An Instruction Guide to GBEES:
Grid-based Bayesian Estimation Exploiting Sparsity

October 4, 2024

Authored by:

Mr. Benjamin L. Hanson

The University of California San Diego

Dept. of Mechanical and Aerospace Engineering

Contents

---

1 Introduction

Consider the state estimation of the nonlinear system

$$\frac{d\boldsymbol{x}}{dt} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{w}), \quad \boldsymbol{y} = \boldsymbol{h}(\boldsymbol{x}, \boldsymbol{v}), \tag{1}$$

where $\boldsymbol{x}$ is the state, $\boldsymbol{f}$ is the system dynamics, $\boldsymbol{w} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{Q})$ is the zero-mean, Gaussian process noise (where $\boldsymbol{Q}$ is the process noise covariance), $\boldsymbol{y}$ is a measurement, $\boldsymbol{h}$ is a measurement function, and $\boldsymbol{v} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{R})$ is the zero-mean Gaussian measurement noise (where $\boldsymbol{R}$ is the measurement noise covariance). The optimal solution to the time-varying uncertainty of a state $p_{\boldsymbol{x}}(\boldsymbol{x}', t)$ driven by the stochastic differential equation $d\boldsymbol{x}/dt$ lies in the Fokker-Planck equation:

$$\frac{\partial p_{\boldsymbol{x}}(\boldsymbol{x}', t)}{\partial t} = -\frac{\partial f_i(\boldsymbol{x}', t) p_{\boldsymbol{x}}(\boldsymbol{x}', t)}{\partial x_i'} + \frac{1}{2}\frac{\partial^2 q_{ij} p_{\boldsymbol{x}}(\boldsymbol{x}', t)}{\partial x_i' x_j'}, \tag{2}$$

where f_i is the i^{th} component of the system dynamics $\boldsymbol{f}$ at realization $\boldsymbol{x}'$ and time t, and q_{ij} is the $(i, j)^{\text{th}}$ element of $\boldsymbol{Q}$. In general, Equation (2) is non-integrable, and $p_{\boldsymbol{x}}(\boldsymbol{x}', t)$ cannot be described by a finite number of parameters. The optimal solution assimilates data via Bayes' theorem:

$$p_{\boldsymbol{x}}(\boldsymbol{x}', t_{k+}) = \frac{p_{\boldsymbol{y}}(\boldsymbol{y}_k | \boldsymbol{x}')\, p_{\boldsymbol{x}}(\boldsymbol{x}', t_{k-})}{C}, \tag{3}$$

where $p_{\boldsymbol{x}}(\boldsymbol{x}', t_{k+})$ is the *a posteriori*, $p_{\boldsymbol{y}}(\boldsymbol{y}_k|\boldsymbol{x}')$ is the likelihood, $p_{\boldsymbol{x}}(\boldsymbol{x}', t_{k-})$ is the *a priori*, and C is a normalization constant. Again, in general, the number of parameters necessary to represent $p_{\boldsymbol{x}}(\boldsymbol{x}', t_{k+})$ is not finite. Thus, the goal of the Recursive Bayesian Filter (RBF) is to approximate and propagate the full probability density function (PDF) with as few parameters as possible while incorporating information from measurements updates.

If $\boldsymbol{Q} = \boldsymbol{0}$, Equation (3) is hyperbolic. If $\boldsymbol{Q} > \boldsymbol{0}$, the equation changes type to elliptic. In practice, the deterministic part of the stochastic differential equation is dominant, thus $\boldsymbol{Q}$ is relatively small. The fluid mechanics community has spent considerable effort developing techniques for the numerical integration of these sorts of systems. Of these techniques, the most well-suited for uncertainty propagation is the Godunov-type finite volume method. This technique treats probability as a fluid, flowing it through discretized phase-space subject to the system dynamics, while considering the general conservation law. These numerical methods have been thoroughly tested for 2D and 3D applications, but marching higher-dimensional, discretized PDFs proves computationally expensive.

Enter Grid-based Bayesian Estimation Exploiting Sparsity (GBEES), an efficient numerical method for propagating 1D-6D, discretized PDFs subject to nonlinear system dynamics with nonlinear measurement functions. By exploiting the fact that the PDF is near-zero in most areas of phase space (otherwise known as sparsity), GBEES adaptively evolves the grid representing the discretized PDF to change size and shape with the true uncertainty. GBEES is a 2^{nd}-order accurate, total variation diminishing (TVD), adaptive time-marching numerical scheme that has been computationally optimized to handle the accurate uncertainty propagation of 1D-6D systems.

The goal of this instruction guide is to provide all of the necessary information needed to completely understand the inner workings of GBEES. The first thing to know about GBEES is that, although the theoretical background is quite dense, it was designed to be as user-friendly as possible. For this reason, GBEES can be implemented in either C or Python with near-equal efficiency (more on this later). Throughout the propagation period, PDF data is saved in .txt files, and thus visualization can be performed in any coding language chosen by the user; Matlab is the language

of choice used for the examples provided in this guide.

Now, some notes on notation: any text written in code blocks...

```
such as this...
```

is meant to represent code. If "$" appears at the start of the line, it is meant to represent a terminal command. Otherwise, it will most likely represent C/Python/Matlab code. Text enclosed in carets, <like_so>, must be replaced by the user, usually in the form of a repository path. Any text written in green font represents output printed in either C/Python/Matlab or to terminal. Anything in quotations is likely a directory or file path, but use your best judgement there.

The rest of the guide is structured as follows: first, installation and the necessary steps to making sure GBEES can run in your local environment are covered. Next, a quick start example is covered. This example does not provide full explanation of the method, but can be used to make sure that GBEES was installed correctly. Then, a deep dive into the architecture of the code is provided. Here, we break down the overall structure of the code, the individual functions, the various inputs and outputs, and what makes it efficient. Lastly, we walk through a few more in-depth examples and provide comment on the results. For any further questions, reference Section A for a complete dictionary of all of the functions/variables in the code and their purposes, or email blhanson@ucsd.edu. Happy GBEESing!

2 Installation

GBEES is available on Github. To install, change your working directory to the location you would like to clone GBEES into. Then run the following command in terminal

```
$ git clone https://github.com/bhanson10/gbees-hash.git
```

Now, everything you need to get started with GBEES should be installed in your working directory.

3 Quick Start (3D Lorenz Example)

To make sure your installation went smoothly and you aren't missing any requirements, we walk through the 3D Lorenz attractor example. The 3D Lorenz attractor, also known as the Butterfly

Effect, is a highly chaotic solution set to the Lorenz system, often used for testing new RBFs. The equations of motion for this system are

$$\boldsymbol{x} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \quad \frac{d\boldsymbol{x}}{dt} = \boldsymbol{f}(\boldsymbol{x}) = \begin{bmatrix} \sigma(y-x) \\ -y - xz \\ -bz + xy - br \end{bmatrix}, \tag{4}$$

with parameter values $\sigma = 4$, $b = 1$, and $r = 48$ used here. In the following example, we propagate initially Gaussian uncertainty for a short period and demonstrate how it becomes highly non-Gaussian in a short period. We then perform one discrete measurement update using measurement function

$$\boldsymbol{y} = \boldsymbol{h}(\boldsymbol{x}) = z. \tag{5}$$

Then, we propagate the uncertainty for a short time and see how it becomes non-Gaussian again.

To begin, choose which language you would like to implement GBEES in, C or Python. To implement this example in C, follow the steps in Section 3.1. To implement this example in Python, follow the steps in Section 3.2. Then, for both C and Python users, proceed to Section 3.3 for a discussion on the output results and their visualization.

3.1 C implementation

Navigate to the 3D Lorenz example using the following:

```
$ cd <path_to_gbees>/gbees/examples/Lorenz3D
```

Now, run the code with the makefile in C mode:

```
$ make MODE=c
```

The resulting PDFs are now located in "./results/c/P0" and "./results/c/P1".

