SMA Stable Point Analyses

September 2, 2022

[1]:
```
# Import necesary packages:
import sympy
import math
import copy
```

[2]:
```
#Set up sympy to display outputs after a latex conversion.
from sympy.interactive import printing
printing.init_printing(use_latex=True)
```

[3]:
```
N, S, I_W, I_Z, I_WZ, beta_W, beta_Z, gamma, gamma_T, epsilon, q, rho, phi =␣
↪sympy.symbols('N S I_W I_Z I_WZ beta_W beta_Z gamma gamma_T epsilon q rho␣
↪phi')
```

1 1 Endemic Equilibria Stability Analyses of the Exclusive Infection Model from Spicknall et al (2013)

1.1 1.1 Deriving Equilibria and Determining Their Stability

[4]:
```
Eqn_S = -beta_W/N*I_W*S-beta_Z/
↪N*I_Z*S+gamma*(1-epsilon)*I_W+gamma_T*epsilon*I_W+gamma*I_Z
Eqn_I_W = beta_W/N*I_W*S-gamma*(1-epsilon)*I_W-gamma_T*epsilon*I_W
Eqn_I_Z = beta_Z/N*I_Z*S-gamma*I_Z
```

[5]:
```
ODEs = sympy.Matrix([Eqn_S,Eqn_I_W,Eqn_I_Z])
ODEs
```

[5]:
$$\begin{bmatrix} I_W \epsilon \gamma_T + I_W \gamma \left(1 - \epsilon\right) - \frac{I_W S \beta_W}{N} + I_Z \gamma - \frac{I_Z S \beta_Z}{N} \\ - I_W \epsilon \gamma_T - I_W \gamma \left(1 - \epsilon\right) + \frac{I_W S \beta_W}{N} \\ - I_Z \gamma + \frac{I_Z S \beta_Z}{N} \end{bmatrix}$$

[6]:
```
equilibria = sympy.solve(ODEs,[S, I_W, I_Z])
equilibria
```

[6]:
$$\left[\left(S,\ 0,\ 0\right), \left(\frac{N \left(-\epsilon\gamma + \epsilon\gamma_T + \gamma\right)}{\beta_W},\ I_W,\ 0 \right), \left(\frac{N\gamma}{\beta_Z},\ 0,\ I_Z \right) \right]$$

The disease free equilibrium (DFE) is the first vector.

[7]:
```
DFE = equilibria[0]
```

We are not interested in whether the disease free equilibrium (DFE), the first equilibria, is stable. This can be found through deriving R0 using PyGom's matrix methodology for determining R0.

Let's look at the stability of the non-DFEs, following recipe 8.2 of Otto & Day (2007). Needs to create Jacobian Matrix of the models ODEs.

[8]:
```
J_of_ODEs = ODEs.jacobian(X=[S,I_W,I_Z])
J_of_ODEs
```

[8]:
$$\begin{bmatrix} -\frac{I_W\beta_W}{N} - \frac{I_Z\beta_Z}{N} & \epsilon\gamma_T + \gamma\left(1-\epsilon\right) - \frac{S\beta_W}{N} & \gamma - \frac{S\beta_Z}{N} \\ \frac{I_W\beta_W}{N} & -\epsilon\gamma_T - \gamma\left(1-\epsilon\right) + \frac{S\beta_W}{N} & 0 \\ \frac{I_Z\beta_Z}{N} & 0 & -\gamma + \frac{S\beta_Z}{N} \end{bmatrix}$$

Substitute the 1st non-disease free equilibrium (non-DFE) into the Jacobian matrix and determine Eigen values.

[9]:
```
non_DFE_1 = {'S':equilibria[1][0],'I_W':equilibria[1][1],'I_Z':equilibria[1][2]}

J_of_ODEs_non_DFE_1 = J_of_ODEs.subs(non_DFE_1)
J_of_ODEs_non_DFE_1

eigs_J_non_DFE_1 = J_of_ODEs_non_DFE_1.eigenvals()
# The above code line enters the Eigen values as keys to a dictionary.
# It would be more useful to have them as elements in a vector.
eigs_J_non_DFE_1 = [k  for  k in  eigs_J_non_DFE_1.keys()]
eigs_J_non_DFE_1
```

[9]:
$$\left[-\frac{\beta_W\gamma + \beta_Z\epsilon\gamma - \beta_Z\epsilon\gamma_T - \beta_Z\gamma}{\beta_W},\ -\frac{I_W\beta_W}{N},\ 0\right]$$

Substitute the 2nd non-disease free equilibrium (non-DFE) into the Jacobian matrix and determine Eigen values.

[10]:
```
non_DFE_2 = {'S':equilibria[2][0],'I_W':equilibria[2][1],'I_Z':equilibria[2][2]}

J_of_ODEs_non_DFE_2 = J_of_ODEs.subs(non_DFE_2)
J_of_ODEs_non_DFE_2

eigs_J_non_DFE_2 = J_of_ODEs_non_DFE_2.eigenvals()
# The above code line enters the Eigen values as keys to a dictionary.
# It would be more useful to have them as elements in a vector.
eigs_J_non_DFE_2 = [k  for  k in  eigs_J_non_DFE_2.keys()]
eigs_J_non_DFE_2
```

[10]:
$$\left[\frac{\beta_W\gamma + \beta_Z\epsilon\gamma - \beta_Z\epsilon\gamma_T - \beta_Z\gamma}{\beta_Z},\ -\frac{I_Z\beta_Z}{N},\ 0\right]$$

1.2 1.2 Function for Determining the Stable Endemic Equilibrium Infection Prevalence

[ ]:
```
# A rounding to sigfig function will prove useful:
def round_sf(number, significant):
    '''
    Rounds to a specified number of significant figures.
    '''
    return round(number, significant - len(str(number)))
```

[12]:
```
def exclus_inf_end_equil(param_values):
    '''
    Calculates the non-disease free equilibria for the exclusive infection␣
↪model ,
    as out lined in Spicknall et al 2013. Returning any non-DFE
    that are biologically reasonable and locally stable.
    '''
    # Note if both strains have R0 values <1 there is no point in using this␣
↪function.

    equil_pops = []
    # Setup Equilibrium populations as an empty list. If any of the endemic
    # equilibria are found to biologically feasible and stable they are␣
↪appended to the list.

    #Basic Reproductive numbers from Spicknall 2013
    R0_W = beta_W/(gamma*(1-epsilon)+gamma_T*epsilon)
    R0_W = R0_W.subs(param_values)
    R0_Z = beta_Z/gamma
    R0_Z = R0_Z.subs(param_values)

    #Formula for the non-Disease Free Equilibria has been worked out in a␣
↪jupyter notebook
    #and pasted here. NOTE with the first non-DFE the I_W population = N-S␣
↪(I_Z=0),
    # and with the second non-DFE the I_Z population = N-S (I_W=0).
    # Therefore, only the formula for S are needed (see jupyter notebook).
    S_non_DFE_1 = N*(-epsilon*gamma + epsilon*gamma_T + gamma)/beta_W
    S_non_DFE_1 = S_non_DFE_1.subs(param_values)
    I_W_non_DFE_1 = param_values['N']-S_non_DFE_1
    I_Z_non_DFE_1 = 0
    S_non_DFE_2 = N*gamma/beta_Z
    S_non_DFE_2 = S_non_DFE_2.subs(param_values)
    I_W_non_DFE_2 = 0
    I_Z_non_DFE_2 = param_values['N']-S_non_DFE_2
```

```
   #The Eigen values associated with Jacobian matricies for the non-DFE have␣
↪been
   #derived in a jupyter notebook and pasted here.
   eigs_J_non_DFE_1 = [-(beta_W*gamma + beta_Z*epsilon*gamma -␣
↪beta_Z*epsilon*gamma_T - beta_Z*gamma)/beta_W,
                       -I_W*beta_W/N,
                       0]
   eigs_J_non_DFE_2 = [(beta_W*gamma + beta_Z*epsilon*gamma -␣
↪beta_Z*epsilon*gamma_T - beta_Z*gamma)/beta_Z,
                       -I_Z*beta_Z/N,
                       0]

   if math.isnan(S_non_DFE_1) or math.isnan(I_W_non_DFE_1) or math.
↪isnan(I_Z_non_DFE_1) or math.isinf(S_non_DFE_1) or math.isinf(I_W_non_DFE_1)␣
↪or math.isinf(I_Z_non_DFE_1):
       DFE_1_feasable = False
   else:
       DFE_1_feasable = True

   if DFE_1_feasable and round_sf(S_non_DFE_1,2) >= 0 and␣
↪round_sf(I_W_non_DFE_1,2) >= 0 and round_sf(I_Z_non_DFE_1,2) >= 0:
       DFE_1_feasable = True
   else:
       DFE_1_feasable = False

   if math.isnan(S_non_DFE_2) or math.isnan(I_W_non_DFE_2) or math.
↪isnan(I_Z_non_DFE_2) or math.isinf(S_non_DFE_2) or math.isinf(I_W_non_DFE_2)␣
↪or math.isinf(I_Z_non_DFE_2):
       DFE_2_feasable = False
   else:
       DFE_2_feasable = True

   if DFE_2_feasable and round_sf(S_non_DFE_2,2) >= 0 and␣
↪round_sf(I_W_non_DFE_2,2) >= 0 and round_sf(I_Z_non_DFE_2,2) >= 0:
       DFE_2_feasable = True
   else:
       DFE_2_feasable = False

   #Determine if R0_W is >=1, R0_Z<1 and the first DFE is biologically␣
↪feasible.
   if R0_W.subs(param_values)>=1 and R0_Z.subs(param_values)<1 and␣
↪DFE_1_feasable:
       equil_pops.append({'S':S_non_DFE_1,'I_W':I_W_non_DFE_1,'I_Z':
↪I_Z_non_DFE_1})
```

```
   #Determine if R0_Z is >=1, R0_W<1 and the second DFE is biologically␣
↪feasible.
   if R0_Z.subs(param_values)>=1 and R0_W.subs(param_values)<1 and␣
↪DFE_2_feasable:
       equil_pops.append({'S':S_non_DFE_2,'I_W':I_W_non_DFE_2,'I_Z':
↪I_Z_non_DFE_2})

   #Determine if R0_W is >=1, R0_Z>=1 and the first DFE is biologically␣
↪feasible but the second is not.
   if R0_W.subs(param_values)>=1 and R0_Z.subs(param_values)>=1 and␣
↪DFE_1_feasable and DFE_2_feasable == False:
       equil_pops.append({'S':S_non_DFE_1,'I_W':I_W_non_DFE_1,'I_Z':
↪I_Z_non_DFE_1})

   #Determine if R0_W is >=1, R0_Z>=1 and the second DFE is biologically␣
↪feasable but the first is not.
   if R0_W.subs(param_values)>=1 and R0_Z.subs(param_values)>=1 and␣
↪DFE_1_feasable  == False and DFE_2_feasable:
       equil_pops.append({'S':S_non_DFE_2,'I_W':I_W_non_DFE_2,'I_Z':
↪I_Z_non_DFE_2})

   #Determine if R0_W is >=1, R0_Z>=1 and both DFEs are biologically feasable.
   if R0_W.subs(param_values)>=1 and R0_Z.subs(param_values)>=1 and␣
↪DFE_1_feasable and DFE_2_feasable:
       # If all the values for non-DFE_1 are >= 0, evaluate eigen values
       # associated with the Jacobian matrix of non-DFE_1.
       # Need to be able to subsititute in equilibria formula, as well as the␣
↪param values:
       vals_to_subs = copy.deepcopy(param_values)
       vals_to_subs['S'] = S_non_DFE_1
       vals_to_subs['I_W'] = I_W_non_DFE_1
       vals_to_subs['I_Z'] = I_Z_non_DFE_1
       if eigs_J_non_DFE_1[0].subs(vals_to_subs) <=0 and eigs_J_non_DFE_1[1].
↪subs(vals_to_subs)<=0:
           # If the non-vero eigen values associated with the Jacobian matrix
           # of non-DFE_1 are <= 0, the equil_pops is appended with non-DFE_1,
           # as it is locally stable.
           equil_pops.append({'S':S_non_DFE_1,'I_W':I_W_non_DFE_1,'I_Z':
↪I_Z_non_DFE_1})

       # If all the values for non-DFE_2 are >= 0, evaluate eigen values
       # associated with the Jacobian matrix of non-DFE_2.
       # Need to be able to subsititute in equilibria formula, as well as the␣
↪param values:
       vals_to_subs = copy.deepcopy(param_values)
       vals_to_subs['S'] =  S_non_DFE_2
```

```
        vals_to_subs['I_W'] = I_W_non_DFE_2
        vals_to_subs['I_Z'] = I_Z_non_DFE_2
        if eigs_J_non_DFE_2[0].subs(vals_to_subs) <=0 and eigs_J_non_DFE_2[1].
↪subs(vals_to_subs)<=0:
            # If the non-vero eigen values associated with the Jacobian matrix
            # of non-DFE_2 are <= 0, the equil_pops is appended with non-DFE_2,
            # as it is locally stable.
            equil_pops.append({'S':S_non_DFE_2,'I_W':I_W_non_DFE_2,'I_Z':
↪I_Z_non_DFE_2})

    return(equil_pops)
```

