

ScientificPython User's Guide

Konrad Hinsen
Centre de Biophysique Moléculaire
CNRS
Rue Charles Sadron
45071 Orléans Cedex 2
France
E-Mail: hinsen@cnrs-orleans.fr

2001-3-29

Chapter 1

Introduction

ScientificPython is a collection of Python modules that are useful for scientific computing. Most modules are rather general, others belong to specific domains and will be of interest to only a small number of users (e.g. the module `Scientific.IO.PDB`). Almost all modules make extensive use of Numerical Python (NumPy), which must be installed prior to Scientific Python. Python 1.5 or later is also required. For more information about Numerical Python and about other packages for scientific computing, see the Topic Guide "Scientific Computing" on the Python home page.

This manual describes version 2.2 of ScientificPython. The 2.x versions are completely revised and *not* compatible with earlier releases. The major difference is the introduction of a package structure; all modules are now submodules of the top-level module `Scientific`. The package structure should prevent name clashes with other modules, which have occurred in the past (e.g. the module `PDB` was indistinguishable from the module `pdb` in the Python standard library on operating systems without case distinction in filenames).

For most code written for versions 1.x of ScientificPython, a change of the import statements should be all that is required to make it compatible with this release. The only module with significantly changed functionality is `Scientific.Statistics.Histogram`; the changes were necessary to implement an algorithm that is much more efficient for large data sets. There are of course also additional modules and enhancements to existing ones, but they don't affect compatibility. The most significant addition is the module `Scientific.IO.NetCDF`, which was previously distributed separately.

It should be noted that ScientificPython 2.x can coexist with an 1.x

Introduction

version, as all module names are different. However, the availability of both versions in parallel makes it difficult to verify that code has been fully ported to use the new one.

Chapter 2

Installation on Unix systems

The first step in installation is compilation of the C modules. There is currently only one C module, the netCDF interface module. If you do not need the netCDF module, you can skip the compilation step. If you do want to compile it, you must first install the netCDF library, version 3 or higher. Then copy the file Src/Setup.in to Src/Setup and edit the copy to reflect the installation of the netCDF library on your system. If the library archive is in /usr/local/lib and the header files in /usr/local/include, nothing has to be changed. Once the file Src/Setup is ready, type `./compile` in the top directory of the distribution.

The second step is the installation of ScientificPython in a publicly accessible place. This is done by typing `./install` in the top directory of the distribution. This copies the directory Scientific to the directory site-packages in the standard Python library. Depending on the location of this directory, installation of ScientificPython may require root privileges. However, ScientificPython can also be installed manually by copying the directory Scientific with all its contents into a directory on the Python search path.

Chapter 3

Reference for Module Scientific

Module Scientific.DictWithDefault

Class DictWithDefault: Dictionary with default values

Constructor: DictWithDefault(**default**)

Instances of this class act like standard Python dictionaries, except that they return a *copy* of **default** for a key that has no associated value.

Module Scientific.Functions

Module Scientific.Functions.Derivatives

This module provides automatic differentiation for functions with any number of variables up to any order. An instance of the class `DerivVar` represents the value of a function and the values of its partial derivatives with respect to a list of variables. All common mathematical operations and functions are available for these numbers. There is no restriction on the type of the numbers fed into the code; it works for real and complex numbers as well as for any Python type that implements the necessary operations.

If only first-order derivatives are required, the module `FirstDerivatives` should be used. It is compatible to this one, but significantly faster.

Example:

```
print sin(DerivVar(2))
```

produces the output

```
(0.909297426826, [-0.416146836547])
```

The first number is the value of $\sin(2)$; the number in the following list is the value of the derivative of $\sin(x)$ at $x=2$, i.e. $\cos(2)$.

When there is more than one variable, `DerivVar` must be called with an integer second argument that specifies the number of the variable.

Example:

```
x = DerivVar(7., 0)
y = DerivVar(42., 1)
z = DerivVar(pi, 2)
print (sqrt(pow(x,2)+pow(y,2)+pow(z,2)))
```

produces the output

```
(42.6950770511, [0.163953328662, 0.98371997197, 0.0735820818365])
```

The numbers in the list are the partial derivatives with respect to x , y , and z , respectively.

Higher-order derivatives are requested with an optional third argument to `DerivVar`.

Example:

```
x = DerivVar(3., 0, 3)
y = DerivVar(5., 1, 3)
print sqrt(x*y)
```

produces the output

```
(3.87298334621,
 [0.645497224368, 0.387298334621],
 [[-0.107582870728, 0.0645497224368],
 [0.0645497224368, -0.0387298334621]],
 [[[0.053791435364, -0.0107582870728],
 [-0.0107582870728, -0.00645497224368]],
 [[-0.0107582870728, -0.00645497224368],
 [-0.00645497224368, 0.0116189500386]]])
```

The individual orders can be extracted by indexing:

```
print sqrt(x*y)[0]
3.87298334621
print sqrt(x*y)[1]
[0.645497224368, 0.387298334621]
```

An n-th order derivative is represented by a nested list of depth n. When variables with different differentiation orders are mixed, the result has the lower one of the two orders. An exception are zeroth-order variables, which are treated as constants.

Caution: Higher-order derivatives are implemented by recursively using DerivVars to represent derivatives. This makes the code very slow for high orders.

Note: It doesn't make sense to use multiple DerivVar objects with different values for the same variable index in one calculation, but there is no check for this. I.e.

```
print DerivVar(3, 0)+DerivVar(5, 0)
```

produces

```
(8, [2])
```

but this result is meaningless.

Class DerivVar: Variable with derivatives

Constructor: `DerivVar(value, index = 0, order = 1)`

`value` the numerical value of the variable

`index` the variable index (an integer), which serves to distinguish between variables and as an index for the derivative lists. Each explicitly created instance of `DerivVar` must have a unique index.

`order` the derivative order

Indexing with an integer yields the derivatives of the corresponding order.

Methods:

- `toOrder(order)`
Returns a `DerivVar` object with a lower derivative order.

Functions

- `isDerivVar(x)`
Returns 1 if `x` is a `DerivVar` object.
- `DerivVector(x, y, z, index=0, order=1)`
Returns a vector whose components are `DerivVar` objects.

`x, y, z` vector components (numbers)

`index` the `DerivVar` index for the `x` component. The `y` and `z` components receive consecutive indices.

`order` the derivative order

Module Scientific.Functions.FindRoot

Functions

- `newtonRaphson(function, lox, hix, xacc)`
Finds the root of `function` which is bracketed by values `lox` and `hix` to an accuracy of $\pm xacc$. The algorithm used is a safe version of Newton-Raphson (see page 366 of NR in C, 2ed). `function` must be a

function of one variable, and may only use operations defined for the DerivVar objects in the module FirstDerivatives.

Example:

```
from Scientific.Functions.FindRoot import newtonRaphson
from math import pi
def func(x):
    return (2*x*cos(x) - sin(x))*cos(x) - x + pi/4.0
newtonRaphson(func, 0.0, 1.0, 1.0e-12)
```

yields 0.952847864655.

Module Scientific.Functions.FirstDerivatives

This module provides automatic differentiation for functions with any number of variables. Instances of the class DerivVar represent the values of a function and its partial derivatives with respect to a list of variables. All common mathematical operations and functions are available for these numbers. There is no restriction on the type of the numbers fed into the code; it works for real and complex numbers as well as for any Python type that implements the necessary operations.

This module is as far as possible compatible with the n-th order derivatives module Derivatives. If only first-order derivatives are required, this module is faster than the general one.

Example:

```
print sin(DerivVar(2))
```

produces the output

```
(0.909297426826, [-0.416146836547])
```

The first number is the value of $\sin(2)$; the number in the following list is the value of the derivative of $\sin(x)$ at $x=2$, i.e. $\cos(2)$.

When there is more than one variable, DerivVar must be called with an integer second argument that specifies the number of the variable.

Example:

```
x = DerivVar(7., 0)
y = DerivVar(42., 1)
z = DerivVar(pi, 2)
print (sqrt(pow(x,2)+pow(y,2)+pow(z,2)))
```

produces the output

```
(42.6950770511, [0.163953328662, 0.98371997197, 0.0735820818365])
```

The numbers in the list are the partial derivatives with respect to x, y, and z, respectively.

Note: It doesn't make sense to use DerivVar with different values for the same variable index in one calculation, but there is no check for this. I.e.

```
print DerivVar(3, 0)+DerivVar(5, 0)
```

produces

```
(8, [2])
```

but this result is meaningless.

Class DerivVar: Variable with derivatives

Constructor: DerivVar(value, index = 0)

value the numerical value of the variable

index the variable index (an integer), which serves to distinguish between variables and as an index for the derivative lists. Each explicitly created instance of DerivVar must have a unique index.

Indexing with an integer yields the derivatives of the corresponding order.

Functions

- `isDerivVar(x)`
Returns 1 if `x` is a `DerivVar` object.
- `DerivVector(x, y, z, index=0)`
Returns a vector whose components are `DerivVar` objects.

`x, y, z` vector components (numbers)

`index` the `DerivVar` index for the `x` component. The `y` and `z` components receive consecutive indices.

Module Scientific.Functions.Interpolation

Class `InterpolatingFunction`: Function defined by values on a grid using interpolation

An interpolating function of `n` variables with `m`-dimensional values is defined by an $(n+m)$ -dimensional array of values and `n` one-dimensional arrays that define the variables values corresponding to the grid points. The grid does not have to be equidistant.

Constructor: `InterpolatingFunction(axes, values, default=None)`

`axes` a sequence of one-dimensional arrays, one for each variable, specifying the values of the variables at the grid points

`values` an array containing the function values on the grid

`default` the value of the function outside the grid. A value of `None` means that the function is undefined outside the grid and that any attempt to evaluate it there yields an exception.

Evaluation: `function(x1, x2, ...)` yields the function value obtained by linear interpolation.

Indexing: all array indexing operations except for the `NexAxis` operator are supported.