3.2 Python implementation

To run GBEES in Python, we must compile the C implementation to a shared object file. The GBEES Python wrapper than dynamically links with this object and runs Python through the linked functions. To compile the C code, return to the parent directory and compile the C code using the following commands:

```
$ cd <path_to_gbees>/gbees
$ make
```

"gbees.so" is the shared object created during compile, which is then called by "gbeespy.py" whenever GBEES is run with Python. Now, we return to the 3D Lorenz example using the following:

```
$ cd <path_to_gbees>/gbees/examples/Lorenz3D
```

Then compile and run the code with the makefile in Python mode:

```
$ make MODE=python
```

The resulting PDFs are now located in "./results/python/P0" and "./results/python/P1".

3.3 Matlab visualization

Although GBEES is run in C or Python, it is visualized using Matlab. To read in the .txt files that were just saved, we need to edit "plot_Lorenz3D.m". In Line 30, change "P_DIR" to be the location of the PDF directory:

```
P_DIR = "./results/<language>";
```

Ensure that "<path_to_gbees>/gbees/examples" has been added to your Matlab search path. Now run the Matlab code to plot the 3D isosurfaces representing the $p = [0.68,\ 0.95,\ 0.997]$ isocurves. The change in color represents a measurement update has occurred.

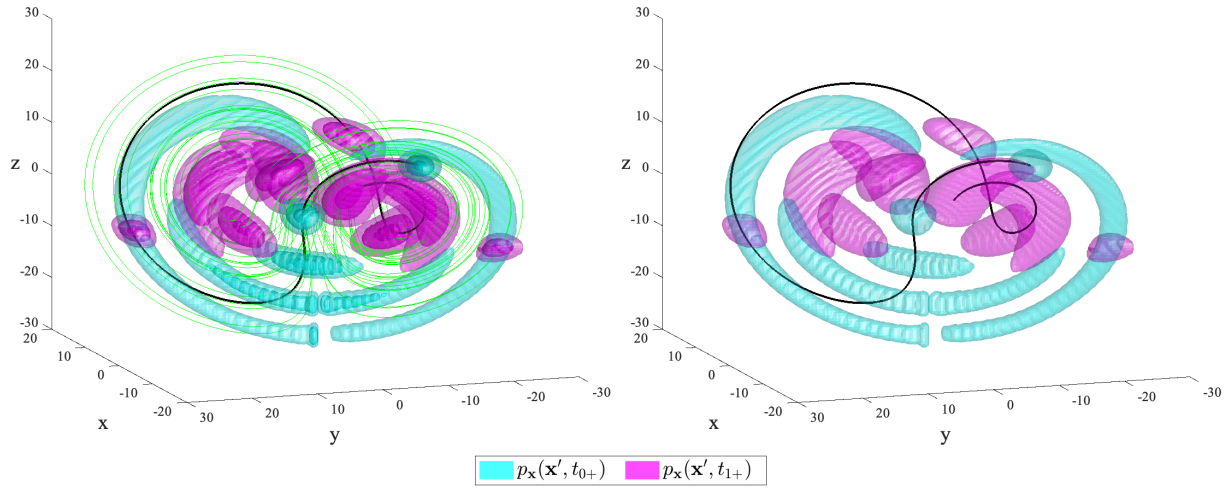

Figure 1: Initially Gaussian uncertainty becoming highly non-Gaussian when subject to the 3D Lorenz attractor dynamics. (*left*) $p = [0.68, 0.95, 0.997]$ isocurves with green background behavior and black nominal trajectory and (*right*) $p = 0.997$ isocurve with black nominal trajectory.

4 Directory Architecture

Now that we have GBEES up and running, we will walk through the directory architecture and discuss what each folder/file means. Here is GBEES printed out as a tree:

```
.
|-- LICENSE
|-- README.md
|-- examples
|   |-- CR3BP
|   |   |-- CR3BP.c
|   |   |-- CR3BP.h
|   |   |-- CR3BP.py
|   |   |-- makefile
|   |   |-- measurements
|   |   |   |-- measurement0.txt
|   |   |-- plot_CR3BP.m
|   |-- Lorenz3D
|   |   |-- Lorenz3D.c
|   |   |-- Lorenz3D.h
```

```
|   |   |-- Lorenz3D.py
|   |   |-- makefile
|   |   |-- measurements
|   |   |   |-- measurement0.txt
|   |   |   |-- measurement1.txt
|   |   |-- plot_Lorenz3D.m
|   |-- PCR3BP
|   |   |-- PCR3BP.c
|   |   |-- PCR3BP.h
|   |   |-- PCR3BP.py
|   |   |-- makefile
|   |   |-- measurements
|   |   |   |-- measurement0.txt
|   |   |   |-- measurement1.txt
|   |   |   |-- measurement2.txt
|   |   |   |-- measurement3.txt
|   |   |-- plot_PCR3BP.m
|   |-- compare_times.m
|   |-- ode87.m
|   |-- plot_nongaussian_surface.m
|-- gbees.c
|-- gbees.h
|-- gbeespy.py
|-- makefile
```

The base code is "<path_to_gbees>/gbees/gbees.c". This program contains all of the functionality necessary for running GBEES. "gbees.so" is a shared object that is created when compiling "gbees.c". Using "gbeespy.py" and "gbees.so", GBEES can run in Python almost as efficiently as it is in C (a demonstration of this capability is provided in Section 6.1). There are three examples provided in "<path_to_gbees>/gbees/examples": Lorenz3D, PCR3BP, and CR3BP. Each examples includes a C implementation ("<model>.c") and a Python implementation ("<model>.py"), as well as a script for visualizing the results in Matlab ("plot_<model>.m"). Each C implementation calls

from “gbees.c” in the parent directory, and each Python implementation calls from “gbeespy.py”, so it is important to keep this structure the same once it is cloned from Github.

4.1 Measurements

Example measurements are also included in the GBEES architecture. The structure of a measurement .txt file is very specific and must be matched exactly in order for GBEES to handle it correctly. Measurements are zero-indexed, meaning the initial uncertainty representing the *a priori* uncertainty is labeled “measurement_0.txt”. Then, any measurements beyond this increment upwards, i.e., “measurement_1.txt”, “measurement_2.txt”, etc.

Let’s take a look at the initial measurement for the 3D Lorenz example, “<path_to_gbees> /gbees/examples/Lorenz3D/measurements/measurement_0.txt”:

```
1   x (LU) y (LU) z (LU)
2   -11.50 -10.00 9.5000
3
4   Covariance(x, y, z)
5   1.000000000000000000 0.000000000000000000 0.000000000000000000
6   0.000000000000000000 1.000000000000000000 0.000000000000000000
7   0.000000000000000000 0.000000000000000000 1.000000000000000000
8
9   T (TU)
10  1
```

Note that the line numbers are added for reference, **but are not actually in the true measurement files and should not be added**. In Line 1, the state coordinates from Equation (4) are provided, with units in parentheses (because the 3D Lorenz attractor system is not physical, we use LU, or length units, as a normalized unit). In Line 2, the d-dimensional measurement mean is provided, with spaces between values. Line 3 is skipped. Line 4 is the covariance label, reiterating which variables the covariance matrix represents. In Lines 5-7, the $d \times d$ covariance matrix is provided with spaces in between. Line 8 is skipped. Line 9 is the period label, T, with units in parentheses (because the 3D Lorenz attractor system is not physical, we use TU, or time units, as a normalized

unit). The period is the amount of time till the next measurement. Line 10 is the period value. The label lines, or Lines 1, 4, and 10 can contain anything, but it is important that all other artifacts are replicated, including the skipped lines between mean and covariance and covariance and period. All values must be separated by spaces. As long as this format is matched, the mean vector and covariance matrix may be any dimension.

Although the initial measurement "measurement_0.txt" should contain a mean vector and covariance matrix with dimensionality that match the dynamics model $\boldsymbol{f}$, all measurements beyond this should contain mean vectors and measurement covariances that match the dimensionality of the measurement model $\boldsymbol{h}$. To further explain this, we now look at the second measurement from the 3D Lorenz example, "<path_to_gbees> /gbees/examples/Lorenz3D/measurements/measurement_1.txt":

```
1   z (LU)
2   -8
3
4   Covariance(z)
5   1
6
7   T (TU)
8   1
```

Here, the mean and covariance are with respect to the measurement state $\boldsymbol{y}$.

