Special Problems Final Report

Lucas plant
Georgia Institute of Technology
lucasplant@gatech.edu

Abstract

Over the Spring 2025 semester, I conducted a special problems course with Professor Verriest. I learned a lot about various topics such as optimal control theory, differential equations, group theory, and physics. I have noticed that this learning is helping me to gain new perspectives and understand concepts from a mathematical and theoretical background. This has helped me in many of my classes this semester such as Control System Design (ECE 4550), Fundamentals of Digital Signal Processing (ECE 4270), and Machine Learning (CS 4641). Additionally, this course has increased my interest in pursuing postgraduate studies and has helped train my skills in complex mathematics that will be needed in my graduate studies. I would like to thank Professor Verriest for all the time he spent teaching me this semester.

1 Optimal Control Fundamentals

Throughout this semester, I learned the fundamentals of optimization within the context of finding the optimal control of a system. To start an optimal control problem, we must first define the cost function that we wish to optimize, which will typically take the form:

$$J = \Phi(x(t_0), x(t_f)) + \int_{t_0}^{t_f} L(x(t), u(t))dt$$

Next, we must define the system dynamics, which take the general nonlinear form:

$$\dot{x}(t) = f(x(t), u(t))$$

We may also define other constraints and boundary conditions to our problem, such as fixing the final time and/or the final state $x(t_f) = T_0$. From here, we will join our cost function using the Hamiltonian and Lagrange Multipliers:

$$\mathcal{H}(x(t), u(t), \lambda(t)) = L(x(t), u(t)) + \lambda(t)^T f(x(t), u(t))$$

We can now solve our optimization problem using the Euler-Lagrange Equations, also known as Pontryagin's Minimum Principle:

$$\dot{\lambda}(t) = -\frac{\partial \mathcal{H}}{\partial x},\ \lambda(t_f) = \frac{\partial \Phi}{\partial x}|_{t_f}$$

And using the optimality condition:

$$\frac{\partial \mathcal{H}}{\partial u} = 0$$

Additionally, many practical problems involve the assumption that our system is linear, which leads to the simplified dynamical constraint:

$$\dot{x}(t) = Ax(t) + Bu(t)$$

Another common simplification is taking the Lagrangian to be a quadratic form with respect to x and u:

$$L(x(t), u(t)) = x^T Q x + u^T R u + x^T N u$$

The combination of the 2 previous simplifications gives rise to the famous LQ problem. The addition of a Gaussian noise term $\nu(t)$ to the state equation gives rise to the famous LQG problem from which the Kalman Filter is derived:

$$\dot{x}(t) = Ax(t) + Bu(t) + \nu(t)$$

1.1 Connecting EL to Physics

In physics, we often make use of a different form of the Euler-Lagrange equation:

$$\frac{\partial L}{\partial q} - \frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}}\right) = 0$$

At first glance, this equation looks quite different than the Euler-Lagrange Equations that we use in optimal control:

$$\dot{\lambda} = -\frac{\partial \mathcal{H}}{\partial x},\ \frac{\partial \mathcal{H}}{\partial u} = 0$$

As I will show, the equation from physics is a special case of the optimization equation. First, consider the state dynamics:

$$\dot{x} = u = f(x(t), u(t))$$

Now substitute these into the Euler-Lagrange Equation:

$$\dot{\lambda} = -\frac{\partial \mathcal{H}}{\partial x} = -(\frac{\partial L}{\partial x} + \dot{x}\frac{\partial}{\partial x}\lambda + \lambda\frac{\partial}{\partial x}\dot{x}) = -\frac{\partial L}{\partial x}$$

Now substitute into the optimality condition:

$$\frac{\partial \mathcal{H}}{\partial u} = \frac{\partial \mathcal{H}}{\partial \dot{x}} = \frac{\partial L}{\partial \dot{x}} + \dot{x}\frac{\partial}{\partial \dot{x}}\lambda + \lambda\frac{\partial}{\partial \dot{x}}\dot{x} = \frac{\partial L}{\partial \dot{x}} + \lambda = 0$$
$$\lambda = -\frac{\partial L}{\partial \dot{x}}$$

Take the time derivative and substitute:

$$\dot{\lambda} = -\frac{d}{dt}\frac{\partial L}{\partial \dot{x}} = -\frac{\partial L}{\partial x}$$

Now we can simply rearrange terms and replace x with q, and we are left with the Euler-Lagrange Equation from physics:

$$\frac{\partial L}{\partial q} - \frac{d}{dt}(\frac{\partial L}{\partial \dot{q}}) = 0$$

1.2 Optimal Temperature Control of a House

Let's consider a simple practice problem concerning finding the optimal temperature control of a house. When we leave the house in the morning, we want it to be a target temperature T_c. To ensure that the system is linear we deffine T_c as the difference from the outside temperature. When we come back from work after a time of t_f we want our house to be at T_c as well. We want to find the optimal control $u(t)$ from 0 to t_f so that $x(0) = x(t_f) = T_c$. Next, we must define a model for the temperature in the house. We can model this using Newton's law of cooling:

$$\dot{x}(t) = -\alpha x(t) + \beta u(t)$$

Here α represents how quickly the house loses heat to the outside. β and u represent our controller's ability to heat or cool the house. Next, we define our cost function, which will simply be our control input squared. There is no need for a coefficient because optimizing the squared u(t) would be equivalent to optimizing a constant times the squared u(t):

$$J = \int_0^{t_f} u(t)^2 dt$$

Now to solve, let's join our cost function and our state dynamics with the Hamiltonian:

$$\mathcal{H}(x(t), u(t), \lambda(t)) =$$
$$u(t)^2 + \lambda(t)(-\alpha x(t) + \beta u(t))$$

Next, we will apply the Euler-Lagrange equations:

$$\dot{\lambda}(t) = -\frac{\partial \mathcal{H}}{\partial x} = \alpha\lambda(t)$$

We did not define a "Parking Fee," also known as an endpoint cost, so we will disregard that portion of the Euler-Lagrange Equations. Now we will define our optimality condition:

$$\frac{\partial \mathcal{H}}{\partial u} = 2u(t) + \beta\lambda(t) = 0$$

Solving for $u^*(t)$ (the optimal u(t)):

$$u^*(t) = -\frac{\beta}{2}\lambda(t)$$

Now we must solve for $\lambda(t)$. using $\dot{\lambda} = \alpha\lambda(t)$ we can state the $\lambda(t)$ takes the form:

$$\lambda(t) = Ce^{\alpha t}$$

Where C is a constant that we must solve for. Because we have defined the problem with a fixed final time lets re define $\lambda(t)$ in terms of the $\lambda_f = \lambda(t_f)$ to make future steps simpler.

$$\lambda(t) = Ce^{\alpha(t-t_f)}$$
$$\lambda(t_f) = \lambda_f = Ce^{\alpha(0)} = C$$
$$\lambda(t) = \lambda_f e^{\alpha(t-t_f)}$$

Next, from differential equation,s we can show that the solution of x(t) is equivalent to the addition of the homogeneous solution plus the driven solution:

$$x(t) = T_c e^{-\alpha t} + \int_0^t T_c e^{-\alpha(t-\tau)}\beta u(\tau) d\tau$$
$$x(t) = T_c e^{-\alpha t} - \int_0^t T_c e^{-\alpha(t-\tau)}\beta\frac{\beta}{2}\lambda_f e^{\alpha(\tau-t_f)} d\tau$$
$$x(t) = T_c e^{-\alpha t} - \frac{\lambda_f T_c \beta^2}{2} e^{-\alpha(t_f+t)} \int_0^t e^{2\alpha\tau} d\tau$$
$$x(t) = T_c e^{-\alpha t} - \frac{\lambda_f T_c \beta^2}{2} e^{-\alpha(t+t_f)} \frac{1}{2\alpha}(e^{2\alpha t} - e^0)$$

Now solve for λ_f by setting t to t_f

$$T_c = T_c e^{-\alpha t_f} - \frac{\lambda_f T_c \beta^2}{2} e^{-2\alpha t_f} \frac{1}{2\alpha}(e^{2\alpha t_f} - 1)$$
$$\lambda_f = \frac{(e^{-\alpha t_f} - 1)4a}{\beta^2 e^{-2at_f}(e^{2\alpha t_f} - 1)}$$

Now we are left with the final solution:

$$u^*(t) = -\frac{\beta}{2}\lambda_f e^{\alpha(t-t_f)}$$

With:

$$\lambda_f = \frac{(e^{-\alpha t_f} - 1)4a}{\beta^2 e^{-2at_f}(e^{2\alpha t_f} - 1)}$$

We can simulate this system in MATLAB. The code can be found at the bottom of this document:

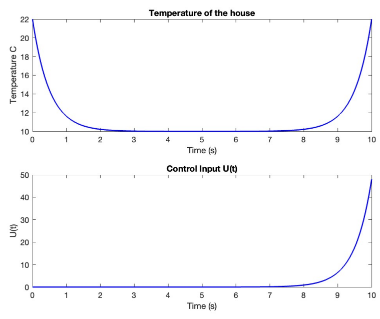


Figure 1: Simulation of the optimal controller of the temperature of a house

2 Maximally Smooth Transitions

Over this semester, I read several papers by Professor Veriest about determining maximally smooth transitions between 2 trajectories. I will briefly summarize what I have learned from these papers.