Methods:

- `selectInterval(first, last, variable=0)`
Returns a new `InterpolatingFunction` whose grid is restricted to the interval from `first` to `last` along the variable whose number is `variable`.
- `derivative(variable=0)`
Returns a new `InterpolatingFunction` describing the derivative with respect to `variable` (an integer).
- `integral(variable=0)`
Returns a new `InterpolatingFunction` describing the integral with respect to `variable` (an integer). The integration constant is defined in such a way that the value of the integral at the first grid point along `variable` is zero.
- `definiteIntegral(variable=0)`
Returns a new `InterpolatingFunction` describing the definite integral with respect to `variable` (an integer). The integration constant is defined in such a way that the value of the integral at the first grid point along `variable` is zero. In the case of a function of one variable, the definite integral is a number.
- `fitPolynomial(order)`
Returns a polynomial of `order` with parameters obtained from a least-squares fit to the grid values.

Class `NetCDFInterpolatingFunction`: Function defined by values on a grid in a netCDF file

A subclass of `InterpolatingFunction`.

Constructor: `NetCDFInterpolatingFunction(filename, axesnames, variablename, default=None)`

`filename` the name of the netCDF file

`axesnames` the names of the netCDF variables that contain the axes information

`variablename` the name of the netCDF variable that contains the data values

default the value of the function outside the grid. A value of `None` means that the function is undefined outside the grid and that any attempt to evaluate it there yields an exception.

Evaluation: `function(x1, x2, ...)` yields the function value obtained by linear interpolation.

Module Scientific.Functions.LeastSquares

Functions

- `leastSquaresFit(model, parameters, data)`

General non-linear least-squares fit using the Levenberg-Marquardt algorithm and automatic derivatives.

The parameter `model` specifies the function to be fitted. It will be called with two parameters: the first is a tuple containing all fit parameters, and the second is the first element of a data point (see below). The return value must be a number. Since automatic differentiation is used to obtain the derivatives with respect to the parameters, the function may only use the mathematical functions known to the module `FirstDerivatives`.

The parameter `parameter` is a tuple of initial values for the fit parameters.

The parameter `data` is a list of data points to which the model is to be fitted. Each data point is a tuple of length two or three. Its first element specifies the independent variables of the model. It is passed to the model function as its first parameter, but not used in any other way. The second element of each data point tuple is the number that the return value of the model function is supposed to match as well as possible. The third element (which defaults to 1.) is the statistical variance of the data point, i.e. the inverse of its statistical weight in the fitting procedure.

The function returns a list containing the optimal parameter values and the chi-squared value describing the quality of the fit.

- `polynomialLeastSquaresFit(parameters, data)`

Least-squares fit to a polynomial whose order is defined by the number of parameter values.

Module Scientific.Functions.Polynomial

Class Polynomial: Multivariate polynomial

Instances of this class represent polynomials of any order and in any number of variables. They can be evaluated like functions.

Constructor: `Polynomial(coefficients)`, where `coefficients` is an array whose dimension defines the number of variables and whose length along each axis defines the order in the corresponding variable.

Methods:

- `derivative(variable=0)`
Returns the derivative with respect to `variable`.
- `integral(variable=0)`
Returns the indefinite integral with respect to `variable`.

Module Scientific.Functions.Romberg

Functions

- `trapezoid(function, interval, numtraps)`
Returns the integral of `function` (a function of one variable) over `interval` (a sequence of length two containing the lower and upper limit of the integration interval), calculated using the trapezoidal rule using `numtraps` trapezoids.

Example:

```
from Scientific.Functions.Romberg import romberg
from Numeric import pi
romberg(tan, (0.0, pi/3.0))
```

yields 0.693147180562

- `romberg(function, interval, accuracy=1e-07, show=0)`
Returns the integral of `function` (a function of one variable) over `interval` (a sequence of length two containing the lower and upper

limit of the integration interval), calculated using Romberg integration up to the specified **accuracy**. If **show** is 1, the triangular array of the intermediate results will be printed.

Module Scientific.Geometry

This subpackage contains classes that deal with geometrical quantities and objects. The geometrical quantities are vectors and tensors, transformations, and quaternions as descriptions of rotations. There are also tensor fields, which were included here (rather than in the subpackage Scientific.Functions) because they are most often used in a geometric context. Finally, there are classes for elementary geometrical objects such as spheres and planes.

Class Tensor: Tensor in 3D space

Constructor: `Tensor([[xx, xy, xz], [yx, yy, yz], [zx, zy, zz]])`

Tensors support the usual arithmetic operations (`t1`, `t2`: tensors, `v`: vector, `s`: scalar):

- `t1+t2` (addition)
- `t1-t2` (subtraction)
- `t1*t2` (tensorial (outer) product)
- `t1*v` (contraction with a vector, same as `t1.dot(v.asTensor())`)
- `s*t1`, `t1*s` (multiplication with a scalar)
- `t1/s` (division by a scalar)

The coordinates can be extracted by indexing; a tensor of rank N can be indexed like an array of dimension N.

Tensors are *immutable*, i.e. their elements cannot be changed.

Tensor elements can be any objects on which the standard arithmetic operations are defined. However, eigenvalue calculation is supported only for float elements.

Methods:

- `asVector()`
Returns an equivalent vector object (only for rank 1).

- `dot(other)`
Returns the contraction with `other`.
- `trace(axis1=0, axis2=1)`
Returns the trace of a rank-2 tensor.
- `transpose()`
Returns the transposed (index reversed) tensor.
- `symmetricalPart()`
Returns the symmetrical part of a rank-2 tensor.
- `asymmetricalPart()`
Returns the asymmetrical part of a rank-2 tensor.
- `eigenvalues()`
Returns the eigenvalues of a rank-2 tensor in an array.
- `diagonalization()`
Returns the eigenvalues of a rank-2 tensor and a tensor representing the rotation matrix to the diagonalized form.
- `inverse()`
Returns the inverse of a rank-2 tensor.

Class Vector: Vector in 3D space

Constructor:

- `Vector(x, y, z)` (from three coordinates)
- `Vector(coordinates)` (from any sequence containing three coordinates)

Vectors support the usual arithmetic operations (`v1`, `v2`: vectors, `s`: scalar):

- `v1+v2` (addition)
- `v1-v2` (subtraction)
- `v1*v2` (scalar product)
- `s*v1`, `v1*s` (multiplication with a scalar)

- $v1/s$ (division by a scalar)

The three coordinates can be extracted by indexing.

Vectors are *immutable*, i.e. their elements cannot be changed.

Vector elements can be any objects on which the standard arithmetic operations plus the functions `sqrt` and `arccos` are defined.

Methods:

- `x()`
Returns the x coordinate.
- `y()`
Returns the y coordinate.
- `z()`
Returns the z coordinate.
- `length()`
Returns the length (norm).
- `normal()`
Returns a normalized copy.
- `cross(other)`
Returns the cross product with vector `other`.
- `asTensor()`
Returns an equivalent tensor object of rank 1.
- `dyadicProduct(other)`
Returns the dyadic product with vector or tensor `other`.
- `angle(other)`
Returns the angle to vector `other`.

Module Scientific.Geometry.Objects3D

Class GeometricalObject3D: Geometrical object in 3D space

This is an abstract base class; to create instances, use one of the subclasses.

Methods:

- **intersectWith(other)**
Returns the geometrical object that results from the intersection with **other**. If there is no intersection, the result is **None**.
Note that intersection is not implemented for all possible pairs of objects. A **ValueError** is raised for combinations that haven't been implemented yet.
- **hasPoint(point)**
Returns 1 if **point** is in the object.
- **distanceFrom(point)**
Returns the distance of **point** from the closest point of the object.
- **volume()**
Returns the volume. The result is **None** for unbounded objects and zero for lower-dimensional objects.

Class Sphere: Sphere

A subclass of **GeometricalObject3D**.

Constructor: **Sphere(center, radius)**, where **center** is a vector and **radius** a float.

Class Plane: Plane

A subclass of **GeometricalObject3D**.

Constructor:

- **Plane(point, normal)**, where **point** (a vector) is an arbitrary point in the plane and **normal** (a vector) indicated the direction normal to the plane.
- **Plane(p1, p2, p3)**, where each argument is a vector and describes a point in the plane. The three points may not be colinear.

Methods:

- **projectionOf(point)**
Returns the projection of **point** onto the plane.
- **rotate(axis, angle)**
Returns a copy of the plane rotated around the coordinate origin.

Class Cone: Cone

A subclass of GeometricalObject3D.

Constructor: Cone(**tip**, **axis**, **angle**), where **tip** is a vector indicating the location of the tip, **axis** is a vector that describes the direction of the line of symmetry, and **angle** is the angle between the line of symmetry and the cone surface.

Class Circle: Circle

A subclass of GeometricalObject3D.

Constructor: Circle(**center**, **normal**, **radius**), where **center** is a vector indicating the center of the circle, **normal** is a vector describing the direction normal to the plane of the circle, and **radius** is a float.

Class Line: Line

A subclass of GeometricalObject3D.

Constructor: Line(**point**, **direction**), where **point** is a vector indicating any point on the line and **direction** is a vector describing the direction of the line.

Methods:

- **projectionOf(point)**
Returns the projection of **point** onto the line.

Class RhombicLattice: Lattice with rhombic elementary cell

A lattice object contains values defined on a finite periodic structure that is created by replicating a given elementary cell along the three lattice vectors. The elementary cell can contain any number of points.

Constructor: RhombicLattice(**elementary_cell**, **lattice_vectors**, **cells**, **function=None**, **base=None**)

elementary_cell a list of the points (vectors) in the elementary cell

lattice_vectors a tuple of three vectors describing the edges of the elementary cell

cells a tuple of three integers, indicating how often the elementary cell should be replicated along each lattice vector

function the function to be applied to each point in the lattice in order to obtain the value stored in the lattice. If no function is specified, the point itself becomes the value stored in the lattice.

base an offset added to all lattice points

Class BravaisLattice: General Bravais lattice

This is a subclass of RhombicLattice, describing the special case of an elementary cell containing one point.

Constructor: BravaisLattice(lattice_vectors, cells, function=None, base=None)

lattice_vectors a tuple of three vectors describing the edges of the elementary cell

cells a tuple of three integers, indicating how often the elementary cell should be replicated along each lattice vector

function the function to be applied to each point in the lattice in order to obtain the value stored in the lattice. If no function is specified, the point itself becomes the value stored in the lattice.

base an offset added to all lattice points

Class SCLattice: Simple cubic lattice

This is a subclass of BravaisLattice, describing the special case of a cubic elementary cell.

