BASIC LINEAR ALGEBRA MATRIX (PYTHON RECIPE) BY BILL MCNEILL

ACTIVESTATE CODE (HTTP://CODE.ACTIVESTATE.COM/RECIPES/189971/)

This recpie defines the Matrix class, an implementation of a linear algebra matrix. Arithmetic operations, trace, determinant, and minors are defined for it.

Python, 506 lines

```python
import types
import operator

"""Linear Algebra Matrix Class

The Matrix class is an implementation of a linear algebra matrix.
Arithmetic operations, trace, determinant, and minors are defined for it.  This
is a lightweight alternative to a numerical Python package for people who need
to do basic linear algebra.

Vectors are implemented as 1xN and Nx1 matricies.  There is no separate vector
class.  This implementation enforces the distinction between row and column
vectors.

Indexing is zero-based, i.e. the upper left-hand corner of a matrix is element
(0,0), not element (1,1).

Matricies are stored as a list of lists, where the top level lists are the rows
and the sub-lists are the columns.  Because of the way Python handles list
references, you have be careful when copying matrix objects.  If you have a
matrix a, assign b=a, and then change values in b, you will change values in a
as well.  Matrix copying should be done with copy.deepcopy.

This implementation has no memory-saving optimization for sparse matricies.  A
derived class may implement a more sophisticated storage method by overriding
the __getitem__ and __setitem__ functions.

Determinants are taken by expanding by minors on the top row.  The private
functions supplied for expansion by minors are more generic than what is needed
by this implementation.  They may be used by a derived class that wishes to do
more efficient expansion of sparse matricies.

By default, Matrix elements are members of the complex field, but if you want
to perform linear algebra on something other than numbers you may redefine
Matrix.null_element, Matrix.identity_element, and Matrix.inverse_element and
override the is_scalar_element function.

References:
        George Arfken, "Mathematical Methods for Physicists", 3rd ed. San Diego:
Academic Press Inc. (1985)
"""

__author__ = "Bill McNeill <billmcn@speakeasy.net>"
__version__ = "1.0"


class Matrix_Error(Exception):
    """Abstract parent for all matrix exceptions
    """
    pass


class Matrix_Arithmetic_Error(Matrix_Error):
    """Incorrect dimensions for arithmetic

    This exception is thrown when you try to add or multiply matricies of
    incompatible sizes.
    """
    def __init__(self, a, b, operation):
        self.a = a
        self.b = b
        self.operation = operation

    def __str__(self):
        return "Cannot %s a %dx%d and a %dx%d matrix" % \
            (self.operation, \
            self.a.rows(), self.a.cols(), \
            self.b.rows(), self.b.cols())


class Matrix_Multiplication_Error(Matrix_Arithmetic_Error):
    """Thrown when you try to multiply matricies of incompatible dimensions.

    This exception is also thrown when you try to right-multiply a row vector or
    left-multiply a column vector.
    """
    def __init__(self, a, b):
        Matrix_Arithmetic_Error.__init__(self, a, b, "multiply")


class Matrix_Addition_Error(Matrix_Arithmetic_Error):
    """Thrown when you try to add matricies of incompatible dimensions.
    """
    def __init__(self, a, b):
        Matrix_Arithmetic_Error.__init__(self, a, b, "add")


class Square_Error(Matrix_Error):
    """Square-matrix only

    This exception is thrown when you try to calculate a function that is only
    defined for square matricies on a non-square matrix.
    """
    def __init__(self, func):
        self.func = func

    def __str__(self):
        return "%s only defined for square matricies." % self.func


class Trace_Error(Square_Error):
    """Thrown when you try to get the trace of a non-square matrix.
    """
    def __init__(self):
        Square_Error.__init__(self, "The trace is")


class Minor_Error(Square_Error):
    """Thrown when you try to take a minor of a non-square matrix.
    """
    def __init__(self):
        Square_Error.__init__(self, "Minors are")


class Determinant_Error(Square_Error):
    """Thrown when you try to take the determinant of a non-square matrix.
    """
    def __init__(self):
        Square_Error.__init__(self, "The determinant is")



class Matrix:
    """A linear algebra matrix

    This class defines a generic matrix and the basic matrix operations from
    linear algebra.  An instance of this class is a single matrix with
    particular values.
    """
    null_element = 0
    identity_element = 1
    inverse_element = -1

    def __init__(self, *args):
        """Matrix constructor

        A matrix can be created in three ways.

        1. A single integer argument is supplied.  The constructor creates a
        null square matrix of that size.  For example

            Matrix(2)

        creates the following matrix

            0       0
            0       0

        2. Two integer arguments are supplied.  The constructor creates a null
        matrix of size first argument x second argument.  For example

            Matrix(2, 3)

        creates the following matrix

            0       0       0
            0       0       0

        3. A list of lists is supplied.  It represents a set of initial matrix
        values.  Each element is a row and each sub-list is a column.
        For example

            Matrix([[1,2,3], [4,5,6], [7,8,9]])

        creates the following matrix

            1       2       3
            4       5       6
            7       8       9

        """
        if not (len(args) == 1 or len(args) == 2):
            raise TypeError("Matrix() takes 1 or 2 arguments (%d given)") % \
                len(args)
        if len(args) == 2:      # Two arguments
            # Create an null n,m matrix.
            row, col = args
            self.create_null_matrix(row, col)
        else:   # One argument
            if isinstance(args[0], types.IntType):
                # Create a square null matrix.
                self.create_null_matrix(args[0], args[0])
            else:
                # Create a matrix from initial values.
                self.m = args[0]
                if __debug__:
                    # Verify correct format for m.
                    if not isinstance(args[0], types.ListType):
                        raise TypeError("Invalid initial data %s" % args[0])
                    for row in args[0]:
                        if not isinstance(row, types.ListType):
                            raise ValueError("Invalid initial data %s" % \
                                args[0])
                        if not (len(row) == len(args[0][0])):
                            raise ValueError("Non-rectangular initial data")
                    if not (self.cols() > 0):
                        raise ValueError("invalid number of columns %d" % \
                            self.cols())
                    if self.rows() == 1 and self.cols() == 1:
                        raise ValueError("Cannot create 1x1 matrix")

    def create_null_matrix(self, row, col):
        """ Create a matrix using the null value

        This is a private function called by __init__.
        """
        if not row > 0:
            raise ValueError("invalid number of rows %d" % row)
        if not col > 0:
            raise ValueError("invalid number of columns %d" % col)
        if row == 1 and col == 1:
            raise ValueError("Cannot create 1x1 matrix")
        # Note, you cannot simply write
        #       self.m = [[self.null_element]*col]*row
        # because this will make all the rows references of a single instance.
        self.m = []
        for i in xrange(row):
            self.m.append([])
            for j in xrange(col):
                self.m[i].append(self.null_element)

    def __str__(self):
        s = ""
        for row in self.m:
            s += "%s\n" % row
        return s

    def __cmp__(self, other):
        if not isinstance(other, Matrix):
            raise TypeError("Cannot compare matrix with %s" % type(other))
        return cmp(self.m, other.m)

    def __getitem__(self, (row, col)):
        """The value at (row, col)

        For example, to get the value of element 1,3 say

            m[(1,3)]
        """
        return self.m[row][col]

    def __setitem__(self, (row, col), value):
        """Sets the value at (row, col)

        For example, to set the value of element 1,3 to 5 say

            m[(1,3)] = 5
        """
        self.m[row][col] = value

    def rows(self):
        """The number of rows in the matrix
        """
        return len(self.m)

    def cols(self):
        """The number of columns in the matrix
        """
        return len(self.m[0])

    def row(self, i):
        """The ith row of the matrix
        """
        return self.m[i]

    def col(self, j):
        """The jth row of the matrix
        """
        r = []
        for row in self.m:
            r.append(row[j])
        return r

    def __add__(self, other):
        """Add matrix self+other
        """
        if not isinstance(other, Matrix):
            raise TypeError("Cannot add a matrix to type %s" % type(other))
        if not (self.cols() == other.cols() and self.rows() == other.rows()):
            raise Matrix_Addition_Error(self, other)
        r = []
        for row in xrange(self.rows()):
            r.append([])
            for col in xrange(self.cols()):
                r[row].append(self[(row, col)] + other[(row, col)])
        return Matrix(r)

    def __neg__(self):
        """Negate the current matrix
        """
        return self.inverse_element*self

    def __sub__(self, other):
        """Subtract matrix self-other
        """
        return self + -other

    def __mul__(self, other):
        """Multiply matrix self*other

        other can be another matrix or a scalar.
        """
        if self.is_scalar_element(other):
            return self.scalar_multiply(other)
        if not isinstance(other, Matrix):
            raise TypeError("Cannot multiply matrix and type %s" % type(other))
        if other.is_row_vector():
            raise Matrix_Multiplication_Error(self, other)
        return self.matrix_multiply(other)

    def __rmul__(self, other):
        """Multiply other*self

        This is only called if other.__add__ is not defined, so assume that
        other is a scalar.
        """
        if not self.is_scalar_element(other):
            raise TypeError("Cannot right-multiply by %s" % type(other))
        return self.scalar_multiply(other)

    def scalar_multiply(self, scalar):
        """Multiply the matrix by a scalar value.

        This is a private function called by __mul__ and __rmul__.
        """
        r = []
        for row in self.m:
            r.append(map(lambda x: x*scalar, row))
        return Matrix(r)

    def matrix_multiply(self, other):
        """Multiply the matrix by another matrix.

        This is a private function called by __mul__.
        """
        # Take the product of two matricies.
        r = []
        assert(isinstance(other, Matrix))
        if not self.cols() == other.rows():
            raise Matrix_Multiplication_Error(self, other)
        for row in xrange(self.rows()):
            r.append([])
            for col in xrange(other.cols()):
                r[row].append( \
                    self.vector_inner_product(self.row(row), other.col(col)))
        if len(r) == 1 and len(r[0]) == 1:
            # The result is a scalar.
            return r[0][0]
        else:
            # The result is a matrix.
            return Matrix(r)

    def is_row_vector(self):
        """Is the matrix a row vector?
        """
        return self.rows() == 1 and self.cols() > 1

    def is_column_vector(self):
        """Is the matrix a column vector?
        """
        return self.cols() == 1 and self.rows() > 1

    def is_square(self):
        """Is the matrix square?
        """
        return self.rows() == self.cols()

    def transpose(self):
        """The transpose of the matrix
        """
        r = []
        for col in xrange(self.cols()):
            r.append(self.col(col))
        return Matrix(r)

    def trace(self):
        """The trace of the matrix
        """
        if not self.is_square():
            raise Trace_Error()
        t = 0
        for i in xrange(self.rows()):
            t += self[(i,i)]
        return t

    def determinant(self):
        """The determinant of the matrix
        """
        if not self.is_square():
            raise Determinant_Error()
        # Calculate 2x2 determinants directly.
        if self.rows() == 2:
            return self[(0, 0)]*self[(1, 1)] - self[(0, 1)]*self[(1, 0)]
        # Expand by minors for larger matricies.
        return self.expand_by_minors_on_row(0)

    def expand_by_minors_on_row(self, row):
        """Calculates the determinant by expansion of minors

        This function returns the determinant of the matrix by doing an
        expansion of minors on the specified row.
        """
        assert(row < self.rows())
        d = 0
        for col in xrange(self.cols()):
            # Note: the () around -1 are needed.  Otherwise you get -(1**col).
            d += (-1)**(row+col)* \
                self[(row, col)]*self.minor(row, col).determinant()
        return d

    def expand_by_minors_on_column(self, col):
        """Calculates the determinant by expansion of minors

        This function returns the determinant of the matrix by doing an
        expansion of minors on the specified column.
        """
        assert(col < self.cols())
        d = 0
        for row in xrange(self.rows()):
            # Note: the () around -1 are needed.  Otherwise you get -(1**col).
            d += (-1)**(row+col) \
                *self[(row, col)]*self.minor(row, col).determinant()
        return d

    def minor(self, i, j):
        """A minor of the matrix

        This function returns the minor given by striking out row i and
        column j of the matrix.
        """
        # Verify parameters.
        if not self.is_square():
            raise Minor_Error()
        if i<0 or i>=self.rows():
            raise ValueError("Row value %d is out of range" % i)
        if j<0 or j>=self.cols():
            raise ValueError("Column value %d is out of range" % j)
        # Create the output matrix.
        m = Matrix(self.rows()-1, self.cols()-1)
        # Loop through the matrix, skipping over the row and column specified
        # by i and j.
        minor_row = minor_col = 0
        for self_row in xrange(self.rows()):
            if not self_row == i:   # Skip row i.
                for self_col in xrange(self.cols()):
                    if not self_col == j:   # Skip column j.
                        m[(minor_row, minor_col)] = self[(self_row, self_col)]
                        minor_col += 1
                minor_col = 0
                minor_row += 1
        return m

    def vector_inner_product(self, a, b):
        """Takes the inner product of vectors a and b

        a and b are lists.
        This is a private function called by matrix_multiply.
        """
        assert(isinstance(a, types.ListType))
        assert(isinstance(b, types.ListType))
        return reduce(operator.add, map(operator.mul, a, b))

    def is_scalar_element(self, x):
        """Is x a scalar

        By default a scalar is an element in the complex number field.
        A class that wants to perform linear algebra on things other than
        numbers may override this function.
        """
        return isinstance(x, types.IntType) or \
                isinstance(x, types.FloatType) or \
                isinstance(x, types.ComplexType)


def unit_matrix(n):
    """Creates an nxn unit matirx

    The unit matrix is a diagonal matrix whose diagonal is composed of
    identity elements.  For example, unit_matrix(3) returns the matrix

        1 0 0
        0 1 0
        0 0 1
    """
    m = Matrix(n)
    for i in xrange(m.rows()):
        m[(i,i)] = m.identity_element
    return m

def row_vector(v):
    """Creates a row vector.

    v is a list of the column values
    """
    if not isinstance(v, types.ListType):
        raise TypeError("Row vector data must be a list")
    return Matrix([v])

def column_vector(v):
    """Creates a column vector.

    v is a list of the row values
    """
    if not isinstance(v, types.ListType):
        raise TypeError("Column vector data must be a list")
    return Matrix(map(lambda x: [x], v))
```

This is a lightweight alternative to a numerical Python package for people who need to do basic linear algebra.

Tags: algorithms

1 COMMENT

Raymond Hettinger 14 years ago

See Also. There is another pure python matrix implementation at http://users.rcn.com/python/download/python.htm .

It goes beyond the basics and provides least squares solutions of matrix equations, QR decompositions, eigenvalues, and curve-fitting.