In [233]:
```
using PyPlot
using Statistics
```

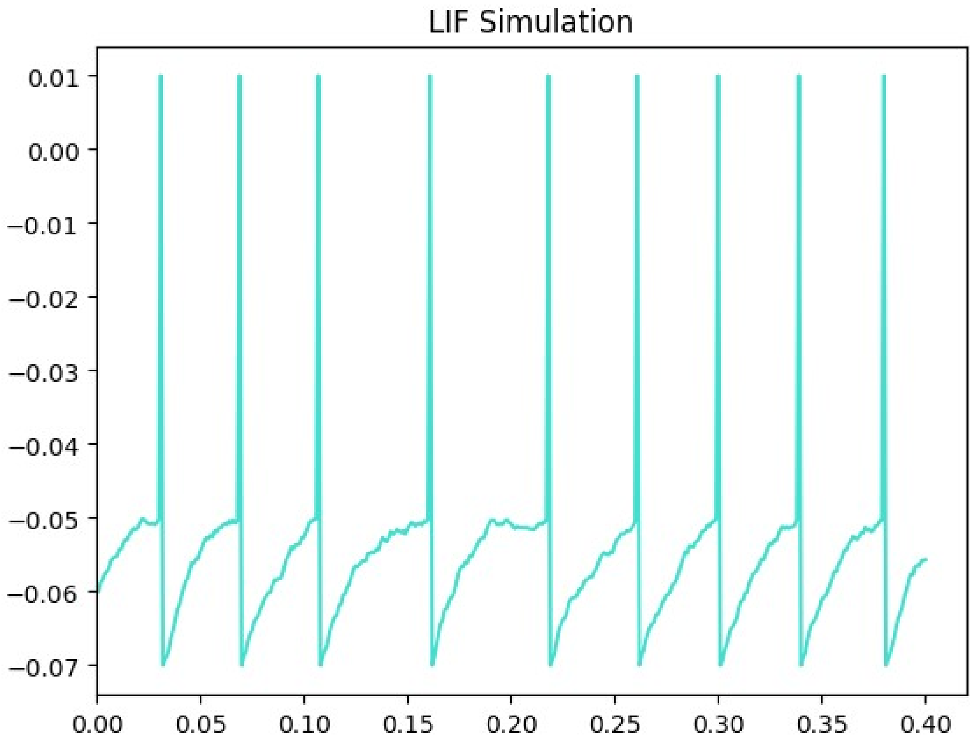
1. The Leaky Integrate-and-Fire (LIF) Neuron

a

In [234]:
```
push!(LOAD_PATH, pwd()); import LIF: LIF_spike
```

b

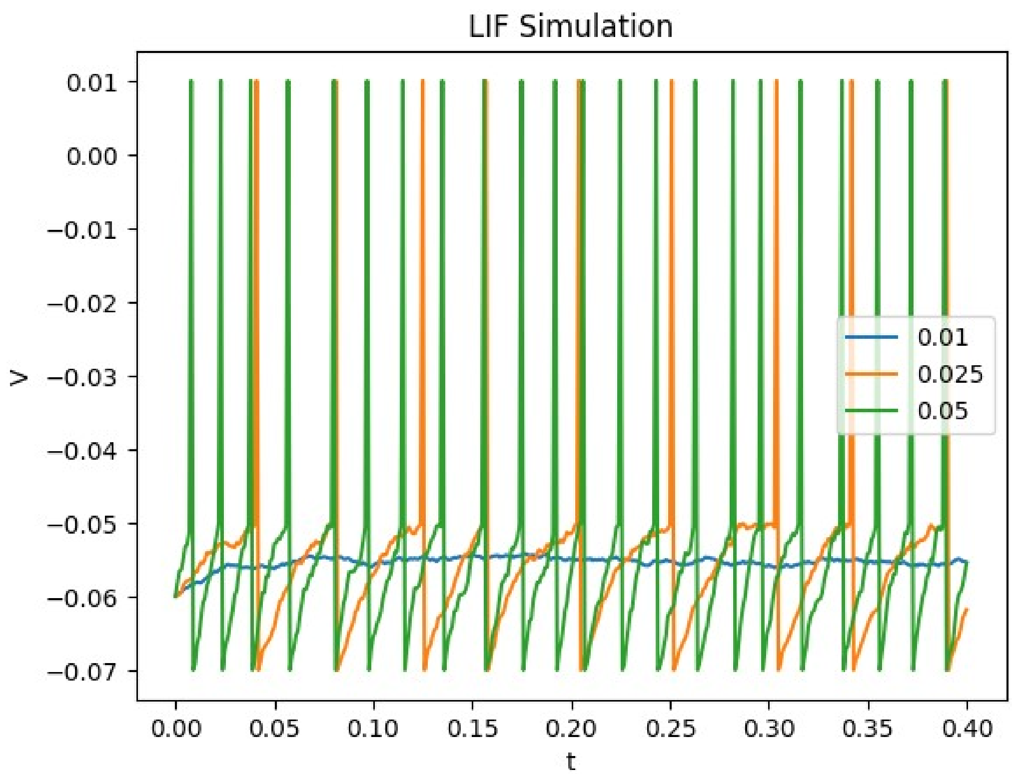
In [309]:
```
time, v, spike_times = LIF_spike()

figure(1)
    title("LIF Simulation")
    plot(time, v, color="turquoise")
    axis(xmin=0)
    println(spike_times)
```




```
Any[0.031, 0.069, 0.107, 0.161, 0.218, 0.261, 0.3, 0.339, 0.38]
```

c

In [305]:
```
means = [ 0.01, 0.025, 0.05];

figure(2)
    title("LIF Simulation")

    xlabel("t")
    ylabel("V")

    for i = 1:length(means)
        time, v, spike_times = LIF_spike(i_mean = means[i]);
        plot(time, v)
    end

    legend(means)
```



Out[305]: PyObject <matplotlib.legend.Legend object at 0x7f1f1ee2be90>

d

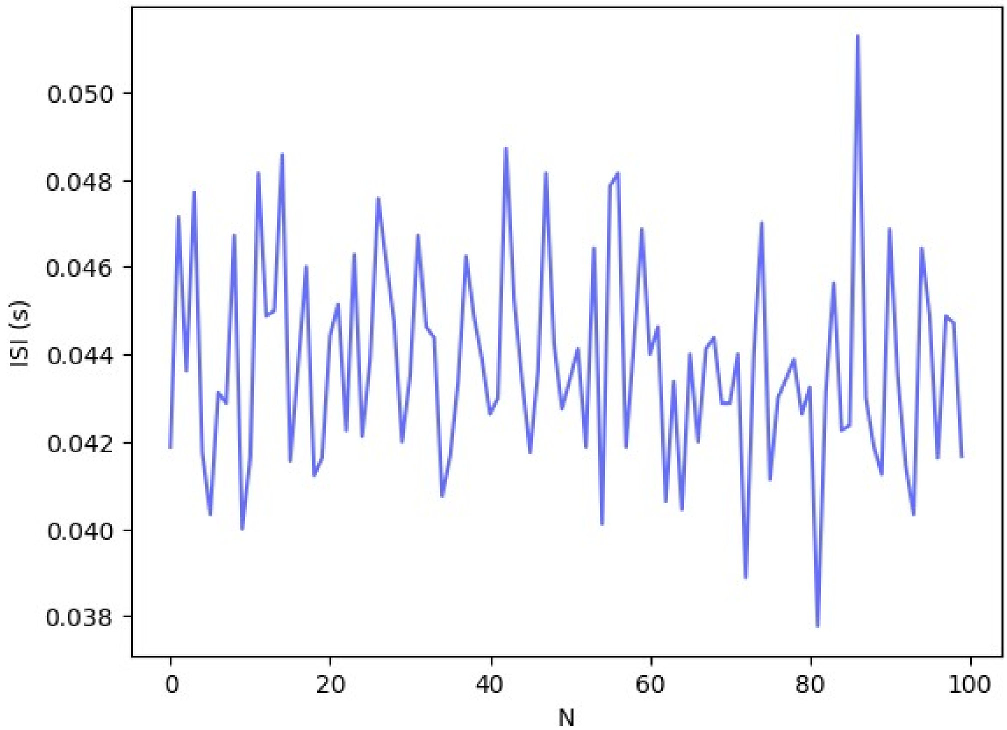
In [240]:

```
"""
avg_ISI(N; i_mean= 25e-3)

This function calculates the average interspike interval (ISI) for one simulation
of the modular implementation of the standard LIF neuron with the LIF_spike functi
on.
It returns one output which is a vector of N length for the average ISIs for each
simulation.
The function can additionally take an optional parameter of specified mean input c
urrent (i_mean).

# PARAMETERS
- N         number of LIF simulations to run

# OPTIONAL PARAMETERS
- i_mean    mean input current


# RETURNS
- ISI     vector representing the average ISI for one simulation

"""

function avg_ISI(N; i_mean = 25e-3)

    ISI = zeros(N);

    for i=1:N
        spike_intervals = []
        time, v, spike_times = LIF_spike(i_mean = i_mean)
        ISI[i]=mean(diff(spike_times))
    end
    return ISI
end
```

Out[240]:

```
avg_ISI (generic function with 1 method)
```

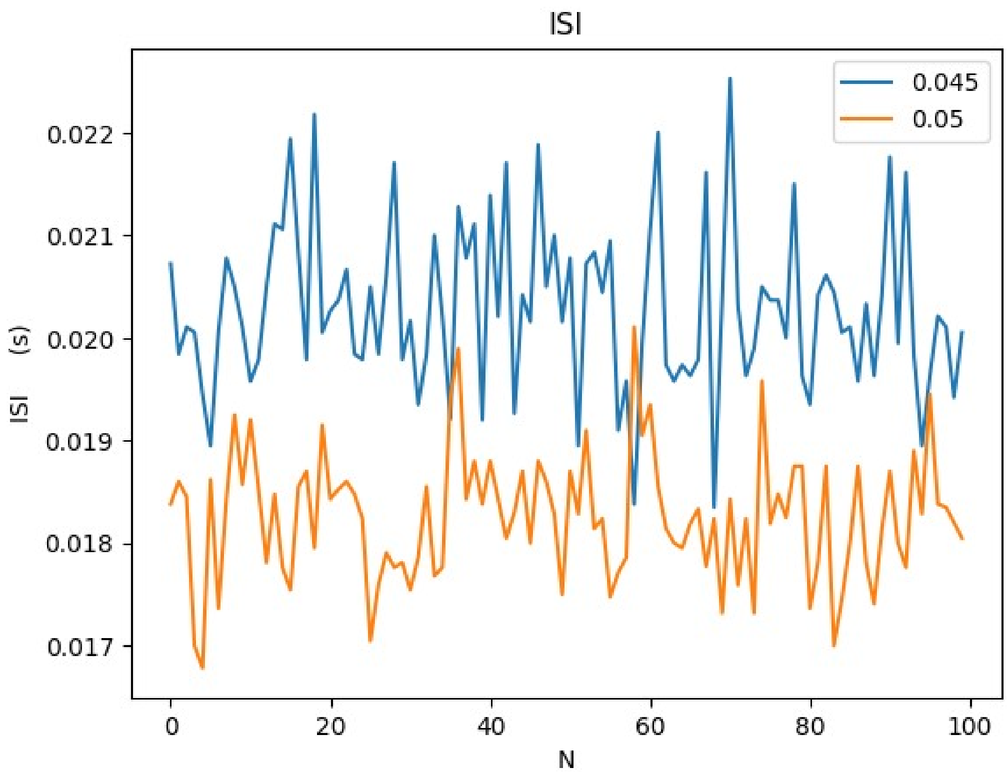
In [244]:
```
ISI = avg_ISI(100)
plot(ISI, color = "#636eff")
xlabel("N")
ylabel("ISI (s)")
```



Out[244]:
```
PyObject Text(24,0.5,'ISI (s)')
```

e

In [246]:
```
i_mean = [0.045, 0.05];

figure(3)
for i = 1:length(i_mean)
    ISI = avg_ISI(100; i_mean = i_mean[i])
    plot(ISI)
end

title("ISI")
legend(i_mean)
xlabel("N")
ylabel("ISI      (s)")
```



Out[246]:
```
PyObject Text(24,0.5,'ISI      (s)')
```

Results

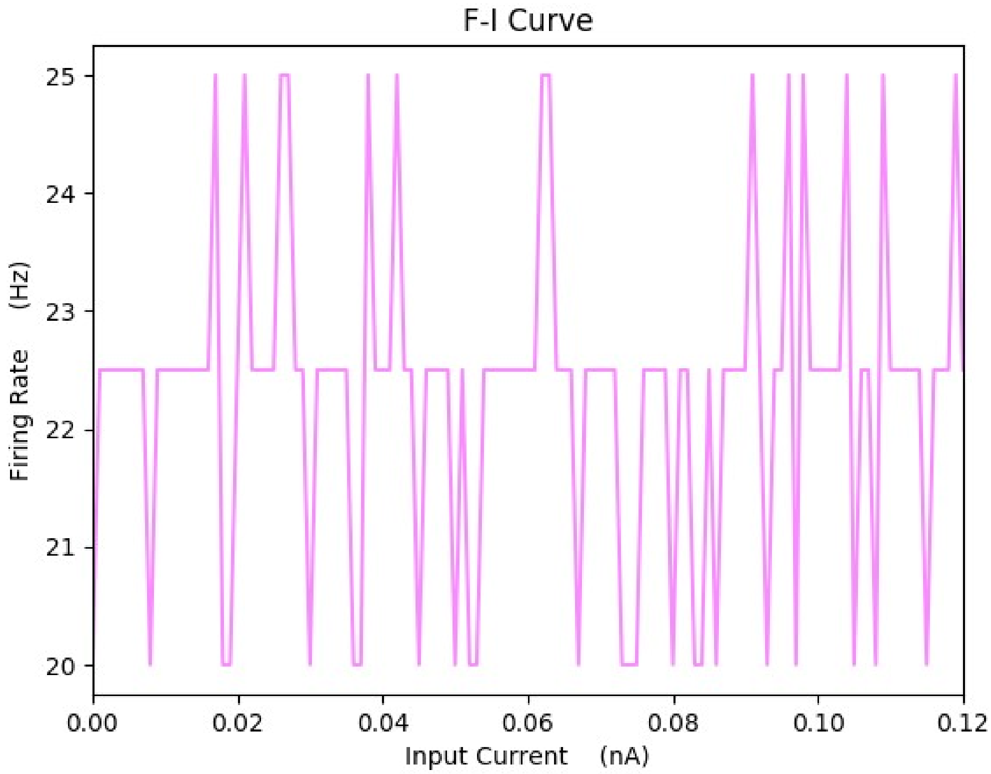
With a lower mean input current (i_mean) (0.045 nA), the interval between each time the neuron fires is longer than it is for a higher mean input current (0.05 nA). This shows that with a greater i_mean the membrane potential of the neuron changes more rapidly and reaches threshold for firing (vth) faster. This is because charged particles diffuse more rapidly across the membrane with a stronger current acting on them cell.

f

In [247]:
```
"""
F_I(i_mean)

This function calculates the firing rates of an LIF neuron as a function
of the mean input current.

# PARAMETERS
- i_mean    mean input current

# RETURNS
- firing_rate     vector representing the firing rate for each input current to th
e LIF neuron

"""
function F_I(i_mean)

    firing_rates = []
    for i=1:length(i_mean)
       time, v, spike_times = LIF_spike()
        push!(firing_rates, length(spike_times)/.4);
    end

    return firing_rates
end
```

Out[247]:
```
F_I
```

In [251]:
```
i_mean = collect(0:0.001:0.12);
firing_rates = F_I(i_mean);

figure(4)
    plot(i_mean, firing_rates, color="#f98aff")
    title("F-I Curve")
    xlabel("Input Current     (nA)")
    ylabel("Firing Rate      (Hz)")
    axis(xmin = 0, xmax = i_mean[end])
```



Out[251]: (0, 0.12, 19.75, 25.25)

g

In [19]:
```
# this is a mess - just a scramble of code I tried and failed with!

t_traces = []

time, v, spike_times = LIF_spike(t_max = 0.9, dt = 0.001)

times = []
for i=1:length(spike_times)
    t = findall(spike_times)
    println(findall(t == time))

end

time, v, spike_times = LIF_spike(t_max = 0.9, dt = 0.001)


i_time = []

for i=1:length(spike_times)
    index = findall(time .== spike_times[i]) #find index in time for spike_times
    push!(i_time, index)
end

#this is not working :(

for i=1:length(i_time)
    t = i_time[i]
    println(t)
    t_step = time[t -15]
    println(t_step)
end

#push!(times, t)
#t = spike_times[i]-15*dt:dt:spike_times[i]-dt
```

```
TypeError: non-boolean (Float64) used in boolean context

Stacktrace:
 [1] iterate at ./iterators.jl:434 [inlined]
 [2] iterate at ./generator.jl:44 [inlined]
 [3] grow_to!(::Array{Int64,1}, ::Base.Generator{Base.Iterators.Filter{typeof(la
st),Base.Iterators.Pairs{Int64,Any,LinearIndices{1,Tuple{Base.OneTo{Int64}}},Arr
ay{Any,1}}},typeof(first)}) at ./array.jl:674
 [4] collect at ./array.jl:617 [inlined]
 [5] findall(::Array{Any,1}) at ./array.jl:2008
 [6] top-level scope at ./In[19]:9
```

h

In [252]:
```
"""
serror(N; i_mean= 25e-3)

This function computes the standard error (SEM) of the average interspike interval
s

# PARAMETERS
- N    number of simulations

# OPTIONAL PARAMETERS
- i_mean     mean input current

# OUTPUT
- SEM    standard error of the mean for averaged ISIs


"""
function serror(N; i_mean= 25e-3)

    ISI = avg_ISI(N; i_mean = i_mean)

    mu = 1/N .* sum(ISI)
       #println(mu)

    std = sqrt(sum((ISI .- mu.^2) ./ (N - 1)))
       #println(std)

    SEM = sqrt(std/N)
       #println(SEM)
    return SEM

end
```

Out[252]: serror

¡

In [253]:
```
serror(100; i_mean = 0.3)
```

Out[253]: 0.02560039709511872

¡

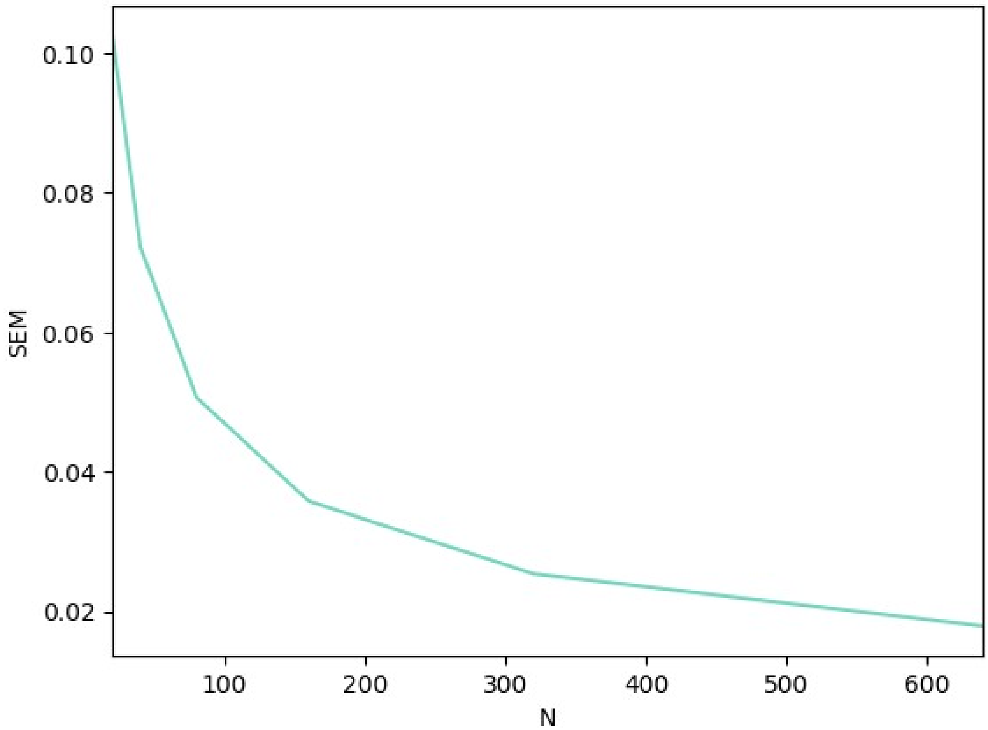
In [301]:

```
"""
serror_plotter(N; i_mean= 25e-3)

This function plots the standard errors (SEM) of the average interspike intervals
for N number of simulations

# PARAMETERS
- N     number of simulations

# OPTIONAL PARAMETERS
- i_mean      mean input current

# OUTPUT
- figure    plots the SEM as a function of N


"""
function serror_plotter(N::Array; i_mean= 25e-3)

    SEM =[]
        for i = 1:length(N)
            push!(SEM, serror(N[i]; i_mean = i_mean))
        end
    figure(1)
        plot(N, SEM, color = "#74d6bf", label="SEM for N simulations")
        axis(xmin=N[1], xmax=N[end])
        xlabel("N")
        ylabel("SEM")
      return SEM
end
```

Out[301]:

```
serror_plotter
```

In [297]:
```
N = [20, 40, 80, 160, 320, 640]

    serror_plotter(N)
```



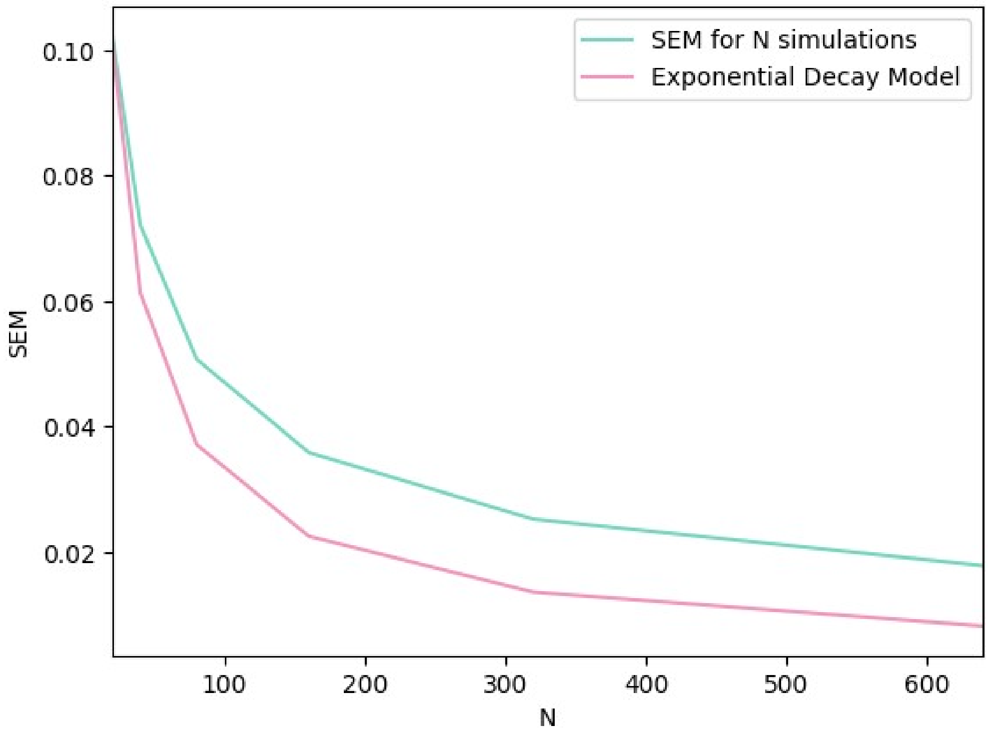
Out[297]:
```
6-element Array{Any,1}:
 0.10250593155912124
 0.07217670438895694
 0.05062055920210368
 0.035752088552001356
 0.025347764212498996
 0.017887709095530387
```

This looks like a function of exponential decay.

To reduce the error bars by a factor of 4, one would have to run the simulation a multiple of 2^4 of N simulations one has performed. In this case, that would be 320*(2^4) = 5120 test runs!

In [302]:

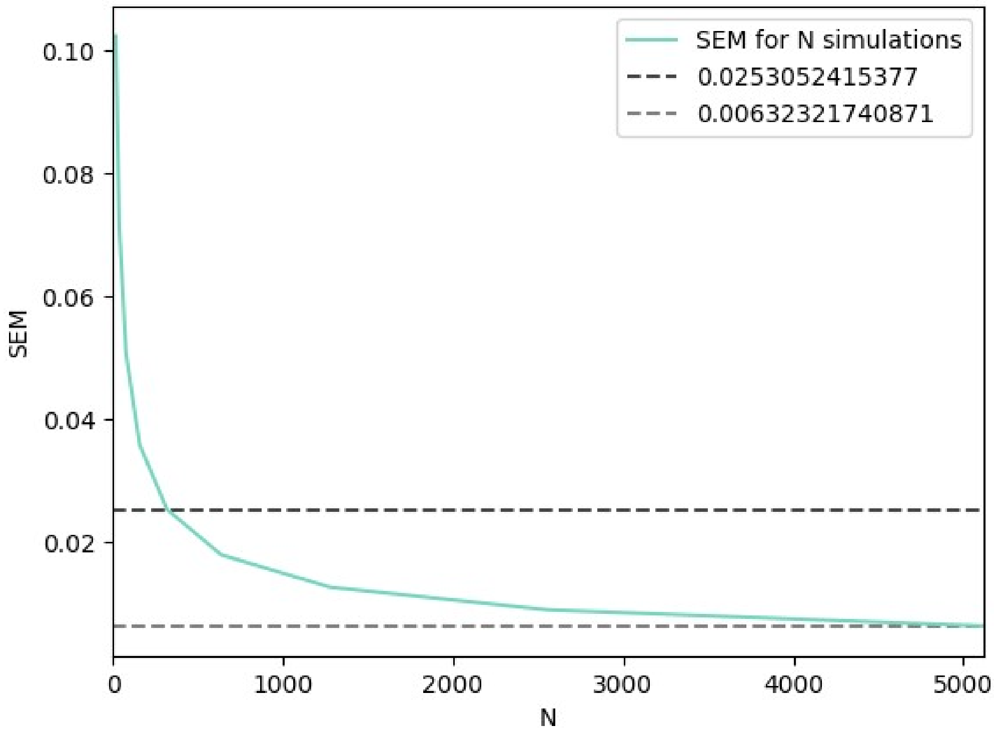
```
### Just taking a look at a model of exponential decay

e = []
    for i=1:length(N)
        push!(e,1/length(N)*2.71828^(-i/2))

    end
println(e)

serror_plotter(N)
plot(N,e, color = "#f294bb", label="Exponential Decay Model")
axis(xmin=N[1], xmax=N[end])
legend()
```




```
Any[0.101088, 0.0613133, 0.0371884, 0.0225559, 0.0136809, 0.00829786]
```

Out[302]: PyObject <matplotlib.legend.Legend object at 0x7f1f1e65bd10>

In [303]:
```
N = [20, 40, 80, 160, 320, 640, 1280, 2560, 5120]

SEM = serror_plotter(N)


# Fun plot
hlines(SEM[5],0, N[end], colors = "#454545", linestyles="dashed", label=SEM[5])
hlines(SEM[end],0, N[end], colors = "grey", linestyles="dashed", label=SEM[end])
    axis(xmin = 0, xmax = N[end])

    legend()
    println("For N=5120 --- SEM(N) = ", SEM[5]/4)
    println(" is ~= ")
    println("for N=320 --- SEM(N)/4 =", SEM[end])
```




```
For N=5120 --- SEM(N) = 0.006326310384433304б
 is ~=
for N=320 --- SEM(N)/4 =0.006323217408710183
```