nil

# Antik

# Table of Contents

# 1 Introduction

Antik provides a foundation for scientific and engineering computation in Common Lisp. It is designed not only to facilitate numerical computations, but to permit the use of numerical computation libraries and the interchange of data and procedures, whether foreign (non-lisp) or Lisp libraries. Notably, GSLL provides an interface to the GNU Scientific Library and is based on Antik.

There are two Common Lisp packages defined by Antik, `antik` and `grid`. Names within these packages have been chosen to match Common Lisp names where the corresponding function could be considered a generalization of the CL function. Therefore, shadowing is used to make sure that within the `antik` package and packages that use it, a reference to the symbol will use the Antik definition and not the CL definition.

Antik is designed and developed to provide a common foundation and interoperability between scientific, engineering and mathematical libraries, whether in lisp or not. Interoperability means that objects created can easily be passed to one or more libraries, and libraries can be mixed and combined to solve a problem. It also means that names of like functions will be the same, differening only in the package. For example, if systems `foo` and `bar` both provide an LU decomposition, that function will be `foo:lu-decomposition` in one and `bar:lu-decomposition` in the other. Should a user wish to compare or switch libraries in such a function call, it is a simple matter of changing the package. If a whole library should be switched, the names can be used without a package prefix and the use-package form changed. This makes it easy to compare results, and select and mix the best libraries for a calculation.

There is a package defined for users to use Antik, `antik-user`. All the appropriate symbols are shadowed. It usees the packages `common-lisp`, `grid`, `antik`, and `iterate`. To make another package with the same use and shadowing properties, call the function `antik:make-user-package`.

`antik:make-user-package` *name* [Function]

Make a package in which it is convenient to use Antik and related systems. If the package already exists, the use-packages and shadowing symbols are updated.

# 2 Mathematics

## 2.1 Generic Functions

Generic mathematical functions are provided so that they may be called on something other than numbers, such as grids. When called with CL numbers, they perform they call the equivalent CL function.

`antik:abs` *num* [Generic Function]

The absolute value.

`antik:acos` *arg* [Generic Function]

The arccosine of the generalized or regular number.

`antik:asin` *arg* [Generic Function]

The arcsine of the generalized or regular number.

`antik:atan` *num &optional den default zero-enough* [Generic Function]

The arctangent of the generalized or regular number. If absolute value of the arguments are below zero-enough, default is returned.

`antik:cosh` *num* [Generic Function]

The hyperbolic cosine of the generalized or regular number.

`antik:cos` *num* [Generic Function]

The cosine of the generalized or regular number.

`antik:=` *a b* [Generic Function]

Numeric equal

`antik:exp` *num* [Generic Function]

The natural exponent e^num of the generalized or regular number.

`antik:expt` *num exponent* [Generic Function]

Raise the number to the exponent.

`antik:floor` *number &optional divisor* [Generic Function]

Greatest multiple of divisor less than number.

`antik:>=` *a b* [Generic Function]

Greater than or equal.

`antik:>` *a b* [Generic Function]

Greater than

`antik:log` *num &optional base* [Generic Function]

The natural logarithm of the generalized or regular number.

`antik:<=` *a b* [Generic Function]

Less than or equal.

`antik:<` *a b* [Generic Function]

Less than.

`antik:minusp` *a* [Generic Function]

Negative

`antik:plusp` *a* [Generic Function]

Positive.

`antik:+` *&rest args* [Function]

Addition of generalized or regular numbers with an arbitrary number of arguments.

`antik:round` *number &optional divisor* [Generic Function]

Nearest multiple of divisor to number.

`antik:signum` *a* [Generic Function]

Signum(a).

`antik:sinh` *num* [Generic Function]

The hyperbolic sine of the generalized or regular number.

`antik:sin` *num* [Generic Function]

The sine of the generalized or regular number.

`antik:/` *&rest args* [Function]

Division of generalized or regular numbers for an arbitrary number of arguments.

`antik:sqrt` *num* [Function]

The square root of the generalized or regular number.

`antik:*` *&rest args* [Function]

Multiplication of generalized or regular numbers with an arbitrary number of arguments.

`antik:tanh` *num* [Generic Function]

The hyperbolic tangent of the generalized or regular number.

`antik:tan` *num* [Generic Function]

The tangent of the generalized or regular number.

`antik:-` *&rest args* [Function]

Subtraction of generalized or regular numbers.

`antik:zerop` *a* [Generic Function]

Zero.

## 2.2 Conditions

Conditions and handlers are defined for common mathematical computations.

The condition `making-complex-number` is signalled when a calculation would make a complex number. If this outcome is desirable, and handler such as defined by the macro `handling-complex-number` is used.

`antik:making-complex-number` [Condition]

Class precedence list: `making-complex-number`, `arithmetic-error`, `error`, `serious-condition`, `condition`, `t`

Signalled when a mathematical calculation would result in a complex number.

`antik:arithmetic-errors-return-nan` *&body body* [Macro]

Return a NaN for a variety of arithmetic errors.

`antik:handling-complex-number` *restart &body body* [Macro]

A handler to make the complex number.

# 3 Organization of Computation

## 3.1 Introduction

Typically scientific and engineering computation starts with input values and computes values or properties of an object, uses those as inputs to compute more values and properties, and eventually produces an output. Sometimes it is necessary to examine the intermediate values, as for education or debugging; sometimes, especially when doing production runs, the variable input and output are the only things of interest, the values that remain unchanged with each calculation are recorded but are not the focus of the computation, and the intermediate values are only need temporarily to get to the next stage of the computation.

This section will describe the definitions for organizing computations. At present very little is written; only *parameters* definitions exist, and these are likely to change.

## 3.2 Parameters

Parameters are named values that are defined and used in computations or in presentation; the concept is distinct from Common Lisp's defparameter macro. They are organized into named categories; Antik itself defines one category, `NF`, which is used for numerical formatting. Each parameter is defined with a default value and description. The value can be changed with `(setf parameter-value)` and used with `parameter-value`, or changed locally (analogous to `let` for CL variables) with `with-parameters`.

```
;;; First create the category
(define-parameter-category kepler)

;;; Then define some parameters
(define-parameter kepler foo 122 fixnum "A fixnum parameter of kepler.")
(define-parameter kepler bar "hi" string "A string parameter of kepler.")

;;; Get their values
(parameter-value kepler bar)
"hi"
(parameter-value kepler foo)
122

;;; Dynamic binding
(defun show-foo-bar ()
  (format t "~&foo: ~a, bar: ~s"
  (parameter-value kepler foo)
  (parameter-value kepler bar)))

(show-foo-bar)
foo: 122, bar: "hi"
NIL

;;; Locally change values
```

```
(with-parameters (kepler foo 143 bar "bye")
  (show-foo-bar))
foo: 143, bar: "bye"
NIL

(show-foo-bar)
foo: 122, bar: "hi"
NIL

;;; Make a mistake
(with-parameters (kepler foo 143 bar -44)
  (show-foo-bar))
Error: Value -44 is of type FIXNUM, not of the required type STRING.

;;; Globally change values
(set-parameter-value kepler bar "a new value")
foo: 122, bar: "a new value"

;;; Set multple values
(set-parameters kepler bar "xyz" foo 1)

;;; Get information about the categories and parameters
(parameter-help)
Parameter categories: KEPLER.

(parameter-help :kepler)
Parameters in KEPLER: BAR and FOO.

(parameter-help :kepler :bar)
BAR: A string parameter of kepler.
Type is: STRING, default value is "hi".

(parameter-value* kepler (mkstr "FO" "O"))
122
```

`antik:define-parameter-category` *name* [Macro]

Define a new category for parameters.

`antik:define-parameter` *category name default type description* [Macro]

Define the parameter name in category.

`antik:parameter-help` *&optional category name stream* [Function]

Print all information known about the parameter. If category is nil (default), names of all categories are printed. If name is nil, all defined parameters in that category are printed.

`antik:parameters-set` *category* [Macro]

A list of parameters that are set.

`antik:parameter-value` *category name &optional default* [Macro]

Get or set the parameter value.

`antik:parameter-value*` *category name &optional default* [Macro]

Get the parameter value dynamically.

`antik:reset-parameters` *category &rest names* [Macro]

Reset the parameter(s) to the default value; if none are specified, all in the category are reset.

