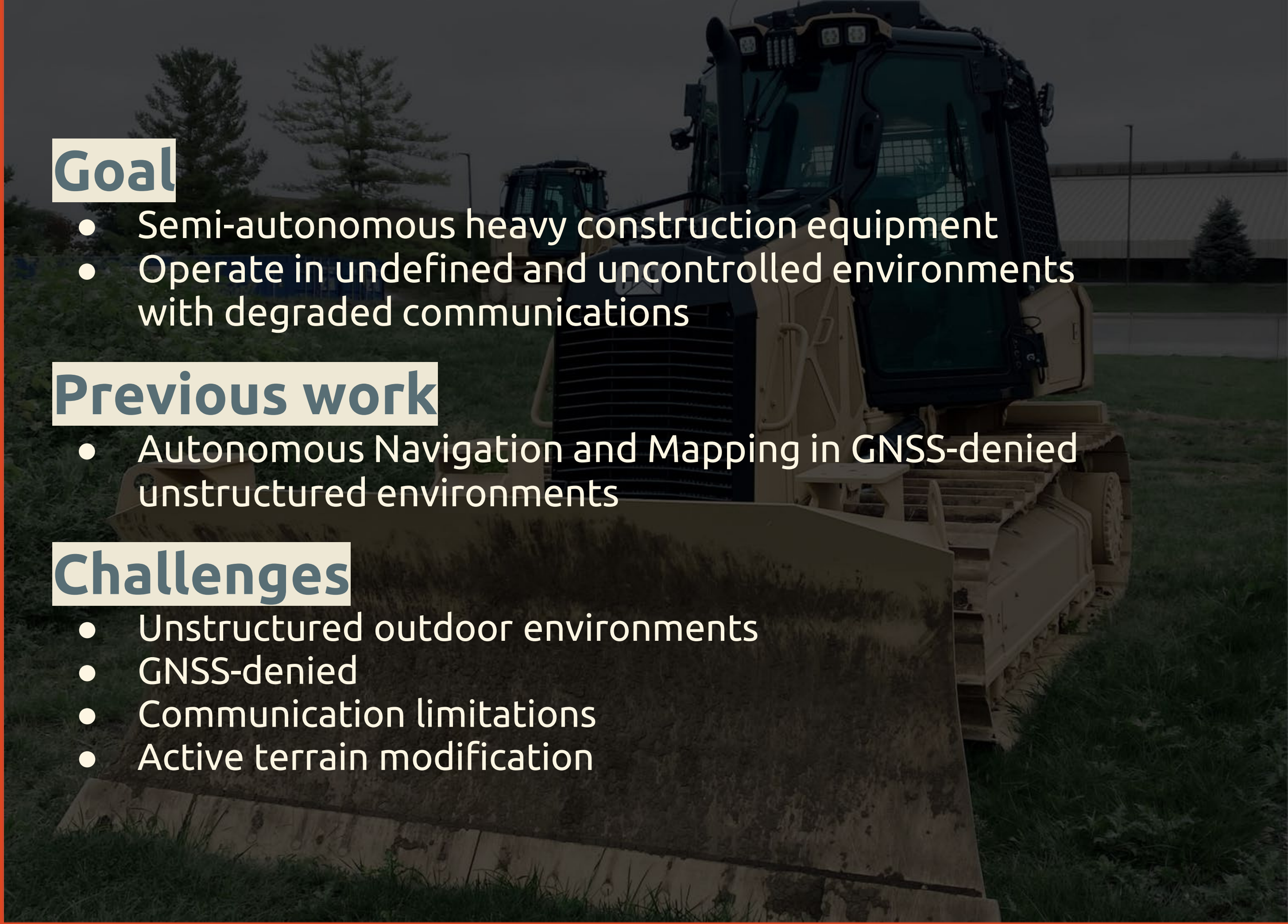
# ros2_j1939

J1939 CAN Device Support in ROS2

*Arturo Saucedo*
*Isaac Blankenau*

# Robotics for Engineer Operations



## Goal

- Semi-autonomous heavy construction equipment
- Operate in undefined and uncontrolled environments with degraded communications

## Previous work

- Autonomous Navigation and Mapping in GNSS-denied unstructured environments

## Challenges

- Unstructured outdoor environments
- GNSS-denied
- Communication limitations
- Active terrain modification

# Robotics for Engineer Operations

# *Vision*

*Problem*

Solution

Why it matters

## Bringing Autonomy to Heavy Equipment

- Existing platforms weren't designed with robotics in mind.
- Modern ECUs manage built-in sensors and by-wire controls.
- Key feedback sensors for autonomy (e.g., hydraulic pressure, piston travel) are often missing.
- The ROS2 ecosystem lacks off-the-shelf drivers for these specialized sensors.
- Integrating new sensors requires accounting for rugged environmental conditions and wiring challenges.

# *Vision*

Problem

***Solution***

Why it matters

## J1939 Bridge for ROS2

**A communication interface between ROS2 and the *J1939 CANBus protocol*, enabling robots to interact with commercial vehicle systems.**

# *Vision*

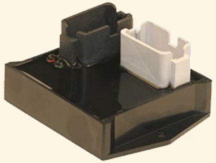
Problem

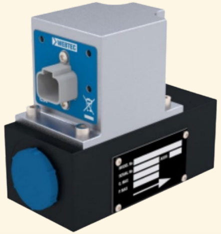
Solution

***Why it matters***

## Benefits

- **Platform Monitoring:** Access built-in platform sensors and controls for monitoring and control without extensive modifications.
- **Seamless Integration**: Use J1939-compatible sensors without the need for custom drivers.
- **Rugged and Reliable**: Components are ready for outdoor, off-road robots requiring durability.





# CAN Bus

## The Controller Area Network (CAN)

- **Complex Comms:** In the 80s, the auto industry faced challenges with increasingly *complicated intra-vehicle communications*.
- **Lacking Protocols**: Existing protocols not really suitable for use in passenger vehicles.
- **Solution:** *CAN* was developed, featuring differential signaling, priority-based arbitration, and decentralized structure.

Today, CAN is the **backbone of communication** between the many ECUs (Electronic Control Units) in modern vehicles.

*Mercedes-Benz was the first auto manufacturer to use the CAN bus in a passenger vehicle*

BOSCH

*Bosch engineers were instrumental in developing CAN*

# CAN Bus

## CAN for Robotics

- **Simplified Wiring**: Reduces cabling complexity and cost by allowing devices to share a bus.
- **Real-Time, Collision-Free Communication**: Priority-based arbitration, ensuring reliable and timely transmission.
- **By-Wire Control**: Many modern vehicles use *CAN for by-wire systems...*👀
- **Rugged Sensors**: Access to a wide range of sensors suitable for use in extreme conditions (temperature, vibration, moisture).

# J1939

## What is J1939?

- Developed by the Society of *Automotive* Engineers (**SAE**).
- **Standardized Message Set**: Includes common messages (e.g., joystick data, engine RPM, temperature).
- **Proprietary Addresses**: Allows OEMs to define their own messages within a reserved range.
- **Message Structure**:
  - *29-bit identifier* (includes PGN, priority, source, and destination addresses).
  - *8-byte data payload*.
- **DBC Files**: File format used to define CAN message structure, or “how to read the CAN messages”

# *J1939*

## Current State in ROS2

***Complete solution does not exist (small parts do)***

- **New Eagle raptor-dbw-ros2**: Integrates with New Eagle's drive by wire kit. It contains a great J1939 DBC parser.
- **Autoware Ros2_socketcan:** ROS2 wrapper around linux kernel socket can, allowing read and write raw frames on CAN bus.
- **ros2_canopen**: Feature-rich CANOpen bridge for ROS2 & ROS2 Control. Industrial-facing.

***Missing ros2_canopen style driver for J1939***

## *Generic CAN Driver*

***Overview***

Configuration

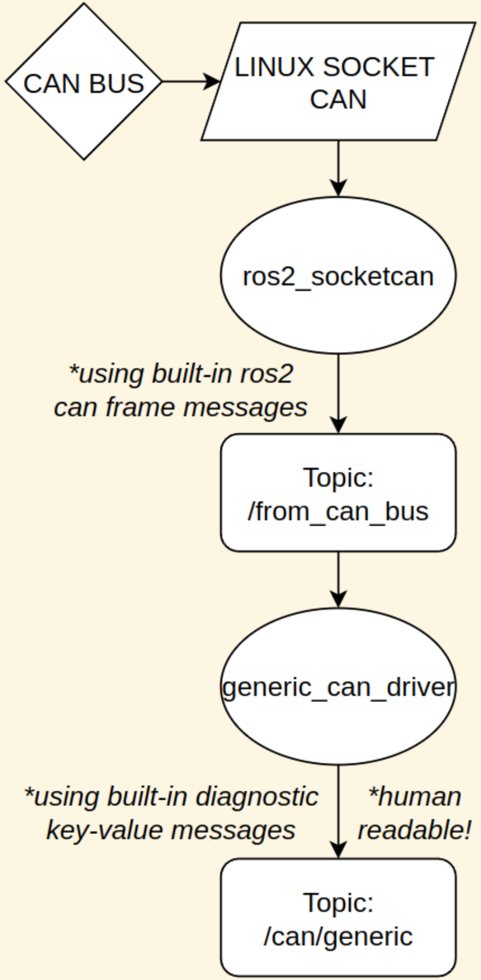
Data Handling

Sensors

ros2_control

# Hardware Agnostic Driver

- **Generate:** J1939 device publishes to the CAN line
- **Receive:** ros2_socketcan receives messages
- **Listen:** Generic CAN driver configured to filter incoming messages by device (source ID)
- **Parse:** User-provided dbc is used to parse incoming messages with can_dbc_parser
- **Publish:** Generic CAN driver publishes *human-readable* data in *auto-configured ros2 topics*.



***Generic CAN Driver***

Overview

***Configuration***

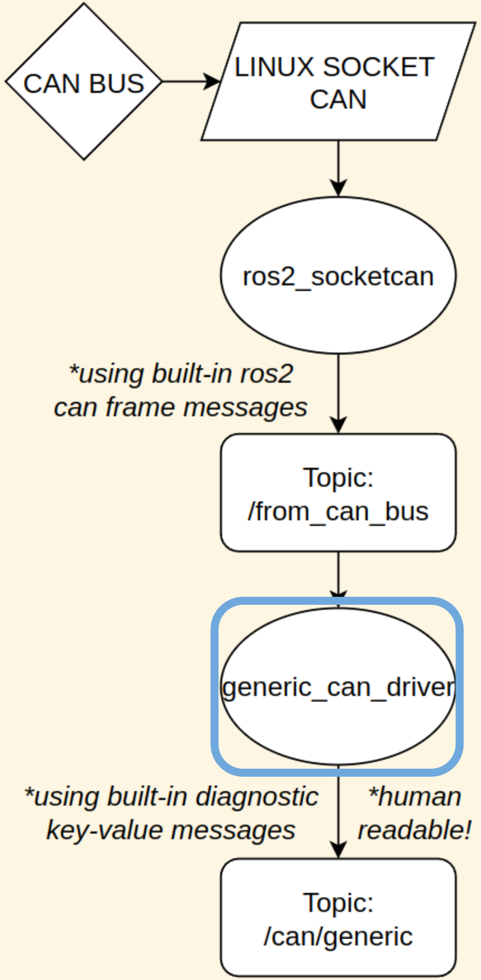
Data Handling

Sensors

ros2_control

# Node Config

- Generic CAN driver node requires basic configuration.
- **Device name:** e.g. IMU_blade
- **Device source ID:** e.g. 227
- **DBC file:** e.g. my_IMU.dbc



# *Generic CAN Driver*

Overview

Configuration

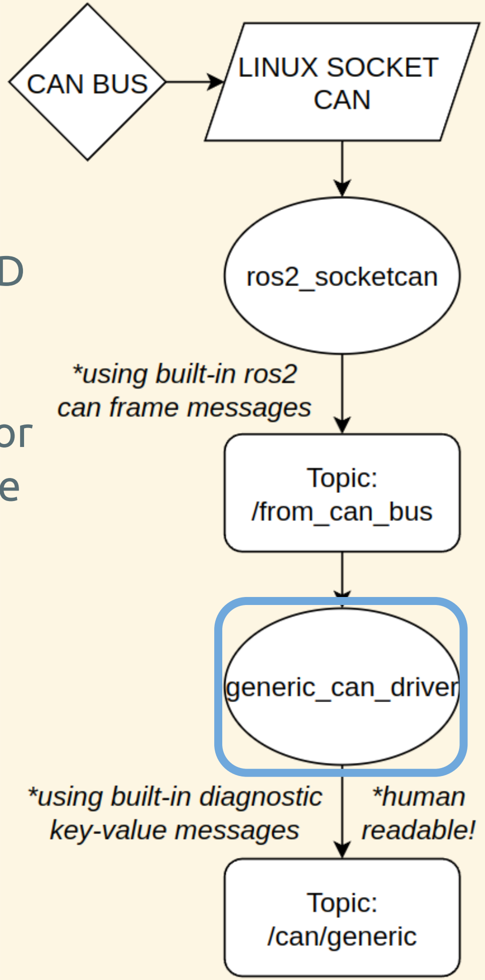
***Data Handling***

Sensors

ros2_control

## Parsing

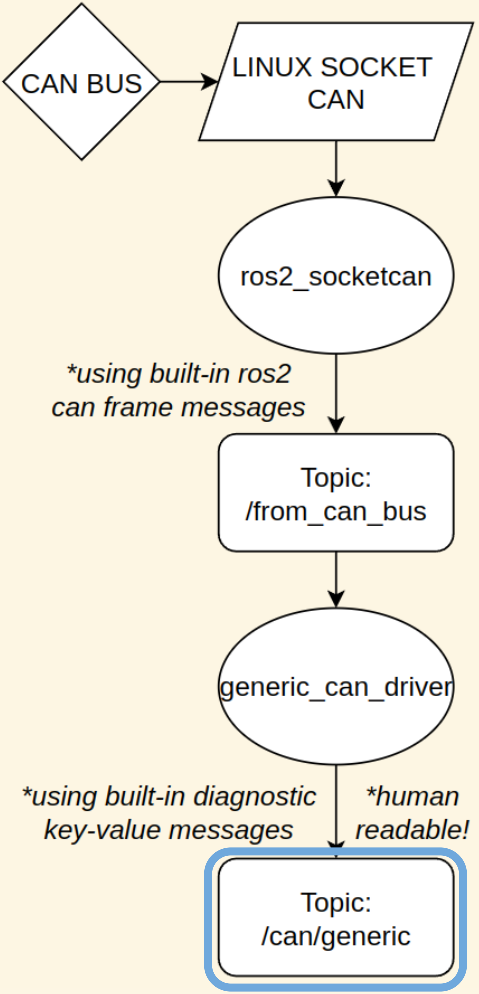
- **Filtering:** Incoming CAN frame's device ID is compared to the ID specified in the driver's config.
- **Lookup:** If they match, the driver looks for the message in the dbc and translates the result.
- **Message:** The result is turned into a modified *diagnostic-type key-value* pair message.



# Publishing

- **Publishers:** Driver *automatically configures* the necessary publishers and topics for the given DBC file.
- **Topics:** Automatically named after message names specified in the DBC:

*Topic: /device_name/message_name*



**Generic CAN Driver**

Overview

Configuration

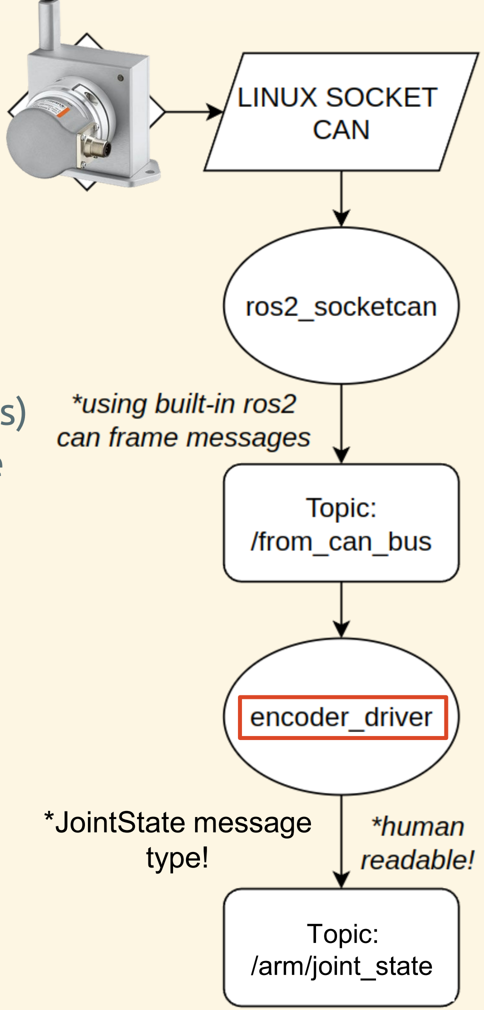
Data Handling

**Sensors**

ros2_control

# Pre-configured sensors

- **Sensor-specific:** We've also included some modified drivers for specific sensors we've encountered (IMUs, encoders, pressure sensors)
- **Types:** These drivers use *specific* ros2 message types (Imu, JointState, FluidPressure).



***Generic CAN Driver***

Overview

Configuration

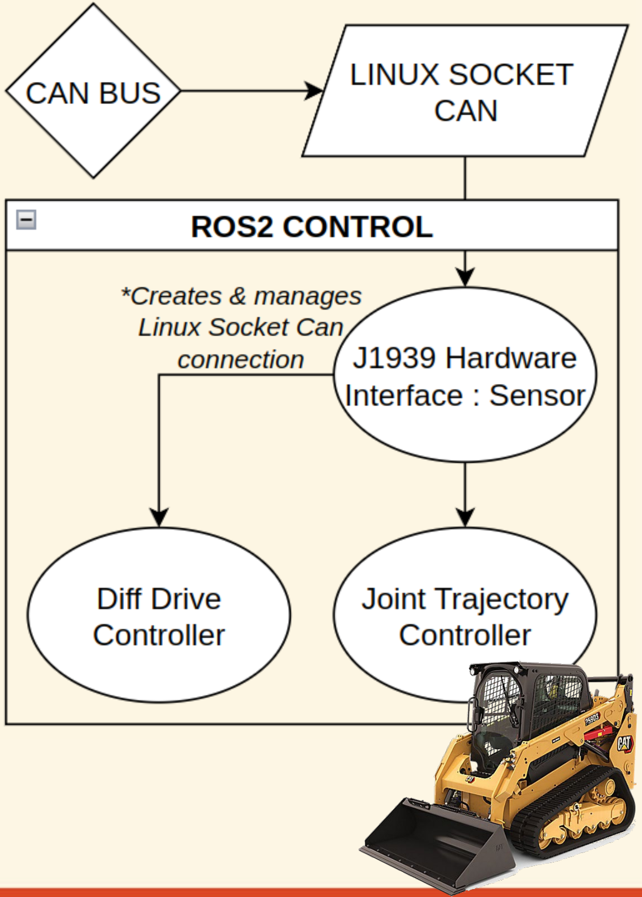
Data Handling

Sensors

***ros2_control***

# Hardware Interface

- **ros2_control:** Our plugin bridges Linux level socket can through ROS2 control.
- **Generic:** A ros2_control hardware interface for CAN J1939 sensors.
- **Bridge:** Each interface spins up its own linux socket can connection.
- **Params:** Interface name, DBC file path, source id, filter list, priority


CAN BUS
LINUX SOCKET CAN
ROS2 CONTROL
*Creates & manages Linux Socket Can connection
J1939 Hardware Interface : Sensor
Diff Drive Controller
Joint Trajectory Controller

# Control of Compact Track Loader

**By-Wire Operation**: We demonstrated by-wire control of critical vehicle functions

**Challenges:** Lack of documentation, multiple Buses, not all functionality on CANBus

**Control:** Access to control of joystick messages, polynomial fit of 5-bar linkage, blade Stabilization

**Navigation:** Nvblox and nav2

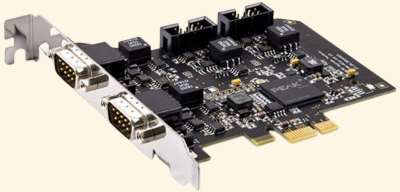
*Getting Started*

***Setup***

Connect

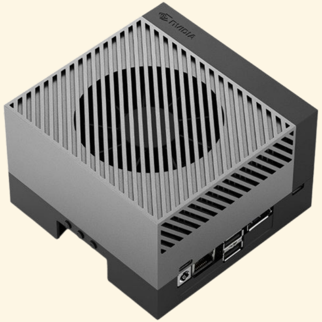
Test

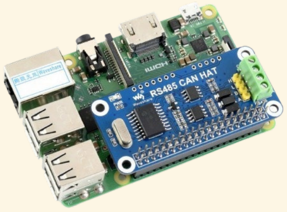
# What You’ll Need:



Now that we’ve shown a working implementation, here’s what you’ll need to get started

- **A J1939 Device:** A device that you want to interface with, typically a sensor or controller
- **DBC File:** Defines the structure of CAN messages.
- **CAN Interface:** CAN Pi-Hat, PeakCAN, USB CAN Reader, etc.
- **Can-utils:** A Computer with socket can.
- **ROS2:** An install of ros2 (devcontainer) to develop and run your nodes.





## *Getting Started*

Setup

***Connect***

Test

# Connect to a CAN network:

- **Wiring:** Signal wire should be twisted pair, with a 120 Ohm terminating resistor
- **Socket:** (Lawicel) CAN USB device run the following in terminal

```
sudo slcand -o -c -f -s5 /dev/ttyUSB0 can0
```

```
sudo ifconfig can0 up
```

- **Verify:** Candump your interface

```
candump can0
```

*Getting Started*

Setup

*Connect*

Test

# Connect to a VCAN network:

- **Socket:** Set up the interface

```
sudo ip link add dev vcan0 type vcan
```

```
sudo ip link set vcan0 up type vcan
```

- **Verify:** Play back a log file through the virtual interface

```
canplayer -i your_can_recording.log vcan0=can0 -v
```

*Getting Started*

Setup

Connect

*Test*

# Devcontainer Demos:

- **Setup:** Follow devcontainer readme for setup instructions.
- **Demos:** We include some simple demo robots for different types of joints/sensors (as well as candump log files and DBCs).
- **ros2_control.xacro**: This is where you load and configure the hardware plugins used by ros2_control for the robot. Make sure to use the correct **interface name** and **DBC** file
- **r_bot.launch.py:** Launch file for the demo, simply run:

```
ros2 launch r_bot r_bot.launch.py
```

Just like that, you can take J1939 sensor data and integrate it with a basic robot in ros2_control

# *Additional Utilities*

**Documentation, scripts, and examples in GitHub, including**

- How to setup a CAN device
- How to rename a CAN device
- How to set up our devcontainer
- How to replace/spoof a J1939 device on an existing network
- Set up a CAN socket with PeakCan & Lawicel USB on host machine
- Some pointers with ros2_control

# *Summary*

**What we've shown:** A barebones ros2 driver for any J1939 sensor, either as a standalone node or as a plugin for ros2_control. (+ centralized documentation!)

*An implementation on real hardware!*

**What you need:** A J1939 sensor, the accompanying .DBC, and ros2.

**What this means:** *Significantly* lower barrier to entry for roboticists who want to use robust commercial-off-the-shelf sensors, or interact with existing J1939 CAN networks.

# How to Contribute

**Add Sensors and Interfaces (DBCs)**: Found a DBC for your hardware? Submit a request to add it, along with any packages for hardware not yet covered.

**Don't violate copyrights** or NDAs (e.g., uploading the full SAE J1939 standard). However, manufacturer-provided or reverse-engineered DBCs are welcome.

Help us build a **Central Repository**, coalescing J1939 hardware and resources in one place to make integration easier for the community.

**Github:** https://github.com/psaucedoa

# Contact Us

***Arturo Saucedo***

Research Aerospace Engineer

arturo.saucedo@usace.army.mil

***Isaac Blankenau***

Research Mechanical Engineer

isaac.j.blankenau@usace.army.mil