In [2]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import ComparativeAmplification as ca
```

We compare amplitude amplification methods as a tool to find the maximum value of a set.

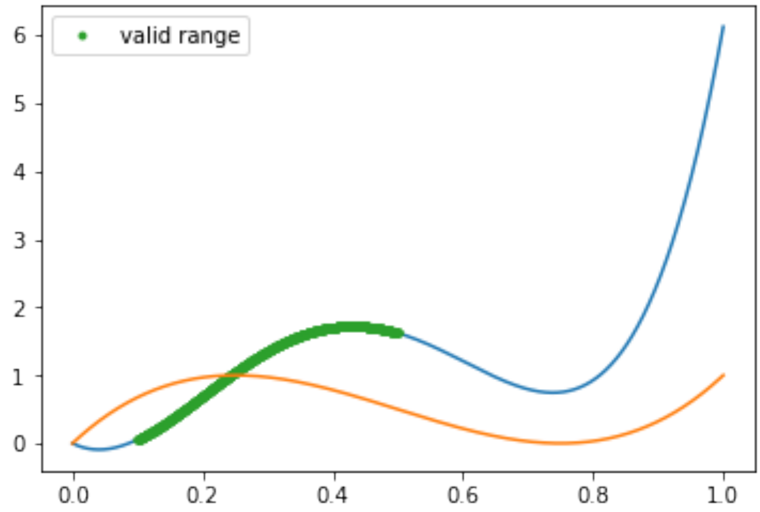
We compare step amplification, used in the Dürr-Høyer algorithm, and compared amplitud amplification, based on the usage of serveral threshold values.

Mean parameters approach

In [5]:
```
def ProbDH(xz):
    '''
    Theretical probability of measuring a state i with F(i) > F(z)
with one Durr-Hoyer amplification step,
    where tz is the number of i's that satisfy that equation among
N elements.
    xz is defined as:
                xz = tz/N
    '''
    return xz*(3-4*xz)**2


def ProbRI(xz, xa, xi, xb, xc):
    '''
    Theretical probability of measuring a state i with F(i) > F(z)
with one superposition amplification step,
    where tz is the number of i's that satisfy that equation among
N elements.
    xz is defined as:
                xz = tz/N
    The parameters statisfy:
                xz: 0 - 1,   solution rate for z
                xa: xz - 1,  solution mean rate for F(a) < F(z)
                xi: 0 - xz,  solution mean rate for F(z) < F(i)
                xb: xi - xz, solution mean rate for F(z) < F(b) < F
(i)
                xc: 0 - xi,  solution mean rate for F(i) < F(c)
    '''
    return xz * ((1-4*xz)**2 * (1-xz) *(3-4*xa)**2 +
             (3-4*xz)**2 * (
                 (xz-xi)*(3-4*xb)**2+xi*(1-4*xc)**2
                      )
             )
```

Firts we will check for an ad-hoc example with x_z as a variable.

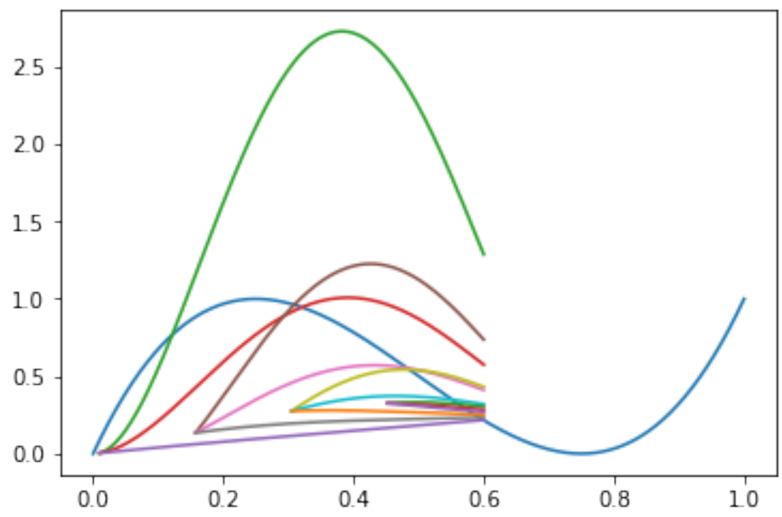
In [72]:
```
xa, xi, xb, xc = .5, .1, .1001, .099
assert xb > xi, 'got xb < xi'
assert xc < xi, 'got xc > xi'

xx = np.linspace(0, 1, 1000)
xvalid = np.linspace(xi, xa, 1000)

plt.plot(xx, ProbRI(xx, xa, xi, xb, xc), 'C0')
plt.plot(xvalid, ProbRI(xvalid, xa, xi, xb, xc), '.C2', label='vali
d range') # this is the valid range
plt.plot(xx, ProbDH(xx), 'C1')
plt.legend()
plt.show()
```



If we try some different values:

In [41]:
```
xa, xi, xb, xc = .6, .1, .2, .01
assert xb > xi, 'got xb < xi'
assert xc < xi, 'got xc > xi'

xx = np.linspace(0, 1, 1000)
plt.plot(xx, ProbDH(xx), 'C0')

cc = ['C1', 'C2', 'C3']
clock = 0
for xi in np.linspace(xc, xa, 5):
    for xb in np.linspace(xi, xa, 3):
        c = 'C{}'.format((clock+1)%9 +1)
        xvalid = np.linspace(xi, xa, 1000)
        plt.plot(xvalid, ProbRI(xvalid, xa, xi, xb, xc), c)
        print(c,' : ', 'xb={:5f}, xi={:5f}'.format(xb, xi))
        clock += 1
plt.show()
```

```
C2  :  xb=0.010000, xi=0.010000
C3  :  xb=0.305000, xi=0.010000
C4  :  xb=0.600000, xi=0.010000
C5  :  xb=0.157500, xi=0.157500
C6  :  xb=0.378750, xi=0.157500
C7  :  xb=0.600000, xi=0.157500
C8  :  xb=0.305000, xi=0.305000
C9  :  xb=0.452500, xi=0.305000
C1  :  xb=0.600000, xi=0.305000
C2  :  xb=0.452500, xi=0.452500
C3  :  xb=0.526250, xi=0.452500
C4  :  xb=0.600000, xi=0.452500
C5  :  xb=0.600000, xi=0.600000
C6  :  xb=0.600000, xi=0.600000
C7  :  xb=0.600000, xi=0.600000
```



All different values

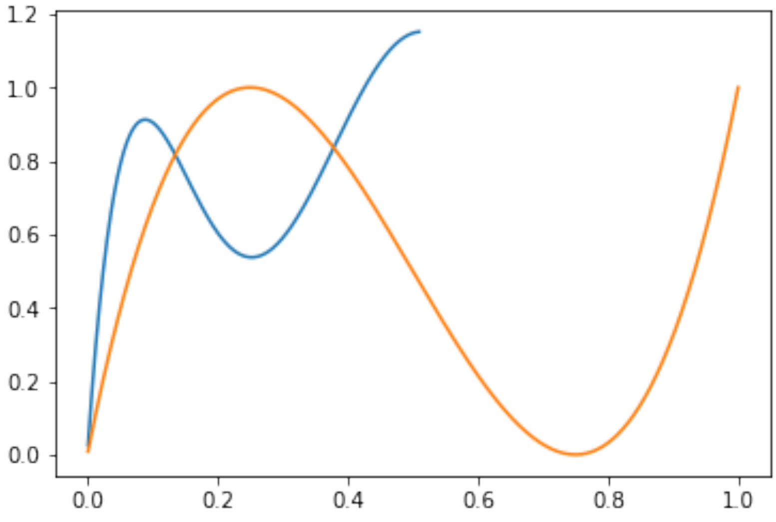
If all the values are differetn, that is $F(i) \neq F(j),\ \forall j \neq i$, then we get the same result in any case:

In [46]:
```
N = 1024
tz = N//4
xz = tz/N

def calc_xi(xz, tz, N):
    return (xz-1/N)/2


def calc_xa(xz, tz, N):
    return 1/4 * (3 - np.sqrt(
    (1-xz-1/N)/(1-xz) * ((3-4*xz)**2 + 8/3*(1-xz)*(2-2*xz-1/N)+4*(3
-4*xz)*(1-xz-1/N))))


def calc_xb(xz, tz, N):
    return 1/4 * (3 - np.sqrt( sum(
            (tz-ti-1)/(tz*(tz-ti)) * ((3-4*ti/N)**2 - 8*(3-4*ti/N)*
(tz-ti-1)/N + 16 * (tz-ti)*(2*tz-2*ti-1)/6/N**2)
                                for ti in range(0, tz-1))))


def calc_xc(xz, tz, N):
    return 1/4 * (1 - np.sqrt( sum(
            1/tz * (1 + 16/N**2 * (ti-1)*(2*ti-1)/6 - 4*(ti-1)/N -
1/ti)
                                for ti in range(1, tz-1))))


xi = calc_xi(xz, tz, N)
xa = calc_xa(xz, tz, N)
xb = calc_xb(xz, tz, N)
xc = calc_xc(xz, tz, N)

print('xa = ', xa)
print('xz = ', xz)
print('xb = ', xb)
print('xi = ', xi)
print('xc = ', xc)
```

```
xa =  -0.15046238147840674
xz =  0.25
xb =  0.26905651188475144
xi =  0.12451171875
xc =  0.05858731346405413
```

In [71]:
```
N = 1024
tz = np.arange(1, N+1)
xz = tz/N


xi = np.array([calc_xi(x, t, N) for x, t in zip(xz, tz)])
xa = np.array([calc_xa(x, t, N) for x, t in zip(xz, tz)])
xb = np.array([calc_xb(x, t, N) for x, t in zip(xz, tz)])
xc = np.array([calc_xc(x, t, N) for x, t in zip(xz, tz)])

pri = ProbRI(xz, xa, xi, xb, xc)
pdh = ProbDH(xz)

plt.plot(xz, pri, 'C0') # parece que algo en las ecuaciones está ma
l, algún factor de 1/2 o así
plt.plot(xz, pdh, 'C1')
plt.show()
```

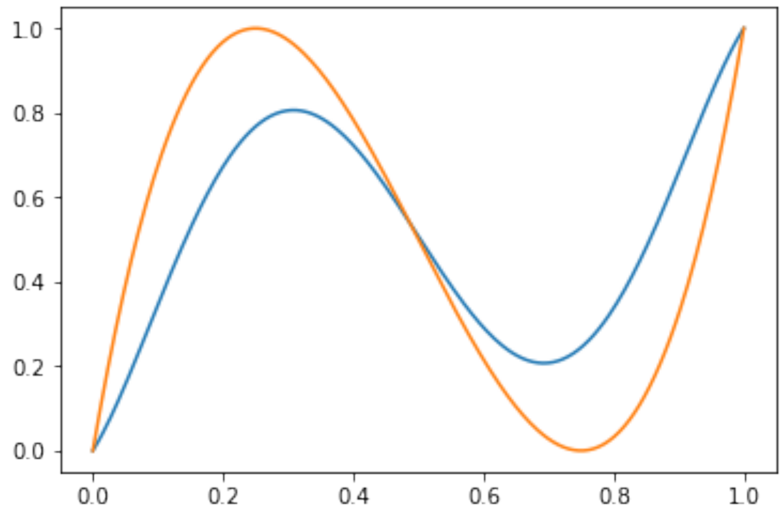
```
/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/si
te-packages/ipykernel_launcher.py:11: RuntimeWarning: divide by ze
ro encountered in double_scalars
  # This is added back by InteractiveShellApp.init_path()
/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/si
te-packages/ipykernel_launcher.py:11: RuntimeWarning: invalid valu
e encountered in sqrt
  # This is added back by InteractiveShellApp.init_path()
/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/si
te-packages/ipykernel_launcher.py:17: RuntimeWarning: invalid valu
e encountered in sqrt
```



Numerical approach, all different values

If we approach it numerically:

In [77]:
```
def probir(ti, tz, N):
    xz = tz/N
    p = 0
    p += 1/N**2 * (1-4*xz)**2 * sum((3-4*t/N)**2 for t in range(tz,
N))
    p += 1/N**2 * (3-4*xz)**2 * sum((3-4*t/N)**2 for t in range(ti,
tz))
    p += 1/N**2 * (3-4*xz)**2 * sum((1-4*t/N)**2 for t in range(0,
ti))
    return p


def probRI(tz, N):
    p = 0
    for ti in range(0, tz):
        p += probir(ti, tz, N)
    return p