`antik:set-parameter-value` *category name value* [Macro]

Set the parameter value, checking that the name is legitimate and the value is of the correct type.

`antik:with-parameters` (*category* **&rest** *name-values*) *&body body* [Macro]

Provide local scope for parameter values, and possibly bind new values.

# 4 Grids

## 4.1 Introduction to grids

### 4.1.1 The grid concept

In scientific and engineering computing, data is frequently organized into a regular pattern indexed by a finite sequence of integers, and thought of having a Cartesian arrangement. We call these *grids*. A standard form is the *array*, but we do not wish to confine ourselves to representation in terms of a Common Lisp array. Other examples might be a list of lists, a C array accessible through a foreign function interface, or an SQL table. We assume a rectangular shape; that is, the range of indices permissible is independent of the other values of the index. Each node of the grid is an *element* (or sometimes a *component*).

Since we are primarily focused on the scientific and engineering applications, the elements will generally be numbers of some type. Most of the definitions here however do not force this to be the case.

The *rank* of a grid is the number of Cartesian axes, and the *dimensions* are a sequence of non-negative integers of length equal to the rank that give the number of index values possible along each axis. We will call a grid of rank 1 a *vector* (where there would not be a confusion with the CL vector) and a grid of rank 2 a *matrix*. For example, a matrix representing rotations in three dimensional space would have rank 2 and dimensions (3 3).

### 4.1.2 Types of grids

Antik defines two types of grids, *array*s, i.e. ordinary Lisp arrays, and *foreign array*s, which are defined in foreign memory and therefore are accessible by foreign libraries. The argument `grid-type` to a function or the variable `grid:*default-grid-type*` should be bound to one of `cl:array` or `grid:foreign-array`, to specify an array or foreign-array respectively.

### 4.1.3 Package

The symbols defined here are in the `grid` package.

### 4.1.4 Tests

There are some tests defined for these modules; to run the tests, make sure that `asdf-system-connections` system is loaded before anything else, and `lisp-unit` is loaded, then

```
(in-package :grid)
(lisp-unit:run-tests)
```

## 4.2 Creating a grid

### 4.2.1 Reader macro and function `grid`

The #m reader macro in the default form creates a vector or matrix of element type double-float, which is the most common type needed for mathematical functions, and grid type given by `grid:*default-grid-type*` (see Section 4.1.2 [Types of grids], page 8). It optionally takes a numeric argument prefix to make an array with a different element type;

a guide to the numeric argument is given below. It should be followed by a list, which is not evaluated. If the list contains `^`, the object created will be a matrix and each row is ended with that symbol.

| **Element type** | **#m prefix** |
|---|---|
| double-float | 1 or empty |
| (complex double-float) | 2 |
| single-float | 3 |
| (complex single-float) | 4 |
| (signed-byte 8) | 7 |
| (unsigned-byte 8) | 8 |
| (signed-byte 16) | 15 |
| (unsigned-byte 16) | 16 |
| (signed-byte 32) | 31 |
| (unsigned-byte 32) | 32 |
| (signed-byte 64) | 63 |
| (unsigned-byte 64) | 64 |

The function `grid:grid` creates a grid of the default type and evaluates its arguments. For example,

```
ANTIK-USER> (setf grid:*default-grid-type* 'grid:foreign-array)
FOREIGN-ARRAY
ANTIK-USER> (grid:grid 1.0d0 2.0d0 3.0d0)
#m(1.000000000000000d0 2.000000000000000d0 3.000000000000000d0)
ANTIK-USER> (type-of @)
VECTOR-DOUBLE-FLOAT
```

Of course, an ordinary (Common Lisp) array may be created in the usual fashion, with the `#` reader macro

```
#(1.0d0 2.0d0 3.0d0)
```

or with `make-array`.

### 4.2.2 The `foreign-array` class

The `foreign-array` class uses `CFFI` to allow the use of C arrays as grids.

Classes of vectors and matrices are named by appending the element type as hypenated words to "vector" or "matrix". The following table shows the classes available on a 64-bit platform:

| **Element type** | **Vector class** | **Matrix class** | **#m prefix** |
|---|---|---|---|
| double-float | `vector-double-float` | `matrix-double-float` | 1 or empty |

| | | | |
|---|---|---|---|
| (complex double-float) | `vector-complex-double-float` | `matrix-complex-double-float` | 2 |
| single-float | `vector-single-float` | `matrix-single-float` | 3 |
| (complex single-float) | `vector-complex-single-float` | `matrix-complex-single-float` | 4 |
| (signed-byte 8) | `vector-signed-byte-8` | `matrix-signed-byte-8` | 7 |
| (unsigned-byte 8) | `vector-unsigned-byte-8` | `matrix-unsigned-byte-8` | 8 |
| (signed-byte 16) | `vector-signed-byte-16` | `matrix-signed-byte-16` | 15 |
| (unsigned-byte 16) | `vector-unsigned-byte-16` | `matrix-unsigned-byte-16` | 16 |
| (signed-byte 32) | `vector-signed-byte-32` | `matrix-signed-byte-32` | 31 |
| (unsigned-byte 32) | `vector-unsigned-byte-32` | `matrix-unsigned-byte-32` | 32 |
| (signed-byte 64) | `vector-signed-byte-64` | `matrix-signed-byte-64` | 63 |
| (unsigned-byte 64) | `vector-unsigned-byte-64` | `matrix-unsigned-byte-64` | 64 |

A foreign array is especially useful when sending or receiving data from a foreign library. In that case, the generic function `grid:foreign-pointer` should be called on the foreign array in order to obtain the pointer that the foreign library needs. The identical contents as a CL array is available with the function `cl-array` under two conditions: the static-vectors system is installed and supported for the CL implementation used, and foreign array was *not* made with Section 4.2.4.6 [make-foreign-array-from-pointer], page 12. If it does exist, this array is the same as the foreign array (not a copy), so for example changing one changes the other:

```
ANTIK-USER> (defparameter *my-fa* (make-foreign-array 'double-float :dimensions 3))
ANTIK-USER> *my-fa*
#m(0.000000000000000d0 1.000000000000000d0 2.000000000000000d0)
ANTIK-USER> (cl-array *my-fa*)
```

```
#(0.0 1.0 2.0)
ANTIK-USER> (setf (grid:aref *my-fa* 1) 33.3d0)
33.3
ANTIK-USER>  *my-fa*
#m(0.000000000000000d0 33.300000000000000d0 2.000000000000000d0)
ANTIK-USER> (cl-array *my-fa*)
#(0.0 33.3 2.0)
ANTIK-USER> (setf (aref (cl-array *my-fa*) 1) 22.2d0)
22.2
ANTIK-USER> *my-fa*
#m(0.000000000000000d0 22.200000000000000d0 2.000000000000000d0)
```

### 4.2.3 Simple grids and default values

A simple grid is one which only has one layer, that is, the contents of the objects are the scalar elements (and not other array-like objects). Both array and foreign-array are simple grids. The function `make-simple-grid` will create a simple grid. It takes the optional keyword arguments (default): `grid-type` (`*default-grid-type*`), `dimensions` (`*default-dimensions*`), `element-type` (`*default-element-type*`), `initial-element`, `initial-contents`.

### 4.2.4 Functions for creation of grids

The functions `make-grid` and `make-foreign-array` are used to make grids from specifications or parameters. The functions `map-grid` and `map-n-grids` can be used to create grids from functions of indices.

#### 4.2.4.1 grid

`grid:grid` *&rest args* [Function]

Make the simple grid with default properties and elements as specified in args.

#### 4.2.4.2 make-simple-grid

`grid:make-simple-grid` *&rest args &key grid-type dimensions element-type initial-element initial-contents* [Function]

Make a simple grid by specfying the grid-type (default *default-grid-type*), dimensions (*default-dimensions*), element-type (*default-element-type*), and optionally initial-element or initial-contents.

#### 4.2.4.3 make-grid

`grid:make-grid` *specification &rest keys &key initial-contents initial-element &allow-other-keys* [Function]

Make a grid object with no or literal values specified. The specification is of the form ((component-type dimensions...) ... element-type) If initial-contents are specified, the dimensions may be omitted, ((component-type) ... element-type).