2 2 Endemic Equilibria Stability Analyses of the Replacement Infection Model from Spicknall et al (2013)

2.1 2.1 Deriving Equilibria

[13]:
```
Eqn_S = -beta_W/N*I_W*S-beta_Z/
↪N*I_Z*S+gamma*(1-epsilon)*I_W+gamma_T*epsilon*I_W+gamma*I_Z
Eqn_I_W = beta_W/N*I_W*S-gamma*(1-epsilon)*I_W-gamma_T*epsilon*I_W+(beta_W/
↪N-beta_Z/N)*I_W*I_Z
Eqn_I_Z = beta_Z/N*I_Z*S-gamma*I_Z+(beta_Z/N-beta_W/N)*I_W*I_Z
```

[14]:
```
ODEs = sympy.Matrix([Eqn_S,Eqn_I_W,Eqn_I_Z])
ODEs
```

[14]:
$$\begin{bmatrix} I_W \epsilon \gamma_T + I_W \gamma (1-\epsilon) - \frac{I_W S \beta_W}{N} + I_Z \gamma - \frac{I_Z S \beta_Z}{N} \\ I_W I_Z \left(\frac{\beta_W}{N} - \frac{\beta_Z}{N}\right) - I_W \epsilon \gamma_T - I_W \gamma (1-\epsilon) + \frac{I_W S \beta_W}{N} \\ I_W I_Z \left(-\frac{\beta_W}{N} + \frac{\beta_Z}{N}\right) - I_Z \gamma + \frac{I_Z S \beta_Z}{N} \end{bmatrix}$$

[15]:
```
equilibria = sympy.solve(ODEs,[S, I_W, I_Z])
equilibria
```

[15]:
$$\left[(S,\ 0,\ 0),\ \left(\frac{I_W \beta_W - I_W \beta_Z + N\gamma}{\beta_Z},\ I_W,\ \frac{-I_W \beta_W^2 + I_W \beta_W \beta_Z - N \beta_W \gamma - N \beta_Z \epsilon \gamma + N \beta_Z \epsilon \gamma_T + N \beta_Z \gamma}{\beta_Z (\beta_W - \beta_Z)} \right),\ \left(\frac{N(-}{} \right.$$

The 2nd vector is difficult to interprate I_W appearing in all three elements, it requires some substituiton work.

2.1.1 2.1.1 Dealing with Unusual Equilibria

[16]:
```
unusaul_equil = copy.deepcopy(equilibria[1])
unusaul_equil
```

[16]:
$$\left(\frac{I_W \beta_W - I_W \beta_Z + N\gamma}{\beta_Z},\ I_W,\ \frac{-I_W \beta_W^2 + I_W \beta_W \beta_Z - N \beta_W \gamma - N \beta_Z \epsilon \gamma + N \beta_Z \epsilon \gamma_T + N \beta_Z \gamma}{\beta_Z (\beta_W - \beta_Z)} \right)$$

2.1.1.1 Treating as a Simultaneous Equation Problem.

[17]:
```
exp2_as_simul=unusaul_equil[1]-I_W
exp3_as_simul=unusaul_equil[2]-I_Z
sol = sympy.solve((exp2_as_simul,exp3_as_simul),(I_W,I_Z))
sol
```

[17]:
$$\left\{I_W : -\frac{I_Z\beta_Z}{\beta_W} + \frac{-N\beta_W\gamma - N\beta_Z\epsilon\gamma + N\beta_Z\epsilon\gamma_T + N\beta_Z\gamma}{\beta_W^2 - \beta_W\beta_Z}\right\}$$

2.1.1.2 Substituting $S = N-(I_W+I_Z)$ and treating as a Simultaneous Equation Problem.

Lets try substituting $S = N - (I_W + I_Z)$ into the 2nd and 3rd equation:

[18]:
```
S_in_N_terms = N-(I_W+I_Z)
exp2_sub_S_in_N_terms = sympy.simplify(unusaul_equil[1].subs(S,S_in_N_terms))
exp3_sub_S_in_N_terms = sympy.simplify(unusaul_equil[2].subs(S,S_in_N_terms))
display(exp2_sub_S_in_N_terms,exp3_sub_S_in_N_terms)
```

I_W

$$\frac{-I_W\beta_W^2 + I_W\beta_W\beta_Z - N\beta_W\gamma - N\beta_Z\epsilon\gamma + N\beta_Z\epsilon\gamma_T + N\beta_Z\gamma}{\beta_Z\left(\beta_W - \beta_Z\right)}$$

Solving both equstions for I_W and I_Z

[19]:
```
exp2_sub_S_in_N_terms=exp2_sub_S_in_N_terms-I_W
exp3_sub_S_in_N_terms=exp3_sub_S_in_N_terms-I_Z
sol = sympy.solve((exp2_sub_S_in_N_terms,exp3_sub_S_in_N_terms),(I_W,I_Z))
sol
```

[19]:
$$\left\{I_W : -\frac{I_Z\beta_Z}{\beta_W} + \frac{-N\beta_W\gamma - N\beta_Z\epsilon\gamma + N\beta_Z\epsilon\gamma_T + N\beta_Z\gamma}{\beta_W^2 - \beta_W\beta_Z}\right\}$$

There is no solution to this approach!

2.2.1.3 Substituting $N = S+I_W+I_Z$ and treating as a Simultaneous Equation Problem.

Lets try substituting $N = S + I_W + I_Z$ into the 2nd and 3rd equation:

[20]:
```
N_in_other_terms = S+I_W+I_Z
exp2_sub_N_in_other_terms = sympy.simplify(unusaul_equil[1].
 ↪subs(N,N_in_other_terms))
exp3_sub_N_in_other_terms = sympy.simplify(unusaul_equil[2].
 ↪subs(N,N_in_other_terms))
display(exp2_sub_N_in_other_terms,exp3_sub_N_in_other_terms)
```

I_W

$$\frac{-I_W\beta_W^2 + I_W\beta_W\beta_Z - \beta_W\gamma\left(I_W + I_Z + S\right) - \beta_Z\epsilon\gamma\left(I_W + I_Z + S\right) + \beta_Z\epsilon\gamma_T\left(I_W + I_Z + S\right) + \beta_Z\gamma\left(I_W + I_Z + S\right)}{\beta_Z\left(\beta_W - \beta_Z\right)}$$

[21]:
```
exp2_sub_N_in_other_terms=exp2_sub_N_in_other_terms-I_W
exp3_sub_N_in_other_terms=exp3_sub_N_in_other_terms-I_Z
```

```
sol = sympy.
↪solve((exp2_sub_N_in_other_terms,exp3_sub_N_in_other_terms),(I_W,I_Z))
sol
```

[21]:
$$\left\{I_W : \frac{I_Z\left(-\beta_W\beta_Z - \beta_W\gamma + \beta_Z^2 - \beta_Z\epsilon\gamma + \beta_Z\epsilon\gamma_T + \beta_Z\gamma\right)}{\beta_W^2 - \beta_W\beta_Z + \beta_W\gamma + \beta_Z\epsilon\gamma - \beta_Z\epsilon\gamma_T - \beta_Z\gamma} + \frac{-S\beta_W\gamma - S\beta_Z\epsilon\gamma + S\beta_Z\epsilon\gamma_T + S\beta_Z\gamma}{\beta_W^2 - \beta_W\beta_Z + \beta_W\gamma + \beta_Z\epsilon\gamma - \beta_Z\epsilon\gamma_T - \beta_Z\gamma}\right\}$$

Whilst not getting rid of the S term in the equations for I_W and I_Z, this does reveal that this equilibrium point is biologically meaningless. Either $\beta_W < \beta_Z$ and I_W is negative, $\beta_Z < \beta_W$ and I_Z is negative or $\beta_W = \beta_Z$ in which case $I_W = undefined$ and $I_Z = -undefined$.

2.2 2.2 Determining Equilibria Stability

We are not interested in whether the disease free equilibrium (DFE), the first equilibria, is stable. This can be found through deriving R0 using PyGom's matrix methodology for determining R0.

Lets look at the stability of the non-DFEs, following recipe 8.2 of Otto & Day (2007). Needs to create Jacobian Matrix of the models ODEs.

The 2nd equilibrium as discussed in 1.2.1 is biologically meaningless. Therefore, we ignore this equilibrium.

[22]:
```
J_of_ODEs = ODEs.jacobian(X=[S,I_W,I_Z])
J_of_ODEs
```

[22]:
$$\begin{bmatrix} -\frac{I_W\beta_W}{N} - \frac{I_Z\beta_Z}{N} & \epsilon\gamma_T + \gamma\left(1-\epsilon\right) - \frac{S\beta_W}{N} & \gamma - \frac{S\beta_Z}{N} \\ \frac{I_W\beta_W}{N} & I_Z\left(\frac{\beta_W}{N} - \frac{\beta_Z}{N}\right) - \epsilon\gamma_T - \gamma\left(1-\epsilon\right) + \frac{S\beta_W}{N} & I_W\left(\frac{\beta_W}{N} - \frac{\beta_Z}{N}\right) \\ \frac{I_Z\beta_Z}{N} & I_Z\left(-\frac{\beta_W}{N} + \frac{\beta_Z}{N}\right) & I_W\left(-\frac{\beta_W}{N} + \frac{\beta_Z}{N}\right) - \gamma + \frac{S\beta_Z}{N} \end{bmatrix}$$

Substitute the 1st non-disease free equilibrium (non-DFE) into the Jacobian matrix and determine Eigen values.

[23]:
```
non_DFE_1 = {'S':equilibria[2][0],'I_W':equilibria[2][1],'I_Z':equilibria[2][2]}

J_of_ODEs_non_DFE_1 = J_of_ODEs.subs(non_DFE_1)
J_of_ODEs_non_DFE_1

eigs_J_non_DFE_1 = J_of_ODEs_non_DFE_1.eigenvals()
# The above code line enters the Eigen values as keys to a dictionary.
# It would be more useful to have them as elements in a vector.
eigs_J_non_DFE_1 = [k  for  k in  eigs_J_non_DFE_1.keys()]
eigs_J_non_DFE_1
```

[23]:
$$\left[-\frac{I_W\beta_W^2 - I_W\beta_W\beta_Z + N\beta_W\gamma + N\beta_Z\epsilon\gamma - N\beta_Z\epsilon\gamma_T - N\beta_Z\gamma}{N\beta_W},\ -\frac{I_W\beta_W}{N},\ 0\right]$$

Substitute the 2nd non-disease free equilibrium (non-DFE) into the Jacobian matrix and determine Eigen values.

[24]:
```
non_DFE_2 = {'S':equilibria[3][0],'I_W':equilibria[3][1],'I_Z':equilibria[3][2]}

J_of_ODEs_non_DFE_2 = J_of_ODEs.subs(non_DFE_2)
J_of_ODEs_non_DFE_2

eigs_J_non_DFE_2 = J_of_ODEs_non_DFE_2.eigenvals()
# The above code line enters the Eigen values as keys to a dictionary.
# It would be more useful to have them as elements in a vector.
eigs_J_non_DFE_2 = [k  for  k in  eigs_J_non_DFE_2.keys()]
eigs_J_non_DFE_2
```

[24]:
$$\left[\frac{I_Z\beta_W\beta_Z - I_Z\beta_Z^2 + N\beta_W\gamma + N\beta_Z\epsilon\gamma - N\beta_Z\epsilon\gamma_T - N\beta_Z\gamma}{N\beta_Z},\ -\frac{I_Z\beta_Z}{N},\ 0\right]$$

2.3 2.3 Function for Determining the Stable Endemic Equilibrium Infection Prevalence

[26]:
```
def replace_inf_end_equil(param_values):
    '''
    Calculates the non-disease free equilibria for the replacement infection␣
↪model ,
    as out lined in Spicknall et al 2013. Reuturning any non-DFE that are
    that are biologically reasonable and locally stable.
    '''
    # Note if both strains have R0 values <1 there is no point in using this␣
↪function.

    equil_pops = []
    # Setup Equilibrium populations as an empty list. If any of the endemic
    # equilibria are found to be biologically feasible and stable they are␣
↪appended to the list.

    #Basic Reproductive numbers from Spicknall 2013
    R0_W = beta_W/(gamma*(1-epsilon)+gamma_T*epsilon)
    R0_Z = beta_Z/gamma

    #Formula for the non-Disease Free Equilibria has been worked out in a␣
↪jupyter notebook
    #and pasted here. NOTE with the first non-DFE the I_W population = N-S␣
↪(I_Z=0),
    # and with the second non-DFE the I_Z population = N-S (I_W=0).
    # Therefore, only the formule for S are needed (see jupyter notebook).
    S_non_DFE_1 = N*(-epsilon*gamma + epsilon*gamma_T + gamma)/beta_W
    S_non_DFE_1 = S_non_DFE_1.subs(param_values)
    I_W_non_DFE_1 = param_values['N']-S_non_DFE_1
    I_Z_non_DFE_1 = 0
    S_non_DFE_2 = N*gamma/beta_Z
```

```
    S_non_DFE_2 = S_non_DFE_2.subs(param_values)
    I_W_non_DFE_2 = 0
    I_Z_non_DFE_2 = param_values['N']-S_non_DFE_2

    #The eigen values associated with Jacobian matricies for the non-DFE have␣
↪been
    #worked out in a jupyter notebook and pasted here.
    eigs_J_non_DFE_1 = [-I_W*beta_W/N,
                        -(I_W*beta_W**2 - I_W*beta_W*beta_Z + N*beta_W*gamma +␣
↪N*beta_Z*epsilon*gamma - N*beta_Z*epsilon*gamma_T - N*beta_Z*gamma)/
↪(N*beta_W),
                        0]
    eigs_J_non_DFE_2 = [-I_Z*beta_Z/N,
                        (I_Z*beta_W*beta_Z - I_Z*beta_Z**2 + N*beta_W*gamma +␣
↪N*beta_Z*epsilon*gamma - N*beta_Z*epsilon*gamma_T - N*beta_Z*gamma)/
↪(N*beta_Z),
                        0]

    if math.isnan(S_non_DFE_1) or math.isnan(I_W_non_DFE_1) or math.
↪isnan(I_Z_non_DFE_1) or math.isinf(S_non_DFE_1) or math.isinf(I_W_non_DFE_1)␣
↪or math.isinf(I_Z_non_DFE_1):
        DFE_1_feasable = False
    else:
        DFE_1_feasable = True

    if DFE_1_feasable and round_sf(S_non_DFE_1,2) >= 0 and␣
↪round_sf(I_W_non_DFE_1,2) >= 0 and round_sf(I_Z_non_DFE_1,2) >= 0:
        DFE_1_feasable = True
    else:
        DFE_1_feasable = False

    if math.isnan(S_non_DFE_2) or math.isnan(I_W_non_DFE_2) or math.
↪isnan(I_Z_non_DFE_2) or math.isinf(S_non_DFE_2) or math.isinf(I_W_non_DFE_2)␣
↪or math.isinf(I_Z_non_DFE_2):
        DFE_2_feasable = False
    else:
        DFE_2_feasable = True

    if DFE_2_feasable and round_sf(S_non_DFE_2,2) >= 0 and␣
↪round_sf(I_W_non_DFE_2,2) >= 0 and round_sf(I_Z_non_DFE_2,2) >= 0:
        DFE_2_feasable = True
    else:
        DFE_2_feasable = False

    #Determine if R0_W is >=1, R0_Z<1 and the first DFE is biologically␣
↪feasable.
```

```
   if R0_W.subs(param_values)>=1 and R0_Z.subs(param_values)<1 and␣
↪DFE_1_feasable:
       equil_pops.append({'S':S_non_DFE_1,'I_W':I_W_non_DFE_1,'I_Z':
↪I_Z_non_DFE_1})

   #Determine if R0_Z is >=1, R0_W<1 and the second DFE is biologically␣
↪feasable.
   if R0_Z.subs(param_values)>=1 and R0_W.subs(param_values)<1 and␣
↪DFE_2_feasable:
       equil_pops.append({'S':S_non_DFE_2,'I_W':I_W_non_DFE_2,'I_Z':
↪I_Z_non_DFE_2})

   #Determine if R0_W is >=1, R0_Z>=1 and the first DFE is biologically␣
↪feasable but the second is not.
   if R0_W.subs(param_values)>=1 and R0_Z.subs(param_values)>=1 and␣
↪DFE_1_feasable and DFE_2_feasable == False:
       equil_pops.append({'S':S_non_DFE_1,'I_W':I_W_non_DFE_1,'I_Z':
↪I_Z_non_DFE_1})

   #Determine if R0_W is >=1, R0_Z>=1 and the second DFE is biologically␣
↪feasable but the first is not.
   if R0_W.subs(param_values)>=1 and R0_Z.subs(param_values)>=1 and␣
↪DFE_1_feasable  == False and DFE_2_feasable:
       equil_pops.append({'S':S_non_DFE_2,'I_W':I_W_non_DFE_2,'I_Z':
↪I_Z_non_DFE_2})

   #Determine if R0_W is >=1, R0_Z>=1 and both DFEs are biologically feasable.
   if R0_W.subs(param_values)>=1 and R0_Z.subs(param_values)>=1 and␣
↪DFE_1_feasable and DFE_2_feasable:
       # If all the values for non-DFE_1 are >= 0, evaluate eigen values
       # associated with the Jacobian matrix of non-DFE_1.
       # Need to be able to subsititute in equilibria formula, as well as the␣
↪param values:
       vals_to_subs = copy.deepcopy(param_values)
       vals_to_subs['S'] = S_non_DFE_1
       vals_to_subs['I_W'] = I_W_non_DFE_1
       vals_to_subs['I_Z'] = I_Z_non_DFE_1
       if eigs_J_non_DFE_1[0].subs(vals_to_subs) <=0 and eigs_J_non_DFE_1[1].
↪subs(vals_to_subs)<=0:
           # If the non-vero eigen values associated with the Jacobian matrix
           # of non-DFE_1 are <= 0, the equil_pops is appended with non-DFE_1,
           # as it is locally stable.
           equil_pops.append({'S':S_non_DFE_1,'I_W':I_W_non_DFE_1,'I_Z':
↪I_Z_non_DFE_1})

       # If all the values for non-DFE_2 are >= 0, evaluate eigenvalues
```

```
        # associated with the Jacobian matrix of non-DFE_2.
        # Need to be able to subsititute in equilibria formula, as well as the␣
↪param values:
        vals_to_subs = copy.deepcopy(param_values)
        vals_to_subs['S'] =  S_non_DFE_2
        vals_to_subs['I_W'] = I_W_non_DFE_2
        vals_to_subs['I_Z'] = I_Z_non_DFE_2
        if eigs_J_non_DFE_2[0].subs(vals_to_subs) <=0 and eigs_J_non_DFE_2[1].
↪subs(vals_to_subs)<=0:
            # If the non-vero eigen values associated with the Jacobian matrix
            # of non-DFE_2 are <= 0, the equil_pops is appended with non-DFE_2,
            # as it is locally stable.
            equil_pops.append({'S':S_non_DFE_2,'I_W':I_W_non_DFE_2,'I_Z':
↪I_Z_non_DFE_2})

    return(equil_pops)
```

3 3 Endemic Equilibria Stability Analyses of the Bi-Conversion Model from Spicknall et al (2013)

3.1 3.1 Deriving Equilibria & The Eigen Values Associated with Their Jacobian Matricies

[27]:
```
Eqn_S = -beta_W/N*I_W*S-beta_Z/
 ↪N*I_Z*S+gamma*(1-epsilon)*I_W+gamma_T*epsilon*I_W+gamma*I_Z
Eqn_I_W = beta_W/
 ↪N*I_W*S-gamma*(1-epsilon)*I_W-gamma_T*epsilon*I_W-rho*epsilon*I_W+phi*(1-epsilon)*I_Z
Eqn_I_Z = beta_Z/N*I_Z*S-gamma*I_Z+rho*epsilon*I_W-phi*(1-epsilon)*I_Z
```

[28]:
```
ODEs = sympy.Matrix([Eqn_S,Eqn_I_W,Eqn_I_Z])
ODEs
```

[28]:
$$\begin{bmatrix} I_W\epsilon\gamma_T + I_W\gamma\left(1-\epsilon\right) - \frac{I_W S\beta_W}{N} + I_Z\gamma - \frac{I_Z S\beta_Z}{N} \\ -I_W\epsilon\gamma_T - I_W\epsilon\rho - I_W\gamma\left(1-\epsilon\right) + \frac{I_W S\beta_W}{N} + I_Z\phi\left(1-\epsilon\right) \\ I_W\epsilon\rho - I_Z\gamma - I_Z\phi\left(1-\epsilon\right) + \frac{I_Z S\beta_Z}{N} \end{bmatrix}$$

[29]:
```
equilibria = sympy.solve(ODEs,[S, I_W, I_Z])
```

[30]:
```
display(equilibria[0],equilibria[1],equilibria[2])
```

$$\left(S,\ 0,\ 0\right)$$

$$\left(\frac{N\left(-\beta_W\epsilon\phi + \beta_W\gamma + \beta_W\phi - \beta_Z\epsilon\gamma + \beta_Z\epsilon\gamma_T + \beta_Z\epsilon\rho + \beta_Z\gamma - \sqrt{\beta_W^2\epsilon^2\phi^2 - 2\beta_W^2\epsilon\gamma\phi - 2\beta_W^2\epsilon\phi^2 + \beta_W^2\gamma^2 + 2\beta_W^2\gamma\phi -\right.}\right.$$

$$\left(\frac{N\left(-\beta_W\epsilon\phi + \beta_W\gamma + \beta_W\phi - \beta_Z\epsilon\gamma + \beta_Z\epsilon\gamma_T + \beta_Z\epsilon\rho + \beta_Z\gamma + \sqrt{\beta_W^2\epsilon^2\phi^2 - 2\beta_W^2\epsilon\gamma\phi - 2\beta_W^2\epsilon\phi^2 + \beta_W^2\gamma^2 + 2\beta_W^2\gamma\phi -\right.\right.$$

The disease free equilibrium (DFE) is the first vector.

[31]:
```
DFE = equilibria[0]
```

We are not interested in whether the disease free equilibrium (DFE), the first equilibria, is stable. This can be found through deriving R0 using PyGom's matrix methodology for determining R0.

Lets look at the stability of the non-DFEs, following recipe 8.2 of Otto & Day (2007). Needs to create Jacobian Matrix of the models ODEs.

[32]:
```
J_of_ODEs = ODEs.jacobian(X=[S,I_W,I_Z])
J_of_ODEs
```

[32]:
$$\begin{bmatrix} -\frac{I_W\beta_W}{N} - \frac{I_Z\beta_Z}{N} & \epsilon\gamma_T + \gamma\left(1-\epsilon\right) - \frac{S\beta_W}{N} & \gamma - \frac{S\beta_Z}{N} \\ \frac{I_W\beta_W}{N} & -\epsilon\gamma_T - \epsilon\rho - \gamma\left(1-\epsilon\right) + \frac{S\beta_W}{N} & \phi\left(1-\epsilon\right) \\ \frac{I_Z\beta_Z}{N} & \epsilon\rho & -\gamma - \phi\left(1-\epsilon\right) + \frac{S\beta_Z}{N} \end{bmatrix}$$

Substitute the 1st non-disease free equilibrium (non-DFE) into the Jacobian matrix and determine Eigen values.

[33]:
```
non_DFE_1 = {'S':equilibria[1][0],'I_W':equilibria[1][1],'I_Z':equilibria[1][2]}

J_of_ODEs_non_DFE_1 = J_of_ODEs.subs(non_DFE_1)
J_of_ODEs_non_DFE_1

eigs_J_non_DFE_1 = J_of_ODEs_non_DFE_1.eigenvals()
# The above code line enters the Eigen values as keys to a dictionary.
# It would be more useful to have them as elements in a vector.
eigs_J_non_DFE_1 = [k  for  k in  eigs_J_non_DFE_1.keys()]
```

[34]:
```
sympy.simplify(eigs_J_non_DFE_1[0])
```

```
C:\Users\mdgru\anaconda3\envs\amr\lib\site-
packages\IPython\lib\latextools.py:126: MatplotlibDeprecationWarning:
The to_png function was deprecated in Matplotlib 3.4 and will be removed two
minor releases later. Use mathtext.math_to_image instead.
  mt.to_png(f, s, fontsize=12, dpi=dpi, color=color)
C:\Users\mdgru\anaconda3\envs\amr\lib\site-
packages\IPython\lib\latextools.py:126: MatplotlibDeprecationWarning:
The to_rgba function was deprecated in Matplotlib 3.4 and will be removed two
minor releases later. Use mathtext.math_to_image instead.
  mt.to_png(f, s, fontsize=12, dpi=dpi, color=color)
C:\Users\mdgru\anaconda3\envs\amr\lib\site-
packages\IPython\lib\latextools.py:126: MatplotlibDeprecationWarning:
The to_mask function was deprecated in Matplotlib 3.4 and will be removed two
minor releases later. Use mathtext.math_to_image instead.
```

```
  mt.to_png(f, s, fontsize=12, dpi=dpi, color=color)
C:\Users\mdgru\anaconda3\envs\amr\lib\site-
packages\IPython\lib\latextools.py:126: MatplotlibDeprecationWarning:
The MathtextBackendBitmap class was deprecated in Matplotlib 3.4 and will be
removed two minor releases later. Use mathtext.math_to_image instead.
  mt.to_png(f, s, fontsize=12, dpi=dpi, color=color)
```

[34]:

$$\frac{-I_Z\beta_W\beta_Z^2\epsilon\rho + I_Z\beta_W\beta_Z\epsilon\left(\beta_W\phi - \beta_Z\gamma + \beta_Z\gamma_T\right) + I_Z\beta_W\beta_Z\left(-\beta_W\gamma - \beta_W\phi + \beta_Z\gamma - \sqrt{\beta_W^2\epsilon^2\phi^2 - 2\beta_W^2\epsilon\gamma\phi - 2\beta_W^2\epsilon\phi}\right.}{}$$

[35]:
```
sympy.simplify(eigs_J_non_DFE_1[1])
```

[35]:

$$\frac{-I_Z\beta_W\beta_Z^2\epsilon\rho + I_Z\beta_W\beta_Z\epsilon\left(\beta_W\phi - \beta_Z\gamma + \beta_Z\gamma_T\right) + I_Z\beta_W\beta_Z\left(-\beta_W\gamma - \beta_W\phi + \beta_Z\gamma - \sqrt{\beta_W^2\epsilon^2\phi^2 - 2\beta_W^2\epsilon\gamma\phi - 2\beta_W^2\epsilon\phi}\right.}{}$$

[36]:
```
sympy.simplify(eigs_J_non_DFE_1[2])
```

[36]:

0

Substitute the 2nd non-disease free equilibrium (non-DFE) into the Jacobian matrix and determine Eigen values.

[37]:
```
non_DFE_2 = {'S':equilibria[2][0],'I_W':equilibria[2][1],'I_Z':equilibria[2][2]}

J_of_ODEs_non_DFE_2 = J_of_ODEs.subs(non_DFE_2)
J_of_ODEs_non_DFE_2

eigs_J_non_DFE_2 = J_of_ODEs_non_DFE_2.eigenvals()
# The above code line enters the Eigen values as keys to a dictionary.
# It would be more useful to have them as elements in a vector.
eigs_J_non_DFE_2 = [k  for  k in  eigs_J_non_DFE_2.keys()]
```

[38]:
```
sympy.simplify(eigs_J_non_DFE_2[0])
```

[38]:

$$\frac{-I_Z\beta_W\beta_Z^2\epsilon\rho + I_Z\beta_W\beta_Z\epsilon\left(\beta_W\phi - \beta_Z\gamma + \beta_Z\gamma_T\right) + I_Z\beta_W\beta_Z\left(-\beta_W\gamma - \beta_W\phi + \beta_Z\gamma + \sqrt{\beta_W^2\epsilon^2\phi^2 - 2\beta_W^2\epsilon\gamma\phi - 2\beta_W^2\epsilon\phi}\right.}{}$$

[39]:
```
sympy.simplify(eigs_J_non_DFE_2[1])
```

[39]:

$$\frac{-I_Z\beta_W\beta_Z^2\epsilon\rho + I_Z\beta_W\beta_Z\epsilon\left(\beta_W\phi - \beta_Z\gamma + \beta_Z\gamma_T\right) + I_Z\beta_W\beta_Z\left(-\beta_W\gamma - \beta_W\phi + \beta_Z\gamma + \sqrt{\beta_W^2\epsilon^2\phi^2 - 2\beta_W^2\epsilon\gamma\phi - 2\beta_W^2\epsilon\phi}\right.}{}$$

[40]:
```
sympy.simplify(eigs_J_non_DFE_2[2])
```

[40]:

0

3.2 3.2 Function for Determining the Stable Endemic Equilibrium Infection Prevelance

[42]:
```
def bi_directional_end_equil(param_values):
    '''
    Calculates the non-disease free equilibria for the uni and bi-directional␣
↪convertion model ,
    as out lined in Spicknall et al 2013. Returning any non-DFE that are
    biologically reasonable and locally stable.
    '''
    # Note if both strains have R0 values <1 there is no point in using this␣
↪function.

    equil_pops = []
    # Setup Equilibrium populations as an empty list. If any of the endemic
    # equilibria are found to be biologically feasible they are appended to the␣
↪list.

    #Basic Reproductive numbers from Spicknall 2013
    R0_W = 2*beta_W*beta_Z/
↪(beta_W*gamma+beta_Z*gamma-beta_Z*epsilon*gamma+beta_Z*epsilon*gamma_T+beta_W*phi-beta_W*ep

                          ␣
↪((beta_W*(gamma+phi-epsilon*phi)+beta_Z*(gamma-epsilon*gamma+epsilon*(gamma_T+rho)))**2+4*b

                                                                              ␣
↪                                                  ␣
↪(-1+epsilon)*epsilon*gamma_T*phi-gamma*(phi+phi*epsilon**2+epsilon*(gamma_T-2*phi+rho))))**
↪5)
    R0_Z = 2*beta_W*beta_Z/
↪(beta_W*gamma+beta_Z*gamma-beta_Z*epsilon*gamma+beta_Z*epsilon*gamma_T+beta_W*phi-beta_W*ep

                          ␣
↪((beta_W*(gamma+phi-epsilon*phi)+beta_Z*(gamma-epsilon*gamma+epsilon*(gamma_T+rho)))**2+4*b

                                                                              ␣
↪                                                  ␣
↪(-1+epsilon)*epsilon*gamma_T*phi-gamma*(phi+phi*epsilon**2+epsilon*(gamma_T-2*phi+rho))))**
↪5)

    #Formula for the non-Disease Free Equilibria has been worked out in a␣
↪jupyter notebook
    #and pasted here.
    non_DFE_1 = {
```

```
            'S':N*(-beta_W*epsilon*phi + beta_W*gamma + beta_W*phi -␣
↪beta_Z*epsilon*gamma + beta_Z*epsilon*gamma_T + beta_Z*epsilon*rho +␣
↪beta_Z*gamma - sqrt(beta_W**2*epsilon**2*phi**2 -␣
↪2*beta_W**2*epsilon*gamma*phi - 2*beta_W**2*epsilon*phi**2 +␣
↪beta_W**2*gamma**2 + 2*beta_W**2*gamma*phi + beta_W**2*phi**2 -␣
↪2*beta_W*beta_Z*epsilon**2*gamma*phi +␣
↪2*beta_W*beta_Z*epsilon**2*gamma_T*phi - 2*beta_W*beta_Z*epsilon**2*phi*rho␣
↪+ 2*beta_W*beta_Z*epsilon*gamma**2 - 2*beta_W*beta_Z*epsilon*gamma*gamma_T +␣
↪4*beta_W*beta_Z*epsilon*gamma*phi - 2*beta_W*beta_Z*epsilon*gamma*rho -␣
↪2*beta_W*beta_Z*epsilon*gamma_T*phi + 2*beta_W*beta_Z*epsilon*phi*rho -␣
↪2*beta_W*beta_Z*gamma**2 - 2*beta_W*beta_Z*gamma*phi +␣
↪beta_Z**2*epsilon**2*gamma**2 - 2*beta_Z**2*epsilon**2*gamma*gamma_T -␣
↪2*beta_Z**2*epsilon**2*gamma*rho + beta_Z**2*epsilon**2*gamma_T**2 +␣
↪2*beta_Z**2*epsilon**2*gamma_T*rho + beta_Z**2*epsilon**2*rho**2 -␣
↪2*beta_Z**2*epsilon*gamma**2 + 2*beta_Z**2*epsilon*gamma*gamma_T +␣
↪2*beta_Z**2*epsilon*gamma*rho + beta_Z**2*gamma**2))/(2*beta_W*beta_Z)

            ,
            'I_W':-I_Z*(beta_W*epsilon*phi - beta_W*gamma - beta_W*phi -␣
↪beta_Z*epsilon*gamma + beta_Z*epsilon*gamma_T + beta_Z*epsilon*rho +␣
↪beta_Z*gamma - sqrt(beta_W**2*epsilon**2*phi**2 -␣
↪2*beta_W**2*epsilon*gamma*phi - 2*beta_W**2*epsilon*phi**2 +␣
↪beta_W**2*gamma**2 + 2*beta_W**2*gamma*phi + beta_W**2*phi**2 -␣
↪2*beta_W*beta_Z*epsilon**2*gamma*phi +␣
↪2*beta_W*beta_Z*epsilon**2*gamma_T*phi - 2*beta_W*beta_Z*epsilon**2*phi*rho␣
↪+ 2*beta_W*beta_Z*epsilon*gamma**2 - 2*beta_W*beta_Z*epsilon*gamma*gamma_T +␣
↪4*beta_W*beta_Z*epsilon*gamma*phi - 2*beta_W*beta_Z*epsilon*gamma*rho -␣
↪2*beta_W*beta_Z*epsilon*gamma_T*phi + 2*beta_W*beta_Z*epsilon*phi*rho -␣
↪2*beta_W*beta_Z*gamma**2 - 2*beta_W*beta_Z*gamma*phi +␣
↪beta_Z**2*epsilon**2*gamma**2 - 2*beta_Z**2*epsilon**2*gamma*gamma_T -␣
↪2*beta_Z**2*epsilon**2*gamma*rho + beta_Z**2*epsilon**2*gamma_T**2 +␣
↪2*beta_Z**2*epsilon**2*gamma_T*rho + beta_Z**2*epsilon**2*rho**2 -␣
↪2*beta_Z**2*epsilon*gamma**2 + 2*beta_Z**2*epsilon*gamma*gamma_T +␣
↪2*beta_Z**2*epsilon*gamma*rho + beta_Z**2*gamma**2))/(2*beta_W*epsilon*rho)

            ,
            'I_Z':I_Z
            }

    # Need to track proportion so a value for I_Z=1 needs to substituted along␣
↪with params values.
    vals_to_subs = copy.deepcopy(param_values)
    vals_to_subs['I_Z'] = 1
    S_non_DFE_1 = non_DFE_1['S'].subs(param_values)
    not_S_non_DFE_1 = param_values['N']-S_non_DFE_1
    prop_I_W_non_DFE_1 = non_DFE_1['I_W'].subs(vals_to_subs)/
↪(1+non_DFE_1['I_W'].subs(vals_to_subs))
    if math.isnan(prop_I_W_non_DFE_1):
        prop_I_W_non_DFE_1 =1
```

```
   I_W_non_DFE_1 = (not_S_non_DFE_1*prop_I_W_non_DFE_1)
   I_Z_non_DFE_1 = (not_S_non_DFE_1*(1-prop_I_W_non_DFE_1))

   non_DFE_2 = {
           'S':N*(-beta_W*epsilon*phi + beta_W*gamma + beta_W*phi -␣
↪beta_Z*epsilon*gamma + beta_Z*epsilon*gamma_T + beta_Z*epsilon*rho +␣
↪beta_Z*gamma + sqrt(beta_W**2*epsilon**2*phi**2 -␣
↪2*beta_W**2*epsilon*gamma*phi - 2*beta_W**2*epsilon*phi**2 +␣
↪beta_W**2*gamma**2 + 2*beta_W**2*gamma*phi + beta_W**2*phi**2 -␣
↪2*beta_W*beta_Z*epsilon**2*gamma*phi +␣
↪2*beta_W*beta_Z*epsilon**2*gamma_T*phi - 2*beta_W*beta_Z*epsilon**2*phi*rho␣
↪+ 2*beta_W*beta_Z*epsilon*gamma**2 - 2*beta_W*beta_Z*epsilon*gamma*gamma_T +␣
↪4*beta_W*beta_Z*epsilon*gamma*phi - 2*beta_W*beta_Z*epsilon*gamma*rho -␣
↪2*beta_W*beta_Z*epsilon*gamma_T*phi + 2*beta_W*beta_Z*epsilon*phi*rho -␣
↪2*beta_W*beta_Z*gamma**2 - 2*beta_W*beta_Z*gamma*phi +␣
↪beta_Z**2*epsilon**2*gamma**2 - 2*beta_Z**2*epsilon**2*gamma*gamma_T -␣
↪2*beta_Z**2*epsilon**2*gamma*rho + beta_Z**2*epsilon**2*gamma_T**2 +␣
↪2*beta_Z**2*epsilon**2*gamma_T*rho + beta_Z**2*epsilon**2*rho**2 -␣
↪2*beta_Z**2*epsilon*gamma**2 + 2*beta_Z**2*epsilon*gamma*gamma_T +␣
↪2*beta_Z**2*epsilon*gamma*rho + beta_Z**2*gamma**2))/(2*beta_W*beta_Z)
           ,
           'I_W':-I_Z*(beta_W*epsilon*phi - beta_W*gamma - beta_W*phi -␣
↪beta_Z*epsilon*gamma + beta_Z*epsilon*gamma_T + beta_Z*epsilon*rho +␣
↪beta_Z*gamma + sqrt(beta_W**2*epsilon**2*phi**2 -␣
↪2*beta_W**2*epsilon*gamma*phi - 2*beta_W**2*epsilon*phi**2 +␣
↪beta_W**2*gamma**2 + 2*beta_W**2*gamma*phi + beta_W**2*phi**2 -␣
↪2*beta_W*beta_Z*epsilon**2*gamma*phi +␣
↪2*beta_W*beta_Z*epsilon**2*gamma_T*phi - 2*beta_W*beta_Z*epsilon**2*phi*rho␣
↪+ 2*beta_W*beta_Z*epsilon*gamma**2 - 2*beta_W*beta_Z*epsilon*gamma*gamma_T +␣
↪4*beta_W*beta_Z*epsilon*gamma*phi - 2*beta_W*beta_Z*epsilon*gamma*rho -␣
↪2*beta_W*beta_Z*epsilon*gamma_T*phi + 2*beta_W*beta_Z*epsilon*phi*rho -␣
↪2*beta_W*beta_Z*gamma**2 - 2*beta_W*beta_Z*gamma*phi +␣
↪beta_Z**2*epsilon**2*gamma**2 - 2*beta_Z**2*epsilon**2*gamma*gamma_T -␣
↪2*beta_Z**2*epsilon**2*gamma*rho + beta_Z**2*epsilon**2*gamma_T**2 +␣
↪2*beta_Z**2*epsilon**2*gamma_T*rho + beta_Z**2*epsilon**2*rho**2 -␣
↪2*beta_Z**2*epsilon*gamma**2 + 2*beta_Z**2*epsilon*gamma*gamma_T +␣
↪2*beta_Z**2*epsilon*gamma*rho + beta_Z**2*gamma**2))/(2*beta_W*epsilon*rho)
           ,
           'I_Z':I_Z
           }

   S_non_DFE_2 = non_DFE_2['S'].subs(param_values)
   not_S_non_DFE_2 = param_values['N']-S_non_DFE_2
   prop_I_W_non_DFE_2 = non_DFE_2['I_W'].subs(vals_to_subs)/
↪(1+non_DFE_2['I_W'].subs(vals_to_subs))
   if math.isnan(prop_I_W_non_DFE_2):
       prop_I_W_non_DFE_2 =1
```

```
    I_W_non_DFE_2 = (not_S_non_DFE_2*prop_I_W_non_DFE_2)
    I_Z_non_DFE_2 = (not_S_non_DFE_2*(1-prop_I_W_non_DFE_2))


    #The Eigen values associated with Jacobian matricies for the non-DFE have␣
↪been
    #derived in a jupyter notebook and pasted here.
```

```
0]
```

```
                        0]


    if math.isnan(S_non_DFE_1) or math.isnan(I_W_non_DFE_1) or math.
↪isnan(I_Z_non_DFE_1) or math.isinf(S_non_DFE_1) or math.isinf(I_W_non_DFE_1)␣
↪or math.isinf(I_Z_non_DFE_1):
        DFE_1_feasable = False
    else:
        DFE_1_feasable = True

    if DFE_1_feasable and round_sf(S_non_DFE_1,2) >= 0 and␣
↪round_sf(I_W_non_DFE_1,2) >= 0 and round_sf(I_Z_non_DFE_1,2) >= 0:
        DFE_1_feasable = True
    else:
        DFE_1_feasable = False

    if math.isnan(S_non_DFE_2) or math.isnan(I_W_non_DFE_2) or math.
↪isnan(I_Z_non_DFE_2) or math.isinf(S_non_DFE_2) or math.isinf(I_W_non_DFE_2)␣
↪or math.isinf(I_Z_non_DFE_2):
        DFE_2_feasable = False
    else:
        DFE_2_feasable = True

    if DFE_2_feasable and round_sf(S_non_DFE_2,2) >= 0 and␣
↪round_sf(I_W_non_DFE_2,2) >= 0 and round_sf(I_Z_non_DFE_2,2) >= 0:
        DFE_2_feasable = True
    else:
        DFE_2_feasable = False

    #Determine if R0_W is >=1, R0_Z<1 and the first DFE is biologically␣
↪feasable.
    if R0_W.subs(param_values)>=1 and R0_Z.subs(param_values)<1 and␣
↪DFE_1_feasable:
        equil_pops.append({'S':S_non_DFE_1,'I_W':I_W_non_DFE_1,'I_Z':
↪I_Z_non_DFE_1})

    #Determine if R0_Z is >=1, R0_W<1 and the second DFE is biologically␣
↪feasable.
    if R0_Z.subs(param_values)>=1 and R0_W.subs(param_values)<1 and␣
↪DFE_2_feasable:
        equil_pops.append({'S':S_non_DFE_2,'I_W':I_W_non_DFE_2,'I_Z':
↪I_Z_non_DFE_2})

    #Determine if R0_W is >=1, R0_Z>=1 and the first DFE is biologically␣
↪feasable but the second is not.
```

```
   if R0_W.subs(param_values)>=1 and R0_Z.subs(param_values)>=1 and␣
↪DFE_1_feasable and DFE_2_feasable == False:
       equil_pops.append({'S':S_non_DFE_1,'I_W':I_W_non_DFE_1,'I_Z':
↪I_Z_non_DFE_1})

   #Determine if R0_W is >=1, R0_Z>=1 and the second DFE is biologically␣
↪feasable but the first is not.
   if R0_W.subs(param_values)>=1 and R0_Z.subs(param_values)>=1 and␣
↪DFE_1_feasable  == False and DFE_2_feasable:
       equil_pops.append({'S':S_non_DFE_2,'I_W':I_W_non_DFE_2,'I_Z':
↪I_Z_non_DFE_2})

   #Determine if R0_W is >=1, R0_Z>=1 and both DFEs are biologically feasable.
   if R0_W.subs(param_values)>=1 and R0_Z.subs(param_values)>=1 and␣
↪DFE_1_feasable and DFE_2_feasable:
       # There are several denominators in eigen values associated with both␣
↪DFEs (see jupyter notebook) that can be 0 if rho or epsilon = 0.
       # This means sympy's substitution function can produce undefined values␣
↪(NaN values).
       # Therefore as Spicknall et al (2013) found that the largest of the R0␣
↪values was the R0 that described the system.
       # We can choose between the DFE's by using R0s.
       if param_values['epsilon']==0 or param_values['rho'] == 0:
           if R0_W.subs(param_values)> R0_Z.subs(param_values):
               equil_pops.append({'S':S_non_DFE_1,'I_W':I_W_non_DFE_1,'I_Z':
↪I_Z_non_DFE_1})
           else:
               equil_pops.append({'S':S_non_DFE_2,'I_W':I_W_non_DFE_2,'I_Z':
↪I_Z_non_DFE_2})
       else:
           # Need to be able to subsititute in equilibria formula, as well as␣
↪the param values:
           vals_to_subs = copy.deepcopy(param_values)
           vals_to_subs['S'] = S_non_DFE_1
           vals_to_subs['I_W'] = I_W_non_DFE_1
           vals_to_subs['I_Z'] = I_Z_non_DFE_1
           # Both DFE associated eigen values can be complex and we are only
           # interested in the real part so:
           eig_1_J_non_DFE_1 = sympy.re(eigs_J_non_DFE_1[0].subs(vals_to_subs))
           eig_2_J_non_DFE_1 = sympy.re(eigs_J_non_DFE_1[1].subs(vals_to_subs))
           if eig_1_J_non_DFE_1 <=0 and eig_2_J_non_DFE_1 <=0:
               # If the non-vero eigen values associated with the Jacobian␣
↪matrix
               # of non-DFE_1 are <= 0, the equil_pops is appended with␣
↪non-DFE_1,
               # as it is locally stable.
```

```
                equil_pops.append({'S':S_non_DFE_1,'I_W':I_W_non_DFE_1,'I_Z':
↪I_Z_non_DFE_1})

            # If all the values for non-DFE_2 are >= 0, evaluate eigen values
            # associated with the Jacobian matrix of non-DFE_2.
            # Need to be able to subsititute in equilibria formula, as well as␣
↪the param values:
            vals_to_subs = copy.deepcopy(param_values)
            vals_to_subs['S'] =  S_non_DFE_2
            vals_to_subs['I_W'] = I_W_non_DFE_2
            vals_to_subs['I_Z'] = I_Z_non_DFE_2
            # Both DFE associated eigen values can be complex and we are only
            # interested in the real part so:
            eig_1_J_non_DFE_2 = sympy.re(eigs_J_non_DFE_2[0].subs(vals_to_subs))
            eig_2_J_non_DFE_2 = sympy.re(eigs_J_non_DFE_2[1].subs(vals_to_subs))
            if eig_1_J_non_DFE_2 <=0 and eig_2_J_non_DFE_2 <=0:
                # If the non-vero eigen values associated with the Jacobian␣
↪matrix
                # of non-DFE_2 are <= 0, the equil_pops is appended with␣
↪non-DFE_2,
                # as it is locally stable.
                equil_pops.append({'S':S_non_DFE_2,'I_W':I_W_non_DFE_2,'I_Z':
↪I_Z_non_DFE_2})

    return(equil_pops)
```

4 4 Endemic Equilibria Stability Analyses of the Super Infection Model from Spicknall et al (2013)

4.1 4.1 Deriving Equilibria

[43]:
```
Eqn_S = - beta_W/N*(I_W+q*I_WZ)*S - beta_Z/N*(I_Z+q*I_WZ)*S +␣
↪gamma*(1-epsilon)*I_W + gamma_T*epsilon*I_W + gamma*I_Z
Eqn_I_W = beta_W/N*S*(I_W+q*I_WZ) - beta_Z/N*I_W*(I_Z+q*I_WZ) -␣
↪gamma*(1-epsilon)*I_W - gamma_T*epsilon*I_W + gamma*I_WZ
Eqn_I_Z = beta_Z/N*S*(I_Z+q*I_WZ) - beta_W/N*I_Z*(I_W+q*I_WZ) - gamma*I_Z +␣
↪(1-epsilon)*gamma*I_WZ + epsilon*gamma_T*I_WZ
Eqn_I_WZ = beta_W/N*I_Z*(I_W+q*I_WZ) + beta_Z/N*I_W*(I_Z+q*I_WZ) -␣
↪(1-epsilon)*gamma*I_WZ - epsilon*gamma_T*I_WZ - gamma*I_WZ
```

[44]:
```
ODE_mat = sympy.Matrix([Eqn_S,Eqn_I_W,Eqn_I_Z,Eqn_I_WZ])
ODE_mat
```

[44]:

$$\begin{bmatrix} I_W\epsilon\gamma_T + I_W\gamma\left(1-\epsilon\right) + I_Z\gamma - \frac{S\beta_W\left(I_W+I_{WZ}q\right)}{N} - \frac{S\beta_Z\left(I_{WZ}q+I_Z\right)}{N} \\ -I_W\epsilon\gamma_T - I_W\gamma\left(1-\epsilon\right) - \frac{I_W\beta_Z\left(I_{WZ}q+I_Z\right)}{N} + I_{WZ}\gamma + \frac{S\beta_W\left(I_W+I_{WZ}q\right)}{N} \\ I_{WZ}\epsilon\gamma_T + I_{WZ}\gamma\left(1-\epsilon\right) - I_Z\gamma - \frac{I_Z\beta_W\left(I_W+I_{WZ}q\right)}{N} + \frac{S\beta_Z\left(I_{WZ}q+I_Z\right)}{N} \\ \frac{I_W\beta_Z\left(I_{WZ}q+I_Z\right)}{N} - I_{WZ}\epsilon\gamma_T - I_{WZ}\gamma\left(1-\epsilon\right) - I_{WZ}\gamma + \frac{I_Z\beta_W\left(I_W+I_{WZ}q\right)}{N} \end{bmatrix}$$

NOTE NEXT CELL MAY RUN FOREVER. A python script containing the same code as this section was run on a server for a week. After this time no solution was found. Hence the code is silenced.

[45]:
```
#equilibria = sympy.solve(ODE_mat,[S, I_W, I_Z,I_WZ])
#equilibria
```

5 5. Endemic Equilibria Stability Analyses of the Full Coinfection Model from Spicknall et al (2013)

5.1 5.1 Deriving Equilibria

[46]:
```
I_WW, I_ZZ = sympy.symbols('I_WW I_ZZ')
```

[47]:
```
Eqn_S = - S*beta_W*(I_W + q*(2*I_WW + I_WZ))/N - S*beta_Z*(I_Z + q*(I_WZ +␣
 ↪2*I_ZZ))/N + gamma*(1-epsilon)*I_W + gamma_T*epsilon*I_W + gamma*I_Z +␣
 ↪gamma*(1-epsilon)*I_WW + gamma_T*epsilon*I_WW + I_ZZ*gamma
Eqn_I_W = S*beta_W*(I_W + q*(2*I_WW + I_WZ))/N - I_W*beta_Z*(I_Z + q*(I_WZ +␣
 ↪2*I_ZZ))/N - I_W*beta_W*(I_W + q*(2*I_WW + I_WZ))/N - gamma*(1-epsilon)*I_W␣
 ↪- gamma_T*epsilon*I_W + gamma*I_WZ
Eqn_I_Z = S*beta_Z*(I_Z + q*(I_WZ + 2*I_ZZ))/N - I_Z*beta_W*(I_W + q*(2*I_WW +␣
 ↪I_WZ))/N - I_Z*beta_Z*(I_Z + q*(I_WZ + 2*I_ZZ))/N - gamma*I_Z +␣
 ↪(1-epsilon)*gamma*I_WZ + epsilon*gamma_T*I_WZ
Eqn_I_WZ = beta_W/N*I_Z*(I_W+q*(2*I_WW + I_WZ)) + beta_Z/N*I_W*(I_Z+q*(I_WZ +␣
 ↪2*I_ZZ)) - (1-epsilon)*gamma*I_WZ - epsilon*gamma_T*I_WZ - gamma*I_WZ
Eqn_I_WW = I_W*beta_W*(I_W + q*(2*I_WW + I_WZ))/N - gamma*(1-epsilon)*I_WW -␣
 ↪gamma_T*epsilon*I_WW
Eqn_I_ZZ = I_Z*beta_Z*(I_Z + q*(I_WZ + 2*I_ZZ))/N - I_ZZ*gamma
```

[48]:
```
ODE_mat = sympy.Matrix([Eqn_S,Eqn_I_W,Eqn_I_Z,Eqn_I_WZ,Eqn_I_WW,Eqn_I_ZZ])
ODE_mat
```

[48]:
$$\begin{bmatrix} I_W\epsilon\gamma_T + I_W\gamma\left(1-\epsilon\right) + I_{WW}\epsilon\gamma_T + I_{WW}\gamma\left(1-\epsilon\right) + I_Z\gamma + I_{ZZ}\gamma - \frac{S\beta_W\left(I_W+q\left(2I_{WW}+I_{WZ}\right)\right)}{N} - \frac{S\beta_Z\left(I_Z+q\left(I_{WZ}+2I_{ZZ}\right)\right)}{N} \\ -I_W\epsilon\gamma_T - I_W\gamma\left(1-\epsilon\right) - \frac{I_W\beta_W\left(I_W+q\left(2I_{WW}+I_{WZ}\right)\right)}{N} - \frac{I_W\beta_Z\left(I_Z+q\left(I_{WZ}+2I_{ZZ}\right)\right)}{N} + I_{WZ}\gamma + \frac{S\beta_W\left(I_W+q\left(2I_{WW}+I_{WZ}\right)\right)}{N} \\ I_{WZ}\epsilon\gamma_T + I_{WZ}\gamma\left(1-\epsilon\right) - I_Z\gamma - \frac{I_Z\beta_W\left(I_W+q\left(2I_{WW}+I_{WZ}\right)\right)}{N} - \frac{I_Z\beta_Z\left(I_Z+q\left(I_{WZ}+2I_{ZZ}\right)\right)}{N} + \frac{S\beta_Z\left(I_Z+q\left(I_{WZ}+2I_{ZZ}\right)\right)}{N} \\ \frac{I_W\beta_Z\left(I_Z+q\left(I_{WZ}+2I_{ZZ}\right)\right)}{N} - I_{WZ}\epsilon\gamma_T - I_{WZ}\gamma\left(1-\epsilon\right) - I_{WZ}\gamma + \frac{I_Z\beta_W\left(I_W+q\left(2I_{WW}+I_{WZ}\right)\right)}{N} \\ \frac{I_W\beta_W\left(I_W+q\left(2I_{WW}+I_{WZ}\right)\right)}{N} - I_{WW}\epsilon\gamma_T - I_{WW}\gamma\left(1-\epsilon\right) \\ \frac{I_Z\beta_Z\left(I_Z+q\left(I_{WZ}+2I_{ZZ}\right)\right)}{N} - I_{ZZ}\gamma \end{bmatrix}$$

NOTE NEXT CELL MAY RUN FOREVER.Finding a solution this way runs into the same problem as the Superinfection model. Hence the code is silenced.

[49]:
```
#equilibria = sympy.solve(ODE_mat,[S, I_W, I_Z,I_WZ,I_WW,I_ZZ])
#equilibria
```

5.2 5.2 Endemic Equilibria Stability Analyses of the Full Co-Infection model just Anti-microbial sensitive strain.

[50]:
```
Eqn_S = - S*beta_W*(I_W + q*(2*I_WW))/N + I_W*(epsilon*gamma_T + gamma*(1 -␣
↪epsilon)) + I_WW*(epsilon*gamma_T + gamma*(1 - epsilon))

Eqn_I_W = S*beta_W*(I_W + q*(2*I_WW))/N - I_W*(epsilon*gamma_T + gamma*(1 -␣
↪epsilon)) - I_W*beta_W*(I_W + q*(2*I_WW))/N

Eqn_I_WW = I_W*beta_W*(I_W + q*(2*I_WW))/N - I_WW*(epsilon*gamma_T + gamma*(1 -␣
↪epsilon))
```

[51]:
```
ODEs = sympy.Matrix([Eqn_S, Eqn_I_W, Eqn_I_WW])
ODEs
```

[51]:
$$\begin{bmatrix} I_W\left(\epsilon\gamma_T + \gamma\left(1-\epsilon\right)\right) + I_{WW}\left(\epsilon\gamma_T + \gamma\left(1-\epsilon\right)\right) - \frac{S\beta_W\left(I_W + 2I_{WW}q\right)}{N} \\ -I_W\left(\epsilon\gamma_T + \gamma\left(1-\epsilon\right)\right) - \frac{I_W\beta_W\left(I_W + 2I_{WW}q\right)}{N} + \frac{S\beta_W\left(I_W + 2I_{WW}q\right)}{N} \\ \frac{I_W\beta_W\left(I_W + 2I_{WW}q\right)}{N} - I_{WW}\left(\epsilon\gamma_T + \gamma\left(1-\epsilon\right)\right) \end{bmatrix}$$

[52]:
```
equilibria = sympy.solve(ODEs,[S, I_W, I_WW])
equilibria
```

[52]:
$$\left[\left(S,\ 0,\ 0\right),\ \left(\frac{-2I_W\beta_W q + I_W\beta_W - N\epsilon\gamma + N\epsilon\gamma_T + N\gamma}{\beta_W},\ I_W,\ -\frac{I_W^2\beta_W}{2I_W\beta_W q + N\epsilon\gamma - N\epsilon\gamma_T - N\gamma}\right)\right]$$

The disease free equilibrium (DFE) is the first vector.

[53]:
```
DFE = equilibria[0]
EE = equilibria[1]
display(DFE,EE)
```

$$\left(S,\ 0,\ 0\right)$$

$$\left(\frac{-2I_W\beta_W q + I_W\beta_W - N\epsilon\gamma + N\epsilon\gamma_T + N\gamma}{\beta_W},\ I_W,\ -\frac{I_W^2\beta_W}{2I_W\beta_W q + N\epsilon\gamma - N\epsilon\gamma_T - N\gamma}\right)$$

The endemic equilibrium for the AMS strain is not that useful lets try:

5.2.1 5.1.1 Substituting $q = 0.5$ in the system.

[54]:
```
ODEs_q_is_a_half = ODEs.subs({q:0.5})
display(ODEs_q_is_a_half)
```

$$\begin{bmatrix} I_W\left(\epsilon\gamma_T + \gamma\left(1-\epsilon\right)\right) + I_{WW}\left(\epsilon\gamma_T + \gamma\left(1-\epsilon\right)\right) - \frac{S\beta_W\left(I_W + 1.0I_{WW}\right)}{N} \\ -I_W\left(\epsilon\gamma_T + \gamma\left(1-\epsilon\right)\right) - \frac{I_W\beta_W\left(I_W + 1.0I_{WW}\right)}{N} + \frac{S\beta_W\left(I_W + 1.0I_{WW}\right)}{N} \\ \frac{I_W\beta_W\left(I_W + 1.0I_{WW}\right)}{N} - I_{WW}\left(\epsilon\gamma_T + \gamma\left(1-\epsilon\right)\right) \end{bmatrix}$$

[55]:
```
equilibria_q_is_a_half = sympy.solve(ODEs_q_is_a_half,[S, I_W, I_WW])
equilibria_q_is_a_half
```

[55]:
$$\left[(S,\ 0.0,\ 0.0),\ \left(\frac{N\left(-\epsilon\gamma+\epsilon\gamma_T+\gamma\right)}{\beta_W},\ I_W,\ -\frac{I_W^2\beta_W}{I_W\beta_W+N\epsilon\gamma-N\epsilon\gamma_T-N\gamma}\right)\right]$$

[57]:
```
EE_q_is_a_half = equilibria_q_is_a_half[1]
display(EE_q_is_a_half)
```

$$\left(\frac{N\left(-\epsilon\gamma+\epsilon\gamma_T+\gamma\right)}{\beta_W},\ I_W,\ -\frac{I_W^2\beta_W}{I_W\beta_W+N\epsilon\gamma-N\epsilon\gamma_T-N\gamma}\right)$$

Solve I_W knowing that $N = S + I_W + I_{WW}$.

[58]:
```
test_eqn = EE_q_is_a_half[0] + EE_q_is_a_half[1] + EE_q_is_a_half[2]
test_eqn
```

[58]:
$$-\frac{I_W^2\beta_W}{I_W\beta_W+N\epsilon\gamma-N\epsilon\gamma_T-N\gamma}+I_W+\frac{N\left(-\epsilon\gamma+\epsilon\gamma_T+\gamma\right)}{\beta_W}$$

[59]:
```
test_sol = sympy.solve(test_eqn-N,I_W)
test_sol
```

[59]:
$$\left[\frac{N\left(-\epsilon\gamma+\epsilon\gamma_T+\gamma\right)\left(\beta_W+\epsilon\gamma-\epsilon\gamma_T-\gamma\right)}{\beta_W^2}\right]$$

[60]:
```
EE_S_q_is_a_half = EE_q_is_a_half[0]
EE_I_W_q_is_a_half = test_sol[0]
display(EE_S_q_is_a_half,EE_I_W_q_is_a_half)
```

$$\frac{N\left(-\epsilon\gamma+\epsilon\gamma_T+\gamma\right)}{\beta_W}$$

$$\frac{N\left(-\epsilon\gamma+\epsilon\gamma_T+\gamma\right)\left(\beta_W+\epsilon\gamma-\epsilon\gamma_T-\gamma\right)}{\beta_W^2}$$

[61]:
```
EE_I_WW_q_is_a_half = sympy.
↪solve(N-(EE_S_q_is_a_half+EE_I_W_q_is_a_half+I_WW),I_WW)
EE_I_WW_q_is_a_half
```

[61]:
$$\left[\frac{N\left(\beta_W^2+2\beta_W\left(\epsilon\gamma-\epsilon\gamma_T-\gamma\right)+\epsilon^2\gamma^2-2\epsilon^2\gamma\gamma_T+\epsilon^2\gamma_T^2-2\epsilon\gamma^2+2\epsilon\gamma\gamma_T+\gamma^2\right)}{\beta_W^2}\right]$$

[62]:
```
EE_I_WW_q_is_a_half = EE_I_WW_q_is_a_half[0].simplify()
EE_I_WW_q_is_a_half
```

[62]:
$$\frac{N\left(\beta_W^2-2\beta_W\left(-\epsilon\gamma+\epsilon\gamma_T+\gamma\right)+\epsilon^2\gamma^2-2\epsilon^2\gamma\gamma_T+\epsilon^2\gamma_T^2-2\epsilon\gamma^2+2\epsilon\gamma\gamma_T+\gamma^2\right)}{\beta_W^2}$$

[63]:
```
EE_q_is_a_half = sympy.Matrix([EE_S_q_is_a_half, EE_I_W_q_is_a_half,␣
↪EE_I_WW_q_is_a_half])
EE_q_is_a_half
```

[63]:
$$\begin{bmatrix} \frac{N(-\epsilon\gamma+\epsilon\gamma_T+\gamma)}{\beta_W} \\ \frac{N(-\epsilon\gamma+\epsilon\gamma_T+\gamma)(\beta_W+\epsilon\gamma-\epsilon\gamma_T-\gamma)}{\beta_W^2} \\ \frac{N\left(\beta_W^2-2\beta_W(-\epsilon\gamma+\epsilon\gamma_T+\gamma)+\epsilon^2\gamma^2-2\epsilon^2\gamma\gamma_T+\epsilon^2\gamma_T^2-2\epsilon\gamma^2+2\epsilon\gamma\gamma_T+\gamma^2\right)}{\beta_W^2} \end{bmatrix}$$

[64]:
```
EE_q_is_a_half.subs({'N':1e3,'beta_W': 0.04 , 'beta_Z': 0.015,'gamma': 0.
↪01,'gamma_T': 0.1,'epsilon':0,'q':0.5})
```

[64]:
$$\begin{bmatrix} 250.0 \\ 187.5 \\ 562.5 \end{bmatrix}$$

[ ]: