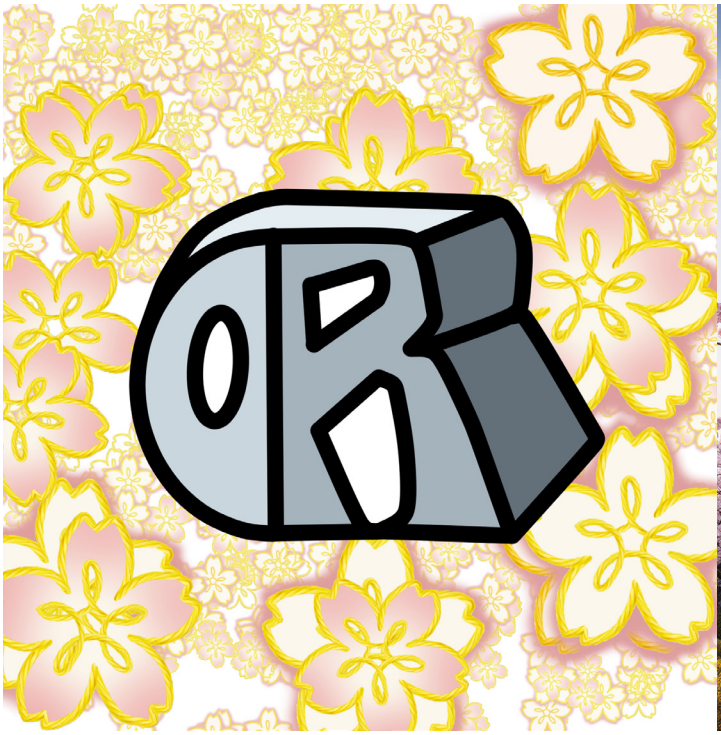


Open Research Institute Inner Circle Newsletter February 2025

The Who What When Where Why

Open Research Institute is a non-profit dedicated to open source digital radio work. We do both technical and regulatory work. Our designs are intended for both space and terrestrial deployment. We're all volunteer.
You can get involved by visiting https://openresearch.institute/getting-started

Membership is free. All work is published to the general public at no cost. Our work can be reviewed and designs downloaded at https://github.com/OpenResearchInstitute

We equally value ethical behavior and over-the-air demonstrations of innovative and relevant open source solutions. We offer remotely accessible lab benches for microwave band radio hardware and software development. We host meetups and events at least once a week. Members come from around the world.

OPEN SOURCE HARDWARE AND SOFTWARE NEWSLETTER SIGNUP
FREE AND 100% VOLUNTEER DRIVEN

Want more Inner Circle Newsletters?
Use the QR code at left or go to
http://eepurl.com/h_hYzL
and sign up.

ORI's AmbaSat Payload Moves Forward

AmbaSat writes: "For any space satellite company, the journey to orbit is filled with challenges. Designing and testing hardware, developing software, and securing a launch provider are all major hurdles. However, one of the most rigorous and complex aspects of this journey is meeting the regulatory requirements for an Orbital Operations Licence.

After an immense amount of work, we're thrilled to announce that we have officially submitted our Orbital Operations Licence application to the UK Civil Aviation Authority (CAA).

This marks a significant milestone for AmbaSat. It's been a long road to get here, filled with dedication, innovation, and perseverance. Of course, submission is just the beginning—there's still plenty of work ahead. But this moment brings us one step closer to orbit, and we couldn't be more excited.

Now, we enter the final phase of preparation—think of it as reaching the last level of a game, where the ultimate challenge awaits: launching into space."

ORI is an early supporter of AmbaSat and has used the board in several ways. One of the modifications to the board can be found at https://github.com/ambasat/AmbaSat-1/pull/10

We made a number of these modified boards and the performance was improved.

We will send our original AmbaSat board in for integration into the payload and a UK launch after the license application process successfully concludes.

We have leveraged AmbaSat to great success as an educational platform and look forward to it reaching space.

Earth-Venus-Earth: Bouncing Signals Off Our Planetary Neighbor

What is the Earth-Venus-Earth Project?

The Earth-Venus-Earth (EVE) project is an ambitious radio communications experiment from the citizen science and amateur radio communities working to bounce radio signals off the planet Venus and receive them back on Earth.

Why Venus?

Venus is our closest planetary neighbor and presents a fascinating target for radio experiments. When Venus is at its closest approach to Earth, we have the best chance of using it as a reflective surface for certain radio frequencies. By bouncing signals off Venus, we can:

1. Push the boundaries of amateur radio capabilities
2. Gather more data about Venus's atmospheric and surface communications reflectivity properties
3. Develop techniques that could be useful for future deep space amateur communications
4. Achieve something extraordinary with open-source technology and collaborative effort

Our Contribution: Can the Connection be Made?

One of the most crucial aspects of any radio communications project is understanding whether a signal can successfully travel from point A to point B (and in our case, back to point A again). This is where our work on the EVE link budget comes in.

What's a Link Budget?

A link budget is essentially an accounting of all the factors that strengthen or weaken a radio signal as it travels. Imagine tracking a water droplet's journey through a series of pipes, pumps, and filters – you need to know

where water is added or lost to understand if enough will reach the destination. Or imagine tracking your income and expenses over the course of a month.

Similarly, with radio signals, we need to account for:

- How much power we can transmit from Earth
- How much signal is lost traveling through space
- How much signal scatters when it hits Venus
- How much returns toward Earth
- How sensitive our receiving equipment must be to detect the returning signal

Our Achievements

Volunteers at Open Research Institute have developed a comprehensive link budget model for the EVE project that demonstrates feasibility, required specifications, timing windows, and open collaboration.

1. Feasibility: We've shown that with access to some of the largest amateur dishes, such as at DSES, Dwingeloo, Stockert, and potentially other sites, that amateur radio equipment and techniques can potentially bounce signals off Venus. Probably the most significant contribution is showing mathematically how difficult EVE is and how nearly all weak signal modes that currently exist in amateur radio will not close the link.

2. Required Specifications: We've determined the minimum requirements for transmitters, antennas, and receivers needed to make this connection. We've contributed several new sections and quantified techniques for this attempt.

3. Timing Windows: We've mapped the optimal time periods when Venus is positioned correctly relative to Earth for successful signal reflection, and identified challenging characteristics in the radio environment.

4. Open Collaboration: All our calculations, models, and results are openly shared, allowing amateur radio operators worldwide to participate in, critique, correct, or reproduce our work.

What Makes This Special

What sets our work apart is that we're approaching this as an open research initiative. The link budget work we've completed serves as a roadmap for anyone interested in participating in or learning more about Earth-Venus-Earth communications.

Next Steps

With our link budget analysis nearing completion we're now considering putting together an Earth-Mars-Earth link budget. We are also studying the possibility of whether distributed receivers can be used for Earth-Moon-Earth to achieve anything of note.

