Introduction to Neural Networks
U. Minn. Psy 5038

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 to justify the assumption of frequency codes

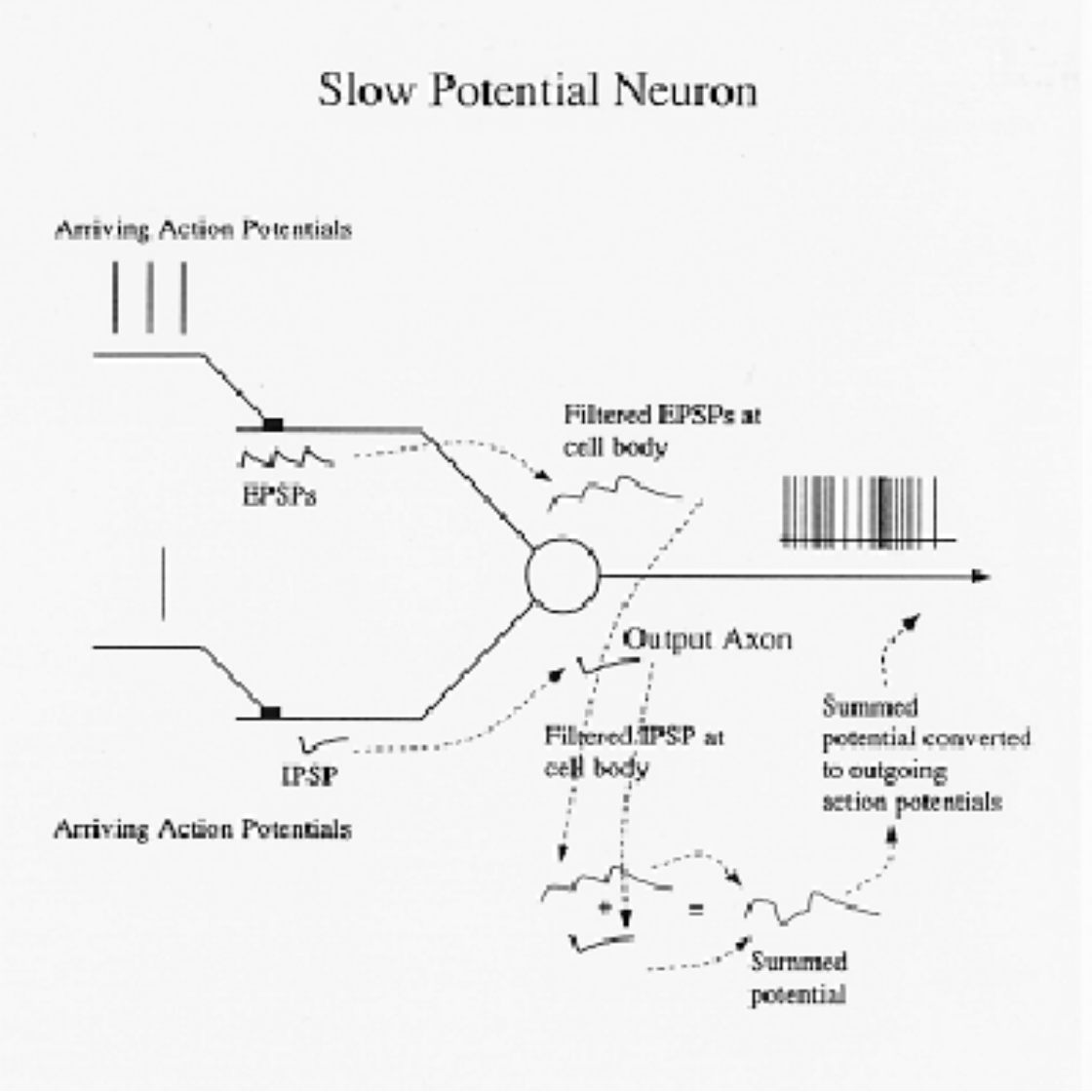
--prepares the stage for a slightly simpler model: the generic (connectionist) model that will be used for a large fraction of the course.

Our pedagogical strategy will be to converge on the "right" model for large-scale simulations--we'll first be too simple (McCulloch-Pitts), then too complex (Integrate-and-fire), and finally just right (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.

Caveat: Not all signal transmission in neural computation is done through action potentials. For example, of the 6 types of cells in the retina of your eye, essentially 1 type, the ganglion cell, uses action potentials as the predominant conveyor of information, the others communicate via slow (also called "graded") potentials.

But spike generation isn't a strictly deterministic process. There is "n**oise" 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.

Models and neural computation

Neural Models

What is a model? 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 description, and allow us to make predictions. Further, to make fine-grain predictions, we need quantitative models. Quantitative models enable us to simulate neural computation.

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 simplication 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 an individual neuron can be so complex, it becomes difficult to characterize how tens of thousands might behave with respect to a specific behavioral function, and 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.

Let's look at the various types of neurons, starting from the simplest to the more complex.

Structure-less ("point") models

Let's look at three classes obtained by making various assumptions about the slow-potential model.

■ 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.

At each time unit, the neuron sums its (excitatory and inhibitory) inputs, and turns on the output if the sum exceeds a threshold.

e.g. *McCulloch-Pitts*, elements of the *Perceptron*, *Hopfield discrete nets*.

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), 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

Action potential responses are interpreted in terms of a single scalar continuous value--the spike frequency--at the ith time interval. Here we ignore the fine-scale temporal structure of a neuron's voltage changes.

The discrete time model gives us the standard the basic building block for the majority of networks considered in this course. It is also the model used in so-called "connectionist" approaches.

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

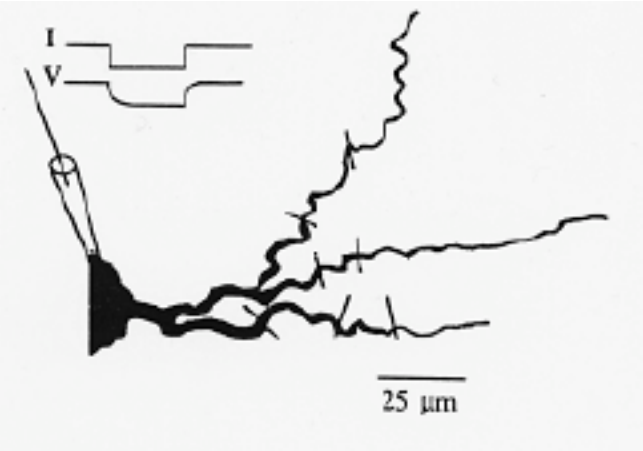
Quantifies above "Slow potential model". Analog. More realistic than discrete models. Emphasizes nonlinear dynamics, dynamic threshold, refractory period, membrane voltage oscillations. Behavior represented by differential equations.

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

Structured models

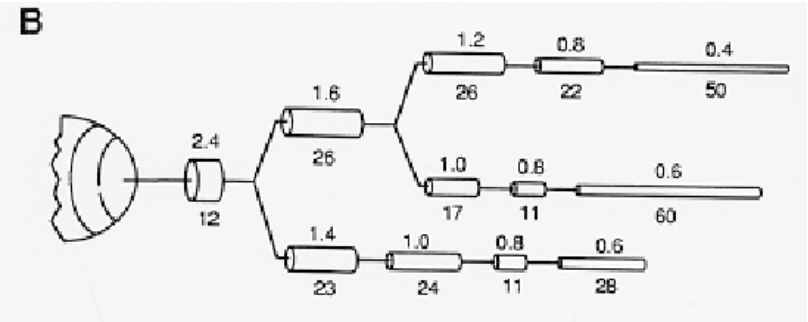
Important for understanding computational properties that are functions of the shape of the neuron, e.g. including 2D and 3D.

■ Passive - cable, compartments



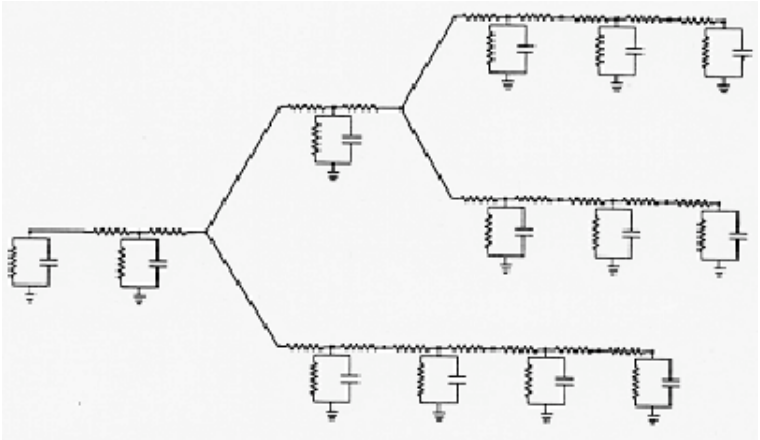
From Segev (1992).

Cable theory - passive trees. Assume membrane is passive. Take into account dendritic morphology or structure. (Rall, 1964). Uses cable equations on segments of dendrites.



From Segev (1992).

Compartments: Represent electrical properties of segments of neurons with RC circuits



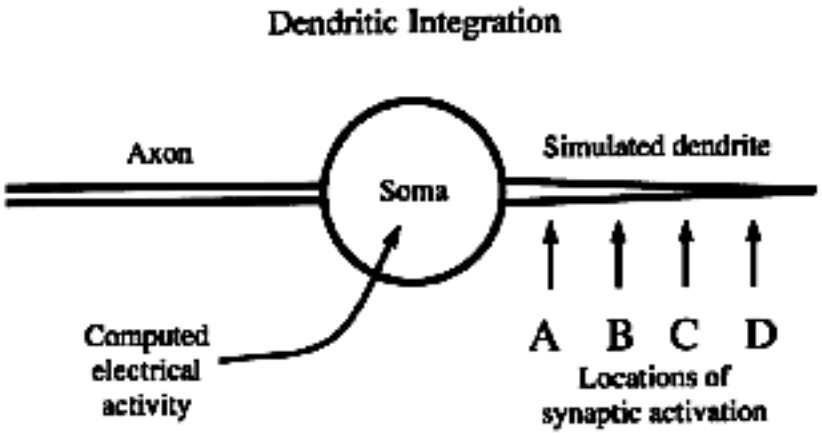
Segev (1992).

Dendritic structure shows what a single neuron can compute--Rall's motion selectivity example

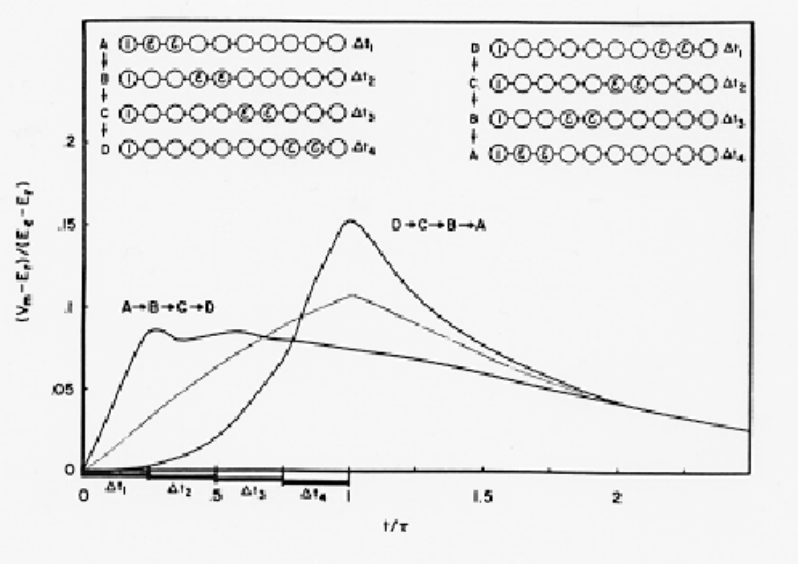
Dendritic structure is important because it can show what a single neuron can compute.

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.

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 axon.) (Anderson, 1995).



The main message is that important functions, such as motion selectivity or sensitivity to timing, may depend on location of the inputs on a dendrite.

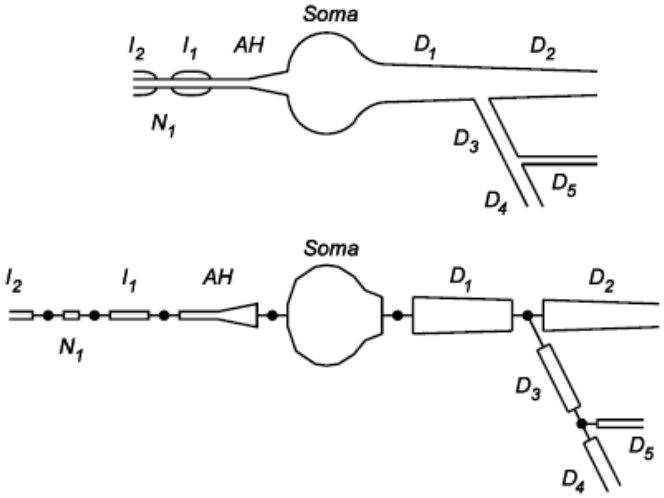
Importance of relating neural computations to perceptual/behavior function: Basis for visual motion selectivity? It is worthwhile pointing out that although this model of motion selectivity has been around for several decades, it has yet to be established that this is right model for motion selectivity of visual neurons. A major problem has been that dendritic transmission is actually too fast to account for the slow velocities that can be detected by animals (Barlow, 1996).

For the purposes of this course, dendritic morphology and its potential for increased computational power will unfortu-

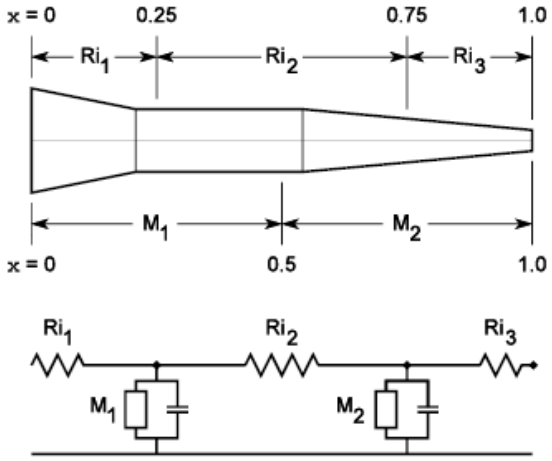
nately largely be ignored. We should remember that simple phenomena such as our sensitivity to motion direction differences may be computed on a single neuron rather than requiring a collection.

■ 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, see Appendix of previous lecture notes), 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 half-dozen or so compartments. The more severe challenge is that computational theories of information processing in such networks becomes more difficult.



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).

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

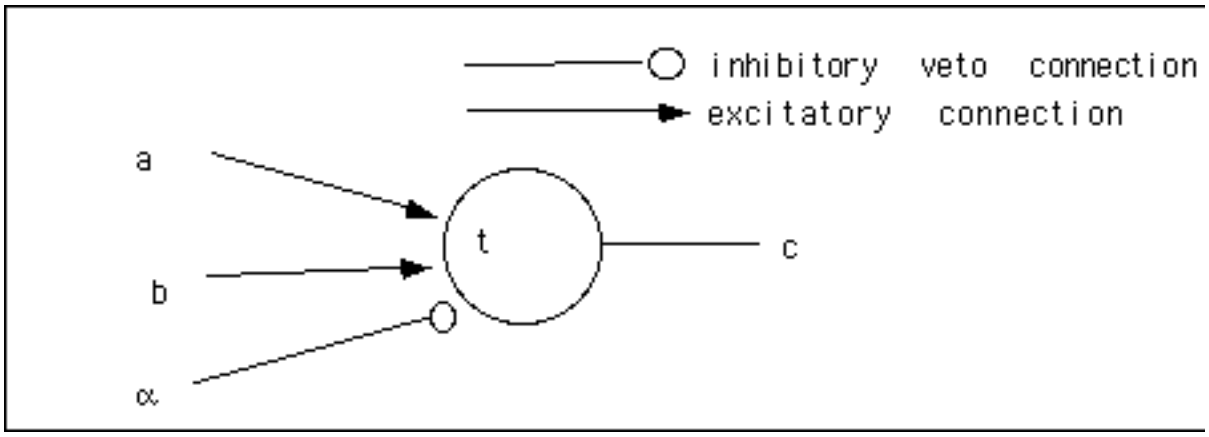
Introduction

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. motion direction selectivity through dendritic cable transmission properties, frequency coding, 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:

- neuron signals are all-or-none
- a certain fixed number of synapses must be excited within a latent period of addition to excite the neuron
- the number of synapses is independent of previous activity and position on the neuron
- the only significant delay is synaptic delay
- there are excitatory and inhibitory synapses
- structure of the net does not change with time



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 {False, 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[1]:=
```
x = True; y = False;
x || y
```

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

In[3]:=
```
Or[x, y]
```

■ AND: And[ ]

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

```
And[x, y]
```

Try AND with C style notation: x&&y

Composite functions, e.g. define Mylogicalfunction

```
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. For simplicity, 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[4]:=
```
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*.

■ Inclusive OR

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

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

```
McCullochPitts[1, 1, 1]
```

```
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[11]:=
```
Table[{a, b, McCullochPitts[a, b, 1]}, {a, 0, 1}, {b, 0, 1}] //
 StandardForm
```

Out[11]//StandardForm=

```
{{{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[6]:=
```
truthtable =
 Flatten[Table[{a, b, McCullochPitts[a, b, 1]}, {a, 0, 1}, {b, 0, 1}], 1] //
  TableForm
```

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[7]:=
```
Insert[truthtable, {"a", "b", "c"}, {1, 1}] // TableForm
```

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

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

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

Exercise: AND

Find a threshold value that will enable a McCulloch-Pitts neuron to realize the **And[]** function whose truth table is shown below.

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

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.

Main result of 1943 paper:

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

This gave 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 good 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 perfer 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:

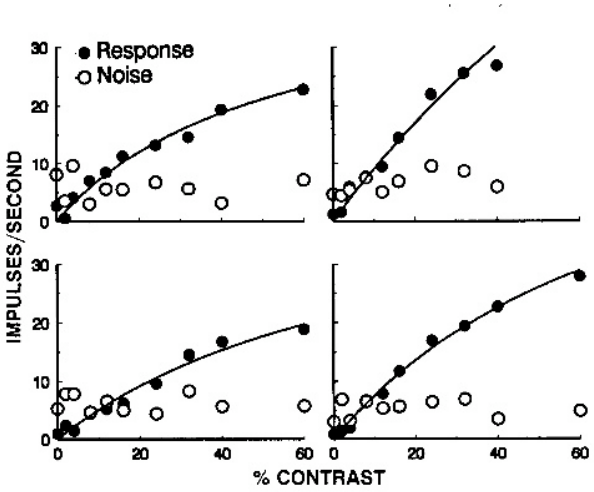
• information theory and statistical decision theory

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.

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



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 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)

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 most of this course.

Integrate and fire model of the neuron

The integrate-and-fire model" of the neuron

One 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.

Let V(t), and s(t) represent the 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, and so 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 it's 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:

$$dV/dt=(1/C)\ s(t) \tag{1}$$

This equation 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 t1 and t2:

$$V(t)=\int_{t1}^{t2} s(t)\, dt \tag{2}$$

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):

$$\theta=\int_{t1}^{t2} s(t)\, dt \approx (t2-t1)s = Ts \tag{3}$$

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 firing is the reciprocal of the period

$$\frac{1}{T}=\frac{s}{\theta}=f \tag{4}$$

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.

In a moment, we will derive the relationship between stimulus input level and the frequency of firing for the "leaky" or "forgetful integrate-and-fire" model. First, we go over some more *Mathematica* basics, and then develop a few tools.

More *Mathematica: rules, functions, and derivatives*

Rules and defining functions.

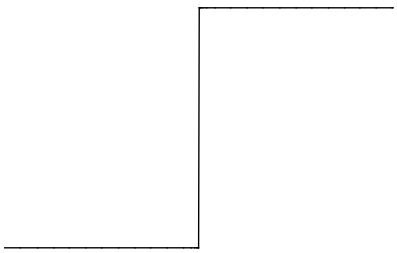
In *Mathematica*, you can define functions in terms of rules. One use of rules is to define functions over specific ranges. For example,

```
Clear[step] (* If you've been playing with the definition, it is a good
idea to clear it *)

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]**).

```
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**:

```
d + 2 e /. d->3 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[12]:=
```
Clear[x]
D[x^3, x]
```

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

In[14]:=
```
Integrate[3 x^2, x]
```

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

In[15]:=
```
NIntegrate[3 x^2, {x, 0, 2}]
```

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)?

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

You have probably noticed that this 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 a second order (second order because it involes a second derivative) equation for an oscillator. The acceleration of a mass on an ideal spring is proportional to the displacement: $\mathbf{d^2X/dt^2 = - k\ x}$. 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:

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

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

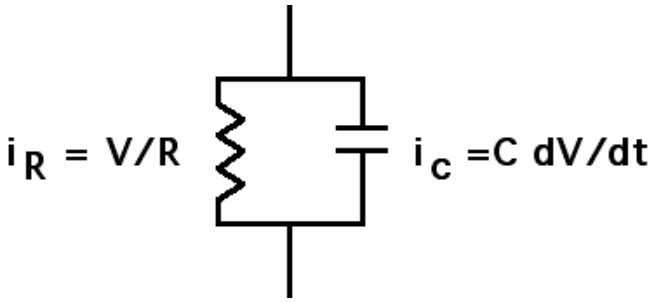
What if the initial speed is zero, but you pull back the end of the spring to -1?

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.

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.



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, or rearranging

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

(What if R->∞?)

We can find the solution using the *Mathematica* function **DSolv**e, given the initial conditions that at time t=0, the voltage is **a**. Let RC =1

In[16]:=
```
Clear[a,s];
DSolve[{V'[t] + V[t] - s/C == 0,
    V[0] == a}, V[t],t]
```

Out[17]=
$$\left\{\left\{V(t) \to \frac{e^{-t}\,(a\,C + e^{t}\,s - s)}{C}\right\}\right\}$$

After t seconds, the voltage reaches threshold, θ, and a spike occurs. We can solve the above equation in terms of t:

In[18]:=
```
Solve[θ == (a*C - s + E^t*s)/(C*E^t), t]
```

Solve::ifun : Inverse functions are being used by Solve, so some
solutions may not be found; use Reduce for complete solution information. More…

Out[18]=
$$\left\{\left\{t \to \log\left(\frac{a\,C - s}{C\,\theta - s}\right)\right\}\right\}$$

E is 2.718..., and E^x is the same as Exp[x]. As we saw before, the frequency is 1/time and is thus given by the following function:

In[20]:=
```
freq[s_,C1_,a_,θ_] :=
    1/(Log[(-(a*C1) + s)/(s - C1*θ)]);
```

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:

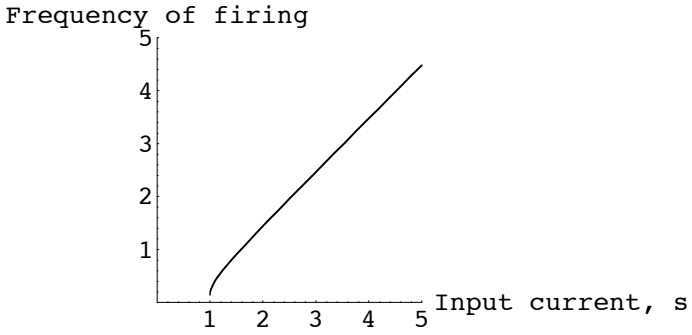
In[21]:=
```
Plot[freq[s,1,0,1], {s, 0, 5},
    PlotRange->{{0,5},{0,5}},AspectRatio->1,AxesLabel->{"Input current,
s","Frequency of firing"}];
```

Plot::plnr : freq(s, 1, 0, 1) is not a machine−size real number at $s = 2.08333 \times 10^{-7}$. More…

Plot::plnr : freq(s, 1, 0, 1) is not a machine−size real number at $s = 0.202835$. More…

Plot::plnr : freq(s, 1, 0, 1) is not a machine−size real number at $s = 0.424044$. More…

General::stop : Further output of Plot::plnr will be suppressed during this calculation. More…



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?

Plot passive response to an input of amplitude inputstep and duration stepduration

---

We take the solution to the leaky integrator for an intial voltage of a1=0 and an input current of s1 =inputstep for the range t<stepduration,

then include the condition for an intial voltage of a1=inputstep and an input current of s1 =0 for the range t>stepduration,

In[22]:=

```
inputstep = 1;
stepduration = 5.0;
a1 = 0; C1 = 1; s1 = inputstep;
a2 = inputstep; C1 = 1; s2 = 0;
r[t_] := (Exp[-t] * (Exp[t] * s1 + C1 * (a1 - s1 / C1))) / C1 /; t < stepduration
r[t_] :=
 (Exp[-(t - stepduration)] * (Exp[(t - stepduration)] * s2 + C1 * (a2 - s2 / C1))) /
   C1 /; t > stepduration

Plot[r[t], {t, 0, stepduration * 3}];
```

Next time

Generic neuron model

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

  This "connectionist" model is one of several abstractions that we saw above.

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

References

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.

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/nctoc.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]
```

```
{{False, True}, {True, True}}
```

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"}}]
```

```
truthtable[Or]
```

a	b	output
False	False	False
False	True	True
True	False	True
True	True	True

■ 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]
```

a	b	c	output
False	False	False	False
False	False	True	False
False	True	False	False
False	True	True	False
True	False	False	False
True	False	True	False
True	True	False	False
True	True	True	True