Learning dimensional constants

In [1]:
```
!pip install smithnormalform
from smithnormalform import snfproblem
from smithnormalform import matrix as snfmatrix
from smithnormalform import z as snfz
import numpy as np
import pylab as plt
import itertools as it
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: smithnormalform in /usr/local/lib/python3.7/di
st-packages (0.6.0)
```

In [2]:
```
# define units-carrying object
class quantity():
    unit_names = ["kg", "m", "s", "K"]

    def __init__(self, x, u):
        self.x = x
        self.u = np.array(u).astype(int)
        assert len(self.u) == len(self.unit_names)

    def unit_string(self):
        foo = ""
        for i, alpha in enumerate(self.u):
            if alpha != 0:
                foo += "\," + self.unit_names[i] + "^{" + str(alpha) + "}"
        return foo

    def __str__(self):
        return "$" + str(self.x) + self.unit_string() + "$"

    def __mul__(self, other):
        return quantity(self.x * other.x, self.u + other.u)

    def __truediv__(self, other):
        return quantity(self.x / other.x, self.u - other.u)

    def __pow__(self, b):
        return quantity(self.x ** b, self.u * b)

    def __add__(self, other):
        assert self.u == other.u
        return quantity(self.x + other.x, self.u)

    def __sub__(self, other):
        assert self.u == other.u
        return quantity(self.x - other.x, self.u)
```

In [3]:
```
# define physical constants
# examples:
# temperature has units [0 0 0 1]
# wavelength has units [0 1 0 0]
c = quantity(299_792_458.0, [0, 1, -1, 0]) #speed of light
kB = quantity(1.380649e-23, [1, 2, -2, -1]) #Boltzmann's constant
h = quantity(6.62607015e-34, [1, 2, -1, 0]) #Planck's constant
print(c, kB, h)
print(h * c)
print(c / kB)
```

```
$299792458.0\,m^{1}\,s^{-1}$ $1.380649e-23\,kg^{1}\,m^{2}\,s^{-2}\,K^{-1}$
$6.62607015e-34\,kg^{1}\,m^{2}\,s^{-1}$
$1.9864458571489286e-25\,kg^{1}\,m^{3}\,s^{-2}$
$2.171387934225136e+31\,kg^{-1}\,m^{-1}\,s^{1}\,K^{1}$
```

In [4]:
```
# define God's truth (the true black body formula)
def truth(lam, T):
    return (quantity(2., [0, 0, 0, 0]) * h * c ** 2 / lam ** 5
            / quantity(np.exp((h * c / (lam * kB * T)).x) + 1., [0, 0, 0, 0]))
```

In [5]:
```
# set up parameters for a training set
lam = quantity(10. ** np.arange(-8, -4, 0.1), [0, 1, 0, 0]) #wavelengths
```

In [6]:
```
# make a training set
# data and physical constants
Ts = [300., 1000., 3000., 10000., 30000.] #temperatures
Ts = [quantity(np.zeros_like(lam.x) + T, [0, 0, 0, 1]) for T in Ts]
B_lams = [truth(lam, T) for T in Ts] #intensities
ys = np.array([B.x for B in B_lams]).flatten()
y_u = B_lams[0].u
n = len(ys)
xs = np.zeros((n, 5)) #values
x_u = np.zeros((5, 4)).astype(int) #units
xs[:, 0] = np.array([lam.x for B in B_lams]).flatten()
x_u[0, :] = lam.u
xs[:, 1] = np.array([T.x for T in Ts]).flatten()
x_u[1, :] = Ts[0].u
xs[:, 2] = c.x
x_u[2, :] = c.u
xs[:, 3] = kB.x
x_u[3, :] = kB.u
xs[:, 4] = h.x
x_u[4, :] = h.u
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:4: RuntimeWarnin
g: overflow encountered in exp
  after removing the cwd from sys.path.
```

In [7]:
```
# make a test set
Ts_test = [6000., 20000.]
Ts_test = [quantity(np.zeros_like(lam.x) + T, [0, 0, 0, 1]) for T in Ts_test]
B_lams_test = [truth(lam, T) for T in Ts_test]
ys_test = np.array([B.x for B in B_lams_test]).flatten()
n_test = len(ys_test)
xs_test = np.zeros((n_test, 5))
xs_test[:, 0] = np.array([lam.x for B in B_lams_test]).flatten()
xs_test[:, 1] = np.array([T.x for T in Ts_test]).flatten()
xs_test[:, 2] = c.x
xs_test[:, 3] = kB.x
xs_test[:, 4] = h.x
```

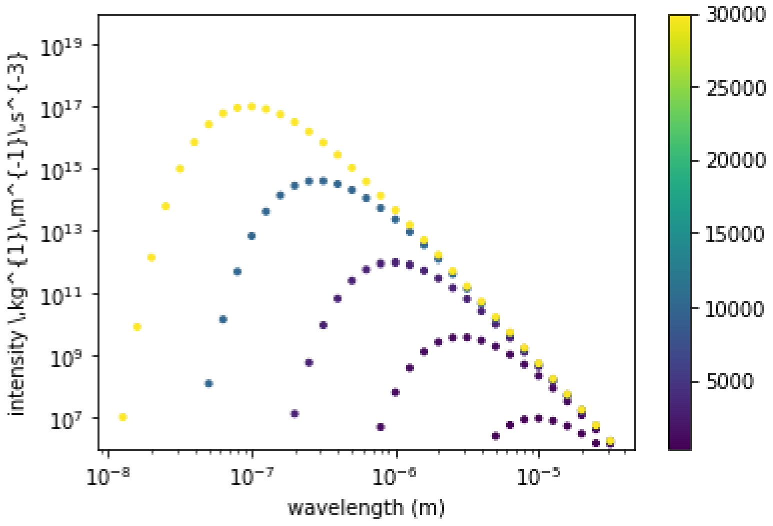
In [8]:
```
# cut on intensity <1e6
y_min = 1e6
i_train = ys > y_min
xs = xs[i_train]
ys = ys[i_train]
i_test = ys_test > y_min
xs_test = xs_test[i_test]
ys_test = ys_test[i_test]
```

In [9]:
```
# what are the units of the labels and features?
print(y_u)
print(x_u)
```

```
[ 1 -1 -3  0]
[[ 0  1  0  0]
 [ 0  0  0  1]
 [ 0  1 -1  0]
 [ 1  2 -2 -1]
 [ 1  2 -1  0]]
```

```
In [10]: # visualize the training set
         plt.scatter(xs[:, 0], ys, marker=".", c=xs[:, 1])
         plt.colorbar()
         plt.ylim(1.e6, 1.e20)
         plt.ylabel("intensity " + truth(quantity(1.0, [0, 1, 0, 0]), Ts[0]).unit_strin
         g())
         plt.xlabel("wavelength (m)")
         plt.loglog()

Out[10]: []
```



Non-units-covariant regression using only wavelengths and temperatures as features

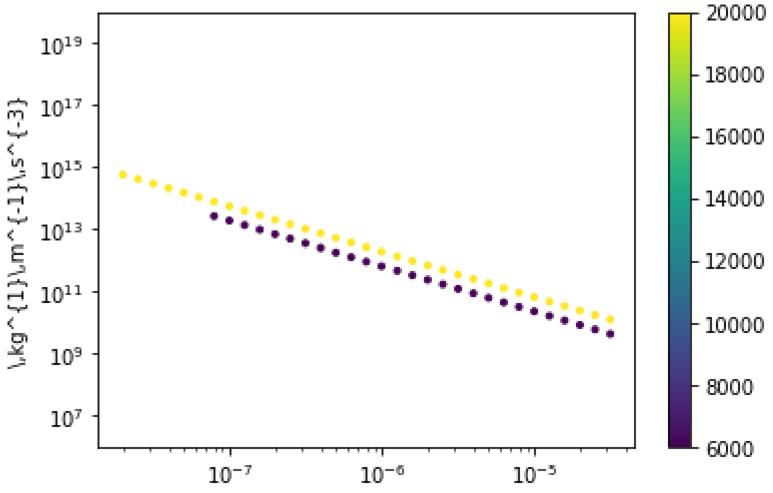
Note: currently the baseline regression does not work

```
In [11]: from sklearn.neural_network import MLPRegressor
         regr = MLPRegressor(hidden_layer_sizes=(20,20,20), random_state=1, max_iter=10
         000).fit(np.log(xs[:,:2]), np.log(ys))

In [12]: ys_hat= regr.predict(np.log(xs_test[:,:2]))
```

In [13]:
```
# visualize the test results
plt.scatter(xs_test[:, 0], np.exp(ys_hat), marker=".", c=xs_test[:, 1])
plt.colorbar()
plt.ylim(1.e6, 1.e20)
plt.ylabel(truth(quantity(1.0, [0, 1, 0, 0]), Ts[0]).unit_string())
plt.loglog()
```

Out[13]:
```
[]
```



Units-covariant regression

```
In [14]: def integer_solve(A, b):
             """
             Find all solutions to Ax=b where A, x, b are integer.

             ## inputs:
             - A - [n, m] integer matrix - n < m, please
             - b - [n] integer vector

             ## outputs:
             - vv - [m] one integer vector solution to the problem Ax=b
             - us - [k, m] set of k integer vector solutions to the problem Ax=0

             ## bugs / issues:
             - Might get weird when k <= 1.
             - Might get weird if k > m - n.
             - Depends EXTREMELY strongly on everything being integer.
             - Uses smithnormalform package, which is poorly documented.
             - Requires smithnormalform package to have been imported as follows:
                 !pip install smithnormalform
                 from smithnormalform import snfproblem
                 from smithnormalform import matrix as snfmatrix
                 from smithnormalform import z as snfz
             """
             ## perform the packing into SNF Matrix format; HACK
             n, m = A.shape
             assert(m >= n)
             assert(len(b) == n)
             assert A.dtype is np.dtype(int)
             assert b.dtype is np.dtype(int)
             smat = snfmatrix.Matrix(n, m, [snfz.Z(int(a)) for a in A.flatten()])
             ## calculate the Smith Normal Form
             prob = snfproblem.SNFProblem(smat)
             prob.computeSNF()
             ## perform the unpacking from SNF Matrix form; HACK
             SS = np.array([a.a for a in prob.S.elements]).reshape(n, n)
             TT = np.array([a.a for a in prob.T.elements]).reshape(m, m)
             JJ = np.array([a.a for a in prob.J.elements]).reshape(n, m)
             ## Find a basis for the lattice of null vectors
             us = None
             zeros = np.sum(JJ ** 2, axis=0) == 0
             us = (TT[:, zeros]).T
             DD = SS @ b
             v = np.zeros(m)
             v[:n] = DD / np.diag(JJ)
             vv = (TT @ v).astype(int)
             return vv, us
```

```
In [15]: def create_dimensionless_features(vv, us, X, X_test):
             exponents = np.vstack([np.zeros_like(vv), us]).T
             features = np.exp(np.log(X)@exponents)
             features_test = np.exp(np.log(X_test)@exponents)
             return features, features_test
```

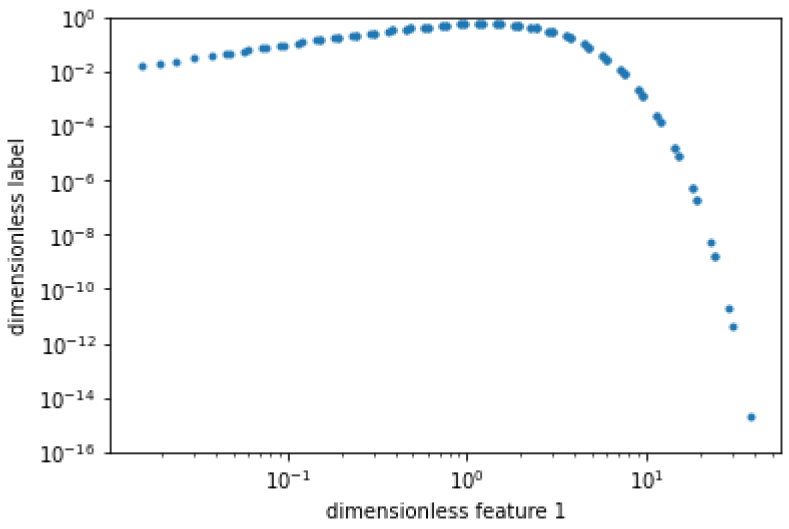
In [16]:
```
def create_dimensionless_labels(vv, X, Y):
    return np.exp(np.log(Y) - np.log(X)@vv)
```

If we use Planck's constant and make the problem fully dimensionless, all curves in the training set collapse into one curve.

In [17]:
```
vv, us = integer_solve(x_u.T, y_u)
XX, XX_test = create_dimensionless_features(vv, us, xs, xs_test)
YY = create_dimensionless_labels(vv, xs, ys)

plt.scatter(XX[:, 1], YY, marker=".")
plt.loglog()
plt.ylim(1e-16, 1e0)
plt.ylabel("dimensionless label")
plt.xlabel("dimensionless feature 1")
```

Out[17]:
```
Text(0.5, 0, 'dimensionless feature 1')
```



In [18]:
```
def dimensionless_regression(X, Y, a_X, a_Y, X_test):
    vv, us = integer_solve(a_X.T, a_Y)
    XX, XX_test = create_dimensionless_features(vv, us, X, X_test)
    YY = create_dimensionless_labels(vv, X, Y)
    regr = MLPRegressor(hidden_layer_sizes=(20,20,20), random_state=1, max_ite
r=10000).fit(np.log(XX), np.log(YY))
    dimensionless_predictions = regr.predict(np.log(XX_test))
    predictions = np.exp(dimensionless_predictions + np.log(X_test)@vv)
    return predictions
```

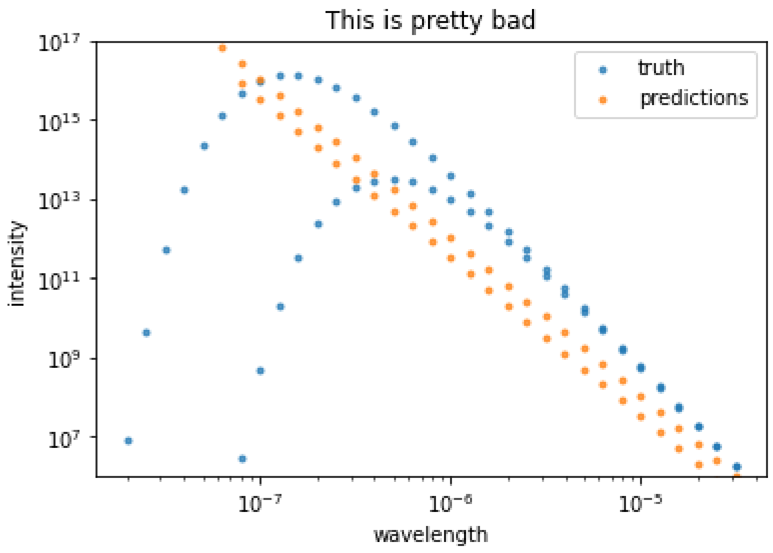
Perform a units-covariant regression without Planck's constant:

```
In [19]: vv, us = integer_solve(x_u[:-1,:].T, y_u) # :-1 removes Plack's constant from
          the features
         print(vv)
         print(us)
         #the outputs show that there is no dimensionless features, but there is an out
         put with the correct dimensions
```

```
[-4  1  1  1]
[]
```

```
In [20]: ys_hat = dimensionless_regression(xs[:,:-1], ys, x_u[:-1,:], y_u, xs_test[:,:-
         1])
         plt.scatter(xs_test[:, 0], ys_test, marker=".",  alpha=.75, label="truth")
         plt.scatter(xs_test[:, 0], ys_hat, marker=".", alpha=.75, label="predictions")
         plt.ylim(1.e6, 1.e17)
         plt.loglog()
         plt.legend()
         plt.xlabel("wavelength")
         plt.ylabel("intensity")
         plt.title("This is pretty bad")
```

```
Out[20]: Text(0.5, 1.0, 'This is pretty bad')
```



Perform a units-covariant regression with Planck's constant:

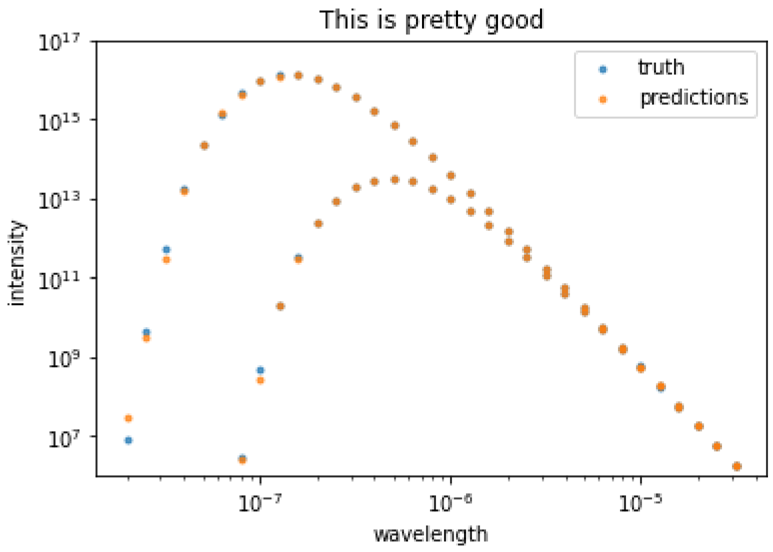
```
In [21]: vv, us = integer_solve(x_u.T, y_u)
         print(vv)
         print(us)
```

```
[-4  1  1  1  0]
[[-1 -1  1 -1  1]]
```

In [22]:
```
ys_hat = dimensionless_regression(xs, ys, x_u, y_u, xs_test)
plt.scatter(xs_test[:, 0], ys_test, marker=".", alpha=.75, label="truth")
plt.scatter(xs_test[:, 0], ys_hat, marker=".", alpha=.75, label="predictions")
rms = np.sqrt(np.nanmean((np.log(ys_test) - np.log(ys_hat))**2))
print("rms", rms)
plt.ylim(1.e6, 1.e17)
plt.loglog()
plt.legend()
plt.xlabel("wavelength")
plt.ylabel("intensity")
plt.title("This is pretty good")
```

```
rms 0.20029584879963894
```

Out[22]:
```
Text(0.5, 1.0, 'This is pretty good')
```



We try lots of dimensional constants in place of Planck's constant. Which ones work?

In [23]:
```
#we are using brute force
#all possible units of the constants we are searching
try_us = list(it.product([-2,-1,0,1,2], repeat=4))
print(try_us[:5])
```

```
[(-2, -2, -2, -2), (-2, -2, -2, -1), (-2, -2, -2, 0), (-2, -2, -2, 1), (-2, -
2, -2, 2)]
```

In [24]:
```
#finds every possible dimensional constant that leads to a good regression

for try_u in try_us:
    xs_aux = xs.copy()
    xs_test_aux = xs_test.copy()
    xs_aux[:,-1] = 1.0
    xs_test_aux[:,-1] = 1.0
    x_u_aux = x_u.copy()
    x_u_aux[-1] = try_u
    vv, us = integer_solve(x_u_aux.T, y_u)
    #print(vv)
    #print(us)
    assert len(us)>0
    XX_aux, XX_aux_test = create_dimensionless_features(vv, us, xs_aux, xs_tes
t_aux)
    YY_aux = create_dimensionless_labels(vv, xs_aux, ys)
    foo = np.nanmedian(XX_aux[:,1])
    #finds the scaling to make the median of the dimensionless features unity
    factor = foo**(1.0 / us[0][-1])
    xs_aux[:,-1] = 1.0 / factor
    xs_test_aux[:,-1] = 1.0 / factor
    XX_aux, XX_aux_test = create_dimensionless_features(vv, us, xs_aux, xs_tes
t_aux)
    YY_aux = create_dimensionless_labels(vv, xs_aux, ys)
    ys_hat = dimensionless_regression(xs_aux, ys, x_u_aux, y_u, xs_test_aux)
    rms = np.sqrt(np.nanmean((np.log(ys_test) - np.log(ys_hat))**2))
    if rms<1:
        print(xs_aux[0,-1], try_u, rms)
    #assert False
```

```
1.0570829621344583e+105 (-2, -2, -2, -2) 0.25567940888234153
2.734642509169893e+94 (-2, -2, -1, -1) 0.24971608263245862
7.069564172748589e+83 (-2, -2, 0, 0) 0.2415520057623676
1.8263511238705562e+73 (-2, -2, 1, 1) 0.16268477874189935
8.198251995753287e+102 (-2, -1, -2, -1) 0.2497160826324932
2.1194020203370332e+92 (-2, -1, -1, 0) 0.2415520057623676
5.475262925962159e+81 (-2, -1, 0, 1) 0.16268477874189935
6.353807411670043e+100 (-2, 0, -2, 0) 0.2415520057623667
1.6414425307704654e+90 (-2, 0, -1, 1) 0.16268477874189935
4.920920909654177e+98 (-2, 1, -2, 1) 0.16268477874189935
4.2009009071295606e+54 (-1, -2, -1, -2) 0.24971608263240164
1.086011734503654e+44 (-1, -2, 0, -1) 0.24155200576237607
2.805602585083024e+33 (-1, -2, 1, 0) 0.16268477874192455
1.259398408762799e+63 (-1, -1, -2, -2) 0.24971608263244757
3.2557812730369343e+52 (-1, -1, -1, -1) 0.24155200576236857
8.410984951531927e+41 (-1, -1, 0, 0) 0.1626847787419094
5.6017884633994514e+20 (-1, -1, 2, 2) 0.17531062577418055
9.760586705541102e+60 (-1, 0, -2, -1) 0.2415520057623685
2.521549852820764e+50 (-1, 0, -1, 0) 0.16268477874190687
1.6793739326385563e+29 (-1, 0, 1, 2) 0.17531062577417436
7.559416283466688e+58 (-1, 1, -2, 0) 0.16268477874191875
5.034636391668421e+37 (-1, 1, 0, 2) 0.17531062577418569
1.5093460189945244e+46 (-1, 2, -1, 2) 0.17531062577416182
16683.086236433377 (0, -2, 0, -2) 0.24155200576237307
4.3099082988723293e-07 (0, -2, 1, -1) 0.16268477874191323
5001463429846.325 (0, -1, -1, -2) 0.2415520057623746
129.20780026735324 (0, -1, 0, -1) 0.16268477874191203
8.605350848797497e-20 (0, -1, 2, 1) 0.1753106257741634
1.4994010152307382e+21 (0, 0, -2, -2) 0.2415520057623741
38735524034.92283 (0, 0, -1, -1) 0.16268477874191264
2.579819282913385e-11 (0, 0, 1, 1) 0.17531062577417236
6.650855286662904e-22 (0, 0, 2, 2) 0.21012642769932655
1.1612617962347577e+19 (0, 1, -2, -1) 0.16268477874190537
0.007734103640204 (0, 1, 0, 1) 0.17531062577417014
1.9938762541909644e-13 (0, 1, 1, 2) 0.21012642769932582
2318625.9407235016 (0, 2, -1, 1) 0.1753106257741709
5.977490631917411e-05 (0, 2, 0, 2) 0.2101264276993254
6.620791427642233e-47 (1, -2, 1, -2) 0.16268477874190215
1.984863335998191e-38 (1, -1, 0, -2) 0.16268477874192644
1.3219360872824093e-59 (1, -1, 2, 0) 0.1753106257741661
5.9504705829298104e-30 (1, 0, -1, -2) 0.1626847787419226
3.9630646892529257e-51 (1, 0, 1, 0) 0.17531062577416592
1.0216905468719191e-61 (1, 0, 2, 1) 0.2101264276993185
1.783906202313212e-21 (1, 1, -2, -2) 0.16268477874192422
1.1880969044041391e-42 (1, 1, 0, 0) 0.17531062577416445
3.062951203620964e-53 (1, 1, 1, 1) 0.21012642769931913
7.890923741225875e-64 (1, 1, 2, 2) 0.2478626167913051
3.5618249131350745e-34 (1, 2, -1, 0) 0.1753106257741744
9.18249670067586e-45 (1, 2, 0, 1) 0.2101264276993284
2.365639424272658e-55 (1, 2, 1, 2) 0.24786261679130367
2.0307307041452025e-99 (2, -1, 2, -1) 0.1753106257741621
6.087977493317602e-91 (2, 0, 1, -1) 0.1753106257741621
1.5694997539048723e-101 (2, 0, 2, 0) 0.21012642769931802
1.82512973697036e-82 (2, 1, 0, -1) 0.1753106257741621
4.705241890535362e-93 (2, 1, 1, 0) 0.21012642769931822
1.21218728193721e-103 (2, 1, 2, 1) 0.24786261679131052
5.471601300152369e-74 (2, 2, -1, -1) 0.1753106257741621
```

```
1.4105960318481614e-84 (2, 2, 0, 0) 0.21012642769931822
3.634046048082946e-95 (2, 2, 1, 1) 0.24786261679131225
9.355753860084212e-106 (2, 2, 2, 2) 0.2439655155754762
```

In [ ]:

```
#check this out: we get Planck's constat to within a factor of 2
#it is this line: 3.5618249131350745e-34 (1, 2, -1, 0) 0.1753106257741744
#todo: interpret the results of the above
```