The Pinsky-Rinzel Model and Optimal Control

Cade Ballew and Ricky Morse

Department of Computational and Applied Mathematics
Rice University, Houston, Texas

June&July, 2019

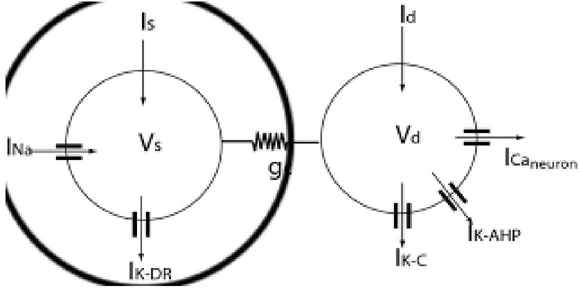
REU Summer 2019

Outline

The Pinsky-Rinzel Model

What is it?

- A system of differential equations
- Models behavior of CA3 neurons in the hippocampus
- Modeled in one or two compartments and extrapolated to larger neurons



- Only consider one compartment (circled part of model)

The One-Compartment Model

$$C_m \frac{d}{dt}\mathbf{V}_s(t) = -I_{leak}(t) - I_{Na}(t) - I_{K-DR}(t) + \frac{I_s}{p}$$
$$\frac{d}{dt}\mathbf{h}(t) = \alpha_h(t)(1 - \mathbf{h}(t)) - \beta_h(t)\mathbf{h}(t)$$
$$\frac{d}{dt}\mathbf{n}(t) = \alpha_n(t)(1 - \mathbf{n}(t)) - \beta_n(t)\mathbf{n}(t)$$

$$I_{Na} = g_{Na} m_\infty^2(t)\mathbf{h}(t)(\mathbf{V}_s(t) - V_{Na})$$
$$I_{K-DR} = g_{K-DR}\mathbf{n}(t)(\mathbf{V}_s(t) - V_K)$$

$$I_{Leak} = g_L(\mathbf{V}_s(t) - V_L)$$
$$m_\infty(t) = \frac{\alpha_m(t)}{\alpha_m(t) + \beta_m(t)}$$

- Want to solve system for $\mathbf{V}_S(t)$, $\mathbf{h}(t)$, $\mathbf{n}(t)$
- αs and βs functions in V_s and t; exact definitions omitted
- Conductances g_{Na}, g_L, g_{K-DR} may be unknown

What do we know?

Assume constants:

- $V_{Na} = 120$, $V_K = -15$, $V_L = 0$
- $p = 0.5$, $C_m = 3$, $I_s = 0.25$

Other 'constants' are uncertain:

- We think $g_L = 0.1$, $g_{Na} = 30$, $g_{K-DR} = 15$
- But only know that $0.01 \leq g_L \leq 2$, $1 \leq g_{Na}, g_{K-DR} \leq 50$

Also have bounds on functions to be solved for:

- We guess that $V_s(t) = -4.6$, $h(t) = 0$, $n(t) = 0$
- And know for certain that $-15 \leq V_s(t) \leq 120$, $0 \leq h(t), n(t) \leq 1$

Solving the Model

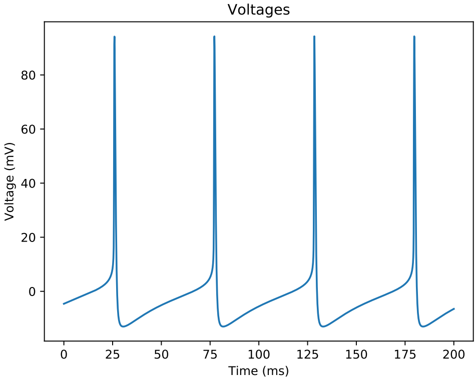
- Recall that

$$
\begin{aligned}
C_m \frac{d}{dt}\mathbf{V}_s(t) &= -I_{leak}(t) - I_{Na}(t) - I_{K-DR}(t) + \frac{I_s}{p} \\
\frac{d}{dt}\mathbf{h}(t) &= \alpha_h(t)(1-\mathbf{h}(t)) - \beta_h(t)\mathbf{h}(t) \\
\frac{d}{dt}\mathbf{n}(t) &= \alpha_n(t)(1-\mathbf{n}(t)) - \beta_n(t)\mathbf{n}(t)
\end{aligned}
$$

- If we set $x(t) = (V_s(t), h(t), n(t))^T$ then $\dot{x}(t) = f(x(t), p)$ where $p = (g_{Na}, g_{K-DR}, g_L)^T$.
- Given m measured values of V_s at various times, $t_1, ..., t_m$, yielding measurements $V_s^{meas}(t_1), ..., V_s^{meas}(t_m)$.
- Problem can be restated as

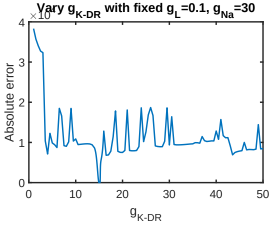
$$
\min_p \sum_{i=1}^{m} (V_s^{meas}(t_i) - V_s^{est}(t_i))^2
$$

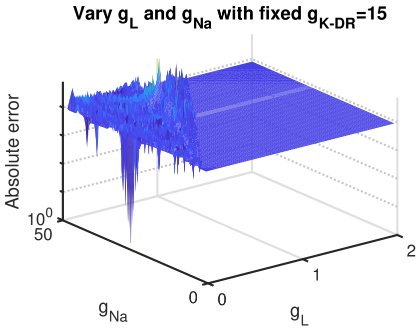
Difficulties



- Spiking behaviour makes modeling with polynomials difficult
- Easy to get stuck in a local minimum where only some spikes are modeled

Difficulty Analysis





- Many local minimum while global minimum is much lower than anything else

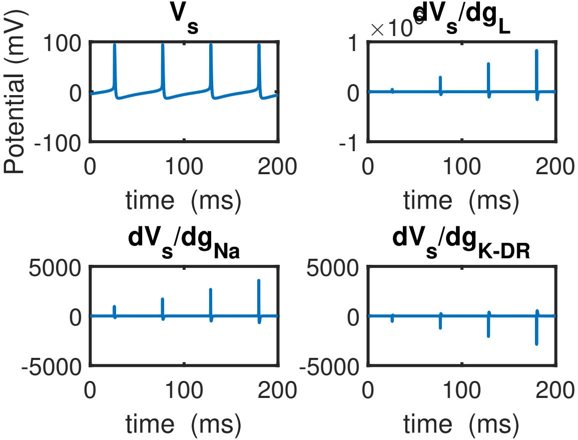
Sensitivity Analysis

Can compute sensitivities for V_S, h, and n as follows:

- First recall problem structure $\dot{x}(t) = f(x(t), p)$
- Compute parital Jacobians $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial p}$
- Set $\frac{d}{dt}W = \frac{\partial f}{\partial x}W + \frac{\partial f}{\partial p}$
- Reshape equation as vector, append to original problem, and solve

Sensitivity Analysis (cont.)

Sensitivities solved for with MATLAB's ode23s function



- Small changes in constants create large errors

Backward/Implicit Euler Discretization

- Need to choose proper discretization to circumvent difficulties
- Want to transform state $x(t)$ and ODE $\dot{x}(t)$ from continuous to discrete

Given n discrete times, $t_1, t_2..., t_n$, approximate a continuous function $y(t)$ by a discrete set $y_1, ..., y_n$as:

$$y_j = y_{j-1} + h * f(t_j, y_j),$$

where h is step size and $f(t_j, y_j)$ is the derivative (or an approximation of it) at $y(t_j)$.

- ODE redefined as $\dot{x}_j = \frac{x_j - x_{j-1}}{t_j - t_{i-j}}$
- Relatively simple discretization, but works better for this problem as complex polynomials can hurt when modeling spiking behavior

Solving the Model Revisited

Recall error problem

$$\min_{p} \sum_{i=1}^{m} (V_s^{meas}(t_i) - V_s^{est}(t_i))^2$$

Applying backward Euler to estimate V_s, problem becomes

$$\min_{p} \sum_{i=1}^{m} (V_s^{meas}(t_i) - (e_i^T x_j)_1)^2$$

$$s.t. \quad x_j = x_{j-1} + h * f(t_j, y_j), \quad j = 1, ..., n$$

- It is also helpful if we minimize over x_j in addition to the parameters, giving us more degrees of freedom, problem becomes

$$\min_{x_j, p} \sum_{i=1}^{m} (V_s^{meas}(t_i) - (e_i^T x_j)_1)^2$$

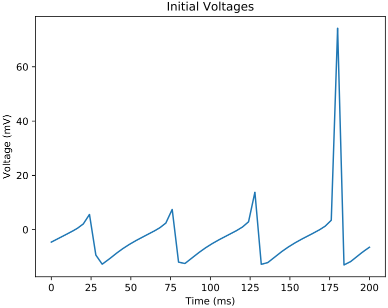
- The ODE is still constraining x_j

Initializing the Model (Warm Start)

- Can deal with the difficulties of solving this model by initializing the voltages to be close to true values.

Given m measured values of the voltage at times $vt_1, ..., vt_m$, and n time steps $t_1, ..., t_n$ initialize $V_{s1}^{est}, ..., V_{sn}^{est}$ following:

- for $t_i, ..., t_n$
- Whenever we have a measured voltage at our current time, initialize V_s^{est} to be that measured voltage
- Otherwise initialize V_s^{est} to be on the line connecting the two closest V_s^{meas}.



V_s^{meas} every 4 seconds

Outline

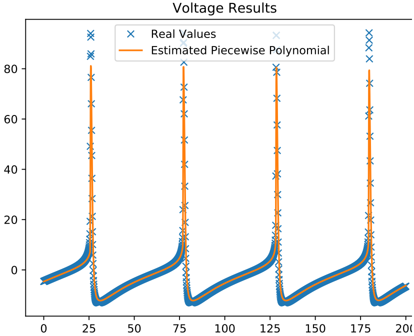
The One Compartment Model

Results

Introduction to Pyomo

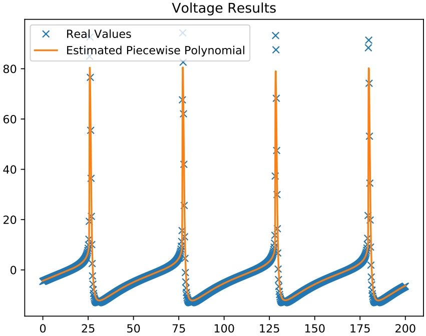
Results

- We fix $n = 2001$, $g_L = 0.1$, and $g_{KDR} = 15$ and solve for g_{Na} varying m each time. The "true" data we are modeling has a time step of 0.05 ms for 200 ms making 4001 times and 4001 measured voltages.



Info: Time Steps = 2001, Measured Voltages = 2001

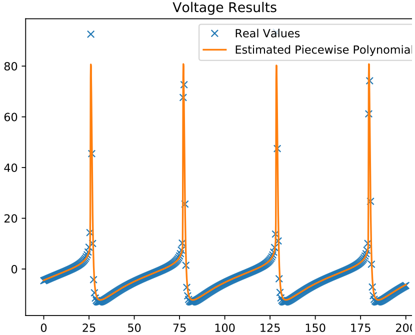
Results: gNa = 28.143, gKDR = 15.0, gL = 0.1
Absolute Error = 9586.836, Relative Error = 4.791
Runtime = 13.928 s



Info: Time Steps = 2001, Measured Voltages = 1001

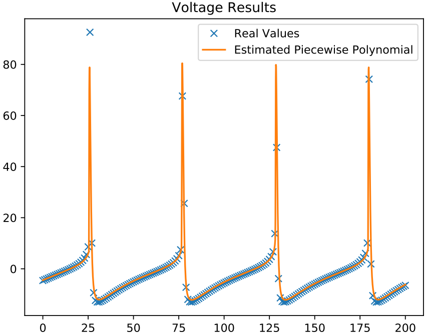
Results: gNa = 27.735, gKDR = 15.0, gL = 0.1
Absolute Error = 4462.891, Relative Error = 4.458
Runtime = 15.454 s

Results



Info: Time Steps = 2001, Measured Voltages = 401

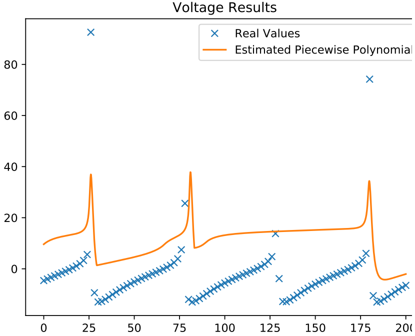
Results: gNa = 28.033, gKDR = 15.0, gL = 0.1
Absolute Error = 1603.461, Relative Error = 3.999
Runtime = 15.797 s

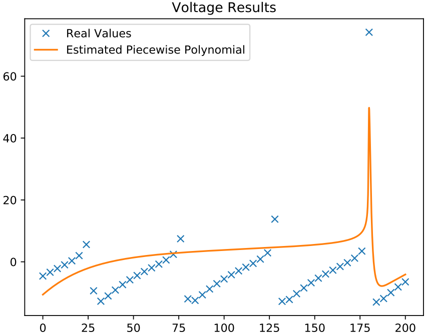


Info: Time Steps = 2001, Measured Voltages = 201

Results: gNa = 27.508, gKDR = 15.0, gL = 0.1
Absolute Error = 1237.714, Relative Error = 6.158
Runtime = 20.831 s

Results





Info: Time Steps = 2001, Measured Voltages = 101

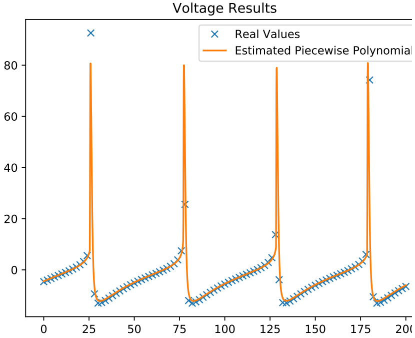
Results: gNa = 3.305, gKDR = 15.0, gL = 0.1
Absolute Error = 30942.137, Relative Error = 306.358
Runtime = 205.359 s

Info: Time Steps = 2001, Measured Voltages = 51

Results: gNa = 6.274, gKDR = 15.0, gL = 0.1
Absolute Error = 4518.712, Relative Error = 88.602
Runtime = 33.559 s

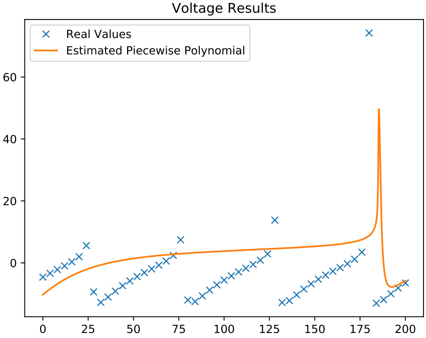
Results

- Initializing g_{Na} to be 30.0 to see if that improves results.



Info: Time Steps = 2001, Measured Voltages = 101

Results: gNa = 27.688, gKDR = 15.0, gL = 0.1
Absolute Error = 3717.657, Relative Error = 36.808
Runtime = 60.294 s



Info: Time Steps = 2001, Measured Voltages = 51

Results: gNa = 6.236, gKDR = 15.0, gL = 0.1
Absolute Error = 8870.891, Relative Error = 173.939
Runtime = 101.68 s

Results

$n = 2001$, $g_L = 0.1$, and $g_{KDR} = 15$.

m	g_{Na}	Relative Error	Runtime
2001	28.143	4.791	13.928
1001	27.735	4.458	15.454
401	28.033	3.999	15.797
201	27.508	6.158	20.831
101	3.305	306.4	205.359
51	6.274	88.6	33.56
101 (init)	27.688	36.8	60.294
51 (init)	6.236	173.9	101.68

Further Research

- How does varying n effect results?
- What happens when we vary other parameters (or all 3)?
- How does initializing the different parameters effect results?
- How does enforcing initial conditions / enforcing ODE's at boundaries effect results?
- How many measurements do we need and how many do we need to precisely initialize?
- Two Compartment Model.

Outline

The One Compartment Model

Results

Introduction to Pyomo

What is Pyomo?

- A way to model optimization problems in Python
- Simple code structure similar to standard problem

$$
\begin{aligned}
& \min_x f(x) \\
s.t. \quad & g(x) \leq 0 \\
& h(x) = 0
\end{aligned}
$$

- User can optimize directly from Python code
- Program does all necessary Lagrangians, Hessians, Jabocbians, etc. implicitly with chosen solver

Installation

- Anaconda Python distribution recommended for easier installation

Using Conda, execute these commands in a shell:

```
conda install -c conda-forge pyomo
conda install -c conda-forge pyomo.extras
```

- Also need to install a solver

For linear problems glpk works well

```
conda install -c conda-forge glpk
```

For differential equations need to use nonlinear solver

```
conda install -c conda-forge ipopt
```

Creating a Model Object

Begin by creating a .py file and importing the Pyomo package

```
from pyomo.environ import *
```

Two types of models:

- Concrete: values for all parameters given as constants

```
model = ConcreteModel()
```

- Abstract: parameters may be represented with Param structure instead of constants but must be defined in separate file; not typically used with DAE extension

```
model = AbstractModel()
```

A Simple Example of a Concrete Model

Consider this model problem

$$\min_{(x_1,x_2)} f(x) = 2x_1 + x_2$$
$$s.t. \quad g(x) = -x_1 - x_2 \leq 0$$
$$h(x) = x_1^2 + x_2^2 - 25 = 0$$

For notation, say $x \in \mathbb{R}^k$ $(k = 2)$ and let M be the set of indices for x (i.e. $\{1, 2\}$) Code these as

```
model.k = 2 #x in R^k
model.M = RangeSet(1, model.k) #set {1,2}
model.x = Var(model.M)
```

Objective and Constraints

Create a Python function for the objective ($f(x)$)

```
def f_(model):
    return 2*model.x[1] + model.x[2]
model.cost = Objective(rule = f_)
```

Create functions for the constraints ($g(x)$ and $h(x)$) as well

```
def g_(model):
    return -sum(model.x[i] for i in model.M) <= 0
model.inequality = Constraint(rule = g_)

def h_(model):
    return (model.x[1])**2 + (model.x[2])**2 - 25 == 0
model.equality = Constraint(rule = h_)
```

- Structure of function g in more general form ideal for more abstract problems

Solving the Simple Model

Solve model by

```
solver = SolverFactory('ipopt') #chosen solver here
solver.solve(model)
results = solver.solve(model,tee=True)
#tee set to true for detailed solver info
```

Can print the solution with the lines

```
print(results)
print(value(model.x[1]))
print(value(model.x[2]))
```

Which yields $x = (-3.5355, 3.5355)$
To add a requirement that $x \geq 0$, change model.x to

```
model.x = Var(model.M, within = NonNegativeReals)
```

which gives $x = (0, 5)$

- Can add similar conditions such as PositiveIntegers and Binary to fit model's needs

Differences with Abstract Models

- Set: used to initialize abstract sets;
  e.g. a list of meal options as

  ```
  model.F = Set()
  ```

- Param: for parameters assigned outside the model
  e.g. list of prices for i goods as

  ```
  model.p = Param(model.i, within = PositiveReals)
  ```

- Have to give data for Sets and Params in a separate .dat file and solve by running command

  ```
  pyomo solve --solver=ipopt file_name.py file_name.dat
  ```

The DAE Extension

DAE: Differential Algebraic Equations; can find optimal functions instead of just vectors
Adds three new structures

- ContinuousSet: an interval between two bounds
  - Typically used to represent time
  - Requires a discretization for the model to be solved
- DerivativeVar: declares a derivative of some variable (Var) w.r.t. a ContinuousSet
- Integral: declares an integral;
  should not be used unless absolutely necessary due to working issues

A DAE Test Problem

Want to determine the minimum force required to move a cart 100 meters in 10 seconds; results in this problem:

$$
\begin{aligned}
& min \int_0^{10} u^2(t)\,\mathrm{d}t \\
s.t. \quad & \frac{d}{dt} f_1(t) = f_2(t), t \in [0,10] \\
& \frac{d}{dt} f_2(t) = \frac{u(t)}{M}, t \in [0,10] \\
& f_1(0) = 0, f_2(0) = 0 \\
& f_1(10) = 100
\end{aligned}
$$

where unknowns are f_1, f_2 (states), and u (control)

Modeling the Cart Problem

Begin by importing packages

```
from pyomo.environ import *
from pyomo.dae import *
```

Use ConcreteModel for DAE problems

```
model = ConcreteModel()
```

Set time as ContinuousSet

```
model.t = ContinuousSet(bounds=(0,10))
```

Take mass to be 1

```
model.M = 1
```

And set unknowns as functions of t

```
model.u = Var(model.t, initialize = 0)
model.f1 = Var(model.t)
model.f2 = Var(model.t)
```

Initialize u to compensate for its lack of boundary conditions

Modeling the Cart Problem (cont.)

To avoid Integral object, define function

```
model.myobj = Var(model.t)
```

where $myobj(t) = \int_0^t u^2(\tau)\,\mathrm{d}\tau$ so objective is

$$minimize\ myobj(10)$$

and

$$\frac{d}{dt}myobj(t) = u^2(t)$$

Define derivatives

```
model.df1dt = DerivativeVar(model.f1, wrt = model.t)
model.df2dt = DerivativeVar(model.f2, wrt = model.t)
model.dmyobjdt = DerivativeVar(model.myobj,
          wrt = model.t)
```

Modeling the Cart Problem (cont.)

Define the objective

```
model.obj = Objective(expr = model.myobj[10])
```

Add original constraints ignoring initial conditions

```
def velo(model, t):
    if t == 0:
        return Constraint.Skip
    return model.df1dt[t] == model.f2[t]
model.consb = Constraint(model.t, rule = velo)

def accel(model, t):
    if t == 0:
        return Constraint.Skip
    return model.df2dt[t] == model.u[t] / model.M
model.consc = Constraint(model.t, rule = accel)
```

Modeling the Cart Problem (cont.)

Add new constraint on myobj

```
def objcond(model, t):
    if t == 0:
        return Constraint.Skip
    return model.dmyobjdt[t] == (model.u[t])**2
model.objcons = Constraint(model.t, rule = objcond)
```

Use ConstraintList object to add boundary conditions

```
def initcond1(model):
    yield model.f1[0] == 0
    yield model.f2[0] == 0
    yield model.f1[10] == 100
    yield model.myobj[0] == 0
    #clear from definition of myobj
    yield ConstraintList.End
model.consd1 = ConstraintList(rule = initcond1)
```

Discretizing and Solving

Need to choose a discretization scheme and apply to the model (using Lagrange-Radau Collocation here)

```
discretizer = TransformationFactory('dae.collocation')
discretizer.apply_to(model, nfe=20, ncp=3,
scheme='LAGRANGE-RADAU')
```

And solving works the same as before

```
solver=SolverFactory('ipopt')
results = solver.solve(model,tee=True)
print(results)
```

Plotting Results

If we want to plot solutions, can use this method:

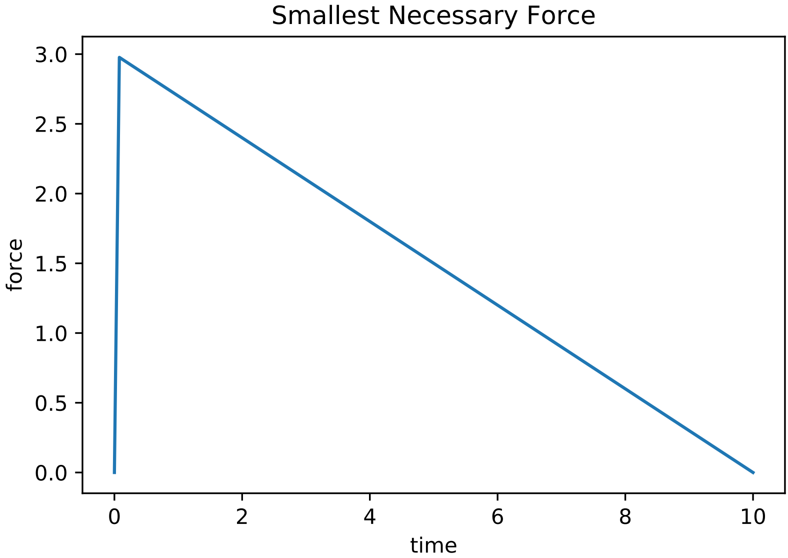
```
f1 = []
f2 = []
u = []
t=[]

for i in sorted(model.t):
    t.append(i)
    f1.append(value(model.f1[i]))
    f2.append(value(model.f2[i]))
    u.append(value(model.u[i]))

import matplotlib.pyplot as plt

plt.plot(t,u) #can also plot f1, f2 here
plt.xlabel('time')
plt.ylabel('force')
plt.title('Smallest Necessary Force')
plt.show()
```

Problem Solution



Pinsky Rinzel in Pyomo

Reference Pinksy-Rinzel-Final in PR Pyomo folder of dropbox.