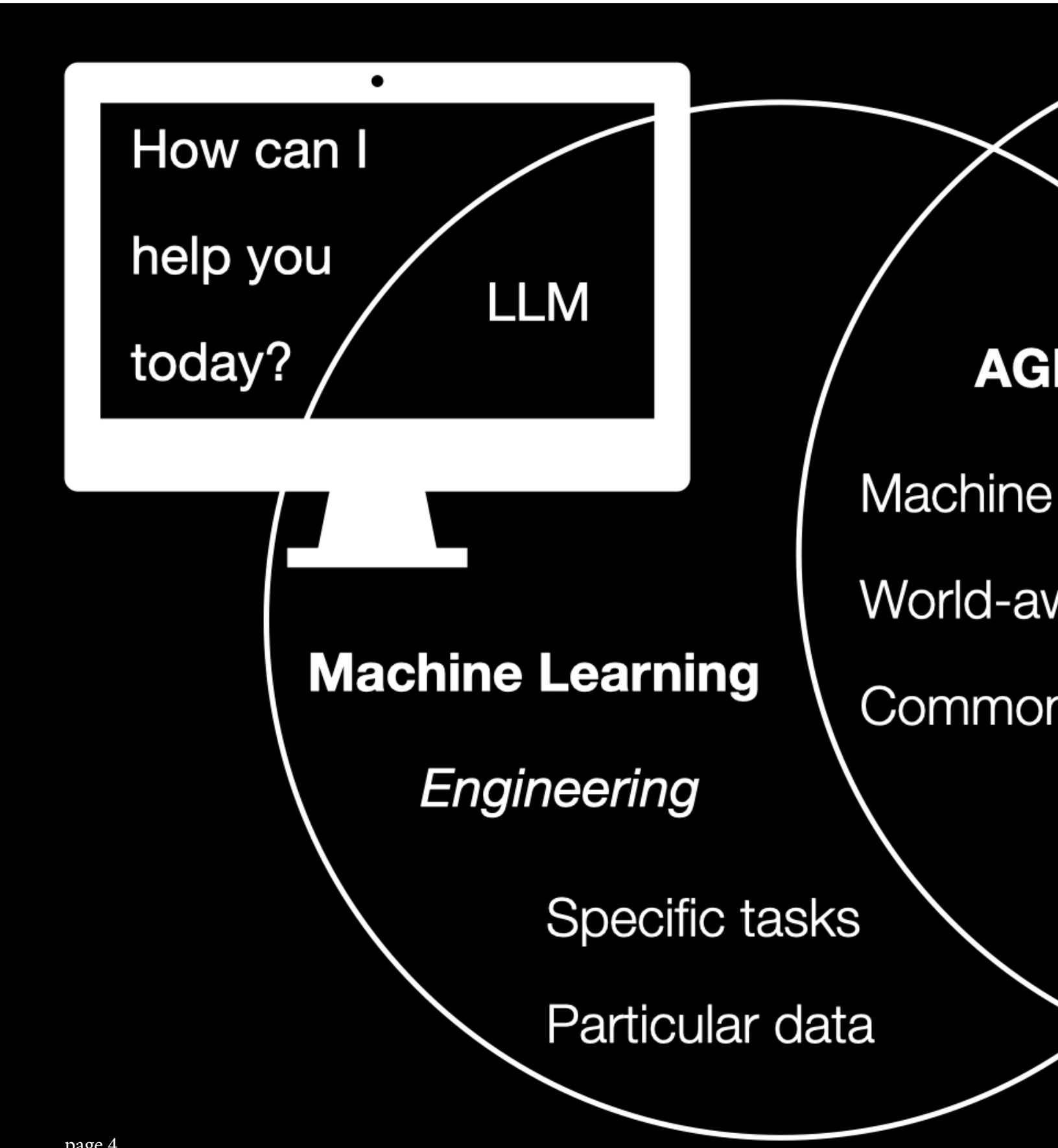
Join the Exploration

If you're interested in radio, space, or innovative open research, we welcome your participation. The beauty of open research is that everyone brings unique perspectives and skills that strengthen the entire project. Visit the Open Research Institute website at https://openresearch.institute/getting-started to learn how you can be part of this exciting journey to bounce signals off our planetary neighbor.

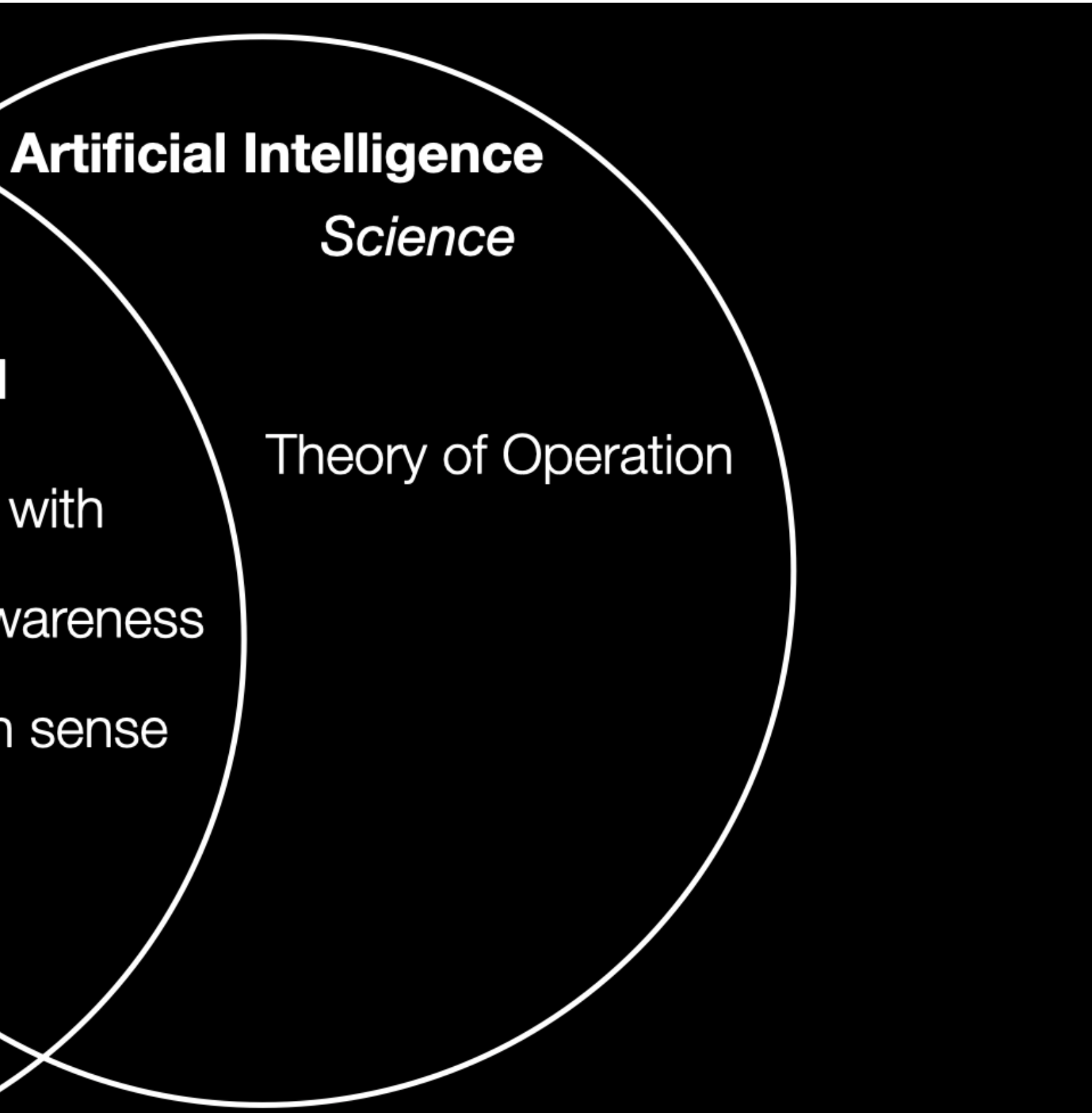
This project represents the collaborative effort of many contributors at Open Research Institute, advancing our understanding of both radio technology and our solar system through open development and shared discovery.

Current link budget can be found at https://github.com/OpenResearchInstitute/documents/blob/master/Engineering/Link_Budget/Link_Budget_Modeling.ipynb

AI/ML at ORI - A


How can I help you today?
LLM
Machine Learning
Engineering
Specific tasks
Particular data
AGI
Machine
World-av
Common

Activity Reports



Successful Collaboration with IEEE Leads to Practical AI/ML Design Work at ORI

Michelle Thompson W5NYV and Matthew Wishek NB0X organized a meetup about the Role of Artificial Intelligence and Machine Learning (AI/ML) in Register Transfer Logic Design (RTL) Generation. The Open Source Digital Radio IEEE Local Group and the San Diego Chapter of the IEEE Information Theory Society co-hosted the online event. The meeting was held on 28 January 2025 and a recording can be found at https://youtu.be/8xDxeUxWTCc

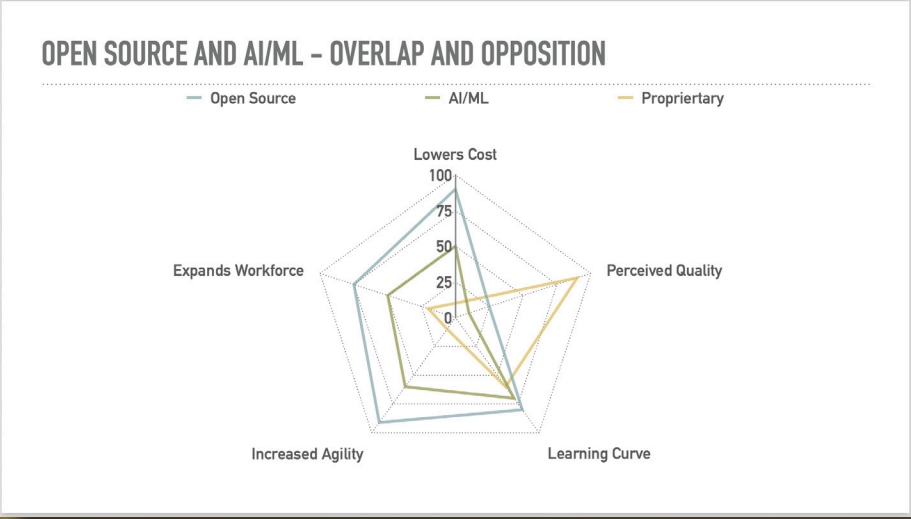
AI/ML in RTL Design Generation

In the meeting, Michelle and Matthew presented about the potential of artificial intelligence and machine learning in Electronic Design Automation (EDA) frameworks. The central question of the meetup was to try and answer where the AI/ML can help the RTL design generation process. They identified core concepts from an open-source perspective and discussed the importance of reducing schedule, technical, and cost risks in the design process. Matthew highlighted the iterative nature of the design process and the need for early identification of design deficiencies and incorrect assumptions. He also suggested the use of AI agents to guide the high-level synthesis process, assist in design space exploration, and improve the performance of place and route.

