Walkthrough of Omega Lokta-Volterra Implementation

In this notebook we will walkthrough our first implementation of Lokta-Volterra using Omega. The simulation is done using the Gillespie algorithm to implement stochastic variation. This implementation was built to use an Omega intialized random variable for every step of Gillespie. This allows us to directly intervene and condition on individual random variables. A downside to this approach is that it can take more time to run.

Load Packages

In [ ]:

```
## Load Packages
using Omega
using StatsBase
using Random
using Plots
using Distributions
```

Main Functions

The main functions of the implementation are defined below. These functions are built to repeat at each step of Gillespie and reference the previous step.

In [2]:

```
function get_hazards(rng, n)
    """
    Compute the hazard function given the current states. "spawn_prey" represents
    the event of a prey being born, "prey2pred" represents a predator consuming
    a new prey and consequently spawning a new predator, "pred_dies" represents
    the death of a predator. The function probabilistically selects one of these
    based on their weights.

    args:
        rng(): julia base random number generator, do not need to explicitly pass
        n(int): An index to the current step in prey and pred lists. Used to pull
                most recent values for calculations
    """

    ecology = Dict("prey" => prey_list[n](rng), "pred" => pred_list[n](rng))

    hazards = Dict(
        "spawn_prey" => theta(rng)["spawn_prey"] * ecology["prey"],
```

```julia
        "prey2pred" => theta(rng)["prey2pred"] * ecology["prey"] * ecology["pred"],
        "pred_dies" => theta(rng)["pred_dies"] * ecology["pred"]
        )

    vals = collect(values(hazards))
    sum_vals = sum(vals)
    prob_vals = vals/sum_vals
    categorical(rng, prob_vals)
end

function one_simulation_prey(rng, n, transitions)

    """
    Simulates one step of gillespie for prey. Takes generated hazards
    adds it to prey and outputs the new value.

    args:
        rng(): julia base random number generator, do not need to explicitly pass
        n(int): An index to the current step in prey and pred lists. Used to pull
                most recent values for calculations
        transitions: Matrix that determines how much prey/pred should change based
                     on selected hazard
    """

    hazard_result = hazards_list[n](rng)
    prey_val = prey_list[n](rng)
    labels = ["spawn_prey", "pred_dies","prey2pred"]
    transition = transitions[labels[hazard_result]]
    new_prey = prey_val + transition[1]

    # Enforce only positive integers
    max(1, new_prey)

end

function one_simulation_pred(rng, n, transitions)

    """
    Simulates one step of gillespie for pred. Takes generated hazards
    adds it to pred and outputs the new value.

    args:
        rng(): julia base random number generator, do not need to explicitly pass
        n(int): An index to the current step in prey and pred lists. Used to pull
                most recent values for calculations
        transitions: Matrix that determines how much prey/pred should change based
```

```
                    on selected hazard
    """

    hazard_result = hazards_list[n](rng)
    pred_val = pred_list[n](rng)
    labels = ["spawn_prey", "pred_dies","prey2pred"]
    transition = transitions[labels[hazard_result]]
    new_pred = pred_val + transition[2]

    # Enforce only positive integers
    max(1, new_pred)

end
```

Out[2]:
```
one_simulation_pred (generic function with 1 method)
```

In [3]:
```
function generate_rates(rng)

    """
    Creates a dictionary with rates needed to calculate hazards. Created in
    a single function so that rates are only generated once and then used
    for the entire simulation.
    """

    Dict("spawn_prey" => spawn_prey(rng),
         "prey2pred" => prey2pred(rng),
         "pred_dies" => pred_dies(rng))

end
```

Out[3]:
```
generate_rates (generic function with 1 method)
```

Build Simulation

In order to simulate Lotka-Volterra we need to use the above functions to create all the steps of Gillespie.

Initalize Parameters

In [4]:
```
## Transition Matrix
Pre = [[1, 0], [1, 1], [0, 1]]
Post = [[2, 0], [0, 2], [0, 0]]
transition_mat = Post - Pre
transitions = Dict("spawn_prey" => transition_mat[1,],
                    "prey2pred" => transition_mat[2,],
                    "pred_dies" => transition_mat[3,])
```

Out[4]:
```
Dict{String,Array{Int64,1}} with 3 entries:
  "spawn_prey" => [1, 0]
  "pred_dies"  => [0, -1]
  "prey2pred"  => [-1, 1]
```

Rates and starting values determine the entire trajectory of the simulation. The lower trajectory was used to recreate the plots in the Omega paper. The higher gives better visualization for this specific simulation (can see the stochasiticity better). There could be some arguement whether the rates should be random variables, but we created them that way here in order to test conditionals and interventions.

One important aspect for reproducibility is to use a random seed. In jupyter notebook we need to intialize the seed in every code block.

In [5]:
```
# Random variables for small cycles
# prey_init = normal(10., .001)
# pred_init = normal(10., .001)

# # Random variables for rates
# spawn_prey = normal(1.5, .01)
# prey2pred = normal(1.0, .0001)
# pred_dies = normal(3.0, .0075)

# Random variables for large cycles
prey_init = normal(50., .001)
pred_init = normal(100., .001)

# Random variables for rates
spawn_prey = normal(.9, .01)
prey2pred = normal(.004, .0001)
pred_dies = normal(.4, .0075)

Random.seed!(1234)
theta = ciid(generate_rates)
```

Out[5]:
```
6:generate_rates()::Any
```

Generate Random Var List (Passed to rand function)

In this section we loop over how many steps we want in the algorithm and create the random variables needed. These random variables are then compiled into a single tuple which can be passed to the rand() function.

In [6]:
```
## Initialize lists and add starting rand vars
```

```
hazards_list = Any[]
prey_list = Any[]
pred_list = Any[]
push!(prey_list, prey_init)
push!(pred_list, pred_init)

## How many time periods to cycle over
N = 5000

## Create a prey/pred/hazard for each time period
for f in 2:N
    last = f - 1
    hazards_temp = ciid(get_hazards, last) # individual step
    prey_temp = ciid(one_simulation_prey, last, transitions) # individual step
    pred_temp = ciid(one_simulation_pred, last, transitions) # individual step
    push!(hazards_list, hazards_temp)
    push!(prey_list, prey_temp)
    push!(pred_list, pred_temp)
end

## Convert lists to single tuple
random_var_tuple = (Tuple(x for x in hazards_list)...,
                    Tuple(x for x in prey_list)...,
                    Tuple(x for x in pred_list)...,
                    Tuple(Any[spawn_prey,prey2pred,pred_dies,theta])...)
print()
```

