```
#! /usr/bin/env python
'''
Rayleigh1D.py

A program to calculate scattering amplitudes from a rough surface.
See usage string for more information.

Created on 2009-09-28.
Copyright (c) 2009 Tor Nordam and Paul Anton Letnes. All rights reserved.
'''

import sys
import optparse
import numpy
import scipy
# Use one of the two below. It seems they both use fftpack, but we should
# check if fft performance becomes an issue.
from numpy.fft import fft
# from scipy.fftpack import fft
from scipy.linalg.lapack import flapack as lapack
from Scientific.IO import NetCDF
import Numeric

######################
# Help messages etc. #
######################
positional_args = 'OMEGA K_RANGE FILE'
usage = '''\
Rayleigh1D.py [options] %s

Calculate the reflection matrix elements from a rough surface, R_\\nu(q|k),
using the Reduced Rayleigh Equation (RRE), for the given light frequency
(OMEGA) and the given parallel wave numbers (K_RANGE). TODO: units! Save
output
to FILE, using the netCDF format. \
''' % positional_args

n_default = -1
n_help = '''\
Number of points used to discretize the surface.
Must be an odd integer. Default = %i.
''' % n_default

v_help = 'Set verbosity level. 0 = errors only, 1 = verbose, 2 = debug.'
```

```
# Exit values
EXIT_SUCCESS = 0
LAPACK_FAILED = 1

# Polarization constants
S_POLARIZATION = 'senkrecht'
P_POLARIZATION = 'parallel'

ERRORS_ONLY = 0
VERBOSE = 1
DEBUG = 2
VERBOSITY_VALUES = (ERRORS_ONLY, VERBOSE, DEBUG)

#######################
# Numerical constants # - working precision, pi, etc.
#######################
# Data types used throughout the calculations.
C_TYPE = numpy.complex64
F_TYPE = numpy.float32
I_TYPE = numpy.int32

# TODO make constants below into input arguments somehow???
# Command line args, input file, what?

# An even number. 2 * N = number of discretization points in x1s
N = 256
# Length of surface measured in $2\pi\lambda$. Pulled out of a hat.
L = 2 * numpy.pi * 20
# Max order of series expansion, i.e. max $j$ in $K_j$
J = 1

# [1], relative dielectric permittivity of the material
EPSILON_R = F_TYPE(2.25)
# [1], relative magnetic permeability of the material
MU_R = F_TYPE(1.0)
# Wave number in vacuum of incident radiation.
Q_LIGHT = F_TYPE(1.0)
# [1 / m], the maximum value of $q$ in our discretization. Adjust if needed.
# At too large or too small values, Rayleigh breaks down!
Q = F_TYPE(5.0 * Q_LIGHT)

####################
# Global variables # - keep these to a minimum...
####################
```

```
# Discretized variables: append an 's' to signify plural.
# Can hence use "for q in qs" in loops later. Convenient!
# Position along surface
x1s = numpy.linspace(-L / 2.0, L / 2.0, 2 * N).astype(F_TYPE)
# Surface height function. Use double precision (64bit float) as we want
# maximum precision in our FFT
zetas = numpy.zeros(2 * N, dtype=numpy.float64)
# Wave numbers of scattered and transmitted waves.
# p, q take on same values and both are read-only.
qs = numpy.linspace(-Q / 2.0, Q / 2.0, N + 1).astype(F_TYPE)
ps = qs
# Use same grid for incoming waves k as the q discretization.
# Store only indices of qs where k_3 is real.
k_indices = numpy.arange(qs.size, dtype=I_TYPE)[abs(qs) < 1.0]
# r = p - q
rs = numpy.linspace(-Q, Q, 2 * N + 1).astype(F_TYPE)
# alpha(q, omega) and alpha_0(q, omega) (tabulated)
alpha_table = numpy.zeros(qs.size, dtype=C_TYPE)
alpha_0_table = numpy.zeros(qs.size, dtype=C_TYPE)
# [m**(j + 1)], K_j[j] is the (tabulated) FFT of $zeta^j(x_1)$.
# Remember that j = 0 is included.
K_j = numpy.zeros((J + 1, rs.size), dtype=C_TYPE)
# Values for j and j! in summation over $K_j$, which sums to I.
js = numpy.arange(J + 1, dtype=I_TYPE)
factorial = lambda tmp: scipy.factorial(tmp, exact=True)
js_factorial = numpy.array(map(factorial, js), dtype=F_TYPE)

# Lookup tables
# Indices of r_{mn} for p_m - q_n; 16 bit int is big enough
r_index = numpy.zeros((N + 1, N + 1), dtype=numpy.uint16)

def fill_r_index():
    'Fill up the index lookup table for r.'
    for m in range(N + 1):
        for n in range(N + 1):
            r_index[m, n] = N + m - n

def alpha(q, epsilon_r, mu_r):
    'Return the alpha value (k_3 below interface) for the given epsilon.'
    # NOT TESTED
    tmp = C_TYPE(epsilon_r * mu_r - q**2)
    alpha = numpy.sqrt(tmp)
    if alpha.imag < 0:
        # Should have exponential decay, so need to flip sign.
        alpha *= -1
    return alpha

def alpha_0(q):
    '''
    Return the alpha_0 value (k_3 above interface).
    Should be opposite sign of alpha, as the wave is travelling in the
    opposite direction, i.e. should have exponential decay orthogonal to
    the interface.
    '''
    # NOT TESTED
    epsilon_r_vacuum = F_TYPE(1.0)
    mu_r_vacuum = F_TYPE(1.0)
    alpha_0 = alpha(q, epsilon_r_vacuum, mu_r_vacuum)
    return alpha_0

def fill_alpha_arrays():
    'Fill alpha array with correct values.'
    # alpha = numpy.zeros(q.size, dtype=C_TYPE)
    # alpha_0 = numpy.zeros(q.size, dtype=C_TYPE)
    alpha_table[:] = [alpha(q, EPSILON_R, MU_R) for q in qs]
    alpha_0_table[:] = [alpha_0(q) for q in qs]

def tabulate_K_j():
    'Tabulate the integral $K_j$ and save in an array.'
    # NOT TESTED
    # TODO: perform the fft using maximum precision, i.e. double?
    # (don't think quad is supported by fftpack!)
    for j in range(J + 1):
        fft_of_zetas = numpy.fft.fft(zetas**j)
        K_j[j, :-1] = numpy.fft.fftshift(fft_of_zetas)
    # Here: handle that FFT[Nyquist] = FFT[-Nyquist]
    K_j[:, -1] = K_j[:, 0]
    # Remember the prefactor coming from discretization
    delta_x1 = x1s[1] - x1s[0]
    prefactor = delta_x1
    K_j[:, :] *= prefactor

def I(m, n, plusminus):
    '''
    Calculates the integral I:
    $I(\alpha(p, \omega)) \mp \alpha_0(q, \omega) | p - q)$
    NB: plusminus is the sign between the alphas!
    '''
    # NOT TESTED
    im_unit = C_TYPE(1.0j)
    alpha_diff = alpha_table[m] + plusminus * alpha_0_table[n]
    prefactors = (-im_unit * (alpha_diff))**js / js_factorial

    r_i = r_index[m, n]
    assert r_i >= 0, 'Make sure r-index is positive.'
    assert r_i < 2 * N + 1, 'Make sure r-index is not too large.'
    integrals = K_j[:, r_i]
    terms_in_sum = prefactors * integrals

    return terms_in_sum.sum()

def kappa(polarization):
    'Returns the prefactor kappa for the given polarization.'
    if polarization == P_POLARIZATION:
        return EPSILON_R
    elif polarization == S_POLARIZATION:
        return MU_R
    else:
        assert False, 'Illegal value for polarization!'

def M(polarization, m, n, plusminus):
    '''
    Calculates the matrix element for the given polarization and sign.
    m and n are the indices in $p_m - q_n$.
    '''
    # NOT TESTED
    assert m >= 0 and n >= 0, 'Make sure indices are positive only.'
    assert m <= N + 1 and n < N + 1, 'Make sure indices are not too large.'
    msg = 'plusminus should be just a sign.'
    assert plusminus == -1 or plusminus == 1, msg

    numerator = (ps[m] - kappa(polarization) * qs[n]) * (ps[m] - qs[n])
    denominator = alpha_table[m] - plusminus * alpha_0_table[n]

    term_1 = numerator / denominator
    term_2 = alpha_table[m]
    term_3 = plusminus * kappa(polarization) * alpha_0_table[n]

    # NB! Sign in integral the opposite of the sign in M!
    integral = I(m, n, -plusminus)
    # prefactor = plusminus * (term_1 + term_2 + term_3)
    prefactor = term_1 + term_2 + term_3

    return prefactor * integral
```

```
def setup_LHS():
    '''Set up the coefficient matrix of the equation system,
    i.e. A in $A x = b$.'''