This is used for making any kind of grid with the same value for each element, or with literally specified values. The first argument is a specification, which has the form `((grid-`

type dimensions) element-type). The keyword arguments are :initial-element or :initial-contents. For example,

```
(make-foreign-array 'double-float :dimensions 3 :initial-element 77.0d0)
#m(77.0d0 77.0d0 77.0d0)
(make-grid '((foreign-array 3) double-float) :initial-element 77.0d0)
#m(77.0d0 77.0d0 77.0d0)
```

#### 4.2.4.4 make-foreign-array

grid:make-foreign-array *element-type &rest keys &key dimensions &allow-other-keys* [Function]

Make a gsll array with the given element type, :dimensions, and :initial-contents, :initial-element or :data.

This function can be used instead of make-grid to make a foreign-array; the first argument is the element-type and the :dimensions are supplied in a keyword argument, for example,

```
(grid:make-foreign-array 'double-float :dimensions '(2 2))
#m((0.000000000000000d0 .0000000000000000d0)
   (.0000000000000000d0 .0000000000000000d0))
```

It is meant as an analogue to cl:make-array for the convenience of users who want to use a function with a similar argument list.

#### 4.2.4.5 ensure-foreign-array

grid:ensure-foreign-array *object &optional dimensions element-type initial-element* [Function]

If object is not a foreign array, make and return a foreign array with the dimensions and element-type. If object is a foreign array, it is returned.

If the first argument to this function is a foreign array, it is returned. If it is not, the foreign array with the desired dimensions, element type, and initial-element is made and returned.

```
(grid:ensure-foreign-array nil '(2 2) 'double-float)
#m((.0000000000000000d0 .0000000000000000d0)
   (.0000000000000000d0 .0000000000000000d0))
```

#### 4.2.4.6 make-foreign-array-from-pointer

grid:make-foreign-array-from-pointer *pointer dimensions element-type &optional finalize* [Function]

Given a foreign pointer to an array, make a Lisp object of class 'foreign-array that references that. This will never make a static-vector. If finalize is true, than the array memory will be freed when this object is garbage collected; otherwise it is presumed that the foreign code will handle that job.

If a foreign pointer already exist (for example, the foreign array was allocated by foreign code), this function will make a foreign array from it. Note that the cl-array of this grid will always be , regardless of whether static-vectors are supported for the CL implementation.

#### 4.2.4.7 Summary: ways to make grids

- General
  - #m reader macro
  - grid:grid
  - grid:make-grid
  - grid:make-simple-grid
- Array
  - cl:make-array
  - # and #2A reader macros
- Foreign array
  - grid:make-foreign-array
  - grid:ensure-foreign-array
  - grid:make-foreign-array-from-pointer

## 4.3 Operations

### 4.3.1 Elements

`grid:aref` *grid &rest indices* [Generic Function]

Select the element from the grid.

`grid:aref*` *grid linearized-index* [Generic Function]

Select the element from the grid using a linearized index.

`(setf grid:aref)` [Generic Function]

Set the element from the grid.

`(setf grid:aref*)` [Generic Function]

Set the element from the grid using a linearized index.

Individual elements are obtained using `grid:aref` (analogous to Lisp's `cl:aref`), and are set with `setf grid:aref`. For example,

```
ANTIK-USER> (setf *default-grid-type* 'foreign-array)
ANTIK-USER> (defparameter *array* #m(2.0d0 1.0d0 -1.0d0))
ANTIK-USER> *array*
#m(2.000000000000000d0 1.000000000000000d0 -1.000000000000000d0)
ANTIK-USER> (grid:aref *array* 1)
1.0
ANTIK-USER> (setf (grid:aref *array* 1) 77.0d0)
77.0
ANTIK-USER> (grid:aref *array* 1)
77.0
ANTIK-USER> *array*
#m(2.000000000000000d0 77.000000000000000d0 -1.000000000000000d0)
```

### 4.3.2 Copying

`grid:copy` *object &key specification grid-type dimensions element-type destination* [Generic Function]

Copy contents into existing or new object.

`grid:copy-to` *object &optional type* [Function]

Make a grid of the specified type from the object.

Copying is performed with the function `copy`. This works between `grid:foreign-arrays`, pointers, and CL arrays. There are two functions provided to extract the dimensions of a vector or array: `dim0` and `dim1`; the latter is applicable only for matrices. To copy to a grid with the same dimensions and element type, use `copy-to`.

### 4.3.3 Iterate

Extensions to the `iterate` system are provided. The following `for` clause iterators are defined:

- `matrix-row`
- `matrix-row-index`
- `matrix-column`
- `matrix-column-index`
- `vector-element`
- `vector-element-index`
- `matrix-element`
- `matrix-element-index`

For example,

```
(defparameter m1 #m(1 2 3 ^ 0 6 8))
(iter:iter (iter:for e :matrix-element m1) (princ e) (princ " "))
1.0 2.0 3.0 0.0 0.0 6.0 8.0
```

The definitions are in the `grid-iterate-extension` extension system which will automatically load if `asdf-system-connnections`, `iterate`, and the `Antik` system are loaded.

### 4.3.4 Composition functions

These functions transform or compose grids to make new grids.

#### 4.3.4.1 codimension-one-subspace

`grid:codimension-one-subspace` *grid position index &optional destination* [Function]

Select a subspace with rank one less than the argument grid. The position is a non-negative number indicating which dimension is to be dropped, and index is the fixed value it should have. If destination is specified, the result will be written to that grid.

`(setf grid:codimension-one-subspace)` [Function]

Set a subspace with rank one less than the argument grid. The index is a non-negative number indicating which dimension is to be dropped, and position is the fixed value it should have.

#### 4.3.4.2 column

`grid:column` *grid index &optional destination* [Function]

The subgrid with the second index set to the specified value.

`(setf grid:column)` [Function]

Set the subgrid with the second index set to the specified value.

The functions `column`, `(setf column)` select or set the column of a matrix. For example, the first column of the above array is

```
(column (test-grid-double-float 'array '(3 4)) 0)
#(0.0d0 10.0d0 20.0d0)
```

#### 4.3.4.3 concatenate-grids

`grid:concatenate-grids` *grid0 grid1 &key axis* [Function]

Concatenate the grids along the axis specified. The new grid will have the grid type specification and element type as grid0.

The function `concatenate-grids` is used to join two grids on an axis whose dimensions are the same on the other axes. For example, join two matrices by adjoining their columns, all of the same length:

```
(map-grid :source (offset-ifd 0.5d0) :source-dims '(3 4))
#2A((0.5d0 1.5d0 2.5d0 3.5d0)
    (10.5d0 11.5d0 12.5d0 13.5d0)
    (20.5d0 21.5d0 22.5d0 23.5d0))
(map-grid :source (offset-ifd 0.1d0) :source-dims '(3 2))
#2A((0.1d0 1.1d0) (10.1d0 11.1d0) (20.1d0 21.1d0))
(concatenate-grids ** * :axis 1)
#2A((0.5d0 1.5d0 2.5d0 3.5d0 0.1d0 1.1d0)
    (10.5d0 11.5d0 12.5d0 13.5d0 10.1d0 11.1d0)
    (20.5d0 21.5d0 22.5d0 23.5d0 20.1d0 21.1d0))
```

#### 4.3.4.4 diagonal

`grid:diagonal` *grid &key offset indices destination start* [Function]

Select a subgrid where the two indices are equal or differ by the offset, e.g. the diagonal affi for the matrix. The offset specifies sub- (offset<0) or super- (offset>0) diagonals.

`(setf grid:diagonal)` [Function]

Set a subgrid where the two indices are equal or differ by the offset, e.g. the diagonal affi for the matrix. The offset specifies sub- (offset<0) or super- (offset>0) diagonals. If grid is not supplied, a grid of one higher dimension than diagonal with default element `0` is used.

`grid:set-diagonal` *grid function-or-value &optional offset value* [Function]

Set the diagonal of the grid by calling the function on the indices. If value is non-nil, then set it to function-or-value, ignoring the indices.

The functions `diagonal`, `(setf diagonal)` (or `set-diagonal`) get or set the part of the grid where two indices are equal or differ by a constant to another grid; that is, either the diagonal or a sub- or super-diagonal. The diagonal of a grid can be set to a fixed value or to a function of its indices.

The diagonal is the collection of elements where there are two indices equal, or differ by a fixed amount. For a matrix (two dimensional grid), this would be for example:

```
ANTIK-USER> (grid::test-grid-double-float 'array '(3 4))
#2A((0.0 1.0 2.0 3.0) (10.0 11.0 12.0 13.0) (20.0 21.0 22.0 23.0))
ANTIK-USER> (diagonal @)
#(0.0 11.0 22.0)
```

The superdiagonal is accessible with the same function,

```
ANTIK-USER> (diagonal (grid::test-grid-double-float 'array '(3 4)) :offset 1)
#(1.0 12.0 23.0)
```

as is the subdiagonal,

```
ANTIK-USER> (diagonal (grid::test-grid-double-float 'array '(3 4)) :offset -1)
#(10.0 21.0)
```

### 4.3.4.5 drop

`grid:drop` *grid &key destination drop* [Function]

Remove singleton axes (axes with dimension 1) if the argument drop is true; otherwise return the input grid unchanged. The destination is an optional pre-existing grid in which to write the result.

The function `drop` will remove singleton axes (axes with dimension one) and create a grid of lower rank than the original grid. For example, a two-dimensional of dimensions 5 x 1

```
(test-grid-double-float 'array '(5 1))
#2A((0.0d0) (10.0d0) (20.0d0) (30.0d0) (40.0d0))
```

will be converted to a vector (one dimensional array),

```
(drop (test-grid-double-float 'array '(5 1)))
#(0.0d0 10.0d0 20.0d0 30.0d0 40.0d0)
```

### 4.3.4.6 identity-matrix

`grid:identity-matrix` *dimension &optional scalar type element-type* [Function]

A rank-two grid with the off-diagonals zero and the diagonal equal to scalar.

The function `identity-matrix` will create a matrix that has the same number on each element of the diagonal and zeros for other elements.

```
ANTIK-USER> (identity-matrix 3 8.0d0)
#2A((8.0 0.0 0.0) (0.0 8.0 0.0) (0.0 0.0 8.0))
```

#### 4.3.4.7 matrix-from-columns

`grid:matrix-from-columns` *&rest columns* [Function]

Make the matrix out of the equal-length vectors. If *default-grid-type* is non-nil, resultant grid will be of the specified type; otherwise, it will be the same as the first column.

#### 4.3.4.8 row

`grid:row` *grid index &optional destination* [Function]

The subgrid with the first index set to the specified value.

`(setf grid:row)` [Function]

Set the subgrid with the first index set to the specified value.

The functions `row`, `(setf row)` select or set the row of a matrix. For example, select the second row from the matrix above:

```
(row (test-grid-double-float 'array '(3 4)) 1)
#(10.0d0 11.0d0 12.0d0 13.0d0)
```

#### 4.3.4.9 slice

`grid:slice` *grid index-selection &key destination drop* [Function]

Select slice(s) from a grid. The index-selection is a list with length equal to the rank of grid. Each element should be one of: an index, indicating the index to be selected, :all, indicating the entire range if indices are to be selected, :rev, indicating the entire range if indices are to be selected in reverse order,

*(:range start end stride), indicating a range of indices to be*
selected; if stride is omitted, it is presumed to be `1`,

(:select value ...), indicating explicit values to be selected.

A *slice* is a subgrid selected by specifying index values. They can be specified as a single value, range of values, or reversed values.

```
(grid::test-grid-double-float 'array '(3 4))
#2A((0.0 1.0 2.0 3.0) (10.0 11.0 12.0 13.0) (20.0 21.0 22.0 23.0))
(slice (grid::test-grid-double-float 'array '(3 4)) '(1 (:range 0 2)) :drop nil)
#2A((10.0d0 11.0d0 12.0d0))
```

#### 4.3.4.10 stride

`grid:stride` *grid stride &key destination* [Function]

Create a new grid with every stride-th element.

A *stride* is the grid resulting from selection of elements at regular intervals.

```
(stride #(1 2 3 4 5 6 7 8) 3)
#(1 4 7)
```

The resultant grid will have rank one (vector) regardless of the source rank.

#### 4.3.4.11 subgrid

`grid:subgrid` *grid dimensions start &key destination drop* [Function]

Create a grid from a block of an existing grid, e.g. a row from a matrix, or a block. The dimensions give the dimensions of the subgrid, and start gives the lowest values of each index which corresponds to the subgrid element with all index values zero. If drop is true, all singleton axes are dropped.

`(setf grid:subgrid)` [Function]

Set the subgrid of the grid. Specify the starting indices with start, and in the case that the subgrid has lower rank than the grid, which axes; default is the first (rank subgrid) axes.

The functions `subgrid`, `(setf subgrid)` select or set a region within a grid as a grid. For example, The 2 by 2 block starting at index 1,2 in the previous matrix is

```
(subgrid (test-grid-double-float 'array '(3 4)) '(2 2) '(1 2))
#2A((12.0d0 13.0d0) (22.0d0 23.0d0))
```

#### 4.3.4.12 transpose

`grid:transpose` *grid &key indices destination start* [Function]

Transpose the grid, optionally putting the result in destination at the element specified by the start indices.

The function `transpose` exchange elements paired by exchange of indices. The grid can be of any rank greater than or equal to 2. For example, the transpose of the above array is

```
ANTIK-USER> (grid::test-grid-double-float 'array '(3 4))
#2A((0.0 1.0 2.0 3.0) (10.0 11.0 12.0 13.0) (20.0 21.0 22.0 23.0))
ANTIK-USER> (transpose @)
#2A((0.0 10.0 20.0) (1.0 11.0 21.0) (2.0 12.0 22.0) (3.0 13.0 23.0))
```

#### 4.3.4.13 Vector products and norms

`grid:cross` *grid0 grid1* [Function]

The cross product of two vectors, using the first three components of each.

`grid:euclidean` *grid &optional kind* [Function]

The norm of the grid. Kind can be :euclidean, for the euclidean, or 2-norm.

`grid:inner` *grid0 grid1* [Generic Function]

The inner product of two grids.

`grid:norm` *grid &optional kind* [Function]

The norm of the grid. Kind can be :euclidean, for the euclidean norm.

`grid:normalize` *grid &optional threshold* [Function]

Find the normalized grid, i.e., each element is divided by grid norm, and the normalization factor. If the norm is less than the non-nil threshold, then nil is returned; if it is zero and threshold is nil, a zero grid is returned.

### 4.3.5 Mapping

There are two more general functions `map-grid` and `map-n-grids` on which the above functions are defined. They will provide the basis for any elementwise mapping of one (for the former) or several (for the latter) grids into a destination grid.

#### 4.3.5.1 elementwise

`grid:elementwise` *function &optional toggle-physical-dimension* [Function]

Make a function on a grid as an elementwise map of a scalar function. If the result has no physical dimension but the argument does, or vice versa, toggle-physical-dimension should be `T`.

```
(funcall (grid:elementwise 'sqrt) #(1.0d0 2.0d0 3.0d0 4.0d0))
#(1.0 1.4142135623730951 1.7320508075688772 2.0)
```

#### 4.3.5.2 map-grid

`grid:map-grid` *&key source source-affi source-dims destination destination-affi destination-specification initial-element element-function combination-function* [Function]

Make a new grid by mapping on an existing grid or on indices. :source The source grid; if not a grid, it is taken as a function to apply to the grid indices to make the element of the destination; if this is supplied, source is ignored and element-function is only applied to the default value.

*:source-affi The affi to be used for extraction; defaults to*
making an affi from source-dims, then destination-specification.

*:source-dims The dimensions of the source if source not supplied;*
if `nil`, dimensions are taken from destination-specification.

*:destination The destination grid, if not supplied, it will be made*
according to destination-affi and destination-specification.

*:destination-affi The affi for injection, defaults to (affi destination)*
if destination is supplied, then makes an affi if destination-specification is supplied, otherwise source-affi.

*:destination-specification The specification to use for the destination to be make,*
defaults to the specification of source.

*:initial-element The default value to set in a newly-created destination.*
*:element-function The function to apply to each element of the source; defaults*
to coercing element to the element type of the destination.

*:combination-function*
A designator for a function of two arguments, or nil (default). If a function, it will be funcalled on the destination element and the transformed source element. If nil, the destination element is overwritten.

