U.S. Department of Transportation
Federal Aviation Administration

# FINAL PROJECT REPORT

Form Approved:
O.M.B. No. 2120-0559
9/30/2013

## PART I - PROJECT IDENTIFICATION INFORMATION

| 1. Institution and Address | 2. FAA Program | 3. FAA Award Number |
|---|---|---|
| Georgia Institute of Technology | Air Transportation COE, AJFE | 13-C-AJFE-GIT-059 |
| 270 Ferst Drive, Atlanta, GA 30332 | 4. Award Period From 6/5/2020 To 9/30/2023 | 5. Cumulative Award Amount 499,999.00 |

6. Project Title

Geospatially Driven Noise Estimation Module - Project 9

## PART II - SUMMARY OF COMPLETED PROJECT (For Public Use)

The goal of this project was to research potential open-source alternative geospatial frameworks for upgrading AEDT, and to test and experiment with different novel computational approached in order to drastically speed up the noise exposure calculations from potential large numbers of UAS being operated for package delivery or other potential UAS applications.

## PART III - TECHNICAL INFORMATION (For Program Management Uses)

| 1. **ITEM** (Check appropriate blocks) | NONE | ATTACHED | PREVIOUSLY FURNISHED | TO BE FURNISHED SEPARATELY TO PROGRAM | |
|---|---|---|---|---|---|
| | | | | Check ( X ) | Approx. Date |
| a. Abstracts of Theses | ✘ | | | | |
| b. Publication Citations | ✘ | | | | |
| c. Data on Scientific Collaborators | | X | | | |
| d. Information on Inventions | ✘ | | | | |
| e. Technical Description of Project and Results | | ✘ | | | |
| f. Other (specify) | | | | | |

| 2. Principal Investigator/Project Director Name (Typed) | 3. Principal Investigator / Project Director Signature | 4. Date |
|---|---|---|
| Dimitri Mavris | Dimitri Mavris | 12/15/23 |

FAA Form 9550-5 (03-03) Supersedes Previous Edition

## PART III – TECHNICAL INFORMATION

a) N/A
b) Publication Citations
N/A

c) Data on Scientific Collaborators

**Investigation Team**

Dimitri Mavris, PI, Georgia Institute of Technology; overall lead.

Holger Pfaender, Co-PI, Georgia Institute of Technology; day-to-day lead supporting all tasks.

Raphael Gautier, Research Engineer, Georgia Institute of Technology, computing lead supporting all tasks.

Michelle Kirby, Senior Research Engineer, division lead, supported reporting and management tasks.

Anthony Markowitz, Graduate Research Assistant, literature search task 1.

Cem Yumuk, Graduate Research Assistant, helped integrating Python frameworks.

Aroua Gharbi, Graduate Research Assistant, contributed to the initial design of the UAS noise analysis.

Martin Delage, Graduate Research Assistant, helped with the demand data organization.

Joaquin Matticoli, Graduate Research Assistant, contributed on tasks2,3, and 4.

Deepika Singla, Graduate Research Assistant, contributed on tasks 2, 3, and 4.

Xi Wang, Graduate Research Assistant, helped on all tasks with processing of geospatial data.

Hugue Chardin, Graduate Research Assistant, implemented the warehouse to staging location and final address allocation algorithm.

Jing Penfei, helped on the visualization of the contours.

# Project 009 Geospatially Driven Noise-estimation Module

## Georgia Institute of Technology

## Project Lead Investigators

Prof. Dimitri N. Mavris
Director, Aerospace Systems Design Laboratory
School of Aerospace Engineering
Georgia Institute of Technology
Phone: 404-894-1557
Fax: 404-894-6596
dimitri.mavris@ae.gatech.edu

Dr. Holger Pfaender
Aerospace Systems Design Laboratory
School of Aerospace Engineering
Georgia Institute of Technology
Phone: 404-385-2779
Fax: 404-894-6596
holger.pfaender@ae.gatech.edu

## University Participants

**Georgia Institute of Technology**

- P.I.s: Prof. Dimitri Mavris, Dr. Holger Pfaender
- FAA Award Number: 13-C-AJFE-GIT-059
- Period of Performance: June 5, 2020 to September 30, 2023
- Tasks:
    1. Literature review and evaluation of geographic information systems (GIS) software
    2. Investigation of emerging computational technologies
    3. Collaboration with the unmanned aircraft system (UAS) computation module development team
    4. Noise computation engine integration

## Project Funding Level

The Georgia Institute of Technology has received $499,999 in funding for this project. The Georgia Institute of Technology has agreed to a total of $250,000 in matching funds. This total includes salaries for the project director, research engineers, and graduate research assistants as well as for computing, financial, and administrative support, including meeting arrangements. The institute has also agreed to provide tuition remission for students whose tuition is paid via state funds.

## Investigation Team

**Georgia Institute of Technology**

Prof. Dimitri Mavris (P.I.)
Dr. Holger Pfaender (co-P.I.)
Research Faculty: Raphael Gautier
Graduate Students: Joaquin Matticoli, Deepika Singla, Xi Wang, Hugues Chardin, Aroua Gharbi, Martin Delage

# Project Overview

## Context and Motivation

The UAS market is expected to grow rapidly in coming years, with projections estimating the civil UAS market at $121 billion in the next decade (Teal Group, 2022). Multiple operators are currently developing and testing various concepts of operations that fall under the umbrella of urban air mobility (UAM), with the two main use cases being drone delivery and e-taxi operations. Similar to traditional aircraft operations, these novel concepts are expected to influence the environment in which they operate, particularly regarding noise. In the same way that noise assessments of terminal operations are carried out today for commercial aviation, noise assessments of UAM operations are expected to be necessary in the future.

## Problem Definition

UAM operations bring unique requirements. First, UAM operations are expected to be denser than current general or commercial aviation operations, possibly by orders of magnitude. Thus, noise-assessment methods should be able to handle such large vehicle densities. Second, the vehicles are expected to be smaller and therefore quieter; for example, small drones for deliveries or helicopter-sized vehicles for e-taxi applications, but these vehicles are also expected to benefit from novel electric-propulsion systems. As a result, the noise footprint of such vehicles is expected to be more localized. Therefore, noise exposure levels should be estimated with sufficient resolution. Third, instead of primarily following fixed trajectories dictated by approach and departure routes around airports, UAM vehicles are expected to operate point-to-point within populated areas. Departure and arrival locations are expected to vary from day to day; delivery drones may depart from warehouses or mobile staging locations and deliver goods to different customers each day, and e-taxis may pick up and drop off customers throughout an urban area. Thus, noise-assessment methods should be sufficiently flexible to accommodate changing flight paths, and the resulting noise assessments should account for corresponding variability.

## Research Objectives

In view of these requirements, the methods used to perform noise assessments in terminal areas, such as the Aviation Environmental Design Tool (AEDT), are not fully suitable for UAM assessments; these methods are usually limited to studies of relatively low-density operations around airports, with vehicles following predefined ground tracks. Thus, there is a need to develop new noise-assessment capabilities tailored to UAM operations, which is the focus of this project.

## Research Approach

Research efforts supporting the development of a UAS noise-assessment tool have been broken down into four tasks.

First, GIS capabilities are expected to play a major part in the development of this tool, as the scenarios under consideration and the resulting noise metrics are to be visualized and overlaid on the geographical area of study. Therefore, Task 1 focuses on a literature review and evaluation of GIS software.

Second, the complexity of assessing noise in the context of UAM-use cases, as discussed in the previous section, calls for an investigation of emerging technologies in multiple computational domains. The size of these problems and the flexibility needed to analyze a wide variety of operational scenarios require the introduction of recent innovations to address the challenges discussed previously. This is the focus of Task 2.

This research was conducted in collaboration with other entities, starting with Mississippi State University (MSU), and followed by subsequent collaborations, which are presented under the umbrella of Task 3.

Finally, Task 4 focuses on the integration of all components investigated or provided by other tasks into the actual UAS noise-assessment engine. Technical details pertaining to the implementation, as well as preliminary results on benchmark test cases, are presented in this section.

# Task 1 - Literature Review and Evaluation of GIS Software

Georgia Institute of Technology

## Task 1 Contents



## 1.1 Objective

This task aims to identify the leading open-source GIS software using preset evaluation criteria.

## 1.2 Research Approach

This review focused on open-source options. For an adequate evaluation of the options, six criteria were set forth:

1. Data import: ability to read shape files with different formats of input geometrical data as well as rasterized (gridded) data
2. Data storage: capability to store geospatial data in either shape/vector formats or as rasterized data
3. Geometric calculations: ability to convert to and from a Cartesian coordinate system and other Earth model coordinates and ability to compute polygon areas and lengths as well as unions and subtractions
4. Geospatial calculations: ability to perform calculations on given vector or raster data and to draw contour plots
5. Display: ability to print raw or processed geospatial data as various map displays and to enable standard desktop and web applications
6. Map data: capability to display results with relation to landmasses, political boundaries such as states and counties, and roads and buildings

In addition to evaluating software, we also investigated GIS applications to examine the option of creating a stand-alone, customized library or component.

## 1.3 GIS Libraries

### 1.3.1 QGIS

QGIS is a user-friendly, open-source GIS written in C++. The latest version is 3.24 (released in February 2022). QGIS runs on Linux, Unix, Mac OSX, Windows, and Android and supports numerous vector, raster, and database formats and functionalities. As well as its intrinsic, built-in functionalities, QGIS allows users to install and create their own plug-ins. New applications can also be created in QGIS through C++ and Python languages. Screenshots of QGIS are shown in Figure 1.


**Figure 1**. QGIS screenshots.

***Evaluation Criteria***

1. Data import: imports shape files such as GPX, GPS, DXF, DWG, and OpenStreetMap, as well as raster files
2. Data storage: stores geospatial data in vector and raster formats

3. Geometric calculations: supports Cartesian (x, y), polar (length, angle), and projected (x-north, y-east) calculations; calculates length or area of geometrical features; and provides overlay, union, and difference between areas
4. Geospatial calculations: creates a vector contour map from an elevation raster and carries out raster-to-vector conversion
5. Display: can provide web mapping with QGIS2Web; can publish data on the internet using a webserver with the University of Minnesota MapServer or GeoServer installed
6. Map data: displays geospatial data such as countries, states, and counties as well as roads

### 1.3.2 OpenJUMP

OpenJUMP is a Java-based, open-source GIS (latest version: 2.0, released in March 2022). OpenJUMP works on Windows, Linux, and Mac platforms with Java 1.7 or later. OpenJUMP's features include reading and writing vector formats, displaying geospatial data, and executing geometric calculations. Additional plug-ins for more capabilities are also available. OpenJUMP is distributed under the GNU General Public License version 2. Screenshots of OpenJUMP are shown in Figure 2.

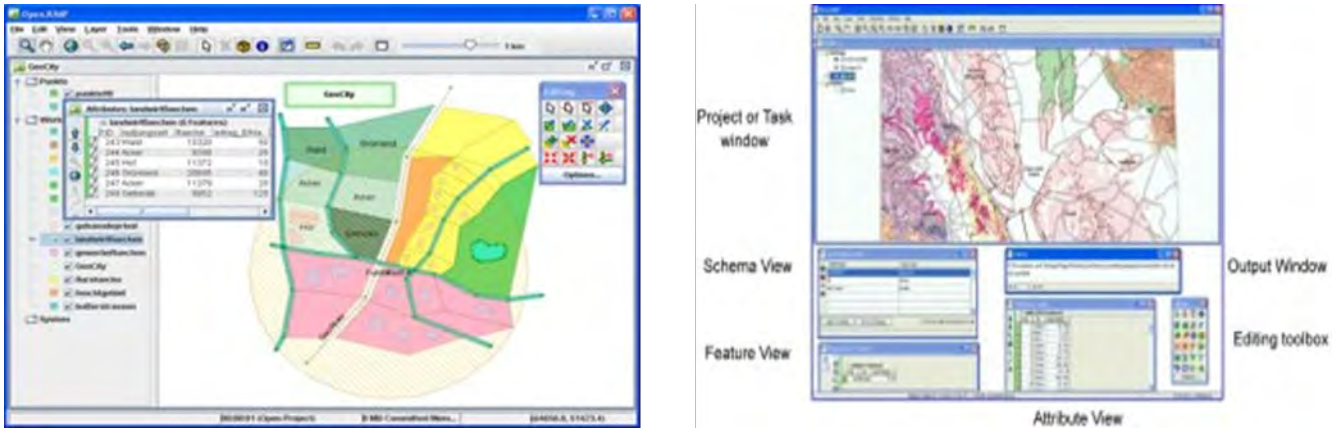


**Figure 2.** OpenJUMP screenshots.

***Evaluation Criteria***

1. Data import: imports shape and raster files
2. Data storage: stores geospatial data in vector and raster formats
3. Geometric calculations: supports coordinate reference system (CRS) (Cartesian [x, y, z], geographic [longitude, latitude, height], and projected [x-north, y-east]) conversions; provides a CRS transformation tool (PROJ4); calculates length or area of geometrical features; provides overlay, union, and subtraction
4. Geospatial calculations: provides conversion between desired file formats (raster-to-vector conversion); does NOT provide contour plots
5. Display: does NOT provide a web application
6. Map data: displays geospatial data such as countries, states, and counties as well as roads

### 1.3.3 System for Automated Geoscientific Analyses (SAGA)

The System for Automated Geoscientific Analyses (SAGA) is an open-source, cross-platform GIS software written in C++ (latest version: 2.0, released in June 2007). SAGA can be run on Windows, Linux, FreeBSD, and Mac (OS X). SAGA provides multiple libraries for GIS calculations: digital terrain analysis, image segmentation, fire spreading analysis and simulation, etc. In addition to these libraries, SAGA allows the scripting of custom models through the command line interface (CLI) and Python interface. Screenshots showing the SAGA environment are shown in Figure 3.

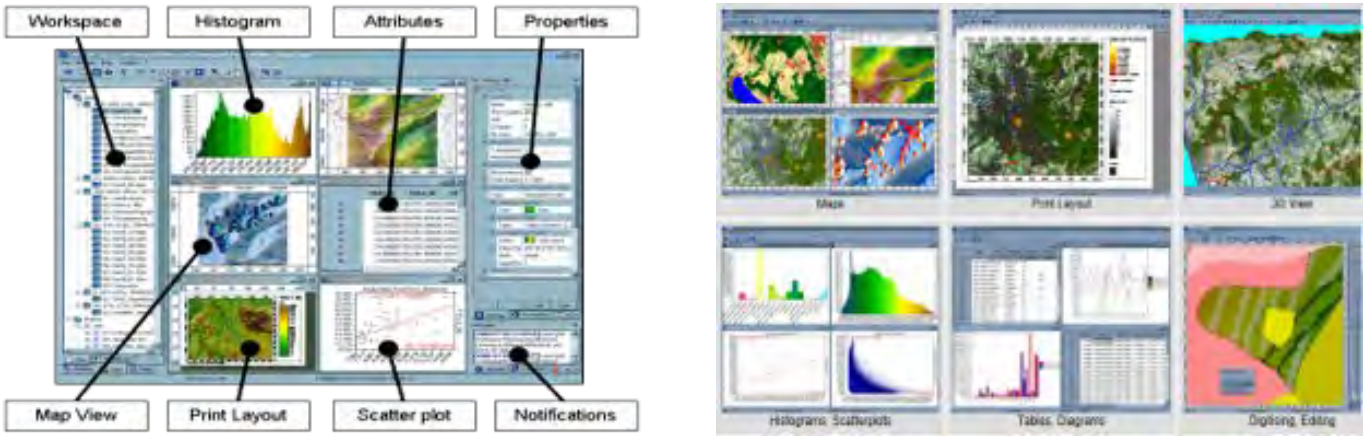


**Figure 3.** Screenshots showing the SAGA environment.

***Evaluation Criteria***

1. Data import: imports shape and raster files
2. Data storage: stores geospatial data in vector and raster formats
3. Geometric calculations: supports geographic coordinate system (latitude, longitude) and Universal Transverse Mercator (UTM) calculations; computes polygon areas or lengths
4. Geospatial calculations: performs raster-to-vector conversions and can create contour lines
5. Display: displays data as histograms and scatter plots
6. Map data: enables visualization of spatial data as cartographic maps; can also import maps from Web Map Service and OpenStreetMap

**1.3.4 Deck.gl**

Deck.gl is a WebGL visualization framework for large datasets (latest version: 8.7.3, released in March 2022). Deck.gl allows users to map data (JavaScript Object Notation [JSON] objects, comma-separated values [CSVs]) into a stack of layers. These layers can be imported directly from a catalog or built by the user.

***Evaluation Criteria***

1. Data import: reads shape files and CSV/GeoJSON files
2. Data storage: can store geospatial data as vector or shape files
3. Geometric calculations: supports geographic coordinate system (latitude, longitude) using Web Mercator; does NOT calculate polygon areas or lengths
4. Geospatial calculations: does not convert raster data to vector data; can create contour lines for a given threshold and cell size
5. Display: offers an architecture for packaging advanced WebGL-based visualizations; enables users to rapid obtain impressive visual results with limited effort
6. Map data: easily displays geospatial data with relation to roads and buildings

**1.3.5 Kepler.gl**

Kepler.gl is an open-source geospatial analysis tool for large-scale datasets (version 2.5.5). The most recent update was made in September 2021. A user interface was created to facilitate the process of saving a map to back-end storage, and a graphics processing unit (GPU) data filter was added, with the ability to create polygon filters in the user interface.

***Evaluation Criteria***

1. Data import: ability to read CSV/GeoJSON files and Kepler.gl's sample datasets; must convert shape files to a GeoJSON file to be consumable by Kepler.gl
2. Data storage: cannot store geospatial data as vector or shape files
3. Geometric calculations: supports geographic coordinate system (latitude, longitude) using Web Mercator; does NOT calculate polygon areas or lengths
4. Geospatial calculations: does not convert raster data to vector data; can create contour lines
5. Display: offers an architecture for packaging advanced WebGL-based visualizations and can easily handle sample data to visualize
6. Map data: easily displays geospatial data with relation to roads and buildings

### 1.3.6 Geographic Resources Analysis Support System GIS

Geographic Resources Analysis Support System (GRASS) is an open-source, Java-based software for vector and raster geospatial data management, geoprocessing, spatial modeling, and visualization. GRASS has compatibilities with QGIS, meaning that QGIS can run some features of GRASS GIS as a plug-in. Already developed add-ons are available, along with the capability to develop additional add-ons. The latest version (8.0, released in March 2022) has an improved graphical user interface (GUI) and Python scripting. GRASS provides rapid linking of external raster files and spatiotemporal data analysis with an improved internal data structure. A vector attribute update was also found with Python syntax. A typical screenshot from GRASS GIS is shown in Figure 4.

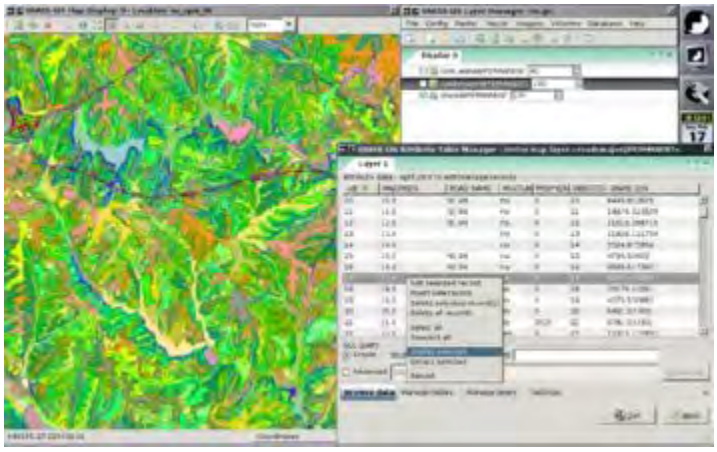

**Figure 4.** Typical screenshot from the Geographic Resources Analysis Support System (GRASS).

***Evaluation Criteria***

1. Data import: imports vector and raster files
2. Data storage: stores geospatial data in vector and raster formats
3. Geometric calculations: supports coordinate reference system (CRS) (Cartesian [x, y, z] and geographic [longitude, latitude, height]) conversions; provides a CRS transformation tool (PROJ4); calculates length or area of geometrical features; provides overlay, union, and subtraction
4. Geospatial calculations: provides conversion between desired file formats (raster-to-vector conversion); creates contour lines
5. Display: provides a Web Mapping Service and graphics display monitor that can be controlled from the command line; can display frames on the user's graphic monitor
6. Map data: displays geospatial data such as countries and states by using Inkspace

### 1.3.7 gvSIG

gvSIG is an open-source GIS written in 2021 that runs on Windows, Linux, and Mac platforms. Screenshots from gvSIG are shown in Figure 5.

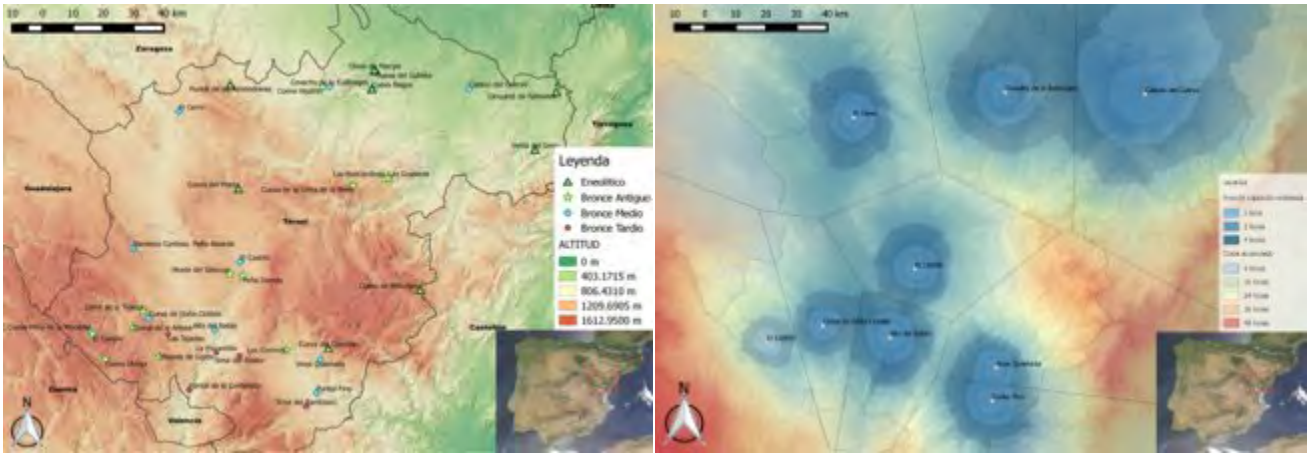


**Figure 5.** Screenshots from gvSIG.

***Evaluation Criteria***

1. Data import: can import shape and raster files
2. Data storage: can store geospatial data in vector and raster formats
3. Geometric calculations: supports geographic coordinate system (latitude, longitude) using Web Mercator; does NOT calculate polygon areas or lengths; supports CRS (Cartesian [x, y, z] and geographic [longitude, latitude, height]) coordinates; provides a CRS transformation tool (PROJ4); calculates length or area of geometrical features; provides overlay, union, and subtraction
4. Geospatial calculations: can convert other file types to the desired file format; does NOT produce contour plots
5. Display: does NOT provide a web application
6. Map data: displays geospatial data such as countries and states by using Inkspace

### 1.3.8 MapWindow GIS

MapWindow GIS is an open-source GIS written in C++ using optimal features from the .NET framework v4/4.5. MapWindow runs on Windows (latest version: 5.3.0, released in 2019), as shown in Figure 6. This version was compiled using VS2017. The new version supports tiles from a local file system and provides extendable snapping events. MapWindow was licensed under the Mozilla Public License.

***Evaluation Criteria***

1. Data import: can import shape and raster files
2. Data storage: can store geospatial data in vector and raster formats
3. Geometric calculations: supports geographic coordinate system (latitude, longitude) and UTM calculations; can calculate length or area of geometrical features
4. Geospatial calculations: can convert other file types to the desired file format; does NOT produce contour plots
5. Display: allows multi-threaded HTTP tile loading
6. Map data: displays geospatial data such as countries and states by using Inkspace

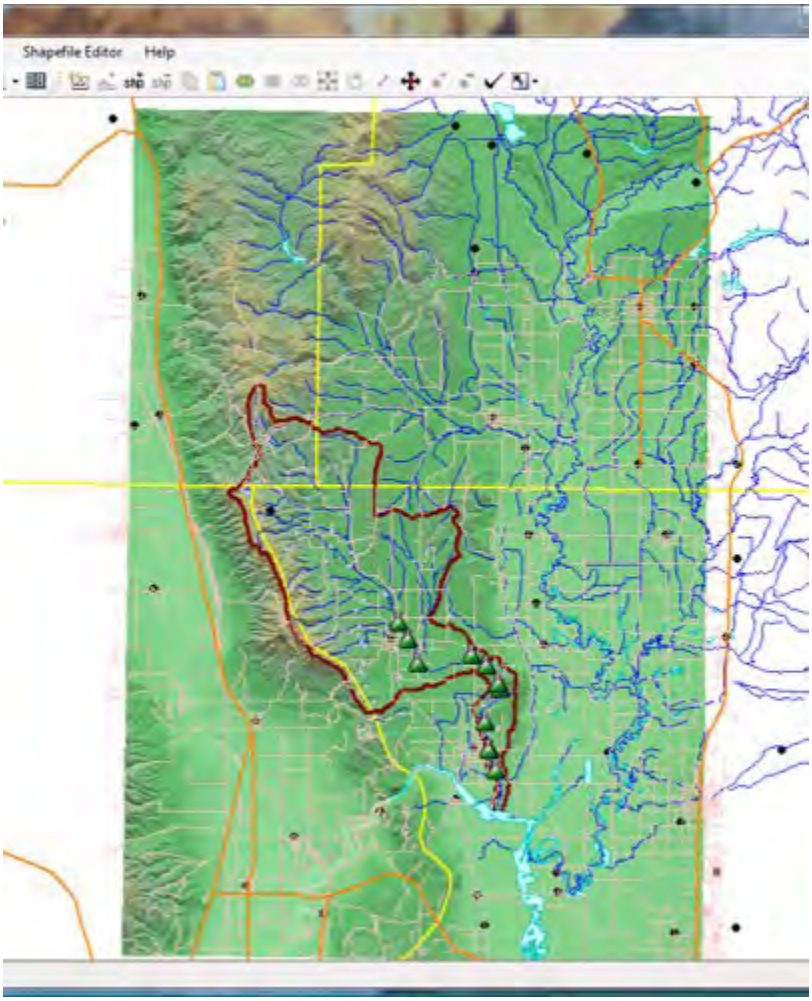


**Figure 6.** Screenshot from the MapWindow geographic information system.

### 1.3.9 GeoPandas

GeoPandas is an open-source project developed in Python to provide a useful library for working with geospatial data, as shown in Figure 7. GeoPandas can run on distributions of Linux and Windows. This software primarily uses the Python packages pandas (as a base for its data storage), shapely (to manipulate the shapes stored in the advanced database), Fiona (for file access), and Descartes and matplotlib (for data visualization). GeoPandas is most adept at displaying discrete sections of data in a geospatial visualization. It is limited in its ability to display graphics outside of the Python environment and does not support conversion to the desired raster/vector formats. The last update was made in 2021, which improved the software from v0.5.0 to v0.10.2 and corrected the regression in the overlay and plotting.

***Evaluation Criteria***

1. Data import: reads almost any vector-based spatial data format
2. Data storage: stores geospatial data in vector and raster formats
3. Geometric calculations: supports CRS calculations; cannot calculate the length or area of geometrical features; has overlay functions, such as intersections between two or more areas, union (merges the areas of one layer to one single area), difference (A-B areas), and polygons
4. Geospatial calculations: does not convert to any desired file formats (no raster-to-vector formats); does not provide a contour plot function

5. Map data: uses various map projections using the Python library Cartopy
6. Display: does not provide a web application; provides a good representation in three-dimensional (3D) color space using matplotlib


**Figure 7.** An example of how GeoPandas can overlay processed geospatial data over existing maps.

### 1.3.10 WorldWind

WorldWind is an open-source, virtual 3D globe-visualization application programming interface (API) developed by NASA in partnership with the European Space Agency. WorldWind is written in both Java (for desktop and Android devices) and JavaScript (for web applications). After its development was suspended in 2019, it was restarted in August 2020. WorldWind can import a variety of input files with geospatial data, stores the data in both raster and vector formats, provides sufficient geometric and geospatial calculations, and produces good visualizations with comprehensive map data. WorldWind finds its application in unmanned aerial vehicle imagery, where such vehicles can provide continuous monitoring of an active fire, with higher resolution and more frequent updates. WorldWind was licensed under NASA Open-Source Agreement Version 1.3. Screenshots of WorldWind are shown in Figure 8.

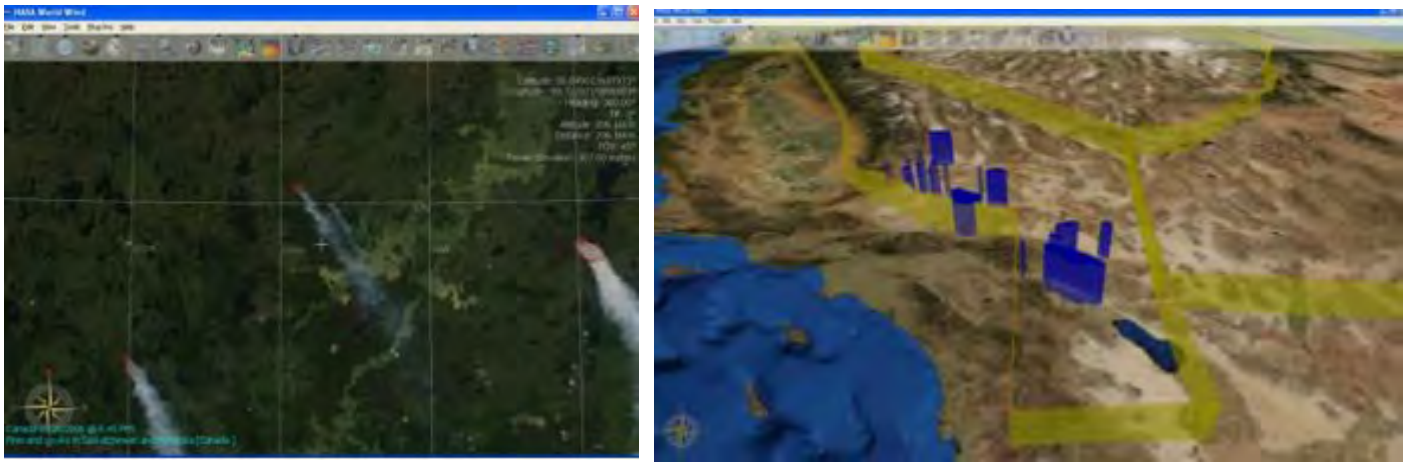

**Figure 8.** Screenshots from WorldWind.

***Evaluation Criteria***

1. Data import: imports shape files, KML, VPF, GML, GeoJSON, GeoRSS, GPX, NMEA, etc.
2. Data storage: stores geospatial data in vector and raster formats
3. Geometric calculations: supports geographic coordinate system (latitude, longitude), UTM, and Military Grid Reference System calculations; can draw and measure distance and area across a terrain
4. Geospatial calculations: displays contour lines on surface terrain at a specified elevation
5. Map data: provides visual representations of scalar values, such as noise, over a grid of geographic positions; can visualize the results on web and Android platforms
6. Display: displays geospatial data divided into country, state, and city

### 1.3.11 Overall Evaluation

An overall evaluation of all of the investigated libraries is provided in Table 1. QGIS seems to surpass the other libraries with respect to our defined metrics.

**Table 1.** Comparison of different libraries.

| | Intuitive GUI | Compatibility | Statistical Analyses | Data Import | Data Storage | Geometrical Calculations | Geospatial Calculations | Map Data | Display | Total |
|---|---|---|---|---|---|---|---|---|---|---|
| QGIS | 3 | 5 | 3 | 5 | 5 | 5 | 5 | 5 | 4 | 40 |
| OpenJUMP | 3 | 4 | 1 | 5 | 5 | 5 | 3 | 5 | 2 | 33 |
| SAGA | 3 | 3 | 4 | 5 | 5 | 4 | 5 | 5 | 4 | 38 |
| Deck.gl | 4 | 3 | 1 | 5 | 5 | 3 | 3 | 5 | 5 | 34 |
| Kepler.gl | 4 | 5 | 1 | 1 | 1 | 3 | 3 | 5 | 5 | 28 |
| GRASS | 4 | 3 | 1 | 5 | 5 | 4 | 5 | 5 | 4 | 36 |
| gvSIG | 3 | 4 | 1 | 5 | 5 | 4 | 3 | 5 | 2 | 32 |
| MapWindow | 3 | 4 | 1 | 5 | 5 | 3 | 3 | 4 | 2 | 30 |
| GeoPandas | 2 | *4* | 1 | 5 | 5 | 4 | 1 | 2 | 2 | 26 |
| WorldWind | 5 | *5* | 1 | 5 | 5 | 4 | 4 | 5 | 5 | 39 |

## 1.4 GIS Applications

GIS applications can be broadly classified in two categories: desktop and web-based applications.

WebGIS applications use web technologies to display and communicate geospatial information to an end user. Every WebGIS application has five common elements:

1. A web application: the interface used by the client, which has tools for visualizing, analyzing, and interacting with geographic information and can be run on a web browser or a GPS-enabled device
2. Digital base maps: the geographical context for the application (e.g., transportation, topography, imagery)
3. Operational layers: the layers used in order for the results of an operation to be displayed (e.g., observations, sensor feeds, query results, analytic results)
4. Tasks and tools: tools to perform operations beyond mapping
5. Geodatabase(s): container of geographical data, which can consist of geodatabases, shape files, tabular databases, computer-aided design files, and so on

WebGIS applications come with multiple advantages and limitations. Table 2 presents a inexhaustive list of these advantages and limitations.

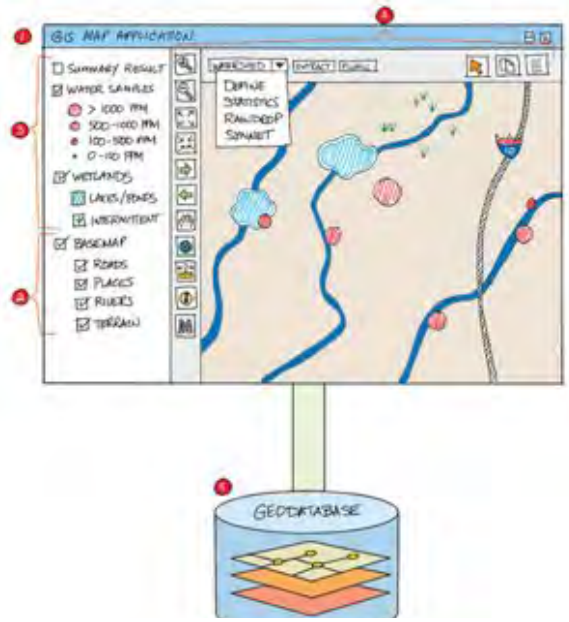


**Figure 9.** Schematic of a web geographic information system (WebGIS) application.

**Table 2.** Advantages and disadvantages of web geographic information systems (WebGISs).

| Advantages of WebGIS | Disadvantages of WebGIS |
|---|---|
| • Provides a broader reach for customers compared **with** a traditional desktop application<br>• Better cross-platform capability with the different web browsers that can be used<br>• Easy to use for customers with different levels of geographic information systems **(GIS)** expertise<br>• Extendable to cloud services, hence allowing manipulation and use of big GIS data<br>• Lower cost to entry (most libraries and tools are open-source with good community support)<br>• Allows real-time analysis | • Harder to build (developers need to have a good knowledge of multiple scripting languages to build the app [Python, JavaScript, html, etc.])<br>• Data security may depend on a third party<br>• Application may need to be hosted outside of the organization |

Our team has started a dialogue with the AEDT development team regarding which GIS functionalities will be required to be able to integrate the UAS noise engine with the AEDT in the future.

# Task 2 - Investigation of Emerging Computational Technologies

Georgia Institute of Technology

## Task 2 Contents



## 2.1 Task 2 Overview

### 2.1.1 Context and Motivation

As explained in the project overview, assessing noise exposure for UASs brings unique requirements that existing frameworks do not meet. Three primary abilities are needed: (a) the ability to analyze scenarios involving large volumes of flights; (b) the ability to cover large areas with small resolution; and (c) the ability to account for sources of uncertainty related to the evolving UAS concepts of operation. Thus, there is a need for the development of a new analysis capability that can fulfill these requirements.

### 2.1.2 Problem Definition

Although the actual estimation of noise exposure levels plays a central role in noise-assessment tools, many other peripheral functions are also needed: inputs must be read and preprocessed, computations must be implemented in such a way that they meet the requirements listed in the previous section, and a visualization of the operational scenario and noise assessment results must be provided in a manner that is intuitive to the user. Each of these functionalities requires a substantial development effort and can leverage specific computational technologies.

### 2.1.3 Research Objectives

In this task, we aim to investigate the emerging technologies that could be used to implement the variety of functions to be performed by the noise-assessment tool. In particular, we are seeking technologies that are compatible with the stringent requirements related to UAS operations.

### 2.1.4 Research Approach

For this task, the following areas of emerging technologies were identified and investigated. Figure 10 presents a partial depiction of these areas and the associated technologies.

First, GIS visualization techniques were investigated. Within the noise-assessment tool, these techniques are used to visualize the defined operational scenarios, such as the flights included in the scenario, as well as the analysis results, in the form of noise levels mapped over a predefined geographical area.

Second, parallel computation approaches were investigated to address the problem of performing noise computations with large problem sizes encountered due to (a) the large flight volumes in UAM scenarios; (b) the low resolution and large areas needed to effectively cover populated areas; and (c) the temporal discretization needed to properly assess noise exposure.

Third, data pre- and postprocessing approaches were investigated, because working with geographical data usually requires many transformations, such as clipping to the analysis area or converting from one CRS to another.

Fourth, motivated by the need to speed up noise computations to enable faster uncertainty quantification, we investigated running the noise engine on a GPU.

Finally, we developed the capability to run the noise engine on cloud-based platforms, specifically AWS, since this approach allows us to scale noise computations for a large number of workers and large amounts of total memory, enabling the analysis of problems the size of which would be prohibitively large for execution on a single machine.

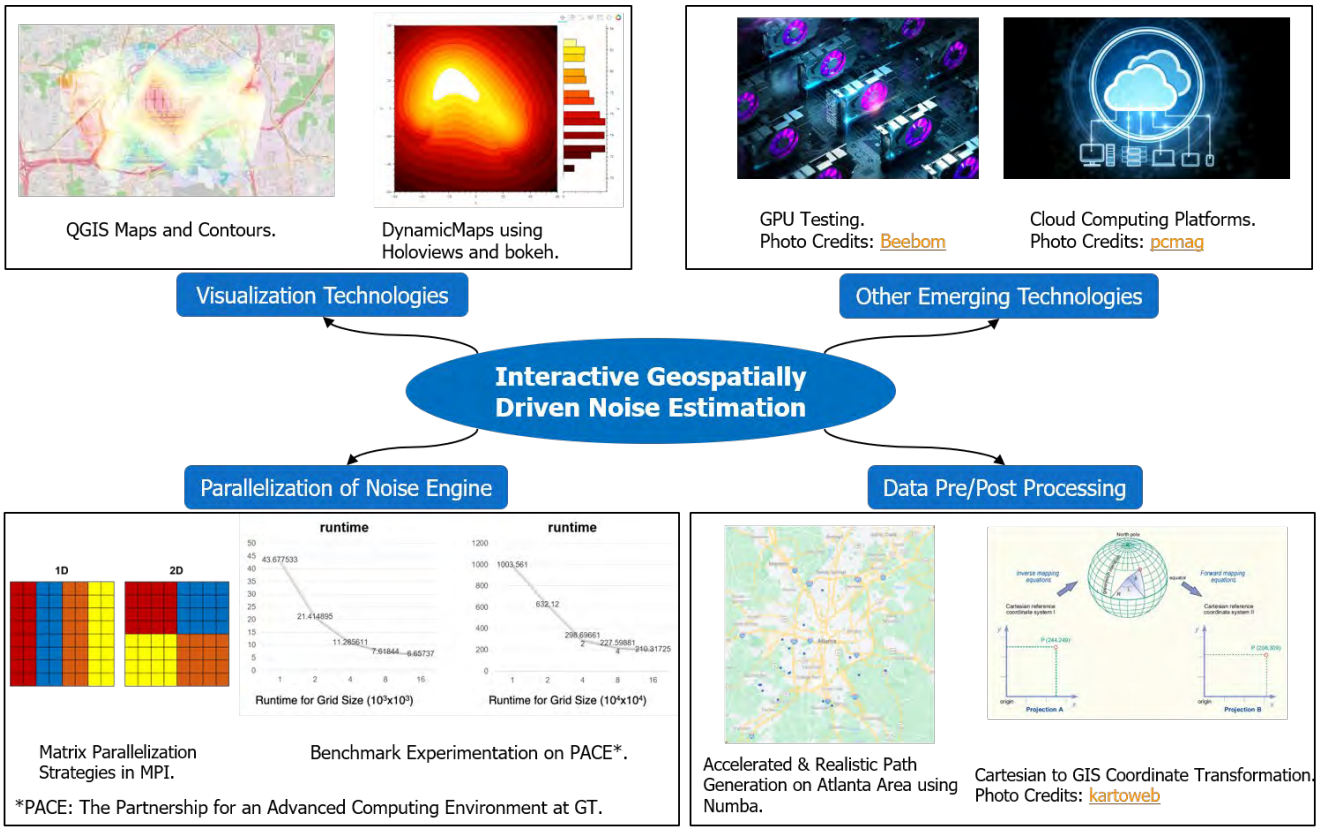


**Figure 10.** Visual summary of the emerging technologies under investigation. GIS: geographic information system; GPU: graphics processing unit; GT: Georgia Institute of Technology; MPI: message passing interface.

## 2.2 GIS-visualization Technologies

The team focused on technologies that provide interactive visualizations of large data on maps, which narrowed the choices to QGIS and interfaces based on Python or JavaScript. Working with large datasets on QGIS requires the use of a structured query language (SQL) plug-in as a conduit for data communication. Furthermore, the GUI aspect of QGIS limits the interactive capabilities that can be achieved.

Therefore, the focus was directed to JavaScript and Python libraries and interfaces, including the D3 library for JavaScript and Bokeh for Python. Bokeh emerged as the preferred choice as it builds on JavaScript visualizations without the need to explicitly use JavaScript. Furthermore, with this library, it is possible to code both the front-end and back-end of a web application using Python.

## 2.3 Parallel-computing Technologies

Parallel computing technologies are critical for calculations that involve large grids. These grids can be expressed as matrices and hence take advantage of their regular structures for the partition of computation tasks.

The team initiated their analysis by exploring the standards for parallel programming via the message passing interface (MPI) implemented on different libraries, such as OpenMPI, MPICH, and MVAPICH. As the noise computation engine is built from common mathematical and computational operations, OpenMPI was selected for its portability and its ability to support most existing platforms.

Parallel algorithms for matrix computations have been well documented in the literature. Typically, the data are partitioned either along one axis of the matrix or both, as shown in Figure 11. These algorithms are usually designed with considerations of the communication overhead and the computation cost for individual processors.

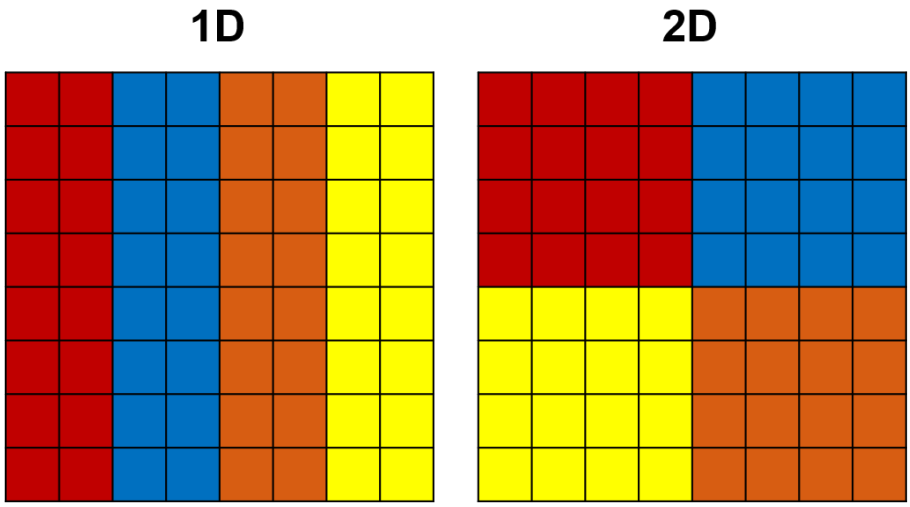


**Figure 11.** Common partition strategies for matrix computations.

The noise engine can be viewed as a large, dense matrix problem in which the calculations for each element do not depend on its neighbors. Instead, these calculations depend on the path of the noise source, which can be modeled as a vector. Hence, the partition strategies shown are theoretically the same, where the main challenge is to manage the data communicated. In addition to communicating the path data to each partition, the engine needs to collect the results and send them to the visualization tool.

These considerations prompted us to examine the input/output (IO) operations in parallel, as shown in Figure 12. There are three main approaches for parallel IO operations, as briefly defined below:

- Nonparallel: A central unit is uniquely responsible for the IO operations.
- Independent parallel: Each process writes to a separate file.
- Cooperative parallel: All processors collaboratively write in one file.

The main advantages and disadvantages for each approach are summarized in Table 3. Although the cooperative parallel approach has the potential to achieve the best performance, it is limited in terms of the file types that can be used, and it may result in performance that is worse than that of the sequential algorithm. Therefore, we did not select a cooperative parallel IO approach. Instead, the choice will depend on other characteristics of the overall noise module.

**Table 3**. Parallel input/output (IO) operations.

| Parallel IO Approach | Advantages | Disadvantages |
|---|---|---|
| **Nonparallel** | • Easy to code | • Poor performance (worse than sequential) |
| **Independent Parallel** | • Easy to parallelize<br>• No interprocess communication | • Generates many small files to manage |
| **Cooperative Parallel** | • Performance can be great<br>• Only one file is needed | • More complex to code<br>• Depends on implementations of concurrent updates in file types, which are rare |

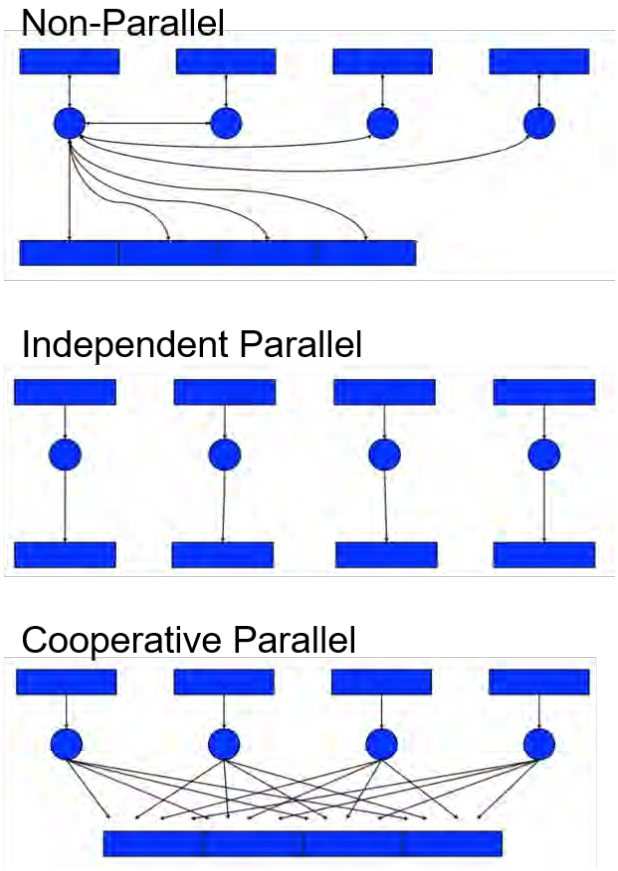


**Figure 12.** Schematic illustrating the input/output operations in a message passing interface.
*Source: William Gropp, Introduction to MPI I/O*

The analysis of parallel IO approaches led to the need to examine the file formats used in parallel as well. Three major categories of file formats are listed in Table 4, along with their major advantages and disadvantages.

**Table 4.** Benefits and drawbacks of file formats.

| File Format | Advantages | Disadvantages |
|---|---|---|
| **ASCII** | • Human-readable<br>• Portable | • Requires a larger amount of storage<br>• Costlier for read/write operations |
| **Binary** | • Efficient storage<br>• Less costly for read/write operations | • Needs formatting to read |
| **Standard scientific libraries (HDF5, NetCDF, etc.)** | • Allows data portability across platforms<br>• Data stored in binary form<br>• Includes data description | • Has a risk of corruption |

This analysis was conducted with a gridded data format in mind. Instances of these files that are encoded in binary format are relatively straightforward to create and manage in parallel because the MPI writes to binary format by default. Instances that use ASCII characters are more difficult to use, however, because a binary–ASCII conversion is needed for formatting.

To showcase the runtime difference between ASCII files and binary files, a test case was run with a fixed problem size and a variable number of processors. The test used the independent parallel approach to eliminate the need for a central unit that collects the results. Figure 13 illustrates the runtimes of text file problems and binary problems for 2 to 16 processors. The "runtime no IO" scenario was included in Figure 13 as a baseline to showcase the cost of communication due to the IO operations. As expected, for a fixed problem, the runtime decreased as the number of processors increased; however, the difference between runtimes with respect to the file formats is quite apparent.

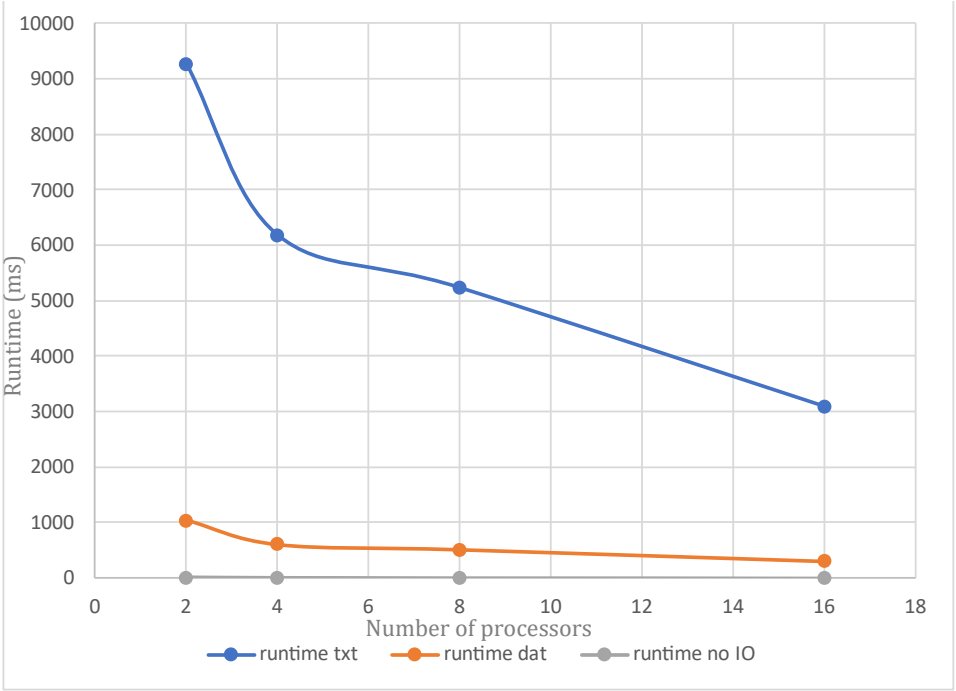


**Figure 13.** Runtime vs. number of processors for different input/output (IO) formats.

Furthermore, for any format used, storage space will be needed to contain the data, as shown in Figure 14, which reveals an exponential growth in size as the grid becomes finer. This test case demonstrates that the available memory of the hardware used will play an important role in the calculation of large grids.

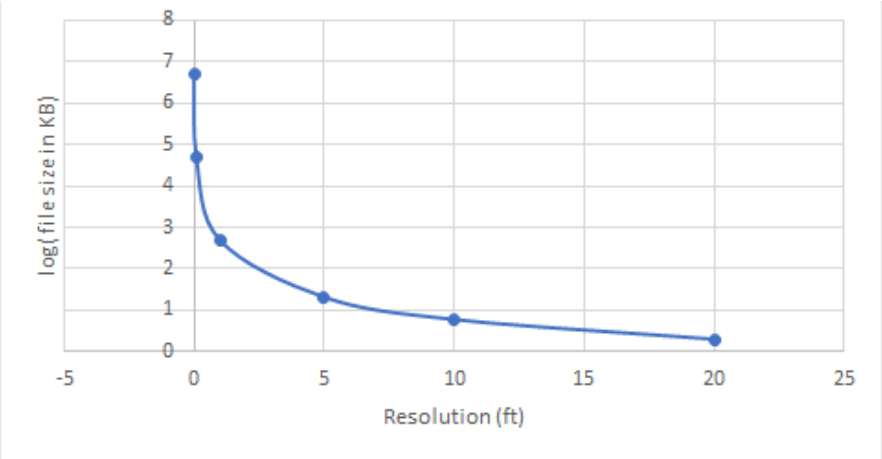


**Figure 14.** Log(file size in kB) vs. resolution (ft).

The choice of programming language is another important aspect to consider in this investigation. Programming languages such as C and C++ combined with MPI libraries are the primary choice of many high-performance computing (HPC) practitioners, as they have some access to low-level machine language, which results in good performance for parallel computations. However, the main challenge in using these languages is the integration with interactive GIS visualization tools. Higher-level languages such as MATLAB and Python provide these libraries with much less scripting and easier integration, but this comes at the expense of speed in running parallel code. In particular, MATLAB requires the setup of a virtual network computing session prior to launching any calculations. Python, despite being slower than C/C++, emerged as an adequate choice for the noise module, as it is better equipped to facilitate large interactive GIS visualizations without greatly sacrificing speed for this particular application while still being able to act as a wrapper for rapid C/C++ implementations of the computational code.

## 2.4 Data-processing Technologies

The team investigated libraries for processing GIS data. As the investigation of visualization techniques favored the use of Python to code the application, libraries such as GeoPandas and GeoTIFF were explored to assess their compatibility with the goals of this project.

The GeoPandas library brings the powerful functionalities of pandas to geospatial operations. The GeoTIFF format allows the embedding of geospatial data into images. GeoPandas is more suited to work with vector data, whereas GeoTIFF supports both raster and vector formats. Each of these libraries has its own merits and utilizations and can be used in the noise calculation engine. The final choice will depend on the data pipeline from the computation to the visualization and the data conversions needed in this process.

## 2.5 Support for GPU-backed Computations and Scaling Study

### 2.5.1 Context and Motivation

The ability to account for variability in operations, as well as other sources of uncertainty emanating from currently unknown parameters, is one of the main requirements for the UAS noise-assessment tool. Indeed, the need for this ability is one of the reasons why existing tools are not adapted for UAS use cases and why the development of a new capability is needed.

Once sources of uncertainty have been characterized and quantified, Monte Carlo simulations are, a priori, the preferred option for propagating the impact of those uncertainty sources to system-level responses of interest. Monte Carlo simulations are preferred because, among the multiple options available to propagate uncertainty, running full Monte Carlo simulations (a) usually does not require any additional assumptions regarding the nature of the uncertainty sources or the system model, and (b) gives access to full probability distributions for system-level responses, which can be used to estimate any statistical quantity related to these responses. In contrast, approximate uncertainty propagation methods (a) may require uncertainty sources and the system model to behave a certain way to produce valid results, and (b) may only approximate a few statistics, such as the mean of the responses.

In the case of UAM operations, the nature of uncertainty sources (e.g., vehicles may depart and arrive in different locations, the number of flights may vary), as well as the nature of the system model, does not immediately appear to be prone to an approximation method; therefore, a full Monte Carlo simulation will be conducted. Applying approximate uncertainty quantification on this problem will be the topic of future research.

### 2.5.2 Problem Definition

An initial Monte Carlo study was conducted using the initial Dask implementation of the noise-assessment tool running on a central processing unit (CPU). The setup and results of this study are discussed under Task 4. One of the main observations was the long runtime required to conduct the study: It took several weeks to complete the study, despite the use of Georgia Tech's HPC environment. This motivated the exploration of methods to speed up the execution of the noise engine.

Multiple options are available for speeding up the execution of the computer code: Applying surrogate modeling and running the code on GPUs were considered as options. In the context of uncertainty propagation, a surrogate would need to take the uncertain parameters as inputs, then output the system-level quantities of interest. Because of the nature of the problem and the sources of uncertainty, building such a surrogate is not immediately possible: it requires multiple steps, which were beyond the scope of this project. Instead, this will be the topic of future research.

In contrast, attempting to run the code on a GPU falls within the scope of this project (under the exploration of emerging computational technologies) and does not require a fundamental change in the computational setup. Moreover, the ability to execute the noise computations on a GPU is fully compatible with other ways of speeding up execution, such as surrogate models, as this would allow training data to be produced more rapidly.

### 2.5.3 Research Objective

The research objective of this subtask was to measure the benefits of running noise computations on a GPU instead of a CPU. This subtask first required that the noise computations be implemented in such a way that they can run on a GPU. Then, two studies were conducted. First, the CPU and GPU runtimes were compared to confirm the benefits brought by the GPU in terms of runtime; because the runtime on a CPU is high, this first study was conducted on relatively small problems. Second, to estimate the ability of GPU-backed computations to handle larger problems, a scaling study was performed, in which the evolution of GPU runtime was estimated as a function of the problem size. Along with runtime, memory requirements also become a challenge for large problems; thus, the memory requirements were estimated.

### 2.5.4 Technical Approach

Dask is a framework for executing parallel processing across many machines, while presenting the user with simple and familiar storage and computational approaches. Internally, Dask includes optimization routines that optimize the flow of code and data across machines. Because Dask's GPU capabilities presented limitations, Google's JAX, another computational framework, was selected to run the noise engine on a GPU. JAX is a cutting-edge computational framework developed at Google that combines XLA, the computational back-end behind TensorFlow, with other tools such as autodiff for automatic differentiation, all while keeping the same simple API as numpy, Python's de facto standard library for numerical computations. JAX allows reuse of the exact same code to run on a GPU instead of a CPU when available.

As discussed previously, runtime and memory use are the two metrics on which we focus to (a) compare CPU and GPU implementation, and (b) study GPU scaling. In our case, runtime is simply measured using wall-clock time: the time instants before and after the computations are recorded, and their difference yields the elapsed wall-clock time. Care was taken to ensure that computations were actually carried out within the measured time interval: Dask, among others, implements the concept of "lazy evaluation," in which expressions may not be actually evaluated until the result is accessed.

Measuring memory use is more challenging, as it depends on the back-end (CPU or GPU). For the CPU, we could not find a way to directly measure the amount of memory used by specific processes. This step is more difficult with Dask because multiple processes may be spawned to manage computations. As a work-around, the total memory use is recorded before computations are started and then continuously updated at regular intervals while the computations are running, and only the maximum system memory use is retained. Memory use is estimated by the difference between maximum memory use during computations and the precomputation system memory use. This estimation assumes that the difference in memory usage can be solely attributed to the noise assessment computations and that other mechanisms, such as memory swapping to disk, do not occur. To avoid swapping, the problem dimensions considered when performing the computations on the CPU were kept relatively small.

Measuring GPU usage was not possible via JAX's built-in functions, as a mismatch was observed between actual GPU memory usage and the value returned by JAX's helper functions. As a consequence, we applied the same approach used for the CPU, except that CUDA-specific commands were issued when polling the GPU memory usage.

As explained previously, the first step of this study was to compare CPU and GPU runtimes. For completeness, the original Dask back-end was also considered in the comparison, both with and without atmospheric absorption improvements (as briefly discussed under Task 4). We varied the problem size by varying the resolution of the square analysis grid. CPU runs were executed locally on a PC equipped with an Intel Core (i7-9700 CPU and 16 GB of RAM). GPU runs were executed on nodes of Georgia Tech's PACE (Partnership for an Advanced Computing Environment) cluster equipped with a Tesla V100 (32 GB) GPU.

In the second step, we focused on GPU runs only. Multiple dimensions of the problem were varied to obtain a wide range of problem size. We varied the resolution, as in the first part of the study, and the number and maximum length of the trajectories. In the current implementation, trajectories are handled sequentially, while for a given trajectory, the complete grid as well as all of the trajectory's time steps are simultaneously computed. Thus, we expect all of those dimensions to influence runtime, while memory use should not be affected by the number of trajectories, since they are treated sequentially.

To increase the maximum allowable GPU memory use, and therefore the maximum size of the problems under consideration, a dual-GPU implementation was developed. This dual-GPU implementation took advantage of the fact that Georgia Tech's PACE cluster offers some nodes with two GPUs, totaling 64 GB of GPU memory. In the current implementation, the analysis grid on which noise exposure levels are computed is split into two regions: the first half is processed on one GPU while the second half is processed on the second GPU. Because all analysis points are independent, this approach does not introduce communication overhead.

### 2.5.5 Results

Figure 15 depicts the evolution of the runtime duration in seconds as a function of the analysis grid resolution for four cases: the original Dask implementation with and without atmospheric absorption, JAX (CPU), and JAX running on a GPU. Here, a log scale is used for the duration on the y-axis. We observe that JAX on a GPU is faster than CPU-based computations by approximately two orders of magnitude: running the same code on a GPU instead of a CPU allows a 100-fold speed-up. This gain is significant, especially when considering the many cases that need to be run as part of an uncertain propagation study using Monte Carlo simulations.

The differences between the different CPU implementations can be justified as follows: First, neglecting atmospheric absorption consistently reduces runtime across the considered grid resolutions compared with the Dask version of the noise engine that accounts for atmospheric absorption. We found that the CPU JAX version of the code initially runs faster than its Dask counterpart for small resolutions, but appears to match the Dask implementation for higher resolutions. We hypothesize that this result is due to the overhead introduced by Dask when setting up its scheduler and workers: while this overhead is significant for low-resolution grids that can be rapidly analyzed, it becomes negligible compared with the actual cost of computations once the resolution increases sufficiently.

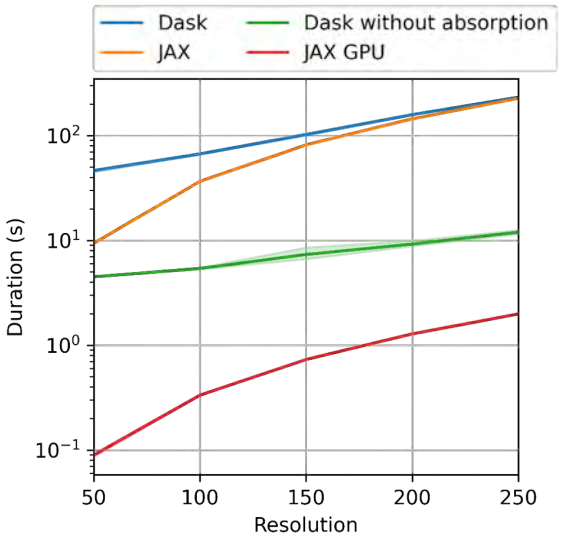


**Figure 15.** Runtime comparison between different implementations of the noise model. GPU: graphics processing unit.

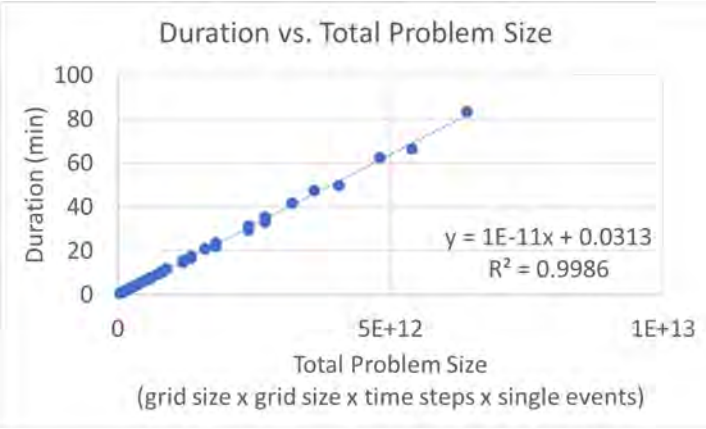


**Figure 16.** Runtime scaling of the graphics processing unit (GPU) implementation.

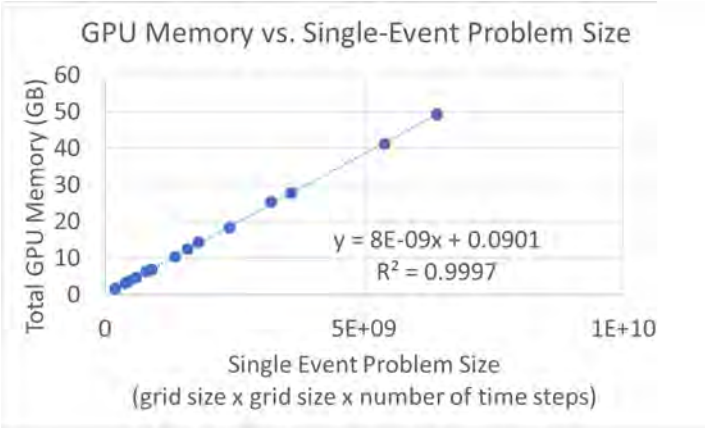


**Figure 17.** Memory-use scaling of the graphics processing unit (GPU) implementation.

Figures 16 and 17 depict (a) the evolution of the runtime duration as a function of the total problem size, and (b) GPU memory as a function of the single-event problem size, respectively. As discussed previously, while the total problem size encompasses all dimensions of the problem, including the number of trajectories (referred to as single events here), the single-event problem size corresponds to the problem size for a given trajectory. As expected, the duration depends on the total problem size, whereas the total GPU memory depends on the single-event problem size because trajectories are processed sequentially by the current implementation. In both cases, linear regression confirms a linear dependence.

These graphs can be used to estimate the runtime and GPU memory use when running a new case: the total and single-event problem sizes can be computed from the individual problem dimensions, and the linear formulas provided here can be used to obtain a runtime and memory use estimate. In practical applications, this information can be used to estimate the total duration of, for example, a Monte Carlo simulation, or to ensure that the memory use will not exceed the available GPU memory.

### 2.5.6 Conclusions

The two studies conducted in this section confirm the benefits brought by GPU computation. Thanks to the JAX framework, the same code can be used on a CPU for local development and testing and then on a GPU when additional speed is needed. These benefits are substantial: a 100-fold increase in speed was observed when the code was run on a GPU compared with that run on a CPU. In the Task 4 Section, we will see that this enables us to run a Monte Carlo simulation in a couple of hours, when it would have taken weeks if run on a CPU.

## 2.6 Cloud-based Computations on AWS

### 2.6.1 Context and Motivation

Among emerging technologies suitable for use in the development of the UAS noise-assessment tool, cloud-based options were retained because they enable a flexible selection of the amount of computational resources allocated to solving a problem. For example, when using Dask paired with AWS Elastic Compute Cloud (EC2), the user can choose the number and characteristics of workers across which computations are distributed: each worker will be executed within a dynamically spawned AWS instance with its own resources, and individual instance resources can be selected based on AWS's offerings. This flexibility allows us to tackle a wide spectrum of problem sizes, from the small problems encountered, for example, when developing and debugging the noise engine to the larger problems encountered when running a full-fledged noise assessment on a large urban area.

### 2.6.2 Problem Definition and Research Objective

While executing the noise engine on AWS EC2 is made easier by using Dask as a computational framework, the level of maturity of these frameworks still does not allow for a plug-and-play experience. Multiple hurdles had to be overcome in order to successfully run noise computations in the cloud. In this section, we document the required steps to ease the process for future users and developers of the tool.

Because ASDL does not have specific resources allocated to AWS EC2, this development effort was conducted using Amazon's free-tier instances, which have limited computational power and system memory (a single virtual CPU and 1 GB of RAM). Therefore, it was not possible to demonstrate the ability to run large problems in the cloud; instead, the objective was to develop a proof-of-concept end-to-end workflow using a simplistic scenario (small grid and very few flights). Scaling to larger problems should not raise additional technical hurdles, but should simply require the allocation of additional resources, which can be easily done by the user via simple configuration parameters.

### 2.6.3 Technical Details

The content of this section is very detailed; at the time of implementation, such details are needed in order to benefit from the advantages of cloud-based computations.

#### *Initial Setup Steps*

The following steps can be followed to set up AWS. Depending on the organizational setup, some steps may be skipped or require different actions. For example, instead of creating a root account and using it to create a lower-privilege account, a lower-privilege account may need to be directly requested from the administrators of the organizational AWS EC2 account.

1. If not already available, create an AWS root account.
2. Create a lower-privilege account. For the security policy, allow programmatic access to EC2 only, "AmazonEC2FullAccess." More details on how to create a user can be found in AWS' documentation.
3. Install and configure AWS CLI on the client machine. Use "pip install awscli" to install the CLI tool, followed by "aws configure" to proceed with the initial configuration. This step requires the user's AWS access key ID as well as their secret access key.
4. Install the dask_cloudprovider library for AWS using "pip install dask_cloudprovider[aws]."
5. The cryptography package is also needed and can be installed via "pip install cryptography."

More details are available in Dask's documentation.

#### *Disable TLS Certificates*

Dask automatically provisions AWS EC2 instances by sending a script via the AWS API. The size of this script is limited to 16 kB. However, Dask's configuration often exceeds 16 kB, mainly due to the transmission of self-signed TLS certificates used to secure cluster communications. This is a known Dask limitation discussed in the project's issue tracker:

- https://github.com/dask/dask-cloudprovider/issues/249
- https://github.com/dask/distributed/pull/4465

The proposed solution is to not use TLS certificates. This is achieved by instantiating the Dask cluster by setting the security keyword argument to False:

```
cluster = EC2Cluster(env_vars=credentials, security=False)
```

As a result, for example, the Dask dashboard is not available through https, only http. Additional steps can be taken to properly secure the dashboard if served from a publicly accessible server.

More details on the user-provided setup scripts for creating AWS EC2 instances can be found in AWS EC2's documentation:

- https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-metadata.html
- https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instancedata-add-user-data.html

#### *Python and Package Versions*

Other errors may arise when using Python 3.9/3.10, and the solution is to use Python version 3.8 or earlier. The relevant tracked issue is located at https://github.com/dask/dask-cloudprovider/issues/359.

We must ensure that package versions match between client and EC2 instances that are automatically set up by Dask. Dask issues a warning when versions mismatch. This is important, because class instances created on EC2 using one version are serialized and sent back to the client, which may not be able to deserialize them.

***Debugging the AWS EC2 Workers***
Debugging is difficult due to the fact that the workers are ephemeral EC2 instances. Workers are automatically terminated by Dask when an error is encountered. The only output that can be easily accessed after EC2 instances are terminated is the system log. Dask can be configured to log to the system log. The steps to achieve this are documented in the Dask and Python logging library documentations:

- https://docs.dask.org/en/stable/how-to/debug.html?highlight=logging#logs
- https://docs.Python.org/3/library/logging.handlers.html

***Manually Copying Scripts to Workers and Manually Configuring Workers' Python Environments***
All of the additional scripts called from the main script used to launch the Dask instance (such as library files not installed through pip) need to be manually copied to the workers (EC2 instances) using client.upload_file(). Likewise, Python environments local to the workers also need to be manually set up, using Dask's PipInstall "worker plug-in."

### 2.6.4 Conclusions
A proof of concept was developed to illustrate how the noise-assessment tool can run in the cloud. Because of the limited resources available to the team, a problem of very limited size was considered. This effort allowed us to gauge the ease of using Dask's cloud functionalities. Although the capability to run a computation with minimal changes to the initial Dask implementation exists, the experience is not yet seamless. Hopefully, the documentation provided here will help streamline the use of Dask in the cloud.

# Task 3 - Collaboration with the UAS Computation Module Development Team
Georgia Institute of Technology

## Task 3 Contents


## 3.1 MSU Collaboration
### 3.1.1 Objective
In this task, we collaborated with the UAS computation module development team at MSU to explore ways in which the teams can effectively exchange data and ideas.

### 3.1.2 Research Approach
The ASCENT9 team met with the team working on the eCommerce project at MSU on a biweekly basis. Led by Dr. Adrian Sescu, this team provided demand data and a data generator to create random UAS paths. The teams discussed the simulation of noise footprints from a notional UAS delivery network in the Memphis area. The ASCENT9 team shared an early version of the noise engine calculation with the MSU team.

The eCommerce project revolved around emerging UAS networks and their implications in national airspace system integration. The project's case study is an analysis of an Amazon UAS delivery network using ground support. The MSU team collected data for warehouses in the greater Memphis area along with the residential addresses served by these warehouses. Trucks were placed in the area to reduce the flight time of the UASs and to help with last-mile delivery. These warehouses are shown in Figure 18. Multiple scenarios were considered in this study:

- 8 drones per warehouse and 4 drones per truck (1,132 drones)
- 12 drones per warehouse and 6 drones per truck (1,698 drones)
- 16 drones per warehouse and 8 drones per truck (2,264 drones)
- 24 drones per warehouse and 12 drones per truck (3,396 drones)
- 32 drones per warehouse and 16 drones per truck (4,528 drones)
- 55 drones per warehouse and 50 drones per truck (12,305 drones)

The ASCENT9 team shared an early version of the noise engine developed under Task 4 with the MSU team, who verified that they were able to run the noise engine on their systems.

The ASCENT9 team used the first scenario to test the noise engine with variable grid precision. These trajectories are shown in Figures 18 and 19. The trajectories span an area of approximately 40 miles, with each trajectory's length varying between 3,000 and 8,000 ft.

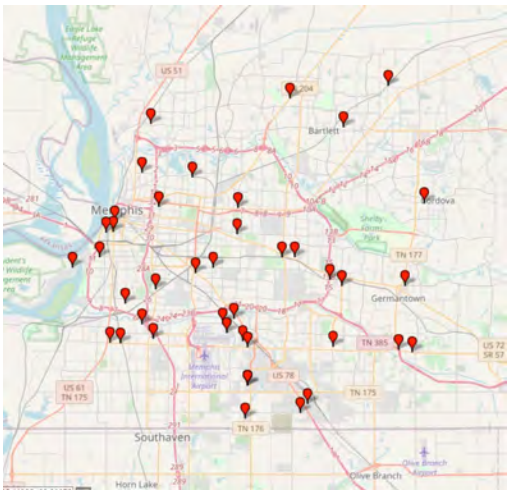


**Figure 18.** Warehouses in the Memphis, TN area.

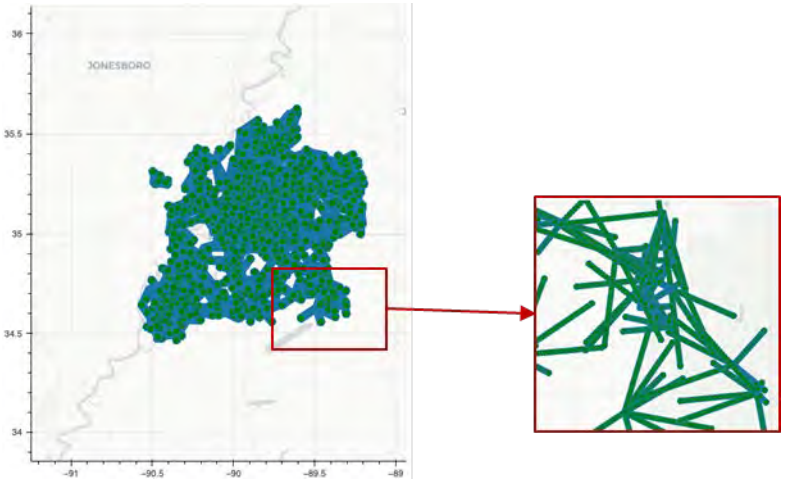


**Figure 19.** Random trajectories provided by Mississippi State University.

## 3.2 Volpe Collaboration

In addition to collaborating with MSU, the ASCENT9 team collaborated with the Volpe Research Center to acquire national transportation noise data. These data consist of combined gridded road, aviation, and railroad noise for the entire United States provided in A-weighted 24-hr exposure levels. These data are used as background noise that is added to the noise calculated by the engine module. A cropped overview of these data for the greater Memphis area is shown in Figure 20.

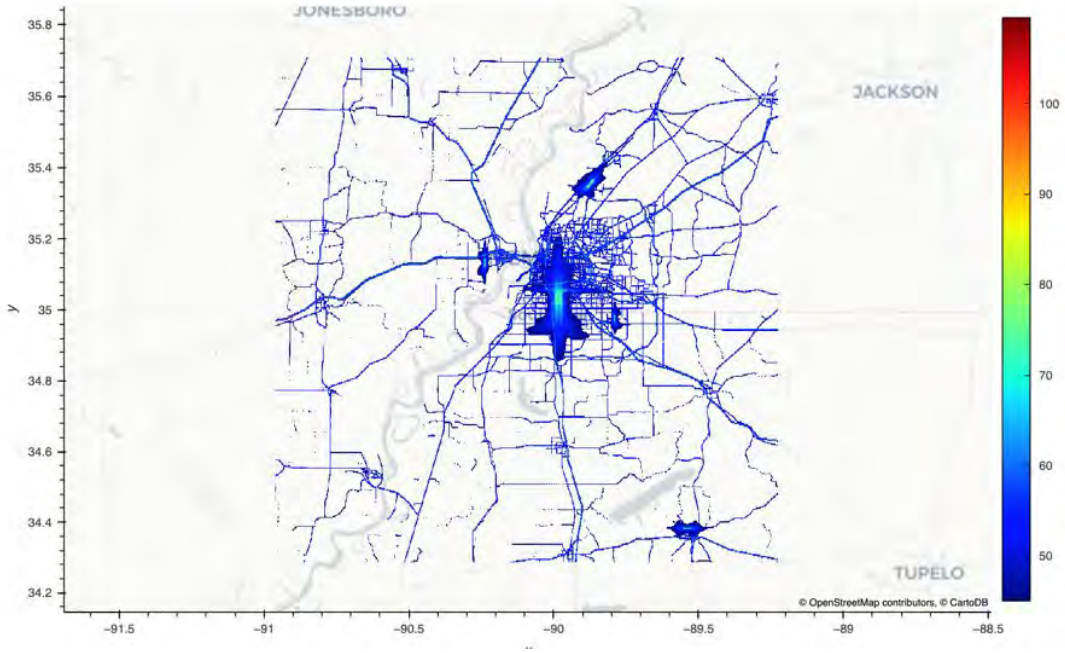


**Figure 20.** National transportation noise data for the greater Memphis area.

## 3.3 PACE Collaboration

In addition to these external collaborations, this research was also supported in part through research cyber-infrastructure resources and services provided by PACE at the Georgia Institute of Technology. This computing environment consists of a large computing cluster that was used to develop and test the noise engine under Task 4. This cluster was also used to conduct experiments and help tune various parameters and aspects of how the noise engine is executed in parallel. For example, parameters range from the number of computing nodes to the amount of memory per node and the number of parallel processes per node.

## 3.4 Improvements to MSU's Trajectory-generation Code

After the MSU collaboration ended, the initial trajectory-generation code was reworked. In addition to introducing a more efficient implementation and increased flexibility, the code was broken down into multiple logical steps that relate to different phases of the workflow, as presented in the Task 4 Section.

Prior to the proper noise computations, the first step consists of creating tuples of staging locations, delivery locations, and vehicles. In general, these locations are the start and end points of a flight. For example, if different use cases are considered, such as for an e-taxi, these locations would map to pick-up and drop-off locations. The generation of these so-called pairings is dictated by the concept of operations, and these pairings are then used as input for the actual noise assessment. Currently, the implementation of this step is simple because only straight trajectories are considered, with either hover, cruise climb, or cruise flight segments. This logic could be made more complex in the future to accommodate new concepts of operations. Here, this logic is separated because it is independent from the noise computations and can therefore be developed in parallel, as long as the data interface between these two steps of the workflow is properly maintained.

In the second step, the definitions of the flights, or flight segments, are discretized in time. We have included this step as part of the preliminary analysis because the need for time discretization is purely an artifact of the current analysis method. If another analysis method were to directly take in flight segments as inputs instead of vehicle locations, then the flight segments would not need to be discretized.

This split also has the advantage of allowing for a more compact representation of a scenario; that is, a set of daily flights.

Among other improvements, the vehicle attributes are now provided externally and stored in a CSV file instead of being hard-coded.

# Task 4 - Noise-computation Engine Integration

Georgia Institute of Technology

## Task 4 Contents



## 4.1 Task Overview

The motivation for developing a noise-assessment tool specific to UASs was presented in the previous sections, and the previous tasks aimed at investigating the building blocks for this tool. Once promising technologies have been identified for the application components, they must be integrated within a coherent and easy-to-use tool, and this is the purpose of Task 4.

The following sections are organized chronologically: An initial implementation was developed and used to conduct an initial benchmark study and an initial Monte Carlo study. Then, a consequent refactor of the code was undertaken to improve both the internal code structure and the user interface. The refactor was intended to make it easier to work with and extend the codebase. This latest iteration was used to study the effect of interactions between trajectories. Finally, a new uncertainty propagation study is discussed, in which we took advantage of the speed-up brought by the GPU implementation discussed and studied in Task 2.

## 4.2 Initial Noise Computation Engine Implementation

The investigation conducted in Task 2 led to the identification of adequate tools to build a high-performance, interactive, GIS-based noise module for UASs. A Python web application was set to be built with the ability to run either locally or in a distributed setting provided by the HPC infrastructure of Georgia Tech PACE. As Python was already determined to be the programming language for this module, different libraries enabling parallel matrix computation and large interactive visualization were explored. The selection process resulted in four libraries, as shown in Figure 21.

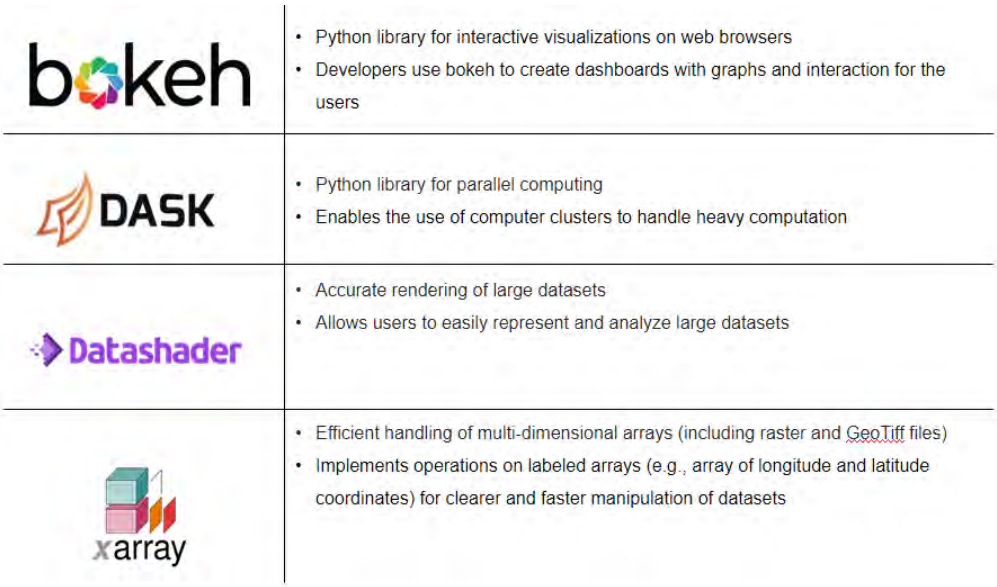


**Figure 21.** Enabling capabilities for the unmanned aircraft system (UAS) noise engine prototype.

Before showcasing the architecture of the web application, we discuss the structure of the Python object for the grid. Noise metrics are built on the distances between the grid and the path of the noise source. In other words, for each point in the path, its distance to every point in the grid must be calculated. This information can be stored as a 3D matrix, where the third dimension matches the number of points in the path. A notional sketch of this structure is shown in Figure 22. This choice benefits from the highly optimized methods of numpy, a Python library for multi-dimensional arrays.

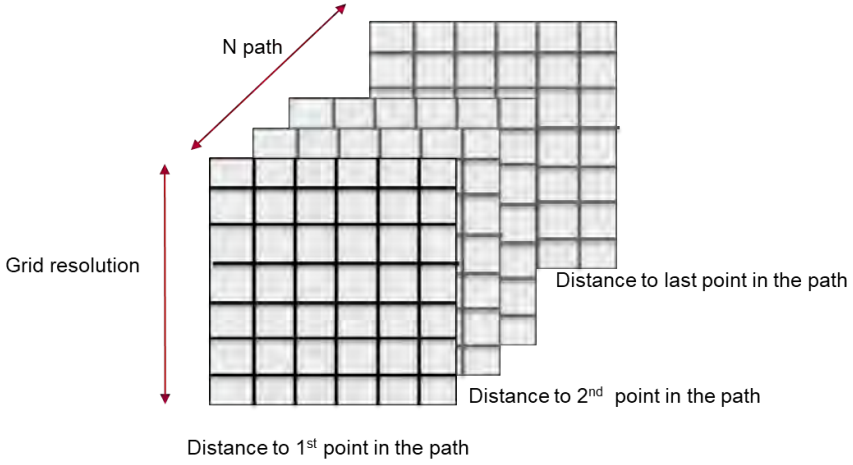


**Figure 22.** Notional structure of the noise module object.

The UAS prototype must demonstrate the calculation and visualization of two types of noise metrics: peak metrics and exposure metrics. The individual steps to calculate each metric are presented in Figure 23.

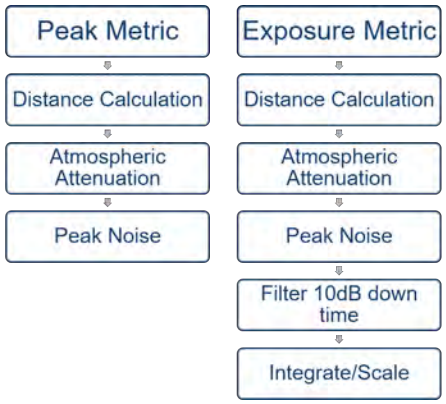


**Figure 23.** Steps for calculating peak and exposure noise metrics.

The parallel execution of the noise engine is conducted using the Dask library, with the following implementation steps:

1. Define computational steps as operations on generic datasets
2. Prepare datasets
3. Define computational resources
4. Launch the dynamic scheduler and map/apply operations on the datasets
5. Collect results

The computational resources are defined by the hardware available for parallel computation, which is characterized by the number of cores or workers and the available memory per core. In addition to allowing parallel computations on single machines, Dask supports cluster schedulers such as PBS and Slurm and is supported by AWS.

The dynamic scheduler is one of the most powerful features of Dask as it handles data partitioning and calculations without much user interference. This scheduler creates an optimized directed acyclic task graph to transfer data and apply computations using the given resources. An example of such a task graph is shown in Figure 24. This graph corresponds to a peak metric event calculation using 10 workers.

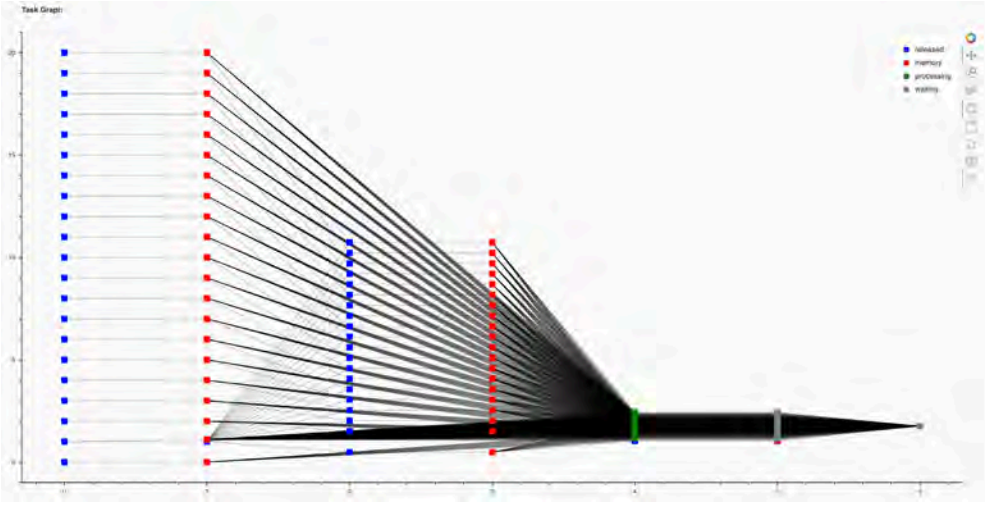

**Figure 24.** Task graph generated by Dask's dynamic scheduler.

The generic implementation steps on Dask are illustrated in Figure 25, where the client refers to the web browser used to visualize the noise contours.

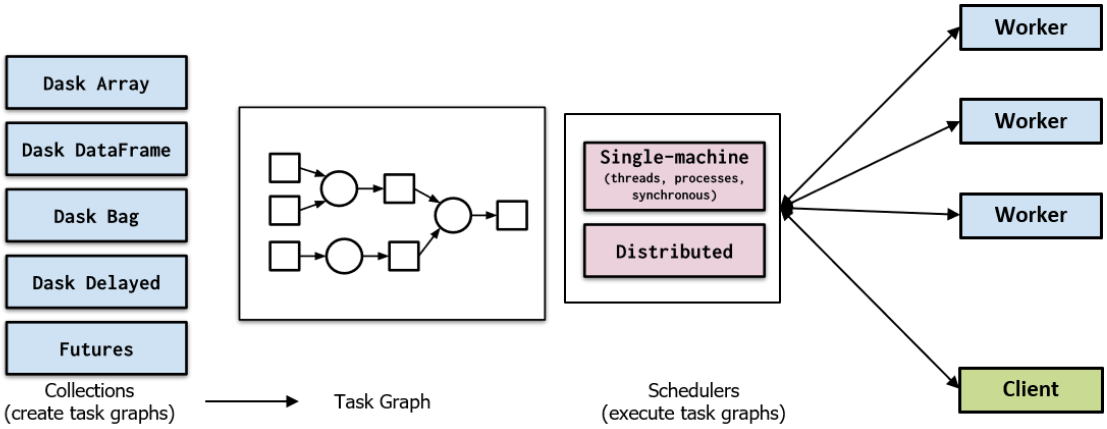


**Figure 25.** Implementation steps in Dask.

To visualize these contours on the browser, the Dask data objects need to undergo packaging operations using xarray and datashader. There is a limitation on the number of points a browser can support; therefore, datashader is used to allow the data to be sampled and visualized in a meaningful way. Datashader objects are integrated in Bokeh, but they do not support Dask arrays. Xarray was used to wrap the Dask objects for use within datashader. This data pipeline is illustrated in Figure 26.

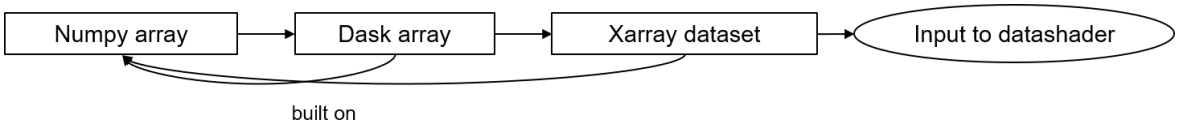


**Figure 26.** The data pipeline from Dask to Bokeh.

The overall architecture of the UAS noise calculation prototype is displayed in Figure 27. The noise contours are calculated and stored on the PACE distributed cluster. For visualization, Bokeh requests a portion of the data that is aggregated and projected using datashader. This step requires continuous communication between the Dask scheduler and the workers writing the data that have been bypassed to files. Alternatively, a central file could be created to collect the results. However, this comes with a high communication cost that must be considered. The data are accessible from the Bokeh server through secure ssh tunneling to the PACE interface. This is a major advantage of web applications over desktop applications, as it provides broader cross-platform access for clients.

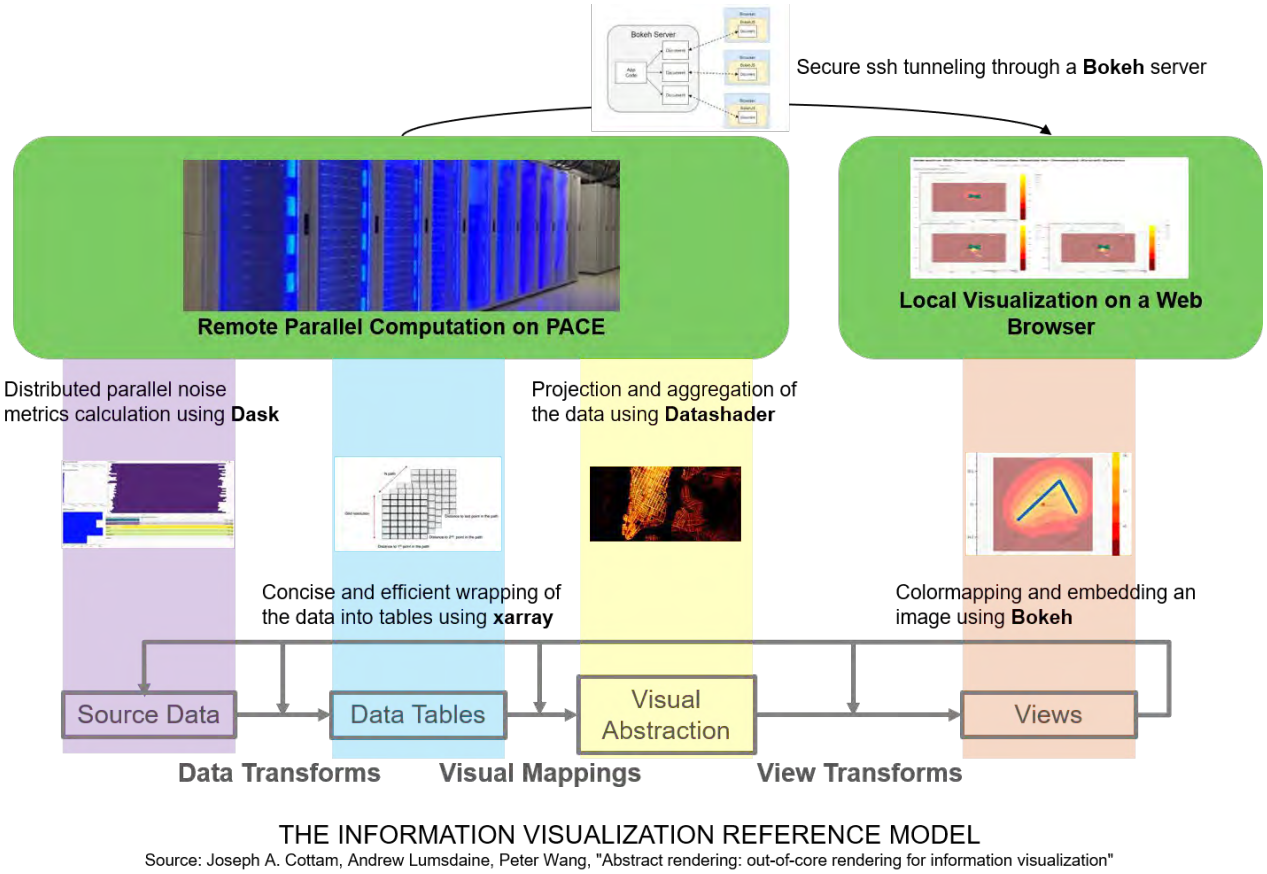


**Figure 27.** Overview of the noise module. PACE: Partnership for an Advanced Computing Environment.

## 4.3 Initial Benchmark Demonstration

This benchmark study aimed to simulate the noise footprint from a notional UAS delivery network in the greater Memphis area. In this study, 40 warehouses serving approximately 30,000 residential addresses were considered. Trucks that serve as UAS staging platforms are positioned near some neighborhoods, which reduces UAS range requirements and delivery times. For this study, eight UASs per warehouse were considered, with four UASs per truck and a total of 1,132 total flights. The paths for these flights are shown in Figure 28.

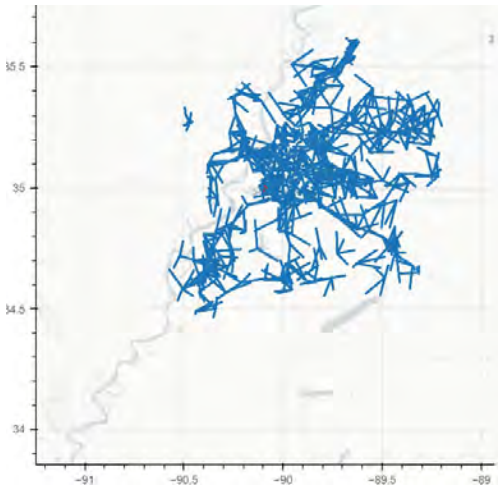


**Figure 28.** Flight paths used in the benchmark study.

The national transportation noise map was used as background to supplement the engine's computations. The contours of this background noise are shown in Figure 29. The cumulative $L_{Aeq}$ noise contours generated uniquely from UAS activities are displayed in Figures 30 and 31. The effect of UAS activity on the existing noise in the greater Memphis area is shown in Figure 32.

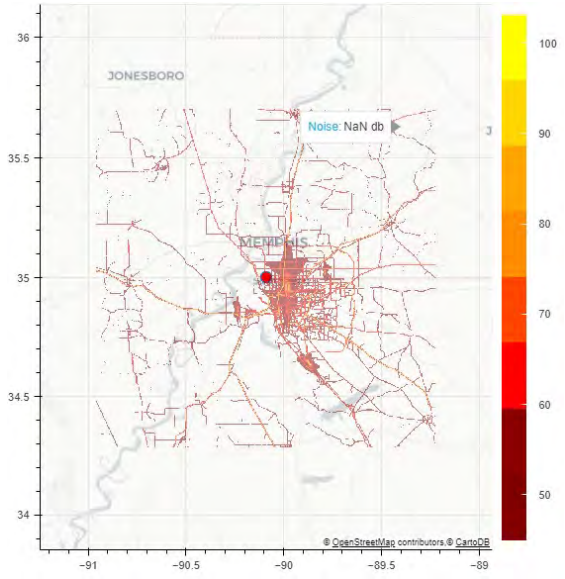


**Figure 29.** National transportation noise map of the greater Memphis area.

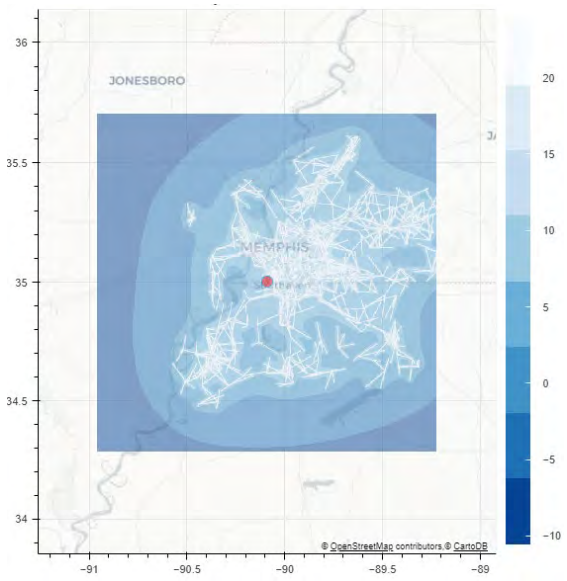


**Figure 30.** Computed unmanned aircraft system (UAS) noise ($L_{Aeq,24hr}$).

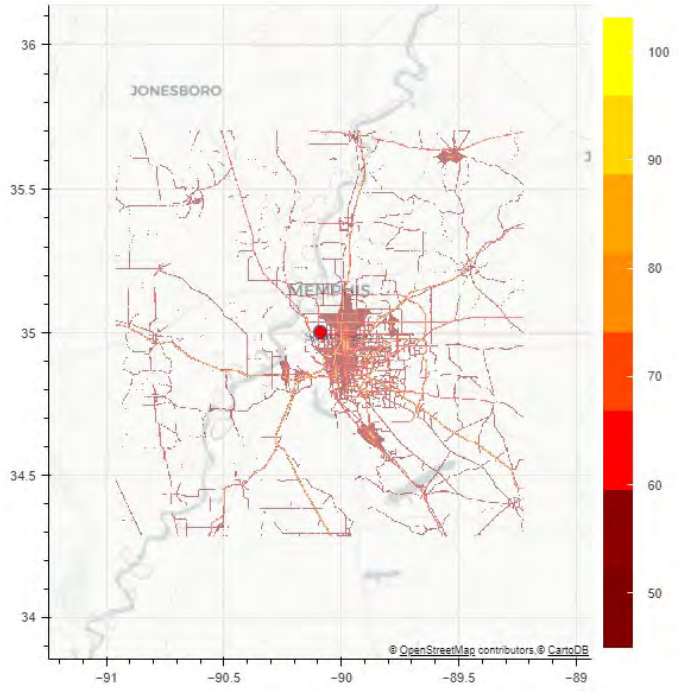


**Figure 31.** Combined noise ($L_{Aeq,24hr}$).

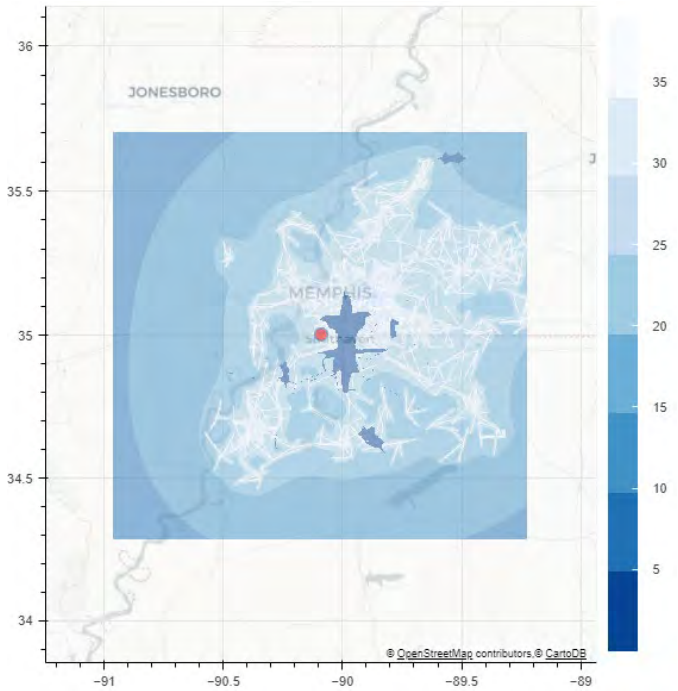


**Figure 32.** Change in $L_{Aeq,24hr}$.

The $L_{A,max}$ value for UAS noise with the interactive demo is illustrated in Figures 33 and 34. This figure indicates the potential difference in noise impacts across areas with high noise exposure levels compared with areas that currently have limited noise-exposure levels. A large difference is found between exposure and peak metrics. The interaction can be better understood by including the background noise.

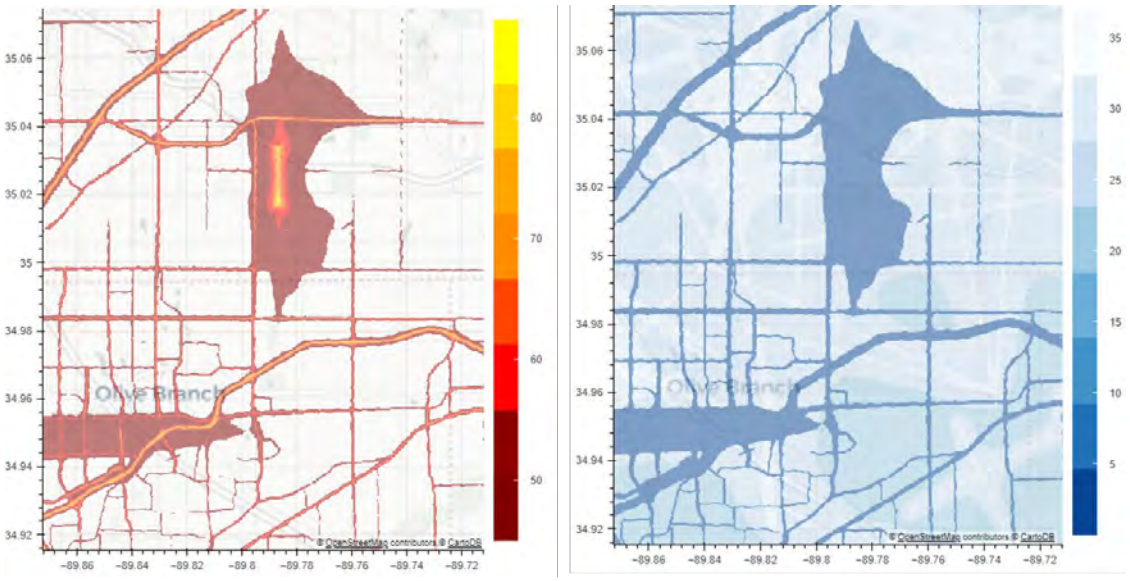


**Figure 33.** Combined noise $L_{Aeq,24h}$ (left) and change in $L_{Aeq,24h}$ (right).

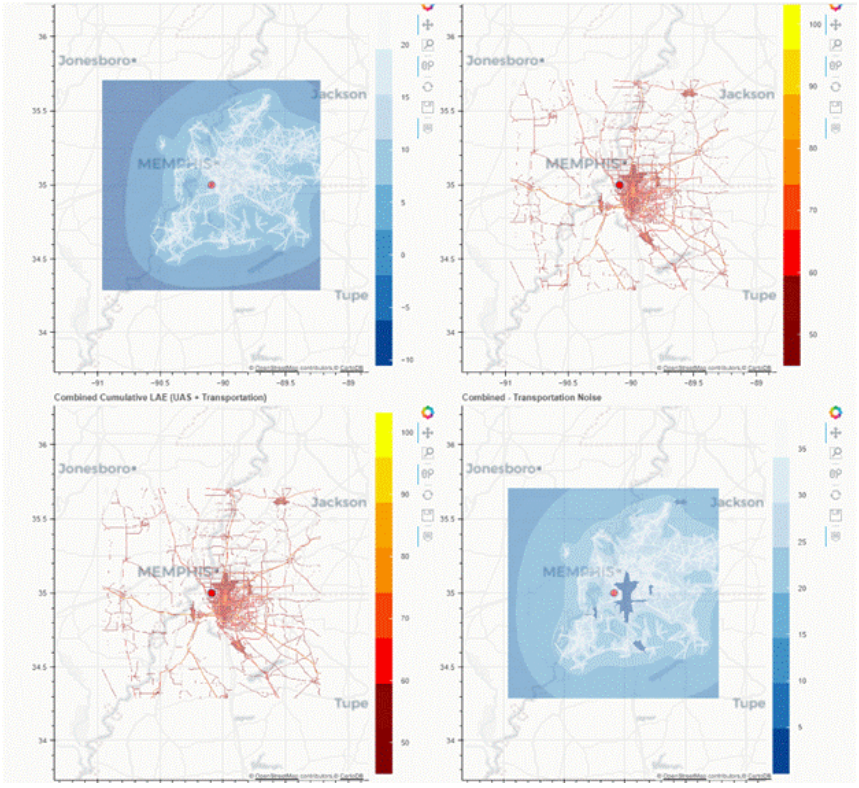


**Figure 34.** Interactive demo.

## 4.4 Initial Monte Carlo Study

Because UAS operations are stochastic in nature, individual flight trajectories for each day depend on daily orders and demand, as shown in Figure 35. In some cases, staging locations can also vary. The operator strategy applied to the trajectory planning can also include noise dispersion and altitude constraints to minimize the noise. The annual average day metrics are not capable of capturing daily changes.

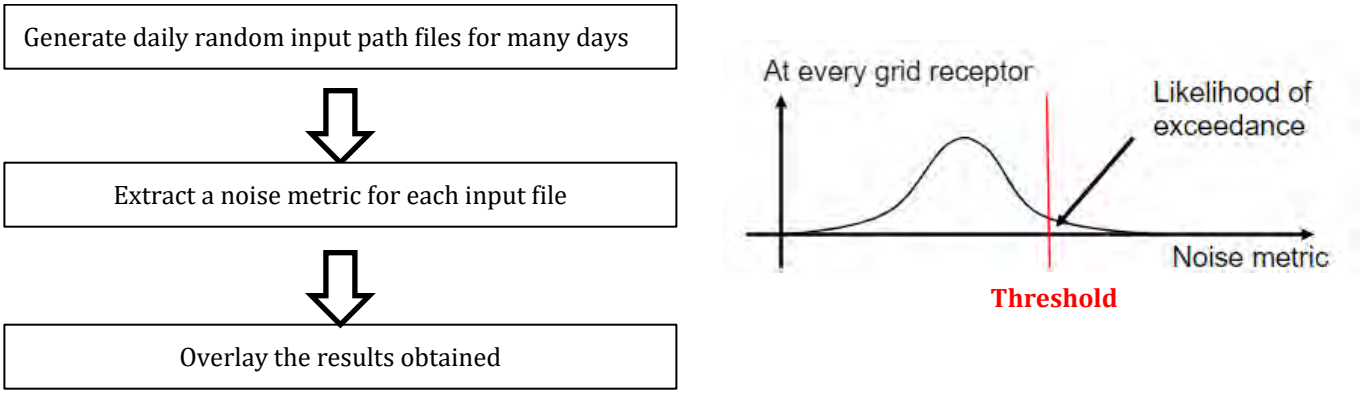


**Figure 35.** Notional workflow for a probabilistic approach to noise assessment.

A question arises: What is the likelihood of exceeding some threshold on any given day and how many locations will exceed this threshold? We first attempted to answer this question using a Monte Carlo simulation. However, this process is

computationally expensive. The goal of this probabilistic assessment is to obtain the likelihood of exceedance contour. The first attempt included 100 daily deliveries for 3,800 days on a coarse grid (250k points). The CPU time included 10,000 simulated days and resulted in collecting multiple noise metrics at the same time. To a first-order approximation, the delivery noise distribution was based on the address/population distribution.

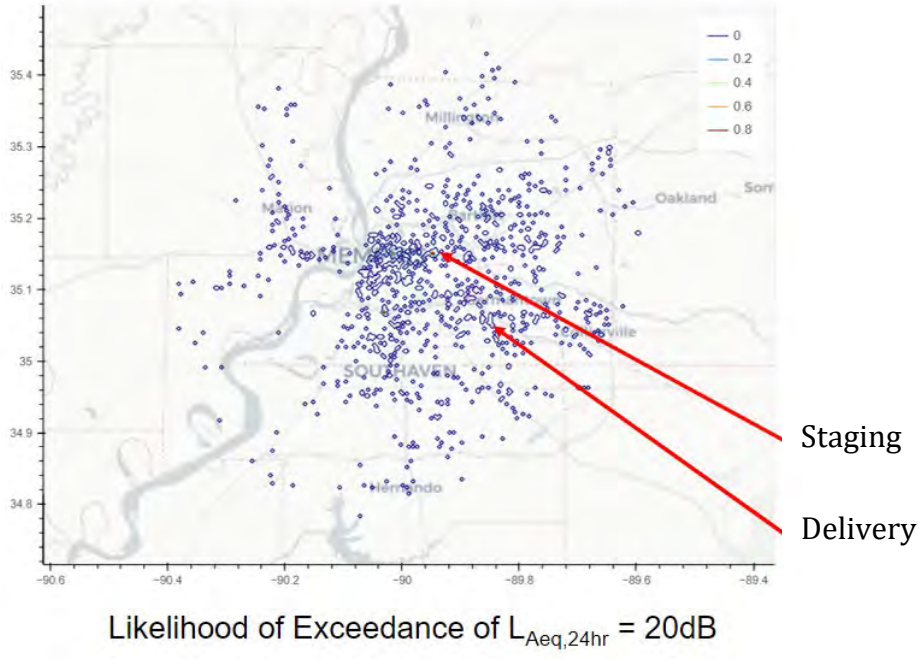


**Figure 36.** Likelihood of exceedance for $L_{Aeq,24hr}$ = 20 dB.

The choice of metric and threshold has a significant impact on the observed results, as demonstrated in Figure 37, which shows the likelihood of exceedance when $L_{Amax}$ is increased from 20 to 50 dB.

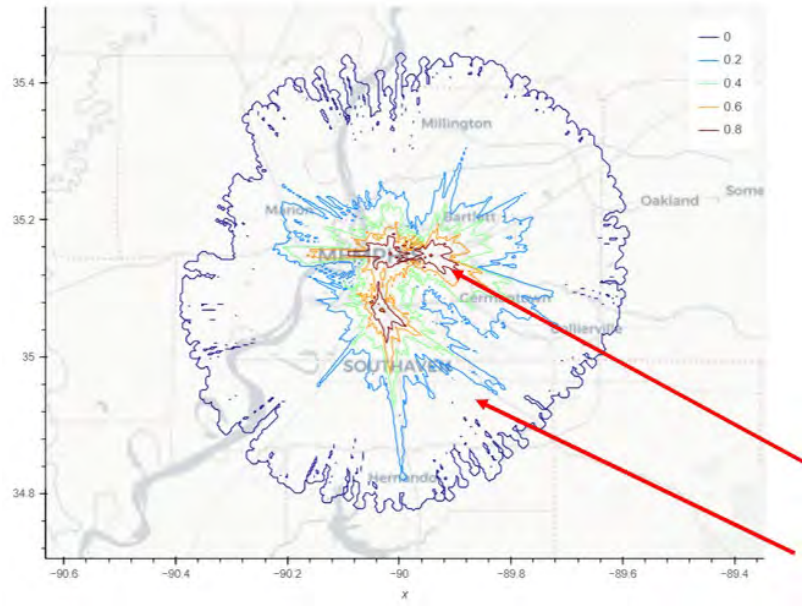


**Figure 37.** Likelihood of exceedance for $L_{Amax}$ = 50 dB.

## 4.5 Implementation of the SAE5534 Atmospheric Absorption Model

The team implemented the atmospheric absorption losses defined in SAE ARP5543. This implementation provides a more realistic method of atmospheric sound absorption than the very simplified method used at the beginning of this project. This approach also builds on earlier standards, such as ARP866A, and allows the modeling of noise absorption to be sensitive to humidity and temperature, as expected. The current implementation works as a function that replaces the simplistic distance scaling. This function is also included in the Dask implementation as a function that utilizes parallel execution. The JAX implementation serves as the basis of a GPU shader function. While there is some penalty in the execution speed in both cases, the current implementation appears to work reasonably well. The team also worked to ensure accuracy in the implementation by comparing the implementation's output to the reference data supplied in ARP5543. In addition, the team compared the current implementation with AEDT's implementation. Both comparisons yielded only minor differences attributable to floating point precision and rounding differences.

## 4.6 Workflow Definition and Code Refactor

### 4.6.1 Workflow Definition

The analysis workflow was formalized in order to drive the development of the new GUI. The resulting workflow is depicted in Figure 38. In the first step, all of the inputs to the analysis are specified and/or loaded. This step includes defining the analysis area, generating or retrieving trajectories, and loading the background transportation noise. In the second step, analysis settings are provided by the user, and the proper noise assessment is executed. Finally, the results are visualized.

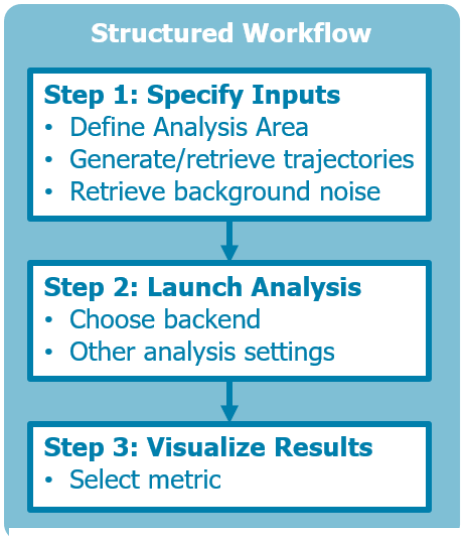


**Figure 38.** New structured workflow.

### 4.6.2 Code Refactor

A code refactor was undertaken to make the codebase more modular. By modular, we mean, for example, logically splitting the code between the GUI-related parts and the analysis-related parts. Within the part of the code devoted to the GUI, modularity means defining clear interfaces between components. For example, the map displayed in the main noise-assessment tool can now easily be reused within Jupyter notebooks in the context of a stand-alone study.

Many new features were developed, including the ability to display more operational scenario details (staging and delivery locations, flights, etc.), the inclusion of multiple input panels allowing the user to specify analysis inputs (instead of hard-coded values), and an integrated display of all output metrics on a single map.

As mentioned in the Task 2 section, the trajectory generation was also enhanced, with more flexibility in defining new vehicles and their noise characteristics.

### 4.6.3 Upgraded GUI

The upgraded GUI is shown in Figure 39. Similar to most GIS software, the map occupies most of the screen. On the left-hand side, a tab-divided panel contains all of the controls necessary to follow the workflow defined in the previous section.

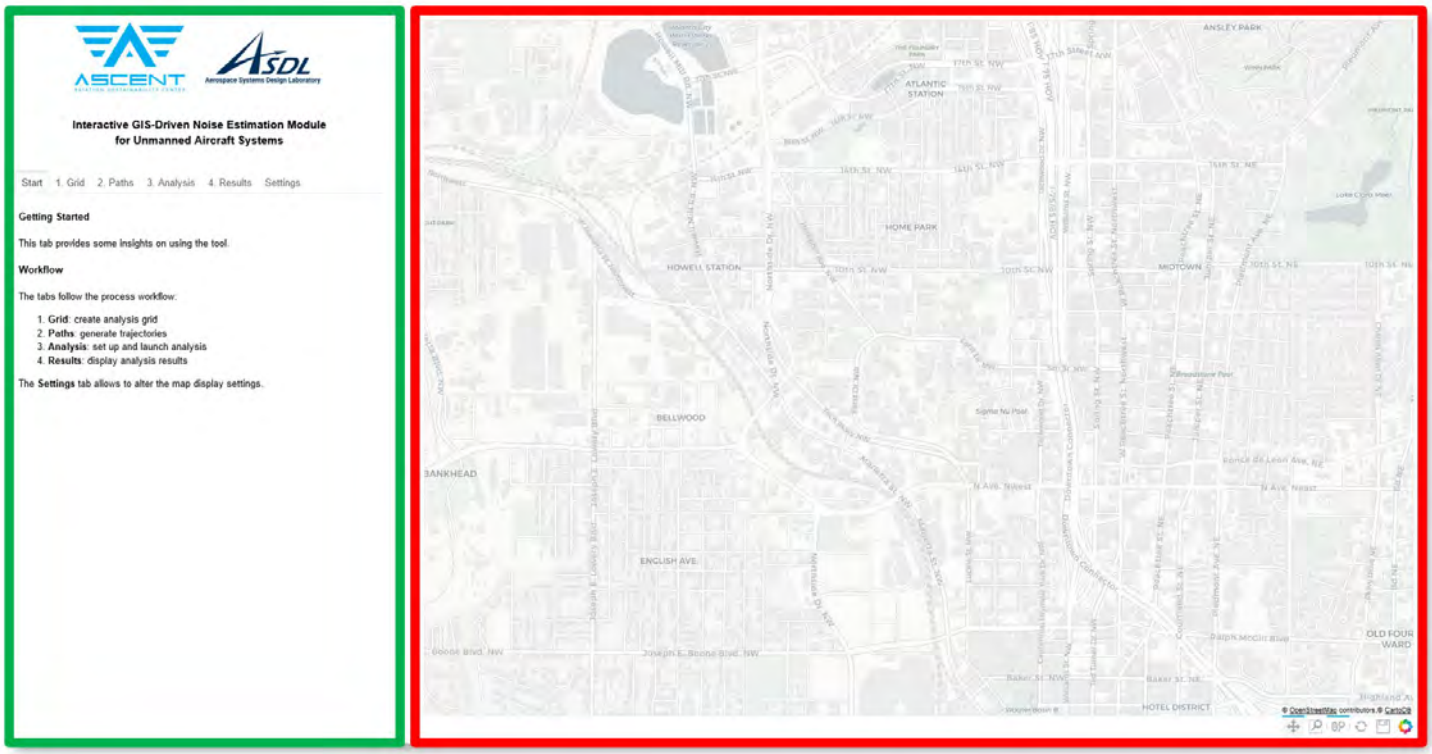


**Figure 39a.** Annotated screenshot of the new graphic user interface. The left-hand side (shown in a green box for emphasis) displays the control panel, featuring tab-based navigation. The visualization map occupies the remaining space on the right-hand side (shown in a red box for emphasis).

**Interactive GIS-Driven Noise Estimation Module for Unmanned Aircraft Systems**

Start 1. Grid 2. Paths 3. Analysis 4. Results Settings

**Getting Started**

This tab provides some insights on using the tool.

**Workflow**

The tabs follow the process workflow:

1. **Grid**: create analysis grid
2. **Paths**: generate trajectories
3. **Analysis**: set up and launch analysis
4. **Results**: display analysis results

The **Settings** tab allows to alter the map display settings.

**Figure 39b.** Screenshot of the control panel of the new graphic user interface.

## 4.7 Study of Interactions Between Trajectories

### 4.7.1 Motivation and Objective

In the current implementation of the noise engine, trajectories are processed separately. Therefore, the combined impact of two flights at the same geographical location (i.e., the same virtual microphone) is not considered. To assess the validity of this simplification in the context of the operational scenarios under consideration, we conducted a study to quantify the discrepancy introduced by this approach.

Instead of using a full operational scenario to characterize the discrepancy, we focused on a smaller test case and proceeded in two steps. First, using two vehicles, we illustrated the error that results when the two vehicles are considered independently. We then generalized this error to a larger number of vehicles; because of the simple noise model being used, the error could be computed analytically as a function of the vehicle number. This first part of the study allowed us to identify the types of situations in which the error was significant; namely, when multiple vehicles were simultaneously close to a microphone location. Then, in a second step, we sought to determine the frequency at which such situations occurred in the considered operational scenarios.

More specifically, we aimed to assess the impact of computing the $L_{A,max}$ metric by considering all vehicles independently, rather than summing the individual sound intensities of nearby vehicles at every microphone location. We focused on the $L_{A,max}$ metric because this is the only metric of interest that is affected by an independent treatment of trajectories. The other metrics result from a time integration, making the concurrency of events irrelevant to their final computed value.

The following simplifications were made compared with the usual noise assessment setup:

- The analysis grid consists of a single microphone;
- All vehicles had the same noise level at 100 ft (65 dBA);
- All vehicles flew at the same altitude/z-coordinate (100 ft); and
- Each vehicle was located at a set distance from the microphone in the horizontal x-y plane.

### 4.7.2 Two Vehicles with Varying Distance from the Microphone

We considered two vehicles and varied their respective distances, $d_1$ and $d_2$, to the microphone in the x-y plane (same altitude, same source noise level). As shown in Figure 40, the difference in the two metrics approaches zero when the vehicles are far from each other (one has a significantly higher contribution than the other; therefore, taking the maximum of the two noise levels becomes a good approximation). When the two vehicles are located the same distance from the microphone, the intensity is underestimated by 1/2 when the maximum is used instead of the sum of intensities, and accordingly, the difference between the two noise level metrics is $10\log(2) = 3.01$.

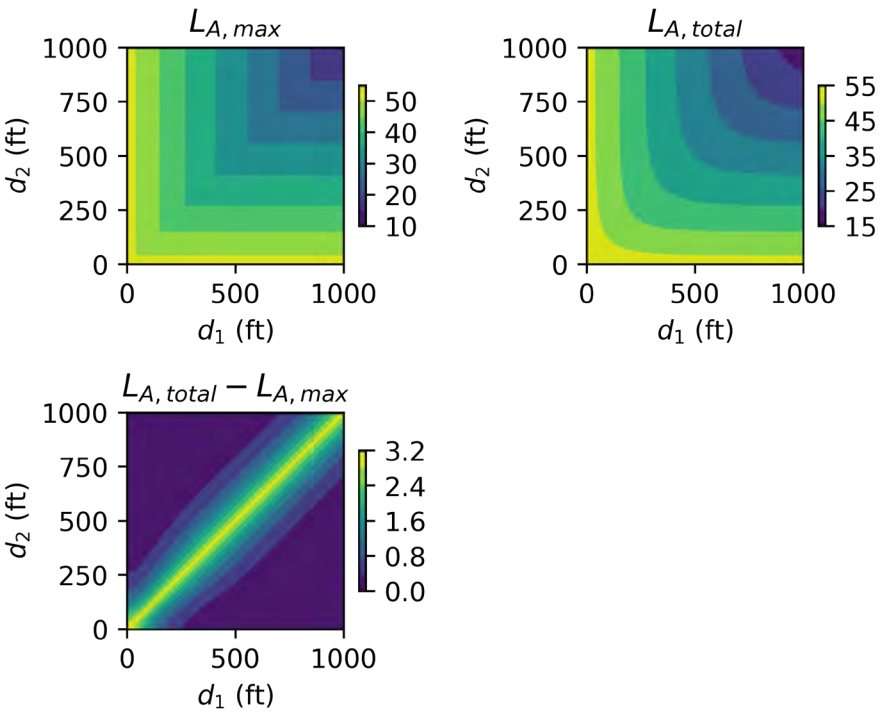


**Figure 40.** Evolution of $L_{A,max}$ (upper left), $L_{A,total}$ (upper right), and the difference between these two quantities (lower left).

### 4.7.3 Varying the Number of Vehicles

We varied the number of vehicles from 2 to 100. All vehicles were assumed to be at an altitude of 100 ft and a distance of 1,000 ft from the microphone. These assumptions correspond to the diagonal of the previously shown plots; that is, the situation in which the difference between the two metrics is the largest.

In the previous section, the noise level was underestimated by $10\log(2)$ because we were considering two vehicles. Here, we expect the difference to be $10\log(N_{vehicles})$, which is confirmed in Figure 41.

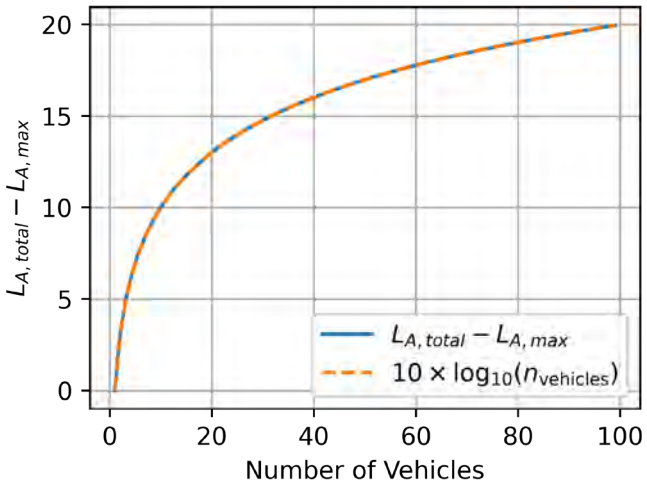


**Figure 41.** Evolution of the approximation error.

### 4.7.4 Assessing Situations in which Multiple Vehicles are Simultaneously Close to a Geographical Location

In the previous sections, we assessed situations in which multiple vehicles are simultaneously located within a relatively close distance to a given geographic location. We sought to verify whether such a situation would arise in a drone delivery scenario. In the following, we applied a fixed time window for the simulation (e.g., 1 hr).

Within this time window, a fixed number of vehicles depart and proceed to deliver packages. The departure times were chosen so as to be uniformly distributed within the time window. Delivery trips that would exceed the time window were truncated.

We plotted the duration during which more than a certain threshold number of vehicles are within a certain threshold distance (measured in feet) of the grid point. Here, we set the threshold number of vehicles to two or five and the threshold distance to 100, 500, or 1,000 ft. The total number of vehicles in the simulation was set to 100, 500, or 1,000 vehicles.

These results correspond to situations in which the value of $L_{A,max}$ may significantly differ based on the method of computation.

***Creating the Flight Schedule***

We created a flight schedule and stored it in a 3D array whose dimensions are (a) the number of time steps, (b) the total number of vehicles in the simulations, and (c) four data values. The last dimension, which has a size of 4, contains the (x, y, z) coordinates of the vehicle and the noise level at 100 ft. In the present study, the vehicle's noise level is not used because we only focus on distances. Here, we keep the number of 1-s-long time steps fixed to 3,600 (1 hr), but we vary the total number of vehicles departing within that flight window.

***Counting the Number of Nearby Flights***

To obtain a quantity that can be easily represented on a map, for each location on the grid, we counted the number of time steps in which the threshold number of vehicles was exceeded within the threshold distance. Both the threshold distance and the threshold number of vehicles were varied.

***Results***

We used Texas data as an example, where the locations and vehicle noise values are loosely based on the Noise Assessment for Wing Aviation [8], with two warehouses used as staging locations for the drones. Results are shown for the different threshold values in Figures 42–44.

***Observations and Conclusions***

From Figures 42–44, we observe the following:

- As we increase the distance threshold, the number of time steps for which the condition is met increases; there are more situations in which vehicles are within a 500-ft radius of a grid point than situations in which they are located within 100 ft.
- As we increase the threshold number of vehicles, the number of time steps in which the condition is met decreases; there are fewer situations in which five vehicles are within a given distance of a grid point than situations in which only two vehicles are within this distance.
- As the total number of vehicles simulated within the 1-hr time window increases, the number of time steps in which multiple vehicles can be found within a given distance of a grid point increases. As the vehicle concentration increases, it becomes easier to find situations in which multiple vehicles are simultaneously within a given distance of a grid point.
- There are two main grid points for which many vehicles may be found simultaneously, corresponding to the two warehouses from which vehicles depart.

These observations match our expectations; the zones of high traffic correspond to the neighborhoods of the staging locations. When the total number of vehicles remains relatively low, there are few situations in which two or more vehicles are found simultaneously near a grid point, and these situations occur only when the radius is set to 500 or 1,000 ft. These correspond to situations in which the sound levels would be relatively low because of the relatively high distance, and the noise would need to be summed for fewer than five vehicles.

As the number of vehicles increases, such situations become more common, and more than five or more vehicles may be found within the threshold distances used in this study. However, such occurrences are relatively rare and are concentrated at locations from which the vehicles depart. For 1,000 vehicles departing within a 1-hr time window, we begin to observe ray-shaped zones, for which multiple vehicles may be present within a given distance. However, this level of traffic most likely exceeds realistic levels (a rate of 1,000 departures per hour corresponds to a departure every 3.6 s). Moreover, if such a high density of traffic were needed, more staging locations would most likely be used, therefore reducing the noise impact at each of the staging locations.

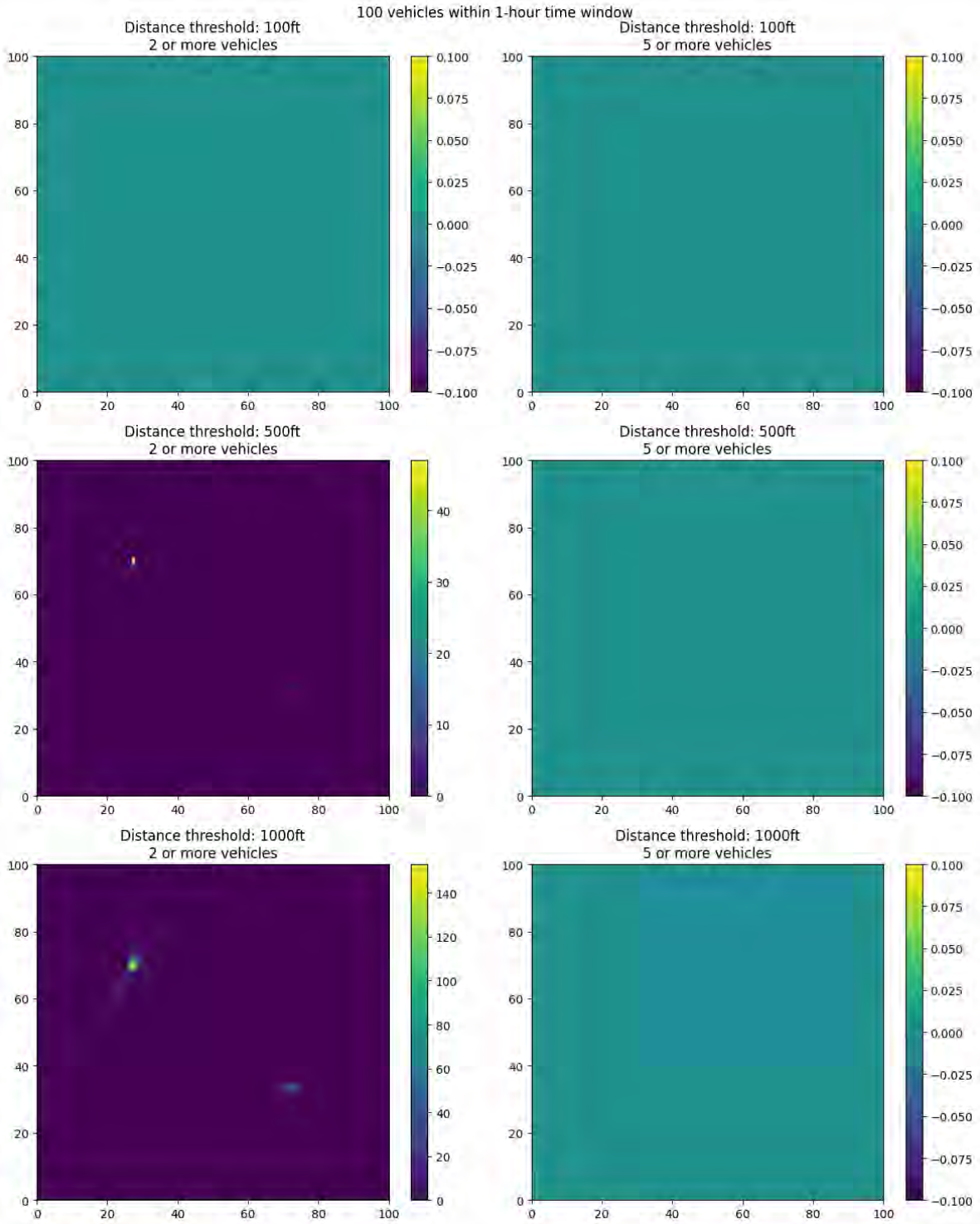


**Figure 42.** Number of occurrences in which the number of vehicles simultaneously flying within a certain threshold distance of a location in the study area exceeds a threshold number of vehicles. Results are shown for the least dense scenario: 100 vehicles within a 1-hr time window.

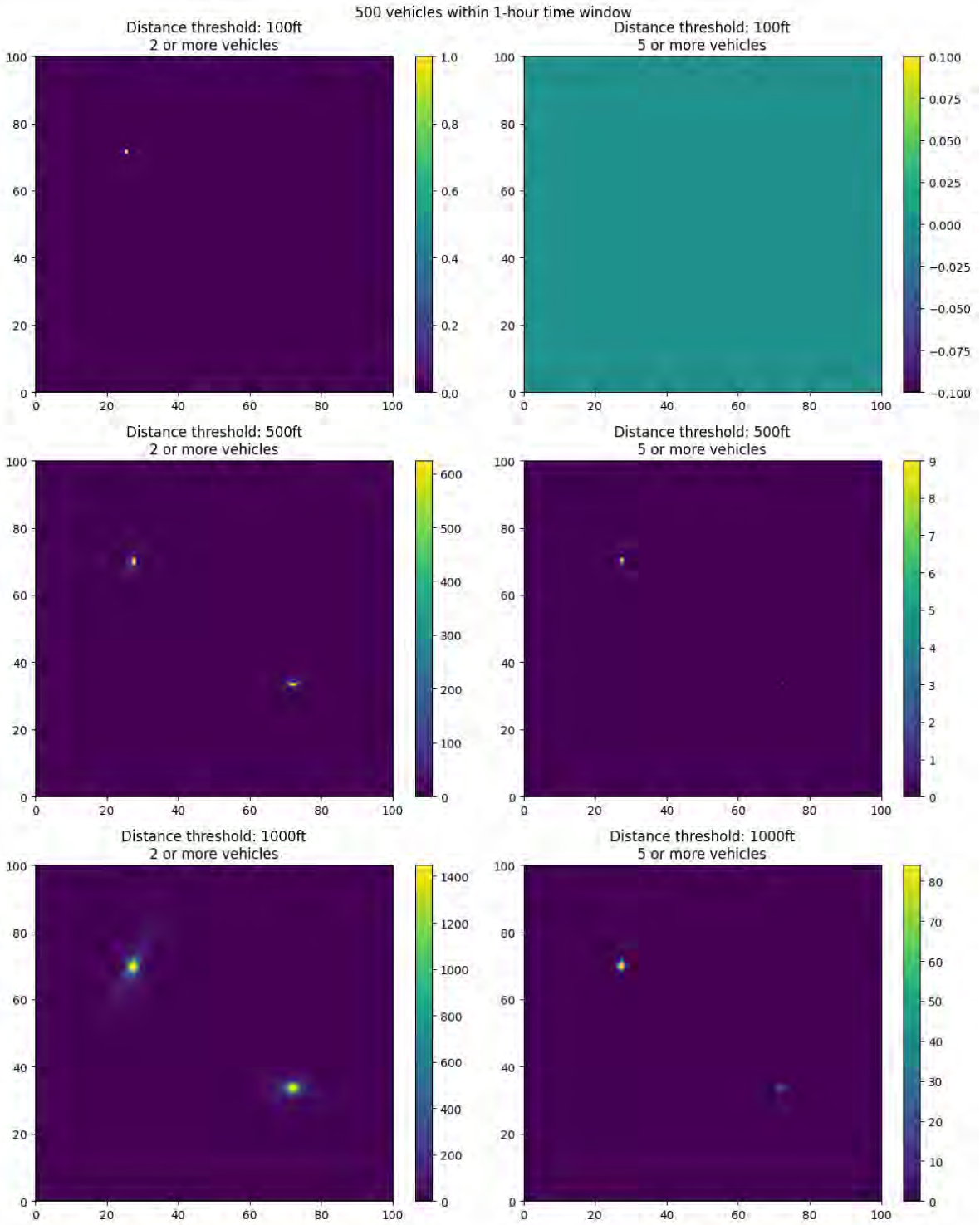


**Figure 43.** Number of occurrences in which the number of vehicles simultaneously flying within a certain threshold distance of a location in the study area exceeds a threshold number of vehicles. Results are shown for the least dense scenario: 500 vehicles within a 1-hr time window.

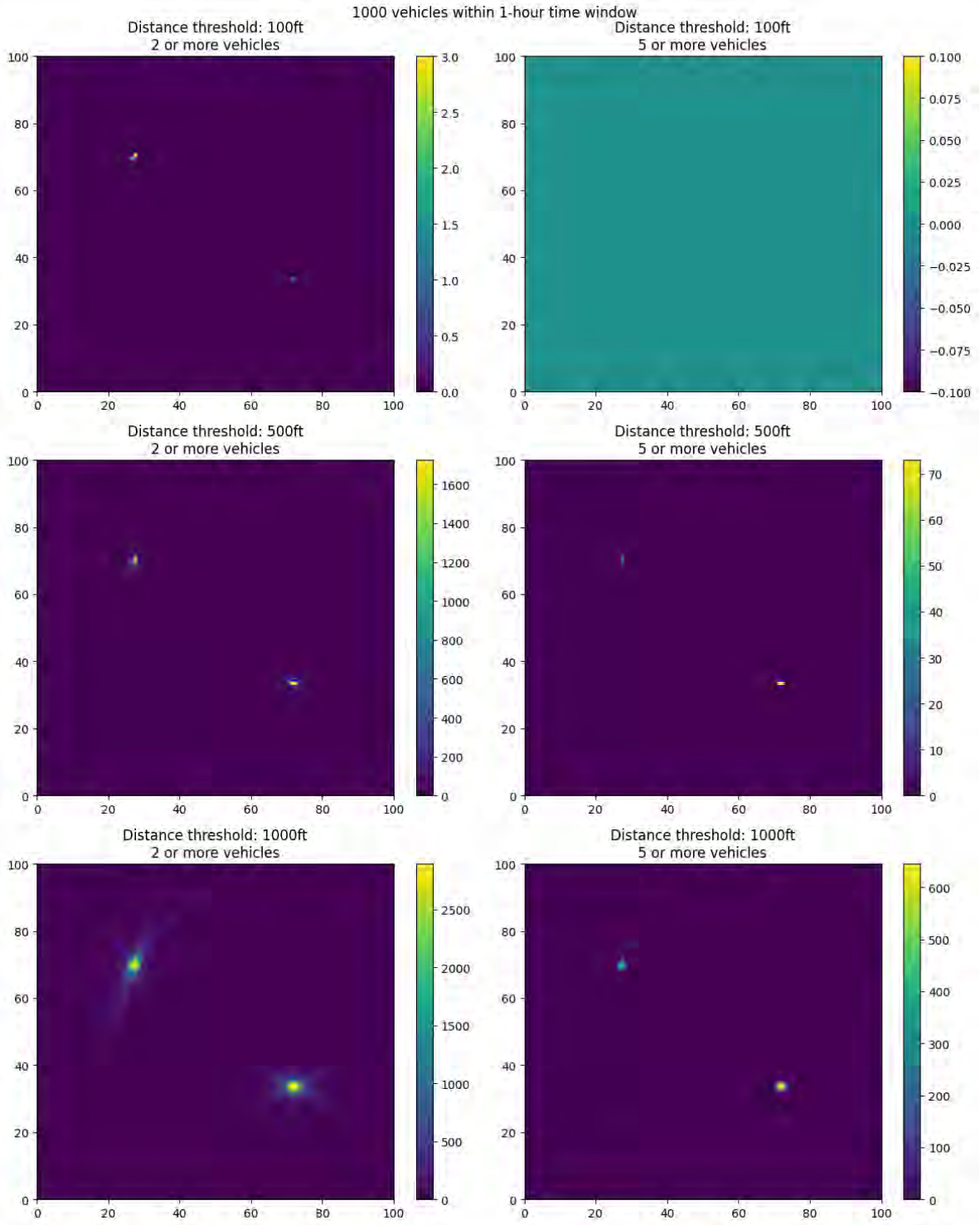


**Figure 44.** Number of occurrences in which the number of vehicles simultaneously flying within a certain threshold distance of a location in the study area exceeds a threshold number of vehicles. Results are shown for the least dense scenario: 1,000 vehicles within a 1-hr time window.

## 4.8 Uncertainty Propagation Leveraging GPU

### 4.8.1 Motivation and Objective

UAS operations are subject to multiple sources of variability, including the following:

- Daily individual flight trajectories are dependent on orders/demand;
- Staging locations may change day to day (e.g., when trucks are used for staging drones); and
- Operator strategies for trajectory planning may include noise dispersion and altitude constraints to minimize noise.

Thus, UAS operations should be modeled as a stochastic process, as annual average day metrics do not capture daily changes. Instead of seeking deterministic measures of noise exposure, we aim to estimate the likelihood of exceedance for some threshold on any given day across the study area. In mathematical terms, we consider $P(L_{A,max} \geq X\ dBA)$ the probability that $L_{A,max}$ will exceed $X\ dBA$. Results are depicted as contours on a map of the study area, with the level $X$ being varied, therefore leading to different contour plots for each value of $X$. This study is the second attempt at a Monte Carlo simulation, which takes advantage of the GPU speed-up.

### 4.8.2 Setup

The study area covers the urban region of Memphis, TN, representing a 60-mile × 60-mile square. Delivery drones may depart from one of 41 warehouses, and delivery locations are uniformly sampled from residential locations within the study area. The resolution of the analysis grid is 1,056 cells in each direction. As a result, the sides of the square cells measure approximately 300 ft. In the considered operational scenario, 500 deliveries are performed per day, and the simulation is repeated for 10,000 days. The only source of variability considered in this study is the day-to-day variability of the flights.

Compared with the first Monte Carlo study, the grid is approximately 4-times larger, and there are approximately 5-times more flights per day, resulting in an overall 20-fold-larger problem size.

### 4.8.3 Results and Conclusions

The results of this study are shown in Figure 45. As discussed previously, each plot corresponds to a different noise exposure level threshold: 10, 20, 30, or 40 dBA. Warehouses are shown as red dots. We observe that the high-probability area shrinks as the threshold increases, consistent with expectations. The geographical areas with a high probability of exceeding 40 dBA are concentrated near the staging locations.

The full Monte Carlo simulation was completed in approximately 7 hr by using 50 GPUs from Georgia Tech's PACE cluster. Each run takes approximately 2 min to complete. This runtime is orders of magnitude shorter than the first Monte Carlo attempt that was run on a CPU

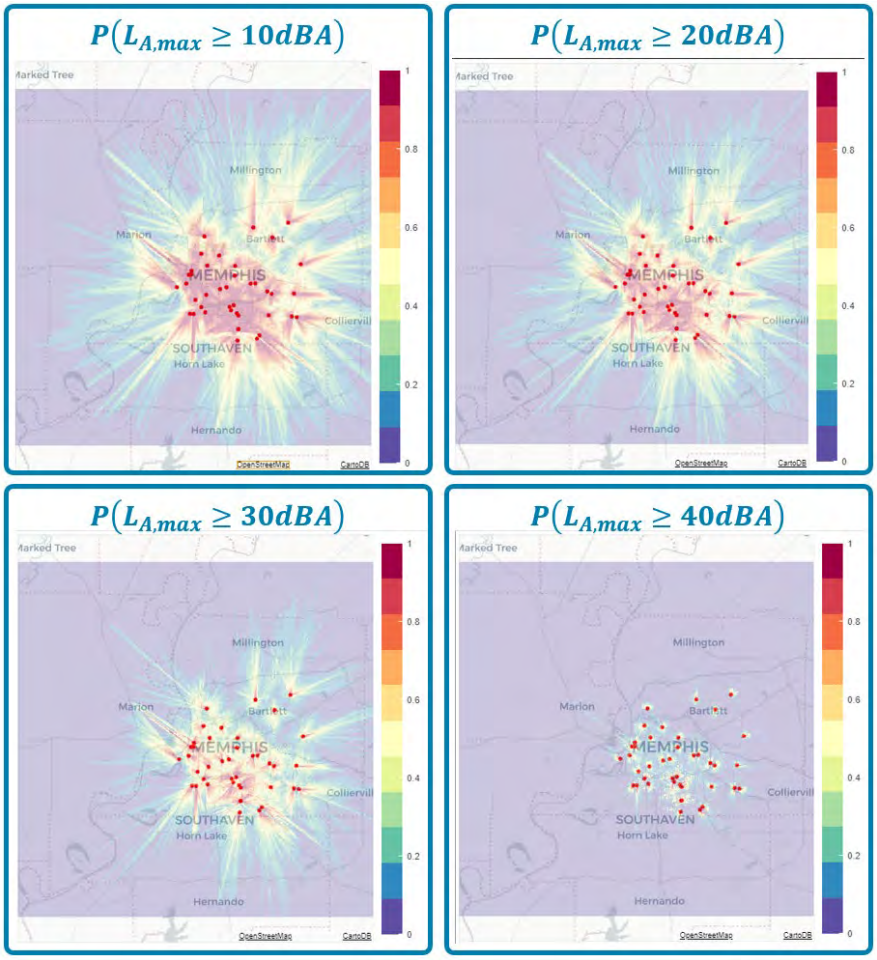


**Figure 45.** Results of the graphics processing unit (GPU) Monte Carlo simulation. Contour plots denote the probability that $L_{A,max}$ exceeds a threshold level. Threshold levels were varied at 10, 20, 30, and 40 dBA.

(hours instead of weeks), thus confirming the benefits achieved when running the noise engine on GPUs.

## 4.9 Packaging of the Noise-assessment Tool

Although Python offers access to a variety of quality off-the-shelf packages that fulfill many computational and analysis needs, it requires setting up a suite of software on the client machine on which the Python tool needs to be executed, including Python itself, as well as third-party packages the software depends on. This complexity compounds with the configuration management problem: Python version and third-party packages must not only be installed; they also need to be compatible versions. To address these shortcomings and remove the setup burden from the user of the noise-assessment tool, the goal of this effort was to package the tool in such a way that the setup on a new client machine required minimal effort.

Among the several options available in the Python ecosystem, PyInstaller stood out as the most mature and promising solution. This is not, however, a plug-and-play solution; although PyInstaller automates most of the process of packaging the Python tool and its dependencies in a stand-alone folder or executable, certain libraries require additional work to be integrated to this executable. A version of the noise-assessment tool relying only on CPU was successfully packaged and tested on Windows. The packaging of the GPU version of the noise-assessment tool was initiated but could not be thoroughly tested due to the lack of access to a machine featuring a GPU.

## Milestone

The team delivered a recommendation for an updated GIS system to the FAA and members of the AEDT development team.

## Major Accomplishments

The team presented an initial prototype of the UAS noise engine with an interactive display while running on a parallel computing cluster to the FAA.

## Publications

None.

## Outreach Efforts

The team engaged in outreach and coordination with the ASSURE Center of Excellence team and their work at MSU. The team also collaborated with the Volpe Center and participated in the NASA UAM Noise Technical Working Groups.

## Awards

None.

## Student Involvement

The Georgia Institute of Technology student team consists of three graduate research assistants. At the beginning of the project, all graduate research assistants engaged in the GIS background research. The team is now divided to tackle the different aspects and implementation of the noise engine, novel computational technology testing, and the creation of benchmark studies that serve as a test bed for testing the computational scaling of different approaches.

## Plans for Next Period

This is the final report for Project 9. Continuation of this work is taking place as part of Project 94, which builds on the capabilities developed in the context of Project 9, extends the modeling to more diverse concepts of operations, and introduces probabilistic aspects to the noise assessment.

## References

Aftomis, M. J., Berger, M. J. & Alonso, J. J. (2006). *Applications of a Cartesian mesh boundary-layer approach for complex configurations* [Meeting presentation]. 44th AIAA Aerospace Sciences Meeting, Reno NV, United States.

Gropp, W. (n.d.). Introduction to MPI I/O [CS 598, Lecture Notes]. University of Illinois, Urbana-Champaign, IL, United States.

The Office of Spatial Analysis and Visualization at the Bureau of Transportation Statistics, U.S. Department of Transportation

Bokeh Development Team. (2018). *Bokeh: Python library for interactive visualization*. http://www.Bokeh.pydata.org

Dask Development Team. (2016). *Dask: Library for dynamic task scheduling*. https://dask.org
Cottam, J. A, Lumsdaine, A., & Wang, P. (2014). Abstract rendering: Out-of-core rendering for information visualization. *Proc. SPIE 9017, Visualization and Data Analysis 90170K*. https://doi.org/10.1117/12.2041200
Teal Group (2022). *2021/2022 Civil UAS Market Forecast*.
HMMH (2021). *Noise assessment for wing aviation proposed package delivery operations in Frisco and Little Elm, Texas*. (HMMH Report No. 309990.003-2.)