4.2 PDFs

Just like the measurements are read in as .txt files, the non-Gaussian PDFs propagated by GBEES are output as .txt files. The saved PDFs are separated by measurement. For instance, the if the most recent measurement update is "measurement_0.txt", then PDFs are stored in directory ".../P0". When "measurement_1.txt" updates the distribution, PDFs will now be stored in ".../P1", and so on. For this reason, it is necessary that the requisite number of ".../P#" folders are created prior to implementing GBEES, as was done in the makefile. The number of measurements should be equal to the number of subfolders, all of which should be in the same parent directory which is provided to the GBEES implementation as was done in Sections 3.1 and 3.2.

Within the "P"-folders, the PDFs are zero-indexed and increment up, i.e., "pdf_0.txt", "pdf_1.txt", etc. Let's take a look at the first few lines of the first PDF saved from the 3D Lorenz example, which should have been saved in "<path_to_gbees> /gbees/examples/Lorenz3D/results/P0/<language>/pdf_0.txt" if Section 3 was carried out correctly:

```
1   0.000000
2   7.9611383961e-03 -1.1500000000e+01 -1.0000000000e+01 9.5000000000e+00
3   7.0256799756e-03 -1.1500000000e+01 -1.0000000000e+01 9.0000000000e+00
4   6.2001408170e-03 -1.1500000000e+01 -1.0500000000e+01 9.0000000000e+00
5   7.0256799756e-03 -1.1500000000e+01 -1.0500000000e+01 9.5000000000e+00
6   6.2001408170e-03 -1.2000000000e+01 -1.0000000000e+01 9.0000000000e+00
7 ...
```

Line 1 is the simulation epoch of the PDF. Lines 2 through the end of the file represent the probability at specific grid cell centers. The first column is the probability, and the second through fourth columns are the grid cell state, or (x, y, z) for the 3D Lorenz attractor example. In general, ".../Pk-1/pdf_n.txt" and ".../Pk/pdf_0.txt" are the PDF at the same epoch before and after measurement update $\boldsymbol{y}_k$, where $n + 1$ is the number of PDFs recorded for each measurement segment.

5 User Input and Output

We will now be walking through the options available to the user when running GBEES. We will be reviewing these options for the 3D Lorenz example code, so for those who ran the C implementation in Section 3, proceed to Section 5.1, and for those who ran the Python implementation, proceed to Section 5.2.

5.1 C Implementation

Open "<path_to_gbees>/gbees/examples/Lorenz3D/Lorenz3D.c" in whatever text editor you prefer. The following is written in Line 4:

```
#include "../../gbees.h"
```

This line is including the parent script that includes all the functionality necessary for GBEES to run. If "gbees.h" is moved for whatever reason, ensure that this line is changed.

Now, move over to "<path_to_gbees>/gbees/examples/Lorenz3D/Lorenz3D.h". In Lines 7 and 8 we have the following:

```
#define DIM_f 3 // State dimension
#define DIM_h 1 // Measurement dimension
```

This is where we define the dimensionality of the system models. The state dimensionality is defined with "DIM_f", and the measurement dimensionality is defined with "DIM_h". Ensure that these values match your upcoming dynamics and measurement functions.

Back to "<path_to_gbees>/gbees/examples/Lorenz3D/Lorenz3D.c". Starting at Line 8, we have the system dynamics function definition:

```
// This function defines the dynamics model - required
void Lorenz3D(double* f, double* x, double t, double* dx, double* coef){
    f[0] = coef[0]*(x[1]-(x[0]+(dx[0]/2.0)));
    f[1] = -(x[1]+(dx[1]/2.0))-x[0]*x[2];
    f[2] = -coef[1]*(x[2]+(dx[2]/2.0))+x[0]*x[1]-coef[1]*coef[2];
}
```

The inputs for this function must be exactly as they are, but the output may be a vector "f" of any size, give that it matches "DIM_f". "f" is where the equations of motion are stored, "x" is the current state, "t" is the current time (which is only needed if the equations of motion are time-varying), "dx" is the grid width (which is only needed if the equations of motion are nonconservative), and if any constants are required for the function, they will be stored in the "coef" variable (more on this later). In this case, the function is the 3D Lorenz attractor equations of motion.

Next, the measurement function is defined, starting in Line 15:

```
// This function defines the measurement model -  required if MEASURE == true
void z(double* h, double* x, double t, double* dx, double* coef){
    h[0] = x[2];
}
```

The inputs for this function must be exactly as they are, but the output may be a vector "h" of any size, give that it matches "DIM_h". "h" is where the measurement is stored, "x" is the current

state, “t” is the current time (which is only needed if the measurement function is time-varying), “dx” is the grid width (which is only needed if the equations of motion are nonconservative), and if any constants are required for the function, they will be stored in the “coef” variable (more on this later). In this case, the function is the z-value.

Now, we enter the main script where GBEES is implemented. First, starting in Line 23, we read in the PDF directory, measurement directory, and first measurement file:

```
char* P_DIR = "./results/c";      // Saved PDFs path
char* M_DIR = "./measurements";     // Measurement path
char* M_FILE = "measurement0.txt"; // Measurement file
```

We then create a Measurement object using this information in Line 26:

```
Meas M = Meas_create(DIM_f, M_DIR, M_FILE);
```

Now, we generate a Grid object. Grid object initialization requires a dimensionality, probability threshold, center, and grid width. In general, the grid width is half of the initial standard deviation from the initial measurement in each direction. Starting in Line 32, we define the grid width:

```
double dx[DIM_f];  // Grid width, default is half of the std. dev. from the
    initial measurement
for(int i = 0; i < DIM_f; i ++){
    dx[i] = pow(M.cov[i][i],0.5)/2;
}
```

We then initialize the Grid object with “DIM_f”, a probability threshold $p = 5\text{e-}6$, the mean of the initial measurement, and “dx”, in Line 36:

```
Grid G = Grid_create(DIM_f, 5E-6, M.mean, dx); // Inputs: (dimension,
    probability threshold, center, grid width)
```

Now, we initialize the Trajectory object. The Trajectory object has all of the coefficients required for calculation of the dynamics function and the measurement function. For the 3D Lorenz attractor, we need σ, b, and r. These values are initialized, and the Trajectory object is defined in Lines 38 and 39:

```
double coef[] = {4.0, 1.0, 48.0}; // Lorenz3D trajectory attributes (sigma,
    beta, r)
Traj T = Traj_create(3, coef);    // Inputs: (# of coefficients, coefficients)
```

The first input to the Trajectory object is the number of coefficients in the list.

Now, we initialize the miscellaneous inputs. Descriptions for each input in Lines 41 through 50 are given in Table 1.

Table 1: Miscellaneous Parameters

Name	Data Type	Description
NUM_DIST	int	Number of distributions recorded per measurement
NUM_MEAS	int	Number of measurements for propagation period
DEL_STEP	int	Number of steps per deletion procedure (more on this in Section A)
OUTPUT_FREQ	int	Number of steps per printing PDF information to terminal
CAPACITY	int	Capacity of hash table (power of 2 for optimal hashing)
OUTPUT	bool	Boolean switch for printing PDF information to terminal
RECORD	bool	Boolean switch for recording PDF information to .txt files
MEASURE	bool	Boolean switch for performing measurement updates
BOUNDS	bool	Boolean switch for including bounding function (more on this in Section A)
COLLISIONS	bool	Boolean switch for tracking collision counts

Finally, we run GBEES with all of the user inputs in Line 52:

```
run_gbees(Lorenz3D, z, NULL, G, M, T, P_DIR, M_DIR, NUM_DIST, NUM_MEAS,
    DEL_STEP, OUTPUT_FREQ, CAPACITY, DIM_h, OUTPUT, RECORD, MEASURE, BOUNDS,
    COLLISIONS);
```

Table 2 provides the description for each of the inputs not already defined.

Table 2: GBEES Inputs

Input No.	Name	Data Type	Description
1	Lorenz3D	function	Dynamics function
2	z	function	Measurement function
3	NULL	function	Bounding function. If no bounding function, put “NULL”
4	G	Grid	Grid object
5	M	Meas	Measurement object
6	T	Traj	Trajectory object
7	P_DIR	char*	Directory location for saving PDFs
8	M_DIR	char*	Directory location for reading measurements

5.2 Python Implementation

Open "<path_to_gbees>/gbees/examples/Lorenz3D/Lorenz3D.py" in whatever text editor you prefer. The following is written in Lines 4 to 6:

```
import sys
sys.path.append('../../')
import gbeespy as gbees # type: ignore
```

This line is including the parent script that includes all the functionality necessary for GBEES to run. If "gbeespy.py" is moved for whatever reason, ensure that this line is changed.

In Lines 8 and 9 we have the following:

```
DIM_f = 3 # State dimension
DIM_h = 1 # Measurement dimension
```

This is where we define the dimensionality of the system models. The state dimensionality is defined with "DIM_f", and the measurement dimensionality is defined with "DIM_h". Ensure that these values match your upcoming dynamics and measurement functions.

Starting at Line 11, we have the system dynamics function definition:

```
# This function defines the dynamics model - required
def Lorenz3D(x, t, dx, coef):
    f1 = coef[0]*(x[1] - (x[0] + (dx[0]/2.0)))
    f2 = -(x[1] + (dx[1]/2.0)) - x[0]*x[2]
    f3 = -coef[1]*(x[2] + (dx[2]/2.0)) + x[0]*x[1] - coef[1]*coef[2]
    return [f1, f2, f3]
```

The inputs for this function must be exactly as they are, but the output may be a vector "f" of any size, give that it matches "DIM_f". "f" is where the equations of motion are stored, "x" is the current state, "t" is the current time (which is only needed if the equations of motion are time-varying), "dx" is the grid width (which is only needed if the equations of motion are non-conservative), and if any constants are required for the function, they will be stored in the "coef" variable (more on this later). In this case, the function is the 3D Lorenz attractor equations of motion.

Next, the measurement function is defined, starting in Line 18:

```
# This function defines the measurement model - required if MEASURE == True
def z(x, t, dx, coef):
    h1 = x[2]
    return [h1]
```

The inputs for this function must be exactly as they are, but the output may be a vector "h" of any size, give that it matches "DIM_h". "h" is where the measurement is stored, "x" is the current state, "t" is the current time (which is only needed if the measurement function is time-varying), "dx" is the grid width (which is only needed if the equations of motion are nonconservative), and if any constants are required for the function, they will be stored in the "coef" variable (more on this later). In this case, the function is the z-value.

Now, we enter the main script where GBEES is implemented. First, starting in Line 26, we read in the PDF directory, measurement directory, and first measurement file:

```
P_DIR = "./results/python"      # Saved PDFs path
M_DIR = "./measurements"     # Measurement path
M_FILE = "measurement0.txt"  # Measurement file
```

We then create a Measurement object using this information in Line 29:

```
M = gbees.Meas_create(DIM_f, M_DIR, M_FILE)
```

Now, we generate a Grid object. Grid object initialization requires a dimensionality, probability threshold, center, and grid width. In general, the grid width is half of the initial standard deviation from the initial measurement in each direction. Starting in Line 35, we define the grid width:

```
dx = [None] * DIM_f  # Grid width, default is half of the std. dev. from the
    initial measurement
for i in range(DIM_f):
    dx[i] = (M.cov[i][i]**(0.5))/2
```

We then initialize the Grid object with "DIM_f", a probability threshold $p = 5\text{e-}6$, the mean of the initial measurement, and "dx", in Line 38:

```
G = gbees.Grid_create(DIM_f, 5E-6, M.mean, dx) # Inputs: (dimension,
    probability threshold, center, grid width)
```

Now, we initialize the Trajectory object. The Trajectory object has all of the coefficients required for calculation of the dynamics function and the measurement function. For the 3D Lorenz attractor, we need σ, b, and r. These values are initialized, and the Trajectory object is defined in Lines 40 and 41:

```
coef = [4.0, 1.0, 48.0] # Lorenz3D trajectory attributes (sigma, beta, r)
T = gbees.Traj_create(len(coef), coef) # Inputs: (# of coefficients,
    coefficients)
```

The first input to the Trajectory object is the number of coefficients in the list.

Now, we initialize the miscellaneous inputs. Descriptions for each input in Lines 43 through 50 are given in Table 3.

Table 3: Miscellaneous GBEES Inputs

Name	Data Type	Description
NUM_DIST	int	Number of distributions recorded per measurement
NUM_MEAS	int	Number of measurements for propagation period
DEL_STEP	int	Number of steps per deletion procedure (more on this in Section A)
OUTPUT_FREQ	int	Number of steps per printing PDF information to terminal
CAPACITY	int	Capacity of hash table (power of 2 for optimal hashing)
OUTPUT	bool	Boolean switch for printing PDF information to terminal
RECORD	bool	Boolean switch for recording PDF information to .txt files
MEASURE	bool	Boolean switch for performing measurement updates
BOUNDS	bool	Boolean switch for including bounding function (more on this in Section A)
COLLISIONS	bool	Boolean switch for tracking collision counts

Finally, we run GBEES with all of the user inputs in Line 54:

```
gbees.run_gbees(Lorenz3D, z, None, G, M, T, P_DIR, M_DIR, NUM_DIST, NUM_MEAS,
    DEL_STEP, OUTPUT_FREQ, CAPACITY, DIM_h, OUTPUT, RECORD, MEASURE, BOUNDS,
    COLLISIONS)
```

Table 4 provides the description for each of the inputs.

Table 4: GBEES Inputs

Input No.	Name	Data Type	Description
1	Lorenz3D	function	Dynamics function
2	z	function	Measurement function
3	None	function	Bounding function. If no bounding function, put "NULL"
4	G	Grid	Grid object
5	M	Meas	Measurement object
6	T	Traj	Trajectory object
7	P_DIR	string	Directory location for saving PDFs
8	M_DIR	string	Directory location for reading measurements

5.3 Terminal output

Now, we discuss the output that is printed to terminal. Here is an example of a line that is printed to terminal, from the 3D Lorenz attractor example:

Timestep: 1-236, Program time: 2.675827 s, Sim. time: 1.250000 TU, Active/Total Cells: 3542/7471

We now breakdown the components of this output. "Timestep: M-N" gives the last measurement number M and the prediction step number N. "Program time: P s" gives the computation runtime up to this point P in seconds. "Sim. time: S TU" gives the simulation propagation time S in time units (TU). "Active/Total Cells: A/T" gives the number of grid cells with probability above threshold A and the total number of grid cells T.

6 Examples

Now, we dive into the more advanced examples provided with GBEES. If you haven't already, make sure you were able to get Section 3 working.

6.1 3D Lorenz Runtime Comparison

Earlier, we stated that the Python wrapper is able to implement GBEES nearly as efficiently as the C code. To put this to the test, we are going to use the output of GBEES to compare computation runtime.

6.1.1 C and Python Implementation

Navigate to "<path_to_gbees>/gbees/examples/Lorenz3D" and run the following command:

```
$ make MODE=c
```

Copy the terminal output into "<path_to_gbees>/gbees/examples/Lorenz3D/results/c/runtime.txt" and remove all of the extra blank lines and the line PERFORMING BAYESIAN UPDATE AT: 1.000000 TU.... Then run the following command:

```
$ make MODE=python
```

Copy the terminal output into "<path_to_gbees>/gbees/examples/Lorenz3D/results/python/runtime.txt" and remove all of the extra blank lines and the line PERFORMING BAYESIAN UPDATE AT: 1.000000 TU....

6.1.2 Matlab Visualization

Now open "<path_to_gbees>/gbees/examples/compare_times.m" in Matlab and change Line 4 to:

```
SYS = "Lorenz3D";
```

Now run the code. The output should look similar to Figure 2. This script compares the distribution cell count and computation runtime of the C and Python implementations (the cell count is plotted to ensure that the two implementations are returning identical results).

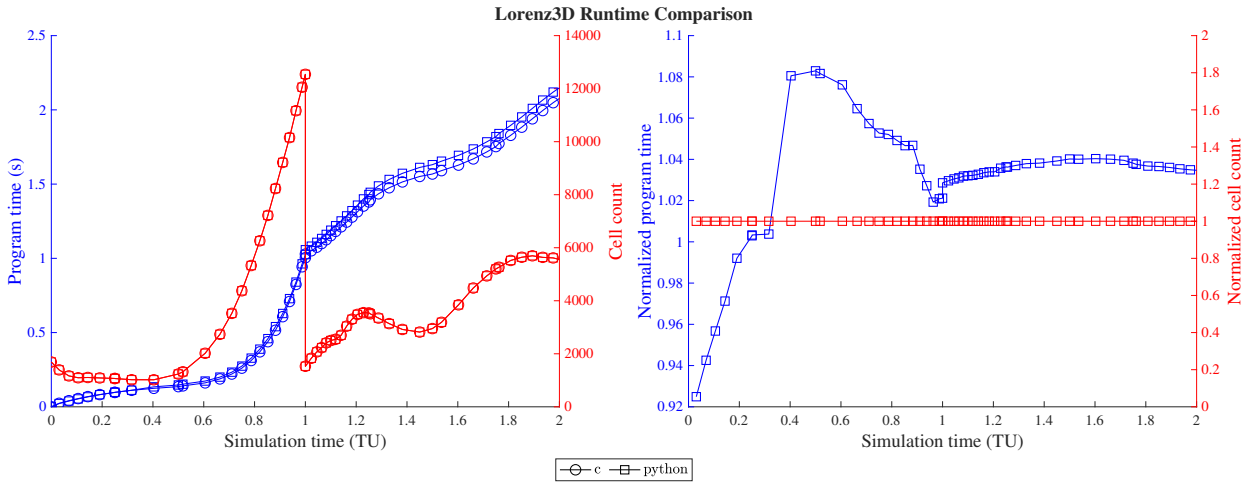

Figure 2: At output epochs, the (*left*) true cell count/computation runtime of the C and Python implementations are compared, and the (*right*) normalized cell count/computation runtimeof the C and Python implementations are compared.

The Python implementation takes $1.0347\times$ the C implementation while returning the identical PDF. This comparison can be performed for the upcoming examples by creating the necessary directories and changing the system in Line 4.

6.2 Saturn-Enceladus 4D Distant Prograde Orbit

For this example, we propagate the state uncertainty of an unstable Saturn-Enceladus Distant Prograde Orbit (DPO) with nonlinear measurement updates with GBEES. DPOs are planar, M_2-centered, stable/unstable periodic orbits whose invariant manifolds provide heteroclinic connections between L_1 and L_2 Lyapunov orbits, meaning a substantial volume of the three-body system may be explored by forming chains of unstable periodic orbits connected by low-energy transfers. Depending on the energy of the third-body, the DPO may be unstable, but operation there may be advantageous, as the L_1-DPO$-L_2$ chain efficiently traverses different volumes of phases space with the option to idle for multiple orbits between L_1 and L_2. Due to the instability, uncertainty propagation for this orbit family is best handled by non-Gaussian filters like GBEES.

The state and orbital dynamics of a spacecraft in the Planar Circular Restricted Three Body

Problem (PCR3BP) in the synodic, non-dimensional frame are

$$\boldsymbol{x} = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix} \quad \text{and} \quad \dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}) = \begin{bmatrix} \dot{x} \\ \dot{y} \\ 2\dot{y} + x - \frac{(1-\mu)(x+\mu)}{r_1} - \frac{\mu(x-1+\mu)}{r_2} \\ -2\dot{x} + y - \frac{(1-\mu)y}{r_1} - \frac{\mu y}{r_2} \end{bmatrix}, \tag{6}$$

where $\mu = \mu_2/(\mu_1+\mu_2)$ is the mass ratio, μ_i represents the gravitational parameter of body M_i, and r_i is the distance to body M_i. This model is used in the prediction step in the upcoming analysis.

The measurement model is

$$\boldsymbol{y} = \begin{bmatrix} \rho \\ \theta \\ \dot{\rho} \end{bmatrix} = \boldsymbol{h}(\boldsymbol{x}) = \begin{bmatrix} \sqrt{(x-1+\mu)^2 + y^2} \\ \tan^{-1}\left(\frac{y}{x-1+\mu}\right) \\ \frac{(x-1+\mu)\dot{x} + y\dot{y}}{\rho} \end{bmatrix}.$$

where ρ is the range, θ is the azimuth angle, and $\dot{\rho}$ is the range-rate, all relative to M_2. Because of the Bayesian nature of this investigation, measurements are taken to be the true state at epoch, transformed by $\boldsymbol{h}$ with zero-mean Gaussian noise. This model is used in the correction step in the upcoming analysis.

One integral of motion exists for the CR3BP, the Jacobi constant, and is defined as

$$C = x^2 + y^2 + \frac{2(1-\mu)}{r_1} + \frac{2\mu}{r_2} + \mu(1-\mu) - \dot{x}^2 - \dot{y}^2 - \dot{z}^2; \tag{7}$$

because GBEES is a 2nd-order accurate numerical scheme, Equation (7) is not necessarily conserved. To compensate for this, the requirement can be hardcoded into the grid generation. The initial discretized PDF has a minimum and maximum C. As the grid grows in phase space, forbidden cells are those that fall outside of this bound and are not created. Admissible cells fall within the initial Jacobi bounds and are inserted into the grid as needed. Using Jacobi bounding, the conservation of C is artificially ensured in GBEES.

An initial state that results in a Saturn-Enceladus DPO is

$$\boldsymbol{x}_0 = \begin{bmatrix} 1.001471 & \text{(LU)} \\ -1.751810\text{e-}5 & \text{(LU)} \\ 7.198783\text{e-}5 & \text{(LU/TU)} \\ 1.363392\text{e-}2 & \text{(LU/TU)} \end{bmatrix} = \begin{bmatrix} 238879.876159 & \text{(km)} \\ -4.178575 & \text{(km)} \\ 9.079038\text{e-}4 & \text{(km/s)} \\ 1.719497\text{e-}2 & \text{(km/s)} \end{bmatrix}.$$

Other properties of the trajectory are provided in Table 6.

Table 5: Saturn-Enceladus DPO properties

μ	LU (km)	TU (s)	C	SI	T (hr)
1.901110e-7	238529	18913	3.0 + 7.809821e-5	3.018700e+2	19.58109

6.2.1 C implementation

Navigate to the PCR3BP example using the following:

```
$ cd <path_to_gbees>/gbees/examples/PCR3BP
```

Now, compile and run the code with the makefile in C mode:

```
$ make MODE=c
```

The resulting PDFs are now located in "./results/c/P0" through "./results/c/P3".

6.2.2 Python implementation

To run GBEES in Python, we must compile the C implementation to a shared object file. The GBEES Python wrapper than dynamically links with this object and runs Python through the linked functions. To compile the C code, return to the parent directory and compile the C code using the following commands:

```
$ cd <path_to_gbees>/gbees
$ make
```

"gbees.so" is the shared object created during compile, which is then called by "gbeespy.py" whenever GBEES is run with Python. Now, navigate to the PCR3BP example using the following:

```
$ cd <path_to_gbees>/gbees/examples/PCR3BP
```

Then compile and run the code with the makefile in Python mode:

```
$ make MODE=python
```

The resulting PDFs are now located in "./results/python/P0" through "./results/python/P3".

6.2.3 Matlab Visualization

Although GBEES is run in C or Python, it is visualized using Matlab. To read in the .txt files that were just saved, we need to edit "plot_PCR3BP.m". In Line 55, change "P_DIR" to be the location of the PDF directory:

```
P_DIR = "./results/<language>";
```

Ensure that "<path_to_gbees>/gbees/examples" has been added to your Matlab search path. Now run the Matlab code to plot the 4D isosurfaces representing the $p = [0.68,\ 0.95,\ 0.997]$ isocurves. The GBEES-propagated distribution is a discretized, 4D PDF, but integrating over the velocity- and position-spaces returns the 2D position and velocity PDFs, respectively. This is done via a numerical implementation of the following formulae:

$$p_{(x,y)}(x',y') = \int_{\Omega_{(\dot{x},\dot{y})}} p_{\boldsymbol{x}}(\boldsymbol{x}')d\dot{x}'d\dot{y}' \quad \text{and} \quad p_{(\dot{x},\dot{y})}(\dot{x}',\dot{y}') = \int_{\Omega_{(x,y)}} p_{\boldsymbol{x}}(\boldsymbol{x}')dx'dy'.$$

The color changes throughout indicate when a measurement update has occurred, where the *a posteriori* $p_{\boldsymbol{x}}(\boldsymbol{x}', t_{k+})$ is the resultant assimilation of the *a priori* $p_{\boldsymbol{x}}(\boldsymbol{x}', t_{k-})$ and the likelihood distribution $p_{\boldsymbol{y}}(\boldsymbol{y}_k|\boldsymbol{x}')$. The distributions shown are not separated by equal time intervals; instead, they are spaced to optimize visualization.

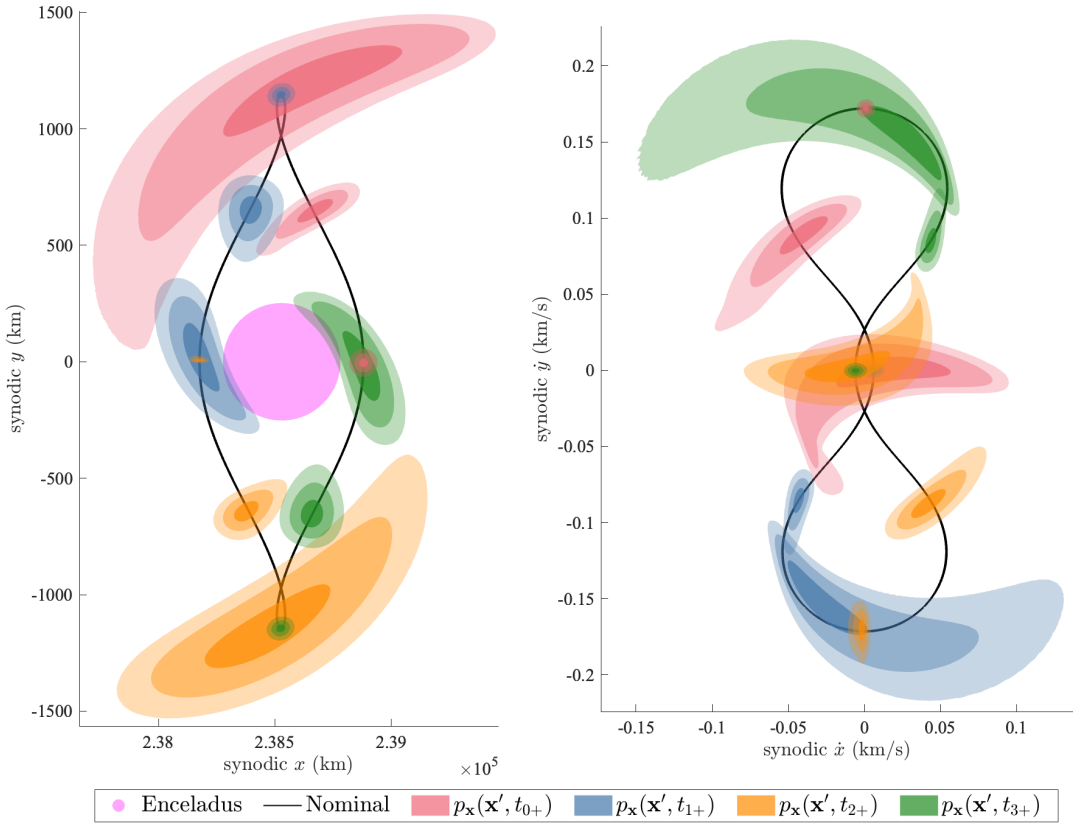

Figure 3: Saturn-Enceladus DPO true synodic state uncertainty.

6.3 Jupiter-Europa 6D Low Prograde Orbit

For this example, we propagate the state uncertainty of a Jupiter-Europa Low Prograde Orbit (LPO). The state and orbital dynamics of a spacecraft in the Circular Restricted Three Body Problem (CR3BP) in the synodic, non-dimensional frame are

$$\boldsymbol{x} = \begin{bmatrix} x \\ y \\ z \\ \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix} \quad \text{and} \quad \dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}) = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ 2\dot{y} + x - \frac{(1-\mu)(x+\mu)}{r_1} - \frac{\mu(x-1+\mu)}{r_2} \\ -2\dot{x} + y - \frac{(1-\mu)y}{r_1} - \frac{\mu y}{r_2} \\ -\frac{(1-\mu)z}{r_1} - \frac{\mu z}{r_2} \end{bmatrix}, \tag{8}$$

where $\mu = \mu_2/(\mu_1+\mu_2)$ is the mass ratio, μ_i represents the gravitational parameter of body M_i, and r_i is the distance to body M_i. This model is used in the prediction step in the upcoming analysis. We do not perform a measurement update for this example, and we use the Jacobi bounding function from Section 6.2.

An initial state that results in a Jupiter-Europa LPO is

$$\boldsymbol{x}_0 = \begin{bmatrix} 1.0169963 & \text{(LU)} \\ -1.069795\text{e-}20 & \text{(LU)} \\ -5.1360140\text{e-}34 & \text{(LU)} \\ -1.393517\text{e-}14 & \text{(LU/TU)} \\ 1.257591\text{e-}2 & \text{(LU/TU)} \\ -3.157220\text{e-}33 & \text{(LU/TU)} \end{bmatrix} = \begin{bmatrix} 6.798813\text{e+}5 & \text{(km)} \\ -7.151785\text{e-}15 & \text{(km)} \\ -3.433523\text{e-}28 & \text{(km)} \\ -1.918358\text{e-}13 & \text{(km/s)} \\ 1.731238\text{e-}1 & \text{(km/s)} \\ -4.346324\text{e-}32 & \text{(km/s)} \end{bmatrix}.$$

Other properties of the trajectory are provided in Table 6.

Table 6: Jupiter-Europa LPO properties

μ	LU (km)	TU (s)	C	SI	T (hr)
2.528018e-5	668519	48562	3.003571	$1.0 + 1.74001\text{e-}9$	29.661406

6.3.1 C implementation

Navigate to the 6D Jupiter-Europa LPO example using the following:

```
$ cd <path_to_gbees>/gbees/examples/CR3BP
```

Now, we compile and run the code with the makefile in C mode:

```
$ make MODE=c
```

The resulting PDFs are now located in "./results/c/P0".

6.3.2 Python implementation

To run GBEES in Python, we must compile the C implementation to a shared object file. The GBEES Python wrapper than dynamically links with this structure and runs Python through the

linked functions. To compile the C code, return to the parent directory and compile the C code using the following commands:

```
$ cd <path_to_gbees>/gbees
$ make
```

"gbees.so" is the shared structure created during compile, which is then called by "gbeespy.py" whenever GBEES is run with Python. Now, navigate to the 6D Jupiter-Europa LPO example using the following:

```
$ cd <path_to_gbees>/gbees/examples/CR3BP
```

Then, compile and run the code with the makefile in Python mode:

```
$ make MODE=python
```

The resulting PDFs are now located in "./results/python/P0".

6.3.3 Matlab Visualization

Although GBEES is run in C or Python, it is visualized using Matlab. To read in the .txt files that were just saved, we need to edit "plot_CR3BP.m". In Line 77, change "P_DIR" to be the location of the PDF directory:

```
P_DIR = "./results/<language>";
```

Ensure that "<path_to_gbees>/gbees/examples" has been added to your Matlab search path. Now run the Matlab code to plot the 6D isosurfaces representing the $p = [0.68,\ 0.95,\ 0.997]$ isocurves. The GBEES-propagated distribution is a discretized, 6D PDF, but integrating over the velocity- and position-spaces returns the 3D position and velocity PDFs, respectively. This is done via a numerical implementation of the following formulae:

$$p_{(x,y,z)}(x',y',z') = \int_{\Omega_{(\dot{x},\dot{y},\dot{z})}} p_{\boldsymbol{x}}(\boldsymbol{x}')d\dot{x}'d\dot{y}'d\dot{z}' \quad \text{and} \quad p_{(\dot{x},\dot{y},\dot{z})}(\dot{x}',\dot{y}',\dot{z}') = \int_{\Omega_{(x,y,z)}} p_{\boldsymbol{x}}(\boldsymbol{x}')dx'dy'dz'.$$

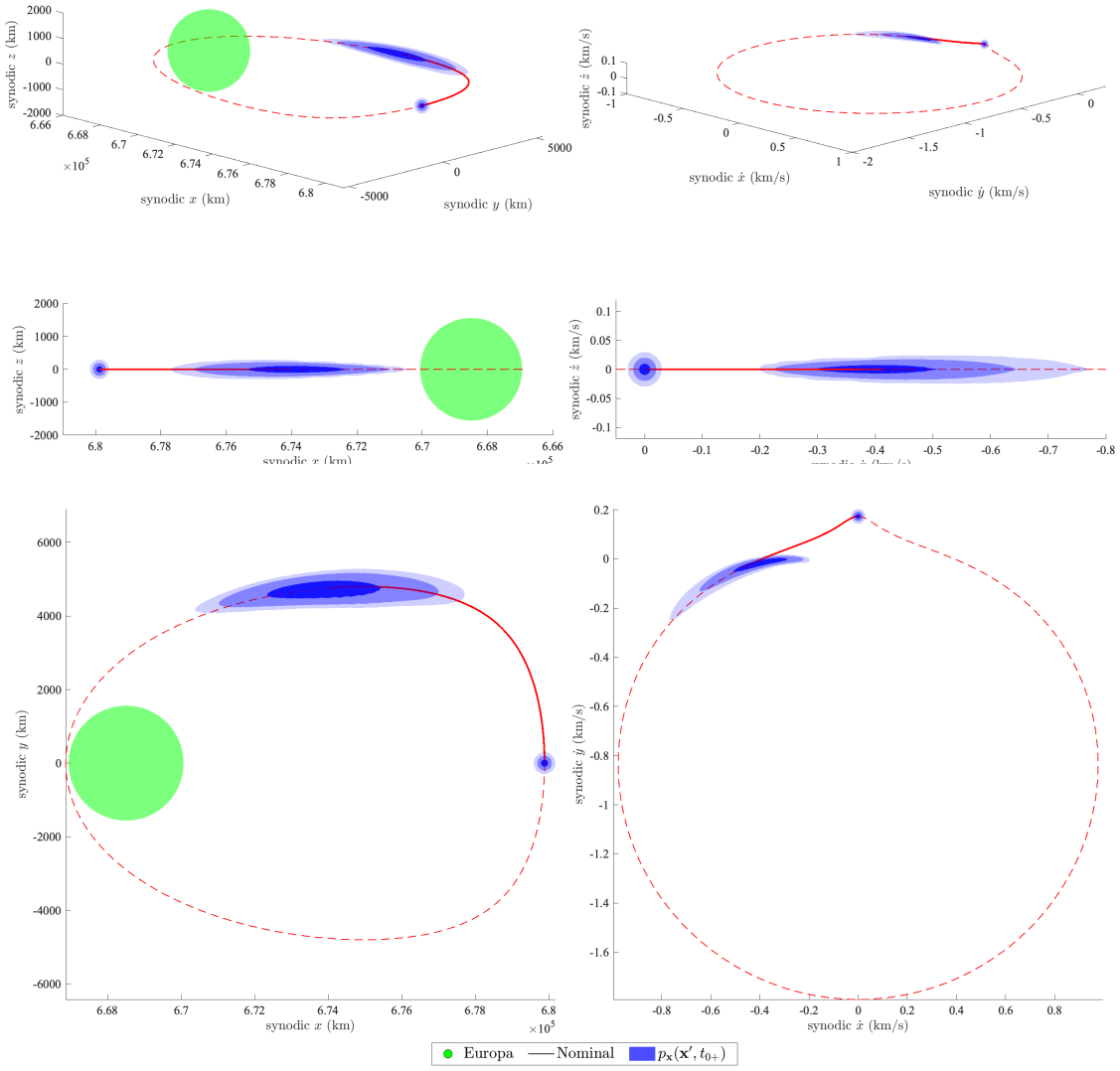

Figure 4: Jupiter-Europa LPO true synodic state uncertainty.

A Structure/Function Dictionary

Now, we list each of the structures and functions of importance in the GBEES code, as well as their purpose, in order as they appear in the code.

Table A1: Structure Definitions in "gbees.h"

Name	Data Type	Description
Meas	struct	• **Parent:** none • Measurement structure where measurement information is stored
Meas::dim	int	• **Parent:** Meas • Dimensionality of Measurement mean and covariance
Meas::mean	ptr to double	• **Parent:** Meas • Mean of Measurement structure
Meas::cov	ptr to ptr to double	• **Parent:** Meas • Covariance of Measurement structure
Meas::T	double	• **Parent:** Meas • Period of continuous-time propagation before next measurement update

Name	Data Type	Description
Grid	struct	• **Parent:** none • Grid structure where grid information is stored
Grid::dim	int	• **Parent:** Grid • Dimensionality of grid
Grid::thresh	double	• **Parent:** Grid • Probability threshold of grid
Grid::t	double	• **Parent:** Grid • Current time of simulation
Grid::dt	double	• **Parent:** Grid • Size of step of time march
Grid::center	ptr to double	• **Parent:** Grid • Center coordinates of grid
Grid::dx	ptr to double	• **Parent:** Grid • Grid width in each dimension

Name	Data Type	Description
Grid::hi_bound	double	• **Parent:** Grid • Max output of bounding function from initial distribution, set to DBL_MAX if bounding isn't implemented
Grid::lo_bound	double	• **Parent:** Grid • Min output of bounding function from initial distribution, set to -DBL_MAX if bounding isn't implemented
Traj	struct	• **Parent:** none • Trajectory structure where trajectory information is stored
Traj::coef	ptr to double	• **Parent:** Traj • List of coefficients used in dynamics and measurement models

Name	Data Type	Description
HashTableEntry	struct	• **Parent:** none • To perform continuous-time marching efficiently, GBEES stores the discretized PDF as a Hash Table, where each grid cell is a HashTableEntry that contains information about the grid cell
HashTableEntry::state	ptr to int	• **Parent:** HashTableEntry • List containing the index coordinates, converted from true coordinate values to integer values via indexing
HashTableEntry:: prob	double	• **Parent:** HashTableEntry • Probability at grid cell
HashTableEntry::v	ptr to double	• **Parent:** HashTableEntry • List containing advection values in each direction at the forward interface for the grid cell

Name	Data Type	Description
HashTableEntry::ctu	ptr to double	• **Parent:** HashTableEntry • List containing corner transport upwind values in each direction at the forward interface for the grid cell
HashTableEntry:: i_nodes	ptr to ptr to HashTableEntry	• **Parent:** HashTableEntry • List containing the pointers to pointers to backward neighboring entries in all dimensions, used for implementing Godunov scheme
HashTableEntry:: k_nodes	ptr to ptr to HashTableEntry	• **Parent:** HashTableEntry • List containing the pointers to pointers to forward neighboring entries in all dimensions, used for implementing Godunov scheme
HashTableEntry::dcu	double	• **Parent:** HashTableEntry • Donor cell upwind value for grid cell

Name	Data Type	Description
HashTableEntry:: cfl_dt	double	• **Parent:** HashTableEntry • Maximum time step that follows the Courant–Friedrichs–Lewy condition for all HashTableEntries in the BST
HashTableEntry:: new_f	int	• **Parent:** HashTableEntry • Flag indicating if the grid cell has just been created in the previous grow step
HashTableEntry:: ik_f	int	• **Parent:** HashTableEntry • Flag indicating if the grid cell requires updating of the backward and forward neighboring HashTableEntries
HashTableEntry:: bound_value	double	• **Parent:** HashTableEntry • Bound value returned by the input bounding function give the true coordinates, set to 0 if no bounding function is input
HashTableEntry:: next	ptr to HashTableEntry	• **Parent:** HashTableEntry • Pointer to next node in HashTableEntry defined as a linked list

Name	Data Type	Description
HashTable	struct	• **Parent:** none • HashTableEntries are stored in the HashTables structure, with extra attributes providing general information of the hash table
HashTable:: entries	ptr to ptr to HashTableEntry	• **Parent:** HashTable • List of linked lists representing HashTableEntries in HashTable
HashTable:: capacity	size_t	• **Parent:** HashTable • Capacity of HashTable, or number of entries in HashTable
HashTable:: a_count	size_t	• **Parent:** HashTable • Count of number of cells above threshold in PDF
HashTable:: tot_count	size_t	• **Parent:** HashTable • Count of number of cells in PDF