Simulations

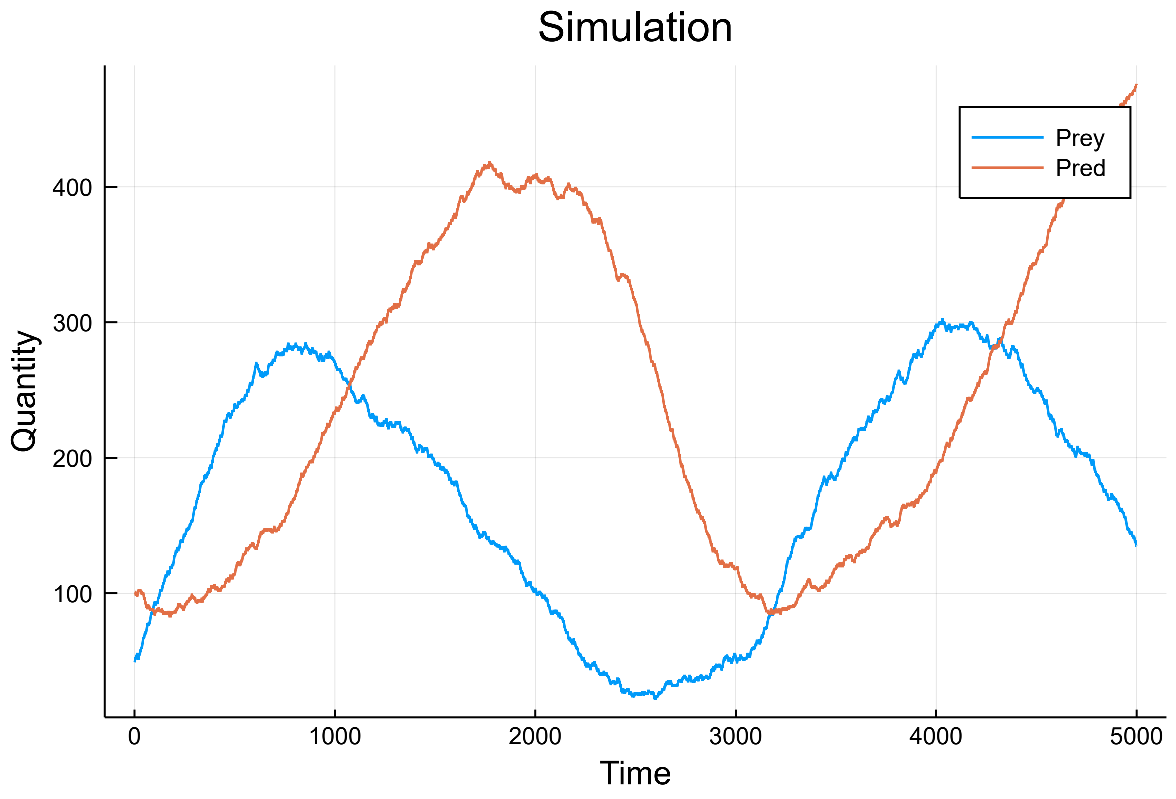
Simple Simulation

We will run the first simulation without any conditionals or interventions.

In [7]:
```
## Sample
Random.seed!(1234) ## Must initialize seed each time if run line by line
samples = rand(random_var_tuple, 1, alg = RejectionSample)

# extract run results and plot
prey_vals = []
pred_vals = []
for x in 1:(N-1)
    push!(prey_vals,samples[1][N+x])
    push!(pred_vals,samples[1][(N*2)+x])
end
```

```
plot(hcat(prey_vals,pred_vals),
        title = "Simulation",
        xlabel = "Time",
        ylabel = "Quantity",
        label = ["Prey" "Pred"],
        lw = 1.25)
```

Out[7]:



Conditional Simulation

Next we will condition the model on some restrictions are run simulations with the conditional.

Recreate Model with Conditional Rand Vars

If model needs to condition on multiple values this is the easiest way to execute it. A single conditional can simply be passed to the rand function.

In [6]:

```
## Initialize lists and add starting rand vars
hazards_list = Any[]
```

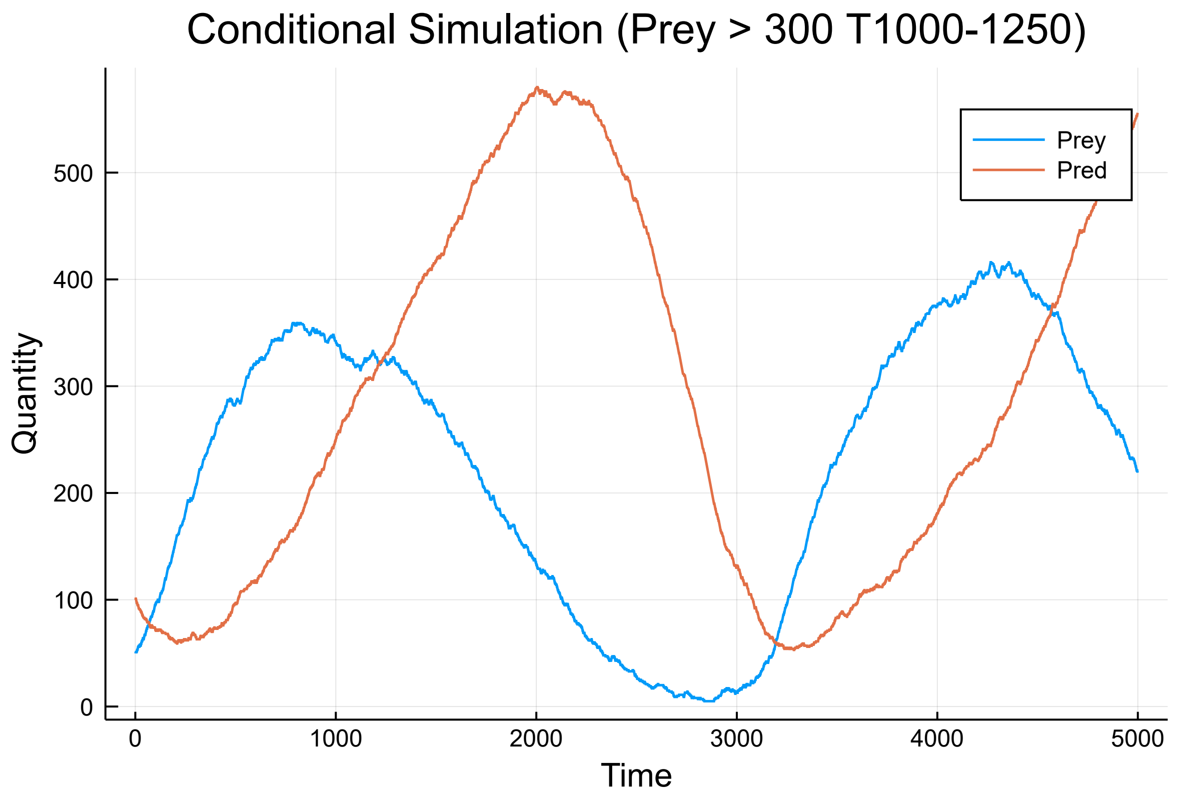
```julia
prey_list = Any[]
pred_list = Any[]

push!(prey_list, prey_init)
push!(pred_list, pred_init)

## How many time periods to cycle over
N = 5000

## Indices to condition on
condition_index = 1000:1250

## Create a prey/pred/hazard for each time period
for f in 2:N
    last = f - 1
    hazards_temp = ciid(get_hazards, last) # individual step

    ## Condition on prey in this example
    if f in condition_index
        prey_temp_ = ciid(one_simulation_prey, last, transitions)
        prey_temp = cond(prey_temp_, prey_temp_ > 300) ## Conditional
    else
        prey_temp = ciid(one_simulation_prey, last, transitions)
    end

    pred_temp = ciid(one_simulation_pred, last, transitions) # individual step

    push!(hazards_list, hazards_temp)
    push!(prey_list, prey_temp)
    push!(pred_list, pred_temp)
end

random_var_tuple = (Tuple(x for x in hazards_list)...,
                Tuple(x for x in prey_list)...,
                Tuple(x for x in pred_list)...,
                Tuple(Any[spawn_prey,prey2pred,pred_dies,theta])...)
print()
```

Next we just simulate as before. One note here is that using the rejection sampling algorithm for conditionals can be dangerous. If the conditional is too restrictive, it can run forever. An alternative to this is to use the soft execution and a different algorithm (SSMH/NUTS) built into Omega. Here our conditional is not that restrictive and this is not necessary.

In [7]:
```julia
Random.seed!(1234)
samples = rand(random_var_tuple,
                1, alg = RejectionSample)
```

```
# extract run results and plot
prey_vals = []
pred_vals = []
for x in 1:(N-1)
    push!(prey_vals,samples[1][N+x])
    push!(pred_vals,samples[1][(N*2)+x])
end

plot(hcat(prey_vals,pred_vals),
        title = "Conditional Simulation (Prey > 300 T1000-1250)",
        xlabel = "Time",
        ylabel = "Quantity",
        label = ["Prey" "Pred"],
        lw = 1.25)
```

Out[7]:



Intervention/Counterfactual

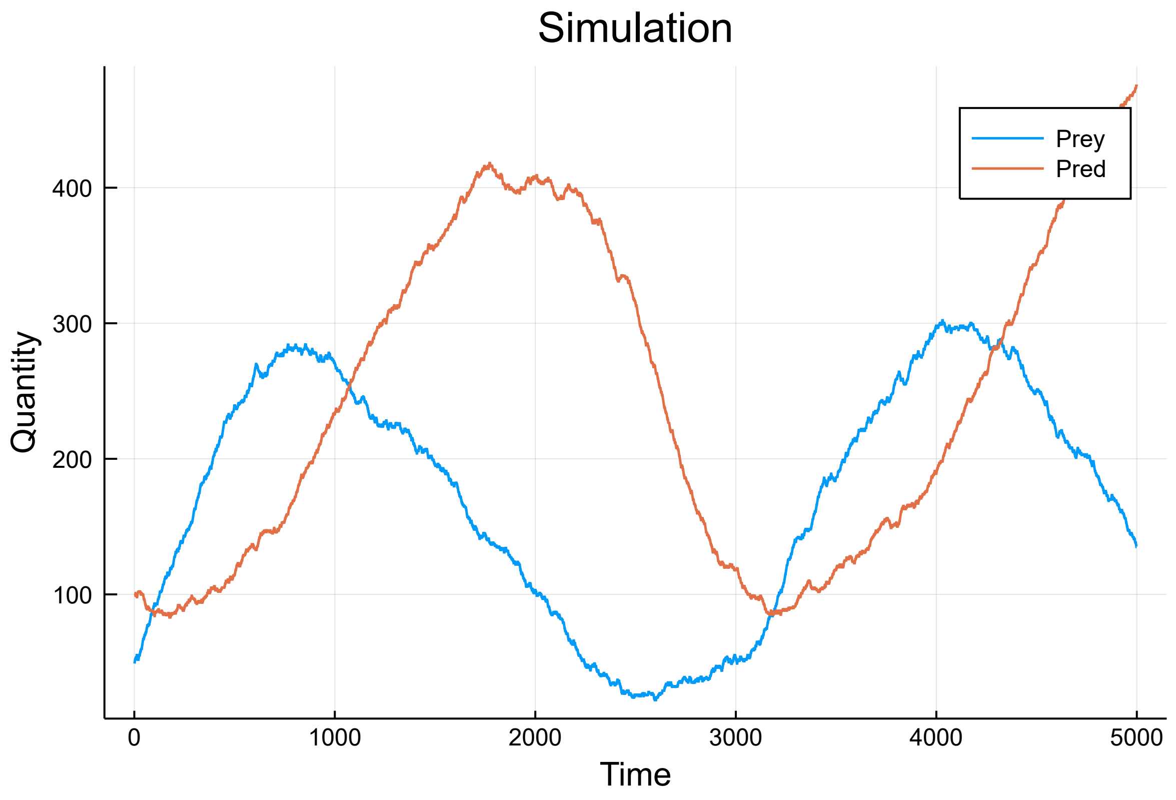
Finally our model also allows us to run interventions and counterfactuals.

Recreate Original Model for comparison

In [8]:
```
## Initialize lists and add starting rand vars
hazards_list = Any[]
prey_list = Any[]
pred_list = Any[]
push!(prey_list, prey_init)
push!(pred_list, pred_init)

## How many time periods to cycle over
N = 5000

## Create a prey/pred/hazard for each time period
for f in 2:N
    last = f - 1
    hazards_temp = ciid(get_hazards, last) # individual step
    prey_temp = ciid(one_simulation_prey, last, transitions) # individual step
    pred_temp = ciid(one_simulation_pred, last, transitions) # individual step
    push!(hazards_list, hazards_temp)
    push!(prey_list, prey_temp)
    push!(pred_list, pred_temp)
end

## Convert lists to single tuple
random_var_tuple = (Tuple(x for x in hazards_list)...,
                Tuple(x for x in prey_list)...,
                Tuple(x for x in pred_list)...,
                Tuple(Any[spawn_prey,prey2pred,pred_dies,theta])...)
print()
```

In [9]:
```
## Original Sim
Random.seed!(1234)
samples = rand(random_var_tuple, 1, alg = RejectionSample)

# extract run results and plot
prey_vals = []
pred_vals = []
for x in 1:(N-1)
    push!(prey_vals,samples[1][N+x])
    push!(pred_vals,samples[1][(N*2)+x])
end

plot(hcat(prey_vals,pred_vals),
        title = "Simulation",
        xlabel = "Time",
        ylabel = "Quantity",
        label = ["Prey" "Pred"],
        lw = 1.25)
```

Out[9]:



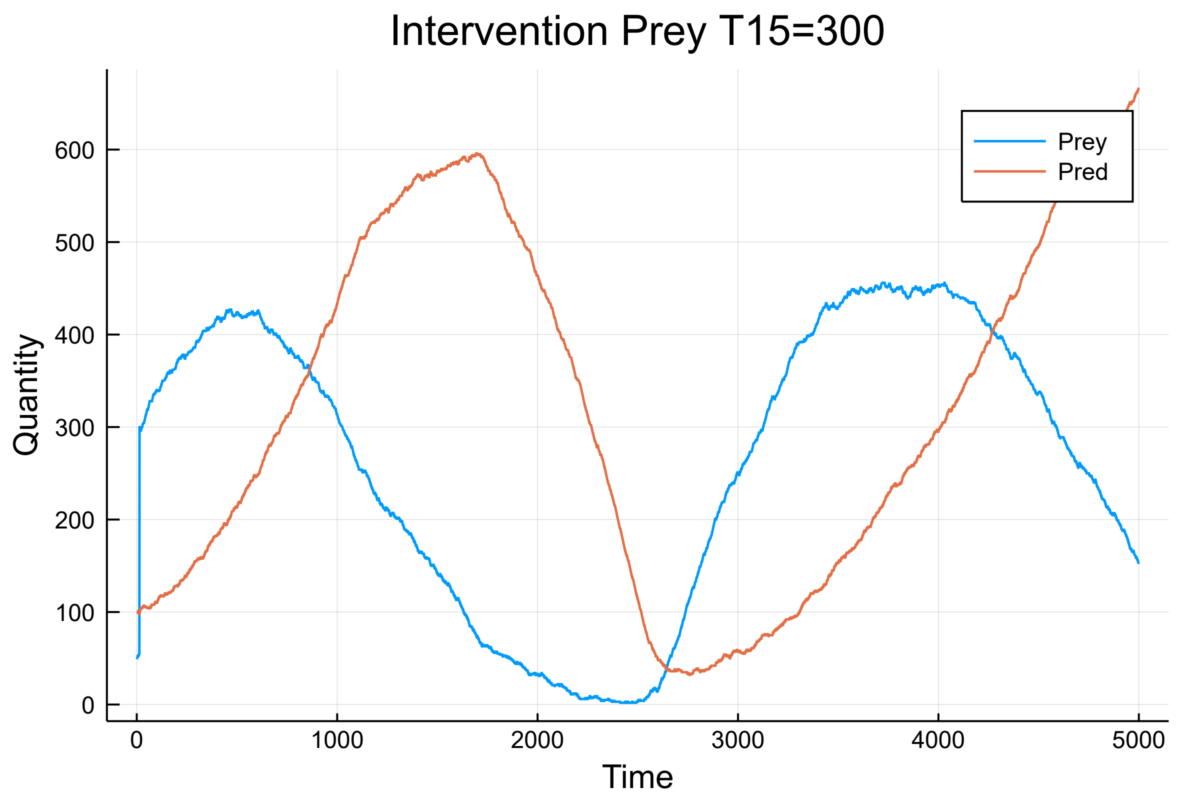
Intervene by increasing prey early

Here we will intervene by making Prey at time 15 equal 300. This is where this implementation starts hitting preformance issues due to the very large amount of random variables in the simulation. The intervention is done early here so that this notebook can be reproduced quickly.

In [10]:

```
## Make intervention and replace rand var with intervention var
replace_index = 15
new_prey = replace(prey_list[replace_index],
                    prey_list[replace_index - 1] => 300.0)
prey_list_replace = prey_list
prey_list_replace[replace_index] = new_prey ## Replace with intervention

## Run sim as normal after intervention
random_var_tuple_int = (Tuple(x for x in hazards_list)...,
                Tuple(x for x in prey_list_replace)...,
                Tuple(x for x in pred_list)...,
                Tuple(Any[spawn_prey,prey2pred,pred_dies,theta])...)

Random.seed!(1234) ## Same seed as above
adj_samples = rand(random_var_tuple_int, 1, alg = RejectionSample)

# extract run results and plot
prey_vals = []
pred_vals = []
for x in 1:(N-1)
    push!(prey_vals,adj_samples[1][N+x])
    push!(pred_vals,adj_samples[1][(N*2)+x])
end

plot(hcat(prey_vals,pred_vals),
        title = "Intervention Prey T15=300",
        xlabel = "Time",
        ylabel = "Quantity",
        label = ["Prey" "Pred"],
        lw = 1.25)
```

Out[10]:



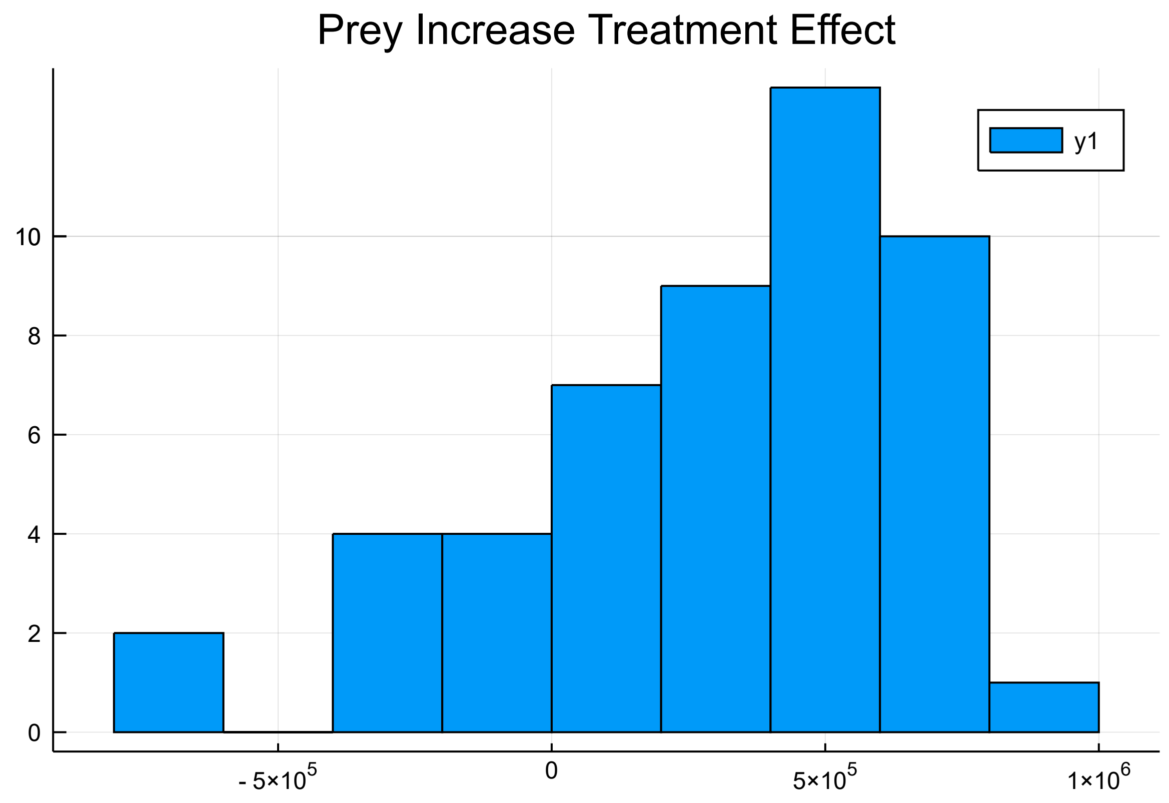
Histogram of Difference in Total Prey

We can use these interventions to answer treatment effect questions such as "If we intervened at time 15 and added enough prey to equal 300, what would be the effect on the total prey of the simulations?"

One way to answer this is to run many simulations, apply our intervention and looking at the difference in total prey between the original run and intervened run. Note because we need to run this multiple times it can take a little while.

In [11]:
```
function increase_prey()

    ## New seed each run
    Random.seed!(rand(1:10000))
    ## Run normal and intervened sims
    samples = rand(random_var_tuple, 1, alg = RejectionSample)
    adj_samples = rand(random_var_tuple_int, 1, alg = RejectionSample)

    # extract run results for prey
    prey_norm_vals = []
    prey_adj_vals = []
    for x in 1:(N-1)
        push!(prey_norm_vals,samples[1][N+x])
        push!(prey_adj_vals,adj_samples[1][(N*2)+x])
    end

    ## Calculate difference in total prey
    return sum(prey_adj_vals) - sum(prey_norm_vals)
end

check = [increase_prey() for x=1:50]

histogram(check, bins = 10,
        title = "Prey Increase Treatment Effect")
```

Out[11]:



In [ ]: