cmax

Jelle Aalbers and Chris Tunnell

April 11, 2014

Part I

$\bar{C}_{\mathbf{max}}(1-\alpha, \mu)$

Computing $\bar{C}_{\max}(C, \mu)$ for optimum interval calculation, where μ is the number of expected events and $1-\alpha$ is how frequently you reject the null hypothesis when it is true.The single-event energy spectrum, that is, the probability density function which tells us which energy depositions are likely to occur, is independent of the chosen WIMP model – we always expect a simple exponential recoil spectrum.

The number of dark matter events detected does depend on the WIMP mass and cross-section. We know, however, that it must follow a Poisson distribution, which leaves the Poisson mean (which equals the expected number of events) as the only parameter left to estimate. From an upper limit on this mean, an upper limit curve in the dark matter mass – cross-section plane can be computed.

- A list_of_energies list of reconstructed energy depositions of single events (from here on simply 'energies'), either measured during some run of an actual detector, or generated using Monte Carlo.)
- An interval is an interval in energy space.
- The size of an interval is the fraction of energies expected in that interval. Clearly, this depends on which energy spectrum we assume, but is independent of the Poisson mean we are trying to constrain. By definition this is a number between 0 and 1.
- The K-largest interval of a run is the largest interval containing K events in that run. Recall our definition of size: a 'large' interval is one which is unusually empty in that run. Clearly k-largest intervals will terminate at (or technically, just before) an observed energy, or at one of the boundaries of our energy space. Again, which interval in a run is the k–largest, depends on our energy spectrum, but not on our Poisson mean.
- The extremeness of a K-largest interval is the probability of finding the K-largest interval in a run to be smaller. This clearly does depend on the Poisson mean: if we expect very few events, large gap sizes are more likely. Clearly extremeness is a number between 0 and 1; values close to 1 indicate unusually large intervals, that is, usually large (almost-)empty regions in the measured energies. For example, if the extremeness of a k-largest interval in a run is 0.8, that means that 80% of runs have k-largest intervals which are smaller than the k-largest interval in this run.
- The optimum interval statistic of a run is extremity of the most extreme k-largest interval in a run.
- The extremeness of the optimum interval statistic is the probability of finding a lower optimum interval statistic, that is, of finding the optimum interval in a run to be less extreme.

The max gap method rejects a theory (places a mean outside the upper limit) based on a run if the 0-largest interval (the largest gap) is too extreme.

The optimum interval method rejects a theory based on a run if the optimum interval statistic is too large.

- The energy cumulant $\epsilon(E)$ is the fraction of energies expected below the energy E. Whatever the (1-normalized) energy distribution dN/dE, $dN/d\epsilon$ is uniform[0,1], where 0 and 1 correspond to the boundaries of our experimental range.

In [16]:
```
import functools
from scipy.optimize import brenth
import matplotlib.pyplot as plt
import numpy as np
import pickle
import os
```

In [17]:
```
def kLargestIntervals(list_of_energies, spectrumCDF = lambda x: x):
    """
    Returns a list of the sizes of the K-largest intervals in that run according to th
    That is, kLargestIntervals(...)[i] is the size of the largest interval containing

    * Transform energies to energy cumulants
    * Add events at 0 and 1
    * Foreach k, compute interval sizes, take max
    """
    answer = {}

    list_of_energies.sort()

    energy_cumulants = spectrumCDF(list_of_energies)

    for interval_size in range(len(energy_cumulants)):
        if (1 + interval_size) >= len(energy_cumulants):
            continue

        temp_data = energy_cumulants.copy()
        gap_sizes = temp_data[(1+interval_size):] - temp_data[0:-1*(1 + interval_size)

        answer[interval_size] = np.max(gap_sizes)

    return answer

assert kLargestIntervals(np.array([0.0, 0.1, 0.2, 0.84, 0.85]))[0] == (0.84 - 0.2)   #
assert kLargestIntervals(np.array([0.0, 0.1, 0.2, 0.84, 0.85]))[2] == (0.84 - 0.0)   #
assert kLargestIntervals(np.array([0.85, 0.0, 0.1, 0.84, 0.2]))[2] == (0.84 - 0.0)   #
```

In [18]:
```
def extremenessOfInterval(x, k, mu):
    """
    Returns the extremeness of a k-largest interval of size, if the poisson mean is mu

    (Number of itvSizes[mu][k] smaller than size) / mcTrials[mu]

    x - also size in above comment
    k - gap (rename k)
    """
    # [0] is because where returns list, where [0] is answer
    if k not in itvSizes[mu]:
        return 0

    return np.where(itvSizes[mu][k] < x)[0].size / mcTrials[mu]
```

In [19]:
```
def optimumItvStatistic(list_of_energies, mu, spectrumCDF = lambda x: x):
    """
    Returns the optimum interval statistic of the run.

    Max of extremenssOfInterval's
    """
    return np.max([extremenessOfInterval(x, k, mu) for k, x in kLargestIntervals(list_
```

In [20]:
```python
def extremenessOfOptItvStat(stat, mu):
    """
    Returns the extremeness of the optimum interval statistic stat, given mu

    (Number of optItvs[mu] smaller than stat) / mcTrials[mu]
    """
    return np.where(optItvs[mu] < stat)[0].size / mcTrials[mu]
```

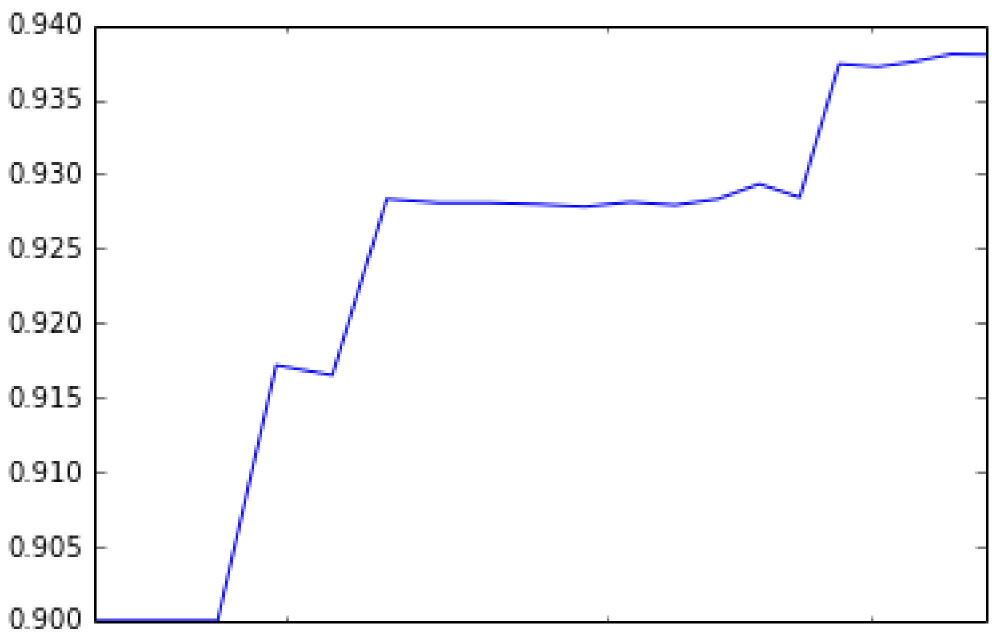
In [26]:
```python
def optItvUpperLimit(list_of_energies, c, spectrumCDF = lambda x: x,
                     n = 1000):
    """
    Returns the c- confidence upper limit on mu using optimum interval

    For which mu is extremenessOfOptItvStat( optimumItvStatistic(run), mu ) = c

    c - e.g., 0.9
    """
    def f(mu, list_of_energies, c, spectrumCDF, n):
        generate_table(mu, n)
        x = optimumItvStatistic(list_of_energies, mu, spectrumCDF)
        prob = extremenessOfOptItvStat(x, mu)
        return prob - c

    mu = 0

    for mu in np.arange(10, 2 * list_of_energies.size):
        if f(mu, list_of_energies, c, spectrumCDF, n) > 0:
            print('Found seed mu=%f' % mu)
            break

    try:
        xsec = brenth(f, mu - 5, mu + 5,
                      args=(list_of_energies, c, spectrumCDF, n),
                      xtol=1e-2)
        print('Improved xsec:', xsec)
    except:
        print("ERROR: could not minimize", mu)
        return mu
    return xsec
```

In [22]:
```python
def generate_trial_experiment(mu, n):
    trials = []

    for index in range(n):
        this_mu = np.random.poisson(mu)

        rand_numbers = np.random.random(size=this_mu)
        rand_numbers = np.append(rand_numbers, [0.0, 1.0])
        rand_numbers.sort()
        trials.append(rand_numbers)

    return trials
```

1 Monte Carlo for populating itvSizes[μ][k] and optItvs[μ]

In [36]:
```
def get_filename():
    return 'saved_intervals.p'

def load_table_from_disk():
    global itvSizes
    global optItvs
    global mcTrials

    if os.path.exists(get_filename()):
        f = open(get_filename(), 'rb')
        itvSizes = pickle.load(f)
        optItvs = pickle.load(f)
        mcTrials = pickle.load(f)
        f.close()


def write_table_to_disk():
    f = open(get_filename(), 'wb')
    pickle.dump(itvSizes, f)
    pickle.dump(optItvs, f)
    pickle.dump(mcTrials, f)
    f.close()

itvSizes = {}
optItvs = {}
mcTrials = {}
load_table_from_disk()

def generate_table(mu, n):
    """    #Generate trial runs"""
    if mu in mcTrials and mcTrials[mu] >= n:
        return

    print("Generating", mu)

    mcTrials[mu] = n
    trials = generate_trial_experiment(mu, mcTrials[mu])

    itvSizes[mu] = {}
    optItvs[mu] = []

    for trial in trials:
        intermediate_result = kLargestIntervals(trial)

        for k, v in intermediate_result.items():
            if k not in itvSizes[mu]:
                itvSizes[mu][k] = []

            itvSizes[mu][k].append(v)

    # Numpy-ize it
    for k, array in itvSizes[mu].items():
        itvSizes[mu][k] = np.array(array)

    for trial in trials:
        optItvs[mu].append(optimumItvStatistic(trial, mu))

    # Numpy-ize it
    optItvs[mu] = np.array(optItvs[mu])




def cache_values(my_max=200, n=100):
    for i in range(3, my_max):
        generate_table(i, n)
    write_table_to_disk()
```

In [52]:
```python
def plot_something():
    x, y = [], []

    for mu in np.linspace(3.5, 6.5, 20):
        generate_table(mu, 10000)
        x.append(mu)

        a = brenth(lambda x: extremenessOfOptItvStat(x, mu) - 0.9,
                   0,
                   1,
                   xtol=1e-2)

        y.append(a)

    plt.plot(x,y)
    plt.xscale('log')
    plt.xlim(3.5, 6.5)

#plot_something()
```



In [71]:
```python
def simple_test_uniform():
    test_list_of_energies = generate_trial_experiment(mu=100, n=1)[0]
    print(len(test_list_of_energies))
    answer = optItvUpperLimit(test_list_of_energies, 0.9)
```

```
107
testing mu=3.000000
... 0.000000, so continuing...
testing mu=4.000000
... 0.000000, so continuing...
testing mu=5.000000
... 0.000000, so continuing...
testing mu=6.000000
... 0.000000, so continuing...
testing mu=7.000000
```

```
... 0.000000, so continuing...
testing mu=8.000000
... 0.000000, so continuing...
testing mu=9.000000
... 0.000000, so continuing...
testing mu=10.000000
... 0.000000, so continuing...
testing mu=11.000000
... 0.000000, so continuing...
testing mu=12.000000
... 0.000000, so continuing...
testing mu=13.000000
... 0.000000, so continuing...
testing mu=14.000000
... 0.000000, so continuing...
testing mu=15.000000
... 0.000000, so continuing...
testing mu=16.000000
... 0.000000, so continuing...
testing mu=17.000000
... 0.000000, so continuing...
testing mu=18.000000
... 0.000000, so continuing...
testing mu=19.000000
... 0.000000, so continuing...
testing mu=20.000000
... 0.000000, so continuing...
testing mu=21.000000
... 0.000000, so continuing...
testing mu=22.000000
... 0.000000, so continuing...
testing mu=23.000000
Generating 23
... 0.000000, so continuing...
testing mu=24.000000
Generating 24
... 0.000000, so continuing...
testing mu=25.000000
Generating 25
... 0.000000, so continuing...
testing mu=26.000000
Generating 26
... 0.000000, so continuing...
testing mu=27.000000
Generating 27
... 0.000000, so continuing...
testing mu=28.000000
Generating 28
... 0.000000, so continuing...
testing mu=29.000000
Generating 29
... 0.000000, so continuing...
testing mu=30.000000
Generating 30
```

```
... 0.000000, so continuing...
testing mu=31.000000
Generating 31
... 0.000000, so continuing...
testing mu=32.000000
Generating 32
... 0.000000, so continuing...
testing mu=33.000000
Generating 33
... 0.000000, so continuing...
testing mu=34.000000
Generating 34
... 0.000000, so continuing...
testing mu=35.000000
Generating 35
... 0.000000, so continuing...
testing mu=36.000000
Generating 36
... 0.000000, so continuing...
testing mu=37.000000
Generating 37
... 0.000000, so continuing...
testing mu=38.000000
Generating 38
... 0.000000, so continuing...
testing mu=39.000000
Generating 39
... 0.000000, so continuing...
testing mu=40.000000
Generating 40
... 0.000000, so continuing...
testing mu=41.000000
Generating 41
... 0.000000, so continuing...
testing mu=42.000000
Generating 42
... 0.000000, so continuing...
testing mu=43.000000
Generating 43
... 0.000000, so continuing...
testing mu=44.000000
Generating 44
... 0.000000, so continuing...
testing mu=45.000000
Generating 45
... 0.000000, so continuing...
testing mu=46.000000
Generating 46
... 0.000000, so continuing...
testing mu=47.000000
Generating 47
... 0.000000, so continuing...
testing mu=48.000000
Generating 48
```

```
... 0.000000, so continuing...
testing mu=49.000000
Generating 49
... 0.000000, so continuing...
testing mu=50.000000
Generating 50
... 0.000000, so continuing...
testing mu=51.000000
Generating 51
... 0.001000, so continuing...
testing mu=52.000000
Generating 52
... 0.000000, so continuing...
testing mu=53.000000
Generating 53
... 0.000000, so continuing...
testing mu=54.000000
Generating 54
... 0.002000, so continuing...
testing mu=55.000000
Generating 55
... 0.001000, so continuing...
testing mu=56.000000
Generating 56
... 0.002000, so continuing...
testing mu=57.000000
Generating 57
... 0.003000, so continuing...
testing mu=58.000000
Generating 58
... 0.000000, so continuing...
testing mu=59.000000
Generating 59
... 0.005000, so continuing...
testing mu=60.000000
Generating 60
... 0.002000, so continuing...
testing mu=61.000000
Generating 61
... 0.007000, so continuing...
testing mu=62.000000
Generating 62
... 0.006000, so continuing...
testing mu=63.000000
Generating 63
... 0.009000, so continuing...
testing mu=64.000000
Generating 64
... 0.013000, so continuing...
testing mu=65.000000
Generating 65
... 0.012000, so continuing...
testing mu=66.000000
Generating 66
```

```
... 0.031000, so continuing...
testing mu=67.000000
Generating 67
... 0.036000, so continuing...
testing mu=68.000000
Generating 68
... 0.034000, so continuing...
testing mu=69.000000
Generating 69
... 0.038000, so continuing...
testing mu=70.000000
Generating 70
... 0.043000, so continuing...
testing mu=71.000000
Generating 71
... 0.054000, so continuing...
testing mu=72.000000
Generating 72
... 0.059000, so continuing...
testing mu=73.000000
Generating 73
... 0.052000, so continuing...
testing mu=74.000000
Generating 74
... 0.072000, so continuing...
testing mu=75.000000
Generating 75
... 0.094000, so continuing...
testing mu=76.000000
Generating 76
... 0.101000, so continuing...
testing mu=77.000000
Generating 77
... 0.121000, so continuing...
testing mu=78.000000
Generating 78
... 0.144000, so continuing...
testing mu=79.000000
Generating 79
... 0.159000, so continuing...
testing mu=80.000000
Generating 80
... 0.177000, so continuing...
testing mu=81.000000
Generating 81
... 0.192000, so continuing...
testing mu=82.000000
Generating 82
... 0.221000, so continuing...
testing mu=83.000000
Generating 83
... 0.232000, so continuing...
testing mu=84.000000
Generating 84
```

```
... 0.268000, so continuing...
testing mu=85.000000
Generating 85
... 0.257000, so continuing...
testing mu=86.000000
Generating 86
... 0.270000, so continuing...
testing mu=87.000000
Generating 87
... 0.310000, so continuing...
testing mu=88.000000
Generating 88
... 0.343000, so continuing...
testing mu=89.000000
Generating 89
... 0.391000, so continuing...
testing mu=90.000000
Generating 90
... 0.455000, so continuing...
testing mu=91.000000
Generating 91
... 0.418000, so continuing...
testing mu=92.000000
Generating 92
... 0.467000, so continuing...
testing mu=93.000000
Generating 93
... 0.492000, so continuing...
testing mu=94.000000
Generating 94
... 0.523000, so continuing...
testing mu=95.000000
Generating 95
... 0.528000, so continuing...
testing mu=96.000000
Generating 96
... 0.536000, so continuing...
testing mu=97.000000
Generating 97
... 0.572000, so continuing...
testing mu=98.000000
Generating 98
... 0.576000, so continuing...
testing mu=99.000000
Generating 99
... 0.644000, so continuing...
testing mu=100.000000
Generating 100
... 0.644000, so continuing...
testing mu=101.000000
Generating 101
... 0.704000, so continuing...
testing mu=102.000000
Generating 102
```

```
... 0.663000, so continuing...
testing mu=103.000000
Generating 103
... 0.698000, so continuing...
testing mu=104.000000
Generating 104
... 0.708000, so continuing...
testing mu=105.000000
Generating 105
... 0.726000, so continuing...
testing mu=106.000000
Generating 106
... 0.748000, so continuing...
testing mu=107.000000
Generating 107
... 0.772000, so continuing...
testing mu=108.000000
Generating 108
... 0.802000, so continuing...
testing mu=109.000000
Generating 109
... 0.806000, so continuing...
testing mu=110.000000
Generating 110
... 0.832000, so continuing...
testing mu=111.000000
Generating 111
... 0.844000, so continuing...
testing mu=112.000000
Generating 112
... 0.832000, so continuing...
testing mu=113.000000
Generating 113
... 0.854000, so continuing...
testing mu=114.000000
Generating 114
... 0.877000, so continuing...
testing mu=115.000000
Generating 115
... 0.838000, so continuing...
testing mu=116.000000
Generating 116
... 0.899000, so continuing...
testing mu=117.000000
Generating 117
... 0.882000, so continuing...
testing mu=118.000000
Generating 118
Found 118.000000 -> 0.902000
```