-object for transferring control and data between threads (left) and the collection of threads put in the thread-pool.

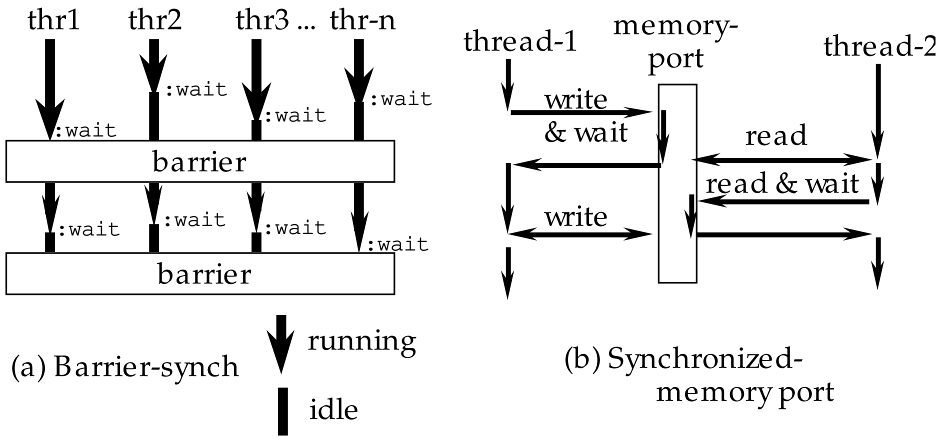


図 8: Barrier synchronization and synchronozed memory port

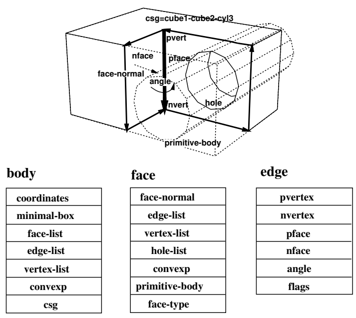


図 9: Arrangements of vertices, edges, and faces

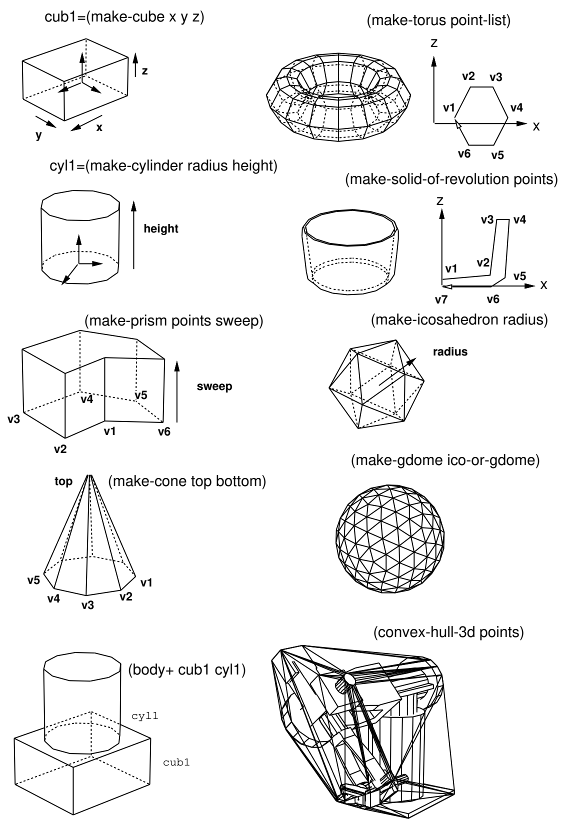


図 10: primitive bodies

The following figures shows examples of constraints. The small arrows in the figures designate the constraints for the pegs.

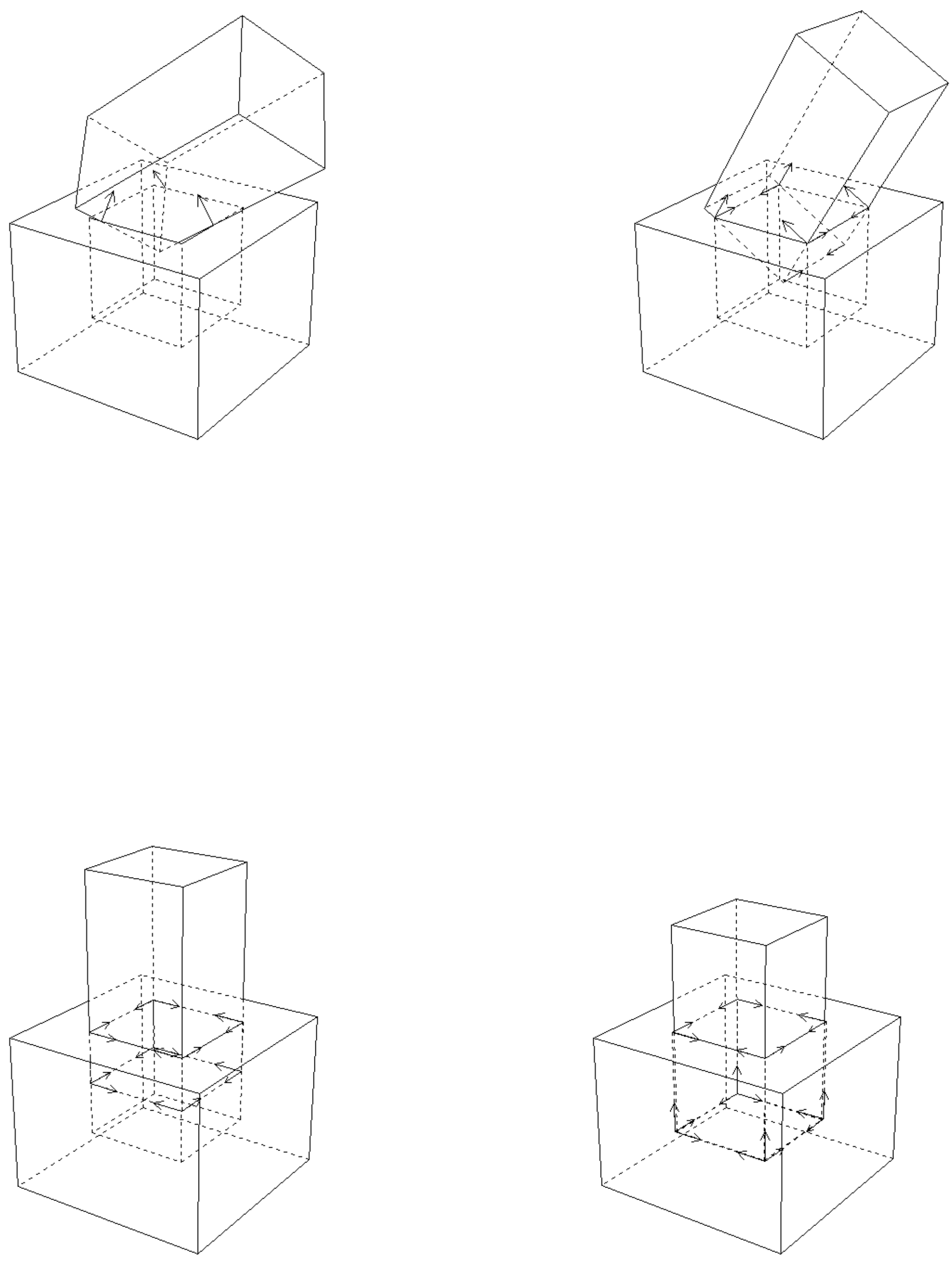


図 11: Constraints for a peg in a hole.

The following figures shows an example of the possible motions of a peg in a hole. The example corresponds to the above program.

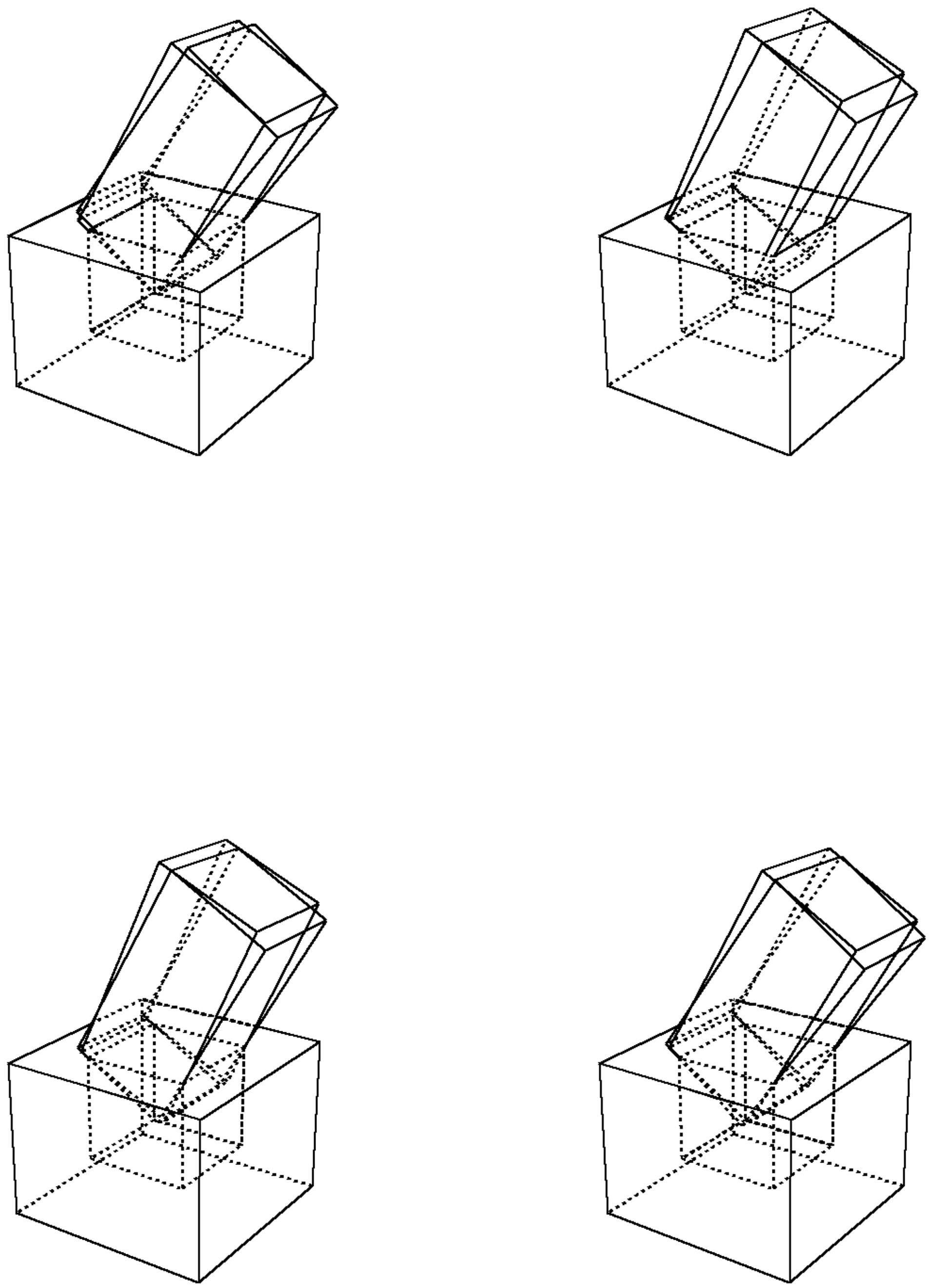


図 12: Possible motions of a peg in a hole

14.9 Voronoi Diagram of Polygons

Author: Philippe PIGNON, ETL Guest Researcher

The program is written in COMMON LISP. I used the method of Fortune, "A sweepline algorithm for Voronoi diagrams", in Proceedings of the 2nd Annual ACM symposium on computational geometry, 1986, 313-322. I adapted it to the polygonal case. This is a sample file with short explanations This program was written under Electrotechnical EUSLISP environment, so graphic connections are provided for it. However, you can use it with any COMMON-LISP; you'll then have to write your own display functions to replace those given in utilities.l file (see below)

PURPOSE: Computation of the voronoi diagram of a set of polygons. Please read the above quoted reference to understand the vocabulary and method used. No explanations about the program itself will be given here.

INPUT: A list of polygons coordinates plus an enclosing frame.

```
DATA= (
       (x11 y11 x12 y12 x13 y13 ...) first polygon,
                                     counterclocwise enumeration of vertices
       (x21 y21 x22 y22 x23 y23 ...) second polygon
               ...
       (xn1 yn1 xn2 yn2 xn3 yn3 ...) nth polygon

       (xf1 yf1 xf2 yf2 xf3 yf3 xf4 yf4) enclosing frame
      )
```

Enclosing frame can occur anywhere in data, and should be clockwise enumerated for outside-inside marking consistency (see below). Polygons must be simple, non intersecting. Aligned or flat edges are not accepted. Neither are isolated points or segments.

OUTPUT: *diagram*: a list of doubly connected edges list (cf utilities.l file). Each edge is a symbol, with property list including the following fields:

```
(start <pointer to a vertex>)
       (end <pointer to a vertex>)
       (pred <pointer to an edge>)
       (succ <pointer to an edge>)
       (left <pointer to a site>)
       (right <pointer to a site>)
       (type <:endpoint or :point-point or :segment-segment or :point-segment>)
       (outflag <t or nil>)
```

A *vertex* is a symbol whose property list contains the field "pos". This field itself contains a cons (xy), (real) planar coordinates of the vertex. *Pred* and *succ* field give counterclockwise predecessor and successor according to the dcel formalism (see Shamos and Preparata, Computational Geometry: An introduction, 1985, pp 15-17). A *site* is also a symbol, whose property list also contains relevant information. Sites describe original input data; they can be of type :point (a polygon vertex) or segment (a polygon edge).

Type is the gender of the bisector, determined by the type of the sites it separates. By convention, outside is the right side of a start-end edge. The voronoi diagram computes ouside as well as inside bisectors. Sort on outflag to keep the ones you want.

SAMPLE: In order to run the program on a short sample, please perform the following steps:

0- Copy the following files in your environment:

utilities.l	Geometric utility functions, plus EUSX graphic functions
polygonalvoronoi.l	The program.
testdata.l	Demonstration data, with the above format.

1- If you do not use EUS, edit the utilities.l file and modify the "compatibility package" according to the instructions.

2- Compile and/or load the following 3 files:

utilities.l	
polygonalvoronoi.l	
testdata.l	This file contains demonstration data,with the above format

3- (pv demoworld) run the program on demonstration data. The global variable *diagram* contains the bisectors of the voronoi diagram.

Under EUSX only (eus with XWindow interface), do the following to display the resulting diagram:

```
(make-display)          ;;Initializes the *display* window object
(dps demoworld *thick*) ;; Shows original data in thick lines
(dbs *diagram*)         ;; Shows the result
```

pv *data* [function]

Compute the Voronoi diagram of polygons from the *data* with the above format.

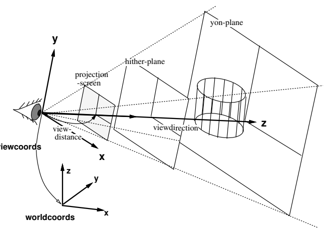


図 13: viewing coords and projection planes

15 Viewing and Graphics

15.1 Viewing

A viewing object manages viewing coordinate system whose origin is located at the position of a virtual camera, -z axis is oriented to the objects observed, and xy-plane is the projection screen. Since viewing inherits class cascaded-coords, it accepts coordinates transformation message such as **:translate, :rotate** and **:transform**. Also, it can be attached to another object derived from cascaded-coords, allowing the simulation of the camera-on-mobile-object system. The main purpose of viewing is to transform vectors represented in the world to the camera coordinates system. The transformation is taken in the opposite direction against usual coordinate transformation where vectors in the local coordinates are transformed into the representation in the world. Therefore, viewing holds the inversed left-handed transformation in the viewcoords slot, which is usually referred as the viewing coordinate system.

viewing [Class]

:super	**cascaded-coords**
:slots	(viewcoords)

defines the viewing transformation.

:viewpoint [method]

returns the position vector of the origin of this viewing.

:view-direction [method]

returns the vector from the origin of the viewing to the center of screen. This is the z-axis direction of the viewing coordinates.

:view-up [method]

returns y-axis vector of this viewing represented in the world coords. Y-axis is the upward direction in the viewport.

:view-right [method]

returns x-axis vector of this viewing represented in the world coords. X-axis is in horizontal direction to the right in the viewport.

:look *from &optional (to #f(0 0 0))* [method]

:look conveniently sets the viewing coords as the eye is located at *from* and looking at *to* point.

:init [method]

&key (target #f(0 0 0))
(view-direction nil)
(view-up #f(0.0 0.0 1.0))
(view-right nil)
&allow-other-keys

Since viewing inherits cascaded-coords, all the *:init* parameters such as *:pos*, *:rot*, *:Euler*, *:rpy*, etc. can be used to specify the location and the orientation of the viewing coordinates. However, viewing's *:init* provides easier way to determine the rotation. If only *:target* is given, view-line (-z axis) is determined to pass the viewpoint and *:target* point, and the *:view-right* vector is determined so that the x-axis is parallel to the xy-plane of the world coordinates. You may specify *:view-direction* instead of *:target* to get the same effect. If you give *:view-up* or *:view-right* parameter in addition to *:target* or *:view-direction*, you can determine all the three rotation parameters by yourself.

15.2 Projection

Class **parallel-projection and perspective-projection** process projection transformation, which is represented with a 4X4 matrix, i.e., the transformation is taken in the three dimensional homogeneous coordinates. Class **projection** is an abstract class for both of these. Since these projection classes inherit the viewing class, two coordinates transformation, world-to-viewing and projection can be performed at the same time. By sending the `:project3` message with a 3D vector to a projection object, a float-vector of four elements is returned. **Homo2normal** function is used to convert this homogeneous vector to the normal representation. The result is a vector represented in so called normalized device coordinates (NDC), in which a visible vector ranges within -1 to 1 in each of x,y, and z dimensions. For the simulation of real cameras in a robot world, the perspective projection is used more often than the parallel-projection. Perspective-projection defines a few more parameters. `Screenx` and `screeny` are the sizes of the window on the viewing plane on which observed objects are projected, and with the larger screen, the wider space is projected. `Viewdistance` which defines the distance between the viewpoint and the viewplane also concerns with the viewing angle. The larger viewdistance maps the smaller region to the window on the view plane. `Hither` and `yon` parameters determine the distance to the front and back depth clipping planes. Objects outside these two planes are clipped out. Actually, this clipping procedure is performed by the viewport object.

projection [Class]

:super **viewing**

:slots (screenx screeny hither yon projection-matrix)

defines projection transformation with a 4x4 matrix.

:projection *&optional pmat* [method]

if *pmat* is given, it is set to the *projection-matrix* slot. **:projection** returns the current 4x4 projection matrix.

:project *vec* [method]

Vec is a three-dimensional homogeneous float-vector of four elements. *Vec* is transformed by projection-matrix, and the resulted homogeneous representation is returned.

:project3 *vec* [method]

Vec is a normal 3D float-vector. *Vec* is homogenized and transformed by projection-matrix, and the resulted homogeneous representation is returned.

:view *vec* [method]

applies viewing transformation and projection transformation to *vec* successively. The resulted homogeneous representation is returned.

:screen *xsize (&optional (ysize xsize))* [method]

changes the size of the viewing screen. The larger the size, the wider view you get.

:hither *depth-to-front-clip-plane* [method]

determines the distance from the viewpoint to the front-clipping plane. Objects before the front-clipping (hither) plane are clipped out.

:yon *depth-to-back-clip-plane* [method]

changes the distance between the viewpoint and the back-clipping plane. Objects behind the back-clipping (yon) plane are clipped out.

:aspect *&optional ratio* [method]

Aspect ratio is the ratio between screen-y and screen-x. If *ratio* is given, the aspect ratio is changed by setting screen-y to screen-x * *ratio*. **:aspect** returns the current aspect ratio.

:init [method]

&key (hither 100.0)
(yon 1000.0)
(aspect 1.0)
(screen 100.0)
(screen-x screen)
(screen-y (* screen-x aspect))
&allow-other-keys

initializes viewing and projection.

parallel-viewing [Class]

:super **projection**
:slots

defines parallel projection. **Hid** (the hidden-line elimination function) cannot handle parallel projection.

:make-projection [method]

perspective-viewing [Class]

:super **projection**
:slots (viewdistance)

defines a perspective projection transformation.

:make-projection [method]

:ray *u v* [method]

returns the normalized direction-vector pointing (u,v) on the normalized screen from the viewpoint.

:viewdistance *&optional vd* [method]

Viewdistance is the distance between viewpoint and the screen. If *vd* is given, it is set to viewdistance. The viewdistance corresponds to the focal length of a camera. The greater the viewdistance, the more zoomed-up view you get. **:viewdistance** returns the current viewdistance.

:view-angle *&optional ang* [method]

set screen size so that the prospective angle of the diagonal of the screen becomes *ang* radian. Note that angles somewhat between 20 degree (approx. 0.4 rad.) and 50 degree (0.9 rad.) can generate a natural perspective view. Wider angle generates a skewed view, and narrower a flat view like orthogonal (parallel) viewing. **:view-angle** returns current or new view angle in radian.

:zoom *&optional scale* [method]

If *scale* is given, the screen is changed relatively to the current size by *scale* (the viewdistance is unchanged). If you give 0.5 for *scale*, you get two times as wide view as before. **:zoom** returns new view angle in radian.

:lookaround *alfa beta* [method]

translates and rotates the viewpoint. The center of rotation is taken at the midst of the hither plane and the yon plane on the viewline. The viewing coordinates is rotated *alfa* radian around world's z-axis and *beta* radian around x-axis locally. **:lookaround** allows you to move around the object in the center of viewing.

:look-body *bodies* [method]

changes view direction, screen sizes, and hither/yon so that all the *bodies* fit in the viewport. Viewpoint does not change. View direction is chosen so that the viewing line penetrate the center of the bounding box of all bodies.

:init *&key (viewdistance 100.0) &allow-other-keys* [method]

15.3 Viewport

Class **viewport** performs three-dimensional viewport clipping in the normalized device coordinates, and maps the result into the device dependent coordinates. The viewport is the term representing the visible rectangular area on a screen. The physical size (dots in x and y) of a viewport should be given with **:init** message as the *:width* and *:height* arguments. *:xcenter* and *:ycenter* arguments determine the physical location of the viewport. These two parameters actually decide the location where objects are drawn on the screen when you are using a primitive display device like tektronics 4014 on which every dimension must be given absolutely to the origin of the screen. If you are using more sophisticated display device like Xwindows where locations can be determined relatively to the parent window, you need not to change viewport's parameters to move the viewport. These parameters are independent of the actual display location.

Viewport class assumes the origin of the viewport at the lower-left corner of the rectangular area and y-axis extends to the upper direction. Unfortunately, in many window systems and display devices, the origin is taken at the upper-left corner and y-axis extends to the lower direction. To work around this problem, a negative value should be given to the *:height* parameter.

homo-viewport-clip *v1 v2* [function]

V1 and *v2*, which are two homogeneous vectors with four elements, represent a line in 3-D space. The line is clipped at the boundary of $x = -1, x = 1, y = -1, y = 1, z = 0, z = 1$, and a list of two vectors are returned. If the line lies completely outside the viewport, NIL is returned.

viewport [Class]

:super **coordinates**
:slots

viewport transformation maps the NDC (normalized device coordinates) to device specific coordinates. Inheriting the **coordinates** class, the `viewport` defines the size and the relative position of the projection screen.

:xcenter *&optional xcenter* [method]

X coordinates of the center of this viewport.

:ycenter *&optional ycenter* [method]

Y coordinates of the center of this viewport.

:size *&optional size* [method]

List of sizes in x direction and y direction.

:width *&optional width* [method]

width of this viewport.

:height *&optional height* [method]

height of this viewport.

:screen-point-to-ndc *p* [method]

p is a float-vector representing the location in the physical screen. *p* is transformed into the representation in the normalized-device coordinates.

:ndc-point-to-screen *p* [method]

NDC representation in this viewport, *p*, is transformed into the physical address on the screen.

:ndc-line-to-screen *p1 p2 &optional (do-clip t)* [method]

Two 3D float-vectors, *p1* and *p2*, define a line in NDC. These two end points are transformed to the representation in the screen space. If *do-clip* is non-nil, the line is clipped.

:init *&key (xcenter 100) (ycenter 100) (size 100) (width 100) (height 100)* [method]

makes a new viewport object.

15.4 Viewer

To get a drawing on a screen, four objects are needed: (1) objects to be drawn, (2) a viewing which defines the viewing coordinates and the projection, (3) a viewport for clipping in NDC and the transformation from NDC to physical screen coordinates, and (4) a viewsurface which performs drawing functions on a physical display device. A **viewer** object holds a viewing, a viewport and a viewsurface object, and controls successive coordinates transformation. Functions **draw** and **hid** described in section **??** use the instances of viewer.

viewer [Class]

:super	**object**
:slots	(eye :type viewint)
	(port :type viewport)
	(surface :type viewsurface)

defines the cascaded coordinates transformation from the viewing via the viewport to the viewsurface.

:viewing *&rest msg* [method]

If *msg* is given, *msg* is sent to the **viewing**`(eye)` object, Otherwise, the **viewing**`(eye)` object is returned.

:viewport *&rest msg* [method]

If *msg* is given, *msg* is sent to the **viewport**`(port)` object, Otherwise, the **viewport**`(port)` object is returned.

:viewsurface *&rest msg* [method]

If *msg* is given, *msg* is sent to the **viewsurface**`(surface)` object, Otherwise, the **viewsurface**`(surface)` object is returned.

:adjust-viewport [method]

When the size of viewsurface has been changed, **:adjust-viewport** changes viewport transformation sending a proper message to port.

:resize *width height* [method]

changes the size of viewsurface by sending :resize message to the viewsurface and :size message to viewport.

:draw-line-ndc *p1 p2 &optional (do-clip t)* [method]

draws a line whose two end points *p1, p2* are defined in NDC.

:draw-polyline-ndc *polylines &optional color* [method]

draws polylines whose end points are defined in NDC.

:draw-star-ndc *center &optional (size 0.01) color* [method]
draws a cross mark in NDC.

:draw-box-ndc *low-left up-right &optional color* [method]
draws a rectangle in NDC.

:draw-arc-ndc *point width height angle1 angle2 &optional color* [method]
draws an arc in NDC. The viewsurface object bound in this viewer must accept **:arc** message.

:draw-fill-arc-ndc *point width height angle1 angle2 &optional color* [method]
draws a filled-arc in NDC.

:draw-string-ndc *position string &optional color* [method]
draws *string* at *position* defined in NDC.

:draw-image-string-ndc *position string &optional color* [method]

:draw-rectangle-ndc *position width height &optional color* [method]

:draw-fill-rectangle-ndc *point width height &optional color* [method]

:draw-line *p1 p2 &optional (do-clip t)* [method]
draws a line whose two end points *p1, p2* are defined in the world coordinates.

:draw-star *position &optional (size 0.01) color* [method]
draws a cross at *position* located in the world.

:draw-polyline *vlist &optional color* [method]
draws polylines whose end points *vlist* are defined in the world.

:draw-box *center &optional (size 0.01)* [method]
draws a rectangular at *center*in the world.

:draw-arrow *p1 p2* [method]
draws an arrow from *p1* to *p2*.

:draw-edge *edge* [method]

:draw-edge-image *edge-image* [method]

:draw-faces *face-list &optional (normal-clip nil)* [method]

:draw-body *body &optional (normal-clip nil)* [method]

:draw-axis *coordinates &optional size* [method]
draws coordinates axes whose length is *size*.

:draw *&rest things* [method]

draws 3D geometric objects. If the object is a 3D float-vector, a small cross is drawn at the position. If it is a list of 3D float-vectors, it is taken as a polyline. If *thing* accepts `:draw` message, the method is invoked with this viewer as its argument. If the object defines `:drawners` method, the **:draw** message is sent to the result of `:drawners`. `Line, edge, polygon, face`, and `body` objects are drawn by corresponding *:draw-xxx* methods defined in viewer.

:erase *&rest things* [method]

draws *things* with background color.

:init *&key viewing viewport viewsurface* [method]

sets *viewing, viewport* and *viewsurface* to `eye, port`, and `surface` slots of this viewer.

view [function]

&key (size 500) (width size) (height size)
(x 100) (y 100)
(title "eusx")
(border-width 3)
(background 0)
(viewpoint #f(300 200 100)) (target #f(0 0 0))
(viewdistance 5.0) (hither 100.0) (yon 10000.0)
(screen 1.0) (screen-x screen) (screen-y screen)
(xcenter 500) (ycenter 400)

creates a new viewer and pushes it in *viewers* list.

15.5 Drawings

draw *&optional viewer &rest thing* [function]

draws *thing*s in *viewer*. *Thing* can be any of coordinates, body, face, edge, float-vector, list of two float-vectors. If you are running `eusx`, `(progn (view) (draw (make-cube 10 20 30)))` draws a cube in a xwindow.

draw-axis *&optional viewer size &rest thing* [function]

draws coordinate-axes of *thing*s in *viewer* with *size* as the length of each coordinates-axis. *Thing* can be any object derived from coordinates.

draw-arrow *p1 p2* [function]

draws an arrow pointing from p1 to p2 in `*viewer*`.

hid *&optional viewer &rest thing* [function]

draws hidden-line eliminated image in *viewer*. *Thing* can be of face or body.

hidd *&optional viewer &rest thing* [function]

is same as **hid**, except that **hidd** draws hidden lines with dashed-lines.

hid2 *body-list viewing* [function]

Generate hidden-line eliminated image represented by edge-image objects. The result is bound to ***hid***.

render *&key bodies faces (viewer *viewer*) (lights *light-sources*) (colormap *render-colormap*) (y 1.0)* [function]

does ray-tracing for *bodies* and *faces* and generates hidden-surface removed images. viewing, viewport, and viewsurface are taken from *viewer*. *lights* is a list of `light-source` objects. *colormap* is xwindow's colormap object. Each of bodies and faces must have color attribute assigned. This can be done by sending `:color` message with the name of color LUT defined in the *colormap*. Currently this function works only in Xlib environment. See examples in `demo/renderdemo.l`.

make-light-source *pos &optional (intensity 1.0)* [function]

make a light-source object located at *pos*. *intensity* is magnifying ratio which multiplies default light intensity. In order to determine the intensity more precisely, use `:intensity` method of a light-source.

tektro *file &rest forms* [macro]

opens file for `*tektro-port*` stream, and evaluates forms. This is used in order to redirect the output of tektro drawings to a file.

kdraw *file &rest forms* [macro]

Kdraw is a macro to produce a [ik]draw-readable postscript file. **Kdraw** opens *file* in `:output` mode, makes a kdraw-viewsurface and a viewport with which `*viewer*` is replaced, and evaluates *forms*. Each of *forms* is a call to any of drawing functions like `draw` or `hid`. Drawing messages from these forms are redirected to a `kdraw-viewsurface`, which transforms the messages into postscript representations that `idraw` or `kdraw` can recognize, and stores them in *file*. When `idraw` or `kdraw` is invoked and *file* is opened, you see the identical figure you drew in a EusViewer window. The figure can be modified by `idraw`'s facilities, and the final drawing can be incorporated into a LaTeXdocument using the `epsfile` environment.

pictdraw *file &rest forms* [macro]

Pictdraw is a macro to produce picture files for Macintosh in PICT format. **Pictdraw** opens *file* in `:output` mode makes a pictdraw-viewsurface and a viewport with which `*viewer*` is replaced, and evaluates *forms*. Each of *forms* is a call to any of drawing functions like `draw` or `hid`. Drawing messages from these forms are redirected to a `kdraw-viewsurface`, which transforms the messages into PICT format that `macdraw` or `teachtext` of Macintosh can recognize, and stores them in *file*.

hls2rgb *hue lightness saturation &optional (range 255)* [function]

Color representation in HLS (Hue, Lightness, and Saturation) is converted to RGB representation. HLS is often referred to as HSL. *Hue* represents a color around a rainbow circle (from 0 to 360). 0 for red, 45 for yellow, 120 for green, 240 for blue, 270 for magenta, and 360 again for red, etc. *Lightness* is a value between 0.0 and 1.0, representing from black to white. The color of lightness value of 0 is always black regardless to the hue and saturation, and the lightness value 1.0 is always white. *Saturation* is a value between 0.0 and 1.0, and represents the strength of the color. The greater the saturation value, the divider the color, and small saturation values generate weak, dull tone colors. *Range* limits the RGB values. If you are using a color display which can assign 8bit value to each of red, green and blue, *range* should be 255. If you use Xwindow, which virtually assigns 16bits integers to RGB, you should specify *range* to 65535. Note the difference between HSV and HLS. In HLS, vivid (rainbow) colors are defined with lightness=0.5.

rgb2hls *red green blue &optional (range 255)* [function]

RGB representation of a color is converted into the corresponding representation in HLS.

15.6 Animation

EusLisp's animation facility provides the pseudo real-time graphics on stock workstations without graphics accelerators. The basic idea is the quick playback of a series of images which have been generated after long computation. Images are retained in two ways: one is to keep a number of xwindow pixmaps each of which holds a complete pixel image, and the other is to keep line segment data obtained by hidden-line elimination. The former is faster and the only way for rendered images, but not suitable for a long animation since it requires much memory in the X server. The latter is more memory efficient and suitable for storing data in disks, but the performance is degraded when the number of line segments increases.

In either way, the user provide a function which gives new configurations to the objects to be drawn and generates drawing on `*viewer*`. **pixmap-animation** calls this function as many times as specified by the *count* argument. After each call, the content of `*viewsurface*`, which is assumed to be an xwindow, is copied to a newly created Xwindow pixmap. These pixmaps are played back by **playback-pixmaps**. Similarly, **hid-lines-animation** extracts visible line segments from the result of **hid**, and accumulates them in a list. The list is then played back by **playback-hid-lines**.

Following functions are defined in `llib/animation.l`, and `demo/animdemo.l` contains a sample animation program using **hid-lines-animation** on the ETA3 manipulator model.

pixmap-animation *count &rest forms* [macro]

forms are evaluated *count* times. After each evaluation, the content of `*viewsurface*` is copied in a new pixmap. A list of *count* pixmaps is returned.

playback-pixmaps *pixmaps &optional (surf *viewsurface*)* [function]

Each pixmap in the *pixmaps* list is copied to *surf* successively.

hid-lines-animation *count &rest forms* [macro]

forms, which are assumed to include call(s) to **hid**, are evaluated *count* times. After each evaluation, the result of **hid** held in ***hid*** is scanned and visible segments are collected in a list of point pairs. A list of length *count* is returned.

playback-hid-lines *lines &optional (view *viewer*)* [function]

lines is a list of lists of point pairs. draws lines successively on *view*. Double buffering technique allocating another pixmap is used to generate flicker-free animation.

list-visible-segments *hid-result* [function]

collects visible segments from the list of edge-images *hid-result*.

16 Xwindow Interface

The Xwindow interface on EusLisp becomes available when EusLisp is invoked by the name of `'eusx'`. [3] The "DISPLAY" environment variable should be properly set to your Xserver, since eusx tries to connect to Xserver referencing the "DISPLAY" environment variable when it starts up.

EusLisp defines three levels of xwindow interface: (1) Xlib functions, (2) Xlib classes, and (3) XToolKit classes. All the xwindow functions described in this section and the following XToolKit section are contained in the "X" package. The function names of the original Xlib are changed so that all constituent letters are converted to upcase and the first 'X' prefix is removed. For example, `XdefaultGC` is named `X:DEFAULTGC`, not `X:XDEFAULTGC`.

The Xlib functions are defined as foreign functions as the lowest level interface to Xwindow system. These Xlib functions should be used carefully, since parameter type check or parameter number check is not performed. For an instance, all the Xlib call requests `x:*display*` argument to identify the connection to Xserver, and if you forget it, Xlib reports an error and the process dies. The second level interface, Xlib classes are provided to avoid this inconvenience and to make the interface object-oriented. This section focuses on this second level interface. Even higher level xwindow library called XToolKit is explained in the next section.

Classes described in this section have the following inheritance hierarchy.

```
propertied-object
   viewsurface
      x:xobject
         x:gcontext
         x:xdrawable
             x:xpixmap
             x:xwindow
   colormap
```

16.1 Xlib global variables and misc functions

x:*display* [variable]

X's display ID (integer).

x:*root* [variable]

default root window object.

x:*screen* [variable]

default screen ID (integer).

x:*visual* [variable]

default visual ID (integer).

x:*blackpixel* [variable]

black pixel = 1

x:*whitepixel* [variable]

[3]Eusx is a symbolic link to eus.

white pixel = 0

x:*fg-pixel* [variable]

default foreground pixel referenced at window creation, normally `*blackpixel*`.

x:*bg-pixel* [variable]

background pixel referenced at window creation, normally `*whitepixel*`.

x:*color-map* [variable]

the system's default color-map.

x:*defaultGC* [variable]

the default gcontext referenced at pixmap creation.

x:*whitegc* [variable]

GC whose foreground color is white.

x:*blackgc* [variable]

GC whose foreground color is black.

gray-pixmap [variable]

the result of `(make-gray-pixmap 0.5)`.

gray25-pixmap [variable]

16x16 pixmap, a quarter of pixels are `*fg-pixel*` and three quarters `*bg-pixel*`.

gray50-pixmap [variable]

16x16 pixmap, a half of pixels are `*fg-pixel*`.

gray75-pixmap [variable]

16x16 pixmap, three quarters of pixels are black.

gray25-gc [variable]

25% gray GC made from `*gray25-pixmap*`.

gray50-gc [variable]

50% gray GC made from `*gray50-pixmap*`.

gray75-gc [variable]

75% gray GC made from `*gray75-pixmap*`.

gray [variable]

`"#b0b0b0"`

bisque1 [variable]

`"#ffe4c4"`

bisque2 [variable]

`"#eed5b7"`

bisque3 [variable]

`"#cdb79e"`

lightblue2 [variable]

"#b2dfee"

lightpink1 [variable]

"#ffaeb9"

maroon [variable]

"#b03060"

max-intensity [variable]

65535

font-cour8 [variable]

(font-id "*-courier-medium-r-*-8-*")

font-cour10 [variable]

(font-id "*-courier-medium-r-*-10-*")

font-cour12 [variable]

(font-id "*-courier-medium-r-*-12-*")

font-cour14 [variable]

(font-id "*-courier-medium-r-*-14-*")

font-cour18 [variable]

(font-id "*-courier-medium-r-*-18-*")

font-courb12 [variable]

(font-id "*-courier-bold-r-*-12-*")

font-courb14 [variable]

(font-id "*-courier-bold-r-*-14-*")

font-courb18 [variable]

(font-id "*-courier-bold-r-*-18-*")

font-helvetica-12 [variable]

(font-id "*-Helvetica-Medium-R-Normal-*-12-*")

font-lucidasans-bold-12 [variable]

(font-id "lucidasans-bold-12")

font-lucidasans-bold-14 [variable]

(font-id "lucidasans-bold-14")

font-helvetica-bold-12 [variable]

(font-id "*-Helvetica-Bold-R-Normal-*-12-*")

font-a14 [variable]

(font-id "*-fixed-medium-r-normal-*-14-*")

x:*xwindows* [variable]

a list of all windows including subwindows created and maintained by EusLisp.

x:*xwindow-hash-tab* [variable]

a hash table to look up the xwindow object by its drawable ID. In the event structure obtained by `x:nextevent` is a window ID, and `x:window-main-loop` calls `x:event-window` to know the corresponding xwindow object using this table.

xflush [function]

sends all commands retained in the Xlib command buffer to Xserver. Since Xlib buffers output to Xserver, commands you issued commands to Xserver are not executed immediately. This is necessary to decrease network traffic and the frequency of process switching. To flush the command buffer to see the effects of the commands, use **xflush** or send **:flush** message to xwindow objects.

find-xwindow *subname* [function]

Each xwindow may have name specified at the creation time. Find-xwindow looks in the *xwindows* list and returns a list of windows that have 'subname' as a substring of its name.

16.2 Xwindow

Xobject [Class]

:super	**geometry:viewsurface**	
:slots		

The common super class for all the Xwindow related classes. Currently, no slots variables and methods are defined.

Xdrawable [Class]

:super	**Xobject**	
:slots	(drawable	; drawable ID
	gcon	; this drawable's default graphic context object
	bg-color	; background color
	width height	; horizontal and vertical dimensions in dots

Xdrawable defines rectangular regions where graphics objects such as lines and strings can be drawn. **Xdrawable** is an abstract class to define common methods for xwindow and xpixmap, and instantiation of this class has no effect.

:init *id* [method]

Id is set to the *drawable* slot as the ID of this drawable. A new GC (graphic context) is created and set to *gcon* as the default GC of this drawable object.

:drawable [method]

returns drawable id.

:flush [method]

flushes commands retained in the Xlib's buffer.

:geometry [method]

returns the list of seven geometric attributes, *root-window-id, x-position, y-position, width, height, border-width* and *visual's depth.*

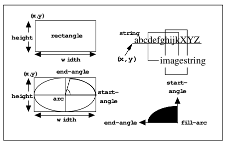


図 14: drawing primitives

:height [method]

returns the height (dots in y direction) of this drawable.

:width [method]

returns width (dots in x direction) of this drawable.

:gc *&rest newgc* [method]

If no *newgc* is given, the current gc object is returned. If *newgc* is an instance of gcontext, it is set to the gc of this drawable. Otherwise, *newgc* is regarded as a message and sent to the current gc.

:pos [method]

returns an integer vector representing the position of this drawable. The position is always defined relative to the parent window, and windows created as direct subwindows of the root window under the intervention of the window manager return the constant coordinates in their surrounding title window regardless to their true position in the root.

:x [method]

returns the x coordinate of this drawable relatively to the parent window.

:y [method]

returns the y coordinate of this drawable relatively to the parent window.

:copy-from *drw* [method]

Drw is another drawable object (xwindow or pixmap). The contents of *drw* is copied to this drawable.

:point *x y &optional (gc gccon)* [method]

draws a point at (x, y) with optional *gc*.

:line *x1 y1 x2 y2 &optional (gc gcon)* [method]

draw a line from *(x1, y1)* to *(x2, y2)* with optional *gc*. *x1, y1, x2,* and*y2* must be integers.

:rectangle *x y width height &optional (gc gcon)* [method]

draws a rectangle whose center is located at *(x, y)* and size is specified by *width* and *height*.

:arc *x y width height angle1 angle2 &optional (gc gcon)* [method]

draws an elliptic arc whose center is *(x, y)* and starting angle at *angle1* and ending angle at *angle2*.

Angles should be given by radian.

:fill-rectangle *x y width height &optional (gc gcon)* [method]

fills in a rectangular region.

:fill-arc *x y width height angle1 angle2 &optional (gc gcon)* [method]

fills in an arc.

:string *x y str &optional (gc gcon)* [method]

displays the string *str* starting at *(x, y)*. The background is not filled.

:image-string *x y str &optional (gc gcon)* [method]

displays an imagestring of *str*. Imagestring fills background.

:getimage *&key x y width height (mask #ffffffff) (format 2)* [method]

gets ximage from the server and returns the pixel data in a string. The pixel data sent from the server is once stored in Xlib's ximage structure, then copied to the string row by row. The ximage structure is automatically destroyed. The image string obtained by `:getimage` can be used to make a `pixel-image`, which can be written to a file in the pbm formats as described in section **??**.

:putimage *image &key src-x src-y dst-x dst-y width height (gc gcon)* [method]

puts *image* to the specified location in this drawable. *image* is a string or a address pointing to an ximage structure.

:draw-line *from to* [method]

is same as **:line** method, and provided for the compatibility with other viewsurface classes.

:line-width *&optional dots* [method]

sets line-width of this drawable's default GC. Use of the `:gc :line-width` message is recommended.

:line-style *&optional dash* [method]

sets line-style of this drawable's default GC. Use of the `:gc :line-style` is preferable.

:color *&optional c* [method]

sets color of this drawable.

:clear [method]

clears full screen. this method calls `:clear-area`

:clear-area *&key x y width height gc* [method]

clears a rectangle using the `:fill-rectangle` method.

Xpixmap [Class]

:super	**Xdrawable**
:slots	

Pixmap is a drawable that is often used as a picture buffer or a background pattern. Unlike xwindow, pixmap itself is not visible until it is copied to xwindow or pixmap does not generate any event.

:init *id* [method]

initializes this pixmap.

:create *&key (width 500) (height 500) (depth 1) (gc *defaultgc*)* [method] creates a *width* x *height* pixmap with *gc* as its default GC.

:create-from-bitmap-file *fname* [method]

creates a pixmap from a bitmap file.

:write-to-bitmap-file *fname* [method]

writes the contents of this pixmap into a bitmap file, which can be read back to create a pixmap by **:create-from-bitmap-file** method.

:destroy [method]

destroys this pixmap and frees X resources.

Xwindow [Class]

:super	**Xdrawable**
:slots	(parent subwindows backing-pixmap event-forward)

Xwindow defines visible rectangular regions of the screen. It is inherited not only by **text-window** and **canvas** where any graphics objects can be drawn, but also by many **panel-items** and **scroll-bars**, which look like graphics objects rather than windows.

:create [method]

&key ((parent *root*)
(x 0) (y 0) (size 256) (width size) (height size) (border-width 2)
(save-under nil) (backing-store :always) (backing-pixmap nil)
(border *fg-pixel*) (background *bg-pixel*)
(map T) (gravity :northwest)
(title "WINDOW") (name title)
(font)
event-mask (:key :button :enterLeave :configure :motion)

creates and initializes a xwindow. When *parent* is given, this window is created as a subwindow of *parent*, and is registered in the *subwindows* list of the *parent*. *X, y, size, width, height* and *border-width* determine the location and the dimensions of this window. *Save-under* and *backing-store* control the Xserver's behaviors taken upon when the window is re-mapped. *Save-under* is either T or NIL, while *backing-store* is either `:notUseful`, `:WhenMapped`, or *:Always*. When *backing-pixmap* is T, a pixmap of the same size as this window is created by EusLisp, and maintained as a backing-store in case the Xserver does not have the capability of backing-store. *Border* and *background* specify the *border_pixel* and *background_pixel* attributes, respectively. *Map* should be set NIL, if this window should not appear immediately after its creation, as is the case many small windows are created as panel-buttons in a **panel**. *Title* is the window title which appears in the title bar of the window. *Name* is the name of the window stored in the property-list of this xwindow object and printed by the printer. X's events reported to this window are determined by *Event-mask*, that is, either an integer representing a bit-coded event-mask or a list of the following symbols: `:key, :button, :enterLeave, :motion` and `:configure`. If more precise control is needed, the following symbols for each event can be specified: *:keyPress, :keyRelease, :ButtonPress, :ButtonRelease, :EnterWindow, :LeaveWindow, :PointerMotion, :PointerMotionHint, :ButtonMotion, :KeyMapState, :Exposure, :VisibilityChange, :StructureNotify, :ResezeRedirect, :SubstructureNotify, :SubstructureRedirect, :FocusChange, :PropertyChange, :ColormapChange* and `:OwnerGrabButton`. `:Key` enables both `:keyPress` and `:KeyRelease`, and `:button` enables both `:ButtonPress` and `:ButtonRelease`. When an event is sent from the server, **window-main-loop** analyzes the event structure and send the

`:KeyPress, :KeyRelease, :buttonPress, :ButtonRelease, :EnterNotify, :LeaveNotify, :MotionNoti` `:ConfigureNotify` message to the window where the event occurred.

:map [method]

makes this xwindow and all the subwindows visible.

:unmap [method]

makes this xwindow and all the subwindows invisible.

:selectinput *event-mask* [method]

Event-mask is either an integer or a list of eventmask symbols. Each event corresponding to the bit turned-on or enumerated in the *event-mask* list becomes to be reported to this window.

:destroy [method]

destroys this xwindow and frees X resource. The corresponding entries in `*xwindows*` and `*xwindow-hash-tab*` are also deleted so that this window object could be garbage-collected. All subwindows are also deleted by sending `:destroy`. This window is dissociated from the subwindow list of the parent window. The *drawable* ID is set to NIL.

:parent [method]

returns the parent window object.

:subwindows [method]

returns the list of all the subwindows. The subwindow most recently created comes first in the list. Only the direct subwindows of this window are listed and subwindows of the subwindows are not.

:associate *child* [method]

register the *child* window as a subwindow of this window.

:dissociate *child* [method]

removes the *child* window of the *subwindows* list.

:title *title* [method]

changes the title of this window. Though the title is in the Xserver, it is maintained and displayed by the window manager.

:attributes [method]

returns an integer-vector representing the attributes of this window.

:visual [method]

returns the visual resource id for this window.

:screen [method]

returns the screen resource id for this window.

:root [method]

returns the root window id.

:location [method]

returns a two dimensional integer-vector describing the x and y coordinates of this window.

:depth [method]

returns the depth (number of color planes) of this window.

:size [method]

returns the size (width and height) of this window.

:colormap [method]

returns colormap resource id for this window.

:move *newx newy* [method]

changes the location of this window to *(newx, newy)*. The coordinates are given relative to the parent window.

:resize *width height* [method]

changes the size of this window. Probably because the size parameters are cached in the Xlib on the client side, `:geometry` message immediately after `:resize` may return wrong (old) result.

:raise [method]

brings this window upfront.

:lower [method]

pushes this window to the back.

:background *pixel* [method]

changes the background pixel value (the index in the color map) to *pixel*. The *pixel* value is also stored in the *bg-color* slot. `:Clear` operation is performed to fill the current background with the specified *pixel*.

:background-pixmap *pixmap* [method]

changes the background with given pixmap.

:border *pixel* [method]

sets the color of the border to *pixel*.

:set-colormap *cmap* [method]

sets colormap.

:clear [method]

clears the entire xwindow.

:clear-area *&key x y width height* [method]

clears the specified rectangular area of this xwindow.

make-xwindow *&rest args* [function]

makes x-window.

init-xwindow *&optional (display (getenv "DISPLAY"))* [function]

is the first function to call when eusx start up. `Init-xwindow` connects to the Xserver specified by *display*, and initializes default variables described in the section **??**. `Init-xwindow` also loads default fonts and sets them to global variables, such as font-courb12, lucidasans-bold-12, etc. This font loading causes the delay at the start-up time. Reduction of the number of fonts loaded or specifying the exact font-names without using the wild-card character "*" will shorten the delay.

16.3 Graphic Context

gcontext [Class]

:super **Xobject**
:slots (gcid GCValues)

defines the graphic context. In EusLisp, every xwindow has its default GC.

:create [method]

&key (drawable defaultRootWindow)
(foreground *fg-pixel* (background *bg-pixel*)
function plane-mask
line-width line-style cap-style join-style
font dash

creates a gc with given attributes. *Drawable* is used by the Xserver to know the screen and depth of the screen. The resulted GC can be used in any drawables as long as they are created on the same screen.

:gc [method]

returns X's GC id.

:free [method]

frees this GC.

:copy [method]

makes a copy of this GC.

:foreground *&optional color* [method]

if *color* is given, it is set to the foreground color. *Color* is a pixel value.

:background *&optional color* [method]

if *color* is given, it is set to the background color. *Color* is a pixel value.

:foreback *fore back* [method]

sets foreground and background colors at once.

:planemask *&optional plane-mask* [method]

sets plane-mask.

:function *x* [method]

sets drawing function. *X* should either be one of the following numbers or keywords: `0=Clear, 1=And, 2=AndReverse, 3=Copy, 4=AndInverted, 5=NoOp, 6=Xor, 7=Or, 8=Nor, 9=Equiv, 10=Invert, 11=XorReverse, 12=CopyInverted, 13=OrInverted, 14=Nand, 15=Set, :clear, :and, :andReverse, :copy, :andInverted, :NoOp, :Xor, :Or, :Nor, :Equiv, :Invert, :XorReverse, :CopyInverted, :OrInverted, :Nand, :Set`.

:font *x* [method]

sets the font attribute of this GC. *X* is either a font-name or a font-ID. If *x* is a font name (string), `:font` calls `x:LoadQueryFont` to decide the font-id. If not found, `"no such font ..."` is warned. If *x* is NIL (not given), the current font-ID of this GC is returned.

:line-width *x* [method]

sets the line width in pixel.

:line-style *x* [method]

sets the line-style (solid, dashed, etc.).

:dash *&rest x* [method]

Each component of *X* is an integer. `:Dash` sets the dash pattern of the line-style.

:tile *pixmap* [method]

sets the tile of this GC to *pixmap*.

:stipple *pixmap* [method]

sets the stipple of this GC to *pixmap*.

:get-attribute *attr* [method]

gets attribute. *Attr* is one of `:function, :plane-mask, :foreground, :background, :line-width, :line-style, :cap-style, :join-style, :fill-style, :fill-rule, :font`. An integer value representing the attribute is returned.

:change-attributes [method]

&key function plane-mask foreground background
line-width line-style cap-style join-style font dash

change attributes. More than one attributes are changed at the same time.

font-id *fontname* [function]

If *fontname* is integer, it is returned regarding it as font-id. If *fontname* is string, font-structure is inquired by using `x:LoadQueryFont`, and its font-id is returned. *Fontname* can be a shorthand of exact name, such as `"*-courier-24-*"` for any 24-point courier font. If the font could not be found, `can't load font` warning is printed.

textdots *str font-id* [function]

returns a list of three integers representing (ascent descent width) of the *str* (string) in dots.

16.4 Colors and Colormaps

colormap [Class]

:super	**object**
:slots	(cmapid planes pixels LUT-list)

defines an xwindow colormap and application oriented color look-up tables. A color is represented by RGB values from 0 through 65535. Color cells in a color map are addressed by their indices, which are between 0 and 255 on 8-bit pseudo color display.

Here we assume your display device has 8bit pseudo color capability which allows you to choose 256 colors at the same time. Basically there are two ways in the use of color maps: to share the system's default color map or to create private color maps. If you use the system's default color map, you have to be careful not to use up all the color cells in the map, since the map is shared among many processes. If you use private color maps, you can allocate all 256 color entries in the map without worrying about other processes, but the map has to be explicitly attached to your private windows. The color map is activated by the window manager when the mouse pointer is moved somewhere in the window.

The system's default color map is set up in `x:*color-map*` which is an instance of the `x:colormap` class when eusx begins execution. If you use private color maps, you create instances of `x:colormap`. These instances correspond to the colormap object defined in the x server and are identified by the `cmapid` stored in each instance.

When you use the system's default color map, you can define *read-only* colors which are shared with other processes or define *read-write* colors which are private to your EusLisp. *Read-only* means that you can define arbitrary color when you allocate the color cell, but you cannot change it after the allocation. On the other hand, *read-write* colors can be altered even after you defined them. Shared colors are *read-only* since other processes expect the colors to be unchanged. This *read-only* or *read-write* attribute is attached to each color entry (often referred to as color cell).

A colormap object defines translation from a color id to a physical representation that is a triplet of red, green and blue components. However, these logical color ids cannot be chosen arbitrarily, especially when you use the the system's default color map. The color id (often referred to as 'pixel') is an index of a particular color in a color map and Xlib chooses one of free indices for a shared color when allocation is requested. Therefore, there is no way, for example, to guarantee many levels of gray colors to be allocated contiguously or to begin from the first (zeroth) index.

From the viewpoint of applications, more logical color naming is needed. For example, a number of gray levels should be referred to with their brightness as indices. A ray trace program may wish to assign contiguous indices to a group of colors of different brightness defined in HLS model.

To cope with this problem, EusLisp's colormap provides another translation table called LUT (look-up table). For a logical group of colors, you can define a LUT and attach a symbolic name to it. More than one LUTs can be defined in a colormap. LUT is an integer vector for the translation of application specific logical color indices into physical pixel values that the Xserver can recognize.

:id [method]

returns the cmap id.

:query *pix* [method]

gets RGB values for the specific pixel number.

:alloc *r g b* [method]

this method is the same as `:store nil r g b`. A new color cell is allocated in this colormap and is assigned with the specified RGB values.

:store *pix r g b* [method]

sets RGB values to the *pix*th color cell.

:store *pix color-name* [method]

:Store is the lowest level method to set a color in a color map. In the first form, you specify the color with the red, green and blue components between 0 and 65535 inclusively. In the second form, you specify the color by name like "red" or "navy-blue". If no such color-name is found, nil is returned. Pixel is either an integer which is the index in a color map or nil. If it is integer, the color cell must be read-write-able. If it is nil, a shared read-only color cell is allocated. `:Store` returns the index of the color cell in the color map.

:store-hls *pix hue lightness saturation* [method] stores the color specified in HLS (Hue, Lightness and Saturation) model in the *pix*th entry of this colormap. If *pix* is NIL, a shared read-only color cell is allocated. `:Store-hls` returns the index to the allocated color cell.

:destroy [method]

destroys this colormap and frees resource.

:pixel *LUT-name id* [method]

looks up in the LUT for the id'th entry and returns its pixel value. *LUT-name* is the name of the look-up-table you defined by `:define-LUT`.

:allocate-private-colors *num* [method]

allocates *num* color cells in the private color map.

:allocate-colors *rgb-list &optional private* [method]

Each element of *rgb-list* is a list of red, green and blue components. Color cells are allocated for each rgb value and an integer-vector whose elements are pixel values is returned.

:define-LUT *LUT-name rgb-list &optional private* [method]

Colors described in *rgb-list* are allocated, and an LUT is registered by the symbolic name of *LUT-name*. In order to define private color cells, set *private* to T.

:define-gray-scale-LUT *LUT-name levels &optional private* [method]

allocates *levels* of color cells that represent linear gray scale colors and returns LUT. For example, `(send x:*color-map* :define-gray-scale-LUT 'gray8 8)` allocates eight gray colors in the system's default color map, and returns an integer vector such as `#i(29 30 31 48 49 50 51 0)`. Physical pixel values can be inquired by sending the `:pixel` message, for example, `(send x:*color-map* :pixel 'gray8 2)` returns 31.

:define-rgb-LUT *LUT-name red green blue &optional private* [method]

defines an LUT for shrunk RGB representation. For example, if red=green=blue=2, totally $2^{2+2+2} = 2^6 = 64$ color cells are allocated.

:define-hls-LUT *LUT-name count hue low-brightness high-brightness saturation &optional private* [method]

allocates *count* colors using the HLS model. Colors of the given *hue* (0..360), *saturation* (0..1), and different levels of brightness between *low-brightness* and *high-brightness* are stored in the color map. A LUT named LUT-name is also created.

:define-rainbow-LUT *LUT-name &optional (count 32) (hue-start 0) (hue-end 360) (brightness 0.5) (saturation 1.0) private* [method]

allocates *count* colors using the HLS model. Colors of the given *brightness* (0..1), *saturation* (0..1), and different hues between *hue-start* and *hue-end* are stored in the color map. A LUT named *LUT-name* is also created.

:LUT-list [method]

returns all LUT list defined in this colormap. Each entry in the list is a pair of the LUT-name and an integer vector.

:LUT-names [method]

returns the name list of all LUT in this colormap.

:LUT *name* [method]

returns the integer-vector (LUT) identified by *name*.

:size *LUT* [method]

returns the length of *LUT*

:planes [method]

returns planes of this colormap.

:set-window *xwin* [method]

associates this colormap to the *xwin* window. This colormap is activated when the cursor enters in *xwin*.

:free *pixel-or-LUT* [method]

frees a specific color cell addressed by pixel, or all the entries in LUT.

:init *&optional cmapid* [method]

initializes this color map with cmap id. All the LUTs registered are discarded.

:create *&key (planes 0) (colors 1) (visual *visual*) contiguous* [method]

creates a new color map object.

XColor [Class]

:super	**cstruct**
:slots	((pixel :integer)
	(red :short)
	(green :short)
	(blue :short)
	(flags :byte)
	(pad :byte))

defines a color in the RGB model. Use **setf** to assign value to each slots. The RGB values are sign extended and the greatest value is represented as -1.

:red [method]

returns the red value of this XColor.

:blue [method]

returns the blue value of this XColor.

:green [method]

returns the green value of this XColor.

:rgb [method]

returns the list of red, green and blue values of this XColor.

:init *pix R G B &optional (f 7)* [method]

initializes XColor.

find-visual *type depth &optional (screen 0)* [function]

finds the visual-ID of the specified *type* and *depth*. *Type* should be either `:StaticGray`, `:GrayScale`, `:StaticColor`, `:pseudoColor`, `:TrueColor` or `:DirectColor`. Usually the *depth* should be either 1, 8 or 24.

17 XToolKit

XToolKit is the highest level X window interface to facilitate composing GUI (Graphical User Interface) by using GUI components such as buttons, pulldown menus, textWindows, etc., as building blocks. The major differences from the Xlib classes are, the XToolKit invokes user-supplied interaction routines corresponding to the Xevents sent from the Xserver, and provides consistent appearance of those interaction-oriented window parts. Classes consisting the XToolKit has the following inheritance structure.

```
xwindow
     panel
          menubar-panel
          menu-panel
          filepanel
          textviewpanel
          confirmpanel
     panel-item
          button-item
               menu-button-item
               bitmap-button-item
          text-item
          slider-item
          choice-item
          joystick-item
     canvas
     textwindow
          buffertextwindow
               scrolltextwindow
          textedit
     scroll-bar
          horizontal-scroll-bar
```

Just below the xwindow class are the five basic XToolKit classes: `panel`, `panel-item`, `canvas`, `textWindow` and `scroll-bar`. `Menubar-panel` and `menu-panel` are defined under the `panel`. A basic strategy to build a new application window and to make it run upon events is the following:

1. **define an application class** An application window class should be defined as a subclass of **panel** that has the capability to lay out XToolKit components.

2. **define event handlers** In the application class, event handlers that are called upon when buttons are pressed or menu items are selected are defined. An event handler ought to be defined as a method with panel-item specific arguments.

3. **define subpanels** If you use a `menubar-panel`, it is placed at the top of the application window, therefore it should be created first by `:create-menubar`. Similarly `menu-panel`s needs to be defined before the `menu-button-item`s to which `menu-panel`s are associated.

4. **create panel-items** Panel-items such as `button-item`, `text-item`, `slider-item`, etc., can be created by `(send-super :create-item` *`class label object method`*`)`. Event handlers defined above are connected to each panel-item. These initialization procedures should be defined in the `:create` method

of the application window class. Do not forget to define `quit` button to make the event dispatcher terminate whenever needed. Any `textWindow` and `canvas` can also be placed in the application window via the `:locate-item` method.

5. **create the entire window** Sending the `:create` message to the application class creates the application window with its XToolKit components properly placed in the window.

6. **run the event dispatcher** In order to receive events from the Xserver and delivers them to the corresponding xwindow, run `window-main-loop`. On Solaris2, `window-main-thread`, which delivers events in a different thread, is available. `Window-main-thread` keeps the toplevel interaction alive. Do not run more than one `window-main-thread`.

17.1 X Event

In the current implementation, an event structure is received in a fixed event buffer (an integer-vector of 25 elements) and the same buffer is reused on all events. The event structure has to be copied when more than one events need to be referenced at the same time.

`Window-main-loop` is the function which captures all events sent from the X server and delivers them to each window where the event happened.

event [variable]

a 25-element integer-vector holding the most recent event structure.

next-event [function]

stores the event structure in `event` and returns it if there is at least one pending event, NIL if there is no pending event.

event-type *event* [function]

returns the keyword symbol representing the event-type in the *event* structure. The event-type keywords are: `:KeyPress` (2), `:KeyRelease` (3), `:ButtonPress` (4), `:ButtonRelease` (5), `:MotionNotify` (6), `:EnterNotify` (7), `:LeaveNotify` (8), `:FocusIn` (9), `:FocusOut` (0), `:KeymapNotify` (1), `:Expose` (12), `:GraphicsExpose` (13), `:NoExpose` (14), `:VisibilityNotify` (15), `:CreateNotify` (16), `:DestroyNotify` (17), `:UnmapNotify` (18), `:MapNotify` (19), `:MapRequest` (20), `:ConfigureNotify` (22), `:ConfigureRequest` (23), `:GravityNotify` (24), `:ResizeRequest` (25), `:CirculateNotify` (26), `:CirculateRequest` (27), `:PropertyNotify` (28), `:SelectionClear` (29), `:SelectionRequest` (30), `:SelectionNotify` (31), `:ColormapNotify` (32), `:ClientMessage` (33), `:MappingNotify` (34), `:LASTEvent` (35).

event-window *event* [function]

returns the window object where the *event* occurred.

event-x *event* [function]

extracts the x coordinate, (i.e., the horizontal position of the mouse pointer relatively in the window) out of the *event*.

event-y *event* [function]

extracts the x coordinate, (i.e., the vertical position of the mouse pointer relatively in the window) out of the *event*.

event-width *event* [function]

returns the eighth element of the *event* structure which represents the width parameter at the `:configureNotify` event.

event-height *event* [function]

returns the ninth element of the *event* structure which represents the height parameter at the `:configureNotify` event.

event-state *event* [function]

returns a list of keywords representing the mouse button and modifier key state. Keywords are: `:shift, :control, :meta, :left, :middle` and `:right`. For example, if left mouse button is pressed while shift key is down, `(:shift :left)` is returned.

display-events [function]

displays all xwindow events captured by `x:nextevent`. Control-C is the only way to terminate this function.

window-main-loop *&rest forms* [macro]

receives Xevents and delivers them to window objects where the event occurred. According to the event-type, methods in the window's class named `:KeyPress, :KeyRelease, :ButtonPress, :ButtonRelease, :MotionNotify, :EnterNotify, :LeaveNotify` and `:ConfigureNotify` are invoked with *event* as the argument. If *forms* is given, evaluates them each time event arrival is checked.

window-main-thread [function]

Do the same thing as `window-main-loop` in a different thread. `Window-main-thread` is only available on Solaris2. `Window-main-thread` installs an error handler which does not enter a read-eval-print loop. After printing the error information, the event processing continues.

17.2 Panel

panel [Class]

:super	**xwindow**
:slots	(pos items fontid
	rows columns ;total number of rows and columns
	next-x next-y
	item-width item-height)

Panel is a xwindow with the capability to lay out panel-items or any xwindows including other panel objects. A panel object supplies the default font for every panel-item created in the panel. Application windows should be defined as subclasses of the `Panel`.

:create [method]

&rest args &key (item-height 30) (item-width 50)
(font font-lucidasans-bold-12) (background *bisque1*)
&allow-other-keys

creates and initializes a panel. Since superclass's `:create` is invoked, all creation parameters for **xwindow**, such as *width, height, border-width*, etc., are allowed. *Item-height* and item-width give the minimum height and width for each panel-item.

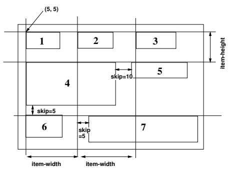


図 15: Item lay-out in panel

:items [method]

returns the list of all items associated.

:locate-item *item &optional x y* [method]

Item is any xwindow object, normally a panel-item. If *x* and *y* are given, the item is located there. Otherwise, *item* is located adjacent to the most recently located item. Items are located from top to bottom, from left to right, as shown in **Fig. ??**. `:Locate-item` also adds *item* in the *items* and *subwindows* list, and makes it visible by sending `:map`.

:create-item [method]

klass label receiver method &rest args
&key (font fontid)
&allow-other-keys

creates an instance of the panel-item class specified by *klass* (i.e., `button-item, menu-button-item, slider-item, joystick-item`, etc.), and place the item in the panel using `:locate-item`. *Args* are passed to `klass`'s `:create` method. *Label* is the identification string drawn in the panel item. *Receiver* and *method* specify the event handler called upon the corresponding event.

:delete-items [method]

delete all panel-items.

:create-menubar [method]

&rest args
&key (font fontid)
&allow-other-keys

creates a *menubar-panel* and locates it at the top of the panel.

The following methods are provided to avoid "subclass's responsibility" warning message when events are sent to panels without event handlers. User applications should override these methods.

:quit *&rest a* [method]

throws `:window-main-loop` and terminates event processing.

:KeyPress *event* [method]

returns NIL.

:KeyRelease *event* [method]

returns NIL.

:ButtonPress *event* [method]

returns NIL.

:ButtonRelease *event* [method]

returns NIL.

:MotionNotify *event* [method]

returns NIL.

:EnterNotify *event* [method]

returns NIL.

:LeaveNotify *event* [method]

returns NIL.

17.2.1 Subpanels (menu-panel and menubar-panel)

menu-panel [Class]

:super	**panel**
:slots	(items item-dots item-height charwidth charheight height-offset highlight-item color-pixels active-color)

`Menu-panel` is a kind of panel that can locate only `button-items` and/or `bitmap-button-items`. Unlike `panel`, however, `menu-panel` is normally invisible and is exposed when the `button-item` to which the `menu-panel` is associated is pressed. If a `menu-panel` is made always visible, it becomes a pinned menu. The response of each button-item to mouse events is slightly different from button-items in other panels, as the mouse button has been pressed somewhere outside the button-item. Creation of a `menu-panel` should follow the order described below:

1. create a menu-panel by `(instance menu-panel :create)`.
2. create button-items or/and bitmap-button-items and locate them in the menu-panel by `(send aMenuPanel :create-item button-item "BTN" obj meth)`.
3. create a menu-button-item in another panel and associate the menu-panel with the menu-button-item by `(instance menu-button-item :create "Option" obj meth :menu-window aMenuPanel)`.

:create [method]

&rest args &keyitems (border-width 0) (font font-courb12)

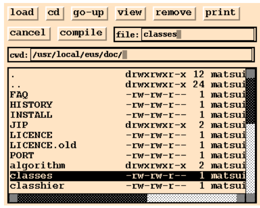


図 16: FilePanel window

(width 100) (height-offset 15) (color *bisque1*) (active *bisque2*)
&allow-other-keys

create a menu-panel window. The size of the window is expanded each time new menu-item is added.

:create-item *class label receiver method &rest mesg* [method]

adds a menu item in this menu-panel window and attatches the corresponding action. The *receiver* objects receives *mesg* when the mouse button is released on the item.

menubar-panel [Class]

:super **panel**
:slots

`Menubar-panel` is a subpanel always located at the top of the parent panel. A menubar-panel resembles with the Macintosh desktop's menubar which lets out several pull-down menus. Panel-items placed in the menubar should be `menu-button-items`. A menubar-panel is created by the panel's `:create-menubar` method.

17.2.2 File Panel

The FilePanel is an application window for the interactive manipulation of files and directories. Using `cd` and `go-up` buttons, any directory can be visited and files contained in the directory are displayed in the `ScrollTextWindow` below. Text files can be displayed in different windows (textViewPanel). Files can also be printed, removed, and compiled by simply cliking buttons. When a file is printed, `a2ps` *`file`* `| lpr` commands are executed in a forked process.

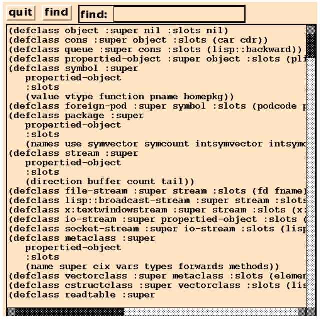


図 17: TextViewPanel window

17.2.3 Text View Panel

TextViewPanel is an application window class to display text files (Fig. ??). The program text is shown to demonstrate how one of the simplest application windows is described. In the `:create` method, the quit button and find button, and a text-item to feed the string to be searched for in the file are created. The view-window is a ScrollTextWindow that displays the file with the vertical and horizontal scroll-bars. The TextViewPanel captures `:ConfigureNotify` event to resize the view-window when the outermost title window is resized by the window manager.

```
(defclass TextViewPanel :super panel
        :slots (quit-button find-button find-text view-window))

(defmethod TextViewPanel
 (:create (file &rest args &key (width 400) &allow-other-keys)
    (send-super* :create :width width args)
    (setq quit-button
          (send self :create-item panel-button "quit" self :quit))
    (setq find-button
          (send self :create-item panel-button "find" self :find))
    (setq find-text
          (send self :create-item text-item "find: " self :find))
    (setq view-window
            (send self :locate-item
                (instance ScrollTextWindow :create
                   :width (setq width (- (send self :width) 10))
                   :height (- (setq height (send self :height)) 38)
                   :scroll-bar t :horizontal-scroll-bar t
                   :map nil      :parent self)))
    (send view-window :read-file file))
 (:quit (event)  (send self :destroy))
```

```
(:find (event)
   (let ((findstr (send find-text :value)) (found)
         (nlines (send view-window :nlines)))
       (do ((i 0 (1+ i)))
           ((or (>= i nlines) found))
          (if (substringp findstr (send view-window :line i)) (setq found i)))
       (when found
          (send view-window :display-selection found)
          (send view-window :locate found))))
(:resize (w h)
   (setq width w height h)
   (send view-window :resize (- w 10) (- h 38)))
(:configureNotify (event)
  (let ((newwidth (send self :width))
        (newheight (send self :height)))
       (when (or (/= newwidth width) (/= newheight height))
         (send self :resize newwidth newheight)))  ) )
```

17.3 Panel Items

panel-item [Class]

:super **xwindow**

:slots (pos notify-object notify-method fontid label labeldots)

Panel-item is an abstract class for all kinds of panel-item windows to invoke *notify-object*'s *notify-method* when item-specific event occurs.

:notify *&rest args* [method]

invokes *notify-object*'s *notify-method*. Responsive events and arguments passed to *notify-method* are item specific:

button-item The button is pressed and released in the same button-item; the argument is the button-item object.

menu-button-item A menu item is selected; the argument is the menu-button-item object.

choice-item A new choice button is selected; the arguments are the choice-item object and the index number of the choice.

text-item A newline or return is entered; the arguments are the text-item object and the entire line (string).

slider-item The slider nob is grabbed and moved; the arguments are the slider-item object and the new value.

joystick-item The joystick is grabbed and moved; the arguments are the slider-item object, the new x and y values.

:create [method]

name reciever method &rest args

&key (width 100) (height 100) (font font-courb12)
&allow-other-keys

creates a panel-item. As panel-item is an abstract class, this method should only be called by the subclasses via `send-super`.

button-item [Class]

:super **panel-item**
:slots

button-item is the simplest panel-item. Button-item has a rectangular box and a label string in it. When clicked, button-item invokes *notify-object*'s *notify-method* with the panel-item object as the only argument.

:draw-label *&optional (state :top) (color bg-color) (border 2) offset* [method]

draws button-item's label.

:create [method]

label revciever method &rest args
&keywidth height (font (send parent :gc :font))
(background (send parent :gc :background))
(border-width 0)
(state :top)
&allow-other-keys

creates a button-item. If button's width and height are not given, the sizes are automatically set to accomodate the label string drawn with the given font. Though the border-width is defaulted to 0, pseudo 3D representation embosses the button. The background color and font are defaulted to the ones defined for the parent window, i.e. a panel.

:ButtonPress *event* [method]

changes the background color to gray, as if the button.

:ButtonRelease *event* [method]

changes *event*'s background color to normal.

menu-button-item [Class]

:super **button-item**
:slots (items item-dots item-labels
charwidth charheight
menu-window window-pos high-light)

defines a pulldown menu. Though a `menu-button-item` looks like a `button-item`, the `menu-button-item` activates associated `menu-panel` below the button when it is pressed, instead of sending an immediate message to the *notify-object*. The actual message is sent when the mouse button is released on one of the menu items.

:create [method]

label reciever method
&rest args
&key menu items (state :flat)

&allow-other-keys

creates a pulldown menu button. *Receiver* and *method* arguments has no effect.

:ButtonPress *event* [method]

reverses the appearance of the pulldown-menu and exposes the associated menu-panel below the button.

:ButtonRelease *event* [method]

unmaps the `menu-panel` below this button and reverts the appearance of the button.

bitmap-button-item [Class]

:super **button-item**
:slots (pixmap-id bitmap-width bitmap-height)

Though `bitmap-button-item`'s function is similar to the `button-item`, its appearance is different. Instead of drawing a simple label string on the button, as is the case for `button-item`, `bitmap-button-item` is drawn by a pixmap which is loaded from a bitmap-file when the button is created.

:draw-label *&optional (state :flat) color bg-color (border 2)* [method]

draws a bitmap/pixmap on the button.

:create [method]

bitmap-file reciever method &rest args
&key width height
&allow-other-keys

creates bitmap-button-item. The first argument, *bitmap-file* replaces the *label* argument of `button-item`.

:draw-label *&optional (state :flat) (color bg-color) (border 2)* [method]

draw a bitmap/pixmap on the button.

:create-bitmap-from-file *fname* [method]

creates pixmap from the bitmap file named *fname*, and stores its id in *pixmap-id*.

choice-item [Class]

:super **button-item**
:slots (choice-list active-choice transient-choice
choice-dots choice-pos button-size)

`choice-item` is a set of round choice buttons. One choice is always active, and only one choice can become active at the same time. `choice-item` provides the similar function as radio-buttons.

:create [method]

label reciever method &rest args
&key (choices '("0" "1")) (initial-choice 0)
(font (send parent :gc :font))
(button-size 13)
(border-width 0)

create a choice-item-button. Each choice button is a circle of radius *button-size*. When a new choice is selected, *notify-object*'s *notify-method* is invoked with the choice-item object and the index of the choice selected.

:value *&optional new-choice invocation* [method]

If *new-choice* is given, it is set as the current active choice, and the corresponding circle is filled black. If *invocation* is also specified, *notify-object*'s *notify-method* is invoked. `:Value` returns the current (or new) choice index.

:draw-active-button *&optional (old-choice active-choice) (new-choice active-choice)* [method]

draw active button.

:ButtonPress *event* [method]

If the mouse button is pressed on any of the choice buttons, its index is recorded in *transient-choice*. No further action is taken until the mouse button is released.

:ButtonRelease *event* [method]

If the mouse button is released on the same button which is already pressed, the *active-choice* is updated and *notify-object*'s *notify-method* is invoked.

slider-item [Class]

:super	**panel-item**
:slots	(min-value max-value value minlabel maxlabel valueformat bar-x bar-y bar-width bar-height valuedots label-base nob-x nob-moving charwidth)

While `choice-item` is used to select a discrete value, `slider-item` is used for the continuous value in the range between *min-value* and *max-value*. Each moment the value is changed, *notify-object*'s *notify-method* is invoked with the slider-item object and the new value as the arguments.

:create [method]

label reciever method &rest args
&key (min 0.0) (max 1.0) parent
(min-label "") (max-label "") (value-format " 4,2f")
(font font-courb12) (span 100) (border-width 0) (initial-value min)

creates slider-item. The sliding knob is displayed as a small black rectangle on a bar. The left end represents the *min* value and the right end *max* value. The length of the bar stretches for the *span* dots. The current value is displayed to the right of the slider-item label in the *value-format*.

:value *&optional newval invocation* [method]

If *newval* is given, it is set as the current value, and the knob is slided to the corresponding location. If *invocation* is also specified non nil, *notify-object*'s *notify-method* is invoked. `:Value` returns the current (or new) value.

joystick-item [Class]

:super	**panel-item**
:slots	(stick-size min-x min-y max-x max-y center-x center-y stick-x stick-y value-x value-y stick-return stick-grabbed fraction-x fraction-y)

`joystick-item` can be regarded as the two-dimensional slider-item. Two continuous values can be specified by the moving black circle on the coaxial chart that looks like a web (Fig. **??**).

:create [method]

name reciever method &rest args
&key (stick-size 5) return
(min-x -1.0) (max-x 1.0)
(min-y -1.0) (max-y 1.0)
&allow-other-keys

Stick-size is the radius of the stick's black circle. The sizes of the circles in the coaxial chart are determined according to the width and height of the joystick-item window. If *return* is non-NIL, the joystick returns to the origin when the mouse button is released. Otherwise, the joystick remains at the released position.

:value *&optional newx newy invocation* [method]

If both *newx* and *newy* are given, they are set as the current values, and the joystick moves to the corresponding location on the coaxial chart. If *invocation* is also specified non nil, *notify-object*'s *notify-method* is invoked with the joystick-item object and x and y values as the arguments. `:Value` returns the list of current (or new) values.

The following short program shows how to use panel-items described above, and Fig. **??** depicts how they appear in a panel.

```
(in-package "X")
(defclass testPanel :super panel
        :slots (quit joy choi sli))
(defmethod testPanel
 (:create (&rest args)
    (send-super* :create :width 210 :height 180
                 :font font-courb12 args)
    (send-super :create-item button-item "quit" self :quit :font font-courb14)
    (send-super :create-item choice-item "choice" self :choice
                :choices '(" A " " B " " C ")
                :font font-courb12)
    (send-super :create-item slider-item "slider" self :slider
                :span 90)
    (send-super :create-item joystick-item "joy" self :joy)
    self)
 (:choice (obj c) (format t "choice: ~S ~d~%" obj c))
 (:slider (obj val) (format t "slider: ~S ~s~%" obj val))
 (:joy (obj x y) (format t "joy: ~S ~s ~s~%" obj x y)) )
(instance testPanel :create)
(window-main-thread)
```

text-item [Class]

:super	**panel-item**
:slots	(textwin)

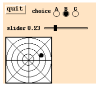


図 18: Panel items created in a panel

`Text-item` is used to display or to input one short line of text, such as a file name. A `text-item` has a label string followed by a small textwindow on the right. When the pointer is put in the textwindow, key input is enabled and the characters typed are buffered. Line editing is available in the textwindow: `control-F` and `control-B` to move forward/backward by one character, `del` to delete the character on the left of the cursor, `control-D` to delete the character on the cursor, and any graphical character to insert it at the cursor position. Clicking a mouse button moves the cursor to the clicked character. Hitting an enter (newline) key causes the buffered text to be sent to the *notify-object*'s *notify-method*.

:create [method]

label revciever method &rest args
&key (font font-courb12) (columns 20) initial-value (border-width 0)
&allow-other-keys

creates text-item. Though the linebuffer of the textwindow may have unlimited length, visible portion is restricted to the *columns* characters.

:getstring [method]

returns the string in the key buffer.

17.4 Canvas

canvas [Class]

:super **xwindow**
:slots (topleft bottomright)

Canvas is a xwindow to interact with figures or images. Currently, only the region selection capability has been implemented. At the buttonPress event, the canvas begins to draw a rectangle with the topleft corner at the pressed position and bottomright corner at the current pointer. ButtonRelease causes the `notify-method` to be sent to the `notify-object`. Use Xdrawable's methods to draw figures or images in the canvas.

17.5 Text Window

There are three textwindow classes, `TextWindow`, `BufferTextWindow` and `ScrollTextWindow`.

textWindow [Class]

```
:super   xwindow
:slots   (fontid
         charwidth charheight charascent dots
         win-row-max win-col-max
         win-row win-col           ;physical current position in window
         x y
         charbuf                   ; for charcode conversion
         keybuf keycount           ;for key input
         echo
         show-cursor cursor-on     ;boolean
         kill delete               ;control character
         notify-object notify-method
         )
```

realizes virtual terminals usable for displaying messages. The displayed contents are not buffered and there is no way to retrieve a line or a character already displayed in the TextWindow. Basically, `TextWindow` has similar capabilities to the dumb terminals, that are, moving the cursor, erasing lines, erasing areas, scrolling displayed texts, inserting strings, etc. Also, the text cursor can be moved to the position designated by the mouse pointer.

:init *id* [method]

initializes *id*th text-window.

:create [method]

&rest args
&key width height (font font-courb14) rows columns
show-cursor notify-object notify-method
&allow-other-keys

creates text-window. The sizes of the window may be specified either by *width* and *height* or by *rows* and *columns*. *Notify-object*'s *notify-method* is invoked when a newline character is typed in.

:cursor *flag* [method]

The *flag* can either be `:on, :off` or `:toggle`. The text cursor is addressed by the *win-row* and *win-col*. The text cursor is displayed if *flag* is `:on`, is erased if *flag* is `:off`, or is reversed if *flag* is `:toggle`. This method must be invoked frequently whenever the character at the cursor is updated.

:clear [method]

clears text-window.

:clear-eol *&optional (r win-row) (c win-col) (csr :on)* [method]

clears the rest of the line after the character addressed by r and *c*, including the character at the cursor.

:clear-lines *lines &optional (r win-row)* [method] clears multiple lines after r-th row.

:clear-eos *&optional (r win-row) (c win-col)* [method]

clears the region after the character addressed by r and c till the end-of-the-screen.

:win-row-max [method]

returns the maximum number of lines displayable in this window.

:win-col-max [method]

returns the maximum number of columns displayable in this window.

:xy *&optional (r win-row) (c win-col)* [method]

calculates the pixel coordinates of the character addressed by r and c.

:goto *r c &optional (cursor :on)* [method]

moves the cursor to r-th row and c-th column.

:goback *&optional (csr :on)* [method]

moves the cursor backward by one.

:advance *&optional (n 1)* [method]

moves the cursor forward by n characters.

:scroll *&optional (n 1)* [method]

scroll textwindow vertically by n lines.

:horizontal-scroll *&optional (n 1)* [method]

horizontally scrolls the text by n columns.

:newline [method]

moves cursor to the beginning of the next line.

:putch *ch* [method]

inserts the character *ch* at the cursor position. The rest of the line is moved forward by one.

:putstring *str &optional (e (length str))* [method]

places *str* at the cursor position.

:event-row *event* [method]

:event-col *event* [method]

returns the text cursor position designated by (x, y) in the *event*.

:KeyPress *event* [method]

inserts the character entered at the cursor position. If the character is newline, notification is sent to the *notify-object*.

textWindowStream [Class]

:super	**stream**
:slots	(textwin)

`TextWindowStream` is an output stream connected to a TextWindow. Characters or strings output to this stream by using `print, format, write-byte`, etc., are displayed in the textwindow. As usual file streams, the output data are buffered.

:flush [method]

flushes buffered text string and send them to the textwindow. `Finish-output` or writing a newline character to this stream automatically calls this method.

make-text-window-stream *xwin* [function]

makes text-window-stream and returns the stream object.

BufferTextWindow [Class]

:super	**TextWindow**
:slots	(linebuf expbuf max-line-length row col)

maintains the line buffer representing the contents of the textwindow. Linebuf is the vector of lines. Expbuf holds tab-expanded text. Only lines displayable in the window are maintained. BufferTextWindows can be used as simple text editors which have several, often only one, lines of text. `Text-item` employs a BufferTextWindow as a displayable line buffer.

:line *n* [method]

returns the contents of the *n*-th line as a string.

:nlines [method]

returns number of lines in the linebuf.

:all-lines [method]

returns the linebuf, which is a vector of strings.

:refresh-line *&optional (r win-row) (c win-col)* [method]

redraws the *r*-th line after the *c*-th column.

:refresh *&optional (start 0)* [method]

redraws the lines after the *start*-th line inclusively.

:insert-string *string* [method]

inserts *string* at the cursor position.

:insert *ch* [method]

inserts the character at the cursor.

:delete *n* [method]

deletes *n* characters after the cursor.

expand-tab *src &optional (offset 0)* [function]

Src is a string possibly containing tabs. These tabs are replaced by spaces assuming the tab stops at every 8th position.

ScrollTextWindow [Class]

:super	**BufferTextWindow**	
:slots	(top-row top-col	;display-starting position
	scroll-bar-window	
	horizontal-scroll-bar-window	

selected-line)

`ScrollTextWindow` defines buffertextwindow with unlimited number of lines, and vertical and horizontal scroll-bars can be attached. ScrollTextWindow can handle `:configureNotify` event to resize itself and accompanying scroll-bar windows, and to redisplay texts. By clicking, a line can be selected.

:create *&rest args &key scroll-bar horizontal-scroll-bar &allow-other-keys* [method]

When scroll-bars are needed, specify T to each keyword argument.

:locate *n* [method]

displays the buffered text by placing the *n*-th line at the top of the window.

:display-selection *selection* [method]

Selection represents the location of the selected line. The entire seleced line is displayed highlighted.

:selection [method]

returns the selected line (string).

:read-file *fname* [method]

reads the textfile specified by *fname* into the *linebuf*, expands tabs, and display in the window. The cursor is put at the beginning of the screen.

:display-string *strings* [method]

Strings is a sequence of lines (strings). The *strings* are copied in the linebuf and displayed in the window.

:scroll *n* [method]

vertically scrolls *n* lines.

:horizontal-scroll *n* [method]

horizontally scrolls *n* columns.

:buttonRelease *event* [method]

The line where the mouse pointer is located is selected. If notification is specified when the window is created, *notify-object*'s *notify-method* is invoked.

:resize *w h* [method]

changes the size of the window and redisplays the contents according to the new size. The same message is sent to scroll-bars if attached.

第III部
irteus Extension

18 Robot Modelling

18.1 Robot Data Structure and modeling

18.1.1 Robot Data Structures and Forward Kinematics

The structure of a robot can be considered to consist of links and joints. As a way to divide the robot into joints and links

- (a)Include joints on the side of the disconnected link
- (b)Include joints in the torso or closer to the torso

can be considered. Considering the data structure of the computer, (a) is used. The reason is that in all links other than the body, The structure always contains one joint, and all links are handled by the same algorithm This is because it is possible to

A tree structure is used to represent the links divided in this way on a computer. It is possible. In general, when creating a tree structure, the data structure is divided into binary trees. The structure is often simplified.

As a method of obtaining the homogeneous transformation matrix in the robot link, Set Σ_j with the origin, and the rotation axis vector seen from the parent link coordinate system is The origin of a_j and Σ_j is b_j, and the joint angle of rotation is q_j.

At this time, the parent-link-relative homogeneous transformation matrix of Σ_j is

$$^iT_j = \begin{bmatrix} e^{\hat{a}_j q_j} & b_j \\ 0\ 0\ 0 & 1 \end{bmatrix}$$

where $e^{\hat{a}_j q_j}$ is the rotation generated by the angular velocity vector of constant velocity. It uses the following Rodrigues formula to give the transpose matrix. around the rotation axis a It is used as a rotation matrix that rotates by $wt[rad]$.

$$e^{\hat{\omega}t} = E + \hat{a}sin(wt) + \hat{a}^2(1 - cos(wt))$$

Assuming that the position and orientation p_i, R_i of the parent link are known, the homogeneous transformation matrix of Σ_i is

$$T_i = \begin{bmatrix} R_i & p_i \\ 0\ 0\ 0 & 1 \end{bmatrix}$$

and from here

$$T_j = T_i\ {}^iT_j$$

can be calculated as this from the root link of the robot to all the links for the first time To calculate posture information from the joint angle information of the whole body of the robot by applying it sequentially This is called forward kinematics.

18.1.2 Geometric Information Modeling with EusLisp

Geometric modeling in EusLisp involves the generation of a basic model (body), the composition function of the body, There are three steps to creating a combined model (bodyset).

So far, we have seen that it is possible to generate and synthesize the following basic models.

```
(setq c1 (make-cube 100 100 100))
(send c1 :locate #f(0 0 50))
(send c1 :rotate (deg2rad 30) :x)
(send c1 :set-color :yellow)
(objects (list c1))

(setq c2 (make-cylinder 50 100))
(send c2 :move-to
      (make-coords
       :pos #f(20 30 40)
       :rpy (float-vector 0 0 (deg2rad 90)))
      :world)
(send c2 :set-color :green)
(objects (list c1 c2))

(setq c3 (body+ c1 c2))
(setq c4 (body- c1 c2))
(setq c5 (body* c1 c2))
```

A bodyset is a composite model introduced in irteus, which can handle multiple objects and It is for handling multiple colors.

```
(setq c1 (make-cube 100 100 100))
(send c1 :set-color :red)
(setq c2 (make-cylinder 30 100))
(send c2 :set-color :green)
(send c1 :assoc c2)          ;;; Remember this
(setq b1 (instance bodyset :init
                   (make-cascoords)
                   :bodies (list c1 c2)))
(objects (list b1))
```

18.1.3 Sample Program Using Parent-Child Relationship of Geometric Information

```
(setq c1 (make-cube 100 100 100))
(setq c2 (make-cube 50 50 50))
(send c1 :set-color :red)
(send c2 :set-color :blue)
(send c2 :locate #f(300 0 0))
(send c1 :assoc c2)
(objects (list c1 c2))
(do-until-key
 (send c1 :rotate (deg2rad 5) :z)
 (send *irtviewer* :draw-objects)
 (x::window-main-one) ;; process window event
 )
```

18.1.4 Modeling a Robot (Multi-Link System) using *bodyset-link* and *joints*

irteus provides a class called bodyset-link (irtmodel.l) as a class for describing robot links. It has mechanical information and geometric information, and the structure of the robot is represented by a general tree structure. In addition, joint information is handled using the joint class.

```
(defclass bodyset-link
  :super bodyset
```

```
  :slots (joint parent-link child-links analysis-level default-coords
               weight acentroid inertia-tensor
               angular-velocity angular-acceleration
               spacial-velocity spacial-acceleration
               momentum-velocity angular-momentum-velocity
               momentum angular-momentum
               force moment ext-force ext-moment))
```

The joint class (irtmodel.l) is used for modeling joints. The joint class is a base class, and actually uses rotational-joint, linear-joint, etc. A joint created by a child class of joint can specify the joint angle with the :joint-angle method.

```
(defclass joint
  :super propertied-object
  :slots (parent-link child-link joint-angle min-angle max-angle
default-coords))
(defmethod joint
  (:init (&key name
              ((:child-link clink)) ((:parent-link plink))
              (min -90) (max 90) &allow-other-keys)
        (send self :name name)
        (setq parent-link plink child-link clink
              min-angle min max-angle max)
        self))

(defclass rotational-joint
  :super joint
  :slots (axis))
(defmethod rotational-joint
  (:init (&rest args &key ((:axis ax) :z) &allow-other-keys)
        (setq axis ax joint-angle 0.0)
        (send-super* :init args)
        self)
  (:joint-angle
   (&optional v)
   (when v
       (setq relang (- v joint-angle) joint-angle v)
       (send child-link :rotate (deg2rad relang) axis)))
     joint-angle))
```

Here we use joint, parent-link, child-links and defualt-coords.

If you make a servo module as an example of a simple one-joint robot,

```
(defun make-servo nil
  (let (b1 b2)
    (setq b1 (make-cube 35 20 46))
    (send b1 :locate #f(9.5 0 0))
    (setq b2 (make-cylinder 3 60))
    (send b2 :locate #f(0 0 -30))
    (setq b1 (body+ b1 b2))
    (send b1 :set-color :gray20)
    b1))

(defun make-hinji nil
  (let ((b2 (make-cube 22 16 58))
       (b1 (make-cube 26 20 54)))
    (send b2 :locate #f(-4 0 0))
    (setq b2 (body- b2 b1))
    (send b1 :set-color :gray80)
    b2))

(setq h1 (instance bodyset-link :init (make-cascoords) :bodies (list (make-hinji))))
(setq s1 (instance bodyset-link :init (make-cascoords) :bodies (list (make-servo))))
(setq j1 (instance rotational-joint :init :parent-link h1 :child-link s1 :axis :z))
;; instance cascaded coords
(setq r (instance cascaded-link :init))
(send r :assoc h1)
```

```
(send h1 :assoc s1)
(setq (r . links) (list h1 s1))
(setq (r . joint-list) (list j1))
(send r :init-ending)
```

Here, a `bodyset-link` of `h1` and `s1` and a `rotational-joint` of `j1` are created, and from here `cascaded-link` A model consisting of connecting links is generated. `cascaded-link` is a child class of `cascaded-coords`, so the parent-child relationship of `r (cascaded-link)`, `h1`, and `s1` is set using `:assoc`.

The notation `(r . links)` accesses `links` which is a slot variable (member variable) of the object `r`. here, Appropriate values are set for `links` and `joint-list`, and necessary initialization is performed as `(send r :init-ending)`.

This will generate one object called `r` containing two links and one joint information. For example, instead of `(objects (list h1 s1))`, the robot can be displayed in the viewer as `(objects (list r))`. You can also use `(send r :locate #f(100 0 0))`.

The following is an example of using the methods of the `cascaded-link` class. Provides access to joint angle vectors, as well as accessors to joint and link lists such as `:joint-list` and `:links` The `:angle-vector` method is important. If you call this without arguments, you can get the current joint angles. If you call this with joint angle vectors as arguments, you can reflect the joint angle vectors indicated by the arguments in the robot model.

```
$ (objects (list r))
(#<servo-model #X628abb0  0.0 0.0 0.0 / 0.0 0.0 0.0>)
;; useful cascaded-link methods
$ (send r :joint-list)
(#<rotational-joint #X6062990 :joint101067152>)
$ (send r :links)
(#<bodyset-link #X62ccb10 :bodyset103598864  0.0 0.0 0.0 / 0.0 0.0 0.0>
 #<bodyset-link #X6305830 :bodyset103831600  0.0 0.0 0.0 / 0.524 0.0 0.0>)
$ (send r :angle-vector)
#f(0.0)
$ (send r :angle-vector (float-vector 30))
#f(30.0)
```

18.1.5 Modeling of Robot (Multi-Link System) Using cascaded-link

On the other hand, there is a class called cascaded-link as a class for modeling multi-link systems. It has slot variables such as links and joint-list, to which lists of instances of bodyset-link and joint are bound and used. The following is an example of how to define a child class of `cascaded-link` and perform initialization processing related to robot modeling here.

```
(defclass cascaded-link
  :super cascaded-coords
  :slots (links joint-list bodies collision-avoidance-links))

(defmethod cascaded-link
  (:init (&rest args &key name &allow-other-keys)
         (send-super-lexpr :init args)
         self)
  (:init-ending
   ()
   (setq bodies (flatten (send-all links :bodies)))
   (dolist (j joint-list)
     (send (send j :child-link) :add-joint j)
     (send (send j :child-link) :add-parent-link (send j :parent-link))
     (send (send j :parent-link) :add-child-links (send j :child-link)))
   (send self :update-descendants))
)


(defclass servo-model
```

```
  :super cascaded-link
  :slots (h1 s1 j1))
(defmethod servo-model
  (:init ()
   (let ()
     (send-super :init)
     (setq h1 (instance bodyset-link :init (make-cascoords) :bodies (list (make-hinji))))
     (setq s1 (instance bodyset-link :init (make-cascoords) :bodies (list (make-servo))))

     (setq j1 (instance rotational-joint :init :parent-link h1 :child-link s1 :axis :z))

     ;; instance cascaded coords
     (setq links (list h1 s1))
     (setq joint-list (list j1))
     ;;
     (send self :assoc h1)
     (send h1 :assoc s1)
     ;;
     (send self :init-ending)
     self))
  ;;
  ;; (send r :j1 :joint-angle 30)
  (:j1 (&rest args) (forward-message-to j1 args))
  )

(setq r (instance servo-model :init))
```

If you define a class like this and run `(setq r (instance servo-model :init))`, you can create an instance of the robot model in the same way, and use the previous method. The advantage of defining a class is that the method definition `(:j1 (&rest args) (forward-message-to j1 args))` provides an accessor to the instance of the joint model. This allows you to use `(send r :j1 :joint-angle)` or `(send r :j1 :joint-angle 30)` It is possible to instruct When moving this robot, just like the previous example

```
(objects (list r))
(dotimes (i 300)
   (send r :angle-vector (float-vector (* 90 (sin (/ i 100.0)))))
   (send *irtviewer* :draw-objects))
```

and so on.

```
(setq i 0)
(do-until-key
  (send r :angle-vector (float-vector (* 90 (sin (/ i 100.0)))))
  (send *irtviewer* :draw-objects)
  (incf i))
```

will keep the program running until the next time you press the keyboard.

In addition, an example of modeling a 3-link 2-joint robot using this by extending it a little is shown below. All joint angle sequences can be specified by giving a vector #f(0 0) to the method :joint-angle.

```
(defclass hinji-servo-robot
  :super cascaded-link)
(defmethod hinji-servo-robot
  (:init
   ()
   (let (h1 s1 h2 s2 l1 l2 l3)
     (send-super :init)
     (setq h1 (make-hinji))
     (setq s1 (make-servo))
     (setq h2 (make-hinji))
     (setq s2 (make-servo))
     (send h2 :locate #f(42 0 0))
     (send s1 :assoc h2)
     (setq l1 (instance bodyset-link :init (make-cascoords) :bodies (list h1)))
```

```
     (setq l2 (instance bodyset-link :init (make-cascoords) :bodies (list s1 h2)))
     (setq l3 (instance bodyset-link :init (make-cascoords) :bodies (list s2)))
     (send l3 :locate #f(42 0 0))

     (send self :assoc l1)
     (send l1 :assoc l2)
     (send l2 :assoc l3)

     (setq joint-list
           (list
            (instance rotational-joint
                      :init :parent-link l1 :child-link l2
                      :axis :z)
            (instance rotational-joint
                      :init :parent-link l2 :child-link l3
                      :axis :z)))
     (setq links (list l1 l2 l3))
     (send self :init-ending)
     )))
(setq r (instance hinji-servo-robot :init))
(objects (list r))

(dotimes (i 10000)
  (send r :angle-vector (float-vector (* 90 (sin (/ i 500.0))) (* 90 (sin (/ i 1000.0)))))
  (send *irtviewer* :draw-objects))
```

18.1.6 Forward Kinematics Calculations in EusLisp

For forward kinematics calculations, use the :worldcoords method defined in the cascaded-corods, bodyset, and bodyset-link classes. The :worldcoords method performs a forward kinematics calculation by calling the :worldcoords method of the parent link backward until the root link is found (the parent link disappears) or until a link is found for which the slot variable changed is nil (a forward kinematics calculation has once been performed). The calculation is performed by calling the :worldcoords method of the parent link. In doing so, the slot variable changed is overwritten with nil. Therefore, the second call to the :worldcoords method does not calculate the forward kinematics of the link that has been calculated once, and the position and orientation information of the link can be retrieved immediately.

The :worldcoords method of the bodyset-link class can take a level argument, and if it is :coords, the forward kinematics calculation of the bodies slot variable of the link is not performed. If the bodies contains facesets that constitute the vertices of the link, then a significant speedup can be expected by omitting the forward kinematics calculation for these facesets. Since the initial value of the level argument is the analysis-level slot variable of the link, if you do not want to always calculate the forward kinematics of bodies, you can use `(send l :analysis-level :coords)` for the instance l of the link. for instance l of the link.

```
(defmethod bodyset-link
  (:worldcoords
   (&optional (level analysis-level))
   (case
    level
    (:coords (send-message self cascaded-coords :worldcoords))
    (t       (send-super :worldcoords)))
   ))

(defmethod bodyset
  (:worldcoords
   ()
   (when changed
     (send-super :worldcoords)
     (dolist (b bodies) (send b :worldcoords)))
   worldcoords))
```

```
(defmethod cascaded-coords
 (:worldcoords  ()      ;calculate rot and pos in the world
   (when changed
      (if parent
          (transform-coords (send parent :worldcoords) self
worldcoords)
          (send worldcoords :replace-coords self))
      (send self :update)
      (setf changed nil))
   worldcoords))
```

18.2 Robot Motion Generation

18.2.1 Inverse Kinematics

In inverse kinematics, the joint angle vector of the manipulator $\boldsymbol{\theta} = (\theta_1, \theta_2, ..., \theta_n)^T$ is obtained from the end-effector's position and orientation ${}^0_n\boldsymbol{H}$.

where position/orientation of end effector $\boldsymbol{r}$ In inverse kinematics, position/orientation of end effector ${}^0_n\boldsymbol{H}$ to joint of manipulator Find the angular vector $\boldsymbol{\theta} = (\theta_1, \theta_2, ..., \theta_n)^T$.

where position/orientation of end effector $\boldsymbol{r}$ is called the joint angle vector

$$\boldsymbol{r} = \boldsymbol{f}(\boldsymbol{\theta}) \tag{1}$$

Equation **??** is Equation **??** and obtain the joint angle vector.

$$\boldsymbol{\theta} = \boldsymbol{f}^{-1}(\boldsymbol{r}) \tag{2}$$

f^{-1} in is generally a nonlinear function. Therefore, by differentiating with respect to time t, we get the linear expression

$$\begin{aligned} \dot{\boldsymbol{r}} &= \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta})\dot{\boldsymbol{\theta}} && (3) \\ &= \boldsymbol{J}(\boldsymbol{\theta})\dot{\boldsymbol{\theta}} && (4) \end{aligned}$$

. where $\boldsymbol{J}(\boldsymbol{\theta})$ is the Jacobian matrix of $m \times n$. m is the dimension of vector $\boldsymbol{r}$ and n is the dimension of vector $\boldsymbol{\theta}$. $\dot{\boldsymbol{r}}$ is the velocity/angular velocity vector.

When the Jacobian matrix is nonsingular, we can solve this linear equation using the inverse matrix $\boldsymbol{J}(\boldsymbol{\theta})^{-1}$ as follows.

$$\dot{\boldsymbol{\theta}} = \boldsymbol{J}(\boldsymbol{\theta})^{-1}\dot{\boldsymbol{r}} \tag{5}$$

However, since Jacobian matrices are generally nonsingular, The pseudo-inverse of the Jacobian matrix $\boldsymbol{J}^{\#}(\boldsymbol{\theta})$ is used (Equation **??**).

$$\boldsymbol{A}^{\#} = \begin{cases} \boldsymbol{A}^{-1} \ (m = n = rank\boldsymbol{A}) \\ \boldsymbol{A}^T \ (\boldsymbol{A}\boldsymbol{A}^T)^{-1}(n > m = rank\boldsymbol{A}) \\ (\boldsymbol{A}^T\boldsymbol{A})^{-1}\boldsymbol{A}^T \ (m > n = rank\boldsymbol{A}) \end{cases} \tag{6}$$

Equation **??** is Equation **??** when $m > n$, Equation **??** as a problem of finding a least-squares solution to minimize, and obtain a solution.

$$\min_{\dot{\boldsymbol{\theta}}} \left(\dot{\boldsymbol{r}} - \boldsymbol{J}(\boldsymbol{\theta})\dot{\boldsymbol{\theta}}\right)^T \left(\dot{\boldsymbol{r}} - \boldsymbol{J}(\boldsymbol{\theta})\dot{\boldsymbol{\theta}}\right) \tag{7}$$

$$\min_{\dot{\boldsymbol{\theta}}} \quad \dot{\boldsymbol{\theta}}^T \dot{\boldsymbol{\theta}} \tag{8}$$
$$s.t. \quad \dot{\boldsymbol{r}} = \boldsymbol{J}(\boldsymbol{\theta})\dot{\boldsymbol{\theta}}$$

The joint angular velocity is obtained as follows.

$$\dot{\boldsymbol{\theta}} = \boldsymbol{J}^{\#}(\boldsymbol{\theta})\dot{\boldsymbol{r}} + \left(\boldsymbol{E}_n - \boldsymbol{J}^{\#}(\boldsymbol{\theta})\boldsymbol{J}(\boldsymbol{\theta})\right)\boldsymbol{z} \tag{9}$$

However, when solving according to Equation **??** , $\left|\dot{\boldsymbol{\theta}}\right|$ becomes large and unstable behavior occurs. Therefore, Nakamura et al.'s SR-Inverse[4] avoids this singularity.

In this study, instead of the pseudo-inverse $\boldsymbol{J}^{\#}(\boldsymbol{\theta})$ of the Jacobian matrix, we use $\boldsymbol{J}^{*}(\boldsymbol{\theta})$ shown in Equation **??** .

$$\boldsymbol{J}^{*}(\boldsymbol{\theta}) = \boldsymbol{J}^T\left(\boldsymbol{J}\boldsymbol{J}^T + \epsilon\boldsymbol{E}_m\right)^{-1} \tag{10}$$

This is obtained by solving an optimization problem that minimizes Equation **??** instead of Equation **??** .

$$\min_{\dot{\boldsymbol{\theta}}}\{\dot{\boldsymbol{\theta}}^T\dot{\boldsymbol{\theta}} + \epsilon\left(\dot{\boldsymbol{r}} - \boldsymbol{J}(\boldsymbol{\theta})\dot{\boldsymbol{\theta}}\right)^T\left(\dot{\boldsymbol{r}} - \boldsymbol{J}(\boldsymbol{\theta})\dot{\boldsymbol{\theta}}\right)\} \tag{11}$$

The index of whether the Jacobian matrix $\boldsymbol{J}(\boldsymbol{\theta})$ is approaching a singularity is the manipulability $\kappa(\boldsymbol{\theta})$[5] is used (Equation **??**).

$$\boldsymbol{r} = \boldsymbol{f}(\boldsymbol{\theta}) \tag{12}$$

Equation **??** is Equation **??** and obtain the joint angle vector.

$$\boldsymbol{\theta} = \boldsymbol{f}^{-1}(\boldsymbol{r}) \tag{13}$$

f^{-1} in is generally a nonlinear function. Therefore, by differentiating with respect to time t, the linear expression is obtained

$$\dot{\boldsymbol{r}} = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta})\dot{\boldsymbol{\theta}} \tag{14}$$
$$= \boldsymbol{J}(\boldsymbol{\theta})\dot{\boldsymbol{\theta}} \tag{15}$$

where $\boldsymbol{J}(\boldsymbol{\theta})$ is the Jacobian matrix of $m \times n$. m is the dimension of vector $\boldsymbol{r}$ and n is the dimension of vector $\boldsymbol{\theta}$. $\dot{\boldsymbol{r}}$ is the velocity/angular velocity vector.

When the Jacobian matrix is nonsingular, we can solve this linear equation using the inverse matrix $\boldsymbol{J}(\boldsymbol{\theta})^{-1}$ as follows:

$$\dot{\boldsymbol{\theta}} = \boldsymbol{J}(\boldsymbol{\theta})^{-1}\dot{\boldsymbol{r}} \tag{16}$$

However, since the Jacobian matrix is generally nonsingular, the pseudo-inverse of the Jacobian matrix $\boldsymbol{J}^{\#}(\boldsymbol{\theta})$ is used (Equation **??**).

$$\boldsymbol{A}^{\#} = \begin{cases} \boldsymbol{A}^{-1} \ (m = n = rank\boldsymbol{A}) \\ \boldsymbol{A}^T\ (\boldsymbol{A}\boldsymbol{A}^T)^{-1}(n > m = rank\boldsymbol{A}) \\ (\boldsymbol{A}^T\boldsymbol{A})^{-1}\boldsymbol{A}^T\ (m > n = rank\boldsymbol{A}) \end{cases} \tag{17}$$

[4] Inverse kinematic solutions with singularity robustness for robot Manipulator control: Y. Nakamura and H. Hanafusa, Journal of Dynamic Systems, Measurement, and Control, vol. 108, pp 163-171, 1986

[5] Robot Arm Manipulability, Tsuneo Yoshikawa, Journal of the Robotics Society of Japan, vol. 2, no. 1, pp. 63-67, 1984.

Equation **??** is Equation **??** when $m > n$, Equation **??** as a problem of finding a least-squares solution to minimize, and obtain a solution.

$$\min_{\dot{\boldsymbol{\theta}}} \left(\dot{\boldsymbol{r}} - \boldsymbol{J}(\boldsymbol{\theta})\dot{\boldsymbol{\theta}}\right)^T \left(\dot{\boldsymbol{r}} - \boldsymbol{J}(\boldsymbol{\theta})\dot{\boldsymbol{\theta}}\right) \tag{18}$$

$$\begin{aligned} \min_{\dot{\boldsymbol{\theta}}} \quad & \dot{\boldsymbol{\theta}}^T \dot{\boldsymbol{\theta}} \\ s.t. \quad & \dot{\boldsymbol{r}} = \boldsymbol{J}(\boldsymbol{\theta})\dot{\boldsymbol{\theta}} \end{aligned} \tag{19}$$

The joint angular velocity is obtained as follows:

$$\dot{\boldsymbol{\theta}} = \boldsymbol{J}^{\#}(\boldsymbol{\theta})\dot{\boldsymbol{r}} + \left(\boldsymbol{E}_n - \boldsymbol{J}^{\#}(\boldsymbol{\theta})\boldsymbol{J}(\boldsymbol{\theta})\right)\boldsymbol{z} \tag{20}$$

However, when solving according to Equation **??** , $\left|\dot{\boldsymbol{\theta}}\right|$ becomes large and unstable behavior occurs. Therefore, Nakamura et al.'s SR-Inverse[6] to avoid this singularity.

In this research, instead of the pseudo-inverse of the Jacobian matrix $\boldsymbol{J}^{\#}(\boldsymbol{\theta})$, $\boldsymbol{J} shownin Equation$ **??** $Use^*(\boldsymbol{\theta})$.

$$\boldsymbol{J}^*(\boldsymbol{\theta}) = \boldsymbol{J}^T \left(\boldsymbol{J}\boldsymbol{J}^T + \epsilon \boldsymbol{E}_m\right)^{-1} \tag{21}$$

This is obtained by solving an optimization problem that minimizes Equation **??** instead of Equation **??** .

$$\min_{\dot{\boldsymbol{\theta}}} \{\dot{\boldsymbol{\theta}}^T \dot{\boldsymbol{\theta}} + \epsilon \left(\dot{\boldsymbol{r}} - \boldsymbol{J}(\boldsymbol{\theta})\dot{\boldsymbol{\theta}}\right)^T \left(\dot{\boldsymbol{r}} - \boldsymbol{J}(\boldsymbol{\theta})\dot{\boldsymbol{\theta}}\right)\} \tag{22}$$

The index of whether the Jacobian matrix $\boldsymbol{J}(\boldsymbol{\theta})$ is approaching a singularity is the manipulability $\kappa(\boldsymbol{\theta})$[7] is used (Equation **??**).

$$\kappa(\boldsymbol{\theta}) = \sqrt{\boldsymbol{J}(\boldsymbol{\theta})\boldsymbol{J}^T(\boldsymbol{\theta})} \tag{23}$$

Task space dimension selection matrix in differential kinematic equations[8] is omitted for the sake of clarity, but it should be noted in advance that it can be applied to all formulas derived below.

18.2.2 Basic Jacobian Matrix

The Jacobian of a manipulator with a one-dimensional pair joint is the fundamental Jacobian matrix [9]. The column vector $\boldsymbol{J}_j$ of the Jacobian corresponding to the j-th joint of the underlying Jacobian matrix is represented as:

$$\boldsymbol{J}_j = \begin{cases} \begin{bmatrix} \boldsymbol{a}_j \\ \boldsymbol{0} \end{bmatrix} & \text{if linear joint} \\ \begin{bmatrix} \boldsymbol{a}_j \times (\boldsymbol{p}_{end} - \boldsymbol{p}_j) \\ \boldsymbol{a}_j \end{bmatrix} & \text{if rotational joint} \end{cases} \tag{24}$$

[6] Inverse kinematic solutions with singularity robustness for robot Manipulator control: Y.Nakamura and H. Hanafusa, Journal of Dynamic Systems, Measurement, and Control, vol. 108, pp 163-171, 1986

[7] Robot Arm Manipulability, Tsuneo Yoshikawa, Journal of the Robotics Society of Japan, vol. 2, no. 1, pp. 63-67, 1984.

[8] Hybrid Position/Force Control: A Correct Formation, William D. Fisher, M. Shahid Mujtaba, The International Journal of Robotics Research, vol. 11, no. 4, pp. 299-311, 1992.

[9] A unified approach for motion and force control of robot manipulators: The operational space formulation, O. Khatib, IEEE Journal of Robotics and Automation, vol. 3, no 1, pp. 43-53, 1987.

$\boldsymbol{a}_j$ and $\boldsymbol{p}_j$ are the joint axis unit vector and the position vector of the jth joint, respectively, and $\boldsymbol{p}_{end}$ is the Jacobian is the position vector of the end effector that controls . In the above, the 1-DOF pair revolute joints and prismatic joints were derived, but the Jacobian can also be defined as a matrix connecting these column vectors for other joints. The two-degree-of-freedom joints representing the motion of the non-omnidirectional bogie can be composed of forward and backward translational joints and rotary joints for turning. The 3-DOF joint that represents the motion of the omnidirectional cart consists of a translational 2-DOF prismatic joint and a rotary joint for turning. A ball-and-socket joint can be regarded as a combination of three rotary joints, assuming that the posture is the posture matrix and the posture change is the equivalent angular axis transformation.

18.2.3 Inverse Kinematics Including Joint Angle Limit Avoidance

In the trajectory generation of the robot manipulator, it is important to consider the joint angle limit in the real machine experiment with the robot. In this section, Inverse kinematics, including avoidance of joint angle limits, is explained by citing formulas and sentences of the literature[10] [11]

We define the weighted norm as follows:

$$\left|\dot{\boldsymbol{\theta}}\right|_{\boldsymbol{W}} = \sqrt{\dot{\boldsymbol{\theta}}^T \boldsymbol{W} \dot{\boldsymbol{\theta}}} \tag{25}$$

where $\boldsymbol{W}$ is $\boldsymbol{W} \in \boldsymbol{R}^{n\times n}$, the weighting coefficient matrix with all elements positive in the object. Using this $\boldsymbol{W}$, $\boldsymbol{J}_{\boldsymbol{W}}, \dot{\boldsymbol{\theta}}_{\boldsymbol{W}}$ is defined in

$$\boldsymbol{J}_{\boldsymbol{W}} = \boldsymbol{J}\boldsymbol{W}^{-\frac{1}{2}}, \dot{\boldsymbol{\theta}}_{\boldsymbol{W}} = \boldsymbol{W}^{\frac{1}{2}}\dot{\boldsymbol{\theta}} \tag{26}$$

Using this $\boldsymbol{J}_{\boldsymbol{W}}, \dot{\boldsymbol{\theta}}_{\boldsymbol{W}}$, we obtain the following equation.

$$\dot{\boldsymbol{r}} = \boldsymbol{J}_{\boldsymbol{W}}\dot{\boldsymbol{\theta}}_{\boldsymbol{W}} \tag{27}$$

$$\left|\dot{\boldsymbol{\theta}}\right|_{\boldsymbol{W}} = \sqrt{\dot{\boldsymbol{\theta}}_{\boldsymbol{W}}^T \dot{\boldsymbol{\theta}}_{\boldsymbol{W}}} \tag{28}$$

With this, the solution of the linear equation can be written from ?? as follows:

$$\dot{\boldsymbol{\theta}}_{\boldsymbol{W}} = \boldsymbol{W}^{-1}\boldsymbol{J}^T\left(\boldsymbol{J}\boldsymbol{W}^{-1}\boldsymbol{J}^T\right)^{-1}\dot{\boldsymbol{r}} \tag{29}$$

Also, a function $H(\boldsymbol{\theta})$ to evaluate how much margin the current joint angle θ has relative to the joint angle limits $\theta_{i,\max}, \theta_{i,\min}$is given by[12]).

$$H(\boldsymbol{\theta}) = \sum_{i=1}^{n} \frac{1}{4} \frac{(\theta_{i,\max} - \theta_{i,\min})^2}{(\theta_{i,\max} - \theta_i)(\theta_i - \theta_{i,\min})} \tag{30}$$

Next, consider a $n \times n$ weighting coefficient matrix $\boldsymbol{W}$ as shown in Equation **??** .

$$\boldsymbol{W} = \begin{bmatrix} w_1 & 0 & 0 & \cdots & 0 \\ 0 & w_2 & 0 & \cdots & 0 \\ \cdots & \cdots & \cdots & \ddots & \cdots \\ 0 & 0 & 0 & \cdots & w_n \end{bmatrix} \tag{31}$$

[10] Exploiting Task Intervals for Whole Body Robot Control, Michael Gienger and Herbert Jansen and Christian Goeric In Proceedings of the 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'06), pp. 2484 - 2490, 2006

[11] A weighted least-norm solution based scheme for avoiding jointlimits for redundant joint manipulators, Tan Fung Chan and Dubey R.V., Robotics and Automation, IEEE Transactions on, pp. 286-292,1995

[12] Efficient gradient projection optimization for manipulators withmultiple degrees of redundancy, Zghal H., Dubey R.V., Euler J.A., 1990 IEEE International Conference on Robotics and Automation, pp. 1006-1011, 1990.

where w_i is

$$w_i = 1 + \left| \frac{\partial \boldsymbol{H}(\boldsymbol{\theta})}{\partial \theta_i} \right| \tag{32}$$

Furthermore, we get the following formula from Equation ?? :

$$\frac{\partial H(\boldsymbol{\theta})}{\partial \theta_i} = \frac{(\theta_{i,\max} - \theta_{i,\min})^2 (2\theta_i - \theta_{i,\max} - \theta_{i,\min})}{4(\theta_{i,\max} - \theta_i)^2 (\theta_i - \theta_{i,\min})^2} \tag{33}$$

If the joint angle moves away from the joint angle limit, there is no need to change the weighting coefficient matrix, so redefine w_i as follows:

$$w_i = \begin{cases} 1 + \left| \frac{\partial \boldsymbol{H}(\boldsymbol{\theta})}{\partial \theta_i} \right| & if \left| \frac{\partial \boldsymbol{H}(\boldsymbol{\theta})}{\partial \theta_i} \right| \geq 0 \\ 1 & if \left| \frac{\partial \boldsymbol{H}(\boldsymbol{\theta})}{\partial \theta_i} \right| < 0 \end{cases} \tag{34}$$

By using w_i and $\boldsymbol{W}$, we can solve inverse kinematics including joint angle limit avoidance.

18.2.4 Inverse Kinematics Including Collision Avoidance

Self-collisions during robot motion and collisions with the environment model can be calculated if a geometric model exists. Efficient collision avoidance calculation proposed by Sugiura et al. [13] [14] In addition to the method of Sugiura et al., the actual implementation uses a point that allows the use of NullSpace in the task workspace to be controlled by a coefficient, and uses SR-Inverse instead of a pseudo-inverse matrix. Points that are robust to singularities have been added.

18.2.5 Joint Angular Velocity Calculation Method for Collision Avoidance

The integration of the target task and collision avoidance in the inverse kinematics calculation is performed by a blending coefficient using the shortest distance between links. As a result, when collision avoidance is not required, the target task is strictly satisfied, and when collision avoidance is required, the target task is given up to perform joint angular velocity calculations for collision avoidance. The final joint angular velocity relation is obtained by Equation ?? . In the following, the subscript of *ca* represents the component for collision avoidance calculation, and the part of *task* represents the task goal other than collision avoidance calculation.

$$\dot{\boldsymbol{\theta}} = f(d)\dot{\boldsymbol{\theta}}_{ca} + (1 - f(d))\,\dot{\boldsymbol{\theta}}_{task} \tag{35}$$

The blending factor $f(d)$ is calculated as a function of the distance between links d and the threshold values d_a and d_b. (Equation ??).

$$f(d) = \begin{cases} (d - d_a)/(d_b - d_a) & if d < d_a \\ 0 & otherwise \end{cases} \tag{36}$$

d_a is the value to start collision avoidance calculation (yellow zone??), and d_b is the threshold for collision avoidance even if it hinders the target task (orange zone??).

[13] Real-Time Self Collision Avoidance for Humanoids by means of Nullspace Criteria and Task Intervals, H. Sugiura, M. Gienger, H. Janssen, C. Goerick, Proceedings of the 2006 IEEE-RAS International Conference on Humanoid Robots, pp. 575-580, 2006.

[14] Real-time collision avoidance with whole body motion control for humanoid robots, Hisashi Sugiura, Michael Gienger, Herbert Janssen, Christian Goerick, In Proceedings of the 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'07), pp. 2053 - 2068, 2007

When the shortest distance and nearest point between two links for collision calculation can be calculated, the motion strategy for avoiding collision is derived from the virtual reaction potential acting between the two links.

Equation ?? describes the velocity calculation derived from the reaction force between two links using the vector $\boldsymbol{p}$ connecting the nearest points between two links.

$$\boldsymbol{\delta x} = \begin{cases} 0 & if\ |\boldsymbol{p}| > d_a \\ (d_a/\,|\boldsymbol{p}| - 1)\boldsymbol{p} & else \end{cases} \tag{37}$$

The joint angular velocity calculation using this is Equation ?? .

$$\dot{\boldsymbol{\theta}}_{ca} = \boldsymbol{J}_{ca}^T k_{joint} \boldsymbol{\delta x} + (\boldsymbol{E}_n - \boldsymbol{J}_{task}^* \boldsymbol{J}_{task}) \boldsymbol{J}_{ca}^T k_{null} \boldsymbol{\delta x} \tag{38}$$

k_{joint} and k_{null} are coefficients that control whether or not the reaction potential is distributed to the NullSpace of the target task.

18.2.6 Conflict Avoidance Calculation Example

The following shows an example of collision avoidance using a robot model and an environment model. In this research, the collision detection library PQP (A Proximity Query Package) was used to detect collisions between links of robots or between links and objects. [15]

In the Fig.?? $d_a = 200[mm]$,$d_b = 0.1 * d_a = 20[mm]$, We set $k_{joint} = k_{null} = 1.0$.

In this collision detection calculation, we devised to set the link as shown in the collision detection.

1. Register link list n_{ca}
2. Compute all link pairs ${}_{n_{ca}}C_2$ from the list of registered links
3. Exclude pairs of adjacent links, pairs of links that always intersect, etc.

Fig.?? In the example, four links for collision detection are registered as "forearm link", "upper arm link", "trunk link", and "base link". In this case, adjacent links are excluded from the number of link pairs of ${}_4C_2$, and all link pairs are "forearm link-trunk link", "forearm link-base link", and "upper arm link-base link".

The three lines (1 red, 2 green) in Fig.?? are the shortest distance vectors that connect the closest points between the collision shape models. Of all link pairs, the red line is the pair with the shortest distance, and inverse kinematics calculations for collision avoidance are performed from this link pair.

18.2.7 Whole-body Coordinated Motion Generation by Non-Block Diagonal Jacobian

Humanoids have a complex structure with branches, and it is necessary to perform coordinated movements with multiple manipulators. (Fig.??).

As an example of the operation of multiple manipulators, in the next section, we describe a non-block-diagonal Jacobian calculation method when there is overlap between links and a joint angular velocity calculation method using it. (Even if the former does not overlap, it can be calculated as part of the latter by the following calculation method):

[15] Fast distance queries with rectangular swept sphere volumes, Larsen E., Gottschalk S., Lin M.C., Manocha D, Proceedings of The 2000 IEEE International Conference on Robotics and Automation, pp. 3719-3726, 2000.

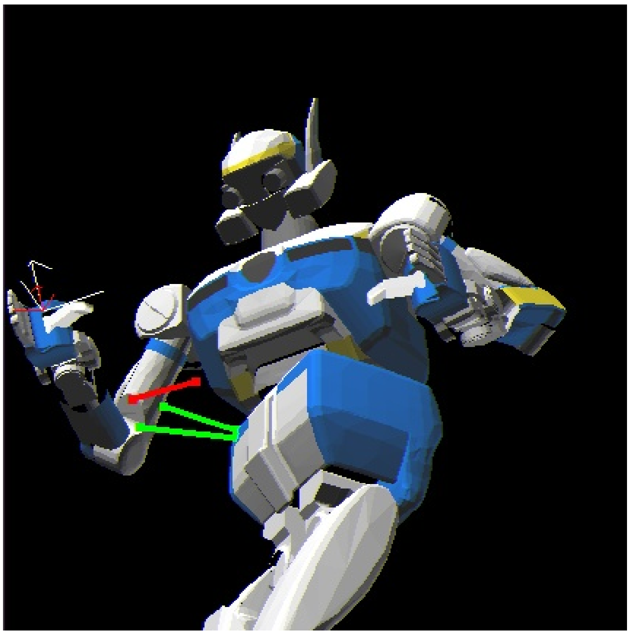


図 19: Example of Collision Avoidance

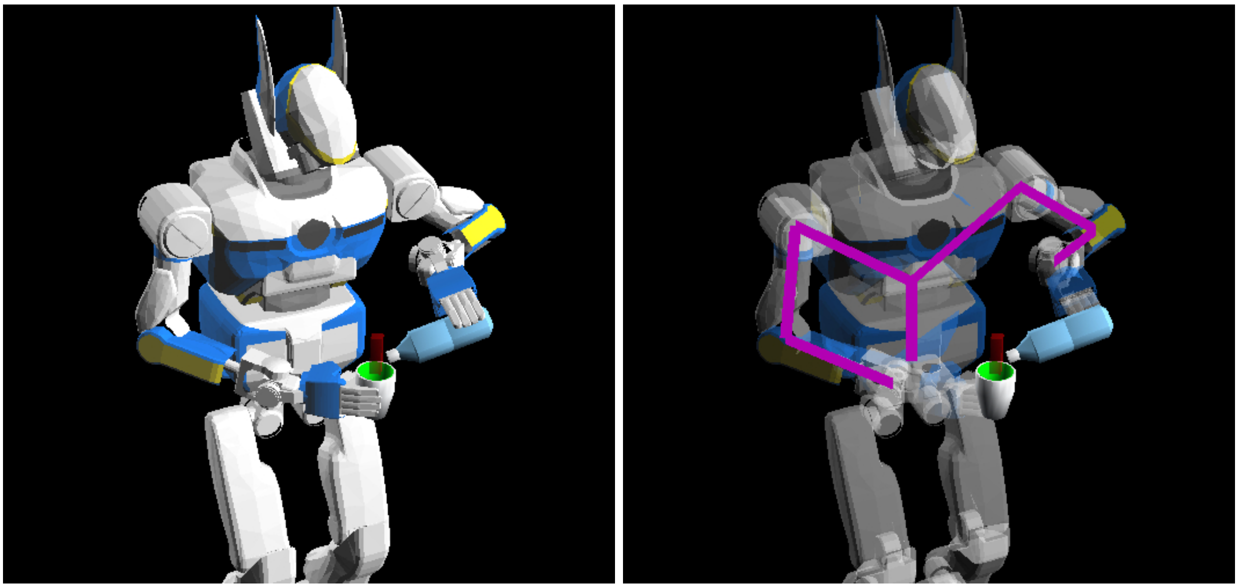


図 20: Duplicate Link Sequence

- If there is no overlap between links
  About each manipulator Calculate the joint angular velocity using the Equation **??** formula. Alternatively, joint angular velocities can be obtained using an equation that combines multiple equations (the Jacobian is a block-diagonal matrix).

- If there are duplicates between links
  If there is duplication between links, it is necessary to consider the Jacobian considering the duplication between links. For example, when performing a double-arm motion, the link sequence of the manipulator of the left arm and the link sequence of the manipulator of the right arm overlap the trunk link sequence, and it is necessary to obtain the joint angular velocity by coordinating the left

and right parts.

18.2.8 Jacobian Calculation with Inter-Link Overlap and Joint Angle Calculation

The conditions for obtaining the differential kinematic equation are shown below:

- Number of manipulators L
- Total number of joints N
- Tip velocity and angular velocity vector of manipulator $[\boldsymbol{\xi}_0^T, ..., \boldsymbol{\xi}_{L-1}^T]^T$
- Each joint angular velocity vector $[\dot{\boldsymbol{\theta}}_{\mathbf{0}}{}^T, ..., \dot{\boldsymbol{\theta}}_{\boldsymbol{L-1}}{}^T]^T$
- Indexed union of joints $S = \{0, \ldots, N-1\}$
  However, using the index set S_i of the manipulator i, S can be expressed as $S = S_0 \cup \ldots \cup S_{L-1}$.
- Joint velocity vector $[\dot{\theta}_0, ..., \dot{\theta}_{N-1}]^T$ based on S

The kinematic relation is Equation **??** .

$$\begin{bmatrix} \boldsymbol{\xi}_0 \\ \vdots \\ \boldsymbol{\xi}_{L-1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{J}_{0,0} & \ldots & \boldsymbol{J}_{0,N-1} \\ \vdots & \boldsymbol{J}_{i,j} & \vdots \\ \boldsymbol{J}_{L-1,0} & \ldots & \boldsymbol{J}_{L-1,N-1} \end{bmatrix} \begin{bmatrix} \dot{\theta}_0 \\ \vdots \\ \dot{\theta}_{N-1} \end{bmatrix} \tag{39}$$

The minor matrix $\boldsymbol{J}_{i,j}$ is obtained as follows:

$$\boldsymbol{J}_{i,j} = \begin{cases} \boldsymbol{J}_j & \text{if } j\text{-th joint} \in i\text{-th link array} \quad (40) \\ \mathbf{0} & \text{otherwise} \quad (41) \end{cases}$$

where $\boldsymbol{J}_j$ is for Equation **??** .

Joint angular velocities can be obtained using Equation **??** as well as the inverse kinematics solution for a single manipulator using SR-Inverse.

The calculation method of the non-block diagonal Jacobian here can derive the Jacobian obtained from the kinematic relational expressions that appear in the motion generation of the arm and multifingered hand. [16]

18.2.9 Whole-body Inverse Kinematics Method Using Base-Link Virtual Joints

In general, to express the motion of a robot with N joints, $N+6$ variables including the positions and orientations of the base links and the degrees of freedom of the joint angles are required. The formulation of the robot's motion using the position and orientation variables that serve as the base link is important not only for space robots[17]but also for humanoid robots [18]that are not fixed to the environment.

[16] Grasping and Manipulation by Arm and Multifingered Hand Mechanism, Kiyoshi Nagai, Tsuneo Yoshikawa, Journal of the Robotics Society of Japan, vol. 13, no. 7, pp. 994-1005, 1995.

[17] Resolved Velocity Control of Space Robot Manipulator Using Generalized Jacobian Matrix, Yoji Umetani, Kazuya Yoshida, Journal of the Robotics Society of Japan, vol. 4, no. 7, pp. 63-73, 1989.

[18] Control of Free-Floating Humanoid Robots Through Task Prioritization, Luis Sentis and Oussama Khatib, Proceedings of The 2005 IEEE International Conference on Robotics and Automation, pp. 1718-1723, 2005

Here, we consider a manipulator configuration in which a linear joint with 3 degrees of freedom and a rotational joint with 3 degrees of freedom are virtually attached to the base link of manipulators such as arms and legs (Fig.**??**). In this study, we call the above virtual 6-DOF joint a base-link virtual joint. By using the base link virtual joints, the humanoid's waist moves and the whole body joints are driven, and it is expected that the kinematics and dynamics solution space will be expanded.

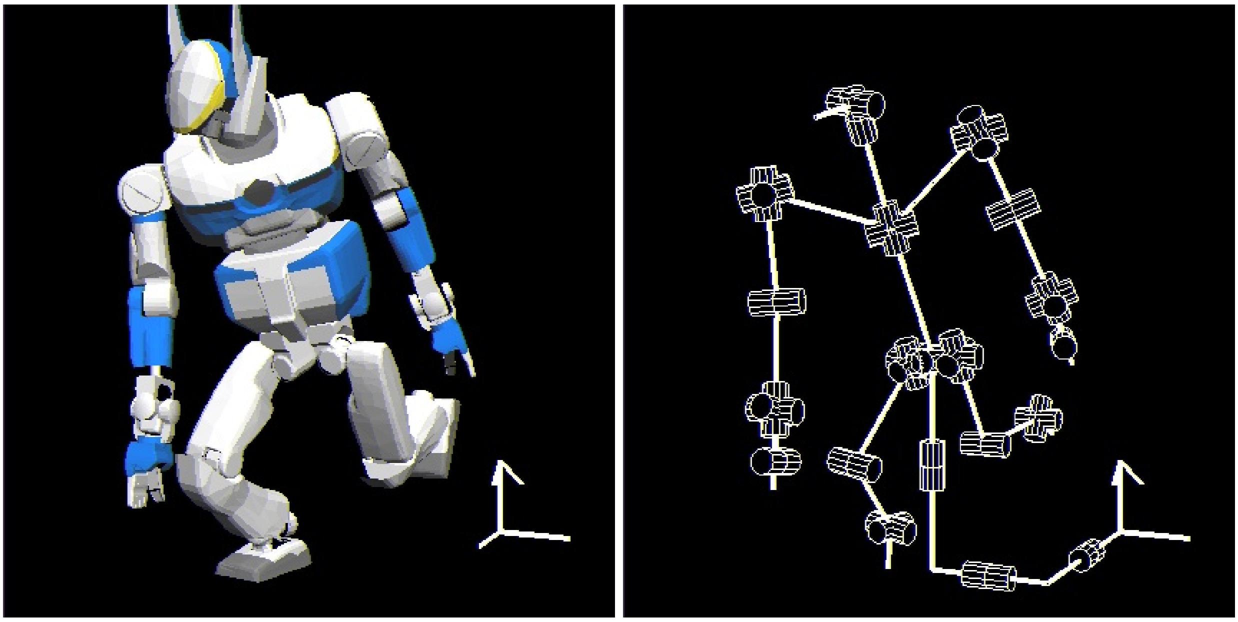


図 21: Concept of the Virtual Joint of the Base Link
(Left figure) Overview of the Robot Model
(Right figure) Skeleton Figure of Robot Model with the Virtual Joint

18.2.10 Base-Link Virtual Joint Jacobian

The Jacobian of the base-link virtual joint uses the calculation of the basic Jacobian matrix (Equation **??**), A 6×6 matrix that connects the prismatic joints around the absolute coordinate system x, y, and z axes and the rotational joints around the absolute coordinate system x, y, and z axes, respectively. By the way, the Jacobian of the root link virtual joint of the translation and rotation components can also be written as follows:

$$\boldsymbol{J}_{B,l} = \begin{bmatrix} \boldsymbol{E}_3 & -\hat{\boldsymbol{p}}_{B \to l} \\ \boldsymbol{0} & \boldsymbol{E}_3 \end{bmatrix} \tag{42}$$

$\boldsymbol{p}_{B \to l}$ is the differential vector from the base link position to the position represented by the subscript l.

18.2.11 Mass Property Calculation

Integrate multiple mass-center-of-inertia matrices into a single set of mass-center-of-inertia matrices We define an arithmetic function to compute $[m_{new}, \boldsymbol{c}_{new}, \boldsymbol{I}_{new}]$ as follows:

$$[m_{new}, \boldsymbol{c}_{new}, \boldsymbol{I}_{new}] = AddMassProperty([m_1, \boldsymbol{c}_1, \boldsymbol{I}_1], \ldots, [m_K, \boldsymbol{c}_K, \boldsymbol{I}_K]) \tag{43}$$

This is an operation such as:

$$m_{new} = \sum_{j=1}^{K} m_j \tag{44}$$

$$\boldsymbol{c}_{new} = \frac{1}{m_{new}} \sum_{j=1}^{K} m_j \boldsymbol{c}_j \tag{45}$$

$$\boldsymbol{I}_{new} = \sum_{j=1}^{K} (\boldsymbol{I}_j + m_j \boldsymbol{D}(\boldsymbol{c}_j - \boldsymbol{c}_{new})) \tag{46}$$

Here, let $\boldsymbol{D}(\boldsymbol{r}) = \hat{\boldsymbol{r}}^T \hat{\boldsymbol{r}}$.

18.2.12 Momentum/Angular Momentum Jacobian

Deriving momentum and angular momentum Jacobian for a serial link manipulator. The momentum and angular momentum around the origin are expressed by each joint variable, and the Jacobian row is calculated by the partial derivative.

Let θ_j be the motion variable of the jth joint. First, we consider a 1-DOF joint of rotation and translation.

$$\boldsymbol{P}_j = \begin{cases} \boldsymbol{a}_j \dot{\theta}_j \times (\tilde{\boldsymbol{c}}_j - \boldsymbol{p}_j)\tilde{m}_j & \text{if rotational joint} \\ \boldsymbol{a}_j \dot{\theta}_j \tilde{m}_j & \text{if linear joint} \end{cases} \tag{47}$$

$$\boldsymbol{L}_j = \begin{cases} \tilde{\boldsymbol{c}}_j \boldsymbol{P}_j + \tilde{\boldsymbol{I}}_j \boldsymbol{a}_j \dot{\theta}_j & \text{if rotational joint} \\ \boldsymbol{0} & \text{if linear joint} \end{cases} \tag{48}$$

Here, $[\tilde{m}_j, \tilde{\boldsymbol{c}}_j, \tilde{\boldsymbol{I}}_j]$ is the AddMassProperty function with the mass property of the link on the distal side of the child link of the j joint, and is actually calculated by recursive calculation[19]. Dividing these by $\dot{\theta}_j$ gives each column vector of the Jacobian.

$$\boldsymbol{m}_j = \begin{cases} \boldsymbol{a}_j \times (\tilde{\boldsymbol{c}}_j - \boldsymbol{p}_j)\tilde{m}_j & \text{if rotational joint} \\ \boldsymbol{a}_j \tilde{m}_j & \text{if linear joint} \end{cases} \tag{49}$$

$$\boldsymbol{h}_j = \begin{cases} \tilde{\boldsymbol{c}}_j \times \boldsymbol{m}_j + \tilde{\boldsymbol{I}}_j \boldsymbol{a}_j & \text{if rotational joint} \\ \boldsymbol{0} & \text{if linear joint} \end{cases} \tag{50}$$

From this, the inertia matrix can be calculated as follows:

$$\boldsymbol{M}_{\dot{\boldsymbol{\theta}}} = [\boldsymbol{m}_1, \ldots, \boldsymbol{m}_N] \tag{51}$$

$$\boldsymbol{H}_{\dot{\boldsymbol{\theta}}} = [\boldsymbol{h}_1, \ldots, \boldsymbol{h}_N] - \hat{\boldsymbol{p}}_G \boldsymbol{M}_{\dot{\boldsymbol{\theta}}} \tag{52}$$

Here, the total number of joints is set to N. The base link is considered to have prismatic joints x, y, and z axes, and rotational joints x, y, and z axes.

$$\begin{bmatrix} \boldsymbol{M}_B \\ \boldsymbol{H}_B \end{bmatrix} = \begin{bmatrix} M_r \boldsymbol{E}_3 & -M_r \hat{\boldsymbol{p}}_{B \to G} \\ \boldsymbol{0} & \tilde{\boldsymbol{I}} \end{bmatrix} \tag{53}$$

Using this, the angular momentum and momentum around the center of gravity are as follows:

$$\begin{bmatrix} \boldsymbol{P} \\ \boldsymbol{L} \end{bmatrix} = \begin{bmatrix} \boldsymbol{M}_B & \boldsymbol{M}_{\dot{\boldsymbol{\theta}}} \\ \boldsymbol{H}_B & \boldsymbol{H}_{\dot{\boldsymbol{\theta}}} \end{bmatrix} \begin{bmatrix} \boldsymbol{\xi}_B \\ \dot{\boldsymbol{\theta}} \end{bmatrix} \tag{54}$$

[19] Resolved Momentum Control:Humanoid Motion Planning based on the Linear and Angular Momentum, S.Kajita, F.Kanehiro, K.Kaneko, K.Fujiwara, K.Harada, K.Yokoi,H.Hirukawa, In Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'03), pp. 1644-1650, 2003

Here, total mass of humanoid M_r, centroid position $\boldsymbol{p}_G$, the inertia tensor $\tilde{\boldsymbol{I}}$ is obtained from the mass property calculation of all links.

$$[M_r, \boldsymbol{p}_G, \tilde{\boldsymbol{I}}] = AddMassProperty([m_1, \boldsymbol{c}_1, \boldsymbol{I}_1], \ldots, [m_N, \boldsymbol{c}_N, \boldsymbol{I}_N]) \tag{55}$$

18.2.13 Centroid Jacobian

Centroid JacobianThe center-of-mass Jacobian is the Jacobian between the center-of-mass velocity and the joint angular velocity. Since the base link virtual joint is used in this paper, it is assumed that the base link has a joint with 6 degrees of freedom. Specifically, the center-of-gravity jacobian is calculated by dividing the momentum jacobian of the base link component $\boldsymbol{M}_B$ and the component $\boldsymbol{M}'_{\dot{\boldsymbol{\theta}}}$ extracted for the used joint by the total mass.

$$\boldsymbol{J}_G = \frac{1}{M_r} \left[\begin{array}{cc} \boldsymbol{M}_B & \boldsymbol{M}'_{\dot{\boldsymbol{\theta}}} \end{array} \right] \tag{56}$$

18.3 Motion Generation Programming for Robots

18.3.1 An Example of Jacobian and Inverse Kinematics Using A Three-Axis Jointed Robot

A robot with 3-axis joints is defined, and examples of inverse kinematics and Jacobian calculations are introduced.

The definition of a robot is as follows:

```
(defclass 3dof-robot
  :super cascaded-link
  :slots (end-coords l1 l2 l3 l4 j1 j2 j3))
(defmethod 3dof-robot
  (:init ()
   (let (b)
     (send-super :init)

     (setq b (make-cube 10 10 20))
     (send b :locate #f(0 0 10))
     (send b :set-color :red)
     (setq l4 (instance bodyset-link :init (make-cascoords) :bodies (list b) :name 'l4))
     (setq end-coords (make-cascoords :pos #f(0 0 20)))
     (send l4 :assoc end-coords)
     (send l4 :locate #f(0 0 100))
     ;;
     (setq b (make-cube 10 10 100))
     (send b :locate #f(0 0 50))
     (send b :set-color :green)
     (setq l3 (instance bodyset-link :init (make-cascoords) :bodies (list b) :name 'l3))
     (send l3 :assoc l4)
     (send l3 :locate #f(0 0 100))
     ;;
     (setq b (make-cube 10 10 100))
     (send b :locate #f(0 0 50))
     (send b :set-color :blue)
     (setq l2 (instance bodyset-link :init (make-cascoords) :bodies (list b) :name 'l2))
     (send l2 :assoc l3)
     (send l2 :locate #f(0 0 20))
     ;;
     (setq b (body+ (make-cube 10 10 20 :pos #f(0 0 10)) (make-cube 300 300 2)))
     (send b :set-color :white)
     (setq l1 (instance bodyset-link :init (make-cascoords) :bodies (list b) :name 'l1))
     (send l1 :assoc l2)
     ;;
     (setq j1 (instance rotational-joint :init :name 'j1
                 :parent-link l1 :child-link l2 :axis :y :min -100 :max 100)
```

```
          j2 (instance rotational-joint :init  :name 'j2
                :parent-link l2 :child-link l3 :axis :y :min -100 :max 100)
          j3 (instance rotational-joint :init  :name 'j3
                :parent-link l3 :child-link l4 :axis :y :min -100 :max 100))
    ;;
    (setq links (list l1 l2 l3 l4))
    (setq joint-list (list j1 j2 j3))
    ;;
    (send self :init-ending)
    self))
  (:end-coords (&rest args) (forward-message-to end-coords args))
  )
```

Here, the coordinates of the robot's hands are stored in a slot variable called `end-coords`, and methods are provided to access them.

As before, it is possible to create a robot model, display it, and specify the joint angles.

```
(setq r (instance 3dof-robot :init))
(objects (list r))
(send r :angle-vector #f(30 30 30))
```

Moreover, it is possible to display the `end-coords`(end-coordinate system) of the robot.

```
(send (send r :end-coords) :draw-on :flush t)
```

But it disappears when a mouse event occurs. For permanent display, use:

```
(objects (list r (send r :end-coords)))
```

Here are some examples of Jacobian and inverse kinematics. First and foremost is the list of links obtained as . This returns a link that can be traced from the root (body) of the robot to the link that is the argument.

```
(send r :link-list (send r :end-coords :parent))
```

The `:calc-jacobian-from-link-list` method takes a list of links and can compute the Jacobian corresponding to the joints in each link. Also, the coordinate system of the end effector is specified by the `:move-target` keyword argument. Other keyword arguments are described later.

```
(dotimes (i 100)
  (setq j (send r :calc-jacobian-from-link-list
               (send r :link-list (send r :end-coords :parent))
               :move-target (send r :end-coords)
               :rotation-axis t
               :translation-axis t))
  (setq j# (sr-inverse j))
  (setq da (transform j# #f(1 0 0 0 0 0)))
  ;;(setq da (transform j# #f(0 0 0 0 -1 0)))
  (send r :angle-vector (v+ (send r :angle-vector) da))
  (send *irtviewer* :draw-objects)
  )
```

Here, the link length (number of joints) is 3, so a 6-by-3 Jacobian (`j`) is calculated. If the inverse matrix (`j#`) is created and the target velocity and angular velocity (`#f(1 0 0 0 0 0)`) of the 6 degrees of freedom in position and posture are given, the corresponding joint velocity (`da`) can be calculated, which is added to the current joint angle (`(v+ (send r :angle-vector) da)`).

Next, we show an example in which the position of the robot's endpoint work is adjusted, but the posture is not constrained and can be left as it is. Here, there are `:rotation-axis` and `:translation-axis` as optional arguments of `:calc-jacobian-from-link-list`, which indicate constraint conditions for position and orientation, respectively. `t` is triaxial constraint, `nil` is no constraint, and `:x`, `:y`, `:z` can be specified.

```
(setq translation-axis t)
(setq rotation-axis nil)
(dotimes (i 2000)
  (setq j (send r :calc-jacobian-from-link-list
               (send r :link-list (send r :end-coords :parent))
               :move-target (send r :end-coords)
               :rotation-axis rotation-axis
               :translation-axis translation-axis))
  (setq j# (sr-inverse j))
  (setq c (make-cascoords :pos (float-vector (* 100 (sin (/ i 500.0))) 0 200)))
  (setq dif-pos (send (send r :end-coords) :difference-position c))
  (setq da (transform j# dif-pos))
  (send r :angle-vector (v+ (send r :angle-vector) da))
  (send *irtviewer* :draw-objects :flush nil)
  (send c :draw-on :flush t)
  )
```

Here, the 3-by-3 Jacobian is calculated by constraining only the three axes of position, and the velocity is given to the joints of the robot from the inverse matrix of this. Furthermore, here the target coordinates are displayed and flash processing is performed.

```
(send *irtviewer* :draw-objects :flush nil)
```

The screen is drawn on `*irtviewer*` as

```
(send c :draw-on :flush t)
```

`:inverse-kinematics` is an inverse kinematics method that summarizes the above computations. Specify the target coordinate system in the first argument, and specify `:move-target`, `:translation-axis`, `:rotation-axis` . Also, if `:debug-view` keyword argument is given with `t`, the state of the calculation is presented visually as well as text.

```
(setq c (make-cascoords :pos #f(100 0 0) :rpy (float-vector 0 pi 0)))
(send r :inverse-kinematics  c
      :link-list (send r :link-list (send r :end-coords :parent))
      :move-target (send r :end-coords)
      :translation-axis t
      :rotation-axis t
      :debug-view t)
```

Let's look at the following program as an example of when reverse kinematics fails.

```
(dotimes (i 400)
  (setq c (make-cascoords
           :pos (float-vector (+ 100 (* 80 (sin (/ i 100.0)))) 0 0)
           :rpy (float-vector 0 pi 0)))
  (send r :inverse-kinematics  c
        :link-list (send r :link-list (send r :end-coords :parent))
        :move-target (send r :end-coords) :translation-axis t  :rotation-axis t)
  (x::window-main-one)
  (send *irtviewer* :draw-objects :flush nil)
  (send c :draw-on :flush t)
  )
```

When I run this program, I get the following error:

```
;; inverse-kinematics failed.
;; dif-pos : #f(11.7826 0.0 0.008449)/(11.7826/1)
;; dif-rot : #f(0.0 2.686130e-05 0.0)/(2.686130e-05/0.017453)
;;  coords : #<coordinates #X4bcccb0  0.0 0.0 0.0 / 0.0 0.0 0.0>
;;  angles : (14.9993 150 15.0006)
;;    args : ((#<cascaded-coords #X4b668a0  39.982 0.0 0.0 / 3.142 1.225e-16 3.14
2>) :link-list (#<bodyset-link #X4cf8e60 l2  0.0 0.0 20.0 / 0.0 0.262 0.0> #<body
set-link #X4cc8008 l3  25.866 0.0 116.597 / 3.142 0.262 3.142> #<bodyset-link #X4
c7a0d0 l4  51.764 0.0 20.009 / 3.142 2.686e-05 3.142>) :move-target;; #<cascaded-
coords #X4c93640  51.764 0.0 0.009 / 3.142 2.686e-05 3.142> :translation-axis t :
rotation-axis t)
```

This is a situation in which the hand cannot reach the target position due to the limitation of the joint drive range. In such a situation, for example, `:rotation-axis nil` can be specified if the position of the hand can be ignored as long as it reaches the target position.

Also, `:thre` and `:rthre` can be used to specify the position and orientation error, which is the termination condition of the inverse kinematics calculation. In situations where exact calculations are not required, it is a good idea to use a value larger than the default `1`, `(deg2rad 1)`.

Also, if the inverse kinematics calculation fails, by default it will return to the posture before starting the inverse kinematics calculation, but if you specify nil for the keyword argument `:revert-if-fail` After repeating the calculation of the number of times, it exits the function with that posture. The specified number of times can also be specified with the keyword argument `:stop`.

```
(setq c (make-cascoords :pos #f(300 0 0) :rpy (float-vector 0 pi 0)))
(send r :inverse-kinematics  c
      :link-list (send r :link-list (send r :end-coords :parent))
      :move-target (send r :end-coords)
      :translation-axis t
      :rotation-axis nil
      :revert-if-fail nil)
```

18.3.2 Example In The irteus Sample Program

The cascaded-coords class provides a method called:

- (:link-list (to &optional form))
- (:calc-jacobian-from-link-list (link-list &key move-target (rotation-axis nil)))

The former takes a link as an argument, calculates the path from the root link to this link, and returns it as a list of links. The latter takes this list of links as an argument and computes the Jacobian with respect to the move-target coordinate system.

concatenate result-type a b concatenates a b and converts it back to result-type type, scale a b multiplies all elements of vector b by a scalar a, and matrix-log computes the matrix logarithm function.

```
(if (not (boundp '*irtviewer*)) (make-irtviewer))

(load "irteus/demo/sample-arm-model.l")
(setq *sarm* (instance sarmclass :init))
(send *sarm* :reset-pose)
(setq *target* (make-coords :pos #f(350 200 400)))
(objects (list *sarm* *target*))

(do-until-key
  ;; step 3
  (setq c (send *sarm* :end-coords))
  (send c :draw-on :flush t)
  ;; step 4
  ;; step 4
  (setq dp (scale 0.001 (v- (send *target* :worldpos) (send c :worldpos))) ;; mm->m
        dw (matrix-log (m* (transpose (send c :worldrot)) (send *target* :worldrot))))
  (format t "dp = ~7,3f ~7,3f ~7,3f, dw = ~7,3f ~7,3f ~7,3f~%"
          (elt dp 0) (elt dp 1) (elt dp 2)
          (elt dw 0) (elt dw 1) (elt dw 2))
  ;; step 5
  (when (< (+ (norm dp) (norm dw)) 0.01) (return))
  ;; step 6
  (setq ll (send *sarm* :link-list (send *sarm* :end-coords :parent)))
  (setq j (send *sarm* :calc-jacobian-from-link-list
                ll :move-target (send *sarm* :end-coords)
                :trnaslation-axis t :rotation-axis t))
  (setq q (scale 1.0 (transform (pseudo-inverse j) (concatenate float-vector dp dw))))
  ;; step 7
  (dotimes (i (length ll))
    (send (send (elt ll i) :joint) :joint-angle (elt q i) :relative t))
```

```
;; draw
(send *irtviewer* :draw-objects)
(x::window-main-one))
```

In actual programming, a method called :inverse-kinematics is prepared, and functions such as singularity and joint limit avoidance and self-collision avoidance are added here.

18.3.3 Real Robot Model

Let's take a look at a practical sample program that uses an actual robot and environment.

First, read the model files of the robot and the environment. These files are stored in $EUSDIR/models, and a program that loads these files and creates instances can be written as follows. `(room73b2)` and `(h7)` are functions defined in these files. The robot model (`robot-model`) is defined in the `irtrobot.l` file and is a child class of the `cascaded-link` class. A robot is defined as a tree of `larm,rarm,lleg,rleg,head` links, and `(send *robot* :larm)` or `(send *robot* :head)` can be used to access the limb of the robot, enabling usage such as inverse kinematics for the right hand and inverse kinematics for the left hand.

```
(load "models/room73b2-scene.l")
(load "models/h7-robot.l")
(setq *room* (room73b2))
(setq *robot* (h7))
(objects (list *robot* *room*))
```

The robot has a method called `:reset-pose` that can be used to take the initial pose.

```
(send *robot* :reset-pose)
```

Next, we want to move the robot around the room. Typical coordinates in the room can be obtained with `(send *room* :spots)`. To obtain the desired coordinates from among these, call the `:spot` method with the name of the coordinates as an argument. By the way, the definition of this method is in `prog/jskeus/irteus/irtscene.l`.

```
(defmethod scene-model
  (:spots
   (&optional name)
   (append
    (mapcan
     #'(lambda(x)(if (derivedp x scene-model) (send x :spots name) nil))
     objs)
    (mapcan #'(lambda (o)
(if (and (eq (class o) cascaded-coords)
 (or (null name) (string= name (send o :name))))
    (list o)))
    objs)))
  (:spot
   (name)
   (let ((r (send self :spots name)))
     (case (length r)
       (0 (warning-message 1 "could not found spot(~A)" name) nil)
       (1 (car r))
       (t (warning-message 1 "found multiple spot ~A for given name(~A)" r name) (car r)))))
  )
```

Since the robot is also a child class of the `coordinates` class, it can use the `:move-to` method. Also, since the origin of this robot is at the waist, move using the `:locate` method so that the feet touch the ground.

```
(send *robot* :move-to (send *room* :spot "cook-spot") :world)
(send *robot* :locate #f(0 0 550))
```

Currently, the robot is small on the screen of `*irtviewer*`, so use the following method to adjust the robot so that it fills the screen.

```
(send *irtviewer* :look-all
      (geo::make-bounding-box
       (flatten (send-all (send *robot* :bodies) :vertices))))
```

Next, an object in the environment is selected. Here we use the `:object` method. This behaves like `:spots, :spot`, so you can find out what objects are by `(send-all (send *room* :objects) :name)`. In addition to `room73b2-kettle`, use `room73b2-mug-cup` or `room73b2-knife`.

```
(setq *kettle* (send *room* :object "room73b2-kettle"))
```

Immediately after the environment model is initialized, the objects are assoc'd in the room, so the parent-child relationship is resolved as follows. If this is not done, there will be problems when grasping objects, etc.

```
(if (send *kettle* :parent) (send (send *kettle* :parent) :dissoc *kettle*))
```

There are the following methods for directing the robot's line of sight to the target object.

```
(send *robot* :head :look-at (send *kettle* :worldpos))
```

For the target object, the coordinate system that should be used to grasp the object may be described as a `:handle` method. Since this method returns a list, the coordinate system can be known as `(car (send *kettle* :han` as follows. `(send (car (send *kettle* :handle)) :draw-on :flush t)`
Therefore, in order to reach this object, it becomes

```
(send *robot* :larm :inverse-kinematics
      (car (send *kettle* :handle))
      :link-list (send *robot* :link-list (send *robot* :larm :end-coords :parent))
      :move-target (send *robot* :larm :end-coords)
      :rotation-axis :z
      :debug-view t)
```

Here, we connect the coordinate systems of the robot's hand and the target object,

```
(send *robot* :larm :end-coords :assoc *kettle*)
```

It can be lifted by 100[mm] in the world coordinate system as follows.

```
(send *robot* :larm :move-end-pos #f(0 0 100) :world
        :debug-view t :look-at-target t)
```

`:look-at-target` is a command to keep looking at the target while moving.

18.3.4 Example of Specifying A Function for target-coords of inverse-kinematics

The argument target-coords of :inverse-kinematics can specify a function that returns a `coordinates` class other than the `coordinates` class. The program shown below uses two arms to shake a cocktail (Fig.**??**).

```
(load "irteus/demo/sample-robot-model.l")
(setq *robot* (instance sample-robot :init))
(setq *obj* (make-cylinder 20 100))
(send *obj* :set-color #f(1 1 0))
(send *robot* :reset-pose)
(objects (list *robot* *obj*))

(send *robot* :inverse-kinematics
      (list (make-coords :pos #f(400 0 0)))
      :move-target
      (list (send *robot* :larm :end-coords))
      :link-list
      (list (send *robot* :link-list
                  (send (send *robot* :larm :end-coords) :parent)
```

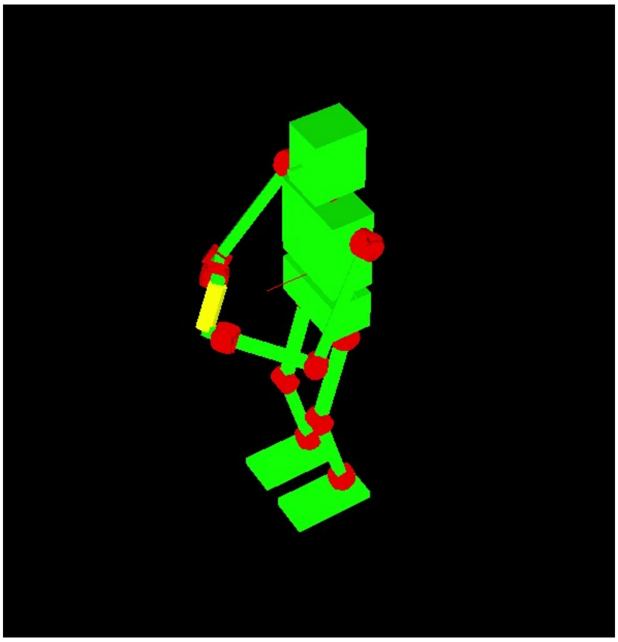


図 22: Example of Dual Arm InverseKinematics

```
                  (car (send *robot* :larm :links))))
      :translation-axis (list t)
      :rotation-axis (list nil))

(let* ((cnt 0.0))
  (do-until-key
   (incf cnt 0.1)
   (send *robot* :inverse-kinematics
         (list (make-coords :pos (float-vector (+ 400 (* 100 (sin cnt))) (* 50 (cos cnt)) 0))
               #'(lambda ()
                   (send (send (send *robot* :larm :end-coords) :copy-worldcoords)
                         :translate #f(0 0 100) :local)))
         :move-target
         (list (send *robot* :larm :end-coords)
               (send *robot* :rarm :end-coords))
         :link-list
         (list (send *robot* :link-list
                     (send (send *robot* :larm :end-coords) :parent)
                     (car (send *robot* :larm :links)))
               (send *robot* :link-list
                     (send (send *robot* :rarm :end-coords) :parent)
                     (car (send *robot* :rarm :links))))
         :translation-axis (list :z t)
         :rotation-axis (list nil :z))
   (send *obj* :newcoords (send (send *robot* :larm :end-coords) :copy-worldcoords))
   (send *irtviewer* :draw-objects)))

         (list (make-coords :pos (float-vector (+ 400 (* 100 (sin cnt))) (* 50 (cos cnt)) 0))
               #'(lambda ()
                   (send (send (send *robot* :larm :end-coords) :copy-worldcoords)
                         :translate #f(0 0 100) :local)))
```

The line actually specifies a function for target-coords. In this example, first determine the position of the left hand holding the cocktail. At this time, `:translation-axis :z`, `:rotation-axis nil`, so the amount of translation in the z-direction and the direction of rotation are not taken into account in the computation of the inverse kinematics of the left hand. Then, by evaluating the function for the determined left hand position, the right hand position is determined for the position 100 in the z direction as seen from the local coordinates of the hand. At this time, the constraint conditions for the right hand are `:translation-axis t`,

`:rotation-axis :z`, which means that inverse kinematics is solved in the z direction, i.e., with the cocktail's length direction as its axis and rotation about that axis allowed. In this way, it is necessary to treat target-coords as a function when solving inverse kinematics based on constraints.

18.3.5 Example of fullbody-inverse-kinematics Considering Center of Gravity Position

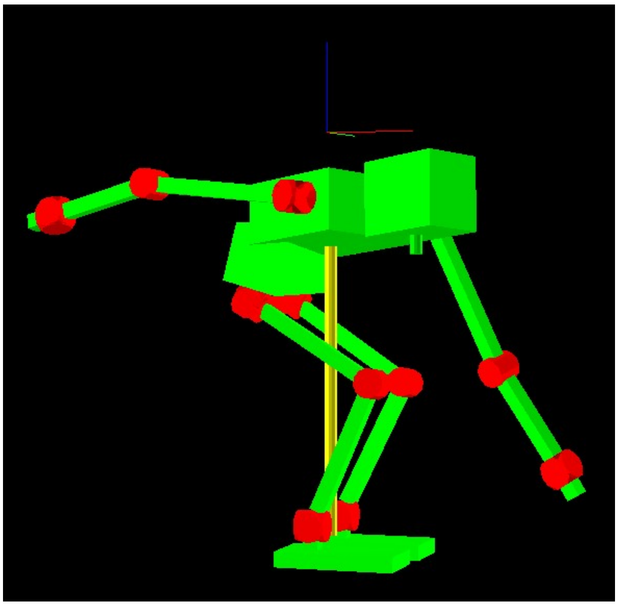


図 23: Example of InverseKinematics with root link virtual joint

:fullbody-inverse-kinematics is a function that solves the inverse kinematics driven base-link virtual joints in addition to the robot's joints. In the program shown below, both feet are fixed on the ground, the center of gravity is positioned above both feet, and the left hand moves to reach the target.

```
(load "irteus/demo/sample-robot-model.l")
(setq *robot* (instance sample-robot :init))
(send *robot* :reset-pose)
(setq *obj* (make-cylinder 10 600))
(send *obj* :rotate pi :x)
(send *obj* :set-color #f(1 1 0))
(objects (list *robot* *obj*))

(let* ((rleg-coords (send *robot* :rleg :end-coords :copy-worldcoords))
       (lleg-coords (send *robot* :lleg :end-coords :copy-worldcoords)))
  (send *robot* :torso :waist-p :joint-angle 10)
  (send *robot* :fullbody-inverse-kinematics
        (list rleg-coords
              lleg-coords
              (make-coords :pos (float-vector 400 100 -600)))
        :move-target
        (list (send *robot* :rleg :end-coords)
              (send *robot* :lleg :end-coords)
              (send *robot* :larm :end-coords))
        :link-list
        (list (send *robot* :link-list (send *robot* :rleg :end-coords :parent))
              (send *robot* :link-list (send *robot* :lleg :end-coords :parent))
              (send *robot* :link-list (send *robot* :larm :end-coords :parent)))
        :translation-axis (list t t t)
        :rotation-axis (list t t nil)
        :target-centroid-pos (midpoint 0.5
                                       (send *robot* :rleg :end-coords :worldpos)
                                       (send *robot* :lleg :end-coords :worldpos))
        :cog-translation-axis :z)
```

```
  (send *obj* :locate (send *robot* :centroid) :world)
  (send *irtviewer* :draw-objects))

(list rleg-coords
      lleg-coords
      (make-coords :pos (float-vector 400 100 -600)))
```

line specifies the target positions and postures of the right foot, left foot, and left hand in target-coords. Since the right and left legs do not move, a copy of the current coordinates is given. At this time `:translation-axis (list t t t)`, `:rotation-axis (list t t nil)`, the position and posture of the right and left legs are completely constrained, and inverse kinematics is solved under the condition that the posture of the left hand is allowed to rotate.

```
:target-centroid-pos (midpoint 0.5 (send *robot* :rleg :end-coords :worldpos)
                                   (send *robot* :lleg :end-coords :worldpos))
:cog-translation-axis :z)
```

The line specifies the inverse kinematics of the centroid. By using `:cog-translation-axis :z` to allow movement of the center of gravity in the z-direction, and by using `:target-centroid-pos` to give the coordinate between the two feet as the target position of the center of gravity, Inverse kinematics can be solved under the condition that the xy-coordinates of the center of gravity coincide with the middle of both feet. These arguments are default values and can be omitted.

18.3.6 Example of Solving fullbody-inverse-kinematics Considering External Forces

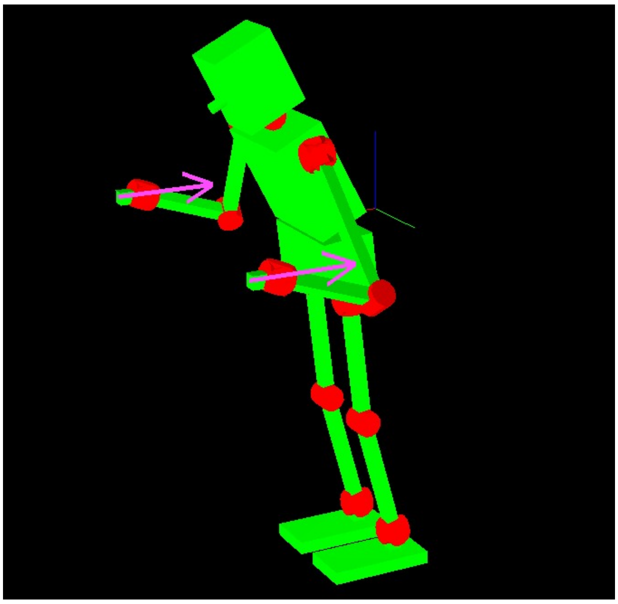


図 24: Example of InverseKinematics with external force

When the robot receives an external force or moment, it can be balanced by offsetting the center of gravity of the robot so as to balance the moment around the sole due to the external force. The program shown below uses inverse kinematics to find a posture in which both hands and feet reach their target positions and are balanced when external forces and moments are applied to both hands.

```
(load "irteus/demo/sample-robot-model.l")
(setq *robot* (instance sample-robot :init))
```

```
(send *robot* :reset-pose)
(setq *obj* (make-cylinder 10 600))
(objects (list *robot*))

(let* ((force-list '(#f(-20 0 0) #f(-20 0 0)))
       (moment-list '(#f(10 0 0) #f(10 0 0))))

  (send *robot* :fullbody-inverse-kinematics
        (list (send *robot* :rleg :end-coords :copy-worldcoords)
              (send *robot* :lleg :end-coords :copy-worldcoords)
              (make-coords :pos #f(400 -300 0))
              (make-coords :pos #f(400 300 0)))
        :move-target (mapcar #'(lambda (x)
                                 (send *robot* x :end-coords))
                             (list :rleg :lleg :rarm :larm))
        :link-list (mapcar #'(lambda (x)
                               (send *robot* :link-list (send *robot* x :end-coords :parent)))
                           (list :rleg :lleg :rarm :larm))
        :centroid-offset-func #'(lambda () (send *robot* :calc-static-balance-point
                                            :force-list force-list
                                            :moment-list moment-list))
        :target-centroid-pos (midpoint 0.5 (send *robot* :rleg :end-coords :worldpos)
                                           (send *robot* :lleg :end-coords :worldpos))
        :cog-translation-axis :z)
  (send *irtviewer* :draw-objects)

  ;; draw force
  (mapcar
   #'(lambda (f cc)
       (let* ((prev-color (send *viewer* :viewsurface :color))
              (prev-width (send *viewer* :viewsurface :line-width)))
         (send *viewer* :viewsurface :color #F(1 0.3 1))
         (send *viewer* :viewsurface :line-width 5)
         (send *irtviewer* :viewer :draw-arrow
               (send cc :worldpos)
               (v+ (send cc :worldpos) (scale 10 f)))
         (send *viewer* :viewsurface :color prev-color)
         (send *viewer* :viewsurface :line-width prev-width)))
   force-list
   (list (send *robot* :rarm :end-coords)
         (send *robot* :larm :end-coords)))
  (send *irtviewer* :viewer :viewsurface :flush)
  )
```

この例では、

```
:centroid-offset-func #'(lambda () (send *robot* :calc-static-balance-point
                                    :force-list force-list
                                    :moment-list moment-list))
```

In this example, external force and external moment are taken into account in the row. force-list is a list of external forces acting on the right hand and left hand, force-list is a list of external moments acting on the right hand and external moments acting on the left hand, and the units are [N] and [Nm], respectively. :calc-static-balance-point is a function that returns the position of the sole pressure center that balances the external moment acting on the current positions of both hands and the gravity acting on the current position of the center of gravity. :centroid-offset-func can specify a function that returns a `float-vector` class, and uses the return value of this function instead of the current position of the center of gravity to reduce the distance to the target position of the center of gravity. By using `:cog-translation-axis :z` to allow movement of the center of gravity in the z-direction, and by specifying the coordinates in the middle of both feet as the target center of gravity position with `:target-centroid-pos`: Inverse kinematics can be solved to match the return value of centroid-offset-func, i.e. the xy coordinates of the center of the sole pressure that balances the external force, to the middle of both feet.

18.4 Robot Model

The body of the robot consists of links and joints, and the `bodyset-link` and `joint` classes are used to create a model picture. The robot's body generates a model as a connected link called `cascaded-link` that includes these elements.

`joint` is actually an abstract class Choose `rotational-joint`,`linear-joint`, `wheel-joint`,`omniwheel-joint`, `sphere-joint`, and for robots with limbs use the `robot-model` class instead of `cascaded-link`.

joint [Class]

:super **propertied-object**

:slots parent-link child-link joint-angle min-angle max-angle default-coords joint-velocity joi

:init [method]

&key (name (intern (format nil joint A (system:address self)) KEYWORD))
((:child-link clink))
((:parent-link plink))
(min -90)
(max 90)
((:max-joint-velocity mjv))
((:max-joint-torque mjt))
((:joint-min-max-table mm-table))
((:joint-min-max-target mm-target))
&allow-other-keys

abstract class of joint, users need to use rotational-joint, linear-joint, sphere-joint, 6dof-joint, wheel-joint or omniwheel-joint.
use :parent-link/:child-link for specifying links that this joint connect to and :min/:min for range of joint angle in degree.

:min-angle *&optional v* [method]

If v is set, it updates min-angle of this instance. :min-angle returns minimal angle of this joint in degree.

:max-angle *&optional v* [method]

If v is set, it updates max-angle of this instance. :max-angle returns maximum angle of this joint in degree.

:parent-link *&rest args* [method]

Returns parent link of this joint. if any arguments is set, it is passed to the parent-link.

:child-link *&rest args* [method]

Returns child link of this joint. if any arguments is set, it is passed to the child-link.

:joint-dof [method]

Returns Degree of Freedom of this joint.

:speed-to-angle *&rest args* [method]

Returns values in deg/mm unit of input value in SI(rad/m) unit.

:angle-to-speed *&rest args* [method]

Returns values in SI(rad/m) unit of input value in deg/mm unit.

:joint-velocity *&optional jv* [method]

If jv is set, it updates joint-velocity of this instance. :joint-velocity returns velocity of this joint in SI(m/s, rad/s) unit.

:joint-acceleration *&optional ja* [method]

If ja is set, it updates joint-acceleration of this instance. :joint-acceleration returns acceleration of this joint in SI(m/s^2, rad/s^2) unit.

:joint-torque *&optional jt* [method]

If jt is set, it updates joint-torque of this instance. :joint-torque returns torque of this joint in SI(N, Nm) unit.

:max-joint-velocity *&optional mjv* [method]

If mjv is set, it updates min-joint-velocity of this instance. :min-joint-velocity returns velocity of this joint in SI(m/s, rad/s) unit.

:max-joint-torque *&optional mjt* [method]

If mjt is set, it updates min-joint-torque of this instance. :min-joint-torque returns velocity of this joint in SI(N, Nm) unit.

:calc-dav-gain *dav i periodic-time* [method]

:calc-jacobian *&rest args* [method]

:joint-min-max-table *&optional mm-table* [method]

:joint-min-max-target *&optional mm-target* [method]

:joint-min-max-table-angle-interpolate *target-angle min-or-max* [method]

:joint-min-max-table-min-angle *&optional (target-angle (send joint-min-max-target :joint-angle))* [method]

:joint-min-max-table-max-angle *&optional (target-angle (send joint-min-max-target :joint-angle))* [method]

:max-joint-torque *&optional mjt* [method]

If mjt is set, it updates min-joint-torque of this instance. :min-joint-torque returns velocity of this joint in SI(N, Nm) unit.

:max-joint-velocity *&optional mjv* [method]

If mjv is set, it updates min-joint-velocity of this instance. :min-joint-velocity returns velocity of this joint in SI(m/s, rad/s) unit.

:joint-torque *&optional jt* [method]

If jt is set, it updates joint-torque of this instance. :joint-torque returns torque of this joint in SI(N, Nm) unit.

:joint-acceleration *&optional ja* [method]

If ja is set, it updates joint-acceleration of this instance. :joint-acceleration returns acceleration of this joint in SI(m/s^2, rad/s^2) unit.

:joint-velocity *&optional jv* [method]

If jv is set, it updates joint-velocity of this instance. :joint-velocity returns velocity of this joint in SI(m/s, rad/s) unit.

:angle-to-speed *&rest args* [method]

Returns values in SI(rad/m) unit of input value in deg/mm unit.

:speed-to-angle *&rest args* [method]

Returns values in deg/mm unit of input value in SI(rad/m) unit.

:joint-dof [method]

Returns Degree of Freedom of this joint.

:child-link *&rest args* [method]

Returns child link of this joint. if any arguments is set, it is passed to the child-link.

:parent-link *&rest args* [method]

Returns parent link of this joint. if any arguments is set, it is passed to the parent-link.

:max-angle *&optional v* [method]

If v is set, it updates max-angle of this instance. :max-angle returns maximum angle of this joint in degree.

:min-angle *&optional v* [method]

If v is set, it updates min-angle of this instance. :min-angle returns minimal angle of this joint in degree.

:init [method]

&key (name (intern (format nil joint A (system:address self)) KEYWORD))
((:child-link clink))
((:parent-link plink))
(min -90)
(max 90)
((:max-joint-velocity mjv))
((:max-joint-torque mjt))
((:joint-min-max-table mm-table))
((:joint-min-max-target mm-target))
&allow-other-keys

abstract class of joint, users need to use rotational-joint, linear-joint, sphere-joint, 6dof-joint, wheel-joint or omniwheel-joint.
use :parent-link/:child-link for specifying links that this joint connect to and :min/:min for range of joint angle in degree.

:joint-min-max-table-max-angle *&optional (target-angle (send joint-min-max-target :joint-angle))* [method]

:joint-min-max-table-min-angle *&optional (target-angle (send joint-min-max-target :joint-angle))* [method]

:joint-min-max-table-angle-interpolate *target-angle min-or-max* [method]

:joint-min-max-target *&optional mm-target* [method]

:joint-min-max-table *&optional mm-table* [method]

:calc-jacobian *&rest args* [method]

:calc-dav-gain *dav i periodic-time* [method]

rotational-joint [Class]

:super **joint**
:slots axis

:init [method]

&rest args &key ((:axis ax) :z)
((:max-joint-velocity mjv) 5)
((:max-joint-torque mjt) 100)
&allow-other-keys

create instance of rotational-joint. :axis is either (:x, :y, :z) or vector. :min-angle and :max-angle takes in radius, but velocity and torque are given in SI units.

:joint-angle [method]

&optional v &key relative
&allow-other-keys

Return joint-angle if v is not set, if v is given, set joint angle. v is rotational value in degree.

:joint-dof [method]

Returns DOF of rotational joint, 1.

:speed-to-angle *v* [method]

Returns degree of given input in radian

:angle-to-speed *v* [method]

Returns radian of given input in degree

:calc-angle-speed-gain *dav i periodic-time* [method]

:calc-jacobian *&rest args* [method]

:angle-to-speed *v* [method]

Returns radian of given input in degree

:speed-to-angle *v* [method]

Returns degree of given input in radian

:joint-dof [method]

Returns DOF of rotational joint, 1.

:joint-angle [method]

&optional v &key relative
&allow-other-keys

Return joint-angle if v is not set, if v is given, set joint angle. v is rotational value in degree.

:init [method]

&rest args &key ((:axis ax) :z)
((:max-joint-velocity mjv) 5)
((:max-joint-torque mjt) 100)
&allow-other-keys

create instance of rotational-joint. :axis is either (:x, :y, :z) or vector. :min-angle and :max-angle takes in radius, but velocity and torque are given in SI units.

:calc-jacobian *&rest args* [method]

:calc-angle-speed-gain *dav i periodic-time* [method]

linear-joint [Class]

:super	**joint**
:slots	axis

:init [method]

&rest args &key ((:axis ax) :z)
((:max-joint-velocity mjv) (/ pi 4))
((:max-joint-torque mjt) 100)
&allow-other-keys

Create instance of linear-joint. :axis is either (:x, :y, :z) or vector. :min-angle and :max-angle takes in [mm], but velocity and torque are given in SI units.

:joint-angle [method]

&optional v &key relative
&allow-other-keys

return joint-angle if v is not set, if v is given, set joint angle. v is linear value in [mm].

:joint-dof [method]

Returns DOF of linear joint, 1.

:speed-to-angle *v* [method]

Returns [mm] of given input in [m]

:angle-to-speed *v* [method]

Returns [m] of given input in [mm]

:calc-angle-speed-gain *dav i periodic-time* [method]

:calc-jacobian *&rest args* [method]

:angle-to-speed *v* [method]

Returns [m] of given input in [mm]

:speed-to-angle *v* [method]

Returns [mm] of given input in [m]

:joint-dof [method]

Returns DOF of linear joint, 1.

:joint-angle [method]

&optional v &key relative
&allow-other-keys

return joint-angle if v is not set, if v is given, set joint angle. v is linear value in [mm].

:init [method]

&rest args &key ((:axis ax) :z)
((:max-joint-velocity mjv) (/ pi 4))
((:max-joint-torque mjt) 100)
&allow-other-keys

Create instance of linear-joint. :axis is either (:x, :y, :z) or vector. :min-angle and :max-angle takes in [mm], but velocity and torque are given in SI units.

:calc-jacobian *&rest args* [method]

:calc-angle-speed-gain *dav i periodic-time* [method]

wheel-joint [Class]

:super **joint**
:slots axis

:init [method]

&rest args &key (min (float-vector *-inf**-inf*))
(max (float-vector *inf**inf*))
((:max-joint-velocity mjv) (float-vector (/ 0.08 0.05) (/ pi 4)))
((:max-joint-torque mjt) (float-vector 100 100))
&allow-other-keys

Create instance of wheel-joint.

:joint-angle [method]

&optional v &key relative
&allow-other-keys

return joint-angle if v is not set, if v is given, set joint angle. v is joint-angle vector, which is (float-vector translation-x[mm] rotation-z[deg])

:joint-dof [method]

Returns DOF of linear joint, 2.

:speed-to-angle *dv* [method]

Returns [mm/deg] of given input in SI unit [m/rad]

:angle-to-speed *dv* [method]

Returns SI unit [m/rad] of given input in [mm/deg]

:calc-angle-speed-gain *dav i periodic-time* [method]

:calc-jacobian *fik row column joint paxis child-link world-default-coords child-reverse move-target transform-coords rotation-axis translation-axis tmp-v0 tmp-v1 tmp-v2 tmp-v3 tmp-v3a tmp-v3b tmp-m33* [method]

:angle-to-speed *dv* [method]

Returns SI unit [m/rad] of given input in [mm/deg]

:speed-to-angle *dv* [method]

Returns [mm/deg] of given input in SI unit [m/rad]

:joint-dof [method]

Returns DOF of linear joint, 2.

:joint-angle [method]

&optional v &key relative
&allow-other-keys

return joint-angle if v is not set, if v is given, set joint angle. v is joint-angle vector, which is (float-vector translation-x[mm] rotation-z[deg])

:init [method]

&rest args &key (min (float-vector ∗-inf∗∗-inf∗))
(max (float-vector ∗inf∗∗inf∗))
((:max-joint-velocity mjv) (float-vector (/ 0.08 0.05) (/ pi 4)))
((:max-joint-torque mjt) (float-vector 100 100))
&allow-other-keys

Create instance of wheel-joint.

:calc-jacobian *fik row column joint paxis child-link world-default-coords child-reverse move-target transform-coords rotation-axis translation-axis tmp-v0 tmp-v1 tmp-v2 tmp-v3 tmp-v3a tmp-v3b tmp-m33* [method]

:calc-angle-speed-gain *dav i periodic-time* [method]

omniwheel-joint [Class]

:super	**joint**
:slots	axis

:init [method]

&rest args &key (min (float-vector ∗-inf∗∗-inf∗∗-inf∗))
(max (float-vector ∗inf∗∗inf∗∗inf∗))
((:max-joint-velocity mjv) (float-vector (/ 0.08 0.05) (/ 0.08 0.05) (/ pi 4)))
((:max-joint-torque mjt) (float-vector 100 100 100))
&allow-other-keys

create instance of omniwheel-joint.

:joint-angle [method]

&optional v &key relative
&allow-other-keys

return joint-angle if v is not set, if v is given, set joint angle. v is joint-angle vector, which is (float-vector translation-x[mm] translation-y[mm] rotation-z[deg])

:joint-dof [method]

Returns DOF of linear joint, 3.

:speed-to-angle *dv* [method]

Returns [mm/deg] of given input in SI unit [m/rad]

:angle-to-speed *dv* [method]

Returns SI unit [m/rad] of given input in [mm/deg]

:calc-angle-speed-gain *dav i periodic-time* [method]

:calc-jacobian *fik row column joint paxis child-link world-default-coords child-reverse move-target transform-coords rotation-axis translation-axis tmp-v0 tmp-v1 tmp-v2 tmp-v3 tmp-v3a tmp-v3b tmp-m33* [method]

:angle-to-speed *dv* [method]

Returns SI unit [m/rad] of given input in [mm/deg]

:speed-to-angle *dv* [method]

Returns [mm/deg] of given input in SI unit [m/rad]

:joint-dof [method]

Returns DOF of linear joint, 3.

:joint-angle [method]

&optional v &key relative
&allow-other-keys

return joint-angle if v is not set, if v is given, set joint angle. v is joint-angle vector, which is (float-vector translation-x[mm] translation-y[mm] rotation-z[deg])

:init [method]

&rest args &key (min (float-vector *-inf**-inf**-inf*))
(max (float-vector *inf**inf**inf*))
((:max-joint-velocity mjv) (float-vector (/ 0.08 0.05) (/ 0.08 0.05) (/ pi 4)))
((:max-joint-torque mjt) (float-vector 100 100 100))
&allow-other-keys

create instance of omniwheel-joint.

:calc-jacobian *fik row column joint paxis child-link world-default-coords child-reverse move-target transform-coords rotation-axis translation-axis tmp-v0 tmp-v1 tmp-v2 tmp-v3 tmp-v3a tmp-v3b tmp-m33* [method]

:calc-angle-speed-gain *dav i periodic-time* [method]

sphere-joint [Class]

:super	**joint**
:slots	axis

:init [method]

&rest args &key (min (float-vector *-inf**-inf**-inf*))
(max (float-vector *inf**inf**inf*))
((:max-joint-velocity mjv) (float-vector (/ pi 4) (/ pi 4) (/ pi 4)))
((:max-joint-torque mjt) (float-vector 100 100 100))
&allow-other-keys

Create instance of sphere-joint. min/max are defind as a region of angular velocity in degree.

:joint-angle [method]

&optional v &key relative
&allow-other-keys

return joint-angle if v is not set, if v is given, set joint angle.
v is joint-angle vector [deg] by axis-angle representation, i.e (scale rotation-angle-from-default-coords[deg] axis-unit-vector)

:joint-angle-rpy *&optional v &key relative* [method]

Return joint-angle if v is not set, if v is given, set joint-angle vector by RPY representation, i.e. (float-vector yaw[deg] roll[deg] pitch[deg])

:joint-dof [method]

Returns DOF of linear joint, 3.

:speed-to-angle *dv* [method]

Returns degree of given input in radian

:angle-to-speed *dv* [method]

Returns radian of given input in degree

:joint-euler-angle [method]

&key (axis-order '(:z :y :x))
((:child-rot m) (send child-link :rot))

Return joint-angle if v is not set, if v is given, set joint-angle vector by euler representation.

:calc-angle-speed-gain *dav i periodic-time* [method]

:calc-jacobian *fik row column joint paxis child-link world-default-coords child-reverse move-target transform-coords rotation-axis translation-axis tmp-v0 tmp-v1 tmp-v2 tmp-v3 tmp-v3a tmp-v3b tmp-m33* [method]

:joint-euler-angle [method]

&key (axis-order '(:z :y :x))
((:child-rot m) (send child-link :rot))

Return joint-angle if v is not set, if v is given, set joint-angle vector by euler representation.

:angle-to-speed *dv* [method]

Returns radian of given input in degree

:speed-to-angle *dv* [method]

Returns degree of given input in radian

:joint-dof [method]

Returns DOF of linear joint, 3.

:joint-angle-rpy *&optional v &key relative* [method]

Return joint-angle if v is not set, if v is given, set joint-angle vector by RPY representation, i.e. (float-vector yaw[deg] roll[deg] pitch[deg])

:joint-angle [method]

&optional v &key relative
&allow-other-keys

return joint-angle if v is not set, if v is given, set joint angle.
v is joint-angle vector [deg] by axis-angle representation, i.e (scale rotation-angle-from-default-coords[deg] axis-unit-vector)

:init [method]

&rest args &key (min (float-vector *-inf**-inf**-inf*))
(max (float-vector *inf**inf**inf*))
((:max-joint-velocity mjv) (float-vector (/ pi 4) (/ pi 4) (/ pi 4)))
((:max-joint-torque mjt) (float-vector 100 100 100))
&allow-other-keys

Create instance of sphere-joint. min/max are defind as a region of angular velocity in degree.

:calc-jacobian *fik row column joint paxis child-link world-default-coords child-reverse move-target transform-coords rotation-axis translation-axis tmp-v0 tmp-v1 tmp-v2 tmp-v3 tmp-v3a tmp-v3b tmp-m33* [method]

:calc-angle-speed-gain *dav i periodic-time* [method]

6dof-joint [Class]

:super	**joint**
:slots	axis

:init [method]

&rest args &key (min (float-vector *-inf**-inf**-inf**-inf**-inf**-inf*))
(max (float-vector *inf**inf**inf**inf**inf**inf*))
((:max-joint-velocity mjv) (float-vector (/ 0.08 0.05) (/ 0.08 0.05) (/ 0.08 0.05) (/ pi 4) (/ pi 4) (/ pi 4)))
((:max-joint-mjt mjt) (float-vector 100 100 100 100 100 100))
(absolute-p nil)
&allow-other-keys

Create instance of 6dof-joint.

:joint-angle [method]

&optional v &key relative
&allow-other-keys

Return joint-angle if v is not set, if v is given, set joint angle vector, which is 6D vector of 3D translation[mm] and 3D rotation[deg], i.e. (find-if #'(lambda (x) (eq (send (car x) :name) 'sphere-joint)) (documentation :joint-angle))

:joint-angle-rpy *&optional v &key relative* [method]

Return joint-angle if v is not set, if v is given, set joint angle. v is joint-angle vector, which is 6D vector of 3D translation[mm] and 3D rotation[deg], for rotation, please see (find-if #'(lambda (x) (eq (send (car x) :name) 'sphere-joint)) (documentation :joint-angle-rpy))

:joint-dof [method]

Returns DOF of linear joint, 6.

:speed-to-angle *dv* [method]

Returns [mm/deg] of given input in SI unit [m/rad]

:angle-to-speed *dv* [method]

Returns SI unit [m/rad] of given input in [mm/deg]

:calc-angle-speed-gain *dav i periodic-time* [method]

:calc-jacobian *fik row column joint paxis child-link world-default-coords child-reverse move-target transform-coords rotation-axis translation-axis tmp-v0 tmp-v1 tmp-v2 tmp-v3 tmp-v3a tmp-v3b tmp-m33* [method]

:angle-to-speed *dv* [method]

Returns SI unit [m/rad] of given input in [mm/deg]

:speed-to-angle *dv* [method]

Returns [mm/deg] of given input in SI unit [m/rad]

:joint-dof [method]

Returns DOF of linear joint, 6.

:joint-angle-rpy *&optional v &key relative* [method]

Return joint-angle if v is not set, if v is given, set joint angle. v is joint-angle vector, which is 6D vector of 3D translation[mm] and 3D rotation[deg], for rotation, please see (find-if #'(lambda (x) (eq (send (car x) :name) 'sphere-joint)) (documentation :joint-angle-rpy))

:joint-angle [method]

&optional v &key relative
&allow-other-keys

Return joint-angle if v is not set, if v is given, set joint angle vector, which is 6D vector of 3D translation[mm] and 3D rotation[deg], i.e. (find-if #'(lambda (x) (eq (send (car x) :name) 'sphere-joint)) (documentation :joint-angle))

:init [method]

&rest args &key (min (float-vector *-inf**-inf**-inf**-inf**-inf**-inf*))
(max (float-vector *inf**inf**inf**inf**inf**inf*))
((:max-joint-velocity mjv) (float-vector (/ 0.08 0.05) (/ 0.08 0.05) (/ 0.08 0.05) (/ pi 4) (/ pi 4) (/ pi 4)))
((:max-joint-mjt mjt) (float-vector 100 100 100 100 100 100))
(absolute-p nil)
&allow-other-keys

Create instance of 6dof-joint.

:calc-jacobian *fik row column joint paxis child-link world-default-coords child-reverse move-target transform-coords rotation-axis translation-axis tmp-v0 tmp-v1 tmp-v2 tmp-v3 tmp-v3a tmp-v3b tmp-m33* [method]

:calc-angle-speed-gain *dav i periodic-time* [method]

bodyset-link [Class]

:super **bodyset**
:slots joint parent-link child-links analysis-level default-coords weight acentroid inertia-tenso

:init [method]

coords &rest args &key ((:analysis-level level) :body)
((:weight w) 1)
((:centroid c) #f(0.0 0.0 0.0))
((:inertia-tensor i) (unit-matrix 3))
&allow-other-keys

Create instance of bodyset-link.

:worldcoords *&optional (level analysis-level)* [method]

Returns a coordinates object which represents this coord in the world by concatenating all the cascoords from the root to this coords.

:analysis-level *&optional v* [method]

Change analysis level :coords only changes kinematics level and :body changes geometry too.

:weight *&optional w* [method]

Returns a weight of the link. If w is given, set weight.

:centroid *&optional c* [method]

Returns a centroid of the link. If c is given, set new centroid.

:inertia-tensor *&optional i* [method]

Returns a inertia tensor of the link. If c is given, set new intertia tensor.

:joint *&rest args* [method]

Returns a joint associated with this link. If args is given, args are forward to the joint.

:add-joint *j* [method]

Set j as joint of this link

:del-joint [method]

Remove current joint of this link

:parent-link [method]

Returns parent link

:child-links [method]

Returns child links

:add-child-links *l* [method]

Add l to child links

:add-parent-link *l* [method]

Set l as parent link

:del-child-link *l* [method]

Delete l from child links

:del-parent-link [method]

Delete parent link

:default-coords *&optional c* [method]

:del-parent-link [method]

Delete parent link

:del-child-link *l* [method]

Delete l from child links

:add-parent-link *l* [method]

Set l as parent link

:add-child-links *l* [method]

Add l to child links

:child-links [method]

Returns child links

:parent-link [method]

Returns parent link

:del-joint [method]

Remove current joint of this link

:add-joint *j* [method]

Set j as joint of this link

:joint *&rest args* [method]

Returns a joint associated with this link. If args is given, args are forward to the joint.

:inertia-tensor *&optional i* [method]

Returns a inertia tensor of the link. If c is given, set new intertia tensor.

:centroid *&optional c* [method]

Returns a centroid of the link. If c is given, set new centroid.

:weight *&optional w* [method]

Returns a weight of the link. If w is given, set weight.

:analysis-level *&optional v* [method]

Change analysis level :coords only changes kinematics level and :body changes geometry too.

:worldcoords *&optional (level analysis-level)* [method]

Returns a coordinates object which represents this coord in the world by concatenating all the cascoords from the root to this coords.

:init [method]

coords &rest args &key ((:analysis-level level) :body)
((:weight w) 1)
((:centroid c) #f(0.0 0.0 0.0))
((:inertia-tensor i) (unit-matrix 3))
&allow-other-keys

Create instance of bodyset-link.

:default-coords *&optional c* [method]

cascaded-link [Class]

:super	**cascaded-coords**
:slots	links joint-list bodies collision-avoidance-links end-coords-list

:init [method]

&rest args &key name
&allow-other-keys

Create cascaded-link.

:init-ending [method]

This method is to called finalize the instantiation of the cascaded-link. This update bodies and child-link and parent link from joint-list

:links *&rest args* [method]

Returns links, or args is passed to links

:joint-list *&rest args* [method]

Returns joint list, or args is passed to joints

:link *name* [method]

Return a link with given name.

:joint *name* [method]

Return a joint with given name.

:end-coords *name* [method]

Returns end-coords with given name

:bodies *&rest args* [method]

Return bodies of this object. If args is given it passed to all bodies

:faces [method]

Return faces of this object.

:angle-vector *&optional vec (angle-vector (instantiate float-vector (calc-target-joint-dimension joint-list)))* [method]

Returns angle-vector of this object, if vec is given, it updates angles of all joint. If given angle-vector violate min/max range, the value is modified.

:link-list *to &optional from* [method]

Find link list from to link to from link.

:plot-joint-min-max-table *joint0 joint1* [method]

Plot joint min max table on Euslisp window.

:calc-jacobian-from-link-list [method]

link-list &rest args &key move-target
(transform-coords move-target)
(rotation-axis (cond ((atom move-target) nil) (t (make-list (length move-target)))))
(translation-axis (cond ((atom move-target) t) (t (make-list (length move-target) :
(col-offset 0)
(dim (send self :calc-target-axis-dimension rotation-axis translation-axis))
(fik-len (send self :calc-target-joint-dimension link-list))
fik
(tmp-v0 (instantiate float-vector 0))
(tmp-v1 (instantiate float-vector 1))
(tmp-v2 (instantiate float-vector 2))
(tmp-v3 (instantiate float-vector 3))
(tmp-v3a (instantiate float-vector 3))
(tmp-v3b (instantiate float-vector 3))
(tmp-m33 (make-matrix 3 3))
&allow-other-keys

Calculate jacobian matrix from link-list and move-target. Jacobian is represented in :transform-coords. Unit system is [m] or [rad], not [mm] or [deg].
Joint order for this jacobian matrix follows link-list order. Joint torque[Nm] order is also the same.
Ex1. One-Arm
(setq *rarm-link-list*(send *robot*:link-list (send *robot*:rarm :end-coords :parent)))
(send-all *rarm-link-list*:joint)
Ex2. Two-Arm
(setq *arms-link-list*(mapcar #'(lambda (l) (send *robot*:link-list (send *robot*l :end-coords :parent))) '(:rarm :larm)))
(send-all (send *robot*:calc-union-link-list *arms-link-list*) :joint)

:move-joints-avoidance [method]

union-vel &rest args &key union-link-list
link-list
(fik-len (send self :calc-target-joint-dimension union-link-list))
(weight (fill (instantiate float-vector fik-len) 1))
(null-space)
(avoid-nspace-gain 0.01)
(avoid-weight-gain 1.0)
(avoid-collision-distance 200)
(avoid-collision-null-gain 1.0)
(avoid-collision-joint-gain 1.0)
((:collision-avoidance-link-pair pair-list) (send self :collision-avoidance-link-pair-from-lin
(cog-gain 0.0)
(target-centroid-pos)
(centroid-offset-func)
(cog-translation-axis :z)
(cog-null-space nil)
(additional-weight-list)
(additional-nspace-list)
(tmp-len (instantiate float-vector fik-len))
(tmp-len2 (instantiate float-vector fik-len))
(tmp-weight (instantiate float-vector fik-len))
(tmp-nspace (instantiate float-vector fik-len))
(tmp-mcc (make-matrix fik-len fik-len))
(tmp-mcc2 (make-matrix fik-len fik-len))
(debug-view)
(jacobi)
&allow-other-keys

:move-joints-avoidance is called in :inverse-kinematics-loop. In this method, calculation of joint position difference are executed and joint positon are moved.
Optional arguments:
:weight
float-vector of inverse weight of velocity of each joint or a function which returns the float-vector or a list which returns the float-vector. Length of the float-vector should be same as the number of columns of the jacobian. If :weight is a function or a list, it is called in each IK loop as (funcall weight

union-link-list) or (eval weight). :weight is used in calculation of weighted norm of joint velocity for sr-inverse. Default is the float-vector filled with 1.

:null-space

float-vector of joint velocity or a function which returns the float-vector or a list which returns the float-vector. Length of the float-vector should be same as the number of columns of the jacobian. If :null-space is a function or a list, it is called in each IK loop as (funcall null-space) or (eval null-space). This joint velocity is applied in null space of main task in each IK loop. Default is nil.

:avoid-nspace-gain

gain of joint velocity to avoid joint limit applied in null space of main task in each IK loop. The avoiding velocity is calculated as $(((t\_max + t\_min)/2 - t)/((t\_max - t\_min)/2))^2$. Default is 0.01.

:avoid-weight-gain

gain of dH/dt in calcuration of avoiding joint limits weight. :weight is devided by this avoiding joint limits weight. Default is 1.0.

If :avoid-nspace-gain is 0, :weight is multipled by :weight instead.

:avoid-collision-distance

yellow zone. 0.1avoid-collision-distance is regarded as orange zone.

If :avoid-collision-joint-gain is smaller than or equal to 0.0, yellow zone and orange zone become inactive. Default is 200[mm].

:avoid-collision-null-gain

k_null. Default is 1.0.

:avoid-collision-joint-gain

k_joint. Default is 1.0.

:collision-avoidance-link-pair

(list (list link1 link2) (list link3 link4) ...) with any length. Collision between paired links is cared. Default is (send self :collision-avoidance-link-pair-from-link-list link-list :obstacles (cadr (memq :obstacles args))).

:additional-weight-list

(list (list target-link1 additional-weight1) (list target-link2 additional-weight2) ...) with any length. The component of :weight corresponding to the parent joint of target-link is scaled by additional-weight. additional-weight should be float (if 1 dof), float-vector with length of the joint dof, or a function which returns the float or float-vector. if additional-weight is a function, it is called in each IK loop as (funcall additional-weight). Default is nil.

:additional-nspace-list

(list (list target-link1 var1) (list target-link2 var2) ...) with any length. In each IK loop, the parent joint of target-link is moved by the amount of var in null space of main task. var should be float (if 1dof), float-vector with the same length of the target joint dof, or a function which returns the float or float-vector.If var is float-vector, it is called in each IK loop as (funcall var). Default is nil.

other-keys

:manipulability-limit

If manipulability of jacobian is smaller than manipulability-limit, diagonal matrix is added in calculation of sr-inverse. Default is 0.1.

:manipulability-gain

Weight of diagonal matrix in calculation of sr-inverse. Weight is calculated as (∗manipulability-gain (expt (- 1.0 (/ manipulability manipulability-limit)) 2)). Default is 0.001.

:collision-distance-limit

Threshold for collision detection. If collision is detected, the distance between the colliding links is considered to be :collision-distance-limit instead of actual distance. Default is 10[mm].

:move-joints-hook

A function which is called right after joints are moved in each IK loop as (funcall move-joints-hook). Default is nil.

:inverse-kinematics-loop [method]

dif-pos dif-rot &rest args &key (stop 1)
(loop 0)
link-list
move-target
(rotation-axis (cond ((atom move-target) t) (t (make-list (length move-target) :initial-
(translation-axis (cond ((atom move-target) t) (t (make-list (length move-target) :init
(thre (cond ((atom move-target) 1) (t (make-list (length move-target) :initial-element
(rthre (cond ((atom move-target) (deg2rad 1)) (t (make-list (length move-target) :init
(dif-pos-ratio 1.0)
(dif-rot-ratio 1.0)
union-link-list
target-coords
(jacobi)
(additional-check)
(additional-jacobi)
(additional-vel)
(centroid-thre 1.0)
(target-centroid-pos)
(centroid-offset-func)
(cog-translation-axis :z)
(cog-null-space nil)
(cog-gain 1.0)
(min-loop (/ stop 10))
debug-view
ik-args
&allow-other-keys

:inverse-kinematics-loop is one loop calculation for :inverse-kinematics.
In this method, joint position difference satisfying workspace difference (dif-pos, dif-rot) are calculated and euslisp model joint angles are updated.
Optional arguments:

:dif-pos-ratio

Ratio of spacial velocity used in calculation of joint position difference to workspace difference (after :p-limit applied). Default is 1.0.

:dif-rot-ratio

Ratio of angular velocity used in calculation of joint position difference to workspace difference (after :r-limit applied). Default is 1.0.

:jacobi

A jacobian of all move-target or a function that returns the jacobian. When a function is given, It is called in every IK loop as (funcall jacobi link-list move-target translation-axis rotation-axis). Default is (send*self :calc-jacobian-from-link-list link-list :translation-axis translation-axis :rotation-axis rotation-axis :move-target move-target method-args).

:additional-check

This argument is to add optional best-effort convergence conditions. Default is nil (no additional check). :additional-check should be function or lambda.

best-effort =>

In :inverse-kinematics-loop, 'success' is overwritten by '(and success additional-check)'

In :inverse-kinematics, 'success is not overwritten.

So, :inverse-kinematics-loop wait until ':additional-check' becomes 't' as possible,

but ':additional-check' is neglected in the final :inverse-kinematics return.

:cog-gain

Ratio of centroid velocity used in calculation of joint position difference to centroid position difference. max 1.0. Default is 1.0.

:min-loop

Minimam loop count. Defalt (/ stop 10).

If integer is specified, :inverse-kinematics-loop does returns :ik-continues and continuing solving IK.

If min-loop is nil, do not consider loop counting for IK convergence.

other-keys

:move-joints-avoidance is internally called and args are passed to it. See the explanation of :move-joints-avoidance.

:p-limit

Maximum spacial velocity of each move-target in one IK loop. Default is 100.0[mm].

:r-limit

Maximum angular velocity of each move-target in one IK loop. Default is 0.5[rad].

:inverse-kinematics [method]

target-coords &rest args &key (stop 50)
(link-list)
(move-target)
(debug-view)
(warnp t)
(revert-if-fail t)
(rotation-axis (cond ((atom move-target) t) (t (make-list (length move-target) :initial-eleme
(translation-axis (cond ((atom move-target) t) (t (make-list (length move-target) :initial-ele
(joint-args)
(thre (cond ((atom move-target) 1) (t (make-list (length move-target) :initial-element 1))))
(rthre (cond ((atom move-target) (deg2rad 1)) (t (make-list (length move-target) :initial-ele
(union-link-list)
(centroid-thre 1.0)
(target-centroid-pos)
(centroid-offset-func)
(cog-translation-axis :z)
(cog-null-space nil)
(dump-command :fail-only)
(periodic-time 0.5)
(check-collision)
(additional-jacobi)
(additional-vel)

&allow-other-keys

Move move-target to target-coords.

;; target-coords, link-list and move-target need to be given.

;; target-coords, move-target, rotation-axis, translation-axis, link-list

;; ->both list and atom OK.

:target-coords

The coordinate of the target, or a function that returns coordinates. When a function is given, it is called in every IK loop as (funcall target-coords). Use a list of targets to solve the IK relative to multiple end links simultaneously.

:stop

Maximum number for IK iteration. Default is 50.

:link-list

List of links to control. When the target-coords is list, this should be a list of lists.

:move-target

Specify end-effector coordinate. When the target-coords is list, this should be list too.

:debug-view

Set t to show debug message and visualization. Use :no-message to just show the irtview image. Default is nil.

:warnp

Set nil to not display debug message when the IK failed. Default is t.

:revert-if-fail

Set nil to keep the angle posture of IK solve iteration. Default is t, which return to original position when IK fails.

:rotation-axis

Use nil for position only IK. :x, :y, :z for the constraint around axis with plus direction, :-x :-y :-z for axis with minus direction. :zz :yy :zz for direction free axis. :xm :ym :zm for allowing rotation with respect to mirror direction. When the target-coords is list, this should be list too. Default is t.

:translation-axis

:x :y :z for constraint along the x, y, z axis. :xy :yz :zx for plane. Default is t.

:joint-args

Arguments passed to joint as (send*joint :joint-angle angle :relative t joint-args). Default nil.

:thre

Threshold for position error to terminate IK iteration. Default is 1 [mm].

:rthre

Threshold for rotation error to terminate IK iteration. Default is 0.017453 [rad] (1 deg).

:union-link-list

a function that returns union of link-list called as (funcall union-link-list link-list). Default is (send self :calc-union-link-list link-list).

:centroid-thre

Threshold for centroid position error to terminate IK iteration. Default is 1 [mm].

:target-centroid-pos

The float-vector of the target centroid position. Default is nil.

:centroid-offset-func

A function that returns centroid position. This function is called in each IK loop as (funcall centroid-offset-func). Default is (send self :centroid).

:cog-translation-axis

:x :y :z for constraint along the x, y, z axis. :xy :yz :zx for plane. t for point. Default is :z.

:cog-null-space

t: centroid position task is solved in null space of the main task. Default is nil.

:dump-command

should be t, nil, :always, :fail-only, :always-with-debug-log, or :fail-only-with-debug-log. Log are success/fail log and ik debug information log.

t or :always : dump log both in success and fail (for success/fail log).

:always-with-debug-log : dump log both in success and fail (for success/fail log and ik debug information log).

:fail-only : dump log only in fail (for success/fail log).

:always-with-debug-log : dump log only in fail (for success/fail log and ik debug information log).

nil : do not dump log.

:periodic-time

The amount of joint angle modification in each IK loop is scaled not to violate joint velocity limit times :periodic-time. Default is 0.5[s].

:check-collision

Set t to return false when self collision occurs at found IK solution. Default is nil. To avoid collision within IK loop, use set links to collision-avoidance-links slot of cascaded-link.

:additional-jacobi

The jacobian of the additional main task, or a function that returns the jacobian. When a function is given, it is called in every IK loop as (funcall additional-jacobi link-list). Use a list of additional-jacobi to solve the IK for multiple additional task. Default is nil.

:additional-vel

The velocity of additional main task, or a function that returns the velocity. When a function is given, it is called in every IK loop as (funcall additional-vel link-list). The velocity should be expressed in the same coordinates as corresponding additional-jacobi. When the additional-jacobi is list, this should be a list of same length. Default is nil.

other-keys

function :inverse-kinematics-loop is internally called and args are passed to it. See the explanation of :inverse-kinematics-loop.

:calc-grasp-matrix [method]

contact-points &key (ret (make-matrix 6 (*6 (length contact-points))))
(contact-rots (mapcar #'(lambda (r) (unit-matrix 3)) contact-points))

Calc grasp matrix.
Grasp matrix is defined as
— E_3 0 E_3 0 ... —
— p1x E_3 p2x E_3 ... —
Arguments:
contact-points is list of contact points[mm].
contact-rots is list of contact coords rotation[rad].
If contact-rots is specified, grasp matrix as follow:
— R1 0 R2 0 ... —
— p1xR1 R1 p2xR2 R2 ... —

:inverse-kinematics-for-closed-loop-forward-kinematics [method]

target-coords &rest args &key (move-target)

(link-list)
(move-joints-hook)
(additional-weight-list)
(constrained-joint-list)
(constrained-joint-angle-list)
(min-loop 2)
&allow-other-keys

Solve inverse-kinematics for closed loop forward kinematics.
Move move-target to target-coords with link-list.
link-list loop should be close when move-target reaches target-coords.
constrained-joint-list is list of joints specified given joint angles in closed loop.
constrained-joint-angle-list is list of joint-angle for constrained-joint-list.

:calc-jacobian-for-interlocking-joints *link-list &key (interlocking-joint-pairs (send self :interlocking-joint-pairs))* [method]

Calculate jacobian to keep interlocking joint velocity same.
dtheta_0 = dtheta_1 =>[... 0 1 0 ... 0 -1 0 ....][...dtheta_0...dtheta_1...]

:calc-vel-for-interlocking-joints *link-list &key (interlocking-joint-pairs (send self :interlocking-joint-pairs))* [method]

Calculate 0 velocity for keeping interlocking joint at the same joint angle.

:set-midpoint-for-interlocking-joints *&key (interlocking-joint-pairs (send self :interlocking-joint-pairs))* [method]

Set interlocking joints at mid point of each joint angle.

:interlocking-joint-pairs [method]

Interlocking joint pairs.
pairs are (list (cons joint0 joint1) ...)
If users want to use interlocking joints, please overwrite this method.

:check-interlocking-joint-angle-validity [method]

&key (angle-thre 0.001)
(interlocking-joint-pairs (send self :interlocking-joint-pairs))

Check if all interlocking joint pairs are same values.

:update-descendants *&rest args* [method]

:find-link-route *to &optional from* [method]

:make-joint-min-max-table *l0 l1 joint0 joint1 &key (fat 0) (fat2 nil) (debug nil) (margin 0.0) (overwrite-collision-model nil)* [method]

:make-min-max-table-using-collision-check *l0 l1 joint0 joint1 joint-range0 joint-range1 min-joint0 min-joint1 fat fat2 debug margin* [method]

:plot-joint-min-max-table-common *joint0 joint1* [method]

:calc-target-axis-dimension *rotation-axis translation-axis* [method]

:calc-union-link-list *link-list* [method]

:calc-target-joint-dimension *link-list* [method]

:calc-inverse-jacobian *jacobi &rest args &key ((:manipulability-limit ml) 0.1) ((:manipulability-gain mg) 0.001) weight debug-view ret wmat tmat umat umat2 mat-tmp mat-tmp-rc tmp-mrr tmp-mrr2 &allow-other-keys* [method]

:calc-gradh-from-link-list *link-list &optional (res (instantiate float-vector (length link-list)))* [method]

:calc-joint-angle-speed *union-vel &rest args &key angle-speed (angle-speed-blending 0.5) jacobi jacobi# null-space i-j#j debug-view weight wmat tmp-len tmp-len2 fik-len &allow-other-keys* [method]

:calc-joint-angle-speed-gain *union-link-list dav periodic-time* [method]

:collision-avoidance-links *&optional l* [method]

:collision-avoidance-link-pair-from-link-list *link-lists &key obstacles ((:collision-avoidance-links collision-links) collision-avoidance-links) debug* [method]

:collision-avoidance-calc-distance *&rest args &key union-link-list (warnp t) ((:collision-avoidance-link-pair pair-list)) ((:collision-distance-limit distance-limit) 10) &allow-other-keys* [method]

:collision-avoidance-args *pair link-list* [method]

:collision-avoidance *&rest args &key avoid-collision-distance avoid-collision-joint-gain avoid-collision-null-gain ((:collision-avoidance-link-pair pair-list)) (union-link-list) (link-list) (weight) (fik-len (send self :calc-target-joint-dimension union-link-list)) debug-view &allow-other-keys* [method]

:move-joints *union-vel &rest args &key union-link-list (periodic-time 0.05) (joint-args) (debug-view nil) (move-joints-hook) &allow-other-keys* [method]

:find-joint-angle-limit-weight-old-from-union-link-list *union-link-list* [method]

:reset-joint-angle-limit-weight-old *union-link-list* [method]

:calc-weight-from-joint-limit *avoid-weight-gain fik-len link-list union-link-list debug-view weight tmp-weight tmp-len* [method]

:calc-inverse-kinematics-weight-from-link-list *link-list &key (avoid-weight-gain 1.0) (union-link-list (send self :calc-union-link-list link-list)) (fik-len (send self :calc-target-joint-dimension union-link-list)) (weight (fill (instantiate float-vector fik-len) 1)) (additional-weight-list) (debug-view) (tmp-weight (instantiate float-vector fik-len)) (tmp-len (instantiate float-vector fik-len))* [method]

:calc-nspace-from-joint-limit *avoid-nspace-gain union-link-list weight debug-view tmp-nspace* [method]

:calc-inverse-kinematics-nspace-from-link-list *link-list &key (avoid-nspace-gain 0.01) (union-link-list (send self :calc-union-link-list link-list)) (fik-len (send self :calc-target-joint-dimension union-link-list)) (null-space) (debug-view) (additional-nspace-list) (cog-gain 0.0) (target-centroid-pos) (centroid-offset-func) (cog-translation-axis :z) (cog-null-space nil) (weight (fill (instantiate float-vector fik-len) 1.0)) (update-mass-properties t) (tmp-nspace (instantiate float-vector fik-len))* [method]

:inverse-kinematics-args *&rest args &key union-link-list rotation-axis translation-axis additional-jacobi-dimension &allow-other-keys* [method]

:draw-collision-debug-view [method]

:ik-convergence-check *success dif-pos dif-rot rotation-axis translation-axis thre rthre centroid-thre target-centroid-pos centroid-offset-func cog-translation-axis &key (update-mass-properties t)* [method]

:calc-vel-from-pos *dif-pos translation-axis &rest args &key (p-limit 100.0) (tmp-v0 (instantiate float-vector 0)) (tmp-v1 (instantiate float-vector 1)) (tmp-v2 (instantiate float-vector 2)) (tmp-v3 (instantiate float-vector 3)) &allow-other-keys* [method]

:calc-vel-from-rot *dif-rot rotation-axis &rest args &key (r-limit 0.5) (tmp-v0 (instantiate float-vector 0)) (tmp-v1 (instantiate float-vector 1)) (tmp-v2 (instantiate float-vector 2)) (tmp-v3 (instantiate float-vector 3)) &allow-other-keys* [method]

:collision-check-pairs *&key ((:links ls) (cons (car links) (all-child-links (car links))))* [method]

:self-collision-check *&key (mode :all) (pairs (send self :collision-check-pairs)) (collision-func 'collision-check)* [method]

:plot-joint-min-max-table *joint0 joint1* [method]

Plot joint min max table on Euslisp window.

:link-list *to &optional from* [method]

Find link list from to link to from link.

:angle-vector *&optional vec (angle-vector (instantiate float-vector (calc-target-joint-dimension joint-list)))* [method]

Returns angle-vector of this object, if vec is given, it updates angles of all joint. If given angle-vector violate min/max range, the value is modified.

:faces [method]

Return faces of this object.

:bodies *&rest args* [method]

Return bodies of this object. If args is given it passed to all bodies

:end-coords *name* [method]

Returns end-coords with given name

:joint *name* [method]

Return a joint with given name.

:link *name* [method]

Return a link with given name.

:joint-list *&rest args* [method]

Returns joint list, or args is passed to joints

:links *&rest args* [method]

Returns links, or args is passed to links

:init-ending [method]

This method is to called finalize the instantiation of the cascaded-link. This update bodies and child-link and parent link from joint-list

:init [method]

&rest args &key name
&allow-other-keys

Create cascaded-link.

:plot-joint-min-max-table-common *joint0 joint1* [method]

:make-min-max-table-using-collision-check *l0 l1 joint0 joint1 joint-range0 joint-range1 min-joint0 min-joint1 fat fat2 debug margin* [method]

:make-joint-min-max-table *l0 l1 joint0 joint1 &key (fat 0) (fat2 nil) (debug nil) (margin 0.0) (overwrite-collision-model nil)* [method]

:find-link-route *to &optional from* [method]

:update-descendants *&rest args* [method]

:check-interlocking-joint-angle-validity [method]

&key (angle-thre 0.001)
(interlocking-joint-pairs (send self :interlocking-joint-pairs))

Check if all interlocking joint pairs are same values.

:interlocking-joint-pairs [method]

Interlocking joint pairs.
pairs are (list (cons joint0 joint1) ...)
If users want to use interlocking joints, please overwrite this method.

:set-midpoint-for-interlocking-joints *&key (interlocking-joint-pairs (send self :interlocking-joint-pairs))* [method]

Set interlocking joints at mid point of each joint angle.

:calc-vel-for-interlocking-joints *link-list &key (interlocking-joint-pairs (send self :interlocking-joint-pairs))* [method]

Calculate 0 velocity for keeping interlocking joint at the same joint angle.

:calc-jacobian-for-interlocking-joints *link-list &key (interlocking-joint-pairs (send self :interlocking-joint-pairs))* [method]

Calculate jacobian to keep interlocking joint velocity same.
dtheta_0 = dtheta_1 =>[... 0 1 0 ... 0 -1 0 ....][...dtheta_0...dtheta_1...]

:inverse-kinematics-for-closed-loop-forward-kinematics [method]

target-coords &rest args &key (move-target)
(link-list)
(move-joints-hook)
(additional-weight-list)
(constrained-joint-list)
(constrained-joint-angle-list)
(min-loop 2)
&allow-other-keys

Solve inverse-kinematics for closed loop forward kinematics.
Move move-target to target-coords with link-list.
link-list loop should be close when move-target reaches target-coords.
constrained-joint-list is list of joints specified given joint angles in closed loop.
constrained-joint-angle-list is list of joint-angle for constrained-joint-list.

:calc-grasp-matrix [method]

contact-points &key (ret (make-matrix 6 (*6 (length contact-points))))
(contact-rots (mapcar #'(lambda (r) (unit-matrix 3)) contact-points))

Calc grasp matrix.
Grasp matrix is defined as
— E_3 0 E_3 0 ... —
— p1x E_3 p2x E_3 ... —
Arguments:
contact-points is list of contact points[mm].
contact-rots is list of contact coords rotation[rad].
If contact-rots is specified, grasp matrix as follow:
— R1 0 R2 0 ... —
— p1xR1 R1 p2xR2 R2 ... —

:inverse-kinematics [method]

target-coords &rest args &key (stop 50)
(link-list)
(move-target)
(debug-view)
(warnp t)
(revert-if-fail t)
(rotation-axis (cond ((atom move-target) t) (t (make-list (length move-target) :initial-eleme
(translation-axis (cond ((atom move-target) t) (t (make-list (length move-target) :initial-ele
(joint-args)
(thre (cond ((atom move-target) 1) (t (make-list (length move-target) :initial-element 1))))
(rthre (cond ((atom move-target) (deg2rad 1)) (t (make-list (length move-target) :initial-ele
(union-link-list)
(centroid-thre 1.0)
(target-centroid-pos)
(centroid-offset-func)
(cog-translation-axis :z)
(cog-null-space nil)
(dump-command :fail-only)
(periodic-time 0.5)
(check-collision)
(additional-jacobi)
(additional-vel)
&allow-other-keys

Move move-target to target-coords.
;; target-coords, link-list and move-target need to be given.
;; target-coords, move-target, rotation-axis, translation-axis, link-list
;; ->both list and atom OK.
:target-coords
The coordinate of the target, or a function that returns coordinates. When a function is given, it is called in every IK loop as (funcall target-coords). Use a list of targets to solve the IK relative to multiple end links simultaneously.
:stop
Maximum number for IK iteration. Default is 50.
:link-list
List of links to control. When the target-coords is list, this should be a list of lists.

:move-target

Specify end-effector coordinate. When the target-coords is list, this should be list too.

:debug-view

Set t to show debug message and visualization. Use :no-message to just show the irtview image. Default is nil.

:warnp

Set nil to not display debug message when the IK failed. Default is t.

:revert-if-fail

Set nil to keep the angle posture of IK solve iteration. Default is t, which return to original position when IK fails.

:rotation-axis

Use nil for position only IK. :x, :y, :z for the constraint around axis with plus direction, :-x :-y :-z for axis with minus direction. :zz :yy :zz for direction free axis. :xm :ym :zm for allowing rotation with respect to mirror direction. When the target-coords is list, this should be list too. Default is t.

:translation-axis

:x :y :z for constraint along the x, y, z axis. :xy :yz :zx for plane. Default is t.

:joint-args

Arguments passed to joint as (send*joint :joint-angle angle :relative t joint-args). Default nil.

:thre

Threshold for position error to terminate IK iteration. Default is 1 [mm].

:rthre

Threshold for rotation error to terminate IK iteration. Default is 0.017453 [rad] (1 deg).

:union-link-list

a function that returns union of link-list called as (funcall union-link-list link-list). Default is (send self :calc-union-link-list link-list).

:centroid-thre

Threshold for centroid position error to terminate IK iteration. Default is 1 [mm].

:target-centroid-pos

The float-vector of the target centroid position. Default is nil.

:centroid-offset-func

A function that returns centroid position. This function is called in each IK loop as (funcall centroid-offset-func). Default is (send self :centroid).

:cog-translation-axis

:x :y :z for constraint along the x, y, z axis. :xy :yz :zx for plane. t for point. Default is :z.

:cog-null-space

t: centroid position task is solved in null space of the main task. Default is nil.

:dump-command

should be t, nil, :always, :fail-only, :always-with-debug-log, or :fail-only-with-debug-log. Log are success/fail log and ik debug information log.

t or :always : dump log both in success and fail (for success/fail log).

:always-with-debug-log : dump log both in success and fail (for success/fail log and ik debug information log).

:fail-only : dump log only in fail (for success/fail log).

:always-with-debug-log : dump log only in fail (for success/fail log and ik debug information log).

nil : do not dump log.

:periodic-time

The amount of joint angle modification in each IK loop is scaled not to violate joint velocity limit times :periodic-time. Default is 0.5[s].

:check-collision

Set t to return false when self collision occurs at found IK solution. Default is nil. To avoid collision within IK loop, use set links to collision-avoidance-links slot of cascaded-link.

:additional-jacobi

The jacobian of the additional main task, or a function that returns the jacobian. When a function is given, it is called in every IK loop as (funcall additional-jacobi link-list). Use a list of additional-jacobi to solve the IK for multiple additional task. Default is nil.

:additional-vel

The velocity of additional main task, or a function that returns the velocity. When a function is given, it is called in every IK loop as (funcall additional-vel link-list). The velocity should be expressed in the same coordinates as corresponding additional-jacobi. When the additional-jacobi is list, this should be a list of same length. Default is nil.

other-keys

function :inverse-kinematics-loop is internally called and args are passed to it. See the explanation of :inverse-kinematics-loop.

:inverse-kinematics-loop [method]

dif-pos dif-rot &rest args &key (stop 1)
(loop 0)
link-list
move-target
(rotation-axis (cond ((atom move-target) t) (t (make-list (length move-target) :initial-
(translation-axis (cond ((atom move-target) t) (t (make-list (length move-target) :init
(thre (cond ((atom move-target) 1) (t (make-list (length move-target) :initial-element
(rthre (cond ((atom move-target) (deg2rad 1)) (t (make-list (length move-target) :init
(dif-pos-ratio 1.0)
(dif-rot-ratio 1.0)
union-link-list
target-coords
(jacobi)
(additional-check)
(additional-jacobi)
(additional-vel)
(centroid-thre 1.0)
(target-centroid-pos)
(centroid-offset-func)
(cog-translation-axis :z)
(cog-null-space nil)
(cog-gain 1.0)
(min-loop (/ stop 10))
debug-view
ik-args
&allow-other-keys

:inverse-kinematics-loop is one loop calculation for :inverse-kinematics.

In this method, joint position difference satisfying workspace difference (dif-pos, dif-rot) are calculated and euslisp model joint angles are updated.

Optional arguments:
:dif-pos-ratio
Ratio of spacial velocity used in calculation of joint position difference to workspace difference (after :p-limit applied). Default is 1.0.
:dif-rot-ratio
Ratio of angular velocity used in calculation of joint position difference to workspace difference (after :r-limit applied). Default is 1.0.
:jacobi
A jacobian of all move-target or a function that returns the jacobian. When a function is given, It is called in every IK loop as (funcall jacobi link-list move-target translation-axis rotation-axis). Default is (send*self :calc-jacobian-from-link-list link-list :translation-axis translation-axis :rotation-axis rotation-axis :move-target move-target method-args).
:additional-check
This argument is to add optional best-effort convergence conditions. Default is nil (no additional check). :additional-check should be function or lambda.
best-effort =>
In :inverse-kinematics-loop, 'success' is overwritten by '(and success additional-check)'
In :inverse-kinematics, 'success is not overwritten.
So, :inverse-kinematics-loop wait until ':additional-check' becomes 't' as possible,
but ':additional-check' is neglected in the final :inverse-kinematics return.
:cog-gain
Ratio of centroid velocity used in calculation of joint position difference to centroid position difference. max 1.0. Default is 1.0.
:min-loop
Minimam loop count. Defalt (/ stop 10).
If integer is specified, :inverse-kinematics-loop does returns :ik-continues and continuing solving IK.
If min-loop is nil, do not consider loop counting for IK convergence.
other-keys
:move-joints-avoidance is internally called and args are passed to it. See the explanation of :move-joints-avoidance.
:p-limit
Maximum spacial velocity of each move-target in one IK loop. Default is 100.0[mm].
:r-limit
Maximum angular velocity of each move-target in one IK loop. Default is 0.5[rad].

:move-joints-avoidance [method]

union-vel &rest args &key union-link-list
link-list
(fik-len (send self :calc-target-joint-dimension union-link-list))
(weight (fill (instantiate float-vector fik-len) 1))
(null-space)
(avoid-nspace-gain 0.01)
(avoid-weight-gain 1.0)
(avoid-collision-distance 200)
(avoid-collision-null-gain 1.0)
(avoid-collision-joint-gain 1.0)

((:collision-avoidance-link-pair pair-list) (send self :collision-avoidance-link-pair-from-lin
(cog-gain 0.0)
(target-centroid-pos)
(centroid-offset-func)
(cog-translation-axis :z)
(cog-null-space nil)
(additional-weight-list)
(additional-nspace-list)
(tmp-len (instantiate float-vector fik-len))
(tmp-len2 (instantiate float-vector fik-len))
(tmp-weight (instantiate float-vector fik-len))
(tmp-nspace (instantiate float-vector fik-len))
(tmp-mcc (make-matrix fik-len fik-len))
(tmp-mcc2 (make-matrix fik-len fik-len))
(debug-view)
(jacobi)
&allow-other-keys

:move-joints-avoidance is called in :inverse-kinematics-loop. In this method, calculation of joint position difference are executed and joint positon are moved.
Optional arguments:
:weight
float-vector of inverse weight of velocity of each joint or a function which returns the float-vector or a list which returns the float-vector. Length of the float-vector should be same as the number of columns of the jacobian. If :weight is a function or a list, it is called in each IK loop as (funcall weight union-link-list) or (eval weight). :weight is used in calculation of weighted norm of joint velocity for sr-inverse. Default is the float-vector filled with 1.
:null-space
float-vector of joint velocity or a function which returns the float-vector or a list which returns the float-vector. Length of the float-vector should be same as the number of columns of the jacobian. If :null-space is a function or a list, it is called in each IK loop as (funcall null-space) or (eval null-space). This joint velocity is applied in null space of main task in each IK loop. Default is nil.
:avoid-nspace-gain
gain of joint velocity to avoid joint limit applied in null space of main task in each IK loop. The avoiding velocity is calculated as $(((t\_max + t\_min)/2 - t)/((t\_max - t\_min)/2))^2$. Default is 0.01.
:avoid-weight-gain
gain of dH/dt in calcuration of avoiding joint limits weight. :weight is devided by this avoiding joint limits weight. Default is 1.0.
If :avoid-nspace-gain is 0, :weight is multipled by :weight instead.
:avoid-collision-distance
yellow zone. 0.1avoid-collision-distance is regarded as orange zone.
If :avoid-collision-joint-gain is smaller than or equal to 0.0, yellow zone and orange zone become inactive. Default is 200[mm].
:avoid-collision-null-gain
k_null. Default is 1.0.
:avoid-collision-joint-gain
k_joint. Default is 1.0.

:collision-avoidance-link-pair

(list (list link1 link2) (list link3 link4) ...) with any length. Collision between paired links is cared. Default is (send self :collision-avoidance-link-pair-from-link-list link-list :obstacles (cadr (memq :obstacles args))).

:additional-weight-list

(list (list target-link1 additional-weight1) (list target-link2 additional-weight2) ...) with any length. The component of :weight corresponding to the parent joint of target-link is scaled by additional-weight. additional-weight should be float (if 1 dof), float-vector with length of the joint dof, or a function which returns the float or float-vector. if additional-weight is a function, it is called in each IK loop as (funcall additional-weight). Default is nil.

:additional-nspace-list

(list (list target-link1 var1) (list target-link2 var2) ...) with any length. In each IK loop, the parent joint of target-link is moved by the amount of var in null space of main task. var should be float (if 1dof), float-vector with the same length of the target joint dof, or a function which returns the float or float-vector.If var is float-vector, it is called in each IK loop as (funcall var). Default is nil.

other-keys

:manipulability-limit

If manipulability of jacobian is smaller than manipulability-limit, diagonal matrix is added in calculation of sr-inverse. Default is 0.1.

:manipulability-gain

Weight of diagonal matrix in calculation of sr-inverse. Weight is calculated as (*manipulability-gain (expt (- 1.0 (/ manipulability manipulability-limit)) 2)). Default is 0.001.

:collision-distance-limit

Threshold for collision detection. If collision is detected, the distance between the colliding links is considered to be :collision-distance-limit instead of actual distance. Default is 10[mm].

:move-joints-hook

A function which is called right after joints are moved in each IK loop as (funcall move-joints-hook). Default is nil.

:calc-jacobian-from-link-list [method]

link-list &rest args &key move-target
(transform-coords move-target)
(rotation-axis (cond ((atom move-target) nil) (t (make-list (length move-target)))))
(translation-axis (cond ((atom move-target) t) (t (make-list (length move-target) :
(col-offset 0)
(dim (send self :calc-target-axis-dimension rotation-axis translation-axis))
(fik-len (send self :calc-target-joint-dimension link-list))
fik
(tmp-v0 (instantiate float-vector 0))
(tmp-v1 (instantiate float-vector 1))
(tmp-v2 (instantiate float-vector 2))
(tmp-v3 (instantiate float-vector 3))
(tmp-v3a (instantiate float-vector 3))
(tmp-v3b (instantiate float-vector 3))
(tmp-m33 (make-matrix 3 3))
&allow-other-keys

Calculate jacobian matrix from link-list and move-target. Jacobian is represented in :transform-coords. Unit system is [m] or [rad], not [mm] or [deg].
Joint order for this jacobian matrix follows link-list order. Joint torque[Nm] order is also the same.
Ex1. One-Arm
(setq *rarm-link-list*(send *robot*:link-list (send *robot*:rarm :end-coords :parent)))
(send-all *rarm-link-list*:joint)
Ex2. Two-Arm
(setq *arms-link-list*(mapcar #'(lambda (l) (send *robot*:link-list (send *robot*l :end-coords :parent))) '(:rarm :larm)))
(send-all (send *robot*:calc-union-link-list *arms-link-list*) :joint)

:self-collision-check *&key (mode :all) (pairs (send self :collision-check-pairs)) (collision-func 'collision-check)* [method]

:collision-check-pairs *&key ((:links ls) (cons (car links) (all-child-links (car links))))* [method]

:calc-vel-from-rot *dif-rot rotation-axis &rest args &key (r-limit 0.5) (tmp-v0 (instantiate float-vector 0)) (tmp-v1 (instantiate float-vector 1)) (tmp-v2 (instantiate float-vector 2)) (tmp-v3 (instantiate float-vector 3)) &allow-other-keys* [method]

:calc-vel-from-pos *dif-pos translation-axis &rest args &key (p-limit 100.0) (tmp-v0 (instantiate float-vector 0)) (tmp-v1 (instantiate float-vector 1)) (tmp-v2 (instantiate float-vector 2)) (tmp-v3 (instantiate float-vector 3)) &allow-other-keys* [method]

:ik-convergence-check *success dif-pos dif-rot rotation-axis translation-axis thre rthre centroid-thre target-centroid-pos centroid-offset-func cog-translation-axis &key (update-mass-properties t)* [method]

:draw-collision-debug-view [method]

:inverse-kinematics-args *&rest args &key union-link-list rotation-axis translation-axis additional-jacobi-dimension &allow-other-keys* [method]

:calc-inverse-kinematics-nspace-from-link-list *link-list &key (avoid-nspace-gain 0.01) (union-link-list (send self :calc-union-link-list link-list)) (fik-len (send self :calc-target-joint-dimension union-link-list)) (null-space) (debug-view) (additional-nspace-list) (cog-gain 0.0) (target-centroid-pos) (centroid-offset-func) (cog-translation-axis :z) (cog-null-space nil) (weight (fill (instantiate float-vector fik-len) 1.0)) (update-mass-properties t) (tmp-nspace (instantiate float-vector fik-len))* [method]

:calc-nspace-from-joint-limit *avoid-nspace-gain union-link-list weight debug-view tmp-nspace* [method]

:calc-inverse-kinematics-weight-from-link-list *link-list &key (avoid-weight-gain 1.0) (union-link-list (send self :calc-union-link-list link-list)) (fik-len (send self :calc-target-joint-dimension union-link-list)) (weight (fill (instantiate float-vector fik-len) 1)) (additional-weight-list) (debug-view) (tmp-weight (instantiate float-vector fik-len)) (tmp-len (instantiate float-vector fik-len))* [method]

:calc-weight-from-joint-limit *avoid-weight-gain fik-len link-list union-link-list debug-view weight tmp-weight tmp-len* [method]

:reset-joint-angle-limit-weight-old *union-link-list* [method]

:find-joint-angle-limit-weight-old-from-union-link-list *union-link-list* [method]

:move-joints *union-vel &rest args &key union-link-list (periodic-time 0.05) (joint-args) (debug-view nil) (move-joints-hook) &allow-other-keys* [method]

:collision-avoidance *&rest args &key avoid-collision-distance avoid-collision-joint-gain avoid-collision-null-gain ((:collision-avoidance-link-pair pair-list)) (union-link-list) (link-list) (weight) (fik-len (send self :calc-target-joint-dimension union-link-list)) debug-view &allow-other-keys* [method]

:collision-avoidance-args *pair link-list* [method]

:collision-avoidance-calc-distance *&rest args &key union-link-list (warnp t) ((:collision-avoidance-link-pair pair-list)) ((:collision-distance-limit distance-limit) 10) &allow-other-keys* [method]

:collision-avoidance-link-pair-from-link-list *link-lists &key obstacles ((:collision-avoidance-links collision-links) collision-avoidance-links) debug* [method]

:collision-avoidance-links *&optional l* [method]

:calc-joint-angle-speed-gain *union-link-list dav periodic-time* [method]

:calc-joint-angle-speed *union-vel &rest args &key angle-speed (angle-speed-blending 0.5) jacobi jacobi# null-space i-j#j debug-view weight wmat tmp-len tmp-len2 fik-len &allow-other-keys* [method]

:calc-gradh-from-link-list *link-list &optional (res (instantiate float-vector (length link-list)))* [method]

:calc-inverse-jacobian *jacobi &rest args &key ((:manipulability-limit ml) 0.1) ((:manipulability-gain mg) 0.001) weight debug-view ret wmat tmat umat umat2 mat-tmp mat-tmp-rc tmp-mrr tmp-mrr2 &allow-other-keys* [method]

:calc-target-joint-dimension *link-list* [method]

:calc-union-link-list *link-list* [method]

:calc-target-axis-dimension *rotation-axis translation-axis* [method]

eusmodel-validity-check *robot* [function]

Check if the robot model is validate

calc-jacobian-default-rotate-vector *paxis world-default-coords child-reverse transform-coords tmp-v3 tmp-m33* [function]

calc-jacobian-rotational *fik row column joint paxis child-link world-default-coords child-reverse move-target transform-coords rotation-axis translation-axis tmp-v0 tmp-v1 tmp-v2 tmp-v3 tmp-v3a tmp-v3b tmp-m33* [function]

calc-jacobian-linear *fik row column joint paxis child-link world-default-coords child-reverse move-target transform-coords rotation-axis translation-axis tmp-v0 tmp-v1 tmp-v2 tmp-v3 tmp-v3a tmp-v3b tmp-m33* [function]

calc-angle-speed-gain-scalar *j dav i periodic-time* [function]

calc-angle-speed-gain-vector *j dav i periodic-time* [function]

all-child-links *s &optional (pred #'identity)* [function]

calc-dif-with-axis *dif axis &optional tmp-v0 tmp-v1 tmp-v2* [function]

calc-target-joint-dimension *joint-list* [function]

calc-joint-angle-min-max-for-limit-calculation *j kk jamm* [function]

joint-angle-limit-weight *j-l &optional (res (instantiate float-vector (calc-target-joint-dimension j-l)))* [function]

joint-angle-limit-nspace *j-l &optional (res (instantiate float-vector (calc-target-joint-dimension j-l)))* [function]

calc-jacobian-from-link-list-including-robot-and-obj-virtual-joint *link-list move-target obj-move-target robot &key (rotation-axis '(t t)) (translation-axis '(t t)) (fik (make-matrix (send robot :calc-target-axis-dimension rotation-axis translation-axis) (send robot :calc-target-joint-dimension link-list)))* [function]

append-obj-virtual-joint *link-list target-coords &key (joint-class 6dof-joint) (joint-args) (vplink) (vplink-coords) (vclink-coords)* [function]

append-multiple-obj-virtual-joint *link-list target-coords &key (joint-class '(6dof-joint)) (joint-args '(nil)) (vplink) (vplink-coords) (vclink-coords)* [function]

eusmodel-validity-check-one *robot* [function]

line [Class]

:super	**propertied-object**
:slots	pvert nvert

:worldcoords [method]

Return a coordinates on the midpoint of the end points

coordinates [Class]

:super	**propertied-object**
:slots	rot pos

:difference-rotation *coords &key (geometry::rotation-axis t)* [method]

return difference in rotation of given coords, rotation-axis can take (:x, :y, :z, :xx, :yy, :zz, :xm, :ym, :zm)

:difference-position *coords &key (geometry::translation-axis t)* [method]

return difference in position of given coords, translation-axis can take (:x, :y, :z, :xy, :yz, :zx).

:axis *geometry::axis* [method]

coordinates [Class]

:super	**propertied-object**
:slots	rot pos

:move-coords *geometry::target geometry::at* [method]

fix 'at' coords on 'self' to 'target'

:transform *geometry::c &optional (geometry::wrt :local)* [method]

:transformation *geometry::c2 &optional (geometry::wrt :local)* [method]

:move-to *geometry::c &optional (geometry::wrt :local) &aux geometry::cc* [method]

cascaded-coords [Class]

:super	**coordinates**
:slots	rot pos parent descendants worldcoords manager changed

:move-to *geometry::c &optional (geometry::wrt :local) &aux geometry::cc* [method]

:transform *geometry::c &optional (geometry::wrt :local)* [method]

:transformation *geometry::c2 &optional (geometry::wrt :local)* [method]

:worldcoords [method]

coordinates [Class]

:super	**propertied-object**
:slots	rot pos

:inverse-rotate-vector *geometry::v &optional geometry::r* [method]

:rotate-vector *geometry::v &optional geometry::r* [method]

cascaded-coords [Class]

:super	**coordinates**
:slots	rot pos parent descendants worldcoords manager changed

:inverse-rotate-vector *geometry::v &optional geometry::r* [method]

:rotate-vector *geometry::v &optional geometry::r* [method]

coordinates [Class]

:super	**propertied-object**
:slots	rot pos

:inverse-transform-vector *geometry::vec &optional geometry::v3a geometry::v3b geometry::m33* [method]

cascaded-coords [Class]

:super	**coordinates**
:slots	rot pos parent descendants worldcoords manager changed

:inverse-transform-vector *geometry::v &optional geometry::v3a geometry::v3b geometry::m33* [method]

bodyset [Class]

:super	**cascaded-coords**
:slots	(geometry::bodies :type cons)

:init [method]

coords &rest args &key (name (intern (format nil bodyset A (system:address self)) KEYWORD))
((:bodies geometry::bs))
&allow-other-keys

Create bodyset object

:bodies *&rest args* [method]

:faces [method]

:worldcoords [method]

:draw-on *&rest args* [method]

:init [method]

coords &rest args &key (name (intern (format nil bodyset A (system:address self)) KEYWORD))
((:bodies geometry::bs))
&allow-other-keys

Create bodyset object

:draw-on *&rest args* [method]

:worldcoords [method]

:faces [method]

:bodies *&rest args* [method]

midcoords *geometry::p geometry::c1 geometry::c2* [function]

Returns mid (or p) coordinates of given two cooridnates c1 and c2

orient-coords-to-axis *geometry::target-coords geometry::v &optional (geometry::axis :z) (geometry::eps *epsilon*)* [function]

orient 'axis' in 'target-coords' to the direction specified by 'v' destructively.
'v' must be non-zero vector.

geometry::face-to-triangle-aux *geometry::f* [function]

triangulate the face.

geometry::face-to-triangle *geometry::f* [function]

convert face to set of triangles.

geometry::face-to-tessel-triangle *geometry::f geometry::num &optional (*epsilon*1.000000e-10)* [function]

return polygon if triangable, return nil if it is not.

body-to-faces *geometry::abody* [function]

return triangled faces of given body

make-sphere *geometry::r &rest args* [function]

make sphere of given r

make-ring *geometry::ring-radius geometry::pipe-radius &rest args &key (geometry::segments 16)* [function]

make ring of given ring and pipe radius

make-fan-cylinder [function]

geometry::radius geometry::height &rest args &key (geometry::segments 12)
(angle 2pi)
(geometry::mid-angle (/ angle 2.0))

make a cylinder whose base face is a fan. the angle of fan
is defined by :angle keyword. and, the csg of the returned body is
(:cylinder radius height segments angle)

x-of-cube *geometry::cub* [function]

return x of cube.

y-of-cube *geometry::cub* [function]

return y of cube.

z-of-cube *geometry::cub* [function]

return z of cube.

height-of-cylinder *geometry::cyl* [function]

return height of cylinder.

radius-of-cylinder *geometry::cyl* [function]

return radius of cylinder.

radius-of-sphere *geometry::sp* [function]

return radius of shape.

geometry::make-faceset-from-vertices *geometry::vs* [function]

create faceset from vertices.

matrix-to-euler-angle *geometry::m geometry::axis-order* [function]

return euler angle from matrix.

geometry::quaternion-from-two-vectors *geometry::a geometry::b* [function]

Comupute quaternion which rotate vector a into b.

transform-coords *geometry::c1 geometry::c2 &optional (geometry::c3 (let ((geometry::dim (send geometry::c1 :dimension))) (instance coordinates :newcoords (unit-matrix geometry::dim) (instantiate float-vector geometry::dim))))* [function]

geometry::face-to-triangle-rest-polygon *geometry::f geometry::num geometry::edgs* [function]

geometry::face-to-triangle-make-simple *geometry::f* [function]

body-to-triangles *geometry::abody &optional (geometry::limit 50)* [function]

geometry::triangle-to-triangle *geometry::aface &optional (geometry::limit 50)* [function]

robot-model [Class]

:super **cascaded-link**

:slots larm-end-coords rarm-end-coords lleg-end-coords rleg-end-coords head-end-coords tor

:camera *sensor-name* [method]

Returns camera with given name

:force-sensor *sensor-name* [method]

Returns force sensor with given name

:imu-sensor *sensor-name* [method]

Returns imu sensor of given name

:force-sensors [method]

Returns force sensors.

:imu-sensors [method]

Returns imu sensors.

:cameras [method]

Returns camera sensors.

:look-at-hand *l/r* [method]

look at hand position, l/r supports :rarm, :larm, :arms, and '(:rarm :larm)

:inverse-kinematics [method]

target-coords &rest args &key look-at-target
(move-target)
(link-list (if (atom move-target) (send self :link-list (send move-target :parent)) (mapcar #'
&allow-other-keys

solve inverse kinematics, move move-target to target-coords
target-coords and move-target should be given.
link-list is set by default based on move-target ->root link link-list
:look-at-target
target head looks at. This task is best-effort and only head joints are used. :look-at-target supports t, nil, float-vector, coords, list of float-vector, list of coords.
other-keys
:inverse-kinematics internally calls :inverse-kinematics of cascaded-cords class and args are passed to it. See the explanation of :inverse-kinematics of cascaded-cords class.

:inverse-kinematics-loop [method]

dif-pos dif-rot &rest args &key target-coords
debug-view
look-at-target
(move-target)
(link-list (if (atom move-target) (send self :link-list (send move-target :parent)) (mapc
&allow-other-keys

move move-target using dif-pos and dif-rot,
look-at-target suppots t, nil, float-vector, coords, list of float-vector, list of coords
link-list is set by default based on move-target ->root link link-list
:inverse-kinematics-loop internally calls :inverse-kinematics-loop function of cascaded-coords class. See the explanation of :inverse-kinematics-loop in cascaded-coords class.

:look-at-target *look-at-target &key (target-coords)* [method]

move robot head to look at targets, look-at-target support t/nil float-vector coordinates, center of list of float-vector or list of coordinates

:init-pose [method]

Set robot to initial posture.

:torque-vector [method]

&key (force-list)
(moment-list)
(target-coords)
(debug-view nil)
(calc-statics-p t)
(dt 0.005)
(av (send self :angle-vector))
(root-coords (send (car (send self :links)) :copy-worldcoords))
(calc-torque-buffer-args (send self :calc-torque-buffer-args))
(distribute-total-wrench-to-torque-method (if (and (not (every #'null (send self :legs))) (not (and

Returns torque vector

:calc-force-from-joint-torque [method]

limb all-torque &key (move-target (send self limb :end-coords))
(use-torso)

Calculates end-effector force and moment from joint torques.

:fullbody-inverse-kinematics [method]

target-coords &rest args &key (move-target)
(link-list)
(min (float-vector -500 -500 -500 -20 -20 -10))
(max (float-vector 500 500 25 20 20 10))
(root-link-virtual-joint-weight #f(0.1 0.1 0.1 0.1 0.5 0.5))
(target-centroid-pos (apply #'midpoint 0.5 (send self :legs :end-coords :worldpos)
(cog-gain 1.0)
(cog-translation-axis :z)
(centroid-offset-func nil)
(centroid-thre 5.0)
(additional-weight-list)
(joint-args nil)
(cog-null-space nil)
(min-loop 2)
&allow-other-keys

fullbody inverse kinematics for legged robot.
necessary args : target-coords, move-target, and link-list must include legs' (or leg's) parameters
ex. (send *robot*:fullbody-inverse-kinematics (list rarm-tc rleg-tc lleg-tc) :move-target (list rarm-mt rleg-mt lleg-mt) :link-list (list rarm-ll rleg-ll lleg-ll))
:min
lower position limit of root link virtual joint (x y z roll pitch yaw). Default is #f(-500[mm] -500[mm] -500[mm] -20[deg] -20[deg] -10[deg]).
:max
upper position limit of root link virtual joint (x y z roll pitch yaw). Default is #f(500[mm] 500[mm] 25[mm] 20[deg] 20[deg] 10[deg]).
:root-link-virtual-joint-weight
float-vector of inverse weight of velocity of root link virtual joint or a function which returns the float-vector (x y z roll pitch yaw). This works in the same way as :additional-weight-list in cascaded-coords::inverse-kinematics. Default is #f(0.1 0.1 0.1 0.1 0.5 0.5).
:joint-args
list of other arguments passed to :init function of root link virtual joint (6dof-joint class).
:max-joint-velocity
limit of velocity of root link virtual joint (x y z roll pitch yaw). Default is #f((/ 0.08 0.05)[m/s] (/ 0.08 0.05)[m/s] (/ 0.08 0.05)[m/s] (/ pi 4)[rad/s] (/ pi 4)[rad/s] (/ pi 4)[rad/s]))
other-keys
:fullbody-inverse-kinematics internally calls :inverse-kinematics and args are passed to it. See the explanation of :inverse-kinematics.

:print-vector-for-robot-limb *vec* [method]

Print angle vector with limb alingment and limb indent.
For example, if robot is rarm, larm, and torso, print result is:

#f(
rarm-j0 ... rarm-jN
larm-j0 ... larm-jN
torso-j0 ... torso-jN
)

:calc-zmp-from-forces-moments [method]

forces moments &key (wrt :world)
(limbs (if (send self :force-sensors) (remove nil (mapcar #'(lambda (fs) (find-i
(force-sensors (mapcar #'(lambda (l) (send self :force-sensor l)) limbs))
(cop-coords (mapcar #'(lambda (l) (send self l :end-coords)) limbs))
(fz-thre 0.001)
(limb-cop-fz-list (mapcar #'(lambda (fs f m cc) (let ((fsp (scale 0.001 (send fs

Calculate zmp[mm] from sensor local forces and moments
If force_z is large, zmp can be defined and returns 3D zmp.
Otherwise, zmp cannot be defined and returns nil.

:foot-midcoords *&optional (mid 0.5)* [method]

Calculate midcoords of :rleg and :lleg end-coords.
In the following codes, leged robot is assumed.

:fix-leg-to-coords [method]

fix-coords &optional (l/r :both) &key (mid 0.5)
&allow-other-keys

Fix robot's legs to a coords
In the following codes, leged robot is assumed.

:move-centroid-on-foot [method]

leg fix-limbs &rest args &key (thre (mapcar #'(lambda (x) (if (memq x '(:rleg :lleg)) 1 5
(rthre (mapcar #'(lambda (x) (deg2rad (if (memq x '(:rleg :lleg)) 1 5))) fix-limbs))
(mid 0.5)
(target-centroid-pos (if (eq leg :both) (apply #'midpoint mid (mapcar #'(lambda (tmp
(fix-limbs-target-coords (mapcar #'(lambda (x) (send self x :end-coords :copy-worldcoo
(root-link-virtual-joint-weight #f(0.1 0.1 0.0 0.0 0.0 0.5))
&allow-other-keys

Move robot COG to change centroid-on-foot location,
leg : legs for target of robot's centroid, which should be :both, :rleg, and :lleg.
fix-limbs : limb names which are fixed in this IK.

:calc-walk-pattern-from-footstep-list [method]

footstep-list &key (default-step-height 50)
(dt 0.1)
(default-step-time 1.0)
(solve-angle-vector-args)
(debug-view nil)
((:all-limbs al) (if (send self :force-sensors) (remove nil (mapcar #'(lamb
((:default-zmp-offsets dzo) (mapcan #'(lambda (x) (list x (float-vector 0
(init-pose-function #'(lambda nil (send self :move-centroid-on-foot :both

(start-with-double-support t)
(end-with-double-support t)
(ik-thre 1)
(ik-rthre (deg2rad 1))
(q 1.0)
(r 1.000000e-06)
(calc-zmp t)

Calculate walking pattern from foot step list and return pattern list as a list of angle-vector, root-coords, time, and so on.
footstep-list should be given.
:footstep-list
(list footstep1 footstep2 ...). :footstep-list can be any length. Each footstep indicates the destinations of swing legs in each step.
footstep should be list of coordinate whose :name is identical with one swing leg and whose coords is the destination of that leg. If number of swing legs in a step is one, the footstep can be a coordinate.
footstep1 is only for intialization and not executed.
:default-step-height
Height of swing leg cycloid trajectories. Default is 50[mm].
:dt
Sampling time of preview control and output walk pattern. Default is 0.1[s].
:default-step-time
Reference time of each step. The first 10 percent and the last 10 percent of default-step-time is double support phase. Default is 1.0[s].
:solve-angle-vector-args
:move-centroid-on-foot is used to solve IK in :calc-walk-pattern-from-footstep-list. :solve-angle-vector-args is passed to :move-centroid-on-foot in the form of (send*self :move-centroid-on-foot ... solve-angle-vector-args). Default is nil.
:debug-view
Set t to show visualization. Default is nil.
:all-limbs
list of limb names. In each walking step, limbs in :all-limbs but not assigned as swing legs by :footstep-list are considered to be support legs. Default is '(:rleg :lleg) sorted in force-sensors order.
:default-zmp-offsets
(list limbname1 offset1 limbname2 offset2 ...). :default-zmp-offsets should include every limb in :all-limbs. offset is a float-vector[mm] and local offset of reference zmp position from end-coords. Default offset is #F(0 0 0)[mm].
:init-pose-function
A function which initialize robot's pose. Walking pattern is generated from this initial pose. :init-pose-function is called once at the start of walking pattern generation in the form of (funcall init-pose-function). Default is #'(lambda () (send self :move-centroid-on-foot :both '(:rleg :lleg))).
:start-with-double-support
At the start of walking pattern generation, the initial position of reference zmp is
t: midpoint of all-limbs.
nil: midpoint of swing legs of footstep1.
Default is t.
:end-with-double-support

At the end of walking pattern generation, the final position of reference zmp is
t: midpoint of all-limbs.
nil: midpoint of support legs of the last footstep.
Default is t.

:ik-thre

Threshold for position error to terminate IK iteration. Default is 1[mm].

:ik-rthre

Threshold for rotation error to terminate IK iteration. Default is (deg2rad 1)[rad].

:q

Weight Q of the cost function of preview control. Default is 1.0.

:r

Weight R of the cost function of preview control. Default is 1e-6.

:calc-zmp

Set t to calculate resultant ZMP after IK. The calculated ZMP is visualized if :debug-view is t, and stored as czmp in return value. Default is t.

:gen-footstep-parameter *&key (ratio 1.0)* [method]

Generate footstep parameter

:go-pos-params->footstep-list [method]

xx yy th &key ((:footstep-parameter prm) (send self :footstep-parameter))
((:default-half-offset defp) (cadr (memq :default-half-offset prm)))
((:forward-offset-length xx-max) (cadr (memq :forward-offset-length prm)))
((:outside-offset-length yy-max) (cadr (memq :outside-offset-length prm)))
((:rotate-rad th-max) (abs (rad2deg (cadr (memq :rotate-rad prm)))))
(gen-go-pos-step-node-func #'(lambda (mc leg leg-translate-pos) (let ((cc (send

Calculate foot step list from goal x position [mm], goal y position [mm], and goal yaw orientation [deg].

:go-pos-quadruped-params->footstep-list *xx yy th &key (type :crawl)* [method]

Calculate foot step list for quadruped walking from goal x position [mm], goal y position [mm], and goal yaw orientation [deg].

:support-polygons [method]

Return support polygons.

:support-polygon *name* [method]

Return support polygon.
If name is list, return convex hull of all polygons.
Otherwise, return polygon with given name

:make-default-linear-link-joint-between-attach-coords *attach-coords-0 attach-coords-1 end-coords-name linear-joint-name* [method]

Make default linear arctuator module such as muscle and cylinder and append lins and joint-list.
Module includes parent-link =>(j0) =>l0 =>(j1) =>l1 (linear actuator) =>(j2) =>l2 =>end-coords.
attach-coords-0 is root side coords which linear actulator is attached to.
attach-coords-1 is end side coords which linear actulator is attached to.
end-coords-name is the name of end-coords.
linear-joint-name is the name of linear actuator.

:calc-static-balance-point [method]

&key (target-points (mapcar #'(lambda (tmp-arm) (send (send self tmp-arm :end-co
(force-list (make-list (length target-points) :initial-element (float-vector 0 0 0)))
(moment-list (make-list (length target-points) :initial-element (float-vector 0 0 0)))
(static-balance-point-height (elt (apply #'midpoint 0.5 (send self :legs :end-coords :w
(update-mass-properties t)

Calculate static balance point which is equivalent to static extended ZMP.
The output is expressed by the world coordinates.
target-points are end-effector points on which force-list and moment-list apply.
force-list [N] and moment-list [Nm] are list of force and moment that robot receives at target-points.
static-balance-point-height is height of static balance point [mm].

:init-ending [method]

:rarm-end-coords [method]

:larm-end-coords [method]

:rleg-end-coords [method]

:lleg-end-coords [method]

:head-end-coords [method]

:torso-end-coords [method]

:rarm-root-link [method]

:larm-root-link [method]

:rleg-root-link [method]

:lleg-root-link [method]

:head-root-link [method]

:torso-root-link [method]

:limb *limb method &rest args* [method]

:inverse-kinematics-loop-for-look-at *limb &rest args* [method]

:gripper *limb &rest args* [method]

:get-sensor-method *sensor-type sensor-name* [method]

:get-sensors-method-by-limb *sensors-type limb* [method]

:larm *&rest args* [method]

:rarm *&rest args* [method]

:lleg *&rest args* [method]

:rleg *&rest args* [method]

:head *&rest args* [method]

:torso *&rest args* [method]

:arms *&rest args* [method]

:legs *&rest args* [method]

:distribute-total-wrench-to-torque-method-default [method]

:joint-angle-limit-nspace-for-6dof *&key (avoid-nspace-gain 0.01) (limbs '(:rleg :lleg))* [method]

:joint-order *limb &optional jname-list* [method]

:draw-gg-debug-view *end-coords-list contact-state rz cog pz czmp dt* [method]

:footstep-parameter [method]

:make-support-polygons [method]

:make-sole-polygon *name* [method]

:calc-zmp-from-forces-moments [method]

forces moments &key (wrt :world)
(limbs (if (send self :force-sensors) (remove nil (mapcar #'(lambda (fs) (find-i
(force-sensors (mapcar #'(lambda (l) (send self :force-sensor l)) limbs))
(cop-coords (mapcar #'(lambda (l) (send self l :end-coords)) limbs))
(fz-thre 0.001)
(limb-cop-fz-list (mapcar #'(lambda (fs f m cc) (let ((fsp (scale 0.001 (send fs

Calculate zmp[mm] from sensor local forces and moments
If force_z is large, zmp can be defined and returns 3D zmp.
Otherwise, zmp cannot be defined and returns nil.

:print-vector-for-robot-limb *vec* [method]

Print angle vector with limb alingment and limb indent.
For example, if robot is rarm, larm, and torso, print result is:
#f(
rarm-j0 ... rarm-jN
larm-j0 ... larm-jN
torso-j0 ... torso-jN
)

:fullbody-inverse-kinematics [method]

target-coords &rest args &key (move-target)

(link-list)
(min (float-vector -500 -500 -500 -20 -20 -10))
(max (float-vector 500 500 25 20 20 10))
(root-link-virtual-joint-weight #f(0.1 0.1 0.1 0.1 0.5 0.5))
(target-centroid-pos (apply #'midpoint 0.5 (send self :legs :end-coords :worldpos)
(cog-gain 1.0)
(cog-translation-axis :z)
(centroid-offset-func nil)
(centroid-thre 5.0)
(additional-weight-list)
(joint-args nil)
(cog-null-space nil)
(min-loop 2)
&allow-other-keys

fullbody inverse kinematics for legged robot.
necessary args : target-coords, move-target, and link-list must include legs' (or leg's) parameters
ex. (send ∗robot∗:fullbody-inverse-kinematics (list rarm-tc rleg-tc lleg-tc) :move-target (list rarm-mt rleg-mt lleg-mt) :link-list (list rarm-ll rleg-ll lleg-ll))
:min
lower position limit of root link virtual joint (x y z roll pitch yaw). Default is #f(-500[mm] -500[mm] -500[mm] -20[deg] -20[deg] -10[deg]).
:max
upper position limit of root link virtual joint (x y z roll pitch yaw). Default is #f(500[mm] 500[mm] 25[mm] 20[deg] 20[deg] 10[deg]).
:root-link-virtual-joint-weight
float-vector of inverse weight of velocity of root link virtual joint or a function which returns the float-vector (x y z roll pitch yaw). This works in the same way as :additional-weight-list in cascaded-coords::inverse-kinematics. Default is #f(0.1 0.1 0.1 0.1 0.5 0.5).
:joint-args
list of other arguments passed to :init function of root link virtual joint (6dof-joint class).
:max-joint-velocity
limit of velocity of root link virtual joint (x y z roll pitch yaw). Default is #f((/ 0.08 0.05)[m/s] (/ 0.08 0.05)[m/s] (/ 0.08 0.05)[m/s] (/ pi 4)[rad/s] (/ pi 4)[rad/s] (/ pi 4)[rad/s]))
other-keys
:fullbody-inverse-kinematics internally calls :inverse-kinematics and args are passed to it. See the explanation of :inverse-kinematics.

:calc-force-from-joint-torque [method]

limb all-torque &key (move-target (send self limb :end-coords))
(use-torso)

Calculates end-effector force and moment from joint torques.

:torque-vector [method]

&key (force-list)
(moment-list)
(target-coords)

(debug-view nil)
(calc-statics-p t)
(dt 0.005)
(av (send self :angle-vector))
(root-coords (send (car (send self :links)) :copy-worldcoords))
(calc-torque-buffer-args (send self :calc-torque-buffer-args))
(distribute-total-wrench-to-torque-method (if (and (not (every #'null (send self :legs))) (not (and

Returns torque vector

:init-pose [method]

Set robot to initial posture.

:look-at-target *look-at-target &key (target-coords)* [method]

move robot head to look at targets, look-at-target support t/nil float-vector coordinates, center of list of float-vector or list of coordinates

:inverse-kinematics-loop [method]

dif-pos dif-rot &rest args &key target-coords
debug-view
look-at-target
(move-target)
(link-list (if (atom move-target) (send self :link-list (send move-target :parent)) (mapc
&allow-other-keys

move move-target using dif-pos and dif-rot,
look-at-target suppots t, nil, float-vector, coords, list of float-vector, list of coords
link-list is set by default based on move-target ->root link link-list
:inverse-kinematics-loop internally calls :inverse-kinematics-loop function of cascaded-coords class. See the explanation of :inverse-kinematics-loop in cascaded-coords class.

:inverse-kinematics [method]

target-coords &rest args &key look-at-target
(move-target)
(link-list (if (atom move-target) (send self :link-list (send move-target :parent)) (mapcar #'
&allow-other-keys

solve inverse kinematics, move move-target to target-coords
target-coords and move-target should be given.
link-list is set by default based on move-target ->root link link-list
:look-at-target
target head looks at. This task is best-effort and only head joints are used. :look-at-target supports t, nil, float-vector, coords, list of float-vector, list of coords.
other-keys
:inverse-kinematics internally calls :inverse-kinematics of cascaded-cords class and args are passed to it. See the explanation of :inverse-kinematics of cascaded-cords class.

:look-at-hand *l/r* [method]

look at hand position, l/r supports :rarm, :larm, :arms, and '(:rarm :larm)

:cameras [method]

Returns camera sensors.

:imu-sensors [method]

Returns imu sensors.

:force-sensors [method]

Returns force sensors.

:imu-sensor *sensor-name* [method]

Returns imu sensor of given name

:force-sensor *sensor-name* [method]

Returns force sensor with given name

:camera *sensor-name* [method]

Returns camera with given name

:joint-order *limb &optional jname-list* [method]

:joint-angle-limit-nspace-for-6dof *&key (avoid-nspace-gain 0.01) (limbs '(:rleg :lleg))* [method]

:distribute-total-wrench-to-torque-method-default [method]

:legs *&rest args* [method]

:arms *&rest args* [method]

:torso *&rest args* [method]

:head *&rest args* [method]

:rleg *&rest args* [method]

:lleg *&rest args* [method]

:rarm *&rest args* [method]

:larm *&rest args* [method]

:get-sensors-method-by-limb *sensors-type limb* [method]

:get-sensor-method *sensor-type sensor-name* [method]

:gripper *limb &rest args* [method]

:inverse-kinematics-loop-for-look-at *limb &rest args* [method]

:limb *limb method &rest args* [method]

:torso-root-link [method]

:head-root-link [method]

:lleg-root-link [method]

:rleg-root-link [method]

:larm-root-link [method]

:rarm-root-link [method]

:torso-end-coords [method]

:head-end-coords [method]

:lleg-end-coords [method]

:rleg-end-coords [method]

:larm-end-coords [method]

:rarm-end-coords [method]

:init-ending [method]

:calc-static-balance-point [method]

&key (target-points (mapcar #'(lambda (tmp-arm) (send (send self tmp-arm :end-co
(force-list (make-list (length target-points) :initial-element (float-vector 0 0 0)))
(moment-list (make-list (length target-points) :initial-element (float-vector 0 0 0)))
(static-balance-point-height (elt (apply #'midpoint 0.5 (send self :legs :end-coords :w
(update-mass-properties t)

Calculate static balance point which is equivalent to static extended ZMP.
The output is expressed by the world coordinates.
target-points are end-effector points on which force-list and moment-list apply.
force-list [N] and moment-list [Nm] are list of force and moment that robot receives at target-points.
static-balance-point-height is height of static balance point [mm].

:make-default-linear-link-joint-between-attach-coords *attach-coords-0 attach-coords-1 end-coords-name linear-joint-name* [method]

Make default linear arctuator module such as muscle and cylinder and append lins and joint-list.
Module includes parent-link =>(j0) =>l0 =>(j1) =>l1 (linear actuator) =>(j2) =>l2 =>end-coords.
attach-coords-0 is root side coords which linear actulator is attached to.
attach-coords-1 is end side coords which linear actulator is attached to.
end-coords-name is the name of end-coords.
linear-joint-name is the name of linear actuator.

:support-polygon *name* [method]

Return support polygon.
If name is list, return convex hull of all polygons.
Otherwise, return polygon with given name

:support-polygons [method]

Return support polygons.

:go-pos-quadruped-params->footstep-list *xx yy th &key (type :crawl)* [method]

Calculate foot step list for quadruped walking from goal x position [mm], goal y position [mm], and goal yaw orientation [deg].

:go-pos-params->footstep-list [method]

xx yy th &key ((:footstep-parameter prm) (send self :footstep-parameter))
((:default-half-offset defp) (cadr (memq :default-half-offset prm)))
((:forward-offset-length xx-max) (cadr (memq :forward-offset-length prm)))
((:outside-offset-length yy-max) (cadr (memq :outside-offset-length prm)))
((:rotate-rad th-max) (abs (rad2deg (cadr (memq :rotate-rad prm)))))
(gen-go-pos-step-node-func #'(lambda (mc leg leg-translate-pos) (let ((cc (send

Calculate foot step list from goal x position [mm], goal y position [mm], and goal yaw orientation [deg].

:gen-footstep-parameter *&key (ratio 1.0)* [method]

Generate footstep parameter

:calc-walk-pattern-from-footstep-list [method]

footstep-list &key (default-step-height 50)
(dt 0.1)
(default-step-time 1.0)
(solve-angle-vector-args)
(debug-view nil)
((:all-limbs al) (if (send self :force-sensors) (remove nil (mapcar #'(lamb
((:default-zmp-offsets dzo) (mapcan #'(lambda (x) (list x (float-vector 0
(init-pose-function #'(lambda nil (send self :move-centroid-on-foot :both
(start-with-double-support t)
(end-with-double-support t)
(ik-thre 1)
(ik-rthre (deg2rad 1))
(q 1.0)
(r 1.000000e-06)
(calc-zmp t)

Calculate walking pattern from foot step list and return pattern list as a list of angle-vector, root-coords, time, and so on.
footstep-list should be given.
:footstep-list
(list footstep1 footstep2 ...). :footstep-list can be any length. Each footstep indicates the destinations of swing legs in each step.
footstep should be list of coordinate whose :name is identical with one swing leg and whose coords is the destination of that leg. If number of swing legs in a step is one, the footstep can be a coordinate.
footstep1 is only for intialization and not executed.
:default-step-height
Height of swing leg cycloid trajectories. Default is 50[mm].
:dt
Sampling time of preview control and output walk pattern. Default is 0.1[s].
:default-step-time
Reference time of each step. The first 10 percent and the last 10 percent of default-step-time is double support phase. Default is 1.0[s].

:solve-angle-vector-args

:move-centroid-on-foot is used to solve IK in :calc-walk-pattern-from-footstep-list. :solve-angle-vector-args is passed to :move-centroid-on-foot in the form of (send∗self :move-centroid-on-foot ... solve-angle-vector-args). Default is nil.

:debug-view

Set t to show visualization. Default is nil.

:all-limbs

list of limb names. In each walking step, limbs in :all-limbs but not assigned as swing legs by :footstep-list are considered to be support legs. Default is '(:rleg :lleg) sorted in force-sensors order.

:default-zmp-offsets

(list limbname1 offset1 limbname2 offset2 ...). :default-zmp-offsets should include every limb in :all-limbs. offset is a float-vector[mm] and local offset of reference zmp position from end-coords. Default offset is #F(0 0 0)[mm].

:init-pose-function

A function which initialize robot's pose. Walking pattern is generated from this initial pose. :init-pose-function is called once at the start of walking pattern generation in the form of (funcall init-pose-function). Default is #'(lambda () (send self :move-centroid-on-foot :both '(:rleg :lleg))).

:start-with-double-support

At the start of walking pattern generation, the initial position of reference zmp is

t: midpoint of all-limbs.

nil: midpoint of swing legs of footstep1.

Default is t.

:end-with-double-support

At the end of walking pattern generation, the final position of reference zmp is

t: midpoint of all-limbs.

nil: midpoint of support legs of the last footstep.

Default is t.

:ik-thre

Threshold for position error to terminate IK iteration. Default is 1[mm].

:ik-rthre

Threshold for rotation error to terminate IK iteration. Default is (deg2rad 1)[rad].

:q

Weight Q of the cost function of preview control. Default is 1.0.

:r

Weight R of the cost function of preview control. Default is 1e-6.

:calc-zmp

Set t to calculate resultant ZMP after IK. The calculated ZMP is visualized if :debug-view is t, and stored as czmp in return value. Default is t.

:move-centroid-on-foot [method]

leg fix-limbs &rest args &key (thre (mapcar #'(lambda (x) (if (memq x '(:rleg :lleg)) 1 5
(rthre (mapcar #'(lambda (x) (deg2rad (if (memq x '(:rleg :lleg)) 1 5))) fix-limbs))
(mid 0.5)
(target-centroid-pos (if (eq leg :both) (apply #'midpoint mid (mapcar #'(lambda (tmp
(fix-limbs-target-coords (mapcar #'(lambda (x) (send self x :end-coords :copy-worldcoo
(root-link-virtual-joint-weight #f(0.1 0.1 0.0 0.0 0.0 0.5))

&allow-other-keys

Move robot COG to change centroid-on-foot location,
leg : legs for target of robot's centroid, which should be :both, :rleg, and :lleg.
fix-limbs : limb names which are fixed in this IK.

:fix-leg-to-coords [method]
fix-coords &optional (l/r :both) &key (mid 0.5)
&allow-other-keys

Fix robot's legs to a coords
In the following codes, leged robot is assumed.

:foot-midcoords *&optional (mid 0.5)* [method]

Calculate midcoords of :rleg and :lleg end-coords.
In the following codes, leged robot is assumed.

:make-sole-polygon *name* [method]

:make-support-polygons [method]

:footstep-parameter [method]

:draw-gg-debug-view *end-coords-list contact-state rz cog pz czmp dt* [method]
make-default-robot-link *len radius axis name &optional extbody* [function]

18.5 Sensor Model

sensor-model [Class]

:super	**body**
:slots	data profile

:profile *&optional p* [method]

:signal *rawinfo* [method]

:simulate *model* [method]

:read [method]

:draw-sensor *v* [method]

:init *shape &key name &allow-other-keys* [method]
:init *shape &key name &allow-other-keys* [method]

:draw-sensor *v* [method]

:read [method]

:simulate *model* [method]

:signal *rawinfo* [method]

:profile *&optional p* [method]

bumper-model [Class]

:super **sensor-model**
:slots bumper-threshold

:init [method]

b &rest args &key ((:bumper-threshold bt) 20)
name

Create bumper model, b is the shape of an object and bt is the threshold in distance[mm].

:simulate *objs* [method]

Simulate bumper, with given objects, return 1 if the sensor detects an object and 0 if not.

:draw *vwer* [method]

:draw-sensor *vwer* [method]

:simulate *objs* [method]

Simulate bumper, with given objects, return 1 if the sensor detects an object and 0 if not.

:init [method]

b &rest args &key ((:bumper-threshold bt) 20)
name

Create bumper model, b is the shape of an object and bt is the threshold in distance[mm].

:draw-sensor *vwer* [method]

:draw *vwer* [method]

camera-model [Class]

:super **sensor-model**
:slots (vwing :forward (:projection :newprojection :screen :view :viewpoint :view-direction :v

:init [method]

b &rest args &key ((:width pw) 320)
((:height ph) 240)
(view-up #f(0.0 1.0 0.0))
(viewdistance 100.0)
(hither 100.0)
(yon 10000.0)
&allow-other-keys

Create camera model. b is the shape of an object

:create-viewer *&optional cv &key (no-window nil)* [method]

Create camera viewer, or set viewer

:width [method]

Returns width of the camera in pixel.

:height [method]

Returns height of the camera in pixel.

:fovy [method]

Returns field of view in degree

:cx [method]

Returns center x.

:cy [method]

Returns center y.

:fx [method]

Returns focal length of x.

:fy [method]

Returns focal length of y.

:screen-point *pos* [method]

Returns point in screen corresponds to the given pos.

:3d-point *x y d* [method]

Returns 3d position

:ray *x y* [method]

Returns ray vector of given x and y.

:draw-on [method]

&rest args &key ((:viewer vwer) *viewer*)
&allow-other-keys

Draw camera raw in irtviewer, ex (send cam :draw-on :flush t)

:draw-objects *objs* [method]

Draw objects in camera viewer, expected type of objs is list of objects

:get-image [method]

&key (with-points)
(with-colors)

Get image objects you need to call :draw-objects before calling this function

:select-drawmode *mode objs* [method]

Change drawmode for drawing with :draw-objects methods. mode is symbol of mode, 'hid is symbol for hidden line mode, the other symbols indicate default mode. objs is the same objects using :draw-objects.

:viewing *&rest args* [method]

:image-viewer *&rest args* [method]

:draw-sensor *vwer &key flush (width 1) (color (float-vector 1 1 1))* [method]

:draw-objects-raw *vwr objs* [method]

:get-image-raw *vwr &key (points) (colors)* [method]

:select-drawmode *mode objs* [method]

Change drawmode for drawing with :draw-objects methods. mode is symbol of mode, 'hid is symbol for hidden line mode, the other symbols indicate default mode. objs is the same objects using :draw-objects.

:get-image [method]

&key (with-points)
(with-colors)

Get image objects you need to call :draw-objects before calling this function

:draw-objects *objs* [method]

Draw objects in camera viewer, expected type of objs is list of objects

:draw-on [method]

&rest args &key ((:viewer vwer) *viewer*)
&allow-other-keys

Draw camera raw in irtviewer, ex (send cam :draw-on :flush t)

:ray *x y* [method]

Returns ray vector of given x and y.

:3d-point *x y d* [method]

Returns 3d position

:screen-point *pos* [method]

Returns point in screen corresponds to the given pos.

:fy [method]

Returns focal length of y.

:fx [method]

Returns focal length of x.

:cy [method]

Returns center y.

:cx [method]

Returns center x.

:fovy [method]

Returns field of view in degree

:height [method]

Returns height of the camera in pixel.

:width [method]

Returns width of the camera in pixel.

:create-viewer *&optional cv &key (no-window nil)* [method]

Create camera viewer, or set viewer

:init [method]

b &rest args &key ((:width pw) 320)
((:height ph) 240)
(view-up #f(0.0 1.0 0.0))
(viewdistance 100.0)
(hither 100.0)
(yon 10000.0)
&allow-other-keys

Create camera model. b is the shape of an object

:get-image-raw *vwr &key (points) (colors)* [method]

:draw-objects-raw *vwr objs* [method]

:draw-sensor *vwer &key flush (width 1) (color (float-vector 1 1 1))* [method]

:image-viewer *&rest args* [method]

:viewing *&rest args* [method]

make-camera-from-param [function]

&key pwidth
pheight
fx
fy
cx
cy
(tx 0)
(ty 0)
parent-coords
name
create-viewer
(no-window nil)

Create camera object from given parameters.

18.6 Environment Model

scene-model [Class]

:super	**cascaded-coords**
:slots	name objs

:init [method]

&rest args &key ((:name n) scene)
((:objects o))
((:remove-wall w))
&allow-other-keys

Create scene model

:objects [method]

Returns objects in the scene.

:add-objects *objects* [method]

Add objects to scene with identifiable names. Returns all objects.

:add-object *obj* [method]

Add object to scene with identifiable name. Returns all objects.

:remove-objects *objs-or-names* [method]

Remove objects or objects with given names from scene. Returns removed objects.

:remove-object *obj-or-name* [method]

Remove object or object with given name from scene. Returns removed object.

:find-object *name* [method]

Returns objects with given name.

:add-spots *spots* [method]

Add spots to scene with identifiable names. All spots will be :assoc with this scene. Returns T if added spots successfly, otherwise returns NIL.

:add-spot *spot* [method]

Add spot to scene with identifiable name. The spot will be :assoc with this scene. Returns T if spot is added successfly, otherwise returns NIL.

:remove-spots *spots* [method]

Remove spots from this scene. All spots will be :dissoc with this scene. Returns removed spots.

:remove-spot *spot* [method]

Remove spot from scene. the spot will be :dissoc with this scene. Returns removed spot.

:spots [method]

Return all spots in the scene.

:object *name* [method]

Return an object of given name.

:spot *name* [method]

Return a spot of given name.

:bodies [method]

:spot *name* [method]

Return a spot of given name.

:object *name* [method]

Return an object of given name.

:spots [method]

Return all spots in the scene.

:remove-spot *spot* [method]

Remove spot from scene. the spot will be :dissoc with this scene. Returns removed spot.

:remove-spots *spots* [method]

Remove spots from this scene. All spots will be :dissoc with this scene. Returns removed spots.

:add-spot *spot* [method]

Add spot to scene with identifiable name. The spot will be :assoc with this scene. Returns T if spot is added successfly, otherwise returns NIL.

:add-spots *spots* [method]

Add spots to scene with identifiable names. All spots will be :assoc with this scene. Returns T if added spots successfly, otherwise returns NIL.

:find-object *name* [method]

Returns objects with given name.

:remove-object *obj-or-name* [method]

Remove object or object with given name from scene. Returns removed object.

:remove-objects *objs-or-names* [method]

Remove objects or objects with given names from scene. Returns removed objects.

:add-object *obj* [method]

Add object to scene with identifiable name. Returns all objects.

:add-objects *objects* [method]

Add objects to scene with identifiable names. Returns all objects.

:objects [method]

Returns objects in the scene.

:init [method]

&rest args &key ((:name n) scene)
((:objects o))
((:remove-wall w))
&allow-other-keys

Create scene model

:bodies [method]

18.7 Dynamics calculation/Walk motion generation

18.7.1 Walking motion generation

A walking motion is generated using anticipatory control. Explained while quoting the formulas and sentences described in the literature.[20][21][22]

A walking motion is generated by the following procedure.

1. Plan the positions and times at which the robot will land on the stride within the range where the inverse kinematics of the legs can be solved.
2. If the foot is not on the ground, cycloidal interpolation is performed to the next position.
3. Generate a center of gravity trajectory that matches the position of the foot on the ground as much as possible.
4. The joint angle trajectory that satisfies the planned trajectory of the foot and the center of gravity is obtained by inverse kinematics.

Here, the 3 steps are explained in detail.

First, consider the following control system for motion in the X direction. A similar argument can be made for motion in the Y direction.

$$\begin{cases} x_{k+1} = Ax_k + bu_k \\ p_k = cx_k \end{cases} \tag{57}$$

$$x_k \equiv \begin{bmatrix} x(k\Delta t) \\ \dot{x}(k\Delta t) \\ \ddot{x}(k\Delta t) \end{bmatrix} \quad u_k \equiv u(k\Delta t) \quad p_k \equiv p(k\Delta t)$$

$$A \equiv \begin{bmatrix} 1 & \Delta t & \Delta t^2/2 \\ 0 & 1 & \Delta t \\ 0 & 0 & 1 \end{bmatrix} \quad b \equiv \begin{bmatrix} \Delta t^3/6 \\ \Delta t^2/2 \\ \Delta t \end{bmatrix} \quad c \equiv \begin{bmatrix} 1 & 0 & -z_c/g \end{bmatrix}$$

x is the position of the center of gravity of the robot, u is the time derivative (jerk) of the acceleration of the center of gravity, and p is the ZMP.

Next, replace this system with the following expansion system.

$$\begin{cases} x^*_{k+1} = \tilde{A}x^*_k + \tilde{b}\Delta u_k \\ p_k = \tilde{c}x^*_k \end{cases} \tag{58}$$

$$\Delta u_k \equiv u_k - u_{k-1} \quad \Delta x_k \equiv x_k - x_{k-1} \quad x^*_k \equiv \begin{bmatrix} p_k \\ \Delta x_k \end{bmatrix}$$

$$\tilde{A} \equiv \begin{bmatrix} 1 & cA \\ 0 & A \end{bmatrix} \quad \tilde{b} \equiv \begin{bmatrix} cb \\ b \end{bmatrix} \quad \tilde{c} \equiv \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}$$

For this Equation **??** system, we obtain control inputs that minimize the following cost function.

$$J_k = \sum_{j=k}^{\infty}\{Q(p_j^{ref} - p_j)^2 + R\Delta u_j^2\} \tag{59}$$

[20] Humanoid Robot (in Japanese), Shuji Kajita, Ohmsha, 2005, ISBN 4-274-20058-2
[21] Biped Walking Pattern Generation by using Preview Control of Zero-Moment Point, Shuji Kajita and Fumio Kanehiro and Kenji Kaneko and Kiyoshi Fujiwara and Kensuke Harada and Kazuhito Yokoi and Hirohisa Hirukawa, ICRA 2003, p.1620-1626, 2006
[22] Optimal Predictive Control and Generalized Predictive Control, Tadashi Egami and Takeshi Tsuchiya, Instrumentation and Control, Vol.39, No.5, p.337-342, 2000

This is obtained as follows. First, consider the following cost function using a sufficiently large natural number M, then set $M \to \infty$ to obtain the control input that minimizes Equation **??** .

$$J_k^M = \sum_{j=k}^{M-1} \{Q(p_j^{ref} - p_j)^2 + R\Delta u_j^2\} \tag{60}$$

Assuming that the minimum value of J_k^M is J_k^{M*}, the following relationship holds from Equation **??** .

$$J_k^{M*} = \min_{\Delta u_k} \{Q(p_k^{ref} - p_k)^2 + R\Delta u_k^2 + J_{k+1}^{M*}\} \tag{61}$$

Now put J_k^{M*} as follows.

$$J_k^{M*} = x_k^{*T} P_k^M x_k^* + \theta_k^{MT} x_k^* + \phi_k^M \tag{62}$$

Using this, the value obtained by partially differentiating the right side of Equation **??** with respect to Δu_k is 0, so the following equation is obtained.

$$\begin{aligned}
0 =& \quad \frac{\partial}{\partial \Delta u_k}\{Q(p_k^{ref} - p_k)^2 + R\Delta u_k^2 + (\tilde{A}x_k^* + \tilde{b}\Delta u_k)^T P_{k+1}^M (\tilde{A}x_k^* + \tilde{b}\Delta u_k) + \theta_{k+1}^{MT}(\tilde{A}x_k^* + \tilde{b}\Delta u_k) + \phi_{k+1}^M\} \\
0 =& \quad \Delta u_k^T R + \Delta u_k^T \tilde{b}^T P_{k+1}^M \tilde{b} + x_k^{*T}\tilde{A}^T P_{k+1}^M \tilde{b} + \frac{1}{2}\theta_{k+1}^{MT}\tilde{b} \\
\Delta u_k =& \quad -(\tilde{b}^T P_{k+1}^M \tilde{b} + R)^{-1}\tilde{b}^T P_{k+1}^M \tilde{A} x_k^* - \frac{1}{2}(\tilde{b}^T P_{k+1}^M \tilde{b} + R)^{-1}\tilde{b}^T \theta_{k+1}^M
\end{aligned} \tag{63}$$

By substituting this into Equation **??** , we get:

$$\begin{aligned}
& x_k^{*T} P_k^M x_k^* + \theta_k^{MT} x_k^* + \phi_k^M \\
= \quad & x_k^{*T}(\tilde{c}^T Q\tilde{c} + \tilde{A}^T P_{k+1}^M \tilde{A} - \tilde{A}^T P_{k+1}^M \tilde{b}(\tilde{b}^T P_{k+1}^M \tilde{b} + R)^{-1}\tilde{b}^T P_{k+1}^M \tilde{A})x_k^* \\
& +\{-2\tilde{c}^T Q p_k^{ref} + \tilde{A}^T \theta_{k+1}^M - \tilde{A}^T P_{k+1}^M \tilde{b}(\tilde{b}^T P_{k+1}^M \tilde{b} + R)^{-1}\tilde{b}^T \theta_{k+1}^M\}^T x_k^* \\
& +Q p_k^{ref2} - \frac{1}{4}\theta_{k+1}^{MT}\tilde{b}(\tilde{b}^T P_{k+1}^M \tilde{b} + R)^{-1}\tilde{b}^T \theta_{k+1}^M + \phi_{k+1}^M
\end{aligned} \tag{64}$$

Since this holds for any x_k^{*T},

$$P_k^M = \tilde{c}^T Q\tilde{c} + \tilde{A}^T P_{k+1}^M \tilde{A} - \tilde{A}^T P_{k+1}^M \tilde{b}(\tilde{b}^T P_{k+1}^M \tilde{b} + R)^{-1}\tilde{b}^T P_{k+1}^M \tilde{A} \tag{65}$$

$$\theta_k^M = -2\tilde{c}^T Q p_k^{ref} + \{\tilde{A}^T - \tilde{A}^T P_{k+1}^M \tilde{b}(\tilde{b}^T P_{k+1}^M \tilde{b} + R)^{-1}\tilde{b}^T\}\theta_{k+1}^M \tag{66}$$

$$\phi_k^M = Q p_k^{ref2} - \frac{1}{4}\theta_{k+1}^{MT}\tilde{b}(\tilde{b}^T P_{k+1}^M \tilde{b} + R)^{-1}\tilde{b}^T \theta_{k+1}^M + \phi_{k+1}^M \tag{67}$$

Considering the boundary conditions, Equation **??** is $J_{k+M}^{M*} = 0$ at $k = M$, so Equation **??** at $k = M$ is Considering the condition that holds identically for any x_M^*, $P_M^M = 0$, $\theta_M^M = 0$, $\phi_M^M = 0$. Therefore, for sufficiently small j, P_{k+j}^M becomes the stationary solution P of Equation **??** , which satisfies the following equation.

$$P = \tilde{c}^T Q\tilde{c} + \tilde{A}^T P\tilde{A} - \tilde{A}^T P\tilde{b}(\tilde{b}^T P\tilde{b} + R)^{-1}\tilde{b}^T P\tilde{A} \tag{68}$$

Henceforth, we consider it as $M \to \infty$. Considering a natural number N and assuming $p_{k+j}^{ref} = p_{k+N}^{ref}$ for $j > N$, Equation **??** can be expressed as (Equation **??** holds only for P_{k+j} for small enough j, but for large enough j $(\tilde{A} - \tilde{b}K)^{Tj} \to assumed 0$ and ignored.)($K \equiv (\tilde{b}^T P\tilde{b} + R)^{-1}\tilde{b}^T P\tilde{A}$.)

$$\begin{aligned}
\theta_k =& \; -2\tilde{c}^T Q p_k^{ref} + (\tilde{A} - \tilde{b}K)^T \theta_{k+1} \\
=& \; -2\{\tilde{c}^T Q p_k^{ref} + (\tilde{A} - \tilde{b}K)^T \tilde{c}^T Q p_{k+1}^{ref} + \cdots + (\tilde{A} - \tilde{b}K)^{TN-2}\tilde{c}^T Q p_{k+N-1}^{ref} \\
& +(\tilde{A} - \tilde{b}K)^{TN-1}\tilde{c}^T Q p_{k+N}^{ref} + (\tilde{A} - \tilde{b}K)^{TN}\tilde{c}^T Q p_{k+N+1}^{ref} + (\tilde{A} - \tilde{b}K)^{T(N+1)}\tilde{c}^T Q p_{k+N+2}^{ref} + \cdots\} \\
=& \; -2\{\tilde{c}^T Q p_k^{ref} + (\tilde{A} - \tilde{b}K)^T \tilde{c}^T Q p_{k+1}^{ref} + \cdots + (\tilde{A} - \tilde{b}K)^{TN-2}\tilde{c}^T Q p_{k+N-1}^{ref} \\
& +(\tilde{A} - \tilde{b}K)^{TN-1}\tilde{c}^T Q p_{k+N}^{ref} + (\tilde{A} - \tilde{b}K)^{TN}\tilde{c}^T Q p_{k+N}^{ref} + (\tilde{A} - \tilde{b}K)^{T(N+1)}\tilde{c}^T Q p_{k+N}^{ref} + \cdots\} \\
=& \; -2\sum_{j=1}^{N-1}\{(\tilde{A} - \tilde{b}K)^{Tj-1}\tilde{c}^T Q p_{k+j}^{ref}\} - 2\sum_{j=N}^{\infty}\{(\tilde{A} - \tilde{b}K)^{Tj-1}\tilde{c}^T Q p_{k+N}^{ref}\}
\end{aligned} \tag{69}$$

Here, if we express Equation **??** using K, we get

$$P = \tilde{c}^T Q \tilde{c} + (\tilde{A} - \tilde{b}K)^T P \tilde{A} \tag{70}$$

By adding $P\tilde{A}$ to both sides and rearranging,

$$(I - (\tilde{A} - \tilde{b}K)^T) P \tilde{A} = P(\tilde{A} - I) + \tilde{c}^T Q \tilde{c} \tag{71}$$

where $\tilde{A} = \begin{bmatrix} 1 & cA \\ 0 & A \end{bmatrix}$, $\tilde{c} = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}$, from the equation in the first column of Equation **??** ,

$$(I - (\tilde{A} - \tilde{b}K)^T) P \tilde{c}^T = \tilde{c}^T Q \tag{72}$$

Substituting this Equation **??** into Equation **??** gives

$$\theta_k = -2 \sum_{j=1}^{N-1} \{(\tilde{A} - \tilde{b}K)^{Tj-1} \tilde{c}^T Q p_{k+j}^{ref}\} - 2(\tilde{A} - \tilde{b}K)^{TN-1} P \tilde{c}^T p_{k+N}^{ref} \tag{73}$$

the optimal control input Δu_k that minimizes Equation **??** is obtained.

$$\Delta u_k = -K x_k^* + \sum_{j=1}^{N} \tilde{f}_j p_{k+j}^{ref} \tag{74}$$

$$\tilde{f}_j = \begin{cases} (\tilde{b}^T P \tilde{b} + R)^{-1} \tilde{b}^T (\tilde{A} - \tilde{b}K)^{Tj-1} \tilde{c}^T Q & (j < N) \\ (\tilde{b}^T P \tilde{b} + R)^{-1} \tilde{b}^T (\tilde{A} - \tilde{b}K)^{TN-1} P \tilde{c}^T & (j = N) \end{cases}$$

Given initial state x_1^* and target ZMP trajectory p_1^{ref}, p_2^{ref}, $\cdots$, the system of Equation **??** and Equation **??** system, Δu_1, x_2^*, Δu_2, x_3^*, $\cdots$ are obtained sequentially. This makes it possible to generate a center-of-gravity trajectory whose ZMP matches the position of the foot on the ground as much as possible.

18.7.2 Example of Walking Motion Generation

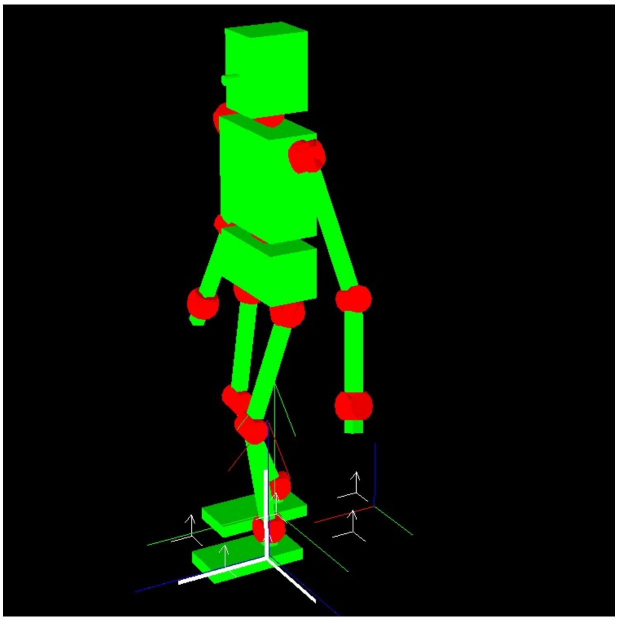


図 25: Example of walk pattern generation

The robot-model class defines a function :calc-walk-pattern-from-footstep-list that generates offline walking motion and returns the walking trajectory. Given a list of foot placement positions for each step, this function generates a walking motion that makes the ZMP as close to this position as possible. The program shown below uses :calc-walk-pattern-from-footstep-list to generate walking motion (Fig.??).

```
(load "irteus/demo/sample-robot-model.l")
(setq *robot* (instance sample-robot :init))
(send *robot* :reset-pose)
(send *robot* :fix-leg-to-coords (make-coords))
(objects (list *robot*))

(let ((footstep-list
       (list (make-coords :coords (send *robot* :rleg :end-coords :copy-worldcoords) :name :rleg)
             (make-coords :coords (send (send *robot* :lleg :end-coords :copy-worldcoords)
                                        :translate #f(100 0 0)) :name :lleg)
             (make-coords :coords (send (send *robot* :rleg :end-coords :copy-worldcoords)
                                        :translate #f(200 0 0)) :name :rleg)
             (make-coords :coords (send (send *robot* :lleg :end-coords :copy-worldcoords)
                                        :translate #f(300 0 0)) :name :lleg)
             (make-coords :coords (send (send *robot* :rleg :end-coords :copy-worldcoords)
                                        :translate #f(400 0 0)) :name :rleg)
             (make-coords :coords (send (send *robot* :lleg :end-coords :copy-worldcoords)
                                        :translate #f(400 0 0)) :name :lleg))))
  (objects (append (list *robot*) footstep-list))

  (send *robot* :calc-walk-pattern-from-footstep-list
        footstep-list
        :default-step-height 50
        :default-step-time 1.0
        :dt 0.1
        :debug-view t)
  )
```

The variable footstep-list is set to the list of footstep positions and given to :calc-walk-pattern-from-footstep-list. Each element of footstep-list must specify which foot it is in :name. The first element of footstep-list is used for initialization, and the robot actually steps to positions after the next element. :default-step-height 50 specifies 50[mm] as the height for cycloidal interpolation of the swing leg. :default-step-time 1.0 specifies 1.0[s] as the time per step. :dt 0.1 specifies 0.1[s] as the sampling time of preview control and the interval between generated trajectories. Note that :default-step-height, :default-step-time and :dt are the default values, so there is no need to actually set these values.

The following program generates walking motions by automatically generating footstep-list when a desired destination is given.

```
(load "irteus/demo/sample-robot-model.l")
(setq *robot* (instance sample-robot :init))
(send *robot* :reset-pose)
(send *robot* :fix-leg-to-coords (make-coords))
(objects (list *robot*
               (apply #'midcoords 0.5 (send *robot* :legs :end-coords))
               (send (send (apply #'midcoords 0.5 (send *robot* :legs :end-coords))
                           :translate #F(500 150 0)) :rotate (deg2rad 45) :z)))

(send *robot* :calc-walk-pattern-from-footstep-list
      (send *robot* :go-pos-params->footstep-list
            500 150 45) ;; x[mm] y[mm] th[deg]
      :debug-view t
      )
```

Give the return value of (send *robot* :go-pos-params->footstep-list 500 150 45) as footstep-list The walk-pattern-from-footstep-list— function generates walking motions.

The :go-pos-params->footstep-list function provides a footstep- This is a function that generates a l and generates a footstep-list that moves 500[mm] forward, 150[mm] left, and rotates 45[deg] to the left. there is

joint [Class]

:super **propertied-object**

:slots parent-link child-link joint-angle min-angle max-angle default-coords joint-velocity joi

:calc-inertia-matrix *&rest args* [method]

rotational-joint [Class]

:super **joint**

:slots parent-link child-link joint-angle min-angle max-angle default-coords joint-velocity joi

:calc-inertia-matrix *mat row column paxis m-til c-til i-til axis-for-angular world-default-coords translation-axis rotation-axis tmp-v0 tmp-v1 tmp-v2 tmp-va tmp-vb tmp-vc tmp-vd tmp-m* [method]

linear-joint [Class]

:super **joint**

:slots parent-link child-link joint-angle min-angle max-angle default-coords joint-velocity joi

:calc-inertia-matrix *mat row column paxis m-til c-til i-til axis-for-angular world-default-coords translation-axis rotation-axis tmp-v0 tmp-v1 tmp-v2 tmp-va tmp-vb tmp-vc tmp-vd tmp-m* [method]

omniwheel-joint [Class]

:super **joint**

:slots parent-link child-link joint-angle min-angle max-angle default-coords joint-velocity joi

:calc-inertia-matrix *mat row column paxis m-til c-til i-til axis-for-angular world-default-coords translation-axis rotation-axis tmp-v0 tmp-v1 tmp-v2 tmp-va tmp-vb tmp-vc tmp-vd tmp-m* [method]

sphere-joint [Class]

:super **joint**

:slots parent-link child-link joint-angle min-angle max-angle default-coords joint-velocity joi

:calc-inertia-matrix *mat row column paxis m-til c-til i-til axis-for-angular world-default-coords translation-axis rotation-axis tmp-v0 tmp-v1 tmp-v2 tmp-va tmp-vb tmp-vc tmp-vd tmp-m* [method]

6dof-joint [Class]

:super **joint**

:slots parent-link child-link joint-angle min-angle max-angle default-coords joint-velocity joi

:calc-inertia-matrix *mat row column paxis m-til c-til i-til axis-for-angular world-default-coords translation-axis rotation-axis tmp-v0 tmp-v1 tmp-v2 tmp-va tmp-vb tmp-vc tmp-vd tmp-m* [method]

bodyset-link [Class]

:super **bodyset**

:slots rot pos parent descendants worldcoords manager changed geometry::bodies joint pare

:calc-inertia-matrix-column *column &rest args &key (rotation-axis nil) (translation-axis t) ((:inertia-matrix im)) (axis-for-angular (float-vector 0 0 0)) (tmp-v0 (instantiate float-vector 0)) (tmp-v1 (instantiate float-vector 1)) (tmp-v2 (instantiate float-vector 2)) (tmp-va (instantiate float-vector 3)) (tmp-vb (instantiate float-vector 3)) (tmp-vc (instantiate float-vector 3)) (tmp-vd (instantiate float-vector 3)) (tmp-ma (make-matrix 3 3)) &allow-other-keys* [method]

:propagate-mass-properties *&key (debug-view nil) (tmp-va (instantiate float-vector 3)) (tmp-vb (instantiate float-vector 3)) (tmp-ma (make-matrix 3 3)) (tmp-mb (make-matrix 3 3)) (tmp-mc (make-matrix 3 3))* [method]

:append-mass-properties *additional-links &key (update t) (tmp-va (float-vector 0 0 0)) (tmp-vb (float-vector 0 0 0)) (tmp-ma (make-matrix 3 3)) (tmp-mb (make-matrix 3 3)) (tmp-mc (make-matrix 3 3)) (tmp-md (make-matrix 3 3)) (additional-weights (send-all additional-links :weight)) (additional-centroids (send-all additional-links :centroid)) (additional-inertias (mapcar #'(lambda (x) (m*(m*(send x :worldrot) (send x :inertia-tensor) tmp-ma) (transpose (send x :worldrot) tmp-mb))) additional-links)) (self-centroid (send self :centroid))* [method]

:append-inertia-no-update *additional-weights additional-centroids additional-inertias self-centroid new-centroid &key (tmp-ma (make-matrix 3 3)) (tmp-mb (make-matrix 3 3)) (tmp-mc (make-matrix 3 3)) (tmp-md (make-matrix 3 3)) (tmp-va (float-vector 0 0 0)) (len (length additional-weights))* [method]

:append-centroid-no-update *additional-weights additional-centroids self-centroid new-weight &key (tmp-va (float-vector 0 0 0)) (tmp-vb (float-vector 0 0 0)) (len (length additional-weights))* [method]

:append-weight-no-update *additional-weights &key (len (length additional-weights))* [method]

cascaded-link [Class]

:super	**cascaded-coords**
:slots	rot pos parent descendants worldcoords manager changed links joint-list bodies collisi

:cog-convergence-check *centroid-thre target-centroid-pos &key (centroid-offset-func) (translation-axis :z) (update-mass-properties t)* [method]

:difference-cog-position *target-centroid-pos &key (centroid-offset-func) (translation-axis :z) (add-draw-on-param) (update-mass-properties t)* [method]

:calc-vel-for-cog *cog-gain translation-axis target-centroid-pos &key (centroid-offset-func) (update-mass-properties t)* [method]

:cog-jacobian-balance-nspace *link-list &rest args &key (cog-gain 1.0) (translation-axis :z) (target-centroid-pos) (centroid-offset-func) (update-mass-properties t) &allow-other-keys* [method]

:calc-cog-jacobian-from-link-list *&rest args &key (link-list (send-all joint-list :child-link)) (rotation-axis nil) (translation-axis t) (axis-dim (send self :calc-target-axis-dimension rotation-axis translation-axis)) (inertia-matrix (make-matrix axis-dim (send self :calc-target-joint-dimension link-list))) (update-mass-properties t) &allow-other-keys* [method]

:calc-inertia-matrix-from-link-list *&rest args &key (link-list (send-all joint-list :child-link)) (rotation-axis nil) (translation-axis t) (axis-dim (send self :calc-target-axis-dimension rotation-axis translation-axis)) (inertia-matrix (make-matrix axis-dim (send self :calc-target-joint-dimension link-list))) (update-mass-properties t) (axis-for-angular) (tmp-v0 (instantiate float-vector 0)) (tmp-v1 (instantiate float-vector 1)) (tmp-v2 (instantiate float-vector 2)) (tmp-va (instantiate float-vector 3)) (tmp-vb (instantiate float-vector 3)) (tmp-vc (instantiate float-vector 3)) (tmp-vd (instantiate float-vector 3)) (tmp-ma (make-matrix 3 3)) (tmp-mb (make-matrix 3 3)) (tmp-mc (make-matrix 3 3)) &allow-other-keys* [method]

:update-mass-properties *&key (tmp-va (instantiate float-vector 3)) (tmp-vb (instantiate float-vector 3)) (tmp-ma (make-matrix 3 3)) (tmp-mb (make-matrix 3 3)) (tmp-mc (make-matrix 3 3))* [method]

joint [Class]

:super	**propertied-object**
:slots	parent-link child-link joint-angle min-angle max-angle default-coords joint-velocity joi

:calc-angular-acceleration-jacobian *&rest args* [method]

:calc-spacial-acceleration-jacobian *&rest args* [method]

:calc-angular-velocity-jacobian *&rest args* [method]

:calc-spacial-velocity-jacobian *&rest args* [method]

rotational-joint [Class]

:super **joint**

:slots parent-link child-link joint-angle min-angle max-angle default-coords joint-velocity joi

:calc-angular-acceleration-jacobian *avj tmp-va* [method]

:calc-spacial-acceleration-jacobian *svj avj tmp-va tmp-vb* [method]

:calc-angular-velocity-jacobian *ax tmp-va* [method]

:calc-spacial-velocity-jacobian *ax tmp-va tmp-vb* [method]

linear-joint [Class]

:super **joint**

:slots parent-link child-link joint-angle min-angle max-angle default-coords joint-velocity joi

:calc-angular-acceleration-jacobian *avj tmp-va* [method]

:calc-spacial-acceleration-jacobian *svj avj tmp-va tmp-vb* [method]

:calc-angular-velocity-jacobian *ax tmp-va* [method]

:calc-spacial-velocity-jacobian *ax tmp-va tmp-vb* [method]

bodyset-link [Class]

:super **bodyset**

:slots rot pos parent descendants worldcoords manager changed geometry::bodies joint pare

:inverse-dynamics *&key (debug-view nil) (tmp-va (float-vector 0 0 0)) (tmp-vb (float-vector 0 0 0)) (tmp-vc (float-vector 0 0 0)) (tmp-ma (make-matrix 3 3)) (tmp-mb (make-matrix 3 3)) (tmp-mc (make-matrix 3 3)) (tmp-md (make-matrix 3 3))* [method]

:forward-all-kinematics *&key (debug-view nil) (tmp-va (float-vector 0 0 0))* [method]

:ext-moment *&optional m* [method]

:ext-force *&optional f* [method]

:moment [method]

:force [method]

:spacial-acceleration *&optional sa* [method]

:spacial-velocity *&optional aa* [method]

:angular-acceleration *&optional aa* [method]

:angular-velocity *&optional aa* [method]

:reset-dynamics [method]

cascaded-link [Class]

:super **cascaded-coords**
:slots rot pos parent descendants worldcoords manager changed links joint-list bodies collisi

:inertia-tensor *&optional (update-mass-properties t)* [method]

Calculate total robot inertia tensor [g mm^2] around total robot centroid in euslisp world coordinates.
If update-mass-properties argument is t, propagate total mass prop calculation for all links and returns total robot inertia tensor.
Otherwise, do not calculate total mass prop, just returns pre-computed total robot inertia tensor.

:centroid *&optional (update-mass-properties t)* [method]

Calculate total robot centroid (Center Of Gravity, COG) [mm] in euslisp world coordinates.
If update-mass-properties argument is t, propagate total mass prop calculation for all links and returns total robot centroid.
Otherwise, do not calculate total mass prop, just returns pre-computed total robot centroid.

:weight *&optional (update-mass-properties t)* [method]

Calculate total robot weight [g].
If update-mass-properties argument is t, propagate total mass prop calculation for all links and returns total robot weight.
Otherwise, do not calculate total weight, just returns pre-computed total robot weight.

:calc-zmp [method]

&optional (av (send self :angle-vector)) (root-coords (send (car (send self :links)) :copy-worldcoords)) &
(dt 0.005)
(update t)
(debug-view)
(calc-torque-buffer-args (send self :calc-torque-buffer-args))

Calculate Zero Moment Point based on Inverse Dynamics.
The output is expressed by the world coordinates,
and depends on historical robot states of the past 3 steps. Step is incremented when this method is called.
After solving Inverse Dynamics, ZMP is calculated from total root-link force and moment.
necessary arguments ->av and root-coords.

If update is t, call inverse dynamics, otherwise, just return zmp from total root-link force and moment.
dt [s] is time step used only when update is t.
pZMPz is ZMP height [mm].
After this method, (send robot :get :zmp) is ZMP and (send robot :get :zmp-moment) is moment around ZMP.

:preview-control-dynamics-filter *dt avs &key (preview-controller-class preview-controller) (cog-method :move-base-pos) (delay 0.8)* [method]

:draw-torque *vwer &key flush (width 2) (size 100) (color (float-vector 1 0.3 0)) (warning-color (float-vector 1 0 0)) (torque-threshold nil) (torque-vector (send self :torque-vector)) ((:joint-list jlist) (send self :joint-list))* [method]

:calc-contact-wrenches-from-total-wrench *target-pos-list &key (total-wrench) (weight (fill (instantiate float-vector (*6 (length target-pos-list))) 1))* [method]

:wrench-list->wrench-vector *wrench-list* [method]

:wrench-vector->wrench-list *wrench-vector* [method]

:calc-cop-from-force-moment *force moment sensor-coords cop-coords &key (fz-thre 1) (return-all-values)* [method]

:calc-torque-from-ext-wrenches *&key (force-list) (moment-list) (target-coords) ((:jacobi tmp-jacobi))* [method]

:calc-av-vel-acc-from-pos *dt av* [method]

:calc-root-coords-vel-acc-from-pos *dt root-coords* [method]

:calc-torque-from-vel-acc *&key (debug-view nil) (jvv (instantiate float-vector (length joint-list))) (jav (instantiate float-vector (length joint-list))) (root-spacial-velocity (float-vector 0 0 0)) (root-angular-velocity (float-vector 0 0 0)) (root-spacial-acceleration (float-vector 0 0 0)) (root-angular-acceleration (float-vector 0 0 0)) (calc-torque-buffer-args (send self :calc-torque-buffer-args))* [method]

:calc-torque-without-ext-wrench *&key (debug-view nil) (calc-statics-p t) (dt 0.005) (av (send self :angle-vector)) (root-coords (send (car (send self :links)) :copy-worldcoords)) (calc-torque-buffer-args (send self :calc-torque-buffer-args))* [method]

:calc-torque *&key (debug-view nil) (calc-statics-p t) (dt 0.005) (av (send self :angle-vector)) (root-coords (send (car (send self :links)) :copy-worldcoords)) (force-list) (moment-list) (target-coords) (calc-torque-buffer-args (send self :calc-torque-buffer-args))* [method]

:calc-torque-buffer-args [method]

:torque-ratio-vector *&rest args &key (torque (send*self :torque-vector args))* [method]

:max-torque-vector [method]

riccati-equation [Class]

:super **propertied-object**
:slots a b c p q r k a-bkt r+btpb-1

:init *aa bb cc qq rr* [method]

:solve [method]

:solve [method]

:init *aa bb cc qq rr* [method]

preview-controller [Class]

:super	**riccati-equation**
:slots	xk uk delay f1-n y1-n queue-index initialize-queue-p additional-data-queue finishedp i

:init [method]

dt &key (q)
(r)
((:delay d))
((:a _a))
((:b _b))
((:c _c))
(state-dim (array-dimension _a 0))
((:output-dim odim) (array-dimension _c 0))
((:input-dim idim) (array-dimension _b 1))
(init-xk (instantiate float-vector (array-dimension _a 0)))
(init-uk (make-matrix (array-dimension _b 1) 1))
((:initialize-queue-p iqp))

Initialize preview-controller.
Q is weighting of output error and R is weighting of input.
dt is sampling time [s].
delay is preview time [s].
init-xk is initial state value.
A, B, C are state eq matrices.
If initialize-queue-p is t, fill all queue by the first input at the begenning, otherwise, do not fill queue at the first.

:update-xk *p &optional (add-data)* [method]

Update xk by inputting reference output.
Return value : nil (initializing) =>return values (middle) =>nil (finished)
If p is nil, automatically the last value in queue is used as input and preview controller starts finishing.

:finishedp [method]

Finished or not.

:last-reference-output-vector [method]

Last value of reference output queue vector (y_k+N_ref).
Last value is latest future value.

:current-reference-output-vector [method]

First value of reference output queue vector (y_k_ref).
First value is oldest future value and it can be used as current reference value.

:current-state-vector [method]

Current state value (xk).

:current-output-vector [method]

Current output value (yk).

:current-additional-data [method]

Current additional data value.

First value of additional-data-queue.

:pass-preview-controller *reference-output-vector-list* [method]

Get preview controller results from reference-output-vector-list and returns list.

:calc-f [method]

:calc-u [method]

:calc-xk [method]

:pass-preview-controller *reference-output-vector-list* [method]

Get preview controller results from reference-output-vector-list and returns list.

:current-additional-data [method]

Current additional data value.

First value of additional-data-queue.

:current-output-vector [method]

Current output value (yk).

:current-state-vector [method]

Current state value (xk).

:current-reference-output-vector [method]

First value of reference output queue vector (y_k_ref).

First value is oldest future value and it can be used as current reference value.

:last-reference-output-vector [method]

Last value of reference output queue vector (y_k+N_ref).

Last value is latest future value.

:finishedp [method]

Finished or not.

:update-xk *p &optional (add-data)* [method]

Update xk by inputting reference output.

Return value : nil (initializing) =>return values (middle) =>nil (finished)

If p is nil, automatically the last value in queue is used as input and preview controller starts finishing.

:init [method]

dt &key (q)
(r)
((:delay d))
((:a _a))
((:b _b))
((:c _c))
(state-dim (array-dimension _a 0))

((:output-dim odim) (array-dimension _c 0))
((:input-dim idim) (array-dimension _b 1))
(init-xk (instantiate float-vector (array-dimension _a 0)))
(init-uk (make-matrix (array-dimension _b 1) 1))
((:initialize-queue-p iqp))

Initialize preview-controller.
Q is weighting of output error and R is weighting of input.
dt is sampling time [s].
delay is preview time [s].
init-xk is initial state value.
A, B, C are state eq matrices.
If initialize-queue-p is t, fill all queue by the first input at the begenning, otherwise, do not fill queue at the first.

:calc-xk [method]

:calc-u [method]

:calc-f [method]

extended-preview-controller [Class]

:super **preview-controller**
:slots orga orgb orgc xk*

:init [method]

dt &key (q)
(r)
((:delay d))
((:a _orga))
((:b _orgb))
((:c _orgc))
(init-xk (instantiate float-vector (array-dimension _orga 0)))
(init-uk (make-matrix (array-dimension _orgb 1) 1))
(state-dim (array-dimension _orga 0))
((:initialize-queue-p iqp))
(q-mat)

Initialize preview-controller in extended system (error system).
Q is weighting of output error and R is weighting of input.
dt is sampling time [s].
delay is preview time [s].
init-xk is initial state value.
A, B, C are state eq matrices for original system and slot variables A,B,C are used for error system matrices.
If initialize-queue-p is t, fill all queue by the first input at the begenning, otherwise, do not fill queue at the first.

:current-output-vector [method]

Current additional data value.
First value of additional-data-queue.

:calc-f [method]

:calc-u [method]

:calc-xk [method]

:current-output-vector [method]

Current additional data value.
First value of additional-data-queue.

:init [method]

dt &key (q)
(r)
((:delay d))
((:a _orga))
((:b _orgb))
((:c _orgc))
(init-xk (instantiate float-vector (array-dimension _orga 0)))
(init-uk (make-matrix (array-dimension _orgb 1) 1))
(state-dim (array-dimension _orga 0))
((:initialize-queue-p iqp))
(q-mat)

Initialize preview-controller in extended system (error system).
Q is weighting of output error and R is weighting of input.
dt is sampling time [s].
delay is preview time [s].
init-xk is initial state value.
A, B, C are state eq matrices for original system and slot variables A,B,C are used for error system matrices.
If initialize-queue-p is t, fill all queue by the first input at the begenning, otherwise, do not fill queue at the first.

:calc-xk [method]

:calc-u [method]

:calc-f [method]

preview-control-cart-table-cog-trajectory-generator [Class]

:super **propertied-object**
:slots pcs cog-z zmp-z

:init [method]

dt _zc &key (q 1)
(r 1.000000e-06)
((:delay d) 1.6)

(init-xk (float-vector 0 0 0))
((:a _a) (make-matrix 3 3 (list (list 1 dt (*0.5 dt dt)) (list 0 1 dt) (list 0 0 1))))
((:b _b) (make-matrix 3 1 (list (list (*(/ 1.0 6.0) dt dt dt)) (list (*0.5 dt dt)) (list dt))))
((:c _c) (make-matrix 1 3 (list (list 1.0 0.0 (- (/ _zc (elt *g-vec*2)))))))
((:initialize-queue-p iqp))
(preview-controller-class extended-preview-controller)

COG (xy) trajectory generator using preview-control convert reference ZMP from reference COG.
dt ->sampling time[s], _zc is height of COG [mm].
preview-controller-class is preview controller class (extended-preview-controller by default).
For other arguments, please see preview-controller and extended-preview-controller :init documentation.

:refcog [method]

Reference COG [mm].

:cart-zmp [method]

Cart-table system ZMP[mm] as an output variable.

:last-refzmp [method]

Reference zmp at the last of queue.

:current-refzmp [method]

Current reference zmp at the first of queue.

:update-xk *p &optional (add-data)* [method]

Update xk and returns zmp and cog values.
For arguments, please see preview-controller and extended-preview-controller :update-xk.

:finishedp [method]

Finished or not.

:current-additional-data [method]

Current additional data value.

:pass-preview-controller *reference-output-vector-list* [method]

Get preview controller results from reference-output-vector-list and returns list.

:cog-z [method]

COG Z [mm].

:update-cog-z *zc* [method]

:cog-z [method]

COG Z [mm].

:pass-preview-controller *reference-output-vector-list* [method]

Get preview controller results from reference-output-vector-list and returns list.

:current-additional-data [method]

Current additional data value.

:finishedp [method]

Finished or not.

:update-xk *p &optional (add-data)* [method]

Update xk and returns zmp and cog values.

For arguments, please see preview-controller and extended-preview-controller :update-xk.

:current-refzmp [method]

Current reference zmp at the first of queue.

:last-refzmp [method]

Reference zmp at the last of queue.

:cart-zmp [method]

Cart-table system ZMP[mm] as an output variable.

:refcog [method]

Reference COG [mm].

:init [method]

dt _zc &key (q 1)
(r 1.000000e-06)
((:delay d) 1.6)
(init-xk (float-vector 0 0 0))
((:a _a) (make-matrix 3 3 (list (list 1 dt (*0.5 dt dt)) (list 0 1 dt) (list 0 0 1))))
((:b _b) (make-matrix 3 1 (list (list (*(/ 1.0 6.0) dt dt dt)) (list (*0.5 dt dt)) (list dt))))
((:c _c) (make-matrix 1 3 (list (list 1.0 0.0 (- (/ _zc (elt *g-vec* 2)))))))
((:initialize-queue-p iqp))
(preview-controller-class extended-preview-controller)

COG (xy) trajectory generator using preview-control convert reference ZMP from reference COG.

dt ->sampling time[s], _zc is height of COG [mm].

preview-controller-class is preview controller class (extended-preview-controller by default).

For other arguments, please see preview-controller and extended-preview-controller :init documentation.

:update-cog-z *zc* [method]

gait-generator [Class]

:super	**propertied-object**
:slots	robot dt footstep-node-list support-leg-list support-leg-coords-list swing-leg-dst-coords

:init *rb _dt* [method]

:get-footstep-limbs *fs* [method]

:get-counter-footstep-limbs *fs* [method]

:get-limbs-zmp-list *limb-coords limb-names* [method]

:get-limbs-zmp *limb-coords limb-names* [method]

:get-swing-limbs *limbs* [method]

:initialize-gait-parameter *fsl time cog &key ((:default-step-height dsh) 50) ((:default-double-support-ratio ddsr) 0.2) (delay 1.6) ((:all-limbs al) '(:rleg :lleg)) ((:default-zmp-offsets dzo) (mapcan #'(lambda (x) (list x (float-vector 0 0 0))) al)) (q 1.0) (r 1.000000e-06) (thre 1) (rthre (deg2rad 1)) (start-with-double-support t) ((:end-with-double-support ewds) t)* [method]

:finalize-gait-parameter [method]

:make-gait-parameter [method]

:calc-hoffarbib-pos-vel-acc *tmp-remain-time tmp-goal old-acc old-vel old-pos* [method]

:calc-current-swing-leg-coords *ratio src dst &key (type :shuffling) (step-height default-step-height)* [method]

:calc-ratio-from-double-support-ratio [method]

:calc-current-refzmp *prev cur next* [method]

:calc-one-tick-gait-parameter *type* [method]

:proc-one-tick *&key (type :shuffling) (solve-angle-vector :solve-av-by-move-centroid-on-foot) (solve-angle-vector-args) (debug nil)* [method]

:update-current-gait-parameter [method]

:solve-angle-vector *support-leg support-leg-coords swing-leg-coords cog &key (solve-angle-vector :solve-av-by-move-centroid-on-foot) (solve-angle-vector-args)* [method]

:solve-av-by-move-centroid-on-foot *support-leg support-leg-coords swing-leg-coords cog robot &rest args &key (cog-gain 3.5) (stop 100) (additional-nspace-list) &allow-other-keys* [method]

:cycloid-midpoint *ratio start goal height &key (top-ratio 0.5)* [method]

:cycloid-midcoords *ratio start goal height &key (top-ratio 0.5) (rot-ratio ratio)* [method]
:cycloid-midcoords *ratio start goal height &key (top-ratio 0.5) (rot-ratio ratio)* [method]

:cycloid-midpoint *ratio start goal height &key (top-ratio 0.5)* [method]

:solve-av-by-move-centroid-on-foot *support-leg support-leg-coords swing-leg-coords cog robot &rest args &key (cog-gain 3.5) (stop 100) (additional-nspace-list) &allow-other-keys* [method]

:solve-angle-vector *support-leg support-leg-coords swing-leg-coords cog &key (solve-angle-vector :solve-av-by-move-centroid-on-foot) (solve-angle-vector-args)* [method]

:update-current-gait-parameter [method]

:proc-one-tick *&key (type :shuffling) (solve-angle-vector :solve-av-by-move-centroid-on-foot) (solve-angle-vector-args) (debug nil)* [method]

:calc-one-tick-gait-parameter *type* [method]

:calc-current-refzmp *prev cur next* [method]

:calc-ratio-from-double-support-ratio [method]

:calc-current-swing-leg-coords *ratio src dst &key (type :shuffling) (step-height default-step-height)* [method]

:calc-hoffarbib-pos-vel-acc *tmp-remain-time tmp-goal old-acc old-vel old-pos* [method]

:make-gait-parameter [method]

:finalize-gait-parameter [method]

:initialize-gait-parameter *fsl time cog &key ((:default-step-height dsh) 50) ((:default-double-support-ratio ddsr) 0.2) (delay 1.6) ((:all-limbs al) '(:rleg :lleg)) ((:default-zmp-offsets dzo) (mapcan #'(lambda (x) (list x (float-vector 0 0 0))) al)) (q 1.0) (r 1.000000e-06) (thre 1) (rthre (deg2rad 1)) (start-with-double-support t) ((:end-with-double-support ewds) t)* [method]

:get-swing-limbs *limbs* [method]

:get-limbs-zmp *limb-coords limb-names* [method]

:get-limbs-zmp-list *limb-coords limb-names* [method]

:get-counter-footstep-limbs *fs* [method]

:get-footstep-limbs *fs* [method]

:init *rb _dt* [method]

calc-inertia-matrix-rotational *mat row column paxis m-til c-til i-til axis-for-angular child-link world-default-coords translation-axis rotation-axis tmp-v0 tmp-v1 tmp-v2 tmp-va tmp-vb tmp-vc tmp-vd tmp-m* [function]

calc-inertia-matrix-linear *mat row column paxis m-til c-til i-til axis-for-angular child-link world-default-coords translation-axis rotation-axis tmp-v0 tmp-v1 tmp-v2 tmp-va tmp-vb tmp-vc tmp-vd tmp-m* [function]

19 Robot Viewer

Create a default Irtviewer(Fig.**??**).

```
(make-irtviewer)
```

Create an irtviewer to draw a grid in the xy plane(Fig.**??**).

```
(make-irtviewer)
(send *irtviewer* :draw-floor 100)
(send *irtviewer* :draw-objects)
```

Create an irtviewer with a white background and a black grid in the xy plane(Fig.**??**).

```
(make-irtviewer)
(send *irtviewer* :change-background (float-vector 1 1 1))
(send *irtviewer* :draw-floor 100)
(send *irtviewer* :floor-color #f(0 0 0))
(send *irtviewer* :draw-objects)
```

viewer [Class]

:super **propertied-object**

:slots geometry::eye geometry::port geometry::surface

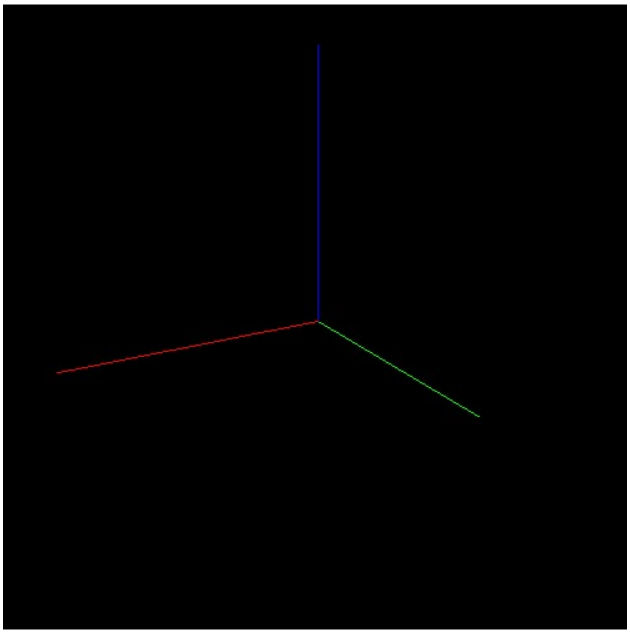


図 26: Default irtviewer

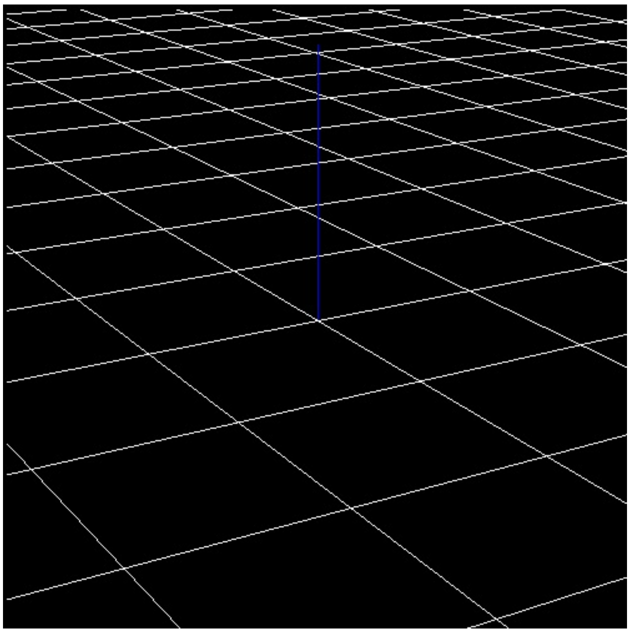


図 27: Irtviewer with floor grid

:draw-objects *&rest args* [method]

:draw-circle *x::c &key (x::radius 50) (x:flush nil) (x::arrow nil) (x::arc 2pi) (x::arrow-scale #f(1.0 1.0))* [method]

x::irtviewer [Class]

:super	**x:panel**
:slots	x::viewer x::objects x::draw-things x::previous-cursor-pos x::left-right-angle x::up-down

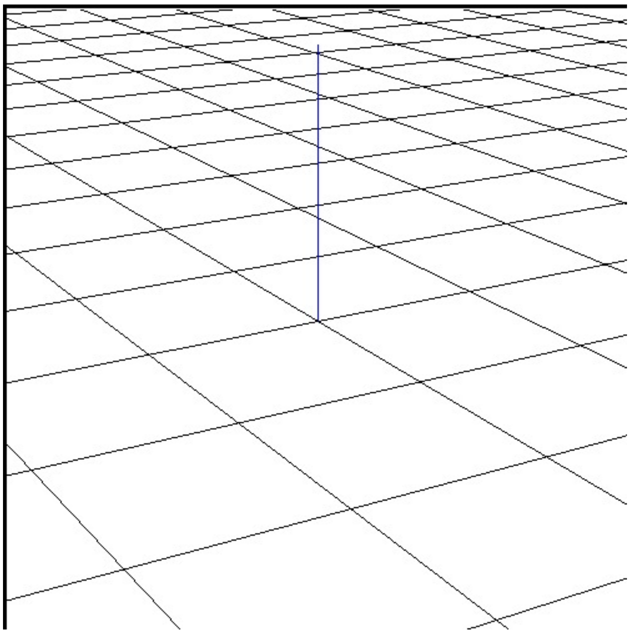


図 28: Irtviewer with white background and floor grid

:draw-origin *&optional (x::tmp-draw-origin :null)* [method]

get/set draw-origin

:draw-floor *&optional (x::tmp-draw-floor :null)* [method]

get/set draw-floor

:floor-color *&optional (x::tmp-floor-color :null)* [method]

get/set floor-color

:logging *&optional (x::flag :on)* [method]

start/stop logging
:clear Clear log
:start Start logging
:restart Stop and restart logging
:stop Stop logging

:save-mpeg [method]

&key (fname anim.mpg)
(x::delay 5)
(delete t)

':save-mpeg' saves logged images as mpeg.
To start image log, run ':logging :start' and to stop log, run ':logging :stop'.
Note that ':logging :stop' did not clear the logged sequence, so you need to run
':logging :clear'.
:save-anmigif' did not stop nor clear the image sequence, so you have to run them manuualy

:save-animgif [method]

&key (fname anim.gif)
(x::delay 5)
(x::transparent t)
(loop t)
(delete t)

':save-anmigif' saves logged images as animation gif.
To start image log, run ':logging :start' and to stop log, run ':logging :stop'.
Note that ':logging :stop' did not clear the logged sequence, so you need to run
':logging :clear'.
:save-anmigif' did not stop nor clear the image sequence, so you have to run them manuualy

:save-image *x::filename* [method]

save curent view to image, supported formats are jpg/png/pnm

:create *&rest args &key (x::title IRT viewer) (x::view-name (gensym title)) (x::hither 200.0) (x::yon 50000.0) (x::width 500) (x::height 500) ((:draw-origin do) 150) ((:draw-floor x::df) nil) ((:floor-color x::fc) #f(1.0 1.0 1.0)) (x::call-super t) &allow-other-keys* [method]

:viewer *&rest args* [method]

:redraw [method]

:expose *x:event* [method]

:resize *x::newwidth x::newheight* [method]

:configurenotify *x:event* [method]

:viewtarget *&optional x::p* [method]

:viewpoint *&optional x::p* [method]

:look1 *&optional (x::vt x::viewtarget) (x::lra x::left-right-angle) (x::uda x::up-down-angle)* [method]

:look-all *&optional x::bbox* [method]

:move-viewing-around-viewtarget *x:event x::x x::y x::dx x::dy x::vwr* [method]

:set-cursor-pos-event *x:event* [method]

:move-coords-event *x:event* [method]

:draw-event *x:event* [method]

:draw-objects *&rest args* [method]

:objects *&rest args* [method]

:select-drawmode *x::mode* [method]

:flush [method]

:change-background *x::col* [method]

:clear-log [method]

:push-image [method]

:clear-image-sequence [method]

:image-sequence [method]

:floor-color *&optional (x::tmp-floor-color :null)* [method]

get/set floor-color

:draw-floor *&optional (x::tmp-draw-floor :null)* [method]

get/set draw-floor

:draw-origin *&optional (x::tmp-draw-origin :null)* [method]

get/set draw-origin

:change-background *x::col* [method]

:flush [method]

:select-drawmode *x::mode* [method]

:objects *&rest args* [method]

:draw-objects *&rest args* [method]

:draw-event *x:event* [method]

:move-coords-event *x:event* [method]

:set-cursor-pos-event *x:event* [method]

:move-viewing-around-viewtarget *x:event x::x x::y x::dx x::dy x::vwr* [method]

:look-all *&optional x::bbox* [method]

:look1 *&optional (x::vt x::viewtarget) (x::lra x::left-right-angle) (x::uda x::up-down-angle)* [method]

:viewpoint *&optional x::p* [method]

:viewtarget *&optional x::p* [method]

:configurenotify *x:event* [method]

:resize *x::newwidth x::newheight* [method]

:expose *x:event* [method]

:redraw [method]

:viewer *&rest args* [method]

:create *&rest args &key (x::title IRT viewer) (x::view-name (gensym title)) (x::hither 200.0) (x::yon 50000.0) (x::width 500) (x::height 500) ((:draw-origin do) 150) ((:draw-floor x::df) nil) ((:floor-color x::fc) #f(1.0 1.0 1.0)) (x::call-super t) &allow-other-keys* [method]

:save-image *x::filename* [method]

save curent view to image, supported formats are jpg/png/pnm

:save-animgif [method]

&key (fname anim.gif)
(x::delay 5)
(x::transparent t)
(loop t)
(delete t)

':save-anmigif' saves logged images as animation gif.
To start image log, run ':logging :start' and to stop log, run ':logging :stop'.
Note that ':logging :stop' did not clear the logged sequence, so you need to run
':logging :clear'.
:save-anmigif' did not stop nor clear the image sequence, so you have to run them manuualy

:save-animgif [method]

&key (fname anim.gif)
(x::delay 5)
(x::transparent t)
(loop t)
(delete t)

':save-anmigif' saves logged images as animation gif.
To start image log, run ':logging :start' and to stop log, run ':logging :stop'.
Note that ':logging :stop' did not clear the logged sequence, so you need to run
':logging :clear'.
:save-anmigif' did not stop nor clear the image sequence, so you have to run them manuualy

:save-mpeg [method]

&key (fname anim.mpg)
(x::delay 5)
(delete t)

':save-mpeg' saves logged images as mpeg.
To start image log, run ':logging :start' and to stop log, run ':logging :stop'.
Note that ':logging :stop' did not clear the logged sequence, so you need to run
':logging :clear'.
:save-anmigif' did not stop nor clear the image sequence, so you have to run them manuualy

:logging *&optional (x::flag :on)* [method]

start/stop logging
:clear Clear log
:start Start logging
:restart Stop and restart logging
:stop Stop logging

:image-sequence [method]

:clear-image-sequence [method]

:push-image [method]

:clear-log [method]

viewing [Class]

:super **cascaded-coords**
:slots rot pos parent descendants worldcoords manager changed geometry::viewcoords

:look *geometry::from &optional (geometry::to (float-vector 0 0 0)) (geometry::view-up (float-vector 0 0 1))* [method]

viewer-dummy [Class]

:super **propertied-object**
:slots nil

:nomethod *&rest args* [method]
:nomethod *&rest args* [method]

irtviewer-dummy [Class]

:super **propertied-object**
:slots objects draw-things

:objects *&rest args* [method]

:nomethod *&rest args* [method]
:nomethod *&rest args* [method]

:objects *&rest args* [method]

irtviewer-no-window [Class]

:super **propertied-object**
:slots irtviewer

:init *&rest args* [method]

:create *&rest args* [method]

:resize *newwidth newheight* [method]

:nomethod *&rest args* [method]
:nomethod *&rest args* [method]

:resize *newwidth newheight* [method]

:create *&rest args* [method]

:init *&rest args* [method]

make-irtviewer *&rest args* [function]

Create irtviewer
:view-name title
:hither near cropping plane
:yon far cropping plane
:width width of the window
:height height of the window
:draw-origin size of origin arrow, use nil to disable it
:draw-floor use t to view floor
:floor-color floor color. default is #f(1 1 1), i.e. white.

x::make-lr-ud-coords *x::lra x::uda* [function]

x::make-mpeg-from-images *x::mpgfile x::images &key (delete t) (x::delay 1)* [function]

x::make-animgif-from-images *x::giffile x::images &key (delete t) x::transparent (loop t) (x::delay 10) (x::background 000000)* [function]

x::draw-things *x::objs* [function]

objects *&optional (objs t) vw* [function]

make-irtviewer-dummy *&rest args* [function]

geometry::default-pixmapsurface *&rest args* [function]

make-irtviewer-no-window *&rest args* [function]

20 Interference Calculation

20.1 Interference Calculation Overview

Interference calculation is the process of judging whether two sets of geometric models intersect and finding the distance between them. It mainly provides the following two functions:

- Collision detection to determine if two models intersect (collision-check function)
- A distance calculator that calculates the shortest distance between two models (collision-distance function)

In irteus, interference calculations are performed by calling external libraries through other language interfaces. As an external library, calls to PQP and Bullet are implemented, and PQP is used by default. The library to be used can be switched by the select-collision-algorithm function as follows:

```
(select-collision-algorithm *collision-algorithm-pqp*) ;; use PQP
(select-collision-algorithm *collision-algorithm-bullet*) ;; use Bullet
```

The features of each external library are described in detail below. See http://gamma.cs.unc.edu/research/collision/ for other interference calculation software packages. (Note that the information may be outdated. For example, Bullet is not listed.)

cascaded-coords [Class]

:super	**coordinates**
:slots	rot pos parent descendants worldcoords manager changed

:make-collisionmodel *&rest args &key &allow-other-keys* [method]

Make collision model and save pointer.

collision-distance *model1 model2 &rest args &key &allow-other-keys* [function]

Calculate collision distance between model1 and model2.

Return value is (list [distance] [nearest point on model1] [nearest point on model2]).

collision-check *model1 model2 &rest args* [function]

Check collision between model1 and model2.

If return value is 0, no collision.

Otherwise (return value is 1), collision.

collision-check-objects *obj1 obj2 &rest args &key &allow-other-keys* [function]

Check collision between obj1 and obj2.

obj1 and obj2 should be list of models.

If return value is nil, no collision.

Otherwise (return value is t), collision.

select-collision-algorithm *alg* [function]

Select collision algorithm.

:pqp and :bullet are supported.

20.1.1 Interference Calculation Example Between Object Shape Models

Below is an example of using collision-check and collision-distance to detect the collision between two cubes, calculate the distance, and draw a line connecting the nearest points. The specifications of the closest point obtained by the collision-distance function when interference occurs are different between PQP and Bullet. For details, see the description of Bullet below:

```
;; Make models
(setq *b0* (make-cube 100 100 100))
(setq *b1* (make-cube 100 100 100))

;; Case 1 : no collision
(send *b0* :newcoords (make-coords :pos #f(100 100 -100)
                                   :rpy (list (deg2rad 10) (deg2rad -20) (deg2rad 30))))
(objects (list *b0* *b1*))
(print (collision-check *b0* *b1*)) ;; Check collision
(let ((ret (collision-distance *b0* *b1*))) ;; Check distance and nearest points
  (print (car ret)) ;; distance
  (send (cadr ret) :draw-on :flush nil :size 20 :color #f(1 0 0)) ;; nearest point on *b0*
  (send (caddr ret) :draw-on :flush nil :size 20 :color #f(1 0 0)) ;; nearest point on *b1*
  (send *irtviewer* :viewer :draw-line (cadr ret) (caddr ret))
  (send *irtviewer* :viewer :viewsurface :flush))

;; Case 2 : collision
(send *b0* :newcoords (make-coords :pos #f(50 50 -50)
                                   :rpy (list (deg2rad 10) (deg2rad -20) (deg2rad 30))))
(objects (list *b0* *b1*))
(print (collision-check *b0* *b1*)) ;; Check collision
(let ((ret (collision-distance *b0* *b1*))) ;; Check distance and nearest points
  (print (car ret)) ;; distance
  ;; In case of collision, nearest points are insignificant values.
  (send (cadr ret) :draw-on :flush nil :size 20 :color #f(1 0 0)) ;; nearest point on *b0*
  (send (caddr ret) :draw-on :flush nil :size 20 :color #f(1 0 0)) ;; nearest point on *b1*
  (send *irtviewer* :viewer :draw-line (cadr ret) (caddr ret))
  (send *irtviewer* :viewer :viewsurface :flush))
```

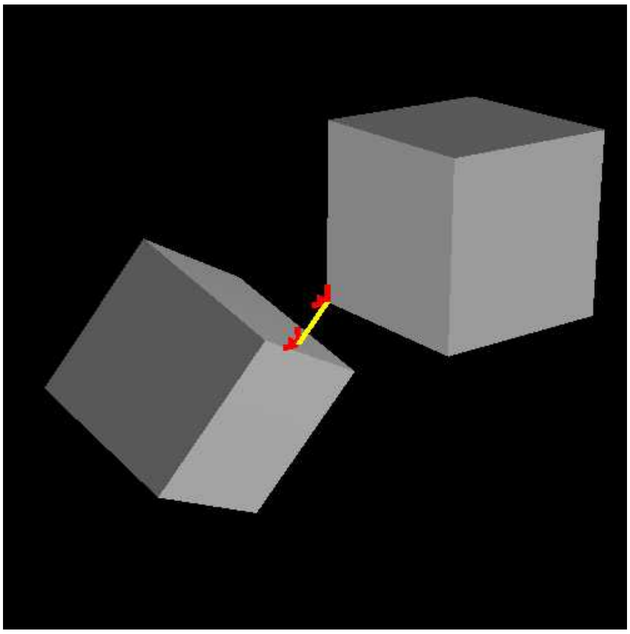


図 29: Collision detection

20.1.2 Robot Motion and Interference Calculation

When performing a static simulation of the action of grabbing an object with the hand, it is possible to investigate the interference between the links of the hand (fingers) and the target object, and stop the action of grabbing the object when this occurs.

```
(load "irteus/demo/sample-arm-model.l")
(setq *sarm* (instance sarmclass :init))
(send *sarm* :reset-pose)
(setq a 42)
(send *sarm* :move-fingers a)
(setq *target* (make-cube 30 30 30))
(send *target* :translate #f(350 200 400))
(objects (list *sarm* *target*))

(send *sarm* :inverse-kinematics *target* :move-target (send *sarm* :end-coords) :debug-view t)
(while (> a 0)
  (if (collision-check-objects
       (list (send *sarm* :joint-fr :child-link)
             (send *sarm* :joint-fl :child-link))
       (list *target*))
      (return))
  (decf a 0.1)
  (send *irtviewer* :draw-objects)
  (send *sarm* :move-fingers a))
(send *sarm* :end-coords :assoc *target*)

(dotimes (i 100)
  (send *sarm* :joint0 :joint-angle 1 :relative t)
  (send *irtviewer* :draw-objects))
(send *sarm* :end-coords :dissoc *target*)
(dotimes (i 100)
  (send *sarm* :joint0 :joint-angle -1 :relative t)
  (send *irtviewer* :draw-objects))
```

Similar functionality is provided by the methods :open-hand and :close-hand in the "irteus/demo/sample-arm-model.l" file.

20.2 Interference Calculation by PQP

PQP is an interference calculation library developed by Lin et al.'s group at the University of North Carolina. The usage of the PQP software package is described in irteus/PQP/README.txt, and can be understood by reading irteus/PQP/src/PQP.h.

The files for using PQP with irteus consist of CPQP.C, euspqp.c, and pqp.l. To determine if two geometric models will collide,

```
(defun pqp-collision-check (model1 model2
       &optional (flag PQP_FIRST_CONTACT) &key (fat 0) (fat2 nil))
  (let ((m1 (get model1 :pqpmodel))  (m2 (get model2 :pqpmodel))
        (r1 (send model1 :worldrot)) (t1 (send model1 :worldpos))
        (r2 (send model2 :worldrot)) (t2 (send model2 :worldpos)))
    (if (null fat2) (setq fat2 fat))
    (if (null m1) (setq m1 (send model1 :make-pqpmodel :fat fat)))
    (if (null m2) (setq m2 (send model2 :make-pqpmodel :fat fat2)))
    (pqpcollide r1 t1 m1 r2 t2 m2 flag)))
```

should be called. r1,r1,r2,t1 are the translation vector and rotation matrix of each object, and (get model1 :pqpmodel) refers to the pointer to the PQP geometric model. This pointer is computed in the :make-pqpmodel method as follows.

```
(defmethod cascaded-coords
  (:make-pqpmodel
   (&key (fat 0))
   (let ((m (pqpmakemodel))
         vs v1 v2 v3 (id 0))
     (setf (get self :pqpmodel) m)
     (pqpbeginmodel m)
     (dolist (f (send self :faces))
       (dolist (poly (face-to-triangle-aux f))
         (setq vs (send poly :vertices)
               v1 (send self :inverse-transform-vector (first vs))
               v2 (send self :inverse-transform-vector (second vs))
               v3 (send self :inverse-transform-vector (third vs)))
         (when (not (= fat 0))
           (setq v1 (v+ v1 (scale fat (normalize-vector v1)))
                 v2 (v+ v2 (scale fat (normalize-vector v2)))
                 v3 (v+ v3 (scale fat (normalize-vector v3)))))
         (pqpaddtri m v1 v2 v3 id)
         (incf id)))
     (pqpendmodel m)
     m)))
```

Here, (pqpmakemodel) is called first. In pqpmakemodel, defined in euqpqp.c,

```
pointer PQPMAKEMODEL(register context *ctx, int n, register pointer *argv)
{
    int addr = PQP_MakeModel();
    return makeint(addr);
}
```

is invoked, which is the same as in CPQP.C

```
PQP\_Model *PQP_MakeModel()
{
    return new PQP_Model();
}
```

PQP_Model() is defined in PQP.h. After the functions in euslisp are passed to the actual PQP library functions in this way, an instance of the PQP geometric model is created with (pqpbeginmodel m) and the surface information is registered as (pqpaddtri m v1 v2 v3 id) to register the surface information.

cascaded-coords [Class]

:super **coordinates**
:slots rot pos parent descendants worldcoords manager changed

:make-pqpmodel *&key (geometry::fat 0) ((:faces geometry::fs) (send self :faces))* [method]

pqp-collision-check [function]

geometry::model1 geometry::model2 &optional (geometry::flag geometry::pqp_first_contact) &
(geometry::fat2 nil)

Check collision between model1 and model2 using PQP.
If return value is 0, no collision.
Otherwise (return value is 1), collision.

pqp-collision-distance [function]

geometry::model1 geometry::model2 &key (geometry::fat 0)
(geometry::fat2 nil)
(geometry::qsize 2)

Calculate collision distance between model1 and model2 using PQP.
Return value is (list [distance] [nearest point on model1] [nearest point on model2]).
If collision occurs, [distance] is 0 and nearest points are insignificant values.

pqp-collision-check-objects *geometry::obj1 geometry::obj2 &key (geometry::fat 0.2)* [function]

return nil or t

20.3 Interference Calculation with Bullet

Bullet is a physics engine, and as part of it, an interference calculation function is provided. The files for using Bullet with irteus consist of CBULLET.cpp, eusbullet.c, and bullet.l. The calling order inside the function is the same as PQP.

The differences between PQP and Bullet are as follows.

- If you call collision-distance when there is interference, PQP will return 0 as the distance and a meaningless point as the closest point. Bullet, on the other hand, returns the shortest distance (called penetration depth) that should be moved to eliminate interference as the distance. Also, as the closest point, the points at both ends of the shortest distance to eliminate interference are returned.
- PQP can handle non-convex mesh models as is, but Bullet internally computes and handles the convex hull of non-convex models.

cascaded-coords [Class]

:super **coordinates**
:slots rot pos parent descendants worldcoords manager changed

:make-btmodel [method]

&key (geometry::fat 0)
((:faces geometry::fs))

Make bullet model and save pointer of the bullet model.

geometry::bt-make-model-from-body [function]

geometry::b &key (csg (send geometry::b :csg)) (geometry::margin 0)

Make bullet model from body.

bt-collision-distance [function]

geometry::model1 geometry::model2 &key (geometry::fat 0) (geometry::fat2 nil) (geometry::qsize)

Calculate collision distance between model1 and model2 using Bullet.
Return value is (list [distance] [nearest point on model1] [nearest point on model2]).
qsize argument is not used, just for compatibility with pqp-collision-distance.

bt-collision-check [function]

geometry::model1 geometry::model2 &key (geometry::fat 0) (geometry::fat2 nil)

Check collision between model1 and model2 using Bullet.
If return value is 0, no collision.
Otherwise (return value is 1), collision.

21 BVH Data

bvh-link [Class]

:super	**bodyset-link**
:slots	type offset channels neutral

:init *name typ offst chs parent children* [method]

create link for bvh model

:type [method]

:offset [method]

:channels [method]

:init *name typ offst chs parent children* [method]

create link for bvh model

:channels [method]

:offset [method]

:type [method]

bvh-sphere-joint [Class]

:super	**sphere-joint**

:slots axis-order bvh-offset-rotation

:init [method]

&rest args &key (order (list :z :x :y))
((:bvh-offset-rotation bvh-rotation) (unit-matrix 3))
&allow-other-keys

create joint for bvh model

:joint-angle-bvh *&optional v* [method]

:joint-angle-bvh-offset *&optional v* [method]

:joint-angle-bvh-impl *v bvh-offset* [method]

:axis-order [method]

:bvh-offset-rotation [method]

:init [method]

&rest args &key (order (list :z :x :y))
((:bvh-offset-rotation bvh-rotation) (unit-matrix 3))
&allow-other-keys

create joint for bvh model

:bvh-offset-rotation [method]

:axis-order [method]

:joint-angle-bvh-impl *v bvh-offset* [method]

:joint-angle-bvh-offset *&optional v* [method]

:joint-angle-bvh *&optional v* [method]

bvh-6dof-joint [Class]

:super **6dof-joint**
:slots scale axis-order bvh-offset-rotation

:init *&rest args &key (order (list :x :y :z :z :x :y)) ((:scale scl)) ((:bvh-offset-rotation bvh-rotation) (unit-matrix 3)) &allow-other-keys* [method]

:joint-angle-bvh *&optional v* [method]

:joint-angle-bvh-offset *&optional v* [method]

:joint-angle-bvh-impl *v bvh-offset* [method]

:axis-order [method]

:bvh-offset-rotation [method]
:bvh-offset-rotation [method]

:axis-order [method]

:joint-angle-bvh-impl *v bvh-offset* [method]

:joint-angle-bvh-offset *&optional v* [method]

:joint-angle-bvh *&optional v* [method]

:init *&rest args &key (order (list :x :y :z :z :x :y)) ((:scale scl)) ((:bvh-offset-rotation bvh-rotation) (unit-matrix 3)) &allow-other-keys* [method]

bvh-robot-model [Class]

:super	**robot-model**
:slots	nil

:init [method]

&rest args &key tree
coords
((:scale scl))

create robot model for bvh model

:make-bvh-link *tree &key parent ((:scale scl))* [method]

:angle-vector *&optional vec (angle-vector (instantiate float-vector (calc-target-joint-dimension joint-list)))* [method]

:dump-joints *links &key (depth 0) (strm *standard-output*)* [method]

:dump-hierarchy *&optional (strm *standard-output*)* [method]

:dump-motion *&optional (strm *standard-output*)* [method]

:copy-state-to *robot* [method]

:fix-joint-angle *i limb joint-name joint-order a* [method]

:fix-joint-order *jo limb* [method]

:bvh-offset-rotate *name* [method]

:init-end-coords [method]

:init-root-link [method]

:init [method]

&rest args &key tree
coords
((:scale scl))

create robot model for bvh model

:bvh-offset-rotate *name* [method]

:fix-joint-order *jo limb* [method]

:fix-joint-angle *i limb joint-name joint-order a* [method]

:copy-state-to *robot* [method]

:dump-motion *&optional (strm *standard-output*)* [method]

:dump-hierarchy *&optional (strm *standard-output*)* [method]

:dump-joints *links &key (depth 0) (strm *standard-output*)* [method]

:angle-vector *&optional vec (angle-vector (instantiate float-vector (calc-target-joint-dimension joint-list)))* [method]

:make-bvh-link *tree &key parent ((:scale scl))* [method]

motion-capture-data [Class]

:super	**propertied-object**
:slots	frame model animation

:init *fname &key (coords (make-coords)) ((:scale scl))* [method]

:model *&rest args* [method]

:animation *&rest args* [method]

:frame *&optional f* [method]

:frame-length [method]

:animate *&rest args &key (start 0) (step 1) (end (send self :frame-length)) (interval 20) &allow-other-keys* [method]
:animate *&rest args &key (start 0) (step 1) (end (send self :frame-length)) (interval 20) &allow-other-keys* [method]

:frame-length [method]

:frame *&optional f* [method]

:animation *&rest args* [method]

:model *&rest args* [method]

:init *fname &key (coords (make-coords)) ((:scale scl))* [method]
:init-root-link [method]

:init-end-coords [method]

rikiya-bvh-robot-model [Class]

:super **bvh-robot-model**
:slots nil

:init *&rest args* [method]

:larm-collar *&rest args* [method]

:larm-shoulder *&rest args* [method]

:larm-elbow *&rest args* [method]

:larm-wrist *&rest args* [method]

:rarm-collar *&rest args* [method]

:rarm-shoulder *&rest args* [method]

:rarm-elbow *&rest args* [method]

:rarm-wrist *&rest args* [method]

:lleg-crotch *&rest args* [method]

:lleg-knee *&rest args* [method]

:lleg-ankle *&rest args* [method]

:rleg-crotch *&rest args* [method]

:rleg-knee *&rest args* [method]

:rleg-ankle *&rest args* [method]

:torso-chest *&rest args* [method]

:head-neck *&rest args* [method]

:copy-joint-to *robot limb joint &optional (sign 1)* [method]

:copy-state-to *robot* [method]
:copy-state-to *robot* [method]

:copy-joint-to *robot limb joint &optional (sign 1)* [method]

:head-neck *&rest args* [method]

:torso-chest *&rest args* [method]

:rleg-ankle *&rest args* [method]

:rleg-knee *&rest args* [method]

:rleg-crotch *&rest args* [method]

:lleg-ankle *&rest args* [method]

:lleg-knee *&rest args* [method]

:lleg-crotch *&rest args* [method]

:rarm-wrist *&rest args* [method]

:rarm-elbow *&rest args* [method]

:rarm-shoulder *&rest args* [method]

:rarm-collar *&rest args* [method]

:larm-wrist *&rest args* [method]

:larm-elbow *&rest args* [method]

:larm-shoulder *&rest args* [method]

:larm-collar *&rest args* [method]

:init *&rest args* [method]

tum-bvh-robot-model [Class]

:super **bvh-robot-model**
:slots nil

:init *&rest args* [method]
:init *&rest args* [method]

cmu-bvh-robot-model [Class]

:super **bvh-robot-model**
:slots nil

:init *&rest args* [method]
:init *&rest args* [method]

read-bvh *fname &key scale* [function]

read bvh file

bvh2eus [function]

fname &rest args &key ((:objects obj) nil)
&allow-other-keys

read bvh file and anmiate robot model in the viewer for Supported bvh data, such as

- CMU motion capture database
(https://sites.google.com/a/cgspeed.com/cgspeed/motion-capture/cmu-bvh-conversion)

- The TUM Kitchen Data Set
(http://ias.cs.tum.edu/download/kitchen-activity-data)

Use

(tum-bvh2eus "Take 005.bvh") ;; tum
(rikiya-bvh2eus "A01.bvh") ;; rikiya
(cmu-bvh2eus "01_01.bvh") ;; cmu

Other Sites are:
(http://www.mocapdata.com/page.cgi?p=free_motions)
(http://www.motekentertainment.com/)
(http://www.mocapclub.com/Pages/Library.htm)

(bvh2eus "poses.bvh")

load-mcd [function]

fname &key (scale)
(coords)
(bvh-robot-model-class bvh-robot-model)
&allow-other-keys

load motion capture data

rikiya-bvh2eus *fname &rest args* [function]

read rikiya bvh file and anmiate robot model in the viewer
(rikiya-bvh2eus "A01.bvh")

cmu-bvh2eus *fname &rest args* [function]

read cmu bvh file and anmiate robot model in the viewer

CMU motion capture database
(https://sites.google.com/a/cgspeed.com/cgspeed/motion-capture/cmu-bvh-conversion)

(cmu-bvh2eus "01_01.bvh" :scale 100.0)

tum-bvh2eus *fname &rest args* [function]

read tum file and anmiate robot model in the viewer

The TUM Kitchen Data Set
(http://ias.cs.tum.edu/download/kitchen-activity-data)

(tum-bvh2eus "Take 005.bvh" :scale 10.0)

parse-bvh-sexp *src &key ((:scale scl))* [function]
make-bvh-robot-model *bvh-data &rest args* [function]
rikiya-file *&key (num 1) (cls 'a)* [function]
tum-kitchen-file *&key (num 1) (cls 0)* [function]
cmu-mocap-file *&key (num 1) (cls 1)* [function]

22 Collada Data

collada::eusmodel-description *collada::model* [function]
convert a 'model' to eusmodel-description

collada::eusmodel-link-specs *collada::links* [function]
convert 'links' to <link-specs>

collada::eusmodel-joint-specs *collada::joints* [function]
convert 'joints' to <joint-specs>

collada::eusmodel-description->collada *name collada::description &key (scale 0.001)* [function]
convert eusmodel-descrption to collada sxml

collada::matrix->collada-rotate-vector *collada::mat* [function]
convert a rotation matrix to axis-angle.

convert-irtmodel-to-collada *collada::model-file &optional (collada::output-full-dir (send (truename ./) :namestring)) (collada::model-name) (collada::exit-p t)* [function]
convert irtmodel to collada model. (convert-irtmodel-to-collada irtmodel-file-path &optional (output-full-dir (send (truename "./") :namestring)) (model-name))

collada::symbol->string *collada::sym* [function]
collada::->string *collada::val* [function]
collada::string-append *&rest args* [function]
collada::make-attr *collada::l collada::ac* [function]
collada::make-xml *collada::x collada::bef collada::aft* [function]
collada::sxml->xml *collada::sxml* [function]
collada::xml-output-to-string-stream *collada::ss collada::l* [function]
collada::cat-normal *collada::l collada::s* [function]
collada::cat-clark *collada::l collada::s collada::i* [function]
collada::verificate-unique-strings *names* [function]
collada::eusmodel-link-spec *collada::link* [function]
collada::eusmodel-mesh-spec *collada::link collada::rt-cds* [function]
collada::eusmodel-joint-spec *collada::_joint* [function]
collada::eusmodel-limit-spec *collada::_joint* [function]
collada::eusmodel-endcoords-specs *collada::model* [function]
collada::eusmodel-link-description *collada::description* [function]
collada::eusmodel-joint-description *collada::description* [function]

collada::eusmodel-endcoords-description *collada::description* [function]

collada::setup-collada-filesystem *collada::obj-name collada::base-dir* [function]

collada::range2 *collada::n* [function]

eus2collada *collada::obj collada::full-root-dir &key (scale 0.001) (collada::file-suffix .dae)* [function]

collada::collada-node-id *collada::link-descrption* [function]

collada::collada-node-name *collada::link-descrption* [function]

collada::links-description->collada-library-materials *collada::links-desc* [function]

collada::link-description->collada-materials *collada::link-desc* [function]

collada::mesh-description->collada-material *collada::mat collada::effect* [function]

collada::links-description->collada-library-effects *collada::links-desc* [function]

collada::link-description->collada-effects *collada::link-desc* [function]

collada::mesh-description->collada-effect *collada::mesh id* [function]

collada::matrix->collada-string *collada::mat* [function]

collada::find-parent-liks-from-link-description *collada::target-link collada::desc* [function]

collada::eusmodel-description->collada-node-transformations *collada::target-link collada::desc* [function]

collada::eusmodel-description->collada-node *collada::target-link collada::desc* [function]

collada::eusmodel-description->collada-library-visual-scenes *name collada::desc* [function]

collada::mesh-description->instance-material *collada::s* [function]

collada::link-description->collada-bind-material *collada::link* [function]

collada::eusmodel-description->collada-library-kinematics-scenes *name collada::desc* [function]

collada::eusmodel-description->collada-library-kinematics-models *name collada::desc* [function]

collada::eusmodel-description->collada-kinematics-model *name collada::desc* [function]

collada::eusmodel-description->collada-library-physics-scenes *name collada::desc* [function]

collada::eusmodel-description->collada-library-physics-models *name collada::desc* [function]

collada::find-root-link-from-joints-description *collada::joints-description* [function]

collada::find-link-from-links-description *name collada::links-description* [function]

collada::eusmodel-description->collada-links *collada::description* [function]

collada::find-joint-from-link-description *collada::target collada::joints* [function]

collada::find-child-link-descriptions *collada::parent collada::links collada::joints* [function]

collada::eusmodel-description->collada-library-articulated-systems *collada::desc name* [function]

collada::eusmodel-endcoords-description->openrave-manipulator *collada::end-coords collada::description* [function]

collada::eusmodel-description->collada-links-tree *collada::target collada::links collada::joints* [function]

collada::joints-description->collada-instance-joints *collada::joints-desc* [function]

collada::joint-description->collada-instance-joint *collada::joint-desc* [function]

collada::eusmodel-description->collada-library-joints *collada::description* [function]

collada::joints-description->collada-joints *collada::joints-description* [function]

collada::collada-joint-id *collada::joint-description* [function]

collada::joint-description->collada-joint *collada::joint-description* [function]

collada::linear-joint-description->collada-joint *collada::joint-description* [function]

collada::rotational-joint-description->collada-joint *collada::joint-description* [function]

collada::eusmodel-description->collada-scene *collada::description* [function]

collada::eusmodel-description->collada-library-geometries *collada::description* [function]

collada::collada-geometry-id-base *collada::link-descrption* [function]

collada::collada-geometry-name-base *collada::link-descrption* [function]

collada::links-description->collada-geometries *collada::links-description* [function]

collada::mesh-object->collada-mesh *collada::mesh id* [function]

collada::link-description->collada-geometry *collada::link-description* [function]

collada::mesh->collada-indices *collada::mesh* [function]

collada::mesh-vertices->collada-string *collada::mesh* [function]

collada::mesh-normals->collada-string *collada::mesh* [function]

collada::float-vector->collada-string *collada::v* [function]

collada::enum-integer-list *collada::n* [function]

collada::search-minimum-rotation-matrix *collada::mat* [function]

collada::estimate-class-name *collada::model-file* [function]

collada::remove-directory-name *fname* [function]

23 Point Cloud Data

pointcloud [Class]

:super **cascaded-coords**

:slots parray carray narray curvature pcolor psize awidth asize box height width view-coord

:init [method]

&rest args &key ((:points mat))
((:colors cary))
((:normals nary))
((:curvatures curv))
((:height ht))
((:width wd))
(point-color (float-vector 0 1 0))
(point-size 2.0)
(fill)
(arrow-width 2.0)
(arrow-size 0.0)

Create point cloud object

:size-change *&optional wd ht* [method]

change width and height, this method does not change points data

:points *&optional pts wd ht* [method]

replace points, pts should be list of points or n*times* matrix

:colors *&optional cls* [method]

replace colors, cls should be list of points or n*times* matrix

:normals *&optional nmls* [method]

replace normals by, nmls should be list of points or n*times*3 matrix

:point-list *&optional (remove-nan)* [method]

return list of points

:color-list [method]

return list of colors

:normal-list [method]

return list of normals

:centroid [method]

retrun centroid of this point cloud

:append *point-list &key (create t)* [method]

append point cloud list to this point cloud.

if :create is true, return appended point cloud and original point cloud does not change.

:filter [method]

&rest args &key create
&allow-other-keys

this method can take the same keywords with :filter-with-indices method.

if :create is true, return filtered point cloud and original point cloud does not change.

:filter-with-indices [method]

idx-lst &key (create)
(negative)

filter point cloud with list of index (points which are indicated by indices will remain).

if :create is true, return filtered point cloud and original point cloud does not change.

if :negative is true, points which are indicated by indices will be removed.

:filtered-indices [method]

&key key
ckey
nkey
pckey
pnkey
pcnkey
negative
&allow-other-keys

create list of index where filter function retrun true.

key, ckey, nkey are filter function for points, colors, normals. They are expected to take one argument and return t or nil.

pckey, pnkey are filter function for points and colors, points and normals. They are expected to take two arguments and return t or nil.

pcnkey is filter function for points, colors and normals. It is expected to take three arguments and return t or nil.

:step [method]

step &key (fixsize)
(create)

downsample points with step

:copy-from *pc* [method]

update object by pc

:transform-points *coords &key (create)* [method]

transform points and normals with coords.

if :create is true, return transformed point cloud and original point cloud does not change.

:convert-to-world *&key (create)* [method]

transform points and normals with self coords. converted points data should be at the same position as displayed

:move-origin-to *neworigin &key (create)* [method]

origin of point cloud is moved to neworigin. moved points data should be at the same position as displayed

:reset-box [method]

:box [method]

:vertices [method]

:size [method]

:width [method]

:height [method]

:view-coords *&optional vc* [method]

:curvatures *&optional curv* [method]

:curvature-list [method]

:set-color *col &optional (_transparent)* [method]

:point-color *&optional pc* [method]

:point-size *&optional ps* [method]

:axis-length *&optional al* [method]

:axis-width *&optional aw* [method]

:clear-color [method]

:clear-normal [method]

:nfilter *&rest args* [method]

:viewangle-inlier *&key (min-z 0.0) (hangle 44.0) (vangle 35.0)* [method]

:image-position-inlier *&key (ipkey) (height 144) (width 176) (cy (/ (float (- height 1)) 2)) (cx (/ (float (- width 1)) 2))* *negative* [method]

:image-circle-filter *dist &key (height 144) (width 176) (cy (/ (float (- height 1)) 2)) (cx (/ (float (- width 1)) 2)) create negative* [method]

:step-inlier *step offx offy* [method]

:generate-color-histogram-hs *&key (h-step 9) (s-step 7) (hlimits (cons 360.0 0.0)) (vlimits (cons 1.0 0.15)) (slimits (cons 1.0 0.25)) (rotate-hue) (color-scale 255.0) (sizelimits 1)* [method]

:set-offset *cds &key (create)* [method]

:drawnormalmode *&optional mode* [method]

:transparent *&optional trs* [method]

:draw *vwer* [method]

:move-origin-to *neworigin &key (create)* [method]

origin of point cloud is moved to neworigin. moved points data should be at the same position as displayed

:convert-to-world *&key (create)* [method]

transform points and normals with self coords. converted points data should be at the same position as displayed

:transform-points *coords &key (create)* [method]

transform points and normals with coords.

if :create is true, return transformed point cloud and original point cloud does not change.

:copy-from *pc* [method]

update object by pc

:step [method]

step &key (fixsize)
(create)

downsample points with step

:filtered-indices [method]

&key key
ckey
nkey
pckey
pnkey
pcnkey
negative
&allow-other-keys

create list of index where filter function retrun true.

key, ckey, nkey are filter function for points, colors, normals. They are expected to take one argument and return t or nil.

pckey, pnkey are filter function for points and colors, points and normals. They are expected to take two arguments and return t or nil.

pcnkey is filter function for points, colors and normals. It is expected to take three arguments and return t or nil.

:filter-with-indices [method]

idx-lst &key (create)
(negative)

filter point cloud with list of index (points which are indicated by indices will remain).

if :create is true, return filtered point cloud and original point cloud does not change.

if :negative is true, points which are indicated by indices will be removed.

:filter [method]

&rest args &key create
&allow-other-keys

this method can take the same keywords with :filter-with-indices method.

if :create is true, return filtered point cloud and original point cloud does not change.

:append *point-list &key (create t)* [method]

append point cloud list to this point cloud.

if :create is true, return appended point cloud and original point cloud does not change.

:centroid [method]

retrun centroid of this point cloud

:normal-list [method]

return list of normals

:color-list [method]

return list of colors

:point-list *&optional (remove-nan)* [method]

return list of points

:normals *&optional nmls* [method]

replace normals by, nmls should be list of points or n$times$3 matrix

:colors *&optional cls* [method]

replace colors, cls should be list of points or n$times$ matrix

:points *&optional pts wd ht* [method]

replace points, pts should be list of points or n$times$ matrix

:size-change *&optional wd ht* [method]

change width and height, this method does not change points data

:init [method]

&rest args &key ((:points mat))
((:colors cary))
((:normals nary))
((:curvatures curv))
((:height ht))
((:width wd))
(point-color (float-vector 0 1 0))
(point-size 2.0)
(fill)
(arrow-width 2.0)
(arrow-size 0.0)

Create point cloud object

:draw *vwer* [method]

:transparent *&optional trs* [method]

:drawnormalmode *&optional mode* [method]

:set-offset *cds &key (create)* [method]

:generate-color-histogram-hs *&key (h-step 9) (s-step 7) (hlimits (cons 360.0 0.0)) (vlimits (cons 1.0 0.15)) (slimits (cons 1.0 0.25)) (rotate-hue) (color-scale 255.0) (sizelimits 1)* [method]

:step-inlier *step offx offy* [method]

:image-circle-filter *dist &key (height 144) (width 176) (cy (/ (float (- height 1)) 2)) (cx (/ (float (- width 1)) 2)) create negative* [method]

:image-position-inlier *&key (ipkey) (height 144) (width 176) (cy (/ (float (- height 1)) 2)) (cx (/ (float (- width 1)) 2)) negative* [method]

:viewangle-inlier *&key (min-z 0.0) (hangle 44.0) (vangle 35.0)* [method]

:nfilter *&rest args* [method]

:clear-normal [method]

:clear-color [method]

:axis-width *&optional aw* [method]

:axis-length *&optional al* [method]

:point-size *&optional ps* [method]

:point-color *&optional pc* [method]

:set-color *col &optional (_transparent)* [method]

:curvature-list [method]

:curvatures *&optional curv* [method]

:view-coords *&optional vc* [method]

:height [method]

:width [method]

:size [method]

:vertices [method]

:box [method]

:reset-box [method]
make-random-pointcloud *&key (num 1000) (with-color) (with-normal) (scale 100.0)* [function]

24 Graph Representation

node [Class]

:super	**propertied-object**
:slots	arc-list image

:init *n &optional image* [method]

:arc-list [method]

:successors [method]

:add-arc *a* [method]

:remove-arc *a* [method]

:remove-all-arcs [method]

:unlink *n* [method]

:image *&optional im* [method]

:add-neighbor *n &optional a* [method]

:neighbors *&optional args* [method]
:image *&optional im* [method]

:unlink *n* [method]

:remove-all-arcs [method]

:remove-arc *a* [method]

:add-arc *a* [method]

:successors [method]

:arc-list [method]

:init *n &optional image* [method]

arc [Class]

:super	**propertied-object**
:slots	from to

:init *from_ to_* [method]

:from [method]

:to [method]

:prin1 *&optional (strm t) &rest msgs* [method]

:prin1 *&optional (strm t) &rest msgs* [method]

:to [method]

:from [method]

:init *from_ to_* [method]

directed-graph [Class]

:super	**propertied-object**
:slots	nodes

:write-to-dot-stream *&optional (strm *standard-output*) result-path (title output)* [method]

write graph structure to stream as dot(graphviz) style

Args:

strm: stream class for output

result-path: list of solver-node, it's result of (send solver :solve graph)

title: title of graph

:write-to-dot *fname &optional result-path (title output)* [method]

write graph structure to dot(graphviz) file
Args:
fname: filename for output
result-path: list of solver-node, it's result of (send solver :solve graph)
title: title of graph

:write-to-file *basename &optional result-path title (type pdf)* [method]
write graph structure to various type of file
Args:
basename: basename for output (output filename is 'basename.type')
result-path: list of solver-node, it's result of (send solver :solve graph)
title: title of graph
type: type of output

:write-to-pdf *fname &optional result-path (title (string-right-trim .pdf fname))* [method]
write graph structure to pdf
Args:
fname: filename for output
result-path: list of solver-node, it's result of (send solver :solve graph)
title: title of graph

:original-draw-mode [method]
change draw-mode to original mode

:current-draw-mode [method]
change draw-mode to latest mode

:draw-both-arc *&optional (bothq :both)* [method]
change draw-mode, if true is set, draw bidirectional arc as two arcs

:draw-arc-label *&optional (writeq :write)* [method]
change draw-mode, if true is set, draw label(name) of arcs

:draw-merged-result *&optional (mergeq :merge)* [method]
change draw-mode, if true is set, draw result arc as red. if not, draw red arc independently

:write-to-png *fname &optional result-path (title (string-right-trim .png fname))* [method]
write graph structure to png
Args:
fname: filename for output
result-path: list of solver-node, it's result of (send solver :solve graph)
title: title of graph

:init [method]

:successors *node &rest args* [method]

:node *name* [method]

:nodes *&optional arg* [method]

:add-node *n* [method]

:remove-node *n* [method]

:clear-nodes [method]

:add-arc-from-to *from to* [method]

:remove-arc *arc* [method]

:adjacency-matrix [method]

:adjacency-list [method]
:adjacency-list [method]

:adjacency-matrix [method]

:remove-arc *arc* [method]

:add-arc-from-to *from to* [method]

:clear-nodes [method]

:remove-node *n* [method]

:add-node *n* [method]

:nodes *&optional arg* [method]

:node *name* [method]

:successors *node &rest args* [method]

:init [method]

costed-arc [Class]

:super	**arc**
:slots	cost

:init *from to c* [method]

:cost [method]
:cost [method]

:init *from to c* [method]

costed-graph [Class]

:super **directed-graph**
:slots nil

:add-arc *from to cost &key (both nil)* [method]

:add-arc-from-to *from to cost &key (both nil)* [method]

:path-cost *from arc to* [method]
:path-cost *from arc to* [method]

:add-arc-from-to *from to cost &key (both nil)* [method]

:add-arc *from to cost &key (both nil)* [method]

graph [Class]

:super **costed-graph**
:slots start-state goal-state

:goal-test *gs* [method]

:path-cost *from arc to* [method]

:start-state *&optional arg* [method]

:goal-state *&optional arg* [method]

:add-arc *from to &key (both nil)* [method]

:add-arc-from-to *from to &key (both nil)* [method]
:add-arc-from-to *from to &key (both nil)* [method]

:add-arc *from to &key (both nil)* [method]

:goal-state *&optional arg* [method]

:start-state *&optional arg* [method]

:path-cost *from arc to* [method]

:goal-test *gs* [method]

:draw-merged-result *&optional (mergeq :merge)* [method]

change draw-mode, if true is set, draw result arc as red. if not, draw red arc independently

:draw-arc-label *&optional (writeq :write)* [method]

change draw-mode, if true is set, draw label(name) of arcs

:draw-both-arc *&optional (bothq :both)* [method] change draw-mode, if true is set, draw bidirectional arc as two arcs

:current-draw-mode [method]

change draw-mode to latest mode

:original-draw-mode [method]

change draw-mode to original mode

:write-to-png *fname &optional result-path (title (string-right-trim .png fname))* [method]

write graph structure to png
Args:
fname: filename for output
result-path: list of solver-node, it's result of (send solver :solve graph)
title: title of graph

:write-to-pdf *fname &optional result-path (title (string-right-trim .pdf fname))* [method]

write graph structure to pdf
Args:
fname: filename for output
result-path: list of solver-node, it's result of (send solver :solve graph)
title: title of graph

:write-to-file *basename &optional result-path title (type pdf)* [method]

write graph structure to various type of file
Args:
basename: basename for output (output filename is 'basename.type')
result-path: list of solver-node, it's result of (send solver :solve graph)
title: title of graph
type: type of output

:write-to-dot *fname &optional result-path (title output)* [method]

write graph structure to dot(graphviz) file
Args:
fname: filename for output
result-path: list of solver-node, it's result of (send solver :solve graph)
title: title of graph

:write-to-dot-stream *&optional (strm *standard-output*) result-path (title output)* [method]

write graph structure to stream as dot(graphviz) style
Args:
strm: stream class for output
result-path: list of solver-node, it's result of (send solver :solve graph)
title: title of graph

:neighbors *&optional args* [method]

:add-neighbor *n &optional a* [method]

arced-node [Class]

:super **node**
:slots nil

:init *&key name* [method]

:find-action *n* [method]

:neighbor-action-alist [method]
:neighbor-action-alist [method]

:find-action *n* [method]

:init *&key name* [method]

solver-node [Class]

:super **propertied-object**
:slots state cost parent action memorized-path

:init *st &key ((:cost c) 0) ((:parent p) nil) ((:action a) nil)* [method]

:path *&optional (prev nil)* [method]

:expand *prblm &rest args* [method]

:state *&optional arg* [method]

:cost *&optional arg* [method]

:parent *&optional arg* [method]

:action *&optional arg* [method]
:action *&optional arg* [method]

:parent *&optional arg* [method]

:cost *&optional arg* [method]

:state *&optional arg* [method]

:expand *prblm &rest args* [method]

:path *&optional (prev nil)* [method]

:init *st &key ((:cost c) 0) ((:parent p) nil) ((:action a) nil)* [method]

solver [Class]

:super **propertied-object**
:slots nil

:init [method]

:solve *prblm* [method]

:solve-by-name *prblm s g &key (verbose nil)* [method]
:solve-by-name *prblm s g &key (verbose nil)* [method]

:solve *prblm* [method]

:init [method]

graph-search-solver [Class]

:super **solver**
:slots open-list close-list

:solve-init *prblm* [method]

:find-node-in-close-list *n* [method]

:solve *prblm &key (verbose nil)* [method]

:add-to-open-list *obj/list* [method]

:null-open-list? [method]

:clear-open-list [method]

:add-list-to-open-list *lst* [method]

:add-object-to-open-list *lst* [method]

:pop-from-open-list *&key (debug)* [method]

:open-list *&optional arg* [method]

:close-list *&optional arg* [method]
:close-list *&optional arg* [method]

:open-list *&optional arg* [method]

:pop-from-open-list *&key (debug)* [method]

:add-object-to-open-list *lst* [method]

:add-list-to-open-list *lst* [method]

:clear-open-list [method]

:null-open-list? [method]

:add-to-open-list *obj/list* [method]

:solve *prblm &key (verbose nil)* [method]

:find-node-in-close-list *n* [method]

:solve-init *prblm* [method]

breadth-first-graph-search-solver [Class]

:super	**graph-search-solver**
:slots	nil

:init [method]

:clear-open-list [method]

:add-list-to-open-list *lst* [method]

:add-object-to-open-list *obj* [method]

:pop-from-open-list *&key (debug)* [method]
:pop-from-open-list *&key (debug)* [method]

:add-object-to-open-list *obj* [method]

:add-list-to-open-list *lst* [method]

:clear-open-list [method]

:init [method]

depth-first-graph-search-solver [Class]

:super	**graph-search-solver**
:slots	nil

:init [method]

:clear-open-list [method]

:add-list-to-open-list *lst* [method]

:add-object-to-open-list *obj* [method]

:pop-from-open-list *&key (debug)* [method]
:pop-from-open-list *&key (debug)* [method]

:add-object-to-open-list *obj* [method]

:add-list-to-open-list *lst* [method]

:clear-open-list [method]

:init [method]

best-first-graph-search-solver [Class]

:super	**graph-search-solver**
:slots	aproblem

:init *p* [method]

:clear-open-list [method]

:add-list-to-open-list *lst* [method]

:add-object-to-open-list *obj* [method]

:pop-from-open-list *&key (debug nil)* [method]

:fn *n p* [method]
:fn *n p* [method]

:pop-from-open-list *&key (debug nil)* [method]

:add-object-to-open-list *obj* [method]

:add-list-to-open-list *lst* [method]

:clear-open-list [method]

:init *p* [method]

a*-graph-search-solver [Class]

:super	**best-first-graph-search-solver**
:slots	nil

:init *p* [method]

:fn *n p &key (debug nil)* [method]

:gn *n p* [method]

:hn *n p* [method]

:hn *n p* [method]

:gn *n p* [method]

:fn *n p &key (debug nil)* [method]

:init *p* [method]

25 irteus Extension

25.1 GL/X Display

polygon [Class]

:super	**plane**
:slots	normal distance convexp edges vertices model-normal model-distance

:draw-on *&key ((:viewer gl::vwer) *viewer*) gl::flush (gl::width 1) (gl::color #f(1.0 1.0 1.0))* [method]

line [Class]

:super	**propertied-object**
:slots	pvert nvert

:draw-on *&key ((:viewer gl::vwer) *viewer*) gl::flush (gl::width 1) (gl::color #f(1.0 1.0 1.0))* [method]

faceset [Class]

:super	**cascaded-coords**
:slots	rot pos parent descendants worldcoords manager changed box faces edges vertices mo

:set-color *gl::color &optional (gl::transparent)* [method]

Set color of given color name, color sample and color name are referenced from http://en.wikipedia.org/wiki/X11_

:paste-texture-to-face *gl::aface &key gl::file gl::image (gl::tex-coords (list (float-vector 0 0) (float-vector 0 1) (float-vector 1 1) (float-vector 1 0)))* [method]

:draw-on *&key ((:viewer gl::vwer) *viewer*) gl::flush (gl::width 1) (gl::color #f(1.0 1.0 1.0))* [method]

coordinates [Class]

:super	**propertied-object**
:slots	rot pos

:draw-on *&key ((:viewer gl::vwer) *viewer*) gl::flush (gl::width (get self :width)) (gl::color (get self :color)) (size (get self :size))* [method]

:vertices [method]

float-vector [Class]

:super	**vector**
:slots	nil

:draw-on *&key ((:viewer gl::vwer) *viewer*) gl::flush (gl::width 1) (gl::color #f(1.0 1.0 1.0)) (size 50)* [method]

:vertices [method]

gl::glvertices [Class]

:super	**cascaded-coords**
:slots	gl::mesh-list gl::filename gl::bbox

:set-color *gl::color &optional (gl::transparent)* [method]

set color as float vector of 3 elements, and transparent as float, all values are betwenn 0 to 1

:actual-vertices [method]

return list of vertices(float-vector), it returns all vertices of this object

:calc-bounding-box [method]

calculate and set bounding box of this object

:vertices [method]

return list of vertices(float-vector), it returns vertices of bounding box of this object

:reset-offset-from-parent [method]

move vertices in this object using self transformation, this method change values of vertices. coordinates's method such as :transform just change view of this object

:expand-vertices [method]

expand vertices number as same number of indices, it enable to set individual normal to every vertices

:use-flat-shader [method]

use flat shader mode, use opengl function of glShadeModel(GL_FLAT)

:use-smooth-shader [method]

use smooth shader mode, use opengl function of glShadeModel(GL_SMOOTH) default

:calc-normals *&optional (gl::force nil) (gl::expand t) (gl::flat t)* [method]

normal calculation
if force option is true, clear current normalset.
if exapnd option is ture, do :expand-vertices.
if flat option is true, use-flat-shader

:mirror-axis [method]

&key (gl::create t)
(gl::invert-faces t)
(gl::axis :y)

creating mirror vertices respect to :axis

:convert-to-faces [method]
&rest args &key (gl::wrt :local)
&allow-other-keys

create list of faces using vertices of this object

:convert-to-faceset *&rest args* [method]
create faceset using vertices of this object

:set-offset *gl::cds &key (gl::create)* [method]
move vertices in this object using given coordinates, this method change values of vertices. coordinates's method such as :transform just change view of this object

:convert-to-world *&key (gl::create)* [method]
move vertices in this object using self's coordinates. vertices data should be moved as the same as displayed

:glvertices *&optional (name) (gl::test #'string=)* [method]
create individual glvertices objects from mesh-list. if name is given, search mesh has the same name

:append-glvertices *gl::glv-lst* [method]
append list of glvertices to this object

:init *gl::mlst &rest args &key ((:filename gl::fn)) &allow-other-keys* [method]

:filename *&optional gl::nm* [method]

:get-meshinfo *gl::key &optional (pos -1)* [method]

:set-meshinfo *gl::key gl::info &optional (pos -1)* [method]

:get-material *&optional (pos -1)* [method]

:set-material *gl::mat &optional (pos -1)* [method]

:expand-vertices-info *gl::minfo* [method]

:faces [method]

:draw-on *&key ((:viewer gl::vwer) *viewer*)* [method]

:draw *gl::vwr &rest args* [method]

:collision-check-objects *&rest args* [method]

:box [method]

:make-pqpmodel *&key (gl::fat 0)* [method]

:append-glvertices *gl::glv-lst* [method]
append list of glvertices to this object

:glvertices *&optional (name) (gl::test #'string=)* [method]

create individual glvertices objects from mesh-list. if name is given, search mesh has the same name

:convert-to-world *&key (gl::create)* [method]

move vertices in this object using self's coordinates. vertices data should be moved as the same as displayed

:set-offset *gl::cds &key (gl::create)* [method]

move vertices in this object using given coordinates, this method change values of vertices. coordinates's method such as :transform just change view of this object

:convert-to-faceset *&rest args* [method]

create faceset using vertices of this object

:convert-to-faces [method]

&rest args &key (gl::wrt :local)
&allow-other-keys

create list of faces using vertices of this object

:mirror-axis [method]

&key (gl::create t)
(gl::invert-faces t)
(gl::axis :y)

creating mirror vertices respect to :axis

:calc-normals *&optional (gl::force nil) (gl::expand t) (gl::flat t)* [method]

normal calculation
if force option is true, clear current normalset.
if exapnd option is ture, do :expand-vertices.
if flat option is true, use-flat-shader

:use-smooth-shader [method]

use smooth shader mode, use opengl function of glShadeModel(GL_SMOOTH) default

:use-flat-shader [method]

use flat shader mode, use opengl function of glShadeModel(GL_FLAT)

:expand-vertices [method]

expand vertices number as same number of indices, it enable to set individual normal to every vertices

:reset-offset-from-parent [method]

move vertices in this object using self transformation, this method change values of vertices. coordinates's method such as :transform just change view of this object

:vertices [method]

return list of vertices(float-vector), it returns vertices of bounding box of this object

:calc-bounding-box [method]

calculate and set bounding box of this object

:actual-vertices [method]

return list of vertices(float-vector), it returns all vertices of this object

:set-color *gl::color &optional (gl::transparent)* [method]

set color as float vector of 3 elements, and transparent as float, all values are betwenn 0 to 1

:make-pqpmodel *&key (gl::fat 0)* [method]

:box [method]

:collision-check-objects *&rest args* [method]

:draw *gl::vwr &rest args* [method]

:draw-on *&key ((:viewer gl::vwer) *viewer*)* [method]

:faces [method]

:expand-vertices-info *gl::minfo* [method]

:set-material *gl::mat &optional (pos -1)* [method]

:get-material *&optional (pos -1)* [method]

:set-meshinfo *gl::key gl::info &optional (pos -1)* [method]

:get-meshinfo *gl::key &optional (pos -1)* [method]

:filename *&optional gl::nm* [method]

:init *gl::mlst &rest args &key ((:filename gl::fn)) &allow-other-keys* [method]

gl::glbody [Class]

:super	**body**
:slots	gl::aglvertices

:glvertices *&rest args* [method]

:draw *gl::vwr* [method]

:set-color *&rest args* [method]
:set-color *&rest args* [method]

:draw *gl::vwr* [method]

:glvertices *&rest args* [method]

gl::find-color *gl::color* [function]

returns color vector of given color name, the name is defined in https://github.com/euslisp/jskeus/blob/master/ir

gl::transparent *gl::abody gl::param* [function]

Set abody to transparent with param

gl::make-glvertices-from-faceset *gl::fs &key (gl::material)* [function] returns glvertices instance

fs is geomatry::faceset

gl::make-glvertices-from-faces *gl::flst &key (gl::material)* [function]

returns glvertices instance

flst is list of geomatry::face

gl::write-wrl-from-glvertices *fname gl::glv &rest args* [function]

write .wrl file from instance of glvertices

gl::set-stereo-gl-attribute [function]

gl::reset-gl-attribute [function]

gl::delete-displaylist-id *gl::dllst* [function]

gl::transpose-image-rows *gl::img &optional gl::ret* [function]

gl::draw-globjects *gl::vwr gl::draw-things &key (gl::clear t) (gl::flush t) (gl::draw-origin 150) (gl::draw-floor nil) (gl::floor-color #f(1.0 1.0 1.0))* [function]

gl::draw-glbody *gl::vwr gl::abody* [function]

gl::_dump-wrl-shape *gl::strm gl::mesh &key ((:scale gl::scl) 1.0) (gl::use_ambient nil) (gl::use_normal nil) (gl::use_texture nil) &allow-other-keys* [function]

x:xwindow [Class]

:super **x:xdrawable**

:slots x::display x::drawable x::gcon x::bg-color x::width x::height x::parent x::subwindows x

:buttonpress *x:event* [method]

:motionnotify *x:event* [method]

:buttonrelease *x:event* [method]

:set-event-proc *type x::method x::receiver* [method]

:clientmessage *x:event* [method]

:quit *&rest x::a* [method]

:event-notify *type x:event* [method]

:create *&rest args* [method]

x:panel [Class]

:super **x:xwindow**

:slots x::display x::drawable x::gcon x::bg-color x::width x::height x::parent x::subwindows x

:quit *&rest args* [method]

x:xscroll-bar [Class]

:super **x:xwindow**

:slots x::display x::drawable x::gcon x::bg-color x::width x::height x::parent x::subwindows x

:redraw [method]

x::tabbed-panel [Class]

:super **x:panel**
:slots x::tabbed-buttons x::tabbed-panels x::selected-tabbed-panel

:create *&rest args* [method]

:add-tabbed-panel *name* [method]

:change-tabbed-panel *x::obj* [method]

:tabbed-button *name &rest args* [method]

:tabbed-panel *name &rest args* [method]

:resize *x::w h* [method]
:resize *x::w h* [method]

:tabbed-panel *name &rest args* [method]

:tabbed-button *name &rest args* [method]

:change-tabbed-panel *x::obj* [method]

:add-tabbed-panel *name* [method]

:create *&rest args* [method]

x::panel-tab-button-item [Class]

:super **x:button-item**
:slots nil

:draw-label *&optional (x::state :up) (x::offset 0)* [method]
:draw-label *&optional (x::state :up) (x::offset 0)* [method]
x::window-main-one *&optional fd* [function]
x::event-far *x::e* [function]
x::event-near *x::e* [function]

25.2 Utility Function

mtimer [Class]

:super **object**
:slots buf

:init [method]

Initialize timer object.

:start [method]

Start timer.

:stop [method]

Stop timer and returns elapsed time in seconds.

:stop [method]

Stop timer and returns elapsed time in seconds.

:start [method]

Start timer.

:init [method]

Initialize timer object.

interpolator [Class]

:super	**propertied-object**
:slots	(position-list :type cons) (time-list :type cons) (position :type float-vector) (time :type

:init [method]

Abstract class of interpolator

:reset [method]

&rest args &key ((:position-list pl) (send self :position-list))
((:time-list tl) (send self :time-list))
&allow-other-keys

Initialize interpolator
position-list: list of control point
time-list: list of time from start for each control point, time in first contrall point is zero, so length of this list is length of control point minus 1

:position-list [method]

returns position list

:position [method]

returns current position

:time-list [method]

returns time list

:time [method]

returns current time

:segment-time [method]

returns time[sec] with in each segment

:segment [method]

returns index of segment which is currently processing

:segment-num [method]

returns number of total segment

:interpolatingp [method]

returns if it is currently processing

:start-interpolation [method]

start interpolation

:stop-interpolation [method]

stop interpolation

:pass-time *dt* [method]

process interpolation for dt[sec]

:pass-time *dt* [method]

process interpolation for dt[sec]

:stop-interpolation [method]

stop interpolation

:start-interpolation [method]

start interpolation

:interpolatingp [method]

returns if it is currently processing

:segment-num [method]

returns number of total segment

:segment [method]

returns index of segment which is currently processing

:segment-time [method]

returns time[sec] with in each segment

:time [method]

returns current time

:time-list [method]

returns time list

:position [method]

returns current position

:position-list [method]

returns position list

:reset [method]

&rest args &key ((:position-list pl) (send self :position-list))
((:time-list tl) (send self :time-list))
&allow-other-keys

Initialize interpolator
position-list: list of control point

time-list: list of time from start for each control point, time in first contrall point is zero, so length of this list is length of control point minus 1

:init [method]

Abstract class of interpolator

linear-interpolator [Class]

:super **interpolator**
:slots nil

:interpolation [method]

Linear interpolator

:interpolation [method]

Linear interpolator

minjerk-interpolator [Class]

:super **interpolator**
:slots velocity acceleration velocity-list acceleration-list

:velocity [method]

returns current velocity

:velocity-list [method]

returns velocity list

:acceleration [method]

returns current acceleration

:acceleration-list [method]

returns acceleration list

:reset [method]

&rest args &key ((:velocity-list vl) (send self :velocity-list))
((:acceleration-list al) (send self :acceleration-list))
&allow-other-keys

minjerk interopolator
position-list : list of control point
velocity-list : list of velocity in each control point
acceleration-list : list of acceleration in each control point

:interpolation [method]

Minjerk interpolator, a.k.a Hoff & Arbib

Example code is:
(setq l (instance minjerk-interpolator :init))
(send l :reset :position-list (list #f(1 2 3) #f(3 4 5) #f(1 2 3)) :time-list (list 0.1 0.18))
(send l :start-interpolation)
(while (send l :interpolatingp) (send l :pass-time 0.02) (print (send l :position)))

:interpolation [method]

Minjerk interpolator, a.k.a Hoff & Arbib

Example code is:
(setq l (instance minjerk-interpolator :init))
(send l :reset :position-list (list #f(1 2 3) #f(3 4 5) #f(1 2 3)) :time-list (list 0.1 0.18))
(send l :start-interpolation)
(while (send l :interpolatingp) (send l :pass-time 0.02) (print (send l :position)))

:reset [method]

&rest args &key ((:velocity-list vl) (send self :velocity-list))
((:acceleration-list al) (send self :acceleration-list))
&allow-other-keys

minjerk interopolator
position-list : list of control point
velocity-list : list of velocity in each control point
acceleration-list : list of acceleration in each control point

:acceleration-list [method]

returns acceleration list

:acceleration [method]

returns current acceleration

:velocity-list [method]

returns velocity list

:velocity [method]

returns current velocity

forward-message-to *to args* [function]

forward _args_ message to _to_ object

forward-message-to-all *to args* [function]

forward _args_ message to all _to_ object

permutation *lst n* [function]

Returns permutation of given list

combination *lst n* [function]

Returns combination of given list

find-extreams [function]

datum &key (key #'identity)
(identity #'=)
(bigger #'>)

Returns the elements of datum which maximizes key function

eus-server *&optional (port 6666) &key (host (unix:gethostname))* [function]

Create euslisp interpreter server, data sent to socket is evaluated as lisp expression

connect-server-until-success [function]
host port &key (max-port (+ port 20)) (return-with-port nil)

Connect euslisp interpreter server until success

format-array *arr &optional (header) (in 7) (fl 3) (strm *error-output*) (use-line-break t)* [function]
print formatted array

his2rgb *h &optional (i 1.0) (s 1.0) ret* [function]
convert his to rgb (0 <= h <= 360, 0.0 <= i <= 1.0, 0.0 <= s <= 1.0)

hvs2rgb *h &optional (i 1.0) (s 1.0) ret* [function]
convert hvs to rgb (0 <= h <= 360, 0.0 <= i <= 1.0, 0.0 <= s <= 1.0)

rgb2his *r &optional g b ret* [function]
convert rgb to his (0 <= r,g,b <= 255)

rgb2hvs *r &optional g b ret* [function]
convert rt to hvs (0 <= r,g,b <= 255)

color-category10 *i* [function]
Choose good color from 10 colors

color-category20 *i* [function]
Choose good color from 20 colors

kbhit [function]
Checks the console for a keystroke. returns keycode value if a key has been pressed, otherwise it returns nil. Note that this does not work well on Emacs Shell mode, run EusLisp program from terminal shell.

piped-fork-returns-list *cmd &optional args* [function]
piped fork returning result as list

make-robot-model-from-name *name &rest args* [function]
make a robot model from string: (make-robot-model "pr2")

mapjoin *expr seq1 seq2* [function]
need-thread *n &optional (lsize (*512 1024)) (csize lsize)* [function]
termios-c_iflag *lisp::s* [function]
set-termios-c_iflag *lisp::s lisp::val* [function]
termios-c_oflag *lisp::s* [function]
set-termios-c_oflag *lisp::s lisp::val* [function]
termios-c_cflag *lisp::s* [function]
set-termios-c_cflag *lisp::s lisp::val* [function]
termios-c_lflag *lisp::s* [function]
set-termios-c_lflag *lisp::s lisp::val* [function]
termios-c_line *lisp::s* [function]
set-termios-c_line *lisp::s lisp::val* [function]
termios-c_cc *lisp::s &optional lisp::i* [function]
set-termios-c_cc *lisp::s lisp::i &rest lisp::val* [function]
termios-c_ispeed *lisp::s* [function]
set-termios-c_ispeed *lisp::s lisp::val* [function]
termios-c_ospeed *lisp::s* [function]

set-termios-c_ospeed *lisp::s lisp::val* [function]

gnuplot [Class]

:super	**propertied-object**
:slots	strm data data-length last-command debug

:init [method]

host &key (clear t)
((:debug _debug))

Initialize gnuplot interface object with given host name

:clear [method]

Clear graph

:draw *&rest vs* [method]

Draw graph with given float vectors,
:range, :xrange, :yrange, : range of each axis
:title : title of graph
:line-width : line width
:direction : direction of the graph (:right, :left)
:xscale, :xoffset : scale and offset for data
:y2tics : list variable to specify when y2 range is used
:y2range : set y2 tics and specify range
:type : specify type of the graph (:lines, :2dmap)

:save *f &key (type postscript eps color "Times-Roman" 24)* [method]

save graph as eps file

:replot [method]

:reset [method]

:command *msg* [method]

:quit [method]

:proc-length *&optional n* [method]

:proc-clear [method]

:proc-one *vs &rest args* [method]

:save *f &key (type postscript eps color "Times-Roman" 24)* [method]

save graph as eps file

:draw *&rest vs* [method]

Draw graph with given float vectors,
:range, :xrange, :yrange, : range of each axis

:title : title of graph
:line-width : line width
:direction : direction of the graph (:right, :left)
:xscale, :xoffset : scale and offset for data
:y2tics : list variable to specify when y2 range is used
:y2range : set y2 tics and specify range
:type : specify type of the graph (:lines, :2dmap)

:clear [method]

Clear graph

:init [method]

host &key (clear t)
((:debug _debug))

Initialize gnuplot interface object with given host name

:proc-one *vs &rest args* [method]

:proc-clear [method]

:proc-length *&optional n* [method]

:quit [method]

:command *msg* [method]

:reset [method]

:replot [method]

gnuplot *&key (host (unix:gethostname))* [function]

Returns gnuplot interface instance
ex)
(setq *g*(gnuplot))
(send *g*:draw #f(0 1 2 3 4 5) #f(5 4 3 2 1 0) :xrange '(0 10) :yrange '(0 10) :title '("data1" "data2"))

see irteus/gnuplotlib.l for more info

graph-view [function]

ordinate-list &optional (abscissa-list (let ((idx -1)) (mapcar #'(lambda (x) (incf idx)) (make-list (len
(xlabel X)
(ylabel Y)
(zlabel Z)
(dump-graph nil)
(graph-fname (format nil A.eps (substitute 95 (elt 0) title)))
(mode lines)
keylist
xrange

yrange
zrange
x11
additional-func
no-dump
((:graph-instance gp) (if (boundp '*gp*) *gp*(setq *gp*(gnuplot))))
(fname (format nil data A (system:address gp)))

plot function for 2d or 3d plot
ordinate-list : list of data for ordinate axis
2D = (list (list y00 y01 ... y0n), ... (list ym0 ym1 ... ymn))
3D = (list (list z00 z01 ... z0n), ... (list zm0 zm1 ... zmn))
abscissa-list : list of data for abscissa axes
2D = (list x0 x1 ... xn)
3D = (list xylist0 ... xylistn) ;; xylist = (list x y)
:title : title of graph
:xlabel, :ylabel, zlabel : label for each axis
:keylist : legend of each data
:xrange, :yrange, :zrange : range of each axis
:mode : "lines" or "points"

25.3 Math Function

inverse-matrix *mat* [function]

returns inverse matrix of mat

inverse-matrix-complex *cmat* [function]

returns inverse matrix of complex square matrix

m*-complex *cmat1 cmat2* [function]

returns complex matrix 1 *complex matrix 2

diagonal *v* [function]

make diagonal matrix from given vecgtor, diagonal #f(1 2) ->#2f((1 0)(0 2))

minor-matrix *m ic jc* [function]

return a matrix removing ic row and jc col elements from m

atan2 *y x* [function]

returns atan2 of y and x (atan (/ y x))

outer-product-matrix *v &optional (ret (unit-matrix 3))* [function]

returns outer product matrix of given v

```
matrix(a) v = a *v
```

```
0 -w2 w1

w2 0 -w0

-w1 w0 0
```

matrix2quaternion *m* [function]
returns quaternion (w x y z) of given matrix

quaternion2matrix *q* [function]
returns matrix of given quaternion (w x y z)

matrix-log *m* [function]
returns matrix log of given m, it returns [-pi, pi]

matrix-exponent *omega &optional (p 1.0)* [function]
returns exponent of given omega

midrot *p r1 r2* [function]
returns mid (or p) rotation matrix of given two matrix r1 and r2

pseudo-inverse *mat &optional weight-vector ret wmat mat-tmp* [function]
returns pseudo inverse of given mat

sr-inverse *mat &optional (k 1.0) weight-vector ret wmat tmat umat umat2 mat-tmp mat-tmp-rc mat-tmp-rr mat-tmp-rr2* [function]
returns sr-inverse of given mat

manipulability *jacobi &optional tmp-mrr tmp-mcr* [function]
return manipulability of given matrix

random-gauss *&optional (m 0) (s 1)* [function]
make random gauss, m:mean s:standard-deviation

gaussian-random *dim &optional (m 0) (s 1)* [function]
make random gauss vector, replacement for quasi-random defined in matlib.c

eigen-decompose-complex *m* [function]
returns eigen decomposition from real square matrix

solve-non-zero-vector-from-det0-matrix *m* [function]
solves non-zero-vector v from real square determinant-zero-matrix mat, when mat∗v=O and det(mat)=0

concatenate-matrix-column *&rest args* [function]
Concatenate matrix in column direction.

concatenate-matrix-row *&rest args* [function]
Concatenate matrix in row direction.

concatenate-matrix-diagonal *&rest args* [function]
Concatenate matrix in diagonal.

vector-variance *vector-list* [function]

returns vector, each element represents variance of elements in the same index of vector within vector-list

covariance-matrix *vector-list* [function]

make covariance matrix of given input vector-list

normalize-vector *v &optional r (eps 1.000000e-20)* [function]

calculate normalize-vector #f(0 0 0)->#f(0 0 0).

pseudo-inverse-org *m &optional ret winv mat-tmp-cr* [function]

sr-inverse-org *mat &optional (k 1) me mat-tmp-cr mat-tmp-rr* [function]

eigen-decompose *m* [function]

lms *point-list* [function]

lms-estimate *res point-* [function]

lms-error *result point-list* [function]

lmeds *point-list &key (num 5) (err-rate 0.3) (iteration) (ransac-threshold) (lms-func #'lms) (lmeds-error-func #'lmeds-error) (lms-estimate-func #'lms-estimate)* [function]

lmeds-error *result point-list &key (lms-estimate-func #'lms-estimate)* [function]

lmeds-error-mat *result mat &key (lms-estimate-func #'lms-estimate)* [function]

25.4 Image Function

read-image-file *fname* [function]

read image of given fname. It returns instance of **grayscale-image** or **color-image24**.

write-image-file *fname image::img* [function]

write img to given fname

read-png-file *fname* [function]

write-png-file *fname image::img* [function]