In [1]:
```
using Random, LinearAlgebra, Statistics, Distributions, StatsBase, Plots
using BAT, IntervalSets, ValueShapes, TypedTables
import SpecialFunctions
using Profile, ProfileView, QuadGK
```

```
Gtk-Message: 02:33:54.672: Failed to load module "canberra-gtk-module"
Gtk-Message: 02:33:54.673: Failed to load module "canberra-gtk-module"
```

In [3]:
```
Threads.nthreads()
```

Out[3]:
```
1
```

In [19]:
```
## Modified error function.
function my_erf(x,coeff,base)
    return coeff/2*(1 .+ SpecialFunctions.erf.(x)) .+ base
end

## Modified step function.
function my_step(x,coeff,base)
    return coeff/2*(1 .+ sign.(x)) .+ base
end

## Model consisting of two error functions (or step functions).
function count(p::NamedTuple{(:offset, :resolution, :k)}, x)
    step1_coeff = 6+p.k
    step2_coeff = 2-p.k
    if p.resolution> 0.0000001
        step1_x = (x .- p.offset[1])/(sqrt(2)*p.resolution)
        step1 = my_erf(step1_x,step1_coeff,4)

        step2_x = (x .- p.offset[2])/(sqrt(2)*p.resolution)
        step2 = my_erf(step2_x,step2_coeff,0.)
    else
        step1_x = (x .- p.offset[1])
        step1 = my_step(step1_x,step1_coeff,4)

        step2_x = (x .- p.offset[2])
        step2 = my_step(step2_x,step2_coeff,0)
    end
    return step1+step2
end

## Area below the model depending on free params.
function get_integral(p::NamedTuple{(:offset, :resolution, :k)},low, high)
    total,error = quadgk(x -> count(p,x),low,high)
    return total
end

## PDF of the model depending on free params.
function pdf(p::NamedTuple{(:offset, :resolution, :k)},x,low, high)
    total = get_integral(p,low,high)
    value = count(p,x)
    return value/total
end

## A simple sampler to construct log-likelihood.
function sampler(p::NamedTuple{(:offset, :resolution, :k)},n,low,high)
    max = count(p,high)
```

```julia
    arr = Array{Float64}(undef, n)
    i = 1
    while i<=n
        y = rand()*max
        x = rand()*(high-low)+low
        if y <= count(p, x)
            arr[i] = x
            i+=1
        end
    end
    return arr
end


true_par_values = (offset = [99, 150], resolution = 5, k = 0)
arr = sampler(true_par_values,10000,0,500)

## Unbinned log-likelihood
likelihood = let data = arr, f =pdf
    params -> begin
        function event_log_likelihood(event)
            log(f(params,event,0,500))
        end
        return LogDVal(mapreduce(event_log_likelihood, +, data))
    end
end

## Flat priors
prior = NamedTupleDist(
    offset = [Uniform(50, 150), Uniform(80, 220)],
    resolution = Uniform(0,20),
    k = Uniform(-6,2)
)

## Posterior based on tutorial
posterior = PosteriorDensity(likelihood, prior)

## running bat_sample
samples = bat_sample(posterior, MCMCSampling(mcalg = MetropolisHastings(), nst
```

```
[ Info: Initializing new RNG of type Random123.Philox4x{UInt64, 10}
[ Info: Using transform algorithm PriorSubstitution()
[ Info: Trying to generate 1 viable MCMC chain(s).
```

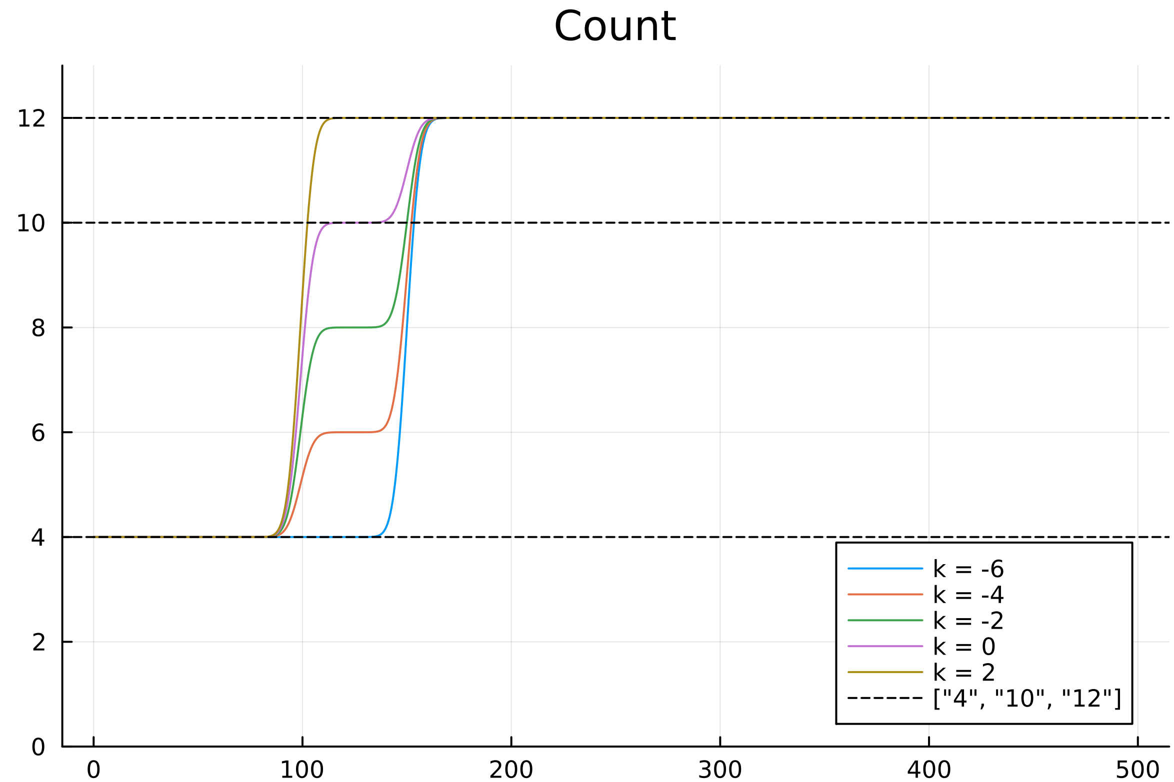
```
AssertionError: length(chains) == nchains

Stacktrace:
 [1] mcmc_init!(rng::Random123.Philox4x{UInt64, 10}, algorithm::MetropolisHast
ings{BAT.MvTDistProposal, RepetitionWeighting{Int64}, AdaptiveMHTuning}, densi
ty::PosteriorDensity{BAT.TransformedDensity{BAT.GenericDensity{var"#36#37"{Vec
tor{Float64}, typeof(pdf)}}, BAT.DistributionTransform{ValueShapes.StructVaria
te{NamedTuple{(:offset, :resolution, :k)}}, BAT.InfiniteSpace, BAT.MixedSpace,
BAT.StandardMvNormal{Float64}, NamedTupleDist{(:offset, :resolution, :k), Tupl
e{Product{Continuous, Uniform{Float64}, Vector{Uniform{Float64}}}, Uniform{Flo
at64}, Uniform{Float64}}, Tuple{ValueAccessor{ArrayShape{Real, 1}}, ValueAcces
sor{ScalarShape{Real}}, ValueAccessor{ScalarShape{Real}}}}}, BAT.TDNoCorr}, BA
T.DistributionDensity{BAT.StandardMvNormal{Float64}, BAT.HyperRectBounds{Float
32}}, BAT.HyperRectBounds{Float32}, ArrayShape{Real, 1}}, nchains::Int64, init
_alg::MCMCChainPoolInit, tuning_alg::AdaptiveMHTuning, nonzero_weights::Bool,
callback::Function)
   @ BAT ~/.julia/packages/BAT/XvOy6/src/samplers/mcmc/chain_pool_init.jl:160
 [2] bat_sample_impl(rng::Random123.Philox4x{UInt64, 10}, target::PosteriorDen
sity{BAT.GenericDensity{var"#36#37"{Vector{Float64}, typeof(pdf)}}, BAT.Distri
butionDensity{NamedTupleDist{(:offset, :resolution, :k), Tuple{Product{Continu
ous, Uniform{Float64}, Vector{Uniform{Float64}}}, Uniform{Float64}, Uniform{Fl
oat64}}, Tuple{ValueAccessor{ArrayShape{Real, 1}}, ValueAccessor{ScalarShape{R
eal}}, ValueAccessor{ScalarShape{Real}}}}}, BAT.HyperRectBounds{Float64}}, BAT.
HyperRectBounds{Float64}, NamedTupleShape{(:offset, :resolution, :k), Tuple{Va
lueAccessor{ArrayShape{Real, 1}}, ValueAccessor{ScalarShape{Real}}, ValueAcces
sor{ScalarShape{Real}}}}}}, algorithm::MCMCSampling{MetropolisHastings{BAT.MvTD
istProposal, RepetitionWeighting{Int64}, AdaptiveMHTuning}, PriorToGaussian, M
CMCChainPoolInit, MCMCMultiCycleBurnin, BrooksGelmanConvergence, typeof(BAT.no
p_func)})
   @ BAT ~/.julia/packages/BAT/XvOy6/src/samplers/mcmc/mcmc_sample.jl:55
 [3] #bat_sample#104
   @ ~/.julia/packages/BAT/XvOy6/src/algotypes/sampling_algorithm.jl:45 [inlin
ed]
 [4] bat_sample
   @ ~/.julia/packages/BAT/XvOy6/src/algotypes/sampling_algorithm.jl:44 [inlin
ed]
 [5] #bat_sample#106
   @ ~/.julia/packages/BAT/XvOy6/src/algotypes/sampling_algorithm.jl:59 [inlin
ed]
 [6] bat_sample(target::PosteriorDensity{BAT.GenericDensity{var"#36#37"{Vector
{Float64}, typeof(pdf)}}, BAT.DistributionDensity{NamedTupleDist{(:offset, :re
solution, :k), Tuple{Product{Continuous, Uniform{Float64}, Vector{Uniform{Floa
t64}}}, Uniform{Float64}, Uniform{Float64}}, Tuple{ValueAccessor{ArrayShape{Re
al, 1}}, ValueAccessor{ScalarShape{Real}}, ValueAccessor{ScalarShape{Real}}}}},
BAT.HyperRectBounds{Float64}}, BAT.HyperRectBounds{Float64}, NamedTupleShape
{(:offset, :resolution, :k), Tuple{ValueAccessor{ArrayShape{Real, 1}}, ValueAc
cessor{ScalarShape{Real}}, ValueAccessor{ScalarShape{Real}}}}}}, algorithm::MCM
CSampling{MetropolisHastings{BAT.MvTDistProposal, RepetitionWeighting{Int64},
AdaptiveMHTuning}, PriorToGaussian, MCMCChainPoolInit, MCMCMultiCycleBurnin, B
rooksGelmanConvergence, typeof(BAT.nop_func)})
   @ BAT ~/.julia/packages/BAT/XvOy6/src/algotypes/sampling_algorithm.jl:57
 [7] top-level scope
   @ In[19]:85
```

In [14]:

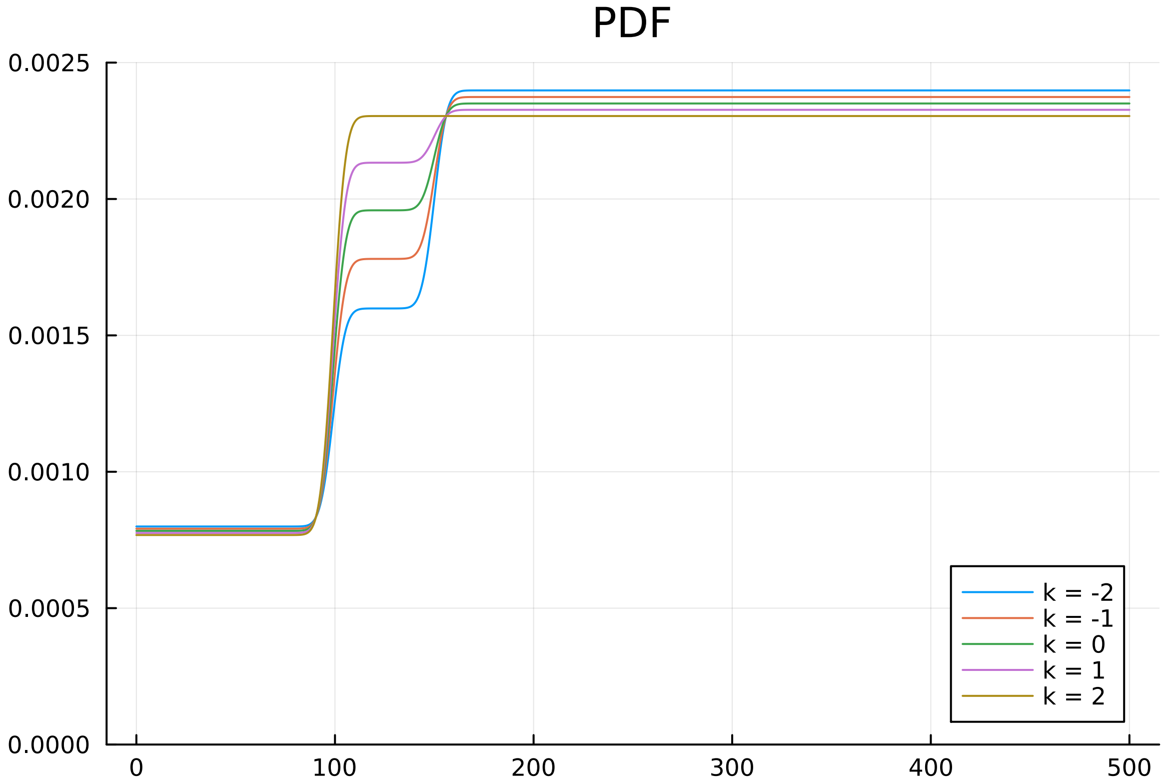
```julia
x = range(0,500,1000)
plt = plot()
for i in -6:2:3
    true_par_values = (offset = [99, 150], resolution = 5, k = i)
    println(true_par_values)
```

```
    ### Plotting count from above
    y = count(true_par_values,x)
    plot!(plt,x,y, label = "k = $(i)")
end
hline!([4,10,12],c=:black,linestyle=:dash,label = ["4","10","12"])
ylims!(0,13)
title!("Count")
display(plt)
```

```
(offset = [99, 150], resolution = 5, k = -6)
(offset = [99, 150], resolution = 5, k = -4)
(offset = [99, 150], resolution = 5, k = -2)
(offset = [99, 150], resolution = 5, k = 0)
(offset = [99, 150], resolution = 5, k = 2)
```



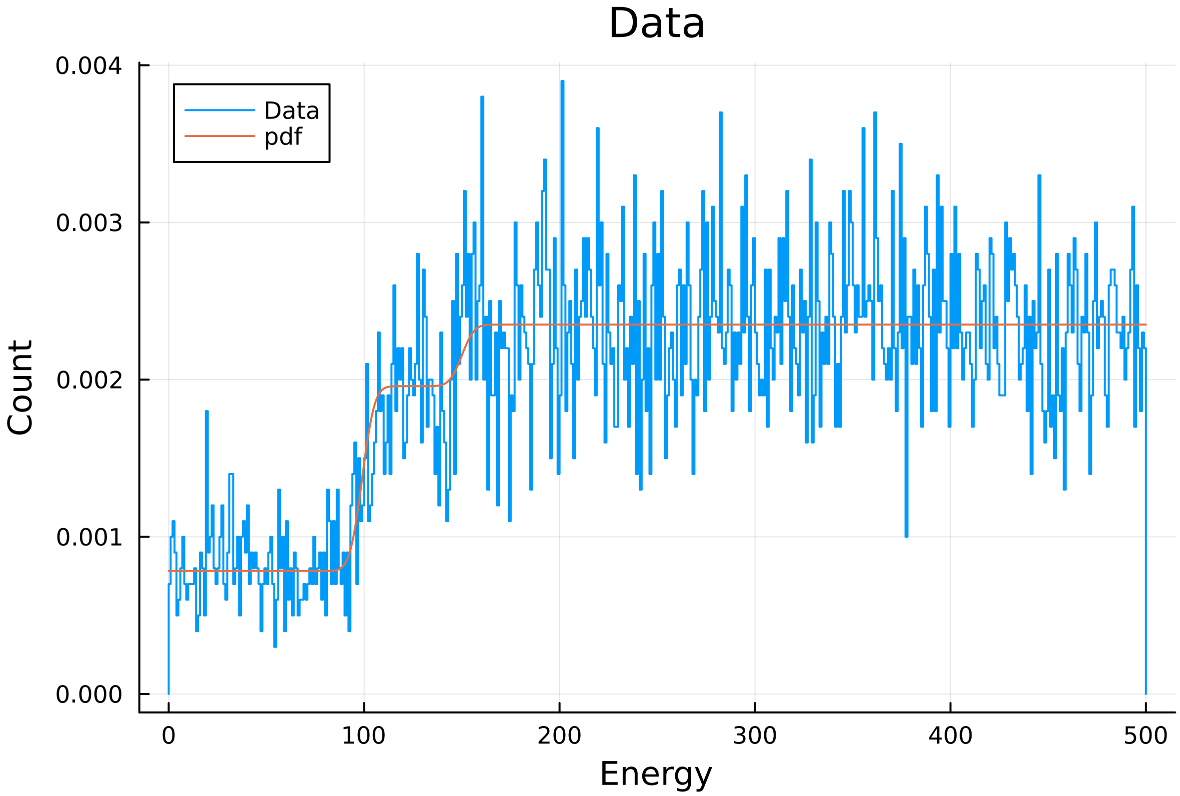
In [13]:
```
x = range(0,500,1000)
configs = [(offset = [99, 150], resolution = 5, k = k) for k in -2:1:2]
plt = plot()
for config in configs
    y = pdf(config,x,0,500)

    ### Plotting PDF from above
    plot!(plt,x,y, label = "k = $(config.k)")
end
ylims!(0,0.0025)
title!("PDF")
display(plt)
```



In [9]:

```
true_par_values = (offset = [99, 150], resolution = 5, k = 0)
arr = sampler(true_par_values,10000,0,500)

hist = append!(StatsBase.Histogram(0:1:500), arr)
plot(StatsBase.normalize(hist, mode=:pdf),st = :step, label = "Data",title = "
x = range(0,500,10000)
y = pdf(true_par_values,x,0,500)
plot!(x,y,label = "pdf")
xlabel!("Energy")
ylabel!("Count")
```

Out[9]:

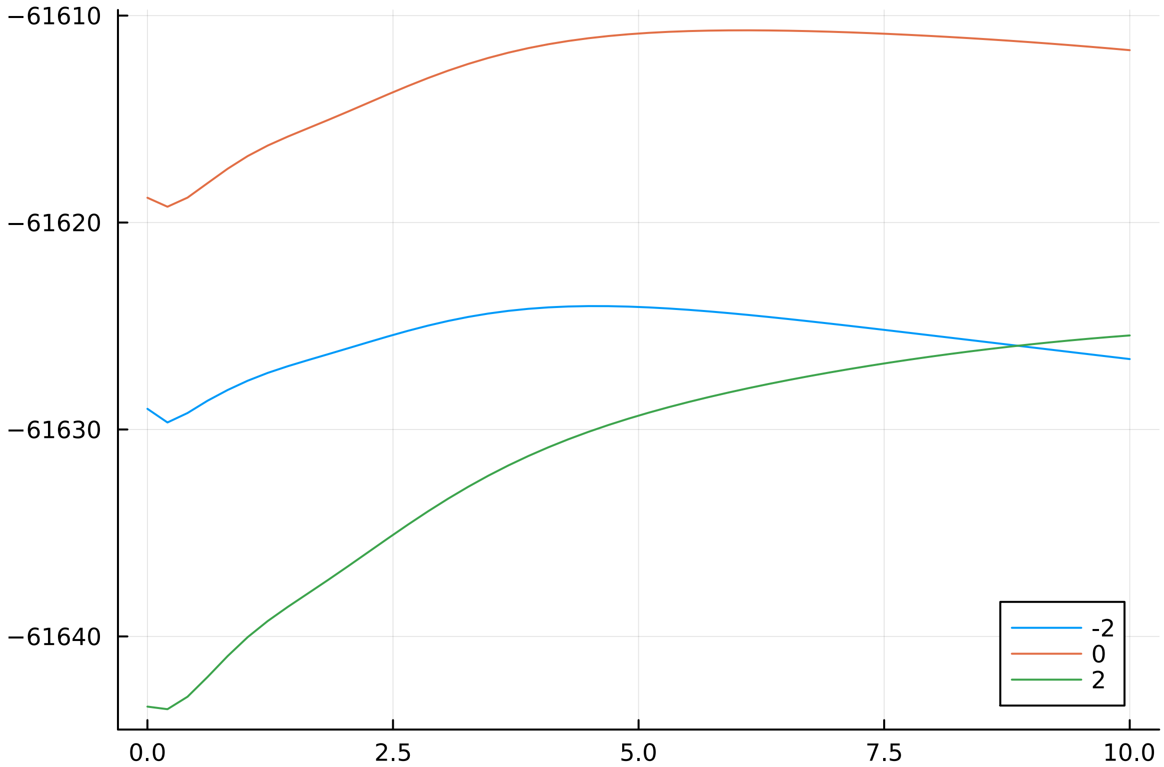


In [16]:

```
true_par_values = (offset = [99, 150], resolution = 5, k = 3)
likelihood(true_par_values)

reso_scan = 50
reso_points = range(0,10,reso_scan)

plt = plot()
for k in -2:2:3
    reso_configs = [(offset = [99, 150], resolution = reso, k = k) for reso in
    y = [logvalof(likelihood(config)) for config in reso_configs]
    plot!(plt,reso_points,y,label = k)
end
```

In [17]:

```
display(plt)
```


−61610
−61620
−61630
−61640
0.0
2.5
5.0
7.5
10.0
-2
0
2