Table A2: Function Definitions in "gbees.c"

Name	Data Type	Description
exit_nomem	function	• **Parent:** none • **Return:** void • This function is used throughout the code to output where a memory allocation failure occured
Meas_create	function	• **Parent:** none • **Return:** Meas • This function uses the measurement directory and measurement file to create a Measurement structure
Meas_free	function	• **Parent:** none • **Return:** void • This function frees the memory associated with a Measurement structure
Grid_create	function	• **Parent:** none • **Return:** Grid • This function creates a Grid structure from the user inputs

Name	Data Type	Description
Grid_free	function	• **Parent:** none • **Return:** void • This function frees the memory associated with a Grid structure
Traj_create	function	• **Parent:** none • **Return:** Traj • This function creates a Traj structure from the user inputs
Traj_free	function	• **Parent:** none • **Return:** void • This function frees the memory associated with a Traj structure
HashTableEntry_create	function	• **Parent:** None • **Return:** ptr to HashTableEntry • Given the user inputs, this function creates a HashTableEntry structure

Name	Data Type	Description
HashTable_create	function	• **Parent:** None • **Return:** ptr to HashTable • Given the user inputs, this function creates a HashTable structure
HashTableEntry_free	function	• **Parent:** None • **Return:** void • This function frees the memory associated with a HashTable made up of HashTableEntries
hash_key	function	• **Parent:** None • **Return:** uint64_t • This function calculates the hash key given a d-dimensional state following the FNV-1a hash function
same_state	function	• **Parent:** None • **Return:**bool • This function determines whether two integer arrays are equal

Name	Data Type	Description
mc	function	• **Parent:** None • **Return:** double • This function calculates the MC flux limiter value given θ
inv_mat	function	• **Parent:** gauss_probability • **Return:** void • This function inverts a matrix
mul_mat_vec	function	• **Parent:** gauss_probability • **Return:** void • This function multiplies a matrix by a vector
dot_product	function	• **Parent:** gauss_probability • **Return:** double • This function takes the dot product of two vectors
gauss_probability	function	• **Parent:** none • **Return:** double • Given a true coordinate and a Gaussian measurement, calculate the probability: $$p = \exp\Big(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\Big)$$

Name	Data Type	Description
HashTableEntry_insert	function	• **Parent:** HashTable • **Return:** void • This function creates a HashTableEntry given the inputs and inserts the entry into the front of the linked list at the index given by the hashing function
HashTable_insert	function	• **Parent:** none • **Return:** void • This function calls HashTableEntry_insert depending on whether or not the provided state falls within the bounding function bounds.
HashTable_search	function	• **Parent:** initialize_ik_nodes • **Return:** ptr to HashTableEntry • This function searches for a HashTableEntry with a specific state, if the state does not exist, the returned ptr is NULL

Name	Data Type	Description
HashTable_delete	function	• **Parent:** none • **Return:** void • This function takes a state and deletes the linked list node within the HashTableEntry associated with the state
get_size	function	• **Parent:** none • **Return:** int • This function gets the size of the HashTable
initialize_adv	function	• **Parent:** none • **Return:** void • This function loops through the HashTableEntries and calculates the advection at each grid cell, based on the dynamics system
initialize_ik_nodes	function	• **Parent:** none • **Return:** void • This function loops through the HashTableEntries and finds the backward and forward neighbors for each grid cell

Name	Data Type	Description
recursive_loop	function	• **Parent:** initialize_grid • **Return:** void • This recursive function performs d nested loops, initializing the HashTableEntries of the initial distribution
initialize_grid	function	• **Parent:** initialize_grid • **Return:** void • This function initializes the discretized PDF stored in the HashTable based off the initial measurement and Grid structure
set_bounds	function	• **Parent:** none • **Return:** void • This function finds the min and max output of the bounding function of the initial discretized PDF
get_sum	function	• **Parent:** normalize_tree • **Return:** void • This function calculates the probability sum of the PDF

Name	Data Type	Description
divide_sum	function	• **Parent:** normalize_tree • **Return:** void • This function divides each grid cell probability in the PDF by the probability sum, while also tracking the active number of grid cells, the total number of grid cells, and the max key value
normalize_tree	function	• **Parent:** normalize_tree • **Return:** void • This function normalizes the PDF, gets the number of active cells, the number of total cells
concat_p	function	• **Parent:** none • **Return:** ptr to char • This function concatenates the PDF directory and file to save a PDF
concat_c	function	• **Parent:** none • **Return:** ptr to char • This function concatenates the PDF directory and file to save a collision count

Name	Data Type	Description
write_file	function	• **Parent:** record_data • **Return:** void • This recursive function writes the probability and grid cell true coordinates of each TreeNode in the BST to a text file
record_data	function	• **Parent:** none • **Return:** void • This function takes a file name and saves the contents of the BST to the text file
create_neighbors	function	• **Parent:** grow_tree • **Return:** void • This recursive function loops through the BST, finds TreeNodes with probabilities above threshold, then inserts all neighbors that do not already exist in the BST.

Name	Data Type	Description
grow_tree	function	• **Parent:** grow_tree • **Return:** void • This function inserts necessary grid cells into the BST, balances the BST, initializes the advection for each grid cell added, and initializes the backward and forward neighbors for each grid cell added
check_cfl_condition	function	• **Parent:** grow_tree • **Return:** void • This function gets the minimum time step for the entire BST, ensuring that the time-marching scheme follows the CFL condition
update_dcu	function	• **Parent:** godunov_method • **Return:** void • This recursive function calculates the donor cell upwind value for each grid cell in the BST (for more information, see J1 in Section B.1)

Name	Data Type	Description
update_ctu	function	• **Parent:** godunov_method • **Return:** void • This recursive function calculates the corner transport upwind values for each grid cell in the BST (for more information, see J1 in Section B.1)
godunov_method	function	• **Parent:** none • **Return:** void • This function performs the Godunov scheme on the discretized PDF, which is 2nd-order accurate and total variation diminishing
update_prob	function	• **Parent:** none • **Return:** void • This recursive function updates the probability of each grid cell in the BST given the donor cell upwind value and corner transport upwind values

Name	Data Type	Description
mark_cells	function	• **Parent:** prune_tree • **Return:** void • This recursive function marks cells that are below probability threshold and do not neighbor down-wind cells that are above probability threshold (for more information, see C2 in Section B.2)
compare_indices	function	• **Parent:** sort_by_double • **Return:** int • This function checks which value at two different indices in a list is higher
sort_by_double	function	• **Parent:** prunte_tree • **Return:** void • This function sorts an index list by a double list
delete_cells	function	• **Parent:** prunte_tree • **Return:** void • This function loops through the sorted key index list of the cells that were marked and deletes them until the probability of the remaining grid cells are above the threshold

Name	Data Type	Description
prune_tree	function	• **Parent:** none • **Return:** void • This function deletes the necessary TreeNodes in the BST, then reinitializes the backward and forward neighbors
meas_up_recursive	function	• **Parent:** none • **Return:** void • This recursive function updates performs a discrete measurement update on the discretized PDF by applying Equation (3) at each grid cell
record_collisions	function	• **Parent:** none • **Return:** void • This function records the number of collisions at each HashTableEntry
run_gbees	function	• **Parent:** none • **Return:** void • This function runs GBEES

B Publications

B.1 Refereed Journal Publications

J1 T. R. Bewley and A. S. Sharma, "Efficient grid-based Bayesian estimation of nonlinear low-dimensional systems with sparse non-Gaussian PDFs," *Automatica*, Vol. 48, No. 7, 2012, pp. 1286-1290.

B.2 Conference Publications

C2 B. L. Hanson, A. J. Rosengren, and T. R. Bewley, "State Estimation of Chaotic Trajectories: A Higher-Dimensional, Grid-Based, Bayesian Approach to Uncertainty Propagation," *AIAA SCITECH 2024 Forum*, AIAA, 2024.

C1 A. Sharma, T. R. Bewley, "Grid-based Bayesian Estimation Exploiting Sparsity for systems with nongaussian uncertainty," *APS Division of Fluid Dynamics Meeting*, Vol. 62, pp. ED–005, 2009.