Michelle discussed the potential of open-source EDA and shared specific recommendations from a European white paper " Roadmap and Recommendations for Open Source EDA in Europe", which can be found at https://fossi-foundation.org/resources/eu-roadmap.

Michelle highlighted the challenges of limited access to EDA software and the semiconductor workforce shortage in the US. These are problems also shared by the tech industry in Europe. She mentioned the positive impact of open-source initiatives like the RISC-V processor and the Google Skywater Process Design Kit (PDK). The technical recommendations from the paper included focusing more on analog and mixed-signal designs, interoperability and verification, and system-on-chip integration. She also emphasized the importance of proper licensing, funding, sustainability, and industry training for both open-source and AI/ML projects. Another recommendation from Free and Open Source Silicon (FOSSi) Foundation was for more conferences, workshops, and events to better distribute the vast amount of innovative and exciting information and developments in this field.

Michelle discussed the use of large language models and automated writing of HDL for design IP. She shared her experience with ORI's Remote Labs Matlab HDL Coder toolbox, which can produce high-quality, human-readable HDL code but requires a lengthy process and is not suitable for complex monolithic designs. She also mentioned the use of AI and ML in deep learning hardware models and their potential to inform the design process. Michelle explained a survey of experts in the field (see summary below), which showed a bias towards proprietary tools being perceived as high-quality, and lesser expecations concerning quality from



both Open Source and AI/ML tools. Matthew then introduced a set of relevant papers he had found in his literature search, focusing on areas such as RTL generation, hardware verification, testbench generation, and formal verification. His call for action was to try to put some of the potential of the work published in these papers to inform and improve the design process into actual Open Source practice.

Matthew talked about about the role of of AI/ML in analog design, signal processing, and network planning. He suggested that AI tools can help with analog design, potentially enabling non-specialists to experiment and learn. He discussed the use of AI/ML in detecting convolutional codes, channel estimation, and radio map generation for network planning. There is significant overlap with information theory in these applications, particularly in tracking entropy levels in codes and signals. Michelle mentioned that the Information Theory and Applications Workshop 9-14 February 2025 would probably have a lot of AI/ML content. She then described successful technology demonstrations by DARPA in 2019 showing improved spectrum efficiency using AI/ML models. This proof of concept from DARPA sparked a great deal of interest from academia, industry, and regulators.

Daniel, Michelle, and Matthew discussed the pros and cons of using proprietary versus open-source Electronic Design Automation (EDA) tools. Michelle described the challenges of open-source projects, such as the lack of funding, limited market size, and difficulty in maintaining quality. Matthew shared his experience with open-source simulators, noting their limitations in terms of feature set parity and language coverage. Daniel suggested that open-source solutions might take longer to develop but could benefit from advancements in AI tools. Michelle and Matthew agreed that the EDA market is small, which contributes to the high cost of proprietary tools. The consensus from the participants was that while there are good open-source projects addressing AI/ML in RTL design generation, many of them may not yet be ready for complex designs.

It was resolved to pick one or more of the many tools and projects mentioned in the presentation, and give it a try. Reports about the first-hand experiences would then be shared in future IEEE meeting collaborations, to put the recommendations from FOSSi and the rubrics from Matthew into better context.

AI RTTY and DeepReceiver

After the EDA meeting, the search for applications of interest to the ORI community sharpened into further focus.

Matthew found a survey paper "Deep Learning in Wireless Communication Receiver: A Study" by Doha and Abdelhadi. One of the references

was a paper about a project called DeepReceiver, titled "DeepReceiver: A Deep Learning-Based Intelligent Receiver for Wireless Communication in the Physical Layer". These papers helped us figure out the next step for the Artificial Intelligent Radio Teletype (RTTY) receiver project.

Up until this point, the machine learning model identified one stand-alone individually transmitted RTTY letter at a time. The DeepReceiver paper provided the inspiration and the education to update the model to something much more realistic.

In order to be useful, an AI RTTY receiver has to be able to translate whole "sentences" of RTTY "speech", and not just individual characters given one at a time.

How is was ORI's artificially intelligent RTTY receiver constructed?

Read on for a draft of RTTY Receiver by Artificial Intelligence.

RTTY Receiver by Artifical Intelligence

Michelle Thompson W5NYV, Open Research Institute, Inc.

20 February 2025

Table of Contents

Introduction

Radio Teletype, or RTTY, is a binary frequency shift keying (FSK) signal that encodes keyboard data for communications. This is a popular amateur radio mode with a long history. An introduction to RTTY can be read at https://en.wikipedia.org/wiki/Radioteletype.

Our end goal of this work is to create a machine learning (ML) model that can demodulate and decode an RTTY message received over the air. ML is one way that artificial intelligence (AI) is implemented. This article captures the current state of this work. We want to show how AI/ML can be used in amateur radio, and explain the process, with an accessible and familiar application.

This work was inspired by a Mathworks AI Workflow seminar held in Carlsbad, CA, in June 2024. At this event, a demonstrated audio example was worked out for attendees. The example can be found at https://www.mathworks.com/help/deeplearning/ug/acoustics-based-machine-fault-recognition.html

Confirm Basic RTTY Functionality in MATLAB

First, we set up, exercised, and verified the basic RTTY functions used in this project. These RTTY functions in MATLAB are from work by Ben Bales.

Bales, Benjamin, "Low Power RTTY and PSK31 Decoder for Ham Radio Applications" (2011). *Chancellor's Honors Program Projects.* https://trace.tennessee.edu/utk_chanhonoproj/1350

```
audiodevreset
```

```
msg = 'HELLO WORLD';
filename = "airtty.txt";
snr = 3;
Fs = 176/(1/45.45);
```

First we reset our audio device to avoid an intermittent error (**Device Error: Internal Device error**) encountered on some versions of MacOS.

The message that we want to send is contained in `msg`, a string.

We write our RTTY values out to a file. We set that filename with the variable `filename`.

We set a signal to noise ratio for our generated RTTY signal with the variable `snr`.

We pick a sample rate that gives an integer number of samples per training window of one symbol period. Our target sample rate is 8000 Hz, which matches the sample rate specified in the function file `gen_rtty_for_mplab_sim.m`. We want our sample rate to match the one used to generate the RTTY signals.

```
gen_rtty_for_mplab_sim(msg, filename, snr)
```

We call this function. It generates the RTTY signal from the message we created at the snr we gave it, and it writes the resulting audio output to a file.

```
play_rtty(msg, snr)
```

We play the signal from the computer speakers to confirm we have created a RTTY signal.

Data Set Generation

We next generate a large data set of RTTY signals. We are using RTTY signal data synthesized by MATLAB to train, validate, and test our machine learning model insteads of signals recorded off the air. We train on all the individual letters (and figures). We will train over a range of randomly generated signal to noise ratios. The data set is a dictionary of signals paired with the RTTY characters used to generate the signal, with a random level of noise added to the signals. Our motivation is to train a model that will decode individual letters correctly. We include a noisy radio environment in the training. The question that we want to answer is "Given a received signal, what was the most likely letter sent?"

Create Data Set of Synthesized RTTY Signals with AWGN

```
data_set_size = 128*100;
```

We decide how large of a data set we are going to work with. This set will be divided into training, validation, and testing subsets. We set the variable `data_set_size` to the total number of signals used for training, validation, and testing.

```
message_length =10;
message = cell(data_set_size,1);
```

We decide how long of a training message we are going to work with. Look at the size of our HELLO WORLD test message. It looks like it has eleven letters. We could and should have random length messages to better match up with real world received signals, but our first experiment was single characters selected at random from the RTTY letters only. After that was working, we added in the figures alphabet. Our current experiment is messages that are ten characters, all randomly selected from the RTTY letters and figures alphabet.

Now that we have determined the message length, we set up a cell array that is `data_set_size` rows by 1 column in size. This array will hold our our randomly generated messages, one message per row. Next we fill up this cell array with randomly generated messages.

`randsample()` arguments are: ([randomly selected number from a particular set], sample length of 1, 'true' indicates we sample with replacement)

`char(my_string1)` converts a numeric array to a character array. This conversion uses unicode.

The character set for RTTY is [0 5 7 10 13 32 33:35 38 40 41 44:59 63 65:90]. `char` function returns the unicode character corresponding to the number randomly selected from the set.

But, it's not quite that simple. RTTY uses Baudot encoding, which means that each character is five bits long. Five bits lets us encode thirty-two different things. But, we have more than thirty-two letters and figures that we want to use. The solution in RTTY is to re-use the five bit encodings. We will have to double up. Letters in one set, and figures in another. But, how do we tell if we sent a letter versus a figure?

There are two additional symbols. LTRS (11111) and FIGS (11011). If we receive a LTRS, then starting with the next character, and continuing until we get a FIGS, we extract the five binary digits that were sent, and then use the **letters** table to look it up. If we receive a FIGS, then starting with the next character, and continuing until we get a LTRS, we extract the five binary digits that were sent, and then use the **figures** table to look it up. We don't print out the LTRS or FIGS. They are control characters that tell us what we really meant when we said "Catch that bat!" Do we chase a small flying mammal, or do we grab a piece of sports equipment that slipped out of a batter's hands? For example, "K" and "(" both use the same binary representation (01111) over the air. LTRS and FIGS tells us which character was intended to be sent by the transmitting station.

```
decimal = [0 5 7 10 13 32 33:35 38 40 41 44:59 63 65:90]';
character = char(decimal);
ascii_table = table(decimal, character)
```

ascii_table = 55×2 table

	decimal	character
1	0	
2	5	
3	7	
4	10	
5	13	
6	32	

	decimal	character
7	33	!
8	34	"
9	35	#
10	38	&
11	40	(
12	41	)
13	44	,
14	45	-
15	46	.
16	47	/
17	48	0
18	49	1
19	50	2
20	51	3
21	52	4
22	53	5
23	54	6
24	55	7
25	56	8
26	57	9
27	58	:
28	59	;
29	63	?
30	65	A
31	66	B
32	67	C
33	68	D
34	69	E
35	70	F
36	71	G
37	72	H
38	73	I
39	74	J

	decimal	character
40	75	K
41	76	L
42	77	M
43	78	N
44	79	O
45	80	P
46	81	Q
47	82	R
48	83	S
49	84	T
50	85	U
51	86	V
52	87	W
53	88	X
54	89	Y
55	90	Z

Note we have a problem. On my system, ENQ and BEL are both a square character representation when run through the char() function. Null, carriage return, space, and line feed all seem to come up as empty. Some character encodings still need some work to properly represent.

We are wanting to use string representations to train our model, so we do need to make sure that we represent our messages correctly.

```
number    character
0         NUL (Null) - defined
5         ENQ (Enquiry) - substituted with $
7         BEL (Bell, Alert) - substituted with single quote
10        LF (Line feed) - undefined
13        CR (Carriage return) - undefined
32        SP (Space) - undefined
```

Let's talk about ENQ and BEL. Enquiry asks the receiving station to send back some sort of identification. For example, a call sign, or tactical ID, or company name. Bell or Alert causes an alert or bell sound to ring at the

receiving station. In RTTY charts, BEL can also be a single quote. And, ENQ can also be $. So, we make this adjustment to get printable characters for BEL and ENQ. The numbers for $ and single quote are 36 and 39. For now, we drop Null, Line Feed, Carriage Return, and Space and substitute in dollar sign and single quote for Enquiry and Bell.

```
%decimal = [0 10 13 32 33:35 36 38:40 41 44:59 63 65:90]';
decimal = [33:35 36 38:40 41 44:59 63 65:90]';
character = char(decimal);
ascii_table = table(decimal, character)
```

ascii_table = 51×2 table

	decimal	character
1	33	!
2	34	"
3	35	#
4	36	$
5	38	&
6	39	'
7	40	(
8	41	)
9	44	,
10	45	-
11	46	.
12	47	/
13	48	0
14	49	1
15	50	2
16	51	3
17	52	4
18	53	5
19	54	6
20	55	7
21	56	8
22	57	9
23	58	:
24	59	;
25	63	?
26	65	A

	decimal	character
27	66	B
28	67	C
29	68	D
30	69	E
31	70	F
32	71	G
33	72	H
34	73	I
35	74	J
36	75	K
37	76	L
38	77	M
39	78	N
40	79	O
41	80	P
42	81	Q
43	82	R
44	83	S
45	84	T
46	85	U
47	86	V
48	87	W
49	88	X
50	89	Y
51	90	Z

Create Messages

```
my_string5 = categorical([1,message_length]);

for q = 1:data_set_size
%for q = 1:2
    for n = 1:message_length
        my_string5(n) = char(randsample(decimal,1,true));
    end
    message{q,1} = my_string5;
end
```

We create `data_set_size` random strings and put them in the `message` cell array. Each row has a message that we will "send" as a RTTY signal. `my_string5` is a `categorical` data type and has a size of 1 row by `message_length` columns. We loop data_set_size times. Each time we loop, we start a new loop, `message_length` long. In this experiment, we loop 10 times because we want messages that are 10 characters long. For this loop, each time we go through it we select one of the values in `decimal` at random, we turn that number into its unicode character representation, and put it in to the categorical array called `my_string5`. This is how we build up a message, character by randomly selected character. When we are done with each individual 10-character message, we drop out of the inner loop, copy `my_string5` over to our `message` array, and move on to the next message, until we've filled in `data_set_size` messages.

Create Signals

```
signals = cell(1,data_set_size);
```

We set up a cell array called signals. This will hold the signals of the time series of binary fsk values calculated by the `gen_rtty` function and modified by `awgn`. Each column is a separate calculated signal that is generated by a corresponding message in `message`. The number of columns is controlled by `set_data_size`. Each signal is a time series representing the voltage values of the RTTY signal.

```
rng('shuffle');
```

We initialize a random number generator based on the current time with 'shuffle' as the argument to rng, resulting in a different sequence of random numbers after each call.

```
snr = (randi([0,60],1,data_set_size));
```

We are going to have an array of values called snr. Our randomly generated signal to noise ratios will be held in here.

The command `randi([imin imax],m,n)` generates a random integer between imin and imax, with the result in an m by n matrix. We generate the random numbers for signal to noise ratio in advance and then index into the array to access them, so that we do not repeat the random number generator evey time we run the loop. This may save time for large data sets.

```
for i = 1:data_set_size
    signals{1,i} = awgn(gen_rtty(Fs, 1275, 1455, 1 / 45.45, 2,
char(message{i}(:))'), snr(i));
    signals{1,i} = signals{1,i}/max(signals{1,i});
end
```

We have a loop. Each time we go through the loop we calculate a set of values that represent a RTTY signal sending a particular message at a particular signal to noise ratio. At the end of the loop we have a full set of synthesized signals. The synthesized signals are stored in the columns of a cell array called `signals`. We have also normalized the data by dividing each element of the signal by the maximum value of that signal. This

gets you the desired SNR as long as there are enough bits of resolution per sample. Since it reduces the signal amplitide to make room for the biggest signal plus noise sample, it also introduces quantization noise in the signal.

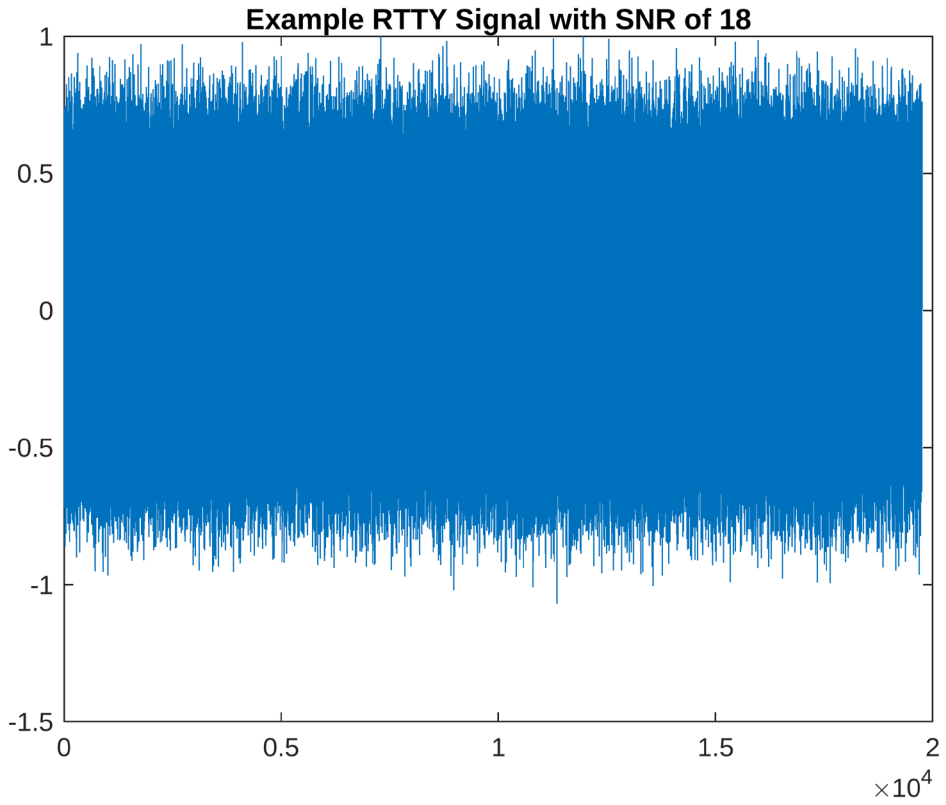
The command `gen_rtty(Fs, Fmark, Fspace, Tsymbol, stopBits, message)` is from Ben Bales' work.

Visualization of an RTTY Signal

```
indx = 1:size(signals{1,4},2);
indy = signals{1,4};
```

A picture is worth a thousand words, so let's see what a signal looks like. We create the inputs to a plot. The x index (indx) is just that, an index of how many values are in the signal. We obtain this by selecting the fourth row of the signals cell array. This element is a row vector, so we want to indicate to the size function that we are looking for the number of columns in the element. The number of columns is the second dimension, which is the reason for the number 2 as the second argument to the size function. The y index (indy) is the row vector itself of the fourth row of the signals cell array. We needed the size to construct our x axis index array. Our y axis is the actual values of the signal.

```
figure(1)
plot(indx,indy)
str = ['Example RTTY Signal with SNR of ', num2str(snr(4))];
title(str);
```



We tell MATLAB that we want a figure, and that it is figure 1. The x axis is the number of elements in the fourth row of the signals array. The y axis is the values in the fourth row of the signals array. We plot the RTTY signal, which is what it looks like sending the fourth message in our data set over the air.

Optionally Play all RTTY Signals

```
false
```

```
ans = logical
   0
```

```
if ans == 1
    for i = 1:data_set_size
        audiodevreset;
        soundsc(signals{1,i}, Fs);
        pause(2);
    end
end
```

We know now many signals are in our data set. Let's listen to all of them! If the check box is active, then we loop through all our signals and play them. We reset the audio devices before we play each signal, to avoid some common problems on some platforms. We include a pause of 2 seconds because that helped prevent problems on the computer used to write the script. We do this to confirm that we are playing a collection of RTTY signals at different signal to noise ratios, each with a different short message. For large data sets, playing all of the audio files would take a very long time, so this off by default.

Configure Sets of Audio Features

```
windowLength = Fs*(1/45.45);
overlapLength = floor(windowLength/2);
f1 = 2125;
f2 = 2295;

afe = audioFeatureExtractor(SampleRate=Fs, ...
    Window=hamming(windowLength,"periodic"),...
    OverlapLength=overlapLength,...
    pitch=true, ...
    spectralCentroid=true, ...
    harmonicRatio=true);
```

We define the audio features that we want to extract from each signal with `audioFeatureExtractor`. We store this configuration in the variable `afe`. For this feature set, we focus on features that let us extract our FSK tones quickly and efficiently. The features we want to extract are marked `true` if they are calculated and `false` if they are skipped.

Window length is how much of the signal we are looking at at one time. `windowLength` is calcluated to be about one RTTY symbol length long. It's parameterized so if we change the sample rate or baud rate then our code will still work. We set the `overlapLength` to be about half of the `windowLength`. This gives us the ability to catch transitions without spending an excessively long time doing math. We have the two baseband amateur radio RTTY tones as references. Right now, we are not using them in the code, but there are some audio features that take frequencies as inputs. We have them assigned "just in case" we need them at some point.

Extract Audio Features from our Signals

```
audio_features = cell(data_set_size,1);
tic
for e = 1:data_set_size
    audio_features{e,1} = extract(afe, signals{1,e}')';
end
disp("Feature extraction took " + toc + " seconds.");
```

```
Feature extraction took 73.5535 seconds.
```

We set up the `audio_features` cell array. It has `data_set_size` rows and one column. We loop through the size of our data set and extract the audio features from each of our signals. Our current experiment produces three sets of audio features per signal. We store the three rows of results in the corresponding row of the `audio_features` cell array. The `extract` function expects the signal data to be a column vector, so we transpose it here. We also transpose the result of the extraction. We do this so that our audio features cell array "looks right" to the functions in the deep learning toolbox that we are going to use. The data hasn't changed, but it does matter to certain functions if it's arranged in columns versus rows.

Calculate Indices for Training, Validation, and Test Sets

```
population_splits = [0.8,0.1,0.1]
```

```
population_splits = 1×3
    0.8000    0.1000    0.1000
```

We create an array that defines the proportions of the training set, the validation set, and the test set. The three elements must add to 1. A common set of proportions is 0.8, 0.1, and 0.1, which stands for 80% training data, 10% validation data, and 10% test data. Training data is data that we are going to use to train our machine learning model. Validation data is used to monitor and adjust the learning process along the way. Test data is used after the machine learning model is trained in order to test its performance.

```
training_indices = [1:floor(population_splits(1)*data_set_size)];
validation_indices = training_indices(end) +
[1:floor(population_splits(2)*data_set_size)];
test_indices = validation_indices(end) +
[1:floor(population_splits(3)*data_set_size)];
```

Using the number of rows in our record, which is equal to the number of signals synthesized, we calculate the indices of the training set, the validation set, and the test set. We use these arrays of index values to fetch rows from the corresponding category. In other words, the elements of `test_indices` will correspond to the rows in the record we have designated for testing.

The first set of indices (`training_indices`) is calculated by multiplying the proportion of the first category times the total number of records in the table, and creating an array of 1 to that number. The second set of indices is calculated by multipling the proportion for the second category times the total number of records in the table, creating an array of 1 to that number, and then adding an offset equal to the ending index from the first category. This shifts the second category array up the correct number of positions so that it falls immediately after the first category. The third set of indices is calculated by multiplying the proportion for the third category times the total number of records in the table, creating an array from 1 to that number, and then adding an offset equal to the ending index of the second category. This creates three disjoint sets of indices following the proportions in `population_splits`. In order to ensure we get integer results from multiplying by a float, we use the `floor` function.

Fill Training, Validation, and Test Sets with Input and Output Data

```
indx = training_indices(1):training_indices(end);
training_set = audio_features(indx,1);
training_labels = message(indx,1);

indx = validation_indices(1):validation_indices(end);
validation_set = audio_features(indx,1);
validation_labels = message(indx,1);

indx = test_indices(1):test_indices(end);
test_set = audio_features(indx,1);
test_labels = message(indx,1);
```

We used logical indexing to create the index array `indx` for our three cases. We use this array to section off three matched sets of audio features in `audio_features` and their corresponding messages in `message`. We now have three tables of training, validation, and test sets of audio features (an analysis of our RTTY signals) with matching labels (the messages that we used to create the RTTY signals).

How Is a Neural Net Trained?

We have matching pairs of input and output data, separated into three sets. We want to use a neural net to predict outputs based on new inputs. We need to teach the neural network how to interpret new input, based on known input-output pairs. How is this done?

Imagine teaching a new soccer referee how to run games. The process would look like this:

Obtain a Library of Previously Played Games (Training Data):

You give the new referee a collection of called games. Infractions are matched up with calls. Each called game is like one training example. The more diverse the calls the new ref sees, the better they'll handle different situations when they have to call a live game.

Learning Process (Training):

The ref starts by making decisions about each of the pre-called games. After each decision, you tell them how close they were to the "right" answer. They adjust their approach based on how far off they were. This happens thousands of times with different scenarios

Fine-tuning (Optimization):

If the ref is changing their style too drastically after each feedback session (we've all seen refs that are inconsistent and we don't want that), then you might tell them to make smaller adjustments. If they're not adapting quickly enough, you might encourage bigger changes. You keep adjusting these learning factors until they're improving at a good pace and "getting the hang of it".

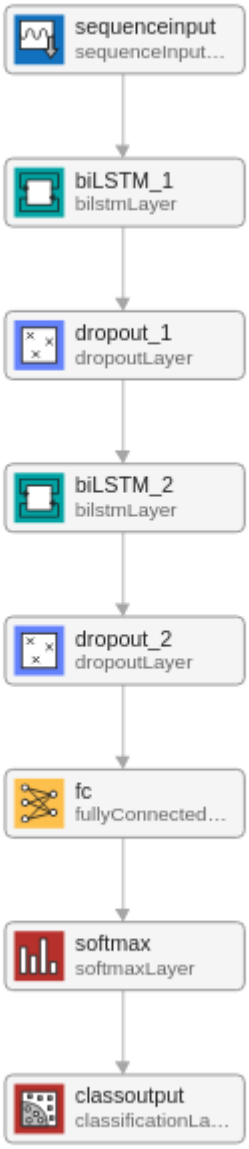
Testing (Validation):

Finally, you give them completely new games they've never seen. This tells you if they've truly learned how to ref a game, or if they've just memorized the practice games. If they just memorized the practice games, then the only infractions they call correctly are ones they've already seen.

The key idea is: Just like a ref learns from many examples and gradually adjusts their approach based on feedback, a neural network learns by seeing many examples and adjusting its internal connections based on how well it's doing.

Define the Machine Learning Model

We define our machine learning model in an array called `layers`. Models are also called networks. A diagram of our model is below.



The first layer is `sequenceInputLayer`. It takes the number of extracted features as an argument. We get the number of features from the audio feature extractor structured variable `afe`.

The next layer is a `bilstmLayer`, which stands for bidirectional long short-term memory (BiLSTM) layer for recurrent neural network (RNN). This is a layer that learns long-term dependencies, in both directions, between time steps of time series or sequence data. Since we have control characters that determine which set of characters we use for translation, there is a long-term memory type of effect from the control character. A BiLSTM layer, set up to translate one sequence to another sequence, seems like a good place to start. The parameter given to this layer is the number of hidden nodes. The number of nodes in a layer is the number of artificial neurons. Hidden nodes encode a characterization of the data from the previous time steps.

The next layer is a dropout layer called `dropoutLayer`. The parameter for this layer is a dropout probability. The main reason for using dropout is to help prevent overfitting. This is where a model performs really well on training data but poorly on new data. We want our model to be able to do some generalization, and this layer helps achieve that goal. We randomly pick the dropout percentage of neurons and erase them.

We have a second bilstmLayer with more neurons, and then another dropout layer.

Next is `fullyConnectedLayer` or a fully connected layer. The parameter is set to the size of our alphabet. This translates from the number of neurons in the previous layer to the number of different things that we need to identify. In other words, the size of the set of items we are trying to distinguish. At the time of this writing, we were going from 128 neural nodes to 51 characters. However, we need to set the number of outputs to 51+1. MATLAB's trainNetwork is expecting our output layer to account for our 51 characters and a potential "background" or "null" character class, which brings the total to 52. This is particularly common when working with image classification tasks and categorical data (like our data).

Next is the `softmaxLayer`. This layer converts the raw outputs from the fully connected layer into probabilities. It does this by taking the exponential of each of the outputs, and then dividing by the sum of all these exponentials. In other words, we normalize our outputs in this layer.

The final layer is a classification layer, or `classificationLayer`. It looks at all the probabilities for all the potential outputs, and selects the one with the highest probability.

```
dropProb = 0.2;
layers = [ ...
    sequenceInputLayer(afe.FeatureVectorLength)
    bilstmLayer(64) %,OutputMode="sequence")
    dropoutLayer(dropProb)
    bilstmLayer(128) %,OutputMode="sequence")
    dropoutLayer(dropProb)
    fullyConnectedLayer(length(decimal)+1) %set to size of our alphabet + 1
    softmaxLayer
    classificationLayer];
```

Define the Hyperparameters

```
options = trainingOptions("adam", ...
    MaxEpochs=100, ...
    MiniBatchSize=32, ...
    GradientThreshold=1, ...
    Plots="training-progress", ...
    ExecutionEnvironment="parallel", ...
    Verbose=false);
```

Imagine you're playing the game Dungeons and Dragons and you are creating a new character. Before you even start rolling stats or choosing spells, you have to make high-level decisions like:

What level will you start at? How many players will be in the party? What's the maximum number of spells you can learn?

Hyperparameters in machine learning are similar to these high level decisions. We set up a structured variable called `options` by using the `trainingOptions` function. The parameters are key value pairs given as arguments to this function. They are set up before we do any training, just like rolling up a character before a role-playing game session. If we choose a character very well-suited to the gaming environment, then we are likely to do better than if we rolled up a weak or ill-suited character. The same thing goes for our hyperparameters.

Some examples from our MATLAB machine learning toolbox include:

Learning rate: This is like choosing how cautiously your character advances. A high learning rate means taking big, bold steps (risking overshooting), while a low rate means moving carefully but potentially taking longer to reach the goal.

Batch size: Similar to deciding how many encounters your party faces before taking a long rest. Larger batches give more stable but slower training, while smaller batches are faster but potentially more erratic.

Number of epochs: Like deciding how many training sessions your character needs before facing the Big Bad Evil Guy (BBEG). More epochs mean more training time but potentially better performance.

Network architecture (layer sizes, number of layers): This is like designing your character's class structure - how many levels in each class, what abilities you'll have access to.

Execution Environment: this lets us do things like let us use our multiple CPU system, which is sort of like being able to play multiple characters at the same time. When we set it to "parallel", we use eight CPUs instead of just one.

Mini Batch Size: this is how many signals we train on at once. We don't want to train on one signal at a time. We probably don't want to train on all our signals at once. Common mini-batch sizes are powers of 2 (32, 64, 128, 256) because they optimize well with GPU memory. The right size depends on several factors, such as the available memory, the dataset size, model complexity, and how stable we want our training to be. In our gaming analogy, this would be like how many monsters we try to take on at the same time. One at a time makes for a very long dungeon crawl. Taking on the entire orc army would result in a brutally short end to our character's career. There is an ideal range of batch size, and getting as close to that as possible increases the quality of our training.

Gradient Threshold: this is a safety mechanism for training. Too much "gain" in the training from step to step can cause bad results, like undefined results, extremely large results, and oscillations. If we get unstable training, then we lower the threshold. If training is too slow, and things aren't converging, then we raise the threshold. If training is always stable and predictable, then we can try removing the threshold entirely to see if our particular model needs this safety mechanism at all. This is like deciding how much risk your character will take in a gaming session. You can rush in and start all the fights as quickly as possible, or you can hang back and investigate everything and take it very slowly.

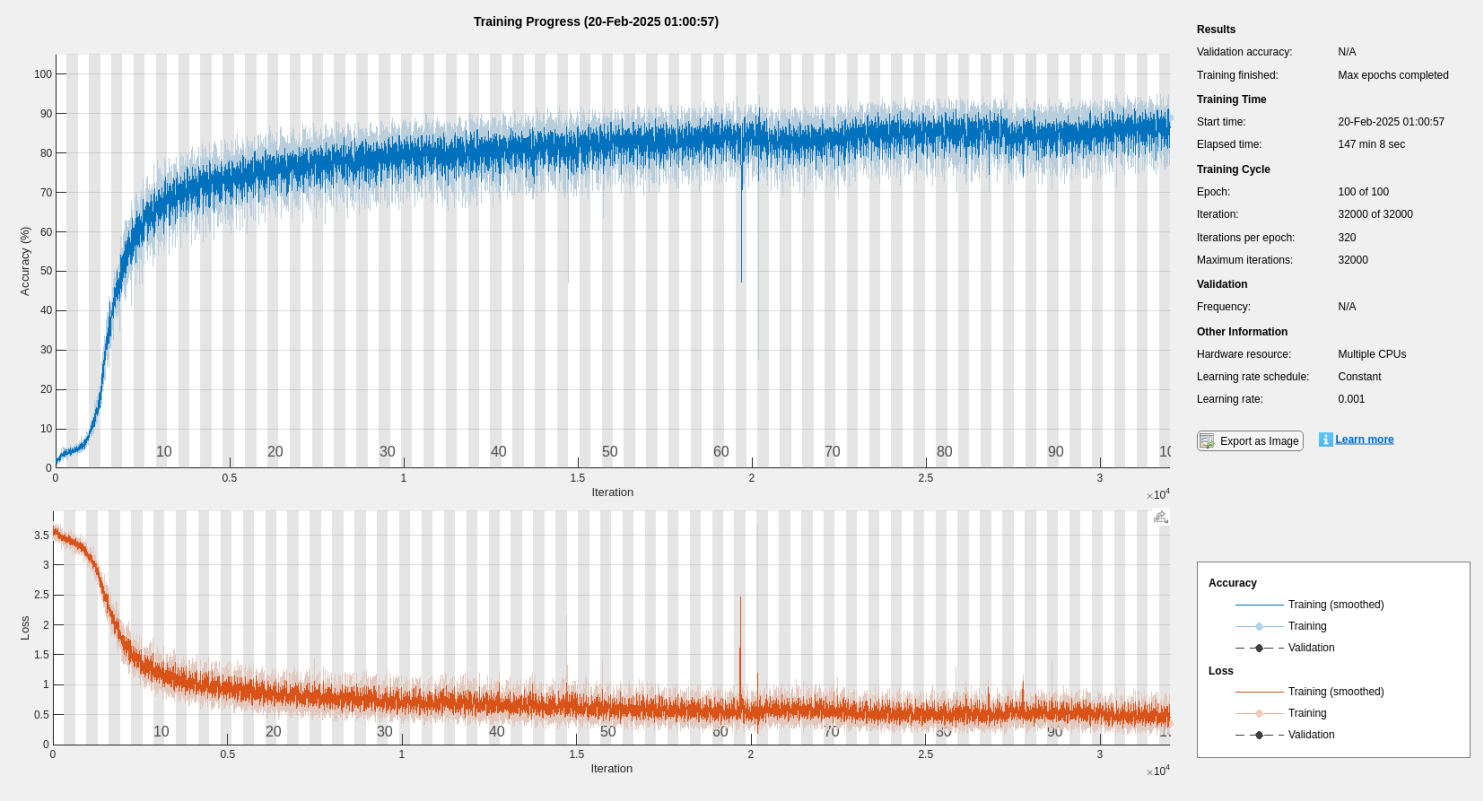
We can also do things like set up plots and make the debug verbosity high, so that we generate more errors and warnings in order to solve problems more quickly.

Train Network

For sequence-to-sequence training in MATLAB, we need to make the dimensions of `training_labels` match the time dimension of `training_set`. We have the same situation for the validation and test sets. Each feature sequence needs its corresponding label sequence of the same length in order for the math to work. What we do is expand the particular message label, the ten characters we sent, out so that there's a letter for every time step in our signal feature extraction. It's about 25 steps per character.

We then train our model on the signal features, alongside our expanded data labels, using the layers we set up, and the options. In the graph, the upper curve is model accuracy. The lower graph is the amount of loss, or the pentalty score for being wrong. We should see the accuracy climb over time and the loss go down. The loss gives insight to the confidence level of the model. For example, a model could make a prediction with just 51% certainty. Another model could make the same prediction with 99% certainty. Both models are "correct", but the loss would show up for 51% certainty as much higher than the 99% certainty. Loss gives us insight into training so that we can improve the confidence or certainty of our model.

```
% Make each label sequence match its corresponding feature sequence
numSequences = numel(training_set);
training_labels_expanded = cell(size(training_set));

for i = 1:numSequences
    % Get current sequence lengths
    [~, numTimeSteps] = size(training_set{i});
    numWords = length(training_labels{i});

    % Calculate steps per word
    stepsPerWord = floor(numTimeSteps/numWords);

    % Expand labels to match time steps
    expanded = [];
    for j = 1:numWords
        expanded = [expanded repmat(training_labels{i}(j), 1, stepsPerWord)];
    end

    % Handle any remaining time steps
    if length(expanded) < numTimeSteps
        padding = repmat(expanded(end), 1, numTimeSteps - length(expanded));
        expanded = [expanded padding];
    end

    training_labels_expanded{i} = expanded;
end

% Now train with expanded labels
trained_net = trainNetwork(training_set, training_labels_expanded, layers,
options);
```



Evaluate Network

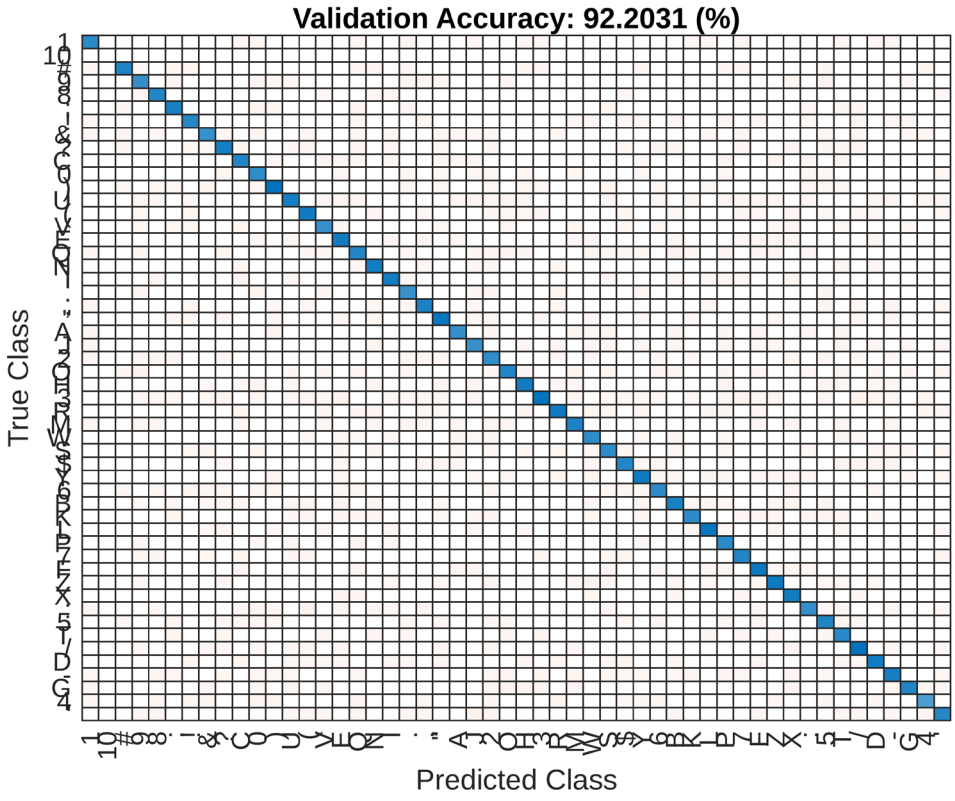
Next we view the confusion chart for the validation data, test data, and histogram of figures and letters used in this training run. A confusion chart is a two-dimensional chart with the character that was sent on the vertical axis, and the classified result from the trained model on the horizontal axis. If we were 100% correct, we'd get a solid diagonal line from upper left to bottom right. Mistakes show up off this diagonal axis of true class versus matching predicted class. We can see the one extra class we put in. It's the missing place in the diagonal on the upper left of the chart. This extra character was included in order to account for the data type we were using. It's blank because it wasn't trained and therefore isn't assigned to any of the received signals.

First we check our validation data. Then, we check our test data. If we see large difference between validation and data sets, then we might have a case of overfitting. We should see roughly the same results from validation as we do from test. Validation data is used as part of the process of training, but the test set is set aside until training is complete, and then used to see how well the model performed.

We need to convert the time-step predictions back into character sequences. Since we had to stretch out the time-step predictions to train, we have to do the reverse process so that we can get the mapping we want.

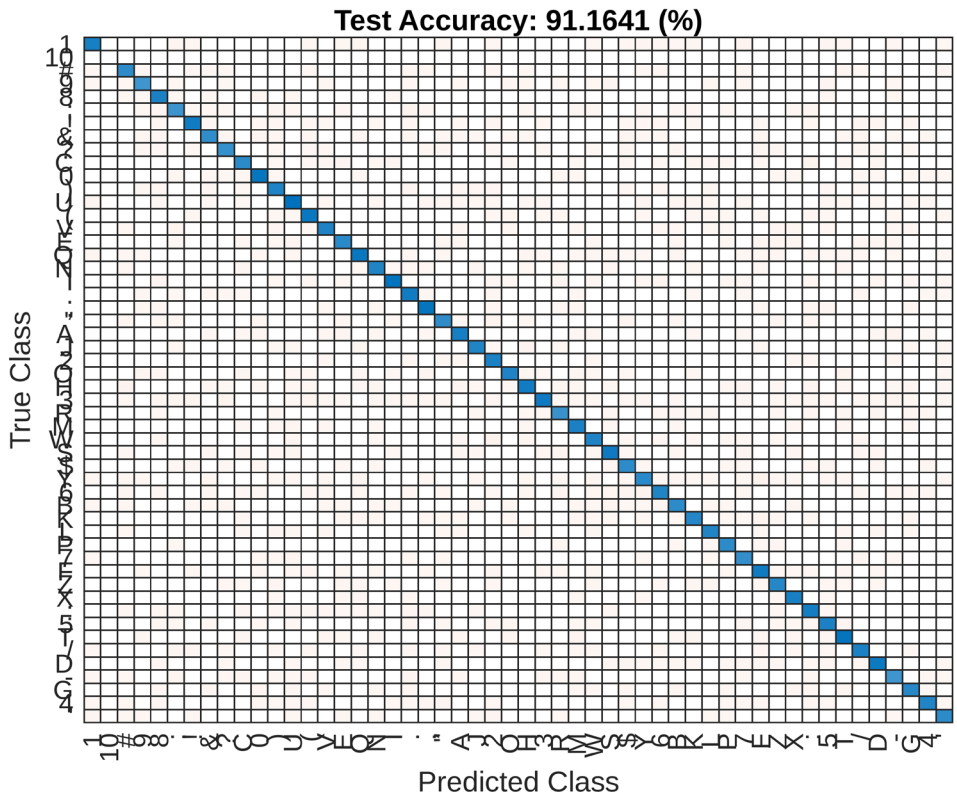
```
% Get predictions for each sequence
validationResults = classify(trained_net, validation_set);

% Convert time-step predictions back to word sequences
numSequences = numel(validation_set);
compressedPredictions = categorical([]);
originalLabels = categorical([]);
```

```
for i = 1:numSequences
    % Get predictions for this sequence
    seqPredictions = validationResults{i};
    numPreds = length(seqPredictions);

    % Each sequence should contain message_length words (10)
    wordsPerSequence = message_length;
    stepsPerWord = floor(numPreds/wordsPerSequence);

    % Compress predictions into words
    wordSeq = categorical([]);
    for j = 1:wordsPerSequence
        startIdx = (j-1)*stepsPerWord + 1;
        endIdx = min(j*stepsPerWord, numPreds);
        segment = seqPredictions(startIdx:endIdx);
        wordSeq = [wordSeq mode(segment)];  % Most common prediction in
segment
    end

    compressedPredictions = [compressedPredictions wordSeq];
    originalLabels = [originalLabels validation_labels{i}];
end

% Now create confusion matrix
confusionchart(compressedPredictions, originalLabels, ...
    'Title', "Validation Accuracy: " + ...
    mean(compressedPredictions == originalLabels)*100 + " (%)");
```




```
% Get predictions for each sequence
testResults = classify(trained_net, test_set);

% Convert time-step predictions back to word sequences
numSequences = numel(test_set);
compressedPredictions = categorical([]);
originalLabels = categorical([]);

for i = 1:numSequences
    % Get predictions for this sequence
    seqPredictions = testResults{i};
    numPreds = length(seqPredictions);

    % Each sequence should contain message_length words (10)
    wordsPerSequence = message_length;
    stepsPerWord = floor(numPreds/wordsPerSequence);

    % Compress predictions into words
    wordSeq = categorical([]);
    for j = 1:wordsPerSequence
        startIdx = (j-1)*stepsPerWord + 1;
        endIdx = min(j*stepsPerWord, numPreds);
        segment = seqPredictions(startIdx:endIdx);
        wordSeq = [wordSeq mode(segment)];  % Most common prediction in
segment
```

```
    end

    compressedPredictions = [compressedPredictions wordSeq];
    originalLabels = [originalLabels test_labels{i}];
end

% Now create confusion matrix
confusionchart(compressedPredictions, originalLabels, ...
    'Title', "Test Accuracy: " + ...
    mean(compressedPredictions == originalLabels)*100 + " (%)");
```



Conclusion

We have successfully trained a model that accepts audio features derived from RTTY audio signals, and outputs the most likely character sequence that was used to create the RTTY signal. We used synthesized RTTY signals and augmented each signal with a random amount of noise in order to produce a variety of signal to noise ratios in the input.

What are the next steps?

Add in the line feed, carriage return, space, and null.

Train on random length messages that have the statistics of RTTY messages, instead of messages that look like a cryptographic code block. This might change (possibly reduce) the size of the trained model.

Move to training on received quadrature modulation (IQ) data instead of audio output, in order to incorporate more of the radio chain and make this receiver more generally useful in modern digital radio systems. An

advantage of the audio version is that it can use any legacy or analog radio, reducing barriers to using neural networks with older radios.

Take this MATLAB model and convert it to hardware descriptive language, so that it can be deployed on a software defined radio.

Technical Resources and Acknowledgements

A listing of the dependent files and MATLAB toolboxes used in this document is printed below.

```
[flist, plist] =
matlab.codetools.requiredFilesAndProducts('airtty_project_five_letters.mlx')
```

```
flist = 1×4 cell
'/home/matt/AIRTTY/airtty_project_five_letters.mlx''/home/matt/AIRTTY/gen_rtty.m ···
plist = 1×6 struct
```

Fields	Name	Version	ProductNumber	Certain
1	'MATLAB'	'9.14'	1	1
2	'Signal Processing Toolbox'	'9.2'	8	1
3	'Deep Learning Toolbox'	'14.6'	12	1
4	'Statistics and Machine Learning Toolbox'	'12.5'	19	1
5	'Communications Toolbox'	'8.0'	36	1
6	'Audio Toolbox'	'3.4'	151	1

Thank you to Open Research Institute (https://openresearch.institute) for making this work possible by providing expert advice, community feedback, a MATLAB license with Deep Learning, and remotely accessible computer resources.

As mentioned in the introduction, the event that sparked this project was a Mathworks AI Workflow seminar held in Carlsbad, CA, in June 2024. At this event, a demonstrated audio example was worked out for attendees. The example can be found at https://www.mathworks.com/help/deeplearning/ug/acoustics-based-machine-fault-recognition.html

RTTY functions in MATLAB are from work by Ben Bales.

Bales, Benjamin, "Low Power RTTY and PSK31 Decoder for Ham Radio Applications" (2011). *Chancellor's Honors Program Projects.* https://trace.tennessee.edu/utk_chanhonoproj/1350

Finally, the large language model Claude was used to quickly search through MATLAB documentation and find recommended design patterns from a wide variety of computer programming forums and discussions on the internet. This saved a very large amount of time compared to "Googling" MATLAB documentation by hand. The code in this document benefited from using these AI/ML searches in two ways. First, by showing a common design pattern for constructing the message cell arrays, which reduced the number of lines of code. Second, by suggesting that stretching the message out to match the feature set, instead of decimating the sample set to match the number of characters in the letters. This improved the flow of the code and preserved data resolution.

ORI Celebrates National Engineers Week 16-23 February 2025

We were delighted to be part of the San Diego County Engineering Council's annual awards banquet and celebration of Engineer's Week.

We were able to present our work and give away some keepsake items at the San Diego Section IEEE booth. Thank you to IEEE for sharing their space with us so that we can spread the word about open source digital radio.

Questions and comments we fielded at the event included:

"How does it work when you give away your solutions? Isn't that stealing?"

"How do you make money doing this?"

"Does anyone use open source designs?"

"I've heard of open source software, but I've never heard of open source hardware."

It was a great opportunity to educate people about open source software, firmware, and hardware, the value it brings to technology in general and the value it brings to digital communications, including the Internet, specifically. Below, we were included with IEEE in a group photo.

“Take This Job”

Interested in Open Source software and hardware? Not sure how to get started? Here’s some places to begin at Open Research Institute. If you would like to take on one of these tasks, please write hello@openresearch.institute and let us know which one. We will onboard you onto the team and get you started.

Opulent Voice:

- Add a carrier sync lock detector in VHDL. After the receiver has successfully synchronized to the carrier, a signal needs to be presented to the application layer that indicates success. Work output is tested VHDL code.
- Bit Error Rate (BER) waterfall curves for Additive White Gaussian Noise (AWGN) channel.
- Bit Error Rate (BER) waterfall curves for Doppler shift.
- Bit Error Rate (BER) waterfall curves for other channels and impairments.
- Review Proportional-Integral Gain design document and provide feedback for improvement.
- Generate and write a pull request to include a Numerically Controlled Oscillator (NCO) design document for the repository located at https://github.com/OpenResearchInstitute/nco.
- Generate and write a pull request to include a Pseudo Random Binary Sequence (PRBS) design document for the repository located at https://github.com/OpenResearchInstitute/prbs.
- Generate and write a pull request to include a Minimum Shift Keying (MSK) Demodulator design document for the repository located at https://github.com/OpenResearchInstitute/msk_demodulator
- Generate and write a pull request to include a Minimum Shift Keying (MSK) Modulator design document for the repository located at https://github.com/OpenResearchInstitute/msk_modulator
- Evaluate loop stability with unscrambled data sequences of zeros or ones.
- Determine and implement Eb/NO/SNR/EVM measurement. Work product is tested VHDL code.
- Review implementation of Tx I/Q outputs to support mirror image cancellation at RF.

Haifuraiya:

- HTML5 radio interface requirements, specifications, and prototype. This is the user interface for the satellite downlink, which is DVB-S2/X and contains all of the uplink Opulent Voice channel data. Using HTML5 allows any device with a browser and enough processor to provide a useful user interface. What should that interface look like? What functions should be prioritized and provided? A paper and/or slide presentation would be the work product of this project.
- Default digital downlink requirements and specifications. This specifies what is transmitted on the downlink when no user data is present. Think of this as a modern test pattern, to help operators set up their stations quickly and efficiently. The data might rotate through all the modulation and coding, transmititng a short loop of known data. This would allow a receiver to calibrate their receiver performance against the modulation and coding signal to noise ratio (SNR) slope. A paper and/or slide presentation would be the work product of this project.

YouTube

https://www.youtube.com/@OpenResearchInstituteInc


RFBitBanger
HF Digital QRP
HF Digital QRP
HF Digital QRP
HF Digital QRP
HF Digital QRP
HF Digital QRP
HF Digital QRP
HF Digital QRP
Get your kit today
while supplies last!
https://www.ebay.com/usr/openresearchinstitute
$175
Shipping
Calculated
Separately

The Inner Circle
Sphere of Activity

February 18, 2025 - San Diego County Engineering Council Annual Awards Banquet. ORI was part of the IEEE Table display in the organizational fair held on site before dinner. ORI was represented by two members.

Schedule is clear until our conference season starts in August with DEFCON 33.

Until then, we'll be working hard in Remote Labs and publishing our work.

Thank you to all who support our work! We certainly couldn’t do it without you.

Anshul Makkar, Director ORI
Frank Brickle, Director ORI (SK)
Keith Wheeler, Secretary ORI
Steve Conklin, CFO ORI
Michelle Thompson, CEO ORI
Matthew Wishek, Director ORI