This Notebook:

"A Maximum Likelihood Bunching Estimator of the Elasticity of Taxable Income" written by Thomas Aronsson, Katharina Jenderny and Gauthier Lanot, accepted for publication in the Journal of Applied Econometrics, MS-13011

This notebook applies the maximum likelihood approach to the estimation of the ETI using Swedish evidence.

This file deals with "ETI_hist_FInk_taxable1000_bin100_2019.dta".

We have first a bunch of declaration (functions etc...), then a presentation of the data, and then some calculations. The calculations are slightly more extensive than those presented in the text of the paper. This may convince the reader that we have carried out due diligence...

In [1]:
```
#Parallelism().set(nproc=6)
#print(Parallelism())
```

In [2]:
```
import numpy  as np
import pandas as pd
import os
```

In [3]:
```
tt=os.getcwd()
show(tt)
```

In [4]:
```
#x,t,y,r = var('x,t,y,r')
x = var('x')
pdfn(x)=1/sqrt(2.*pi)*exp(-1/2.*x*x)
##cdfn(x) = integrate(pdfn(t),(t,-15,x))
#cdfn(x)=0.5*(1.+erf(x/sqrt(2.)))
#assume(abs(r)<=1)
#assume(abs(t)<=1)
#pdfn2(x,y,t) = pdfn(x)*pdfn((y-t*x)/sqrt(1.-t^2))/sqrt(1.-t^2)
```

In [5]:
```
%%cython
clib: m

cdef extern from "math.h":
    double sin(double x)
    double exp(double x)
    double log(double x)
    double sqrt(double x)
    double erf(double x)
    double tanh(double x)
    double pow(double x, double y)
    double fmax(double x, double y)
    double fmin(double x, double y)


def grd(faf,x0,*args):
    x00 = [j for j in x0]
    r = len(x00)
    faf0 = faf(x00,*args)
    grd0 = [0.] * r
    petit = 1.3e-6
    for i in range(r):
        x = [j for j in x0]
        x[i]=x[i]*(1.+petit)+petit
        grd0[i] = (faf(x,*args)-faf0)/(x[i]-x00[i])

    return grd0

# central gradient...
def grdcd(faf,x0,*args):
    x00 = [j for j in x0]
    r = len(x00)

    grd0 = [0.] * r
    petit = 1.3e-6
    for i in range(r):
        x = [j for j in x0]
        xi = x[i]
        x[i] = (1.0+petit)*xi+petit
        fafp = faf(x,*args)
        dx = 2.0*petit*xi+2*petit
        x[i] = (1.0-petit)*xi-petit
        fafm = faf(x,*args)
        grd0[i] = (fafp-fafm)/(dx)

    return grd0

# central gradient...long version
# faf returns a 'vector' of size nfaf
def grdcd_long(faf,x0,nfaf,*args):
    x00 = [j for j in x0]
    r = len(x00)
    #import pdb; pdb.set_trace()
    grd0 = [[0.]*nfaf] * r
    petit = 1.3e-6
    for i in range(r):
        x = [j for j in x0]
        xi = x[i]
        x[i] = (1.0+petit)*xi+petit
        fafp = faf(x,*args)
```

```
        dx = 2.0*petit*xi+2*petit
        x[i] = (1.0-petit)*xi-petit
        fafm = faf(x,*args)
        grd0[i] = [ (fafp[j]-fafm[j])/(dx) for j in range(nfaf)]

    return grd0


#central hessian...
def hescd(faf,x0,*args):
    x00 = [j for j in x0]
    r = len(x00)
    faf0 = faf(x00,*args)
    dh0  = [0.] * r
    hes0 = [[0. for j in range(r)] for l in range(r)]
    petit = 1.03e-4
    for i in range(r):
        dh0[i] = (x00[i]*(1.+petit)+petit)-x00[i]

    #print(hes0)
    #petit = 1.3e-5
    for i in range(r):
        for k in range(i,r):
            if i==k:
                x1 = [j for j in x00]
                x1[i] = x1[i] + 2*dh0[i]
                fafp2 = faf(x1,*args)
                x1 = [j for j in x00]
                x1[i] = x1[i] + dh0[i]
                fafp1 = faf(x1,*args)
                x1 = [j for j in x00]
                x1[i] = x1[i] - dh0[i]
                fafm1 = faf(x1,*args)
                x1 = [j for j in x00]
                x1[i] = x1[i] - 2*dh0[i]
                fafm2 = faf(x1,*args)
                hes0[i][k]=(-fafp2+16*fafp1-30*faf0+16*fafm1-fafm2)/12/dh0[i]/dh
            else:
                x1 = [j for j in x00]
                x1[i] = x1[i] + dh0[i]
                x1[k] = x1[k] + dh0[k]
                fafp1p1 = faf(x1,*args)
                x1 = [j for j in x00]
                x1[i] = x1[i] + dh0[i]
                x1[k] = x1[k] - dh0[k]
                fafp1m1 = faf(x1,*args)
                x1 = [j for j in x00]
                x1[i] = x1[i] - dh0[i]
                x1[k] = x1[k] + dh0[k]
                fafm1p1 = faf(x1,*args)
                x1 = [j for j in x00]
                x1[i] = x1[i] - dh0[i]
                x1[k] = x1[k] - dh0[k]
                fafm1m1 = faf(x1,*args)
                hes0[i][k]=(fafp1p1-fafp1m1-fafm1p1+fafm1m1)/4/dh0[i]/dh0[k]
                hes0[k][i]=hes0[i][k]

    return hes0
```

In [6]:
```
# predicts the distribution
```

In [7]:
```
#hessian, for var cov+ checking optimum...
# hh0b=matrix(RDF,hes(lik0,q00b))
# hh0b.eigenvalues()
# hh0ib = hh0b.inverse()
# sqrtN =sqrt(sum(n[1] for n in nobs))
# results log-likelihood estimates...
# res_lik_print0(q00b,hh0ib,sqrtN,sum(n[1] for n in nobs),"Likelihood Estimates,
    tse = [0 for n in range(len(dens))]
    td  = [[0,0] for n in range(len(dens))]
    t   = [[0,0] for n in range(len(dens))]
    tu  = [[0,0] for n in range(len(dens))]
```

In [8]:
```
# this is for UMea...what is the swedish average...?
taxinfo = [ [2001,0.677,0.477,252000],
          [2002,0.677,0.477,273800],
          [2003,0.677,0.477,284300],
          [2004,0.677,0.477,291800],
          [2005,0.669,0.469,298600],
          [2006,0.669,0.469,306000],
          [2007,0.669,0.469,316700],
          [2008,0.669,0.469,328800],
          [2009,0.669,0.469,367600],
          [2010,0.669,0.469,372100],
          [2011,0.669,0.469,383000],
          [2012,0.669,0.469,401100],
          [2013,0.669,0.469,413200],
          [2014,0.664,0.464,420800],
          [2015,0.664,0.464,430200],
          [2016,0.6635,0.4635,430200],
          [2017,0.6585,0.46585,438900],
          [2018,0.6585,0.46585,455300],
          [2019,0.6585,0.46585,490700]
        ]
    print(ttl)
```

In [ ]:
```
    if len(qq)==3:
```

In [9]:
```
# so how do we select a symmetric range around the kink...?
# we check the distance from the kink to the min or the max,
# and select only if the difference from the kink
# is less that the minimum between those 2 differences... simple...
year = 2019
Info_in_year = flatten([ nn for nn in taxinfo if nn[0]==year])
Kink = Info_in_year[3]/1000.
lnk=ln(Kink).n()
```

In [10]:
```
os.chdir(tt)
tt2=os.getcwd()
show(tt2)

    return qq,se,ll
```

Data:

Here comes the data.

In [11]:
```
import numpy  as np
import pandas as pd
import os
```

In [12]:
```
#newd="D://Umeå universitet//ETI - General//JAE Revisions//scb data//hist_data"
#newd = "/Users/gauthierlanot/Library/CloudStorage/OneDrive-SharedLibraries-Umeå
#newd = "//Users/gauthierlanot/Downloads/"
#ETI - General\\JAE Revisions\\scb data"
newd = "D:\ETIprojectfiles\JAE revisions"
show(newd)
```

```
<>:5: DeprecationWarning: invalid escape sequence \E
<>:5: DeprecationWarning: invalid escape sequence \E
<>:5: DeprecationWarning: invalid escape sequence \E
<ipython-input-12-69121d137286>:5: DeprecationWarning: invalid escape sequence \E
  newd = "D:\ETIprojectfiles\JAE revisions"
```

In [13]:
```
# loading data from stata file
os.chdir(newd)
filename = "ETI_hist_FInk_taxable1000_bin100_2019.dta"
```

In [14]:
```
delta, STEP_BIN = var('delta STEP_BIN')
delta = (0.1*.99999)
STEP_BIN= 0.1
```

In [15]:
```
delta
```

Out[15]:
```
0.0999990000000000
```

In [16]:
```
dd0 = pd.read_stata(filename)
```

In [17]:
```
dd = dd0.to_numpy()
```

In [18]:
```
dd = [(nn[0],nn[1],nn[2],nn[3]/1000.,nn[4],nn[5],nn[6]/1000.) for nn in dd]
```

In [19]:
```
# ...and here we find that the data is not centered on the kink!
# typical last minute job! no attention to details!
#df0
```

In [20]:
```
dd[0:10]
```

Out[20]:
```
[(1.0, 17.0, 2019.0, 422.3, 422300.0, 0.0, 490.7),
 (2.0, 25.0, 2019.0, 422.4, 422400.0, 1.0, 490.7),
 (3.0, 23.0, 2019.0, 422.5, 422500.0, 1.0, 490.7),
 (4.0, 21.0, 2019.0, 422.6, 422600.0, 1.0, 490.7),
 (5.0, 15.0, 2019.0, 422.7, 422700.0, 1.0, 490.7),
 (6.0, 28.0, 2019.0, 422.8, 422800.0, 1.0, 490.7),
 (7.0, 20.0, 2019.0, 422.9, 422900.0, 1.0, 490.7),
 (8.0, 19.0, 2019.0, 423.0, 423000.0, 1.0, 490.7),
 (9.0, 22.0, 2019.0, 423.1, 423100.0, 1.0, 490.7),
 (10.0, 21.0, 2019.0, 423.2, 423200.0, 1.0, 490.7)]
```

In [21]:
```
# so how do we select a symmetric range around the kink...?
# we check the distance from the kink to the min or the max,
# and select only if the difference from the kink
# is less that the minimum between those 2 differences... simple...
year = 2019
Info_in_year = flatten([ nn for nn in taxinfo if nn[0]==year])
Kink = Info_in_year[3]/1000.
lnk=ln(Kink).n()
```

In [22]:
```
Info_in_year
```

Out[22]:
```
[2019, 0.658500000000000, 0.465850000000000, 490700]
```

In [23]:
```
Kink
```

Out[23]:
```
490.700000000000
```

In [24]:
```
dd[0]
```

Out[24]:
```
(1.0, 17.0, 2019.0, 422.3, 422300.0, 0.0, 490.7)
```

In [25]:
```
#distmin2k = df0.kink[[0]].array[0]-df0.wh[[0]].array[0]
#distk2max = df0.wh[[len(df0)-1]].array[0]-df0.kink[[0]].array[0]
#topdistfromk = min(distmin2k,distk2max)+0.001
#distmin2k,distk2max,topdistfromk
distmin2k = dd[0][6]-dd[0][3]
distk2max = dd[-1][3]-dd[-1][6]
topdistfromk = min(distmin2k,distk2max)+0.001
distmin2k,distk2max,topdistfromk
```

Out[25]:
```
(68.39999999999998, 79.59999999999997, 68.40099999999998)
```

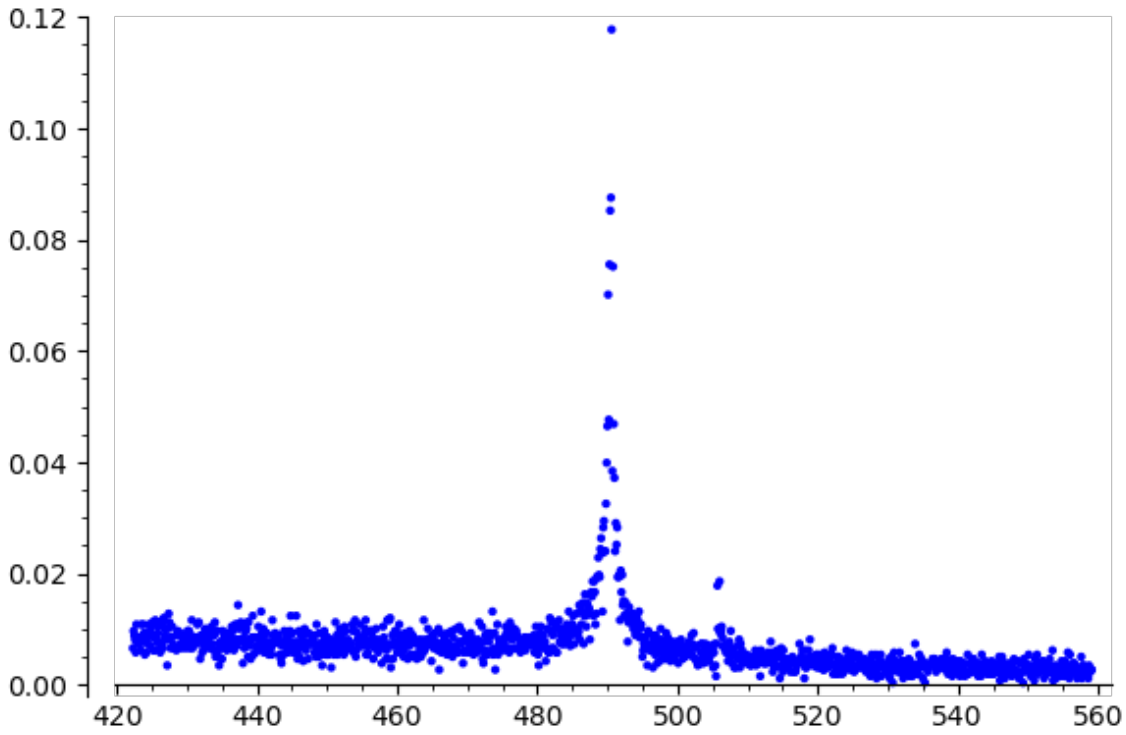
In [26]:
```
def mak_density(datai,Kinki,topdistfromki,stp_bn,dlt):
    df = [nn for nn in datai if abs(nn[3]-Kinki)<=topdistfromki ]
    df_yy_cnt = [ (dd[3],dd[1]) for dd in df]
    Nobs = int(sum([nn[1] for nn in df]))
    #show(Nobs)
    dens = [(nn[3],nn[1]/Nobs) for nn in df]
    dens = np.array(dens)
    #show(len(dens))

    #densa = [(nn[3],nn[1]/Nobs/0.1) for nn in df if not(((ln(nn[3])+0.1/252.)>=

    densa = [(nn[3],nn[1]/Nobs/stp_bn) for nn in df if nn[3]<(Kinki-dlt)]+ \
            [(nn[3],nn[1]/Nobs) for nn in df if (nn[3]>=(Kinki-dlt)) and (nn[3])
            [(nn[3],nn[1]/Nobs/stp_bn) for nn in df if nn[3]>Kinki]

    densb = [(nn[3],nn[1]/Nobs/stp_bn) for nn in df]
    return {
            'df':df,
            'df_yy_cnt':df_yy_cnt,
            'Nobs':Nobs,
            'dens':dens,
            'densa':densa,
            'densb':densb}
```

In [27]:
```
mkd=mak_density(dd,Kink,topdistfromk,STEP_BIN,delta)
```

In [ ]:

In [28]:
```
list_plot(mkd['densa'])
```

Out[28]:



In [29]:
```
#sum(dens[n][1] for n in range(0,len(dens)) if n!= KinkPos)+dens[KinkPos][1]
```

In [30]:
```
os.chdir(tt)
tt2=os.getcwd()
show(tt2)
```

In [31]:
```
490.7*0.85, 490.7*1.15
```

Out[31]:
```
(417.095000000000, 564.305000000000)
```

Transformations:

The data is transformed in several ways:

- dens: contains the data as density everywhere but at the kink where it is a probability, used for least squares fitting, model with perfect bunching;
- densa: containes the data as proportion of the sample, used for maximum likelihood, model with perfect bunching;
- densb: contains the data as density everywhere, used for least squares fitting, model with imperfect bunching.

In [32]:
```
totnobs = mkd['Nobs']

KinkPos = int((len(mkd['df'])-1)/2)

print(sum( n[1] for n in mkd['dens']))

#best normal fit...
# measure mean and variance
print("level")
m  = sum(n[1]*n[0]   for n in mkd['dens'])
m2 = sum(n[1]*n[0]^2 for n in mkd['dens'])
v2 = m2-m^2
print(['mean',m],['std dev',sqrt(m2-m^2)])

print("logs")
lm = sum( n[1]*log(n[0]) for n in mkd['dens'])
lm2 = sum(n[1]*log(n[0])^2 for n in mkd['dens'])
lv2 = lm2-lm^2
print(['mean',lm],['std dev',sqrt(lm2-lm^2)])
# -1. to deal with the data to the left of the lower bound...
# to copy in the spyx file...until I find how to transfer data on the fly!

lnebu = ln(min(n[0] for n in mkd['dens'])-.1)-0.001
lnebo = ln(max(n[0] for n in mkd['dens']))+0.001



sum(mkd['dens'][n][1] for n in range(0,len(mkd['dens'])) if n!= KinkPos)+mkd['de
sum(mkd['dens'][n][1] for n in range(0,len(mkd['dens'])) )
```

```
1.0000000000000042
level
['mean', 480.4303364695853] ['std dev', 33.10937618086997]
logs
['mean', 6.172316342035386] ['std dev', 0.06874764297418277]
```

Out[32]: (1.0000000000000042, 1.0000000000000042)

In [33]:
```
lnebu,lnebo
```

Out[33]: (6.04447913541422, 6.3273283480326)

In [34]:
```
KinkPos,mkd['dens'][KinkPos]
```

Out[34]: (684, array([4.90700000e+02, 3.84094165e-02]))

In [35]:
```
# allows us to check the data!
#list_plot(dens,ymin=0)+ list_plot(densa,color='gold')
```

In [ ]:
```

```

In [36]:
```
Info_in_year
```

Out[36]: [2019, 0.658500000000000, 0.465850000000000, 490700]

In [37]:
```
#list(zip(dens,densa))
Info_in_year[1]
```

Out[37]:
```
0.658500000000000
```

Tax parameters:

In [38]:
```
pl1,pl2,ps,ps1,ps2,pr12,lw = var('pl1,pl2,ps,ps1,ps2,pr12,lw')

#/Kink

# this is complicated since their application aggregates many years...here I tak
lnt1c = ln(Info_in_year[1])
lnt2c = ln(Info_in_year[2])
lt1cplt2c = lnt1c^2 + lnt2c^2
lt1cmlt2c = lnt1c^2 - lnt2c^2
```

In [39]:
```
delta,delta/(lnt1c-lnt2c)
```

Out[39]:
```
(0.0999990000000000, 0.288930254180252)
```

In [40]:
```
delta,lnk,mkd['dens'][KinkPos][0],Kink
```

Out[40]:
```
(0.0999990000000000, 6.19583294309586, 490.7, 490.700000000000)
```

In [41]:
```
#len(df_yy_cnt),Kink
#ck['IG'][0:10],ck['FREQ'][0:10]
#show(list(zip(ck['IG'].list(),ck['FREQ'].list())))
```

Cython code...

In [42]:
```
#newd = "C:\\Users\\gala0013\\Dropbox\\sagemath\\alternativeETI"
#os.chdir(newd)
```

In [43]:
```
#compile some useful routines
#load("~/Users/gauthier/Dropbox/sagemath/alternativeETI/GLbunching.spyx")
#load('C:\Users\gala0013\Dropbox\sagemath\alternativeETI\GLbunching.spyx')
load("GLbunching2.spyx")
```

```
Compiling ./GLbunching2.spyx...
```

Least Squares and Likelihood Routines

The routines use various parametrisation. This matters when calculating the second derivatives and the ML precision.