Although this function has other uses, it can be used to create a grid using a function of the index values. For example, in the file '`antik/grid/tests/grids.lisp`' is a function `index-fill-decadal` that multiplies increasing powers of ten by each argument in succession, and adds the result. The array `*array-3-4-double-float*` is created with this function:

```
(defparameter *array-3-4-double-float*
  (map-grid :source 'index-fill-decadal :source-dims '(3 4)))
```

which gives

```
*array-3-4-double-float*
#2A((0.0d0 1.0d0 2.0d0 3.0d0)
    (10.0d0 11.0d0 12.0d0 13.0d0)
    (20.0d0 21.0d0 22.0d0 23.0d0))
```

(see also the function 'test-grid-double-float).

Take the square root of every element of an array:

```
(map-grid :source #m((0.0d0 1.0d0 2.0d0) (10.0d0 11.0d0 12.0d0) (20.0d0 21.0d0 22.0d0)
```

Make a foreign vector with each element the square root of its index:

```
(map-grid :source 'sqrt :destination-specification '((foreign-array 6) double-float))
```

#### 4.3.5.3 map-n-grids

`grid:map-n-grids` *&key sources destination destination-affi destination-specification initial-element element-functions combination-function combine-destination* [Function]

Map on multiple source grids. The arguments are: sources: A list of (grid-or-function affi), each one representing either a grid or function from which data will be drawn. If it is a grid, the grid values are used. If it is a function of the appropriate number of non-negative integer arguments, values will be created by calling that function on the indices. If affi is not specified and grid-or-function is a grid, the affi will default to the `affi` of that grid.

*destination:*

A grid in which to put the results, or nil if one is to be created.

*destination-affi:*

An `affi` for the destinattion.

*destination-specification:*

A specification for the destination, if destination-affi is not given.

*initial-element: Initial value to give to the destination grid.*

*element-functions:*

A list of functions or nil, one for each source. This function is applied to the source element before combination.

*combination-function:*

A function that combines each of the source values after being passed through the element-functions. If nil, then just take the value from the first source.

*combine-destination:*
If true, pass as the first argument to the combination-function the prior value of the destination grid. Defaults to `t` if :desination is specified; otherwise nil.

This is a more general form of `map-grid` which can take multiple source grids, instead of only one. For example, combine arrays a and b as a+2b:

```
(map-n-grids :sources ‘((,#31m(1 2 3) nil) (,#31m(9 8 7) nil))
             :combination-function (lambda (a b) (+ a (* 2 b))))
```

### 4.3.6 Properties

These functions give properties of a grid.

`grid:contents` *grid* [Function]
The contents of the grid as nested lists. This can be used as the :initial-contents argument in making a new grid.

`grid:dim0` *object* [Function]
The first dimension of the object.

`grid:dim1` *object* [Function]
The second dimension of the object.

`grid:dimensions` *grid* [Generic Function]
A list representing the dimensions of the generalized array.

`grid:dimensions` (*foreign-array foreign-array*) [Method]
automatically generated reader method

`grid:gridp` *object* [Function]
Object is a grid; type is returned.

`grid:rank` *grid* [Generic Function]
The rank (number of dimensions) of the generalized array.

`grid:specification` *grid* [Generic Function]
The grid specification.

### 4.3.7 AFFI

`extrude` will transform an AFFI into one that will make it appear that the grid has an extra dimension. This is useful in e.g. `map-n-grids`.

## 4.4 Input and Output

Note: the definitions for reading and writing grids are at present minimal and incomplete.

These definitions allow reading and writing of grids from and to files.

`grid:read-data-file` *filename &optional separator include-count discard-header* [Function]

Read the data in the file into a list.

`grid:read-data-stream` *&optional separator include-count stream* [Function]

Read the data from the stream and collect a list of lists of values from columns separated by whitespace. If include-count is a number, line count starting at that number will be prepended as the first field.

`grid:read-vector-from-string` *string count &optional start end* [Function]

Read the count element vector from a string.

## 4.5 Physical quantities

A grid may also be a physical quantity. In this case, the magnitude is a numeric grid, and the units are either scalar, meaning the same unit applies to all numbers, or mixed, meaning that one set of units applies to each element.

```
ANTIK-USER> #_#m(1.0 2.0 3.0)_km
#_#m(1000.000000000000000d0 2000.000000000000000d0 3000.000000000000000d0)_m
ANTIK-USER> #m(#_1.0_km #_2.0_s #_3.0_kg)
#m(#_1000.000000000000000d0_m #dPT2.000s #_3.000000000000000d0_kg)
```

See Chapter 5 [Physical Quantities], page 26 for more details on physical quantities, keeping in mind that whereever a scalar magnitude is shown there, a grid may be used instead.

## 4.6 Capabilities and Internals

The capabilities of the foreign-array system depend on the CL implementation and the platform. This section describes ways of obtaining information about the capabilities, and other internal information. It will be of greatest interest to someone writing an interface to a foreign numeric library.

### 4.6.1 Classes

`grid:foreign-array` [Class]

Class precedence list: `foreign-array, standard-object, t`

Arrays that can be referenced by foreign functions.

`grid:matrix` [Class]

Class precedence list: `matrix, foreign-array, standard-object, t`

Foreign matrices.

`grid:mvector` [Class]

Class precedence list: `mvector, foreign-array, standard-object, t`

Foreign vectors.

The subclasses of either mvector or matrix are named with the element type, as described in the table in Section 4.2.2 [The foreign-array class], page 9.

### 4.6.2 Functions and Macros

`grid:all-types` *alist &optional right-side* [Function]

A list of all types defined by symbol or definition.

`grid:cffi-cl` *cffi-type* [Function]

The `cl` type from the `cffi` element type.

`grid:check-dimensions` *dimensions &optional errorp* [Function]

Check that dimension specification for a grid is a list of positive integers.

`grid:cl-cffi` *cl-type* [Function]

The `cffi` element type from the `cl` type.

`grid:data-class-name` *category-or-rank element-type &optional making-class* [Function]

The class name from the type of element.

`grid:dcref` *double &optional index* [Macro]

Reference `c` double(s).

`grid:element-size` *object* [Function]

The size of each element as stored in `c`.

`grid:element-types` *element-types* [Function]

All the element types available of a given category.

`grid:element-type` *grid* [Generic Function]

The element type of the grid.

`grid:element-type` (*foreign-array foreign-array*) [Method]

automatically generated reader method

`grid:lookup-type` *symbol alist &optional reverse error* [Function]

Lookup the symbol defined in the alist. If error is non-nil it should be a string describing the class of types, and this function will signal an error if the type wasn't found.

`grid:metadata-slot` *object name* [Macro]

Access a slot in the foreign-metadata.

`grid:number-class` *type* [Function]

Find the class corresponding to the numeric type.

`grid:spec-scalar-p` *specification* [Function]

Specification is for a scalar; if so, return that type.

`grid:st-pointerp` *decl* [Function]

If this st represents a pointer, return the type of the object.

### 4.6.3 Condition

`grid:array-mismatch` [Condition]

Class precedence list: `array-mismatch`, `error`, `serious-condition`, `condition`, `t`

An error indicating that the two arrays do not have the same dimensions.

### 4.6.4 Constants and Variables

`grid:+foreign-pointer-class+` [Constant]

The class in which foreign pointers fall.

`grid:+foreign-pointer-type+` [Constant]

The type of foreign pointers.

`grid:*array-element-types-no-complex*` [Variable]

All the array element types supported except for complex types.

`grid:*array-element-types*` [Variable]

All the array element types supported.

`grid:*complex-types*` [Variable]

All the supported complex array element types.

`grid:*cstd-cl-type-mapping*` [Variable]

An alist of the `c` standard types as keywords, and the `cl` type The exception is complex types, which don't have a definition in the `c` standard; in that case, the `c` type is the foreign struct definition.

`grid:*cstd-integer-types*` [Variable]

List of integer types supported by `cffi`, from the `cffi` docs.

`grid:*double-types*` [Variable]

All the supported double float element types.

`grid:*float-complex-types*` [Variable]

All the float or complex array element types supported.

`grid:*float-types*` [Variable]

All the float array element types.

`grid:*grid-types*` [Variable]

A list of (disjoint) types that are accepted as grids. Not every object of the given type is necessarily a grid, however.

`grid:*print-contents*` [Variable]

Print the contents of the foreign-array.

`grid:*print-foreign-array-readably*` [Variable]

Print the contents of the foreign-array with the #m reader macro.

## 4.7 Efficiency

Access to elements of foreign arrays, for getting or setting, can be very slow if they are not declared. These declarations can take several forms: `declare`, `declaim`, `proclaim`, or `the`. With the exception of the `the` form, these declarations will only be effective in SBCL and CCL. Users of other implementations that support the environment function `variable-information` should contact the Antik maintainer to have support added. If declarations are present and the compiler honors the compiler macros that foreign-array defines, the `aref` and related forms are macro-expanded directly into foreign array calls, considerably increasing the speed of execution.

When using matrices, declarations with explicit dimensions are also helpful for speed, e.g.

```
(declare (type (grid:matrix-double-float 100 100) my-matrix new-matrix))
```

For setting array elements, it is better to use the macro `grid:gsetf` than `setf` when declarations are present. It is hoped that eventually this macro can be eliminated.

For example, the declarations in the following function, and the use of `grid:gsetf` instead of `setf`, help decrease the execution time when running this function:

```
(defun foreign-array-test (dim)
  (let ((input (grid:make-foreign-array
'double-float
:dimensions dim
:initial-element 1.0d0))
(output (grid:make-foreign-array 'double-float :dimensions dim)))
    (declare (type grid:vector-double-float input output))   ; declaration of foreign
    (let ((tv0 0.0d0) (tv1 0.0d0))
      (declare (type double-float tv0 tv1))
      (iter (for i from 0 below dim)
    (setf tv0 0.0d0)
    (iter (for m from 0 to i)
  (iter (for n from i below dim)
(setf tv1 0.0d0)
(iter (for k from m to n)
      (incf tv1 (grid:aref input k)))
(incf tv0 (expt tv1 -2))))
    (grid:gsetf (grid:aref* output i) (- (grid:aref input i) tv0))))))
```

# 5 Physical Quantities

## 5.1 Units

## 5.2 Introduction

*Physical quantities* are numerical values that are used in science, engineering, and other numerical computations. There are several aspects to them:

- They may have *physical dimension*, such as length, mass, time, etc.
- They may have *attributes* related an object, such as the diameter of a circle.

The symbols defined for physical quantities are in the `antik` package.

## 5.3 Physical dimension

### 5.3.1 Introduction

Antik has a class `physical-quantity` which is used to represent values that have physical dimension, such as mass or length; e.g.

```
ANTIK-USER> #_35_km
#_35000.000000000000000_m
ANTIK-USER> (* #_5_m #_8_kg)
#_40.000000000000000_m-kg
ANTIK-USER> (expt #_12_m 2)
#_144.000000000000000_m^2
(* #_5_kg #_8_m/s^2)
#_40.000000000000000d0_N
```

Mathematical operations are allowed on these objects,

```
ANTIK-USER> (+ #_35_km #_12_miles)
#_54312.128000000000000_m
```

Of course, such math must be sensible,

```
ANTIK-USER(3): (+ #_35_km #_12_kg)
debugger invoked on a SIMPLE-ERROR in thread #<THREAD "initial thread" RUNNING {100402
  The quantities 3.500d+4m and 12.00kg are not both physical quantities with the same
0]
```

Note also that fractional exponents are acceptable:

```
ANTIK-USER> (expt #_16_m 1/2)
#_4.000000000000000_m^1/2
```

Each of these quantities has *units* associated with it – meters, kilograms, etc., and those units must be of the appropriate dimension for the physical quantity. All quantities are stored internally in the SI system of units, and converted to desired units either when printed or when an explicit conversion is requested.

The inspiration for this software came from Gordon Novak's work. Names, abbreviations, and definitions of units come from NIST.

### 5.3.2 Making physical dimension quantities

There are two main ways to make physical quantities: the function `make-pq`, and the reader macro `#_`

```
ANTIK-USER> (make-pq 24.5d0 'm/s)
#_24.500000000000000d0_m/s
ANTIK-USER> #_24.5_m/s
#_24.500000000000000d0_m/s
```

To get the magnitude and units of a physical quantity, use `pqval`. It returns three values: the magnitude, units, and whether a single unit expression applies to all elements of the grid magnitude:

```
ANTIK-USER> (pqval #_24.5_m/s)
24.5
(/ METER SECOND)
T
ANTIK-USER> (pqval #m(#_1.0_km #_2.0_s #_3.0_kg))
#m(1000.000000000000000d0 2.000000000000000d0 3.000000000000000d0)
#(METER SECOND KILOGRAM)
NIL
```

`antik:physical-quantity` [Class]

Class precedence list: `physical-quantity, standard-object, t`

A quantity with a dimension and possibly a unit, e.g. length or meter.

`antik:check-dimension` *obj units &optional errorp zeros-have-any-dimension* [Function]

`t` if one of these cases hold:

- obj is a pdq and units have the same dimension as obj,
- obj is zero and zeros-have-any-dimension is `t`,
- obj and units represent a dimensionless quantity,
- obj and units are grids of the same length, and for each pair of corresponding elements, one of the above is true.

`antik:make-pq` *magnitude unit-expression &optional sysunits scalar-dimension* [Function]

Make a physical dimension quantity (PDQ). Allow for the possiblity that magnitude is already a `pdq`; in this case, check that the physical dimension is correct, and pass it on. This can be used as a way of assigning default units to a number. If sysunits is specified, unitless dimensions (e.g., 'length) are interpeted as units in that system. If scalar-dimension is `t` and magnitude is not a scalar, the unit-expression applies to each element of the magnitude. Otherwise it must be a grid or list of the shape as the magnitude.

`antik:new-units` *pq &optional sysunits-additional-units stream* [Function]

Format the physical dimension quantity in units different than the default. Note that sysunits-additional-units must be a list.

`antik:pqval` *pq &rest sysunits-additional-units* [Generic Function]

Get the numerical value and the units of the physical dimension quantity. Returns the magnitude, the units, and whether units are the same for each element of a sequence or grid.

`antik:with-pq` *physical-quantities &body body* [Macro]

Ensure that the named physical dimension quantities are of the right dimensions, or convert them from plain numbers using the current system of units as specified by *system-of-units*.

`antik:*zero-is-dimensionless*` [Variable]

Numbers with zero magnitude are made dimensionless.

### 5.3.3 Dimensions

The fundamental physical dimensions handled by Antik are:

- length
- time
- mass
- temperature
- current
- substance
- luminosity
- money
- angle

In addition, there are derived dimensions:

- force
- area
- volume
- power
- energy
- velocity
- momentum
- angular-momentum
- acceleration
- pressure
- density
- charge
- electric-potential
- capacitance
- resistance
- conductance

- magnetic-field
- magnetic-flux
- inductance
- frequency
- dose

For units defined by Antik, see the source file ‘`antik/physical-quantities/unit-definitions.lisp`’. Most of the common units for each of the listed physical dimension are defined, together with common abbreviations.

### 5.3.4 Systems of units

A *system of units* is a set of units, one for each of the physical dimensions that Antik can handle. A system of units is needed by Antik for printing output and in special cases for converting physical quantities that have a physical dimension defined (e.g. length), but no units. The system of units defined by `*system-of-units*` is used to determine the units in which the output is printed. There are pre-defined several systems of units, `*siu*`, `*englishu*`, `*cgsu*`:

```
ANTIK-USER> (defparameter *length1* #_10_m)
*LENGTH1*
ANTIK-USER> *length1*
#_10.000000000000000_m
ANTIK-USER> (setf *system-of-units* *englishu*)
(((1 0 0 0 0 0 0 0 0) . FOOT) ((0 0 0 0 0 0 0 0 1) . RADIAN)
 ((0 0 1 0 0 0 0 0 0) . SLUG) ((0 1 0 0 0 0 0 0 0) . SECOND)
 ((1 -2 1 0 0 0 0 0 0) . POUND-FORCE) ((2 -3 1 0 0 0 0 0 0) . HORSEPOWER)
 ((2 -2 1 0 0 0 0 0 0) . FOOT-POUND)
 ((-1 -2 1 0 0 0 0 0 0) . POUNDS-PER-SQUARE-INCH))
ANTIK-USER> *length1*
#_32.808398950131235_ft
```

So by setting the system of units, you can convert values as needed,

```
ANTIK-USER> (setf *system-of-units* *siu*)
ANTIK-USER> #_12_feet
#_3.657600000000000d0_m
ANTIK-USER> (setf *system-of-units* *englishu*)
ANTIK-USER> #_3_m
#_9.842519685039370d0_ft
```

A new system of units may be defined with `define-sysunits`. One need not define all physical dimensions, but if a unit is needed for a particular dimension and it’s not defined, an error will be signalled. For convenience, a system of units may be defined by alteration of an existing system of units. For example, suppose that you wish to have a system of units that is like SI except that lengths are measured in km instead of meters. This will define such a system:

```
(define-sysunits *kmu* (km) *siu* "Kilometer system.")
```

To set the default system of units, use `set-default-sysunits`. To define a symbol macro to set a default system of units, use `setsys`. The SI system is the initial default system of units. It may be reset as the system of units with the symbol macro `si`.

For formatted output of angles, the units (degrees or radians) may be set with the nf option `:degrees`, so the system of units may be maintained whether degrees or radians are desired on output.

`antik:define-sysunits` *system-name units &optional base-sysunits docstring* [Macro]

Make a system of units with the given name and units, optionally agumenting the existing system of units base-sysunits.

`antik:set-default-sysunits` *sysunits &rest additional-units* [Function]

Set the default system of units to the specified system, possibly augmenting it with other units. additional-units may be `nil` to augment the current default system.

`antik:setsys` *symbol sysunits &rest additional-units* [Macro]

Set the symbol macro symbol to make the system of units sysunits default. additional-units may be useform{NIL} to augment the current default system.

`antik:*system-of-units*` [Variable]

Default system of units used by pqval.

`antik:*siu*` [Variable]

Systeme internationale system of units.

`antik:*englishu*` [Variable]

The English system of units.

`antik:*cgsu*` [Variable]

The `cgs` system of units.

## 5.4 Physical constants

It is possible to define physical constants which may then be used either in a mathematical expression or in a physical quantity definition

```
ANTIK-USER> (define-physical-constant smoot
    #_1.70180_m (ors) smoot
  "The unit of length named after Oliver R. Smoot.")
ANTIK-USER> (* 3 smoot)
#_5.105399999999999d0_m
ANTIK-USER> #_3_ors
#_5.105399999999999d0_m
```

`antik:define-physical-constant` *name value &optional synonyms print-name docstring* [Macro]

Define the variable to be the physical dimension quantity given, and also make it a unit of measure.

## 5.5 Date and time

Dates and times can be represented in Antik with the `timepoint` class. A timepoint is a specific point in Newtonian time, so includes a date and time, and a *timescale* (best thought of as a timezone). Timepoints are most conveniently made with the *#d* reader macro; the default timescale is UTC, and the standard format for specification of timepoints is ISO 8601

```
ANTIK-USER> #d2011-08-04T12:00
2011-08-04 12:00:00.000
#d2011-08-04T12:00EST
2011-08-04 17:00:00.000
```

Other timescales available are UT1, TAI, GPS, and any of the customary time zone abbreviations that define offsets from UTC; these are listed in the variable `*timescales*`. In order to convert to and from UT1, it is necessary to obtain earth orientation parameters from the US Naval Observatory; if **real-ut1-utc** is , then UT1 is taken the same as UTC.

The function `read-time` will read a timepoint or time interval in ISO8601 format. Timepoints may be read in any format using the function `read-timepoint`. The default is the ISO8601 format, with a broadened allowance for separators between the components. A day without a specific time will be marked :day-only. The function `read-us-date` is provided to read US-style dates (month/day/year).

```
ANTIK-USER> (read-time "2011-08-04T12:00")
2011-08-04 12:00:00.000
ANTIK-USER> (read-time "1999?03/30    % 12-33-45")
1999-03-30 12:33:45.000
ANTIK-USER> (read-timepoint "20/12/2011" '(2 1 0)) ; Read a European-style date
2011-12-20
ANTIK-USER> (read-us-date "12/20/2000 3:41:12")
2000-12-20 03:41:12.000
ANTIK-USER> (read-us-date "12/20/11 3:41:12") ; Two-digit years are acceptable
2011-12-20 03:41:12.000
```

Dates are output in ISO8601 format. The function `write-us-date` can be used to format in US style.

**`antik:timepoint`** [Class]

Class precedence list: `timepoint, dtspec, standard-object, t`

Specification of a point in time, including a scale.

**`antik:*real-ut1-utc*`** [Variable]

If `nil`, `ut1` will be taken the same as `utc`.

**`antik:read-time`** *string* [Function]

Parse the datime or time interval string and create a timepoint object.

**`antik:read-timepoint`** *string &optional pos-ymdhms scale* [Function]

Read a timepoint from a string with specification for the position in the string of each component; pos-ymdhms is a list of year, month, day, hour, minute, and second as sequence numbers for the integers in the string. Scale is the timescale (zone) as a string or symbol. If pos-ymdhms has only three components, or only a date is provided, the timepoint created will be specifed as day-only. The default reads an `iso8601` string like `1999-03-30t12:33:45`.

`antik:read-us-date` *string &optional day-only* [Function]

Read dates and times in customary `us` format MM/DD/YYYY; times may be included as well if day-only is nil.

`antik:write-us-date` *datime* [Function]

Write dates and times in customary `us` format MM/DD/YYYY.

## 5.6 Time arithmetic and time intervals

The difference of two timepoints may be used to compute a time interval, and an interval added to a timepoint to create another timepoint. Intervals are ordinary physical dimension quantities with dimension of time, and may be format for input or output in several ways. The reader macro `#d` and the function `read-time` will read in an interval in ISO8601 format; it should begin with "P" as required by that standard.

All components are of fixed size; for example, a month is 30 days, and a year is 12 months. This may produce some odd-looking results because a year will have only 360 days.

```
ANTIK-USER> (+ #d2011-08-01T12:00 #DP1m3d)
2011-09-03 12:00:00.000
ANTIK-USER> (- #d2011-09-03T12:00:00.000 #d2011-08-01T12:00)
#dP1m3dT
ANTIK-USER> (- #d2011-09-03T12:00:00.000 #d2010-09-03T12:00:00.000)
#dP1y5dT
```

Formatting for an interval is specified by setting the nf parameter `:time` to `:tud` for an ISO8601 time unit designator, `:alternative` for the ISO8601 alternate, or  for the physical dimension time. Default is `:tud`.

```
ANTIK-USER> (set-nf-options :time :tud)
ANTIK-USER> #_12345_seconds
#dPT3h25m45.000s
ANTIK-USER> (set-nf-options :time nil)
NIL
ANTIK-USER> #dP8m3dT5h
#_21013200.000000000000000d0_s
```

# 6 Numerical Output Format

The function `nf` is provided to format numbers according to specific criteria set as parameters in the `:nf` category (see Section 3.2 [Parameters], page 5). There is precise control over the number of digits printed, how grids are printed, etc. To show all parameters available, evaluate `(parameter-help :nf)`; to get information about each one, `(parameter-help :nf` *`name`*`)`.

| **Parameter** | **Values** | **Description** |
|---|---|---|
| `components` | list of strings | Names of vector components to use when `vector-format` is `:coordinate-unit-vectors`. |
| `date-time-separator` | , `t`, character | Character to place between date and time in ISO8601 datime output. If nil, use a space, if T, use #\T. |
| `degrees` | boolean | Whether to format angles in degrees and angular rates in Hertz. |
| `fracpart-digits` | , fixnum | The number of digits to the right of the decimal floating point numbers when :significant-figures is . |
| `full-precision` | , `t` | If not , ignore `:fracpart-digits`, `:intpart-digits`, `:significant-figures`, and format floating point numbers to full precision. |
| `horizontal-element-separator` | character | What to put between horizontally separated elements for plain style. |
| `ignore-day-only` | boolean | If true, a timepoint specified day-only will show the time part as well. |
| `intpart-digits` | , fixnum | The minimum space allowed for the whole-number part of numbers when :significant-figures is . If this is larger than the actual number of digits, pad to the left with spaces. If this value is , allow enough space to accomodate the number. |
| `no-units` | boolean | Don't print units if true. |
| `print-sign` | boolean | Whether leading '`+`' is printed. |
| `significant-figures` | , fixnum | The number of significant figures formatted for numbers. If , formatting of numbers is done using :intpart-digits and, if a float, :fracpart-digits. |
| `style` | , `:tex`, fixnum | Style of format: plain (standard or shortened) or LaTeX. |
| `tex-decimal-align` | boolean | Align columns of numbers on decimal point in LaTeX. |
| `tex-element-separator` | string | Character to put between vertically separated matrix rows for LaTeX. |
| `time` | ,`:tud`,`:alternative` | How to format time intervals: with unit for time (), ISO8601 time-unit designator (`:tud`), ISO8601 alternative (`:alternative`). |

| | | |
|---|---|---|
| `timepoint-linear` | symbol, list | Convert timepoints to a linear scale if specified as a list of epoch time and unit. For example `'(*midnight-2000* :year)` will present time points as a real number of calendar years, including fractions, such as 3.3223. If the value is a symbol representing a unit, like :year, the epoch is taken as 0. |
| `vector-format` | `:horizontal`, `:vertical`, `:coordinate-unit-vectors` | Vectors are formatted as rows, columns or as linear combination of coordinate unit vectors. |
| `vertical-element-separator` | character | Character to put between vertically separated elements for plain. |

**`antik:nf`** *object &optional stream* [Generic Function]

Format output for numerical objects. If stream is nil, use *standard-output*.

**`antik:nf`** (*object float*) *&optional* (*stream *standard-output**) [Method]

Returns two values: the string and prints-as-zero.

**`antik:nf-option`** *name* [Macro]

Get/set the nf option named.

**`antik:nf-readably`** [Function]

The format produced will be readable to Lisp.

**`antik:nf-string`** *object* [Function]

Format output for numerical objects to a new string.

**`antik:set-nf-options`** *&rest name-values* [Macro]

Set the numerical formatting options.

**`antik:with-nf-options`** (**&rest** *name-values*) *&body body* [Macro]

Set the options for all nf calls within body to inherit.

**`antik:object-as-nf`** *object* [Function]

Define a new object exactly as the current nf options print an existing one. This function can be used for example to define a number exactly as the rounded print format of another one.

# 7 Mathematical Utility

This module has definitions for mathematical utilities.

`antik:angle-law-of-cosines` *length-opp length-adj1 length-adj2* [Function]

Solve for the angle when lengths of a triangle are known.

`antik:prime-factors` *n* [Function]

Find the prime factors of the integer n.

# 8 Three Dimensional Space

This module has definitions related to Cartesian 3-space and rotations.

## 8.1 Cartesian

`antik:coordinate-unit-vector` *i &optional length* [Function]

A coordinate unit vector.

`antik:coplanar` *vect1-or-matrix &optional vect2 vect3* [Function]

The sine of the angle between vect1 and the cross product of vect2 and vect3. This should be near zero.

`antik:distance` *a b* [Function]

The length of the vector and the vector difference a-b.

`antik:first-3vector` *vec* [Function]

Extract the first 3-vector.

`antik:right-angle` *vector* [Function]

Find an arbitrary right angle to the vector.

`antik:second-3vector` *vec* [Function]

Extract the second 3-vector.

`antik:vector-angle` *a b* [Function]

The short angle between a and b, and the rotation vector.

## 8.2 Polar

`antik:polar-to-rectangular` *azimuth elevation &optional radius sign1* [Function]

Convert polar coordinates to rectangular coordinates. Argument sign1 should be set to `-1` for accomodating topocentric azimuth, which is measured from North instead of South.

`antik:rectangular-to-polar` *vector &optional sign1* [Function]

Convert rectangular coordinates to polar coordinates. Argument sign1 should be set to `-1` for accomodating topocentric azimuth, which is measured from North instead of South. Returns a list of azimuth, elevation, and radius, and the plane distance.

## 8.3 Rotation

`antik:euler-angle-rotation` *rot3a rot1 rot3b &optional passive* [Function]

Compute the matrix for a `3-1-3` rotation. See Curtis (9.118).

`antik:rotate` *angle &optional passive* [Function]

Create a rotation matrix in two dimensions. Passive rotation if passive is true, otherwise active.

`antik:rotate-3d` *axis angle &optional passive* [Function]

Create a rotation matrix about the (0, `1`, 2) axis in three dimensions. Passive rotation if passive is true, otherwise active.

`antik:rotate-3daa` *rotation-axis angle &optional passive* [Function]

Rotation about an arbitrary axis in `3d`. See Goldstein, Poole, Safko (4.62) or http://mathworld.wolfram.com/RotationFormula.html.

`antik:tait-bryan-angles` *dcm* [Function]

Find the three angles psi, theta, phi of a `3-2-1` (Tait-Bryan) rotation for the direction cosine matrix.

`antik:tait-bryan-rotation` *rot1 rot2 rot3 &optional passive* [Function]

Compute the matrix for a `3-2-1` (yaw-pitch-roll) rotation. See Curtis (9.124). This transforms a vector in the body frame into a vector in the external frame.

# 9 Higher

These functions provide a more straightforward interface to foreign libraries like GSLL. For example, GSLL supplies several functions to invert a matrix, but the function defined here `invert-matrix` uses LU decomposition with all the necessary preparation.

## 9.1 Linear Algebra

These functions require that GSLL be loaded.

`antik:determinant` *matrix* [Function]

Find the determinant of the matrix.

`antik:invert-matrix` *matrix* [Function]

Invert the matrix; return the inverse and determinant.

## 9.2 Optimization

These functions require that GSLL be loaded.

`antik:maximize-1d` *function x-lower x-upper &optional x-maximum absolute-error relative-error method* [Function]

Find the maximum of the function of one variable (x) between x-lower and x-upper. The optional argument x-maximum is a guess of x for the maximum value of the function. Returns the argument and value at the maximum.

`antik:minimize-1d` *function x-lower x-upper &optional x-minimum absolute-error relative-error method* [Function]

Find the minimum of the function of one variable (x) between x-lower and x-upper. The optional argument x-minimum is a guess of x for the minimum value of the function. Returns the argument and value at the minimum.

`antik:root-1d` *function initial derivative &optional absolute-error relative-error method* [Function]

Find the root of a real function of a real variable. Returns the argument value of the root and the computed value of the function at that value. The argument and return values may be physical quantities.

# 10 Internals

This chapter describes some of the designs used internally, for the benefit of those who wish to port libraries to use some of the definitions in Antik, or those who are merely curious. If you wish to simply use Antik, you can skip this chapter.

## 10.1 Foreign Library Kit

Antik is designed with the idea that scientific and engineering libraries, particularly foreign (non-Lisp) libraries, can easily be integrated, so that data can be exchanged with and among the libraries. Function and other definitions, when duplicated, will be named the same, differing only in the package. For example, if libraries '`foo`' and '`bar`' both define a singular value decomposition (SVD), they would use the function names `foo:sv-decomposition` and `bar:sv-decomposition`, and the arguments would be as similar as possible. Then a user may call either one on a matrix (two-dimensional) grid, and compare results.

To facilitate the exchange of foreign arrays, the class `grid:foreign-array` has a slot `grid::foreign-metadata`. The interface code for a foreign library can save information there as a plist that the library needs in order to use the array, such as the dimensions.

## 10.2 Antik tests

Regression (unit) tests are available for both `antik` and `grid` if the lisp-unit system is installed,

# 11 Download and installation

The recommended way to download and install this software is through quicklisp. Install quicklisp following the directions, then do one of:

```
(ql:quickload "antik")
```

Users that need the very latest version will need to fetch the repository via git. The gitweb page gives a view of the commit history and information on cloning. To clone,

```
git clone git://repo.or.cz/antik.git
```

The `Antik` system depends on

- alexandria cffi, trivial-garbage, and split-sequence. In addition, there are two optional systems, FSBV, which permits usage of foreign arrays of complex number elements, and static-vectors gives foreign arrays a native Lisp view via the function `cl-array` for foreign arrays made in Lisp (not created in foreign code) for supported implementations (currently SBCL, CCL, ECL, LispWorks, and Allegro).

Note: FSBV is for the time being mandatory. Only one file is needed, and no foreign libraries, but until it is split into two separate systems, the whole thing must be loaded.

# 12 Copying

All code and documentation in this collection is copyright as indicated at the top of this document. It may be copied under the GPL v3.

# Index

## Concept index

## Function and Macro index

## Constants and Variables index