    # Arbitrary constant, needs to become an argument somehow
    polarization = S_POLARIZATION

    # Coefficient matrix (LHS) in A*x = b
    A = numpy.zeros((N + 1, N + 1), dtype=C_TYPE)
    for m in range(N + 1):
        for n in range(N + 1):
            A[m, n] = M(polarization, m, n, +1)

    # Weight end points of $q_n$ sum by half, i.e. use trapezoidal rule.
    A[:, 0] *= 0.5
    A[:, -1] *= 0.5

    # Prefactor for LHS: $\frac{\Delta q}{2 \pi}$
    delta_q = qs[1] - qs[0]
    n_pref = N # TODO I don't know where this comes from, but it is needed
for normalization... Suspect it is off by a little, as results are further
away from 0.
    prefactor = C_TYPE(delta_q * n_pref) / (2.0 * numpy.pi)

    return prefactor * A

def setup_RHS():
    '''Set up multiple Right Hand Sides in the equation system,
    i.e. b in $A x = b$.'''

    # Arbitrary constant, needs to become an argument somehow
    polarization = S_POLARIZATION

    # Assume k is on the "q-grid", i.e. k = ks[k_index].
    # This allows reuse of the tabulated integrals and whatnot.
    nk = len(k_indices)
    bs = numpy.zeros((N + 1, nk), dtype=C_TYPE)
    for m in range(N + 1):
        for k_i, k_index in enumerate(k_indices):
            bs[m, k_i] = M(polarization, m, k_index, -1)
    return bs

#############################################
# Below: general programming (not numerics) #
```

```
#############################################

def parse_options(argv):
    'Parse options given on the command line using the optparse module.'
    parser = optparse.OptionParser(usage=usage)
    parser.add_option('-n', type='int', dest='n', metavar='N',
default=n_default,
        help=n_help)
    parser.add_option('-v', '--verbose', type='int', dest='verbose',
        metavar='LEVEL', default=0, help=v_help)

    # Verify validity of options and arguments.
    # Put arguments into the options object.
    options, arguments = parser.parse_args(argv)
    if not options.verbose in VERBOSITY_VALUES:
        msg = '\n\tOption -v takes one of three values: %i, %i or %i.\n'
        msg += '\tHigher values makes the program more verbose.'
        parser.error(msg % VERBOSITY_VALUES)
    n_pos_args = len(positional_args.split())
    if not len(arguments) == n_pos_args:
        msg = '\n\t%i positional arguments are required: %s'
        msg = msg % (n_pos_args, positional_args)
        parser.error(msg)

    try:
        options.omega = float(arguments[0])
    except ValueError as e:
        msg = '\n\tOMEGA should be a floating point number: '
        msg += str(arguments[0])
        parser.error(msg)

    try:
        options.k = float(arguments[1])
    except ValueError as e:
        msg = '\n\tK should be a floating point number: '
        msg += str(arguments[1])
        parser.error(msg)

    options.outputfile = arguments[2]

    return options

def setup_arrays():
    'Perform filling of arrays and lookup tables.'
    setupfuncs = [fill_r_index, fill_alpha_arrays, tabulate_K_j]
    [f() for f in setupfuncs]

def check_errors():
    '''
    Check consistency of arguments, options, etc.
    Not sure if we need this.
    '''
    assert N % 2 == 0, 'N should be even for practical reasons.'
    # TODO add more consistency checks.
    # or TODO move them someplace more sensible?

def save_2D_array(a, var_name, dimensions, ncfile, datatype=Numeric.Float32):
    '''
    Save the 2D array *a* with *dimensions* as *var_name* in *ncfile*
    (a netCDF format file).
    '''
    file_var = ncfile.createVariable(var_name, datatype, dimensions)
    M, N = a.shape
    for m in range(M):
        for n in range(N):
            file_var[m, n] = a[m, n]

def save_1D_array(a, var_name, dimension, ncfile, datatype=Numeric.Float32):
    '''
    Save the 1D array *a* with *dimension* as *var_name* in *ncfile*
    (a netCDF format file).
    '''
    file_var = ncfile.createVariable(var_name, datatype, dimension)
    N = a.size
    for n in range(N):
        file_var[n] = a[n]

def save_result(R, polarization, filename):
    'Save the scattering amplitude R_\nu(q|k)'
    ncfile = NetCDF.NetCDFFile(filename, 'w')

    ncfile.polarization = polarization
    ncfile.epsilon_r = EPSILON_R
    ncfile.mu_r = MU_R

    N_qs_name = 'No._of_qs'
    ncfile.createDimension(N_qs_name, N + 1)
    N_k_name = 'No._of_k_indices'
    ncfile.createDimension(N_k_name, len(k_indices))
    N_zetas_name = 'Surf_disc._points'
    ncfile.createDimension(N_zetas_name, len(zetas))

    save_1D_array(qs, 'qs', (N_qs_name, ), ncfile)
    save_1D_array(zetas, 'zetas', (N_zetas_name, ), ncfile)
    dt = Numeric.Int32
    save_1D_array(k_indices, 'k_indices', (N_k_name, ), ncfile, datatype=dt)

    dims = (N_qs_name, N_k_name)
    # R_save = R[k_indices, :]
    R_save = R
    save_2D_array(R_save.real, 'R(q_m,k_n).real', dims, ncfile)
    save_2D_array(R_save.imag, 'R(q_m,k_n).imag', dims, ncfile)

    ncfile.close()

class Logger(object):
    'Convenience class for logging with verbosity.'
    def __init__(self, verbosity, stream=sys.stdout):
        super(Logger, self).__init__()
        assert verbosity in VERBOSITY_VALUES
        self.verbosity = verbosity
        self.stream = stream

    def log(self, *msg, **kwargs):
        '''Print log msg to the given stream. One legal **kwargs:
            vb = ERRORS_ONLY: Verbosity threshold
        '''
        if 'vb' in kwargs:
            vb = kwargs['vb']
        else: # Default
            vb = ERRORS_ONLY
        write = lambda m: self.stream.write(str(m))
        if self.verbosity >= vb:
            map(write, msg)
            write('\n')



def main(argv=None):
    'The main method. Execution begins here.'
    if not argv:
        argv = sys.argv[1:]
    arguments = parse_options(argv)
    logger = Logger(arguments.verbose)

    logger.log('Command line arguments:', arguments, '\n', vb=DEBUG)
    logger.log('Filling lookup tables and arrays', vb=DEBUG)
    setup_arrays()
    logger.log('Checking for internal consistency', vb=DEBUG)
    check_errors()

    logger.log('Solving RRE with N = %5i' % N, vb=VERBOSE)

    # RHS
    logger.log('Setting up RHS vectors b in A x = b', vb=DEBUG)
    bs = setup_RHS()
    logger.log('RHS vectors complete.', vb=DEBUG)

    # LHS
    logger.log('Setting up LHS matrix...', vb=DEBUG)
    A = setup_LHS()
    logger.log('LHS complete.', vb=DEBUG)

    # Solve A x = b
    logger.log('Entering LAPACK equation solver routine', vb=DEBUG)
    # Call LAPACK routine to solve equation system $A x = b$. Docs:
    # print fl.cgesv.__doc__
    # http://www.netlib.org/lapack/complex/cgesv.f
    # TODO: lag if-setning som velger likningsloeser avh. av presisjon
    lu, piv, R, info = lapack.cgesv(A, bs)
    if info:
        print >> sys.stderr, 'Lapack failed to solve equation system.'
        print >> sys.stderr, 'Value of "info": %i' % info
        sys.exit(LAPACK_FAILED)
    logger.log('Lapack found a solution', vb=DEBUG)

    # Save results to file
    logger.log('Saving results to file', vb=DEBUG)
    save_result(R, S_POLARIZATION, arguments.outputfile)

    return EXIT_SUCCESS

if __name__ == '__main__':
    sys.exit(main())


# TODO LIST
a = """
[Check] - Units
- Investigate normalization of FFT
[Check] - Main method - put things into several subfunctions
```

```
[Check] - Several right hand sides
- lag if-setning som velger likningsloeser avh. av presisjon
- Testing the small funcs, alpha, etc
- Test program for correctness
    - Energy conservation
    - Flat interface: recreate Fresnel's reflection formula
    - Periodic interface
- Weight the RRE integral, i.e. use [*] something more "fancy" than the
simple
    box method. This modifies the equation system a bit.
    [*] NR, eq. 4.1.14, section "classical formulas for equally spaced
abscissas."
- Add options for s- and p-polarization; other inputs as well
- Change user interface, so that K is no longer an input (we solve for all k
values in the propagating part of the q-grid anyway).
- Profiling
- Optimizing (most likely M, I are the two functions that are important)
- Fortran-implementering
"""
```