In [1]:
```
%matplotlib inline
from tensorscaling import scale, capacity, unit_tensor, random_tensor, marginal, random_unitary, random_orthogonal
import numpy as np
import scipy as scipy
import cvxpy as cp
import matplotlib.pyplot as plt
from numpy import matrix
```

Tensor scaling

Scale 3x3x3 unit tensor to certain non-uniform marginals:

In [ ]:
```
shape = [3, 3, 3]
targets = [(.5, .25, .25), (.4, .3, .3), (.7, .2, .1)]

res = scale(unit_tensor(3, 3), targets, eps=1e-4)
res
```

We can also access the scaling matrices and the final scaled state:

In [ ]:
```
print(res.gs[0], "\n")
print(res.gs[1], "\n")
print(res.gs[2])
```

Let's now check that the W tensor *cannot* be scaled to uniform marginals:

In [22]:
```
shape = [2, 2, 2, 2]
W = np.zeros(shape)
W[1, 0, 0, 0] = W[0, 1, 0, 0] = W[0, 0, 1, 0] = W[0, 0, 0, 1] = .5
targets = [(.5, .5)] * 4
print(targets)

scale(W, targets, eps=1e-4, max_iterations=1000)
```

```
[(0.5, 0.5), (0.5, 0.5), (0.5, 0.5), (0.5, 0.5)]
```

Out[22]:
```
Result(success=False, iterations=1000, max_dist=0.5934653559719874, ...)
```

To see more clearly what is going on, we can set the `verbose` flag:

In [ ]:
```
res = scale(W, targets, eps=1e-4, max_iterations=10, verbose=True)
```

We see that at each point in the algorithm, one of the marginals has Frobenius distance ≈ 0.59 to being uniform. Indeed, we know that the entanglement polytope of the W tensor does not include the point corresponding to uniform marginals -- see here for an interactive visualization!

Tuples of matrices and tensors

We can just as well only prescribe the desired spectra for subsystems. Note that prescribing two out of three marginals amounts to *operator scaling*.

In [ ]:
```
shape = [3, 3, 3]
targets = [(.4, .3, .3), (.7, .2, .1)]

res = scale(unit_tensor(3, 3), targets, eps=1e-6)
res
```

Indeed, the last two marginals are as prescribed, while the first marginal is arbitrary.

In [ ]:
```
print(marginal(res.psi, 0).round(5), "\n")
print(marginal(res.psi, 1).round(5), "\n")
print(marginal(res.psi, 2).round(5))
```

Duality

The scaling way: The below computes $\frac{1}{t}\inf_{\det L=\det R=1}\langle L\otimes R, e^{tC}\rangle$ for inputs $C = uni(spec)uni^{\dagger}$, dimension n, and weight t.

In [25]:
```
def scalingot(spec,uni,n,weight):
    #marginals p and q are just normalized identities
    targets = [tuple((n**(-1))*np.ones(n)), tuple((n**(-1))*np.ones(n))]

    expcost = uni.copy()
    #make the unitary a list of nxn matrices
    expcost=expcost.reshape([n**2,n,n])


    expspec = np.exp(weight*spec)
    #multiply the i^{th} eigenvector by e^(weight*spec[i])
    for i in range(0,n**2):
        expcost[i]*=expspec[i]
    cap = capacity(expcost, targets, eps=1e-4, max_iterations=400,randomize=False, verbose=False)

    return cap/weight
```

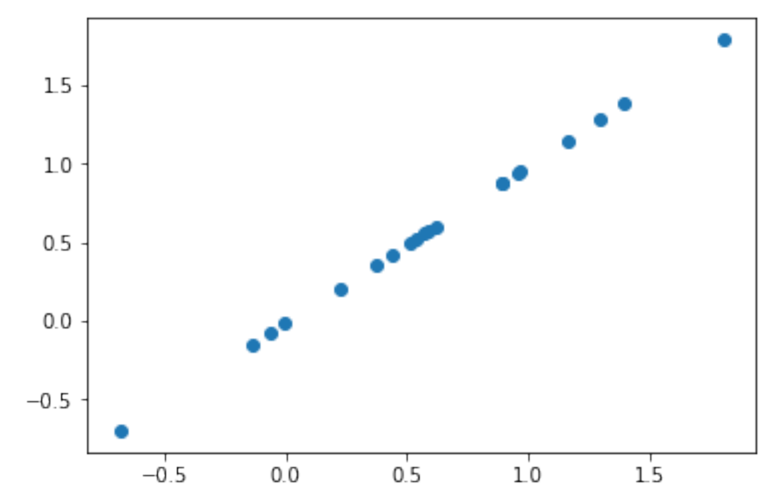
The SDP way: Compute the sdp $\max C\cdot\rho$ subject to $E_{ij}\otimes I_n\cdot\rho = P_{ij}$ and $I_n\otimes E_{ij}\cdot\rho = Q_{ij}$ and, of course $\rho\succeq 0$.

In [13]:
```
def sdpot(spec, uni, n):
    #compute the cost matrix C = uni*spec*uni^T
    C = uni.dot(np.diag(spec).dot(np.conj(uni).T))
    #currently the marginals are just the normalized identities
    p = np.eye(n,n)/n
    q = np.eye(n,n)/n


    X = cp.Variable((n**2,n**2), symmetric=True)
    constraintos = [X >> 0]

#add the partial trace constraints

    for i in range(n):
        for j in range(i+1):
            a = np.kron(np.kron(np.eye(1,n,i), np.eye(n,1,-j)), np.eye(n,n))        #c.reshape([n**2,n**2])
            a = (a + a.T)/2

            constraintos += [cp.trace(a @ X) == p[i,j]]



    for i in range(n):
        for j in range(i+1):
            a = np.kron(np.eye(n,n), np.kron(np.eye(1,n,i), np.eye(n,1,-j)))        #c.reshape([n**2,n**2])
            a = (a + a.T)/2

            constraintos += [cp.trace(a @ X) == q[i,j]]


    prob1 = cp.Problem(cp.Maximize(cp.trace(C @ X)),
                 constraintos)

    prob1.solve()

    return prob1.value
```

Sanity check: when the cost C is diagonal these algorithms should match

In [26]:
```
n = 2
scal = []
sdp = []

for i in range(20):
    #choose random spectrum for cost matrix
    spec = np.random.randn(n**2)
    #C will be uni*spec*uni^T, for now we make C diagonal
    uni = np.eye(n**2)
    #compute the mincost using both algorithms
    sc = scalingot(spec,uni,n,20)
    sd = sdpot(spec,uni,n)
    #put them in a list
    scal.append(sc)
    sdp.append(sd)
#scatter plot it
plt.scatter(scal,sdp)
```

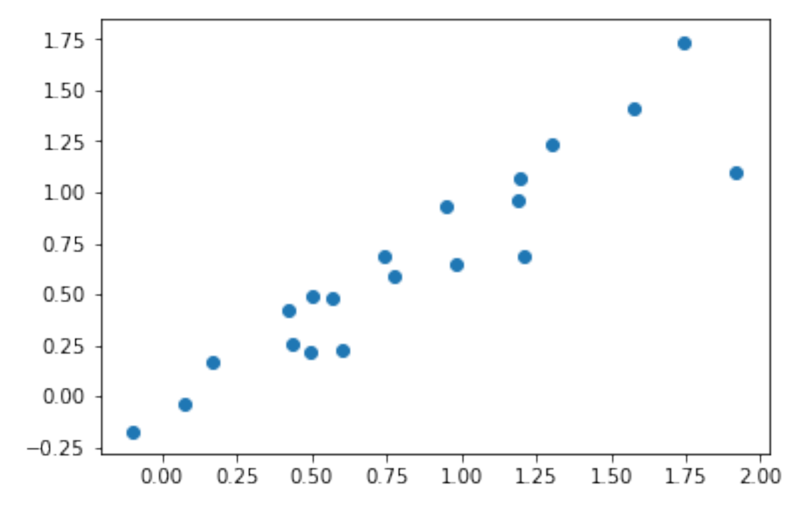
Out[26]:
```
<matplotlib.collections.PathCollection at 0x11d59c908>
```



In general: we don't know what will happen, but it doesn't look like they are the same.

In [27]:
```
n = 2
scal = []
sdp = []

for i in range(20):
    spec = np.random.randn(n**2)
    #same except this time we use random unitary so that C is quite random
    uni = random_orthogonal(n**2)
    sc = scalingot(spec,uni,n,20)
    sd = sdpot(spec,uni,n)
    scal.append(sc)
    sdp.append(sd)

plt.scatter(scal,sdp)
```

Out[27]:
```
<matplotlib.collections.PathCollection at 0x11d6cebe0>
```



scratchwork

In [ ]:
```
tuple(np.ones(3))
```

In [ ]:
```

```