N = 64
tz = np.arange(0, N+1)
xz = tz/N

pri = np.array([probRI(tzi, N) for tzi in tz])
pdh = ProbDH(xz)

plt.plot(xz, pri, 'C0') # parece que algo en las ecuaciones está ma
l, algún factor de 1/2 o así
plt.plot(xz, pdh, 'C1')
plt.show()
```



Examples

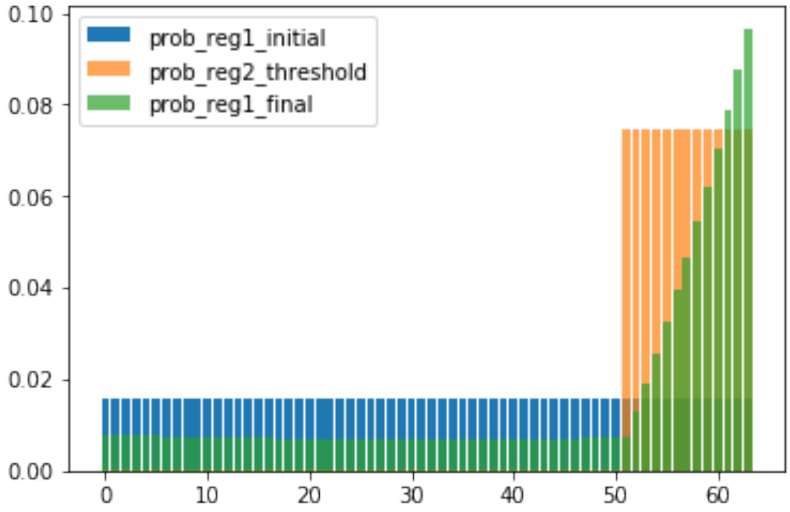
We will take the example of an unsorted list and a linarly increasing function. The results should match with each oder and with the approach above.

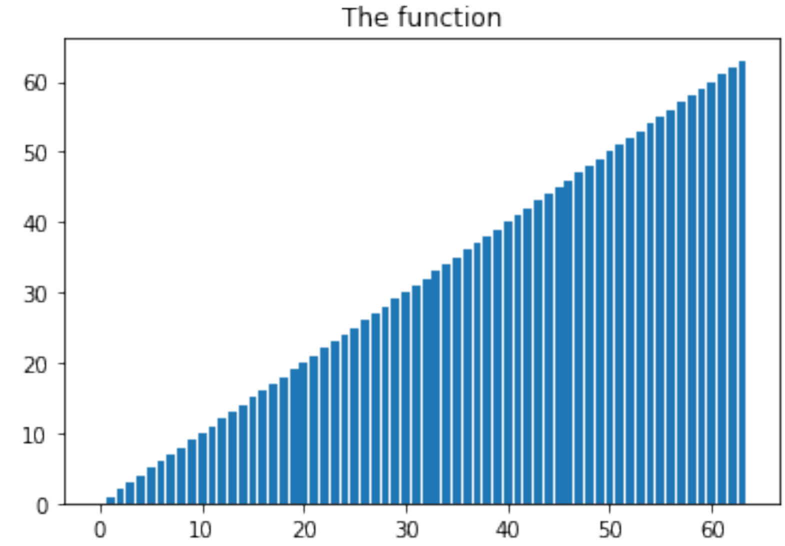
In [1]:
```
import ComparativeAmplification as ca
```

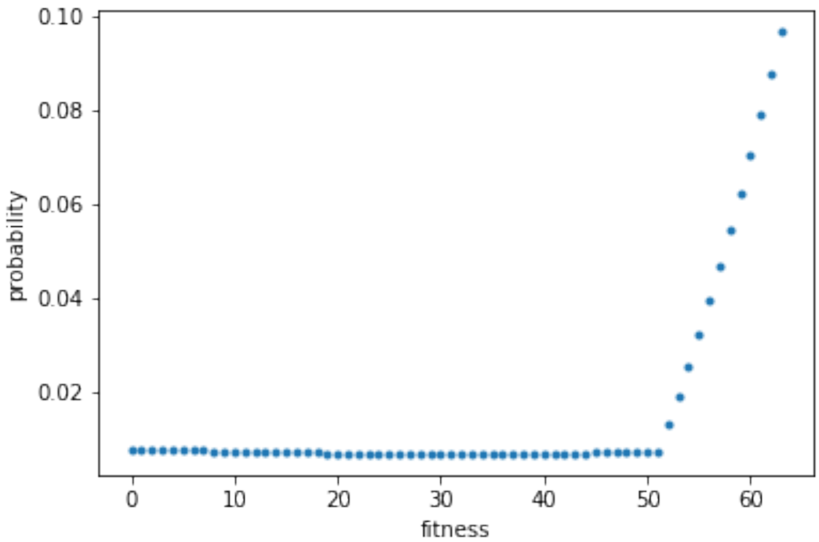
Linear function

In [2]:
```
N = 64
aa = ca.step(N, minval=0, maxval=N, step=1)
linfunction = lambda x: x
ca.test_function(N, aa, 'x', linfunction, 50)
```

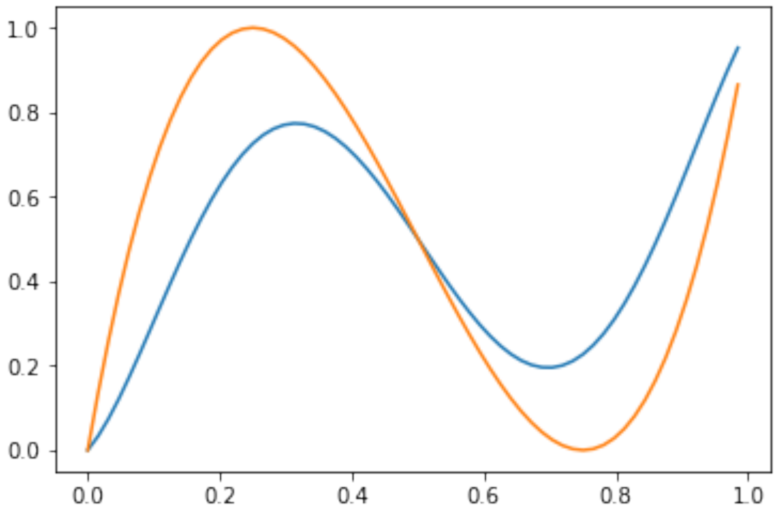
```
| function     | z & tz/N     | bad (f<%20)  | good (f>%80) | exce
l (f>%90)|
| ------------ | ------------ | ------------ | ------------ | ----
-------- |
|            x |   50 & 0.2031 |     0.091616 |     0.626802 |
0.450087 |
```




The function
0
10
20
30
40
50
60


probability
fitness
0.02
0.04
0.06
0.08
0.10
0
10
20
30
40
50
60

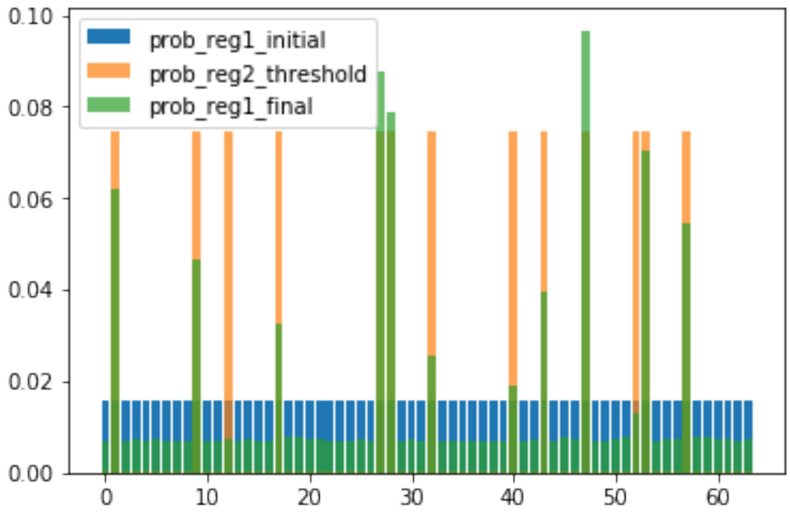
In [7]:
```
pca_z = []
xz = []

for z in range(N):
    pca = sum(ca.prob_compared_amplification(i, z, N, function=linf
unction) for i in ca.get_above_index(z, N, linfunction))
    xz.append(ca.get_above_number(z, N, linfunction) / N)
    pca_z.append(pca)

plt.plot(xz, pca_z)
plt.plot(xz, [ProbDH(xzi) for xzi in xz])
plt.show()
```

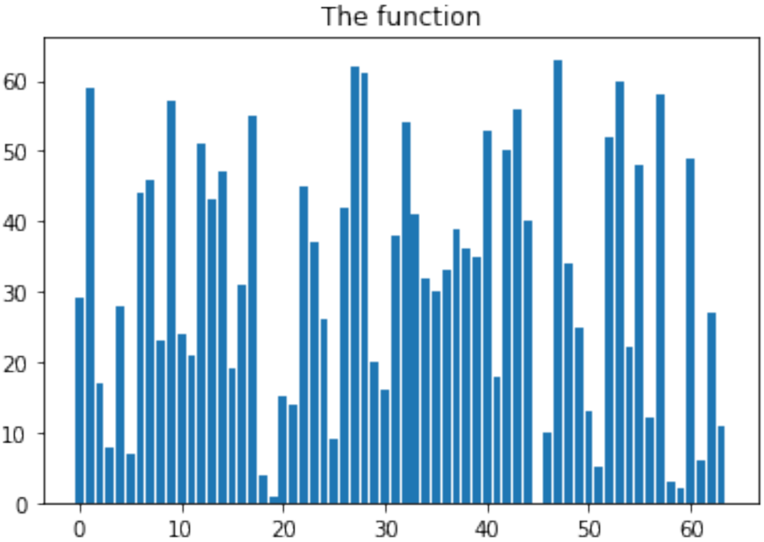


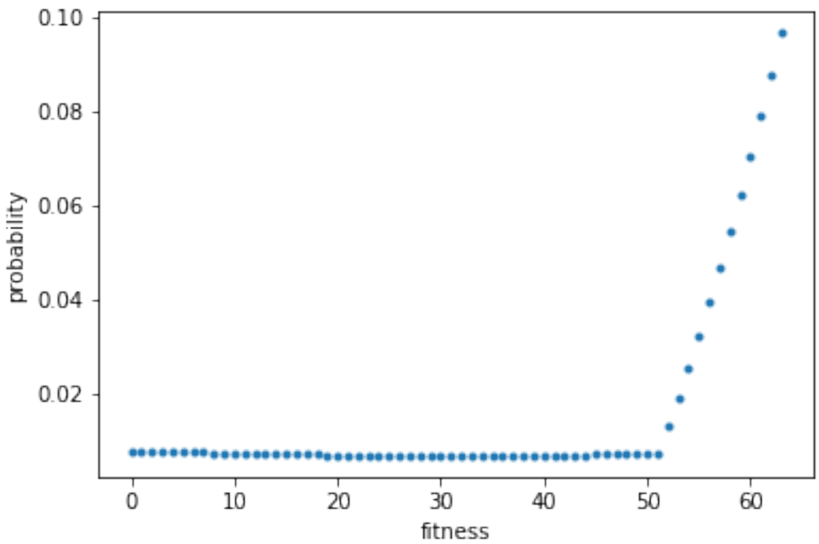
Unsorted list

In [8]:
```
N = 64
aa = ca.step(N, minval=0, maxval=N, step=1)
array = np.array(range(N))
np.random.seed(12)
np.random.shuffle(array)
unsorted = lambda x: array[x]
ca.test_function(N, aa,'unsorted', unsorted, 42)
```

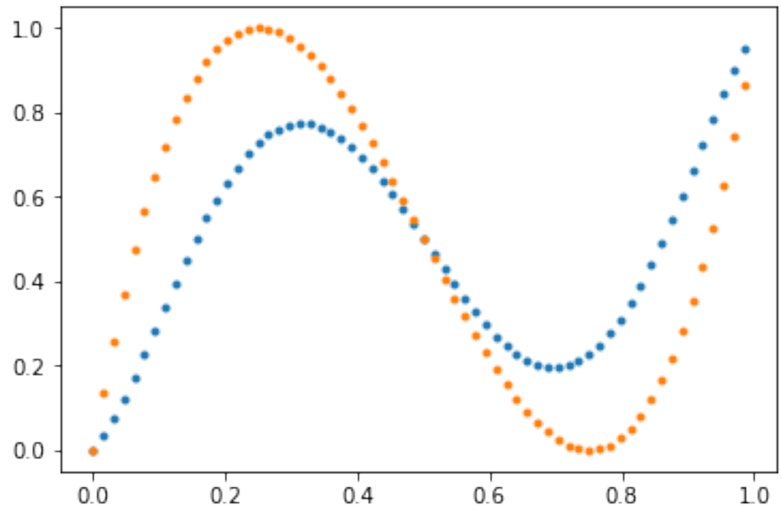
```
| function     | z & tz/N     | bad (f<%20)  | good (f>%80) | exce
l (f>%90)|
| ------------ | ------------ | ------------ | ------------ | ----
-------- |
|     unsorted |  42 & 0.2031 |     0.091616 |     0.626802 |
0.450087 |
```


prob_reg1_initial
prob_reg2_threshold
prob_reg1_final
0.10
0.08
0.06
0.04
0.02
0.00
0
10
20
30
40
50
60


The function
60
50
40
30
20
10
0
0
10
20
30
40
50
60


0.10
0.08
0.06
0.04
0.02
probability
0
10
20
30
40
50
60
fitness

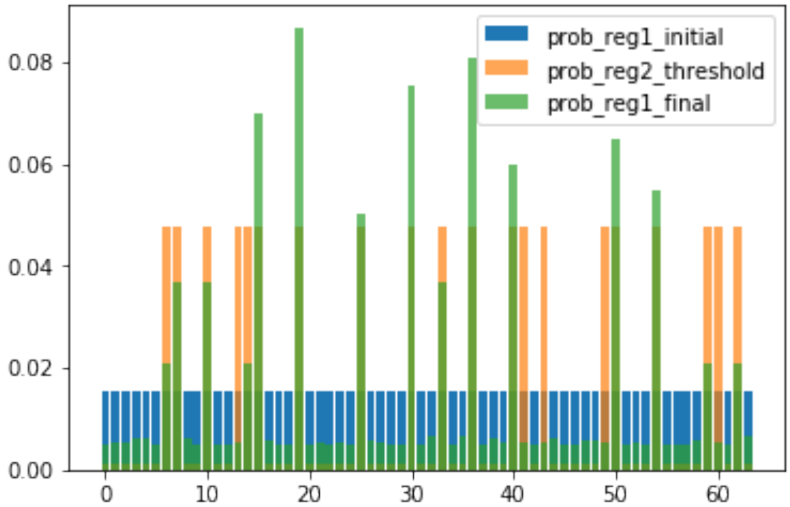
In [11]:
```
pca_z = []
xz = []

for z in range(N):
    pca = sum(ca.prob_compared_amplification(i, z, N, function=unsorted) for i in ca.get_above_index(z, N, unsorted))
    xz.append(ca.get_above_number(z, N, unsorted) / N)
    pca_z.append(pca)

plt.plot(xz, pca_z, '.')
plt.plot(xz, [ProbDH(xzi) for xzi in xz], '.')
plt.show()
```

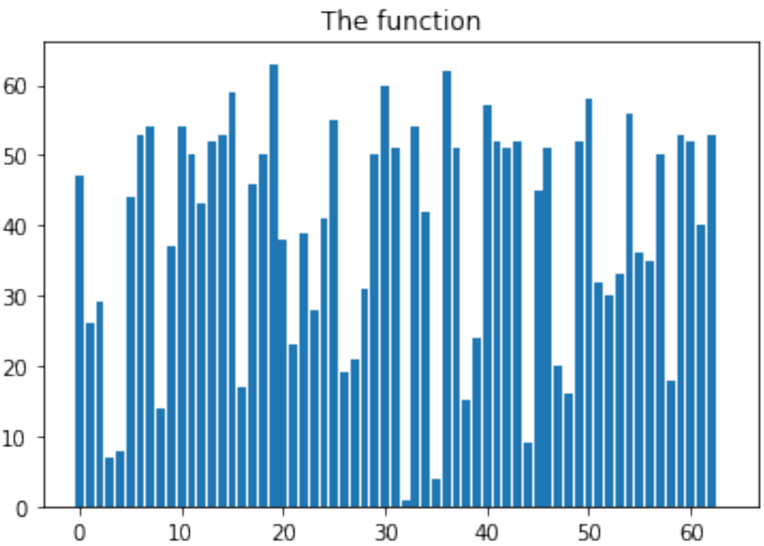


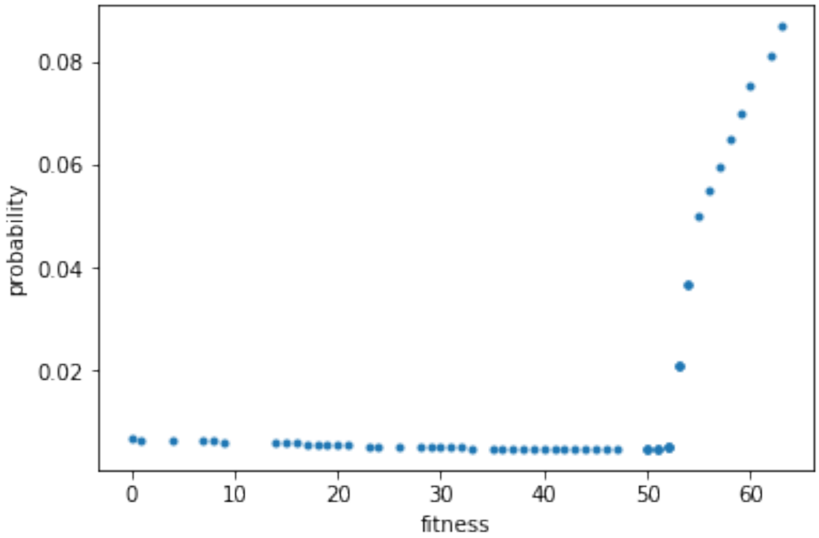
Unsorted list, with some repetitions

In [28]:
```
N = 64
aa = ca.step(N, minval=0, maxval=N, step=1)
repnumber = 20
array = np.array(range(N))
np.random.seed(12)
# this is not the bet way to produce repetition,
# not only the index should be random, but also
# the value...
i = np.random.randint(0, N, repnumber)
array[i] = 50 + i % 5
# -------------------------------------------------
np.random.shuffle(array)
unsorted_rep = lambda x: array[x]
ca.test_function(N, aa,'unsorted with rep', unsorted, 42)
```

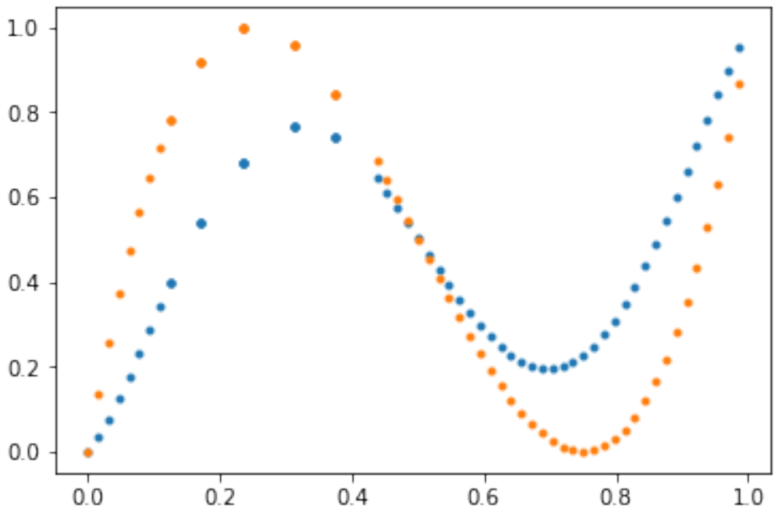
```
| function      | z & tz/N      | bad (f<%20)   | good (f>%80) | exce
l (f>%90)|
| ------------ | ------------ | ------------ | ------------ | ----
-------- |
| unsorted with rep |   42 & 0.3125 |      0.074141 |      0.674872 |
0.438186 |
```







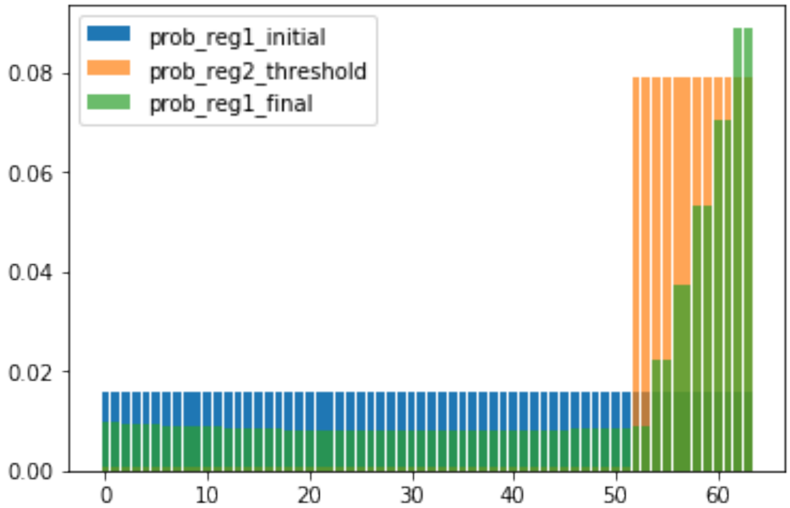
In [29]:
```
pca_z = []
pdh_z = []
xz = []

for z in range(N):
    pca = sum(ca.prob_compared_amplification(i, z, N, function=unso
rted_rep) for i in ca.get_above_index(z, N, unsorted_rep))
    pca_z.append(pca)

    pdh = sum(ca.prob_one_threshold(i, z, N, function=unsorted_rep)
for i in ca.get_above_index(z, N, unsorted_rep))
    pdh_z.append(pdh)

    xz.append(ca.get_above_number(z, N, unsorted_rep) / N)

plt.plot(xz, pca_z, '.')
plt.plot(xz, pdh_z, '.') # The DH analytic formula is not valid wit
h repetitions
plt.show()
```

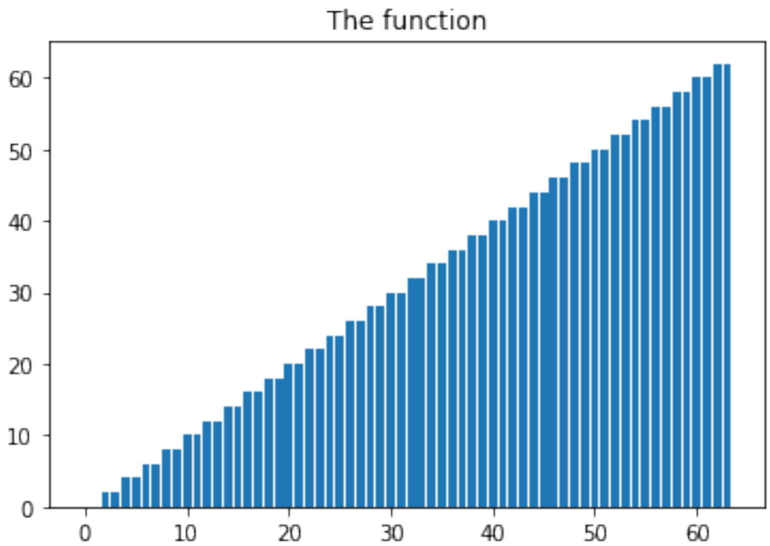


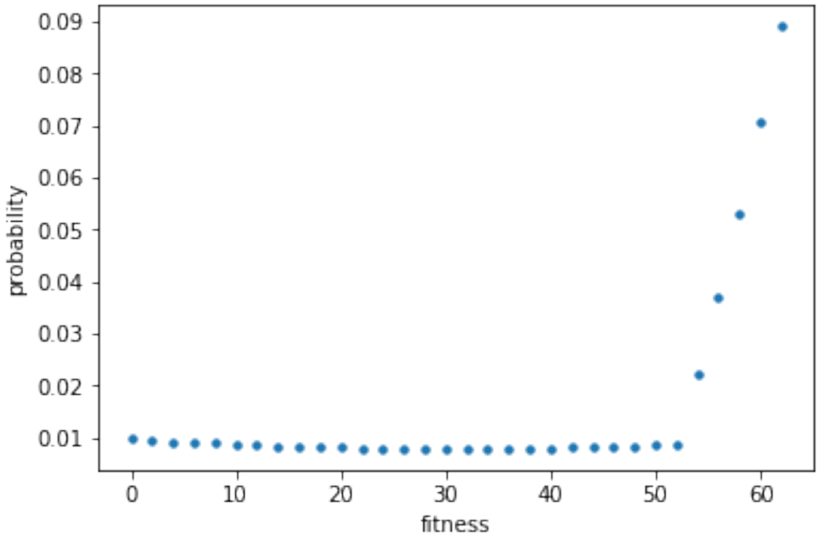
Linear function, with some repetitions

In [34]:
```
N = 64
aa = ca.step(N, minval=0, maxval=N, step=1)
linfunction_rep = lambda x: x - x % 2
ca.test_function(N, aa, 'x rep', linfunction_rep, 50)
```

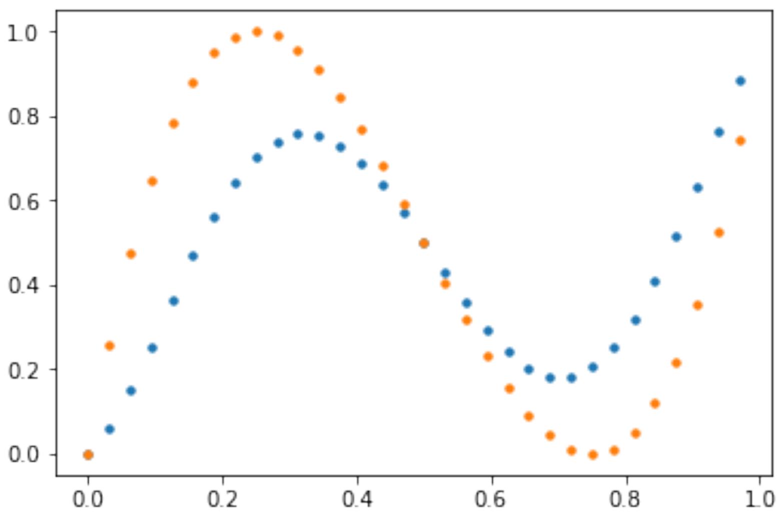
```
| function     | z & tz/N     | bad (f<%20)  | good (f>%80) | exce
l (f>%90)|
| ------------ | ------------ | ------------ | ------------ | ----
-------- |
|        x rep |  50 & 0.1875 |     0.110748 |     0.562408 |
0.425812 |
```







In [31]:
```
pca_z = []
pdh_z = []
xz = []
function = linfunction_rep
for z in range(N):
    pca = sum(ca.prob_compared_amplification(i, z, N, function=func
tion) for i in ca.get_above_index(z, N, function))
    pca_z.append(pca)

    pdh = sum(ca.prob_one_threshold(i, z, N, function=function) for
i in ca.get_above_index(z, N, function))
    pdh_z.append(pdh)

    xz.append(ca.get_above_number(z, N, function) / N)

plt.plot(xz, pca_z, '.')
plt.plot(xz, pdh_z, '.') # The DH analytic formula is not valid wit
h repetitions
plt.show()
```



The Final Countdown ninonino

Some simulaitons were performed in order to compare both methods.

We call DHCA (Durr-Hoyer compared amplification) to the usage of a single threshold in each step for the comparision in the oracle. The real Durr-Hoyer algorithm uses a more complex approach (BBHT) in each step that empowers the search.

We call SCA (superposition compared amplification) to the usage of a superposition of thresholds en each step for the coparision in the oracle, this means *2 oracle calls* in each step.

$\langle T \rangle (N)$

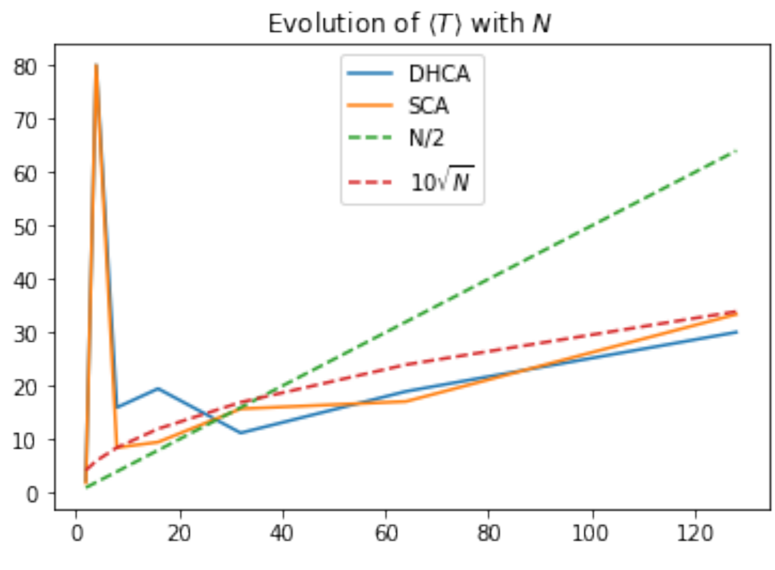
This is the average result obtained for the average number of steps, T, as N varies. Starting from 1 (worst case) and performing a maximum of 80 steps. 100 trials were made.

A guess of $\langle T \rangle (N) \sim 10\sqrt{N}$ is also drawn.

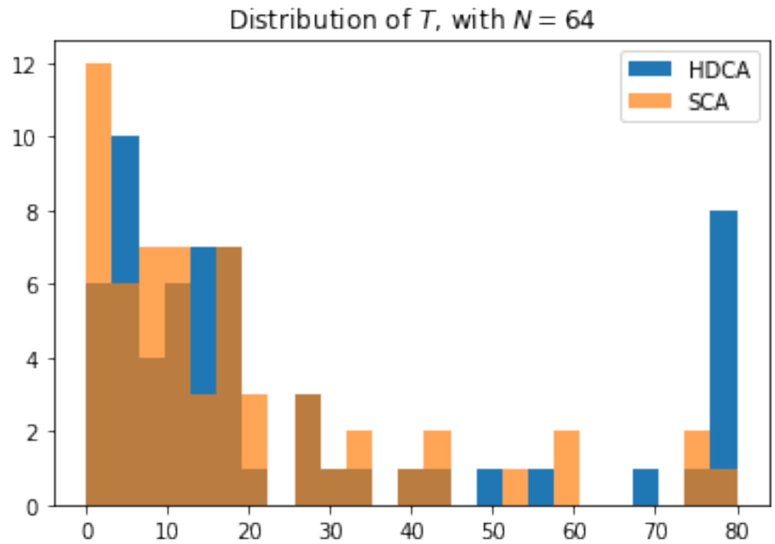
But the actual distribution for each N is not a normal distribution.

In [14]:
```
av_tdhca = []
av_tsca = []
nn = np.array([2, 4, 8, 16, 32, 64, 128])
for n in nn:
    tdhca, tsca = ca.parse_T_dhca_vs_sca(60, 'data/T_dhca_vs_sca_c8
0_z0_r60_n{}.data'.format(n))
    tdhca.sort()
    tsca.sort()
    tdhca = np.array(tdhca)
    tsca = np.array(tsca)
    av_tdhca.append(sum(tdhca)/len(tdhca))
    av_tsca.append(sum(tsca) / len(tsca))

    print("N = {}".format(n))
    print('tdhca = {:5.3f} ({:5.3f})'.format(sum(tdhca)/len(tdhca),
                                            np.sqrt(sum(tdhca**2)/
len(tdhca) - (sum(tdhca)/len(tdhca))**2)))
    print('tsca = {:5.3f} ({:5.3f})'.format(sum(tsca) / len(tsca),
                                           np.sqrt(sum(tsca ** 2)
/ len(tsca) - (sum(tsca) / len(tsca)) ** 2)))
    print()
plt.plot(nn, av_tdhca, label='DHCA')
plt.plot(nn, av_tsca, label='SCA')
plt.plot(nn, nn/2, '--', label='N/2')
plt.plot(nn, 3*np.sqrt(nn), '--', label='$10\sqrt{N}$')
plt.title('Evolution of $\\langle T \\rangle$ with $N$')
plt.legend()
plt.show()
```

```
N = 2
tdhca = 2.300 (1.735)
tsca = 1.933 (1.289)

N = 4
tdhca = 80.000 (0.000)
tsca = 80.000 (0.000)

N = 8
tdhca = 15.983 (24.153)
tsca = 8.500 (6.262)

N = 16
tdhca = 19.517 (25.683)
tsca = 9.517 (8.829)

N = 32
tdhca = 11.233 (14.044)
tsca = 15.750 (15.570)

N = 64
tdhca = 19.000 (20.123)
tsca = 17.117 (16.820)

N = 128
tdhca = 30.067 (24.296)
tsca = 33.417 (27.636)
```



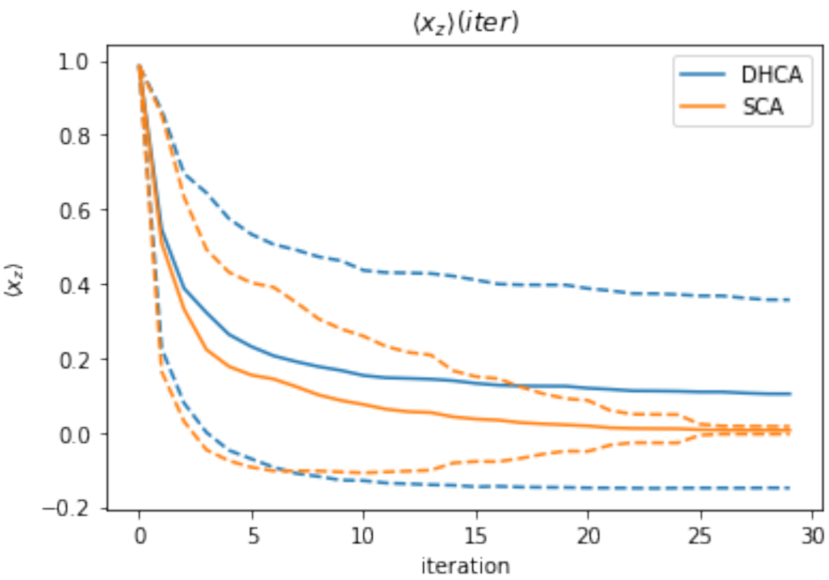
In [12]:
```
tdhca, tsca = ca.parse_T_dhca_vs_sca(60, 'data/T_dhca_vs_sca_c80_z0
_r60.data'.format(n)) # N = 64
plt.hist(tdhca, label='HDCA', range=(0,80), bins=25)
plt.hist(tsca, label='SCA', alpha=.7, range=(0,80), bins=25)
plt.legend()
plt.title('Distribution of $T$, with $N=64$')
plt.show()
```



$\langle x_z \rangle (iter)$

This is the average result obtained for the evolution of x_z in each iteration, starting from 1 (worst case) and performing 30 steps. 100 trials were made.

In [17]:
```
xzdhca, xzsca = ca.parse_xz_dhca_vs_sca(100, 'data/xz_dhca_vs_sca_c
30_z0_r100.data')
a, b = ca.parse_xz_dhca_vs_sca(20, 'data/xz_dhca_vs_sca_c30_z0_r20.
data')
xzdhca += a
xzsca += b
xzdhca, xzsca = np.array(xzdhca), np.array(xzsca)

xz_dhca = sum(xzdhca) / len(xzdhca)
dev_dhca = np.sqrt(sum(xzdhca ** 2) / len(xzdhca) - xz_dhca ** 2)

xz_sca = sum(xzsca) / len(xzsca)
dev_sca = np.sqrt(sum(xzsca ** 2) / len(xzsca) - xz_sca ** 2)

plt.plot(xz_dhca, 'C0', label='DHCA')
plt.plot(xz_dhca + dev_dhca, '--C0')
plt.plot(xz_dhca - dev_dhca, '--C0')

plt.plot(xz_sca, 'C1', label='SCA')
plt.plot(xz_sca + dev_sca, '--C1')
plt.plot(xz_sca - dev_sca, '--C1')

plt.legend()
plt.xlabel('iteration')
plt.ylabel('$\\langle x_z \\rangle$')
plt.title('$\\langle x_z \\rangle (iter)$')
plt.show()
```



In [ ]: