Received: 29 July 2025 | Revised: 17 April 2026 | Accepted: 11 May 2026 | Published online: 17 July 2026

RESEARCH ARTICLE

Journal of Computational and Cognitive Engineering
2026, Vol. 00(00) 1–16
DOI: 10.47852/bonviewJCCE62026975

MOD-NAV: Scalable Modular Inertial Navigation Framework for Resource-Constrained Embedded Hardware

Mahender Katukuri[1,*], Satyanarayana JV[1], Ruchir Gupta[2] and Bhaskar Biswas[2]

[1] *Research Centre Imarat, India*

[2] *Department of Computer Science, Indian Institute of Technology (BHU) Varanasi, India*

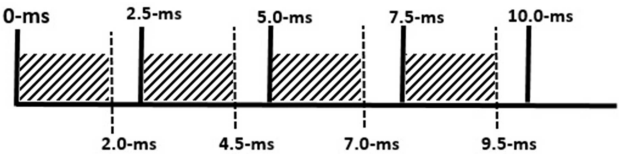
Abstract: Inertial navigation algorithms are inherently computationally intensive, often necessitating high-performance embedded processors for real-time execution. This requirement poses a significant challenge for deploying such algorithms on resource-constrained, small-scale embedded platforms commonly used in cost-sensitive and power-limited applications. To address this limitation, this paper presents a scalable and modular inertial navigation framework designed to adapt to varying application requirements. The proposed framework decomposes the navigation system into independent functional modules, enabling selective inclusion or exclusion based on specific operational needs. This modular architecture allows developers to tailor the computational load by integrating only the essential components required for a given application, thereby reducing unnecessary processing overhead. As a result, the framework significantly lowers computational demand, memory usage, and power consumption. Furthermore, the scalability of the framework ensures that it can be efficiently deployed across a wide spectrum of hardware platforms, ranging from low-power microcontrollers to high-end processors. The design also supports ease of maintenance, extensibility, and system upgrades, making it suitable for diverse navigation applications such as unmanned systems and portable devices. Experimental validation demonstrates that the proposed approach achieves a substantial reduction in computational complexity while maintaining the accuracy of the navigation output. This makes it a practical and efficient solution for implementing inertial navigation systems on resource-constrained embedded hardware.

Keywords: inertial navigation, embedded systems, modular software, resource-constrained embedded hardware

1. Introduction

The inertial navigation algorithm framework used for avionics applications is complex in nature from computational perspective. This computational complexity requires a high-end processor to execute the algorithm implementation. A zynq-7000 processor based embedded system with a clock-speed of 600 MHz requires 2 m-s of time to execute one cycle of inertial navigation algorithm. The execution scenario is depicted in Figure 1. The algorithm needs to be executed with a periodicity of 2.5 m-s., i.e., each cycle of execution should be completed within 2.5 m-s. During run time, the zynq-7000 processor takes 2 m-s for execution of each cycle, out of the allocated 2.5 m-s. The same execution scenario on an Arduino-processor based system is depicted in Figure 2. The Arduino is a low-end processor and runs with a clock speed of 16 MHz. It takes an execution time of 45-m-s, which is supposed to be completed within 2.5 m-s. It exceeds the allocated execution time by a huge margin. It is practically not possible to fit the algorithm to execute within 2.5 m-s. To overcome this problem, we propose an approach by which the Nav algorithm can be scaled down to smaller modules and the required modules only will be selected for addition into the Nav application. By doing this, we can reduce the number of modules to be executed in each cycle and the algorithm can be fit in the available computational bandwidth. We have carried out the work on how to divide the algorithm into smaller modules, how to decouple each computational elements, how to reduce the interdependency across modules and created a framework of modules, identified the required inputs & outputs for each of the module.

Figure 1
Nav-Alg execution scenario on zynq processor

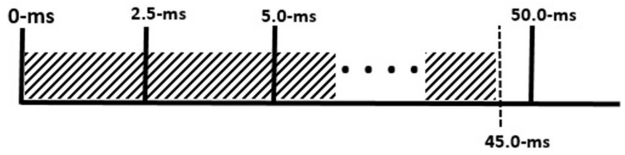


2. Background

An inertial navigation system (INS) provides the position, velocity, and orientation of an object based on the measurements

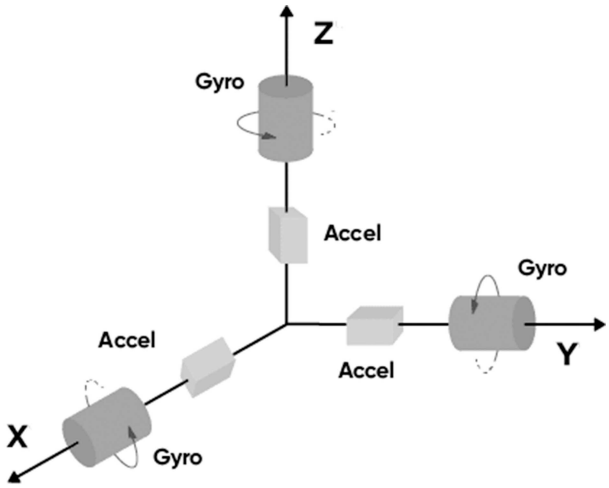
***Corresponding author:** Mahender Katukuri, Research Centre Imarat, India. Email: mahender.k.rci@gov.in

Figure 2
Nav-Alg execution scenario on Arduino processor accessibility



received from inertial sensors. INS typically contains three orthogonal gyroscopes and three orthogonal accelerometers measuring angular and linear accelerations, respectively. The output of these sensors is passed as input to the inertial navigation algorithms to generate the body rates, linear & angular accelerations, quaternions, positions, and velocities in three axes(x, y, z). These parameters will be further used in the control of trajectory, orientation of the object. The mounting of the accelerometers and gyros, in three axes in an orthogonal orientation, is depicted in Figure 3.

Figure 3
Gyro, accelerometer sensor mounting of INS in three axis



The accelerometers sense the linear accelerations, and gyros sense the angular accelerations. The sensed accelerations are in analog form. Through appropriate ADCs, the analog signal will be digitized. This digital data correspond to the linear, angular accelerations in the previous time quantum. The time quantum is user-defined. It can be fixed with a few milliseconds (for ex., 2.5 m-s, 5m-s, 10 m-s, 20 m-s, or any custom-specific periodicity) based on the application's accuracy requirements and sensor measurement characteristics. For this work, it is fixed as 2.5 m-s. Every time quantum, the sensor measurements will be taken and passed as input to the navigation algorithm to generate the navigation data consisting of body rates, accelerations, quaternions, positions, and velocities in the three axes. The navigation algorithms are computationally complex and demand higher processing power. One iteration execution of the full-scale navigation module of the current implementation takes 2 m-s on a zynq-7000 series of processor with a time quantum of 2.5 m-s. The exact configuration takes 45 m-s of execution time on an Arduino board. For a productive application, the execution time of each iteration should be less than the time quantum so that the execution of the algorithm with the current sensor measurement can be completed and navigation output can be generated for the current measurement sample before the next measurement sample becomes available. Though a zynq-7000 series processor with a 600 MHz clock speed can run the full-scale navigation stack with comfortable time margins, an Arduino type of resource limited computer cannot run a full-scale navigation stack. But, with the widespread usage of the low-end embedded hardware and associated applications, it has become necessary to run various applications on the resource limited hardware. Inertial navigation has been used in complex systems like aircrafts, spacecrafts, ships, submarines, and other high-level applications. But in recent days, this system has found its applications in less complex applications like small-scale drones, robots, autonomous vehicles, and other applications in which data about position, velocity, and orientation is required. These less complex applications are limited by constrained resources, so it may not be possible to run the complex algorithms of the Inertial Navigation System on a full scale. This limitation can be overcome by using a scalable Navigation Framework in which the complex algorithms of the Inertial Navigation system are divided into different modules and each module is implemented separately. The user can then customize the software by selecting only the modules required for the small-scale application in use. This modular, scalable inertial navigation framework has been termed as MOD-NAV, which can be used to scale the INS with respect to the computational resource constraints of the embedded hardware being used.

2.1. Contributions

The following are the contributions of this work:

1) A proposal of modularization of inertial navigation algorithms: We have classified the inertial navigation algorithms into various cohesive modules. This facilitates in customization & scaling of the framework as per the requirements. This makes the inertial navigation algorithms efficient so that they can be employed in a wide variety of low-end resource-constrained hardware. Based on the accuracy requirements of the navigation output, the algorithms can be customized when they are modular.
2) Implementation of a modular inertial navigation framework: We have implemented the modularized & cohesive modules and tested them for the functionality & accuracy. The framework is tested in the full-scale configuration on a Zynq-7000 series of processor-based embedded hardware and tested in the minimal configuration on an Arduino-uno microcontroller-based hardware.

2.2. Motivation

The complex computational nature of the inertial navigation algorithms demands higher computational resources. This makes the algorithms to run only on high-end computational hardware. The emergence of navigational applications like small-scale drones, autonomous vehicles, robots, and similar applications created a need to optimize the hardware and software resource requirements to run navigational algorithms. The resource requirements of the hardware & software are interlinked together. Higher complex software requires a higher computational hardware resource and vice versa. The cost of the system is also proportional to the resource requirement. To optimize the system, there is a need to optimize the software. With this intention, we decided to carry out a modularized structure for the navigation algorithms so that we can customize

the software by adding only suitable modules. This drastically reduces the computational requirements of the hardware. With this requirement, we have made a proposal for modularization of the inertial navigation algorithms and implemented the software with this framework. This framework can be scaled up/down by adding and deleting necessary modules based on the application requirements.

3. Terminology

The measurements of the inertial sensors (Accelerometers, Gyros) will be sampled for each time-quantum. An accelerometer senses the linear accelerations of the body in the axis in which it is connected. By connecting three accelerometers in orthogonal axes, the linear accelerations of the body in three axes(x, y, z) can be sensed for every time quantum. The same thing is applicable for gyros. The Gyro senses the rotational acceleration. The output of the sensors will be in analog voltage. Through appropriate analog-to-digital converters (ADCs), the digital data will be obtained for every time quantum. Let us assume the time quantum is 2.5 m-s. In such a case, the ADCs can be sampled at every 2.5 m-s and the linear accelerations (δ_v) in the three axes for the past 2.5 m-s can be obtained. Rotational accelerations (δ_θ) can be obtained by sampling the ADCs of the gyros with a periodicity of the time-quantum. The measurements (δ_θ, δ_v) will be passed to the inertial navigation algorithms to generate the orientation, positions, and velocities of the body in three axes. The input to the algorithms is the sensor output (δ_θ, δ_v), and the output of the algorithms is the attitude, positions, and velocities of the body in three axes. This transformation requires an initialization of the algorithms, execution of the necessary equations,and generation of the output. We have classified these computations into various cohesive modules so that the required modules can be added to the custom-specific modules and other modules can be dropped. This facilitates the creation of a tailor-made navigation system and optimizes computational needs. The computations are described in the following sections. While describing the computations, specific notation is used to represent various parameters. The notation is given in Table 1.

4. Related Work

Accurate inertial navigation is essential for aerial vehicles, yet many platforms must operate on resource-constrained embedded hardware where power, computation, and payload are limited. Consequently, a modular navigation architecture—one that can flexibly combine lightweight sensors, scalable algorithms, and interchangeable processing units—is required to deliver high-precision positioning without overburdening the onboard processor. Recent studies illustrate how such modularity and algorithmic efficiency are achieved across a wide range of INS-centric solutions. Li et al.[1] propose a distributed relative-pose estimator that fuses IMU data with inter-UAV data-link measurements via a sliding-window factor-graph optimizer, enabling high-accuracy formation positioning in GNSS-denied environments. This demonstrates the need for efficient fusion. The proposed modular framework replaces the heavy optimiser with lightweight EKF/UKF modules. Wang et al.[2] introduce a UWB-IMU relative-localization scheme that employs adaptive gradient observers and DREM-based finite-time estimation to reach centimeter-level accuracy without persistent excitation. This shows the algorithmic efficiency in achieving high accuracy with IMU setups. Wang et al.[3] integrate a strapdown INS with infrared scene-matching through a Kalman filter, markedly improving UAV positioning when GNSS is unavailable. It provides a vision-inertial module that can be swapped out for lighter visual-odometry or omitted in low-cost configurations. Deliparaschos et al. [4] describe a low-cost modular quadrotor platform equipped with Jetson Nano/Z-turn and a suite of sensors (IMU, UWB, stereo camera, LiDAR, ultrasonic) managed by ROS 2, enabling flexible indoor navigation research on embedded processors. This directly aligns with our hardware-agnostic modularity. We extend this concept to the navigation-software stack. Zhu et al.[5] proposed a tightly coupled multi-frequency PPP-RTK/INS model that merges atmospheric corrections and ambiguity resolution, enhancing urban vehicle positioning under GNSS interruptions. This illustrates a high-accuracy GNSS-centric modular framework. Cohen and Klein[6] survey deep-learning techniques for IMU calibration,

Table 1
Notations

λ	Latitude	μ	Longitude
h	Altitude	M	Mach
ψ	Azimuth	ϕ	Roll
θ	Pitch	ψ	Yaw
gc	Geocentric	gd	Geodetic
hsea	Height from sea level	Q	Quaternion
ToE	Turn rate of Earth	w	Earth rotation rate
e	Ellipticity of earth	ecc	Eccentricity of earth
$\delta\theta$	Incremental Angles for the last time quantum		
δ_v	Incremental Velocities for the last time quantum		
Ra	Radius of semi major axis of the earth		
Rb	Radius of semi minor axis of the earth		
Rm	Range/Distance from center of earth		
Rn	Meridian radius of curvature		
Re	Transverse radius of curvature		
radi	Radi vector		

denoising, and sensor fusion, highlighting performance gains across land, air, and marine platforms while emphasizing lightweight network designs suitable for embedded deployment. It provides learning-based modules that can be integrated as optional plug-ins when resources permit. Mei et al. [7] present a vision-inertial fusion scene-matching system that combines MEMS inertial mechanization with optimized image matching, delivering robust and computationally efficient UAV navigation. This serves as a vision-fusion alternative. The tool can enable/disable the fusion based on sensor availability. Hsan et al. [8] provide a comprehensive review of UAV classifications, swarm systems, charging methods, and security challenges, underscoring the need for modular hardware that can adapt to diverse mission requirements. Saha et al. [9] examined the inertial navigation on ultra-low-power hardware. Discussed the EKF-based methods and the issues of resource-constrained platforms. Cucci et al. [10] evaluates stochastic calibration techniques (Allan-variance linear regression and GMWM) for integrated INS/GNSS system. Demonstrated their impact on system accuracy while remaining computationally tractable. Pritzl et al. [11] proposed a cooperative framework where a LiDAR-equipped UAV provides 3-D relative localization for a smaller camera-UAV, enabling precise guidance in GNSS-denied settings with modest processing loads. Indraganti et al. [12] demonstrated a fused GPS/INS attitude estimation for high-altitude platforms, achieving superior accuracy over GPS-only solutions using a modular filter architecture. Kulkarni et al. [13] describe an advanced GPS/IMU sensor-fusion INS that leverages sophisticated filtering for enhanced navigation performance on embedded processors. This serves as a high-performance fusion option for GPS and INS data. Singh et al. [14] compare Kalman-filter and recurrent-neural-network integration of INS/GPS for radar-aided navigation, finding the RNN-based approach. This yields better results while maintaining a lightweight implementation. This work highlights a learning-based fusion alternative for navigation data. Hadjiloizou et al. [15] introduced a real-time multi-sensor pose estimator that fuses IMU, vision, and UWB data with a Kalman filter to handle intermittent communication during indoor quadrotor flights, emphasizing modular sensor integration. Li et al. [16] developed a cooperative UAV swarm navigation framework based on weighted-uncertainty belief propagation, exploiting INS uncertainty and inter-UAV information for GNSS-denied positioning with scalable computation. Li et al. [17] proposed a tightly integrated PPP-RTK/MEMS/vision system that achieves centimeter-level vehicle navigation in urban environments via a multistate-constraint Kalman filter, designed for embedded platforms. Fesenko et al. [18] improve UAV inertial navigation accuracy by applying an enhanced Madgwick filter within a platform-less MEMS INS, outperforming traditional filters while keeping computational demand low. It Supplies a lightweight filter option that can be used in a minimal navigation stack configuration. Patel & Faruque [19] assessed multi-IMU fusion strategies—including virtual IMU, federated, and augmented EKF—showing improved attitude estimation when GPS is unavailable. This implementation can be realized even on modest microcontrollers. Gu et al. [20] propose a multi-GNSS PPP/INS tightly coupled model with atmospheric augmentation, delivering sub-meter accuracy for urban vehicle navigation with a modular software stack. Klein et al. [21] introduce a radar-aided navigation system for small drones that uses omnidirectional radar Doppler analysis to reduce INS drift in GPS-denied environments, employing a lightweight processing pipeline. It provides a radar-fusion alternative for navigation. Lai & Yang [22] presented a robust Kalman filter for INS/GPS integration, enhancing resilience against sensor errors while remaining suitable for embedded execution. Wei et al. [23] combines adaptive Kalman filtering with a wavelet neural network to maintain INS/GPS performance during GPS outages, using a compact network architecture. This demonstrates a hybrid learning-filter module. Herath et al. [24] describe RoNIN, a data-driven inertial navigation approach that leverages diverse IMU motion data for robust outdoor positioning, implemented with efficient neural models. It provides a data-driven navigation alternative that can be integrated for platforms with sufficient resources. Chen et al. [25] developed L-IONet, an LSTM-based pedestrian inertial navigation model achieving high accuracy and low memory usage on mobile devices, demonstrating the feasibility of deep-learning-enhanced INS on constrained hardware. It shows feasibility of deep-learning INS on constrained hardware. Hamdy et al. [26] implemented a GPS/INS EKF algorithm on a Cortex-M4 processor, demonstrating real-time feasibility for land-vehicle navigation with modest computational resources. Ren et al. [27] designed an ultra-tightly coupled INS/GPS system using an unscented particle filter, targeting high accuracy on low-end hardware. It provides a high-accuracy particle-filter option that can be selected when computational hardware processing power supports. Li et al. [28] proposed a hybrid CKF-GRU algorithm that merges a cubature Kalman filter with a gated recurrent unit to improve GPS/INS integration while keeping the model lightweight. Wang [29] presents an IMM-EKF based GPS/INS positioning method for drones, employing multiple-model filtering to enhance accuracy in complex environments with limited processing capacity. It provides a multi-model fusion alternative that can be included as an advanced module for demanding missions. Collectively, these works demonstrate solutions for improving the accuracy of the navigation and to sustain the outages of GPS. The problem of extending a high accurate navigation system to low end aerial vehicle applications is not being considered. This work proposes a methodology of extending high accurate navigation to low end system. This is achieved by reducing the software computational load and thereby reducing the embedded computational hardware cost.

4.1. Limitations of existing methods

1) The major limitation of existing inertial navigation implementations is the higher demand for computational hardware. This requires the usage of high-end computational hardware, thereby increasing the cost of the overall system.
2) Non-availability of modularized, cohesive implementable structure of the algorithms. A modularized structure facilitates in scaling up/down of the framework as per the custom-specific requirements so that we can include the bare minimum required modules into the application. This optimizes the computational overhead of the implementation.
3) Non-availability of suitable implementation for resource-limited hardware. A scalable architecture facilitates to overcome this limitation.

5. MOD-NAV Framework

The MOD-NAV framework proposal constitutes the classification of inertial navigation algorithms into cohesive modules. Every navigational application may not require the complete implementation of the navigational framework. Though the

high-end applications need complete implementation of the framework, many low-end applications like small-scale drones, micro-robots, and similar applications do not require the complete framework. However, they require portions of the navigational algorithms. Some applications require position updates, velocity updates, and quaternion updates with a high frequency; some applications may require only quaternion updates and may not require position, velocity updates, and so on. The modularization proposed as part of this work facilitates the customization of the algorithms by adding only required modules and eliminating unnecessary modules of the navigational framework into the target application. Complete code of the algorithms as per the proposed modularization criteria is implemented in the navigational framework. The user application can be generated from the full-fledged framework by selective addition of required modules from the framework. The user has the provision to add the required modules and remove the not-required modules into the application from the framework. This optimizes the code and reduces the demand for the target computational hardware. This facilitates in running the navigational algorithms even on extremely resource-constrained embedded hardware. The scalability offered by the framework ensures that it can run on resource-constrained embedded hardware. The complex algorithms of inertial navigation software are divided into different modules and implemented separately as part of the framework. The modularization and the implementation is described in the following sections. The algorithms are broadly classified into modules of Initialization, Core Navigation, Earth Parameters Computation, and Navigation Utilities, along with the updation modules of Quaternion, Velocities, and Positions. Further, each module will be divided into sub-modules.

5.1. Initialization module

The navigation system will be initialized with parameters, viz, initial position, velocity, orientation, and earth components related parameters. These parameters get updated based on the sensor measurement in each cycle. The data-initialization part has been classified into an initialization module. The following are the data items initialized by this module: Initial Position (Lat(λ), Long(μ), Altitude(h)), orientation (syi(Ψ), theta(θ), phi(Φ)), earth-related components (ω_0, semi-major, and semi-minor radius of the earth(R_a and R_b, respectively), ellipticity(e), and eccentricity(ecc)). A sample initialization data are depicted in the following algorithm.

$$Q_{Z0} = cos(\psi / 2.0); \quad (1)$$

$$Q_{Z1} = 0.0; \quad (2)$$

$$Q_{z2} = 0.0; \quad (3)$$

$$Q_{z3} = sin(\psi / 2.0) \quad (4)$$

$$Q_{y0} = cos(\theta / 2.0); \quad (5)$$

$$Q_{y1} = 0.0; \quad (6)$$

$$Q_{y2} = sin(\theta / 2 : 0); \quad (7)$$

$$Q_{y3} = 0.0 \quad (8)$$

$$Q_{x0} = cos(\phi / 2.0); \quad (9)$$

$$Q_{x1} = sin(\phi / 2.0); \quad (10)$$

$$Q_{x2} = 0.0; \quad (11)$$

$$Q_{x3} = 0.0 \quad (12)$$

$$Q_{zy} = Q_z * Q_y; \quad (13)$$

$$Q_i = Q_{zy} * Q_x; \quad (14)$$

Algorithm 1: InitialiseNavData

Function InitialiseNavData ()
- λ = 17.257658*DegToRad
- μ = 78.499978*DegToRad
- h = -50.0
- Ψ = 54.5*DegToRad
- θ = 90.0*DegToRad
- Φ = 0.0*DegToRad
- velNed[0] = 400.0
- velNed[1] = 200.0
- velNed[2] = -100.0
- posNed[0] = 0.0
- posNed[1] = 0.0
- posNed[2] = 0.0
- R_a = 6378137.0
- R_b = 6356752.3142
- ω_0 = 7.292115e-5
- e = (R_a - R_b) / R_a
- ecc^2 = (R^2a - R^2b) / R^2a

5.2. Core navigation module

The core navigation module is the integrator module. This reads the input sensor measurements δ_θ, δ_v and generates the navigation data output. In the process, this module calls all other navigation modules. This runs an indefinite loop (*while* (1)) and calls all other modules and generates the final output. The module is described using the below algorithm.

The *UpdateEarthComponentsBasedonLatLo*ng() module updates the earth-related components like TOE rates, ω_{ned}, coriolis acceleration, and centripetal acceleration. The *ReadImuIncAngV*el() module reads the δ_θ, δ_v corresponding to the last time quantum, from the sensors. These data will be used by other modules in the generation of the navigation output. If the sensor data are not available for testing of the module, then simulated data can be pumped from this module through a text file. The *CoreNav*() module updates the current quaternion of the body from the measured δ_θ, δ_v. The *VelUpdat*e() module updates the velocities of the body. The *PosUpdat*e() module updates the position.

Algorithm 2: CoreNav

input: void
output: void
Function CoreNav()
  call InitialiseNavData();
  while true **do**
    call UpdateEarthComponentsBasedonLatLong()
    call ReadImuIncAngVel()
    call QuatUpdate()
    call VelUpdate()
    call PosUpdate()

5.3. Earth parameters computation module

This module computes the earth-related parameters, viz. Radi, R_m, R_n, R_e, ToE. These parameters will be used in *PositionUpdate, VelocityUpdate,* and *QuatUpdate* modules. These are the parameters associated with the earth, and their quantity keeps changing as the body's position and velocity keep changing. So, they need to be updated every cycle of the navigation.

$$radi = R_a * t1, \; where; \; t1 = 1.0 - e * sin(\lambda_d) * sin(\lambda_d) \tag{15}$$

$$R_n = \frac{t1}{(t2)^{\frac{3}{2}}}, \; where \; t1 = R_a\,(1.0 - eccentricity^2) \tag{16}$$

$$t2 = 1.0 - ecc^2 * \sin(\lambda_d) * \sin(\lambda_d) \tag{17}$$

$$ecc^2 = (R_a^2 - R_b^2) / R_a^2; \text{ where } e = (R_a - R_b) / R_a \tag{18}$$

$$R_e = t1 / (t2)^{1/2}; \text{ where } t1 = R_a \tag{19}$$

$$TOE[0] = vel\_east / (R_e + h_{sea}) \tag{20}$$

$$TOE[1] = vel\_north / (R_n + h_{sea}) \tag{21}$$

$$TOE[2] = -vel_{east} X \tan(\lambda_d) / (R_e + h_{sea}) \tag{22}$$

5.3.1. Computation of ω_{ned}

This module takes latitude and earth-related parameters as input and computes the ω_{ned} vector.

$$\omega_{ned[0]} = \omega_0 * cos(\lambda_d); \tag{23}$$

$$\omega_{ned[1]} = 0.0; \tag{24}$$

$$\omega_{ned[2]} = -\omega_0 * sin(\lambda_d); \tag{25}$$

5.3.2. Computation of coriolis acceleration

This module takes three vectors, namely ω_{ned}, TOE_Ned and Vel_Ned as input and performs computations and outputs the Coriolis acceleration vector.

$$Coriolis_{acceleration(ned)} = (2.0 * \omega_{ned} + TOE_{ned}) * vel_{ned} \tag{26}$$

5.3.3. Computation of centripetal acceleration

$$CPA = \omega_{ned} * (\omega_{ned} * radi) \tag{27}$$

$$radi_{gc} = [0.0;\; 0.0; -(radi + h_{sea})] \tag{28}$$

$$radi_{gd} = dcm(geocentric\_to\_geodetic) - radi_{gc} \tag{29}$$

$$dcm\_geocentric\_geodetic = \begin{bmatrix} cos(-dev_{angle}) & 0 & -sin(-dev_{angle}) \\ 0 & 1 & 0 \\ sin(-dev_{angle}) & 0 & cos(-dev_{angle}) \end{bmatrix} \tag{30}$$

$$radi_{gd} = \begin{bmatrix} cos(-dev_{angle}) & 0 & -sin(-dev_{angle}) \\ 0 & 1 & 0 \\ sin(-dev_{angle}) & 0 & cos(-dev_{angle}) \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -(radi + h_{sea}) \end{bmatrix} \tag{31}$$

$$radi_{gd} = \begin{bmatrix} -radi.sin(dev_{angle}) \\ 0.0 \\ -radicos(dev_{angle}) - h_{sea} \end{bmatrix} \tag{32}$$

$$h_{sea}.sin(dev_{angle}) = 0.0; \tag{33}$$

$$h_{sea}.cos(dev_{angle}) = h_{sea}; \tag{34}$$

$$radi_{gd[0]} = -radi.\; sin(dev_{angle}); \tag{35}$$

$$radi_{gd[1]} = 0.0 \tag{36}$$

$$radi_{gd[2]} = -radi.\; cos(dev_{angle}) - h_{sea}; \tag{37}$$

5.4. Quaternion update module

The quaternion represents the orientation of the body. This needs an updation every cycle based on the change of orientation in the last time quantum. Quaternion Update Consists of the following steps. The Miller method computation for Quaternion Update, Earth rate turn of reference frame computation, TOE turn of reference frame computation and Quaternion Normalization. These steps are illustrated in Section 5.4.1 to Section 5.4.3.

5.4.1. Miller method for quaternion update

In this module, one quaternion is taken as input and Δq is calculated by using the miller algorithm and outputs a quaternion which is the product of input quaternion and Δq. For miller 2 samples method,

$$\beta = \alpha_{sum} + (2.0 / 3.0) \cdot (\alpha_1 * \alpha_2); \tag{38}$$

$$\alpha_{sum} = \alpha_1 + \alpha_2 \tag{39}$$

For miller 3 samples method,

$$\beta = \alpha_{sum} + 0.45(\alpha * \alpha_3) + 0.675\, \alpha_2 * (\alpha_3 - \alpha_1) \tag{40}$$

$$\alpha_{sum} = \alpha_1 + \alpha_2 + \alpha_3 \tag{41}$$

For miller 4 samples method,

$$\beta = t_1 + t_2 + t_3 + t_4; \tag{42}$$

$$t_1 = \alpha_{sum} = \alpha_1 + \alpha_2 + \alpha_3 + \alpha_4 \tag{43}$$

$$t_2 = (54.0 /105.0)\, (\alpha_1 * \alpha_4) \tag{44}$$

$$t_3 = (92.0 /105.0)\, (b_1.t_{51} + b_2.t_{52}) \tag{45}$$

$$t_{51} = \alpha_1 * \alpha_3; \tag{46}$$

$$t_{52} = \alpha_2 * \alpha_4 \tag{47}$$

$$t_4 = (214.0 /105.0)\, (\alpha_1.t_{61} + \alpha_2.t_{62} + \alpha_3.t_{63}) \tag{48}$$

$$t_{61} = \alpha_1 * \alpha_2; \tag{49}$$

$$t_{62} = \alpha_2 * \alpha_3 \tag{50}$$

$$t_{63} = \alpha_3 * \alpha_4; \tag{51}$$

$$\alpha_1 + \alpha_2 + \alpha_3 = 1 \tag{52}$$

$$b_1 + b_2 = 1 \text{ is the condition to follow} \tag{53}$$

$$\alpha_{sum} = \text{sum of inc.angles} \tag{54}$$

5.4.2. Earth rate turn of reference frame and TOE turn of reference frame

This module takes quaternion update interval and two vectors namely, ω_{ned}, and TOE_{ned} as input and computes Δq. The output quaternion will be the product of the quaternion obtained after the miller algorithm and the Δq.

$$\beta = -(\omega_{ned} + TOE_{ned}).quaternion\ update\ time \tag{55}$$

$$\beta_{mag} = \sqrt{(\beta[0].\beta[0] + \beta[1].\beta[1] + \beta[2].\beta[2])} \tag{56}$$

$$c = cos\,(\beta_{mag} / 2.0) = 1 - t_1 + t_2 \tag{57}$$

$$t_1 = \beta^2_{mag} / 8.0; \tag{58}$$

$$t_2 = \beta^4_{mag} / 480.0;. \tag{59}$$

$$s = (1.0 /\beta_{mag}) * sin\,(\beta_{mag} / 2.0) = 0.5 - t_1 \tag{60}$$

where $t_1 = \beta^2{}_{mag} /48.0;$

$$\Delta q = (c,\ s.\ \beta[0],\ s.\ \beta[1].s.\ \beta[2])^T \tag{61}$$

$$q(updated) = \Delta q * q(previous) \tag{62}$$

$$\Delta q:\ Quaternion\ from\ ned\_new\ to\ ned\_old \tag{63}$$

5.4.3. Quaternion normalization

This module takes one quaternion as input and outputs one quaternion which is the normalized form of input quaternion.

$$q0 = q0 / qmag; \tag{64}$$

$$q1 = q1 / qmag; \tag{65}$$

$$q2 = q2 / qmag; \tag{66}$$

$$q3 = q3 / qmag; \tag{67}$$

$$where\ qmag = sqrt(q_0^2 + q_1^2 + q_2^2 + q_3^2) \tag{68}$$

5.5. Velocity update module

The velocity of the body in three axes needs to be updated based on the change of velocity in the last time quantum.

$$Inc\ (ned\ sensed) = dcm(body\ to\ ned).Inc\ vel(body) \tag{69}$$

$$Inc\ vel(ned\ inertial) = inc\ vel(ned\ sensed) + gravity(nedd) * delt\ nav \tag{70}$$

$$Inc\ vel\ (ned\ rotating) = inc\ vel(ned\ inertial) - corilolisaccn * delta\ nav - cpa.del\ nav \tag{71}$$

$$Velocity(new) = velocity(old) + inc\ vel(rotating) \tag{72}$$

$$velocity(average) = [velocity(old) + velocity(new)] / 2.0 \tag{73}$$

$$Inc\ velocities(NED\ rotating) = Inc\ velocities(NED\ inertial) - Coriolisaccn.delta\_nav - cpa.delta\ nav \tag{74}$$

$$Dynamic\ Pressure(Q) = 0.5 * \rho * v^2 \tag{75}$$

$$velocity(new) = velocity(old\ or\ beginning) + Inc\ vel(NED\ rotating) \tag{76}$$

$$Average\ velocity = (velocity\ new + velocity\ old) / 2.0 \tag{77}$$

$$t_1 = (3/2) * j2 * (R_e / R_m)^2 * (3cos^2(\Phi) - 1.0) \tag{78}$$

$$t_2 = 2 * j3 * (R_e / R_m)^3 * (5\ cos^3(\Phi) - 3\ cos(\Phi)) \tag{79}$$

$$t_3 = (5/8) * j4 * (R_e/R_m)^4 * (35cos^4(\Phi) - 30cos^2(\Phi) + 3) \tag{80}$$

$$G_r = (-\mu / r^2)[1.0 - t_1 - t_2 - t_3],; \tag{81}$$

$$t_4 = j2 \tag{82}$$

$$t_5 = (j3/2) * (R_e / R_m) * (5\ cos^2(\Phi) - 1) / cos(\Phi) \tag{83}$$

$$t_6 = (5/6) * j4 * (R_e / R_m)^2 * (7cos^2(\Phi) - 3) \tag{84}$$

$$Gphi = 3 * (\mu/r^2) * (R_e/R_m)^2 * \sin(\Phi)\ cos(\Phi)[t4 + t5 + t6] \tag{85}$$

$$\begin{bmatrix} GnedCentric[0] \\ GnedCentric[1] \\ GnedCentric[2] \end{bmatrix} = \begin{bmatrix} -Gphi \\ 0.0 \\ -Gr \end{bmatrix} \tag{86}$$

$$\begin{bmatrix} \text{g\_ned\_c (north)} \\ \text{g\_ned\_c (east)} \\ \text{g\_ned\_c (down)} \end{bmatrix} = \text{Dcm(centric to detic)} \begin{bmatrix} \text{g\_ned\_c (north)} \\ \text{g\_ned\_c (east)} \\ \text{g\_ned\_c (down)} \end{bmatrix} \tag{87}$$

$$\text{Dcm(centric to detic)} = \begin{bmatrix} \cos(theta) & 0 & -sin(theta) \\ 0 & 1 & 0 \\ sin(theta) & 1 & -cos(theta) \end{bmatrix} \tag{88}$$

$$\theta = -deviationangle; \tag{89}$$

$$devitationangle = \lambda_d - \lambda_c \tag{90}$$

$$Gravity\ model\ in\ NED\ geodetic = dcm * gravity(geocentric) \tag{91}$$

$$Inc\ velocities(NED\ inertial) = Inc\ velocities(NED\ sensed) + gravity(NED\ detic).delt\_nav \tag{92}$$

$$Inc\ velocities(NED\ rotating) = Inc\ velocities(NED\ inertial) - coriolisaccn.delt\_nav - cpa.delt\_nav \tag{93}$$

$$velocity(new) = velocity(old\ or\ beginning) + Inc\ vel(NED\ rotating) \tag{94}$$

$$Average\ velocity = (velocity\ new + velocity\ old)/2.0 \tag{95}$$

5.6. Position update module

This module considers the computations done in the earth parameters computation module and updates the position in both Cartesian and polar coordinates position in Cartesian coordinates,

$$f_1 = R_n / (R_n + h_{sea}) \tag{96}$$

$$\delta_{pos(N)} = velocity(average)north * \delta_{nav} * f_1 \tag{97}$$

$$position(N) = position\ old(N) + \delta_{pos(N)} \tag{98}$$

$$f_2 = R_e.cos(\lambda_d) / [(R_e + h_{sea})cos(\lambda_d)] = R_e /(R_e + h_{sea}) \tag{99}$$

$$\delta_{pos(E)} = velocity(average)east * delt\_nav * f_2 \tag{100}$$

$$position(E) = position\ old(E) + \delta_{nav} \tag{101}$$

$$\delta_{pos(D)} = velocity(average)down * \delta_{nav} \tag{102}$$

$$position(D) = position\ old(N) + \delta_{pos(N)} \tag{103}$$

position in polar coordinates,

$$\delta_\lambda = \delta_{pos(n)} / R_n \tag{104}$$

$$\lambda_d = \lambda(previous) + \delta_\lambda \tag{105}$$

$$\delta_\mu = \delta_{pos(e)} / [R_e.cos(\lambda_d)] \tag{106}$$

$$\mu = \mu(previous) + \delta_\mu \tag{107}$$

$$Height(from\ sea\ level : h_{sea}) = launch\ height ((from\ sea\ level : lh_{sea}) + platform\ (h\_pl) \tag{108}$$

$$platform\ height(h\_pl) = -position(down) \tag{109}$$

$$h_{sea} = lh_{sea} + platform\ height \tag{110}$$

$$h_{sea} = lh_{sea} + h\_pl \tag{111}$$

$$\lambda_d = 17.257658\ d \tag{112}$$

$$\mu = 7.499978\ d \tag{113}$$

$$h_{sea} = 560m \tag{114}$$

$$R_b = 63781370.0; \tag{115}$$

$$R_b = 6357752.3142 \tag{116}$$

$$\omega_0 = 7.292115e-5 \tag{117}$$

5.7. Nav utilities module

We have classified some computations as utility modules. These modules will be used across all other modules.

5.7.1. Quaternion update module

The module is named as *QuatMul*. This module takes input as two quaternions and outputs one quaternion, which is a product of both of the input quaternions.

$$cq = aq * bq \tag{118}$$

$$cq[0] = aq[0].bq[0] - aq[1].bq[1] - aq[2].bq[2] - aq[3].bq[3] \tag{119}$$

$$cq[1] = aq[0].bq[1] - aq[1].bq[0] - aq[2].bq[3] - aq[3].bq[2] \tag{120}$$

$$cq[2] = aq[0].bq[2] - aq[2].bq[0] - aq[3].bq[1] - aq[1].bq[3] \tag{121}$$

$$cq[3] = aq[0].bq[3] - aq[3].bq[0] - aq[1].bq[2] - aq[2].bq[1] \tag{122}$$

5.7.2. Quaternion inverse module

This module takes one quaternion as input and outputs one quaternion, which will be the inverse of the input quaternion.

$$aq[0] = bq[0] \tag{123}$$

$$aq[1] = -bq[1]; \tag{124}$$

$$aq[2] = -bq[2] \tag{125}$$

$$aq[3] = -bq[3]; \tag{126}$$

5.7.3. Vector multiplication module

This module takes input as two vectors and outputs one vector, which is a product of both the input vectors.

$$v3 = v1Xv2; \tag{127}$$

$$v3[0] = v1[1].v2[2] - v1[2].v2[1]; \tag{128}$$

$$v3[1] = v1[2].v2[0] - v1[0].v2[2]; \tag{129}$$

$$v3[2] = v1[0].v2[1] - v1[1].v2[0]; \tag{130}$$

5.7.4. Vector scale module

This module takes one vector and one constant as input and the output will be the multiplication of this constant value to each term in the vector.

$$v = v * A \tag{131}$$

$$v[0] = v[0] * A \tag{132}$$

$$v[0] = v[1] * A \tag{133}$$

$$v[0] = v[2] * A \tag{134}$$

5.7.5. Vector addition module

This module takes input as two vectors and outputs one vector, which is a sum of both the input vectors.

$$v3 = v1 + v2 \tag{135}$$

$$v3[0] = v1[0] + v2[0] \tag{136}$$

$$v3[1] = v1[1] + v2[1] \tag{137}$$

$$v3[2] = v1[2] + v2[2] \tag{138}$$

5.7.6. Vector subtraction module

This module takes input as two vectors and outputs one vector, which is a difference of both the input vectors.

$$v3 = v1 - v2; \tag{139}$$

$$v3[0] = v1[0] - v2[0] \tag{140}$$

$$v3[1] = v1[1] - v2[1]; \tag{141}$$

$$v3[2] = v1[2] - v2[2] \tag{142}$$

5.7.7. DCM multiplication module

This module takes input as one vector and DCM matrix and outputs one vector, which is the product of the input vector and the DCM matrix.

$$v2 = dcm12.v1 \tag{143}$$

$$v2[0] = dcm12[0][0].v1[0] + dcm12[0][1].v1[1] + dcm12[0][2].v1[2] \tag{144}$$

$$v2[1] = dcm12[1][0].v1[0] + dcm12[1][1].v1[1] + dcm12[1][2].v1[2] \tag{145}$$

$$v2[2] = dcm12[2][0].v1[0] + dcm12[2][1].v1[1] + dcm12[2][2].v1[2] \tag{146}$$

5.7.8. DCM from quaternion module

This module generates the DCM corresponding to an input quaternion. The computation is given in the matrix mentioned in the below matrix.

$$\mathrm{dcm} = \begin{vmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2(q_1q_2 + q_0q_3) & 2(q_1q_3 - q_0q_2) \\ 2(q_1q_2 - q_0q_3) & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2(q_2q_3 + q_0q_1) \\ 2(q_1q_3 + q_0q_2) & 2(q_2q_3 - q_0q_1) & q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{vmatrix} \tag{147}$$

5.7.9. Copy quaternion module

This module takes one quaternion as input and outputs one quaternion, which will be the replica of the input quaternion.

$$bq[0] = aq[0] \tag{148}$$

$$bq[1] = aq[1] \tag{149}$$

$$bq[2] = aq[2] \tag{150}$$

$$bq[3] = aq[3] \tag{151}$$

5.7.10. Copy vector module

This module takes one quaternion as input and outputs one quaternion, which will be the replica of the input quaternion.

$$v2 = v1 \tag{152}$$

$$v1[0] = v2[0] \tag{153}$$

$$v1[1] = v2[1] \tag{154}$$

$$v1[2] = v2[2] \tag{155}$$

6. Results

The ModNav is a modular and scalable navigation framework. In the full-scale configuration, it works like the original navigation algorithm that generates the same output and requires the same execution times. It can be scaled down from the full-scale configuration based on the requirements of the application. The scaling down can be done to a minimal configuration, where the framework requires minimum execution time. Comparative analysis of the ModNav framework results is carried out with reference to the original implementation. The comparison is done from two perspectives: one is the output accuracy and the other is the execution time requirements. The accuracy of the output generated by the framework is compared with the original implementation in the full-scale and minimal configuration. The comparison is carried out for all the navigational output parameters viz. positions, velocities, quaternions, rates, and accelerations. It is observed that the results of the framework are exactly matching with the original implementation of the algorithm. This is shown in Figures 4, 5, 6, 7. In the full-scale configuration, it generates all the output parameters, and in the scaled-down configurations, it generates the limited output parameters for which it is configured. The framework is tested for execution time in both full-scale, minimal configurations on high-end and low-end processor hardware. For high-end hardware, we have used Zynq-7000 processor-based embedded hardware running at 600 MHz. For low-end hardware, we have used Arduino-Uno running at 16MHz. The execution time measurements of the framework in full-scale configuration and minimal configurations on both of the hardware are presented in Figures 8 and 9. This shows that the navigation framework in a scaled-down configuration can be made to run on resource-constrained embedded hardware, which was not possible with the original implementation.

6.1. Discussion

The input to the Nav algorithm is the δ_θ, δ_v. These are measured by the sensors (accelerometers, gyros) for each time-quantum and sent as input to the algorithm. δ_θ, δ_v is the measurement output of the accelerometer and gyro, respectively. Since three sets of sensors will be present in the system, accordingly three sets of δ_θ, δ_v will be generated for every time quantum. For testing of the framework, we have collected an input data set of δ_θ, δ_v. with an interval time (time quantum) of 2.5 m-s for a duration of 500 s. So, the total input data set size is 50 k samples. Each sample consists of seven parameters: the time, three sets of δ_θ, & three sets of δ_v. With this input data, six experiments are carried out for comparative analysis of the results. The details of the experimental runs are given in Table 2. The first two runs are