Constructor: SCLattice(cellsize, cells, function=None, base=None)

cellsize the edge length of the cubic elementary cell

cells a tuple of three integers, indicating how often the elementary cell should be replicated along each lattice vector

function the function to be applied to each point in the lattice in order to obtain the value stored in the lattice. If no function is specified, the point itself becomes the value stored in the lattice.

base an offset added to all lattice points

Module Scientific.Geometry.Quaternion

Class Quaternion: Quaternion (hypercomplex number)

This implementation of quaternions is not complete; only the features needed for representing rotation matrices by quaternions are implemented.

Constructor:

- Quaternion(q0, q1, q2, q3) (from four real components)
- Quaternion(q) (from a sequence containing the four components)

Quaternions support addition, subtraction, and multiplication, as well as multiplication and division by scalars. Division by quaternions is not provided, because quaternion multiplication is not associative. Use multiplication by the inverse instead.

The four components can be extracted by indexing.

Methods:

- norm()
Returns the norm.
- normalized()
Returns the quaternion scaled to norm 1.
- inverse()
Returns the inverse.
- asMatrix()
Returns a 4x4 matrix representation.
- asRotation()
Returns the corresponding rotation matrix (the quaternion must be normalized).

Functions

- isQuaternion(x)
Returns 1 if x is a quaternion.

Module Scientific.Geometry.TensorAnalysis

Class TensorField: Tensor field of arbitrary rank

A tensor field is described by a tensor at each point of a three-dimensional rectangular grid. The grid spacing may be non-uniform. Tensor fields are implemented as a subclass of InterpolatingFunction from the module Scientific.Functions.Interpolation and thus share all methods defined in that class.

Constructor: `TensorField(rank, axes, values, default=None)`

rank a non-negative integer indicating the tensor rank

axes a sequence of three one-dimensional arrays, each of which specifies one coordinate (x, y, z) of the grid points

values an array of **rank+3** dimensions. Its first three dimensions correspond to the x, y, z directions and must have lengths compatible with the axis arrays. The remaining dimensions must have length 3.

default the value of the field for points outside the grid. A value of `None` means that an exception will be raised for an attempt to evaluate the field outside the grid. Any other value must be a tensor of the correct rank.

Evaluation:

- `tensorfield(x, y, z)` (three coordinates)
- `tensorfield(coordinates)` (any sequence containing three coordinates)

Methods:

- `zero()`
Returns a tensor of the correct rank with zero elements.
- `derivative(variable)`
Returns the derivative with respect to `variable`, which must be one of 0, 1, or 2.

- `allDerivatives()`
Returns all three derivatives (x, y, z).

Class ScalarField: Scalar field (tensor field of rank 0)

Constructor: `ScalarField(axes, values, default=None)`
A subclass of `TensorField`.

Methods:

- `gradient()`
Returns the gradient (a vector field).
- `laplacian()`
Returns the laplacian (a scalar field).

Class VectorField: Vector field (tensor field of rank 1)

Constructor: `VectorField(axes, values, default=None)`
A subclass of `TensorField`.

Methods:

- `divergence()`
Returns the divergence (a scalar field).
- `curl()`
Returns the curl (a vector field).
- `strain()`
Returns the strain (a tensor field of rank 2).
- `divergenceCurlAndStrain()`
Returns all derivative fields: divergence, curl, and strain.
- `laplacian()`
Returns the laplacian (a vector field).
- `length()`
Returns a scalar field corresponding to the length (norm) of the vector field.

Module Scientific.Geometry.Transformation

Class Transformation: Linear coordinate transformation.

Transformation objects represent linear coordinate transformations in a 3D space. They can be applied to vectors, returning another vector. If \mathbf{t} is a transformation and \mathbf{v} is a vector, $\mathbf{t}(\mathbf{v})$ returns the transformed vector. Transformations support composition: if $\mathbf{t1}$ and $\mathbf{t2}$ are transformation objects, $\mathbf{t1}*\mathbf{t2}$ is another transformation object which corresponds to applying $\mathbf{t1}$ *after* $\mathbf{t2}$.

This class is an abstract base class. Instances can only be created of concrete subclasses, i.e. translations or rotations.

Methods:

- `rotation()`
Returns the rotational component.
- `translation()`
Returns the translational component. In the case of a mixed rotation/translation, this translation is executed *after* the rotation.
- `inverse()`
Returns the inverse transformation.
- `screwMotion()`
Returns the four parameters (`reference`, `direction`, `angle`, `distance`) of a screw-like motion that is equivalent to the transformation. The screw motion consists of a displacement of `distance` (a float) along `direction` (a normalized vector) plus a rotation of `angle` radians around an axis pointing along `direction` and passing through the point `reference` (a vector).

Class Translation: Translational transformation.

This is a subclass of Transformation.

Constructor: `Translation(vector)`, where `vector` is the displacement vector.

Methods:

- `displacement()`
Returns the displacement vector.

Class `Rotation`: Rotational transformation.

This is a subclass of `Transformation`.

Constructor:

- `Rotation(tensor)`, where `tensor` is a tensor object containing the rotation matrix.
- `Rotation(axis, angle)`, where `axis` is a vector and `angle` a number (the angle in radians).

Methods:

- `axisAndAngle()`
Returns the axis (a normalized vector) and angle (a float, in radians).

Class `RotationTranslation`: Combined translational and rotational transformation.

This is a subclass of `Transformation`.

Objects of this class are not created directly, but can be the result of a composition of rotations and translations.

Module Scientific.IO

Module Scientific.IO.ArrayIO

This module contains elementary support for I/O of one- and two-dimensional numerical arrays to and from plain text files. The text file format is very simple and used by many other programs as well:

- each line corresponds to one row of the array
- the numbers within a line are separated by white space
- lines starting with `#` are ignored (comment lines)

An array containing only one line or one column is returned as a one-dimensional array on reading. One-dimensional arrays are written as one item per line.

Numbers in files to be read must conform to Python/C syntax. For reading files containing Fortran-style double-precision numbers (exponent prefixed by `D`), use the module `Scientific.IO.FortranFormat`.

Functions

- `readArray(filename)`
Return an array containing the data from file `filename`.
- `writeArray(array, filename)`
Write array `a` to file `filename`.
- `writeDataSets(datasets, filename, separator='')`
Write each of the items in the sequence `datasets` to the file `filename`, separating the datasets by a line containing `separator`. The items in the data sets can be one- or two-dimensional arrays or equivalent nested sequences. The output file format is understood by many plot programs.

Module Scientific.IO.FortranFormat

Fortran-compatible input/output

This module provides two classes that aid in reading and writing Fortran-formatted text files.

Examples:

Input:

```
s = ' 59999'
format = FortranFormat('2I4')
line = FortranLine(s, format)
print line[0]
print line[1]
```

prints

```
5
9999
```

Output:

```
format = FortranFormat('2D15.5')
line = FortranLine([3.1415926, 2.71828], format)
print str(line)
```

prints

```
3.14159D+00 2.71828D+00
```

Class FortranLine: Fortran-style record in formatted files

FortranLine objects represent the content of one record of a Fortran-style formatted file. Indexing yields the contents as Python objects, whereas transformation to a string (using the built-in function `str`) yields the text representation.

Constructor: `FortranLine(data, format, length=80)`

data either a sequence of Python objects, or a string formatted according to Fortran rules

format either a Fortran-style format string, or a FortranFormat object. A FortranFormat should be used when the same format string is used repeatedly, because then the rather slow parsing of the string is performed only once.

length the length of the Fortran record. This is relevant only when **data** is a string; this string is then extended by spaces to have the indicated length. The default value of 80 is almost always correct.

Restrictions:

- 1) Only A, D, E, F, G, I, and X formats are supported (plus string constants for output).
- 2) No direct support for complex numbers; they must be split into real and imaginary parts before output.
- 3) No overflow check. If an output field gets too large, it will take more space, instead of being replaced by stars according to Fortran conventions.

Class FortranFormat: Parsed fortran-style format string

Constructor: `FortranFormat(format)`, where **format** is a format specification according to Fortran rules.

Module Scientific.IO.NetCDF

Class NetCDFFile: netCDF file

Constructor: `NetCDFFile(filename, mode="r")`

filename name of the netCDF file. By convention, netCDF files have the extension ".nc", but this is not enforced. The filename may contain a home directory indication starting with "~".

mode access mode. "r" means read-only; no data can be modified. "w" means write; a new file is created, an existing file with the same name is deleted. "a" means append (in analogy with serial files); an existing file is opened for reading and writing, and if the file does not exist it is created. "r+" is similar to "a", but the file must already exist. An "s" can be appended to any of the modes listed above; it indicates that the file will be opened or created in "share" mode, which reduces buffering in order to permit simultaneous read access by other processes to a file that is being written.

A NetCDFFile object has two standard attributes: `dimensions` and `variables`. The values of both are dictionaries, mapping dimension names to their associated lengths and variable names to variables, respectively. Application programs should never modify these dictionaries. All other attributes correspond to global attributes defined in the netCDF file. Global file attributes are created by assigning to an attribute of the NetCDFFile object.

Methods:

- `close()`
Closes the file. Any read or write access to the file or one of its variables after closing raises an exception.
- `createDimension(name, length)`
Creates a new dimension with the given `name` and `length`. `length` must be a positive integer or `None`, which stands for the unlimited dimension. Note that there can be only one unlimited dimension in a file.
- `createVariable(name, type, dimensions)`
Creates a new variable with the given `name`, `type`, and `dimensions`. The `type` is a one-letter string with the same meaning as the typecodes for arrays in module Numeric; in practice the predefined type constants from Numeric should be used. `dimensions` must be a tuple containing dimension names (strings) that have been defined previously.

The return value is the NetCDFVariable object describing the new variable.
- `sync()`
Writes all buffered data to the disk file.

Class NetCDFVariable: Variable in a netCDF file

NetCDFVariable objects are constructed by calling the method `createVariable` on the NetCDFFile object.

NetCDFVariable objects behave much like array objects defined in module Numeric, except that their data resides in a file. Data is read by indexing and written by assigning to an indexed subset; the entire array can be accessed by the index `[:]` or using the methods `getValue` and

assignValue. NetCDFVariable objects also have attribute "shape" with the same meaning as for arrays, but the shape cannot be modified. There is another read-only attribute "dimensions", whose value is the tuple of dimension names.

All other attributes correspond to variable attributes defined in the netCDF file. Variable attributes are created by assigning to an attribute of the NetCDFVariable object.

Note: If a file open for reading is simultaneously written by another program, the size of the unlimited dimension may change. Every time the shape of a variable is requested, the current size will be obtained from the file. For reading and writing, the size obtained during the last shape request is used. This ensures consistency: `foo[-1]` means the same thing no matter how often it is evaluated, as long as the shape is not re-evaluated in between.

Methods:

- **assignValue(value)**
Assigns `value` to the variable. This method allows assignment to scalar variables, which cannot be indexed.
- **getValue()**
Returns the value of the variable. This method allows access to scalar variables, which cannot be indexed.
- **typecode()**
Return the variable's type code (a string).

Module Scientific.IO.PDB

This module provides classes that represent PDB (Protein Data Bank) files and configurations contained in PDB files. It provides access to PDB files on two levels: low-level (line by line) and high-level (chains, residues, and atoms).

Caution: The PDB file format has been heavily abused, and it is probably impossible to write code that can deal with all variants correctly. This module tries to read the widest possible range of PDB files, but gives priority to a correct interpretation of the PDB format as defined by the Brookhaven National Laboratory.

A special problem are atom names. The PDB file format specifies that the first two letters contain the right-justified chemical element name. A later modification allowed the initial space in hydrogen names to be replaced by a digit. Many programs ignore all this and treat the name as an arbitrary left-justified four-character name. This makes it difficult to extract the chemical element accurately; most programs write the "CA" for C_alpha in such a way that it actually stands for a calcium atom! For this reason a special element field has been added later, but only few files use it. The low-level routines in this module do not try to deal with the atom name problem; they return and expect four-character atom names including spaces in the correct positions. The high-level routines use atom names without leading or trailing spaces, but provide and use the element field whenever possible. For output, they use the element field to place the atom name correctly, and for input, they construct the element field content from the atom name if no explicit element field is found in the file. Except where indicated, numerical values use the same units and conventions as specified in the PDB format description.

Example:

```
conf = Structure('example.pdb')
print conf
for residue in conf.residues:
    for atom in residue:
        print atom
```

Class HetAtom: HetAtom in a PDB structure

A subclass of Atom, which differs only in the return value of the method `type()`.

Constructor: `HetAtom(name, position, **properties)`.

Class Group: Atom group (residue or molecule) in a PDB file

This is an abstract base class. Instances can be created using one of the subclasses (Molecule, AminoAcidResidue, NucleotideResidue).

Group objects permit iteration over atoms with for-loops, as well as extraction of atoms by indexing with the atom name.

Methods:

- `addAtom(atom)`
Adds `atom` (an `Atom` object) to the group.
- `deleteAtom(atom)`
Removes `atom` (an `Atom` object) from the group. An exception will be raised if `atom` is not part of the group.
- `deleteHydrogens()`
Removes all hydrogen atoms.
- `changeName(name)`
Sets the PDB residue name to `name`.
- `writeToFile(file)`
Writes the group to `file` (a `PDBFile` object or a string containing a file name).

Class Chain: Chain of PDB residues

This is an abstract base class. Instances can be created using one of the subclasses (`PeptideChain`, `NucleotideChain`).

Chain objects respond to `len()` and return their residues by indexing with integers.

Methods:

- `sequence()`
Returns the list of residue names.
- `addResidue(residue)`
Add `residue` at the end of the chain.
- `removeResidues(first, last)`
Remove residues starting from `first` up to (but not including) `last`. If `last` is `None`, remove everything starting from `first`.
- `deleteHydrogens()`
Removes all hydrogen atoms.

- `writeToFile(file)`
Writes the chain to file (a PDBFile object or a string containing a file name).

Class Molecule: Molecule in a PDB file

A subclass of Group.

Constructor: `Molecule(name, atoms=None, number=None)`, where `name` is the PDB residue name. An optional list of `atoms` can be specified, otherwise the molecule is initially empty. The optional `number` is the PDB residue number.

Note: In PDB files, non-chain molecules are treated as residues, there is no separate molecule definition. This module defines every residue as a molecule that is not an amino acid residue or a nucleotide residue.

Class PDBFile: PDB file with access at the record level

Constructor: `PDBFile(filename, mode="r")`, where `filename` is the file name and `mode` is "r" for reading and "w" for writing, The low-level file access is handled by the module `Scientific.IO.TextFile`, therefore compressed files and URLs (for reading) can be used as well.

Methods:

- `readLine()`
Returns the contents of the next non-blank line (= record). The return value is a tuple whose first element (a string) contains the record type. For supported record types (HEADER, ATOM, HETATM, ANISOU, TERM, MODEL, CONECT), the items from the remaining fields are put into a dictionary which is returned as the second tuple element. Most dictionary elements are strings or numbers; atom positions are returned as a vector, and anisotropic temperature factors are returned as a rank-2 tensor, already multiplied by 1.e-4. White space is stripped from all strings except for atom names, whose correct interpretation can depend on an initial space. For unsupported record types, the second tuple element is a string containing the remaining part of the record.
- `writeLine(type, data)`

Writes a line using record type and data dictionary in the same format as returned by `readLine()`. Default values are provided for non-essential information, so the data dictionary need not contain all entries.

- `writeComment(text)`
Writes `text` into one or several comment lines. Each line of the text is prefixed with `REMARK` and written to the file.
- `writeAtom(name, position, occupancy=0.0, temperature_factor=0.0, element='')`
Writes an `ATOM` or `HETATM` record using the `name`, `occupancy`, `temperature` and `element` information supplied. The residue and chain information is taken from the last calls to the methods `nextResidue()` and `nextChain()`.
- `nextResidue(name, number=None, terminus=None)`
Signals the beginning of a new residue, starting with the next call to `writeAtom()`. The residue name is `name`, and a `number` can be supplied optionally; by default residues in a chain will be numbered sequentially starting from 1. The value of `terminus` can be `None`, `"C"`, or `"N"`; it is passed to export filters that can use this information in order to use different atom or residue names in terminal residues.
- `nextChain(chain_id=None, segment_id='')`
Signals the beginning of a new chain. A chain identifier (string of length one) can be supplied as `chain_id`, by default consecutive letters from the alphabet are used. The equally optional `segment_id` defaults to an empty string.
- `terminateChain()`
Signals the end of a chain.
- `close()`
Closes the file. This method *must* be called for write mode because otherwise the file will be incomplete.

Class Atom: Atom in a PDB structure

Constructor: `Atom(name, position, **properties)`, where `name` is the PDB atom name (a string), `position` is a atom position (a vector), and `properties`

can include any of the other items that can be stored in an atom record. The properties can be obtained or modified using indexing, as for Python dictionaries.

Methods:

- `type()`
Returns the six-letter record type, ATOM or HETATM.
- `writeToFile(file)`
Writes an atom record to file (a PDBFile object or a string containing a file name).

Class AminoAcidResidue: Amino acid residue in a PDB file

A subclass of Group.

Constructor: `AminoAcidResidue(name, atoms=None, number=None)`, where `name` is the PDB residue name. An optional list of `atoms` can be specified, otherwise the residue is initially empty. The optional `number` is the PDB residue number.

Methods:

- `isCTerminus()`
Returns 1 if the residue is in C-terminal configuration, i.e. if it has a second oxygen bound to the carbon atom of the peptide group.
- `isNTerminus()`
Returns 1 if the residue is in N-terminal configuration, i.e. if it contains more than one hydrogen bound to be nitrogen atom of the peptide group.

Class NucleotideResidue: Nucleotide residue in a PDB file

A subclass of Group.

Constructor: `NucleotideResidue(name, atoms=None, number=None)`, where `name` is the PDB residue name. An optional list of `atoms` can be specified, otherwise the residue is initially empty. The optional `number` is the PDB residue number.

Methods:

- `hasRibose()`
Returns 1 if the residue has an atom named O2*.
- `hasDesoxyribose()`
Returns 1 if the residue has no atom named O2*.
- `hasPhosphate()`
Returns 1 if the residue has a phosphate group.
- `hasTerminalH()`
Returns 1 if the residue has a 3-terminal H atom.

Class PeptideChain: Peptide chain in a PDB file

A subclass of Chain.

Constructor: `PeptideChain(residues=None, chain_id=None, segment_id=None)`, where `chain_id` is a one-letter chain identifier and `segment_id` is a multi-character chain identifier, both are optional. A list of `AminoAcidResidue` objects can be passed as `residues`; by default a peptide chain is initially empty.

Methods:

- `isTerminated()`
Returns 1 if the last residue is in C-terminal configuration.

Class NucleotideChain: Nucleotide chain in a PDB file

A subclass of Chain.

Constructor: `NucleotideChain(residues=None, chain_id=None, segment_id=None)`, where `chain_id` is a one-letter chain identifier and `segment_id` is a multi-character chain identifier, both are optional. A list of `NucleotideResidue` objects can be passed as `residues`; by default a nucleotide chain is initially empty.

Class ResidueNumber: PDB residue number

Most PDB residue numbers are simple integers, but when insertion codes are used a number can consist of an integer plus a letter. Such compound residue numbers are represented by this class.

Constructor: `ResidueNumber(number, insertion_code)`

Class Structure: A high-level representation of the contents of a PDB file

Constructor: `Structure(filename, model=0, alternate_code="A")`, where `filename` is the name of the PDB file. Compressed files and URLs are accepted, as for class `PDBFile`. The two optional arguments specify which data should be read in case of a multiple-model file or in case of a file that contains alternative positions for some atoms.

The components of a system can be accessed in several ways (`s` is an instance of this class):

- `s.residues` is a list of all PDB residues, in the order in which they occurred in the file.
- `s.peptide_chains` is a list of `PeptideChain` objects, containing all peptide chains in the file in their original order.
- `s.nucleotide_chains` is a list of `NucleotideChain` objects, containing all nucleotide chains in the file in their original order.
- `s.molecules` is a list of all PDB residues that are neither amino acid residues nor nucleotide residues, in their original order.
- `s.objects` is a list of all high-level objects (peptide chains, nucleotide chains, and molecules) in their original order.

An iteration over a `Structure` instance by a for-loop is equivalent to an iteration over the residue list.

Methods:

- `deleteHydrogens()`
Removes all hydrogen atoms.
- `splitPeptideChain(number, position)`
Splits the peptide chain indicated by `number` (0 being the first peptide chain in the PDB file) after the residue indicated by `position` (0 being the first residue of the chain). The two chain fragments remain adjacent in the peptide chain list, i.e. the numbers of all following nucleotide chains increase by one.

- `splitNucleotideChain(number, position)`
Splits the nucleotide chain indicated by `number` (0 being the first nucleotide chain in the PDB file) after the residue indicated by `position` (0 being the first residue of the chain). The two chain fragments remain adjacent in the nucleotide chain list, i.e. the numbers of all following nucleotide chains increase by one.
- `joinPeptideChains(first, second)`
Join the two peptide chains indicated by `first` and `second` into one peptide chain. The new chain occupies the position `first`; the chain at `second` is removed from the peptide chain list.
- `joinNucleotideChains(first, second)`
Join the two nucleotide chains indicated by `first` and `second` into one nucleotide chain. The new chain occupies the position `first`; the chain at `second` is removed from the nucleotide chain list.
- `renumberAtoms()`
Renumber all atoms sequentially starting with 1.
- `writeToFile(file)`
Writes all objects to `file` (a PDBFile object or a string containing a file name).

Module **Scientific.IO.TextFile**

Class TextFile: Text files with line iteration and transparent compression

`TextFile` instances can be used like normal file objects (i.e. by calling `readline()`, `readlines()`, and `write()`), but can also be used as sequences of lines in for-loops.

`TextFile` objects also handle compression transparently. i.e. it is possible to read lines from a compressed text file as if it were not compressed.

Compression is deduced from the file name suffixes

`.Z`(compress/uncompress), `.gz` (gzip/gunzip), and `.bz2` (bzip2).

Finally, `TextFile` objects accept file names that start with `~` or `user` to indicate a home directory, as well as URLs (for reading only).

Reference for Module Scientific

Constructor: `TextFile(filename, mode="r")`, where `filename` is the name of the file (or a URL) and `mode` is one of `"r"` (read), `"w"` (write) or `"a"` (append, not supported for `.Z` files).

Module Scientific.MPI

This module contains a Python interface to the Message Passing Interface (MPI), and standardized library for message-passing parallel computing. Please read an introduction to MPI before using this module; some terms in the documentation do not make much sense unless you understand the principles of MPI.

This module contains an object, `world`, which represents the default communicator in MPI. This communicator can be used directly for sending and receiving data, or other communicators can be derived from it.

A number of global constants are also defined (`max`, `min`, `prod`, `sum`, `land`, `lor`, `lxor`, `band`, `bor`, `bxor`, `maxloc` and `minloc`). They are used to specify the desired operator in calls to the `reduce` and `allreduce` methods of the communicator objects.

Class MPICommunicator: MPI Communicator

There is no constructor for MPI Communicator objects. The default communicator is given by `Scientific.MPI.world`, and other communicators can only be created by methods on an existing communicator object.

A communicator object has two read-only attributes: `rank` is an integer which indicates the rank of the current process in the communicator, and `size` is an integer equal to the number of processes that participate in the communicator.

Methods:

- `duplicate()`
Returns a new communicator object with the same properties as the original one.
- `send(data, destination, tag)`
Sends the contents of `data` (a string or any contiguous NumPy array except for general object arrays) to the processor whose rank is `destination`, using `tag` as an identifier.
- `nonblockingSend(data, destination, tag)`
Sends the contents of `data` (a string or any contiguous NumPy array except for general object arrays) to the processor whose rank is

destination, using **tag** as an identifier. The send is nonblocking, i.e. the call returns immediately, even if the destination process is not ready to receive.

The return value is an `MPIRequest` object. It is used to wait till the communication has actually happened.

- `receive(data, source=None, tag=None)`
Receives an array from the process with rank **source** with identifier **tag**. The default **source=None** means that messages from any process are accepted. The value of **data** can either be an array object, in which case it must be contiguous and large enough to store the incoming data; it must also have the correct shape. Alternatively, **data** can be a string specifying the data type (in practice, one would use `Numeric.Int`, `Numeric.Float`, etc.). In the latter case, a new array object is created to receive the data.

The return value is a tuple containing four elements: the array containing the data, the source process rank (an integer), the message tag (an integer), and the number of elements that were received (an integer).

- `receiveString(source=None, tag=None)`
Receives a string from the process with rank **source** with identifier **tag**. The default **source=None** means that messages from any process are accepted.

The return value is a tuple containing three elements: the string containing the data, the source process rank (an integer), and the message tag (an integer).

- `nonblockingReceive(data, source=None, tag=None)`
Receives an array from the process with rank **source** with identifier **tag**. The default **source=None** means that messages from any process are accepted. The value of **data** must be a contiguous array object, large enough to store the incoming data; it must also have the correct shape. Unlike the blocking receive, the size of the array must be known when the call is made, as nonblocking receives of unknown quantities of data is not implemented. For the same reason there is no `nonblocking_receiveString`.

The return value is an `MPIRequest` object. It is used to wait until the data has arrived, and will give information about the size, the source and the tag of the incoming message.

- `nonblockingProbe(source=None, tag=None)`
Checks if a message from the process with rank `source` and with identifier `tag` is available for immediate reception. The return value is `None` if no message is available, otherwise a `(source, tag)` tuple is returned.
- `broadcast(array, root)`
Sends data from the process with rank `root` to all processes (including `root`). The parameter `array` can be any contiguous NumPy array except for general object arrays. On the process `root`, it holds the data to be sent. After the call, the data in `array` is the same for all processors. The shape and data type of `array` must be the same in all processes.
- `share(send, receive)`
Distributes data from each process to all other processes in the communicator. The array `send` (any contiguous NumPy array except for general object arrays) contains the data to be sent by each process, the shape and data type must be identical in all processes. The array `receive` must have the same data type as `send` and one additional dimension (the first one), whose length must be the number of processes in the communicator. After the call, the value of `receive[i]` is equal to the contents of the array `send` in process `i`.
- `barrier()`
Waits until all processes in the communicator have called the same method, then all processes continue.
- `abort()`
Aborts all processes associated with the communicator. For emergency use only.
- `reduce(sendbuffer, receivebuffer, operation, root)`
Combine data from all processes using `operation`, and send the data to the process identified by `root`.

`operation` is one of the operation objects defined globally in the module: `max`, `min`, `prod`, `sum`, `land`, `lor`, `lxor`, `band`, `bor`, `bxor`, `maxloc` and `minloc`.

- `allreduce(sendbuffer, receivebuffer, operation, root)`
Combine data from all processes using `operation`, and send the data to all processes in the communicator.

`operation` is one of the operation objects defined globally in the module: `max`, `min`, `prod`, `sum`, `land`, `lor`, `lxor`, `band`, `bor`, `bxor`, `maxloc` and `minloc`.

Class `MPIError`: MPI call failed

Class `MPIRequest`: MPI Request

There is no constructor for MPI Request objects. They are returned by nonblocking send and receives, and are used to query the status of the message.

Methods:

- `wait()`
Waits till the communication has completed. If the operation was a nonblocking send, there is no return value. If the operation was a nonblocking receive, the return value is a tuple containing four elements: the array containing the data, the source process rank (an integer), the message tag (an integer), and the number of elements that were received (an integer).

Module `Scientific.MPI.IO`

Class `LogFile`: File for logging events from all processes

Constructor: `LogFile(filename, communicator=None)`

`filename` the name of the file

`communicator` the communicator in which the file is accesible. The default value of `None` means to use the global world communicator, i.e. all possible processes.

The purpose of LogFile objects is to collect short text output from all processors into a single file. All processes can write whatever they want at any time; the date is simply stored locally. After the file has been closed by all processes, the data is sent to process 0, which then writes everything to one text file, neatly separated by process rank number.

Note that due to the intermediate storage of the data, LogFile objects should not be used for large amounts of data. Also note that all data is lost if a process crashes before closing the file.

Methods:

- `write(string)`
Write `string` to the file.
- `flush()`
Write buffered data to the text file.
- `close()`
Close the file, causing the real text file to be written.

Module Scientific.NumberDict

Class NumberDict: Dictionary storing numerical values

Constructor: `NumberDict()`

An instance of this class acts like an array of number with generalized (non-integer) indices. A value of zero is assumed for undefined entries. `NumberDict` instances support addition, and subtraction with other `NumberDict` instances, and multiplication and division by scalars.

Module Scientific.Physics

Module Scientific.Physics.PhysicalQuantities

Physical quantities with units.

This module provides a data type that represents a physical quantity together with its unit. It is possible to add and subtract these quantities if the units are compatible, and a quantity can be converted to another compatible unit. Multiplication, subtraction, and raising to integer powers is allowed without restriction, and the result will have the correct unit. A quantity can be raised to a non-integer power only if the result can be represented by integer powers of the base units.

The values of physical constants are taken from the 1986 recommended values from CODATA. Other conversion factors (e.g. for British units) come from various sources. I can't guarantee for the correctness of all entries in the unit table, so use this at your own risk!

Class PhysicalQuantity: Physical quantity with units

Constructor:

- `PhysicalQuantity(value, unit)`, where `value` is a number of arbitrary type and `unit` is a string containing the unit name.
- `PhysicalQuantity(string)`, where `string` contains both the value and the unit. This form is provided to make interactive use more convenient.

`PhysicalQuantity` instances allow addition, subtraction, multiplication, and division with each other as well as multiplication, division, and exponentiation with numbers. Addition and subtraction check that the units of the two operands are compatible and return the result in the units of the first operand. A limited set of mathematical functions (from module `Numeric`) is applicable as well:

`sqrt` equivalent to exponentiation with 0.5.

`sin`, `cos`, `tan` applicable only to objects whose unit is compatible with `rad`.

Methods:

- `convertToUnit(unit)`
Changes the unit to `unit` and adjusts the value such that the combination is equivalent. The new unit is by a string containing its name. The new unit must be compatible with the previous unit of the object.
- `inUnitsOf(*units)`
Returns one or more `PhysicalQuantity` objects that express the same physical quantity in different units. The units are specified by strings containing their names. The units must be compatible with the unit of the object. If one unit is specified, the return value is a single `PhysicalObject`. If several units are specified, the return value is a tuple of `PhysicalObject` instances with with one element per unit such that the sum of all quantities in the tuple equals the the original quantity and all the values except for the last one are integers. This is used to convert to irregular unit systems like hour/minute/second. The original object will not be changed.

Functions

- `isPhysicalQuantity(x)`
Returns 1 if `x` is an instance of `PhysicalQuantity`.

Module Scientific.Physics.Potential

This module offers two strategies for automagically calculating the gradients (and optionally force constants) of a potential energy function (or any other function of vectors, for that matter). The more convenient strategy is to create an object of the class `PotentialWithGradients`. It takes a regular Python function object defining the potential energy and is itself a callable object returning the energy and its gradients with respect to all arguments that are vectors.

Example:

```
def _harmonic(k,r1,r2):  
    dr = r2-r1  
    return k*dr*dr  
harmonic = PotentialWithGradients(_harmonic)  
energy, gradients = harmonic(1., Vector(0,3,1), Vector(1,2,0))
```

```
print energy, gradients
```

```
prints
```

```
3.0
```

```
[Vector(-2.0,2.0,2.0), Vector(2.0,-2.0,-2.0)]
```

The disadvantage of this procedure is that if one of the arguments is a vector parameter, rather than a position, an unnecessary gradient will be calculated. A more flexible method is to insert calls to two function from this module into the definition of the energy function. The first, `DerivVectors()`, is called to indicate which vectors correspond to gradients, and the second, `EnergyGradients()`, extracts energy and gradients from the result of the calculation. The above example is therefore equivalent to

```
def harmonic(k, r1, r2):
    r1, r2 = DerivVectors(r1, r2)
    dr = r2-r1
    e = k*dr*dr
    return EnergyGradients(e,2)
```

To include the force constant matrix, the above example has to be modified as follows:

```
def _harmonic(k,r1,r2):
    dr = r2-r1
    return k*dr*dr
harmonic = PotentialWithGradientsAndForceConstants(_harmonic)
energy, gradients, force_constants = harmonic(1.,Vector(0,3,1),Vector(1,2,0))
print energy
print gradients
print force_constants
```

The force constants are returned as a nested list representing a matrix. This can easily be converted to an array for further processing if the numerical extensions to Python are available.

Module Scientific.Statistics

Functions

- `mean(data)`
Returns the mean (average value) of `data` (a sequence of numbers).
- `variance(data)`
Returns the variance of `data` (a sequence of numbers).
- `standardDeviation(data)`
Returns the standard deviation of `data` (a sequence of numbers).
- `median(data)`
Returns the median of `data` (a sequence of numbers).
- `skewness(data)`
Returns the skewness of `data` (a sequence of numbers).
- `kurtosis(data)`
Returns the kurtosis of `data` (a sequence of numbers).
- `correlation(data1, data2)`
Returns the correlation coefficient between `data1` and `data2`, which must have the same length.

Module Scientific.Statistics.Histogram

Class Histogram: Histogram in one variable

Constructor: `Histogram(data, bins, range=None)`

`data` a sequence of data points

`bins` the number of bins into which the data is to be sorted

`range` a tuple of two values, specifying the lower and the upper end of the interval spanned by the bins. Any data point outside this interval will be ignored. If no range is given, the smallest and largest data values are used to define the interval.

The bin index and the number of points in a bin can be obtained by indexing the histogram with the bin number. Application of `len()` yields the number of bins. A histogram thus behaves like a sequence of bin index - bin count pairs.

Methods:

- `addData(data)`
Add the values in `data` (a sequence of numbers) to the originally supplied data. Note that this does not affect the default range of the histogram, which is fixed when the histogram is created.
- `normalize(norm=1.0)`
Scales all counts by the same factor such that their sum is `norm`.
- `normalizeArea(norm=1.0)`
Scales all counts by the same factor such that the area under the histogram is `norm`.

Module Scientific.Threading

Module Scientific.Threading.TaskManager

Class TaskManager: Parallel task manager for shared-memory multiprocessor machines

This class provides a rather simple way to profit from shared-memory multiprocessor machines by running several tasks in parallel. The calling program decides how many execution threads should run at any given time, and then feeds compute tasks to the task manager, who runs them as soon as possible without exceeding the maximum number of threads.

The major limitation of this approach lies in Python's Global Interpreter Lock. This effectively means that no more than one Python thread can run at the same time. Consequently, parallelization can only be achieved if the tasks to be parallelized spend significant time in C extension modules that release the Global Interpreter Lock.

Constructor: `TaskManager(nthreads)`

`nthreads` the maximum number of compute threads that should run in parallel. Note: This does not include the main thread which generated and feeds the task manager!

Methods:

- `runTask(function, args)`
Add a task defined by `function`. This must be a callable object, which will be called exactly once. The arguments of the call are the elements of the tuple `args` plus one additional argument which is a lock object. The task can use this lock object in order to get temporary exclusive access to data shared with other tasks, e.g. a list in which to accumulate results.
- `terminate()`
Wait until all tasks have finished.

Module Scientific.TkWidgets

Class `FilenameEntry`: Filename entry widget

Constructor: `FilenameEntry(master, text, pattern, must_exist_flag=1)`

`master` the master widget

`text` the label in front of the filename box

`pattern` the filename matching pattern that determines the file list in the file selection dialog

`must_exists_flag` allow only names of existing files

A `FilenameEntry` widget consists of three parts: an identifying label, a text entry field for the filename, and a button labelled **browse** which call a file selection dialog box for picking a file name.

Methods:

- `get()`
Return the current filename. If `must_exist_flag` is true, verify that the name refers to an existing file. Otherwise an error message is displayed and a `ValueError` is raised.

Class `FloatEntry`: An entry field for float numbers

Constructor: `FloatEntry(master, text, initial=None, lower=None, upper=None)`

`master` the master widget

`text` the label in front of the entry field

`initial` an optional initial value (default: blank field)

`upper` an optional upper limit for the value

`lower` an optional lower limit for the value

A FloatEntry widget consists of a label followed by a text entry field.

Methods:

- `set(value)`
Set the value to `value`.
- `get()`
Return the current value, verifying that it is a number and between the specified limits. Otherwise an error message is displayed and a `ValueError` is raised.

Class IntEntry: An entry field for integer numbers

Constructor: `IntEntry(master, text, initial=None, lower=None, upper=None)`

`master` the master widget

`text` the label in front of the entry field

`initial` an optional initial value (default: blank field)

`upper` an optional upper limit for the value

`lower` an optional lower limit for the value

A IntEntry widget consists of a label followed by a text entry field.

Methods:

- `get()`
Return the current value, verifying that it is an integer and between the specified limits. Otherwise an error message is displayed and a `ValueError` is raised.

Class ButtonBar: A horizontal array of buttons

Constructor: `ButtonBar(master, left_button_list, right_button_list)`

`master` the master widget

`left_button_list` a list of (text, action) tuples specifying the buttons on the left-hand side of the button bar

`right_button_list` a list of (text, action) tuples specifying the buttons on the right-hand side of the button bar

Class `StatusBar`: A status bar

Constructor: `StatusBar(master)`

`master` the master widget

A status bar can be used to inform the user about the status of an ongoing calculation. A message can be displayed with `set()` and removed with `clear()`. In both cases, the `StatusBar` object makes sure that the change takes place immediately. While a message is being displayed, the cursor form is changed to a watch.

Module `Scientific.TkWidgets.TkPlotCanvas`

Class `PolyLine`: Multiple connected lines

Constructor: `PolyLine(points, **—attr—)`

`points` any sequence of (x, y) number pairs

`attr` line attributes specified by keyword arguments:

- `width`: the line width (default: 1)
- `color`: a string whose value is one of the color names defined in Tk (default: "black")
- `stipple`: a string whose value is the name of a bitmap defined in Tk, or `None` for no bitmap (default: `None`)

Class `VerticalLine`: A vertical line

Constructor: `VerticalLine(xpos, **—attr—)`

`xpos` the x coordinate of the line

`attr` line attributes specified by keyword arguments:

- `width`: the line width (default: 1)
- `color`: a string whose value is one of the color names defined in Tk (default: "black")
- `stipple`: a string whose value is the name of a bitmap defined in Tk, or `None` for no bitmap (default: `None`)

Class HorizontalLine: A horizontal line

Constructor: HorizontalLine(ypos, **attr—)

ypos the y coordinate of the line

attr line attributes specified by keyword arguments:

- **width**: the line width (default: 1)
- **color**: a string whose value is one of the color names defined in Tk (default: "black")
- **stipple**: a string whose value is the name of a bitmap defined in Tk, or **None** for no bitmap (default: **None**)

Class PolyMarker: Series of markers

Constructor: PolyPoints(points, **attr—)

points any sequence of (x, y) number pairs

attr marker attributes specified by keyword arguments:

- **width**: the line width for drawing the marker (default: 1)
- **color**: a string whose value is one of the color names defined in Tk, defines the color of the line forming the marker (default: black)
- **fillcolor**: a string whose value is one of the color names defined in Tk, defines the color of the interior of the marker (default: black)
- **marker**: one of **circle** (default), **dot**, **square**, **triangle**, **triangle_down**, **cross**, **plus**

Class PlotGraphics: Compound graphics object

Constructor: PlotGraphics(objects)

objects a list whose elements can be instances of the classes PolyLine, PolyMarker, and PlotGraphics.

Class PlotCanvas: Tk plot widget

Constructor: PlotCanvas(master, width, height, **—attributes—).

The arguments have the same meaning as for a standard Tk canvas. The default background color is white and the default font is Helvetica at 10 points.

PlotCanvas objects support all operations of Tk widgets.

There are two attributes in addition to the standard Tk attributes:

zoom a logical variable that indicates whether interactive zooming (using the left mouse button) is enabled; the default is 0 (no zoom)

select enables the user to select a range along the x axis by dragging the mouse (with the left button pressed) in the area *under* the x axis. If **select** is 0, no selection is possible. Otherwise the value of **select** must be a callable object that is called whenever the selection changes, with a single argument that can be None (no selection) or a tuple containing two x values.

Methods:

- **draw(graphics, xaxis=None, yaxis=None)**
Draws the graphics object **graphics**, which can be a PolyLine, PolyMarker, or PlotGraphics object. The arguments **xaxis** and **yaxis** specify how axes are drawn: **None** means that no axis is drawn and the graphics objects are scaled to fill the canvas optimally. **"automatic"** means that the axis is drawn and a suitable value range is determined automatically. A sequence of two numbers means that the axis is drawn and the value range is the interval specified by the two numbers.
- **clear()**
Clears the canvas.
- **redraw()**
Redraws the last canvas contents.
- **select(range)**
Shows the given **range** as highlighted. **range** can be None (no selection) or a sequence of two values on the x-axis.

Module Scientific.TkWidgets.TkVisualizationCanvas

Class PolyLine3D: Multiple connected lines

Constructor: PolyLine(**points**, ****—attr—**), where **points** is any sequence of (x, y, z) number triples and **attr** stands for line attributes specified by keyword arguments, which are **width** (an integer) and **color** (a string whose value is one of the color names defined in Tk). The default is a black line of width 1.

Class VisualizationGraphics: Compound graphics object

Constructor: VisualizationGraphics(**objects**), where **objects** is a list whose elements can be instances of the classes PolyLine3D and VisualizationGraphics.

Class VisualizationCanvas: Tk visualization widget

Constructor: VisualizationCanvas(**master**, **width**, **height**, ****—attributes—**). The arguments have the same meaning as for a standard Tk canvas. The default background color is white and the default font is Helvetica at 10 points.

VisualizationCanvas objects support all operations of Tk widgets.

Interactive manipulation of the display is possible with click-and-drag operations. The left mouse button rotates the objects, the middle button translates it, and the right button scales it up or down.

Methods:

- **draw(graphics)**
Draws the graphics object **graphics**, which can be a PolyLine3D or a VisualizationGraphics object.
- **clear(keepscale=0)**
Clears the canvas.

Module Scientific.Visualization

The modules in this subpackage provide visualization of 3D objects using different backends (VRML, VMD, VPython), but with an almost identical interface. It is thus possible to write generic 3D graphics code in which the backend can be changed by modifying a single line of code.

The intended application of these modules is scientific visualization. Many sophisticated 3D objects are therefore absent, as are complex surface definitions such as textures.

Module Scientific.Visualization.Color

This module provides color definitions that are used in the modules VRML, VRML2, and VMD.

Class Color: RGB Color specification

Constructor: `Color(rgb)`, where `rgb` is a sequence of three numbers between zero and one, specifying the red, green, and blue intensities.

Color objects can be added and multiplied with scalars.

Class ColorScale: Mapping of a number interval to a color range

Constructor: `ColorScale(range)`, where `range` can be a tuple of two numbers (the center of the interval and its width), or a single number specifying the widths for a default center of zero.

Evaluation: `colorscale(number)` returns the Color object corresponding to `number`. If `number` is outside the predefined interval, the closest extreme value of the interval is used.

The color scale is blue - green - yellow - orange - red.

Class SymmetricColorScale: Mapping of a symmetric number interval to a color range

Constructor: `SymmetricColorScale(range)`, where `range` is a single number defining the interval, which is `-range` to `range`.

Evaluation: `colorscale(number)` returns the Color object corresponding to `number`. If `number` is outside the predefined interval, the closest extreme value of the interval is used.

The colors are red for negative numbers and green for positive numbers, with a color intensity proportional to the absolute value of the argument.

Functions

- `ColorByName(name)`
Returns a Color object corresponding to `name`. The known names are black, white, grey, red, green, blue, yellow, magenta, cyan, orange, violet, olive, and brown. Any color can be prefixed by "light " or "dark " to yield a variant.

Module Scientific.Visualization.VMD

This module provides definitions of simple 3D graphics objects and scenes containing them, in a form that can be fed to the molecular visualization program VMD. Scenes can either be written as VMD script files, or visualized directly by running VMD.

There are a few attributes that are common to all graphics objects:

material a Material object defining color and surface properties

comment a comment string that will be written to the VRML file

reuse a boolean flag (defaulting to false). If set to one, the object may share its VRML definition with other objects. This reduces the size of the VRML file, but can yield surprising side effects in some cases.

This module is almost compatible with the modules VRML and VRML2, which provide visualization by VRML browsers. There is no Polygon objects, and the only material attribute supported is `diffuse_color`. Note also that loading a scene with many cubes into VMD is very slow, because each cube is represented by 12 individual triangles.

Example:

```
from VMD import *
scene = Scene([])
```

```
scale = ColorScale(10.)
for x in range(11):
    color = scale(x)
    scene.addObject(Cube(Vector(x, 0., 0.), 0.2,
                        material=Material(diffuse_color = color)))
scene.view()
```

Class Scene: VMD scene

A VMD scene is a collection of graphics objects that can be written to a VMD script file or fed directly to VMD.

Constructor: Scene(**objects**=None, ****—options—**)

objects a list of graphics objects or **None** for an empty scene

options options as keyword arguments. The only option available is "scale", whose value must be a positive number which specifies a scale factor applied to all coordinates of geometrical objects *except* for molecule objects, which cannot be scaled.

Methods:

- **addObject(object)**
Adds **object** to the list of graphics objects.
- **writeToFile(filename, delete=0)**
Writes the scene to a VRML file with name **filename**.
- **view()**
Start VMD for the scene.

Class Molecules: Molecules from a PDB file

Constructor: Molecules(**pdb_file**)

Class Sphere: Sphere

Constructor: Sphere(center, radius, **—attributes—)

center the center of the sphere (a vector)

radius the sphere radius (a positive number)

attributes any graphics object attribute

Class Cube: Cube

Constructor: Cube(center, edge, **—attributes—)

center the center of the cube (a vector)

edge the length of an edge (a positive number)

attributes any graphics object attribute

The edges of a cube are always parallel to the coordinate axes.

Class Cylinder: Cylinder

Constructor: Cylinder(point1, point2, radius, faces=(1, 1, 1),
**—attributes—)

point1, point2 the end points of the cylinder axis (vectors)

radius the radius (a positive number)

attributes any graphics object attribute

faces a sequence of three boolean flags, corresponding to the cylinder hull and the two circular end pieces, specifying for each of these parts whether it is visible or not.

Class Cone: Cone

Constructor: Cone(point1, point2, radius, face=1, **—attributes—)

point1, point2 the end points of the cylinder axis (vectors). point1 is the tip of the cone.

radius the radius (a positive number)

attributes any graphics object attribute

face a boolean flag, specifying if the circular bottom is visible

Class Line: Line

Constructor: Line(point1, point2, **—attributes—)

point1, point2 the end points of the line (vectors)

attributes any graphics object attribute

Class Arrow: Arrow

An arrow consists of a cylinder and a cone.

Constructor: Arrow(point1, point2, radius, **—attributes—)

point1, point2 the end points of the arrow (vectors). point2 defines the tip of the arrow.

radius the radius of the arrow shaft (a positive number)

attributes any graphics object attribute

Class Material: Material for graphics objects

A material defines the color and surface properties of an object.

Constructor: Material(**—attributes—)

The accepted attributes are "ambient_color", "diffuse_color", "specular_color", "emissive_color", "shininess", and "transparency". Only "diffuse_color" is used, the others are permitted for compatibility with the VRML modules.

Functions

- `DiffuseMaterial(color)`
Returns a material with the `diffuse color` attribute set to `color`.

Module Scientific.Visualization.VPython

Class Scene: VPython scene

A VPython scene is a collection of graphics objects that can be shown in a VPython window. When the "view" method is called, a new window is created and the graphics objects are displayed in it.

Constructor: `Scene(objects=None, **—options—)`

`objects` a list of graphics objects or `None` for an empty scene

`options` options as keyword arguments: "title" (the window title, default: "VPython scene"), "width" (the window width, default: 300), "height" (the window height, default: 300), "background" (the background color, default: `black`)

Methods:

- `addObject(object)`
Adds `object` to the list of graphics objects.
- `view()`
Open a VPython window for the scene.

Class Sphere: Sphere

Constructor: `Sphere(center, radius, **—attributes—)`

`center` the center of the sphere (a vector)

`radius` the sphere radius (a positive number)

`attributes` any graphics object attribute

Class Cube: Cube

Constructor: Cube(center, edge, **—attributes—)

center the center of the cube (a vector)

edge the length of an edge (a positive number)

attributes any graphics object attribute

The edges of a cube are always parallel to the coordinate axes.

Class Cylinder: Cylinder

Constructor: Cylinder(point1, point2, radius, **—attributes—)

point1, point2 the end points of the cylinder axis (vectors)

radius the radius (a positive number)

attributes any graphics object attribute

Class Arrow: Arrow

Constructor: Arrow(point1, point2, radius, **—attributes—)

point1, point2 the end points of the cylinder axis (vectors)

radius the radius (a positive number)

attributes any graphics object attribute

Class Cone: Cone

Constructor: Cone(point1, point2, radius, **—attributes—)

point1, point2 the end points of the cylinder axis (vectors). point1 is the tip of the cone.

radius the radius (a positive number)

attributes any graphics object attribute

Class PolyLines: Multiple connected lines

Constructor: PolyLines(points, **—attributes—)

points a sequence of points to be connected by lines

attributes any graphics object attribute

Class Line: Line

Constructor: Line(point1, point2, **—attributes—)

point1, point2 the end points of the line (vectors)

attributes any graphics object attribute

Class Polygons: Polygons

Constructor: Polygons(points, index_lists, **—attributes—)

points a sequence of points

index_lists a sequence of index lists, one for each polygon. The index list for a polygon defines which points in **points** are vertices of the polygon.

attributes any graphics object attribute

Class Material: Material for graphics objects

A material defines the color and surface properties of an object.

Constructor: Material(**—attributes—)

The attributes are "ambient_color", "diffuse_color", "specular_color", "emissive_color", "shininess", and "transparency".

Functions

- DiffuseMaterial(color)
Returns a material with the **diffuse_color** attribute set to **color**.
- EmissiveMaterial(color)
Returns a material with the **emissive_color** attribute set to **color**.

Module Scientific.Visualization.VRML

This module provides definitions of simple 3D graphics objects and VRML scenes containing them. The objects are appropriate for data visualization, not for virtual reality modelling. Scenes can be written to VRML files or visualized immediately using a VRML browser, whose name is taken from the environment variable VRMLVIEWER (under Unix).

There are a few attributes that are common to all graphics objects:

material a Material object defining color and surface properties

comment a comment string that will be written to the VRML file

reuse a boolean flag (defaulting to false). If set to one, the object may share its VRML definition with other objects. This reduces the size of the VRML file, but can yield surprising side effects in some cases.

This module used the original VRML definition, version 1.0. For the newer VRML 2 or VRML97, use the module VRML2, which uses exactly the same interface. There is another almost perfectly compatible module VMD, which produces input files for the molecular visualization program VMD.

Example:

```
from Scientific.Visualization.VRML import *
scene = Scene([])
scale = ColorScale(10.)
for x in range(11):
    color = scale(x)
    scene.addObject(Cube(Vector(x, 0., 0.)), 0.2,
                    material=Material(diffuse_color = color))
scene.view()
```

Class Scene: VRML scene

A VRML scene is a collection of graphics objects that can be written to a VRML file or fed directly to a VRML browser.

Constructor: Scene(objects=None, cameras=None, **options—)

objects a list of graphics objects or `None` for an empty scene

cameras a list of cameras (not yet implemented)

options options as keyword arguments (none defined at the moment; this argument is provided for compatibility with other modules)

Methods:

- **addObject(object)**
Adds `object` to the list of graphics objects.
- **addCamera(camera)**
Adds `cameras` to the list of cameras.
- **writeToFile(filename)**
Writes the scene to a VRML file with name `filename`.
- **view()**
Start a VRML browser for the scene.

Class Sphere: Sphere

Constructor: `Sphere(center, radius, **—attributes—)`

center the center of the sphere (a vector)

radius the sphere radius (a positive number)

attributes any graphics object attribute

Class Cube: Cube

Constructor: `Cube(center, edge, **—attributes—)`

center the center of the cube (a vector)

edge the length of an edge (a positive number)

attributes any graphics object attribute

The edges of a cube are always parallel to the coordinate axes.

Class Cylinder: Cylinder

Constructor: `Cylinder(point1, point2, radius, faces=(1, 1, 1), **—attributes—)`

`point1, point2` the end points of the cylinder axis (vectors)

`radius` the radius (a positive number)

`attributes` any graphics object attribute

`faces` a sequence of three boolean flags, corresponding to the cylinder hull and the two circular end pieces, specifying for each of these parts whether it is visible or not.

Class Cone: Cone

Constructor: `Cone(point1, point2, radius, face=1, **—attributes—)`

`point1, point2` the end points of the cylinder axis (vectors). `point1` is the tip of the cone.

`radius` the radius (a positive number)

`attributes` any graphics object attribute

`face` a boolean flag, specifying if the circular bottom is visible

Class Line: Line

Constructor: `Line(point1, point2, **—attributes—)`

`point1, point2` the end points of the line (vectors)

`attributes` any graphics object attribute

Class PolyLines: Multiple connected lines

Constructor: `PolyLines(points, **—attributes—)`

`points` a sequence of points to be connected by lines

`attributes` any graphics object attribute

Class Polygons: Polygons

Constructor: Polygons(points, index_lists, **—attributes—)

points a sequence of points

index_lists a sequence of index lists, one for each polygon. The index list for a polygon defines which points in **points** are vertices of the polygon.

attributes any graphics object attribute

Class Arrow: Arrow

An arrow consists of a cylinder and a cone.

Constructor: Arrow(point1, point2, radius, **—attributes—)

point1, **point2** the end points of the arrow (vectors). **point2** defines the tip of the arrow.

radius the radius of the arrow shaft (a positive number)

attributes any graphics object attribute

Class Material: Material for graphics objects

A material defines the color and surface properties of an object.

Constructor: Material(**—attributes—)

The attributes are "ambient_color", "diffuse_color", "specular_color", "emissive_color", "shininess", and "transparency".

Functions

- DiffuseMaterial(color)
Returns a material with the **diffuse_color** attribute set to **color**.
- EmissiveMaterial(color)
Returns a material with the **emissive_color** attribute set to **color**.

Module Scientific.Visualization.VRML2

This module provides definitions of simple 3D graphics objects and VRML scenes containing them. The objects are appropriate for data visualization, not for virtual reality modelling. Scenes can be written to VRML files or visualized immediately using a VRML browser, whose name is taken from the environment variable VRML2VIEWER (under Unix).

There are a few attributes that are common to all graphics objects:

material a Material object defining color and surface properties

comment a comment string that will be written to the VRML file

reuse a boolean flag (defaulting to false). If set to one, the object may share its VRML definition with other objects. This reduces the size of the VRML file, but can yield surprising side effects in some cases.

This module used the VRML 2.0 definition, also known as VRML97. For the original VRML 1, use the module VRML, which uses exactly the same interface. There is another almost perfectly compatible module VMD, which produces input files for the molecular visualization program VMD. Example:

```
from Scientific.Visualization.VRML2 import *
scene = Scene([])
scale = ColorScale(10.)
for x in range(11):
    color = scale(x)
    scene.addObject(Cube(Vector(x, 0., 0.)), 0.2,
                    material=Material(diffuse_color = color))
scene.view()
```

Class Scene: VRML scene

A VRML scene is a collection of graphics objects that can be written to a VRML file or fed directly to a VRML browser.

Constructor: Scene(objects=None, cameras=None, **—options—)

objects a list of graphics objects or `None` for an empty scene

cameras a list of cameras

options options as keyword arguments (none defined at the moment; this argument is provided for compatibility with other modules)

Methods:

- **addObject(object)**
Adds `object` to the list of graphics objects.
- **addCamera(camera)**
Adds `camera` to the list of cameras.
- **writeToFile(filename)**
Writes the scene to a VRML file with name `filename`.
- **view()**
Start a VRML browser for the scene.

Class Camera: Camera/viewpoint for a scene

Constructor: `Camera(position, orientation, description, field_of_view)`

position the location of the camera (a vector)

orientation an (axis, angle) tuple in which the axis is a vector and angle a number (in radians); axis and angle specify a rotation with respect to the standard orientation along the negative z axis

description a label for the viewpoint (a string)

field_of_view the field of view (a positive number)

Class NavigationInfo: Navigation Information

Constructor: `NavigationInfo(speed, type)`

speed walking speed in length units per second

type one of `WALK`, `EXAMINE`, `FLY`, `NONE`, `ANY`

Class Sphere: Sphere

Constructor: Sphere(center, radius, **—attributes—)

center the center of the sphere (a vector)

radius the sphere radius (a positive number)

attributes any graphics object attribute

Class Cube: Cube

Constructor: Cube(center, edge, **—attributes—)

center the center of the cube (a vector)

edge the length of an edge (a positive number)

attributes any graphics object attribute

The edges of a cube are always parallel to the coordinate axes.

Class Cylinder: Cylinder

Constructor: Cylinder(point1, point2, radius, faces=(1, 1, 1),
**—attributes—)

point1, point2 the end points of the cylinder axis (vectors)

radius the radius (a positive number)

attributes any graphics object attribute

faces a sequence of three boolean flags, corresponding to the cylinder hull
and the two circular end pieces, specifying for each of these parts
whether it is visible or not.

Class Cone: Cone

Constructor: `Cone(point1, point2, radius, face=1, **—attributes—)`

`point1`, `point2` the end points of the cylinder axis (vectors). `point1` is the tip of the cone.

`radius` the radius (a positive number)

`attributes` any graphics object attribute

`face` a boolean flag, specifying if the circular bottom is visible

Class Line: Line

Constructor: `Line(point1, point2, **—attributes—)`

`point1`, `point2` the end points of the line (vectors)

`attributes` any graphics object attribute

Class PolyLines: Multiple connected lines

Constructor: `PolyLines(points, **—attributes—)`

`points` a sequence of points to be connected by lines

`attributes` any graphics object attribute

Class Polygons: Polygons

Constructor: `Polygons(points, index_lists, **—attributes—)`

`points` a sequence of points

`index_lists` a sequence of index lists, one for each polygon. The index list for a polygon defines which points in `points` are vertices of the polygon.

`attributes` any graphics object attribute

Class Arrow: Arrow

An arrow consists of a cylinder and a cone.

Constructor: Arrow(point1, point2, radius, **—attributes—)

point1, **point2** the end points of the arrow (vectors). **point2** defines the tip of the arrow.

radius the radius of the arrow shaft (a positive number)

attributes any graphics object attribute

Class Material: Material for graphics objects

A material defines the color and surface properties of an object.

Constructor: Material(**—attributes—)

The attributes are "ambient_color", "diffuse_color", "specular_color", "emissive_color", "shininess", and "transparency".

Functions

- DiffuseMaterial(color)
Returns a material with the `diffuse_color` attribute set to `color`.
- EmissiveMaterial(color)
Returns a material with the `emissive_color` attribute set to `color`.

Module **Scientific.indexing**

This module provides a convenient method for constructing array indices algorithmically. It provides one importable object, `index_expression`. For any index combination, including slicing and axis insertion, `a[indices]` is the same as `a[index_expression[indices]]` for any array `a`. However, `index_expression[indices]` can be used anywhere in Python code and returns a tuple of indexing objects that can be used in the construction of complex index expressions.

Sole restriction: Slices must be specified in the double-colon form, i.e. `a[:,:]` is allowed, whereas `a[:]` is not.