Pulse Computation

by

Gill Andrews Pratt

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

at the

Massachusetts Institute of Technology

December, 1989

Signature of Author

Department of Electrical Engineering and Computer Science

Certified by

Thesis Supervisor

Accepted by

Chairman, Departmental Committee on Graduate Students

JUN 06 1990

LIBRARIES ARCHIVES
Pulse Computation

Gill Andrews Pratt
Massachusetts Institute of Technology
December, 1989

ABSTRACT

This thesis considers the design of a computer that represents information with impulses instead of the usual binary signals. In such a machine, computation is effected through the fine grained interaction of continuously timed impulses arriving at the inputs of simple elements with finite life memory. The impulses themselves are atomic events, individually indistinguishable from one another. Thus, unlike binary symbols, the "meaning" of any impulse is given entirely by its context in place and time.

We begin with a look at pulse computation in animal nervous systems, demonstrating how fine grained impulse computation can, and most likely is, put to use by nature. Some compelling evidence in conflict with the traditional activity based (frequency modulated) view of nerve cell coding will be presented, and the important role of conduction block as a computational process explored.

Because of the brain's incredible connectivity and the microscopic scale of its circuitry, modern neuroscience enjoys no lack of conflicting, non-disprovable theories about neural coding and computation. To elevate the theory of fine grained pulse computation from this morass, the evidence presented from nature will be followed by the constructive approach of demonstrative design. Given hard constraints on the characteristics of a basic device, we will address the task of designing efficient pulse circuits that implement useful functions. Methods for simulating, programming, and evaluating the performance of pulse circuits will be described.

The encoding of large dimensional input spaces into single fiber temporal rhythms is a major conjecture of the natural pulse computation theory, and we conclude with an automatic method for teaching artificial pulse systems to solve specific problems in single fiber rhythmic feature classification.
Table of Contents

Cover Page .................................................................................................................. 1
Abstract ...................................................................................................................... 2
Table of Contents ...................................................................................................... 3
Acknowledgments ...................................................................................................... 5
On the Matter of Style ................................................................................................. 6
Introduction ................................................................................................................ 7
The Relationship of Media and Method ...................................................................... 9
Natural Pulse Computers ............................................................................................ 13
  Introduction to Natural Neural Networks ............................................................... 13
  The Computation Properties of Neurons ............................................................... 16
  Physiological Recordings ..................................................................................... 26
  Correlative Effects of Anesthesia ........................................................................... 44
  Temporal Encoding of Visual Patterns ................................................................. 45
Artificial Pulse Computers ....................................................................................... 49
  Devices - The Integrative Blocker ......................................................................... 50
  Details of the Integrator ....................................................................................... 52
  Details of the Blocker ........................................................................................... 60
  Absolute Blockers in Series and Parallel .............................................................. 71
  The Simulation of Pulse Circuits .......................................................................... 84
  Training a General Rhythm Classifier .................................................................. 105
  Examples of Blocking Layer Performance ......................................................... 110
  Training the Coincidence Detector ........................................................................ 118
  An Example of Coincidence Layer Learning Performance .................................... 123
Conclusions .............................................................................................................. 130
References ............................................................................................................... 134
Appendix A: FROG-11 Recording Instrumentation Hardware .............................................. 136
Appendix B: FROG-11 FSM Code ......................................................................................... 149
Appendix C: Frequency Analysis of the Non-Retriggerable Blocker ........................................ 156
Appendix D: Determining Proper Training Statistics ................................................................. 162
  Noise in Digital and Pulse systems ......................................................................................... 162
  Transmission Rates of Interval Encoded Channels ................................................................. 169
  Power Considerations ............................................................................................................. 179
Appendix E: Terminology of Information Capacity .................................................................. 185
Appendix F: Listing of Simulator Event Kernel ....................................................................... 187
Appendix G: Listing of Absolute Simulator ............................................................................. 191
Appendix H: Listing of Exponential Simulator ......................................................................... 198
Appendix I: Listing of Back Propagator .................................................................................. 208
Acknowledgments

In the late 1960's, Steve Raymond and Jerry Lettvin first proposed that conduction block might be a useful computational mechanism in nerve. Lettvin had earlier found the first supporting evidence for non-FM encoding in the form of multi-channel timing patterns in the frog's dimming detector. Raymond built the first hunter circuit to measure post-depolarization threshold shifts under various conditions, and hypothesized on the possible role of this shift and its ability to decode Lettvin's multi-channel patterns. For these intellectual accomplishments, and for Jerry's and Steve's help with the development of my own ideas, I am in their debt.

Steve Raymond, in particular, by recalling to me the outlines of unpublished work he did with Robert Bobrow in the early 1970's, led me to the design of generalized two-layer networks for pattern recognition.

Jerry Lettvin, my thesis supervisor, was as wonderful a mentor as I could have desired. I can only hope that future students, with other advisors, find themselves as fortunate.

Cam Searle and Toni Knight, my readers, have also been good friends in the course of my career.

For Janey, my wife, I must express my appreciation and my love. She strengthened my will to complete this work and willed me strength to do it with cheer.

A number of students worked with me over the years on various aspects of pulsed systems, and they deserve both recognition and thanks:

<table>
<thead>
<tr>
<th>Name</th>
<th>Research Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chris Musil</td>
<td>A Programmable Interconnect and Recording Instrument for Small Neural Networks</td>
</tr>
<tr>
<td>Peter Schmidt</td>
<td>Transfer Characteristic of Neurons in a Pulse-Code Neural Network</td>
</tr>
<tr>
<td>Carlton Sparrell</td>
<td>A VLSI Implementation of a Pulsed Neural Network in CMOS</td>
</tr>
<tr>
<td>Jeff Thompson</td>
<td>Learned Pattern Recognition in a Simulated Neural Network</td>
</tr>
<tr>
<td>Rob Webster</td>
<td>An Interactive Software Environment for the Investigation of Pulsed Neural Networks</td>
</tr>
<tr>
<td>Rolf Wyss</td>
<td>Hardware Implementation of a Pulsed Neural Network</td>
</tr>
</tbody>
</table>

This research was made possible by funding from the Siemens Corporation and I.B.M., with the support of Michael Dertouzos, director of the MIT Laboratory for Computer Science.
On the Matter of Style

Many students, myself included, often find themselves struggling long and hard to decode a straightforward idea hidden in the all but straightforward language of a scientific article. After emerging victorious from several of these battles and having had the opportunity to judge objectively the true difficulty of the material just presented, I have become convinced that technical authors do the majority of their readers a disservice by taking the "high road" of scientific writing. In reality, the subject at hand is usually far less difficult in broad concept than one would assume from the intimidating manner of its description.

What a shame! Except for the purposes of archival reference, there is really little excuse for overly formal presentation.

Like all human beings, every scientific author experiences the desire to be appreciated for his or her work. We also like to think of ourselves as intelligent, able to describe ideas succinctly through the use of powerful notation. But if we are to be true to our reader's interests instead of our own, these forces must be resisted. Elegantly formal writing inspires awe, but it is often not the best path towards understanding. To be truly effective, writing must educate, not impress.

This thesis is presented in a developmental style intended to allow readers with little background in neurophysiology, neural networks, or computer architecture to understand the issues without difficulty. These three fields are still in their infancy, and since this work both challenges a fundamental assumption about how neurons are put to work by nature, and also how neuron-like systems might be put to work by us, it makes sense to discuss the fundamentals. If you are already familiar with them, I beg your patience, your perseverance, and your forgiveness. The new ideas are at the end of this thesis, but their justification is to be found throughout.
Introduction

Of all the design tenets shared by modern computers, the adoption of a discrete representation for computed and communicated values is the most universal. This is not surprising, for two reasons: First is human-kind’s long acquaintance with cardinal numbers, which no doubt began with the counting of our digits and has since led to the rich mathematics of logical functions and discrete automata. Second is our growing virtuosity at constructing transistor switches that can quickly sense and generate binary digital signals.

But not all computers are digital. Not so long ago, analog\(^1\) machines were more popular, representing in their continuous time varying signals the values of state variables in systems of simultaneous differential equations. The invention of the analog computer by human-kind is also not surprising, for like the digital machine that parallels our abilities to count, the analog machine is in keeping with our natural talents at gauging continuous magnitudes and forming continuous responses. That this is true can be found by interviewing any experienced analog engineer; one finds that they think of voltages and currents not only in terms of abstract equations, but in the same manner Goldilocks considered porridge and beds.

Regardless of this success in analog and digital computing, there is no reason to believe that all worthwhile models of computation follow “natural” human modes of thought. Most provoking (although by no means paradoxical) is that the computational mechanisms by which our very thoughts are realized may fail to fit the paradigms those same thoughts follow regarding computation. And this is also to be expected, because unless one engages in neurophysiological research, nowhere in the natural world is an environment found that would develop our intuition along the proper avenues.

\(^1\) The word analog originated from the close relationship of the voltages or currents in analog machines to the scalar quantities of some real world problem. These quantities were always continuous, so an "analog circuit" came to mean one that manipulated continuous signals, even if no real world analogy existed.
The pulse computing paradigm may justifiably be viewed as being midway between the analog and digital realms. A pulse, since it marks a discrete instant of time, is quantized in its existence just like a digital '1' or '0'. On the other hand, a pulse may arrive at any moment in continuous time, in the same manner an analog signal can take on any continuous value within a range of voltages or currents.

Thus, pulse systems enjoy a portion of the noise immunity of digital signalling while preserving some of the hardware efficiency more common to analog systems. This combination, until now, has been used only in communications systems (e.g. the No. 101 ESS telephone switch\(^2\)) but not in computation systems. This paper's purpose is to demonstrate the feasibility of efficient computation in pulsatile media. In particular, we will focus our attention on the task of rhythmic pattern classification, and show how the monostable blocking oscillator, a device not ordinarily considered a computational element, can become a powerful one when combined with combinatoric coincidence detectors and decay prone integrators.

---

The Relationship of Media and Method

The digital and analog computing methodologies have evolved, in part, because of their suitability to available computing and communications media. Of course, the reverse is also true, because these computing methods have provided very strong impetus for the development of their media. Each development feeds and is fed by the other.

But this inseparability of media and method also has a deleterious effect; it causes us to ignore competing strategies (or competing media) because they don’t match very well with present techniques.

As one might expect from this prelude, pulse computing happens not to be well matched to conventional electronic media. Conventional analog electronics has been designed to manipulate continuous time-varying voltages. This is done by lumped components with continuous input-output relations and memory characteristics. Analog signals are conveyed by continuous voltages or currents on passive wires.

Digital systems, on the other hand, have been developed to signal one of two symbols at high rates with absolute reliability. Here the match of media and computing paradigm is very highly developed, with two power supply rails (e.g. ground and + 5 volts) providing the binary alphabet of transmitted symbols. Simple switch devices (e.g. N and P channel MOSFETs in CMOS) provide the elegantly simple means by which the to-be-transmitted symbol is selected. As in analog systems, passive wires are used for communication.

The idea of conveying a time event is not directly handled by either the digital or analog techniques. Modulation can be used, but this involves unavoidable inefficiencies in signalling capacity. For example, time events are needed in digital systems as clocks for memory state transitions, and these clocks are usually conveyed by the rising or falling edges of digital voltage signals. But there are two types of edges in digital systems, and a pulse system requires only one. It is no great loss to signal a time event by reversing the state of a digital line, but the required receiving hardware that is sensitive to both rising and falling edges is clumsy to build. Clearly, such a receiver would fall far short of the design economy
found in a CMOS inverter.

Sending time events as rising, then falling, edges (i.e. digital pulses) also has drawbacks. If the pulse is made wide, a good deal of a signalling channel’s information capacity is wasted. If the pulse is short, the difference in the system’s response to the rising and falling edge may cause pulses traveling through a cascade of computation units to either increase in duration (wasting capacity), or decrease in duration (and vanish). Thus, unless pulses are re-timed at each stage (such as they were in DEC’s circa 1969 PDP-10), a digital system will not process pulses reliably.

The same arguments apply to the analog handling of pulses. Pulses, repeatedly processed by analog hardware, tend either to widen because of dispersion or, in certain non-linear systems, to become more narrow.

It is obvious, then, that to build properly a system that handles time events, the creation of the time event signal as an atomic entity must be basic to the hardware of the system itself. Of course, the flip side of this coin is that if we have a media that is well suited to creating (and sensing) such signals, a pulsatile mode of computation and communication is particularly effective.

Even ignoring references referring to radar and other unrelated fields, the number of books that have been published on “pulse” techniques is large. Unfortunately, this is misleading, because in the early days of digital logic, tubes were used to create flip-flops (called “binaries”) from which larger systems were built. Because of the large conduction power loss of these circuits and also the large mis-match of plate and grid voltages, tube flip-flops were usually triggered by AC coupled signals (i.e. pulses). Thus, the pulse became synonymous with a digital signal, even after digital signals were treated by their circuits as DC levels, not as transients.

Today, edge-sensitive logic design is looked down upon as being unreliable, mostly because if a transition is “missed” by a mis-designed circuit, the initial source of trouble is very hard to track down. DC state transition circuits are easier to design and debug.
An additional source of confusion has come from the communications industry. Because of their small hardware complexity, "pulse height" and later "pulse width" modulation techniques were used in early half-digital telephone switchers to improve signal to noise ratio, allow for time slot multiplexing of several signals on one wire, and improve reliability. As hardware costs declined, the advance was made to total A/D - D/A techniques where a digital code represents the instantaneous amplitude of a sampled waveform. The etymology being what it was, this method was called "pulse code" modulation even though pulses had very little to do with it.

Average pulse frequency techniques have also been used occasionally to construct pseudo-analog computers (e.g. the MIDADA) with a little economy in integration hardware. But here the pulse was being used only as a carrier, not as the fundamental token of information, and the time-relationship of individual pulses was not nearly as important as the average frequency or (time integrated) count.

Several varieties of unary computers have also been designed. These machines encode data so that successive (or adjacent) bits have equal, rather than exponential, weight, and can achieve hardware or reliability economies given certain design criteria. The equal bit weighting seems similar, at the surface, to the equal significance given pulses by the circuits to be described here, but unary machines are more appropriately considered conventional digital designs with a peculiar encoding technique rather than time-event based computation engines.

Finally, there are the "stochastic" digital circuit designs, where the properties of uncorrelated random events are used to effect continuous arithmetic functions with boolean gates. As with the MIDADA, in these systems it is the collective interaction of pulses

---

3 Peter Elias - personal communication
averaged over a long time period which is considered significant, not the individual pulse interactions upon which we shall concentrate.

Thus, the search for references on truly pulsatile computation systems leaves one disappointed. Except for rudimentary oscillators and frequency dividers, few artificial circuits have been developed that operate in non-trivial ways on the pulses themselves.

Research to be described here leads us to believe that nature has done better. Natural nerve membrane is an example of a fundamentally pulsatile communications media where pulses are the fundamental signalling tokens of computation. The day may someday come when an artificial medium as well suited to pulsatile signalling is at hand. It will be then that the results of what we describe here may be of some use.

Before we begin our investigation of natural and simulated pulse systems, we might address the issue of practicality and summarize the media and method link by wondering what might have happened if human-kind was gifted not with today's conventional electronic media, but instead with an efficient pulsatile media, one which could support neither base band digital signalling nor direct analog signalling. In this case, would not the chance of the digital or analog computing paradigms being developed before one based on time-events be small?

It is on this argument that we proceed.
Natural Neural Networks

Theoretic models for various parts of the nervous system abound, and our view of the role of the neural pulse is in disagreement with many of them. Despite this, the reader must understand that it is not our purpose here to model living systems. Instead, what we learn about what nature might do, and even what she most likely does, will be used as suggestive grist for our own engineering mill. Since systems using similar signalling tend at some level to be similar in operation, what we learn through these efforts will reflect back on neural models in animals. But the nervous system is very complicated, and good tests for the many reasonable models are hard to find.

We will thus resist the temptation to offer a complete model for any nervous process. Instead we shall explore a general engineering discipline for a particular mode of signaling, a mode that also happens to be used in living neural networks.

All-or-None

Today the prevailing model of the nervous system is an analog one. Despite this, natural nervous systems, at the electronic level, are really not analog circuits. Low level information in an analog circuit is conveyed by continuous signals. In the nervous system, however, information is conveyed for the most part by the transmission of electrical impulses, or action potentials, that demarcate discrete instants in continuous time. This is a singularly remarkable, but nonetheless often overlooked, fact.

It was in the mid 1800’s that the nervous system was first found to conduct information via the transmission of electrical impulses\textsuperscript{6}. It was later discovered that axonal nerve membrane, when sufficiently excited, became highly regenerative, undergoing a saturated, self-timed cycle of electrical depolarization\textsuperscript{7}. The depolarization of one membrane patch


serves as excitation for neighboring patches, and the spread of locally transient depolarizations propagates an electrical impulse along a nerve fiber’s axis. If started in the center of a fiber, two pulses propagate in both directions away from the excitation source. If started at one end, a single pulse propagates to the other end. The speed of propagation depends on the size and shape of the impulse’s rising edge, since it is at the exact instant this edge crosses the neighboring patch’s threshold that propagation takes place. Propagation speed also depends on the size of the fiber, due to the relationship of axoplasmic conductance and membrane capacitance to fiber size. Finally, the ever-changing threshold of the membrane patch being excited specifies to what level the exciting waveform must rise before propagation takes place, and thus also plays a role in determining propagation speed.

As a particular pulse travels down a fiber, it is recurrently subject to the high gain and saturation of many membrane patches. Because of this, the pulse’s “shape”, that is, its voltage waveform vs. time, upon emerging from the fiber’s end, is determined almost entirely by the terminal membrane’s characteristics. The shape is not significantly influenced by the details of the initial excitation that occurred a long distance away. Reliable transmission of shape-modulated pulses is impossible, since the fiber can only propagate pulses in an all or none fashion. One may conclude that the shape of a nerve pulse, in and of itself, carries no information\(^8\), and that only the time of each pulse’s arrival, in the context of other pulses (from the same and other fibers), conveys meaning.

Traditional analog neural models view the impulse trains carried by real neurons as noisy frequency modulations of relatively slowly changing variables, and it is these slow, continuous variables that are assumed to be the real data. In this view, the message of the modulating variables (i.e. its “activity”) may be recovered from an impulse train by taking the appropriate short term average of the measured firing rate. Neural circuits are correspondingly described in terms of their transfer functions in the mean firing rate domain.

Except in a few cases, this sweeping abstraction from individual impulse timings to mean firing rate is probably not justified. We will support this assertion about natural systems in three ways:

1) Existence of single pulse computational properties in the characteristics of nerve membrane

2) Supporting evidence from the statistics of electrophysiological recordings

3) Correlative effects of anesthesia noted by Raymond on consciousness and single pulse computational properties
The Computation Properties of Neurons

The traditional view of the neuron's computational powers concentrates on the action of the dendritic tree. It is on the dendritic tree that the axonal twigs of other neurons terminate. When a pulse arrives at a termination, it brings about the release of neurotransmitter which crosses the synaptic junction and, depending on the synapse type, either inhibits or excites the receiving membrane. The action of excitation or inhibition changes the impedance of either sodium (excitation) or potassium (inhibition) channels in the receiving membrane so that the balance of charge flow is disturbed and the local voltage changed. The equilibrium voltage of membrane at rest is near to that which would occur under maximum inhibition, so the effect of inhibition is to shunt away excitation rather than subtract an increment of voltage similar to that added by an excitatory event.

Spatial arrangement of the synapses is important, for two reasons. First, synapses which are near each other can, if activated nearly simultaneously, have cumulative (usually non-linear) effects on the receiving membrane. Second, because it takes time for inputs to propagate down to the root of the tree, the location of a synapse may be such as to give it maximum effect a certain time after the activation of other synapses. For example, inhibitory inputs, by their location, can have veto\(^9\) power during a certain time window over excitatory wave fronts traveling past them towards the nerve soma. Evidence of full depolarization in the input dendritic tree has also become less of an oddity\(^{10}\), leading us to realize that a great deal of non-linear local processing may be taking place in the individual branches of dendrites.

The role of these temporally and spatially sensitive excitatory and inhibitory mechanisms is to effect what we shall call "combinatorial coincidence detection", namely,

\(^9\) T. Poggio; C. Koch; and V. Torre: Microelectronics in Nerve Cells: Dendritic Morphology and Information Processing, MIT AI Memo 650, October, 1981.

the detection of a set of input conditions which have arrived within a certain window of time. The idea of the dendritic tree as a time-windowed combinatorial logic element is different than the traditional neural model because it occurs on a fast time scale, taking into account the phase of the pulsatile inputs (i.e. their fine-grained timing in relation to each other) and also allows, through the differential spatial placement of the inputs, for more complex functions than could be generated by a single homogeneous linear threshold device.

It is true that threshold systems, given a sufficient number of elements, can be used to compose any logical function.\textsuperscript{11} But with many inputs, a threshold gate has much more severe noise requirements for reliable operation than does a locally sensitive coincidence gate.

Before presenting evidence of fast combinatoric co-incidence detection in the optic tectum of the frog, we will show how fine grained input phase differences can be generated in a reliable fashion and thus allow the full logic power of the neuron’s dendritic coincidence detection circuitry to be put to use. To do this, we must explore what other computational powers are inherent in the neuron.

\textbf{Threshold Variation and Conduction Block}

Nerve fiber, like any real transmission media, has limited bandwidth. Every nerve fiber can carry pulses only up to a maximum frequency. This is the result of conduction block, a phenomenon which has traditionally been considered a parasitic limitation to neuronal signalling capacity.

For a time after the generation of a propagating impulse, the sensitivity of membrane to electrical stimulation is systematically altered. A good understanding of this change, and how it might be used to advantage by living membrane, may be gained through an understanding of the apparatus by which the threshold is measured. This device, the

threshold hunter\textsuperscript{12}, excites a test fiber with precisely timed pairs of impulses, sensing whether a regenerative response occurred due to the second impulse. The energy of the first conditioning pulse of each pair is always above the fiber's resting threshold, and is sure to cause a regenerative response. Over many trials, the energy of the second test pulse is adjusted by the hunter in a negative feedback loop so as to achieve a running average depolarization success rate of 50\%. The device thus "hunts" for the 50\% threshold by increasing the test pulse energy if the success rate is low, or lowering it if the success rate is high. The time interval, from conditioning pulse to test pulse, is then slowly swept while the hunter maintains a 50\% success rate, and a plot made of the feedback stabilized test pulse energy. Assuming the fiber to be initially at rest, this plot is a good indicator of the membrane threshold as a function of post-depolarization time.

Hunter plots show that the threshold varies in four distinct phases:

1. Immediately after depolarization, a membrane patch enters what is called an absolute refractory period, when no biologically feasible level of excitation will cause another depolarization.

2. The absolute refractory period is followed by a relative refractory period wherein the membrane once again becomes sensitive to reasonable levels of excitation, but of greater energy than were necessary when the membrane was at rest.

3. Next comes a surprise; a super-excitabile phase exists when depolarization can be triggered by a smaller impulse than when rested.

4. Finally, a longer period (measurable in minutes) of depression is entered, when the threshold is slightly raised, and then gradually falls back to its resting value.

A generalized threshold curve is shown below:

The magnitude and extent of each phase depends on the recent level of activity seen by the patch, and on the local geometry of the fiber (small diameter fibers exhibit longer refractory periods than those of large diameter). Systemic chemistry (e.g. concentration of $\text{CO}_2$) also has a profound affect on the threshold recovery curve.

**Regenerative Variations**

The depressive phase is brought about by the action of the nerve membrane’s ion pump. Each depolarization causes the loss of potassium ions from the inside of the nerve, and the import of sodium ions from the outside. To maintain the proper concentration of these two ions, the ion pump actively exports positive sodium ions. This pumping causes the membrane to become hyper-polarized, i.e. for its inside to become more negative with respect to the outside. Eventually, as positive potassium ions diffuse through the membrane toward the inside, the membrane’s voltage returns to normal. Since the membrane responds to positive (measured inside to out) voltages to induce depolarization, when hyper-polarized, the membrane is less sensitive to incoming pulses.

The effective pump rate depends on the extent of the nerve’s ion imbalance, and is thus dependent upon the history of recent firings. The more activity recently seen by the nerve, the higher the effective pumping rate, and the more profound the post-depolarization threshold depression.
But another effect also occurs. Since the membrane is hyperpolarized, when regeneration does occur, the swing in voltage potential is greater, and thus so is the energy of a propagating pulse. Thus, the effects of threshold shift are not simple, and even if we know the variations in energy required to depolarize a section of nerve as a function of recent activity, we must also account for the changes in excitation level.

Membrane Operations

When patches of membrane are joined to form a fiber, an upward threshold shift in a particular patch can block the conduction of certain impulses in the input train. Ignoring, for the moment, the mechanism by which impulse trains are first created, axonal nerve membrane is capable of two computational operations: the complete deletion of selected impulses and, as well, their variable delay. We consider the second of these operations first.

Delay

Delay can be brought about by several mechanisms. When marginally excited, a particular patch of membrane may teeter on the edge of depolarization for a time which, theoretically, has no bound. The result is a highly variable delay, which, in the upper limit, is equivalent to total conduction block. Deterministically controlling this delay requires extremely precise timing of incoming impulses and tight control of patch threshold. The delay’s high sensitivity to cross coupling and noise makes this metastable mechanism an unusable capability in real membrane. Because of the unpredictable results, the transmission of certain input rhythms may be specifically avoided when some of the rhythm’s pulses would arrive when the receiver’s blocking threshold is approximately equal to the normal input excitation.

A similar problem is addressed routinely in the design of reliable digital computers. Because a flip-flop compares the arrival time of its data and clock inputs, digital engineers insure by design that data inputs to flip-flops remain stable during the setup and hold time
windows surrounding the transition of the clock.

There are, however, two other delay mechanisms that deserve consideration. One is involved with the generation of impulse trains, the second, with the time-shifting of pulses as they propagate.

**Pulse Generation**

The generation and propagation of a neuron's impulses begins at the first axon segment beyond the axon hillock. This is where the output of the dendritic tree is considered as a continuous time-varying function, and compared against the current segment threshold. When the dendritic excitation exceeds the segment threshold, the segment fires, and its refractory threshold shift prevents its refiring for a time.

In the simple case of a constant (DC) dendritic input, the period of segment firing is given by the functional inverse of its post depolarization refractory threshold plot. This is done simply by drawing a line at the given excitation voltage and seeing when the threshold first drops below the given voltage. The first segment can thus be made to function as a non-linear voltage-controlled oscillator.

If we adopt an individual pulse view of a nerve's operation instead of a frequency modulated view, the types of waveforms we expect at the output of the dendritic tree will be anything but constant. On the contrary, they will be pulsatile in nature, but, since they arise (in a passive dendrite) from inactive processing in a capacitive media, they will have slower rise and fall times than axonal depolarizations.

Research on heart muscle\(^{13}\) has shown how post-depolarization threshold variation can interact with independent pacemaker excitation to account for timing irregularities such as the Wenchenback period. Despite the success of this model, we are still a good distance

away from a generalized theory of neural soma action and its effect on spike pattern generation.

**Pulse Entrainment**

Entrainment is a phenomenon described in good detail by Kocsis, Swadlow, Waxman, and Brill\textsuperscript{14} in 1979. Since conduction velocity varies with threshold and threshold varies in the course of time following the propagation of every pulse, the speed with which a pulse propagates past a point on a nerve fiber depends on the time which has elapsed since the last pulse was propagated. The lower the instantaneous threshold created by the last pulse, the faster the next pulse will be propagated. If we consider the differential propagation velocity as a wave function traveling along with a pulse along the fiber, there will be regions where the differential velocity is positive (i.e. the second pulse is gaining on the first), regions where it is negative (i.e. the second pulse is falling behind), and points at which it is zero (i.e. stationary).

A simple model of the propagation phenomena considers the fiber as composed of a series of segments where, in the style of dominos, each segment is excited by its predecessor and, in turn, is the excitor of its successor. Each of these sections has a post-depolarization threshold curve like that shown in the figure on page 19. For a more accurate model, these segments can be made indefinitely small in length and great in number, until the proper limiting behavior is found.

The manner in which one membrane patch excites the next is a complicated matter, involving both voltage and current, and the charging of the membrane capacitance through a non-linear conductance. However, if we assume the rising edge of the excitation in the region of interest to be a linearized ramp, then the propagation time for each segment will be

proportional to the instantaneous threshold. In this simple model, the propagation speed is then proportional to the inverse of the threshold, with the addition of a constant baseline velocity.

From the frame of reference of the original pulse, which, let us say, is traveling down a fiber at constant velocity, this waveform of second pulse velocity versus time becomes a waveform of second pulse velocity versus distance. Wherever this function has a velocity less than the speed of the original pulse, the second pulse will fall further behind. Wherever the speed is greater, the second pulse will advance.

For the curve shown above (which is the inverse of the threshold curve on page 19) there are three stable points of zero differential velocity (indicated as 1, 2, and 3).

The long-interval zero point (3) is effectively stable because the amount of lag increment is small, and the direction of increased lag makes the increment still smaller. The middle zero point (2) is meta-stable, as a pulse arriving a small time/distance from it will be driven away. The very short interval zero point (1) is truly stable, because there the slope of the threshold function traps the second pulse at a fixed interval from the first. This is the
stable entrainment interval.

Given a sufficiently long stretch of fiber, a non-linear arbitration is thus performed around point number 2. Intervals above this time are stretched to make them longer, while intervals below it become stabilized to the entrainment length.

For typical lengths of nerve fiber, this effect does not have much influence on the longer intervals; entrainments over a short distance requires either a high slope of threshold function or a very slow excitation waveform. Pulses that arrive well into the depressive recovery of a membrane patch have neither characteristic.

Pulse entrainment, like refractory meta-stable variable delay, is a seductive computational use of the threshold variation curve. But it is also a very difficult operation to describe precisely and to use in a drift-free manner. For our simple purposes, we will ignore its effects, save for accommodating to its existence so that it does not interfere with the computational operations we hypothesize.

**Conduction Block**

Post-depolarization threshold shift can also cause a nerve fiber to block conduction of insufficiently large excitations, and it is this aspect of the threshold curve we shall use.

Exactly which impulses of a particular train are blocked depends on the time series of intervals that make up the train, and, as well, on the membrane characteristics of the fiber segments through which the train has passed. A neuron is usually inhomogeneous in this regard, changing diameter and branching as well. To simplify matters, we can abstract the nerve into lumped components, namely, relatively homogeneous lengths of high *transmission safety* where almost all pulses survive, and nodes of low transmission safety where a significant fraction of pulses may be lost. This abstraction is not without biological basis: the terminal arbor of a nerve is a tapered tree of relatively untapered axon segments, forked off in twos from larger diameter parent segments. It is at the bifurcation of each large segment into two smaller ones that we place our nodes of low conduction safety, where they
act like transmission gates to pulses arriving at the parent segment’s end. Each gate, in fact, is nothing but the beginning portion of the smaller daughter segments, but if successful at its penetration, a pulse is very likely to make it to a daughter segment’s end. It is interesting, though perhaps inevitable\textsuperscript{15}, that neural teledendrons always fork into fibers of differing diameter. If the region of low conduction safety is performing a computation, as we believe, it is not done identically in both branches. Thus, depending on the past history of pulses which have been received, a particular pulse may invade one branch, the other, neither, or both. Because the branches are dissimilar, the bifurcations can serve as a complex parser of time domain encoding to a spatially differential pattern.

Physiological Recordings

Of the evidence suggesting that individual pulse (non FM) computation occurs in nature, the most obvious is the irregular timing of successive pulse generations on a single fiber. Pulse intervals recorded from the non-peripheral nervous system of living animals tend to have very high variance on an interval-to-interval basis.

In the traditional frequency-modulated view of nervous activity, this irregularity is considered to be a consequence of parasitic noise. The integrative properties of synaptic chemistry and post-synaptic dendritic membrane capacitance are called upon to low pass filter the “noisy” pulse train and extract the “real” message of slowly modulated average pulse frequency.

As discussed earlier, this view provides for little bandwidth of communication and fails to make use of the phase-sensitive receiving logic present in the dendritic tree.

Multiplexing

But such inefficiency is not a conclusive argument against slow FM encoding. To more convincingly dispute the FM model, we note that a slowly modulated FM signal can encode only one dimension of a scalar signal on any single physical channel. Multiplexing of the modulating signal can expand this dimensionality, but this multiplexing can occur at most at a rate on the order of the modulation bandwidth, which is significantly less than the carrier frequency. Certainly, a multiplexing scheme heaped on top of a slow-FM modulation does not select amongst its inputs on a pulse to pulse basis.

Thus, if through some method of data analysis, we discover several channels being multiplexed on a pulse to pulse basis in the interval train of a single fiber, a very direct blow is dealt to the theory of slow FM encoding. The exposition of such evidence is our purpose here.
The Frog’s Dimming Detector

The evidence to be presented here is not completely new. Frog retina dimming detectors were first characterized by Lettvin in 1959 as “off fibers”\textsuperscript{16}, i.e. those which fire when the lights are shut off. In the course of his experiments, Lettvin noticed a peculiar steady state firing pattern being generated by dimming detectors with the frog in complete darkness.

After the dimming fiber’s initial burst and period of silence (which lasted many minutes), a slow “muttering” would begin. The muttering had a particular rhythmic style, and this style changed under extremely small changes in incident light. Because the style was complex and seemed to indicate the history of received illuminance as well as the current light level, Lettvin suspected that the dimming detector’s irregular rhythm simultaneously encoded at least two channels of information.

To visually display these multiplexed channels, Lettvin devised the “CLOOGE” plot format, an acronym for “Continuous Log Of On-going Events”. In a CLOOGE plot, each pulse’s arrival is marked by a corresponding point on a two dimensional graph. The X coordinate of a point corresponds to the pulse’s time of arrival (measured from the start of the experiment), whereas the Y coordinate is given by the log of the elapsed time since the last pulse.

To actually record an hour-long experiment, Lettvin built a logarithmic time-sweep generator to deflect the Y axis of his oscilloscope, and a slow gear motor potentiometer to sweep the X axis. An intensification amplifier brightened the oscilloscope beam at the arrival of every pulse and also reset the Y sweep. Recordings were taken off the scope screen on Polaroid film.

The results of Lettvin's studies were published in "Multiple Meanings in Single Visual Units". They showed a banded structure in the CLOOGE plots, parts of which varied depending on absolute light level, and other parts with the level of adaptation. With a little experience on the part of the viewer, these two channels were separately readable from a single recording. Furthermore, the two channels were seen to be multiplexed on a pulse to pulse basis.

Lettvin's equipment, while clever in design, required great patience on the part of the experimenter. Little feedback as to the success or failure of each recording was available until hours of experimental work were completed, and once the experiment was finished, no re-adjustment of the apparatus could be made - one had to start all over.

Given the nature of Lettvin's equipment, it was not unreasonable to wonder if the banded CLOOGE structure was an artifact of amplitude thresholding or a too large electrode receiving spikes from multiple fibers. To answer this question, scanning electron microscopy was first used to photograph typical electrodes so as to ascertain their true size. The electrodes were made in standard fashion from drawn glass, broken and back filled with metal. The tips were then plated with gold and then platinum black. An SEM photograph of a medium-sized electrode appears below:

---

As can be seen, the tip has a diameter of approximately 3-4 micrometers and a very large surface area. To accurately record the height, width, and time of arrival of every pulse-shaped event received by the electrode, a new computer instrumentation system was designed and constructed, named the FROG-11. A detailed description of this hardware, including schematics and micro-code listings, is given in Appendix A and B. An essential feature of the new hardware was that the recording threshold could be set very low, causing a great deal of noise to be recorded, but more importantly, allowing no significant pulses to escape detection. Further thresholding to filter out low amplitude noise and to select a single fiber when several detectable fibers were present was done later by computer software, so that very little adjustment of the apparatus was required during the recording process of an experiment\(^\text{19}\).

Roughly a dozen new recordings were made with this hardware, and the results were in agreement with those in the Multiple Meanings\(^\text{20}\) paper. Indeed, the CLOOGE plots were remarkably similar to those made with Lettvin’s equipment. The two best recordings were made in complete darkness, and are presented below. Due to time constraints, experiments with changing illumination were not undertaken.

The software thresholded, but otherwise raw, data is given on the following pages by three plots. The X axis on all three is experiment time, measured in minutes. The bottom plot displays pulse height, while the middle gives pulse width (actually 50% of the fall time - see Appendix A). The pulse width plot’s Y-axis uses a logarithmic scale, in units of seconds. The purpose of these two plots is to guarantee that the recording is in fact that of a single nerve fiber, and not two. A two unit recording, for example, could have explained Lettvin’s data without any multiplexed use of a single fiber. With consistent heights and widths,

\(^{19}\) The need for record-time adjustment was later reduced even further by the use of a digital audio tape recorder that recorded the entire session (sub-threshold noise and all) in complete detail at a 48 KHz sampling rate. This tape could be re-played into the detector as many times as needed.

however, we can be certain that only one fiber is being recorded.

The top plot on each page is in the format of Lettvin’s CLOOGE plots, and displays the logarithm of pulse interval (in seconds) versus the logarithm of experiment time.

The first recording (named tectum) was made from the supposed termination of the optic nerve at the optic tectum, not from the optic nerve itself. The tectum is located at the top of the frog’s brain and shakes visibly in synchrony with the beating of the heart. This shaking, in combination with the tectum’s gelatinous consistency, makes for a more slippery place to rest an electrode than the nerve. The pulse amplitude of tectal recordings correspondingly tends to vary significantly over the course of an hour long experiment. Nevertheless, in the pulse height plot below, we clearly see one fiber being approached and then slowly drifted away from by the electrode. The clipped amplitude in minutes 10 through 38 of the recording is due to an overdriving of the FROG-11’s A/D. This had no other undesirable effect on the experiment, however, because the noise was so low and no other unit rose above the software threshold of $\frac{1}{2}$. Because of the high quality of the recording, no width thresholding was used.
The second recording (nervé), shown below, was taken directly from the optic nerve, and the pulse amplitude drifted much less.\textsuperscript{21} The software amplitude threshold for this data was set at .35, and as before, no width threshold was required.

\textsuperscript{21} I am indebted to Arthur Grant for making this recording, which was digitally taped and later replayed through my detection hardware.
What the Plots Show

The two CLOOGE plots are quite different, although both show the banded structure first noted by Lettvin. Both recordings were made in complete darkness, but it must be remembered that in each, the frog’s eye was illuminated just prior to recording. Thus, a degree of adaptation related variation can be seen in the CLOOGE plots. This time correlative variation is minor, however, in relation to the consistency of the interval probability spectrum throughout the experiment, and we thus concentrate our analysis on the distribution of interval times across the entire experiment.

Direct judgments can be made about interval densities by viewing the CLOOGE plots (perhaps with a defocused eye) from a distance or end-on, but some simple computer analysis makes the analysis significantly less subjective. To derive a smooth probability density function for the recorded samples, a normalized discrete cumulative distribution function (CDF) was first formed. The entire experiment’s intervals were sorted in numerical order, and a curve with successively incremented Y values was drawn. Since the recorded samples did not occur at even interval spacings, the X co-ordinates of this curve were not evenly spaced. Furthermore, because of the finite sample size (6451 points for tectum, 4225 points for nerve), this raw CDF curve was somewhat jagged.

In the next step, the raw CDF curve’s X co-ordinates were converted to base 10 logarithmic measure and a gaussian curve with a standard deviation .5 percent of the (logarithmic) X range was convolved with the raw CDF. The convolution was carried out at 500 evenly spaced points along the logarithmic axis, and took into account all data points lying within 4 standard deviations of the mean. The smoothed results were normalized with respect to the number of considered points, which yielded a smooth curve with even X co-ordinates and roughly 10% the number of data points as the raw data. The raw (left) and smoothed (right) curves for the tectum recording are shown below:
As can be seen, the narrow width of the filtering function created very little distortion.

Successive sample differences were then taken of the filtered CDF to create a probability distribution function (PDF) of the data such that

$$\int_{x_1}^{x_2} PDF(x) \, dx = CDF(x_2) - CDF(x_1)$$
Below are the CDF and PDF curves of the second recording, *nerve*:

Because of the small sample size, the high-frequency ripples visible in these PDFs are not statistically significant. These ripples could have been smoothed out with a wider filtering function, but this was not done so as to preserve the similarity of the raw and smoothed CDFs. Since a PDF is accurate only insofar as its definite integral indicates actual probabilities, this method of filtering the CDF and then differentiating yields a more accurate
(though less smooth) result than does a histogram-type convolution of a gaussian with the actual distribution of sample points (i.e. the defocused vision method).

As a final note on filtering, the small spike visible to the left of the peak of nerve's PDF (located at about .004, .2) is an artifact of the discretely calculated smoothing function interacting with the extremely sudden rise of nerve's raw CDF.

**Similarities and Differences**

Both recordings have a bimodal distribution, but they are also quite different. In the first (tectum), the sharp peak occurs in the long intervals (about 1 second) while in the second (nerve), the sharp peak occurs in the shortest intervals (about 50 milliseconds). This first observation seems to indicate that the recordings came from two very different types of fibers, and in our next section we shall see that these two recording differ in a more profound way.

Also of significance, we believe, is the depth of the notch between the tall peak of each PDF and its neighboring hill. Such a significant "forbidden zone" is indicative of discrete treatment (e.g. sorting) of the incoming signal by a receiver. Such sorting is what we believe to be the function of axonal arborization.
Joint Probabilities

Now that we have had a look at first order statistics, a natural question arises: is a more complex structure also present?

Any history dependent receiving process based on decaying thresholds must by nature consider intervals in the recent past more significantly than those of the distant past. Thus, it makes sense for us to first look for structure in second order joint probabilities. We wish to know how the interval probability distribution changes as a function of the preceding interval. If no dependence is found, then because of the receiver's structure, it is unlikely that significant dependence exists in higher order statistics. Furthermore, as the dimensionality of the joint probabilities under question increases, the amount of data required to give a meaningful answer goes up exponentially. Thus, our data set, which was not large for one dimensional analysis, will be barely adequate for two.

The *n-gram* scatter plot, commonly used as a means to establish the merit of a pseudo random number generator, can give us our first look at the joint probability distribution. To generate a two dimensional (digram) plot, a sliding window of length 2 was passed over the sequence of recorded intervals, and each pair of co-ordinates so generated was plotted as a single point. Each interval in the original recording thus appeared twice, once as X co-ordinate and once as Y. In the plots below, the X co-ordinate represents the earlier interval, whereas the Y co-ordinate represents the latter:
Like CLOOGE plots, digram plots are hard to objectively read, particularly because the first order statistics establish a bias which tends to mask the second order effects. Nevertheless, one effect can be read directly from the scatter plots which will not be apparent further on. This is the matter of symmetry. The tectum recording is very symmetric about the line $X = Y$ whereas nerve is symmetric to a lesser extent. This means that the order of the paired intervals is important as well as their combined values, and for nerve, short intervals are almost never followed by intervals from the sharp short-interval peak (4 ms) whereas a variety of other intervals are often followed by a 4 ms interval. The inhibition of a third pulse soon after two quickly successive firings is most likely an effect of accumulating depressive block.
First Order Normalization

To see the second order statistics more clearly, we can sort the interval pairs into bins dependent on the first interval. To achieve significant sample sizes, decade width bins were used, and in the same manner as was done for the first order statistics, a smoothed CDF and corresponding PDF plot were drawn for the second intervals.

The joint CDF and PDF for *tectum* are shown below. Three decades of previous interval are given, and the decade range is indicated to the right of each trace.
Each CDF curve's Y axis was normalized from 0 to 1, and the PDFs all drawn on the same scale (indicated by the left hand bracket). Thus, the bias of first order statistics was removed from the Y scale of these plots. From top down, the number of sample points included in each plot was: 747 (1 - 10 sec), 2554 (.1 - 1 sec), and 3076 (.01 - .1 sec). 74 interval pairs fell outside the three decade range and were not plotted.

As can be seen, the 1 second peak is slightly broader for the longer previous intervals, although the effect of this should not be overly stressed. The area underneath these peaks is not so different, as can be noted on the CDF plots. In all, the three curves are remarkably similar.

Such similarity did not, however, exist in the nerve recording:
The sample sizes for this plot are: 1404 (1 - 10 sec), 1175 (.1 - 1 sec), 503 (.01 - .1 sec), and 1111 (.001 - .01 sec). 32 interval pairs were out of range and not plotted.

Clearly, *nerve* shows very strong dependency on previous interval whereas *tectum* does not. This disparity was an unexpected, but quite welcome, discovery of this research.
Physiological Recordings - Conclusions

The strong lack of adjacent interval dependency in the tectum recording compared to the nerve recording leads to several conclusions:

1.

The two recordings do not, in fact, both represent the activity of dimming detector fibers of the optic nerve. Specifically, the tectum recording is most likely a measurement of the activity of active dendritic bulbs firing post-synaptically due to the activity of several optic nerve fibers.

Arthur Grant\textsuperscript{22} first demonstrated an intriguing variance in tectally recorded bug detector pulse shapes that could not be explained if tectal recordings were identical to those of optic nerve axons. The statistical evidence presented here is in agreement with this result, but for dimming detectors.

2.

Adjacent interval dependency would be expected if the time constant of integration for synaptic inputs (and their processing by whatever dendrites are present) is long compared to the average period of firing. In other words, strong adjacent interval dependency is expected in cases where a voltage controlled oscillator’s input changes slowly compared to its threshold modulation. Lack of such dependency indicates that the controlling input consisted mostly of fast transients.

Optic nerve fibers are the axons of retinal ganglion cells and as such, directly represent the firing of those cells caused by continuous inputs. The adjacent interval dependency seen in the nerve recording, by which a short interval prohibits another succeeding short interval,

is in keeping with a reasonable model of low frequency ganglion cell excitation and fast block. It is quite natural to expect the patterns from these cells to exhibit strong second order statistics because each firing causes a significant shift in threshold which influences the time of next firing under the slowly changing inputs given by the circuitry of the retina.

Not so for the coincidence detection circuitry of the hypothesized post synaptic dendritic bulb, which takes pulsatile inputs from several fibers and fires under conditions of coincident excitation. Coincidence is unlikely to occur at as high a rate as nerve firing (which is corroborated by the first order statistics shown previously), and when it does occur, each coincidence is not predictive of the time of the next. Hence the lack of adjacent interval dependency in the tectum recording makes sense.

If tectal recordings are indeed indicative of post-synaptic coincidence detection rather than optic nerve activity, then a good means is at hand for accessing both input and output, and eventually understanding the preliminary circuitry of the frog’s visual tectum. This seems a ripe area for future research.

What is clear now, from the independence of successive tectally recorded intervals, is that dendritic computation is occurring on a fast time scale. Per-pulse coincident computation, rather than integrative computation on average firing rates, is taking place.
Correlative Effects of Anesthesia

Steve Raymond, who first brought history dependent conduction block under quantitative investigation with the invention of the double pulse threshold hunter, has continued the study of the physiologic factors influencing conduction block.

Most exciting in his recent work is the discovered correlation between the action of systemic anesthetics on consciousness and their specific effect on the depressive phase of conduction block.

The actual method by which anesthetics affect consciousness has always been somewhat of a mystery. At clinically relevant concentrations, both the threshold and latency of neuronal membrane do not increase sufficiently (compared to hypothesized statistical factors) to explain loss of consciousness. Raymond has found, however, that anesthetics have a profound effect on the modulation of threshold following successful impulse transmission. Specifically, the depressive phase of post-activity threshold oscillations is strongly suppressed when anesthetic concentrations surpass the levels where consciousness is lost.

Thus, there is reason to believe that history dependent conduction block may be essential for sustaining consciousness.
Other Evidence from Nature - Temporal Encoding of Visual Patterns

In 1987, Barry Richmond, Lance Optican, Michael Podell, and Hedva Spitzer\textsuperscript{23} reported the results of post stimulus activity histograms recorded from monkey visual cortex. 64 Walsh pattern stimuli were presented, representing a linearly independent component space for arbitrary 8x8 element pictures. A post-stimulus pulse histogram was compiled from 8 presentations, and principal component analysis then used to quantify each histogram as a small dimensional vector representing, in compact form, the temporal probability signature of each post-stimulus response. Information theoretic measures were then made on the specificity of these vectors compared to scalar measures of pulse count (totalled for a short time after each presentation). As might be expected, the information encoded in the temporal probability signature was significantly (2 - 3 times) larger than that encoded by the simple pulse count measure.

Richmond and Optican's work is very supportive of the "time coding is important" notion advocated by this thesis. However, some critical points regarding their work should be noted. The use of the "orthogonal" Walsh stimuli in their study is misleading, because nowhere is it assumed (or proven) that a response to a linear sum of two Walsh patterns is the linear sum of individual responses (i.e. the visual system is anything but linear, so investigating responses to linearly independent images is not necessarily predictive of the responses to arbitrary images). Nevertheless, the 64 Walsh patterns did form a broad visual space for their study, and this is all that was really required.

Also not demonstrated is whether temporal encoding is actually used by the visual system, or is rather a mere correlative phenomenon. As the weather changes outside of one's home, the types of sounds one is likely to hear emerging from the hallway will vary - the sounds of coats, umbrellas, etc ... However, unless starved of other information, we don't use

\textsuperscript{23} Richmond; Optican; Podell; and Spitzer: Temporal Encoding of Two-Dimensional Patterns by Single Units in Primate Inferior Temporal Cortex, Journal of Neurophysiology, Vol 57, No. 1, January 1987.
those sounds as a means of gauging the weather outside; we look out the window.

As shown below, the individual pulse trains that Richmond and Optican recorded from each single presentation are extremely noisy compared to the smoothed average probability distributions:

It is hard to imagine a signature programmed detector being any good at detecting which visual pattern was presented from the pulse timings of a single trial. Nevertheless, there may be finer structure in the raw pulse trains than is given by the trial-averaged signature waveforms, and it is possible that a more sophisticated temporal parser would be able to accomplish the recognition task.

Why this is so may be seen by considering the following example, from the structure of English words. Let's say we wanted to build a detector of all words that had the suffix "ing". Here are several examples:

motorboating
sailing
rowing
thing
ping
Now let's say we made a map of individual letters to binary symbols such that the letters 'i', 'n', and 'g' were all assigned the value 1, and all other letters were assigned the value 0. If we then took a post-start-of-word average histogram of all words suffixed with "ing" and all words not suffixed with "ing", we would see the following signatures:\(^{24}\):

![Diagram of histogram graphs]

The X axis of these graphs denotes letter position, while the Y axis gives the probability of finding an 'i', 'n', or 'g'. The two signatures are distinctly different. But does this mean we could build a machine that used these signatures to tell us if any particular word ended in "ing"? Of course not, because the process of creating the signatures significantly smears the information we seek.

In the neural timing domain, the process of multi-experiment post-stimulus histogramming destroys the contextual information of single trial rhythm. Certainly, commonalities in rhythm will bring bias to the multi-trial histograms, but a great deal of detail is lost.

In the second half of this thesis, we will look at systems that automatically learn to recognize context dependent temporal patterns. This naturally suggests a wonderful experiment, namely that of taking the raw spike train data from an experiment like

---

\(^{24}\) These signatures were computed from a 250,000 word dictionary, of which 5,533 words had the suffix "ing".
Richmond’s and Optican’s, and attempting to train an artificial detailed rhythm pattern classifier in their discrimination. Success in this venture would prove not only that temporal modulations exist in visual cortex, but that different visual stimuli causing varying single trial pulse rhythms can be discriminated amongst using very simple properties of nerve membrane.
Artificial Pulse Computers

Having examined evidence in support of natural fine grained pulse computation, we now turn our attention to the design of artificial pulse computers. Specifically, we investigate the construction of a machine that can learn to classify temporal firing patterns.

The most powerful property of this machine will be its ability to learn by example. The classifier will be able to learn to recognize a subset of a digram space, to detect a particular Optican and Richmond style signature, or to classify a set of input timings whose only commonality is that they are denoted as belonging to the same group. The classifier will be the temporal counterpart of a digital ROM, but with automatic term minimization.

We start with the specification of a basic hardware device, the Integrative Blocker. After describing its parts, we’ll attempt to analyze its precise temporal input/output relationships. Some progress will be made, but the task will be seen to be quite difficult. As an alternative, we’ll look at strategies for simulating the device’s behavior, and find that through the use of simulation, the classifier’s design can be accomplished by methods of automatic learning.

This idea is very powerful, because for a given recognition problem, we won’t be able to easily deduce a specific network that will do the job, but we will be able to apply a simple learning algorithm to a general network so that the network trains itself to accomplish the desired task.
Devices

The selection of a basic device for the construction of any computation system has great influence on the performance and efficiency of the final design. Unfortunately, the measures of merit for a complete system are different than those used to evaluate a single device, and because many devices are universal, an a priori choice is difficult to make. Fortunately, here our choice is limited to those devices that are functionality isomorphic to natural nerve membrane. Taking the "minimalist yet functionalist" approach, we will strive to endow our basic computational element with those characteristics of nerve membrane shown to be important in previous chapters.

Short Term Memory

A stringent constraint on our basic element's design will be a finite limit on memory lifetime. Ordinary digital state devices (such as flip-flops) have functionally infinite lifetime memories. While some type of long term memory is obviously required in any computer to provide for program storage, our computing system will effect such storage through changes in hardware connectivity rather than internal state.

As an example of what our systems won't contain, consider a binary memory module, much like a digital set-reset flip-flop. Two pulse inputs could be used for the set and reset functions, and a third used to sense the state of the device. A machine composed of these elements and the co-incidence detectors described below is trivially equivalent to a digital state machine, and is not interesting.

Physiological nerve cells are believed to completely reset their electronic state under conditions of deep surgical anesthesia. Despite this, we know that long term memories survive surgery. Thus the nervous system must store long term information in configurational changes, and this fact, in large part, is why we shall limit our artificial system's device state to short term storage.
This means that every part of our pulse computer will, if starved of input for a long enough time, respond identically as if given a stream of data and then starved again. No local long term state memory will be permitted.

The Integrative Blocker

Our basic element is called an integrative blocker, because it executes a simple model of stimulus integration feeding a pulse generator that undergoes conduction block. It must be understood that this device, while it does have several inputs, is not meant to model an entire neuron. Rather, each unit models the role of an active membrane segment. Although the conventional model of dendritic input arborizations is a passive one, the passing of information by action potentials within our simulated “dendrites” will not hamper their power significantly. This is because our notion of the dendritic tree’s role is one of combinatoric co-incidence detection, and pulsatile events do a good job carrying information about coincidense.
The Integrative Blocker

As shown below, an integrative blocker is made up of two components: an input integrator and an output blocker:

The integrator takes several pulsatile inputs and produces a continuous time-varying output signal \( I(t) \). The blocker compares \( I(t) \) against its internal threshold \( B(t) \), producing an output pulse whenever \( I(t) > B(t) \). By side effect, the production of the output pulse modifies \( B(t) \), making the blocker refractory for a time to further excitation by \( I(t) \). After each pulse is produced, \( B(t) \) decays back to its resting value according to some specified relaxation function.

The Integrator

Each integrator input wire has associated with it a “synaptic” weight \( W_i \) that specifies the influence of an incoming pulse on the integrator’s excitation or inhibition conductance. Interconnection wires have specifiable constant delays, but are otherwise passive. This lumped approximation is necessary to limit computer simulation time to reasonable values.

An electrical model for the integrator is shown below:
In this and all further diagrams, resistor values (such as $G$, $W_{\text{excite}}$, $W_{\text{inhibit}}$) indicate a resistor's conductance, the inverse of its resistance.

This model is a simple attempt at capturing the essence of ion channel controlled conductance. Input pulses do not affect the integrator's state voltage directly; they only cause changes to the conductance of the two variable resistors. Pulses arriving on input wires with positive weights add that weight to the excitation conductance, whereas negative weighted pulses have the negative of their weight added to the inhibition conductance. All pulses are considered to have identical duration, so that changes to integrator input conductance caused by any given pulse last for a fixed time. When no pulses are acting on the integrator, the excitation and inhibition conductances are 0.

Under conditions of maximal excitation the output can reach a maximum voltage of $I_{\text{max}}$, while under inhibition, the minimum voltage is $I_{\text{min}}$. The integration leakage $G$ is assumed to have much smaller conductance than the smallest non-zero input conductance, and its effect is ignored whenever $W_{\text{excite}}$ or $W_{\text{inhibit}}$ are non-zero. When $W_{\text{excite}}$ and $W_{\text{inhibit}}$ are both zero, the integrator's state value $I(t)$ decays exponentially towards $I_{\text{rest}}$ with time constant $I_c$. 

Single Excitation or Inhibition

Consider the effect of a single input pulse of positive weight $W$ and duration $D$ arriving at time $t = 0$. While this single pulse is active, the input circuit becomes:

![Integrator Circuit Diagram]

If the integrator is initially at a voltage $I(0)$, at the end of the pulse duration time $D$ it will be:

$$I(D) = I_{\text{max}} + (I(0) - I_{\text{max}})e^{-\frac{WD}{C}}$$

or equivalently,

$$I(D) = I(0)e^{-\frac{WD}{C}} + I_{\text{max}}(1 - e^{-\frac{WD}{C}})$$

Note that $1 - e^{-\frac{WD}{C}}$ is independent of $I(0)$, and thus can be viewed as the fraction towards $I_{\text{max}}$ that the integrator's state voltage is pumped at the end of each excitation of weight $W$.

If the single input pulse has negative (inhibitory) weight, $I(D)$ is pumped to:

$$I(D) = I(0)e^{-\frac{WD}{C}} + I_{\text{min}}(1 - e^{-\frac{WD}{C}})$$
Time-Weight Equivalence

An important characteristic of the integrator's structure is that two excitatory pulses, arriving near-simultaneously, have the same effect as if they arrived in coincidence. This can be seen by considering the effect of two successive positive weight pumps, $W_1$ and $W_2$, occupying time $t = 0 \rightarrow d$ and $t = d \rightarrow 2d$:

$$I(0) = I(0) e^{-\frac{W_1 D}{C}} + I_{max} \left(1 - e^{-\frac{W_1 D}{C}}\right)$$

$$I(2D) = I(d) e^{-\frac{W_2 D}{C}} + I_{max} \left(1 - e^{-\frac{W_2 D}{C}}\right)$$

or

$$I(2D) = \left[I(0) e^{-\frac{W_1 D}{C}} + I_{max} \left(1 - e^{-\frac{W_1 D}{C}}\right)\right] e^{-\frac{W_2 D}{C}} + I_{max} \left(1 - e^{-\frac{W_2 D}{C}}\right)$$

Multiplying this out yields:

$$I(2D) = I(0) e^{-\frac{(W_1 + W_2) D}{C}} + I_{max} e^{-\frac{W_2 D}{C}} - I_{max} e^{-\frac{(W_1 + W_2) D}{C}} + I_{max} - I_{max} e^{-\frac{W_1 D}{C}}$$

or

$$I(2D) = I(0) e^{-\frac{(W_1 + W_2) D}{C}} + I_{max} \left(1 - e^{-\frac{(W_1 + W_2) D}{C}}\right)$$

which is exactly the response to two overlapping pulses of weights $W_1$ and $W_2$. This result is actually obvious from the initial equations for $I(D)$ because a pulse of doubled duration is treated identically to a pulse of doubled weight.

The equivalence of near-simultaneity to true simultaneity is important, because it means that simultaneity is not a singular point of operation.

The Non-Equivalence of Excitation and Inhibition

Because the learning algorithm we will use expects uniformity between the effects of excitation and inhibition, excitation and inhibition must have complimentary but equally scaled roles. This is achieved by making $I_{min} = -I_{max}$. Since scale doesn't matter, we choose
\( I_{\text{max}} = 1 \) and \( I_{\text{min}} = -1 \). Given these definitions, let's investigate what happens when an excitatory input of weight \( W_1 \) is immediately followed by an inhibitory input of weight \( W_2 \):

\[
I(D) = I(0)e^{-\frac{W_1D}{C}} + (1 - e^{-\frac{W_1D}{C}})
\]

\[
I(2D) = I(d)e^{-\frac{W_1D}{C}} - (1 - e^{-\frac{W_1D}{C}})
\]

or

\[
I(2D) = \left[ I(0)e^{-\frac{W_1D}{C}} + (1 - e^{-\frac{W_1D}{C}}) \right] e^{-\frac{W_1D}{C}} - (1 - e^{-\frac{W_1D}{C}})
\]

As before, we multiply this out to:

\[
I(2D) = I(0)e^{-\frac{(W_1 - W_2)D}{C}} + e^{-\frac{W_1D}{C}} - e^{-\frac{(W_1 - W_2)D}{C}} - 1 + e^{-\frac{W_1D}{C}}
\]

or

\[
I(2D) = I(0)e^{-\frac{(W_1 - W_2)D}{C}} + 2e^{-\frac{W_1D}{C}} - e^{-\frac{(W_1 - W_2)D}{C}} - 1
\]

To calculate the effect of simultaneous arrival of \( W_1 \) and \( W_2 \), we transform the two conductance circuit to its Thevenin equivalent:

![Thevenin equivalent circuit](image)

Note that this Thevenin circuit has both a variable open circuit voltage and a variable impedance. Simultaneous excitation and inhibition \((W_{\text{excite}} = W_1, W_{\text{inhibit}} = -W_2)\) causes \( I \) to
change by:

\[
I(D) = I(0) e^{-\frac{(W_1-W_2)D}{C}} + \frac{W_1+W_2}{W_1-W_2} (1 - e^{-\frac{(W_1-W_2)D}{C}})
\]

The \( I(0) \) decay term of the near-simultaneous and simultaneous equations are identical. If we assume the fraction of pumping is small (i.e. \( \frac{D}{C} \) is small), then we can estimate \( e^x \) by \( 1 + x \).

Using this, the near-simultaneous excitation and inhibition becomes:

\[
I(2D) = I(0) \left[ 1 - \frac{(W_1-W_2)D}{C} \right] + 2 \left[ 1 + \frac{W_2D}{C} \right] - \left[ 1 - \frac{(W_1-W_2)D}{C} \right] - 1
\]

which reduces to

\[
I(2D) = I(0) \left[ 1 - \frac{(W_1-W_2)D}{C} \right] + \frac{(W_1+W_2)D}{C}
\]

The simultaneous excitation and inhibition case under the approximation \( e^x = 1 + x \) becomes:

\[
I(D) = I(0) \left[ 1 - \frac{(W_1-W_2)D}{C} \right] + \frac{W_1+W_2}{W_1-W_2} \left[ 1 - \left( 1 - \frac{(W_1-W_2)D}{C} \right) \right]
\]

which also simplifies to:

\[
I(D) = I(0) \left[ 1 - \frac{(W_1-W_2)D}{C} \right] + \frac{(W_1+W_2)D}{C}
\]

Thus, even for mixed excitatory and inhibitory inputs, as long as the pump fraction is small, we can expect similar behavior for pulse overlap and near-overlap. As before, this was actually apparent from inspection of the schematic diagram, because for small pump fractions, the integrator's state variable does not change much, and connecting a conductance to the capacitor for a short time is equivalent to hooking it to a current source. Thus, connecting a positive current source and then a negative one will yield the same results as hooking both up simultaneously.

In the actual operation of our recognizer, the pump fraction will be very large - a little over \( \frac{1}{2} \). Thus, we will not be able to assume identical behavior for different degrees of time
overlap of excitatory and inhibitory pulses. This will be solved in our architecture by never generating near-misses when we did not intend coincident behavior, and utilizing large pulse durations \( D \) to insure that intended coincident action occurs reliably despite unavoidable skew in arrival times. Because borders of forbidden arrival times will surround this coincidence window, in no circumstance will arbitration be necessary.

It is important to understand the difference between long pulse duration \( D \) and long integration time \( I_r \). Physiologically, the pulse duration refers to the time constant of the synaptic gap, whereas the integration time represents the decay of postsynaptic membrane voltage. The entire argument of fine grained pulse computation hinges on using small values for \( I_r \) (so that the action of individual pulses, rather than their average frequency, is felt). However, the engineering necessity of detecting coincidence in the face of skew will dictate a non-zero value for \( D \). This does not mean that \( D \) will be larger than \( I_r \). It does mean that the mathematical effect of pulses may be described by their quantized effect on the receiving membrane, but nevertheless a real system will require non-zero pulse durations not only because zero width pulses are impossible to make, but because non-zero pulse widths allow for the construction of real coincidence detectors.

**Integrator Leakage**

As stated before, real nerve membrane has a resting voltage close to the maximum inhibitory potential, and thus inhibitory inputs are more correctly viewed as having veto power over excitatory ones than having lasting effects of their own. For example, if a barrage of inhibitory pulses arrives at a patch of membrane, the membrane's state after the barrage is not much different than the resting state.

In keeping with this characteristic, but without attempting any sort of precise neural modelling, we set \( I_{\text{rest}} = I_{\text{min}} = -1 \). Thus, when devoid of input, the integrator decays from an initial voltage of \( I(0) \) according to:

\[
I(t) = (I(0) + 1) e^{-\frac{t}{I_r}} - 1
\]
Implementation of Logic Functions

In the manner of linear threshold logic\textsuperscript{25}, the weights of various integrator inputs may be so arranged as to evaluate certain boolean functions. It is assumed that the integrator will be feeding into a blocking device whose resting threshold is 0, thus, for large pump fractions, the criterion for triggering of the rested blocker is

\[ W_{\text{excita}} > W_{\text{inhibit}} \]

where, as stated before, \( W_{\text{excita}} \) is the sum of all coincidently active inputs with positive weights and \( W_{\text{inhibit}} \) is the negative sum of all coincidently active inputs with negative weights.

Threshold logic requires not only the linear weighting and summation of input values, but comparison of this sum against a variable threshold as well. In our circuits, this variable threshold is implemented by arranging for an input that is always active (i.e. has a pulse arriving) in coincidence with the threshold unit's evaluation time. Since this input can shift the above inequality, the blocker's resting threshold can be fixed at 0.

The Artificial Block

In earlier chapters, we saw that pulse transmission in real nerve membrane is accompanied by threshold modulation, and that this modulation may in turn bring about the block of future pulse transmission. In designing an artificial blocking device, we attempt to replicate this phenomenon, preserving some essential features while of necessity ignoring others.

Let's start with the most fundamental type of blocker, and with a familiar example from the world of hydraulics: the common flush toilet. The blocking action of a flush toilet is easy to describe - the flush toilet flushes successfully if one waits long enough since the last attempted flush.

The flush toilet is similar to nerve block because an output "event" is generated the moment the input "waveform" exceeds the blocker's internal threshold. In this case, the input waveform is the position of the handle, and the blocking function rises from some constant value to infinity for a fixed length of time. Here's a timing diagram of a typical flush toilet trigger event:

![Timing Diagram](image)

Because the blocking function rises to infinity, different gradations of excitation on the flush handle do not influence the chance of success, even late in the blocking period. In other words, when flushing is blocked, it is blocked absolutely. The real world imperfection of half-successful flushes is a different issue and can, for the present discussion, be safely
ignored.

Because it seems so simple, the absolute blocker will be our first topic of study. It will not, however, be our final chosen form of threshold curve. Instead, we will settle on a block whose threshold decays exponentially from a certain maximum towards a fixed resting value. Unlike the flush toilet, the exponentially decaying threshold will be capable of blocking small inputs when larger ones would have successfully caused regenerative response. This time-varying sensitivity to different magnitude inputs will allow more information about the integrator’s state to appear in the blocker’s output pulse train.

Exponentially decaying block is still much simpler than the blocking action of real nerve membrane. First of all, the exponential blocker has only one phase of response - a refractory one, whereas real nerve has three - refractory, super-excitabile, and depressed. Furthermore, in real nerve we saw that there were significant modulations to the threshold curve due to activity previous to the last regenerated impulse. Our simple exponential block, on the other hand, uses the same threshold curve after every trigger. The entire state of the artificial blocker is thus given by a single variable - the time of last trigger, while in nature it seems that at least three state variables are involved\(^{26}\). Although such complex behavior would give even more input history dependence to the blocker’s output, it was found that sufficient history dependence could be had by using simple exponential block, and a more complex model was not investigated.

Presented below is an electronic schematic for the simple exponential blocker:

The comparator triggers the pulse generator whenever the input waveform exceeds the voltage on capacitor C. The output pulse also causes the FET switch to close, raising the capacitor voltage to $B_{\text{max}}$. This, in turn, because $B_{\text{max}}$ is higher that any possible input, causes the comparator output to be de-asserted (go to 0), so that the pulse generator is no longer triggered (how long this process takes is not well defined, but it really doesn’t matter). Finally, after the pulse has ended, the capacitor voltage decays as:

$$B(t) = B_{\text{rest}} + (B_{\text{max}} - B_{\text{rest}}) e^{-\frac{t}{\tau}}$$

**Absolute Behavior in the Exponential Blocker**

As one might imagine, the interaction of the time varying integrator's output with the time varying input threshold of the exponential blocker is a complicated matter. Thus, we might ask if such complexity is necessary for building a general recognizer. This proved to be the case, although a very simple class of single interval discriminators can be constructed using only absolute blockers. Because these latter machines (unlike the general recognizers) can be easily analyzed, we will look at them first.

It should be noted that given appropriate input, an exponential blocker can be made to behave like an absolute one. Such action occurs when blocker inputs are normalized to a constant height (i.e. the integrator has only one input) and decay very rapidly (i.e. $\tau = 0$). In this case, the integrator has no function whatsoever and treats all pulses identically. Thus,
any blocking threshold over the normalized input pulse height is effectively infinite and absolute behavior is obtained.
Absolute Blockers

In this section we take a look at the function and powers of the absolute blocker.

Retriggerability

Because absolute block treats input signals as binary events, there is a choice as to the blocker’s type: a blocker can be retriggerable or non-retriggerable, depending on whether the threshold pump is derived from the blocker’s input or from its output.

In digital system form, the constant-time artificial blocker could be implemented as shown below:

![Diagram of constant-time artificial blocker](image)

As is indicated by the inversion bubble on its input, the 1-shot is triggered by the trailing edge of a pulse. Thus, if the 1-shot is quiescent, the output will be high during the high portion of an input pulse. At the end of the input pulse, the 1-shot is triggered, which causes the AND gate to block output pulses until the timing period has expired.

The issue of retriggerability concerns the sensitivity of the 1-shot during the blocking period. If the 1-shot is retriggerable, then additional input pulses, while not passing through to the output, will still causes a restart of the 1-shot’s blocking period.

The retriggerable blocker allows for a very simple operational description: a pulse is allowed to pass through only if an interval of silence greater than the blocking time has preceded it. For example, to its user’s usual dismay, the flush-toilet is a retriggerable blocker.
As another example, the retriggerable 1-shot function is also often used in computer systems for "watchdog" reset timers. System software recurrently performs some sort of systems health check and, if successful, triggers the retriggerable 1-shot. This is done with a period less than the 1-shot's timeout time, so that under normal operation the timer's (inverted) output is always "0". If the software detects a bad system state, or if the watchdog software itself fails to operate, the "fail safe" 1-shot completes its timeout period, and its (inverted) output goes to "1", causing a hardware system reset. In essence, the retriggerable 1-shot is a detector of long intervals.

Non-retriggerable 1-shots are much less commonly used in digital systems, but they are made for other applications. In the above schematic, a non-retriggerable 1-shot would create a non-retriggerable blocker, i.e. one where an input pulse occurring during the 1-shot's blocking time is not only prevented from reaching the output, but has no effect on the system's state whatsoever.

Natural neuronal block is mostly non-retriggerable. However, some retriggerable effects do occur as a result of blocked impulses, particularly when one branch of an axon tree is invaded, but the other is not. In this case, the electrotonic continuity of the two branches causes them to share a degree of threshold modulation.

A functional description of a pure non-retriggerable absolute blocker is difficult to state simply. Whereas the appearance of a pulse on the output of a retriggerable blocker said something very definite about the history of past inputs (i.e. that there haven't been any for at least so long), as shown in the following diagram, an output pulse may be emitted by a non-retriggerable blocker an arbitrarily short time after an input pulse has been received:
For the non-retriggerable blocker, it is the appearance of an interval on the output that gives an equivalent description of the input. An interval of time $I$ (where $I$, of course, must be greater than the blocking time $B$) indicates that there must have been a pulse on the input at the start of the interval, that there could have been pulses on the input for a time $B$ following this pulse, and that there were no pulses for a time $I-B$ following the time of possible pulses. Finally, the appearance of the pulse terminating the output interval must have come from an input pulse occurring at the same time:

A simple change in the schematic can make our blocker non-retriggerable no matter what type of 1-shot is used, because it insures that inputs to the 1-shot are suppressed during the blocking period:
In this configuration, the non-retriggerable blocker, unlike its retriggerable cousin, can be transformed into analog signal processing form:

Here, the pulse trains on the input and outputs are mathematically described as a sum of time-offset dirac delta functions: $\sum_{n} \delta(t - T_n)$, where $T_n$ is the time of occurrence of the $n$th pulse.

A short delay $D$ after each output pulse occurs, the one-shot is triggered. The impulse response of the one-shot is a box-car waveform lasting $B$ seconds:
It doesn’t matter that a real one-shot is a non-linear device, since only pulse trains with inter-pulse spacing greater than $B$ will be output by the blocker. For these signals, the oneshot performs just like a linear device with the impulse response shown above.

If a pulse has passed through less than $B + D$ seconds ago, the output of the one-shot will be 1, and all input signals will cancel in the subtractor. Once the blocking time $B + D$ has passed, the one-shot output will return to zero, and a new pulse will pass through.

The delay $D$ is necessary to take care of a sticky issue of causality. If $D$ is zero, a race occurs between the rising of the one shot’s output and the pulse which caused the one shot to fire in the first place. How the system should resolve such a race is hard to answer. An output pulse, in some sense, is a self-contradiction. To avoid the question, we assume $D$ to be a small, but non-zero, positive delay.

Since $D$ is greater than zero, the blocker is not quite ideal; it remains receptive for a short time following the transmission of a pulse instead of blocking immediately. But this is a problem only if we feed the blocker pulses separated by very short intervals, and for any minimum expected interval, we can make the blocker functionally ideal by simply making $D$ small enough.

The ability to represent the non-retriggerable blocker in a signal processing diagram leads one to consider analysis by frequency domain methods, with the hope that since periodic pulse rhythms can be considered as a summation of several simple periodic trains
with different phase shifts, a frequency analysis should help us see how absolute blockers transform periodic pulse rhythms. Unfortunately, the analysis is so tedious that this does not seem to be a fruitful approach. The form of the output usually must be assumed, and once this is done, one already has the answer. For the interested reader, a frequency analysis of constant interval behavior which proves that block does not occur at low frequencies is presented in Appendix C.

**Response to Constant Interval Pulse Trains**

The behavior of the non-retriggerable blocker for constant interval input pulse trains may be determined by intuition:

<table>
<thead>
<tr>
<th>Range of $T$</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B &lt; T$</td>
<td>Input</td>
</tr>
<tr>
<td>$\frac{B}{2} &lt; T &lt; B$</td>
<td>Input with every other pulse deleted</td>
</tr>
<tr>
<td>$\frac{B}{3} &lt; T &lt; \frac{B}{2}$</td>
<td>Input with every 2 out of 3 pulses deleted</td>
</tr>
<tr>
<td>$\frac{B}{4} &lt; T &lt; \frac{B}{3}$</td>
<td>Input with every 3 out of 4 pulses deleted</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

A plot of output frequency vs. input frequency looks like:
It is because of this aliasing behavior that vacuum tube blockers (called blocking oscillators) were used in the past as pulse frequency dividers.

As a contrast, the retriggerable blocker has the frequency transfer graph:
Absolute Blockers in Series

Natural axonal arbors contain several blockers in series, so we examine here what occurs when artificial absolute blockers are placed in series. Our artificial serial system will be labeled as follows:

\[ \text{In} \rightarrow \overrightarrow{B1} \rightarrow \overrightarrow{B2} \rightarrow \text{Out} \]

\( B_1 \) is the blocking time of the first device, \( B_2 \) the blocking time of the second.

For both retriggerable and non-retriggerable blockers, if \( B_2 < B_1 \), then the first blocker will prevent pulse intervals shorter than \( B_2 \) (and thus shorter than \( B_1 \)) from reaching the second blocker. Thus, the second blocker's blocking action will never be put to the test, and the behavior of the entire system will be identical to that of the first blocker in the string.

If \( B_1 < B_2 \), a situation in agreement with the ordering of blocking time in natural terminal arbors, things get more interesting.

We begin with the retriggerable case, comparing the action of a single blocker of blocking time \( B_2 \) to that of the cascaded string. The single blocker would allow to pass only those pulses that were preceded by a time \( B_2 \) of silence. Since \( B_2 > B_1 \), these pulses would also pass through the first blocker of the string system. Presumably, other pulses would also make it to the intermediate node, namely those with pre-silence periods between \( B_1 \) and \( B_2 \). It is clear that all pulses which would make it through the single \( B_2 \) system would make it through the series system, and that pulses which don’t make it past \( B_1 \) can’t emerge miraculously from \( B_2 \). Is it possible for some of the pulses with pre-silence periods between \( B_1 \) and \( B_2 \) to make it through as well? To do so would require the deletion by the first blocker of the pulse marking the start of the pre-silence period. This can indeed happen, as we see below:
The effect of a preliminary short-time blocker can also be seen on certain sequences in the non-retrigerable case:

**Constraint Propagation**

To determine which input sequences will generate a particular output sequence we can try an analytical method that propagates the constraint relationship imposed by a blocker in both directions on its input and output.

We will concentrate here on the non-retrigerable blocker, since that is the device with more complex behavior and the one that better corresponds to natural block. Working our way from output to input, we first note the constraint of \( B2 \) given an output interval \( I > B2 \) (shorter output intervals are impossible):
Note that two things have happened here. The original output interval created the input constraint of two pulses which must have occurred and the two hatched regions where a pulse may have occurred. In the reverse direction, since a blocker cannot put out intervals longer than its blocking time, the output has been constrained to have silence periods of length $B_2$ preceding and following the output interval.

The input of $B_2$ preceding the first output pulse has been left unspecified on purpose, because whether or not pulses are allowed there depends on the interval preceding the one given for the output - the possible input pulse cross hatch is indicated only when we know for sure that an input pulse, if it appeared, would not affect the output.

Since the input to $B_2$ comes from the output of $B_1$, and $B_1 < B_2$, this forward constraint further affects the signal on the intermediate point:

Note that the duration of the cross-hatched region depends not only on $B_1$ and $B_2$, but on $I$ as well. If $I > B_1 + B_2$ then this possible-pulse region reaches a maximum:
Back-propagating the constraint of $B_1$, we reach a description of possible inputs for the given string output. In the case of $I < B_1 + B_2$, this is:

The back stroked cross hatch indicates where $B_1$ is responsible for the block, whereas the forward stroked cross hatch indicates where $B_2$ did the blocking. When $I > B_1 + B_2$, the input conditions are the same as for $B_2$ alone:

The far right hand cross hatch region is long in both cases, but may be shorter due to the constraints of succeeding intervals. Thus, the effect of $B_1$ on the classification of inputs which will produce the same output is to possibly shorten the time of permissible, but
ignored, input pulses from $B_2$ to $I - B_1$.

Returning to our original example, we can see why the cascade handled the string differently than $B_2$ alone. Since $I$ was less than $B_1 + B_2$, the interval output through just $B_2$, which ignored the middle input pulse, did not allow its existence in the cascaded system. Thus, its existence forced the cascaded system to emit a different output interval (in this case, a very long one).

**Longer Cascades**

The important result of the preceding section is that, while series connections can shorten the interval of allowed, but ignored, pulses on an input stream, they do not alter the basic form of the input constraints (i.e. must-exist pulse, might-exist region, silence region, must-exist pulse ...).

We have seen that the length of the might-exist input region is $\min(I - B_1, B_2)$. Adding another blocker (say, $B_0$) to the input stage of blocking time involves two possibilities.

If $B_0 > B_1$, then, as discussed before, the series of $B_0$ and $B_1$ acts like a blocker of time $B_0$. If $B_0 < B_1$, then the new might-exist input region will have length:

$$\min(I - B_0, \min(I - B_1, B_2))$$

But since $B_0 < B_1$, $I - B_0 > I - B_1$ and:

$$\min(I - B_0, \min(I - B_1, B_2)) = \min(I - B_1, B_2)$$

Thus, for a given output pattern, a string with monotonically increasing time blocks from input to output can have all but its final two blockers deleted and still have the same input sensitivity. Since any particular input pattern will result in some output pattern, the behavior of such a string is transparently equivalent to that with all but the final two blockers removed. This was a result noted qualitatively by Raymond\textsuperscript{27} and Bobrow in their early

\textsuperscript{27} Personal Communication: "Fiddling with the higher blockers didn't seem to change things much"
simulations, but never formally explained.

But Wait a Moment..

Unfortunately, the above analysis ignores one subtle effect of two blockers in series, and that is the added blocking time allowed by the first blocker when a pulse is blocked by the second blocker. This effect comes into play for a small but significant fraction of the input interval rhythms.

Let us assume again that \( B_1 < B_2 \), and furthermore, that \( B_1 < \frac{L}{2} \). As long as these conditions hold, if an intermediate pulse arrives during the last \( B_2 - B_1 \) seconds of \( B_2 \)'s block, that pulse will not be blocked by \( B_1 \), but it will be blocked by \( B_2 \). At the same time, by starting a new block by \( B_1 \), the string is rendered insensitive to new input pulses for the next \( B_1 \) seconds. This is longer than the previously calculated zone of "might occur" input pulses:

It doesn't matter whether \( I > B_1 + B_2 \), only that \( I > 2B_1 \). For the sake of illustration, however, we'll use the case of \( I > B_1 + B_2 \). In the last section, we computed the following input constraint for the output interval \( I \):

But if a pulse arrives towards the end of \( B_2 \), we see things will be different:
Why didn’t our former analysis pick this up? The oversight lies in not handling the output to input constraint resulting from a region of possible output pulses. Since no such regions existed in B2’s output, no mistake was made there. However, the intermediate node had such regions, and while an interval of possible output pulses does correspond to an interval of possible input pulses, (as long as no other silence constraints apply), it also creates a possible expanded region of ignored input pulses.

The simple time-line language we have described is too poor to express this constraint properly, because the actual time-line of ignored pulses will depend on when actual pulses occurred in other “possible” regions of the time-line.

What we ultimately seek is some sort of a set description of the input rhythms that parse to a particular output. Unfortunately, this set cannot be described in terms of a single time line with our three interval symbols. The first element that is needed is an OR-operator to merge two possible time lines. Furthermore, a notion of the allowed shift of a time-line must be incorporated so that the “pulse somewhere in this region creates a possible pulse interval for so long after” notion can be expressed.

We also need the ability to pass the time-line description through another blocker further up towards another input and still be able to express the result in the same time-line language. The description must be universal, and the operation of the blockers upon it must be closed.
Finally, a simplifier of some sort must exist that reduces OR-ed descriptions of possible time lines at a single node into simplified form. This is a very difficult problem, because the simplest description may not be that found by a hierarchical merging of less simple constraints.

The more one considers this problem, the more apparent it becomes that an automatic propagation and reduction analysis tool would be of great assistance. With such a tool, the reduction rules could be written algorithmically and their application and completeness tested in a complex problem environment. The creation of such a automatic tool has been begun by the author, but it is apparent that this task is quite formidable.

**What a Time Line Constraint Propagator is Good For**

The automated tool just described has applications outside that of pulse computation. Because digital signals may be completely described by their times of transition, a constraint system that handles pulse events will, with small modification, be useful for the verification of timing in asynchronous circuits of digital logic.

Most exciting about such a system would be its ability to verify correct operation of a circuit under the entire range of MIN and MAX element parameters. If we consider the actual parameters such a propagation speed, setup time, etc ... of all digital blocks in a circuit as being the components of a multi-dimensional space, then the proof of correct operation everywhere within this space is our goal. Because our time line constraint system propagates constraints on possible time lines, rather than one particular time line as is common in current simulators, true timing margins for the ranges of device specifications may be established. This is an increasingly necessary operation for large, fast, computer systems, where the skew of signals and clocks is on the same order as the clock periods.
Absolute Blockers in Parallel

The next topology to address is that of blockers in parallel. Here we introduce the notion of a coincidence gate being the reader of blocking action in a set of several parallel blockers fed from a common input. As discussed before, the coincidence gate can be implemented by one or more layers of selectively weighted input integrators (with small blocking time constants) acting as linear threshold elements.

Consider the following circuit:

![Circuit Diagram]

In this circuit, two blockers, of blocking time $B1$ and $B2$, are fed with identical pulse trains. Their output is monitored by an AND coincidence gate, which fires whenever a pulse is emitted by $B1$ and not $B2$. What does this condition indicate about the history of pulses in the input stream?

This question is difficult to answer, because a variety of $B1$ and $B2$ output waveforms result in the same AND gate output waveform. Without more powerful notation, we cannot draw a single, or even a finite, set of time lines to represent the output relationships of the two blockers.

If we use retriggerable blockers, however, a top-down analysis is possible.
Last Interval Recognition

We remember that a retriggerable blocker passes an impulse conditional only upon whether a minimum of time has passed since the last input pulse. Thus, if we arbitrarily label $B_1$ and $B_2$ so that $B_1 < B_2$, we can make a table of blocking conditions and their cause.

<table>
<thead>
<tr>
<th>Interval ($i$)</th>
<th>B1</th>
<th>B2</th>
<th>AND Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i &lt; B_1$</td>
<td>blocked</td>
<td>blocked</td>
<td>null</td>
</tr>
<tr>
<td>$B_1 &lt; i &lt; B_2$</td>
<td>open</td>
<td>blocked</td>
<td>pulse</td>
</tr>
<tr>
<td>$B_2 &lt; i$</td>
<td>open</td>
<td>open</td>
<td>null</td>
</tr>
</tbody>
</table>

Thus, the above circuit, with retriggerable $B_1$ and $B_2$, outputs a pulse whenever an input pulse is received that terminates an interval lying between $B_1$ and $B_2$.

More complex last-interval detection functions are easy to implement, using the following architecture:

![Diagram showing the architecture of a last interval recognition system with blockers B1 to Bn and a combinatorial coincidence detection unit.]

The upper layer of blockers provide last interval detection for a range of durations.
\( B_1 < B_2 < B_3 < \cdots < B_k \). Thus, for an input pulse terminating an interval of duration \( i \) where \( B_k < i < B_{k+1} \), the input to the combinatorial coincidence detector will be a set of \( k \) parallel coincident pulses from the outputs of blockers \( B_1 \) through \( B_k \).

Once this analog to unary digital conversion of the last interval has been accomplished, a combinatoric threshold gate can then be designed to map the desired interval values to an output pulse. For example, the combinatoric coincidence gate could output a pulse whenever one was received from \( B_1 \) and not \( B_2 \), or from \( B_3 \) and not \( B_4 \). For monotonically increasing blocking values, this would translate to detecting intervals falling between \( B_1 \) and \( B_2 \), or between \( B_3 \) and \( B_4 \):

```
<table>
<thead>
<tr>
<th>Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last Interval Duration 0</td>
</tr>
</tbody>
</table>
```

### Logic Functions with Negated Inputs

An integrate and block implementation of the combinatoric coincidence detector gets into trouble when an output pulse must be generated when all of the blockers (\( B_1 \) through \( B_n \)) block the input pulse. For example, consider the case of detecting intervals less than \( B_1 \). Because this causes no combinatoric input to be active, the combinatoric coincidence detector cannot know when to output a pulse. Since DC levels are not an appropriate output, one cannot simply output a "1" whenever no inputs are active.

To solve this problem, and to provide the "always on" input for threshold bias (as discussed in the previous chapter), we provide every combinatoric gate with an unblocked, but properly delayed, copy of the input (or, equivalently, one passed through a zero blocking time blocker \( B_0 \)). This allows for pulse generation in the condition of total block by all other
blockers. It should be noted that this “all zeroes” condition is, speaking generally, very rare, and that if another thresholding mechanism is available (e.g. built in membrane bias), one is not giving up much by abandoning the detection of all zeroes.

More Complex Functions

Unfortunately, last interval classification is about as far as analytic methods can take us, because multiple contingencies of time and space compound the complexity of analysis enormously. This occurs in the time domain when non-retriggegrable blockers are used, or when series strings (or trees) of blockers are used. It also occurs in the space domain as soon as parallel structures feed input to dimension-reducing gates (like the AND gate seen above). Bottom up analysis is just too hard in these cases.

Note that all architectures with a single layer of retriggegrable blockers can only be history sensitive to the last interval, since all retriggegrable blockers have their state reset by every incoming pulse regardless of their current state. Thus, the retriggegrable blocker is prone to history erasure by each incoming pulse. This can somewhat be overcome by the use of series strings, but the added memory is poor compared to the amount of extra hardware.

The non-retriggegrable absolute blocker can remember past pulses even when presented with new ones, because a new one may arrive during block, and thus not affect the blocker’s state at all. This is also troublesome, however, because it means that each pulse which does trigger the non-retriggegrable blocker causes it to be information “blind” to future pulses during the blocking time. This can be somewhat alleviated by providing for an array of parallel and series blockers so that a fraction of blockers will always be receptive, but again, the amount of added hardware required to compensate for this blindness is large.

Now we begin to see why the exponential integrator-blocker is so attractive. Because integrator outputs last for some time, and, if they arrive close together, are cumulative, the exponential blocker can discriminate integrator inputs based more on their history than a single pulse arrival time. Because the block threshold is never infinite, no blocker is ever
blind to all input rhythms, and because the block decays slowly, the phase of the input rhythm in relation to the history of output firings is taken into account. There is inherent feedback because of this, which leads to greater state retention and good history dependence with a minimum of hardware.

The power of the exponential blocker will be explored in a few chapters, but first, since analysis of its behavior is so intractable, we show how the behavior of pulse circuits can be simulated, and how circuit design can be effected by learning instead of analytical construction.
Simulation of Pulse Circuits

Any simulator of integrative blocker circuits must be able to calculate the generation time of impulses by the blocker as well as the effects of incoming impulses that pump the integrator. The simulator must also effectively trace the evolution of the functions \( I(t) \) and \( B(t) \), and this latter task threatens to require large amounts of computer time. Luckily, although the integrator output \( I(t) \) and blocking threshold \( B(t) \) are continuous functions, they appear as signals only inside of the integrative blocker. As far as the rest of the circuit is concerned, they matter only in their effect on impulse generation.

Thus, because \( I(t) \) and \( B(t) \) evolve so simply, and because their interaction is locally confined, we can implement a simulator that predicts the time of threshold crossing inside each integrative blocker and takes giant leaps forward in time to the moment of that crossing. This is much faster then tracing out the evolution of analog state variables in small time increments and changing circuit connectivity when thresholds are crossed. Our efficient alternative is very similar to the event driven or relaxation mode techniques used in modern VLSI simulators,\(^{28}\) except that the binary results are unimportant - we care only about transition times!

Before we consider the threshold crossing problem, let's get familiar with the general idea of event driven simulation. We start by simulating the action of a circuit with zero time constant integrators and absolute blockers. In such a circuit, an absolute blocker is triggered only if its threshold is zero and the integrator inputs obey:

\[
W_{excit} > W_{inhibit}
\]

With a retriggerable absolute block, this is the type of circuit that can implement last interval detection, as discussed in the previous chapter.

The Event Queue

An event driven simulator works by having a queue of simulated future events. This type of queue is almost identical to the timeout queues commonly used in operating systems to call specific functions after a given passage of time. The only difference lies in the manner in which time is advanced. In the timeout queue, time advances because of a real-time clock. In the event queue, time advances instantaneously to the earliest future event. The operation of the event queue is insensitive to processor speed because time is a virtual, not physical, commodity and its rate of advance is decoupled from the real time taken to execute functions. Event queue functions execute in effectively zero virtual time.

Each queue element describes a function to be called, its arguments, and the "pause" time which is to elapse before the call:

```c
struct e
{
    int (*funcp)(); /* Event function */
    int arg; /* Event Argument */
    double darg; /* Double argument */
    double time; /* Pause time before call */
    e *nep; /* Next event */
    e *pep; /* Previous event */
};

typedef struct e e; /* Type shorthand */

extern e *fep, *lep; /* Pointer to first and last events */
```

For those readers unfamiliar with modern C, the `typedef` line is just shorthand for making the named structure a new data type of the same name.

The queue is a doubly linked, ordered list of events, from the earliest (*fep) to the latest (*lep). As such, the pause time recorded in each element is relative, not absolute, and indicates for each queue element how much time should elapse after the moment of the previous queue element's function call.

Elements are added to the queue by calling the function `event()` in the following fashion:
event (funcp, arg, time, undup, darg)

Where:

funcp is a pointer to the function to be called
arg is an integer argument to the called function
time is the time to elapse before the function is called
undup is a flag that causes identical (funcp, arg) elements to be removed first
darg is an optional double precision floating point argument

The purpose of undup will be discussed later. After allocating a queue element (let's call it new) for storage, event() fills in new with event()'s arguments and sorts it into proper place in the queue. This is done by passing over each queue element, from earliest to latest, and successively subtracting the considered element's pause time from new's pause time. Eventually, the point is reached where a subtraction would reduce new's time below zero, and rather than have this happen, new is inserted in the queue before the current element. The current element's pause time is then reduced by new's remaining pause time, so that the total time of future elements is not disturbed. The actual code for event() is found in Appendix F.

Running the Queue

Now that a consistent way of scheduling events into the queue is at hand, we need only find the means to execute events from the queue. For the sake of reference, the global variable Time is maintained to indicate absolute time (beginning at 0 at the start of the simulation run). In a simple loop, the function run() executes the following:

remove the earliest event from the queue
advance Time by the removed element's pause time
call the removed element's (*funcp)(arg, darg)

This process continues until there are no elements left in the queue or until Time exceeds a
given limit. Note that Time is not used whatsoever by event(); the time argument given to event() indicates that a function should be called in so many seconds, i.e. at absolute time Time + time.

The functions called from the queue by run() typically call event() to place future events on the queue. Because only future events can be scheduled, at the end of each run loop, it is impossible that an event should occur prior to the earliest element in the queue, and thus it is safe to advance Time by the pause time of the earliest element.

Input Events

We will wish to simulate the reaction of our pulse circuits to a stream of input events given by a file or the output of another program. The safest means to accomplish this would be to read all input events into the queue before the simulation's start, thus guaranteeing their call at the proper future time. However, this means that the simulator must store all input events before beginning the simulation, and in the case of long input streams, this is an extremely cumbersome solution.

What can be done instead is to place on the queue, rather than the input events themselves, a single call to a function which effects one input event and reads from the input the time and location of the next. Thus, each input event causes, as a side effect of its execution, the loading of the next future event into the queue. Because input events are assumed to be in time-order and one future input event is always in the queue, this guarantees that no input events are missed. The simulator can run (and give output results) with only one event worth of storage for input, and the input stream may be infinite without inhibiting the simulation of circuit events. Of course, the first input event must be primed into the queue before the circuit simulation starts.
Simulation of Absolute Block Circuits

Below is a slightly simplified version of the data structure allocated for each integrative blocker, called unit, as used by the "absolute blocking without integration" simulator.

```c
struct unit
{
    double firet;        /* Time of last excitation */
    double btime;        /* Blocking time */
    struct link *links;  /* Output list */
};

typedef struct unit unit;
```

The structure link looks like this:

```c
struct link
{
    unit *from;        /* Source unit */
    unit *to;          /* Destination unit */
    double weight;     /* Link weight */
    double delay;      /* Link delay */
    struct link *next; /* Next link */
};

typedef struct link link;
```

Because its time constant is zero, the simulated integrator requires no state. However, the blocker does need one continuous variable, and that is firet, the time which marks the beginning of the last blocking period.

When a integrative blocker outputs a pulse (or a pulse is read from the input), the fire() function spreads the impulse to other parts of the circuit after the proper delay. fire() is short enough for us to consider here (the full program is in Appendix G):

```c
fire (up)
    unit *up;
{
    link *lp;

    up->firet = Time;

    for (lp = up->links; lp; lp = lp->next)
        event (excite, lp->to, lp->delay, 0, lp->weight);
    }
`excite()`, although slightly more complicated, is also understandable with little effort:

```c
excite (up, weight)
    unit *up;
    double weight;
{
    e *ep, *nep;
    double future = 0. ;

    /* Swallow up excitation to this unit in the near future */
    for (ep = fep; ep && (future += ep->time) < SIMUL; ep = nep)
    {
        nep = ep->nep;

        if (((unit *) ep->arg == up && ep->funcp == excite))
        {
            weight += ep->darg;

            if (nep)
                nep->time += ep->time;

            remove (ep);
        }
    }

    if (weight > 0 .)
    {
        if (Time - up->firet > up->btime)
            fire (up);
        else if (RETRIG)
            up->firet = Time;
    }
}
```

If the "swallowing" for loop seems mysterious, it should, because what is being done breaks the abstraction between the general event driven functions discussed in the previous chapter and the simulator specific function being discussed here.

**Simulated Coincidence Detection**

The "swallow" code is a cheat of sorts. Because all connections between units are assumed to have significant delay compared to the width of an individual pulse, when the first excitation arrives at the input to a unit, the simulator can peek at what other excitations might be arriving in the very near future. In this manner, a quick calculation can be done to determine whether the total coincident activity is sufficient to generate response.
Essentially, excite()'s first for loop begins at the earliest element in the queue and looks at successively later elements until the end of the queue is reached or until the total future time goes past SIMUL seconds. SIMUL is a small value indicative of pulse width, which as discussed previously, is the means by which simultaneity is measured. Each considered queue element is checked to see if it is an incoming excitation to the same unit, and if so, it is removed from the queue and its effective weight added to the accumulating variable weight.

At the end of this process, the total weight of all coincident pulses has been accumulated and all coincident pulses removed from the queue. The sign of the accumulated weight is then checked. If positive, and the blocking interval has passed, then fire() is called to effect a unit firing. Otherwise, a check is made to see if the simulator is running in retriggerable block mode, in which case the blocking time is reset even if a firing did not occur.

These two functions, fire() and excite(), form the heart of the absolute blocking simulator: Other functions exist for the purposes of input/output, etc ..., and are included in the complete listing in Appendix G.

An Example

Consider the following circuit, composed of retriggerable absolute blockers and coincident logic, which was designed to recognize pulses terminating intervals \( i \) where \( 0 < i < 1 \) or \( 2 < i < 3 \):
The numbers inside each unit's circle represent the blocking time, while those adjacent to each link give the link weight. The coincident AND-OR gate can be transformed as follows:

Thanks to the form of the input and the hardware efficiency of threshold logic, only one output device is needed. With the circuit in a form appropriate for the simulator, we can feed
in a sequence of input pulses and see what happens. First, however, we must decide how to correlate observations of input and output activity.

**Triggered Snapshots**

To describe simulated system behavior, we use the concept of an output triggered input snapshot. Each time a pulse is emitted from the circuit’s output, we take a snapshot of the input history. The trigger time is negatively (non-causally) adjusted for the circuit delay time from input to output so that the snapshot taken shows only the history of firings previous to and including the pulse that was responsible for the trigger.

In this manner, we can feed a wide variety of input pulse trains into the circuit and collect a stack of snapshots whose terminating pulse caused an output pulse. By processing this collection of snapshots, we can then determine if the desired history dependence criterion was detected, and also whether histories which should not have been detected were falsely recognized.

We also take a snapshot of every input history whose terminating pulse failed to cause an output pulse. These negative snapshots are separately compiled in the same manner as the positive ones, so that we get an idea of how many histories should have been detected, but were falsely rejected.

**Interval Digrams**

The most straightforward snapshot compilation technique is that of the interval digram scatter plot. Here, two co-ordinates are extracted from each snapshot: that of the the last interval \((X)\), and that of the next to last \((Y)\). The point so described is plotted on a 2-D graph, and the collection of scattered points gives an indication of the history detection performance. For the circuit above, upon being fed 1000 samples of a Poisson random process for input, the following interval digram scatter plot is obtained:
We see that the desired performance was obtained perfectly, as only those snapshots showing last intervals between 0 and 1 or between 2 and 3 were recognized positively, while no such intervals failed to be recognized.

**Gaussian Smoothed Pre-trigger Probability Signature**

Another method of compiling the snapshots is to overlay them, aligned on their terminating pulse, and then estimate the pre-terminating pulse probability signature using methods identical to those used on the physiological data presented earlier. The resulting signatures indicate the probability of having an input pulse as a function of time before a pulse that is recognized. For the circuit above, this compilation technique results in the following signatures:
The effect of the two-banded last interval recognition criterion can be seen, but the signature notches do not fall to 0. This can be explained by realizing that although the first pulse previous to that resulting in recognition cannot occupy certain pre-recognition times, pulses earlier than the first may occur at any time. Thus, the following snapshot is classified positively, and acts to "fill in" the –1 to –2 notch:

![Diagram of time intervals]

Although the pre-recognition pulse probability signature is a poor indicator of performance for last interval recognition, it can be a good indicator if the recognition criterion is different. For example, a simple appropriate criterion would be: "at least one pulse preceded the recognized one in the time interval 0 < i < 1".
Simulation of Exponential Block Circuits

As discussed previously, the absolute block and zero integration time-constant unit has poor performance due to blindness or oversensitivity to successive pulses. Much better history dependence can be had by allowing the integrator to accumulate inputs and the blocker to effect variable degrees of block. The simulation of the exponential integrative blocker will be explained in this section.

Unit Definition

Because more state is involved, the definition of a unit in the exponential simulator is slightly more complex than it was for absolute block:

```c
struct unit {
    double fiRet; /* Time of last blocker firing */
    double btau;  /* Blocker time constant */
    double itau;  /* Integrator time constant */
    double iv;    /* Last integrator voltage */
    double it;    /* Time of last integrator voltage */
    struct link *links; /* Pointer to output link list */
};
```

Fortunately, we can use the same link structure as before. The structure elements `btau` and `itau` represent the blocking time constant $B_t$ and integration time constant $I_t$, respectively. The absolute time of each firing is recorded in `fiRet` so that the blocker's threshold decays in between firings as:

$$B(t) = (B_{max} - B_{rest}) e^{-\frac{(t-fiRet)}{B_t}} + B_{rest}$$

Unlike the blocker, the integrator is pumped to varying voltages before it decays, and two variables are needed. The variable `iv` marks the integrator voltage at absolute time `it`. Future values of $I(t)$ can be predicted from these two values by:

$$I(t) = (iv - I_{rest}) e^{-\frac{(t-it)}{I_t}} + I_{rest}$$

Note that at any time (Time), the following operation may be executed without affecting the
future values of $I(t)$:

$$iv = I(Time)$$

$$it = Time$$

Since $Time$ is maintained by the event simulation code as the current absolute time, this has the effect of making $iv$ equal to the current integrator voltage.

**Pumping**

In the same manner that the absolute simulator's `excite()` function swallowed up incoming coincident pulses, the exponential simulator's `excite()` function swallows up active weights, categorizing them into positive and negative accumulators:

```c
excite (up, weight) /* Excite a unit, firing if conditions are correct */
    unit *up;
    double weight;
{
    e *ep, *np;
    double future = 0.;
    double posweights, negweights;
    double volts, mhos;

    if (weight > 0.)
    {
        posweights = weight;
        negweights = 0.;
    }
    else
    {
        negweights = -weight;
        posweights = 0.;
    }

    /* Swallow up excitation to this unit in the near future */
    for (ep = fep; ep && (future += ep->time) < SIMUL; ep = nep)
    {
        nep = ep->nep;

        if (((unit *) ep->arg == up && ep->funcp == excite)
        {
            if ((weight = ep->darg) > 0.)
                posweights += weight;
            else
                negweights -= weight;
```
if (nep)
    nep->time += ep->time;

remove (ep);
}

/* Calculate Thevenin source equivalent */

mhos = posweights + negweights;
volts = (posweights * IMAX + negweights * IMIN) / mhos;

up->iv = I(up, Time);
up->it = Time;

up->iv = PUMP(up, volts, mhos);
relax (up);

Once the positive and negative conductances have been accumulated, the Thevenin equivalent of the input resistors is calculated. The integrator state variables \( iv \) and \( it \) are updated as described above, and then \( iv \) is pumped towards the Thevenin voltage \( volts \) by the macro \( \text{PUMP}() \):

\[
\text{PUMP}(up, volts, mhos) = volts - (volts - up->iv)e^{-\text{TIMCAP}mhos}
\]

\( \text{TIMCAP} \) is the standard pulse duration divided by the integration capacitance. So that a 1 mho input weight causes a pump fraction a little over \( \frac{1}{2} \), \( \text{TIMCAP} \) is set to \( \frac{7}{10} \).

Relaxation

Having updated and pumped the integrator voltage, the simulator must now check if the pump results in an immediate or perhaps a delayed firing. This is done by the function \( \text{relax}() \).

Immediate firing is easy to check; this occurs if the just pumped \( I(\text{Time}) \) is greater than the present \( B(\text{Time}) \). Furthermore, because \( I_{\text{rest}} = -1 \) and \( B(t) \) is always positive, if \( I(\text{Time}) \) is already negative, we know it can never grow larger than any future \( B(t) \).
The only remaining case is if $I(\text{Time})$ is positive and $B(\text{Time}) > I(\text{Time})$. In this case, delayed firing can occur:

Because output pulses are generated out of sync with input pulses, we will endeavor to avoid this mode of firing in our designs. However, it is important that the simulator still be capable of predicting when such firing will occur, and for future designs, perhaps this mode will be useful.

To predict the moment of delayed trigger, we wish to find the instant $I(t)$ exceed $B(t)$, or in other words, when

$$I(t) = B(t)$$

that is, when

$$(\text{iv} - I_{\text{rest}}) e^{\frac{(t-\text{iv})}{B_0}} + I_{\text{rest}} = (B_{\text{max}} - B_{\text{rest}}) e^{\frac{(t-t_{\text{reset}})}{B_0}} + B_{\text{rest}}$$

Unfortunately, no analytical solution is know for this equation. However, since the derivatives of $I(t)$ and $B(t)$ are easily found, Newton's method can be used to converge on the solution. First, we find the point of maximum overtrigger, i.e. when $I(t) - B(t)$ is maximum. This is done by solving:

$$\frac{dl(t)}{dt} - \frac{dB(t)}{dt} = 0$$
Substituting the formulas for $I(t)$ and $B(t)$ into this equation, we have

\[
\frac{(iv - I_{rest}) e^{-\frac{(u-iv)}{l_t}}}{-I_t} - \frac{(B_{max} - B_{rest}) e^{-\frac{(u-t)\mu_{iv}}{B_{t}}}}{-B_{t}} = 0
\]

which, after some algebra, can be solved for $t$ as:

\[
t = \frac{B_t l_t \ln \left( \frac{B_t (iv - I_{rest})}{l_t (B_{max} - B_{rest})} \right) + B_t it - I_t \text{fired}}{B_t - I_t}
\]

This value of $t$ can now be plugged into $I(t) - B(t)$, and if the result is positive, we know that a zero crossing lies in the future. Starting half-way to this point of maximum over-trigger, Newton’s method can then be used to find, in a few iterations, the approximate trigger time. Below is a slightly simplified version of the C code for `relax()`:

```c
relax (up)    /* Relax a unit, checking for blocker trigger */
    unit *up;
{
    double t, vdelta, vdiff;

    /* Check if we are over threshold from the start */
    if (I(up, Time) > B(up, Time))
    {
        event (fire, up, 0., 1);
        return;
    }

    /* Check if integrator is inhibited */
    if (up->iv <= IREST)
        return;

    /* Find point of maximum over-trigger */
    t = (up->btau == up->itau) ? LONGLTIME : ZERO (up);

    /* Check if we’ll ever fire */
    if (t < Time || I(up, t) < B(up, t))
        return;

    /* Using Newton’s method, converge on future trigger time */
    for (t = (t + Time) / 2.; ; t -= vdelta / vdiff)
```
{ vdelta = I(up, t) - B(up, t);  

if (fabs (vdelta) < EPSILON)  
{  
    event (fire, up, t - Time, 1);  
    return;  
}  

vdiff = dI(up, t) - dB(up, t);  
}

The macro \texttt{ZERO()} calculates the time of maximum overtrigger as explained above. A simplified version of the exponential simulator's \texttt{fire()} function is shown below:

\begin{verbatim}
fire (up)  
    unit *up;  
    {  
        link *lp;  
        up->iv = I(up, Time);  
        up->it = Time;  
        up->firt = Time;  
        for (lp = up->links; lp; lp = lp->next)  
            event (excite, lp->to, lp->delay, 0, lp->weight);  
        relax (up);  
    }
\end{verbatim}

Because \texttt{relax()} assumes that iv and it have been updated to the present Time, \texttt{fire()} begins by updating these values. The instant of the firing is recorded in \texttt{firt}, and its effects spread to linked units as in the absolute block simulator. It is possible, if integration time is significantly longer than blocking time, for a single unit, after self-blocking, to re-fire with no additional input. This occurs when the input integrator has not decayed much but the blocking threshold has fallen. To check for this condition, \texttt{relax()} is called at the end of \texttt{fire()}. Because \texttt{relax()} can schedule future firings, to prevent recursive runaway, all \texttt{fire()} scheduling in \texttt{relax()} is done with \texttt{event()} calls rather than direct subroutine calls, even if the time to pause before calling \texttt{fire()} is 0.
The Use of undup

Because the predicted time of a past relaxation becomes invalid when a new integration input pump occurs, relax() calls event() with the undup flag set. This has the effect of removing from the event queue past calculations of future delayed firings.
Learning

We’ve seen how efficient pulse circuit simulation can be carried out, and how the “output triggered input snapshot” can be used to evaluate classifier circuit performance on a variety of inputs. We’ve also constructed a simple classifier of last intervals that used a set of tuned retrigerable absolute blockers followed by a coincidence gate to accomplish its task.

Now we consider the goal of building a general system to parse complex input sequences based on a learned criterion.

Learning by Example

A story is told 29 of the method used to teach new farm hands to determine the sex of baby chickens. It is advantageous, for practical reasons, to separate male and female chicks shortly after hatching. Although no set of visual attributes is known that differentiate the young chicks according to sex, some experienced farmers have a well honed “knack” of simply looking at a chick and coming up with the right answer. To train a new person in this art, a conveyer belt of chicks is passed in front of a seated student. The student takes an intuitive guess as each one goes by, while an experienced instructor watches from behind, tapping the student on the right shoulder for a correct answer and on the left for an incorrect response. After some time, the student acquires the knack for sexing chicks, even though neither the teacher nor the student can describe what allows them to make the distinction.

This notion of hard-to-describe learning is not only funny, it is the same “learn by example” methodology we’ll use in our artificial circuits. Like the visual differences between male and female chicks, the mapping of input rhythms by exponential blockers is very hard to describe. To make our circuits learn despite this difficulty, we provide them with an instructor giving corrective advice during the presentation of a variety of inputs:

29 by Jerry Lettvin
As with chicken sexing, both instructor and student calculate an answer, and after comparison, the student learns to do better. In the case of pulse circuits learning rhythm classification, we'll use a piece of software to monitor the timing of the input pulse stream and decide, for each input pulse, whether that pulse terminated a desirable or an undesirable rhythm.

**Input Statistics**

It is a well known result of information theory\(^{30}\) that an optimally encoded band-limited channel has the same statistics as a channel carrying band-limited white noise. This leads us to suspect that a noise function can justifiably be used to generate the "variety of inputs" that both student and teacher consider during the learning process. We want to teach, with correctly balanced statistics, the repertoire of inputs the circuit will face during its operational "career". If the input channel is to be used efficiently, that "career" will consist of parsing messages from an efficiently coded input line, i.e. one that looks like noise.

---

There are, however, many noise functions to choose from, and although white noise is most efficient for an analog band-limited channel, it is reasonable to think another distribution may be optimal for active pulse transmission lines. To answer this question with certainty, Appendix D undertakes a study of the limits of signalling capacity in pulsatile transmission media, and derives the optimal statistics for pulse channels of non-zero recovery time, non-zero jitter, and limited power.

As is shown in the Appendix, a Poisson process is representative of optimal information encoding on a finite power pulse channel with exponentially decaying receiver time resolution. Thus, we will use a Poisson process to create the "variety of inputs" from which to train the coincident layer.
Training a General Rhythm Classifier

The trainable rhythm classifier uses a two layer design similar to that used by the absolute blocking last interval detector:

Both the blocking layer and the coincidence detection layer are implemented from the exponential integrative blockers described previously. In natural nervous system terminology, we are constructing a rough analog of one neuron's terminal arbor connected to another's dendritic input tree. In the first (blocking) layer, we use integrative blockers with unit-weight inputs and varying integration and blocking time constants ($I_\tau$ and $B_\tau$). In the second (coincidence) layer, only the logical input properties of the IB units are used, and so the input weights vary but $I_\tau$ and $B_\tau$ are made very small. It is to be understood that natural neurons have both integrative-blocking and variable weighting in both their dendritic input trees and terminal arbors. Additionally, we mix negative (inhibitory) weights and positive (excitatory) weights in the connections of the two artificial neurons, whereas a natural system would require an "interneuron" to effect inhibitory action.

The delay on any path from system input to blocking layer output is uniform, so that the coincidence layer is always acting on a non-skewed vector of signals. Thus, given a
definition of the circuit topology and time constants of the blocking layer, a map can be made of each input rhythm's terminating pulse to the corresponding set of invaded blocking layer outputs. Each input pulse generates a mid-section bit vector of successful (1) or unsuccessful (0) invasions.

The first task of the learning process is to make certain the blocking layer provides sufficient sensitivity and specificity for the rhythmic features we wish to detect.

**Designing the Blocking Layer**

To check for sensitivity and specificity, a representative variety of input rhythms are presented to the input of the blocking layer. For each pulse that enters the system, a record is made of which blocking layer outputs were invaded by that pulse. Simultaneously, a software "instructor" passes judgment on whether the input pulse terminated a desirable rhythm, and the invasion pattern is assigned a tag of 1 (wish to detect) or 0 (do not wish to detect). This is done for a large number (from 1,000 to 10,000) input pulses, yielding a stack of records.

The records are then binned by invasion pattern so that all records with identical invasion pattern are classified in the same bin. Each bin thus has a number of positive records corresponding the instructor saying "detect (1)" and a number of negative records created when the instructor said "don't detect (0)".
If only one type of tag is present in any particular bin, the meaning of that invasion pattern (under the trial space given by the input set) is clear. If both types of tag are present, then the invasion pattern is ambiguous. Since our coincidence detection circuitry is by definition a deterministic map, the best we can do is to apply democratic voting to the process; whichever type of instructor response is in predominance within any one bin wins. A tradeoff of result uncertainty and hardware complexity will be discussed in the next section on coincidence layer learning.

The importance of picking proper statistics for the input pulse stream can now be seen. Since different statistics bias the voting majority of ambiguous invasion patterns, it is important that the training statistics be representative of the use statistics.

When the bin voting process is complete, we have in our hands the specification of a combinatoric Read Only Memory (ROM) that provides, for each invasion pattern, a single bit response:
Non-ambiguous bins will always give correct response, while vote decided bins will sometimes give improper response. The number of suppressed minority votes as a fraction of the majority and shutout (non-ambiguous) votes gives an indication of the sensitivity and specificity of the blocking function to the particular rhythmic class we wish to detect.

The circuit topology used to implement the general blocker is the same simple one layer structure that was used for absolute block last interval detection:

Note that a copy of the input is passed through unblocked to provide for threshold shift and all-block detection by the coincidence layer.
No general algorithm has yet been found to assign the various values of \( I_t \) and \( B_t \) so as to maximize performance. However, a manual hill climbing approach was used by the author in the examples to be presented below, and very good results were obtained with a 16 unit blocker using all possible combinations of \( B_t = \{1.0, 1.5, 2.0, 2.5\} \) and \( I_t = \{0.0, 0.6, 1.2, 1.8\} \). This variety of history response was sufficient to allow different coincidence detectors to recognize different classification criterion with low probability of error, all using the same 16 pair set of blocking and integration parameters.

**Selection of \( B_{\text{max}} \)**

The maximum blocking threshold \( B_{\text{max}} \), it will be remembered, specifies to what level the blocker's threshold rises after each pulse. Initially, this was set to 2, or twice the maximum possible value of \( I(t) \) (remember that \( I_{\text{max}} = 1 \)). Once the negative effects of non-retriggerable blindness were understood, however, \( I(t) \) was lowered to 1.1, a value which is close to the minimum required for self-block of impulse generation under maximum excitation. Although no rigorous means are available for showing this to be the best choice, the same manner of \( B_t \) and \( I_t \) parameter hill climbing was undertaken in each case, and a significant increase in performance (from 80% correct discrimination to 95% correct) was obtained. The choice as to the correct weight for blocker input links was made in the same manner; eventually settling on that weight which pumped the integrator output just over 0 for a single pulse arriving with the integrator at rest (thus causing an output firing). With these choices, the maximum positive dynamic range of the integrator was made available for the accumulation of future impulses.
Examples of Blocker Performance

The following examples were simulated using the 16 unit exponential blocker just described. The "output triggered input snapshot" method was used to compile the plots, with a majority programmed ROM for the second stage. Thus, these plots indicate the best system performance that can be achieved with the given 16 unit input blocking stage. A 1000 pulse input stream was used to generate all plots except the last.

Short Interval Detection

The instructor software for this example was programmed to give a positive response if the last interval had duration less than 5. After reading the entire history of input and output pulses into a large time-ordered event array, the following function was called for each input pulse:

```c
wantle5(ep)
    event *ep;
{
    double now = ep->time;

    if ((ep = previnevent(ep)) && now - ep->time < 5)
        return (1);
    else
        return (0);
}
```

To understand this code, one need only know that previnevent(), given a pointer to an input event, searches backwards in the event array for the most recent previous input event. Thus, the instructor software calls wantle5() with each input event as argument, and wantle5() returns 1 if the previous input event happened less than 5 seconds before, and 0 otherwise.

Under the wantle5() rule, 67% of the Poisson process generated input pulses were positive (i.e. wantle5() returned 1), and 33% were negative. Of the positive examples, 97.3% could be classified correctly (i.e. had unambiguous or majority blocker invasion patterns), while all (100%) of the negative examples were classified correctly. The left hand
plots below show the pre-trigger probability signature for this criterion, while the right hand ones are pre-trigger interval digram plots:

For this task, the exponential blocker's performance is not as perfect as would be possible with a well tuned non-retriggerable absolute blocker, but it is still very good.
Double Short Interval Detection

We really start seeing the power of the exponential block in this next example. Recall that memory of more than the last interval is not possible in a single layer of retriggerable absolute blockers. It is, however, possible in a single layer of exponential integrative blockers. In the following example, the instructor software was programmed to give a positive response if both of the last two intervals were less than 5 seconds long:

```c
wantquad(ep)
    event *ep;
{
    double now = ep->time;
    double delta1, delta2;
    event *ep1, *ep2;

    if ((ep1 = previnevent (ep)) && (ep2 = previnevent (ep1)))
    {
        delta1 = now - ep1->time;
        delta2 = ep1->time - ep2->time;

        if (delta1 < 5 && delta2 < 5)
            return (1);
        else
            return (0);
    }
}
```

42% of the input pulses were classified positively under this rule, with 92.1% being classified correctly. Of the remaining 58% negative responses, 99.7% could be classified correctly.
Again, some unavoidable blocker ambiguity causes a few incorrect responses, but overall, the performance was quite good.

**Double Interval Range Detection**

Here we again use the digram paradigm as the means to express our desired detected rhythm, but instead of a single inequality for each interval, we specify a range:

```plaintext
wantbox(ep)
```
event *ep;
{
    double now = ep->time;
    double delta1, delta2;
    event *ep1, *ep2;

    if ((ep1 = previnevent (ep)) && (ep2 = previnevent (ep1)))
    {
        delta1 = now - ep1->time;
        delta2 = ep1->time - ep2->time;

        if (delta1 > 2 && delta1 < 7 && delta2 > 2 && delta2 < 7)
            return (1);
        else
            return (0);
    }
}

For this instructor criterion, only 19% of the input pulses were positive examples. Still, 93.5%
% of these could be classified correctly, whereas 99.4% of the negative examples could be
classified correctly:
Signature Programming

All of the previous examples have been based on a specification of the last one or last two successive intervals. Now we see if the array of exponential blockers can provide sensitive and selective rhythm differentiation based on differing pulse probability signatures in the style of Optican and Richmond. In the following example, we specify that at least one input pulse must have occurred between 0 and 2 seconds previous to the output trigger pulse, that
no pulses can have occurred between 2 and 4 seconds previous to the output trigger, and that at least one pulse must have occurred between 4 and 6 seconds previous to the output trigger. The C code for this instructor criterion is as follows:

```c
wantcomb2(ep)
    event *ep;
{
    double now = ep->time;
    double delta;
    int count02 = 0, count24 = 0, count46 = 0;

    while (ep = previnevent(ep))
    {
        delta = now - ep->time;

        if (delta < 2)
            count02++;
        else if (delta < 4)
            count24++;
        else if (delta < 6)
            count46++;
        else
        {
            if (count02 > 0 && count24 == 0 && count46 > 0)
                return (1);
            else
                return (0);
        }
    }
    return (0);
}
```

Three counter variables \(\text{count02, count24, count46}\) record the number of input pulses in the three pre-trigger time ranges. After all three variables have been filled, we see if the criterion was met. Using this rule, only 7.5 % of the input pulses terminated positive rhythms, and, perhaps because such a large majority of input examples were negative, only 70% of the positive input pulses could be classified correctly. Of the negative patterns, however, 95% could be judged correctly:
Because of the low positive example count, 5000 input pulses were used instead of 1000. Note how well the pre-trigger probability signature matches with the desired criterion. This sort of filter could do a good job detecting the firing patterns hypothesized by Optican and Richmond to encode visual information. The diagonal region apparent in the digram plot is due to the sum criterion of interval times to make up the earliest (count 46) pulse of the three pre-trigger time zone sequence.
Training the Coincidence Detector

With a bank of exponential blockers providing different invasion patterns for various rhythms, there remains the task of building a combinatoric coincidence layer. This layer will map sets of blocking layer invasion patterns to a single output pulse, signifying the detection of a member of the desired input rhythm class.

The ROM address binning process just described gives a specification of the best possible input to output map. To program a coincidence layer to implement this map, we can take advantage of the back propagation\textsuperscript{31} learning algorithm recently developed for feed forward multi-layer perceptrons. Our pulse coincidence layer is identical in function to a single output back-propagation network, except that events are transmitted internally by pulses instead of continuous values.

This difference is not a disabling one, because with a sufficient number of nodes, back-propagation networks infrequently use the graded region of their activation function for computation - use of the saturated regions is more typical. However, back-propagation learning in multi-layer systems does depend on the continuity of the activation function, and thus we are forced to use a continuous activation function in the learning phase.

Note that we are proposing back-propagation as a learning process inside the dendritic structure of a single neuron, not in a structure of several neurons. Our artificial system uses only impulse generating elements to construct its "dendritic tree" because any combinatoric function can be designed out of such elements. However, this in no way excludes the feasibility of graded action occurring in the dendrites of natural neurons - in fact this is probably more generally the case than not.

The continuity of the interior of one neuron's dendrites allows for far more "backwards" analog communication than does a multi-neuron circuit. Furthermore, such

\textsuperscript{31} Rumelhart, David E. and McClelland, James L.: \textit{Parallel Distributed Processing, MIT Press, 1986.}
communication does not require that analog values be encoded as frequency modulations of multi-pulse signals. We have seen that dendritic integration time constants can be quite small, and the theory of FM encoding stands on shaky ground. By confining backpropagation learning to the dendritic tree of a single neuron, we supply it with a reasonable method of transporting the necessary learning information and yet keep it consistent with real observations (made in the first half of this thesis) about real dendritic characteristics.

Any boolean function can be computed in 2 layers of linearly weighted threshold elements. This is easily seen by considering the boolean function in terms of its AND - OR specification. Since n-input AND and OR gates can be implemented as single unit threshold functions and negation is possible by changing the sign of a weight, we know that a two layer threshold network is sufficient for any function. In particular, the following 2 layer architecture, given a sufficient number of "hidden" nodes, can compute any boolean map:

If the number of hidden nodes is limited, we know that each node can at least represent one AND term, and thus we have a minimum complexity for what can be represented.
When back propagation is applied to the two layer single output architecture, it is quite possible that a more efficient formulation than AND-OR will be acquired to represent a particular map. However, knowing that the network is sufficient to learn an AND-OR specification of certain size gives us a measure of the minimum power of the network's expressivity.

**Applying Back-Propagation**

While being trained, the units of the coincidence layer compute continuous functions. Three phases make up each learning cycle. In the first phase, forward activations \( A_i \) are computed by each unit, from input to output. After all activations have been computed, a backwards computation is done, from output to input, which roughly measures each unit's error contribution \( E_i \). Finally, the link weights between units are adjusted according to the computed \( A_i \) and \( E_i \).

In the forward phase, each unit first computes its total stimulus \( S_d \) by summing the activations of its feeding units multiplied by the weights of the connecting links:

\[
S_d = \sum_i W_{sd} A_i
\]

where \( W_{sd} \) is the weight of a link hooking source unit \( s \) to destination unit \( d \).

Each unit's stimulus is subject to a sigmoidal logistic activation function to determine the unit's output activation:

\[
A_d = \frac{1}{1 + e^{-S_d}}
\]

This activation is propagated forward to successive units, and so forth.

When the output unit's activation is finally computed, the backwards propagation begins. Each unit computes a total output botch factor \( B_s \) in a similar manner as the total input stimulus \( S_d \) was computed, but in the reverse direction:

\[
B_s = \sum_d W_{sd} E_d
\]
The output unit's botch factor is given as:

\[ B_{\text{output}} = A_{\text{correct}} - A_{\text{output}} \]

Where \( A_{\text{correct}} \) is the training instructor's binary decision as to the correct result. Each unit computes its error from the botch factor as follows:

\[ E_s = B_s \frac{\delta A_s}{\delta S_s} \]

\[ = B_s A_s (1 - A_s) \]

After all \( E_s \) values are computed, each link's weight is adjusted in the following fashion:

\[ \Delta W_{sd} = .05 A_s E_d + .9 \Delta W'_{sd} \]

where \( \Delta W'_{sd} \) is the previously computed \( \Delta W_{sd} \). This corresponds to a learning rate of .05 and a "momentum" of \( \frac{9}{10} \).

Justifications for this learning process will not be presented here, but are given in PDP.\(^{32}\) The complete program used for back propagation is listed in appendix I.

When the circuit is not being trained, the standard integrative-blocker behavior takes place, with very small blocking and integration time. Thus, a pulse is generated whenever the stimulus is greater than 0. This corresponds to an infinitely steep activation function.

**Training by ROM Listing**

In the previous section, we saw how a ROM could be programmed to provide the best possible coincidence layer function. Training examples could be drawn from this ROM's contents, and we would, in effect, be trying to duplicate the ROM with the coincident back

propagation network. Since network resources are limited, we would expect only a certain (hopefully large) fraction of the network’s responses to be correct compared to the ROM.

But the ROM only has one bit of storage per address, and this single bit gives little information as to which input addresses are used most frequently (i.e. are most important). Training the coincident network from the ROM’s contents assigns equal importance to all ROM addresses, regardless of their frequency under typical inputs. Unfortunately, this means that neural network vs. ROM discrepancies will not be biased towards the less important address, as they could be.

Training by Actual First Layer Output

Certain entries in the ROM are also the result of majority votes, where a degree of ambiguity existed as to the proper response. A way to include such ambiguity, as well as the popularity (i.e. importance) of different addresses, is to not train the network from the ideal ROM specification, but rather from the actual blocking layer’s output under the random input. Ambiguous inputs will have effect in proportion to both their positive and negative tags, and frequently occurring patterns will be trained more strongly than those occurring less frequently. This mode of training is also more justifiable naturally than the pre-compilation of an “optimal” ROM, and it is this mode which was used for the following example.
An Example of Coincidence Layer Learning

In this final section, we show how coincidence layer learning actually performs on one non-trivial example: double short interval detection. Training of 750,000 pulses consumed approximately 24 hours of computer time on a DEC MicroVAX II, and this prevented the exploration of other examples in this thesis. Nonetheless, the example to be considered is representative of a "hard" rhythm recognition problem. It requires memory which persists beyond one input pulse, and an analytical design of the coincidence network is by no means obvious.

The blocking layer used for this example is identical to that used in the double short interval detector of the previous chapter. In the network described below, this blocking layer feeds a 2 layer coincidence network of 16 hidden units. Each hidden unit is connected to each of the blocking layer’s 16 outputs, and as well, to a delayed copy of the system input. As discussed previously, this latter signal provides for threshold control and the ability to detect complete block. The 16 hidden units, and another delayed but unblocked copy of the input, are then fed to the output unit.

Before we discuss the performance of the back propagator, let’s take a look at the input vector set on which the network was trained. The table below gives a bin addressed histogram of positive and negative vectors produced when 5,000 Poisson process generated pulses were fed to the blocking layer:
<table>
<thead>
<tr>
<th># of examples per bin</th>
<th># of vectors</th>
<th>% of all examples</th>
<th>Positive</th>
<th># of vectors</th>
<th>% of all examples</th>
<th>Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>617</td>
<td>30.3</td>
<td>362</td>
<td>12.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>93</td>
<td>9.1</td>
<td>112</td>
<td>7.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>41</td>
<td>6.0</td>
<td>46</td>
<td>4.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>24</td>
<td>4.7</td>
<td>33</td>
<td>4.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>17</td>
<td>4.2</td>
<td>21</td>
<td>3.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>3.2</td>
<td>16</td>
<td>3.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>17</td>
<td>5.8</td>
<td>14</td>
<td>3.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>0.8</td>
<td>4</td>
<td>1.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>1.8</td>
<td>7</td>
<td>2.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>1.0</td>
<td>6</td>
<td>2.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>1.1</td>
<td>1</td>
<td>0.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>7</td>
<td>1.2</td>
<td>3</td>
<td>1.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>0.6</td>
<td>3</td>
<td>1.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td>1</td>
<td>0.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>2.2</td>
<td>2</td>
<td>1.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td>4</td>
<td>2.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td>1</td>
<td>0.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>0.9</td>
<td>3</td>
<td>1.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>1.0</td>
<td>1</td>
<td>0.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>2.2</td>
<td>1</td>
<td>1.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>1</td>
<td>1.2</td>
<td>1</td>
<td>0.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td>1.3</td>
<td>1</td>
<td>0.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>1</td>
<td>1.3</td>
<td>1</td>
<td>0.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>1</td>
<td>1.3</td>
<td>1</td>
<td>0.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td>1.4</td>
<td>1</td>
<td>0.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>1.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>1</td>
<td>1.7</td>
<td>1</td>
<td>1.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>1</td>
<td>1.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>1</td>
<td>1.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>2</td>
<td>4.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td></td>
<td></td>
<td>1</td>
<td>1.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>1</td>
<td>2.3</td>
<td>1</td>
<td>1.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>49</td>
<td></td>
<td></td>
<td>1</td>
<td>1.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>51</td>
<td></td>
<td></td>
<td>1</td>
<td>1.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>57</td>
<td></td>
<td></td>
<td>1</td>
<td>1.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>79</td>
<td></td>
<td></td>
<td>1</td>
<td>2.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
<td>1</td>
<td>2.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>89</td>
<td></td>
<td></td>
<td>1</td>
<td>3.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>107</td>
<td></td>
<td></td>
<td>1</td>
<td>3.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>137</td>
<td></td>
<td></td>
<td>1</td>
<td>4.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>157</td>
<td>1</td>
<td>7.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>511</td>
<td></td>
<td></td>
<td>1</td>
<td>17.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>850</strong></td>
<td><strong>2037 examples</strong></td>
<td><strong>656</strong></td>
<td><strong>2963 examples</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

By telling us how many example vectors binned to the same location, this table gives insight into the blocking layer’s hash-function properties. Note that roughly 30% of the positive examples resulted in unique vectors. A coincident layer network that learned each of the 1506 vector bin addresses as a unique “memory” would be enormous.

But the above table does not show the commonality amongst bits in each vector, nor does it show the bit relationships between vectors. Thus, without encoding each of the ROM’s addresses separately, a simple network with a small number of hidden units may still be able to extract the essence of similarity within each category (positive and negative), and
learn to distinguish between them. The real coincidence layer network, because of its small size, forces a sort of soft logic minimization on the mapping function.

Network Convergence

The following graph shows the performance evolution of the 16 hidden unit coincidence network. The X axis represents the number of input pulses, while the Y axis gives percent recognition performance for the positive examples. To save computer time, the invasion patterns training the coincidence layer were repeated every 5,000 pulses. For clarity, only performance measures at multiples of 5,000 pulses are marked below:

![Graph showing network convergence](image)

The slightly noisy non-monotonic behavior is due to the pulsatile evaluation of the network, which was trained with a continuous activation evaluation function.

The performance on the training data, after 750,000 input pulses, was:

<table>
<thead>
<tr>
<th>Match Category</th>
<th>Coincidence Network</th>
<th>ROM (Best Possible)</th>
</tr>
</thead>
<tbody>
<tr>
<td>positive</td>
<td>83 %</td>
<td>90 %</td>
</tr>
<tr>
<td>negative</td>
<td>97 %</td>
<td>98 %</td>
</tr>
<tr>
<td>overall</td>
<td>91 %</td>
<td>95 %</td>
</tr>
</tbody>
</table>

This performance corresponds to the following 1000 pulse “output triggered input snapshot” plots, which may be compared with the ROM second layer plots of the previous chapter:
Plots for the full 5000 pulse input train are shown below:

Although the 16 hidden unit coincident network is not perfect, it does a remarkably good job at detecting the desired input criterion 5 by 5 "box".

The final test was to feed the network an input pulse stream other than that which made up the training set. This was done by allowing the poisson process to produce 5,000 pseudorandom generated pulses, and then using the next 5,000. This gave the following
performance:

<table>
<thead>
<tr>
<th>Match Category</th>
<th>Percent Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>positive</td>
<td>80 %</td>
</tr>
<tr>
<td>negative</td>
<td>96 %</td>
</tr>
<tr>
<td>overall</td>
<td>89 %</td>
</tr>
</tbody>
</table>

which corresponded to the following 5000 point "output triggered input snapshot" plots:

It is interesting to contrast this with the performance of the ROM based system trained on 5,000 input pulses, but evaluated with 5,000 other pulses. In this situation, 20 % of the evaluation vector addresses were unknown to the ROM. The plots below represent ROM-
trained performance on a novel input set, with a random classification choice for unknown vectors:

As can be seen, the ROM's performance was significantly worse then that of the coincident network. This is because the training algorithm caused the hardware limited coincident network to generalize its map of input vectors to output classes. With this generalized map, the network could then extrapolate good responses to input vectors it had never seen before.
Conclusions

Since the time biological neurons were first found to convey pulsatile signals, two theories have been proposed to explain the methodology of neural information encoding.

The first, and simplest, theory was that of frequency modulation. As was physiologically observed in nerves controlling muscle (and also in nerves receiving simple stimuli), higher firing rates meant larger magnitudes of some fiber-specific quantity. Since neurons controlling muscle worked this way, wouldn’t interior neurons operate in the same manner? In this view, neural impulses took on the role of an FM carrier, modulated by an analog quantity - the supposed real meaning of the neural message. “Neural computation” became a study of slow non-linear analog circuits, with the underlying pulsatile signalling being an implementation detail to be abstracted away. This view is still by far the most prevalent today.

The other neural encoding paradigm (as proposed by McCulloch and Pitts\textsuperscript{33}) resembled that of digital logic. Linear threshold units were used in place of less powerful digital gates, with unit delays between elements providing for well behaved discrete time behavior. The theoretical universality of the McCulloch - Pitts model was easily shown, but in practical terms, the construction of long term memories and multi-stage digital calculations using pulse logic and unit delays was not practical. Skew accumulates in chained delay circuits, and pulse train feedback cannot be expected to reliably store (and make available) short term information. The movement of modern computer design away from McCulloch-Pitts type pulse and delay circuits toward DC encoded synchronous finite state machines is indicative of the lack of practicality in the former methodology.

This thesis (and the work done by Raymond and Lettvin which inspired it) has demonstrated both the likelihood and feasibility of a third approach.

In the first half of the thesis, the theory of universal FM encoding was disputed by two laboratory observations. First, the rhythmic structure of frog optic nerve firings discovered by Lettvin was verified. Second, it was shown, by second order statistics, that the first stage of circuitry in the frog’s optic tectum did not integrate its pulsatile inputs for a long (multi-pulse) period of time. Without long integration time, average firing rate cannot be decoded from a nerve fiber, and we conclude that the first layer of tectal circuitry must operate on the coincidence of individual impulses rather than on their average rates.

With natural per-pulse encoding and computation established, we were faced with the question: how does a neural circuit decode rhythmic patterns encoded on a single fiber? The second half of the thesis addressed this question by describing a simple device to model the essential per-pulse characteristics of nerve membrane: synaptic weighting, input integration, and output block.

Membrane trees with varied integration and block time constants were shown able to translate rhythmic variability on a single wire into a spatial invasion encoding on several wires. It was then shown that a lookup table ROM could be designed to map the multi-line invasion patterns to a single bit. This output bit could then signify whether a particular rhythmic class had just appeared at the network’s input.

With the feasibility of constructing such a rhythmic feature detector shown, we next demonstrated how automatic learning techniques could be used to program the weights of a pulse-mode linear threshold network so as to capture the essence of invasion pattern differences. Finally, we showed that a network so trained could extrapolate class information for novel inputs, resulting in higher performance (with less hardware complexity) than would be possible in a lookup table system.

The generation of a single output pulse to indicate recognition of a particular input rhythmic class is an artificial choice of design. A more realistic notion of natural neuronal action is that the dendritic tree, after performing coincident logic on its plethora of inputs, presents a highly transient excitation and load variation signal to the neuron’s pulse
generator. The pulse generator executes a complex time-sampling function on the dendritic signal, which results in a pulse rhythm (rather than one pulse) being generated.

The details of this encoding process may not be important. It is, however, important that enough information about the dendritic tree's time varying activity is encoding in the rhythm-modulated signal.

The multiple paths of the axonal arbor then play the role of differentially blocking pulses generated by the initial axon segment. What results at the terminating axonal twigs is a particular spatial pattern of invasion for each initially generated pulse. The pulse coincident logic of secondary dendritic trees fed by the differentially invaded terminals then detects various features of the rhythms, which correspond to various state histories of the first neuron's dendritic activity.

We have shown how back-propagation, given simple corrective advice, can be used to train the coincident level circuitry of a receiving neuron's dendrites, without any need for FM encoding of analog values. Most likely, learning also takes place in the axon's terminal arbor so that sufficient sensitivity to the axonal rhythm variations is achieved at axonal terminations. It is possible that the axonal rhythm is itself responsible for the differential development of the blocking arbor.

We are left with a view in which each neuron first integrates its inputs in a combinatoric manner and then processes the intermediate results by means of some sampling function, encoding the result as a pulsatile rhythm. This rhythm is then parsed by the blocking action of the axonal arbor, followed by the coincidence logic of successive dendrites. Thus, rather than being a non-linear processor of frequency modulated pulses, the neuron is elevated to an encoder of input conditions, followed by a trainable feature extractor. While this mechanism is more complex and less easily analyzed than the FM model, it is also significantly more powerful.
If correct, the fine grained pulse computing theory is of fundamental importance to neuroscience, because it redefines the relevant functional properties of neurons. The relevance of this theory to modern computer design is more difficult to state with certainty. As discussed in the second chapter, applicability depends on the development of a pulsatile hardware technology. If a physical device with intrinsic impulse propagation properties like nerve membrane is ever developed or discovered, then the types of circuits we’ve discussed here could be practical alternatives to digital feature recognition machines. As was shown in the last example, a fairly complex recognizer can be implemented out of pulse devices with good performance and very few parts.
References


Appendix A - Physiological Recording Instrumentation

The instrumentation used to record the physiological data given in this paper was designed to detect action potentials in the face of noise, to allow discrimination among action potentials that arose from multiple fibers, and to precisely record the time of occurrence of every action potential. The detailed shape of each impulse was not measured.

The electronics was separated into seven physical components, the first 5 of which were located inside a shielded recording room, and were shielded themselves to prevent electromagnetic contamination of the room’s interior. In order of signal flow, they were:

- Pre-amplifier
- First oscilloscope
- Frog-11 analog front end
- Second oscilloscope
- Audio monitor
- Frog-11 digital electronics
- Parallel interface

Pre-amplifier

A straightforward AC-coupled circuit utilizing a low noise, high input impedance op-amp gave the electrode signal its initial gain boost of 50.

Physically, the pre-amplifier sat atop the electrode positioner so as to keep its input leads short and unstressed during positioning. The pre-amp’s power supply was attached via
a flexible cable and located a short distance away.

Some data was recorded by Arthur Grant on digital audio tape using an identical pre-amplifier. These tapes were then played back as if emerging from the pre-amplifier just described.

**First oscilloscope**

The pre-amplifier output was passed through a Tektronix 1A7A oscilloscope plug-in, where it was subject to adjustable gain and bandpass filtered to 1-3 KHz. The resultant signal was sent to the Frog-11 interface and also displayed on the oscilloscope screen. The screen’s horizontal trace was triggered externally by the Frog-11’s detection electronics.

**Frog-11**

Electronically, the Frog-11 consisted of 5 circuits:

- Detector
- Digitizer
- Multiplexor
- FIFO Buffer
- Control FSM

Analog portions of the detector were located in the recording cage. The digital circuitry of all other circuits resided in the unshielded computer room.

**Detector**

The detector’s input was declared to be a valid action potential when it rose above a preset threshold. Once so declared, retriggering was held off until the input voltage fell below half of this same threshold. This insured that a particular pulse was counted only once, even when contaminated by noise.

After triggering, the detector waited for the waveform to crest by waiting for it to drop below 90% of its highest value since the trigger moment. The exact time of the 90% fall
point was recorded, as well as the value of the 100% voltage used for comparison. Next, the
detector waited for the input to fall below 50% of the peak value and another time stamp was
made. In this fashion, information regarding both the time of arrival, the height, and the fall
time was recorded for every detected pulse. Finally, the detector reset its circuitry and waited
for the next impulse.

Detector Hardware

The detector circuit began with an externally adjustable variable gain amplifier that
boosted the amplitude of the incoming signal up to 100 times.

The amplified, AC-coupled, signal was then sent to several threshold comparators
(outside the shielded room), the track/peak/hold circuit, the vertical amplifier of the monitor
oscilloscope, and the audio monitor.

At the heart of the Frog-11’s detector was the track/peak/hold (tiplh) circuit. This
circuit had an analog input, an analog output, and two digital control inputs. The digital
inputs caused the circuit to operate in one of three modes:

<table>
<thead>
<tr>
<th>Control Input</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Track</td>
<td></td>
</tr>
<tr>
<td>Peak</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>X</td>
</tr>
<tr>
<td>L</td>
<td>H</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
</tr>
</tbody>
</table>

In Track mode, the analog output followed the analog input. In peak mode, the output was
allowed to rise to match the input’s value, but not to fall, while in hold mode, the analog
output was not affected by the input at all, but held at its last value.

The Detection Process in Detail

Each impulse was detected in 5 phases:
Sub-Threshold

While waiting for a trigger, the t/p/h circuit was put into track mode. A comparator measured the input voltage against the settable threshold, and when the input exceeded the threshold, a trigger occurred.

Rise

At the instant of triggering, the t/p/h circuit was put in peak mode, with its output, for all purposes, identical to its input. The first oscilloscope was also triggered at this instant, allowing a visual display of the entire pulse. As the input continued to rise, the output voltage increased so as to follow the input. Small non-monotonicities in the input waveform did not affect the output or the detector’s control logic, as a 10% drop was required before a peak was declared.

Peak

When the input dropped below 90% of its peak, the first (interval) time stamp was made, and the t/p/h circuit put into hold mode. The analog to digital conversion of the 100% peak height was begun, and the detector waited for the input to drop to 50% of its peak value. Once this happened, the second (width) time stamp was made.

Fall

After recording the data just taken, the detector waited (for reasons stated above) for the waveform to drop below half the original threshold voltage.

Slew

Next, the t/p/h circuit was put back into track mode, whereupon its output slewed from the previously held peak down to the current input. The slew interval was terminated when the t/p/h output voltage approached the input (i.e. the input was over 90% of the tracking
output).

Finally, with the circuitry settled down, the detector was ready for a new impulse.

**Digitizer**

Each pulse was characterized by three numbers. A 32 bit counter clocked at 1 MHz provided time stamps for the 90% and 50% marks. All 32 bits were recorded for the 90% fall point, allowing inter-pulse intervals of over 1 hour to be recorded with a resolution of 1 microsecond.

Only the low order 16 bits of the time stamp were recorded for the 50% mark. This provided for 90% → 50% fall times of up to 65 milliseconds.

A 10 bit A/D converter produced the 100% analog height as an unsigned binary number with .1% resolution.

**Multiplexor**

As the bus interface transferred only 16 bits at a time, four successive words were transmitted for every impulse. First, the multiplexor fed the FIFO the low order interval word, then the high order interval word followed by the single fall time word. Finally, the 10 bit A/D output was presented, left padded with 6 ones.

The Vax 11/750’s recording software synchronized the four word frames by searching for the 6 high order ones in the trailing word. While it was possible for other words to have six ones in the high order bits, such false synchronization could persist for at most \(2^{26}\) microseconds, or about one minute. As the parallel link to the computer was reliable, once proper framing was achieved, there was no reason to lose framing synchronization. Also, the device driver reset the entire FROG-11 when the recording program was not being used, so the first frame was in proper synchronization. In practice, framing errors were never encountered.
FIFO Buffer

A Signetics 8X60 FIFO controller, in conjunction with 4096 words of static RAM, formed a first-in first-out queue for the multiplexed data, allowing fast pulse bursts to be recorded even during periods of high computer response latency.

Control FSM

A simple 5 chip FSM, synchronously clocked at 1 MHz, controlled the timing of all signals in the Frog-11. Two 2764 EPROMs held the FSM code, which is included in appendix B.

Second Oscilloscope

Three outputs from the detector were sent to the second oscilloscope to form a real time pulse quality display. The scope's horizontal sweep was triggered on the falling edge of the t/p/h circuit's Peak control signal, i.e. the 90% fall point. The t/p/h input signal was applied to the vertical amplifier. At the 50% point (the falling edge of Track), a short, high voltage, pulse was generated to greatly intensify the CRT beam. Thus, as each pulse was received, a small line, approaching a dot, was flashed upon the face of the CRT. Its vertical position was proportional to the peak height, while the horizontal position indicated time elapsed between the 90% and 50% marks.

The very same criteria that were used by software to separate pulses from multiple fibers in later processing were thus shown in real time on the face of the CRT. In theory, a high quality recording could be predicted by locating the electrode so as to produce a small number of, concentrated, well separated, spot "clouds". This display also was used to adjust the detector gain so as not to overload the A/D.
Audio monitor

The amplified signal was also sent to an audio amplifier and speaker. The \textit{t/p/h Track} signal, when asserted, squelched the audio output, thus masking noise below the detector's threshold. The monitor's circuit was so arranged as to shift its zero point to the input's value the moment squelch was turned off, thus suppressing a pop regardless of the threshold level.

Bus interface

To simplify design, a standard DEC DR-11C interfaced the custom \textbf{Frog-11} circuitry to the \textbf{Vax 11/750}'s UNIBUS. A special UNIX device driver, taking advantage of the 4096 word FIFO buffer in the \textbf{Frog-11}, transferred data very efficiently despite the DR-11C's lack of direct memory access (DMA).

The DR-11C's 40 wire "B" connector accepted 16 bits of parallel data and 1 bit of control input, labeled \textit{REQA}. \textit{REQA} was asserted by the \textbf{Frog-11} when its FIFO was half full, causing an interrupt on the \textbf{Vax 11/750}. Recording software responded to the interrupt by quickly reading out 2048 words of data. Also present on the "B" connector was a control output, \textit{CSR0}, which, when asserted, enabled the \textbf{Frog-11}'s recording circuitry. When de-asserted, \textit{CSR0} caused the \textbf{Frog-11} to enter a set-up mode where the detector was operated, but data discarded. When data was being recorded, a handshake output, \textit{DTRANS}, from the DR-11C caused the FIFO output to advance one word. Finally, an initialization line, \textit{INIT} reset the \textbf{Frog-11} whenever the \textbf{Vax 11/750} was initialized.
Appendix B - FROG-11 FSM Code

The following program was executed once on a VAX 11/750 UNIX system to generate the contents of the FROG-11’s finite state machine ROM.

To allow for good association between the generation program’s text and the corresponding actions of the FSM hardware, the generation code was written so it seems as if the generation program itself, not its output, is being executed in the FROG-11 hardware. This handy confusion of action and description was made possible through the use of a generic driver to probe the FROG-11 specific code for appropriate action under all possible state and input conditions. This action was then noted and became the contents of the FSM ROM at the corresponding state and input address. The FSM was thus constrained, in every situation, to mimmick the actions of the generation program.

With a little practice, this format of ROM specification is very easy to read. Moreover, it is very easy to debug, because the human programmer can follow the code and “play FSM” just like a programmer “plays computer” in the usual process of debugging a program. The actions of FSMS are algorithmic, and it thus makes sense to take advantage of the powerful algorithm description constructs available in computer programming languages. More primitive ROM specification strategies based on regular expression matching and complex substitutions are markedly inferior in this regard.

Generic Driver

```c
#include <stdio.h>

main(argc, argv)
    int argc;
    char *argv[];
{
    extern int nroms, nwords;
    FILE *files[4];
    char s[80];
    int i, j, k;

    for (i = 0; i < nroms; i++)
```
```c
{ printf (s, "%s.%d", argv[0], i);

    files[i] = fopen (s, "w");
}

for (i = 0; i < nwords; i++)
{
    j = rom(i);

    if ((i % 1024) == 0)
    {
        fprintf (stderr, "%d words\n", i);
    }

    for (k = 0; k < nroms; k++)
    {
       putc (((j & 0377), files[k]);
        j >>= 8;
    }
}

fprintf (stderr, "%d words.\n", nwords);

for (i = 0; i < nroms; i++)
    fflush (files[k]);

fprintf (stderr, "%d roms.\n", nroms);
}

FROG-11 Specific Code

/*
 * Specification for frog-11 interface FSM roms
 *
 * Gill Pratt
 *
 * 1986 (revised into new format August, 1989)
 */

#include <stdio.h>

struct in
{
    unsigned g_vth:1; /* > Vth */
    unsigned g_vth2:1; /* > Vth / 2 */
    unsigned g_50:1; /* > 50 % */
    unsigned g_90:1; /* > 90 % */
    unsigned n_adready:1; /* A/D not ready */
    unsigned n_ack:1; /* not ACK */
    unsigned csr0:1; /* CSR0 */
    unsigned init:1; /* init */
```
unsigned state:5; /* Current state */

struct out
{
    unsigned load_phase:1; /* Load the phase latch */
    unsigned load_width:1; /* Load the width latch */
    unsigned sel:2; /* Multiplexor control */
    unsigned n_convert:1; /* 0 = run A/D conversion */
    unsigned n_si:1; /* 0 = do fifo shift in */
    unsigned peak:1; /* peak input signal */
    unsigned track:1; /* track input signal */
    unsigned n_reset:1; /* reset fifo controller */
    unsigned n_clear:1; /* clear phase counter */
    unsigned latch_ad:1; /* latch A/D output */
    unsigned state:5; /* Next state */
} defout = { 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0 }; /* Default output */

int nwords = 8192;
int nroms = 2;

union
{
    int inword;
    struct in in;
} inunion;

union
{
    int outword;
    struct out out;
} outunion;

struct out *outp = & outunion.out;
struct in *inp = & inunion.in;

rom(inword)
{
    int inword;

    inunion.inword = inword;

    frog();

    return (outunion.outword);
}
#define loop() (outp->state = inp->state)
#define advance() (outp->state = inp->state + 1)
#define go(x) (outp->state = (x))
#define set(x) (outp->x = 1)
#define clear(x) (outp->x = 0)

frog()
{
*outp = defout;

if (inp->init)
{
    clear (n_reset);
    clear (n_clear);
    set (track);
    set (peak);
    go (0);
}
else
{
    switch (inp->state)
    {
        case 0:
            if (inp->g_vth)  /* Just got triggered */
            {
                set (peak);
                advance();
            }
            else /* Waiting for trigger */
            {
                set (track);
                loop();
            }
        break;
        case 1:
            if (inp->g_vth2)
            {
                if (inp->g_90) /* Wait for peak */
                {
                    set (peak);
                    loop();
                }
                else /* Load phase, convert */
                {
                    clear (n_convert);
                    set (load_phase);
                    advance();
                }
            }
            else /* Droop bailout */
            {
                set (track);
                go (0);
            }
        break;
        case 2:
            if (inp->g_vth2)
            {
                if (inp->g_50) /* Wait for the fall */
                {
                    clear (n_convert);
                    loop();
                }
            }
        default: /* Do nothing */
            break;
    }
}
else if (inp->csr0) /* Load width, write phase */
{
    set (load_width);
    shiftin (0);
    advance();
}
else /* Recording disabled */
{
    clear (n_clear);
    clear (n_reset);
    set (load_width);
    go (10);
}
else /* Droop bailout */
{
    set (track);
    go (0);
}
break;
case 3:
    waitack (0);
    break;
case 4: /* Write high phase */
    shiftin (1);
    advance();
    break;
case 5:
    waitack (1);
    break;
case 6: /* Write width */
    shiftin (2);
    advance();
    break;
case 7:
    waitack (2);
    break;
case 8:
    if (inp->n_adready) /* Wait for A/D ready */
    {
        set (latch_ad);
        clear-(n_convert);
        outp->sel = 3;
        loop();
    }
    else /* Write height */
    {
        set (latch_ad);
        shiftin (3);
        advance();
    }
    break;
case 9:
if (! inp->n_ack)   /* Got ack */
{ 
    if (inp->g_vth)   /* Ack, > vth */
    {
        advance();
    }
    else       /* Ack, < Vth */
    {
        go (11);
        set (track);
    }
}
else if (inp->csr0) /* Wait for ack */
{
    set (latch_ad);
    shiftin (3);
    loop();
}
else                /* Recording disabled */
{
    clear (n_clear);
    clear (n_reset);
    go (10);
    break;
}

case 10:
    if (inp->g_vth)   /* Wait for input to drop below vth */
    {
        loop();
    }
    else       /* < vth */
    {
        set (track);
        advance();
    }
    break;

case 11:       /* Slew tracker circuit */
set (track);

    if (! inp->g_vth) 
        (! inp->g_vth2) 
            &&
        inp->g_50 
            &&
        inp->g_90)
    {
        go (0);
    }
else
{
    loop();
}
break;
}
shiftin (sel)
    int sel;
{
    outp->sel = sel;
    clear (n_si);
    clear (n_convert);
}

waitack(sel)
    int sel;
{
    if (! inp->n_ack) /* Got ack */
    {
        clear (n_convert);
        outp->sel = sel + 1;
        advance();
    }
    else if (inp->csr0) /* Wait for ack */
    {
        clear (n_si);
        clear (n_convert);
        outp->sel = sel;
        loop();
    }
    else /* Recording disabled */
    {
        outp->sel = 0;
        clear (n_clear);
        clear (n_reset);
        go (10);
    }
}
Appendix C - Frequency Domain Analysis of the Non-Retriggerable Blocker

The system we wish to analyze has the following diagram:

Our analysis will consider the effect of the absolute non-retriggerable blocker on constant frequency pulse trains, i.e.

\[ \sum_{n=-\infty}^{\infty} \delta(t-nT) \]  \hspace{2cm} (1.1)

where \( T \) is the period between pulse events.

In the frequency domain, a simple periodic pulse input is transformed as:

\[ \text{Input} = \sum_{n=-\infty}^{\infty} \delta(t-nT) \longleftrightarrow \frac{1}{T} \sum_{n=-\infty}^{\infty} \delta(f-n/T) \]  \hspace{2cm} (1.2)

We shall confine ourselves to look at the case where \( B < T \), i.e. where the input should pass through unaffected. In this case, the output is also:

\[ \text{Output} = \sum_{n=-\infty}^{\infty} \delta(t-nT) \longleftrightarrow \frac{1}{T} \sum_{n=-\infty}^{\infty} \delta(f-n/T) \]  \hspace{2cm} (1.3)

The transform of the one-shot's impulse response with the delay is:

\[ H(f) = \frac{\sin(\pi f B)}{\pi f} e^{-j\pi f (B+2D)} \]  \hspace{2cm} (1.4)

Thus, the one-shot's output in the frequency domain is given by:
\[
\frac{1}{T} \sum_{n=-\infty}^{\infty} \delta(f - \frac{n}{T}) \frac{\sin(\pi f B)}{\pi f} e^{-j\pi f (B + 2D)}
\] (1.5)

This function is non-zero only at multiples of \( \frac{n}{T} \), where it is an impulse of magnitude \( \frac{\sin(\pi f B)}{T \pi f} \) and phase shift \(-\pi f (B + 2D)\). When multiplied by the input in time, this signal is convolved in frequency:

\[
\left[ \frac{1}{T} \sum_{n=-\infty}^{\infty} \delta(f - \frac{n}{T}) \right] \ast \left[ \frac{1}{T} \sum_{n=-\infty}^{\infty} \delta(f - \frac{n}{T}) \frac{\sin(\pi f B)}{\pi f} e^{-j\pi f (B + 2D)} \right]
\] (1.6)

This convolution is non-zero only at multiples of 1 over \( T \), where it takes on the value:

\[
M = \frac{1}{T^2} \sum_{n=-\infty}^{\infty} \frac{\sin(\pi n B / T)}{\pi n / T} e^{-j2\pi n (B + 2D) / T}
\] (1.7)

Graphically, this looks like:

Expanding the sine:

\[
M = \frac{1}{T^2} \sum_{n=-\infty}^{\infty} \frac{e^{j\pi n B / T} - e^{-j\pi n B / T}}{2j} e^{-j2\pi n (B + 2D) / T}
\] (1.8)

combining exponents:

\[
M = \frac{1}{T^2} \sum_{n=-\infty}^{\infty} \frac{e^{-j2\pi n D / T} - e^{-j2\pi n (B + D) / T}}{2j \pi n / T}
\] (1.9)
and simplifying, we get:

\[ M = \frac{1}{2\pi jT} \sum_{n=-\infty}^{\infty} \frac{e^{-j2\pi a \frac{D}{T}} - e^{-j2\pi a \frac{(B+D)}{T}}}{n} \]  

(1.10)

To evaluate this summation, we begin by looking at

\[ \frac{e^{-j2\pi a \frac{D}{T}} - e^{-j2\pi a \frac{(B+D)}{T}}}{n} \]  

(1.11)

when \( n = 0 \).

By l'Hôpital's rule, this is:

\[ -j2\pi a \frac{D}{T} - j2\pi a \frac{B+D}{T} = j2\pi a \frac{B}{T} \]  

(1.12)

Returning to the original summation, we have

\[ M = \frac{1}{2\pi jT} \sum_{n=-\infty}^{\infty} \frac{e^{-j2\pi a \frac{D}{T}} - e^{-j2\pi a \frac{(B+D)}{T}}}{n} = \frac{B}{T^2} + \frac{1}{2\pi jT} \sum_{n=0}^{\infty} \frac{e^{-j2\pi a \frac{D}{T}}}{n} - \frac{1}{2\pi jT} \sum_{n=0}^{\infty} \frac{e^{-j2\pi a \frac{(B+D)}{T}}}{n} \]  

(1.13)

Clearly, we need a formula for

\[ \sum_{n=0}^{\infty} \frac{e^{jkn}}{n} \]  

(1.14)

This is best done by separating real and imaginary parts:

\[ \sum_{n=0}^{\infty} \frac{e^{jkn}}{n} = \sum_{n=0}^{\infty} \frac{\cos(kn)}{n} + j \sum_{n=0}^{\infty} \frac{\sin(kn)}{n} \]  

(1.15)

Since cosine is an even function, \( \frac{\cos(kn)}{n} \) is an odd function, and thus:

\[ \sum_{n=0}^{\infty} \frac{\cos(kn)}{n} = 0 \]  

(1.16)

Note that it is not having to worry about \( n = 0 \) that allows us to do this so easily.

To handle the real part, we use the discrete time fourier transform pair:

\[ \sum_{n=-\infty}^{\infty} \frac{\sin(2\pi \phi \eta)}{\pi n} e^{-j2\pi \phi n} = \begin{cases} 1, & \text{if} -m < \phi < \phi_0, \ -\infty < m < +\infty \\ 0, & \text{elsewhere} \end{cases} \]  

(1.17)
where $m$ is an integer and $0 < \phi_0 < \frac{1}{2}$. Letting $k = 2\pi \phi_0$ and $\phi = m$,

$$\sum_{n=-\infty}^{\infty} \frac{1}{\pi} \frac{\sin(kn)}{n} e^{-j2\pi mn} = 1 \quad (0 < k < \pi) \tag{1.18}$$

Since sine is cyclic, this is also true for $k+2\pi$, where $l$ is any integer. For $-\pi < k < 0$, we note that sine is an odd function, so

$$\sum_{n=-\infty}^{\infty} \frac{1}{\pi} \frac{\sin(kn)}{n} e^{-j2\pi mn} = -1 \quad (-\pi < k < 0) \tag{1.19}$$

Again, this is also true for $k+2\pi$, where $l$ is any integer. For $k = l\pi$, all terms of the sum are clearly zero except $n = 0$, where

$$\lim_{n \to 0} \frac{1}{\pi} \frac{\sin(kn)}{n} e^{-j2\pi mn} = \frac{k\cos(kn)}{\pi} = l \quad (k = l \pi) \tag{1.20}$$

Since $e^{-j2\pi mn} = 1$ for all integer $m$ and $n$,

$$\sum_{n=-\infty}^{\infty} \frac{\sin(kn)}{n} = \begin{cases} \pi, & 2\pi < k < (2l+1)\pi \\ l\pi, & k = l\pi \\ -\pi, & (2l-1)\pi < k < 2l\pi \end{cases} \tag{1.21}$$

Thus

$$\sum_{n \neq 0} \frac{e^{j2\pi n}}{n} = j \sum_{n \neq 0} \frac{\sin(kn)}{n} = j \left[ \sum_{n=-\infty}^{\infty} \frac{\sin(kn)}{n} \right] \lim_{n \to 0} \frac{\sin(kn)}{n} = \begin{cases} j (\pi-k), & 2\pi < k < (2l+1)\pi \\ 0, & k = l\pi \\ j (-\pi-k), & (2l-1)\pi < k < 2l\pi \end{cases} \tag{1.22}$$

Graphically, this looks like:
Thus, we can simplify our formula to:

\[
\sum_{n=0}^{\infty} \frac{e^{j\pi n}}{n} = \begin{cases} 
  j[\pi-(k \mod 2\pi)], & k \mod 2\pi \neq 0 \\
  0, & k \mod 2\pi = 0 
\end{cases} \tag{1.23}
\]

Finally, back to our equation for M:

\[
M = \frac{B}{T^2} + \frac{1}{2\pi j T} \sum_{n=0}^{\infty} e^{-j2\pi \frac{n}{T}} - \frac{1}{2\pi j T} \sum_{n=0}^{\infty} e^{-j2\pi \frac{(B+D)}{T}} 
\tag{1.24}
\]

For the first summation, \( k = -2\pi \frac{D}{T} \) while for the second, \( k = -2\pi \frac{B+D}{T} \). Since \( D \ll T \), in the first summation, \(-2\pi < k < 0\). For the second summation, we need to use the fact that our blocking time is less than the period of the waveform to ensure that \( B+D < T \) for \(-2\pi < k < 0\). For this range of operation,

\[
M = \frac{B}{T^2} + \frac{1}{2\pi j T} j \left[ -\pi - (-2\pi \frac{D}{T}) \right] - \frac{1}{2\pi j T} j \left[ -\pi - (-2\pi \frac{B+D}{T}) \right] 
\tag{1.25}
\]

Simplifying, we have

\[
M = \frac{B}{T^2} + \frac{2\pi D}{T^2} - 2\pi \frac{B+D}{T^2} = \frac{B}{T^2} + \frac{D}{T^2} - \frac{\pi}{T^2} - \frac{B+D}{T^2} + \frac{\pi}{T^2} = 0 \tag{1.26}
\]

Thus, for \( B+D < T \), the suppression signal is zero, which is consistent with pulses passing through unaffected.
Frequency Division

A similar, but more involved, analysis can be done in the case when \( B + D > T \). Since we expect the output period to be a multiple of the input period, we denote the output period \( T_{out} = cT_{in} \). Thus, equation (1.2), representing the input in the frequency domain, becomes

\[
\frac{1}{T_{in}} \sum_{n=-\infty}^{\infty} \delta(f - \frac{n}{T_{in}})
\]

while equation (1.5), representing the one shot's output, becomes:

\[
\frac{1}{T_{out}} \sum_{n=-\infty}^{\infty} \delta(f - \frac{n}{T_{out}}) \frac{\sin(\pi f B)}{\pi f} e^{-j\pi f (B + 2D)}
\]

The product of these two is less easy to compute than before because \( T_{out} \) is not at first obviously related to \( T_{in} \). But since we know the behavior of the device from time domain analysis, we can assume \( T_{out} \) to be a multiple of \( T_{in} \). This would lead us to a similar tautology as was done for the above case, and we would see that the inhibition signal has the proper form.

Unfortunately, the business of assuming the output to have a certain form ruins the purpose of this analysis, for it is not proof we are searching for, but a systematic method of determining output from input.

Thus, both propagation of constraint and computer simulation are more efficient tools for carrying out this task, and we continue along the lines of frequency analysis no further.
Appendix D - Determining Proper Training Statistics

Noise in Digital and Pulse Systems

Every electronic circuit known to man is unavoidably subject to the effects of electrostatic and electromagnetic noise. Despite this, almost all digital circuits assume total noise immunity from the output of one circuit element to the input of another. To bring this about, digital devices are manufactured so that precisely specified input noise margins surround the valid output voltage ranges. The diagram below shows these margins for standard TTL:

![TTL Logic Levels Diagram]

Corrupted signals received within a noise margin are taken to signify the valid output symbol to which the noise margin is adjacent. In the above diagram, the "outer" margins, i.e. above 5 volts or less than 0 volts, exist to insure that the device cannot be damaged by very high or very low inputs. The "inner" margins help discern the difference between a noise corrupted high and low. A standard TTL gate outputs less than 0.4 volts to signify an "L" (low) and greater than 2.4 volts to signify an "H" (high). On the other hand, the maximum input voltage that will be received as an "L" is 0.8 volts while the minimum input voltage for an "H" is 2.0 volts. The outer margins are larger than the inner ones, thus noise of an instantaneous amplitude up to .4 volts can be injected between transmitter and receiver with no effect.
At first glance, the assumption of absolute noise immunity would seem to be based upon a false premise, because real random noise, even when band and power limited, has a non-zero probability of taking on any instantaneous value. White noise, for example, has a Gaussian instantaneous amplitude probability density function:

$$\frac{d \, p(v)}{dv} = \frac{1}{\nu_n \sqrt{2\pi}} e^{-\frac{v^2}{2\nu_n^2}}$$

where the standard deviation $\nu_n$ is the rms noise voltage. Integrating $d \, p(v)/dv$ from a particular $v_{\text{min}}$ to $\infty$, we arrive at the probability of finding the instantaneous noise voltage over $v_{\text{min}}$:

$$p(v>V) = \int_{v_{\text{min}}}^{\infty} \frac{1}{\nu_n \sqrt{2\pi}} e^{-\frac{v^2}{2\nu_n^2}} dv$$

The standard compliment error function is defined as:

$$\text{erfc}(x) = \sqrt{\frac{2}{\pi}} \int_x^{\infty} e^{-y^2} \, dy$$

so that:

$$p(v>V) = \frac{1}{2\nu_n} \text{erfc} \left( \frac{V}{\nu_n \sqrt{2}} \right)$$

In and of itself, this probability of instantaneously discovering Gaussian noise above a certain voltage does not tell us how reliable a digital system will be. To determine this, we must insure that all noisy input signals will stay within the noise margins of every system element throughout the lifetime of the system.

**Calculating Reliability**

Every electronic receiver has a certain limited bandwidth. Although the noise introduced on a channel may have power outside this band, for the most part, noise power above the bandwidth of the receiver is averaged out and ignored.
Let us assume that the noise has significant power above the bandwidth of the receiver. When this noise is considered in the relatively large time scale of the receiver's response time, the noise from one moment to the next will be essentially uncorrelated (i.e. the noise coherence time will be much less than the receiver's response time). A reasonable method\textsuperscript{34} of determining overall system reliability is to consider the probability of always finding the input signal inside the noise margins on successive samples at the receiver's Nyquist rate.

A typical degree of reliability to demand of modern computer systems is one error every 10 years. If we take the bandwidth of a TTL gate to be 50 MHz, we have a Nyquist frequency of 100 MHz. 10 years contain 315,360,000 seconds, or roughly $3 \times 10^{16}$ Nyquist samples.

Thus, with a noise margin of .4 volts and, say, $10^4$ nodes to consider, a mean time between failures of 10 years requires an RMS noise that obeys:

$$p_{v > A} (v_n) = \frac{1}{10^4} \frac{1}{3 \times 10^{16}} = 3 \times 10^{-21}$$

Even when plotted on a log scale, $p_{v > A} (v_n)$ is extremely steep:

As seen above, the solution to $p_{v > A} (v_n) = 3 \times 10^{-21}$ is found at $v_n = .041$ (41 millivolts rms). Note that increasing the system size to $10^5$ nodes reduces the maximum tolerated RMS noise by

only 4 millivolts. Taking the rule of thumb\textsuperscript{35} of 7 times the rms noise value being the peak-to-peak value observed over the course of a few seconds, we have a short term observed peak-to-peak value of about .3 volts, i.e. an amplitude of .15 volts. This is the measured random noise a TTL system can tolerate and with no significant impact on reliability.

Two observations can now be made:

1. The fast fall-off of Gaussian distributions causes the average observed peak-to-peak value of tolerable random noise to be roughly 40\% of the noise margin required for reliable operation. For all intents and purposes, systems with absolute noise margins over three times the observed short-term peak noise will be reliable. In channel capacity terms, this means that a system with reasonably absolute reliability in the face of maximum gaussian noise sacrifices only $\frac{2}{3}$ of the available voltage signalling space compared to one with a high error rate.

2. The random noise observed on a digital wire in a real computer system is typically 3 orders of magnitude less than what has been shown above to be required for reliable operation.

So why are the noise margins so large? Manufacturing variations, drift, and aging do not play a part, because the .4 volt margin specified for TTL is that which is established across any two manufactured parts for their entire life, under all reasonable operating conditions. Usually outputs are of higher quality than the minimum specifications, and the noise margins correspondingly greater. What the noise margins are really for is not random noise, but correlated noise.

When an oscilloscope probe is placed on a wire within a typical digital system, crosstalk on the order of a tenth of a volt is commonly observed. Coupling is both capacitive and magnetic, and is also caused by dips in the positive power supply and common ground.

Regardless of the source, the important point about non-random noise in a finite digital system is that it has a finite amplitude bound. Unlike random noise, having all of the wires of a digital system switch from low to high will only induce a certain maximum of amplitude noise.

Thus, in terms of effect on absolute reliability, because cross-talk is so much greater a source of interference than random, unbounded processes, it makes sense to talk about digital system noise in terms of bounded amplitude waveforms rather than unbounded statistical waveforms of limited bandwidth and power.

Pulse systems are similar to digital ones in this respect because of the thresholding performed by their receivers. Thus, as in the digital case, we will consider the "important" noise in pulse systems to be bounded in amplitude. In a pulse machine, signals carry information in their time of arrival, so the noise under consideration will be the noise of time jitter rather than the noise of voltage shift. As shown below, additive amplitude noise on a passive pulse communications channel, when passed through a threshold triggered receiver, translates to equivalent time jitter.
Note how the pulse’s fast rise time serves to reduce the large amplitude noise into a small amount of time jitter\textsuperscript{36}.

For geometric reasons, noise added to a pulse signal transmitted on a passive, linear transmission line cannot cause time jitter in excess of the duration of a single pulse. However, if the transmission line is regenerative along its entire length, input thresholding is performed everywhere instead of just at the termination. Given a long enough regenerative line as such, jitter could then exceed the width of one pulse.

**Final Arguments in Support of Discrete Analysis**

In natural neural systems, and artificial pulse systems as well, the history controlled transmission gate "blocker" has been shown to be a powerful computational element. When an interval train receiver uses such a device to classify input messages, since the transmission or blocking of each pulse is a go / no-go decision, the classification of inputs rhythms is by nature a discrete operation.

This is not to say that a continuous phase encoding methodology could not also be employed within the discrete one. For example, let us say that we build an interval discriminator that routes pulses following intervals greater than 1 second to the right, and pulses following intervals less than 1 second to the left. Even though this left / right decision is discrete, the transmitter might still use an analog method of encoding values within the time intervals permitted in each range (i.e. .25 - .75 seconds for left-parsed intervals and 1.25 - 1.75 seconds for right parsed intervals).

But regardless of such auxiliary encoding, what fundamentally limits the partitioning of left hand intervals from right hand ones is noise, and since we have shown that the discriminator can be made significantly reliable with relatively small increases in the jitter

\textsuperscript{36} The figures above were created by passing a half cosine wave and white noise of 100\% and 20\% magnitudes, respectively, through an ideal low pass filter, with the receiver threshold drawn at 50\%.
margin over what would bring about unreliable parsing, it is reasonable to concentrate our arguments on very reliable, discrete systems.

There are three other arguments in support of this decision which we should reiterate:

First, we have seen that "forbidden" regions looking suspiciously like noise margins surround the peaks of interval probability distributions actually recorded from animals.

The existence of these forbidden regions is also necessary if interval times are not to be non-deterministically distorted. Leaving out time jitter noise margins means that the receiver will frequently be forced to decide intervals near its parsing point, and such decisions not only give indeterminate answers, they cause pulses to be generated at unpredictable times after excitation due to slow resolution of metastable conditions.

Finally, the coincidence detection function present in dendritic trees receiving input from the terminal arbors is expressing, in each detected coincidence, a particular combination of axonal transmission gate failures and successes. This coincidence detection is a discrete operation that is significantly upset by noise. Logical functions, by and large, are not forgiving of small errors in their inputs, and so it behooves us to avoid making errors of parsing in the first place.
Transmission Rates of Interval Encoded Channels

With a bounded model of noise, we now turn our attention to the issue of efficient transmission. For the unfamiliar reader, appendix E presents a derivation of the standard measures of information and channel capacity. We shall examine, in this section, what interval statistics we would expect to find on a pulse transmission line.

In keeping with our device limitations, a typical receiver of pulse messages has limited memory. Thus, we will consider only those impulse channels which convey intervals of finite length. Extremely long intervals may occur, but no discrimination is possible amongst them, and by our reckoning, they are all equivalent to the longest discernible interval.

As mentioned in the last chapter, our receiver's detection mechanism is discrete, so we will consider discrete symbol alphabets of finite size \( n \), where the interval time of each alphabet symbol from \( i = 0 \) to \( i = n - 1 \) is given by the variable \( t_i \). The average time to reliably transmit interval encoded symbols for a given set of \( t_i \) is thus:

\[
T(P) = \sum_{i=0}^{n-1} t_i p_i
\]

where \( p_i \) is the probability of sending symbol \( i \).

The average information of these symbols is:

\[
H(P) = - \sum_{i=0}^{n-1} p_i \log_2(p_i)
\]

where the utilized channel capacity is defined as \( C(P) = \frac{H(P)}{T(P)} \)

What distribution of symbol probabilities \( P \) maximizes the channel utilization \( C(P) \)? As first stated by Shannon\(^{37}\), an optimum encoding assigns symbol probabilities such that the information content of each symbol is in proportion to its transmission duration:

This can be shown by the method of Lagrange multipliers, and since we will use this technique later to address a more complex problem, let's become familiar with the algebra now:

We wish to maximize

$$C(P) = \frac{H(P)}{T(P)}$$

subject to the constraint that

$$\sum_{i=0}^{n-1} p_i = 1$$

By Lagrange's method, we let

$$G(P) = \left[ \sum_{i=0}^{n-1} p_i \right] - 1$$

and find a $\lambda$ which satisfies:

$$G(P) = 0 \quad \text{and} \quad \nabla C(P) + \lambda \nabla G(P) = 0$$

simultaneously. Substituting this $\lambda$ back into the first equation should yield the desired $P$.

To solve the right hand equation, we consider the individual components of each dimension:

$$\frac{\partial}{\partial p_j} \left[ \frac{H(P)}{T(P)} \right] + \lambda \frac{\partial}{\partial p_j} \left[ \sum_{i=0}^{n-1} p_i \right] - 1 = 0$$

evaluating the partials, we have:

$$\frac{T(P) \frac{\partial H(P)}{\partial p_j} - H(P) \frac{\partial T(P)}{\partial p_j}}{T^2(P)} + \lambda = 0$$

the partials of $H(P)$ and $T(P)$ are:

$$\frac{\partial H(P)}{\partial p_j} = -\left( \frac{1}{\ln(2)} + \log_2(p_j) \right) \quad \text{and} \quad \frac{\partial T(P)}{\partial p_j} = t_j$$

thus,

$$-T(P) \left[ \frac{1}{\ln(2)} + \log_2(p_j) \right] - H(P) t_j$$

moving all the given variables to the right yields:

$$\log_2(p_j) = \lambda T(P) - C(P) t_j - \frac{1}{\ln(2)}$$

or, in terms of $p_j$,

$$p_j = 2^{\left[ \frac{\lambda T(P) - 1}{\ln(2)} \right] - C(P) t_j}$$
To determine $\lambda$, we substitute the form of $p_j$ into our original definition of $H$:

$$H(P) = -\sum_{j=0}^{\lambda-1} p_j \log_2(p_j)$$

$$= -\sum_{j=0}^{\lambda-1} p_j \log_2\left(2^{\frac{\lambda T(P) - \frac{1}{\ln(2)}}{\ln(2)}} 2^{-C(P) t_j}\right)$$

$$= -\sum_{j=0}^{\lambda-1} p_j \left(\lambda T(P) - \frac{1}{\ln(2)} - C(P) t_j\right)$$

$$= -\left(\lambda T(P) - \frac{1}{\ln(2)}\right) \sum_{j=0}^{\lambda-1} p_j + C(P) \sum_{j=0}^{\lambda-1} p_j t_j$$

but $\sum_{j=0}^{\lambda-1} p_j = 1$ and $\sum_{j=0}^{\lambda-1} p_j t_j = T$, so that

$$H(P) = -\left(\lambda T(P) - \frac{1}{\ln(2)}\right) + H(P)$$

that is:

$$\lambda T(P) - \frac{1}{\ln(2)} = 0$$

solving for $\lambda$, we arrive at:

$$\lambda = \frac{1}{T(P) \ln(2)}$$

which causes our formula for $p_j$ to simplify to:

$$p_j = 2^{-C(P) t_j}, \quad \text{or} \quad -\log_2(p_j) = C(P) t_j$$

Thus, we see that the optimal probability distribution does indeed make the information content of each interval proportionate to the interval’s transmission time. The constant of proportionality, $C(P)$, is also the maximum utilized information rate of the channel. To determine this for a particular set of $t_j$, one simply solves:

$$\sum_{j=0}^{\lambda-1} 2^{-C(P) t_j} - 1 = 0$$

Since $t_j$ and $C(P)$ are always positive, the left hand side of this equation is monotonic in $C(P)$, making the solution easy to find. Although an equation for $C(P)$ cannot be stated analytically, Newton’s method with an initial guess of $C(P) = 0$ converges quickly. The needed derivative of the above equation with respect to $C(P)$ is:

$$-\ln(2) \sum_{j=0}^{\lambda-1} 2^{-C(P) t_j} t_j$$
So as to become familiar with this distribution, we consider now the case of a channel whose time jitter is independent of interval length. In this case, the $t_i$ values are evenly spaced, with a spacing of $w$ seconds for maximum time jitter of $\pm \frac{w}{2}$ seconds.

The non-zero width of each pulse (and, perhaps, the recovery time of the media after the passage of each pulse) also creates a minimum interval $t_{\min}$ between pulses.

Below is shown a typical transmission of a few symbols with $n = 4$:

```
'2'   '2'   '1'

\[ t_{\min} \]

\[ t_{\min} \]

\[ w \]

Possible pulses

Actual pulses
```

If the successively increasing interval times are taken as the symbols 0, 1, 2, and 3, the transmitted symbol stream shown from left to right is [2, 2, 1]. For $t_{\min} = w$, interval encoding is equivalent, in digital terms, to the comma code where a '1' (pulse) is followed by a certain number of '0's (silence) to represent each symbol.

To determine the optimal probabilities, we proceed as follows:

\[ t_j = t_{\min} + w \cdot j \quad \text{and} \quad p_j = 2^{-C(P) t_{\min}} 2^{-C(P) w j} \]

we enforce $\sum_{j=0}^{n-1} p_j = 1$, which gives:

\[ 2^{-C(P) t_{\min}} \sum_{j=0}^{n-1} 2^{-C(P) w j} = 1 \]

applying the simplification:
\[ \sum_{i=0}^{n-1} 2^{-\alpha i} = \frac{1 - 2^{-\alpha n}}{1 - 2^{-\alpha}} \]

we have

\[ 2^{-C(P)\tau_{\text{min}}} \frac{1 - 2^{-C(P)\epsilon n}}{1 - 2^{-C(P)\epsilon}} \]

which, when re-arranged, yields:

\[ 2^{-C(P)\epsilon n} + 2^{C(P)\tau_{\text{min}}} - 2^{C(P)(\tau_{\text{min}} - \epsilon)} - 1 = 0 \]

Given \( \tau_{\text{min}}, n, \) and \( \epsilon, \) this can be numerically solved for \( C(P). \) Note that as the alphabet size \( n \to \infty, \) if \( \tau_{\text{min}} = \epsilon \) (i.e. is its smallest possible value), then \( C(P) \to 1/\epsilon \) and \( p_i = 2^{-i + 1}. \) In this circumstance, each transmission time window has exactly one half chance of being occupied by a pulse, and as far as an outside observer is concerned, the channel is being utilized exactly as it would by a straight binary encoding (i.e. with the statistics of discrete white noise).

For a particular \( \tau_{\text{min}}, \epsilon, \) and \( n, \) once the optimal symbol probability distribution is determined, a suitable algorithm can be used to encode intervals with optimal statistics from input of non-optimal statistics. For example, to generate an approximate exponential distribution from a flat probability bit stream, we can encode interval lengths counting the number of successive 0's (or equivalently, successive 1's) in the input stream. In effect, this un-does the interval encoding to recreate the original bit stream on the transmission line.

**The effect of \( n \)**

The probability of long intervals being utilized goes down exponentially no matter how large one makes \( n, \) giving a practical limit to worthwhile interval lengths (in other words, forcing a pulse to occur every so often does not, even for small maximum interval lengths, decrease the channel utilization much). This will be important later, when we consider details of real receiver construction, so it is worthwhile to see graphically the effect of maximum interval length. Let us once again let \( \tau_{\text{min}} = \epsilon \) and plot the utilized capacity \( C \)
This curve approaches full utilization very quickly indeed. A digression of practical interest is in order now concerning the impressive result of 70% channel utilization for \( n = 2 \). This corresponds to transmit stream statistics of no more than a single successive time window not occupied by a pulse!

When a computer is faced with the task of encoding a base band serial bit stream onto a network cable or magnetic media, the statistics of the base band signal often conflict with that of the media. Serial bits allow for an unlimited number of successive 0’s (or 1’s) while the frequency response of the media often does not extend down to low frequencies. A simple modulation scheme is usually used to raise the guaranteed rate of signal transition. In most of these schemes the rate of signal transition is raised from 0-1 to 1-2 (normalized to the original bit rate). For example, one could arrange in every input bit cell time to transit the transmission line at the beginning of the cell, and transit it again in the middle if the bit to be transmitted is a one. If the bit is a zero, the line would stay quiescent until the end of the bit time. One transit per cell is thus guaranteed, and the maximum frequency of transitions is doubled.

It is clear from this that a medium modulated in this fashion is only sending bits at one half the maximum transition rate. But an optimal interval encoded scheme given the same medium and maximum interval time uses 70% of the channel capacity rather than 50%. How is this possible?
The wasted capacity occurs because once the middle transition occurs in a bit cell, the value of the bit is already known. The remaining time (to the next transition) is wasted; it could be the beginning of a new cell. We could improve matters by signifying a ‘1’ by two transitions spaced one unit apart and a ‘0’ by two transitions spaced two units apart. Since a ‘1’ takes less time to transmit than a ‘0’ and can immediately be followed by another bit, the average transmission duration is reduced from 2 units to 1.5. This raises the channel capacity from 50% to 66%.

To make it the rest of the way to 70%, the probabilities of sending a one or a zero must be brought into proportion with their transmission durations. This requires an encoding system to be used prior to the transmitter and a decoder after the receiver.

In summary, we see that an alphabet as small as \( n = 4 \) allows over 90% channel utilization, which means we can design an efficient interval communications system using transmitters and receivers that need not be capable of transmitting or receiving long intervals.

**The effect of \( t_{\text{min}} \)**

The effect of a minimum interval time is more dramatic, because it is reducing the channel’s transmission rate in the domain of the most information laden symbols - the short intervals. Here, plotted on a log scale, is both the effect of increased \( t_{\text{min}} \) (normalized to its minimum value of \( w \)) on straight binary coding (lower curve) and optimal interval coding (upper curve) with \( n = 50 \).
Because straight binary coding incurs the expense of $t_{\text{min}}$ for every bit, its utilization of the channel is significantly worse than interval coding, which incurs one $t_{\text{min}}$ time penalty per symbol, where each symbol encodes a larger number of bits.

**Receiver noise**

So far, we have considered noise only in term of constant channel time jitter, independent of the signal pattern being transmitted. But it is also important to consider how the receiver actually discerns various intervals, and with what precision.

A digital engineer, asked to design a receiver to measure time intervals between pulses, would most likely employ a high speed counter clocked by some precision oscillator (e.g. a quartz crystal). The intervals generated by the crystal are roughly gaussian in statistics, with an extremely small standard deviation. Compounding these statistics for many cycles causes the mean error to accumulate at a slow linear rate, and for most reasonable channel time jitters, would not be the limiting factor of precision.

If analog methods are used, however, it is imperative to know what form of signal is being used to measure time, and whether this signal evolves at a linear or (more likely) an inverted exponential rate.
As was previously discussed, the most simple analog time interval discriminator is the exponential blocker. The exponential blocker arbitrates whether a pulse arrives before or after a certain interval by comparing the input signal with a decaying threshold. Since the input pulse amplitude is assumed to be constant, a specific interval block corresponds to the instant where the receiver threshold crosses the nominal input pulse height.

Because the discrimination is being done in the amplitude domain, the rate of change of the threshold determines the precision of the receiver. A threshold that crosses the nominal pulse height slowly will be less precise in discerning time than one that crosses more rapidly. Thus, the precision of a blocking discriminator will depend on the length of interval to be blocked.

Two approaches can be taken to estimating this precision. One is to consider the parameters (time constants, etc ...) of the blocker to be fixed, and vary the interval to be discerned by the nominal input pulse height. The second is to keep the pulse height constant and vary the time constants of the blocking receiver.

In the first case, the derivative of the threshold function at various times is inversely proportional to the precision of measurement for that interval. In the second, we pick a particular pulse height and vary the time constants of the blocker, noting the derivative at the intercept of the threshold function and the input pulse height.

The most common normalized decaying threshold function is the inverse exponential:

\[ T = e^{-\frac{t}{\tau}} \]  

Its rate of change is: 

\[ \frac{dT}{dt} = -\frac{1}{\tau} e^{-\frac{t}{\tau}} = -\frac{T}{\tau} \]

The two parametric equations describing the jitter vs. interval time are thus:

\[ t = -\tau \ln(T) \]

\[ t_j \propto \frac{\tau}{T} \]

Eliminating \( \tau \) from these equations yields the relation of \( t_j \) to \( t \) when \( \tau \) (the decay's time constant) is varied and \( T \) is fixed:
\[ t_j \propto -\frac{t}{T \ln(T)} \]

When \( T \) (the pulse height) is varied and \( \tau \) is fixed,

\[ t_j \propto \tau e^{\frac{t}{\tau}} \]

Thus, we are faced with either a linear or exponential increase in time jitter with interval length.

With variable jitter, transmission durations are no longer evenly spaced. Instead, where jitter increases linearly with interval length,

\[ t_i = t_{i-1} + t_k + t_{i-1} - t_{i-2} \]

When jitter increases exponentially,

\[ t_i = t_{i-1} + t_k (t_{i-1} - t_{i-2}) \]

where \( t_0 = t_{\text{min}} \), \( t_1 = t_{\text{min}} + t_j \) and \( t_k \) in the above equations gives the rate of interval growth.

Interval dependent jitter does not change the optimality of assigning symbol information content in proportion to the interval length, but it does change the distribution of available intervals. This change is reflected in the statistics one would expect of received (noisy) intervals over the range of possible times. Where pulse height is used to control the receiver’s interval tuning, one would expect a distribution that followed:

\[ \frac{dp(t)}{dt} \propto 2^{-C(p) t} e^{-\frac{t}{\tau}} \]

while, for time-constant receiver tuning, the optimal distribution would obey:

\[ \frac{dp(t)}{dt} \propto 2^{-C(p) t} \frac{1}{t} \]
Power Considerations

A story is told in which a group of technologically advanced but artistically impoverished extraterrestrials pay us a visit, wishing to return home with a copy of all the poetry in the library of congress. After considering and rejecting our primitive storage techniques, they decide instead on the following scheme: A bar of otherwordly stable material 1 foot long is engraved in a most precise manner with a single scratch. The position of this scratch, as placed and measured by the excruciatingly advanced instruments of the alien civilization, is made accurate to as many bits as there are bits of poetry in our entire library. By simply reading out the position of the scratch, all the data is recovered. The aliens return home, bar in hand, overjoyed by their bounty of human creativity.

Now of course for us, a scratch in a one foot long bar could probably be read to an accuracy of perhaps 20 bits, which is about 4 alphabetic characters, never mind a whole library. But the hypothetical option to encode data this way does exist. It is the design philosophy of trying to minimize the number of marks used in an encoding, taken to the ridiculous extreme of a single mark.

When this idea is not taken to such extreme, it can make sense. A more realistic stipulation is the following: The number of scratches (marks) used to encode data conveyed by a communication media can become a cost constraint on par with the constraints of limited bandwidth and accuracy.

For example, in the nervous system, the transmission of every pulse along a nerve fiber brings about the consumption of an unavoidable quantity of energy, and in all living beings, energy is a very valuable commodity. It is a remarkable fact that the human brain, accomplishing the most complex of tasks, consumes only about 40 watts (and yet this is still a significant fraction of the resting body’s total power consumption). Surely, power consumption was an issue in the evolution of the nervous system, and per-mark energy management is also an inherent constraint in artificial pulse transmission systems.
Digital signalling systems with transition dependent energy consumption (such as CMOS) exhibit a per-transition cost similar to the per-pulse cost of pulsatile systems. A problem of this sort was recently related to me by Jeff Tabor, a master’s student of Professor Tom Knight:

When a bus of wires is used to communicate values between two sections of a computer system, the wires on that bus are usually synchronized in their possible transition times. To reduce noise and minimize transmit power, it is advantageous to employ a coding scheme that minimizes the number of wires simultaneously changing state. Doing so minimizes the energy consumed charging the bus wiring capacitance and also minimizes the pulsatile ground return current.

Jeff’s task is to design a system to encode the $2^n$ possible symbols from a source of $n$ bits onto a wider bus of $m$ wires where the number of bit transitions made by that $m$ wire bus from one symbol to the next will be less than the number made on the original bus.

In our last section, we saw that increasing the probability of short intervals led to better utilization of a channel, i.e. higher information efficiency in bits per second. This led to the ultimate inverse exponential distribution which transmitted close to a full bit of information per interval resolution time. Unfortunately, this optimization is bad from a power consumption point of view, since short intervals being more likely leads to a higher average pulse frequency, and thus higher power consumption. Like Jeff, we may wish to sacrifice some of the possible bandwidth of our channel so as to consume less power, even though an increase in channel resolution may be demanded to maintain a constant bit rate.

**Power Efficiency: The Case Against Information Proportionate Statistics**

To simply maximize the power efficiency of a finite resolution pulse channel, we maximize $H(P)$, the bits transmitted per pulse:

$$H(P) = -\sum_{i=0}^{n-1} p_i \log_2(p_i)$$

Along with the side constraint that:
\[ \sum_{i=0}^{n-1} p_i = 1 \quad \text{and} \quad G(P) = \sum_{i=0}^{n-1} p_i - 1 \]

As in our maximization of \( C(P) \), we solve \( \nabla H(P) + \lambda \nabla G(P) = 0 \), subject to \( G(P) = 0 \):

\[
- \left( \frac{1}{\ln(2)} + \log_2(p_j) \right) + \lambda = 0.
\]

Thus

\[ p_j = 2^{\left(-\alpha - \frac{1}{\ln(2)}\right)} \]

In other words, all \( p_j \) are equal. Since \( \sum_{i=0}^{n-1} p_i = 1 \):

\[ p_j = \frac{1}{N} \quad (j = 0 \ldots N - 1) \]

This makes sense intuitively, because \( H(P) \) does not consider transmission time whatsoever, and hence every interval is just like any other. It makes sense to allow for maximum free choice among the symbols.

Given this fixed value of \( p_j \), the bits transmitted per symbol is:

\[ H(P) = -\sum_{j=0}^{n-1} \frac{1}{N} \log_2\left(\frac{1}{N}\right) = -\log_2\left(\frac{1}{N}\right) = \log_2(N) \]

which, of course, is just the number of bits of information in choosing 1 of \( N \) equally likely possibilities.

To increase this maximum of power efficiencies, we must increase \( N \) (i.e. the length of the longest interval).

**The Trouble with Maximum Power Efficiency**

Unfortunately, as we increase \( N \), while the number of bits per symbol goes up logarithmically, the rate of symbol transmission drops faster - linearly. Thus, increasing channel resolution to make up for lost bit rate will result in higher total power. Increasing \( N \)
in this fashion is also unrealistic, because we have seen that analog interval discriminators have trouble discerning long intervals.

**Maximum Capacity with Limited Power**

Let us thus state our problem more carefully. With a specified power (or, equivalently, average interval \(T(P) = T_0\)), maximum alphabet size \(N\), and particular distribution of available intervals \((t_j)\), what probability distribution \(P\) maximizes \(C(P)\)?

Employing the same Lagrange multiplier method as before, but with an additional constraint of \(T(P) - T_0 = 0\), we have:

\[
\nabla C(P) + \lambda_1 \nabla G(P) + \lambda_2 \nabla (T(P) - T_0) = 0
\]

component-wise, this is:

\[
- T(P) \left[ \frac{1}{\ln(2)} + \log_2(p_j) \right] - \frac{H(P) t_j}{T^2(P)} + \lambda_1 + \lambda_2 t_j = 0
\]

moving all given variables to the right yields:

\[
\log_2(p_j) = \lambda_1 T(P) - \frac{1}{\ln(2)} + (\lambda_2 T(P) - C(P)) t_j
\]

or

\[
p_j = 2^{\left( \lambda_1 T(P) - \frac{1}{\ln(2)} \right) / \lambda_2 T(P) - C(P))} t_j
\]

as in the unlimited power case, we substitute this back into our definition of \(H(P)\):

\[
H(P) = - \left( \lambda_1 T(P) - \frac{1}{\ln(2)} \right) \sum_{j=0}^{n-1} p_j - \left[ \lambda_2 T(P) - C(P) \right] \sum_{j=0}^{n-1} p_j t_j
\]

noting that \(\sum_{j=0}^{n-1} p_j = 1\) and \(\sum_{j=0}^{n-1} p_j t_j = T(P) = T_0:\)

\[
H(P) = - \left( \lambda_1 T_0 - \frac{1}{\ln(2)} \right) - \lambda_2 T_0^2 + H(P)
\]
\[
\lambda_1 T_0 - \frac{1}{\ln(2)} = -\lambda_2 T_0^2
\]

thus,

\[
\rho_j = 2^{-\lambda_1 T_0^2} 2^{\alpha x - \frac{H(P)}{T_0}} t_j
\]

In this equation, \( T_0 \) is known, leaving two unknown variables, \( \lambda_2 \) and \( H(P) \). We can rewrite the equation for \( \rho_j \) in terms of two simpler independent variables:

\[
a = 2^{-\lambda_1 T_0^2} \quad \text{and} \quad b = \left( \frac{H(P)}{T_0} - \lambda_2 T_0 \right)
\]

in which case:

\[
\rho_j = a 2^{-b t_j}
\]

thus, from our valid probability sum and average power constraints:

\[
a \sum_{j=0}^{n-1} 2^{-b t_j} = 1 \quad \text{and} \quad a \sum_{j=0}^{n-1} 2^{-b t_j} t_j = T_0
\]

dividing the right hand equation by the left removes the variable \( a \):

\[
\frac{\sum_{j=0}^{n-1} 2^{-b t_j} t_j}{\sum_{j=0}^{n-1} 2^{-b t_j}} - T_0 = 0
\]

As before, this can’t be solved analytically, but since \( T_0 \) and all \( t_j \)'s are known and positive, Newton’s method can be employed. The derivative of the above function with respect to \( b \) is:

\[
\ln(2) \left[ \sum_{j=0}^{n-1} 2^{-b t_j} \right]^{n-1} \sum_{j=0}^{n-1} 2^{-b t_j} t_j^2 - \ln(2) \left[ \sum_{j=0}^{n-1} 2^{-b t_j} t_j \right]^2
\]

Once \( b \) is known, \( a \) can be found by:

\[
a = \frac{1}{\sum_{j=0}^{n-1} 2^{-b t_j}}
\]
Input Statistics - Conclusions

We have seen that, given limited power, the probability of choosing \( t_j \) length symbols for optimal utilization of channel capacity should follow:

\[
P_j = a \ 2^{-b\cdot j}
\]

Where \( b \) is a function of the power limit and \( a \) is determined once \( b \) is chosen.

Lowering the power limit has the effect of sloshing the exponential curve from a decreasing function of \( t_j \) (positive \( b \)) to an increasing function of \( t_j \) (negative \( b \)). The greatest power efficiency is had with a flat distribution \( (b = 0, a = \frac{1}{n}) \).

The effect of interval-dependent noise due to receiver implementation does not change the validity of this analysis. However, if we want to establish a post-noise probability curve for various interval times, we must divide the optimal symbol distribution by the noise margin width of each symbol. For an optimal power efficiency (i.e. flat) distribution, with an exponentially decaying threshold receiver and receiver tuning by pulse height, this results in:

\[
\frac{dp(t)}{dt} \propto e^{-\frac{t}{\tau}}
\]

while, for time-constant receiver tuning, the optimal distribution would obey:

\[
\frac{dp(t)}{dt} \propto \frac{1}{t}
\]

This first distribution of intervals is that of the Poisson process, where every time interval has a constant, independent, probability of being occupied by a pulse. It is this distribution we shall use for the simulator's input.
Appendix E - Terminology of Information Capacity

The computer industry has created some confusion in the terminology of information by mislabelling telephone modem channel capacity in baud rate instead of bits per second. Below is a brief review of discrete channel capacity in correct terminology.

When a machine sends discrete symbols over a communications channel, be they amplitudes, pulses, frequencies, phase shifts, or what have you, it chooses each symbol from an alphabet of possible symbols. Assuming the data stream to be memoryless, each symbol has an independent probability of being chosen for transmission. For some encodings (e.g. morse code), each symbol also has a different length of time that is required for transmission [e.g. a morse code "o" (three dashes) takes longer to send than a more code "e" (one dot)].

The average rate at which symbols are transmitted is correctly called the "baud rate". Because each symbol may signify a different amount of information, the average amount of information that is communicated (i.e. the channel capacity C) is correctly termed the "bit rate".

As an illustrative example, so-called 9600 "baud" modems actually transmit a single symbol from an equal probability alphabet of 16 possibilities 2400 times per second (they do this because a telephone line has only a 3000 Hertz bandwidth). Thus, their symbol rate is actually 2400 baud, not 9600.

The information content of a symbol probabilistically chosen from an alphabet is defined as:

\[ H_i = - \log_2(p_i) \]

where \( i \) is the index of the particular symbol and \( p_i \) the probability of choosing that symbol. Thus, the weighted average information of all symbols in the alphabet is:

\[ H_{avg} = - \sum_i p_i \log_2(p_i) \]

The average symbol rate is similarly defined (in baud):
\[ R_{\text{avg}} = \frac{1}{\sum_i p_i t_i} \]

Where \( t_i \) is the time required to send message \( i \). The channel capacity \( C \) in bits per second is then:

\[ C = H_{\text{avg}} R_{\text{avg}} = -\frac{\sum_i p_i \log_2(p_i)}{\sum_i p_i t_i} \]

Returning to our example of the so-called 9600 baud modem, each symbol has probability \( \frac{1}{16} \), thus encoding \( H_{\text{avg}} = 4 \) bits of information. All symbols take \( \frac{1}{2400} \) second to transmit, leading to a symbol rate \( R_{\text{avg}} = 2400 \) baud and a channel capacity of \( C = 9600 \) bits per second.
Appendix F - Listing of Event Driven Simulator Kernal

/*
 * event.h: event queue header file
 */

typedef struct e e;

struct e
{
    int (*funep)();    /* Event function */
    int arg;          /* Event Argument */
    double darg;      /* Double argument */
    double time;      /* Time to wait before triggering */
    e     *nep;       /* Next event */
    e     *pep;       /* Previous event */
};

extern e *fep, *lep;    /* First and last events */

extern double Time;     /* What time it is now */

/*
 * Event driven simulator routines
 *
 * Gill Pratt
 *
 * 1989
 */

/*
 * Access to this code is made via two routines:
 *
 * event (funep, arg, time, undup, darg)
 *     int *funep, arg, undup, darg;
 *     double time;
 *     double darg;
 *
 * causes (*funep)(arg, darg) to be called in the
 * future after relative "time" has passed from now. The second argument of the call, "darg"
 * is an optional double precision floating point argument. The external double "Time" is
 * available, giving the elapsed time since the beginning of the run.
 *
 * if "undup" is set, all previous events with
 * identical funep and arg are first deleted
 * from the future event queue.
 *
 * run(tlimit)
 * */
* causes events to run till the queue is empty or
* the Time > 'tlimit'.
 *
* The debugging routine prevents() prints the current event queue on
* the standard error.
* /

#include <stdio.h>
#include "event.h"

e *fep, *lep;                /* Pointer to first and last events */
double Time;                  /* Now */

event (funcp, arg, time, undup, darg)
  int (*funcp)();
  int arg;
  double time;
  int undup;
  double darg;
{
  e *ep, *nep, *next;

  if (! (ep = (e *) malloc(sizeof (e))))
  {
    fprintf (stderr, "Out of memory\n");
    exit (0);
  }

  if (undup) /* Remove duplicates if desired */
  for (nep = fep; nep; nep = next)
    if (nep->funcp == funcp && nep->arg == arg)
    {
      if (next = nep->nep)
        next->time += nep->time;

        remove (nep);
    }
  else
    next = nep->nep;

  for (nep = fep; nep && nep->time < time; nep = nep->nep)
    time -= nep->time;

  ep->funcp = funcp;
  ep->arg = arg;
  ep->time = time;
  ep->darg = darg;

  if (nep)
    nep->time -= time;

  insert (ep, nep);
}
run(tlimit)
    double tlimit;
{
    int (*funcp)();
    int arg;
    double darg;

    for (Time = 0.; fep && Time < tlimit; )
    {
        Time += fep->time;

        funcp = fep->funcp;
        arg = fep->arg;
        darg = fep->darg;

        remove (fep);

        (*funcp)(arg, darg);
    }

    while (fep)
        remove (fep);
}

/* Insert event ep in front of nep; (! nep) means the end of the queue */
insert (ep, nep)
    e *ep, *nep;
{
    if (nep)
    {
        if (ep->pep = nep->pep)
            nep->pep->nep = ep;
        else
            fep = ep;

        nep->pep = ep;
        ep->nep = nep;
    }
    else
    {
        if (ep->pep = lep)
            lep->nep = ep;
        else
            fep = ep;

        lep = ep;
        ep->nep = 0;
    }
}

remove (ep)      /* Remove an event from the queue and free it */
    e *ep;
{

if (ep->nep)
    ep->nep->pep = ep->pep;
else
    lep = ep->pep;

if (ep->pep)
    ep->pep->nep = ep->nep;
else
    fep = ep->nep;

free (ep);
}

prevents()
{
    e *ep;

    for (ep = fep; ep; ep = ep->nep)
        fprintf (stderr, "%g\t%f (%x, %g)",
                     ep->time, ep->funcp, ep->arg, ep->darg);
}
Appendix G: Listing of Absolute Blocking No Integration Simulator

/*
 * Simple blocker event driven pulse network simulator
 *
 * Gill Pratt
 *
 * 1989
 *
 * Use: block [[-option_name option_value] ...] w d b < in
 *
 *       where:
 *
 *       w     is the weight matrix file
 *       d     is the delay matrix file
 *       b     is the blocking time list file
 *       in    is the time ordered file of input events
 */

#include <math.h>
#include <stdio.h>
#include "../event/event.h"

struct param
{
    char *name;
    double value;
};

typedef struct param param;

param params[] =
{
    "t",  1.e6,  /* Time limit */
    "d",  0.,   /* Debug option */
    "r",  0.,   /* Retrigger option */
    "s",  1.e-10, /* Simultaneity criterion */
    "c",  0.,   /* Input count option */
    0,     0.,
};

#define TLIMIT (params[0].value)
#define DEBUG (params[1].value > 0.)
#define RETRIG (params[2].value > 0.)
#define SIMUL (params[3].value)
#define COUNT (params[4].value)

char *black();
char *white();

struct unit
{
double first;  /* Time of last excitation */
double unitno;  /* Floating point of unit number for fast i*/
double btime;  /* Blocking time */
struct link *links;  /* Ptr to first of output link list */
);

typedef struct unit unit;
unit *units, *endunits;

unit initunit =
{
  -1.e20,  /* I fired a long time ago (in a galaxy far away ...) */
  0.,    /* Will be calculated */
  0.,    /* Will be read in */
  0,     /* Initially, I have no output links */
};

int N;  /* Number of units */

struct link
{
  unit *from;  /* Source unit */
  unit *to;    /* Destination unit */
  double weight;  /* Link weight */
  double delay;  /* Link delay */
  struct link *next;  /* Next of source unit's output links */
};

typedef struct link link;

#define advance(argc, argv, n) (argc -= n, argv += n)  /* Advance to the n'th argument */

char obuf[BUFSIZ], ibuf[BUFSIZ];  /* Stdio buffers to make operation faster */

main(argc, argv)
int argc;
char **argv;
{
  int n;

  setbuf (stdin, ibuf);
  setbuf (stdout, obuf);

  advance (argc, argv, 1);  /* Skip command name */

  n = argparse(argc, argv);

  advance (argc, argv, n);

  if (argc != 3)
    fatal ("Use: block [-option_name option_value] ...] weightfile delay

  readnet (argv[0], argv[1], argv[2]);
inqueue();

run (TLIMIT);

fflush (stdout);

if (COUNT)
    fprintf (stderr, "%g\n", COUNT);

exit (0);
}

argparse (argc, argv) /* Parse optional arguments */
    int argc;
    char **argv;
{
    param *param;
    int oargc = argc;

    while (argc && argv[0][0] == '-')
    {
        for (param = params; param->name; param++)
            if (! strcmp (param->name, argv[0] + 1))
                break;

        if (param->name)
            advance (argc, argv, 1);

        if (sscanf (argv[0], "%lf", &param->value) != 1)
            fatal ("Bad option value: %s\n", argv[0]);

        advance (argc, argv, 1);
    }
    else
        fatal ("Unknown option (%s)\n", argv[0]);

    return (oargc - argc);
}

inqueue() /* Queue up the next input firing event */
{
    int n;
    unit u;
    int infire();

    if (fread (&u, 2 * sizeof (double), 1, stdin) == 1)
    {
        n = u.unitno;

        if (n < 0 || n >= N)
            fatal ("%d: bad unit index in input line %g\n", n, COUNT);
        else if (COUNT && ((int) (++COUNT) % 100) == 0)
fprintf (stderr, "%g\r", COUNT);

event (infire, units + n, u.firet - Time, 0);
}

infire (up) /* Fire an input unit and cause the next input event to be queued */
{
    unit *up;
    fire (up);
    inqueue();
}

fire (up) /* Indicate the firing of a unit and queue up excitation of other units */
{
    unit *up;
    int excite();
    link *lp;
    up->firet = Time;
    fwrite (up, 2 * sizeof (double), 1, stdout);
    for (lp = up->links; lp; lp = lp->next)
        event (excite, lp->to, lp->delay, 0, lp->weight);
}

excite (up, weight) /* Excite a unit, firing if conditions are correct */
{
    unit *up;
    double weight;
    e *ep, *nep;
    double future = 0.0;

    /* Swallow up excitation to this unit in the near future */
    for (ep = fep; ep && (future + ep->time) < SIMUL; ep = nep)
    
        nep = ep->nep;

        if ((unit *) ep->arg == up && ep->funcp == excite)
            
                weight += ep->darg;

                if (nep)
                    nep->time += ep->time;

                    remove (ep);
            
    }

    if (weight > 0.)
    {
if (Time - up->firet > up->btime)
    fire (up);
else if (RETRIG)
    up->firet = Time;
}

readnet (wfn, dfn, bfn) /* Read in the network information */
    char *wfn, *dfn, *bfn;
{
    FILE *wf, *df, *bf, *myopen();
    char ws[256], ds[256], bs[256];
    char *wcp, *dcp;
    double weight;
    unit *up;
    link *lp;
    int line, i;

    wf = myopen (wfn);
    df = myopen (dfn);
    bf = myopen (bfn);

    for (line = 1; fgets(ws, sizeof ws, wf); line++, up++)
    {
        myfgets (ds, sizeof ds, df, dfn);
        myfgets (bs, sizeof bs, bf, bfn);

        if (! units)
        {
            for (N = 0, wcp = ws; *(wcp = black(wcp)); N++)
                wcp = white (wcp);

            up = units = (unit *) malloc (N * sizeof (* units));
            endunits = units + N;
        }

        if (up >= endunits)
            linerror ("more rows than columns in first row", wfn, line);

        *up = initunit;
        up->unitno = up - units;

        sscanf (bs, "%lf", &up->btime);

        for (i = 0, wcp = ws, dcp = ds; *(wcp = black(wcp)); i++, wcp = white
        {
            if (! *(dcp = black (dcp)))
                linerror ("fewer columns than in first row of weight file", d);

            sscanf (wcp, "%lf", &weight);

            if (weight != 0.)
            {
                lp = (link *) malloc (sizeof * lp);
            }
lp->weight = weight;
sscanf (dcp, "%lf", &lp->delay);
lp->from = up;
lp->to = units + i;
lp->next = up->links;
up->links = lp;
}
if (i != N)
   linerror ("different number of columns than in first row", wfn, li);
}
if (up < endunits)
   linerror ("less rows than columns in first row", wfn, line);
fclose (wf);
fclose (df);
fclose (bf);

FILE *myopen (name)   /* Just like fopen, but complains if can't open */
    char *name;
{
    FILE *result;
    if (result = fopen (name, "r"))
      return (result);
    else
      fatal ("Can't open %s\n", name);
}

myfgets (s, ssize, stream, fname)   /* Just like fgets, but complains if e
    char *s;
    int ssize;
    FILE *stream;
    char *fname;
{
    if (! (fgets (s, ssize, stream)))
      fatal ("Premature end of file from %s\n", fname);
}

linerror (trouble, fname, line)   /* Shorthand function for fatal troub
    char *trouble;
    char *fname;
    int line;
{
    fatal ("%s (line %d) : %s\n", fname, line, trouble);
}

int debug (s, a1, a2, a3, a4, a5, a6)   /* Mention debugging information */
    char *s;
int a1, a2, a3, a4, a5, a6;
{
    fprintf (stderr, "pulse: debug - ");
    fprintf (stderr, s, a1, a2, a3, a4, a5, a6);
}

fatal (s, a1, a2, a3, a4, a5, a6)  /* Blurt out a message and die */
char *s;
int a1, a2, a3, a4, a5, a6;
{
    fprintf (stderr, "pulse: fatal error - ");
    fprintf (stderr, s, a1, a2, a3, a4, a5, a6);
    exit (1);
}
Appendix H - Listing of Exponential Blocking Exponential Integration Simulator

/*
   Dual Exponential relaxation event driven pulse network simulator

   Gill Pratt

   1989

   Use: pulse [\{-option_name option_value\} ...] w d h i < in

   where:

   w is the weight matrix file
   d is the delay matrix file
   b is the blocker time constant list file
   i is the integrator time constant list file
   in is the time ordered file of input events
*/

#include <math.h>
#include <stdio.h>
#include "../event/event.h"

struct param
{
    char *name;
    double value;
};

typedef struct param param;

param params[] =
{
    "imax", 1., /* Maximum integrator voltage */
    "imin", -1., /* Minimum integrator voltage */
    "irest", -1., /* Resting integrator voltage */
    "itaumin", 1.e-7, /* Shortest integrator time constant */
    "bmax", 1.01, /* Maximum value of blocker threshold */
    "bmin", 0., /* Minimum value of blocker threshold */
    "btaumin", 1.e-6, /* Shortest blocker time constant */
    "t", 1.e6, /* Time limit */
    "d", 0., /* Debug option */
    "s", 1.e-10, /* Simultaneous criterion */
    "c", 0., /* Input count option */
    "e", .0001, /* Firing voltage hysteresis criterion */
    "t/c", .7, /* Seconds / Farads per excitation */
    0, 0.,
};

#define IMAX (params[0].value)
#define IMIN (params[1].value)
#define IREST (params[2].value)
#define ITAUMIN (params[3].value)
#define BMAX (params[4].value)
#define BMIN (params[5].value)
#define BTAUMIN (params[6].value)
#define TLIMIT (params[7].value)
#define DEBUG (params[8].value > 0.)
#define SIMUL (params[9].value)
#define COUNT (params[10].value)
#define EPSILON (params[11].value)
#define TIMCAP (params[12].value)

    /* Relaxation of integrator voltage */
#define I(up,t) \  
((up->iv - IREST) * exp (- (t - up->it) / up->itau) + IREST) 

    /* Derivative of I() with respect to t */
#define dI(up,t) \ 
((up->iv - IREST) * exp (- (t - up->it) / up->itau) / (- up->itau)) 

    /* Relaxation of blocker threshold after firing */
#define B(up,t) \ 
((BMAX - BMIN) * exp (- (t - up->firet) / up->btau) + BMIN) 

    /* Derivative of B() with respect to t */
#define dB(up,t) \ 
((BMAX - BMIN) * exp (- (t - up->firet) / up->btau) / (- up->btau)) 

    /* Time of maximum over-trigger */
#define ZERO(up) \ 
((up->btau * up->itau * \ 
log (up->btau * (up->iv - IREST) / (up->itau * (BMAX - BMIN))) + \ 
up->btau * up->it - up->itau * up->firet) / (up->btau - up->itau)) 

    /* Pumping of integrator voltage */
#define PUMP(up,volts,mhos) \ 
(volts - (volts - up->iv) * exp (- TIMCAP * mhos)) 

char *black();
char *white();

struct unit 
{ 
    double firet;    /* Time of last blocker firing */
    double unitno;   /* Floating point of unit number for fast i/o */
    double btau;     /* Blocker time constant */
    double itau;     /* Integrator time constant */
    double iv;       /* Last integrator voltage */
double it;  /* Time of last integrator voltage */
struct link *links;  /* Ptr to first of output link list */
int flags;  /* Binary flag bits */
);

typedef struct unit unit;

#define WARNED 1  /* Flag bit */

unit *units, *endunits;

#define LONGTIME 1.e20

unit initunit =
{
  -LONGTIME,  /* I fired a long time ago (in a galaxy far away ... */
    0.,
    0.,
    0.,
    0.,
    0.,
    0,
    0,
};

int N;  /* Number of units */

struct link
{
  unit *from;  /* Source unit */
  unit *to;  /* Destination unit */
  double weight;  /* Link weight (conductance) */
  double delay;  /* Link delay */
  struct link *next;  /* Next of source unit's output links */
};

typedef struct link link;

#define advance(argc,argv,n)  (argc -= n, argv += n)

char obuf[BUFSIZ], ibuf[BUFSIZ];

main(argc, argv)
  int argc;
  char **argv;
{
  int n;

  setbuf (stdin, ibuf);
  setbuf (stdout, obuf);

  advance (argc, argv, 1);  /* Skip command name */

  n = argpaparse(argc, argv);
advance (argc, argv, n);

if (argc != 4)
    fatal ("Use: pulse [{-oname ovalue} ...] wfile dfile bfile ifile");

readnet (argv[0], argv[1], argv[2], argv[3]);
inqueue();
run (TLIMIT);
fflush (stdout);

if (COUNT)
    fprintf (stderr, "%g\n", COUNT);
exit (0);

argparse (argc, argv) /* Parse optional arguments */
        int argc;
        char **argv;
{
    param *paramp;
    int oargc = argc;

    while (argc && argv[0][0] == '-')
    {
        for (paramp = params; paramp->name; paramp++)
            if (!strcmp (paramp->name, argv[0] + 1))
                break;

        if (paramp->name)
            {
                advance (argc, argv, 1);

                if (sscanf (argv[0], "%lf", &paramp->value) != 1)
                    fatal ("Bad option value: %s", argv[0]);

                advance (argc, argv, 1);
            }
        else
            fatal ("Unknown option (%s)", argv[0]);
    }
return (oargc - argc);
}
inqueue() /* Queue up the next input firing event */
{
    int n;
    unit u;
    int infire();

    if (fread (&u, 2 * sizeof (double), 1, stdin) == 1)
n = u.unitno;

if (n < 0 || n >= N)
    fatal ("%d: bad unit index in input line %g", n, COUNT);
else if (COUNT & ((int) (++COUNT) % 100) == 0)
    fprintf (stderr, "%g\r", COUNT);

    event (infire, units + n, u.firet - Time, 0);
}
}

infire (up)    /* Fire an input unit and cause the next input event to be que
    unit *up;
    !
    fire (up);
    !
    inqueue();
    */

fire (up)    /* Indicate the firing of a unit and queue up excitation of li
    unit *up;
    {;
    int excite();
    link *lp;

    up->iv = I(up, Time);
    up->it = Time;
    up->firet = Time;

    fwrite (up, 2 * sizeof (double), 1, stdout);
    for (lp = up->links; lp; lp = lp->next)
        event (excite, lp->to, lp->delay, 0, lp->weight);
    
    relax (up);
    }

excite (up, weight)    /* Excite a unit, firing if conditions are correct */
    unit *up;
    double weight;
    {
    e *ep, *nep;
    double future = 0.;
    double posweights, negweights;
    double volts, mhos;

    if (weight > 0.)
    {
        posweights = weight;
        negweights = 0.;
    }
    else
negweights = -weight;
posweights = 0.;
}

/* Swallow up excitation to this unit in the near future */
for (ep = fep; ep && (future += ep->time) < SIMUL; ep = nep)
{
    nep = ep->nep;
    if (((unit *) ep->arg == up && ep->funcp == excite)
        if ((weight = ep->darg) > 0.)
            posweights += weight;
        else
            negweights -= weight;
    if (nep)
        nep->time += ep->time;
    remove (ep);
}

mhos = posweights + negweights;
vols = (posweights * IMAX + negweights * IMIN) / mhos;
up->iv = I(up, Time);
up->it = Time;
up->iv = PUMP(up, vols, mhos);
if (DEBUG)
    debug ("%g excite %d\tnew integrator voltage = %g", Time, up - units,
        relax (up);
}

relax (up)    /* Relax a unit, checking for blocker trigger */
unit *up;
{
double t, vdelta, vdiff;
if (DEBUG)
    debug ("%g: relax unit %d - S = %g, H = %g", Time, up - units, I(up, T
    /* Check if we are over threshold from the start */
if (I(up, Time) > B(up, Time))
{
    if (DEBUG)
        debug ("\timmediate fire");
    event (fire, up, 0., 1);
return;
}

/* See if we will cross in the future */
if (up->iv <= IREST) /* Inhibited */
{
    if (DEBUG)
        debug ("\tinhibited");

    return;
}

t = (up->btau == up->itau) ? LONGTIME : ZERO (up);

if (t < Time || I(up, t) < B(up, t))
{
    if (DEBUG)
        debug ("\twon’t fire");

    return;
}

/* Using Newton’s method, converge on future intersect */
for (t = (t + Time) / 2. ; ; t -= vdelta / vdiff)
{
    vdelta = I(up, t) - B(up, t);

    if (fabs (vdelta) < EPSILON)
    {
        event (fire, up, t - Time, l);

        if (DEBUG)
            debug ("\twill fire in %g seconds", t - Time);

        if (! (up->flags & WARNED))
        {
            debug ("warning - unit %d delayed firing (%g seconds)", up - u, up->flags != WARNED;

        }

        return;
    }

    vdiff = dI(up, t) - dB(up, t);
}

readnet (wfn, dfn, hfn, sfn) /* Read in the network */
    char *wfn, *dfn, *hfn, *sfn;
{
    char ws[256], ds[256], hs[256], ss[256];
    char *wcp, *dcp;
double weight;
unit *up;
link *lp;
int line, i;

wf = myopen (wfn);
df = myopen (dfn);
hf = myopen (hfn);
sf = myopen (sfn);

for (line = 1; fgets(ws, sizeof ws, wf); line++, up++)
{
    myfgets (ds, sizeof ds, df, dfn);
    myfgets (hs, sizeof hs, hf, hfn);
    myfgets (ss, sizeof ss, sf, sfn);

    if (! units)
    {
        for (N = 0, wcp = ws; *(wcp = black(wcp)); N++)
            wcp = white (wcp);

        up = units = (unit *) malloc (N * sizeof (* units));
        endunits = units + N;
    }

    if (up >= endunits)
        linerror ("more rows than columns in first row", wfn, line);

    *up = initunit;
    up->unitnc = up - units;
    up->iv = IREST;

    sscanf (hs, "%lf", &up->btau);

    if (up->btau < BTAUMIN)
        up->btau = BTAUMIN;

    sscanf (ss, "%lf", &up->itau);

    if (up->itau < ITAUMIN)
        up->itau = ITAUMIN;

    for (i = 0, wcp = ws, dcp = ds; *(wcp = black(wcp)); i++, wcp = white
    {
        if (! *(dcp = black (dcp)))
            linerror ("fewer columns than in first row of weight file", df

        sscanf (wcp, "%lf", & weight);

        if (weight != 0.)
            {  
            lp = (link *) malloc (sizeof * lp);

            lp->weight = weight;  
        }
sscanf (dcp, "%lf", &lp->delay);
lp->from = up;
lp->to = units + i;
lp->next = up->links;
up->links = lp;
}
}

if (i != N)
    linerror ("different number of columns than in first row", wfn, li)
}

if (up < endunits)
    linerror ("less rows than columns in first row", wfn, line);

fclose (wf);
fclose (df);
fclose (hf);
fclose (sf);
}

FILE *myopen (name)  /* Just like fopen, but complains if can’t open */  
char *name;
{
    FILE *result;

    if (result = fopen (name, "r"))
        return (result);
    else
        fatal ("Can’t open %s", name);
}

myfgets (s, ssize, stream, fname)  /* Just like fgets, but complains if e 
char *s;  
int ssize;
FILE *stream;
char *fname;
{
    if (! (fgets (s, ssize, stream)))
        fatal ("Premature end of file from %s", fname);
}

linerror (trouble, fname, line)  /* Shorthand function for fatal troubl 
char *trouble;
char *fname;
int line;
{
    fatal ("%s (line %d) : %s", fname, line, trouble);
}

debug (s, a1, a2, a3, a4, a5, a6)  /* Mention debugging information */  
char *s;
int a1, a2, a3, a4, a5, a6;
{
    fprintf (stderr, "pulse: debug - ");
    fprintf (stderr, s, a1, a2, a3, a4, a5, a6);
    fprintf (stderr, "\n");
}

fatal (s, a1, a2, a3, a4, a5, a6)  /* Blurt out a message and die */
    char *s;
    int a1, a2, a3, a4, a5, a6;
{
    fprintf (stderr, "pulse: fatal error - ");
    fprintf (stderr, s, a1, a2, a3, a4, a5, a6);
    fprintf (stderr, "\n");
    exit (1);
Appendix I - Listing of Back Propagation Learning Program

#include <stdio.h>
#include <math.h>
#include <signal.h>

/*
 * Back propagation
 *
 * Use: backprop w outunit [inunits] < binfile
 */

#define logistic(net) ((1.0 / (1.0 + exp(- (net)))))
#define advance(argc,argv) (--argc,argv++)

#define LRATE .05
#define MOMENTUM .9
#define MAXNEURONS 69

struct link {
    struct neuron *np;      /* Target neuron */
    double weight;          /* Link weight */
    double delay;           /* Link delay (not used, but copied) */
    double odelayw;         /* Old delta w for momentum */
};

typedef struct link link;

struct neuron {
    int flags;            /* Various flags */
    double sum;           /* Accumulating activation input or delta */
    int nsum;             /* Number of items added to the activation or delta so far */
    int nouts;            /* Number of output links */
    int nins;             /* Number of input links */
    double activation;    /* Activation, when known */
    double delta;         /* Delta, when known */
    struct link outs[MAXNEURONS]; /* Output links */
    struct link ins[MAXNEURONS]; /* Input links */
} neurons[MAXNEURONS];

typedef struct neuron neuron;

#define ACTVALID 001/* Activation valid */
#define DELVALID 002/* Delta valid */
#define INUNIT 004/* This is an input unit */
#define OUTUNIT 010/* This is an output unit */
#define EXIST 020/* Existance flag */

int N;            /* Actual number of neurons */
char *wfile;

int inunits[32], outunits, ninunits;
char ibuf[BUFSIZE];

int npasses = 1;

main(argc, argv)
    int argc;
    char **argv;
{
    setbuf (stdin, ibuf);

    advance (argc, argv);

    if (argc < 3)
    {
        fprintf (stderr, "Use: backprop [-npasses] w outunit [inunits ...] < b
        exit (0);
    }

    if (argv[0][0] == '-')
    {
        sscanf (argv[0] + 1, "%d", &npasses);
        advance (argc, argv);
    }

    getN (wfile = * argv);
    readmat (wfile);
    advance (argc, argv);

    neurons[outunit = atoi (*argv)].nouts = 1;
    neurons[outunit].flags |= OUTUNIT;
    advance (argc, argv);

    for (ninunits = 0; argc; ninunits++, advance(argc, argv))
    {
        neurons[inunits[ninunits] = atoi (*argv)].nins = 1;
        neurons[inunits[ninunits]].flags |= INUNIT;
    }

    while (npasses--)
    {
        rewind (stdin); /* Only works for file input */
        backprop();
        writeout (wfile);
    }
}

forward (np)    /* Propagate activations forward */
    neuron *np;
{
    neuron *target;
    link *lp;

    np->activation = logistic(np->sum);
    np->flags |= ACTVALID;
if (! (np->flags & OUTUNIT))
{
    for (lp = np->outs; lp < np->outs + N; lp++)
    {
        if (target = lp->np)
        {
            target->sum += lp->weight * np->activation;

            if (++(target->nsum) == target->nins)
                forward (target);
        }
    }
}
}

backward (np)
neuron *np;
{
    neuron *source;
    link *lp;

    np->delta = np->sum * np->activation * (1. - np->activation);
    np->flags |= DELVALID;

    if (! (np->flags & INUNIT))
    {
        for (lp = np->ins; lp < np->ins + N; lp++)
        {
            if (source = lp->np)
            {
                source->sum += lp->weight * np->delta;

                if (++(source->nsum) == source->nouts)
                    backward (source);
            }
        }
    }
}

backprop()
{
    char s[128];
    int output;
    int input;
    neuron *np;
    int i;
    int count;

    for (count = 0; gets(s); count++)
    {
        if ((count % 100) == 0)
            fprintf (stderr, "%d\r", count);
sscanf (s, "%o %d", &input, &output);

for (np = neurons; np < neurons + N; np++)
{
    np->sum = 0;
    np->nsum = 0;
    np->flags &= ~ACTVALID;
}

for (i = 0; i < ninunits; i++)
{
    np = neurons + inunits[i];
    np->sum = (input & (1 << i)) ? 10. : -10.;
    np->nsum = 1;
    forward (np);
}

for (np = neurons; np < neurons + N; np++)
{
    np->sum = 0;
    np->nsum = 0;
    np->flags &= ~DELVALID;
}

np = neurons + outunit;

np->sum = output - np->activation;
np->nsum = 1;
backward (np);

adjustweights();
}

fprintf (stderr, "%d\n", count);
}

adjustweights()
{
    neuron *np, *target;
    link *lp;
    double delw;

    for (np = neurons; np < neurons + N; np++)
    {
        if (np->flags & ACTVALID)
        {
            for (lp = np->outs; lp < np->outs + N; lp++)
                if (((target = lp->np) & (target->flags & DELVALID))
                    {
                        delw = LRATE * target->delta * np->activation +
                                MOMENTUM * lp->odelw;
                        lp->weight += delw;
                        lp->odelw = delw;
                    }
        }
    }
}
target->ins[np - neurons].weight = lp->weight;

char *
white(cp)
register char *cp;
{
    register char c;

    while(c = *cp)
    {
        switch(c)
        {
        case ' ':
        case '\t':
        case '\n':
        case '\r':
            return (cp);
        default:
            cp++;
            break;
        }
    }
    return (cp);
}

char *
black(cp)
register char *cp;
{
    register char c;

    while(c = *cp)
    {
        switch(c)
        {
        case ' ':
        case '\t':
        case '\n':
        case '\r':
            cp++;
            break;
        default:
            return (cp);
        }
    }
    return (cp);
}

getN (filename)
char *filename;
{
    FILE *stream;
    char string[512];
    char *cp;

    if (stream = fopen (filename, "r"))
    {
        if (fgets (string, sizeof string, stream))
            for (N=0, cp = string; *(cp = black(cp)); N++)
                cp = white (cp);
        else
            N = 0;

        fclose (stream);
    }
    else
        N = 0;
}

readmat(filename)
    char *filename;
{
    FILE *stream;
    int i, j;
    char string[512];
    char *cp;

    if (! (stream = fopen(filename, "r")))
    {
        fprintf(stderr, "Can’t open %s\n", filename);
        exit(0);
    }

    for (i=0; fgets(string, sizeof string, stream); i++)
    {
        for(j=0, cp = string; *(cp = black(cp)); j++)
        {
            ascanf(cp, "%lf", &neurons[i].outs[j].weight);

            if (((neurons[j].ins[i].weight = neurons[i].outs[j].weight) != 0.)
            {
                neurons[i].outs[j].np = neurons + j;
                neurons[i].flags |= EXISTS;
                neurons[i].nouts++;

                neurons[j].ins[i].np = neurons + i;
                neurons[j].flags |= EXISTS;
                neurons[j].nins++;
            }

            cp = white (cp);
        }
        if (j != N)
{  
    fprintf(stderr, "Inconsistent dimension \n");  
    exit(0);  
}
}

if (i != N)  
{
    fprintf(stderr, "%d,%d - rectangular\n", i, N);  
    exit(0);  
}

fclose(stream);
}

writeout(wfile)  
char *wfile;
{
    FILE *wstream;
    int i, j;

    if (! (wstream = fopen(wfile, "w")))  
    {  
        fprintf(stderr, "can't open w");  
        exit(0);  
    }

    for (i=0; i<N; i++)  
    {
        if (neurons[i].flags & EXISTS)  
        {
            for (j=0; j<N; j++)  
            {  
                fprintf(wstream, "%g\t", neurons[i].outs[j].weight);
            }
        }
        else  
        {
            for (j=0; j<N; j++)  
            {  
                fprintf(wstream, "%g\t", 0.);
            }
        }
        fprintf(wstream, "\n");
    }
  
  fclose(wstream);
}