Figure 4
Navigation output (quaternion) in fullscale configuration

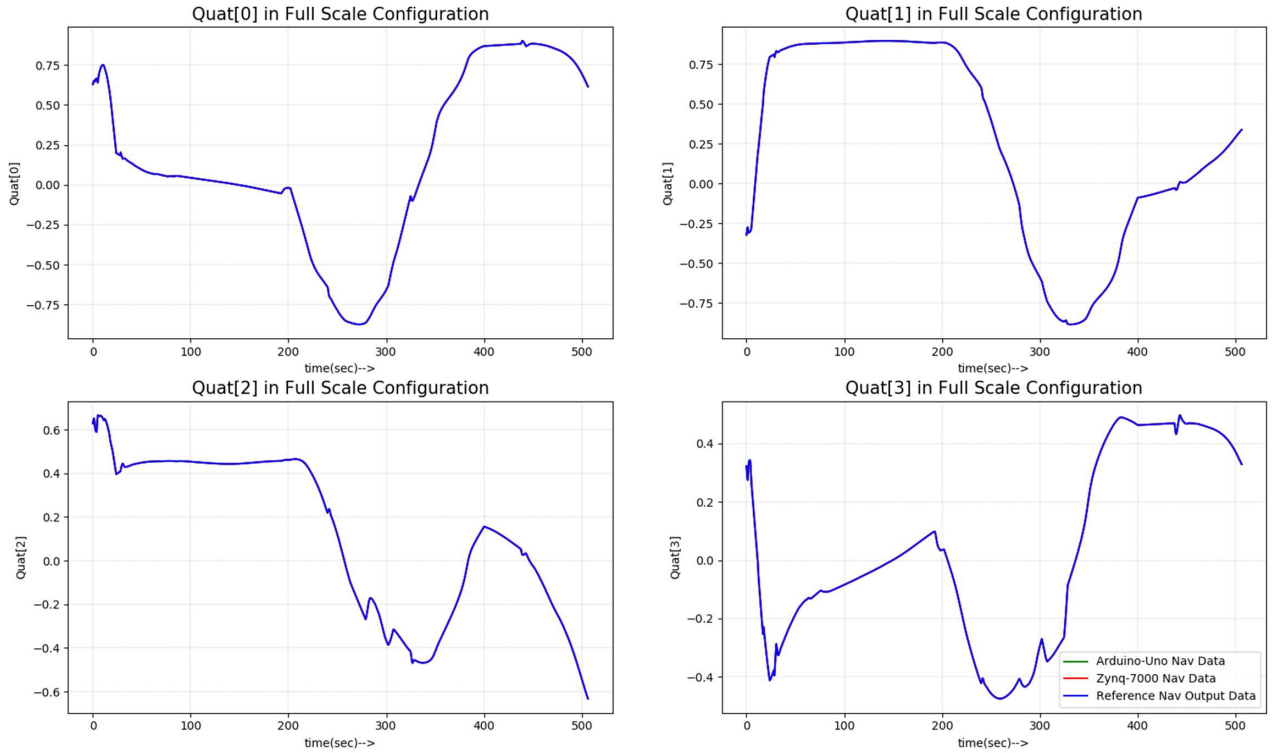


Figure 5
Navigation output (velocities and positions) in fullscale configuration

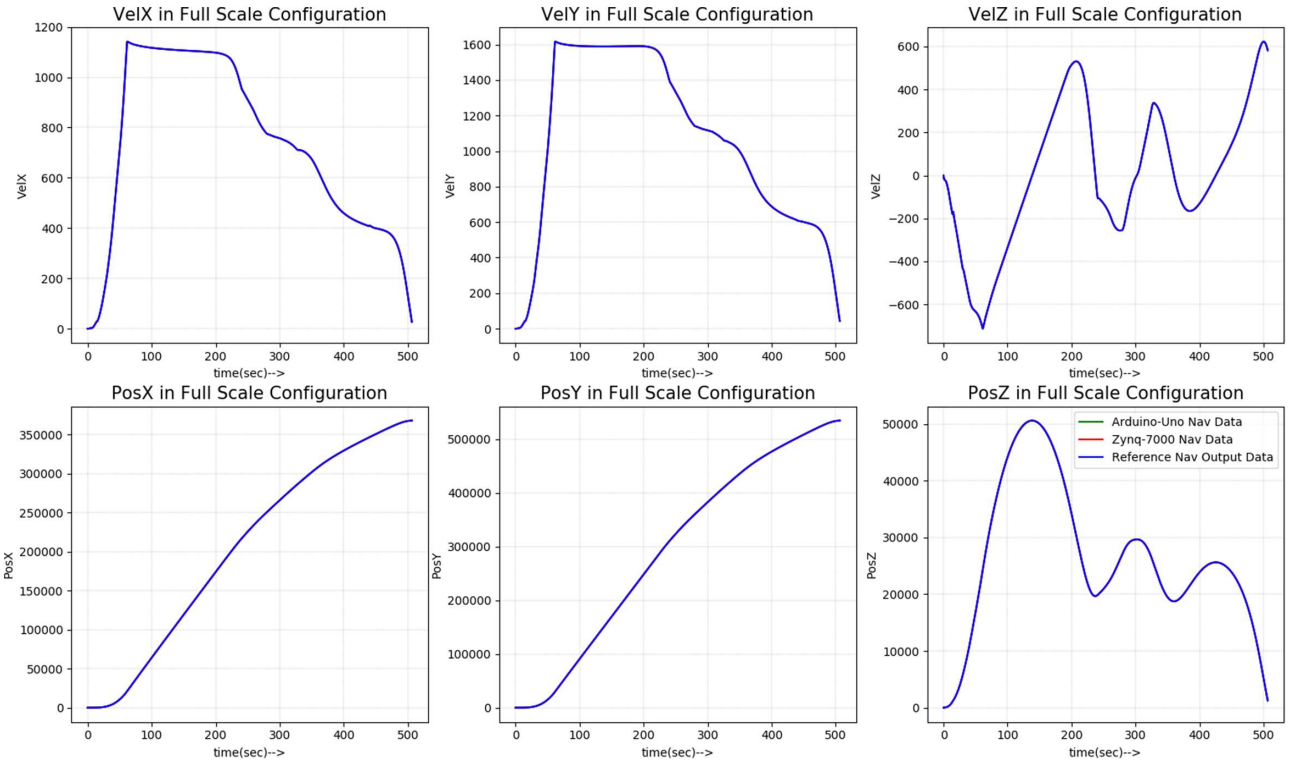


Figure 6
Navigation output (quaternion) in minimal configuration

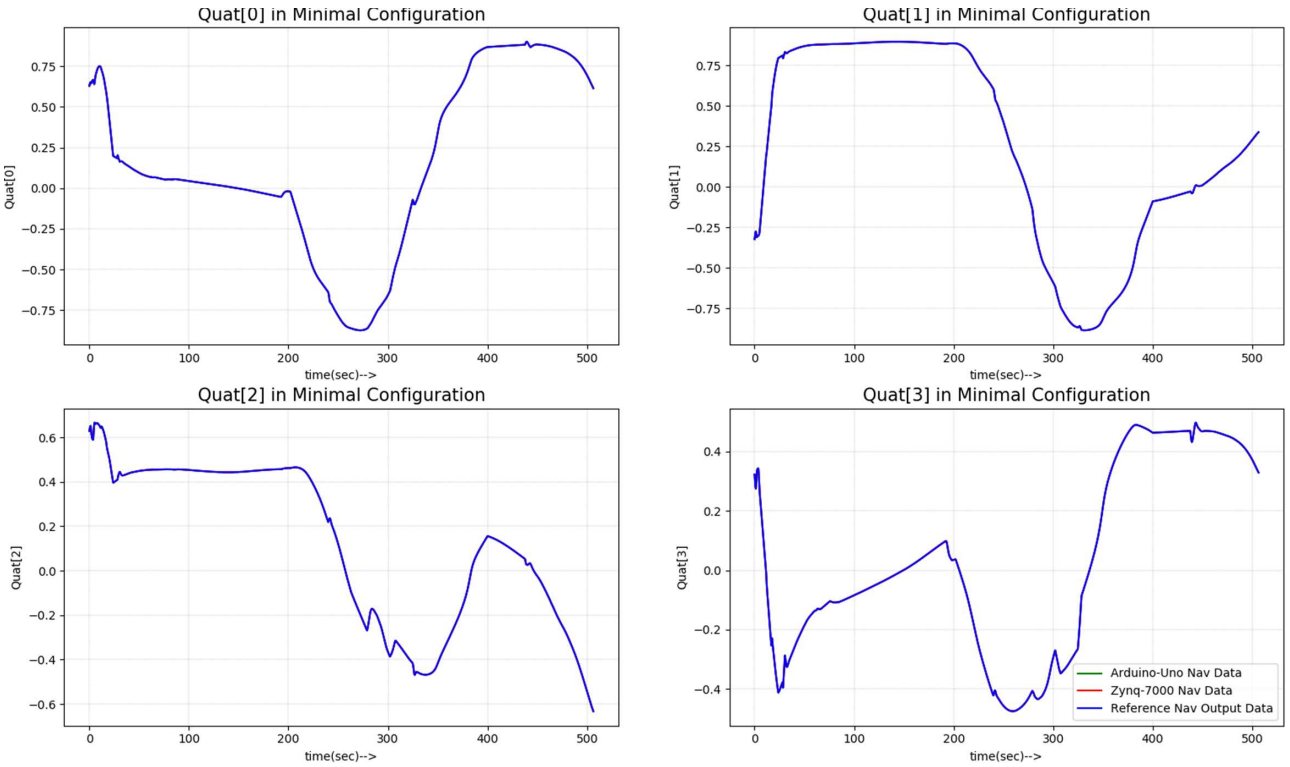


Figure 7
Navigation output (velocities and positions) in minimal configuration

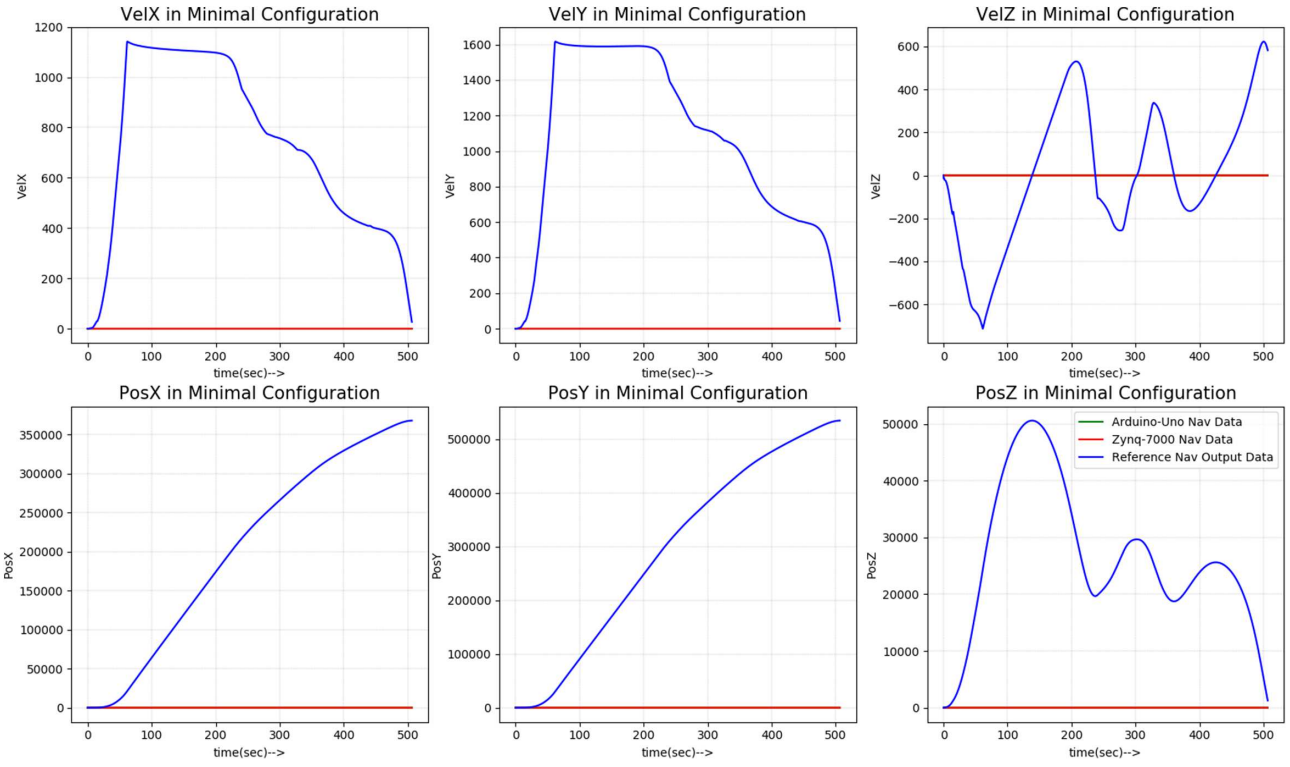


Figure 8
Execution times in fullscale, minimal configuration for high-end and low-end hardware

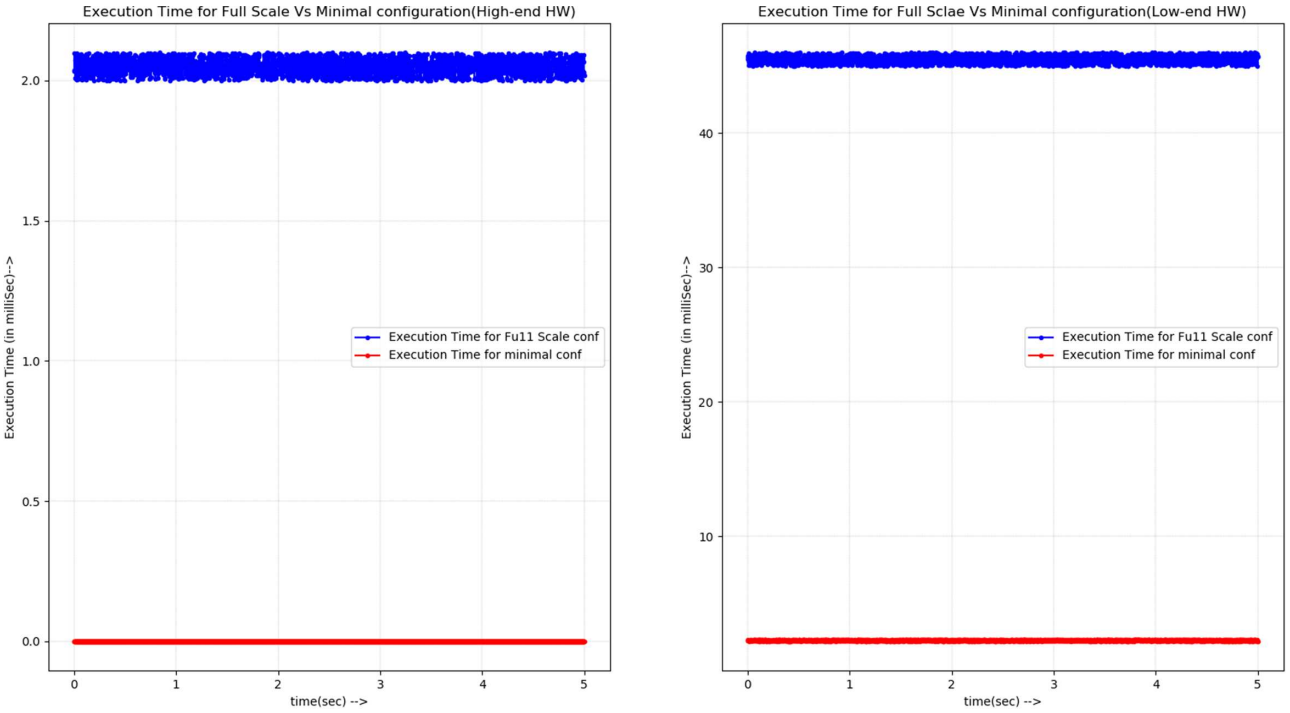


Figure 9
Execution times of high-end and low-end hardware with MOD-NAV configuration

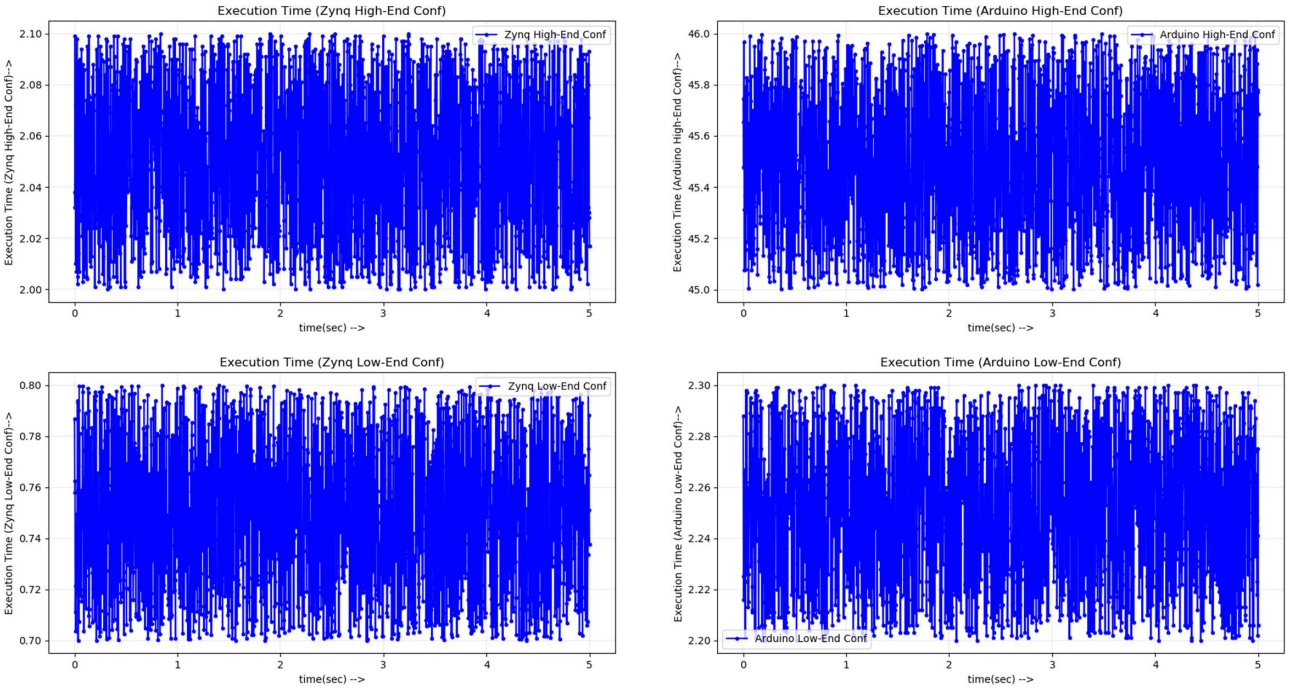


Table 2
Details of experimental runs

S.No.	Experimental run configuration	Hardware used
1	Original Configuration	Zynq-7000
2	Original Configuration	Arduino-Uno
3	ModNav in Full-Scale Configuration	Zynq-7000
4	ModNav in Minimal Configuration	Zynq-7000
5	ModNav in Full-Scale Configuration	Arduino-Uno
6	ModNav in Minimal Configuration	Arduino-Uno

given with the original implementation: one with Zynq hardware and the other with arduino hardware. Two runs are given with ModNav in full-scale configuration. Each run corresponds to a different hardware. Accordingly, another two runs are given with ModNav in minimal configuration. In the minimal configuration, we have configured the framework to generate the orientation data only. Positions and velocities computations are discarded.

6.2. Navigation output results (fullscale configuration)

The applications requiring position, velocity & quaternion as output with maximum accuracy have to run the full-scale configuration of the framework. The framework takes δ_θ, δ_v as continuous input data for every cycle and generates the positions, velocities, and orientation at the end of each navigation cycle. This configuration requires a high-end processor for reliable output generation. Running the full-scale framework on a low-end processor requires a higher time quantum. On Arduino-Uno, this takes around 45–46 m-s for the execution of each navigation cycle. If the navigation data periodicity demand of the application is more than 46-m-s, then it can be used on low-end hardware also; otherwise, by customization of the framework, with reduced accuracy, it can be executed. The plots mentioned in Figure 5 correspond to the positions (x, y, z), velocities (V_x, V_y, V_z) generated by the framework in fullscale configuration on high-end, low-end hardware compared with that of the reference navigation output. Overlapped plots of the three sets of the results are given. The three sets of output are matching one to one, so that, the curve looks as single curve, even though three curves are present in the plot. Similarly, the quaternions are mentioned in Figure 4. The generated navigation output matches the reference output exactly, so the plots overlapped. While running on low-end hardware, the execution time crosses the time quantum because of the CPU-time resource constraints. To overcome that, we have passed the time from the simulated timer variable instead of real-time. On completion of each navigation cycle, we increment the timer variable with the time-quantum and this timer variable will be considered for all the timing requirements in the algorithm. In cases where the execution time is less than the time quantum, the real-time or simulated time will be the same, so either can be used for timing generation.

6.3. Navigation output results (minimal configuration)

In this minimal configuration, only the orientation computation is enabled in the ModNav framework configuration. Position and velocity updates are eliminated as part of the configuration. This configuration is useful for a non-moving body with only orientation change; for example, a robotic arm whose position or velocity does not change because it is fixed at one position, but its orientation may change as part of its activity. In this configuration, only the quaternion computation will be enabled. Since position and velocity computations are not present, they remain zero. The plots mentioned in Figure 7 correspond to the positions (x, y, z) and velocities (V_x, V_y, V_z), generated by the framework in the minimal configuration on high-end, low-end hardware, compared with that of the reference navigation output. Here, it can be observed that the generated positions and velocities are zero because we have eliminated the position and velocity computation as part of the configuration. The reference output will have computed positions and velocities as the curve but the ModNav output will be zero, because this computation is eliminated in the configuration. The comparison of the generated quaternion and reference quaternion is given in the plot of Figure 6. Here, it can be observed that the orientation perfectly matches the reference output because the quaternion computation is enabled in the configuration.

6.4. Execution time results

The execution times are measured in all the six experimental runs mentioned in Table 2. The execution times of the original implementation on both of the hardware are given in Table 3. The zynq hardware can execute each of the navigation cycle within 2–2.1 m-s. However, the Arduino takes around 45–46 m-s for the

Table 3
Execution times for original implementation

Configuration	Hardware	Computational need/executiontime
Full-Scale Conf	High-end (Zynq-7000)	2–2.1 m-s
Minimal-Conf	High-end (Zynq-7000)	700–800 micro-sec
Full-Scale Conf	Low-end (Arduino-Uno)	45–46 m-s
Minimal-Conf	Low-end (Arduino-Uno)	2.2–2.3 m-s

Table 4
Execution times with MOD-NAV implementation

Configuration	Hardware	Computational need/execution time
Original implementation	Zynq-7000	2–2.1 m-s
Original implementation	Arduino-Uno	45–46 m-s

same execution. If the navigation time-quantum is 2.5 m-s, then it is feasible to run the original algorithm only on the zynq hardware and not feasible on the resource constrained Arduino hardware.

The execution time of the framework in full-scale, minimal configuration with both of the hardware configurations is given in Table 4. The ModNav in Full-Scale configuration takes around 2–2.1 m-s on zynq hardware and same things takes 45–46 m-s on Arduino hardware. This is same as the execution times of the original implementation on both of the hardware. Since the original implementation is like the full-scale configuration of the Modular framework, both take the same execution times. In the minimal configuration, it takes the execution time of 700–800 micro-sec on zynq and 2.2–2.3 m-s on Arduino. This shows that the ModNav framework in minimal configuration can be executed even on resource-constrained embedded hardware for all practical nav time-quantum. The comparison of the execution times in full-scale and minimal configuration on both of the hardware is given in Figure 8. The instantaneous measurement of the execution times is given in Figure 9. This is given for a duration of 5-s. Rest of the duration of the 500-s of the run follows the same pattern of execution time. Here, the execution time variations are depicted; the band of variation is already presented with corresponding numbers in Tables 3 and 4. With this analysis, we conclude that the inertial nav algorithm can be made to execute even on resource-constrained embedded hardware with the proposed modularization and configuration, which is not feasible with the original implementation.

7. Conclusion

A proposal for the modularization of the inertial navigation algorithms has been presented in this work. The modularization framework & its implementation is described. The modularization facilitates in scaling up or down of the framework based on the application requirements. For a full-fledged navigation application, the entire framework can be used; for resource-constrained hardware, the framework can be scaled down to a minimal configuration and can be used for navigation. Thereby, this modularization ensures that the navigation framework runs even on low-end computational hardware like Arduino-Uno. Each of the modules with its inputs, outputs, and functionality has been presented. We have tested the framework in its full-scale configuration, minimal configuration on high-end embedded hardware consisting of a Zynq-7000 processor and low-end hardware consisting of Arduino-Uno. The navigation output results of the four configurations and execution time analysis for both hardware have been presented. The navigation output on both of the hardware is observed to be exactly matching the reference output; the execution time is observed to be appropriate to the computational capability of the processors. The minimal configuration framework is proved to be running, even on a resource-constrained Arduino-Uno-based embedded hardware.

Ethical Statement

This study does not contain any studies with human or animal subjects performed by any of the authors.

Conflicts of Interest

The authors declare that they have no conflicts of interest to this work.

Data Availability Statement

Data are available from the corresponding author upon reasonable request.

Author Contribution Statement

Mahender Katukuri: Conceptualization, Methodology, Software, Writing – original draft, Writing – review & editing, Visualization. **Satyanarayana JV:** Validation, Formal analysis, Data curation, Supervision, Project administration. **Ruchir Gupta:** Validation, Investigation, Resources, Supervision, Project administration. **Bhaskar Biswas:** Validation, Investigation, Resources, Supervision, Project administration.

References

[1] Li, K., Bu, S., Li, J., Xia, Z., Wang, J., & Li, X. (2025). Distributed relative pose estimation for multi-UAV systems based on inertial navigation and data link fusion. *Drones*, *9*(6), 405. https://doi.org/10.3390/drones9060405

[2] Wang, Y., Yang, Q., Cui, H., & Fang, H. (2025). Relative localization with non-persistent excitation using UWB-IMU measurements. *IEEE Transactions on Automation Science and Engineering*, *22*, 7092–7104. https://doi.org/10.1109/TASE.2024.3460811

[3] Wang, Y., Wang, Q., Hao, Z., & Chen, P. (2025). An integrated navigation method based on the strapdown inertial navigation system/scene-matching navigation system for UAVs. *Sensors*, *25*(11), 3379. https://doi.org/10.3390/s25113379

[4] Deliparaschos, K. M., Loizou, S. G., & Zolotas, A. C. (2024). A modular UAV hardware platform for aerial indoor navigation research and development. In *2024 International Conference on Unmanned Aircraft Systems*, 398–404. https://doi.org/10.1109/ICUAS60882.2024.10557007

[5] Zhu, H., Yao, Y., Ma, X., & Zhang, Q. (2024). Tightly coupled multi-frequency PPP-RTK/INS integration model and its application in an urban environment. *Geo-spatial Information Science*, *27*(5), 1685–1699. https://doi.org/10.1080/10095020.2023.2251535

[6] Cohen, N., & Klein, I. (2024). Inertial navigation meets deep learning: A survey of current trends and future directions. *Results in Engineering*, *24*, 103565. https://doi.org/10.1016/j.rineng.2024.103565

[7] Mei, C., Fan, Z., Zhu, Q., Yang, P., Hou, Z., & Jin, H. (2023). A novel scene matching navigation system for UAVs based on vision/inertial fusion. *IEEE Sensors Journal*, *23*(6), 6192–6203. https://doi.org/10.1109/JSEN.2023.3241330

[8] Mohsan, S. A. H., Othman, N. Q. H., Li, Y., Alsharif, M. H., & Khan, M. A. (2023). Unmanned aerial vehicles (UAVs): Practical aspects, applications, open challenges, security issues, and future trends. *Intelligent Service Robotics*, *16*(1), 109–137. https://doi.org/10.1007/s11370-022-00452-4

[9] Saha, S. S., Du, Y., Sandha, S. S., Garcia, L. A., Jawed, M. K., & Srivastava, M. (2023). Inertial navigation on extremely resource-constrained platforms: Methods, opportunities, and challenges. In *2023 IEEE/ION Position, Location and Navigation Symposium* (pp. 708–723). https://doi.org/10.1109/PLANS53410.2023.10139997

[10] Cucci, D. A., Voirol, L., Khaghani, M., & Guerrier, S. (2023). On performance evaluation of inertial navigation systems: The case of stochastic calibration. *IEEE Transactions on Instrumentation and Measurement*, *72*, 8502417. https://doi.org/10.1109/TIM.2023.3267360

[11] Pritzl, V., Vrba, M., Štěpán, P., & Saska, M. (2022). Cooperative navigation and guidance of a micro-scale aerial vehicle by an accompanying UAV using 3D LiDAR relative localization. In *2022 International Conference on Unmanned Aircraft Systems*, 526–535. https://doi.org/10.1109/ICUAS54217.2022.9836116

[12] Indraganti, V., Gupta, A., & Sharma, R. (2022). Attitude estimation of HAPs using fused global positioning system and inertial navigation system. In *2022 8th International Conference on Signal Processing and Communication*, 54–57. https://doi.org/10.1109/ICSC56524.2022.10009474

[13] Kulkarni, D., Narkhede, G. G., & Motade, S. N. (2022). SENSOR FUSION: An advanced inertial navigation system using GPS and IMU. In *2022 6th International Conference on Computing, Communication, Control and Automation*, 1–5. https://doi.org/10.1109/ICCUBEA54992.2022.10010952

[14] Singh, B., Menghal, P. M., & Suri, N. (2022). Integration of INS and GPS for radar-aided inertial navigation system. In *2022 IEEE 7th International Conference for Convergence in Technology*, 1–5. https://doi.org/10.1109/I2CT54291.2022.9824660

[15] Hadjiloizou, L., Deliparaschos, K. M., Makridis, E., & Charalambous, T. (2022). Onboard real-time multi-sensor pose estimation for indoor quadrotor navigation with intermittent communication. In *2022 IEEE Globecom Workshops* (pp. 154–159). https://doi.org/10.1109/GCWkshps56602.2022.10008590

[16] Li, J., Yang, G., Cai, Q., Niu, H., & Li, J. (2022). Cooperative navigation for UAVs in GNSS-denied area based on optimized belief propagation. *Measurement*, *192*, 110797. https://doi.org/10.1016/j.measurement.2022.110797

[17] Li, X., Li, X., Li, S., Zhou, Y., Sun, M., Xu, Q., & Xu, Z. (2022). Centimeter-accurate vehicle navigation in urban environments with a tightly integrated PPP-RTK/MEMS/vision system. *GPS Solutions*, *26*(4), 124. https://doi.org/10.1007/s10291-022-01306-3

[18] Fesenko, O. D., Bieliakov, R. O., Radzivilov, H. D., Sasin, S. A., Borysov, O. V., Borysov, I. V., . . . , & Kovalchuk, O. O. (2022). Method of improving the accuracy of navigation MEMS data processing of UAV inertial navigation system. *Radio Electronics, Computer Science, Control*, *3)*, 196. https://doi.org/10.15588/1607-3274-2022-3-18

[19] Patel, U. N., & Faruque, I. A. (2022). Multi-IMU based alternate navigation frameworks: Performance and comparison for UAS. *IEEE Access*, *10*, 17565–17577. https://doi.org/10.1109/ACCESS.2022.3144687

[20] Gu, S., Dai, C., Fang, W., Zheng, F., Wang, Y., Zhang, Q., . . . , & Niu, X. (2021). Multi-GNSS PPP/INS tightly coupled integration with atmospheric augmentation and its application in urban vehicle navigation. *Journal of Geodesy*, *95*(6), 64. https://doi.org/10.1007/s00190-021-01514-8

[21] Klein, K. T. J., Uysal, F., Cuenca, M. C., Otten, M. P. G., & de Wit, J. J. M. (2021). Radar-aided navigation system for small drones in GPS-denied environments. In *2021 IEEE Radar Conference*, 1–6. https://doi.org/10.1109/RadarConf2147009.2021.9455317

[22] Lai, X., & Yang, F. (2021). A robust Kalman filter for SINS/GPS integrated navigation. In *2021 IEEE 4th Advanced Information Management, Communications, Electronic and Automation Control Conference*, 238–242. https://doi.org/10.1109/IMCEC51613.2021.9482280

[23] Wei, X., Li, J., Feng, K., Zhang, D., Li, P., Zhao, L., & Jiao, Y. (2021). A mixed optimization method based on adaptive Kalman filter and wavelet neural network for INS/GPS during GPS outages. *IEEE Access*, *9*, 47875–47886. https://doi.org/10.1109/ACCESS.2021.3068744

[24] Herath, S., Yan, H., & Furukawa, Y. (2020). RoNIN: Robust neural inertial navigation in the wild: Benchmark evaluations & new methods. In *2020 IEEE International Conference on Robotics and Automation*, 3146–3152. https://doi.org/10.1109/ICRA40945.2020.9196860

[25] Chen, C., Zhao, P., Lu, C. X., Wang, W., Markham, A., & Trigoni, N. (2020). Deep-learning-based pedestrian inertial navigation: Methods, data set, and on-device inference. *IEEE Internet of Things Journal*, *7*(5), 4431–4441. https://doi.org/10.1109/JIOT.2020.2966773

[26] Hamdy, A., Ouda, A. N., Kamel, A. M., & Elhalwagy, Y. Z. (2020). Land vehicle navigation algorithm implementation on Cortex-M4 embedded processor. In *2020 12th International Conference on Electrical Engineering*, 343–349. https://doi.org/10.1109/ICEENG45378.2020.9171718

[27] Ren, J. X., Zi, J. L., Guan, H. Y., & Li, J. (2020). Design of an ultra-tightly coupled integrated INS/GPS navigation system based on UPF. In *2020 27th Saint Petersburg International Conference on Integrated Navigation Systems*, 1–4. https://doi.org/10.23919/ICINS43215.2020.9133853

[28] Li, D., Wu, Y., & Zhao, J. (2020). Novel hybrid algorithm of improved CKF and GRU for GPS/INS. *IEEE Access*, *8*, 202836–202847. https://doi.org/10.1109/ACCESS.2020.3035653

[29] Wang, R. (2020). IMM-EKF based GPS/INS combined positioning method for drone. In *2020 International Conference on Computing and Data Science*, 160–164. https://doi.org/10.1109/CDS49703.2020.00039

How to Cite: Katukuri, M., JV, S., Gupta, R., & Biswas, B. (2026). MOD-NAV: Scalable Modular Inertial Navigation Framework for Resource-Constrained Embedded Hardware. *Journal of Computational and Cognitive Engineering*. https://doi.org/10.47852/bonviewJCCE62026975