2.1 Lyapunov analysis of networks

The equations of motion of a fully connected network

\[ \tau \dot{u}_i = -u_i + \sum_j T_{ij} V_j + I_i \]

cause the following Lyapunov energy to decrease with time:

\[ L = -\frac{1}{2} \sum_{i,j} T_{ij} V_i V_j + \sum_i G(V_i) - \sum_i V_i I_i \]

where \( G(V_h) = \int_0^V g^{-1}(z)dz \) and \( g^{-1}(V_i) = u_i \) is the inverse gain function.

- To show this, let’s take the gradient of this energy (the vector of partials with respect to each output \( V_h \)):

\[ \frac{\partial L}{\partial V_h} = -\frac{1}{2} \sum_j (T_{kj} + T_{jk}) V_j + g^{-1}(V_h) - I_h \]

where we have remembered enough calculus to realize that \( \frac{d}{dt} \int_0^t f(x)dx = f(t) \) and also we have thought hard about which terms to keep from the first double summation and which ones drop out as zero. Now we assume \( T \) is symmetric (so that \( T_{kj} = T_{jk} \)), and substitute \( u_h = g^{-1}(V_h) \),

\[ \frac{\partial L}{\partial V_h} = +u_h - \sum_j T_{kj} V_j - I_h \]

and we notice that this is exactly the negative of the expression for \( \tau \dot{u}_h \):

\[ \frac{\partial L}{\partial V_h} = -\tau \dot{u}_h \]

This means that the neural network variables \( u_i \) descend the \( V \) gradient of the energy:

\[ \dot{u}_i = -\frac{1}{\tau} \frac{\partial L}{\partial V_i} \]

careful that this is \emph{not the same as} having the \( u_i \) dynamics follow the negative \( u \) gradient of the energy as would occur in a “gradient descent” algorithm:

\[ \frac{\partial L}{\partial u_i} = \frac{\partial L}{\partial V_i} \frac{\partial V_i}{\partial u_i} = -\tau g'(u_i) \dot{u}_i \neq -\dot{u}_i \]

- Now, it’s no surprise that the differential equation always decreases with the energy. We will assume that \( g(\cdot) \) is bounded and monotonic increasing and that \( g(0) = 0 \) although it is possible to assume slightly less than this.

\[ \dot{L} = \sum_i \frac{\partial L}{\partial V_i} \dot{V}_i = -\tau \sum_i \dot{u}_i V_i = -\tau \sum_i \dot{u}_i \frac{\partial V_i}{\partial u_i} \dot{u}_i = -\tau \sum_i (\dot{u}_i)^2 g'(U_i) \leq 0 \]

(since \( g'(\cdot) \) is always positive).
Lastly we want to show that $L$ is bounded from below. We derive the bound by examining the expression for $L$ term by term:

$$L = -\frac{1}{2} \sum_{i,j} T_{ij} V_i V_j + \sum_i G(V_i) - \sum_i V_i I_i$$

The first term can never be less than $-\frac{V_m^2}{2N} \sum_{i,j} |T_{ij}|$, where $V_m$ is the value of the bound on $|g(.)|$. The second term can never be less than zero (because it is the sum of terms which are always either the positive sense integral of a positive function or the negative sense integral of a negative function); the third term can never be less than $-V_m \sum_i |I_i|$, so the entire function must be bounded below.

What good does this do? Well, if the system energy is bounded (meaning it can’t escape to negative infinity) and if the energy keeps decreasing, the system will not oscillate. For, if the system did follow a closed path in state space, then at some point it would have the same energy as before, which violates the fact that the energy always decreases. Also note from (3) that $\dot{L} = 0$ if and only if all the $\dot{u}_i = 0$. This means that $L$ can’t stop decreasing unless we are at a fixed point, and so it is also impossible to oscillate on a path that keeps $L$ constant.

What does the Lyapunov function look like? Let’s ignore the third term (assume no inputs $I_i$ to the network). Then, there are two terms which fight. The first term (shown on the left below) is known as a quadratic form. If you plot a typical slice through state space (i.e. Lyapunov energy versus some $V_i$ or $u_j$), the first term would look like a parabola, either concave upward or downward, bounded of course by $-V_m \leq V_i \leq +V_m$.

The second term $G(V_i)$ (shown in the middle below) also acts to keep $V_i$ bounded by putting up (possibly infinitely) steep walls at $V_i = -V_m$ and $V_i = +V_m$. The sum of these two terms is shown on the right below (assuming a downward pointing parabola).

Notice that the energy does not make it easy to predict where the local minima are, or which minima can be reached from where. If one uses the outer product formula to specify the $T_{ij}$ as we do, one digs deep energy wells where the memories are, but one is not guaranteed that those will be local minima, or that they will be unique or reachable. In practice, for few enough stored memories, the memories will be local minima, although usually not unique.

### 2.2 Simple Neural Network

- It is essentially the same as the two neuron network analyzed in lecture, with some sign changes: we have reciprocal negative (inhibitory) connections (but still zero self-connections). The circuit analyzed in class had stable points when both neurons were on or both were off and one has stable points when one is on and one is off. We want to find the minimum gain required to have stable points away from $(0,0)$ (remember that “gain” is defined as the slope of the neuron transfer function at zero, or, $\beta = \frac{dg(u)}{du} \bigg|_{u=0}$).

In general, we will find a fixed point for this system wherever the dynamic equations tell us that the state is stationary (we have written the equations in vector form, but they are still the same
equations in $u_1$ and $u_2$ that you are used to):

$$\tau \dot{u} = 0 = -u + TV \quad \Rightarrow \quad u = TV = Tg(u)$$

where $u = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \end{pmatrix}$ and $V = \begin{pmatrix} V_1 \\ V_2 \\ \vdots \end{pmatrix} = g(u)$. For our perfectly symmetrical 2x2 network (something we can have, almost, in a computer simulation, but not in real life), we have $T_{11} = T_{22} = 0$; $T_{12} = T_{21} = -1$ exactly; and we can simplify the fixed-point condition to the requirement that:

$$-u_1 = V_2 = g(u_2) \quad \text{and} \quad -u_2 = V_1 = g(u_1)$$

which tells us that there is either a single fixed point at the origin or three fixed points (one near the origin and two near $\pm(1, 1)$) depending on the value of the gain $\beta$ of function $g$. This situation is illustrated in the figure below for high and low gains using $g(x) = tanh(x)$:

Please see the function simhop.m in the Appendix.

- Now here we go to the plotting part, I made a little change to the plothop2.m function (now it is plothop1.m) and I made a easy.m function to do the whole job (please see the plothop1.m and easy.m in the Appendix). Then the plotting will be:
Next, we want to know something about the stability of the fixed points, that is, if we move a small distance away from the point, will the network dynamics return us to that point, or will we drift away to somewhere else? In general, a way to find this out is to linearize the dynamic equations around the fixed point and examine the resulting linear equations for stability. How do we do this? Again, the idea is essentially playing with Taylor expansions: expand and discard high order terms. We will work through a two dimensional example, but the general linearization process can be done for any size network. If we have a fixed point at \( u = u_0 = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \) then we are interested in the behavior near this point, so we linearize the dynamic equations around \( u_0 \):

\[
\begin{align*}
\tau \dot{u}_1 &= -u_1 + T_{12} g(u_2) \\
\tau \dot{u}_2 &= -u_2 + T_{21} g(u_1)
\end{align*}
\]

If we take only linear terms, assuming that \((u - u_0)\) is small, we get

\[
\begin{align*}
\tau \dot{u}_1 &= -u_1 + T_{12} \left[ g(x_2) + (u_2 - x_2) \frac{dg(u)}{du} \right]_{u=x_2} = -(u_1 - a_1) + b_1 (u_2 - x_2) \\
\tau \dot{u}_2 &= -u_2 + T_{21} \left[ g(x_1) + (u_1 - x_1) \frac{dg(u)}{du} \right]_{u=x_1} = -(u_2 - a_2) + b_2 (u_1 - x_1)
\end{align*}
\]

This is now a linear system of the form

\[
\dot{u} = Au - b
\]

whose stability you already know how to analyze from previous problems. Essentially, the linear system is unstable if any of the eigenvalues of \( A \) have real parts greater than zero. So if we do our simple 2 neuron example and we linearize about the fixed point \((0,0)\) then we get the equation:

\[
\begin{pmatrix}
\dot{u}_1 \\
\dot{u}_2
\end{pmatrix} =
\begin{pmatrix}
-1 - \beta \\
-\beta - 1
\end{pmatrix}
\begin{pmatrix}
u_1 \\
u_2
\end{pmatrix}
\]

which means that \( A \) has eigenvalues \( \lambda = -1 \pm \beta \). So in order to be stable, we must keep the eigenvalues negative which means \( \beta < 1 \).

What? Where did all this stuff about real parts less than zero come from, you ask? Don’t panic, this is stuff you know. Here are two easy ways to think about it. First, the engineer’s way: think of doing Euler simulation of the equation \( \dot{u} = Au \). We will simulate \( u_{t+1} = u_t + \eta \dot{u}_t = u_t + \eta Au_t = (I + \eta A)u_t \). This means that \( u_t = (I + \eta A)^t u_0 = M^t u_0 \). How do we raise a matrix \( M \) to the power \( t \)? We diagonalize it into the form \( M = E \Lambda E^{-1} \) (where \( D \) is diagonal with the eigenvalues of \( M \) on the diagonal) so that \( M^t = E \Lambda^t E^{-1} \). Now we don’t want any eigenvalues of \( M \) to to have magnitude larger than one, even for arbitrarily small (and positive) timesteps \( \eta \) or else \( D^t \) will blow up for some \( u_0 \). But what are the eigenvalues of the \( M = (I + \eta A) \)? They are just \(^1\) one plus \( \eta \) times the eigenvalues of \( A \). So to keep their magnitude less than one, the eigenvalues of \( A \) must have real part less than zero, no matter how small \( \eta \) is.

Another way to look at this is the mathematician’s way: notice that \( \dot{u} = Au - b \) is a linear system which (if \( A \) is symmetric) has solutions of the form \( u = e^{At} u_0 + A^{-1} b \) (where we have used the

\(^1\) Essential basic linear algebra straight from the review! If we know that \( \lambda \) are the eigenvalues of a matrix \( X \) then what are the eigenvalues of \( aX + bI \) for scalars \( a \) and \( b \) and identity matrix \( I \)? The answer is easy, let \( \psi \) be an eigenvector of \( X \) and now what is \( (aX + bI) \psi \)? It is just \((a\lambda + b)\psi \). So the eigenvectors of \( aX + bI \) are the same as those of \( X \) and the eigenvalues are just \( a\lambda + b \).
matrix exponential\(^2\)). Again we can rotate into the eigenbasis \( \mathbf{A} = \mathbf{E D E}^{-1} \) and now \( \mathbf{D} \) becomes diagonal with \( \mathbf{A} \)'s eigenvalues on the diagonal. So in each coordinate we have \( e^{\lambda_i t} \) which blows up if any eigenvalues of \( \mathbf{A} \) have real part greater than zero. This is the multidimensional extension of the basic exponential thing, which blows up if the time coefficient has a positive real part\(^3\).

For the flip-flop network in the homework, the condition for stable points away from zero turned out to be very simple: the neuron gains must be greater than one. We haven’t formally shown that when the fixed point at the origin becomes unstable then the other fixed points must be stable. But it is true and can be argued as follows: By symmetry the two fixed points away from the origin are identical and so must either both be stable or both unstable. From the Lyapunov analysis we know that the system always has some stable fixed point, therefore when the origin becomes unstable the two fixed points away from the origin must both become stable.

You will find that in the general case in which each neuron can have a different gain, that the product of the gains of the two neurons must be greater than one. This is what electrical engineers sometimes call “loop gain,” and the condition we are trying to produce is usually considered pathological, or “unstable,” since in a linear system we would go directly from stable points only at zero to no finite stable points at all. Luckily for us, the nonlinearity of our neurons gives us two new stable points away from the origin!

All of this should have been born out in the computer simulations. In particular, you should have found two different behaviours for \( \beta > 1 \) and \( \beta < 1 \). With high gain, there are two fixed points of the system near \((u_1, u_2) = (1, -1)\) and \((u_1, u_2) = (-1, 1)\) and an unstable saddle point at the origin. In the low gain case there is only one stable point, at the origin.

Appendix

simhop.m:

function [utrace,vtrace] = simhop(T,beta,uinit,eta,iterations)
% This function simulates a Hopfield network using simple Euler integration.
% T is the connection matrix (should be symmetric)
% beta is the gain of the activation function (tanh by default)
% uinit is the initial condition vector for the u's
% eta is the timestep for the Euler integration
% iterations is the number of iterations to simulate
% utrace is the u vector state of the network over time,
% one column per timestep in the simulation
% vtrace is the V vector state of the network over time,
% one column per timestep in the simulation

N = size(T,1);
utrace = zeros(N,iterations);
vtrace = zeros(N,iterations);
utrace(:,1) = uinit(:);
vtrace(:,1) = tanh(beta*utrace(:,1));
for i=2:iterations
  utrace(:,i)= (1-eta) * utrace(:,i-1) + eta * T * vtrace(:,i-1);
  vtrace(:,i)= tanh(beta * utrace(:,i));

\(^2\)Just as \( e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots \) we define \( e^M = I + M + \frac{M^2}{2!} + \frac{M^3}{3!} + \ldots \)

\(^3\)Extensive coverage of the topic of linear stability is available in control systems texts, such as one by Kailath, and another by DeSoer.
end

plothopl.m:

function plothopl(T,beta,uinit,eta,iter)

%plothopl(T,beta,uinit,eta,iter)

% $
% T$
% This function simulates a two neuron Hopfield net by calling
% sinh and then plots the resulting traces in the u and V phase planes.
% $\beta$ is the connection matrix (should be symmetric)
% $\beta$ is the gain of the activation function (tanh by default)
% $\beta$ is the initial condition vector for the u's
% $\beta$ is the timestep for the Euler integration
% $\beta$ is the number of iterations to simulate
%
[utrace,vtrace] = simhop(T,beta,uinit,eta,iter);

figure(1)
plot(utrace(1,:),utrace(2,:),'o',utrace(1,:),utrace(2,:));

figure(2)
plot(vtrace(1,:),vtrace(2,:),'o',vtrace(1,:),vtrace(2,:));

easy.m

function easy(bb,ee)

T=[0 -1; -1 0]
beta=bb
eta=ee
iter=10

figure(1);clf;
title=sprintf('U Phase Plane ( $\beta = %2.1f, \eta = %2.1f$ )',beta,eta);
title=title('FontSize',20);
axis([-1.5 1.5 -1.5 1.5]); axis('square'); grid;
xlabel('u_1','FontSize',16); ylabel('u_2','FontSize',16);
hold on;

figure(2);clf;
title2=sprintf('V Phase Plane ( $\beta = %2.1f, \eta = %2.1f$ )',beta,eta);
title2=title2('FontSize',20);
axis([-1 1 -1 1]); axis('square'); grid;
xlabel('V_1','FontSize',16); ylabel('V_2','FontSize',16);
hold on
ui=-1.5;for u1=-1.5:0.3:1.5
plothopl(T,beta,[ui;u2],eta,iter);
end

u2=1.5;for u1=-1.5:0.3:1.5
plothopl(T,beta,[ui;u2],eta,iter);
end

ui=1.5;for u2=1.5:-0.3:-1.5
plothopl(T,beta,[ui;u2],eta,iter);
end

u2=-1.5;for u1=1.5:-0.3:-1.5
plothopl(T,beta,[ui;u2],eta,iter);
end