In [44]:

```
%%cython

clib: m

cdef extern from "math.h":
    double sin(double x)
    double exp(double x)
    double log(double x)
    double sqrt(double x)
    double erf(double x)
    double tanh(double x)
    double pow(double x, double y)
    double fmax(double x, double y)
    double fmin(double x, double y)

def param1(pprm,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk):
    pps  = exp(fmax(fmin(pprm[2],5.),-5))
    ppl2 = abs(pprm[0])*lnt2c +  pprm[1]
    ppl2 = pps*ppl2
    ppl1 = abs(pprm[0])*lnt1c +  pprm[1]
    ppl1 = pps*ppl1

    return (ppl1,ppl2,pps)

def paramib(pprm,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk):
    #pprm[1] = fmax(fmin(pprm[1],5.),-5)
    #pl1 = abs(pprm[0])*lnt1c + lnebu + (lnebo-lnebu)/(1+exp(-pprm[1]))
    #pl2 = abs(pprm[0])*lnt2c + lnebu + (lnebo-lnebu)/(1+exp(-pprm[1]))
    pl1 = abs(pprm[0])*lnt1c + pprm[1]
    pl2 = abs(pprm[0])*lnt2c + pprm[1]

    if len(pprm)==4:
        #homogenous model with imperfect bunching
        ps1  = exp(-fmax(fmin(pprm[2],5.),-5))
        ps2 = ps1
        vs = exp(fmax(fmin(pprm[3],10.),-5))
        return [pl1/ps1,pl2/ps2,1/ps1,1/ps2,1.,vs]

    elif len(pprm)==5:
        # heterogenous model with imperfect bunching
        # independence between disutility and response remains equal to 0
        prr = 0.0
    #elif len(pprm)==5:
    #    prr = tanh(pprm[4])

    # variance of eti can't be too large under the normal model...
    sa = abs(pprm[0])/2.5/(1+exp(-fmax(fmin(pprm[3],5.),-5)))
    st = exp(-fmax(fmin(pprm[2],5.),-5))

    #ps1 = pprm[2]^2 + 2.*prr*abs(pprm[2]*pprm[3])*lnt1c         + (lnt1c*lnt1c)
    ps1 = st^2 + 2.*prr*abs(st*sa)*lnt1c         + (lnt1c*lnt1c)*(sa^2)
    ps1 = sqrt(ps1)
    ps2 = st^2 + 2.*prr*abs(st*sa)*lnt2c         + (lnt2c*lnt2c)*(sa^2)
    ps2 = sqrt(ps2)
    pc12= st^2 +    prr*abs(st*sa)*(lnt1c+lnt2c) + (lnt1c*lnt2c)*(sa^2)
    pc12= pc12/ps1/ps2

    vs = exp(fmax(fmin(pprm[4],10.),-5))
    #print(pc12)#
    pc12 = fmin(fmax(pc12,-1.+1e-5),1.-1e-5)
```

```
    #pc12 = pc12/ps1/ps2
    return [pl1/ps1,pl2/ps2,1/ps1,1/ps2,pc12,vs]
```

In [45]:

```
import math

def ff0n(w,pl1,pl2,ps,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk):
    ppprm = param1([pl1,pl2,ps])
    ppl1 = ppprm[0]
    ppl2 = ppprm[1]
    pps  = ppprm[2]
    cndI = fmax(cdfn(pps*lnebo-ppl2)-cdfn(pps*lnebu-ppl1),0.0000001)

    if math.log(w)<log(Kink-delta):
        return pdfn(pps*math.log(w)-ppl1)*pps/w/cndI

    if math.log(w)<=lnk and math.log(w)>=log(Kink-delta):
        return (cdfn(pps*lnk-ppl2)-cdfn(pps*log(Kink-delta)-ppl1))/cndI

    if math.log(w)>lnk:
        return pdfn(pps*math.log(w)-ppl2)*pps/w/cndI

def ff0na(w,pprm,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk):
    ppprm = param1(pprm)
    pl1 = ppprm[0]
    pl2 = ppprm[1]
    ps  = ppprm[2]

    lw = math.log(w)
    lKmd = math.log(Kink-delta)

    if lw<lKmd:
        return pdfn(ps*lw-pl1)*ps/w
    if lw>lnk:
        return pdfn(ps*lw-pl2)*ps/w
    if lw<=lnk and lw>=lKmd:
        return (cdfn(ps*lnk-pl2)-cdfn(ps*lKmd-pl1))

def prob_int(pprm,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk):
    # this parametrisation is better behaved...
    # this feels very ad hoc but it works (at least here!)
    ppprm = param1(pprm)
    ppl1 = ppprm[0]
    ppl2 = ppprm[1]
    pps  = ppprm[2]

    cndI  = (cdfn(pps*lnebo-ppl2)-cdfn(pps*lnebu-ppl1))
    return cndI


def h0iba1(ww,pl1,pl2,ps1,ps2,pr12,vs,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk):
    # density as in the paper...
    #import pdb; pdb.set_trace()
    lnw = math.log(ww)
    ffss1 = ps1
    ffss12=ffss1*ffss1
    ffmm1 = ffss1*lnk-pl1
    ffss2 = ps2
    ffss22=ffss2*ffss2
    ffmm2 = ffss2*lnk-pl2
    evs = vs
```

```
    evs2 = evs*evs
    return   (ffss1/sqrt(evs2+ffss12)*pdfn((evs*ffss1*lnw-(pl1*evs-1/2*(ffss1/ev
             cdfn((-evs2*lnw +(evs2+ffss12)*lnk-1/2-pl1*ffss1)/sqrt(evs2+ffss12)

def h0iba2(ww,pl1,pl2,ps1,ps2,pr12,vs,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk):
    # density as in the paper...
    #import pdb; pdb.set_trace()
    lnw = math.log(ww)
    ffss1 = ps1
    ffss12=ffss1*ffss1
    ffmm1 = ffss1*lnk-pl1
    ffss2 = ps2
    ffss22=ffss2*ffss2
    ffmm2 = ffss2*lnk-pl2
    evs = vs
    evs2 = evs*evs
    return   (ffss2/sqrt(evs2+ffss22)*pdfn((evs*ffss2*lnw-(pl2*evs-1/2*(ffss2/ev
             cdfn((evs2*lnw-(evs2+ffss22)*lnk+1/2+pl2*ffss2)/sqrt(evs2+ffss22))/

def h0ibak(ww,pl1,pl2,ps1,ps2,pr12,vs,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk):
    # density as in the paper...
    #import pdb; pdb.set_trace()
    lnw = math.log(ww)
    ffss1 = ps1
    ffss12=ffss1*ffss1
    ffmm1 = ffss1*lnk-pl1
    ffss2 = ps2
    ffss22=ffss2*ffss2
    ffmm2 = ffss2*lnk-pl2
    evs = vs
    evs2 = evs*evs
    if (pr12>=1):
        FFk= cdfn(ffmm2)-cdfn(ffmm1)
    else:
        FFk= cdfn(ffmm2)-cdfbvn(ffmm1,ffmm2,pr12)
    return   (pdfn(evs*(lnw-lnk)+1/2/evs)/ww*FFk)
    #return FFk/evs

def h0iba(ww,pl1,pl2,ps1,ps2,pr12,vs,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk):
    # density as in the paper...
    #import pdb; pdb.set_trace()
    lnw = math.log(ww)
    ffss1 = ps1
    ffss12=ffss1*ffss1
    ffmm1 = ffss1*lnk-pl1
    ffss2 = ps2
    ffss22=ffss2*ffss2
    ffmm2 = ffss2*lnk-pl2
    evs = vs
    evs2 = evs*evs
    if (pr12>=1):
        FFk= cdfn(ffmm2)-cdfn(ffmm1)
    else:
        FFk= cdfn(ffmm2)-cdfbvn(ffmm1,ffmm2,pr12)
    #if tanh(pr12)<0.9999:
    #    FFk= cdfN(ffmm2)-cdfbvn(ffmm1,ffmm2,tanh(pr12))
    #else:
    #    FFk= cdfN(ffmm2)-cdfN(ffmm1)
    #print(FFk.n())
    # removed *evs
```

```
    return   (pdfn(evs*(lnw-lnk)+1/2/evs)/ww*FFk+\
       ffss2/sqrt(evs2+ffss22)*pdfn((evs*ffss2*lnw-(pl2*evs-1/2*(ffss2/evs)))/sq
       cdfn((evs2*lnw-(evs2+ffss22)*lnk+1/2+pl2*ffss2)/sqrt(evs2+ffss22))/ww+ \
       ffss1/sqrt(evs2+ffss12)*pdfn((evs*ffss1*lnw-(pl1*evs-1/2*(ffss1/evs)))/sq
       cdfn((-evs2*lnw +(evs2+ffss12)*lnk-1/2-pl1*ffss1)/sqrt(evs2+ffss12))/ww)*


def cndI0ib(prm,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk):
    pprm = paramib(prm,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk)

    cndI = prob_alt(math.exp(lnebo),*pprm,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk

    return cndI


def prob_alt(ww,pl1,pl2,ps1,ps2,pr12,vs,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk):
    #a formal expression...
    lnw = math.log(ww)
    ffss1 = ps1
    ffss12=ffss1*ffss1
    ffmm1 = ffss1*lnk-pl1
    ffss2 = ps2
    ffss22=ffss2*ffss2
    ffmm2 = ffss2*lnk-pl2
    evs = vs
    evs2 = evs*evs
    #import pdb; pdb.set_trace()
    FFk= cdfn(ffmm2)-cdfn(ffmm1)
    #show(FFk)
    P0 = FFk*cdfn(evs*(lnw-lnk)+1/2/evs)

    #show(ffss1/sqrt(evs2+ffss12))
    y2 = (evs*ffss1*lnw-(pl1*evs-1/2*(ffss1/evs)))/sqrt(evs2+ffss12)
    P1 = cdfbvn(ffmm1,y2,evs/sqrt(evs2+ffss12))

    y2 = (evs*ffss2*lnw-(pl2*evs-1/2*(ffss2/evs)))/sqrt(evs2+ffss22)
    #y1 = (evs*ffss2*lnk-(pl2*evs-1/2*(ffss2/evs)))/sqrt(evs2+ffss22)
    P2 = cdfbvn(-ffmm2,y2,-evs/sqrt(evs2+ffss22))
    #-cdfbvn(-ffmm2,y1,-evs/sqrt(evs2+ffss22))
    #show((P0.n(),P1.n(),P2.n()))
```

In [46]:
```
import math

def lik0(pprm,densi,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk):
    #ppprm = param0(pprm)
    ppprm = param1(pprm,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk)
    ppl1 = ppprm[0]
    ppl2 = ppprm[1]
    pps  = ppprm[2]

#    print(pl1,pl2)
    #cndI  = (cdfn(pps*lnebo-ppl2)-cdfn(pps*lnebu-ppl1))
    cndI = max(cdfn(pps*lnebo-ppl2)-cdfn(pps*lnebu-ppl1),0.000000001)
    if cndI<1.e-9:
        cndI = 1.e-9
        print(pprm,cndI)

    ores = [ n[1]*math.log(ff0na(n[0],pprm,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,ln
    #print(res)
    #res = sum(res).n()-ln(cndI).n()
    return -sum(ores)
```

```
# least square with minimize; rough and ready!
def lsq0(pprm,densi,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk):
    #ppprm = param0(pprm)
    ppprm = param1(pprm,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk)
    ppl1 = ppprm[0]
    ppl2 = ppprm[1]
    pps  = ppprm[2]

    #cndI  =    cdfn(pps*lnebo-ppl2)-cdfn(pps*lnebu-ppl1)
    cndI = max(cdfn(pps*lnebo-ppl2)-cdfn(pps*lnebu-ppl1),0.000000001)
    if cndI<1.e-9:
        cndI = 1.e-9
        #print(pprm,cndI)

    ores = [ (n[1]-(ff0na(n[0],pprm,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk)/cndI
    ores = sum(ores)
    #print(pprm,res)
    return ores
def lik0ib(prm,densi,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk):
    pprm = paramib(prm,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk)
    cndI = prob_alt(math.exp(lnebo),*pprm,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk
    #if cndI<=0.00001:
    #        return len(densi)*50
    #show(cndI)
    #import pdb; pdb.set_trace()
    ores = 0
    p_bef = prob_alt(math.exp(lnebu),*pprm,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,ln
    longueur = len(densi)

    for ii in range(longueur):
        p_at = prob_alt(densi[ii][0],*pprm,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,ln
        if ((p_at-p_bef+1e-50)/cndI)<1e-10:
            #show([ii,(p_at-p_bef),cndI,densi[ii]],prm,pprm)
            ores -= densi[ii][1]*(-75)
        else:
            ores -= densi[ii][1]*(math.log((p_at-p_bef+1e-50))- math.log(cndI) )
        p_bef = p_at
    #show([*pprm,cndI,ores])
```

In [47]:

```
def mak_vc(loglik,loglik_long,x,lendti,N,stp_bn,*args):
    #construct the variance covariance of the ML estimator...
    #Gradients, Hessian, Information matrices
    #adjusting for density etc...
    G_all = stp_bn*matrix(RDF,grdcd(loglik,x,*args))
    hh0ib = stp_bn*matrix(RDF,hescd(loglik,x,*args))
    hh0ibi = hh0ib.inverse()
    G00 = grdcd_long(loglik_long,x,lendti,*args)
    G0ib  = stp_bn*matrix(RDF,G00)
    jj0ib = N*G0ib*G0ib.transpose()
    return {
            'G_all':G_all,
            'hh0ib':hh0ib,
            'hh0ibi':hh0ibi,
            'jj0ib':jj0ib
    }

def lsq0ib(prm,densi,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk):
    pprm = paramib(prm,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk)

    cndI = (prob_alt(math.exp(lnebo),*pprm,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,ln
```

In [48]:
```
def mak_chi2(datai,N,x,*par):
    #make a chi squared stat...
    OO2 = [ [nn[0],nn[1]*totnobs] for nn in datai]
    TT2 = [ [n[0],float(h0iba(n[0],*paramib(x,*par),*par))/cndI0ib(x,*par)*N] fo
    csq = [ float((OO2[ii][1]-TT2[ii][1])^2/TT2[ii][1]) for ii in range(len(data
    csq2 = [ [OO2[ii][0],float((OO2[ii][1]-TT2[ii][1])^2/TT2[ii][1])] for ii in
    return {
        'Observed #':OO2,
        'Theoretical #':TT2,
        'Chi_squared stat': sum(csq),
        'obs vs ind chi2 stat': csq2
    }
```

In [49]:
```
def mak_OLS_R2(O2,T2):
    data = [ (T2[ii][1],O2[ii][1]) for ii in range(len(O2))]
    N = sum([n[1] for n in O2])
    # design a model with adjustable parameters a,b,c that describes the data
    a,b,x = var('a, b, x')
    model(x) = a * x + b

    # calculate the values of the parameters that best fit the model to the data
    sol = find_fit(data,model)
    # define f(x), the model with the parameters set to the fitted values
    f(x) = model(a=sol[0].rhs(),b=sol[1].rhs())

    # create an empty plot object
    a = plot([])
    # add a plot of the model, with respect to x from -0.5 to 12.5
    a += plot(f(x),x,[min(nn[1] for nn in T2),max(nn[1] for nn in T2)])

    # add a plot of the data, in red
    a += list_plot(data,color='red')

    V_obs = variance([nn[1] for nn in O2])
    V_the = variance([f(nn[1]) for nn in T2])
    R2sq = V_the/V_obs

    return {
        'a':sol[0].rhs(),
        'b':sol[1].rhs(),
        'Nobs':N,
        'graph':a,
        'Var obs':V_obs,
        'Var theoretical':V_the,
        'R2':R2sq
    }
```

In [50]:
```
%load_ext cython
```

In [51]:
```
#%%cython
#clib: m
#cdef extern from "math.h":
#    double sin(double)
#    double exp(double)
#    double log(double)
#    double sqrt(double)
#    double erf(double)
```

In [52]:
```
#[ n for n in dens if (((ln(n[0])+2*delta)>=lnk) and (ln(n[0])<=lnk)) ],
Kink,exp(lnk)
```

Out[52]:
```
(490.700000000000, 490.700000000000)
```

Model with Imperfect Bunching.

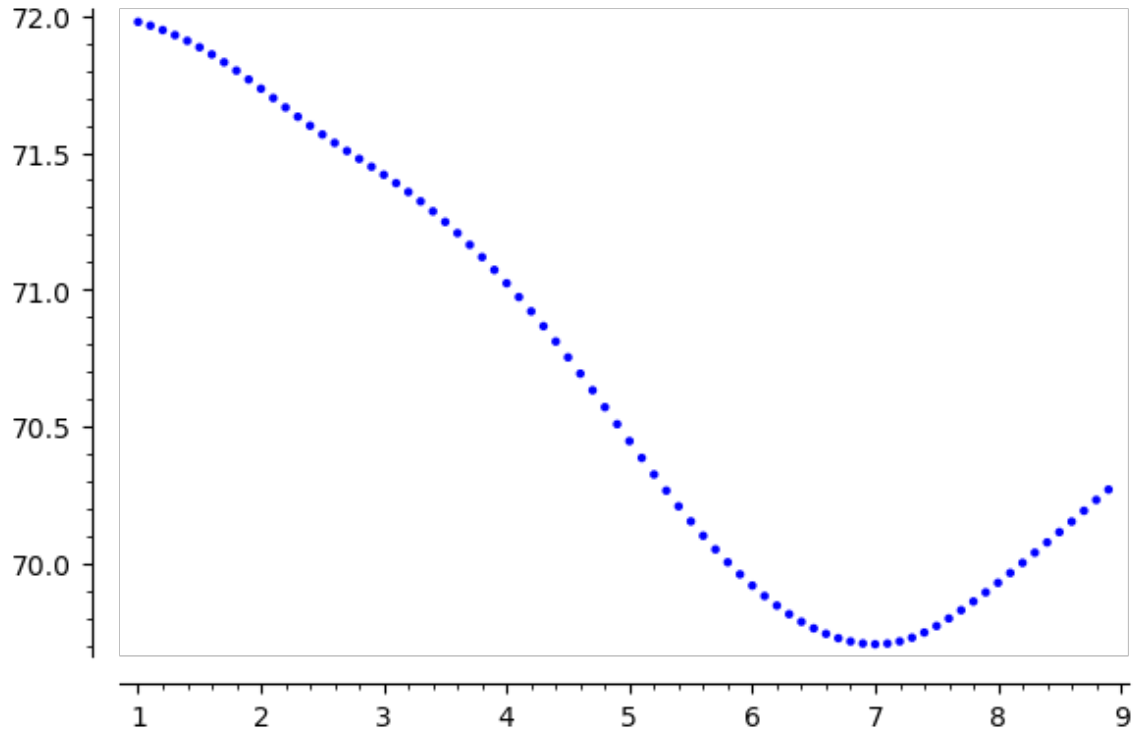
model of density throughout. Log Normal-Log Normal model...

All Observations.

We start with some more routines...

In [59]:
```
densb = mkd['densb']
sqrtN =sqrt(mkd['Nobs'])
tax_par = (densb,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk)
# starting from almost correct solutions! searching for best standard error of f
#list_plot([ [xx,lik0ib([0.06, 6.15, 2.,xx],*tax_par)] for xx in srange(1.0,9.,0
#list_plot([ [xx,lik0ib([xx, 6.15, 2., 7.0],*tax_par)] for xx in srange(0.02,0.2
#list_plot([ [xx,lik0ib([0.07, xx, 2., 7.0],*tax_par)] for xx in srange(5.5,6.5,
#list_plot([ [xx,lik0ib([0.07, 6.2, xx, 7.0],*tax_par)] for xx in srange(1.75,2.
list_plot([ [xx,lik0ib([0.07, 6.2, 2.2,xx],*tax_par)] for xx in srange(1.0,9.,0.
```

Out[59]:



In [60]:
```
#maximum likelihood estimator.
#tax_par = (('delta', delta),('lnt1c',lnt1c),('lnt2c',lnt2c),('lnebu',lnebu),('l
tax_par = (densb,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk)
solib_Lll_all = minimize(lik0ib,[0.07, 6.2, 2.2, 6.9],args=tax_par, verbose=True
print(solib_Lll_all)
```

```
Optimization terminated successfully.
         Current function value: 69.650189
         Iterations: 96
         Function evaluations: 171
(0.10206311716886357, 6.199949427585897, 2.0260598905446336, 6.776668576169403)
```

In [ ]:
```
lnebu,lnebo
```

In [61]:
```
mkd['Nobs'],STEP_BIN
```

Out[61]:
```
(25827, 0.100000000000000)
```

In [62]:
```
#calculation of var cov of parameters
densb = mkd['densb']
#(loglik,loglik_long,x,lendti,N,stp_bn,*args)
VCM= mak_vc(lik0ib,lik0ib_long,solib_Lll_all,len(densb),mkd['Nobs'],STEP_BIN,*ta
```

In [63]:
```
show(VCM['hh0ib'].eigenvalues(algorithm='symmetric'))
show(VCM['jj0ib'].eigenvalues(algorithm='symmetric'))
```

In [64]:
```
VCM['G_all']
```

Out[64]:
```
[-5.926834844953962e-05  6.653922667567939e-05  -7.89604745242042e-06 1.55727108
02076364e-06]
```

In [65]:
```
# printing results... Hessian+ Sandwich...
```

In [66]:
```
res_lik_print0(lik0ib,solib_Lll_all,VCM['hh0ibi'],sqrtN,int(mkd['Nobs']),"Likeli
```

```
Likelihood Estimates, Homogenous Model, Imperfect Bunching, Hessian
Parameter Estimate Std.Err.
  eti      0.1021   0.0025368
  mu       6.200    0.0018761
  sigma1   2.026    0.018897
  v        6.777    0.033948
  number of observations |               25827
          log_likelihood | -1798855.4288572774
```

Out[66]:
```
((0.10206311716886357, 6.199949427585897, 2.0260598905446336, 6.77666857616940
3),
 [0.0025368, 0.0018761, 0.018897, 0.033948],
 -1798855.4288572774)
```

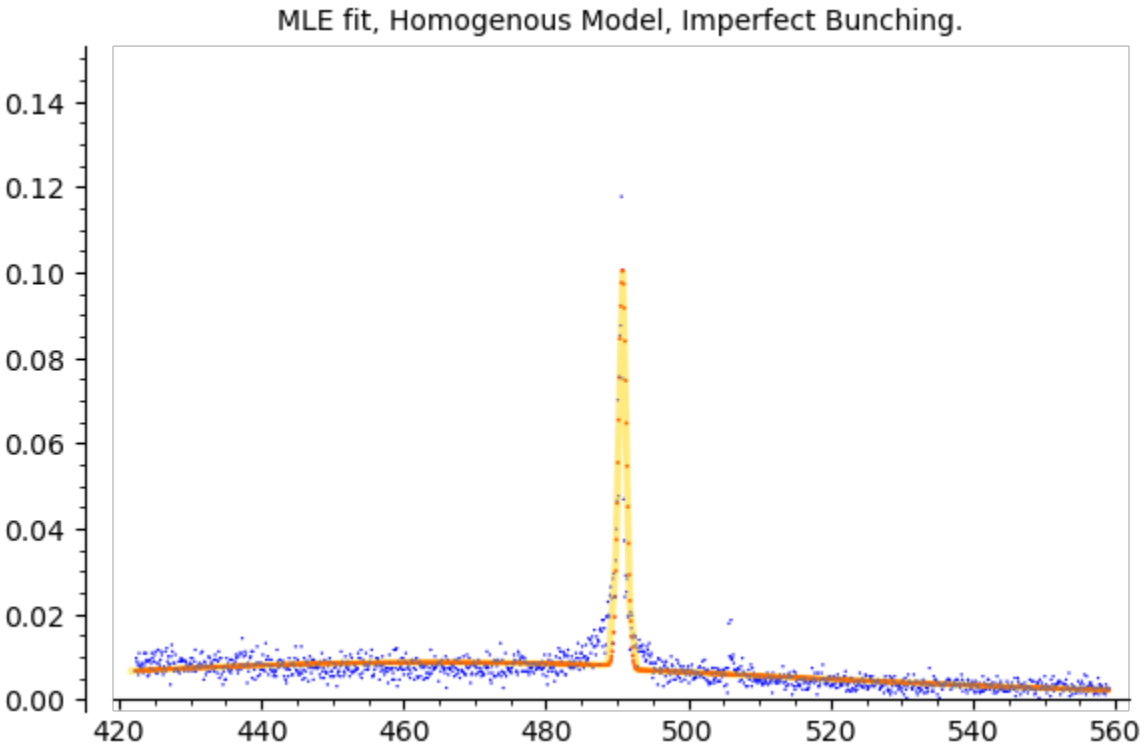
In [67]:
```
res_lik_print0(lik0ib,solib_Lll_all,VCM['hh0ibi']*VCM['jj0ib']*VCM['hh0ibi'],sqr
```

```
Likelihood Estimates, Homogenous Model, Imperfect Bunching, Sandwich
Parameter Estimate Std.Err.
  eti      0.1021   0.028797
  mu       6.200    0.017409
  sigma1   2.026    0.14023
  v        6.777    0.96796
  number of observations |               25827
          log_likelihood | -1798855.4288572774
```

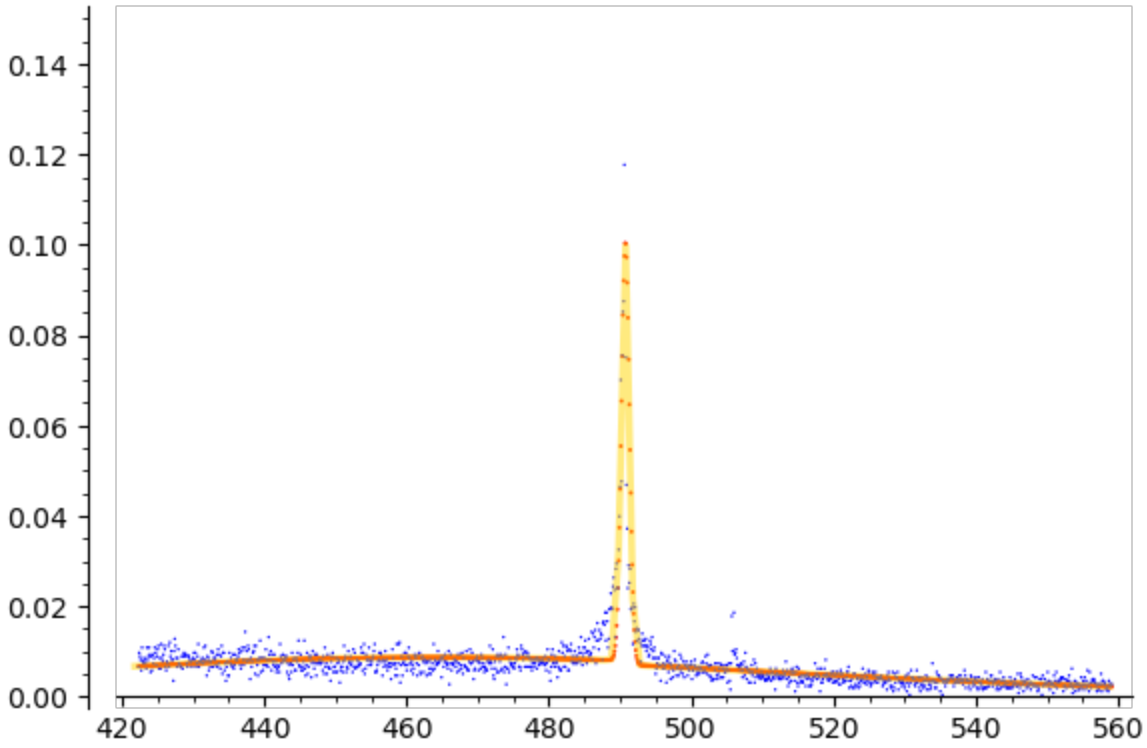
Out[67]:
```
((0.10206311716886357, 6.199949427585897, 2.0260598905446336, 6.77666857616940
3),
 [0.028797, 0.017409, 0.14023, 0.96796],
 -1798855.4288572774)
```

In [68]:
```
tri = solib_Lll_all
tax_par2 = (delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk)
#plot( h0iba(x,*paramib(tri))/cndI0ib(tri),(x,exp(lnebu),exp(lnebo)),color='red'
resll0 = [ [n[0],\
    ((prob_alt(n[0],*paramib(tri,*tax_par2),*tax_par2)-prob_alt(n[0]-.1,*paramib
    ] for n in mkd['densa'] ]
resll1 = [ [n,\
    (h0iba(n,*paramib(tri,*tax_par2),*tax_par2))/cndI0ib(tri,*tax_par2)\
    ] for n in srange(exp(lnebu),exp(lnebo),1) ]
```

In [69]:
```
grfitib = list_plot(resll1,color='gold',plotjoined=True,thickness=2.5,alpha=0.5)
          list_plot(resll0,color='red',size=2)+\
          list_plot(mkd['densb'],size=1)
grfitib.show(title='Maximum Likelihood fit, Imperfect Bunching.',ymax=0.15)
#          list_plot(resls1,color='black',plotjoined=True,linestyle=':')+\
```



In [72]:
```
grfitib.show(ymax=0.15)
```



In [70]:
```
cndI0ib(solib_Lll_all,*tax_par2).n()*mkd['Nobs'],float(h0iba(490.,*paramib(solib
```

Out[70]: (19221.0403988721, 0.03763736978359802)

Chi Squared testing

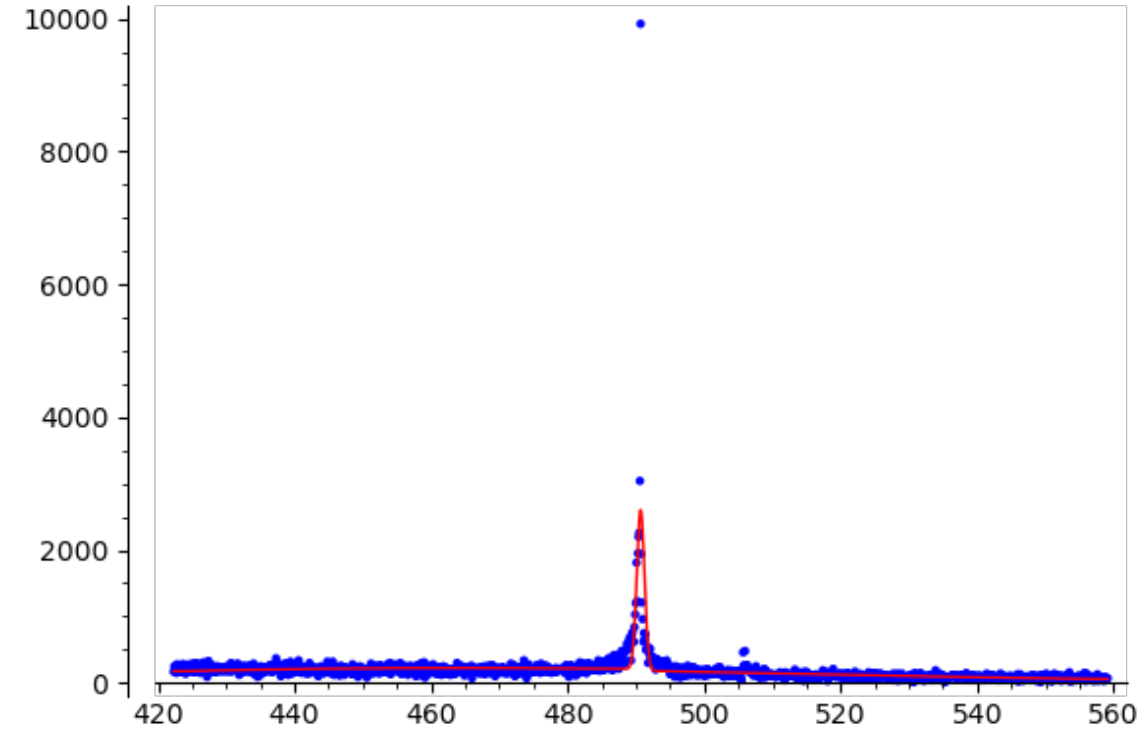
In [ ]:
```
```

In [71]:
```
MKC2=mak_chi2(densb,totnobs,solib_Lll_all,*tax_par2)
```

In [73]:
```
show(n(MKC2['Chi_squared stat'],digits=8))
```

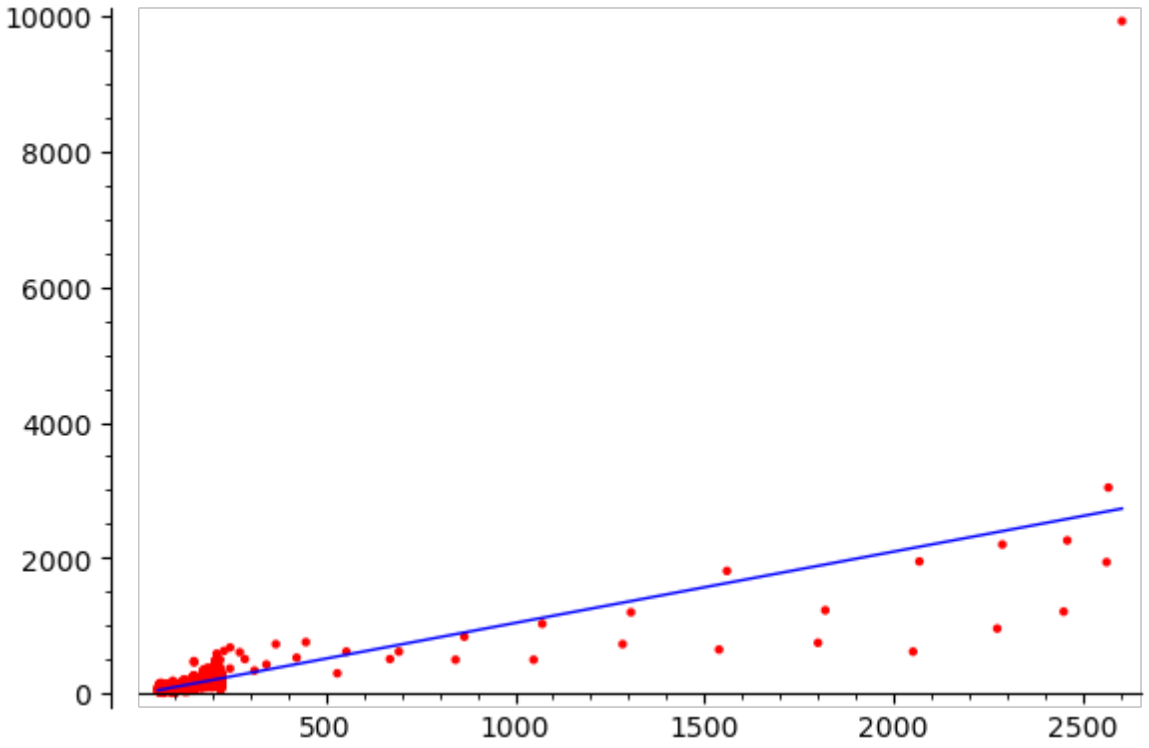
In [74]:
```
list_plot(MKC2['Observed #'],color='blue')+list_plot(MKC2['Theoretical #'],color
```

Out[74]:



In [75]:
```
MKOR = mak_OLS_R2(MKC2['Observed #'],MKC2['Theoretical #'])
```

In [76]:
```
MKOR['graph'].show()
```



In [77]:
```
MKOR['R2']
```

Out[77]:
```
0.5078799073387024
```

Selected Observations, +/- 40 around kink

In [78]:
```
#+/- 40 around kink
mkd2=mak_density(dd,Kink,40+0.001,STEP_BIN,delta)
totnobs = mkd2['Nobs']

KinkPos = int((len(mkd2['df'])-1)/2)

print(sum( n[1] for n in mkd2['dens']))

lnebu = ln(min(n[0] for n in mkd2['dens'])-STEP_BIN)-0.001
lnebo = ln(max(n[0] for n in mkd2['dens']))+0.001
```

```
1.0000000000000067
```

In [79]:
```
len(densb), mkd2['df'][KinkPos], mkd2['densb'][KinkPos]
```

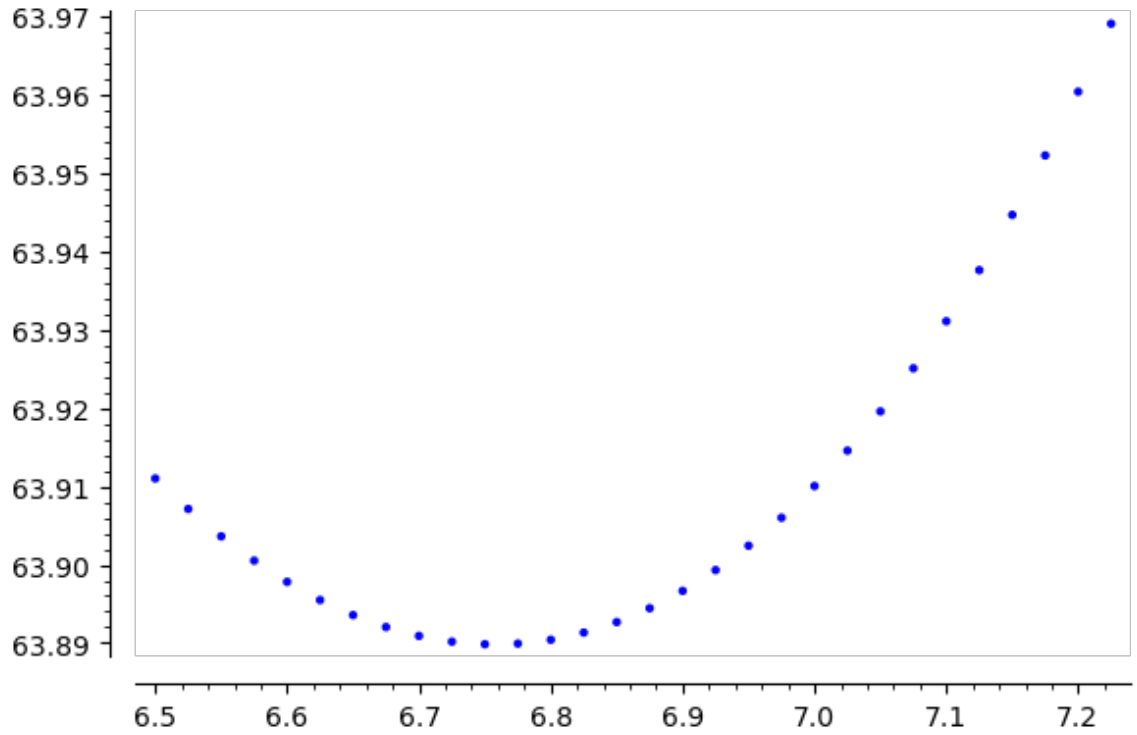
Out[79]:
```
(1369,
 (685.0, 992.0, 2019.0, 490.7, 490700.0, 1.0, 490.7),
 (490.7, 0.5602937023439706))
```

In [80]:
```
min(n[0] for n in mkd2['dens']),max(n[0] for n in mkd2['dens'])
```

Out[80]:
```
(450.7, 530.7)
```

In [87]:
```
densb = mkd2['densb']
sqrtN =sqrt(mkd2['Nobs'])
tax_par = (densb,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk)
#list_plot([ [xx,lik0ib([0.1021, 6.2, 2.02,xx],*tax_par)] for xx in srange(1.0,9
#list_plot([ [xx,lik0ib([xx, 6.2, 2.02, 6.8],*tax_par)] for xx in srange(0.02,0.
#list_plot([ [xx,lik0ib([0.105, xx, 2.02, 6.8],*tax_par)] for xx in srange(5.5,6
#list_plot([ [xx,lik0ib([0.105, 6.2, xx, 6.8],*tax_par)] for xx in srange(1.75,2
list_plot([ [xx,lik0ib([0.105, 6.2, 2.1,xx],*tax_par)] for xx in srange(6.5,7.25
```

Out[87]:



In [ ]:
```
#[ [xx,lik0ib([0.075, 6.2, 2.4,xx],*tax_par)] for xx in srange(5.0,8.0,0.1)]
```

In [ ]:
```
(delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk)
```

In [ ]:
```
list_plot(densb)
```

In [88]:
```
lik0ib([0.105, 6.2, 2.1,6.76],*tax_par), lik0ib([0.105, 6.2, 2.1,6.76],*tax_par)
```

Out[88]:
```
(63.88983850077644, 63.88983850077644, 801, (490.7, 0.5602937023439706))
```

In [89]:
```
lik0ib([0.105, 6.2, 2.1,6.76],*tax_par)
```

Out[89]:
```
63.88983850077644
```

In [90]:
```
exp(lnebo),exp(lnebu),lnk,exp(6.9)
```

Out[90]:
```
(531.2309654384723, 450.14962522491857, 6.19583294309586, 992.274715605026)
```

In [91]:
```
#maximum likelihood estimator.
#tax_par = (('delta', delta),('lnt1c',lnt1c),('lnt2c',lnt2c),('lnebu',lnebu),('l
solib_Lll_2 = minimize(lik0ib,[0.105, 6.2, 2.1,6.76],args=tax_par, verbose=True)
show(solib_Lll_2)
```

```
Optimization terminated successfully.
         Current function value: 63.786805
         Iterations: 114
         Function evaluations: 197
```

In [92]:
```
#calculation of var cov of parameters
VCM2= mak_vc(lik0ib,lik0ib_long,solib_Lll_2,len(densb),mkd2['Nobs'],STEP_BIN,*ta
```

In [93]:
```
res_lik_print0(lik0ib,solib_Lll_2,VCM2['hh0ibi'],sqrtN,int(mkd2['Nobs']),"Likeli
```

```
Likelihood Estimates, Homogenous Model, +/-40 around Kink, Imperfect Bunching, He
ssian
Parameter Estimate Std.Err.
  eti      0.07697   0.0020665
  mu       6.222     0.0012378
  sigma1   2.642     0.019910
  v        6.947     0.031959
  number of observations |               17705
          log_likelihood | -1129345.3869945295
```

Out[93]:
```
((0.07697433607912332, 6.2220818539092955, 2.6421099399934747, 6.94674566554306
7),
 [0.0020665, 0.0012378, 0.019910, 0.031959],
 -1129345.3869945295)
```

In [94]:
```
res_lik_print0(lik0ib,solib_Lll_2,VCM2['hh0ibi']*VCM2['jj0ib']*VCM2['hh0ibi'],sq
```

```
Likelihood Estimates, Homogenous Model, +/-40 around Kink, Imperfect Bunching, Sa
ndwich
Parameter Estimate Std.Err.
  eti      0.07697   0.023121
  mu       6.222     0.013156
  sigma1   2.642     0.14559
  v        6.947     0.84801
  number of observations |               17705
          log_likelihood | -1129345.3869945295
```

Out[94]:
```
((0.07697433607912332, 6.2220818539092955, 2.6421099399934747, 6.94674566554306
7),
 [0.023121, 0.013156, 0.14559, 0.84801],
 -1129345.3869945295)
```

In [95]:
```
tax_par2 = (delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk)
MKC2b=mak_chi2(densb,totnobs,solib_Lll_2,*tax_par2)
```

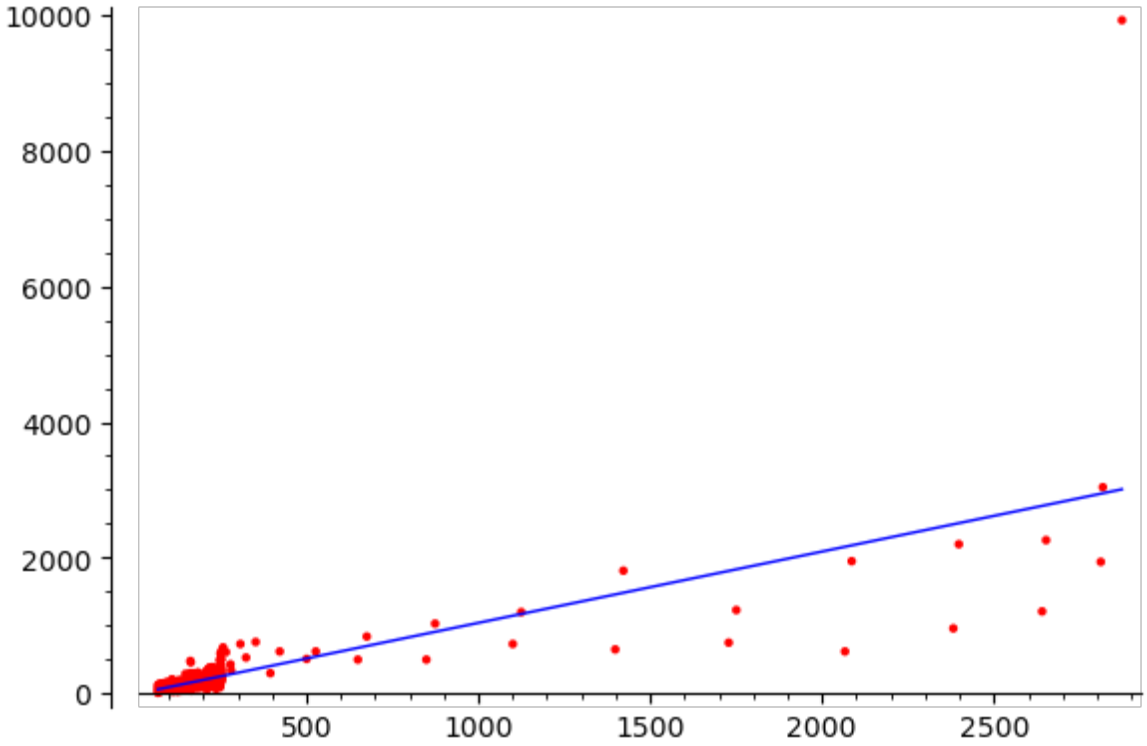
In [96]:
```
show(n(MKC2b['Chi_squared stat'],digits=8))
```

In [97]:
```
MKOR = mak_OLS_R2(MKC2b['Observed #'],MKC2b['Theoretical #'])
```

In [98]:
```
MKOR
```

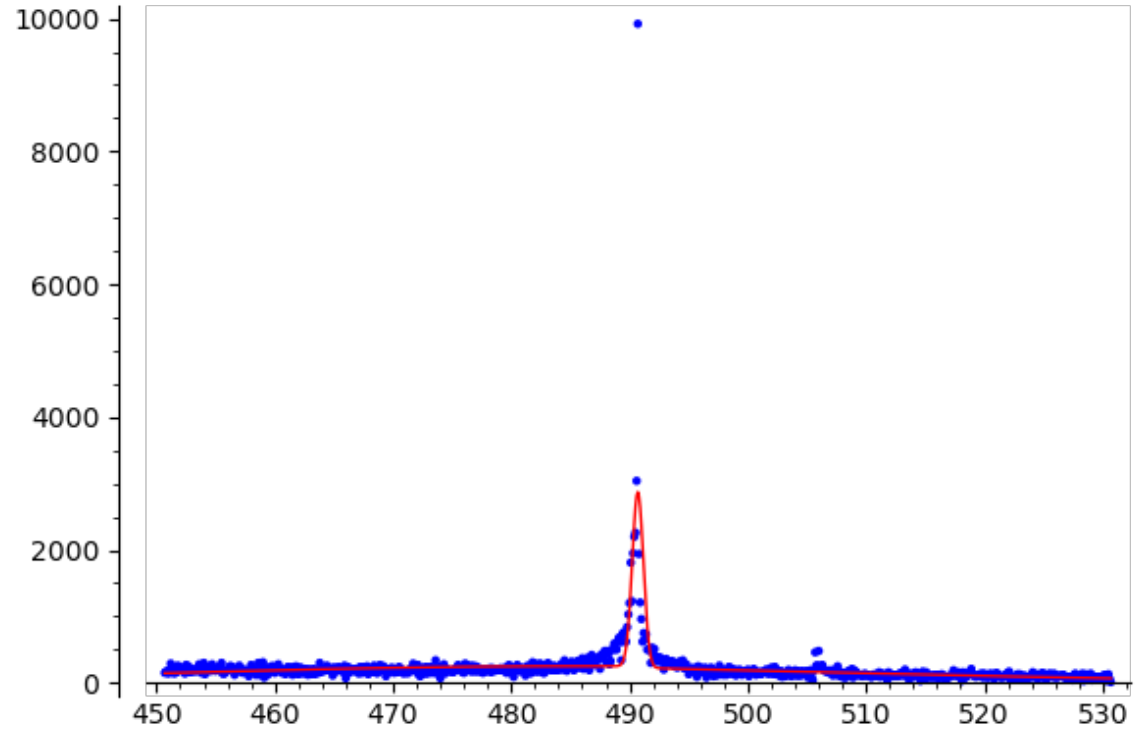
Out[98]:
```
{'a': 1.0508611262986605,
 'b': -9.862987837244344,
 'Nobs': 177050.0,
 'graph': Graphics object consisting of 2 graphics primitives,
 'Var obs': 165178.04993757835,
 'Var theoretical': 86629.24626999044,
 'R2': 0.5244597953706809}
```

In [99]:
```
MKOR['graph'].show()
```



In [100…
```
list_plot(MKC2b['Observed #'],color='blue')+list_plot(MKC2b['Theoretical #'],col
```

Out[100]:



In [102…
```
MKOR['R2']
```

Out[102]:
```
0.5244597953706809
```

Selected Observations, +/- 15 around kink

In [103…
```
#+/- 40 around kink
mkd3=mak_density(dd,Kink,15+0.001,STEP_BIN,delta)
totnobs = mkd3['Nobs']

KinkPos = int((len(mkd3['df'])-1)/2)

print(sum( n[1] for n in mkd3['dens']))

lnebu = ln(min(n[0] for n in mkd3['dens'])-STEP_BIN)
lnebo = ln(max(n[0] for n in mkd3['dens']))
```

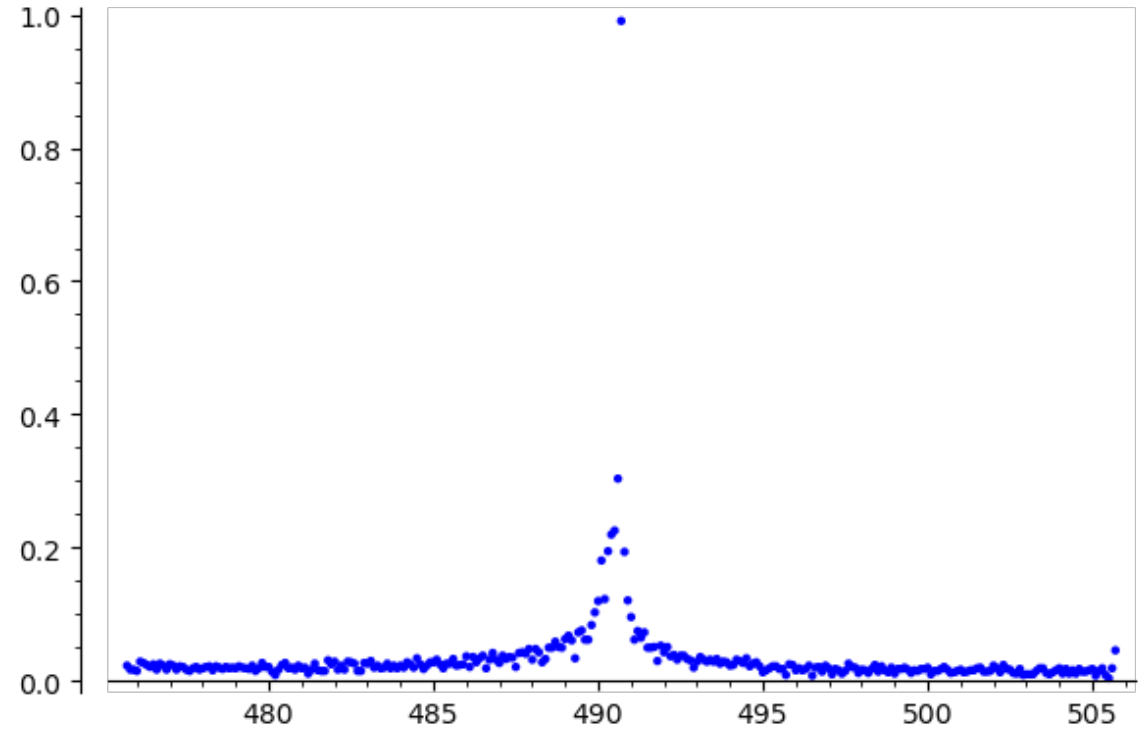
```
1.0000000000000004
```

In [104…
```
len(mkd3['densb']), mkd3['df'][KinkPos], mkd3['densb'][KinkPos]
```

Out[104]:
```
(301,
 (685.0, 992.0, 2019.0, 490.7, 490700.0, 1.0, 490.7),
 (490.7, 0.9907120743034056))
```

In [105…
```
list_plot(mkd3['densb'])
```

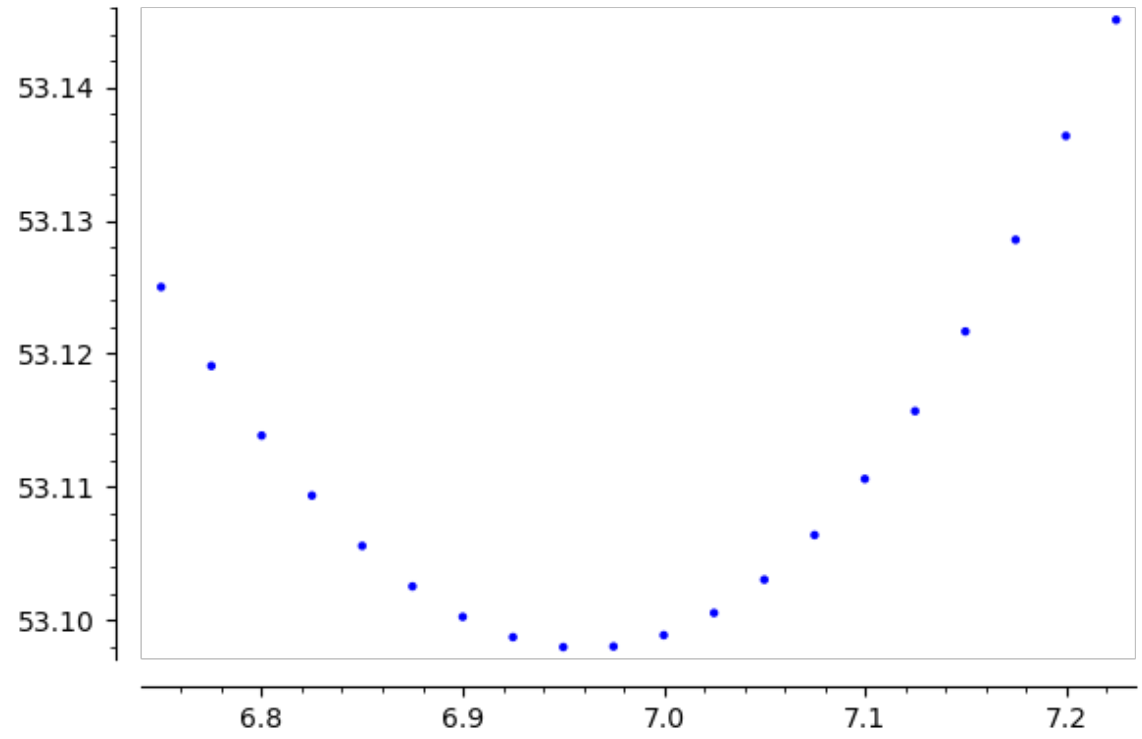
Out[105]:



In [112…

```
densb = mkd3['densb']
sqrtN =sqrt(mkd3['Nobs'])
tax_par = (densb,delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk)
#0.07697433607912332,6.2220818539092955,2.6421099399934747,6.946745665543067
#list_plot([ [xx,lik0ib([0.077, 6.22, 2.64,xx],*tax_par)] for xx in srange(1.0,9
#list_plot([ [xx,lik0ib([xx, 6.22, 2.64, 7.],*tax_par)] for xx in srange(0.02,0.
#list_plot([ [xx,lik0ib([0.075, xx, 2.64, 7.],*tax_par)] for xx in srange(5.95,6
#list_plot([ [xx,lik0ib([0.075, 6.22, xx, 7.],*tax_par)] for xx in srange(1.95,3
list_plot([ [xx,lik0ib([0.075, 6.22, 2.9,xx],*tax_par)] for xx in srange(6.75,7.
```

Out[112]:



In [ ]:

```
(delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk)
```

In [113…

```
lik0ib([0.075, 6.22, 2.9,6.95],*tax_par)
```

Out[113]: 53.09796074054919

In [114…
```
#maximum likelihood estimator.
#tax_par = (('delta', delta),('lnt1c',lnt1c),('lnt2c',lnt2c),('lnebu',lnebu),('l
tax_par2 = (delta,lnt1c,lnt2c,lnebu,lnebo,Kink,lnk)
solib_Lll_3 = minimize(lik0ib,[0.075, 6.22, 2.9,6.95],args=tax_par,verbose=True)
show(solib_Lll_3)
```

```
Optimization terminated successfully.
         Current function value: 52.949239
         Iterations: 102
         Function evaluations: 179
```

In [ ]:

In [115…
```
#calculation of var cov of parameters
VCM3= mak_vc(lik0ib,lik0ib_long,solib_Lll_3,len(densb),mkd3['Nobs'],STEP_BIN,*ta
```

In [116…
```
res_lik_print0(lik0ib,solib_Lll_3,VCM3['hh0ibi'],sqrtN,int(mkd3['Nobs']),"Likeli
```

```
Likelihood Estimates, Homogenous Model, +/-15 around Kink, Imperfect Bunching, He
ssian
Parameter Estimate Std.Err.
  eti      0.04463   0.0015396
  mu       6.218     0.00087062
  sigma1   3.651     0.027079
  v        7.204     0.032682
  number of observations |              10013
          log_likelihood | -530180.7251128439
```

Out[116]:
```
((0.04463087870540995, 6.218027418792525, 3.651499987197437, 7.20440347634293
8),
 [0.0015396, 0.00087062, 0.027079, 0.032682],
 -530180.7251128439)
```

In [117…
```
res_lik_print0(lik0ib,solib_Lll_3,VCM3['hh0ibi']*VCM3['jj0ib']*VCM3['hh0ibi'],sq
```

```
Likelihood Estimates, Homogenous Model, +/-15 around Kink, Imperfect Bunching, Sa
ndwich
Parameter Estimate Std.Err.
  eti      0.04463   0.017466
  mu       6.218     0.0096911
  sigma1   3.651     0.24322
  v        7.204     0.84649
  number of observations |              10013
          log_likelihood | -530180.7251128439
```

Out[117]:
```
((0.04463087870540995, 6.218027418792525, 3.651499987197437, 7.20440347634293
8),
 [0.017466, 0.0096911, 0.24322, 0.84649],
 -530180.7251128439)
```

In [118…
```
MKC2c=mak_chi2(densb,totnobs,solib_Lll_3,*tax_par2)
```

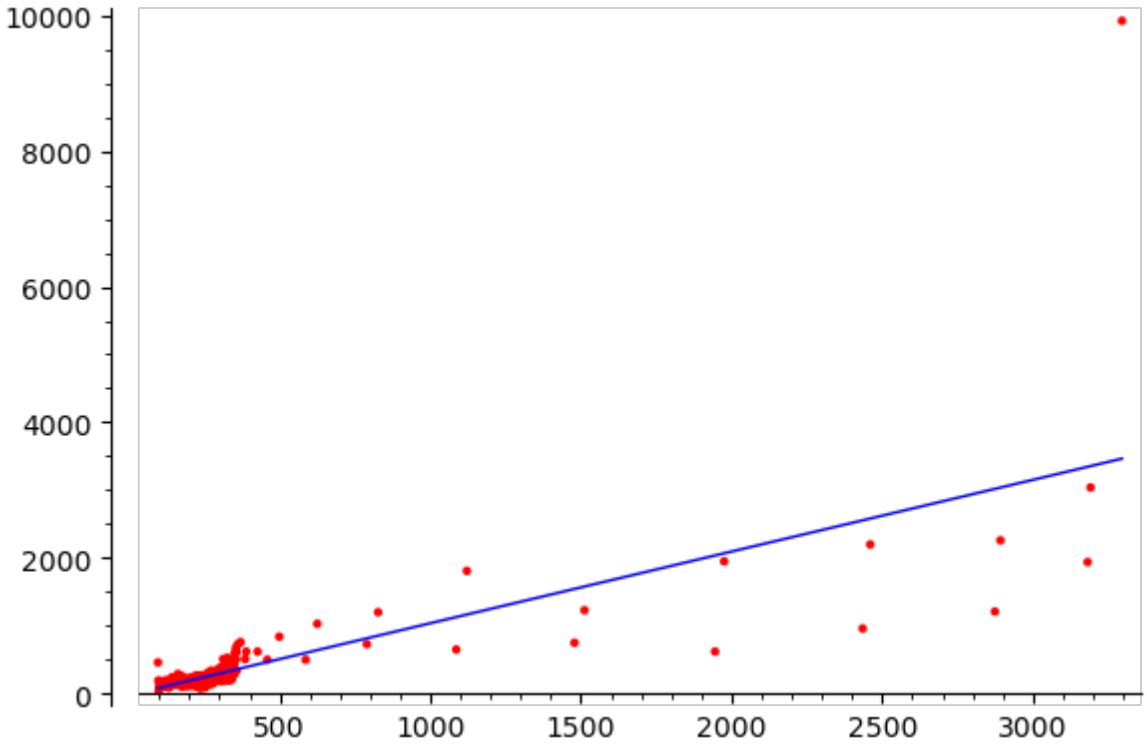
In [119…
```
show(n(MKC2c['Chi_squared stat'],digits=8))
```

In [120…
```
MKOR = mak_OLS_R2(MKC2c['Observed #'],MKC2c['Theoretical #'])
```

In [121…
```
MKOR
```

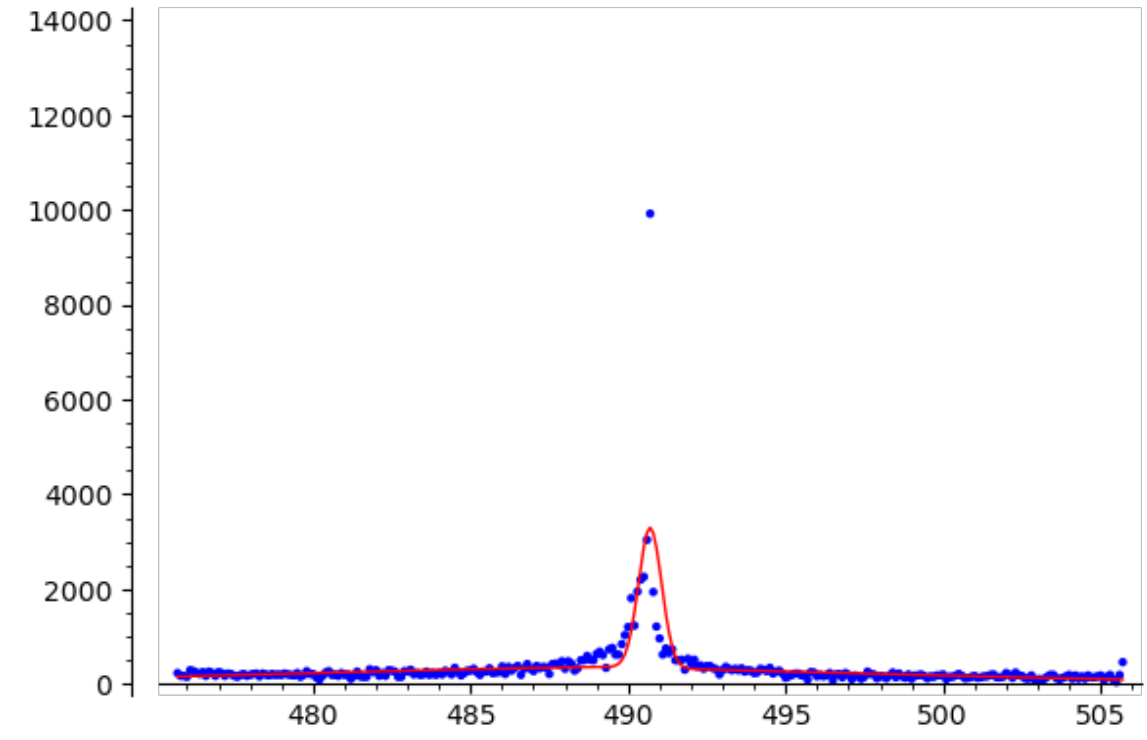
Out[121]:
```
{'a': 1.0569348604789939,
 'b': -18.833386069345213,
 'Nobs': 100130.0,
 'graph': Graphics object consisting of 2 graphics primitives,
 'Var obs': 414082.9125138428,
 'Var theoretical': 227132.9376678212,
 'R2': 0.5485204310627722}
```

In [122…
```
MKOR['graph'].show()
```



In [123…
```
list_plot(MKC2c['Observed #'],color='blue')+list_plot(MKC2c['Theoretical #'],col
```

Out[123]:



In [124…
```
MKOR['R2']
```

Out[124]: 0.5485204310627722

In [ ]: