Numerical Recipes in Astrophysics

Assignment 1

Brendon Walter (s2078864)

April 9, 2019

Contents

1 Routines

I created a file called `routines.py` which contains various functions that are used throughout this exercise. The full output is seen in Appendix A.

The entire assignment is run through the script `handin1.py`. The relevant parts are shown in snippets throughout the rest of the file.

Listing 1: `handin1.py` : header

```
#!/usr/bin/env python3

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import time

import nur.routines as rt
from nur.RNG import RNG

random = RNG(11)
print('Random seed is', random.seed)


def write(path, text):
    ''' for easy printing and writing into a file '''
    print('\t%s' %text)
    with open(path, 'w') as f:
        f.write(str(text))
```

1.1 Poisson Distribution

This function implements the Poisson distribution, defined as

$$P(k) = \frac{\mu^k e^{-\mu}}{k!} \tag{1}$$

Listing 2: `poisson.py`

```
#!/usr/bin/env python3

import numpy as np

import nur.routines as rt


def poisson(mu, k):
    '''
    Calculate the value k for a Poisson distribution with mean mu.

    Parameters
    ----------
    mu : unsigned np.float64
        The mean of the distribution
    k : np.int64
        The number of times an event occurs (should be >0)
    '''
    if mu < 0:
        raise ValueError('Negative value passed for mu')
    if k < 0:
        return 0

    mu = np.float64(mu)
    k = np.int64(k)

    return (mu**k * np.exp(-mu)) / rt.prod(range(1, k+1))
```

Listing 3: `handin1.py` : 1a

```
from nur.poisson import poisson

s = ''
for mu, k in [(1, 0), (5, 10), (3, 21), (2.6, 40)]:
    s += 'P(%s, %s) = %.6f\t' %(mu, k, poisson(mu, k))

write('output/1a_poisson.txt', s)
```

```
P(1, 0) = 0.367879  P(5, 10) = 0.018133 P(3, 21) = 0.000000 P(2.6, 40) =
    0.000000
```

1.2 Random Number Generator

Here, I create a random number generator that uses a mix of multiplicative linear congruential generation:

$$x_{j+1} = ax_j + c \mod m \tag{2}$$

and a 64-bit XOR-shift:

$$\begin{aligned} x_{j+1} &= x_j \wedge (x_j \ll a) \\ x_{j+1} &= x_j \wedge (x_j \gg b) \\ x_{j+1} &= x_j \wedge (x_j \ll c) \end{aligned} \tag{3}$$

I use the suggested values for the multipliers from the textbook.

When called with `RNG.rand()`, the generator generates a number between 0 and 1. An additional function, `RNG.rand_range()` allows the user to generate a float between two values.

Typically usage of the generate uses the current Unix time in seconds as the starting seed. This ensures that it is as random as possible on start. For the purpose of this assignment, where I want the output to be reproducible, I assign a starting seed (see 1).

In order to ensure that the same generator is used throughout the assignment, I create a singleton class from which the RNG inherits.

Figure 1 below shows the output of the first 1,000 and 1,000,000 randomly generated numbers. It seems to perform well.

Listing 4: `RNG.py`

```
#!/usr/bin/env python3

'''
Usage: At beginning of program, add the line:

    from nur.RNG import random

This will create an instance of the RNG using the current unix time as
the starting seed. Generating a random number is then as easy as calling

    random.rand()

To define your own seed, use

    from nur.RNG import RNG
    random = RNG(seed)
    random.rand()
'''

import time


class Singleton:
    '''
    Implementation of singleton class comes from Alex Martelli's 'Borg':
    http://www.aleax.it/Python/5ep.html
    '''
    _shared_state = {}
    def __init__(self):
        self.__dict__ = self._shared_state
```

```
class RNG(Singleton):

    def __init__(self, seed=1):
        ''' Random Number Generator '''
        Singleton.__init__(self)

        if seed <= 0:
            raise ValueError('Starting seed must be greater than 0')

        self.seed = int(seed)
        self.state = int(seed)
        self.max = (2**64)-1   # maximum value (64 bits)


    def rand(self):
        '''
        Generate a random value between 0 and 1 using a combination
        of a multiplicative linear congruential generator and a
        64-bit XOR-shift.

        Values for the multipliers taken from Press et al. (2007)
        '''

        # MLCG
        a = 3935559000370003845
        c = 2691343689449507681

        self.state = (a * self.state + c) % self.max

        # XOR shift
        self.state ^= (self.state >> 21) & self.max
        self.state ^= (self.state << 35) & self.max
        self.state ^= (self.state >> 4)  & self.max

        return self.state / self.max


    def rand_range(self, lower, upper):
        '''
        Generate a random number within a range

        '''
        return lower + (random.rand() * (upper-lower))


# random seed from current unix time
random = RNG(time.time())
```

Listing 5: `handin1.py` : 1b

```
rand = [random.rand() for _ in range(int(1e6+1))]
x, y = rand[:-1], rand[1:]

# scatter plot
plt.figure(figsize=(4,4))
plt.scatter(x[:1000], y[:1000], alpha=.5, marker='.')
plt.savefig('output/1b_RNG-scatter.png')
plt.clf()

# 2d histogram
plt.figure(figsize=(5,4))
im = plt.hist2d(x, y, bins=20, cmap='Blues')
plt.colorbar(im[3])
plt.savefig('output/1b_RNG-hist2d.png')
plt.clf()
```

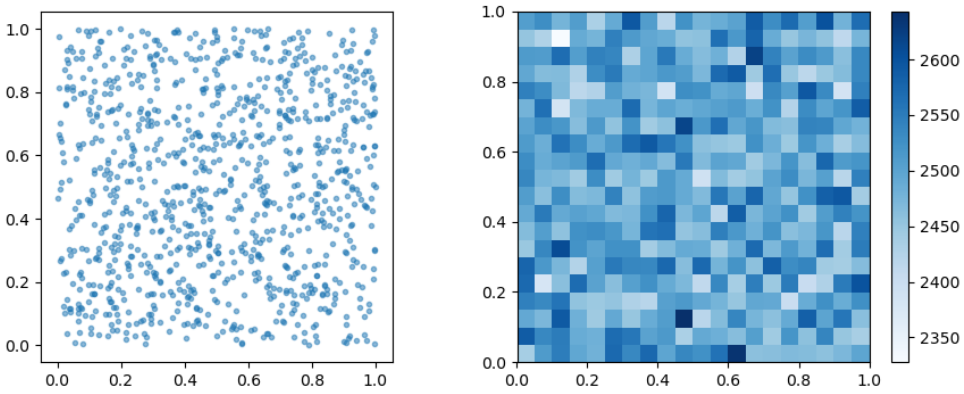


Figure 1: The first thousand (left) and million (right) pseudo random numbers from the above generator. The coordinates of each point are given as (x_j, x_{j+1})

2 Satellite Galaxy Haloes

In this problem, we create a number density profile for satellites galaxies in a dark matter halo. The profile is defined as

$$n(x) = A \langle N_{\text{sat}} \rangle \left(\frac{x}{b}\right)^{a-3} \exp\left[-\left(\frac{x}{b}\right)^{c}\right] \tag{4}$$

I created class called `GalaxyProfile` for easy usage of this profile:

Listing 6: `gal_profile.py`

```
#!/usr/bin/env python3

import numpy as np

from nur.RNG import random
import nur.integration as integ
import nur.derivation as deriv


class GalaxyProfile:

    def __init__(self, Nsat=100, a=None, b=None, c=None):
        '''
        Create a number density profile of satellite galaxies in a
        dark matter halo:

            n(x) = A <Nsat> (x/b)^(a-3) exp[-(x/b)^c]

        where x is the radius relative to the virial radius x = r/r_vir

        If the free parameters a, b, and c are not provided, random
            values are
        assigned.

        Parameters
        ----------
        Nsat : int (optional)
            The number of galaxies
        a : float (optional)
            Controls the small-scale slope
        b : float (optional)
            Controls the transition scale
        c : float (optional)
            Controls the steepness of the exponential dropoff
        '''
        self.a = a if a is not None else random.rand_range(1.1, 2.5)
        self.b = b if b is not None else random.rand_range(0.5, 2)
        self.c = c if c is not None else random.rand_range(1.5, 4)

        self.Nsat = Nsat
        self.A = self.normalize()


    def normalize(self, Nsat=None):
        '''
        Find the normalization factor A such that

            4 pi \int_0^5 n(x) x^2 = <Nsat>
        '''
        if Nsat is None:
```

```
            Nsat = self.Nsat

        self.A = self.Nsat / integ.spherical_integration(self.dist, 1e-4,
             5)
        return self.A


    def dist(self, x):
        ''' unormalized distriubtion '''
        return (x/self.b)**(self.a-3) * np.exp(-(x/self.b)**self.c)

    def logdist(self, x):
        ''' log of the unormalized distribution '''
        return np.log10(self.dist(x))

    def n(self, x):
        ''' the normalized number density profile '''
        return self.dist(x) * self.A

    def logn(self, x):
        ''' the log of the normalized number density profile '''
        return self.logdist(x) * self.A

    def dn(self, x):
        ''' the derivative of the profile at radius x '''
        return deriv.central(self.n, x)

    def probability(self, x):
        '''
        the probability distribution

            p(x) dx = 4 pi x^2 n(x) dx
        '''
        return 4 * np.pi * x**2 * self.n(x) / self.Nsat


    def __str__(self):
        return 'Nsat=%s      a = %.4f      b = %.4f      c = %.4f     A = %.4
            f' \
                %(self.Nsat, self.a, self.b, self.c, self.A)
```

The integration and derivation routines can be found in Appendices B and C respectively. More detail is also given in the following sections.

2.1 Random values for a, b, and c

When calling `GalaxyProfile` without specifying the free parameters a, b, and c, the class automatically generates random numbers for these in the range specified by the assignment sheet.

The class then finds the normalization factor A using a spherical integrator, which in turn uses Simpson integration. While I did also implement a trapezoidal integrator (see Appendix B), timing tests showed that Simpson was slightly faster for spherical integration.

Listing 7: `integration.py` : simpson, spherical

```
def simpson(f, a, b, N=10000000):
    '''
    Integral approximation via Simpson's rule
    '''
    h = (b-a)/N

    x = np.linspace(a, b, N)
    y = f(a) + f(b)

    y += np.sum(2*f(x[2::2]))
    y += np.sum(4*f(x[1::2]))

    return (h*y)/3


def spherical_integration(f, a, b):
    '''
    Integrate a function over 3 dimensions in spherical coordinates

        \int \int \int_V f(r) dV

    where dV = r^2 \sin \phi d\phi d\theta

    Thus

        4 \pi \int_a^b f(r) r^2
    '''

    # timing shows that simpson tends to be faster
    return 4 * np.pi * simpson(lambda r: f(r)*r**2, a, b)
```

Printing the object then shows the values for the generated parameters:

Listing 8: `handin1.py` : 2a

```
from nur.gal_profile import GalaxyProfile

dist = GalaxyProfile()
write('output/2a.txt', dist.__str__())
```

```
Nsat=100     a = 1.6630     b = 0.8805     c = 1.8560     A = 20.1881
```

2.2 Interpolation

I used two different interpolators – linear and polynomial via Neville's algorithm – to attempt this problem. The full code (including the `bisect()` function used by `linear()`) is available in Appendix D.

Listing 9: `interpolation.py` : linear

```
def linear(x, y, xinterp):
    '''
    Linear interpolation

    Parameters
    ----------
    x, y : float, float
        The known data points in x and y
    xinterp : list (1D)
        The list of x values to be interpolated

    Returns
    -------
    yinterp : list (1D)
        An array holding the values of the interpolated y
        values for each xinterp passed to the function
    '''

    x, y = np.array(x), np.array(y)
    xinterp = np.sort(xinterp)
    yinterp = []

    for xval in xinterp:

        #find the neighboring points
        xlow, xhigh = bisect(x, xval)

        # get the index of the neighboring points
        ilow =  rt.find_closest(x, xlow)[0]
        ihigh = rt.find_closest(x, xhigh)[0]

        # get y values of the neighboring points
        ylow, yhigh = y[ilow], y[ihigh]

        slope = (yhigh - ylow) / (xhigh - xlow)

        yinterp.append(slope * (xval - xhigh) + yhigh)

    return np.array(yinterp)
```

Listing 10: `interpolation.py` : polynomial

```
def polynomial(x, y, xinterp):
    '''
    Polynomial interpolation via Neville's algorithm

    Parameters
    ----------
    x, y : float, float
        The known data points in x and y
    xinterp : list (1D)
        The list of x values to be interpolated
```

```
    Returns
    -------
    yinterp : list (1D)
        An array holding the values of the interpolated y
        values for each xinterp passed to the function
    '''
    M = len(x)
    yinterp = []
    for xval in xinterp:
        P = np.zeros((M, M))
        P[:, 0] = y
        for i in range(1, M):
            for j in range(1, i+1):

                A = (xval - x[i-j]) * P[i,   j-1]
                B = (xval - x[i])   * P[i-1, j-1]
                C = (x[i] - x[i-j])

                P[i,j] = (A - B) / C

        yinterp.append(P[-1,-1])

    return yinterp
```

Listing 11: handin1.py : 2b

```
import nur.interpolation as interp

xmin, xmax = 1e-4, 5

# true values for x and y
truex = np.linspace(xmin, xmax, 100)
logtruex = np.log10(truex)
logtruey = dist.logn(truex)

# the data points
datax = np.array([1e-4, 1e-2, 1e-1, 1, 5])
logdatax = np.log10(datax)
logdatay = dist.logn(datax)

# the x points to be interpolated
xinterp = np.linspace(xmin, xmax, 100)
logxinterp = np.log10(xinterp)

# interpolation via two different methods
loglinear = interp.linear(logdatax, logdatay, logxinterp)
logpoly = interp.polynomial(logdatax, logdatay, logxinterp)

# plot results
plt.plot(logtruex, logtruey, lw=5, alpha=.3, label='true')
plt.scatter(logdatax, logdatay, c='k', marker='x', label='data')

plt.plot(logxinterp, loglinear, lw=2, linestyle='dashed', label='linear')
plt.plot(logxinterp, logpoly, lw=2, linestyle='dotted', label='polynomial
    ')

plt.xlabel('$\log_{10}(x)$')
plt.ylabel('$\log n(x)$')
```

```
    plt.legend(loc='lower left')
    plt.savefig('output/2b_interpolation.png')
    plt.clf()
```

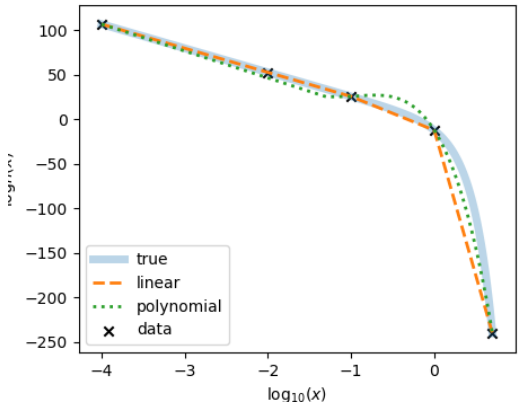


Figure 2: Interpolation between data points using two different interpolation schemes.

Both methods seem roughly comparable for the 100 interpolated points, however as `len(xinterp)` increases, `polynomial()` performs worse. I interpret this to mean that the fewer the points, the more `polynomial()` approximates `linear()` and performs better. This means that `linear` is the more appropriate interpolator for this problem.

2.3 Derivation

I used the central difference formula to find the derivative of Equation 4 at the generated value for b.

This was implemented as a method `dn()` for the `GalaxyProfile` class.

Listing 12: `derivation.py` : central

```
def central(f, x, dx=1e-6):
    '''
    derivation via the central differences formula

    TODO: error estimation (eps = h^2 f'")
    '''
    return (f(x+dx) - f(x-dx))/(2*dx)
```

Listing 13: `handin1.py` : 2c

```
write('output/2c_derivative.txt', dist.dn(dist.b))
```

```
-26.932839939153297
```

I did not get a chance to calculate the error or compare the numerical derivation to the analytical value.

2.4 Galaxy position generation in 3D

There are two parts to this problem: First for sampling the radius and second for the angles.

I sample the probability distribution,

$$P(x)dx = n(x) \cdot 4\pi x^2 dx / \langle N_{sat} \rangle \tag{5}$$

using rejection sampling. Although this method is often slow, I found that this function was broad enough that the number of attempts to sample from the distribution was still reasonable.

Listing 14: `sampling.py` : rejection

```
def rejection(f, n, xrange=(0,1), yrange=(0,1)):
    '''
    Rejection sampling

    Parameters
    ----------
    f : function
        The function being sampled
    n : int
        The number of samples to draw
    xrange : (float, float) (default: (0,1))
        The range of x values to sample from
    yrange : (float, float) (default: (0,1))
        The range of y values to sample from

    Returns
    -------
    values : array (N, 2)
        An array with the (x, y) sampled pairs. The length
        N <= n is the number of samples in which y < f(x)
    '''

    values = []

    while len(values) < n:
        x = random.rand_range(xrange[0], xrange[1])
        y = random.rand_range(yrange[0], yrange[1])

        if y < f(x):
            values.append((x,y))

    return np.array(values)
```

The next step is to sample θ and ϕ from a spherical distribution.

Listing 15: `sampling.py` : spherical

```
def spherical(n):
    '''
    Sample values randomly from the surface of a sphere
    '''

    values = []

    for _ in range(n):
        theta = 2 * np.pi * random.rand()
        phi = np.arccos(1 - 2*random.rand())
        values.append((theta, phi))

    return np.array(values)
```

With these two sampling functions defined, a set a box around the function and sample from within the box to find x. For each x value found, I then randomly generate values for θ and ϕ.

Listing 16: `handin1.py` : 2d

```
import nur.sampling as samp

# sample along r
r = samp.rejection(dist.probability, 100, xrange=(xmin, xmax), yrange=(0,
    4)).T

plt.plot(truex, dist.probability(truex), lw=5, alpha=.8)
plt.scatter(r[0], r[1], marker='.')

plt.xlabel('x')
plt.ylabel('P(x)')

plt.savefig('output/2d_r-samples.png')
plt.clf()


# sample theta and phi
angles = samp.spherical(100).T
theta, phi = angles[0], angles[1]

# generate a table of (r ,theta, phi) for 100 galaxies
galaxies = np.array([r[0], theta, phi])
galaxies = pd.DataFrame(galaxies.T, columns=['r', 'theta', 'phi'])
galaxies.to_csv('output/2d_galaxies.csv', sep='\t')
```

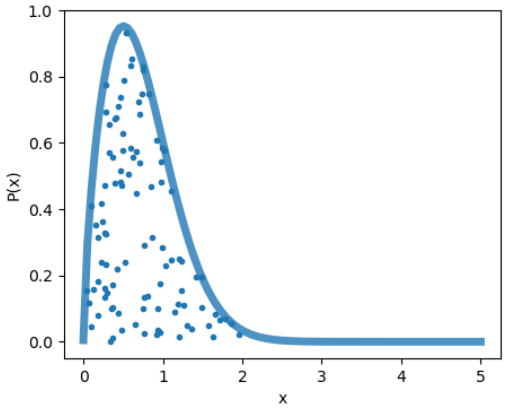


Figure 3: Sampling P(x) with rejection sampling

Below is the first ten of the 100 galaxies generated. The full list can be seen in Appendix F

Listing 17: `galaxies.csv`

```
     r    theta    phi
0    0.3947025909536362  5.539505336075609    1.6656697259710946
1    0.0916921519764658  3.3564521637306264   1.050470897092962
2    0.6893474165664725  2.384514000055732    2.070754654628161
3    0.6703353256854144  2.742781445160191    1.318429693400268
4    0.2291229261130093  1.6598071453836898   2.0269797198539927
5    0.9379558849557447  5.634314926915802    0.8999076415908493
6    0.33927296093340104 2.316001267928581    1.028554516024993
7    0.2618143634625172  3.8858383938546264   2.137708252912163
8    0.8285449303755207  0.29565695934626446  1.9555689903251363
9    1.4848998129388358  2.580448432275089    0.5703046807813118
```

2.5 1000 haloes

Here, I create 1000 haloes with 100 satellites each, drawing x from the distribution:

$$N(x) = n(x) \cdot 4\pi x^2 \tag{6}$$

I save these values in a (1000x3x100) datacube, which I then use to average all 1000 haloes in x to create a histogram.

Listing 18: `handin1.py` : 2e

```
N = lambda x: dist.probability(x)*dist.Nsat

# sample 1000 haloes
Nsat = 100
Nhalo = 1000
halos = np.ndarray((Nhalo, 3, Nsat))

for h in range(Nhalo):
    r = samp.rejection(dist.probability, Nsat,
                       xrange=(xmin, xmax), yrange=(0, 4)).T
    theta, phi = samp.spherical(Nsat).T
    galaxies = np.array([r[0], theta, phi])
    halos[h, :, :] = galaxies


# create a table of histograms for each of the 1000 halos
nbins = 20
bins = np.log10(np.logspace(np.log10(xmin), np.log10(xmax), nbins+1))

halo_hist = np.ndarray((Nhalo, nbins))
for h, halo in enumerate(halos):
    logr = np.log10(halo[0])
    halo_hist[h, :]  = np.histogram(logr, bins)[0]


# plot the function and histogram
plt.plot(np.log10(truex), N(truex), lw=5, alpha=.8)
plt.bar(bins[:-1], rt.mean(halo_hist), .2, alpha=.5, color='C1')

plt.yscale('log')
plt.ylim([1e-3, 1e4])

plt.xlabel('log$_{10}(x)$')
plt.ylabel('$N(x)$')

plt.savefig('output/2e_average-r.png')
plt.clf()
```

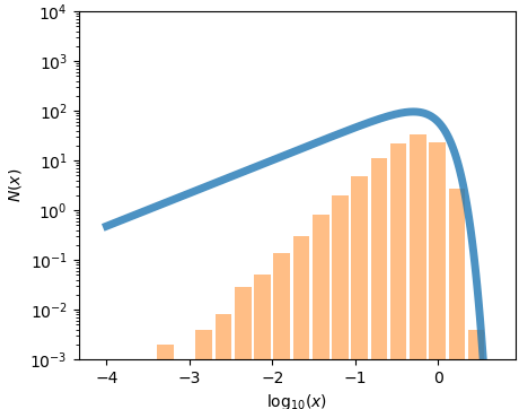


Figure 4: Average values of x for 1000 haloes compared to $N(x)$

2.6 Minimization and root finding

In order to find the solutions of $N(x) = N_{max}/2$, I first found the function maximum using a golden ratio search. This method quickly converges if the initial bracket is small, however it is difficult to generalize for this problem as if the bracket is too large it is very slow to converge (and often does not find the maximum at all).

Listing 19: `minimization.py`

```
#!/usr/bin/env python3

import numpy as np

import nur.routines as rt


def golden(f, a, b, eps=1e-8):
    '''
    Find the minimum of a function using golden function search

    This algorithm was taken from wikipedia
    https://en.wikipedia.org/wiki/Golden-section_search#Algorithm

    (sorry if that's not allowed - I really don't know how I can change
        this so
    it's not just the exact same algorithm)
    '''

    # ensure that f(b) < f(a)
    assert(a != b)
    if f(a) < f(b):
        a, b = rt.switch(a, b)

    # find two new points between c and d
    golden_ratio = 1.618
    c = b - (b-a) / golden_ratio
    d = a + (b-a) / golden_ratio

    # iteratively tighten the bracket around the function minimum
    while abs(c-d) > eps:
        if f(c) > f(d):
            b = d
        else:
            a = c

        c = b - (b-a) / golden_ratio
        d = a - (b-a) / golden_ratio

    # minimum is in the middle of the bracket
    return (b+a)/2
```

Next, I shift the function down by $N_{max}/2$ in order to use root-finding to find the x-values. The algorithm I use is bisection, which iteratively checks if the root is between two points and then tightens the bracket.

Listing 20: `root_finding.py`

```
#!/usr/bin/env python3

import nur.routines as rt
```

```
def bisect(f, a, b, eps=1e-4, maxiter=50):
    '''
    Find the root via bisection

    Parameters
    ----------
    f : function
        The function for which the roots are found
    a : float
        x value to the left of the root
    b : float
        y value to the right of the float
    eps : float (default: 1e-4)
        Stop once the difference between `a` and `b` becomes smaller than
        this value
    maxiter : int (default: 50)
        Stop after this many iterations if the method fails to converge
        fast enough

    Returns
    -------
    c : float
        The location on the x axis where f(x) = 0
    '''

    # ensure there is a root between a and b
    assert(f(a)*f(b) < 0)

    # ensure a is less than b
    assert(a != b)
    if a > b:
        a, b = rt.switch(a, b)

    delta = abs(b-a)

    i = 0
    while (delta > eps) and i<=maxiter:

        # calculate midpoint between a and b
        c = a + delta/2

        if f(a)*f(c) < 0:
            b = c  # root is between a and c
        elif f(b)*f(c) < 0:
            a = c  # root is between b and c

        # update delta to the new values of a and b
        delta = abs(b-a)

        i += 1

    return c
```

Listing 21: `handin1.py : 2f`

```
import nur.minimization as minim
import nur.root_finding as rf

# find the function maximum
```

```
Nmaxx = minim.golden(lambda x: -N(x), .1, 1)
Nmaxy = N(Nmaxx)

# plot the location of N(y/2) and y/2
plt.plot(truex, N(truex), lw=5, alpha=.8, label='true')
plt.axvline(Nmaxx, c='k', linestyle='dashed', lw=2)
plt.axhline(Nmaxy/2, c='k', linestyle='dotted', lw=2)

plt.xlabel('x')
plt.ylabel('N(x)')

plt.savefig('output/2f_max.png')
plt.clf()


# find N(x) = y/2
Nshift = lambda x: N(x)-Nmaxy/2
r1 = rf.bisect(Nshift, 1e-4, Nmaxx)
r2 = rf.bisect(Nshift, Nmaxx, 5)

write('output/2f_roots.txt', 'r1: %s\tr2 %s' %(r1, r2))

# plot location of the roots
plt.plot(truex, N(truex), lw=5, alpha=.8, label='true')
plt.axhline(Nmaxy/2, c='k', linestyle='dotted', lw=2)

plt.axvline(r1, c='k', linestyle='dashed', lw=2)
plt.axvline(r2, c='k', linestyle='dashed', lw=2)

plt.xlabel('x')
plt.ylabel('N(x)')

plt.savefig('output/2f_roots.png')
plt.clf()
```

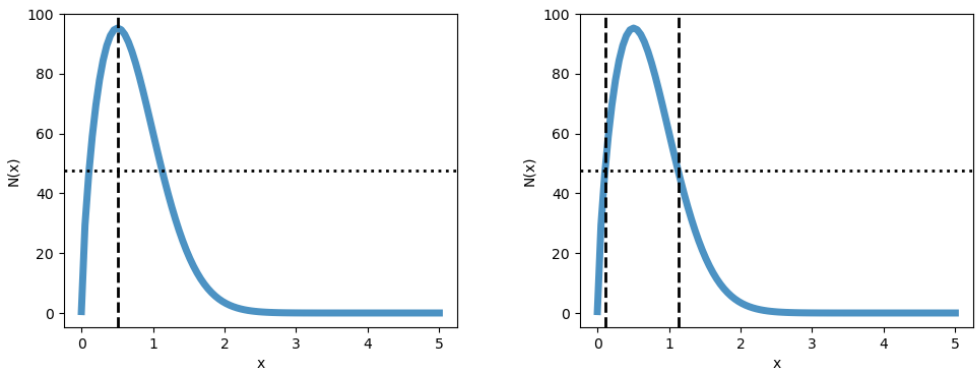


Figure 5: The location of the maximum and $N_{max}/2$ (left) and the solutions to the roots (right).

Listing 22: `2f_roots.txt`

```
r1: 0.10694445820910341 r2 1.1272565023846253
```

2.7 Sorting and comparison with Poisson distribution

In order to sort the values in the largest bin, I first attempted to do a heap sort. I was able to successfully put a list of around 10 values into a heap, however once I tried it with the 1000 values it failed to sort it correctly. My attempt can be seen in Appendix G.

Failing the heap sort, I opted to use quick sort instead, which recursively choices a pivot point and sorts the values to the left and then to the right.

Once the values are sorted, I find the median, the 16th and 84th percentiles, and finally I put the galaxies in the max radial bin into a histogram and compare it to the Poisson distribution.

Listing 23: `sorting.py` : quick sort

```
def quick_sort(values, _low=None, _high=None):
    '''
    Sort an array using the 'quick sort' method. This recursively chooses
        a
    pivot element and sorts all values on the left and the right of the
        pivot
    with swapping.

    Sorting is done in place. Pass a copy of the values if you want to
        keep the
    original ordering.

    Parameters
    ----------
    values : array-like (1D)
        The array of values to be sorted
    '''
    _low = 0 if _low is None else _low
    _high = len(values)-1 if _high is None else _high

    if _low < _high:
        pivot = values[_low]
        mid = _low
        for curr in range(_low+1, _high+1):
            if values[curr] < pivot:
                mid += 1
                values[mid], values[curr] = rt.switch(values[mid], values
                    [curr])

        values[_low], values[mid] = rt.switch(values[_low], values[mid])

        quick_sort(values, _low, mid-1) # left half
        quick_sort(values, mid+1, _high) # right half
```

Listing 24: `handin1.py` : 2g

```
import nur.sorting as sort

# find the maximum bin
max_bin = np.argmax(rt.mean(halo_hist))
ngals = halo_hist[:, max_bin]

# sort galaxies in the radial bin
sort.quick_sort(ngals)

# find median and percentiles
median = rt.median(ngals)
```

```
p16, p84 = rt.percentile(ngals, 0.16), rt.percentile(ngals, 0.84)

write('output/2g_median-percentiles.txt',
      'Median: %s\t16th Percentile: %s\t84th Percentile: %s' \
      %(median, p16, p84))


# plot poisson distribution and histogram
x = np.arange(ngals[0]-5, ngals[-1]+5)
plt.plot(x, [poisson(rt.median(ngals), k) for k in x], lw=5, alpha=.8)

ngals_hist, ngals_bins = np.histogram(ngals, bins=x)
plt.bar(ngals_bins[:-1], ngals_hist/len(ngals), alpha=.5, color='C1')

# plot median and percentile lines
plt.axvline(rt.median(ngals), c='k', lw=2, linestyle='dashed', label='
    median')
plt.axvline(rt.percentile(ngals, .16), c='k', lw=2, linestyle='dotted',
    label='16th')
plt.axvline(rt.percentile(ngals, .84), c='k', lw=2, linestyle='dotted',
    label='84th')

plt.xlabel('Number of galaxies')

plt.savefig('output/2g_ngal.png')
```

```
Median: 33.0    16th Percentile: 28.0    84th Percentile: 38.0
```

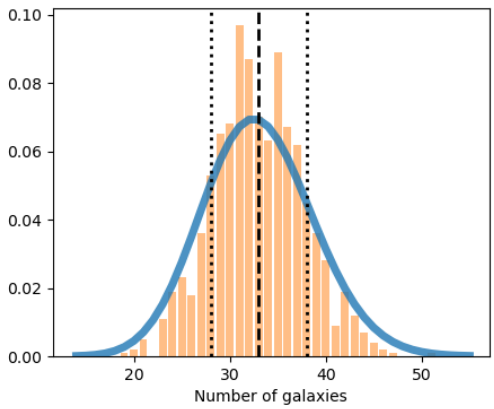


Figure 6: The number of galaxies in the maximum radial bin for 1000 haloes as compared to the Poisson distribution. The vertical lines show the median and 16th and 84th percentiles. As can be seen, the distribution of galaxies follows the Poisson distribution quite well.

2.8 Finding A

I was able to find all 6240 values of A however I was not able to write the 3D interpolator. This method ended up being quite slow – it takes around 1.2 seconds to find A for a single realization of a, b, and c, but with 6240 values this ended up taking 2 hours to run. I saved the output in a file which then can be loaded for future use.

Listing 25: `handin1.py` : 2f

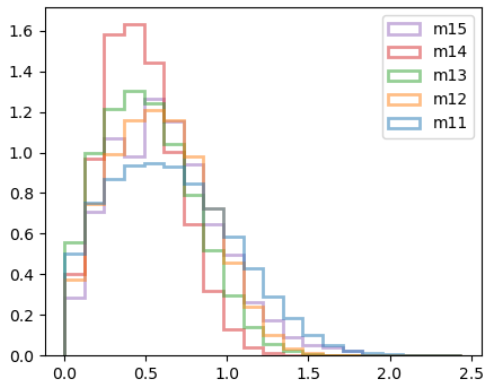
```
from scipy.interpolate import RegularGridInterpolator

step_size = 0.1
a_range = np.arange(1.1, 2.5+step_size, step_size)
b_range = np.arange(0.5, 2+step_size, step_size)
c_range = np.arange(1.5, 4+step_size, step_size)

A_table = np.ndarray((len(a_range), len(b_range), len(c_range)))

try:
    A_table = np.load('output/A_table.npy')
except FileNotFoundError:
    print("!!! THIS SHOULDN'T HAVE TO RUN BECAUSE IT TAKES 2 HOURS !!!")
    step_size = 0.1
    a_range = np.arange(1.1, 2.5+step_size, step_size)
    b_range = np.arange(0.5, 2+step_size, step_size)
    c_range = np.arange(1.5, 4+step_size, step_size)

    A_table = np.ndarray((len(a_range), len(b_range), len(c_range)))
    for i, a in enumerate(a_range):
        print(i)
        for j, b in enumerate(b_range):
            for k, c in enumerate(c_range):
                A_table[i, j, k] = GalaxyProfile(a=a, b=b, c=c).A

    np.save('output/A_table', A_table)

fn = RegularGridInterpolator((a_range, b_range, c_range), A_table)
```

3 Galaxy data from files

I did not have time to complete the problem. All I was able to accomplish was reading in and parsing the files, and showing a histogram of the different mass profiles.

Listing 26: `handin1.py : 3`

```
print('3 parsing data files')

haloes = {}
for mass in [11, 12, 13, 14, 15]:
    data = []
    with open('data/satgals_m%s.txt' %mass, 'r') as f:
        for line in f.readlines()[4:]:
            if line.startswith('#'): continue
            data.append(line)

    for i, val in enumerate(data):
        data[i] = np.array(val.strip().split(), dtype='float')

    data = np.array(data)

    m = 'm%s' %mass
    haloes[m] = {}
    haloes[m]['r'] = data[:,0]
    haloes[m]['theta'] = data[:,1]
    haloes[m]['phi'] = data[:,2]


plt.hist([haloes['m11']['r'], haloes['m12']['r'], haloes['m13']['r'],
    haloes['m14']['r'], haloes['m15']['r']],
        histtype='step', lw=2, alpha=.5, label=['m11', 'm12', 'm13', 'm14
            ', 'm15'], density=True, bins=20)

plt.legend()
```



A routines.py

Listing 27: routines.py

```
#!/usr/bin/env python3

import numpy as np


def switch(a, b):
    '''
    Switch the values of a and b

    Ex:
        a=1; b=2
        a, b = switch(a, b)
        print(a) # 2
        print(b) # 1
    '''
    return b, a


def mean(array):
    '''
    Calculate the mean of a 1D array
    '''
    return sum(array) / len(array)


def median(array):
    '''
    Calculate the median of a 1D array

    Note: the array should be sorted
    '''
    return percentile(array, .5)


def percentile(array, k):
    '''
    Find the 'k'th percentile of a 1D array.

    Note: the array should be sorted
    '''
    assert(k>=0 and k<=1)
    index = len(array)*k
    if index % 1 == 0.:          # if not a whole number
        index = int(index)+1     # round up to the nearest whole number
        return array[index]

    return mean([array[index], array[index+1]])


def std(array):
    '''
    Calculate the standard deviation of a 1D array
    '''
    array = np.asarray(array)
    numer = sum((array - mean(array))**2)
    denom = len(array)
```

```
    return (numer / denom)**.5


def prod(array):
    '''
    Calculate the product of an array

        prod = x_i * x_{i+1} * ... * x_{N}
    '''
    total = 1
    for value in array:
        total *= value

    return total


def factorial(k):
    '''
    Calculate the factorial

        k! = k * (k-1) * ... * 1
    '''
    return prod(range(1, k+1))


def find_closest(points, value):
    '''
    Find the index and nearest value in an array

    Example
    -------
    points = [1, 5, 10, 2, 8]
    value = 2.3
    print(find_closest(points, value))
    # (3, 2)
    # 2 is the value closest to 2.3 and is at index 3

    Parameters
    ----------
    points : list-like
        list of values to search
    value : float
        value to find in points

    Returns
    -------
    (index, closet_val) : (int, float)
        The index and closest value found in the array
    '''

    index = np.argmin(np.abs(np.array(points) - value))
    closest_val = points[index]

    return (index, closest_val)
```

B integration.py

Listing 28: integration.py

```
#!/usr/bin/env python3

import numpy as np


def trapezoid(f, a, b, N=10000000):
    '''
    Integral approximation via the trapezoid rule
    '''
    h = (b-a)/N
    x = np.linspace(a, b, N)

    y = (f(a) + f(b))/2
    y += np.sum(f(x))

    return h*y


def simpson(f, a, b, N=10000000):
    '''
    Integral approximation via Simpson's rule
    '''
    h = (b-a)/N

    x = np.linspace(a, b, N)
    y = f(a) + f(b)

    y += np.sum(2*f(x[2::2]))
    y += np.sum(4*f(x[1::2]))

    return (h*y)/3


def spherical_integration(f, a, b):
    '''
    Integrate a function over 3 dimensions in spherical coordinates

        \int \int \int_V f(r) dV

    where dV = r^2 \sin \phi d\phi d\theta

    Thus

        4 \pi \int_a^b f(r) r^2
    '''

    # timing shows that simpson tends to be faster
    return 4 * np.pi * simpson(lambda r: f(r)*r**2, a, b)


if __name__=='__main__':

    # running some tests
    print('Triple integral to approximate the volume of a sphere (R=5):')
    print('\tApprox\t', spherical_integrator(lambda r: 1, 0, 5))
    print('\tReal\t', (4/3) * np.pi * 5**3)
```

C derivation.py

Listing 29: derivation.py

```
#!/usr/bin/env python3

import numpy as np


def central(f, x, dx=1e-6):
    '''
    derivation via the central differences formula

    TODO: error estimation (eps = h^2 f'")
    '''
    return (f(x+dx) - f(x-dx))/(2*dx)


def ridders(f, x, dx=1e-6):
    '''
    derivation with Ridder's method

    TODO: do this
    '''

    pass
```

D interpolation.py

Listing 30: interpolation.py

```
#!/usr/bin/env

import numpy as np

import nur.routines as rt


def bisect(points, value):
    '''
    Use the bisection algorithm to find the closest points
    to a given value

    Parameters
    ----------
    points : list-like (1D)
        The list of the points from which you are interpolating
    value : float
        The point that you want interpolated

    Returns
    -------
    (left, right) : (float, float)
        The points closest to the interpolated point
    '''

    points = np.sort(points)
    n = len(points)

    left = points[:int(n/2)]
    right = points[int(n/2):]

    if value < left[0] or value > right[-1]:
        raise Exception('Value falls outside of given range')

    elif value < left[-1]: return bisect(left, value)
    elif value > right[0]: return bisect(right, value)
    else: return (left[-1], right[0])


def linear(x, y, xinterp):
    '''
    Linear interpolation

    Parameters
    ----------
    x, y : float, float
        The known data points in x and y
    xinterp : list (1D)
        The list of x values to be interpolated

    Returns
    -------
    yinterp : list (1D)
        An array holding the values of the interpolated y
        values for each xinterp passed to the function
    '''
```

```
    x, y = np.array(x), np.array(y)
    xinterp = np.sort(xinterp)
    yinterp = []

    for xval in xinterp:

        #find the neighboring points
        xlow, xhigh = bisect(x, xval)

        # get the index of the neighboring points
        ilow =  rt.find_closest(x, xlow)[0]
        ihigh = rt.find_closest(x, xhigh)[0]

        # get y values of the neighboring points
        ylow, yhigh = y[ilow], y[ihigh]

        slope = (yhigh - ylow) / (xhigh - xlow)

        yinterp.append(slope * (xval - xhigh) + yhigh)

    return np.array(yinterp)


def polynomial(x, y, xinterp):
    '''
    Polynomial interpolation via Neville's algorithm

    Parameters
    ----------
    x, y : float, float
        The known data points in x and y
    xinterp : list (1D)
        The list of x values to be interpolated

    Returns
    -------
    yinterp : list (1D)
        An array holding the values of the interpolated y
        values for each xinterp passed to the function
    '''
    M = len(x)
    yinterp = []
    for xval in xinterp:
        P = np.zeros((M, M))
        P[:, 0] = y
        for i in range(1, M):
            for j in range(1, i+1):

                A = (xval - x[i-j]) * P[i,   j-1]
                B = (xval - x[i])   * P[i-1, j-1]
                C = (x[i] - x[i-j])

                P[i,j] = (A - B) / C

        yinterp.append(P[-1,-1])

    return yinterp
```

E sampling.py

Listing 31: sampling.py

```
#!/usr/bin/env python3

import numpy as np

from nur.RNG import random


def rejection(f, n, xrange=(0,1), yrange=(0,1)):
    '''
    Rejection sampling

    Parameters
    ----------
    f : function
        The function being sampled
    n : int
        The number of samples to draw
    xrange : (float, float) (default: (0,1))
        The range of x values to sample from
    yrange : (float, float) (default: (0,1))
        The range of y values to sample from

    Returns
    -------
    values : array (N, 2)
        An array with the (x, y) sampled pairs. The length
        N <= n is the number of samples in which y < f(x)
    '''

    values = []

    while len(values) < n:
        x = random.rand_range(xrange[0], xrange[1])
        y = random.rand_range(yrange[0], yrange[1])

        if y < f(x):
            values.append((x,y))

    return np.array(values)


def spherical(n):
    '''
    Sample values randomly from the surface of a sphere
    '''

    values = []

    for _ in range(n):
        theta = 2 * np.pi * random.rand()
        phi = np.arccos(1 - 2*random.rand())
        values.append((theta, phi))

    return np.array(values)
```

F galaxies.csv

Listing 32: galaxies.csv

```
    r   theta   phi
0   0.3947025909536362  5.539505336075609   1.6656697259710946
1   0.0916921519764658  3.3564521637306264  1.050470897092962
2   0.6893474165664725  2.384514000055732   2.070754654628161
3   0.6703353256854144  2.742781445160191   1.318429693400268
4   0.2291229261130093  1.6598071453836898  2.0269797198539927
5   0.9379558849557447  5.634314926915802   0.8999076415908493
6   0.33927296093340104 2.316001267928581   1.028554516024993
7   0.2618143634625172  3.8858383938546264  2.137708252912163
8   0.8285449303755207  0.29565695934626446 1.9555689903251363
9   1.4848998129388358  2.580448432275089   0.5703046807813118
10  0.9616476661498307  6.010366605363992   2.113083673330574
11  1.103098034963741   4.622021388693754   0.9353477882461629
12  0.26556887281079655 2.484615406432123   2.0148250231763036
13  1.6528310964892319  5.104748410090186   2.2687742357014593
14  0.5886235432960951  4.333023917638118   1.0562462986556655
15  0.5935732457797236  3.1365188535077593  0.2129648394479579
16  0.8519900204547607  2.9617774482355466  2.266103554101457
17  0.9781475974600913  3.007985742605298   2.8182609690338323
18  0.7594885710731902  5.617882505487017   1.2606819715219015
19  1.2369049015866067  1.548305536170741   1.3860843648223435
20  0.7330913621468416  4.875700856713954   1.2510670456246946
21  1.0219358596232806  6.1598767673993295  1.2801873473188066
22  0.1857767595376985  1.8985958222721349  0.8593613577081629
23  0.9580330261553501  3.5793416126544244  0.6894705906154119
24  1.4878516102019645  5.1351290756045005  0.5033123659935352
25  0.42371737074338217 1.5224060273285012  2.6162427237840653
26  0.18912449717212526 5.0124714015017515  2.5957260511303404
27  1.5682992282852262  3.9185925042475476  1.5297564661722172
28  0.9814992130592566  2.3764437394381748  0.3347214778647096
29  0.04257835123029744 1.9483345817078856  2.853565193286381
30  0.4631801561840723  4.66841179860276    0.931134681268114
31  0.7067851515065505  3.8561704834366437  1.3320333301525276
32  1.7175875801849694  6.006338943251911   2.6066143454131714
33  0.6045457453850481  0.1924335913082727  1.3559584622011367
34  1.099913355083943   4.832209973419252   2.1538505926119815
35  0.9376430472981663  0.8079117571447663  1.304316868195243
36  0.46877640877716853 5.29116241330836    2.3025071077658543
37  0.4935252096607154  1.281168789487524   1.5477679364371426
38  0.2709953156707687  3.062433830487863   2.693370587501471
39  0.749192114953772   0.6663827602589198  1.9333349814249976
40  0.4805189827879855  0.14184351010408008 2.066316371944877
41  0.23927325979473688 5.173124816927456   1.4642438828849826
42  0.44151741983231113 1.7436037542117058  1.9280793204801427
43  0.23245670480288547 0.37065576873695266 2.030999904138143
44  0.7549855695883286  6.276093161605616   2.0678739480159427
45  0.8075848891260251  0.07471335991038183 1.7963285875231494
46  0.36241490191345777 0.6939068770818817  1.3877989655591956
47  1.1865086460758765  5.558047895686034   2.1668738384961683
48  0.535525745906852   4.814575441121884   1.1287760654201706
49  0.3007113220890282  4.72023158369952    1.0172912673025643
50  1.1425627203873712  5.138942984881625   0.5467165547256857
51  1.7871292066335849  3.5879306828995046  2.5331505944676644
52  0.12554234945350615 6.235182656562409   2.426588195130584
53  1.9508957691788424  1.5727845425135067  1.136370878426268
54  1.2340843271063864  4.266832155446338   1.6095190754643054
55  0.7596490837018829  0.7814527232424783  1.9470808757941769
```

56	0.27603246888940225	2.6809625147685714	1.078495340417599
57	0.1774379688936298	2.2002158586794587	1.6536146773539644
58	0.5279395140055386	2.8106806130373756	0.953328013938007
59	0.28597175133131936	0.9392301257775056	2.00164072679719
60	0.7489457426462115	4.778043836894104	1.1152502543646066
61	0.41583414057708357	0.3294743321137541	1.668693730730418
62	0.15658329196716386	5.461334751920941	1.4053047928763684
63	0.4834230364082359	5.268014599683456	3.059726597288281
64	1.2082082403707055	4.913058948372035	0.20986249004606283
65	0.9228494628294356	4.112734045934136	1.4557394980973237
66	0.49886820798674	5.735940349234949	0.35360816380835713
67	0.8614547698744979	2.4793727335042393	0.9390025811773465
68	1.0404130120319168	5.006881945651275	1.2715714469048516
69	0.5155115476611812	0.7070232115155793	2.33766379862859
70	0.9873043887314078	3.9598006058159445	2.179627574185707
71	0.10482613314882151	0.66326380437729	1.3979058057659053
72	0.28772713789966176	4.629328483802589	0.25178126013971047
73	0.36005415456318	6.021209585994502	0.8163155888412308
74	0.6689967240385887	3.4332119377192907	1.9485455314715487
75	0.6560535325898762	1.0500076927998354	0.6812687658594905
76	0.994852665787628	1.0169697760807013	1.7330830211189694
77	0.7600457554185966	1.012101020662183	1.3519275034504492
78	1.2992785430627949	6.001989268966303	1.112446352248371
79	1.257729282080912	1.6870139107190902	1.4753255467451556
80	1.6263079100380933	4.52754845590209	1.8031835496710111
81	0.3247650057554131	0.5292294782896991	2.0223563584461006
82	0.28236262384336075	4.703858057854235	0.8389067948613075
83	1.3545401061066393	3.38615683866579	0.3914093790124544
84	0.3705036483047302	2.8791585119240106	0.4387058164947563
85	0.9274906770176661	3.2920287574168348	1.4893000917848795
86	0.07540934727746357	5.7610742015275145	2.119575248761535
87	1.862385815784794	3.045052607502409	1.2510993714844976
88	0.3946055122837304	3.340030698103959	1.6701575487199602
89	0.3186661044043466	5.789882713670429	0.9397922480984844
90	0.3721033905270471	1.869356420169736	1.4157104165429575
91	0.5727593440444689	3.4208908185836924	1.6717465311782636
92	1.2116983903225638	4.17020088051413	0.5833618992301244
93	0.6183940090166672	3.6527090074237307	0.5268394477675815
94	0.4324614564924536	5.822071742762191	1.4041879494027756
95	0.2619688271321546	4.4421280711627595	2.1670755002570643
96	0.47129585311631267	3.031660565342671	1.4209084082191634
97	0.36569997827579626	0.8760531441243352	0.642293620960293
98	1.4216840122476373	2.4314977570875973	2.448691070627248
99	0.702096969866075	2.151940879599076	0.31025291051449916

G sorting.py

Listing 33: sorting.py

```
#!/usr/bin/env python3

import nur.routines as rt


def quick_sort(values, _low=None, _high=None):
    '''
    Sort an array using the 'quick sort' method. This recursively chooses
        a
    pivot element and sorts all values on the left and the right of the
        pivot
    with swapping.

    Sorting is done in place. Pass a copy of the values if you want to
        keep the
    original ordering.

    Parameters
    ----------
    values : array-like (1D)
        The array of values to be sorted
    '''
    _low = 0 if _low is None else _low
    _high = len(values)-1 if _high is None else _high

    if _low < _high:
        pivot = values[_low]
        mid = _low
        for curr in range(_low+1, _high+1):
            if values[curr] < pivot:
                mid += 1
                values[mid], values[curr] = rt.switch(values[mid], values
                    [curr])

        values[_low], values[mid] = rt.switch(values[_low], values[mid])

        quick_sort(values, _low, mid-1) # left half
        quick_sort(values, mid+1, _high) # right half



''' Attempt at heap sorting. Doesn't work! '''

class BinaryNode:
    def __init__(self, value=None, parent=None):
        self.value = value
        self.parent = parent
        self.left = None
        self.right = None


    def __str__(self):
        '''
        Print the node value and the values of its pointers in the format
            :

            P       <- parent value
```

```
            C       <- current node value
          L   R     <- left and right node values

        (note this only looks nice for single digit numbers)
        '''
        p = '  ' + str(self.parent.value) + '\n' if self.parent else ''
        c = '  ' + str(self.value) + '\n'
        l = str(self.left.value) + '   ' if self.left else ''
        r = str(self.right.value) if self.right else ''
        return p + c + l + r

    def __repr__(self):
        return '%s:\n%s' %(type(self), self.__str__())


class BinaryTree:
    '''
    An unsorted binary tree
    '''

    def __init__(self, values):
        self.values = values

        self.build_tree()


    def build_tree(self):
        '''
        Recursively construct an (unsorted) binary tree from a list of
            values
        '''
        def insert(root, i):
            if i < len(self.values):
                curr = BinaryNode(self.values[i], root)
                curr.left = insert(curr, 2*i+1)
                curr.right = insert(curr, 2*i+2)

                root = curr

            return root

        self.root = insert(None, 0)

        return self


class HeapTree(BinaryTree):

    def __init__(self, values):
        super().__init__(values)
        self.build_heap()

    def build_heap(self):
        '''
        Sort the binary tree into a heap where the parent node is always
        larger than either of its two children.
        '''
        def heap_swap(node):
            if node.left and (node.left.value > node.value):
                print('L: %s <-> %s' %(node.left.value, node.value))
                node.left.value, node.value = rt.switch(node.left.value,
```

```
                                                        node.value)
                heap_swap(node.left)

            if node.right and (node.right.value > node.value):
                print('R: %s <-> %s' %(node.right.value, node.value))
                node.right.value, node.value = rt.switch(node.right.value
                    ,
                                                         node.value)
                heap_swap(node.right)

            if node.parent and (node.value >= node.parent.value):
                print('U: %s <-> %s' %(node.value, node.parent.value))
                node.parent.value, node.value = rt.switch(node.parent.
                    value,
                                                          node.value)
                heap_swap(node.parent)

        heap_swap(self.root)


    def sort(self):
        pass
```