

Introduction to Neural Networks

Daniel Kersten

Lecture 3

Introduction

Last time

1. Overview of structure of the neuron
2. Basic electrophysiology

Today

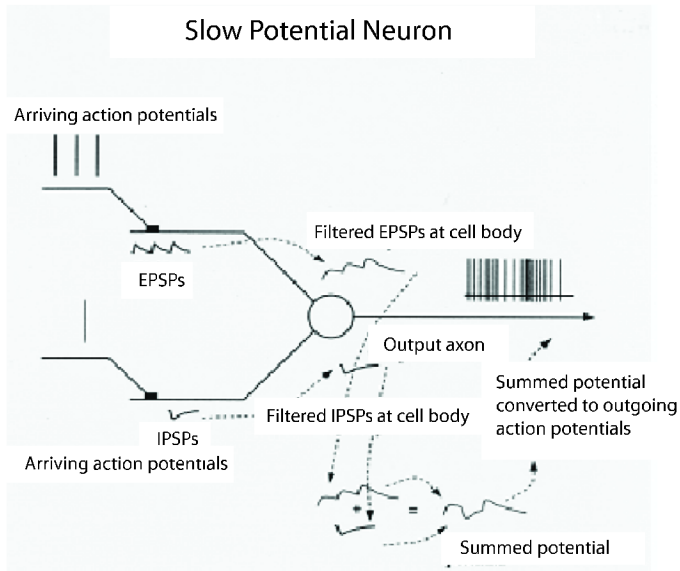
- Summarize the qualitative features of the neuron as a signal processing device
- Levels of abstraction in neural modeling
- The McCulloch-Pitts model
- Develop the "integrate and fire" model, and its relation to frequency codes
 - prepares the stage for a simpler model: a generic model that will be used for a large fraction of the course.

Our strategy will be to converge on the "right" model for large-scale simulations--we'll look at ones that are simple (McCulloch-Pitts), then complex (leaky integrate-and-fire), and then somewhere in the middle (the generic neural model).

After reviewing the basic slow potential model, we'll discuss approaches to quantifying neuron models. Then we'll introduce our first quantitative model--the McCulloch-Pitts neuron, and see how to implement it in Mathematica. Although it is interesting from a historical and theoretical perspective, we'll argue that it is biologically too unrealistic. The "leaky integrate and fire" neuron model better captures the fundamental properties of the slow potential model, and provides a justification for the generic connectionist model ("structure-less, continuous signal").

Qualitative summary of slow potential neuron model

Let's summarize the essential qualitative features of signal integration and transmission of a neuron with what is called the "slow potential model".



Slow potential at axon hillock waxes and wanes (because of low-pass temporal characteristics and the spatial distribution of the inputs) depending on the number of active inputs, whether they are excitatory or inhibitory, and their arrival times.

The slow integrated voltage potential now and then exceeds threshold producing an axon potential. Further, if the slow potential goes above threshold, frequency of firing is related to size of slow potential. But spike generation isn't a strictly deterministic process. There is **"noise" or random fluctuation** that can be due to several factors:

- ion channels open and close probabilistically, quantized
- neurotransmitter release in discrete packages
- sensory receptors can produce spontaneous signals, due to random fluctuations at the input (e.g. photon absorption in rod receptors).

Over long distances spike train frequency is roughly like a Poisson process (more general--a Gamma distribution on the inter-spike intervals) whose mean is modulated by the already noisy slow potential.

In order to compute with models, we need more precision--we need to make our models quantitative.

Neuron modeling abstractions

Neural Models

What is a model? It is a simplification of something complicated to help our understanding. The schematic of the slow potential model above does that. It reduces complexity but still preserves essential features of the phenomenon. A good model, however, should go beyond descriptions consistent with past data, and allow us to make predictions. Further, to make fine-grain predictions and ones involving complex interactions, we need quantitative models.

How can we make the slow-potential model precise so that we can compute outputs from inputs? There are several levels of abstraction in neural models that are useful in computational neuroscience. One

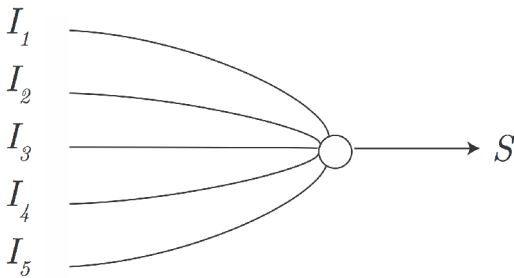
simplification is to ignore the spatial structure of a neuron, and to assume that the essential computational nature of a neuron is captured by how its inputs are integrated at a given time. This simplification will lead us to three classes of "**structure-less**" or "point" models (Segev, 1992).

But what if we want to go beyond the above slow potential model to understand how the geometry of a neuron, its dendritic tree, affects its signalling? Then we'd want a neuron model that takes into account the morphology of the neuron--"**structured**" models.

The upside of structured models is that they include sufficient detail to make testable detailed electrophysiological predictions at the neuron level. The downside is that the model of even one individual neuron can be very complex, making it difficult to characterize how tens of thousands might enable a specific behavioral function. That's where the simplifying assumptions of the "structure-less" models is useful. Structure-less models make simulation simpler, and they make the theory easier.

We now look at three types of models obtained by making various assumptions about the slow-potential model.

Structure-less ("point") models



Discrete (binary) signals--discrete time

The action potential is the key characteristic in these models. Signals are discrete (on or off), and time is discrete. These models have a close analog to digital logic circuits. At each time unit, the neuron sums its (excitatory and inhibitory) inputs, and sets the output to "high" if the sum exceeds a threshold, and "low", if it doesn't.

Examples include: *McCulloch-Pitts*, elements of the *Perceptron*, *Hopfield discrete networks* (Hopfield, 1982). A gross simplification...but the collective computational power of a large network of these simple model neurons can be great.

And when the model neuron is made more realistic (inputs are graded, last on the order of milliseconds or more, output is delayed), under certain constraints the computational properties of these networks can be preserved (Hopfield, 1984).

Below, we'll briefly discuss the computational properties of networks of **McCulloch-Pitts** neurons.

Continuous signals -- discrete time

As we'll emphasize below, action potential responses can also interpreted in terms of a single scalar continuous value--the spike frequency relative to an assumed temporal window, at the i th time interval.

Here we ignore the fine-scale temporal structure of a neuron's voltage changes.

The continuous signal, discrete time model gives us the standard the basic building block for so-called "connectionist" models, and is often associated with the phrase "neural networks". We'll study these networks in more depth in later lectures.

Both of the above two classes are useful for large scale models (thousands of neurons). Below, we'll see how the continuous signal model is an approximation of the "leaky integrate and fire" model. The leaky integrate and fire model is an example of a structure-less continuous-time model.

Structure-less continuous-time

The structure-less continuous-time model quantifies the "Slow potential model" shown in the figure above. These are analog models, and more realistic than discrete models. They are used to study nonlinear dynamics, dynamic threshold, refractory period, membrane voltage oscillations. Behavior in networks is represented by systems of coupled differential equations.

- "integrate and fire" model -- takes into account membrane capacitance. Threshold is a free parameter.

- Hodgkin-Huxley model--Realistic characterization of action potential generation at a point. Parameters defining the model have a physical interpretation (e.g. various sodium and potassium ion currents), that can be used to account for threshold. Models the form and timing of action potentials.

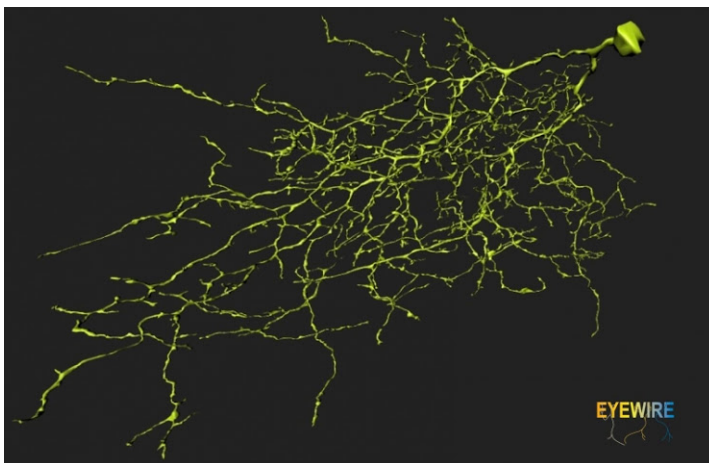
Try out a simulation:

With the right values, note how firing rate increases with current input I . We'll derive this property below using simpler assumptions.

```
nb = CreateDocument [ Import [
  "http://gandalf.psych.umn.edu/users/kersten/kersten-lab/courses/Psy5038WF2014
  /Lectures/Lect_2_TheNeuron/Hodgkin-Huxley.nb" ] ] ;
```

Structured models

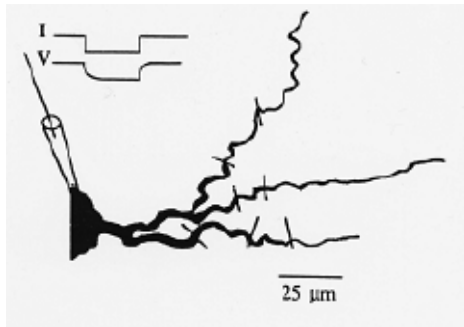
Compare the simplicity of the structure-less model with the anatomy of a real cell from the eyewire project:



Structured models are important for understanding computational properties that are functions of the shape of the neuron, e.g. the length, diameter of the dendrites, as well as the structure of the dendritic tree, i.e. its “morphology”.

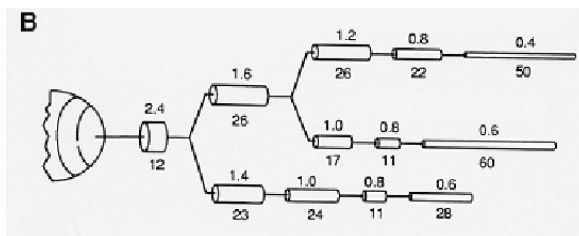
Passive - cable, compartments

In the previous lecture, we introduced the cable equation for modeling how voltage changes across space and time as a function of a current pulse input at a point.



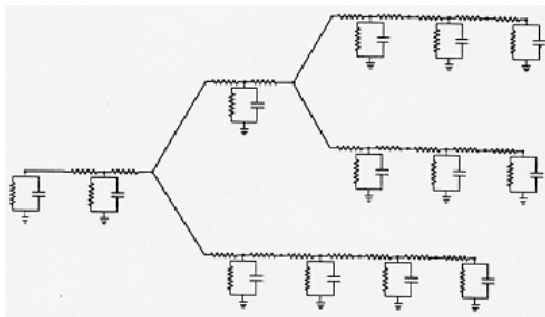
From Segev (1992).

Cable theory can be extended to model passive conduction properties of dendritic trees to take into account dendritic morphology or structure. (Rall, 1964). This can be done by applying the cable equations to segments of dendrites.



From Segev (1992).

Alternatively, one can model the tree in terms of RC "compartments": Represent electrical properties of segments of neurons with RC circuits, whose discrete resistance and capacitance values are a function of the local morphology of segments of the dendrites.

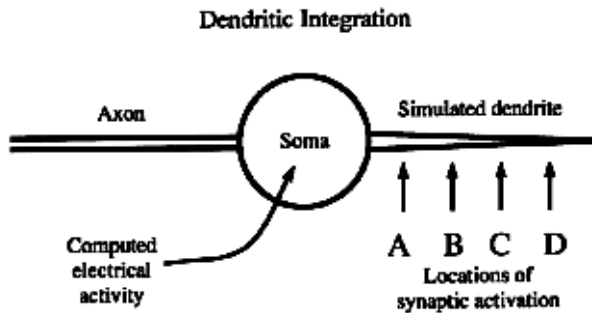


Segev (1992).

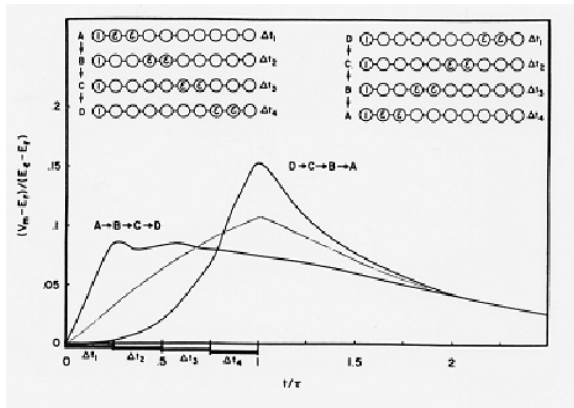
Dendritic structure is important for studying what a single neuron could compute--Rall's motion selectivity example

A model of motion selectivity provides an example of the kind of useful computation that requires a consideration of the effects of dendritic structure on integration. Suppose there is a cell in the retina that receives inputs from light sensitive cells arranged in a regular grid, as in a CCD camera. The idea is that signals from inputs further from the cell body take longer to arrive at the axon hillock than those that are closer. If there is a sequence of inputs along the dendrite, there will be an asymmetry in how a moving pattern of stimulation gets integrated--i.e. the signals at the hillock sum up differently depending on whether the inputs are sequentially activated going towards or away from the soma. In other words, the neurons response will depend on the direction of motion of the stimulus driving the cell.

A "motion selective" neuron



Consider the sequential stimulation of the dendrite from left to right (ABCD) vs. right to left (DCBA). (Recall that information flow in the neuron as a whole is from the dendritic arbor towards the soma to the axon.) (Anderson, 1995). In the DCBA direction, the voltages get stacked up, e.g. the one from D arrives at the cell body close to the time that the one from A does (see figure below, from Anderson, 1995).



The main message is that important functions, such as motion selectivity or more generally, sensitivity to timing, may depend on location of the inputs on a dendrite, and require a "structured" model.

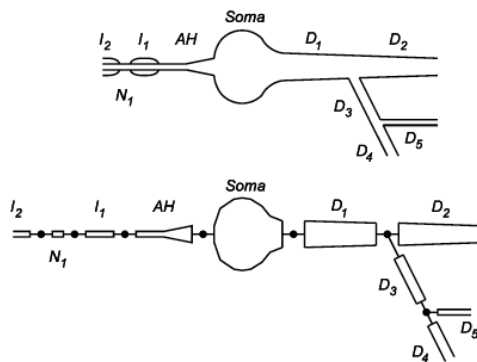
Importance of relating neural computations to perceptual/behavior function: Basis for visual motion selectivity? It is worthwhile pointing out that although this model of motion selectivity has been around for several decades, the story is more complex (e.g. Fried et al., 2005). One problem has been that dendritic transmission seems to be too fast to account for the slow velocities that can be detected by animals (Barlow, 1996). One should keep in mind too that motion processing involves much more than detection.

For the purposes of this course, dendritic morphology and its potential for increased computational

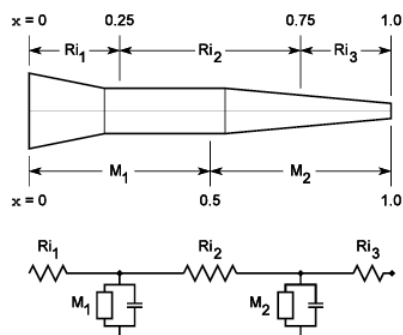
power will largely be ignored. But we should remember that simple phenomena such as our sensitivity to motion direction differences may be computed on a single neuron rather than requiring a collection. Further, there is growing evidence that quite sophisticated visual processing occurs as early as the retina of some animals (Gollisch & Meister, 2010).

Dynamic - compartmental models

A more complete, description of neuron processing takes into account the non-linear active properties of spike generation together with the morphological properties of neurons. Add Na^+ and K^+ conductance components to the compartmental RC-circuits (using Hodgkin and Huxley equations rather than passive RC circuits, see Hodgkin-Huxley.nb), and one can create non-linear trees to model non-linear dynamical properties of neurons. Computer simulations are necessary. Modern computers can handle networks on the order of tens of thousands of neurons, each with a dozen or so compartments. The relevant scientific questions are typically directed at hypotheses testable neurophysiologically. The trade-off is that computational theories of information processing, i.e. higher behavioral functions, such as pattern recognition, perception, motor control, etc. in such network models is more difficult. The reason is that there are too many parameters to manage whose precise values may not be critical for understanding the function.



From: Hines, M.L. and Carnevale, N.T. (1997) The NEURON simulation environment. *Neural Computation* 9:1179-1209.



From: Hines, M.L. and Carnevale, N.T. (1997) The NEURON simulation environment. *Neural Computation* 9:1179-1209. Rectangular block in figure contains the non-capacitive contributions to current change, ionic currents and membrane resistance (as in the Hodgkin-Huxley equation).

For software for realistic simulations see: <http://www.neuron.yale.edu/neuron/>. And <http://genesis-sim.org>. For simpler simulations using python see: <http://briansimulator.org/faq/>

McCulloch-Pitts: Discrete-time, discrete state (binary)

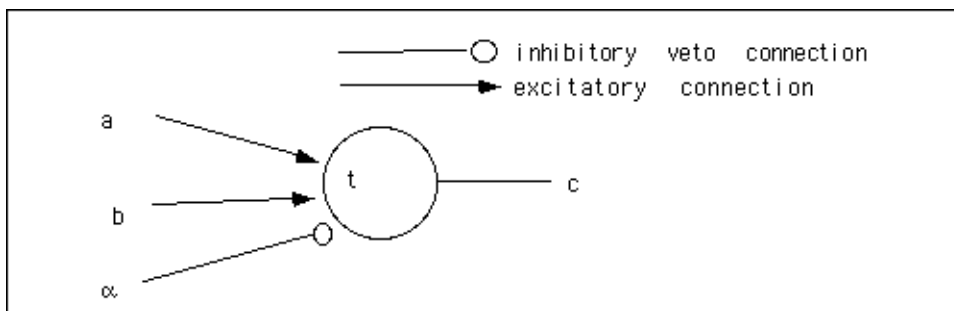
Introduction

Published in 1943, the McCulloch-Pitts model is famous and important because it showed that with a few simple assumptions, networks of neurons may be capable of computing the full repertoire of logical operations. Although some of the basic facts about the physiology were wrong, the notion of neurons as computational devices remains with us.

The model ignores some of the very properties we just looked at that might be important for certain kinds of neural processing (e.g. dendritic cable transmission properties, frequency coding in terms of the rate of neural spike discharge, noise). But the model abstracts properties that at the time seemed the most essential.

Apart from the understanding of the biochemical basis of neural transmission, by the 1940's the basic signalling properties of neurons were well-known. In their seminal paper, McCulloch and Pitts formalized what was known, and developed a theory of neural networks that related the functioning of the brain to the then infant field of digital computers. They made the following assumptions. (See the parenthetical comments for limitations to their assumptions.)

- neuron signals are all-or-none (true for action potentials)
- a certain "fixed number of synapses must be excited" within a latent period of addition to excite the neuron (not that simple, also depends on strength of activation, distance from soma, etc. as discussed above)
- the number of synapses is independent of previous activity and position on the neuron (Neurons adapt, and synaptic weights change with learning, changing the "effective" number of synapses. But still a common simplifying assumption.)
- the only significant delay is synaptic delay (there are other delays as we have seen above)
- there are excitatory and inhibitory synapses (true)
- structure of the net does not change with time (can change, but still a common assumption)



With the right choice of parameters, the McCulloch-Pitts neuron can do various logical operations.

Review of basic logical operations

Before looking at the McCulloch-Pitts model, let's review simple two-input logical functions where the inputs are a and b , and the output is c . Inputs and outputs can take on only two states $\{\text{False}, \text{True}\} = \{0, 1\}$.

Inclusive OR: Or[]

The output of a two-input inclusive OR is false if and only if both inputs are false. You can use "C" programming style notation for OR:

```
In[15]= x = True; y = False;
        x || y
```

```
Out[16]= True
```

...or try the built-in function notation: Or[x,y]

```
Or[x, y]
```

```
True
```

AND: And[]

And[] outputs true if and only if both inputs are true.

```
In[3]= And[x, y]
```

```
Out[3]= False
```

Try AND with C style notation: x&&y

Composite functions, e.g. define Mylogicalfunction

```
In[4]= Mylogicalfunction[a_, b_] := Or[And[Not[a], b], b]
```

What input combinations produce a "True" output for the logical function "Exclusive OR": Xor[]?

McCulloch-Pitts: Inclusive OR, AND

Let's construct a McCulloch-Pitts neuron to compute OR and AND. Let's set alpha to zero (no inhibition). The McCulloch-Pitts neuron sums its (binary) inputs, tests to see if the sum is bigger or less than the threshold. If bigger or equal, the output is set to 1, otherwise it is set to 0. We can model the McCulloch-Pitts neuron's response like this:

```
In[18]= McCullochPitts[a_, b_, t_] := If[a + b >= t, 1, 0];
```

By convention True <=> 1, and False <=> 0.

The McCulloch-Pitts neuron is said to be computing *threshold logic*.

Exercise Inclusive OR

Set the threshold to 1, and find out what kind of function the neuron is computing:

$$c = \begin{cases} 1 & \text{if } a + b \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

```
In[37]:= McCullochPitts[1, 1, 1]
```

```
Out[37]= 1
```

Try it out for various values of a and b.

Truth tables

There are only 2 inputs with 2 possible values each, so let's list all the neuron's possible responses, defining a truth table. We use two *Mathematica* functions, **Table[]**, and **Flatten[]** to define a list that we'll call **truthtable**. **Table[]** is used to make lists, and because we have two indices (a, b), and a list as the first argument to **Table[]**, it makes a list of lists of lists.

```
In[7]:= Table[{a, b, McCullochPitts[a, b, 1]}, {a, 0, 1}, {b, 0, 1}]
```

```
Out[7]= {{{0, 0, 0}, {0, 1, 1}}, {{1, 0, 1}, {1, 1, 1}}}
```

Sidenote: Enumerating all possible combinations crops up so often in mathematics that *Mathematica* has a special function **Outer[]** function that can be used to compute functions on all possible combinations (see **Appendix**). E.g. Try **Outer[List, {False, True}, {False, True}, {False, True}]**. This is just one use of **Outer[]**. Later when we study Hebbian learning, we'll use **Outer** to produce all possible products of inputs and outputs in a neural network. Learning weights are proportional to these "outer products". **Outer[]** is also useful for calculating the covariance between variables.

The above list has dimensions 2x2x3 (because of the extra curly brackets). We use **Flatten[1]** to flatten **truthtable** to level 1 to convert it to a 4x3 matrix. Then we can display it in "truth table" format, and view the result in "TableForm", "MatrixForm" or "TraditionalForm" (which is the default in *Mathematica*).

```
In[8]:= truthtable =
  Flatten[Table[{a, b, McCullochPitts[a, b, 1]}, {a, 0, 1}, {b, 0, 1}], 1] // TableForm
```

```
Out[8]/TableForm=
```

```
0  0  0
0  1  1
1  0  1
1  1  1
```

Sidenote: *Mathematica* provides a wide range of list operations which you can read about in "**Lists and Matrices: List Operations**" in the **Built-in Functions** under **Help**. *Mathematica* also allows type-mixing in lists, so for example we can insert labels into our truthtable like this

```
In[9]:= Insert[truthtable, {"a", "b", "c"}, {1, 1}] // TableForm
```

```
Out[9]/TableForm=
```

```
a  b  c
0  0  0
0  1  1
1  0  1
1  1  1
```

Compare with *Mathematica*'s predefined **Or[]** function:

```
In[10]:= truthtable =
  Flatten[Table[{a, b, Or[a == 1, b == 1]}, {a, 0, 1}, {b, 0, 1}], 1] // TableForm
```

```
Out[10]/TableForm=
  0  0  False
  0  1  True
  1  0  True
  1  1  True
```

TableForm can also be specified as a function `TableForm[]`.

Exercise: AND

Using `McCullochPitts[a, b, t]`, find a threshold value that will enable a McCulloch-Pitts neuron to realize the **And[]** function whose truth table is shown below.

```
In[11]:= truthtable = Flatten[Table[{a, b, And[a == 1, b == 1]}, {a, 0, 1}, {b, 0, 1}], 1];
  TableForm[truthtable, TableHeadings -> {{}, {"a", "b", "c"}}]
```

```
Out[12]/TableForm=
  | a  b  c
  |---|---|
  | 0  0  False
  | 0  1  False
  | 1  0  False
  | 1  1  True
```

Main conclusions

It is straightforward to show that **Not** can be implemented with the inhibitory input. **And**, **Or**, and **Not** are a complete set in the sense that any logical operation can be built out of them. This led to the main result of 1943 paper:

Any finite logical expression can be realized by McCulloch-Pitts neurons.

Their paper helped to give rise to the idea that the brain was very much like a digital computer. John von Neumann made explicit reference to the McCulloch-Pitts model in his famous 1945 technical report on the EDVAC.

How realistic is the McCulloch-Pitts model?

The deterministic computer view of the brain did not hold under close examination. Over the decades following McCulloch & Pitts, progress in theory and experimental findings led to a significant change in the way neural systems were seen to operate. Today many people prefer to think of neural networks as doing statistical pattern processing, rather than deterministic logical computation.

The developments that led to this change of view included:

The statistical nature of "natural" information processing:

- information theory and statistical decision theory

which emphasized the statistical nature of information transmission and processing in the presence of uncertainty and noise that was more characteristic of "real world" information processing.

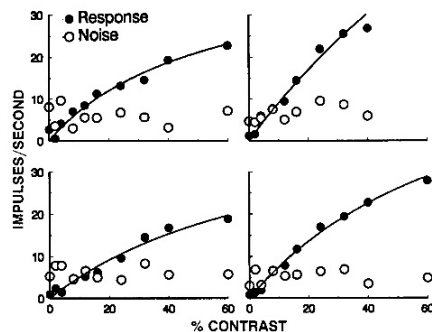
- Psychophysics and signal detection theory showed the importance of noise, false alarms, in human perception and decision making

- Physiological data on inherent statistical and analog nature of sensory coding by neurons became apparent, e.g. input intensities are probably coded in terms of average frequency of spike trains, not in terms of a precise sequence—but this is still debated. The neural code is noisy. The

McCulloch-Pitts model is a structure-less discrete time model--sensory experiments suggested that "structure-less continuous signal, and continuous-time" models would be better.

Deeper understanding of the electrophysiology:

- The Hodgkin-Huxley model of neural discharge added substantial richness to our understanding of the mechanism of spike generation, and neural conduction.



The above figure shows the number of action potentials per second in a primate retinal ganglion cell as a function of the contrast of sinusoidal gratings of light. Illustrates relationship between firing rate on contrast, and the "noisiness" of action potential rates. From: Croner, L. J., Purpura, K., & Kaplan, E. (1993). Response variability in retinal ganglion cells of primates. *Proc Natl Acad Sci U S A*, 90(17), 8128-8130.

Around 1956, from his deathbed, John von Neumann realized that the deterministic logical computation view was wrong, and wrote:

"The language of the brain is not the language of mathematics" and "...the message-system used in the nervous system, as described in the above, is of an essentially *statistical* character. In other words what matters are not the precise positions of definite markers, digits, but the statistical characteristics of their occurrence, i.e. frequencies of the periodic or nearly periodic pulse-trains, etc." Italics are his.

He predicted that brain theory would eventually come to resemble the physics of statistical mechanics and thermodynamics. Later on in this course, we'll see how by the 1980s von Neumann's prediction came true, at least in theoretical modeling of brain functioning. (Von Neumann died in February, 1957). And today, there are new twists that we'll discuss later in the course when we introduce the idea that populations of neurons may explicitly represent probability distributions.

Let us now examine more closely the rationale for continuous-response, and continuous-time models, and look at some simple models of the neuron that incorporate a frequency response. Then we will formally introduce the "generic neuron model" that we will use for much of this course.

Integrate and fire model of the neuron

The "integrate-and-fire model" of the neuron

A major limitation of the McCulloch-Pitts model is that it assumed that the fundamental language of neural communication was binary. From sensory studies, we know that neurons often encode information about stimulus intensity in terms of rate of firing. Let's see how this might arise with some simple assumptions about how action potentials are generated. The "integrate-and-fire model" of the neuron is a structure-less, continuous time model.

We'll consider one effective input, $s(t)$, which could result from the summed potential (from lots of synaptic inputs) at an axon hillock at time t .

Let $V(t)$, and $s(t)$ represent the output membrane voltage potential and stimulus input (ionic current) to a neuron, respectively. Because of the membrane's capacitance, the rate of change of the membrane potential is proportional to the input current. Thus over time the potential, $V(t)$, grows as more and more current pumps into the cell. When $V(t)$ reaches some critical value, an action potential is generated, after which the neuron returns to its resting state, and begins integrating again. We can model the rate of change in voltage, up to, but not including the action potential event, as:

$$dV / dt = (1 / C) s (t) \quad (1)$$

This equation is not special to neurons, but comes from basic electronics that tells us that the charge across a capacitor is proportional to the voltage: $q = CV$, where C is a constant called the capacitance. Current, s , is the rate of change of charge: $s = dq/dt = C dV/dt$. For simplicity, we'll let the capacitance $C=1$, and integrate to obtain the voltage change, $V(t)$, between time t_1 and t_2 :

$$V (t) = \int_{t_1}^{t_2} s (t) dt \quad (2)$$

For small time intervals, and a smooth input, the integral is approximately the area of the rectangle under s (where s is plotted against t):

$$\theta = \int_{t_1}^{t_2} s (t) dt \approx (t_2 - t_1) s = Ts$$

After time T , the neuron's potential increases up to some point, say θ . Let θ be the critical threshold point at which a spike is generated, after which time the voltage gets reset. So the time (or period, T) between spikes is θ/s . The frequency of action potential firing is the reciprocal of the period

$$\frac{1}{T} = \frac{s}{\theta} = f \quad (3)$$

... all this to conclude that:

$$f \propto s \quad (4)$$

So we've shown that: **the frequency of firing, f , is proportional to the input current level, s .** As we will see below, this model assumes that the membrane integrates current with no leakage--i.e. it is a pure capacitance, with no resistance. We can improve the model by including both resistive and capacitance elements to the equation--this is a **leaky integrate-and-fire** or "forgetful integrate-and-fire" model. The calculus gets a bit more sophisticated, so we'll use *Mathematica* to solve the equations for us. Before we derive the relationship between stimulus input level and the frequency of firing for the "leaky" or "forgetful integrate-and-fire" model, let's go over some more *Mathematica* basics, and then develop a few tools.

More *Mathematica*: rules, functions, and derivatives

Rules and defining functions.

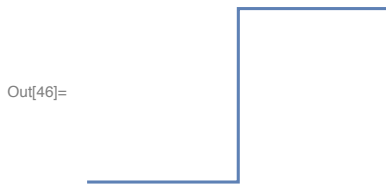
In *Mathematica*, you can define functions in terms of rules. One use of rules is to define functions over specific ranges. For example,

```
In[42]:= Clear[step] (* If you've been playing with the definition,
it is a good idea to clear it, because additional constraints accumulate *)

step[x_] := 1 /; x >= 0
step[x_] := -1 /; x < 0
```

Here `/;` means the function is defined "with the following rule...". The rules can be incompatible and *Mathematica* will evaluate the rules in a specific order, usually the order that you specified. You can find out what order the rules will be evaluated by typing `?step`. The `Clear[]` function clears any prior definition of the function `step[]`. `Clear` can take multiple arguments (e.g. `Clear[f,g,h]`).

```
In[46]:= Plot[step[x], {x, -3, 3}, Axes -> False]
```



The rule for replacing a variable with a value in an expression is denoted by the operator `/.` meaning **given that** and `->` meaning **goes to**:

```
In[47]:= d + 2 e /. d->3 e
```

Out[47]= 5 e

Derivatives and integrals

The derivative of $f[x]$, with respect to x is $D[f[x], x]$. For example, here is the derivative of x^3 :

```
In[48]:= Clear[x, f]
D[x^3, x]
```

Out[49]= 3 x²

Mathematica also uses the "prime" symbol to indicate derivative of a function.

```
In[50]:= f[x_] := x^3;
f'[x]
D[f[x], x]
```

Out[51]= 3 x²

Out[52]= 3 x²

We can use *Mathematica* to calculate the indefinite integral of this function:

```
In[53]:= Integrate[3 x^2, x]
```

```
Out[53]= x^3
```

You can also do a numerical integration, which is particularly useful when a closed form solution isn't available:

```
In[54]:= NIntegrate[3 x^2, {x, 0, 2}]
```

```
Out[54]= 8.
```

Differential equations

Some illustrations of differential equation solution

The dynamics of many natural systems can be described in terms of differential equations. Later on we will see how the dynamics of models of large scale neural systems can be described in terms of coupled differential equations. A differential equation captures a set of constraints on the rates of change of some dependent variables. Given these rates of change, we would often like to find out how the dependent variable itself changes with time. In the above integrate and fire model, we used the result from basic electronics that tells us that the charge across a capacitor is proportional to the voltage: $q = CV$, where C is a constant called the capacitance. Current, s , is the rate of change of charge: $s = dq/dt = C dV/dt$. So we are given the rate of change of voltage, and would like to find out how the voltage itself changes with time. In general the answer isn't unique without additional constraints, such as specifying the initial conditions (i.e. at time =0).

To illustrate, suppose, we know that $s[t] = \cos(t)$, and that at time $t=0$, $V=0$, then what is the dependent variable $V(t)$?

```
In[55]:= DSolve[{V'[t] - (1/C) Cos[t] == 0,
               V[0] == 0}, V[t], t]
```

```
Out[55]= {{V[t] -> \frac{Sin[t]}{c}}}
```

(The result shows that the voltage response is a phase-shifted version of the input.). You have probably noticed that the differential equation problem could have been easily solved by integration, (e.g. using **Integrate[]**). But as you will see below for the leaky integrate-and-fire neuron model, you can't always simply solve an integral to find the solution. In particular, this happens when the rate of change of V depends on V itself. For example, here is familiar example from physics--a second order (second order because it involves a second derivative) equation for an oscillator. The acceleration of a mass on an ideal spring is proportional to the displacement: $d^2X/dt^2 = -k X$. Let $k = 1$, and assume initial conditions $X[0] = 0$, $X'[0] = 1$, then **DSolve** tells us that the mass on the spring will oscillate sinusoidally:

```
In[56]:= DSolve[{X'[t] + X[t] == 0,
                X[0] == 0, X'[0] == 1}, X[t], t]
```

```
Out[56]= {{X[t] -> Sin[t]}}
```

What is the solution if the initial speed is zero and the start position is zero too?

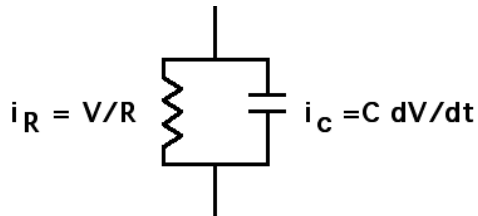
What if the initial speed is zero, but you pull back the end of the spring to -1?

The "Leaky integrate-and-fire model" of the neuron using DSolve[]

The integrate and fire model of the neuron is a simple extension of the integrate and fire model where we now assume (up until depolarization threshold, θ) that the neuron membrane is a passive resistor and capacitor with some input current source. The current input leads to an increase in the resting potential until sufficient depolarization triggers an action potential. Like the integrate-and-fire model, it is a structure-less continuous time model, and with a scalar driving input.

We will develop the model in two parts. Our goal is to find how firing rate depends on the input current and threshold. First, we'll derive the relationship between membrane potential and time. Then we will derive the relationship between frequency of firing and input current, similar to how we did it for the simple integrate-and-fire model above.

The input current, $s[t]$ is conserved and is determined by the sum of the current through the resistance and the capacitor.



So by Kirchoff's current law, the current *in* equals the current *out* :

$$s = i_R + i_C,$$

Using Ohm's law, and the definitions of capacitance ($q = CV$) and current ($i=dq/dt$):

$$s[t] = V/R + C dV/dt, \text{ or rearranging}$$

$$dV/dt = s/C - V/(RC).$$

(What if $R \rightarrow \infty$?)

We can find the solution using the *Mathematica* function **DSolve**, given the initial conditions that at time $t=0$, the voltage is **a**. For simplicity, let $RC=1$

```
In[57]:= Clear[a, s];
          DSolve[{V'[t] + V[t] - s/C == 0,
                V[0] == a}, V[t], t]
```

```
Out[58]= {{V[t] -> \frac{e^{-t} (a C - s + e^t s)}{C}}}
```

After T seconds, the voltage reaches threshold, θ , and a spike occurs. We can use **Solve[]** to find

solutions to the above equation in terms of t:

```
In[59]:= Solve[ $\theta == (a * C - s + E^{T * s}) / (C * E^T)$ , T]
```

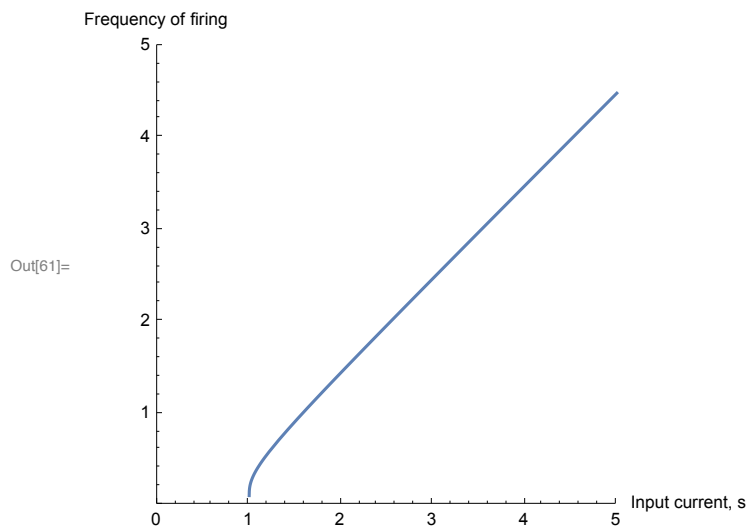
```
Out[59]:= {{T -> ConditionalExpression[ $2 i \pi C[1] + \text{Log}\left[\frac{a C - s}{-s + C \theta}\right]$ , C[1] ∈ Integers]}}
```

E is 2.718..., and E^x is the same as $\text{Exp}[x]$. As we saw before, the frequency is 1/time. So we define a new function **frequency[]** as the the reciprocal of the above function :

```
In[60]:= frequency[s_, C1_, a_,  $\theta$ _] :=  
1 / (Log[ $-(a * C1) + s$ ] / (s - C1 *  $\theta$ ));
```

If we plot it for a capacitance of 1, and threshold of 1, frequency of firing as a function of input strength (current) looks like:

```
In[61]:= Plot[frequency[s, 1, 0, 1], {s, 0, 5}, PlotRange -> {{0, 5}, {0, 5}},  
AspectRatio -> 1, AxesLabel -> {"Input current, s", "Frequency of firing"}]
```



To review what we've done, the "leaky integrate and fire" neuron shows two properties characteristic of many neurons. First, it shows a threshold--it doesn't begin firing until the input current is sufficiently high. Second, once threshold is exceeded, the frequency of firing grows in proportion to the input current. One characteristic we haven't modeled is the absolute refractory period.

How would the shape of the above plot change if we included the effects of the absolute refractory period?

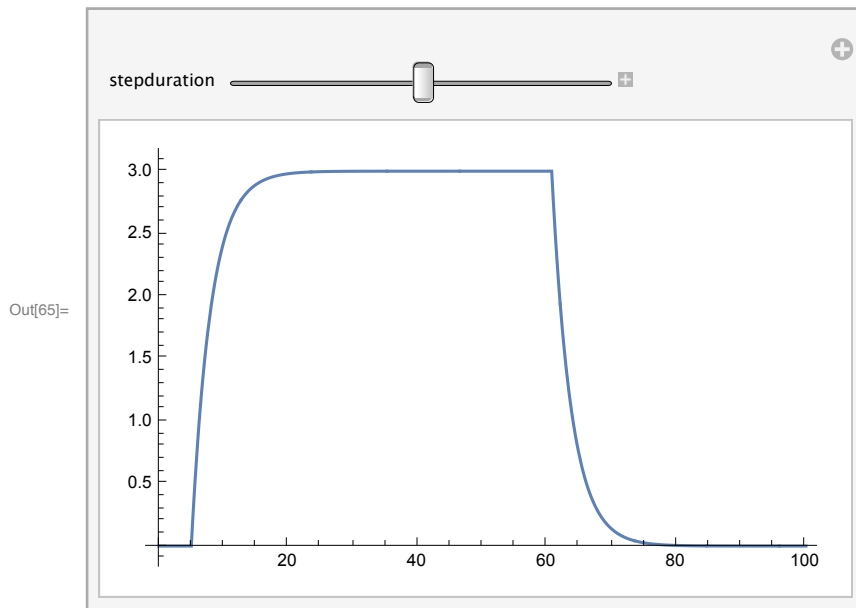
Use `NDSolve[]` to Plot the passive RC response to a current input of duration = *stepduration*

We can solve the above differential equation for the rate of change of voltage as a function of current input $s[t]$, resistance $R1$, and capacitance $C1$. $s[t]$ now is a pulse of duration "stepduration". This example illustrates the use of the *Mathematica* function `Module[]`.

```

In[64]:= Clear[s, t, R]
Manipulate[Module[{V0, R1, C1, t}, V0 = 0.0; R1 = 3.0; C1 = 1.0;
  s[t_, stepduration_] := If[t < 5 || t > 5 + stepduration, 0, 1];
  R = NDSolve[{V'[t] ==  $\frac{1}{C1} \left( s[t, stepduration] - \frac{V[t]}{R1} \right)$ , V[0] == V0}, V, {t, 0, 100}];
  Plot[Evaluate[V[t] /. R], {t, 0, 100}, PlotRange -> All],
  {{stepduration, 50.0}, 10.0, 100.0}]

```



Next time

Generic neuron model

We develop a "structure-less, continuous signal, and discrete time" generic neuron model and from there build a network.

This generic model is a linear approximation, one of several abstractions that we saw above.

We review basic linear algebra. Motivate linear algebra concepts from neural networks.

References

Anderson, J. C., Binzegger, T., Kahana, O., Martin, K. A., & Segev, I. (1999). Dendritic asymmetry cannot account for directional responses of neurons in visual cortex. *Nature Neuroscience*, 2(9),

820–824. doi:10.1038/12194

Barlow, H. (1996). Intraneuronal information processing, directional selectivity and memory for spatio-temporal sequences. *Network: Computation in Neural Systems*, *7*, 251-259.

Dayan, P., & Abbott, L. F. (2001). *Theoretical neuroscience : computational and mathematical modeling of neural systems*. Cambridge, Mass.: MIT Press.

Fried, S. I., Muench, T. A., & Werblin, F. S. (2005). Directional Selectivity Is Formed at Multiple Levels by Laterally Offset Inhibition in the Rabbit Retina. *Neuron*, *46*(1), 117–127. doi:10.1016/j.neuron.2005.02.007

Gollisch, T., & Meister, M. (2010). Eye Smarter than Scientists Believed: Neural Computations in Circuits of the Retina. *Neuron*, *65*(2), 150–164. doi:10.1016/j.neuron.2009.12.009

Hines, M.L. and Carnevale, N.T. (1997) The NEURON simulation environment. *Neural Computation* 9:1179-1209. nsimenv.pdf (for Adobe Acrobat). <http://www.neuron.yale.edu/neuron/papers/nc97/nc-toc.htm>, NEURON software package: <http://www.neuron.yale.edu/neuron/>

Hopfield, J. J. (1984). Neurons with graded response have collective computational properties like those of two-state neurons. *Proc. Natl. Acad. Sci. USA*, *81*, 3088-3092.

Koch, C., & Segev, I. (2000). The role of single neurons in information processing. *Nat Neurosci*, *3 Suppl*, 1171-1177.

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, *5*, 115-133.

Meunier, C., & Segev, I. (2002). Playing the devil's advocate: is the Hodgkin-Huxley model useful? *Trends Neurosci*, *25*(11), 558-563.

Rall, W. (1967). Distinguishing theoretical synaptic potentials computed for different soma-dendritic distributions of synaptic input. *J Neurophysiol*, *30*(5), 1138-68.

Segev, I. (1992). Single neurone models: oversimple, complex and reduced. *Trends in Neuroscience*, *15*(11), 414-421.

Appendix

Mathematica functions for generating and plotting lists

Using Tables to make Lists.

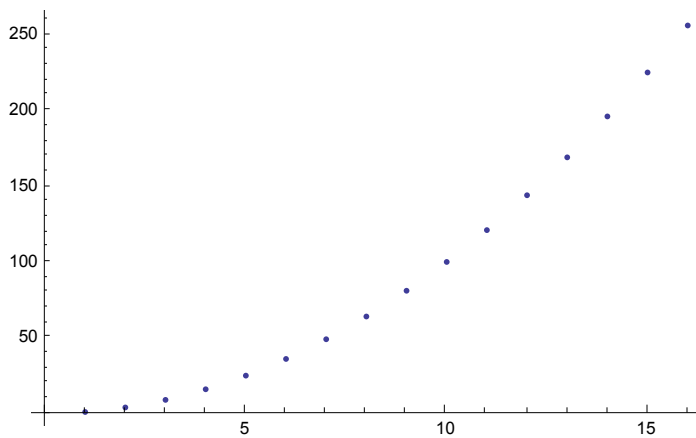
Often we will have to define a list of input values to a neuron, or a list of synaptic weights. A convenient way of defining lists in *Mathematica* is to use the `Table[]` function. For example, you can make a list whose elements are the squares of the element location.

```
s = Table[x^2, {x, 1, 16}];
```

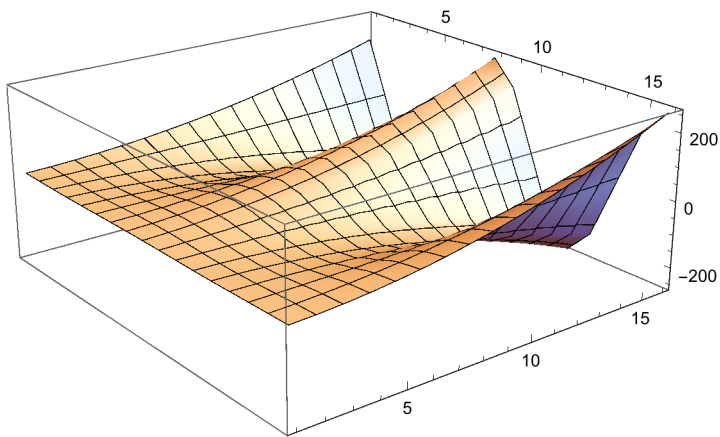
You can also use `Table` to make a list of lists, Here is a 16x16 matrix:

```
A = Table[x^2*Cos[2 Pi (1/8) y], {x,1,16}, {y,1,16}];
```

To graph a one-dimensional list, you have to use `ListPlot`:

ListPlot[s]

To graph a two-dimensional list, you have to use ListPlot3D:

ListPlot3D[A]

Other *Mathematica* functions for generating truth tables

The Outer[] function

Let's list all possible input states a:

```
a = {False, True};
```

All possible pairs of inputs can be obtained using the **Outer[]** function with the **List** function as the first argument:

```
Outer[List, a, a]
```

```
({False, False} {False, True}
 {True, False}  {True, True})
```

If we want a function, say logical **Or**, of all possible combinations of the inputs, we write:

Outer[Or, a, b]

Outer::ipnfm: Positive machine-size integer or Infinity expected at position 3 in Outer[Or, {False, True}, b]. >>

Outer[Or, {False, True}, b]

And if we want to summarize the **Or** function with a truth table, we can just put it together by hand:

Inclusive OR

a	b	c
0	0	0
0	1	1
1	0	1
1	1	1

...or to automate the process of truth table generation, we can create the following rather messy function composed of a bunch of built-in list manipulation functions:

Truth table function

```
truthtable[logicfunction_] :=
  TableForm[Transpose[Append[Transpose[Flatten[Outer[List, a, a], 1]],
    Flatten[Outer[logicfunction, a, b]]]],
    TableHeadings -> {{}, {"a", "b", "output"}}]
```

```
SetDelayed::write: Tag List in  $\begin{pmatrix} 0 & 0 & \text{False} \\ 0 & 1 & \text{False} \\ 1 & 0 & \text{False} \\ 1 & 1 & \text{True} \end{pmatrix}$  [logicfunction_] is Protected. >>
```

\$Failed

truthtable[Or]

```
 $\begin{pmatrix} 0 & 0 & \text{False} \\ 0 & 1 & \text{False} \\ 1 & 0 & \text{False} \\ 1 & 1 & \text{True} \end{pmatrix}$  [Or]
```

3-input truth tables

```
truthtable3[logicfunction_] := TableForm[Transpose[Append[
  Transpose[Flatten[Outer[List, {False, True}, {False, True}, {False, True}], 2]],
  Flatten[Outer[logicfunction, {False, True}, {False, True}, {False, True}]]],
  TableHeadings -> {{}, {"a", "b", "c", "output"}}]
```

truthtable2[And]

truthtable2(And)