Often in robotics applications, we have 2 trajectories that our robot can follow, also known as gaits. These can be the gaits that allow our robot to walk or run. The question of maximally smooth transitions is how we can connect or transition between 2 gaits in a maximally smooth manner.

The first technique mentioned in the paper is referred to as the image representation. In this, we define ϕ as a map that maps us from our function space $\mathcal{X}$ to a parameter space Θ. We can then define our starting and ending trajectories by θ_i and θ_f. The maximally smooth trajectory can then be shown to be a geodesic from θ_i to θ_f in Θ. A simple example of this considered in the paper is one in which we transition between 2 sinusoidal trajectories where the geodesic involves linearly scaling the parameters at a constant rate from their initial to their final values.

The second representation is referred to as the Kernel Representation. In this, we expand our function space $\mathcal{X}$ to a larger space $\mathcal{Y}$. We then define some operator whose kernel is $\mathcal{X}$. We then seek the trajectory in $\mathcal{Y}$ that minimizes the previously defined operator.

3 Modeling the Root Locus With Electromagnetics

Root Locus theory from classical control shares a lot of mathematical similarities with electrostatics. This is mainly because both of these equations satisfy the equation $\nabla^2 f = 0$ when there are no charges/poles/zeros. We can take this analogy a step further by using point charges to represent poles and zeros.

First, let's look at the voltage equations. From Gauss' law in derivative form, we have:

$$\nabla \cdot E = -\nabla^2 V = \frac{\rho}{\epsilon_o}$$

We see from this that our function is analytic in all locations that don't contain a charge. We can also observe Gauss's law in integral form:

$$\oint_S E{\cdot}da = \frac{1}{\epsilon_o}(\sum(+Charges) - \sum(-Charges))$$
$$\oint_S \nabla V da =$$
$$\frac{1}{\epsilon_o}(\sum(-Charges) - \sum(+Charges))$$

Now, let's take a look at the equations governing H(s). First, it can be shown that if our transfer function takes the form:

$$H(S) = \frac{b(s)}{a(s)}$$

Where a and b are polynomial functions of s, then ln(H(s)) is an analytic function wherever we don't have poles and zeros. Implying:

$$\nabla^2 H(s) = 0$$

Next, we can show that the contour integral of the logarithmic derivative of H(s) takes the form:

$$\int_\Gamma \frac{d\ln(H(S))}{dS} = \sum(\alpha_i) - \sum(\beta_i)$$

Where α_i corresponds to the system zeros and β_i corresponds to the system poles. We can now draw a direct parallel between the surface $ln(H(S))$ and the potential field. We must simply place point charges with charge = ϵ_o where we have poles and zeros. Using MATLAB code that can be found in the appendix, we can plot this surface using both the potential method and by plotting the log of the magnitude of the transfer function. By doing this, I observe that these surfaces don't look the same. I am not entirely sure why this is. 1 possible explanation I have is that the location of discontinuities, as well as the surface being harmonic, may not uniquely define the surface. Another explanation I have is that we must stretch or shrink the domain or axes of the functions in some way to get the 2 figures to align.

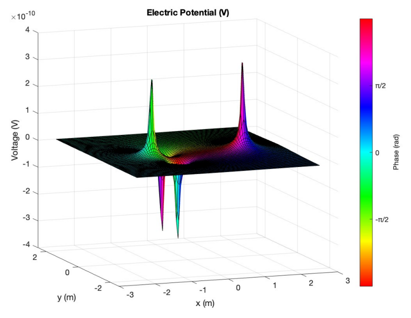


Figure 2: The Electric Potential surface

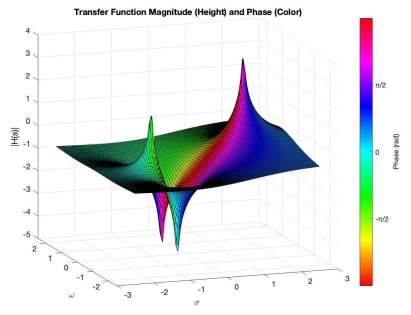


Figure 3: The log of the magnitude of the transfer function on the complex plane

4 Delayed Differential Equations

At the start of the semester, I read a few papers and learned about delayed differential equations. Here I will give a brief overview of what I learned.

In undergraduate differential equations and controls courses, we typically look at linear Ordinary Differential Equations with the state space form:

$$\dot{x} = Ax + Bu$$

We can expand this to be able to include a time-delayed version of the state. We also omit the input u for simplicity:

$$\dot{x}(t) = Ax(t) + Bx(t - \tau)$$

We can then take the Laplace transform of the system:

$$SX(S) = AX(S) + BX(S)e^{-\tau s}$$

From this we can rearrange our equation to find the eigenvalues of the system:

$$X(S)(SI - A - Be^{S\tau}) = 0$$
$$\det\big(\lambda I - A - Be^{\lambda\tau}\big) = 0$$

This equation has infinitely many solutions, meaning that our delay systems have an infinite number of modes whose locations are determined by the Lambert W function.

4.1 Delayed Differential Equations for Command Shaping

One use case for Delayed Differential Equations that I observed this semester was in command shaping. Given a strongly oscillatory plant, how can we command it to a certain value without stimulating oscillatory modes? In ECE 4550, we discuss this and apply it to the problem of moving a pendulum in its downward position without swinging it.

To perform this action, first consider a plant whose step response contains an oscillation $x(t) = u(t)\sin(\pi t)$. If we shape our input to contain an initial step and a step 1 second in the future, we will get a response:

$$x(t) = u(t)\sin(\pi t) - u(t-1)\sin(\pi t - \pi)$$

For $t > 1$, we see that the oscillations cancel each other out. From a frequency domain perspective, we can see that our time delay shaping filter has the frequency response:

$$h(t) = \sum \alpha_i \delta(t - (i-1)\tau)$$
$$H(S) = \frac{1+e^{-(s+\alpha)\tau}}{1+e^{-a\tau}}$$

We can strategically place the zeros of these transfer functions over or near the stable oscillatory poles to help damp them out in the final response of the system.

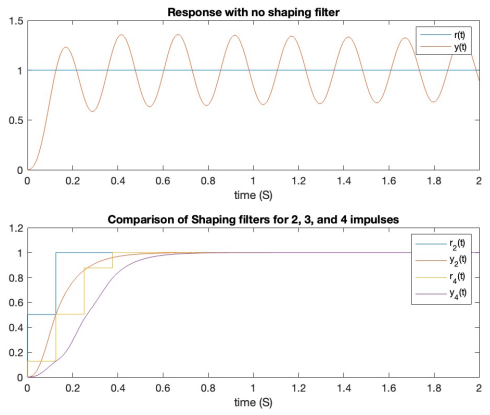


Figure 4: Using 2-step and 4-step delay systems to suppress oscillations in an inverted pendulum plant

4.2 Connection to PDEs

Delayed differential equations and partial differential equations have many similarities. First, many real-world delay systems come from systems that can be described by PDEs. The example most commonly seen in Electrical Engineering is the transmission line. These can be described by the wave equation. In lossless nondispersive media, the voltage at the end of a transmission line is simply a delayed version of the voltage seen at the input plus the reflected wave. Similarly, in physics, gravity does not travel instantaneously due to the nature of Einstein's General Relativity equations. This can be approximated by adding delays to Newtonian Gravitation as seen in Dr. Verriest's paper.

Another similarity between delayed differential equations and partial differential equations is that they can both be described by infinite-dimensional ordinary differential equations. For the delayed case, we can have a state variable for x(t) and also $x(t-\tau)$ for every τ from zero to the maximum delay that we see in the equation. For the partial differential equation version, we can describe each point in space as a different state variable. Both of these cases will lead to an uncountable infinite number of state variables.

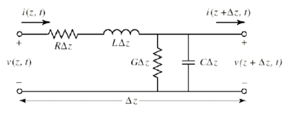


Figure 5: Discrete element model of transmission lines from Dr. Tenzeris' class on electromagnetic applications ECE 4350

4.3 Simulation of Transmission Lines with FDTD

This semester, I was interested in simulating transmission lines and the wave equation.

To start off, I want to derive a finite-dimensional state space model for a transmission line. To start, we can model transmission lines as a cascade of an infinite number of resistors, inductors, and capacitors, or we can use the Telegrapher's equations, which describe transmission lines as a partial differential equation:

$$\frac{\partial v(z,t)}{\partial z} = -Ri(z,t) - L\frac{\partial i(z,t)}{\partial z}$$
$$\frac{\partial i(z,t)}{\partial z} = -Gv(z,t) - L\frac{\partial v(z,t)}{\partial z}$$

R is the resistance per unit length, G is the conductance per unit length, L is the inductance per unit length, and C is the capacitance per unit length. We can expand out our spatial derivatives as limits using the Yee Lattice. The Yee lattice discretizes our problem in which voltages/electric fields and currents/magnetic fields are defined at different alternating points in space (Oskooi et.al). This allows us to define our derivative between our 2 end points, giving us second-order accuracy with respect to Δz. Re-arranging the equation and expanding out the derivative in this way:

$$\frac{\partial i}{\partial t}(t, z + \tfrac{\Delta z}{2}) =$$
$$-\frac{R}{L}i(t, z + \tfrac{\Delta z}{2}) - \frac{1}{L}\lim_{\Delta z \to 0}(\frac{v(t,z+\Delta z)-v(t,z)}{\Delta z})$$
$$\frac{\partial v}{\partial t}(t,z) =$$
$$-\frac{G}{C}v(t,z) - \frac{1}{C}\lim_{\Delta z \to 0}(\frac{i(t,z+\frac{\Delta z}{2})-i(t,z-\frac{\Delta z}{2})}{\Delta z})$$

If we take Δz to be non-zero zero we now have a finite-dimensional continuous-time system. To derive the A matrix for our system, consider the following setup of our state vector:

$$\vec{x}(t) = \begin{bmatrix} v(t,0) \\ v(t,\Delta z) \\ ... \\ v(t,N\Delta z) \\ i(t,\frac{\Delta z}{2}) \\ i(t,\frac{\Delta z}{2}+\Delta z) \\ ... \\ i(t,\frac{\Delta z}{2}+N\Delta z) \end{bmatrix}$$

We can now define our state space A matrix as a partitioned matrix:

$$\dot{\vec{x}}(t) = Ax(t)$$
$$A = \begin{bmatrix} \mathcal{G} & \mathcal{C} \\ \mathcal{L} & \mathcal{R} \end{bmatrix}$$

$\mathcal{G}$ represents the conductance of the system, $\mathcal{C}$ represents the capacitance of the system, $\mathcal{L}$ represents the inductance of the system, and $\mathcal{R}$ represents the resistance of the system. We define these submatrices as follows:

$$\mathcal{G} = -\frac{G}{C}\mathcal{I}_N$$
$$\mathcal{R} = -\frac{R}{L}\mathcal{I}_N$$
$$\mathcal{C} = \frac{1}{C\Delta z}\begin{bmatrix} 0 & 0 & 0 & ... & 0 & 0 & 0 \\ -1 & 1 & 0 & ... & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & ... & 0 & 0 \\ & & & ... & & & \\ 0 & 0 & 0 & ... & 0 & -1 & 1 \end{bmatrix}$$
$$\mathcal{L} = \frac{1}{C\Delta z}\begin{bmatrix} -1 & 1 & 0 & ... & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & ... & 0 & 0 \\ & & & ... & & & \\ 0 & 0 & 0 & ... & 0 & -1 & 1 \\ 0 & 0 & 0 & ... & 0 & 0 & 0 \end{bmatrix}$$

Where $\mathcal{I}_N$ is the NxN identity matrix. Now, we can use the forward Euler method to discretize in time to get a discrete-time system of the form:

$$\vec{x}(t+\Delta t) = A_d\vec{x}(t)$$
$$A_d = \begin{bmatrix} \mathcal{G}_d & \mathcal{C}_d \\ \mathcal{L}_d & \mathcal{R}_d \end{bmatrix}$$
$$\mathcal{G}_d = \Delta t\mathcal{G} + I_N, \mathcal{R}_d = \Delta t\mathcal{R} + \mathcal{I}_N, \mathcal{C}_d = \Delta t\mathcal{C},$$
$$\text{and } \mathcal{L}_d = \Delta t\mathcal{L}$$

Simulating this system with Python, we observe that this is unstable: Initially, I thought that this instability would need to be solved with filtering; however, after further investigation, this error comes from the fact that we used the staggered Yee discretization in space but not time. To solve this, we can implement an algorithm that first calculates the current/magnetic field at a half-time step in the future and then calculates the voltage/electric field:

$$\vec{i}(t+\tfrac{\Delta t}{2}) = \mathcal{L}_d\vec{v}(t) + \mathcal{R}_d\vec{i}(t-\tfrac{\Delta t}{2})$$
$$\vec{v}(t+\Delta t) = \mathcal{G}_d\vec{v}(t) + \mathcal{C}_d\vec{i}(t+\tfrac{\Delta t}{2})$$

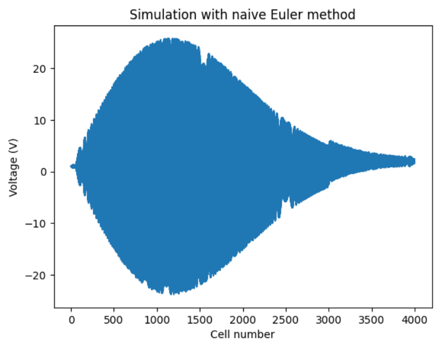


Figure 6: Visualization of instabilities in forward Euler

We can substitute the current update equation into the voltage equation:

$\vec{v}(t+\Delta t) = \mathcal{G}_d\vec{v}(t) + \mathcal{C}_d(\mathcal{L}_d\vec{v}(t) + \mathcal{R}_d\vec{i}(t-\frac{\Delta t}{2}))$

We can now formulate the stable discrete A matrix A_{Yee}:

$$x(t+\Delta t) = A_{Yee}x(t)$$
$$A_{Yee} = \begin{bmatrix} \mathcal{G}_d + \mathcal{C}_d\mathcal{L}_d & \mathcal{C}_d\mathcal{R}_d \\ \mathcal{L}_d & \mathcal{R}_d \end{bmatrix}$$

Simulating with the following update, we can now see that the simulation is stable:

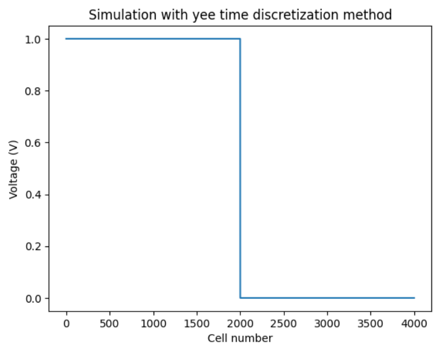


Figure 7: A stable simulation of a step function propagating down a transmission line

To explore why forward Euler is unstable but the Yee discretization is stable, we can look at the eigenvalues of the discrete A matrix. In the figure below, we observe that the Yee discretization has all of its eigenvalues on the unit circle corresponding to pure undamped oscillatory behavior. The forward Euler discretization has a line of Eigenvalues that are mostly outside of the unit circle, leading to instability.

5 Modeling the Watts Regulator

The Watts Regulator, also known as the Watts Governor, is a device that was used for control before modern electrical control systems. It consists of 2 masses spinning around a shaft. The angular velocity can be seen as a control input, and the output of the system is the height of a sleeve. This effectively gives us a mechanical feedback system that could be used to control fluid flow. This system was often used to control plants such as steam engines. As the engine speeds up, it spins the masses, which raise the sleeve and would restrict the flow of steam. This acts as a sort of negative feedback loop that would prevent the engine from spinning out of control. Let's now derive the equations governing the Watts Regulator.

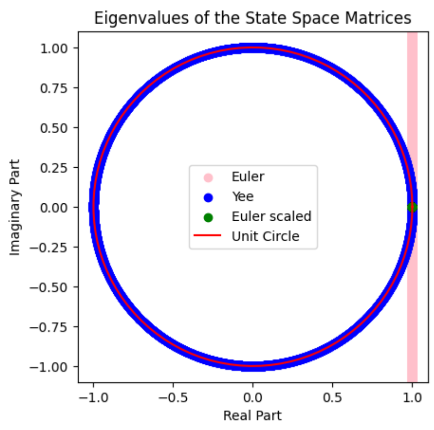


Figure 8: Plot of the eigenvalues of the different time discretization methods

ϕ is the angle that the lever makes from the vertical.

l is the length of the lever. m is the mass of 1 of the 2 or more spinning masses.

ω is the angular velocity of the shaft.

g is the force of gravity.

h is the vertical distance of the masses from the top of the shaft.

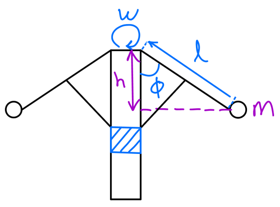


Figure 9: The setup of the watt regulator problem

First, the height of the masses can be described by:

$$h = l\cos(\phi)$$

Next, there are 3 forces acting on the mass: gravity, the centripetal force caused by the balls spinning, and the force from the rod keeping the mass in place. The gravitational force is: $F_g = mg$. The centripetal force is:

$$F_c = m\omega^2 \sin(\phi)$$

We will simply model the forces from the rod as counteracting the other forces to keep the ball in place. Now we can break the forces into their components parallel with the lever and their components parallel to the lever:

$$F_c^\perp = m\omega^2 \sin(\phi)\cos(\phi)$$
$$F_g^\perp = mg\sin(\phi)$$

These forces represent the torques that can either move the spinning masses up or down. We can take the moment of inertia in the vertical plane to be $J = ml$. Putting these all together, we get the dynamics:

$$\ddot{\phi} = \frac{1}{ml}(m\omega^2 l \sin(\phi)\cos(\phi) - mg\sin(\phi))$$

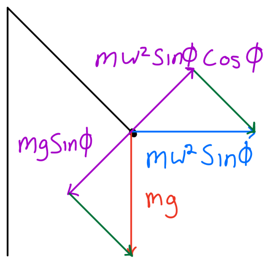


Figure 10: Diagram of forces for the Watts Regulator

5.1 Case Without Dynamics

The simplest way to analyze this system is to assume steady state or that the controller transitions settle much faster than the dynamics of the system that we are trying to control:

$$\ddot{\phi} = \dot{\phi} = 0$$

From this, we can simplify our original equation and solve for h:

$$m\omega^2 \sin(\phi)(l\cos(\phi)) = mg\sin(\phi)$$
$$h = \frac{g}{\omega^2}$$

We can then relate the height of the sleve to the negative of h, giving us a nonlinear gain from ω to u:

$$u = 2l - \frac{g}{\omega^2}$$

This gain can be tuned or adjusted by changing gear ratios of other aspects of the regulator's geometry.

5.2 Dynamical Model

If we choose to model the transients of the system, then we wind up with a nonlinear system described by:

$$\ddot{\phi} = \frac{1}{ml}(m\omega^2 l \sin(\phi)\cos(\phi) - mg\sin(\phi))$$

These nonlinear dynamics are discussed in many papers, including Pontryagin's book on Differential Equations (Pontryagin 1962).

References

A. Oskooi, D. Roundy, M. Ibanescu, P. Bermel, J.D. Joannopoulos, and S.G. Johnson 2010 "MEEP: A flexible free-software package for electromagnetic simulations by the FDTD method," Computer Physics Communications, Vol. 181, pp. 687-702

L. S. Pontryagin. Translated by Leonas Kacinskas, and Walter B. Counts 1962 Ordinary Differential Equations Library of Congress

Table of Contents

Define the domain that we will plot on

```
minmax = 2.5;
sigma = linspace(-minmax, minmax, 100);  % Real part
w = linspace(-minmax, minmax, 100);      % Imaginary part
[sigma_grid, w_grid] = meshgrid(sigma, w);
s = sigma_grid + 1j * w_grid;

x = sigma_grid;
y = w_grid;
```

Define the system

```
system_zeros = [(-1+1i)/2, (-1-1i)/2];  % Two complex conjugate zeros
system_poles = [-2/2, 3/2];        % Two real system_poles
K = 0.5;                     % System gain
```

Calculate the transfer function

H(s) = PI(system_zeros) / PI(system_poles)

```
H = K * ones(size(s));

% Add system_zeros to transfer function
for i = 1:length(system_zeros)
    H = H .* (s - system_zeros(i));
end

% Add system_poles to transfer function
for i = 1:length(system_poles)
    H = H ./ (s - system_poles(i));
end

mag_H = abs(H);
logmag_H = log(mag_H);
phase_H = angle(H);
phase_H_normalized  = (phase_H + pi) / (2*pi);
```

Calculate the electric potential and field

```
voltage = zeros(length(x), length(y));
k = 8.99e9;  % Coulomb's constant in N□m²/C²
```

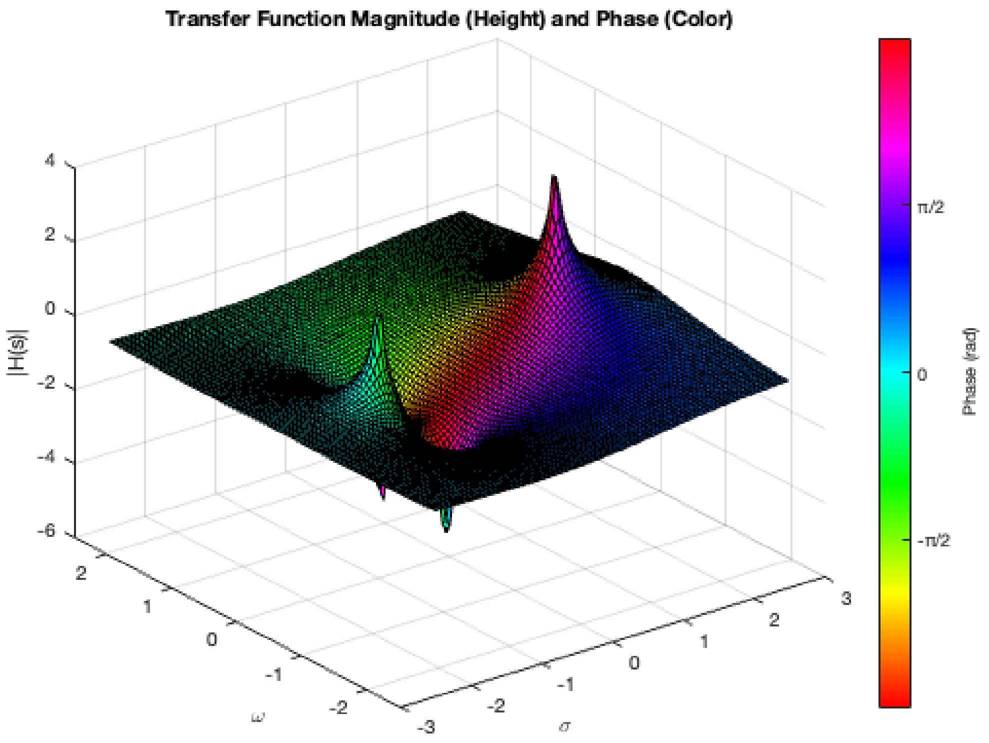
```
charge_mult = 8.854e-12

charges_negative = -1 * ones(1, length(system_zeros)) * charge_mult;
posititons_negative = zeros(length(system_zeros), 2);

for i = 1:length(charges_negative)
    posititons_negative(i, 1) = real(system_zeros(i));
    posititons_negative(i, 2) = imag(system_zeros(i));
end

charges_pos = ones(1, length(system_poles)) * charge_mult;
posititons_pos = zeros(length(system_poles), 2);

for i = 1:length(charges_pos)
    posititons_pos(i, 1) = real(system_poles(i));
    posititons_pos(i, 2) = imag(system_poles(i));
end

charges = [charges_pos, charges_negative];
positions = vertcat(posititons_pos, posititons_negative);

for i = 1:length(charges)
    r = sqrt((x - positions(i,1)).^2 + (y - positions(i,2)).^2);
    voltage = voltage + charges(i) ./ r;
end
```

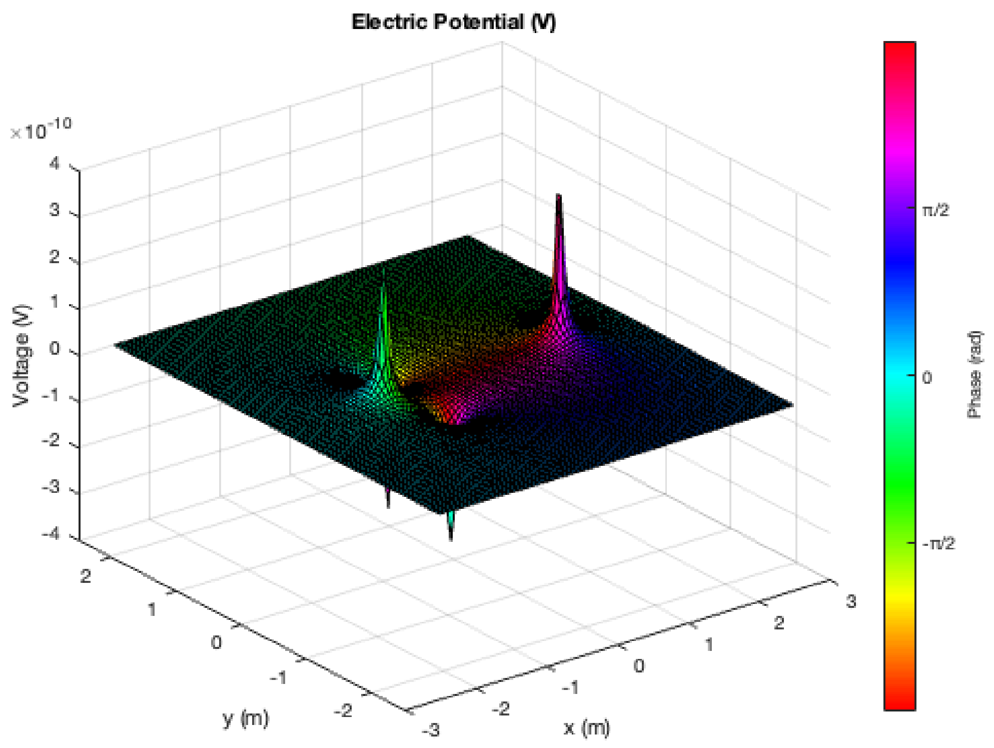
```
charge_mult =

   8.8540e-12
```

Polt

```
figure

% subplot(2,2,1)
surf(sigma, w, logmag_H, phase_H_normalized);

% Customize the colormap for phase
colormap(hsv);  % HSV colormap is circular, good for phase
c = colorbar;
c.Label.String = 'Phase (rad)';

% Convert colorbar ticks from [0,1] back to [-π,π]
c.Ticks = 0:0.25:1;
c.TickLabels = {'-π', '-π/2', '0', 'π/2', 'π'};

% Add labels and title
xlabel('\sigma');
ylabel('\omega');
zlabel('|H(s)|');
title('Transfer Function Magnitude (Height) and Phase (Color)');
```

```
% Surface plot with colormap
figure
% subplot(2,2,2);
surf(x, y, voltage, phase_H_normalized); % give the same coloring for clarity

% Customize the colormap for phase
colormap(hsv);  % HSV colormap is circular, good for phase
c = colorbar;
c.Label.String = 'Phase (rad)';
% Convert colorbar ticks from [0,1] back to [-π,π]
c.Ticks = 0:0.25:1;
c.TickLabels = {'-π', '-π/2', '0', 'π/2', 'π'};title('Electric Potential
(V)');

xlabel('x (m)');
ylabel('y (m)');
zlabel('Voltage (V)');
% zlim([-5, 5])
```





Published with MATLAB® R2024b

Table of Contents

Deffine our problem paramiters

```
target = 22; % target temperature of the house
ambient = 10; % ambient temperature outside
tf = 10; % the time gap

% paramiters from physics
alpha = 2;
beta = 1;

% calculate paramiters
Tc = target - ambient;
```

Initialize simulation

```
h = 0.001;
t = 0:h:tf;

lambda_f = (Tc * (exp(-alpha * tf) - 1) * 4 * alpha) / (beta^2 * exp(-2 *
alpha * tf) * (exp(2 * alpha * tf) - 1));

% initialize state variables
x = NaN(1, length(t));
u = -(beta / 2) * lambda_f * exp(alpha * (t - tf));

x(:, 1) = Tc;
```

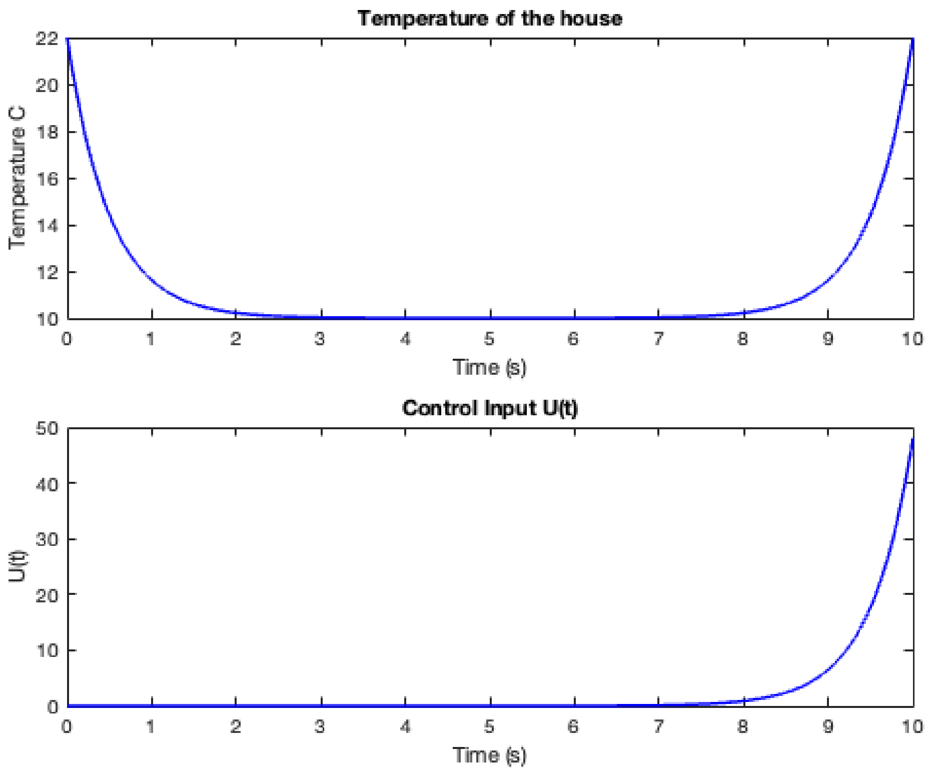
Run sim forward euler

```
for i = 1:length(t) - 1
    x(i+1) = x(i) + h*(-alpha * x(i) + beta * u(i));
end
y = x + ambient;
```

Plot

```
figure

subplot(2,1,1)
plot(t, y, 'b', 'LineWidth',1.5)
title("Temperature of the house")
```

```
xlabel("Time (s)")
ylabel("Temperature C")

subplot(2,1,2)
plot(t, u, 'b', 'LineWidth',1.5)
title("Control Input U(t)")
xlabel("Time (s)")
ylabel("U(t)")
```



Published with MATLAB® R2024b

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        import matplotlib.animation as animation
        import scipy.sparse as sp
```

```
In [2]: # Define constants for a typical coax transmission line
        L =  0.14e-6 # inductance per unit length (H/m)
        C = 80.3e-12 # capacitance per unit length (F/m)
        R = 2.08e-3 # resistance per unit length (Ohm/m)
        G = 0 # conductance per unit length (S/m)
```

```
In [3]: # Calculation of transmission line parameters
        Z0 = np.sqrt(L / C)  # characteristic impedance (Ohm)
        print(f"Characteristic Impedance (Z0): {Z0:.2f} Ohm")
        # Calculate the propagation velocity
        v = 1 / np.sqrt(L * C)  # propagation velocity (m/s)
        print(f"Propagation Velocity (v): {v:.2f} m/s")
```

```
Characteristic Impedance (Z0): 41.75 Ohm
Propagation Velocity (v): 298248459.76 m/s
```

```
In [4]: line_length = 20e-6
        dz = 5e-9 # dz
        N_cells = int(line_length/dz) # jmax
        print(f"The domain is discretized into {N_cells} cells")
```

```
The domain is discretized into 4000 cells
```

```
In [5]: dt = dz/v # time step (s)
        simulation_time = 1e-13
        N_time = int(simulation_time/dt)
        print(f"The simulation is run for {N_time} time steps")
        print(f"Time step (dt): {dt:.2e} s")
```

```
The simulation is run for 5964 time steps
Time step (dt): 1.68e-17 s
```

```
In [6]: state = np.zeros(N_cells * 2) # State vector consisting of alternating Ex ar
        print(f"state vector shape: {state.shape}")
```

```
state vector shape: (8000,)
```

```
In [7]: # Make sub arrays for state space analysis
        # See paper for more details on the derivation of the state space model
        Gm = (1 - dt*G/C) * np.eye(N_cells)
        Rm = (1 - dt*R/L) * np.eye(N_cells)
        # Gm = np.eye(N_cells)
        # Rm = np.eye(N_cells)

        Cm = np.zeros((N_cells, N_cells))
        for i in range(1, N_cells):
            Cm[i, i-1] = -1
            Cm[i, i] = 1

        Cm = dt / (C * dz) * Cm
```

```
Lm = np.zeros((N_cells, N_cells))
for i in range(0, N_cells-1):
    Lm[i, i+1] = 1
    Lm[i, i] = -1

Lm = dt / (L * dz) * Lm

print(Cm.shape)
print(Lm.shape)
print(Gm.shape)
print(Rm.shape)
```

```
(4000, 4000)
(4000, 4000)
(4000, 4000)
(4000, 4000)
```

In [8]:
```
# Naive approach with simple Eulers method
scale_dt = 1e3
Ad_euler = np.block([[Gm, Cm], [Lm, Rm]])
# Make a new matrix with scaled dt for demonstration purposes
Ad_euler_scaled = ((Ad_euler - np.eye(2 * N_cells)) / scale_dt) + np.eye(2 *

# Convert to sparse matrix for faster computation
Ad_euler = sp.csr_matrix(Ad_euler)
Ad_euler_scaled = sp.csr_matrix(Ad_euler_scaled)

print(f"matrix shape: {Ad_euler.shape}")
```

```
matrix shape: (8000, 8000)
```

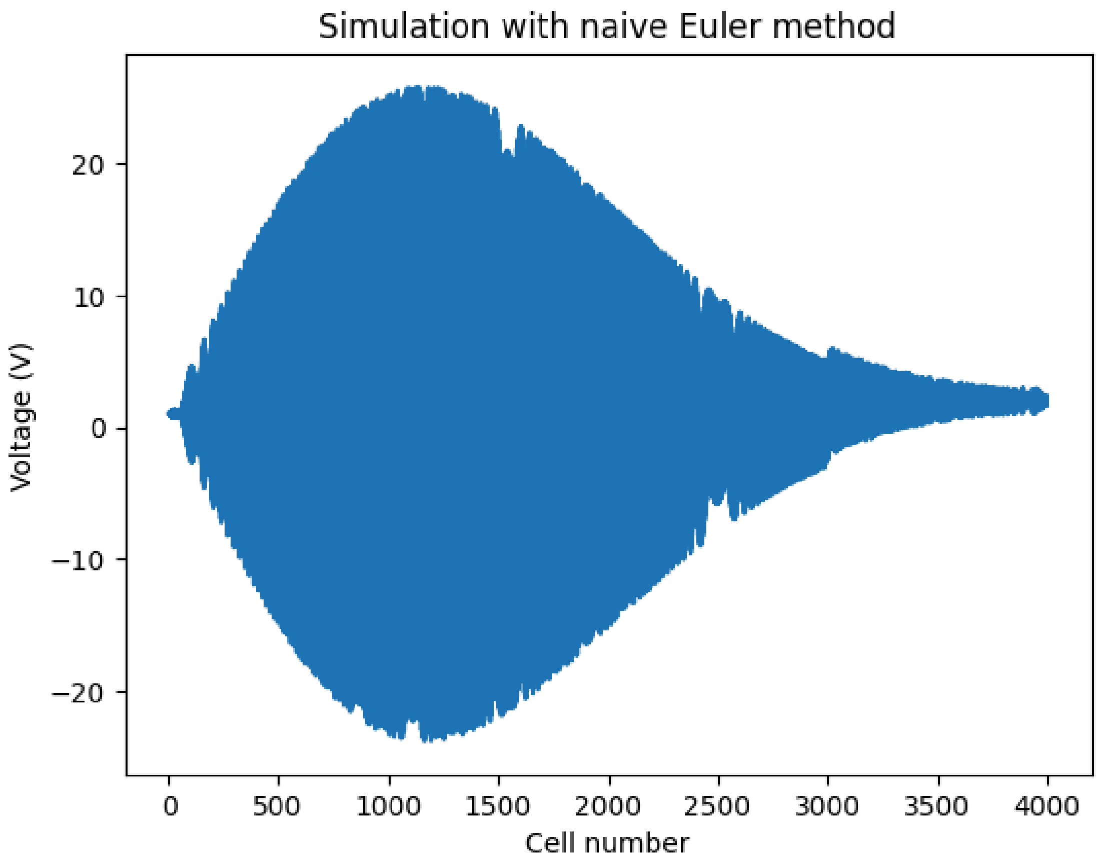
In [9]:
```
def run_simulation(Ad, state, N_time, save_every=100, source = lambda t: 1):
    "Run a simulation with a given state space matrix"
    V_movie = []
    for n in range(N_time):
        # Update state vector
        state = Ad @ state

        # add source (for now just a step function)
        state[0] = source(n * dt)

        # Save the electric field
        if n % save_every == 0:
            V = state[0:N_cells]
            print(f"step: {n})")
            print(np.min(V),np.max(V))
            V_movie.append(V)
    return V_movie
```

In [10]:
```
# Run simulation with naive forward Euler method
# Reset state vector
state = np.zeros(N_cells * 2) # State vector consisting of alternating Ex ar
euler_movie = run_simulation(Ad_euler_scaled, state, N_time * int(scale_dt),
print(f"euler_movie shape: {np.array(euler_movie).shape}")
step: 0)
0.0 1.0
step: 100000)
0.0 1.2594967523054121
step: 200000)
0.0 1.274893315541931
step: 300000)
0.0 1.2778197745442266
step: 400000)
0.0 1.2766236525027395
step: 500000)
0.0 1.280391066969833
step: 600000)
0.0 1.2827875331040985
step: 700000)
0.0 1.281585115758393
step: 800000)
0.0 1.2846088924389314
step: 900000)
0.0 1.2844871717141706
step: 1000000)
0.0 1.2849716620080245
step: 1100000)
0.0 1.286959287319911
step: 1200000)
0.0 1.2852961932077467
step: 1300000)
0.0 1.2873735713067012
step: 1400000)
0.0 1.2885326070216319
step: 1500000)
0.0 1.287442998833998
step: 1600000)
0.0 1.2883399737091659
step: 1700000)
0.0 1.2898975006193223
step: 1800000)
0.0 1.2898406815074526
step: 1900000)
0.0 1.2885003991105697
step: 2000000)
0.0 1.2903806110397278
step: 2100000)
0.0 1.291414290143484
step: 2200000)
0.0 1.291395904226017
step: 2300000)
0.0 1.2905044084817363
step: 2400000)
0.0 1.3001231146233467
step: 2500000)
0.0 1.3503637069816294
step: 2600000)
0.0 1.409461557615731
step: 2700000)
0.0 1.4792792805158768
```

```
step: 2800000)
0.0 1.5637355208604957
step: 2900000)
0.0 1.6626097236722204
step: 3000000)
0.0 1.7781293759068928
step: 3100000)
0.0 1.9192695976633551
step: 3200000)
-0.08325637873927118 2.0862911481357775
step: 3300000)
-0.28338168309623735 2.2834027400911032
step: 3400000)
-0.518142346035404 2.5181771520109115
step: 3500000)
-0.7985908577318085 2.7957065266056995
step: 3600000)
-1.1308395173061396 3.129164943563412
step: 3700000)
-1.5221059335182723 3.529080600929388
step: 3800000)
-2.00314229879829 3.9966081674686444
step: 3900000)
-2.5611929261118562 4.5678112623616345
step: 4000000)
-3.2424962927592085 5.237811867052592
step: 4100000)
-4.043664072125267 6.045237294499682
step: 4200000)
-5.010057275255279 7.003506244638208
step: 4300000)
-6.164216071739437 8.15923164665627
step: 4400000)
-7.530962035991124 9.537707742836943
step: 4500000)
-9.188605752231426 11.18667642505515
step: 4600000)
-11.149133360137938 13.162755314895445
step: 4700000)
-13.500887309690143 15.509524744971788
step: 4800000)
-16.350648799101208 18.35343914412425
step: 4900000)
-19.74408652783739 21.732809275589318
step: 5000000)
-23.809609181326444 25.791745137546407
step: 5100000)
-28.67191854862715 30.673111250622064
step: 5200000)
-34.527107872212355 36.51343596135316
step: 5300000)
-41.45478914857423 43.48593302297351
step: 5400000)
-49.852554867152215 51.937401596405905
step: 5500000)
-60.000305811343516 62.037005456579884
```

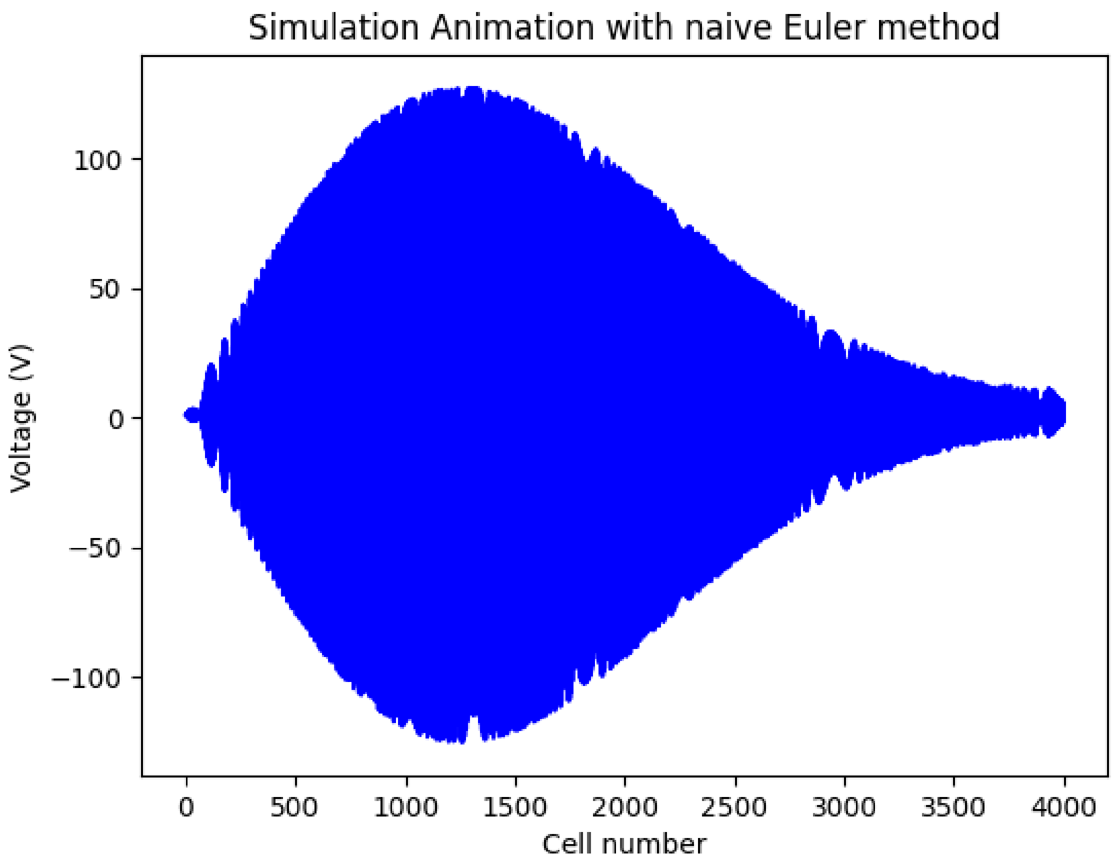
```
step: 5600000)
-72.19110956306096 74.00181455244743
step: 5700000)
-86.76213540826379 88.75016791064017
step: 5800000)
-104.18980719796514 106.29875020863778
step: 5900000)
-125.29855610069374 127.23633272819735
euler_movie shape: (60, 4000)
```

In [11]:
```
plt.plot(euler_movie[50])
plt.title("Simulation with naive Euler method")
plt.xlabel("Cell number")
plt.ylabel("Voltage (V)")
plt.show()
```



In [12]:
```
frames = [] # for storing the generated images
fig = plt.figure()
ax = fig.add_subplot(1,1,1)

for i in range(len(euler_movie)):
    im, = ax.plot(euler_movie[i],color = 'blue')
    frames.append([im])
ani = animation.ArtistAnimation(fig, frames, interval=20, blit=True,
                                repeat_delay=1000)

# plt.show()
plt.title("Simulation Animation with naive Euler method")
plt.xlabel("Cell number")
plt.ylabel("Voltage (V)")
ani.save('euler.gif', writer='imagemagick', fps=30)
```

```
MovieWriter imagemagick unavailable; using Pillow instead.
```



In [13]:
```
# Use the Yee discretization in time
Ad_yee = np.block([[Gm + Cm @ Lm, Cm @ Rm], [Lm, Rm]])

# Convert to sparse matrix for faster computation
Ad_yee = sp.csr_matrix(Ad_yee)

print(f"matrix shape: {Ad_yee.shape}")
```

```
matrix shape: (8000, 8000)
```

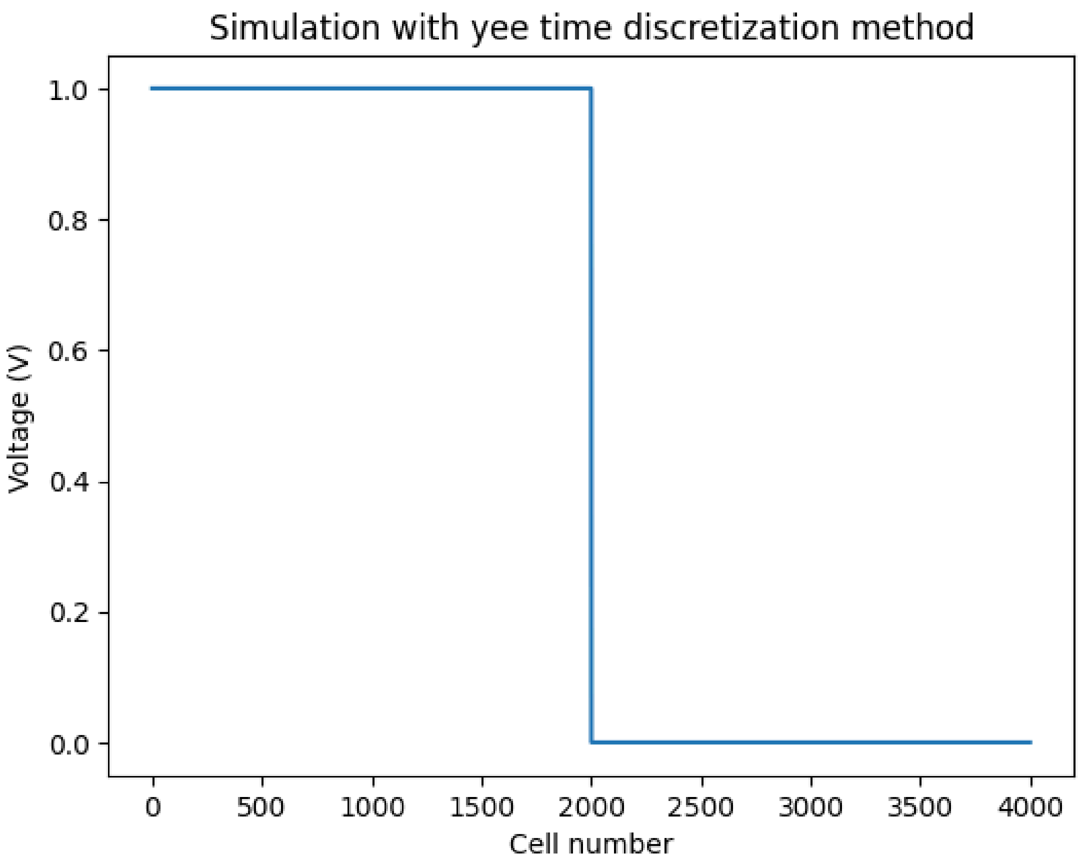
In [14]:
```
# Run simulation with yee discretization
# Reset state vector
state = np.zeros(N_cells * 2) # State vector consisting of alternating Ex ar
yee_movie = run_simulation(Ad_yee, state, N_time)
```

```
step: 0)
0.0 1.0
step: 100)
0.0 1.0000000000000762
step: 200)
0.0 1.0000000000002962
step: 300)
0.0 1.0000000000006777
step: 400)
0.0 1.0000000000011555
step: 500)
0.0 1.0000000000017923
step: 600)
0.0 1.0000000000025246
step: 700)
0.0 1.0000000000034246
step: 800)
0.0 1.000000000004442
step: 900)
0.0 1.0000000000056128
step: 1000)
0.0 1.0000000000069837
step: 1100)
0.0 1.0000000000084805
step: 1200)
0.0 1.000000000010135
step: 1300)
0.0 1.0000000000119096
step: 1400)
0.0 1.0000000000138047
step: 1500)
0.0 1.0000000000157816
step: 1600)
0.0 1.0000000000179439
step: 1700)
0.0 1.0000000000202336
step: 1800)
0.0 1.0000000000227742
step: 1900)
0.0 1.0000000000253606
step: 2000)
0.0 1.0000000000280727
step: 2100)
0.0 1.0000000000308882
step: 2200)
0.0 1.0000000000338265
step: 2300)
0.0 1.000000000036971
step: 2400)
0.0 1.0000000000402738
step: 2500)
0.0 1.0000000000437332
step: 2600)
0.0 1.0000000000472697
step: 2700)
0.0 1.0000000000509908
```

```
step: 2800)
0.0 1.0000000000548388
step: 2900)
0.0 1.000000000058736
step: 3000)
0.0 1.0000000000627707
step: 3100)
0.0 1.0000000000670046
step: 3200)
0.0 1.0000000000713878
step: 3300)
0.0 1.0000000000759313
step: 3400)
0.0 1.0000000000805802
step: 3500)
0.0 1.000000000085336
step: 3600)
0.0 1.0000000000902327
step: 3700)
0.0 1.0000000000953249
step: 3800)
0.0 1.0000000001004377
step: 3900)
0.0 1.000000000105653
step: 4000)
0.99999999900381 1.9999999990047612
step: 4100)
0.9999999990072787 1.9999999990079027
step: 4200)
0.9999999990115829 1.9999999990121895
step: 4300)
0.9999999990171261 1.9999999990177209
step: 4400)
0.999999999023832 1.9999999990244428
step: 4500)
0.9999999990316226 1.9999999990322046
step: 4600)
0.9999999990404616 1.999999999041076
step: 4700)
0.999999999050398 1.9999999990510036
step: 4800)
0.9999999990614663 1.9999999990620645
step: 4900)
0.999999999073718 1.9999999990743291
step: 5000)
0.9999999990871594 1.9999999990877515
step: 5100)
0.9999999991017412 1.9999999991023532
step: 5200)
0.9999999991174214 1.9999999991180184
step: 5300)
0.9999999991340572 1.9999999991346689
step: 5400)
0.9999999991517858 1.999999999152387
step: 5500)
0.9999999991707339 1.9999999991713233
```

```
step: 5600)
0.9999999991907423 1.9999999991913213
step: 5700)
0.9999999992117832 1.999999999212384
step: 5800)
0.9999999992338815 1.9999999992344775
step: 5900)
0.9999999992572154 1.9999999992578175
```

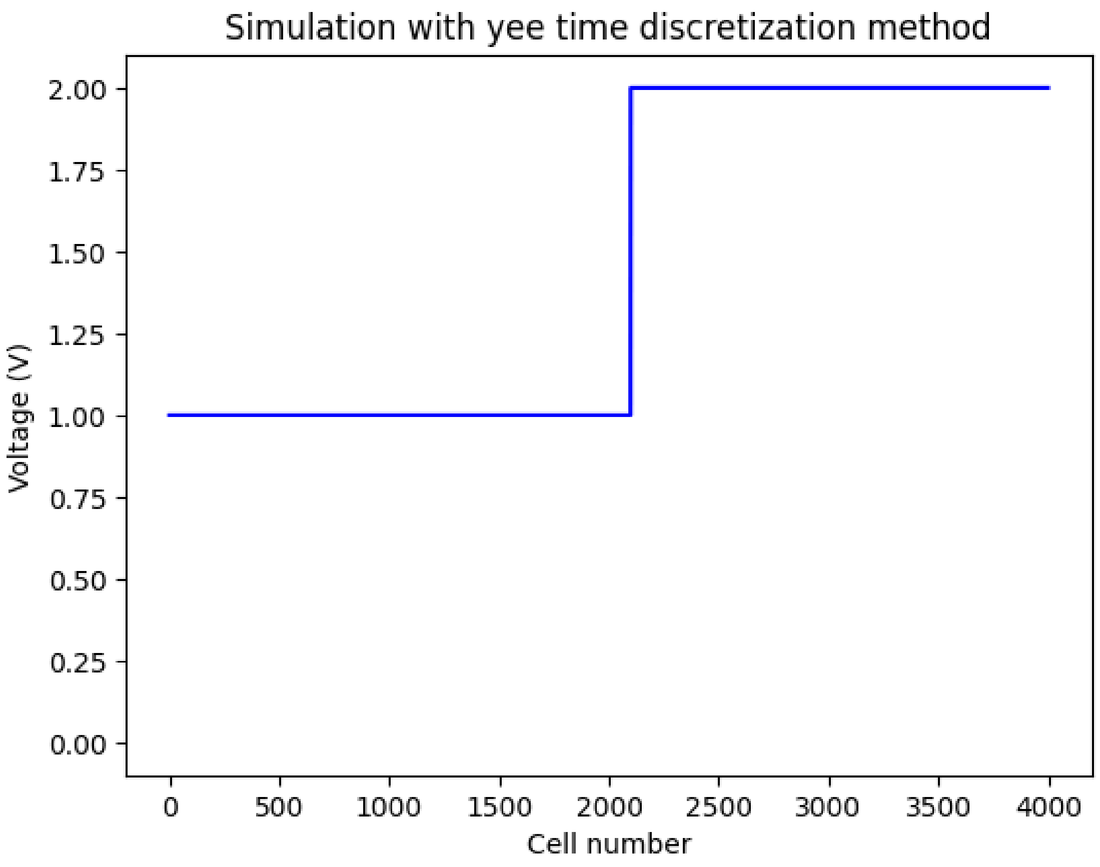
In [15]:
```python
plt.plot(yee_movie[20])
plt.title("Simulation with yee time discretization method")
plt.xlabel("Cell number")
plt.ylabel("Voltage (V)")
plt.show()
```



In [16]:
```python
frames = [] # for storing the generated images
fig = plt.figure()
ax = fig.add_subplot(1,1,1)

for i in range(len(yee_movie)):
    im, = ax.plot(yee_movie[i],color = 'blue')
    frames.append([im])
ani = animation.ArtistAnimation(fig, frames, interval=20, blit=True,
                                repeat_delay=1000)


plt.title("Simulation with yee time discretization method")
plt.xlabel("Cell number")
plt.ylabel("Voltage (V)")
```

```
# plt.show()
ani.save('yee.gif', writer='imagemagick', fps=30)
```

```
MovieWriter imagemagick unavailable; using Pillow instead.
```



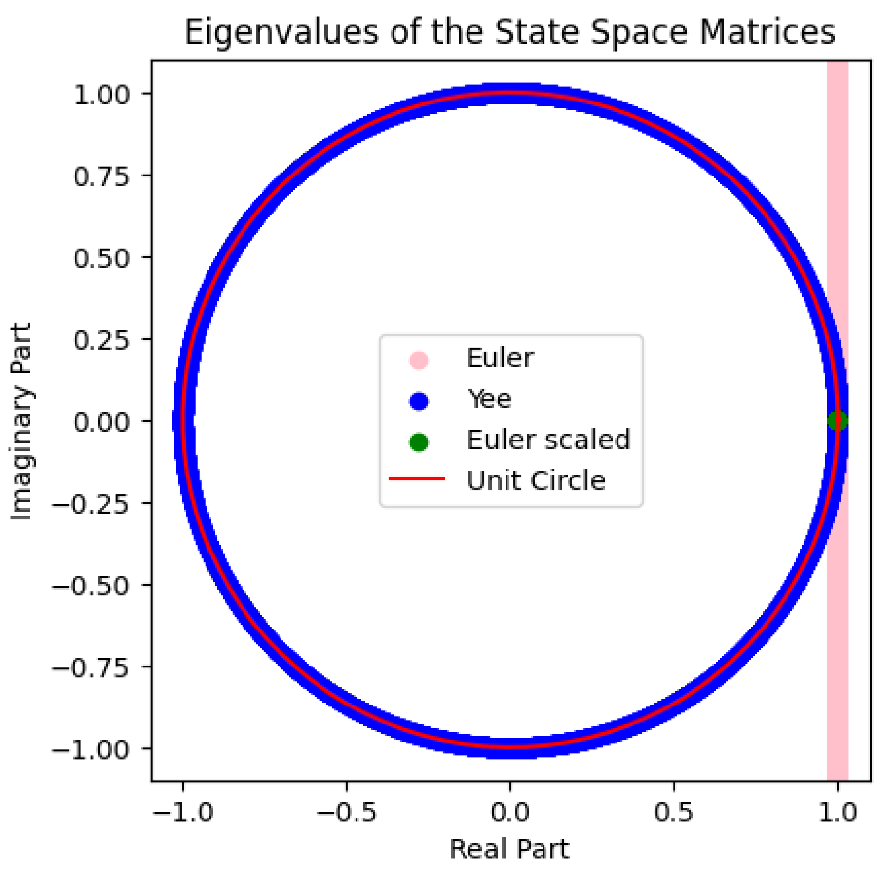
In [17]:
```
Ad_euler = np.array(Ad_euler.todense())
Ad_euler_scaled = np.array(Ad_euler_scaled.todense())
Ad_yee = np.array(Ad_yee.todense())

eigvals_euler, _ = np.linalg.eig(Ad_euler)
eigvals_yee, _ = np.linalg.eig(Ad_yee)
eigvals_euler_scaled, _ = np.linalg.eig(Ad_euler_scaled)
```

In [18]:
```
# Sort eigenvalues in descending order
eigvals_euler = np.sort(eigvals_euler)[::-1]
eigvals_yee = np.sort(eigvals_yee)[::-1]
eigvals_euler_scaled = np.sort(eigvals_euler_scaled)[::-1]
```

In [19]:
```
# Print first 10 eigenvalues
print("Eigenvalues of Euler matrix:")
for i in range(10):
    print(f"Eigenvalue {i}: mag: {np.abs(eigvals_euler[i])} angle: {np.angle
print("============================")
print("Eigenvalues of Yee matrix:")
for i in range(10):
    print(f"Eigenvalue {i}: mag: {np.abs(eigvals_yee[i])} angle: {np.angle(e
Eigenvalues of Euler matrix:
Eigenvalue 0: mag: 1.908685509861771 angle: 1.0193487454823402
Eigenvalue 1: mag: 1.908685509861771 angle: -1.0193487454823402
Eigenvalue 2: mag: 1.6443185318583764 angle: 0.9170624315746728
Eigenvalue 3: mag: 1.6443185318583764 angle: -0.9170624315746728
Eigenvalue 4: mag: 1.1974374353318773 angle: 0.5824514211265115
Eigenvalue 5: mag: 1.1974374353318773 angle: -0.5824514211265115
Eigenvalue 6: mag: 1.9928680868081552 angle: 1.0451301419874828
Eigenvalue 7: mag: 1.9403851982098923 angle: 1.029366792932062
Eigenvalue 8: mag: 1.9403851982098923 angle: -1.029366792932062
Eigenvalue 9: mag: 1.9928680868081552 angle: -1.0451301419874828
===========================
Eigenvalues of Yee matrix:
Eigenvalue 0: mag: 1.0 angle: 0.0
Eigenvalue 1: mag: 0.999999999999751 angle: 0.0
Eigenvalue 2: mag: 0.999999999999886 angle: 0.00039274817522138965
Eigenvalue 3: mag: 0.999999999999886 angle: -0.00039274817522138965
Eigenvalue 4: mag: 0.9999999999998801 angle: 0.0011782445256658637
Eigenvalue 5: mag: 0.9999999999998801 angle: -0.0011782445256658637
Eigenvalue 6: mag: 0.9999999999998768 angle: 0.001963740876101948
Eigenvalue 7: mag: 0.9999999999998768 angle: -0.001963740876101948
Eigenvalue 8: mag: 0.9999999999998863 angle: 0.0027492372265442854
Eigenvalue 9: mag: 0.9999999999998863 angle: -0.0027492372265442854
```

In [20]:
```python
# Plot eigenvalues
plt.figure()
plt.scatter(np.real(eigvals_euler), np.imag(eigvals_euler), color='pink', la
plt.scatter(np.real(eigvals_yee), np.imag(eigvals_yee), color='blue', label=
plt.scatter(np.real(eigvals_euler_scaled), np.imag(eigvals_euler_scaled), co

# Draw unit circle
theta = np.linspace(0, 2 * np.pi, 100)
plt.plot(np.cos(theta), np.sin(theta), 'red', label='Unit Circle')

# Set aspect ratio and limits so the unit circle isn't stretched
plt.gca().set_aspect('equal', adjustable='box')
plt.xlim([-1.1, 1.1])
plt.ylim([-1.1, 1.1])
plt.xlabel('Real Part')
plt.ylabel('Imaginary Part')
plt.legend()
plt.title('Eigenvalues of the State Space Matrices')
# plt.legend("Eigenvalues Euler", "Eigenvalues Yee")
```

Out[20]:
```
Text(0.5, 1.0, 'Eigenvalues of the State Space Matrices')
```


Eigenvalues of the State Space Matrices
Imaginary Part
Real Part
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.0
−0.5
0.0
0.5
1.0
Euler
Yee
Euler scaled
Unit Circle