PROBABILISTIC ALGORITHMS FOR TRANSITION ALTITUDE OPTIMIZATION IN BALLISTIC AIRDROP

A Thesis
Presented to
The Academic Faculty

by

Christopher J. Jumonville

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Mechanical Engineering

Georgia Institute of Technology
August, 2016

PROBABILISTIC ALGORITHMS FOR TRANSITION ALTITUDE OPTIMIZATION IN BALLISTIC AIRDROP

To those who could not finish the journey with me: Marie, Eulalie, Albert, Buck, Nester, Roy, and Gabby

ACKNOWLEDGEMENTS

I would first like to thank my advisor, Dr. Jonathan Rogers. His mentoring and guidance through this process was critical to the success of this thesis. He helped shape my graduate experience, and I will be forever grateful for the opportunities he has given me.

Adam Gerlach deserve special mention for his contributions to the software implementation of the algorithm. I want to thank him for his patience and help.

I would also like to thank my thesis committee members, Dr. Jun Ueda and Dr. Aldo Ferri, for graciously accepting my request. Their classes were instrumental in shaping my graduate education, and I am happy they were able to serve on my committee.

The iREAL lab members deserve special mention: Brady, Caroline, Jonathan, Kyle, and Laura. I would like to thank them for providing all the help they have given me during this last year and showing me that it is ok to feel lost at times.

I would particularly like to thank my friends: Huong, Brian, Ben, Ahmed A., Ahmed N., Benni, Caroline, Eui, Marius, Nadine, Thomas, and Xiashou. They made this stressful and trying period of my life so much easier to get through with their welcomed distractions, adventures, and support.

I would finally, and especially, like to thank my mother and father for always pushing me to be the best person I can be. I feel extremely fortunate to have parents that are so supportive of my goals and aspirations. Without their encouragement throughout these last 6 years, I never would have been able make it as far as I have.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

CARP	Computed Air Release Point
HALO	High-Altitude, Low-Opening
$\hat{I}, \hat{J}, \hat{K}$	Unit vectors in North-East-Down frame
x, y, z	Location of parachute-package system in North-East-Down frame
C_d	Parachute drag coefficient
S	Parachute reference area
ρ	Air density
g	Gravitational acceleration
m	Parachute-package mass
m_a	Apparent mass
k_a	Apparent mass coefficient
R	Parachute radius
$\vec{x}$	Parachute-package state vector
w_x, w_y, w_z	Magnitude of wind in the $\hat{I}, \hat{J}, \hat{K}$ directions
$\overline{w}_{mag}$	Mean wind magnitude
$\overline{w}_{dir}$	Mean wind direction
n_p	Number of uncertain parameters
$\sigma_{mag}, \sigma_{dir}, \sigma_c$	Standard deviations in wind mag., wind dir., and drag coefficient
$\varphi_{PI}(x, y)$	Desired impact distribution
$\varphi_{xy}(x, y, z)$	x-y marginal probability density
$\varphi_d(z)$	Marginal PDF along predicted drogue trajectory
$\bar{\vec{s}}_i$	Discretized value of augmented state vector

N	Number of points used in discretization of joint probability density
p, q	Number of bins in x and y used during marginalization
$\vec{s}_i$	Guess vector in BVP method
z_t	Transition altitude
$z_{t,min}$, $z_{t,max}$	Minimum and maximum allowed transition altitudes
z_t^*	Optimized transition altitude
z_{inf}	Inflation distance
$\dot{z}_{ss}$	Quasi-steady-state descent rate under main parachute
CSC	Custom Scenario Creator
D	Distance matrix
M	Minimum distance matrix
R	Road cost map
G	Geodesic cost map
J	Final cost map

SUMMARY

The development of a transition altitude optimization algorithm for ballistic airdrops is detailed. Ballistic airdrops are unguided, high-altitude, low-opening cargo drops for military or humanitarian purposes. Compared to their guided airdrop counterparts, unguided airdrops are cheaper but have less accurate impact locations since the flight path of unguided airdrops is not controlled. Because of their ability to be deployed in large quantities, efforts have been taken to improve the impact dispersion of unguided airdrops. The algorithm described here aims to increase accuracy and shape the impact dispersion while accounting for relevant sources of uncertainty by optimizing the parachute-package system's transition altitude, e.g. the altitude of main parachute deployment.

A simulation framework that consists of the airdrop dynamic model, atmospheric air density and wind model, and a parachute inflation model is generated. This framework serves as a test bed for algorithm development and testing. Additionally, the creation of complex impact distributions based on real-world map data is detailed. Nonlinear uncertainty propagation is employed to back-propagate the impact distribution through space and time from ground to airdrop altitude. This time-history of the impact distribution is leveraged for the purposed of optimal transition altitude selection. Finally, example results are presented and discussed.

CHAPTER 1

INTRODUCTION

Airdrops are a standard technique for delivering various equipment to inaccessible areas. Used in both military and humanitarian efforts, cargo is deployed from the aircraft and allowed to safely descend to ground under a parachute. Depending on the mission requirements, guided or unguided airdrops are used.

Guided airdrops are used in cases where highly accurate and precise impacts are required. For these situations, the parachutes are controlled by a mechanical system as the package descends, as shown in Figure 1.1. This controllable flight path allows these guided airdrops to excel in complex impact scenarios – such as hitting a specific target or avoiding certain obstacles – and results in small impact dispersions. However, these guided airdrop systems are often expensive and impractical for large numbers of drops [1]. The current state of the art in guided airdrops is the Precision Airdrop System (PADS) [2-4]. The PADS control system uses a model predictive framework in which a trajectory is propagated from the aircraft release point to the desired ground impact location. While this system provides favorable accuracy with respect to a desired impact location, the error dispersion cannot be shaped or changed.

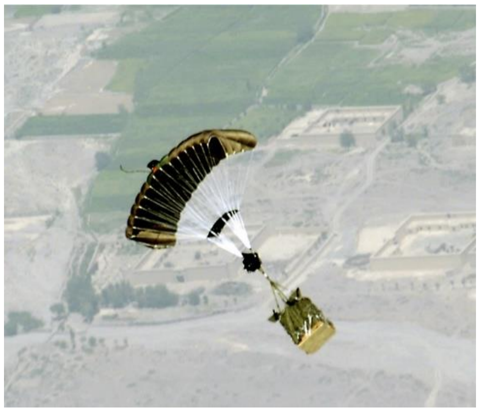


Figure 1.1: The guided PADS being controlled during its descent (Image credit: http://www.defenseindustrydaily.com/jpads-making-precision-airdrop-a-reality-0678/)

Unguided airdrop, also called ballistic airdrop, is used in cases where specific impacts are unimportant, and an impact dispersion is preferable instead. For example, it might be desirable to land packages anywhere between a forward operating base and a body of water, i.e. as long as the package does not hit some exclusion region, it does not matter where it lands. While not as accurate as guided airdrops, these unguided airdrops utilize knowledge of the atmosphere, parachute dynamics, and aircraft state to compute an air release point that allows packages to land in a desired location. Because unguided airdrops are less expensive and able to handle large numbers of drops at once, it is desirable to increase the impact accuracy. An example of unguided airdrops is shown in Figure 1.2.

Figure 1.2: Unguided airdrop of humanitarian relief after the Haiti earthquake of 2010 (Image credit: http://media.defense.gov/2010/Jan/19/2000402702/-1/-1/0/100118-F-4177H-657.JPG)

There have been many previous attempts to improve ballistic airdrop accuracy. Avenues explored include minimizing the primary sources of uncertainty by using high resolution wind modeling systems [5-6], increasing the fidelity of the dynamic models that compute the CARP [7-8], and accounting for the coupled aircraft-package dynamics and relating the drag characteristics of the package to its inertial properties [9]. Unfortunately, wind and atmospheric uncertainties play a large role in airdrop impact accuracy and dispersion, and higher fidelity dynamic models cannot account for this. Additionally, current mission planning techniques [2-4] rely on a single desired impact point and cannot control the shape of the unavoidable dispersion error.

High Altitude, Low Opening (HALO) airdrops are considered as the ballistic airdrop scheme in this paper. When released from the aircraft, the package descends under a drogue parachute until it reaches the determined transition altitude. The drogue then detaches from the parachute-package system, and the main parachute deploys and inflates, significantly slowing the package's descent to ground. The transition altitude is pre-programmed in current systems. By altering the transition altitude, the package impact location is moved along a "line of control" that is determined by wind conditions [10]. Thus, controlling the transition altitude can control the package impact location along this line of control.

Being able to specify an impact distribution affords the ability to account for various complex impact scenarios. This specification of an impact distribution proves advantageous in increasing the accuracy of ballistic airdrops. However, due to the multivariate uncertainty, the Computed Air Release Point (CARP) calculation must now take into account the desired impact distribution, i.e. use the desired impact distribution as an input to the mission planner instead of as an output as in current mission planners. The proposed solution described in this paper is as follows: given a desired impact distribution or cost map on the ground and knowledge of winds in a specific volume, an airdrop package selects an optimized transition altitude that allows the package to impact on a given target with the only control input being the main parachute deployment, i.e. the transition altitude is treated as a single-use control input.

The transition altitude optimization is accomplished in 4 main steps. First, a desired impact distribution or cost map that would result in the ideal impact dispersion pattern at ground is specified. This impact distribution is then combined with model uncertainty and wind uncertainty to produce a joint probability density function (PDF) at ground. Next, this PDF is propagated backwards through time using the Stochastic Liouville Equation (a simplified form of the Fokker-Planck Equation) [11], a method of nonlinear uncertainty propagation. This now provides a time-history of the PDF from the ground to the drop altitude. Next, the PDF at altitude is analyzed to pick the CARP and aircraft heading, a process beyond the scope of this thesis (a full description of this procedure can be found in [12]). Finally, the intermediate PDFs are analyzed in real-time as the package falls under drogue descent to determine the best transition altitude for the main parachute in order to reduce dispersion error. The selection of the transition altitude is based on various criteria discussed in Chapter 3 and is repeated in a feedback manner during drogue descent to reduce the effects of trajectory prediction errors.

This thesis presents the following contributions to the transition altitude optimization algorithm scheme: implementation and validation of the simulation framework, exploration of different boundary value problem (BVP) solution methods for trajectory generation in the uncertainty propagation procedure, and creation of the Custom Scenario Creator tool for producing complex desired impact distributions. To contextualize these contributions, previous research from [9], [12], and [23] is used as a base of understanding.

The thesis is outlined as follows. The simulation framework is first explored in Chapter 2. This includes the derivation of the parachute-package dynamic model. Atmospheric and uncertainty models are also introduced in this section. Chapter 3 begins the description of the proposed transition altitude optimization algorithm, detailing the steps of the scheme. A detailed description of alternative boundary value problem solution methods is provided in this section. A tool for creating complex desired impact distribution and inverse cost maps in presented in detail in Chapter 4. Development of this tool was vital in producing real-world scenarios. Chapter 5 presents the results of the proposed algorithm for a variety of example cases involving spatially varying winds, exclusion regions, and complex impact distributions. Finally, Chapter 6 provides the conclusion and a look at future work in this topic.

CHAPTER 2

SIMULATION FRAMEWORK

In this section, the simulation framework is detailed, starting first with the derivation of the parachute-package dynamics. Descriptions of the atmospheric and uncertainty models then follow, and a brief discussion of the nonlinear uncertainty propagation process is described. Finally, the software workflow of the simulation framework is briefly described.

Parachute-Package Dynamics

A HALO ballistic airdrop refers to the high altitude package deployment and low altitude during main parachute inflation, as shown in Figure 2.1. It is assumed that the drogue parachute inflates instantly upon package deployment and that the parachute-package system reaches a quasi-steady-state descent at the stabilization point. After reaching a pre-computed transition altitude, z_t, the drogue parachute detaches from the package, and the main parachute is deployed. The parachute-package system then enters another quasi-steady-state decent under the main parachute until ground impact at point PI.

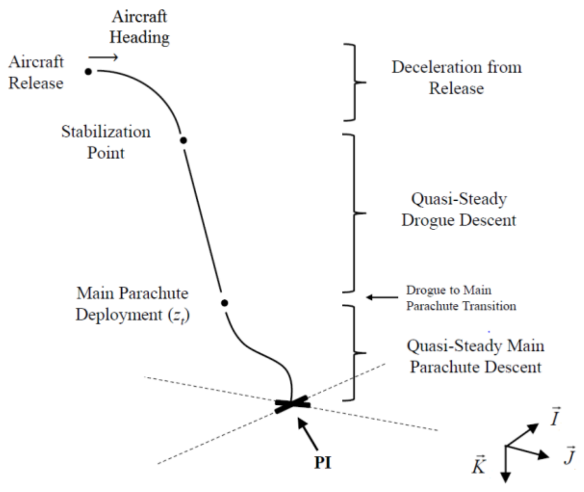

Aircraft Heading
Aircraft Release
Stabilization Point
Main Parachute Deployment (z_t)
Deceleration from Release
Quasi-Steady Drogue Descent
Drogue to Main Parachute Transition
Quasi-Steady Main Parachute Descent
PI
$\vec{I}$
$\vec{J}$
$\vec{K}$

Figure 2.1: HALO Airdrop Schematic

In deriving the dynamics of the parachute-package system, an inertial North-East-Down coordinate system is used with the package center of mass position given as,

$$\vec{r}_{C/O} = x\hat{I} + y\hat{J} + z\hat{K} \tag{1}$$

where the inertial coordinate system is denoted with $\hat{I}, \hat{J}$, and $\hat{K}$. Starting from a simple drag equation, the drag force on the parachute-package system is determined as,

$$\vec{F}_D = \frac{1}{2}\rho S C_d \left\|\vec{V}_{rel}\right\| \vec{V}_{rel} \tag{2}$$

where $\rho = f(x, y, z)$ is the air density at the current point, S is the parachute reference area (i.e. a circle with the same radius as the parachute), C_d is the parachute coefficient of drag, and $\vec{V}_{rel}$ is the velocity of the parachute-package system relative to the wind in the inertial frame,

$$\vec{V}_{rel} = \begin{bmatrix} \dot{x} - w_x \\ \dot{y} - w_y \\ \dot{z} - w_z \end{bmatrix} = \vec{v}_p - \vec{v}_w \tag{3}$$

with $\vec{v}_w = w_x\hat{I} + w_y\hat{J} + w_z\hat{K}$ being the wind velocity and $\vec{v}_p = \dot{x}\hat{I} + \dot{y}\hat{J} + \dot{z}\hat{K}$ being the parachute-package system velocity.

The air mass captured by the parachute presents non-negligible effects on the parachute-package system. This mass, called the apparent mass m_a, is commonly approximated as the volume of air mass enclosed by a sphere that has the same radius as the parachute reference area,

$$m_a = \frac{4}{3}k_a \rho \pi R^3 \tag{4}$$

where R is the parachute radius and k_a is the apparent mass coefficient [9]; k_a is determined by the parachute geometry and porosity.

Because the parachute-package system mass is changing during flight due to the apparent mass, Newton's Second Law cannot be applied solely to the parachute-package system. Instead, define a control volume around the parachute-package system with the following assumptions: (a) the air in the apparent mass volume is moving at the same velocity as the package, and (b) the air outside of the apparent mass volume is moving at the same velocity as the wind field. Newton's Second Law is now applied to this control volume,

$$\vec{F}_D + m\vec{g} = \frac{d}{dt}\left[m_{ext}\vec{v}_w + (m + m_a)\vec{v}_p\right] \tag{5}$$

where m_{ext} is the air mass in the control volume but outside of the apparent mass volume, and the derivative is taken in the inertial frame. Additionally, note that the weight of the apparent mass is cancelled by its buoyancy, and therefore, the term is not in the equation. Substituting $m_{int} = m + m_a$ into Eq. (5) and expanding the derivative yields,

$$\vec{F}_D + m\vec{g} = \dot{m}_{ext}\vec{v}_w + m_{ext}\dot{\vec{v}}_w + \dot{m}_{int}\vec{v}_p + m_{int}\dot{\vec{v}}_p \tag{6}$$

Knowing $\dot{m}_{ext} = -\dot{m}_a$ and assuming $\dot{\vec{v}}_w = 0$,

$$\vec{F}_D + m\vec{g} = -\dot{m}_a\vec{v}_w + \dot{m}_a\vec{v}_p + m_{int}\dot{\vec{v}}_p \tag{7}$$

$$\vec{F}_D + m\vec{g} = \dot{m}_a\left(\vec{v}_p - \vec{v}_w\right) + (m + m_a)\dot{\vec{v}}_p \tag{8}$$

From Eq. (4), the time derivative can be computed,

$$\dot{m}_a = 4k_a\rho\pi R^2\dot{R} \tag{9}$$

where the assumption is made that the contributions from $\dot{\rho}$ are negligible to the parachute-package weight. Substituting Eq. (2), Eq. (3), and Eq. (9) into Eq. (8), $\dot{\vec{v}}_p$ can be solved for,

$$\frac{1}{2}\rho S C_d\left\|\vec{V}_{rel}\right\|\vec{V}_{rel} + m\vec{g} = 4k_a\rho\pi R^2\dot{R}\vec{V}_{rel} + (m + m_a)\dot{\vec{v}}_p \tag{10}$$

$$\dot{\vec{v}}_p = \left[\frac{\rho S C_d\left\|\vec{V}_{rel}\right\| - 8k_a\rho\pi R^2\dot{R}}{2(m + m_a)}\right]\vec{V}_{rel} + \frac{m}{m + m_a}\vec{g} \tag{11}$$

Expanding this equation for clarity yields the equations of motion,

$$\begin{bmatrix}\ddot{x}\\ \ddot{y}\\ \ddot{z}\end{bmatrix} = \frac{\rho S C_d\left\|\vec{V}_{rel}\right\| - 8k_a\rho\pi R^2\dot{R}}{2(m + m_a)}\begin{bmatrix}\dot{x} - w_x\\ \dot{y} - w_y\\ \dot{z} - w_z\end{bmatrix} + \frac{m}{m + m_a}\begin{bmatrix}0\\ 0\\ g\end{bmatrix} \tag{12}$$

From [9], the rate of change of the parachute radius is given by,

$$\dot{R}(t) = \frac{6R_0(1 - \eta)}{t_0^6}\left[(1 - \eta)t^5 + \eta t_0^3 t^2\right] \tag{13}$$

where R_0 is the fully inflated parachute radius, t_0 is the parachute opening time, and η is "the ratio of projected mouth area at line stretch to the steady-state projected frontal area" [9]. The parachute opening time t_0 is modeled base on experimental data as,

$$t_0 = \left(\frac{0.339}{0.121 - C_{eff}}\right)\frac{2R_0}{V_S} \tag{14}$$

where C_{eff} is the parachute effective porosity and V_S is the snatch velocity of the parachute deployment [9]. Similarly, η is modeled experimentally as,

$$\eta = \frac{\sqrt{\frac{S}{S_0}} - \left(\frac{t}{t_0}\right)^3}{1 - \left(\frac{t}{t_0}\right)^3} \tag{15}$$

where S_0 is the fully inflated parachute reference area and S is the instantaneous parachute reference area based on the current radius [9].

By substituting Eq. (13), Eq. (14), and Eq. (15) into Eq. (12), a system of nonlinear ordinary differential equations is produced that is solved numerically with a 4th order Runge-Kutta integrator. The parachute-package system state vector is constructed as,

$$\vec{x} = [x \quad y \quad z \quad \dot{x} \quad \dot{y} \quad \dot{z}]^T = [x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6]^T \tag{16}$$

and thus,

$$\dot{\vec{x}} = [\dot{x}_1 \quad \dot{x}_2 \quad \dot{x}_3 \quad \dot{x}_4 \quad \dot{x}_5 \quad \dot{x}_6]^T = [x_4 \quad x_5 \quad x_6 \quad \dot{x}_4 \quad \dot{x}_5 \quad \dot{x}_6]^T \tag{17}$$

Atmospheric Model

Three-dimensional wind field data from Weather Research and Forecasting (WRF), an atmospheric prediction and modeling tool, is examined [29]. This model gives three-dimensional wind vectors on a grid of (x, y, z) locations, which can then be interpolated to determine the wind at any point on an airdrop trajectory.

Although referenced above, the wind vector at a given (x, y, z) is now defined as,

$$\vec{v}_w(x, y, z) = \vec{w}(x, y, z) = w_x\hat{I} + w_y\hat{J} + w_z\hat{K} \tag{18}$$

The wind azimuth angle is simply defined as,

$$\bar{w}_{dir} = \tan^{-1}(w_y/w_x) \tag{19}$$

and the wind magnitude simply as,

$$\bar{w}_{mag} = \|\vec{w}\| = \sqrt{w_x^2 + w_y^2 + w_z^2} \tag{20}$$

Finally, instead of using a spatially-varying air density model, a simpler model is used instead where the air density varies as a function of altitude [13],

$$\rho(z) = 1.2257 \text{ x } (1.0 + 0.00002257z)^{4.258} \tag{21}$$

Should a more complex model be desired, this computation can simply be replaced in the algorithm.

Uncertainty Model

For aerospace vehicles, wind uncertainty is normally modeled using the high-fidelity Dryden gust and turbulence model [14]. However, uncertainty propagation with this model can present several computational difficulties [15-16]. Therefore, the wind uncertainty is simplified by assuming the wind errors are a perturbation to the total magnitude, Eq. (20), and direction, Eq. (19), and are uniform across the entire wind field. Thus, the wind magnitude and direction becomes,

$$w_{mag}(x,y,z) = \bar{w}_{mag} * \hat{w}_{mag} \tag{22}$$

$$w_{dir}(x,y,z) = \bar{w}_{dir} + \hat{w}_{dir} \tag{23}$$

with the wind magnitude uncertainty and wind direction uncertainty terms defined as,

$$\hat{w}_{mag} = \mathcal{N}(1, \sigma_{mag}) = 1 \pm \sigma_{mag} \tag{24}$$

$$\hat{w}_{dir} = \mathcal{N}(0, \sigma_{dir}) = 0 \pm \sigma_{dir} \tag{25}$$

where $\mathcal{N}(*,*)$ is the normal distribution.

Another major source of error comes from uncertainty of the dynamic model, which is manifested as uncertainty of the drag coefficient of the main parachute. Note that the drag coefficient of the drogue parachute does not present appreciable uncertainties since the transition altitude optimization algorithm is used in a feedback manner, i.e. the optimal transition altitude is continually recomputed during the drogue descent. Similar to the wind uncertainty, the drag coefficient error is assumed as a perturbation to the mean value,

$$C_d = \mathcal{N}(\bar{C}_d, \sigma_c) = \bar{C}_d \pm \sigma_c \tag{26}$$

Since wind and dynamic model uncertainty account for most of the impact error of ballistic airdrops [17], the uncertainty model focuses only on these areas.

Uncertainty Propagation

Monte Carlo simulation is normally used as the standard method of uncertainty propagation in airdrop computation because of its ease of implementation and lack of alternative methods. By simulating a large number of trajectories from randomly sampled initial conditions, a history of the uncertainty evolution can be built. However, Monte Carlo methods break down for high-dimensional systems: computationally scalability is difficult to achieve [18]; implementation of necessary Markov Chain Monte Carlo methods is non-trivial [19]; and uncoherent quantification of uncertainty evolution [20-21] (i.e. the PDF cannot be examined at regular time intervals) present themselves.

Instead, a direct method of uncertainty propagation is used whereby the PDF transport equation is solved directly compared to approximating the solution from various samples. The Fokker-Planck-Kolmogorov (FPK) equation is the general PDF transport equation that describes the time evolution of the joint probability density over the state space [22]. However, a solution to the FPK is largely computationally intractable expect for low-dimensional problems because it is a partial differential equation in the number of dimensions of the state space [15-16]. This problem is mitigated by reducing the FPK equation to the Stochastic Liouville Equation (SLE) by neglecting noise in the system and only considering parametric uncertainty. The SLE is a quasi-linear partial differential equation that is first order in both time and space.

While the full derivation of the SLE is beyond the scope of this thesis, it is noted that the SLE can be reduced to an ordinary differential equation of the form,

$$\frac{d\varphi(\vec{X},t)}{dt} = -\varphi(\vec{X},t)\Psi(\vec{x}(t)) \tag{27}$$

along a trajectory of the dynamic system where $\boldsymbol{\varphi}(\vec{\boldsymbol{X}},\boldsymbol{t})$ is the time-varying joint PDF of the augmented state $\vec{\boldsymbol{X}} = [\vec{\boldsymbol{x}} \;\; \vec{\boldsymbol{p}}]^{\boldsymbol{T}}$ with $\vec{\boldsymbol{x}} \in \mathbb{R}^{\boldsymbol{n_s}}$ and the parameter vector $\vec{\boldsymbol{p}} \in \mathbb{R}^{\boldsymbol{n_p}}$. $\boldsymbol{\Psi}(\vec{\boldsymbol{x}}(\boldsymbol{t}))$ is the trace of the Jacobian of $\boldsymbol{h}(\vec{\boldsymbol{x}},\vec{\boldsymbol{p}})$ evaluated at $\vec{\boldsymbol{x}}(\boldsymbol{t})$ defined as,

$$\Psi(\vec{x}(t)) \equiv \sum_{i=1}^{n_s} \frac{\partial h_i}{\partial x_i}\bigg|_{\vec{x}(t)} \tag{28}$$

where $\dot{\vec{\boldsymbol{x}}} = \boldsymbol{h}(\vec{\boldsymbol{x}},\vec{\boldsymbol{p}})$. Eq. (27) is then numerically integrated from an initial condition $\boldsymbol{\varphi_0}$.

Since the ODE form of the SLE is valid only along a trajectory of the dynamic system, Eq. (12) must be numerically integrated followed by Eq. (27) for each parachute-package system trajectory. To compute the time and space evolution of the entire PDF, the initial joint density $\boldsymbol{\varphi_0}$ must first be discretized across the uncertainty space; this creates an ensemble of initial conditions for the SLE at ground. From these discretized initial conditions, trajectories are propagated to package drop altitude through a boundary value problem (BVP) solver. The SLE is then computed along these trajectories, resulting in a time-varying joint PDF from ground to package drop altitude. For a full description and analysis of this procedure, the reader is directed to [23].

Software Workflow

The software workflow is shown in Figure 2.2. A full parachute dynamics simulation is implemented in MATLAB, including the inflation model and uncertainty model described above. The parachute simulation requires as input an initial condition, a wind model, and simulation parameters. The wind model can take the form of constant winds, wind stick data, or 3D wind fields. The simulation parameters including data logging and the ability to turn off the transition altitude optimization algorithm, among others. The mission planner and transition altitude optimization are discussed in Chapter 3.

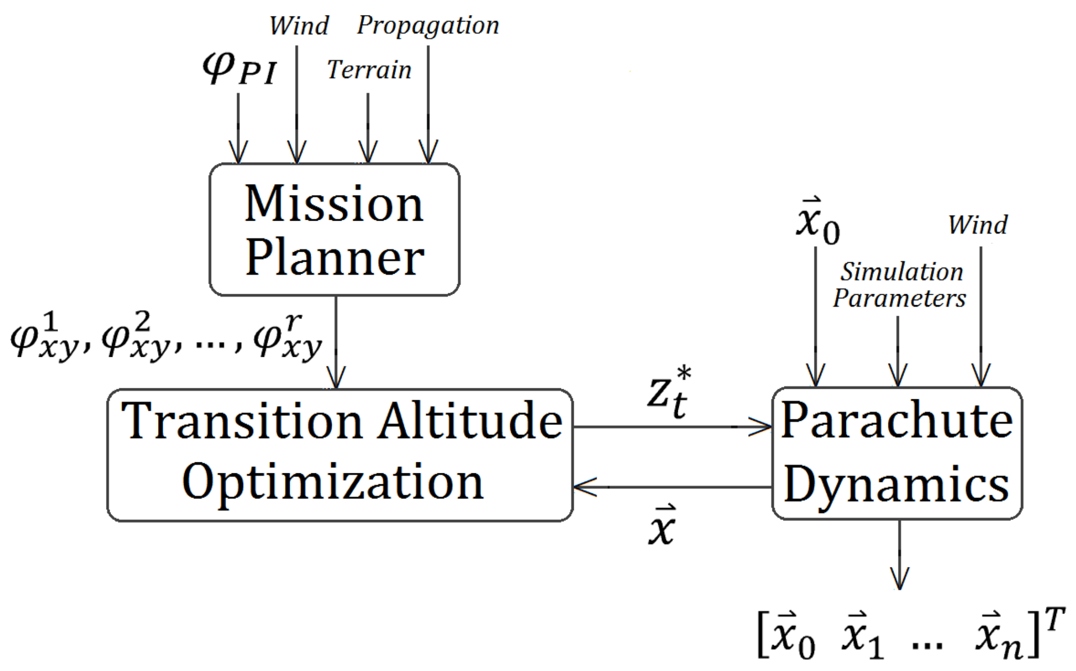

Wind
Propagation
φ_{PI}
Terrain
Mission Planner
$\vec{x}_0$
Wind
Simulation Parameters
$\varphi_{xy}^1, \varphi_{xy}^2, \dots, \varphi_{xy}^r$
z_t^*
Transition Altitude Optimization
Parachute Dynamics
$\vec{x}$
$[\vec{x}_0 \;\; \vec{x}_1 \;\; \dots \;\; \vec{x}_n]^T$

Figure 2.2: Software workflow

The path of execution is as follows. A wind model, propagation model, terrain model, and desired impact distribution are used as input to the mission planner. The mission planner then computes the marginal $\boldsymbol{x}$-$\boldsymbol{y}$ probability densities. The transition altitude optimization takes the marginal densities and determines the optimal transition altitude $\boldsymbol{z_t^*}$. These two processes are detailed in Chapter 3. As the parachute-package system dynamics are simulated from drop altitude to ground, the transition altitude optimization receives the current state vector of the parachute-package system at some specified frequency. This simulates the transition altitude optimization algorithm receiving state data from onboard sensors in a real-world application. The parachute dynamics simulation then checks the transition altitude value. When the parachute-package system has descended to or below the transition altitude, the main parachute is deployed. A full state time-history is produced upon ground impact.

CHAPTER 3

TRANSITION ALTITUDE OPTIMIZATION ALGORITHM

The overall transition altitude optimization scheme is shown in Figure 3.1. The offline mission planner procedure follows. First, a desired impact distribution is specified. Next, the joint probability density is created and discretized at ground, i.e. the joint PDF at ground is now known. For each discretized point in the ground PDF, a BVP is solved to produce the package trajectory from the maximum transition altitude to ground impact. Back-propagating the SLE along each trajectory then yields the joint PDF for discrete locations along the trajectory. The $\boldsymbol{x}$-$\boldsymbol{y}$ marginal density at discrete altitudes is then computed, producing "slices" of the joint PDF. These "slices" are then stored on the package for onboard computation of the transition altitude optimization.

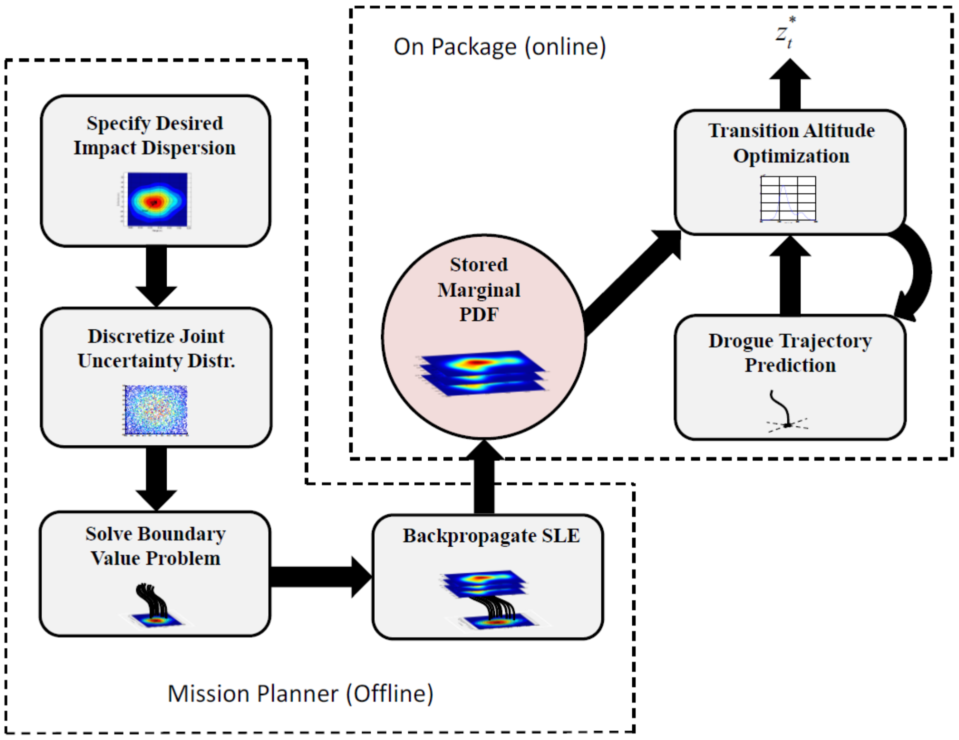

On Package (online)
z_t^*
Specify Desired Impact Dispersion
Transition Altitude Optimization
Stored Marginal PDF
Discretize Joint Uncertainty Distr.
Drogue Trajectory Prediction
Solve Boundary Value Problem
Backpropagate SLE
Mission Planner (Offline)

Figure 3.1: Probabilistic transition altitude optimization process

The onboard procedure now follows. A maximum transition altitude $\boldsymbol{z_{t,max}}$ and minimum transition altitude $\boldsymbol{z_{t,min}}$ (i.e. maximum and minimum altitudes that the main parachute can deploy) are set. Using position and velocity feedback, the drogue descent trajectory is predicted from its current altitude down to $\boldsymbol{z_{t,min}}$. The marginal PDF along this trajectory is then computed, and the optimal transition altitude $\boldsymbol{z_t^*}$ based on various criteria is selected. The determination of $\boldsymbol{z_t^*}$ is then repeated in a feedback loop as the descent continues. Once the package altitude drops to or below $\boldsymbol{z_t^*}$, the main parachute deploys. These individual steps are examined in the sections below.

Joint Distribution Specification and Discretization

A desired impact distribution, denoted $\varphi_{PI}(x, y)$, is required as input to the algorithm. This distribution is a normalized function of x and y, taking the form of an impact probability density or an inverse cost map (i.e. high cost regions represent desired landing regions and lost cost regions represent regions to avoid). Examples of desired impact distributions include 2D uniform distributions, Gaussian distributions, or complex distributions based on arbitrary map data, as seen in Figure 3.2.

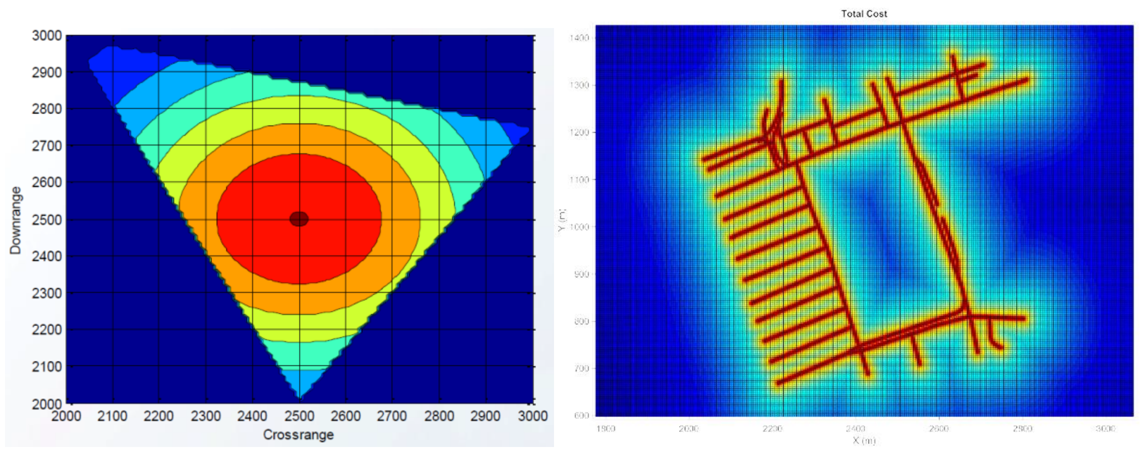


Figure 3.2: Examples of desired impact distributions

Once a desired impact distribution is achieved, the joint probability density can be computed. Comprised of the uncertainty distribution of the desired impact point and the uncertainty distributions for the n_p uncertain parameters, the joint probability density at the ground φ_0 is the product density of the parametric uncertainty distributions and impact distribution,

$$\varphi_0 = \varphi_{PI}(x,y)\prod_{i=1}^{n_p}\varphi_i(p_i) \tag{29}$$

where $\varphi_i(p_i)$ is the uncertainty distribution associated with the i^{th} uncertain parameter. Note that it is assumed all of the uncertainty distributions are uncorrelated at the ground impact time. Additionally, the uncertainty distributions for the uncertainty parameters may take the form of any valid probability distribution function.

The ground joint probability density is then discretized into N vectors of $n_p + 2$ dimension, where N is the number of points in the desired impact distribution. The i^{th} discretized vector in the uncertainty space is denoted $\bar{\vec{s}}_\imath$, where $\bar{\vec{s}}_\imath \in R^{n_p+2}$ and $i \in [1, N]$. Therefore, the discretization process results in N vectors $\bar{\vec{s}}_\imath$ and N associated probability values. The $\bar{\vec{s}}_\imath$ vectors are referred to as "samples" throughout the rest of the thesis. Note, however, that these samples are not drawn randomly from the underlying distributions as in Monte Carlo, but are points in the discretized uncertainty space.

Boundary Value Solution

To solve the SLE as described in Eq. (27), trajectories of the dynamic system must first be generated for each sample $\bar{\vec{s}}_\imath$. While the impact point of a particular sample is known from the discretization process of the desired impact distribution, the trajectory of the sample – using that particular sample's realization of the uncertain parameters – is unknown. The trajectory of the parachute-package system is computed from $z = 0$ to $z = z_{t,max}$ under the main parachute dynamics. This trajectory generation problem is a two-point boundary value problem (BVP) that is formulated as,

Given $\bar{\vec{s}}_\iota$ *and initial state values of the system at* $z_{t,max}$ *as,*

$$\begin{cases} z(t_0) = z_{t,max} = z_0 \\ \dot{x}(t_0) = w_{mag}\left(x_0, y_0, z_{t,max}\right) cos\left(w_{dir}\left(x_0, y_0, z_{t,max}\right)\right) = \dot{x}_0 \\ \dot{y}(t_0) = w_{mag}\left(x_0, y_0, z_{t,max}\right) sin\left(w_{dir}\left(x_0, y_0, z_{t,max}\right)\right) = \dot{y}_0 \\ \dot{z}(t_0) = \dot{z}_{ss} = \dot{z}_0 \end{cases} \tag{30}$$

find the package position $x(t_0) = x_0$, $y(t_0) = y_0$ *subject to the dynamics in Eq. (12)*

where the initial time $t_0 = 0$ is defined such that $z(t = t_0) = z_{t,max}$, x_0 and y_0 are the package positions at $z_{t,max}$, and $\dot{z}_{ss}$ is the quasi-steady-state parachute-package descent speed under the main parachute given by [30],

$$\dot{z}_{ss} = \sqrt{\frac{2mg}{\rho(z)C_dS}} \tag{31}$$

Due to the larger number of samples, it is important to have an efficient BVP solver. A single shooting method (SSM) is currently used for solving BVPs in the existing mission planner. Its convergence failure rate in 3D wind fields is higher than desired, thus necessitating a more robust method; the SSM is thus compared with a more robust method, the modified simple shooting method (MSSM), below.

Single Shooting Method

The single shooting method for solving boundary value problems transforms the BVP into an initial value problem (IVP). For the trajectory generation in Eq. (30), there are n = 6 total states and m = 4 known states at the initial condition t_0. Thus, there are $n - m$ = 2 unknown states that must be "guessed", the initial x and y positions of the parachute-package system at drop altitude $z_{t,max}$. The fully complete initial state vector $\vec{x}(t_0)$ is then used as the initial condition to the dynamic system, and the system is integrated to ground impact at time t_f. The error between the desired impact point (x_d, y_d) and the actual impact point (x_a, y_a) is computed. Based on this error, a damped Newton-like Method is used to reduce the error to 0 and generate a new guess for x_0 and y_0 using an adaptive damping factor λ. The BVP solution algorithm proceeds as follows:

(1) *Choose initial guesses for* x_k *and* y_k *to complete the intial state vector* $\vec{x}(t_0) = [x_k \;\; y_k \;\; z_0 \;\; \dot{x}_0 \;\; \dot{y}_0 \;\; \dot{z}_0]^T$ *as the initial conditions, initialize iteration counter* k *to 0, and choose the error tolerance for solving completion as* $\varepsilon > 0$

(2) *Integrate the dynamic system of Eq. (12) to ground impact*

(3) *Compare the distance between the actual impact point and desired impact point* $\begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} x_a - x_d \\ y_a - y_d \end{bmatrix}$ *and compute an error formulation* $e_{k+1} = \Delta x^2 + \Delta y^2$

(4) *If* $e_{k+1} < \varepsilon$*, break out of solver; otherwise, continue to step (5)*

(5) *(Only compute this step if* $k > 0$*) If* $e_{k+1} < e_k$*, increase* λ *by a small amount; otherwise, decrease* λ *by a small amount*

(6) *Determine a new guess using,* $\begin{bmatrix} x_{k+1} \\ y_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ y_k \end{bmatrix} - \lambda \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix}$

(7) *Increment* k

(8) *Return to step (2)*

This method is illustrated in Figure 3.3. The SSM is advantageous because it is easy to implement and fast to compute, but may fail to converge without a good starting guess in some wind fields.

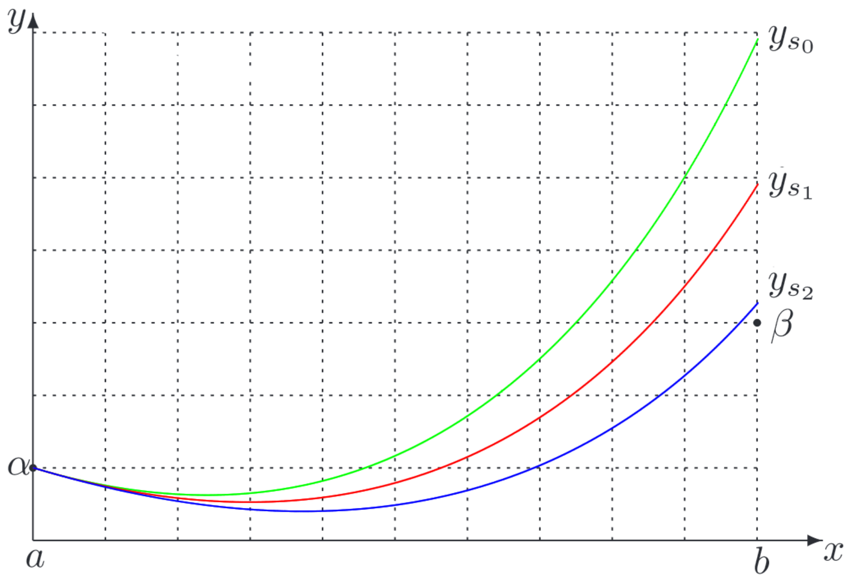


Figure 3.3: A visualization of the single shooting method, where it takes three iterations to get close to the final boundary conditions [25]

Modified Simple Shooting Method

The modified simple shooting method begins similar to the SSM in that the BVP is transformed to an IVP and initial guesses must be chosen. The notion of a "reference path" is introduced. The reference path $\vec{r}(t)$ is a Lipschitz continuous function from the initial conditions to the final conditions such that $\vec{r}(t_0) = \vec{x}_0$ and $\vec{r}(t_f) = \vec{x}_f$; in most cases, a line from initial conditions to final conditions is used as the reference path. Bounds of width ε_b are placed around the reference path. As the system is integrated, violations of these boundaries are checked. If the system does go out of bounds, the integration is stopped at $t_{violate}$. The MSSM then enters another iteration loop to correct the guessed parameters using a modified Newton Method. In this correction iteration, the system is only integrated to $t_{violate}$. Once the system is within a certain tolerance ε of the reference path at $t_{violate}$, the correction iteration stops, and the system is integrated until another boundary violation occurs or the final condition is successfully reached. This procedure is illustrated in Figure 3.4, where "shot 1" (green) uses 3 corrections, shot 2 (red) uses 3 corrections, and shot 3 (blue) uses 2 corrections; the reference path is the solid black line from (a, α) to (b, β). In Figure

3.5 the reference path and bounds are more clearly illustrated; the reference path is the dashed black line from $(0,1)$ to $(1,2)$ and the bounds are shown as dashed green lines. A full description of the MSSM is provided in [24] and [25].

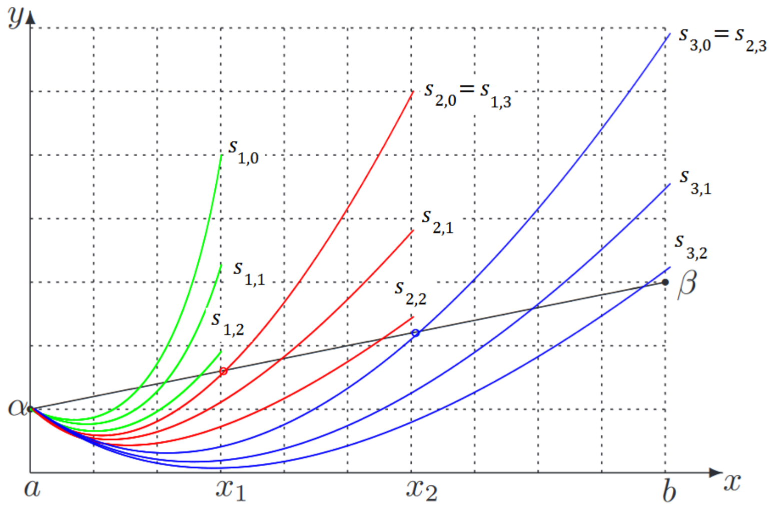


Figure 3.4: Visualization of the modified simple shooting method [25]

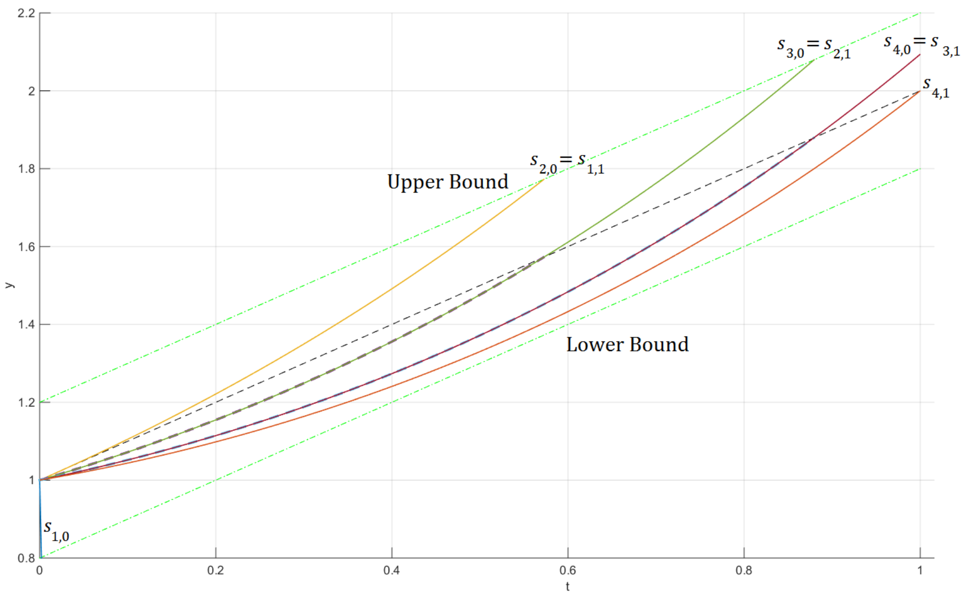


Figure 3.5: Visualization of the modified simple shooting method

Unfortunately, the BVP in Eq. (30) is not formulated in the exact manner for which the MSSM was defined in reference [25]. However, some small modifications to the method result in a formulation that is more applicable to the BVP considered here. First, z is chosen as the independent variable. Next, instead of a reference path from z_0 to z_f, a constant desired value is used. Finally, the parallel bounds are replaced by bounds that continuously shrink as they approach the final condition. These modifications are shown in Figure 3.6, where the desired x value is 2000 m, the desired y is 3000 m, z_0 is -1000 m, z_f is 0 m (i.e. ground), and the bounds decrease from an ε_b of 10,000 m at the initial conditions to 20 m at the final conditions. The remaining solving conditions and methods are kept the same as in the standard MSSM. Thus, the final algorithm proceeds as follows:

(1) *Choose initial guesses for* $x_{k,c}$ *and* $y_{k,c}$ *to complete the intial state vector* $\vec{x}(t_0) = \begin{bmatrix} x_{k,c} & y_{k,c} & z_0 & \dot{x}_0 & \dot{y}_0 & \dot{z}_0 \end{bmatrix}^T$ *as the initial conditions, define initial conditions guesses as* $\vec{s}_{k,c} = \begin{bmatrix} x_{k,c} \\ y_{k,c} \end{bmatrix}$ *and thus the initial guess associated with "shot" 1as* $\vec{s}_{1,0} = \begin{bmatrix} x_{1,0} \\ y_{1,0} \end{bmatrix}$, *initialize iteration counter* k *to 1, initialize correction counter* c *to 0, and choose the error tolerance for solving completion as* $\varepsilon > 0$

(2) *Integrate the dynamic system of Eq. (12) until a boundary is violated*

(3) *If no boundary is violated (i.e. the system stayed in bounds all the way to ground impact), go to step (8); otherwise, stop integration at the boundary violation, now called* $z_{violate}$, *and continue to step (4)*

(4) *Compute a "correction" guess using a modified or damped Newton Method,*

$$\vec{s}_{k,c+1} = \vec{s}_{k,c} - \lambda(DF)^{-1}F$$

where λ *is the adaptive damping factor,* DF *is the Jacobian, and* F *is the error between actual and desired position.*

a. *The Jacobian is defined,*

$$DF = \begin{bmatrix} \frac{\partial x_f}{\partial x_i} & \frac{\partial x_f}{\partial y_i} \\ \frac{\partial y_f}{\partial x_i} & \frac{\partial y_f}{\partial y_i} \end{bmatrix}$$

$$\therefore DF = \begin{bmatrix} \frac{1}{2h}\left[x_f(x_i+h,y_i) - x_f(x_i-h,y_i)\right] & \frac{1}{2h}\left[x_f(x_i,y_i+h) - x_f(x_i,y_i-h)\right] \\ \frac{1}{2h}\left[y_f(x_i+h,y_i) - y_f(x_i-h,y_i)\right] & \frac{1}{2h}\left[y_f(x_i,y_i+h) - y_f(x_i,y_i-h)\right] \end{bmatrix}$$

where the notation $x_f(x_i + h, y_i)$ *represents the final* x*-position of the system at altitude* $z_{violate}$ *using* $x_i + h$ *and* y_i *as guesses to complete the initial conditions. Thus, four additional integrations are computed from* z_0 *to* $z_{violate}$*: one for each variation of guesses.*

b. *The error is defined,* $F = \begin{bmatrix} e_x \\ e_y \end{bmatrix} = \begin{bmatrix} x_a - x_d \\ y_a - y_d \end{bmatrix}$ *where* x_a *is the actual* x*-position of the system at* $z_{violate}$ *and* x_d *is the desired impact* x*-position (and also the reference path)*

c. *The damping factor* λ *is defined according to the increasing or decreasing of a scalar error. Compute an error formulation* $e_{c+1} = {e_x}^2 + {e_y}^2$*. If* $e_{c+1} < e_c$*, increase* λ *by a small amount; otherwise, decrease* λ *by a small amount (only compute this comparison when* $k > 1$*).*

(5) *Integrate the dynamic system of Eq. (12) from* z_0 *to* $z_{violate}$ *using guess* $\vec{s}_{k,c+1}$

(6) *Compute a final tolerance formulation* $e_{tol} = \sqrt{(x_a - x_d)^2 + (y_a - y_d)^2}$ *where* x_a *is now the actual* x*-position at altitude* $z_{violate}$ *using guess* $\vec{s}_{k,c+1}$

(7) *Increment* c*. If* $e_{tol} > \varepsilon$*, return to step (4); otherwise, set* $\vec{s}_{k+1,0} = \vec{s}_{k,c}$*, reset* c *to 0, increment* k*, and return to step (2).*

(8) *The system has successfully stayed in bounds all the way to ground impact. Perform a final Newton iteration to produce the final correct guess as described in steps (4)-(7)*

A fully detailed implementation of the modified MSSM written in C++ is provided to the interested reader in Appendix A.

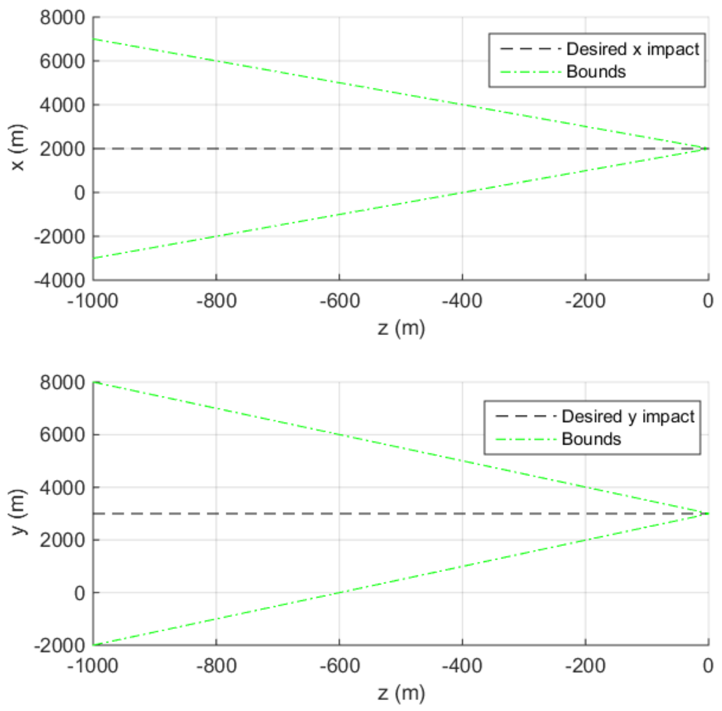


Figure 3.6: Custom realization of the MSSM showing the setup of the method

Method Comparison

Practical implementations introduce a further consideration: the methods cannot get stuck in an infinite loop. While not a regular occurrence, the solvers can sometimes get caught in a local minimum and fail to converge to a solution. The adaptive damping factor helps overcome this problem, but some cases have been noted where convergence is never reached. To prevent this, the SSM solving routine is aborted after 50 iterations. Similarly, the MSSM solving routine is aborted after 50 combined main shooting and correction iterations. Furthermore, the Newton correction routine of the MSSM switches from a 1st order approximation,

$$\vec{s}_{k,c+1} = \vec{s}_{k,c} - \lambda(DF)^{-1}F \tag{32}$$

to a 0th order approximation,

$$\vec{s}_{k,c+1} = \vec{s}_{k,c} - \lambda F \tag{33}$$

after 10 iterations have elapsed in the correction routine. Due to the high convergence rate of Newton Methods, it is assumed that after 10 iterations the solver is close to a solution, in which case switching to the 0th order approximation will still quickly arrive at the solution, or the system will never converge to a solution, in which case it is desirable to get as close to the solution as possible and additional computation time should not be wasted by numerically computing the Jacobian.

Both of the above methods are tested using the parachute-package dynamic system described in Eq. (12). The methods are implemented in C++ on an Ubuntu operating system. 500 points in the joint probability distribution at ground are randomly sampled to produce desired impact points, as well as values for the uncertainty parameters of C_d, w_{mag}, and w_{dir}.

Additionally, three different wind models are used in the comparison: constant wind, wind stick data, and 3D wind fields. The constant wind model assume a constant and uniform 7.0 m/s wind speed in the $+x$ direction (i.e. North) throughout the entire scenario; no wind exists in the y or z directions. The wind stick model keeps the x and y components of the wind vector constant for a given altitude with no velocity component in the z direction. For example, at altitude $z = z_1$, all wind vectors would be of the form $\vec{w} = w_{x,1}\hat{I} + w_{y,1}\hat{J}$, whereas at altitude $z = z_2$, all wind vectors would be of the form $\vec{w} = w_{x,2}\hat{I} + w_{y,2}\hat{J}$. Finally, the 3D wind fields varies all three components of the wind vector throughout the scenario space.

For the constant wind case, an ideal solver should be able to produce a solution in only 2 iterations. Once one iteration has been completed, the difference between actual and desired impact locations is simply applied to the original starting position; all choices of initial conditions will produce the same trajectory, they will just be translated in space. The comparison of the methods for constant wind is presented in Table 3.1. Both the SSM and MSSM successfully converged for all trials. However, the MSSM was able to converge for all 500 samples in identically 2 iterations, the ideal amount. The SSM, meanwhile, required many more iterations. This is due to the initial choice of $\lambda = 0.3$ for the SSM and $\lambda = 1$ for the

MSSM. Changing the value of λ for the SSM breaks the solver for other wind models. It is also noteworthy that the MSSM is able to produce the initial conditions that result in identically 0 error at ground impact, which the SSM cannot replicate.

Table 3.1: Constant wind BVP comparison

	SSM	**MSSM**
Total Runs	500	500
Number Not Converged	0	0
Average Number of Iterations	8.78	**2**
Average Iteration Time	**0.09 s**	0.10 s
Average Iterations / Runtime	**99.66**	20.54
Average Distance to Target at Ground	0.72 m	**0.0 m**

Similar to the constant wind case, an ideal solver using a wind stick model should also be able to produce a solution in only 2 iterations. Because the system always has the same wind values for any trajectory starting from the same altitude, the system trajectory is constant for any choice of initial conditions. The comparison for wind stick data is presented below in Table 3.2. Again, the MSSM is able to successfully converge for all 500 samples in the ideal 2 iterations, whereas the SSM converges for all 500 samples in identically 8 iterations. Similar to the constant wind case, the SSM's 8 iterations are due to the initial choice of $\lambda = 0.3$ for the SSM and $\lambda = 1$ for the MSSM. The SSM is slightly faster but produces a larger error at ground compared to the MSSM, although this error value is negligible for the purposes of this work.

Table 3.2: Wind Stick BVP comparison

	SSM	**MSSM**
Total Runs	500	500
Number Not Converged	0	0
Average Number of Iterations	8	**2**
Average Iteration Time	**0.09 s**	0.11 s
Average Iterations / Runtime	**104.19**	20.18
Average Distance to Target at Ground	0.79 m	**0.0 m**

While constant wind and wind stick models are useful for testing purposes, 3D wind fields represent the real-world data sets that these solvers need to work with. The comparison for 3D wind is presented below in Table 3.3. The MSSM is slightly more robust than the SSM, failing to converge for only 2.0% of its samples compared to 3.4% for the SSM. Additionally, it is seen that the MSSM converges much more rapidly than the SSM at the cost of slightly longer iteration times. Finally, the MSSM still results in trajectories that are closer to the desired impact points compared to the SSM.

Table 3.3: 3D Wind BVP comparison

	SSM	**MSSM**
Total Runs	500	500
Number Not Converged	17 (3.4%)	**10 (2.0%)**
Average Number of Iterations	13.17	**4.96**
Average Iteration Time	**0.25 s**	0.32 s
Average Iterations / Runtime	**60.34**	20.96
Average Distance to Target at Ground	0.85 m	**0.42 m**

While the performance of the MSSM is not as decisive for the 3D wind model, a further step is taken. The bounds of the MSSM are turned off, reducing the method to the currently used SSM, except with a 1st order approximation for the new guesses. The results are presented below in Table 3.4; note that these results were conducted with a 3D wind model. Noting that the full MSSM performs slightly better than the boundary-less version, these results show that the 0th order approximation in the currently used SSM is reducing performance. Using a 1st order approximation in the current SSM would likely produce results comparable with the full MSSM.

Table 3.4: Bounds vs No Bounds BVP comparison

	Bounds	**No Bounds**
Total Runs	500	500
Number Not Converged	10 (2.0%)	10 (2.0%)
Average Number of Iterations	4.96	**4.18**
Average Iteration Time	**0.32 s**	0.34 s
Average Iterations / Runtime	**20.96**	14.03
Average Distance to Target at Ground	**0.42 m**	0.47 m

The similar performance between the full MSSM and a 1st order SSM ultimately stems from the quality of the initial guess. The same initial guess was chosen for all the comparisons presented above (i.e. MSSM sample i and SSM sample i use the same initial guess, …, MSSM sample n and SSM sample n use the same initial guess). However, a "good" guess is generated for every sample by coarsely stepping the ground impact position backwards through time to the start altitude using the wind x and y components. Without this good initial guess, the MSSM would perform more favorably than any SSM.

Uncertainty Propagation

The joint probability density at ground needs to be evolved through the nonlinear dynamics of the parachute-package system to the maximum transition altitude. To accomplish this, the SLE is integrated backwards in time for each sample along its trajectory. The independent variable of the SLE is changed from time t to altitude z since different samples have different descent rates. This results in the following equation for the time-varying joint PDF,

$$\varphi\big(\vec{x}(z)\big) = \varphi_0 \exp\left(- \int_0^z \frac{\mathrm{Tr}\left(J_{\dot{\vec{x}}}(\xi)\right)}{\dot{x}_3} d\xi \right) \tag{34}$$

where $J_{\dot{\vec{x}}}$ is the Jacobian of $\dot{\vec{x}}$ from Eq. (17) and $\mathrm{Tr}(*)$ is the trace. The full derivation of Eq. (34) is beyond the scope of this thesis but is presented in detail in [23]. Eq. (34) is then able to be solved along the trajectory of every sample since the trajectories are now known from the BVP solution, thus producing the joint PDF as a function of altitude along each sample's trajectory.

This joint PDF represents the evolution of the final density φ_0 from ground to $z_{t,max}$ through the uncertain dynamic system and assuming only main parachute dynamics. Consider the joint PDF at some altitude $\bar{z} \in [z_{t,max}, 0]$. If packages were deployed at altitude $\bar{z}$, and their initial positions and uncertain parameters were randomly sampled according to the joint PDF at $\bar{z}$, then the resulting ground impact distribution would match exactly the desired impact distribution. This assumes the packages are already in steady-state under the main parachute and infinite samples are being used. In reality, the final impact distribution will never exactly match the desired impact distribution due to discretization errors, errors in uncertainty distributions, and finite sampling.

The transition altitude optimization algorithm uses the x-y marginal probability density $\varphi_{xy}(x, y, z)$ rather than the joint PDF from Eq. (34). An example of the procedure is presented in Figure 3.7. The joint PDF exists in $(n_p + 2)$-dimensional space; thus, when viewed in the x-y plane, the joint PDF looks noisy. In the marginalization process, a grid of $p \times q$ total bins is layered over the joint PDF. To produce the marginal, each bin is computed as the average of the joint PDF values in that bin and is normalized by the bin area and total probability across all bins. The procedure is repeated at discrete altitude steps to produce marginal x-y "slices" φ_{xy}^i. The accuracy of φ_{xy} depends on the number of samples N, the number of uncertain parameters n_p, and the number of bins p and q; for example, lower values of N lead to higher noise in the marginal. A grid size of 60×60 bins is assumed for the remainder of the thesis.

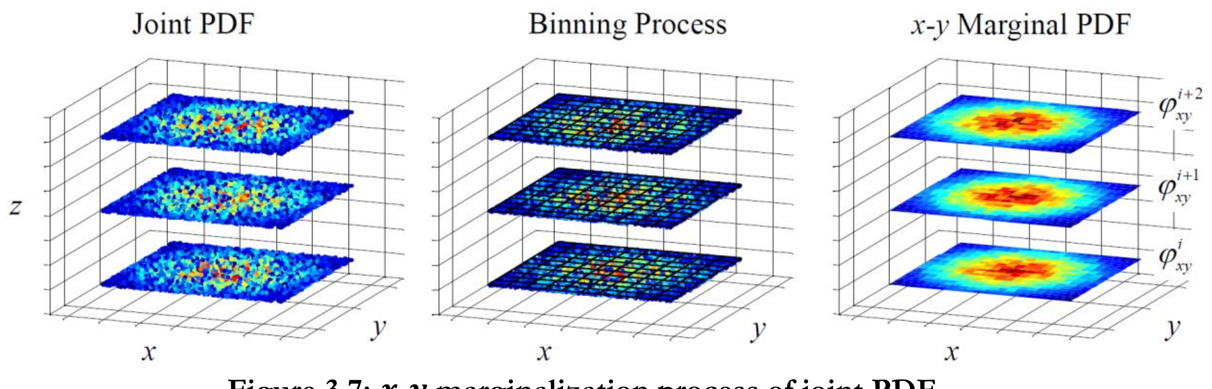


Figure 3.7: x-y marginalization process of joint PDF

It is worth noting that the PDF values computed with Eq. (34) can become quite small over a small altitude range, leading to numerical ill-conditioning of the joint PDF. To resolve this problem, the dynamics of Eq. (12) are non-dimensionalized by an arbitrary acceleration value γ. This factor decreases the magnitude of the Jacobian in Eq. (34), improving the computational performance of the algorithm. A value of γ = 50 m/s is used for the remainder of the thesis.

Transition Altitude Optimization

Every step prior to this section takes place offline in the mission planning process, i.e. prior to package deployment. The marginal x-y PDF slices φ_{xy}^{i} are stored in a computer onboard the package which computes the real-time optimized transition altitude. The transition altitude optimization algorithm is described below.

While under the drogue descent, the package's current state is measured. The trajectory is propagated from the measured state to an altitude of $z_{t,min}$ under drogue parachute dynamics only using an analytical descent solver that predicts the drogue trajectory using a radial basis function approximation [26]. This analytical trajectory propagator is beneficial because the solution time is deterministic and many orders of magnitude faster than numerical integration; this makes the propagator practical for a real-time implementation on low-cost embedded hardware. The analytical solution in [26] assumes that winds do not vary spatially in x or y. However, this assumption results in minimal error because the package vertical velocity is much greater than its translation velocity under drogue descent.

The predicted package trajectory computed by the analytical solution is defined as $x_p(z) = f_x(z)$ and $y_p(z) = f_y(z)$. The marginal probability density is determined along this trajectory, resulting in a function $\varphi_d(z)$ that measures the PDF values that this trajectory is predicted to intersect during drogue descent. This process is shown in Figure 3.8, where the predicted drogue trajectory is shown in black. The predicted trajectory passes through regions of relatively low probability on slices φ_{xy}^{i}, φ_{xy}^{i+1}, and φ_{xy}^{i+3}, whereas it reaches a high probability region on slice φ_{xy}^{i+2}. The translation of the marginal slices is due to the effects of wind on the main parachute motion.

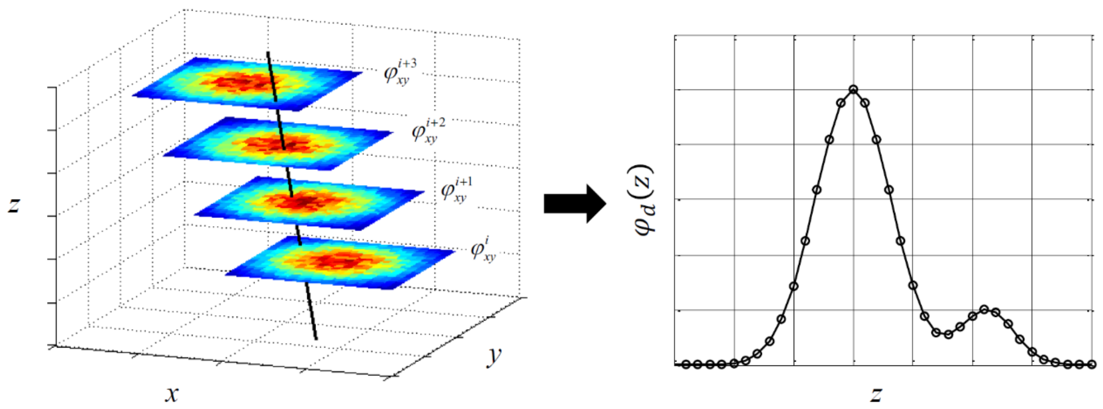


Figure 3.8: Methodology for constructing $\varphi_d(z)$ from drogue descent trajectory prediction

During drogue descent, $\varphi_d(z)$ is continuously recomputed based on the position and velocity feedback from the onboard sensors. There are many methods for choosing a transition altitude from $\varphi_d(z)$. Two methods are presented here.

The optimal transition altitude z_t^* may be selected as,

$$z_t^* = \underset{z \in \mathcal{Z}}{\arg\max}\, \varphi_d(z) \tag{35}$$

where $\mathcal{Z} = [z_{t,min}, z_{t,max}]$. Selecting z_t^* as the altitude that maximizes $\varphi_d(z)$ will push packages towards higher probability regions. Eq. (35) is most appropriate when the desired impact distribution φ_{PI} is specified as an inverse cost map since packages that can reach the highest probability regions will always do so and low probability regions will always be avoided if possible. Thus, the impact distribution will likely not match φ_{PI} if given as a desired impact distribution.

In cases where it is desirable to match the impact distribution to the desired impact distribution, z_t^* may be considered to be a random variable with distribution given by $\varphi_d(z)$ according to,

$$z_t^* \in \varphi_d(z) \tag{36}$$

where z_t^* is generated by randomly sampling from $\varphi_d(z)$ as each new prediction for $\varphi_d(z)$ is obtained. Thus, packages will probabilistically achieve the desired impact distribution given a large enough number of trials. Applications of these two methods are discussed in detail in Chapter 5.

In practice, the selection process for choosing z_t^* using either method is performed every time $\varphi_d(z)$ is recomputed when new state measurements are received. When the measured altitude crosses the currently selected z_t^*, the main parachute deploys and inflates. This constant recalculation process allows the optimal transition altitude to be updated based on unexpected wind and dynamic model errors that can cause the actual trajectory to drift from the predicted trajectory.

It is important to note a few considerations in this scheme. First, the transition altitude optimization only acts to control a package along its "line of control" along the ground. This line of control, determined by the wind field, is fixed at the time of package deployment based on the initial location. Therefore, it is not possible to control the accuracy in the crosswind direction by changing the transition altitude alone. This limitation restricts the ability of this process to realize an arbitrary desired impact distribution. However, results shown in Chapter 5 demonstrate how transition altitude optimization can be used to shape the dispersion pattern in interesting ways despite being unable to reduce dispersion in the crosswind direction.

Another consideration is the BVP solutions assume that the parachute-package system is in a quasi-steady-state descent under the main parachute throughout the entire BVP trajectory. When deploying the main parachute at z_t^*, it still takes several meters of altitude (on the order of 100+ m for the cases presented here) to reach steady-state descent. An inflation distance z_{inf} is added to z_t^* such that when the measured z altitude crosses over $z_t^* + z_{inf}$, the main parachute is deployed, and the system achieves steady-state descent by the time is reaches z_t^*. A value of $z_{inf} = 130$m was determined via simulation that provided an accurate approximation of the inflation distance for the example cases below.

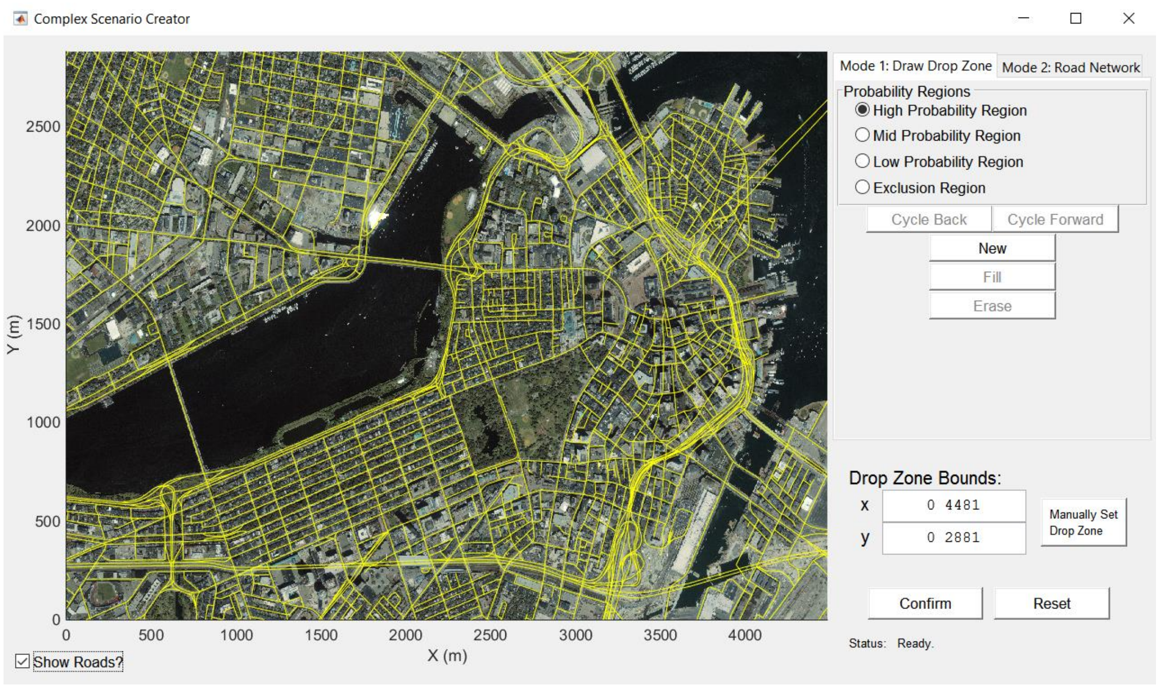
CHAPTER 4

CUSTOM SCENARIO CREATOR

The potential use of the proposed algorithm in complex real-world scenarios is of primary concern. Examples include impacting at the shortest geodesic location to a desired point and avoiding undesirable impact areas. For example, perhaps it is desirable to land cargo as close to the entrance of a military forward operating base (FOB) as possible but avoid landing inside the actual base. Perhaps instead it is desirable to land cargo in any of the surrounding area around a FOB, but the packages should land as close to the existing roads as possible to make the package retrieval easier. Further still, perhaps it is desirable to land as many packages as possible in an open park, but it is not important where inside the park they land. Reliably and repeatedly creating these complex scenarios is a non-trivial task. To solve this problem, the Custom Scenario Creator (CSC) graphical user interface was created, allowing a user to easily generate a complex desired impact distribution.

GUI Operation

The CSC is a MATLAB program that utilizes the Mapping Toolbox and Image Processing Toolbox to allow a user to create a desired impact distribution based on a real-world map, as shown in Figure 4.1. In the examples shown, map and road data from Boston, Massachusetts is used because it is provided as example data in the Mapping Toolbox. Additional map and road data can be easily gathered from various sources online; the interested reader is directed to [27]. Two modes of operation currently exist in the CSC: "Draw Drop Zone" mode and "Road Network" mode.



Drop Zone Bounds: x 0 4481; y 0 2881

Status: Ready.

Figure 4.1: Custom Scenario Creator

Draw Drop Zone Mode

In this mode, the user can manually draw any number of regions of high, medium, and low probability, along with exclusion regions, as seen in Figure 4.2. Using the Boston map as an example, the user can mask out desired regions of interest, as shown in Figure 4.3. The color scheme is as follows: red regions are regions of highest probability (set to 99%), yellow regions have medium probability (set to 66%), green regions have low probability (set to 33%), and black regions are exclusion regions that have no probability (set to 0%). These probability levels can be easily tailored or generalized as needed.

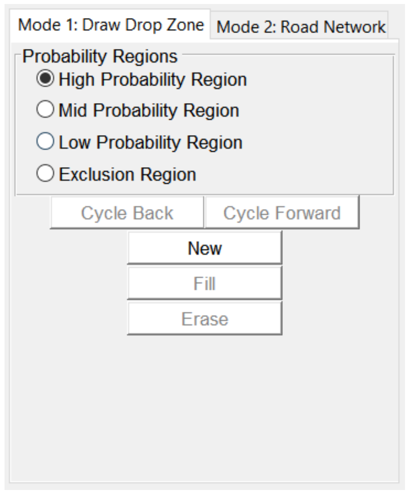


Figure 4.2: Draw Drop Zone mode tools

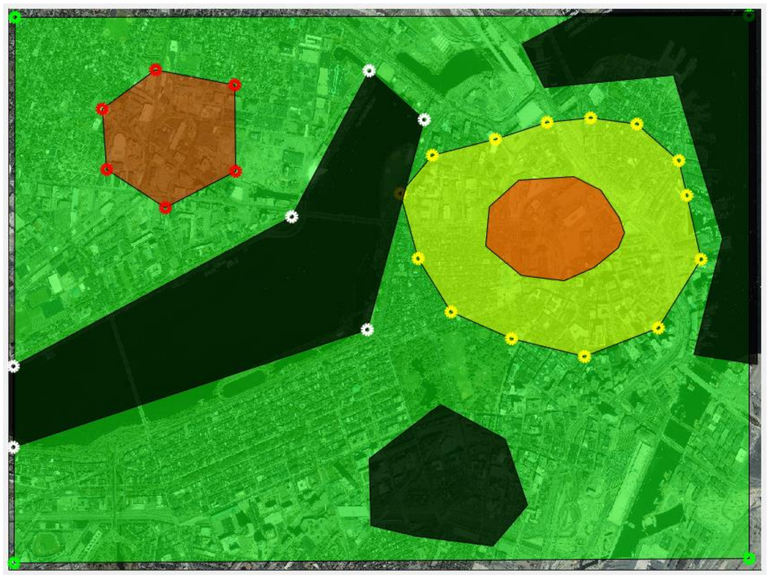


Figure 4.3: Custom scenario created with Draw Drop Zone mode

In the example, a low probability across the entirety of the map is set, and the desired exclusion areas are masked out, i.e. the packages should be able to land anywhere in the map except for the exclusion regions. Additionally, two regions of high probability are added where the packages should ideally impact, if possible given the atmospheric conditions. Finally, a region of medium probability is added around one of the high probability regions to further push the package toward the desired landing area.

Once the user is satisfied with the scenario, a desired impact distribution is generated, as seen in Figure 4.4. This impact distribution is then able to be directly used in the existing mission planner for uncertainty propagation. It is noted that no efforts were taken to smooth the transitions between probability regions. Different interpolation approaches can be investigated to produce a smoother impact distribution, but the implementation of that feature is left for future work.

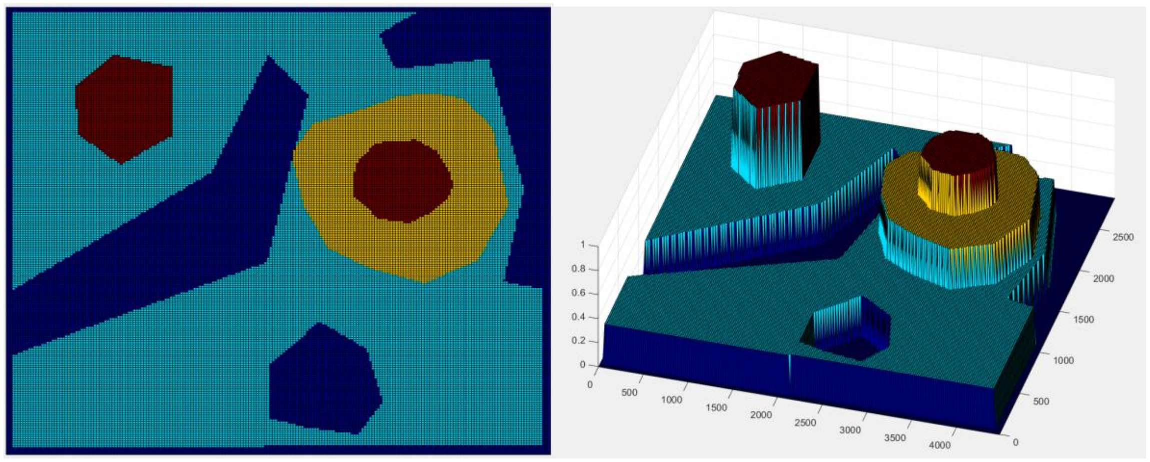
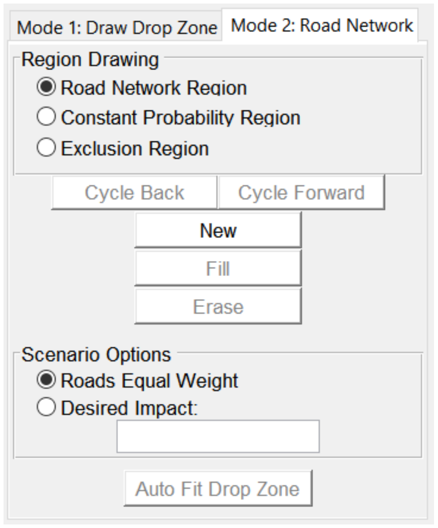


Figure 4.4: Desired impact distribution created from the CSC showing the probability and exclusion regions

Road Network Mode

In this mode, the user can manually draw a road network region of interest and any number of constant probability and exclusion regions. Additionally, the user can choose between having all roads weighted equally and specifying a desired impact point, as seen in Figure 4.5. By choosing all roads weighed equally, a desired impact distribution is produced where all the roads in a given region will have the same probability of impact. Conversely, by choosing a desired impact point, a desired impact distribution is produced where the probability of each road is a function of the geodesic distance from a given point on the road to the desired impact location. Both options will be explored further below.



Region Drawing
● Road Network Region
○ Constant Probability Region
○ Exclusion Region

Scenario Options
● Roads Equal Weight
○ Desired Impact:

Figure 4.5: Road Network mode tools

The user first creates a road network region. The road network region is an area of interest in the map that is turned into a desired impact distribution. Figure 4.6 shows this area in green around a park. The road network region is then zoomed in to maximize its area while maintaining the aspect ratio of the map, as seen in Figure 4.7

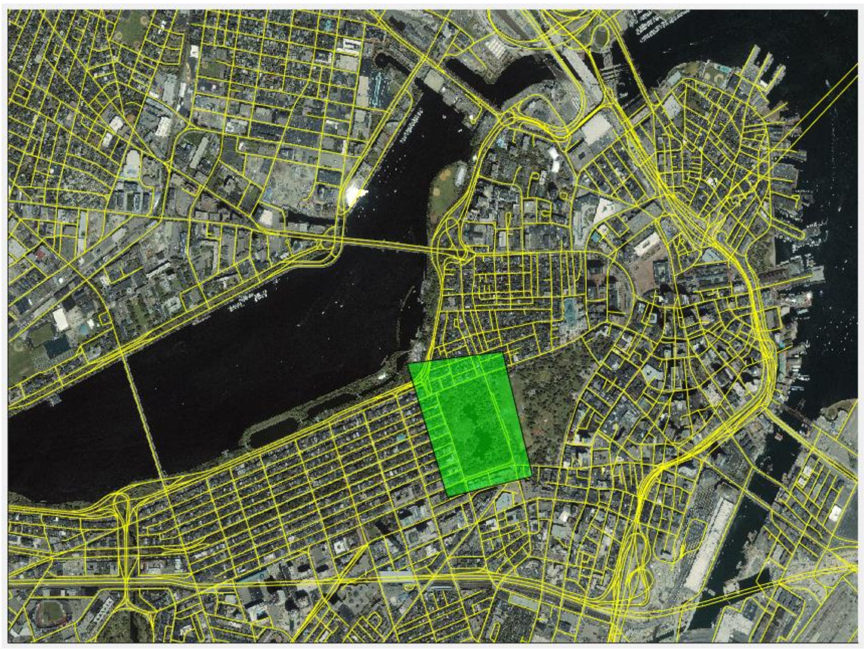


Figure 4.6: Road network region of interest drawn on the map

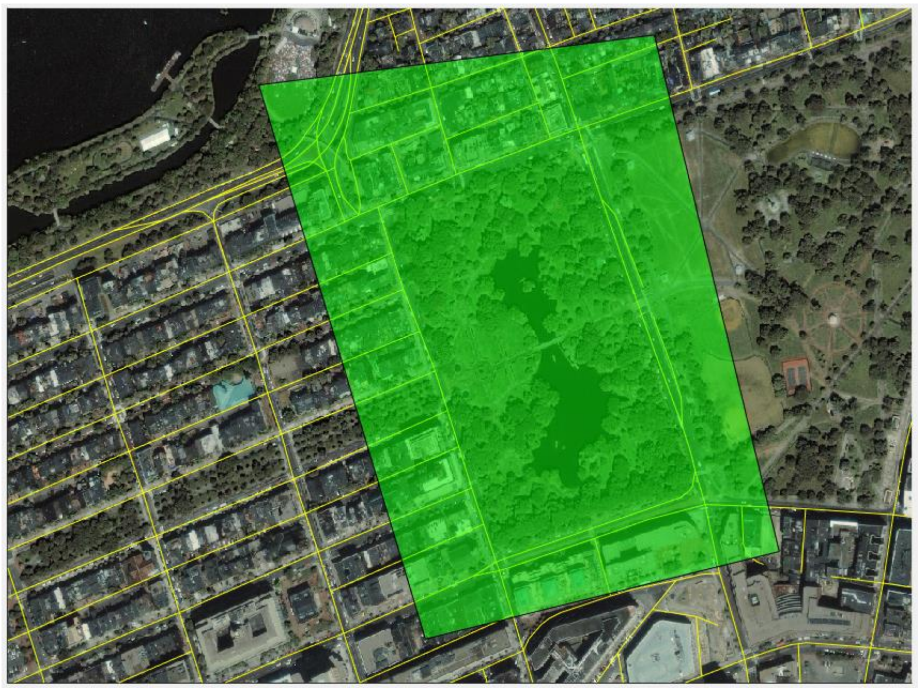


Figure 4.7: Road network region of interest drawn on the map

A region of constant probability is added in the middle of the park, denoted in red, as well as an exclusion region, denoted in black, as shown in Figure 4.8. This region of constant probability is a region where the desired probability is 100%, i.e. as many packages as possible should ideally land in that area subject to the atmospheric conditions. Conversely, the exclusion region is a region where the desired probability is 0%, i.e. no packages should land in that area. Finally, a desired impact point is added to the road near the northwest region of the park denoted as point A in Figure 4.8. This desired impact point will produce a geodesic distance computation for every road, i.e. the farther away from the desired impact point, the lower the probability (according to a geodesic distance metric).

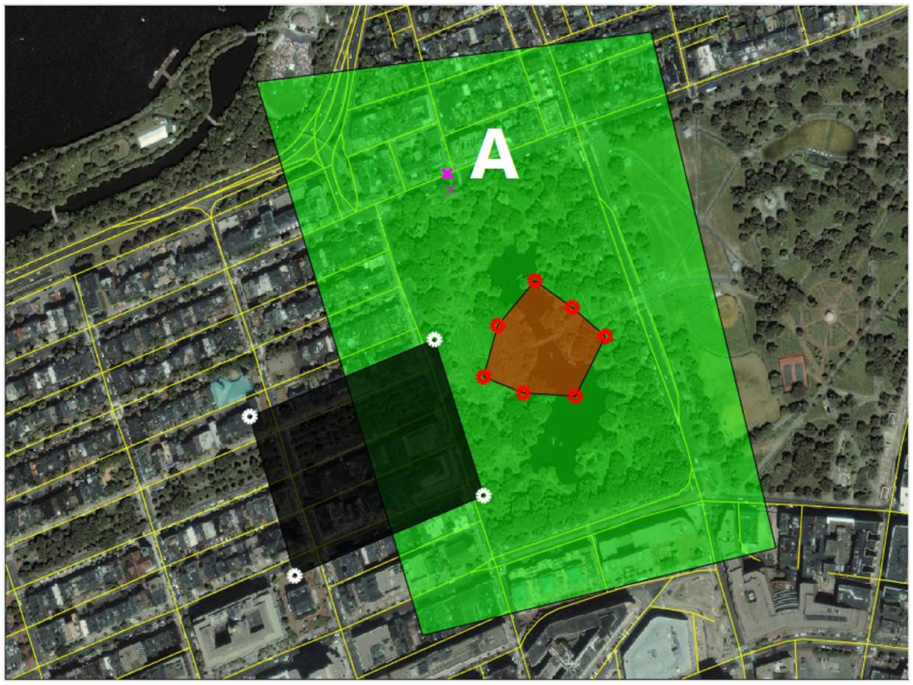


Figure 4.8: Road network region (green) with constant probability region (red), exclusion region (black), and desired impact point (pink) at A

Overall, the example scenario is designed to proceed as follows. As many packages as possible should land in the middle of the park. If the packages cannot reach the middle of the park, they should land as close to any road as possible. However, it is also desirable to land as close as possible to point A. Thus, the ultimate goal for a package is to land as close as possible to any road but also as close to point A as possible. No packages should land in the exclusion region. This scenario is realized in Figure 4.9 where the region of constant probability, the exclusion region, and the probability decreasing out from point A can easily be seen.

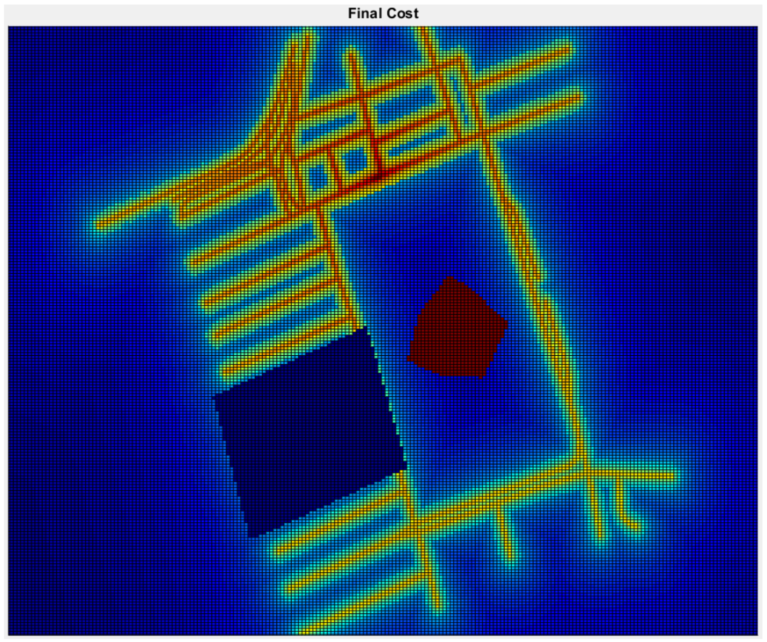


Figure 4.9: Desired impact distribution created from the CSC using a desired impact point

If instead all roads are chosen to be weighed equally, the scenario would build similar to before. However, now the probability no longer diminishes from point A, as seen in Figure 4.10.

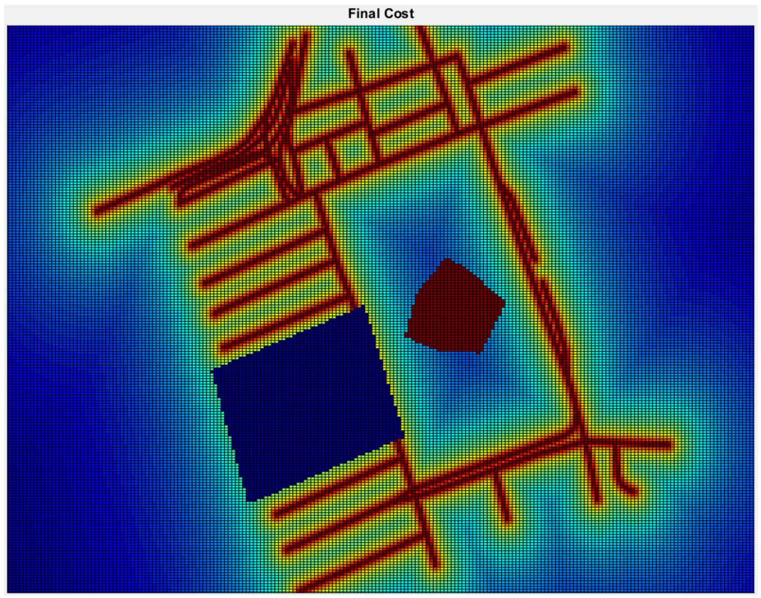


Figure 4.10: Desired impact distribution created from the CSC using all roads weighed equally

Scenario Computation

The procedure for computing the complex scenarios is now examined. A road network similar to Figure 4.8 is chosen as the starting point. This scenario contains every currently available option: multiple roads, exclusion region, constant probability region, and a desired impact point. Note that the drop zone spans from approximately 1900 m to 2900 m in the x-direction and approximately 700 m to 1300 m in the y-direction, as seen in Figure 4.11.

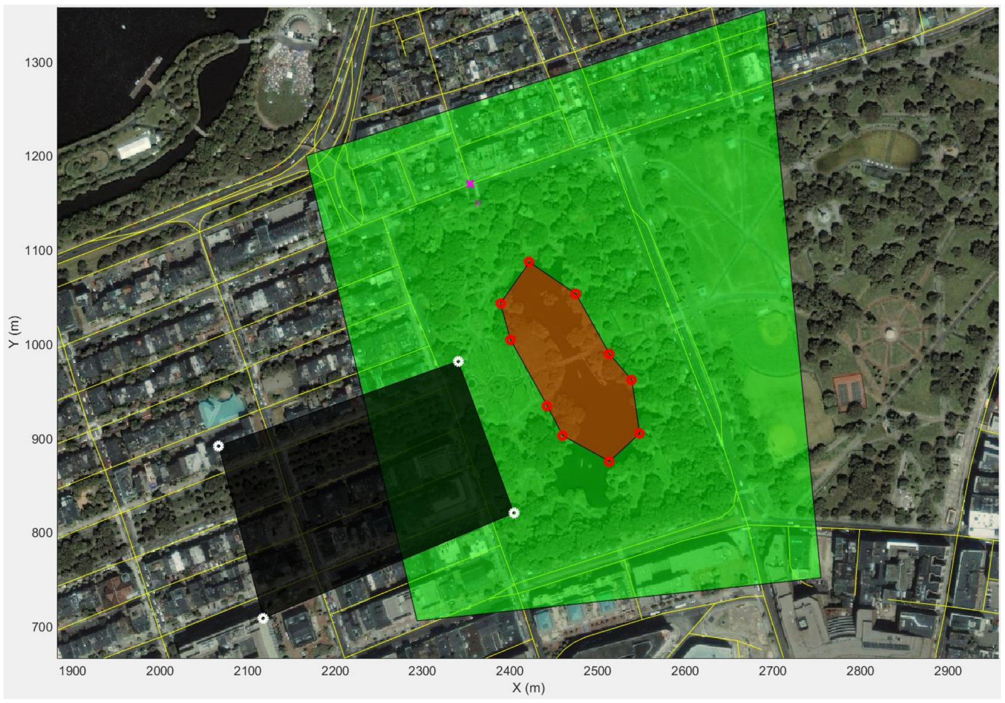


Figure 4.11: New road network region (green) with constant probability region (red), exclusion region (black), and desired impact point (pink)

The drop zone is first discretized into a 2D grid. For the examples in this thesis, a 250 × 250 grid is used because of its fast computation time and visual clarity (i.e. larger grids were too dense to display). For each road in the scenario, the minimum distance from every grid point to the road curve is computed, producing a distance matrix $\boldsymbol{D_k}$, where $\mathbf{1} \leq \boldsymbol{k} \leq \boldsymbol{n}$ and $\boldsymbol{n}$ is the number of roads in the scenario. Thus, $\boldsymbol{n}$ 250 × 250 matrices are produced that contain this distance information. A minimum distance matrix $\boldsymbol{M}$ is then created by taking the minimum value from every distance matrix for each grid point,

$$M_{ij} = \min(D_{1,ij}, D_{2,ij}, \dots, D_{n,ij}) \tag{27}$$

This matrix is visualized in figure 4.12. From the minimum distance matrix, the road cost $\boldsymbol{R}$ is computed by inverting $\boldsymbol{M}$,

$$R = \frac{1}{M^{0.1}} \tag{27}$$

The minimum distance matrix is taken to the power 0.1 because it provides a gentler gradient of the resulting data. The road cost matrix is then saturated at a value of 0.95 to eliminate spikes in the data and is normalized, as seen in Figure 4.13 and Figure 4.14.

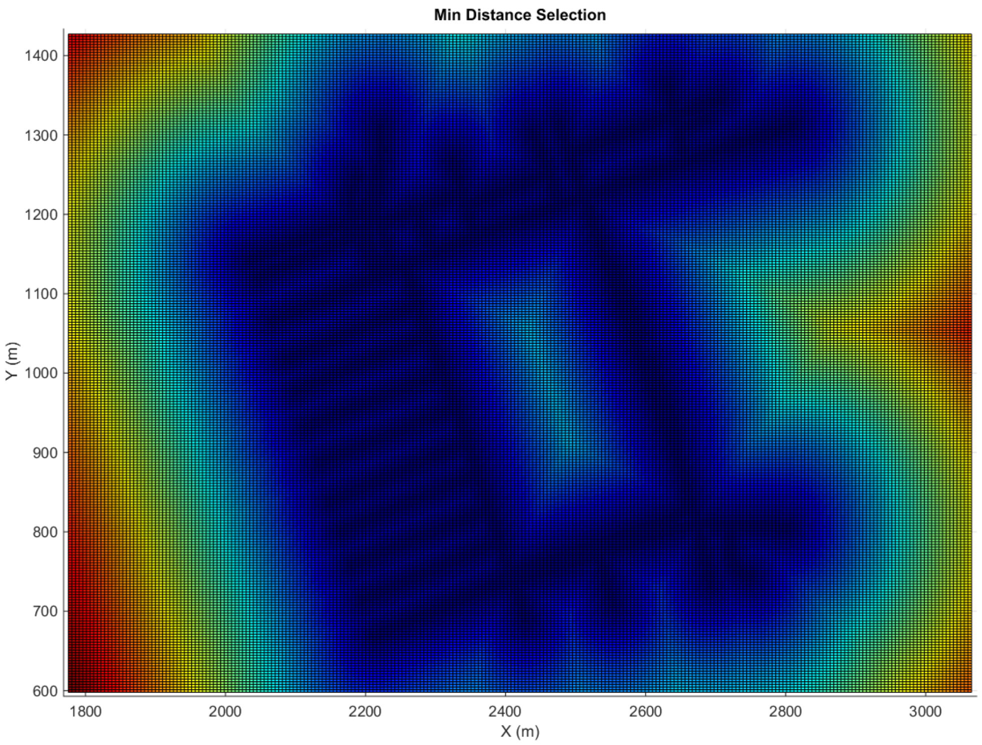


Figure 4.12: Minimum distance matrix

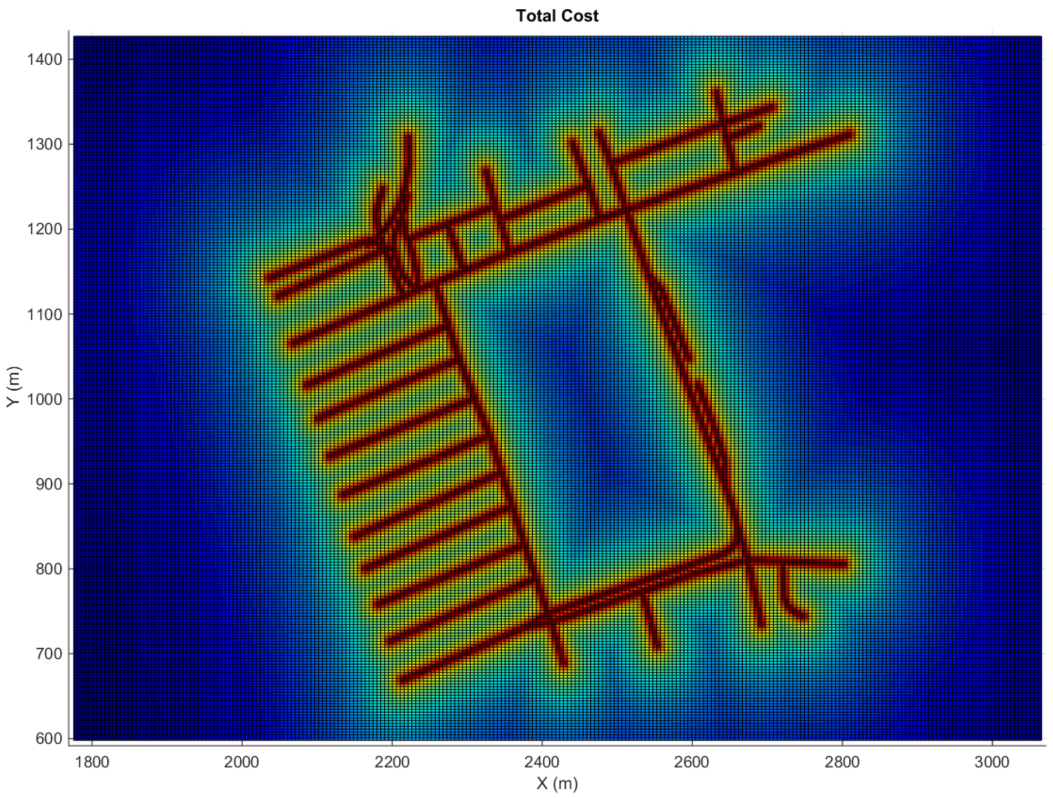


Figure 4.13: Road cost matrix

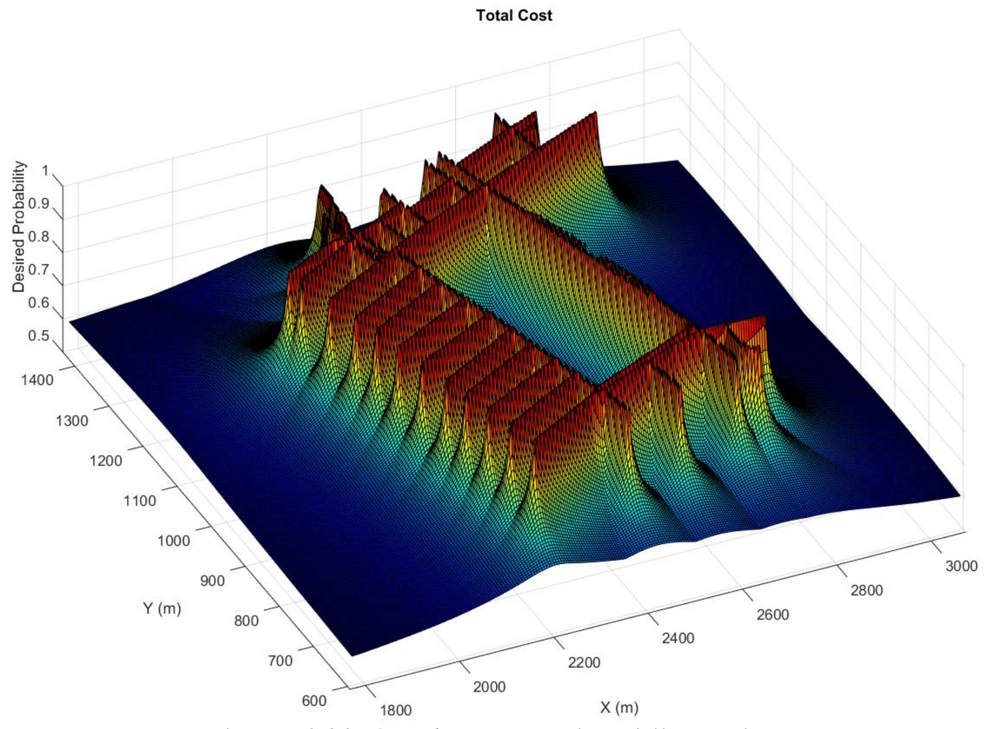


Figure 4.14: Road cost matrix, oblique view

The geodesic cost $\boldsymbol{G}$ is next computed by utilizing MATLAB's Image Processing Toolbox. For every road in the scenario, the road curve is discretized into a large binary matrix that maintains the aspect ratio of the scenario. Indices in the binary matrix where a road appears are valued as 1, and everything else is valued at 0. The binary matrix is then transformed into a logical matrix so that image processing operations can be performed. The discretized road curve data is quite sparse in the binary matrix, as seen in Figure 4.15a. Note that the procedure now switches to "image space" where every pixel is an index of the logical matrix. The MATLAB `bwmorph(*,'dilate')` operation is then used to widen the line and connect some adjacent pixels, as seen in Figure 4.15b. However, the line still has discontinuities and is too small to perform significant operations. The `bwmorph(*,'thicken',*)` operation is used to significantly widen the data line, as seen in Figure 4.15c. To remove the remaining discontinuities, a final `bwmorph(*,'dilate')` operation is performed to obtain a smooth and wide road curve in the logical matrix, as seen in Figure 4.15d. Remember that this procedure is done for every road. Every logical matrix is summed together to produce a final logical matrix $\boldsymbol{R_{network}}$ containing the entire road network, as seen in Figure 4.16.

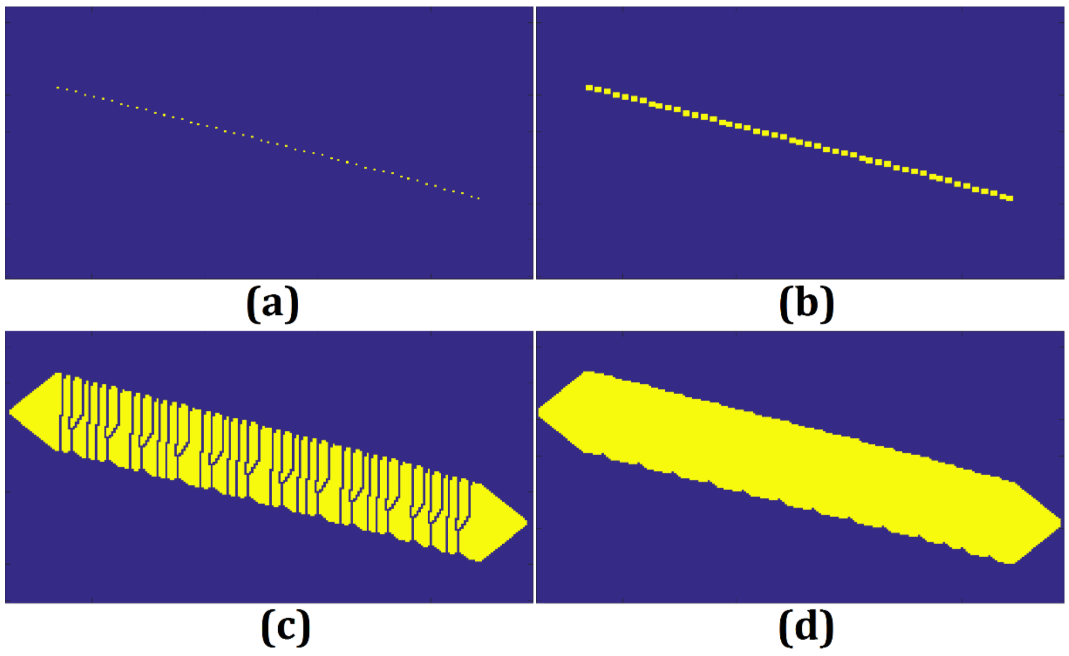


Figure 4.15: Modification of road data in image space – (a) raw road data discretized to image space, (b) dilated line to connect adjacent pixels, (c) widened line, (d) dilated line again to fully connect data

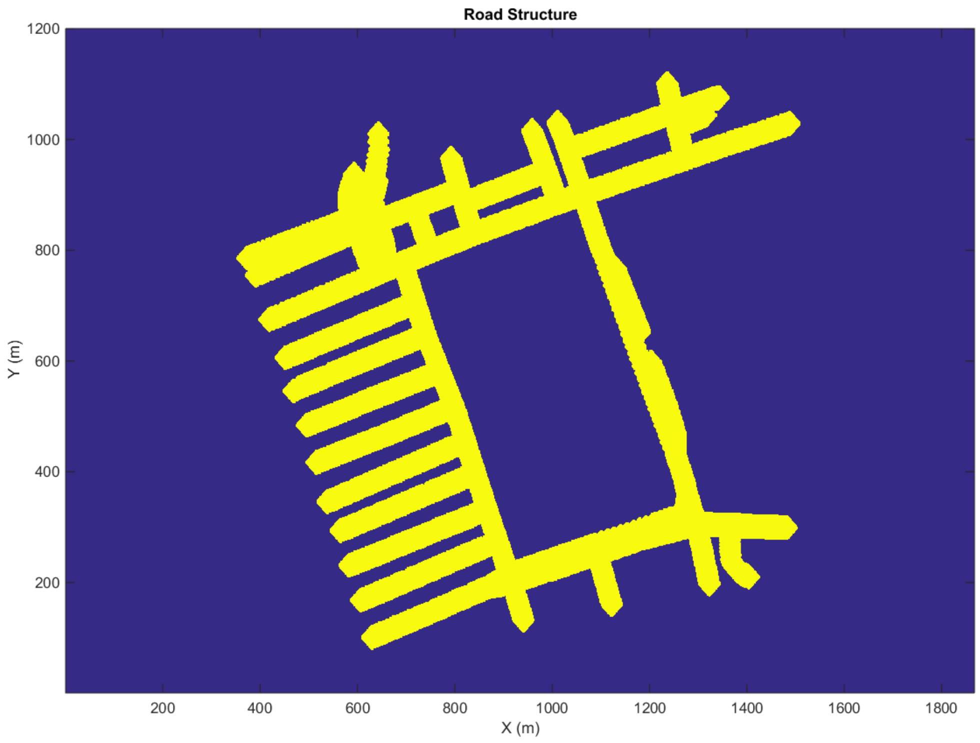


Figure 4.16: Total road network (Note the units on the axes are pixels since this is an image)

The binary image of the road network is used to compute the geodesic distance transform of the impact point using the operation `bwdistgeodesic(`$\boldsymbol{R_{network}}$`, impactX, impactY, 'quasi-euclidean')`, where `impactX` and `impactY` are the $\boldsymbol{x}$ and $\boldsymbol{y}$-coordinate of the desired impact point, respectively. The geodesic distance can be thought of as the shortest distance along a given path, e.g. the distance from the desired impact point to any arbitrary point in the road network along valid paths of the road network. This is displayed in Figure 4.17, where the white area of the image is considered a valid path and the black area is an invalid path. Valid and invalid paths of the geodesic computation are specified by $\boldsymbol{R_{network}}$ where values of 1 and 0 represent valid and invalid paths, respectively. The newly computed geodesic distance matrix $\boldsymbol{G_{dist}}$ is then inverted to produce the geodesic cost $\boldsymbol{G}$,

$$G = \frac{1}{{G_{dist}}^{0.1}} \tag{27}$$

This is the same inversion used for the road cost calculation. Values of NaN are set to 0, values of Inf are set to 1, and the geodesic cost is normalized over the road network structure to produce a simple distribution between 0 and 1. Note the geodesic cost is still in an image state at this point around 1800 x 1200 pixels in size, which is much larger than the road cost matrix. This image is then discretized to a matrix of the same dimension as the road cost to produce the final geodesic cost, as seen in Figure 4.18.

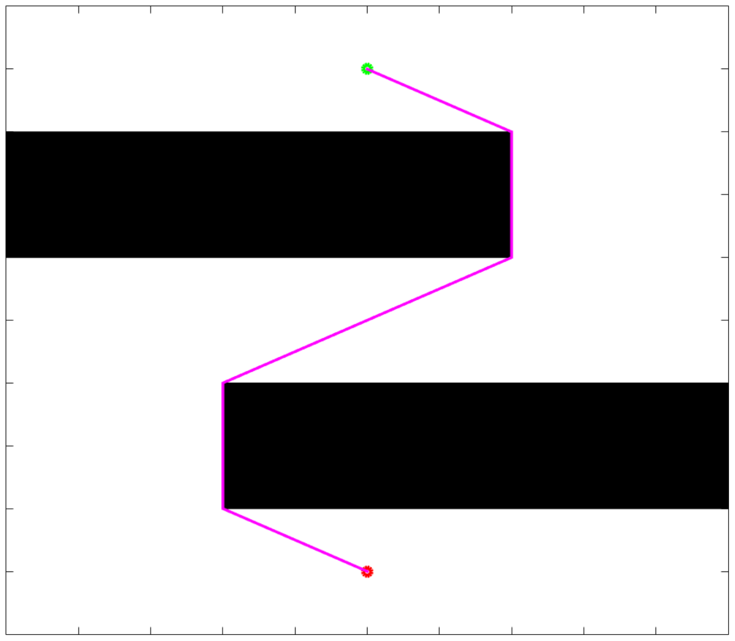


Figure 4.17: Geodesic distance example, white areas are valid travel regions, black areas are obstacles

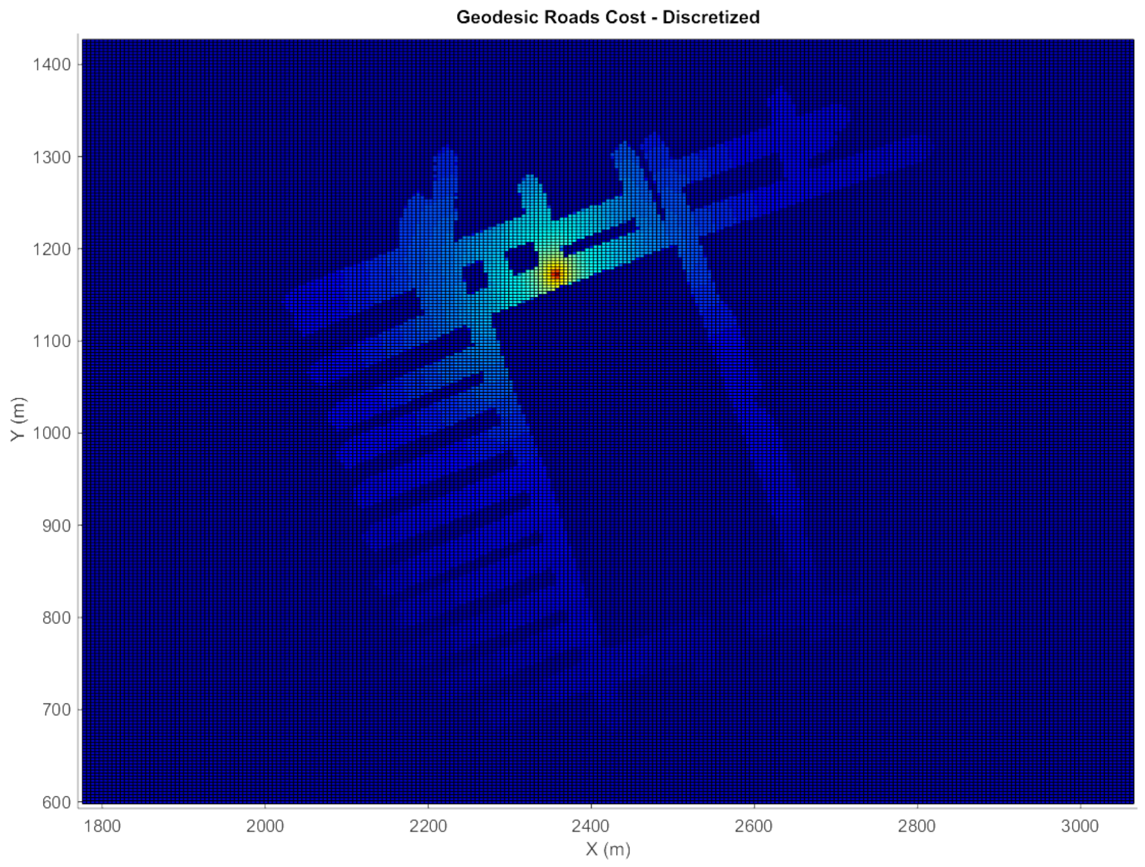


Figure 4.18: Geodesic cost

The road cost $\boldsymbol{R}$ and geodesic cost $\boldsymbol{G}$ are then combined by a generic addition to produce the final cost $\boldsymbol{J}$,

$$J = \alpha R^{\gamma} + \beta G^{\delta} \tag{27}$$

where $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, $\boldsymbol{\gamma}$, and $\boldsymbol{\delta}$ are scalar parameters that are manually tuned to produce the desired impact distribution. The effects of changes to these parameters are shown in Figure 4.19 for a more primitive scenario example that contains four roads and one exclusion region.

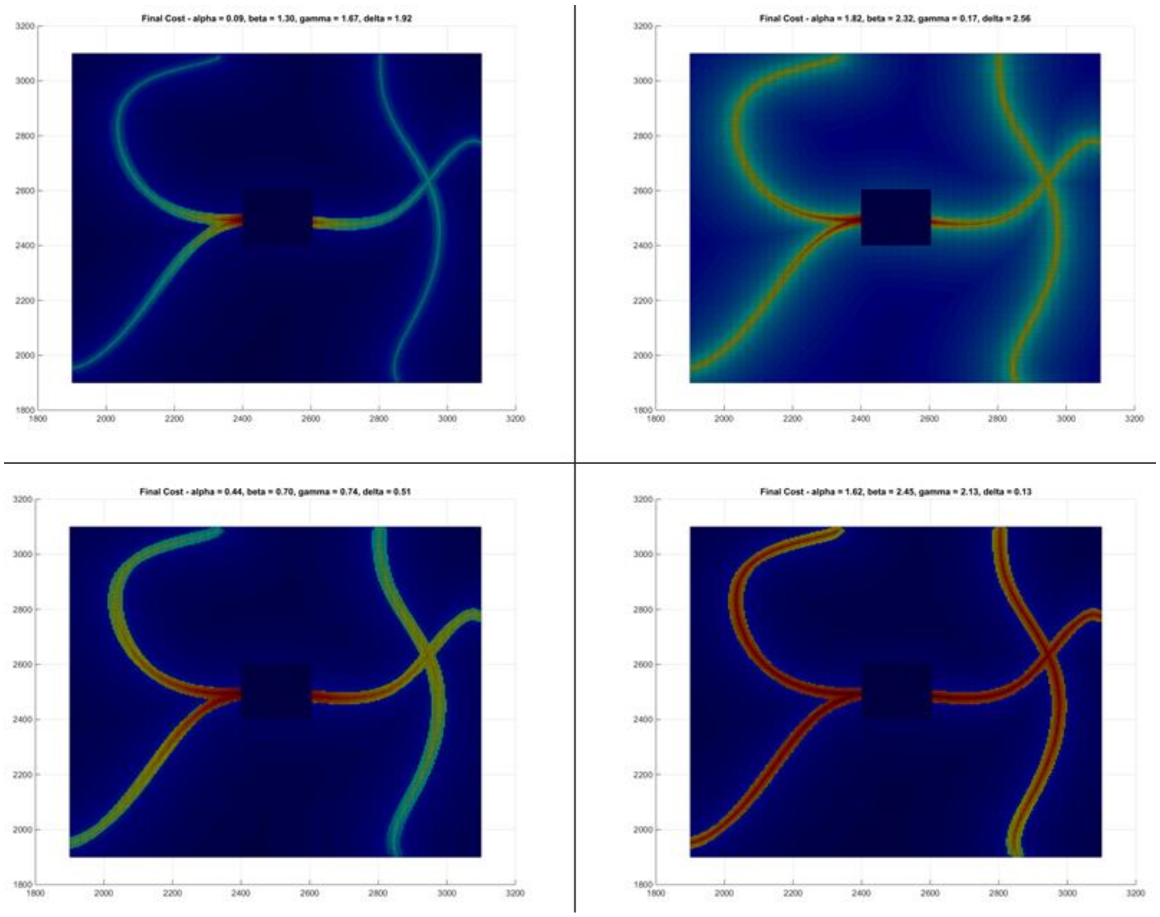


Figure 4.19: Comparisons of different values of $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, $\boldsymbol{\gamma}$, and $\boldsymbol{\delta}$ for the same road network

While many sets of parameters could produce valid probability distributions, it is desirable to produce distributions that show a clean ramp down in probability along the roads originating from the desired impact point, as well as a further reduction in probability emanating from the roads themselves.

Finally, "matrix masks" of the exclusion and constant probability regions are created to apply to the final cost. These masks essentially act as filters, setting certain indices of the final cost matrix to specific values and letting all the other indices remain unchanged. For example, the exclusion mask combines all exclusion regions into a discretized matrix; any index in the exclusion mask that falls within an exclusion region forces that same index in the final cost matrix to have a value of 0. Thus, the final cost calculation becomes,

$$J_{ij} = \begin{cases} 0, & if\ E_{ij} = 1 \\ 1, & if\ C_{ij} = 1 \\ \alpha {R_{ij}}^{\gamma} + \beta {G_{ij}}^{\delta}, & otherwise \end{cases} \tag{27}$$

where $\boldsymbol{E}$ is the exclusion region mask and $\boldsymbol{C}$ is the constant probability mask. The final cost is then scaled between 0 and 1 to have all desired probabilities in the form of a percent. The final cost is shown in Figure 4.20. In Figure 4.21, an oblique view of the distribution is shown. Note the exclusion region reaches a floor of zero probability, while the constant probability reaches the ceiling of 1 across its whole area. Additionally, note the probability peaks at the desired impact point and slowly decays along all the roads paths.

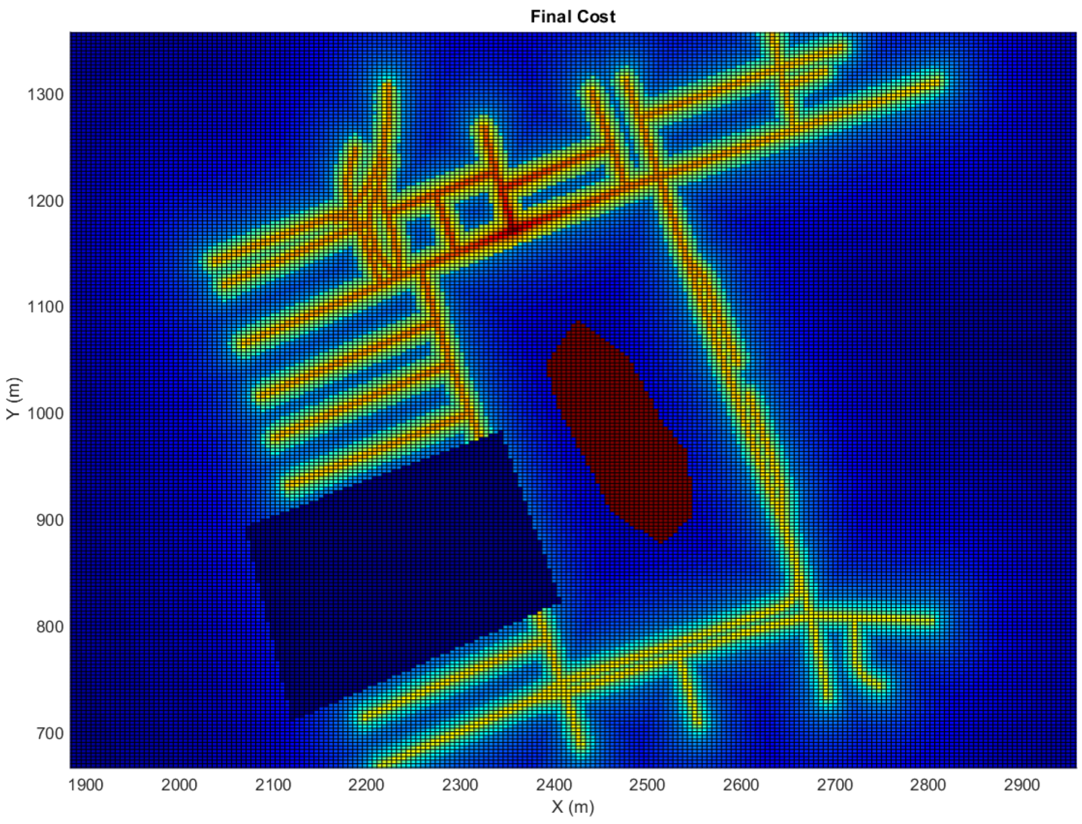


Figure 4.20: Final complex scenario

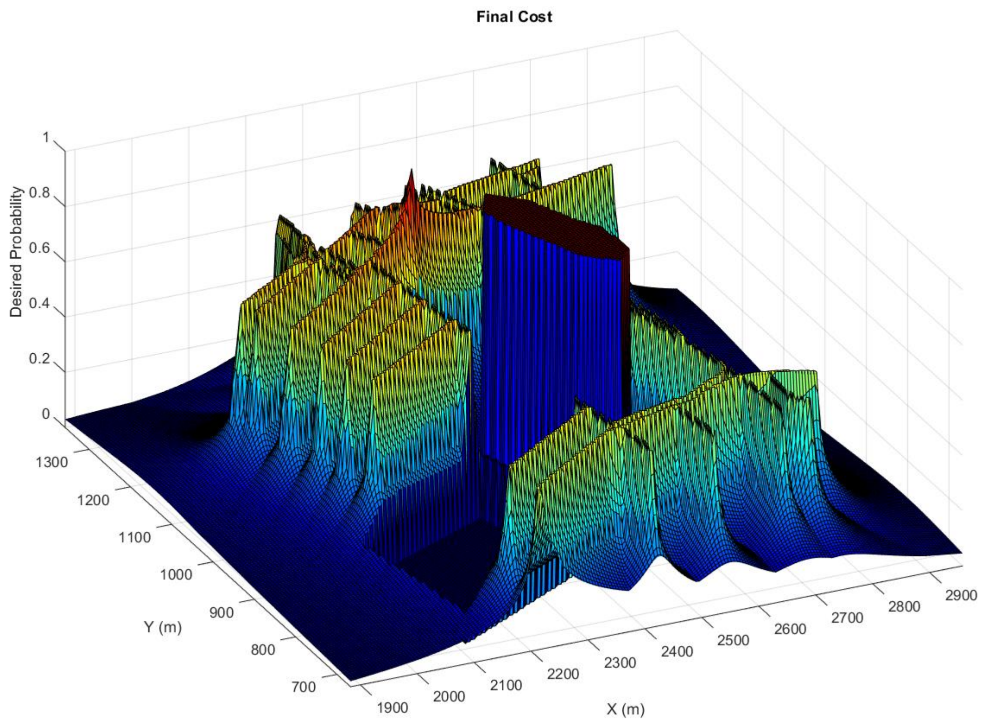


Figure 4.21: Final complex scenario, oblique view

The above process detailed the computation of a road network scenario with a desired impact point. If a road network scenario with all roads being equal weight is desired instead, none of the geodesic calculations have to be computed, and the total road cost of Figure 4.13 would serve instead as the output distribution.

Furthermore, if a drawn drop zone scenario is chosen instead of a road network scenario, a much simpler computation is undertaken. The concept of matrix masks presented above is utilized to compute the final cost,

$$J_{ij} = \begin{cases} 0, & if\ E_{ij} = 1 \\ 0.33, & if\ L_{ij} = 1 \\ 0.66, & if\ M_{ij} = 1 \\ 0.99, & if\ H_{ij} = 1 \end{cases} \tag{27}$$

where $\boldsymbol{H_{ij}}$ is the high probability region mask, $\boldsymbol{M_{ij}}$ is the medium probability mask, $\boldsymbol{L_{ij}}$ is the low probability mask, and $\boldsymbol{E_{ij}}$ is the exclusion region mask. Higher probability regions take precedent over lower probability regions, e.g. if a high probability region is on top of a middle probability region which is on top of a low probability region, the high probability value is chosen.

CHAPTER 5

RESULTS

In this section, a comprehensive set of simulation results is presented demonstrating performance of the proposed transition altitude optimization algorithm in a variety of scenarios. Simulation results are first shown for two example cases where the propagated probability density is sampled at altitude and the desired impact distribution is recovered at the ground, highlighting the physical meaning of the evolution of the desired impact distribution as it is back-propagated through the dynamic system. Next, several example cases are presented showing the ability of the transition altitude optimization algorithm to reduce dispersion, avoid obstacles, and shape the impact distribution in complex scenarios.

Three uncertain parameters ($n_p = 3$) are assumed throughout this section: main parachute drag coefficient, wind direction, and wind magnitude. A normal distribution is assumed for these parameters according to Eq. (22) – Eq. (26). The specific parachute-package system used in these examples is based on the G-12E with a total package mass of 1,048 kg. All inertial and aerodynamic properties used in the following cases are presented in Appendix B.

PDF Back-Propagation Examples

These examples demonstrate the physical meaning of the back-propagated probability distribution. By choosing random samples from the joint distribution at altitude and simulating those samples to ground impact under the main parachute dynamics, the desired impact distribution can be recovered. Note that the transition altitude optimization does not factor into these cases.

A desired ground impact distribution is first specified, as well as uncertainty distributions for $\boldsymbol{C_d}$, wind magnitude, and wind direction. The joint density at the ground is sampled as described in Chapter 2, Section "Joint Distribution Specification and

Discretization" and the parameter values listed in Appendix B. The joint density at ground is then back-propagated along BVP solution trajectories using the SLE to an altitude of $\boldsymbol{z} = \mathbf{700\ m}$, and the marginal distribution $\boldsymbol{\varphi_{xy}^{700}}$ is calculated. Next, new samples are generated by randomly sampling drop locations according to the $\boldsymbol{x}$-$\boldsymbol{y}$ marginal distribution $\boldsymbol{\varphi_{xy}^{700}}$ and the other uncertainty distributions for $\boldsymbol{C_d}$, wind magnitude, and wind direction. Rejection sampling [28] is used to draw samples from the marginal density in light of its arbitrary shape. The new samples are then integrated to ground impact to construct the marginal density at ground impact. The number of grid points used in these cases is $\boldsymbol{N} = 500{,}000$.

The correlation between the desired impact distribution and the achieved reconstruction is quite favorable. Case 1 is presented in Figure 5.1. A static wind field, constant with altitude, is used, where $\boldsymbol{w_x} = 0$ m/s and $\boldsymbol{w_y} = 2$ m/s except for a jet of $\boldsymbol{w_x} = 5$ m/s located between $\boldsymbol{y} = 700$m and $\boldsymbol{y} = 800$m. The desired distribution is chosen as a Gaussian centered at $\boldsymbol{x} = 1000$m, $\boldsymbol{y} = 700$m with a diagonal covariance matrix, and equal variances in $\boldsymbol{x}$ and $\boldsymbol{y}$ of 10^5 m^2. The top left of Figure 5.1 shows the marginal density at the ground computed from the joint distribution using the binning procedure. The top right of Figure 5.1 shows the back-propagated marginal density $\boldsymbol{\varphi_{xy}^{700}}$; the effect of the wind jet is clearly seen in the portion of the distribution smeared in the negative $\boldsymbol{x}$-direction. The bottom left of Figure 5.1 shows the histogram of 100,000 drop locations sampled from $\boldsymbol{\varphi_{xy}^{700}}$ using rejection sampling. Finally, the bottom right of Figure 5.1 shows the histogram of impact locations for the samples generated, which is an approximately Gaussian distribution with a nearly identical mean and covariance to the desired distribution.

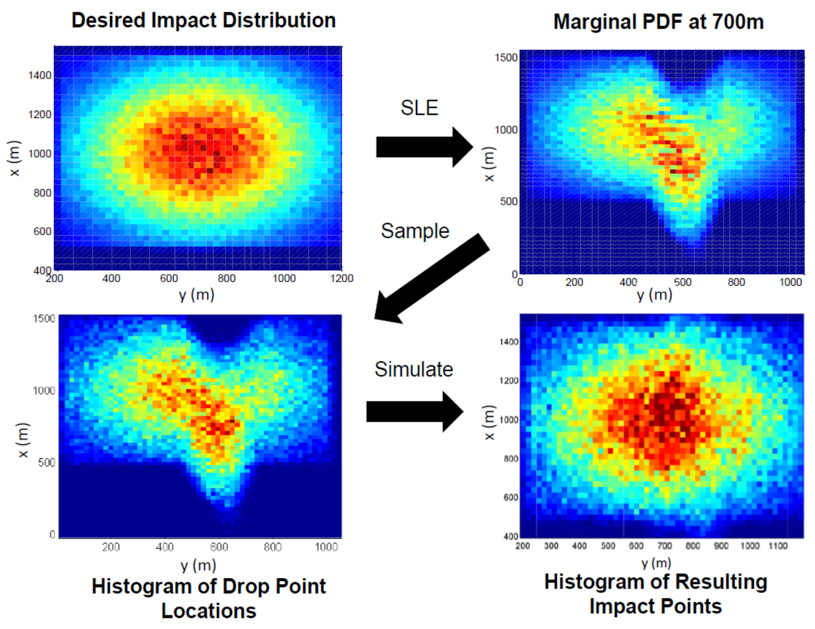


Figure 5.1: Case 1 – (Top Left) Desired marginal probability density at $z = 0$ m. (Top Right) Marginal probability density at $z = 700$ m. (Bottom Left) Histogram of sampled marginal probability density for validation simulations. (Bottom Right) Histogram of impact point locations from validation simulations

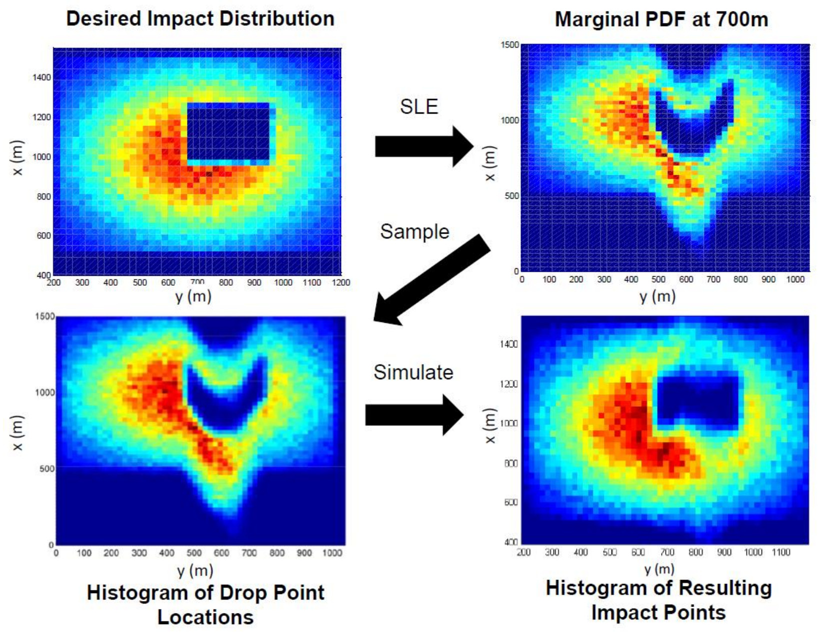


Figure 5.2: Case 2 – (Top Left) Desired marginal probability density at $z = 0$ m. (Top Right) Marginal probability density at $z = 700$ m. (Bottom Left) Histogram of sampled marginal probability density for validation simulations. (Bottom Right) Histogram of impact point locations from validation simulations

Case 2 is presented in Figure 5.2, where an exclusion zone is placed in the target region. Note the desired impact distribution $\boldsymbol{\varphi_{PI}(x,y)}$ is set to zero in this region. The same winds as Case 1 are used. The top right of Figure 5.2 is again the marginal density at 700m, sampled using rejection sampling and smeared from the wind jet. Finally, the bottom right of Figure 5.2 again shows the reconstruction of the original desired impact distribution, strongly resembling the overall Gaussian shape and avoidance of the exclusion region. The fuzzy boundary of the exclusion region in the top left of Figure 5.2 is due to the discretization during the binning process; discontinuous drops in probability across the exclusion region cannot be perfectly represented.

Dispersion Reduction

The transition altitude optimization algorithm's ability to reduce dispersion is examined in this section. Additionally, the effects of various algorithm parameters on dispersion reduction are explored. In the follow examples, a Gaussian centered at $x = 0$m, $y = 1300$m with diagonal covariance matrix and equal variances of 10^5 m^2 in the x and y directions is used as the desired impact distribution. This is meant to be interpreted as an inverse cost map, therefore Eq. (35) is used for selection of z_t^*. Winds are assumed to be constant in the y direction at 7 m/s and are uniform with respect to altitude. Again, all inertial and aerodynamic parameters, as well as uncertainty values, are given in Appendix B.

A complete HALO airdrop is simulated for these examples. The drop altitude is set to 4000m. The simulation executes the transition altitude optimization algorithm under the drogue descent and deploys the main parachute once passing the selected z_t^* and accounting for inflation distance. The main parachute dynamics are used for the remainder of the simulation until ground impact is recorded.

Figure 5.3 shows a plot of altitude vs downrange distance for two example trajectories dropped at initial locations $y = 150$m (Case 3) and $y = 350$m (Case 4). The simulations of these examples incorporate uncertainty in C_d and wind magnitude but not wind direction. Since there is no wind component in x, $x = 0$ for all time. It is seen that the transition altitude optimization sufficiently selects z_t^* such that the package impacts very near the center of the Gaussian distribution.

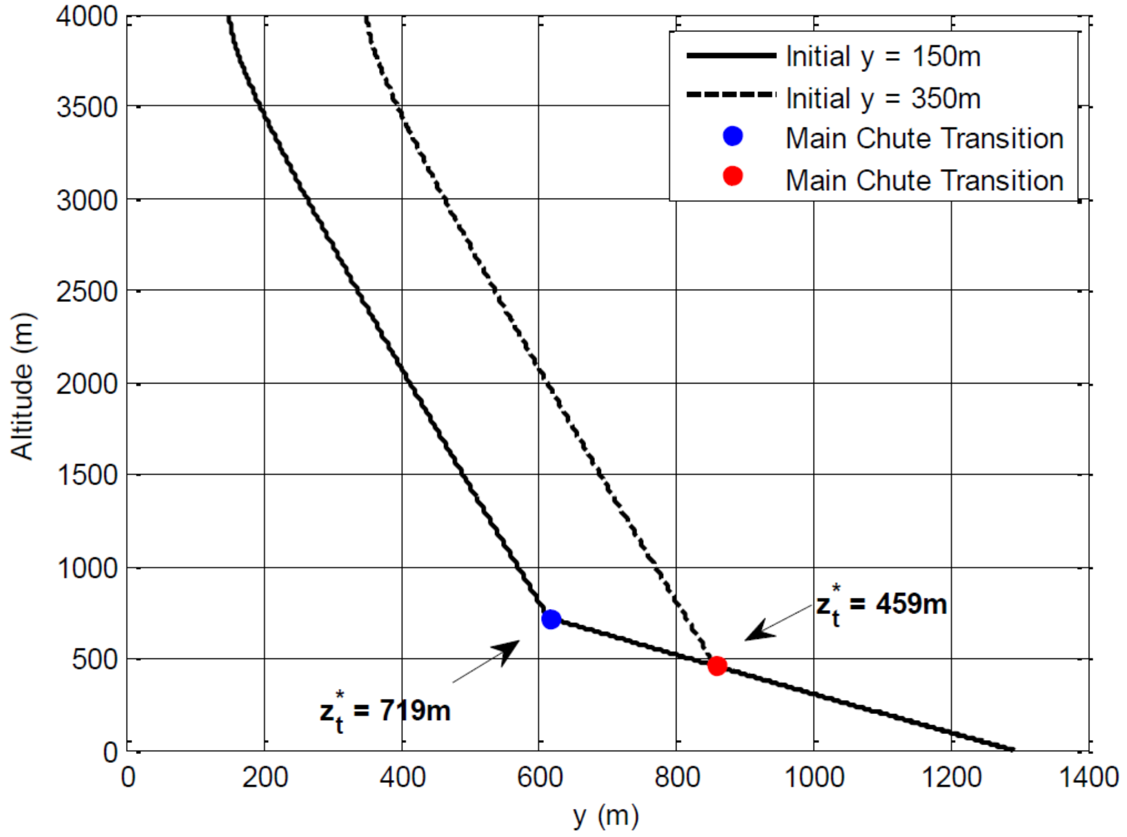


Figure 5.3: Altitude vs downrange for example trajectories (Case 3 and 4) (Note both cases are overlaid after main parachute inflation)

A plot of $\varphi_d(z)$ vs altitude at a specific instant of the trajectory for Case 4 is shown in Figure 5.4. Two different marginal densities are shown, corresponding to N = 250,000 (blue) and N = 1,250,000 (red). It is clear that using more points results in less noise in the marginal probability density, thus resulting in less noise in $\varphi_d(z)$. Less noise in $\varphi_d(z)$ is beneficial for maximum z_t^* selection described by Eq. (35). Unfortunately, increasing the number of points N significantly increases the offline computation time. In these cases, increasing N from 250,000 to 1,250,000 resulted in an 18x increase in runtime.

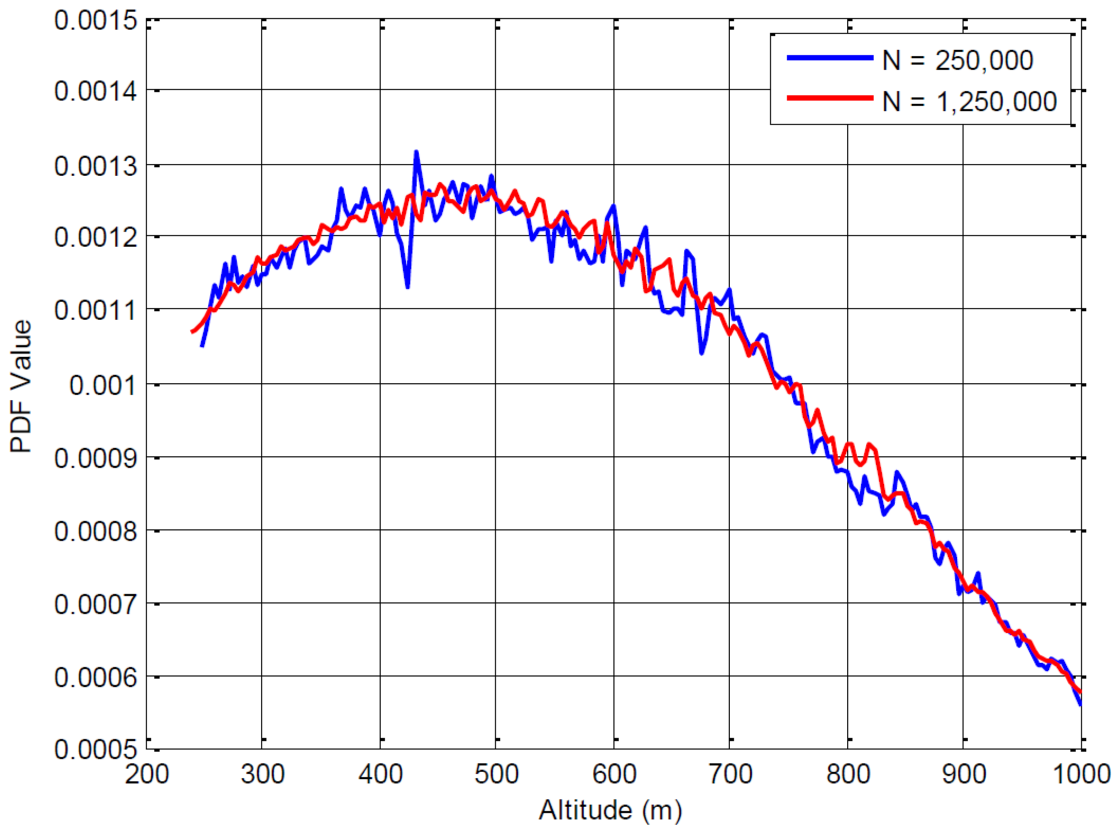


Figure 5.4: $\varphi_d(z)$ vs altitude for Case 4 using N = 250,000 (blue), and N = 1,250,000 (red)

The effect of N and the altitude resolution at which the marginal densities are stored are examined in a trade study that further explores the tradeoffs described above. The altitude resolution Δr represents the altitude between adjacent "slices" of the x-y marginalization φ_{xy}^{i}. Two sets of Monte Carlo simulations of 50 runs each were computed using $N = 500{,}000$ and $\Delta r = 4\text{m}$ where the initial y location of the bundle was randomly varied according to a uniform distribution. Additionally, C_d and wind magnitude were also varied. One set used an optimized transition altitude, while the other used a fixed transition altitude of $z_t = 625\text{m}$. The results are shown in Figure 5.5, where the fixed transition altitude case produces a target-centered 50% circular error probable (CEP) of 81m and the optimized transition altitude produces a target-centered 50% CEP of 19m.

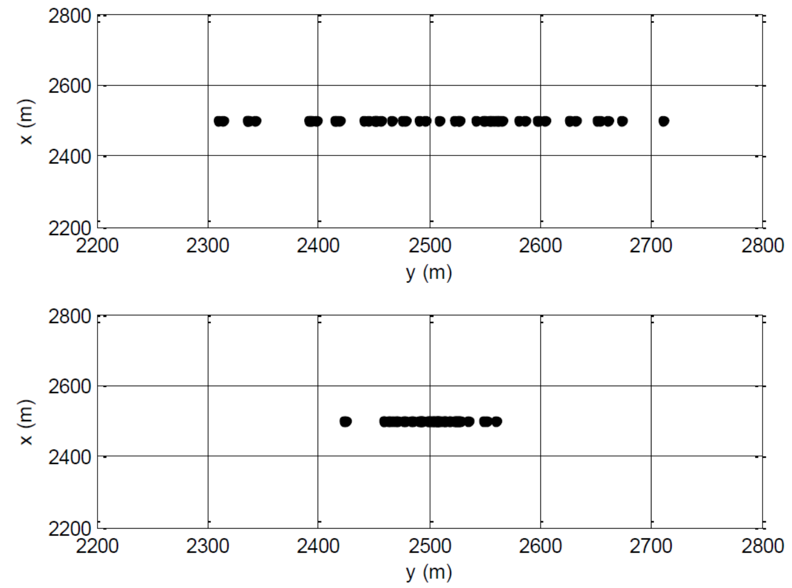


Figure 5.5: Monte Carlo impact dispersion locations for fixed transition altitude case (top) and optimized transition altitude case (bottom)

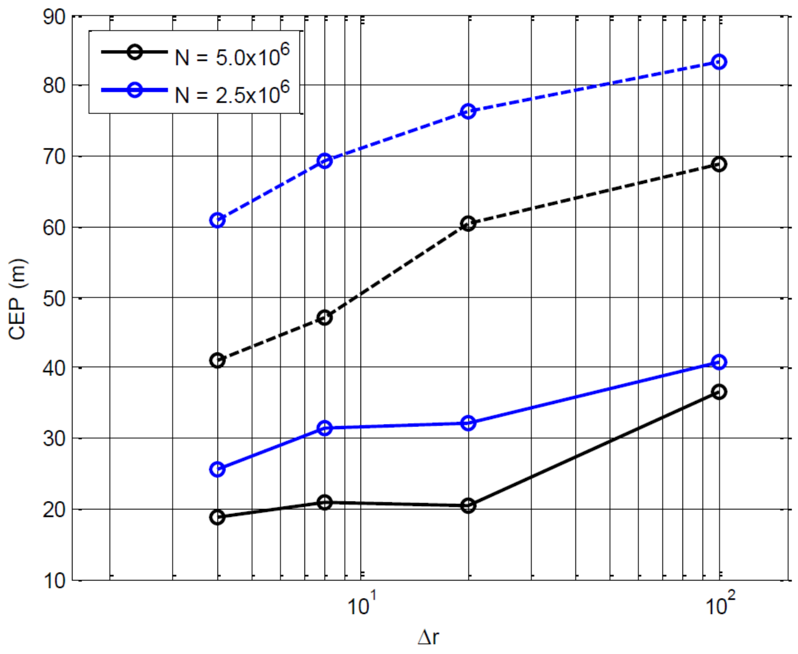


Figure 5.6: CEP vs altitude resolution and *N*, solid lines represent 50% CEP, dashed lines represent 90% CEP

The study is extended further in Figure 5.6, showing how 50% and 90% CEP changes as function of N and Δr. Note that accuracy generally improves as altitude resolution and N both increase. Increasing altitude resolution reduces dispersion only to a certain point, after which it is necessary to increase N to provide further reductions.

Exclusion Regions

The proposed transition altitude optimization algorithm is particularly advantageous because of its ability to enforce exclusion regions in the drop zone, where packages should avoid landing. This capability is demonstrated as follows. A Gaussian inverse cost map is created and centered at $\boldsymbol{x} = 0$, $\boldsymbol{y} = 0$ with diagonal covariance matrix and equal variances in $\boldsymbol{x}$ and $\boldsymbol{y}$ of 10^5 m^2. A rectangular exclusion region is created, as shown in Figure 5.7. Inside of the exclusion region, the inverse cost map $\boldsymbol{\varphi_{xy}}$ is set to zero. The wind values in these examples represent wind stick data obtained from actual dropsonde measurements at Yuma Proving Ground [26]. The wind stick data varies with altitude but not spatially with $\boldsymbol{x}$ or $\boldsymbol{y}$. A "line of control", which represents the possible impact locations obtained by varying the transition altitude between $\boldsymbol{z_{t,min}}$ and $\boldsymbol{z_{t,max}}$, is also shown. Note that the line of control only translates throughout the drop zone as the drop location moves but does not otherwise change shape since the winds are constant in $\boldsymbol{x}$ and $\boldsymbol{y}$.

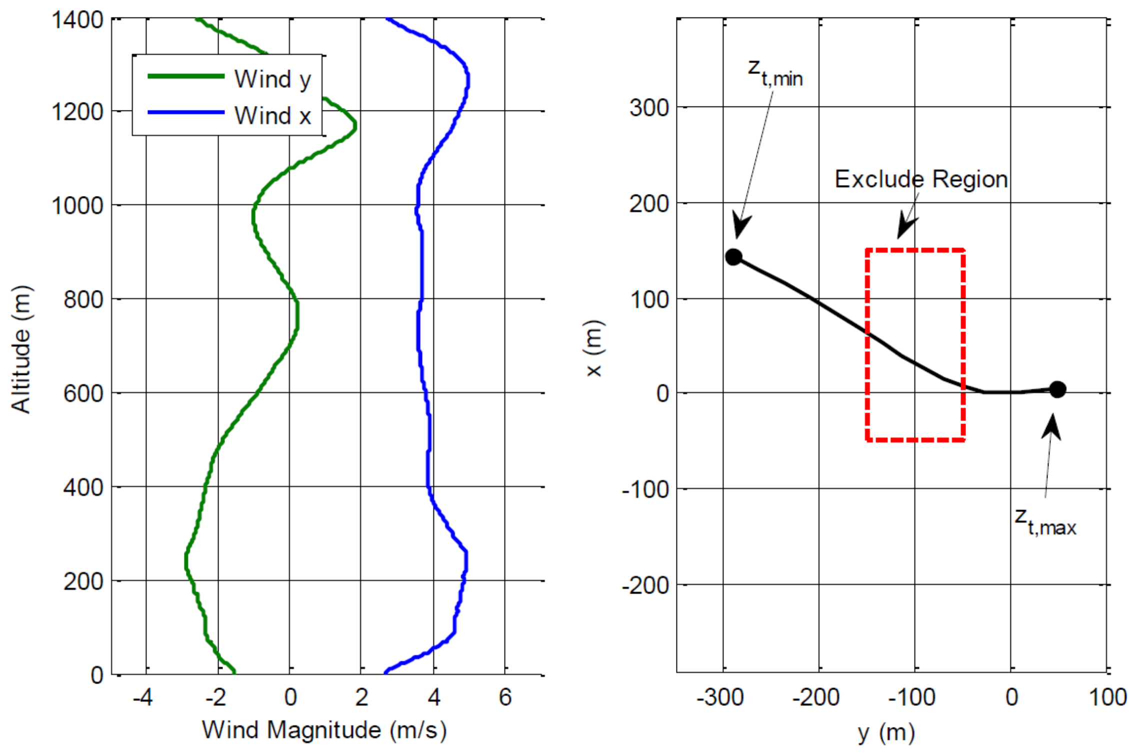


Figure 5.7: Wind magnitude vs altitude (left), Exclusion region geometry and line of control for cases 5-8 (right)

Cases 5-8 all used Monte Carlo simulations of 500 runs each, randomly selecting the drop location according to a uniform distribution in a 500m × 500m area. Additionally, random perturbations to $\boldsymbol{C_d}$, wind magnitude, and wind direction were selected according to Appendix B. Case 5 used a fixed transition altitude of $\boldsymbol{z_t}$ = 625m; case 6 used the optimized transition altitude but did not use the exclude region; and case 7 used the optimized transition altitude and the exclude region. In cases 6 and 7, $\boldsymbol{z_t^*}$ was selected according to Eq. (35) since $\boldsymbol{\varphi_{xy}}$ represents an inverse cost map. The results are presented below in Figures 5.8, 5.9, and 5.10, respectively. Additionally, impact statistics are presented in Table 5.1.

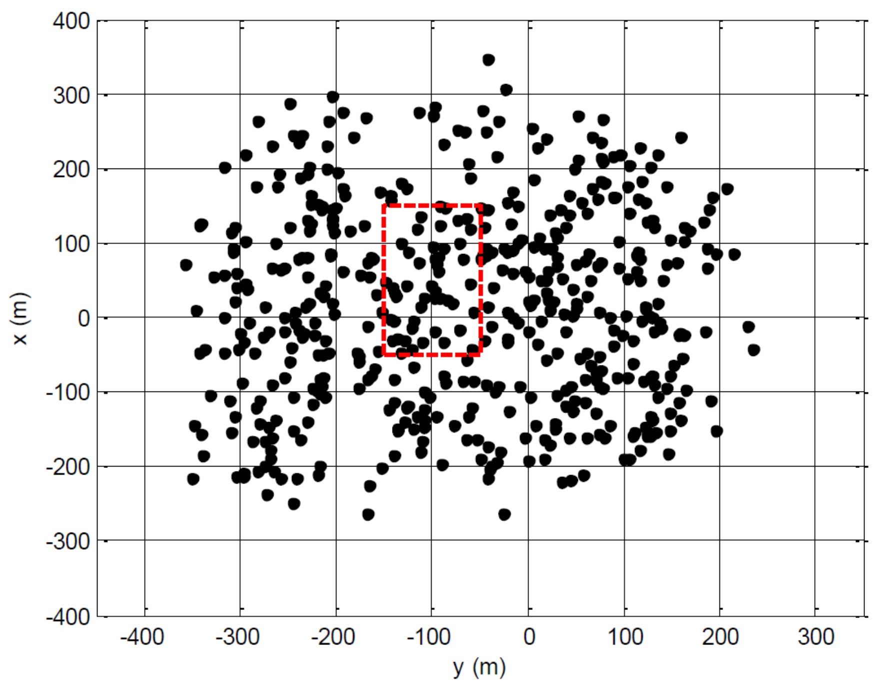


Figure 5.8: Case 5 – Monte Carlo impact locations, fixed transition altitude

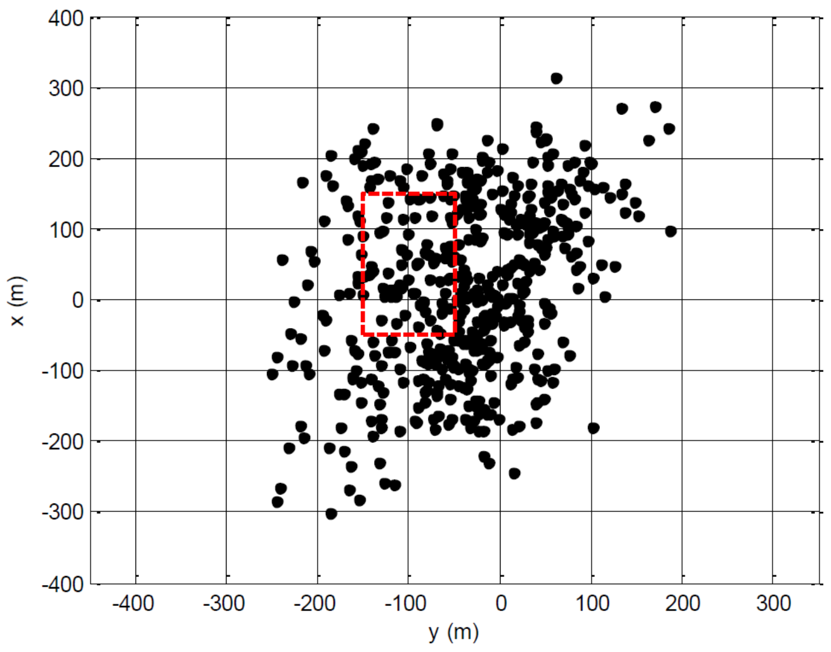


Figure 5.9: Case 6 – Monte Carlo impact locations, optimized transition altitude with no exclusion region

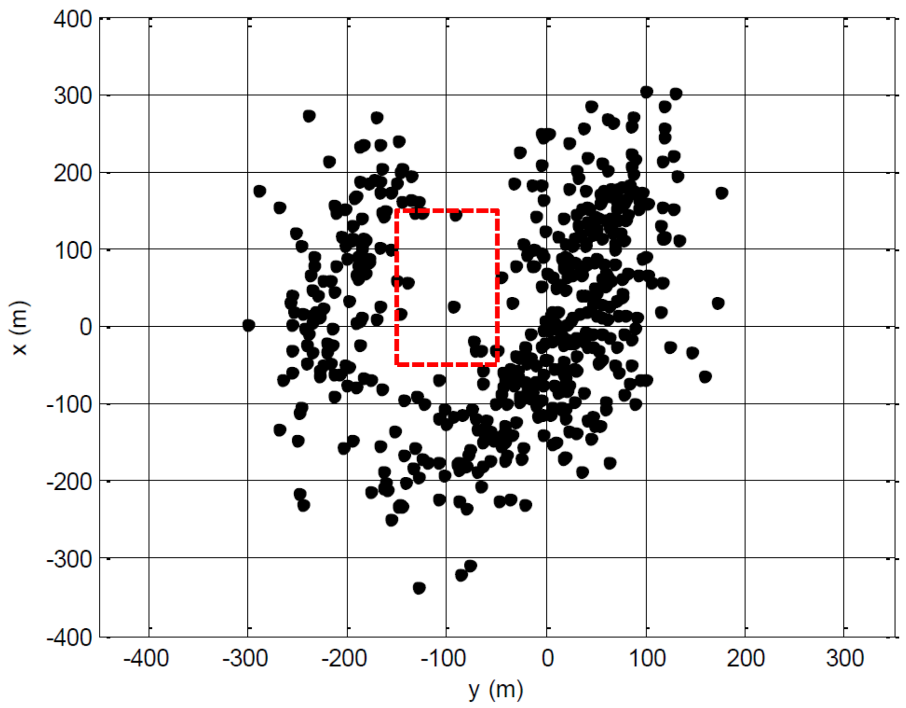


Figure 5.10: Case 7 – Monte Carlo impact locations, optimized transition altitude with exclusion region

As expected, the fixed transition altitude case yields relatively poor CEP, yielding nearly 10% of impacts within the exclusion region. With the optimized transition altitude (but not using the exclusion region), the CEP is reduced drastically since many of the impacts are pushed towards the target but also into the exclusion region. Finally, using the optimized transition altitude and the exclusion region produces a milder CEP reduction compared to case 6, but the number of impacts within the exclusion region is reduced to 1.8%.

The final Monte Carlo simulation, case 8, demonstrates the importance of the uncertainty propagation process. During the definition of the joint probability for this case, the standard deviations of all parameter distributions ($\boldsymbol{C_d}$, wind magnitude, and wind direction) are set to 0, i.e. the PDF is propagated assuming perfect knowledge of all system parameters and wind. However, these parameters are still perturbed during each Monte Carlo run. Case 8 is shown in Figure 5.11 shows. Over twice as many impacts land in the exclusion region compared to case 7. Furthermore, there appears to be a buffer region between the impact density and exclusion region boundary in case 7 that is not present in case 8. This forms because of the presence of uncertainty in generation of the joint PDF, and the resulting diffusion that occurs during propagation. When the PDF is propagated to altitude, the low probability area of the exclusion region slowly increases in sizes, and the boundaries become "fuzzy". This fuzziness, or diffusion, becomes greater as the parameters become more uncertain. This results in a larger buffer region emerging which is desirable when uncertainty in the model and wind conditions increases. This behavior is entirely emergent from the probabilistic planning process and represents the main advantage of this method over a similarly-defined deterministic scheme.

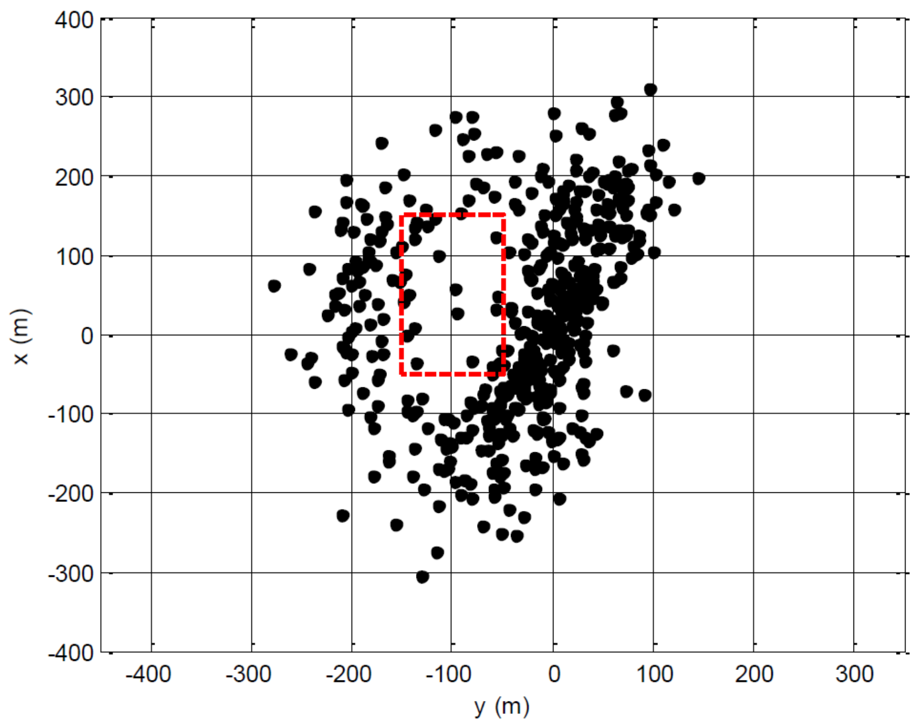


Figure 5.11: Case 8 – Monte Carlo impact locations, optimized transition altitude with no parameter uncertainty used in PDF propagation

Table 5.1: Impact statistics for cases 5-8

	50% CEP	**Number of cases landing inside exclude region**
Case 5	186 m	48
Case 6	130 m	73
Case 7	153 m	9
Case 8	136 m	23

Probabilistic z_t^* Calculation

The ability of the proposed algorithm to match a desired ground impact distribution is further examined in another case study. A scenario is considered in which packages are to be dispersed over multiple areas. A real-world example would be several nearby operating bases or several nearby areas for humanitarian aid delivery. It is desirable that packages distribute themselves amongst the multiple drop zones according to some pre-defined statistical distribution, i.e. the majority of packages should not impact near a single location.

To realize this scenario, consider a humanitarian aid example whereby the desired impact distribution is defined by the union of two disjoint uniform distributions $\boldsymbol{U_1(x,y)}$ and $\boldsymbol{U_2(x,y)}$, and the magnitude of $\boldsymbol{U_1(x,y)}$ is double that of $\boldsymbol{U_2(x,y)}$. By setting $\boldsymbol{\varphi_{xy} = U_1 \cup U_2}$, a joint PDF is created using the parameter distributions in Appendix B and then backpropagated. A steady wind, uniform with altitude, of $\boldsymbol{w_x} = 0$ and $\boldsymbol{w_y} = 7$ m/s is assumed. A Monte Carlo run of 300 simulations is performed, where parameter values are sampled from distributions in Appendix B, all packages are dropped from a single location $\boldsymbol{x} = 0$ and $\boldsymbol{y} = -1{,}016$ m, and $\boldsymbol{z_t^*}$ is selected according to Eq. (36) since $\boldsymbol{\varphi_{PI}(x,y)}$ represents a desired statistical impact distribution.

The results of this scenario, case 9, are shown in Figure 5.12, where $\boldsymbol{U_1}$ is the blue region and $\boldsymbol{U_2}$ is the red region. The resulting impact distribution should result in 2/3 of the impacts being in $\boldsymbol{U_1}$ and 1/3 of the impacts being in $\boldsymbol{U_2}$. Indeed, case 9 yielded 202/300 (67.3%) impacts in $\boldsymbol{U_1}$ and 98/300 (32.7%) impacts in $\boldsymbol{U_2}$. Comparing these results to the same setup using a fixed transition altitude of $\boldsymbol{z_t} = 773$ m results in 158/300 (52.7%) impacts in $\boldsymbol{U_1}$ and 142/300 (47.3%) impacts in $\boldsymbol{U_2}$; these results are not shown below.

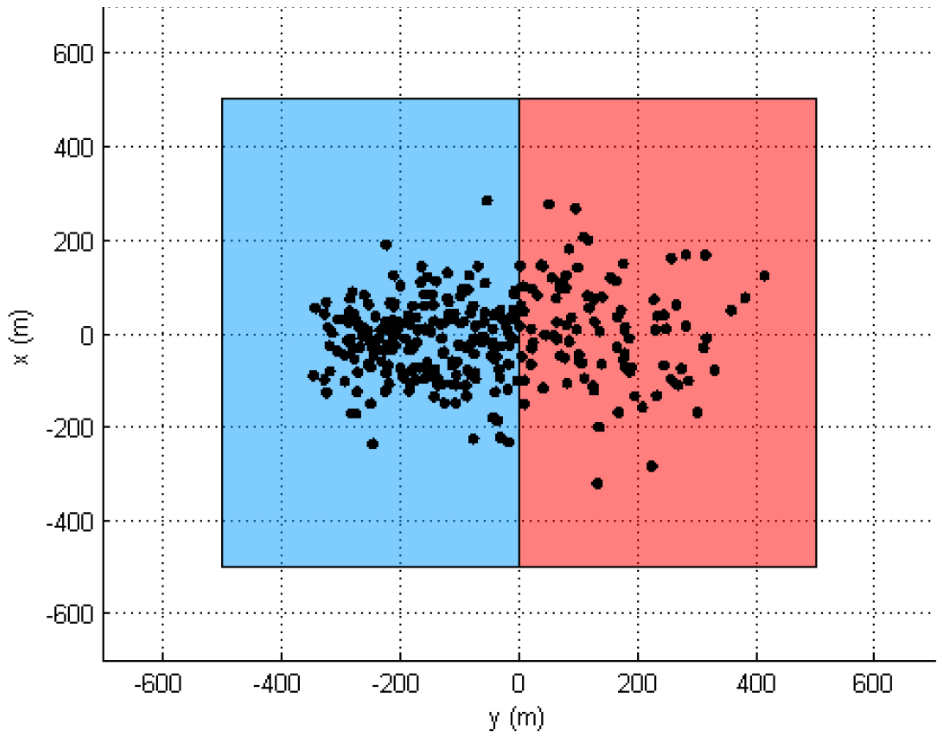


Figure 5.12: Case 9 – Desired distribution and Monte Carlo impact locations

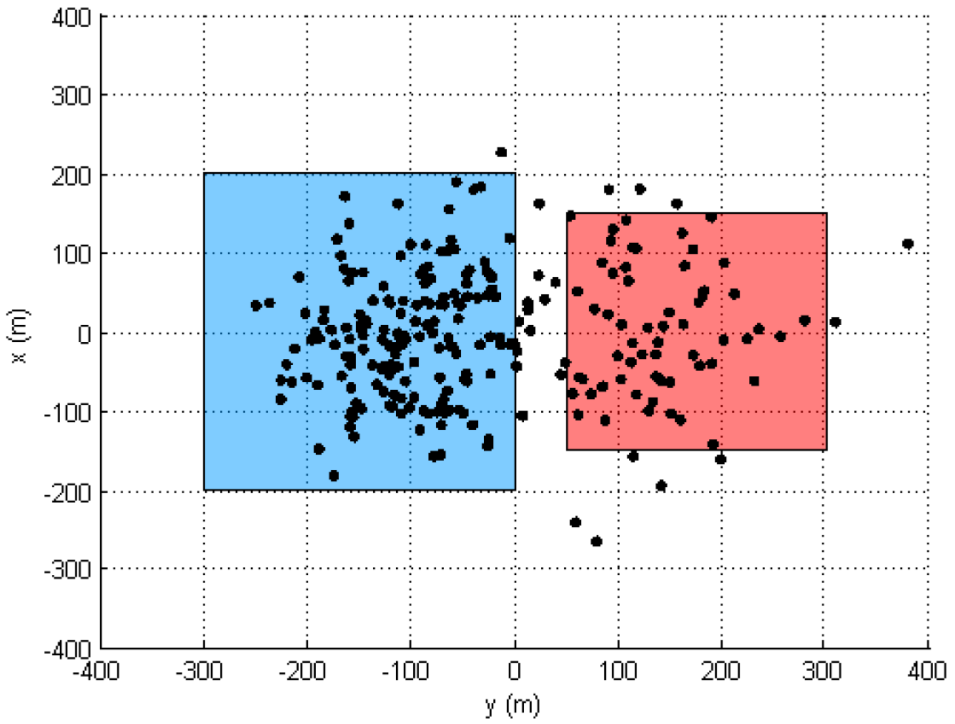


Figure 5.13: Case 10 – Desired distribution and Monte Carlo impact locations

A second example, case 10, is considered in Figure 5.13. Two non-adjacent distributions with slightly smaller supports are employed. The same PDF values for $\boldsymbol{U_1}$ and $\boldsymbol{U_2}$ as above are used, and the difference in the sizes of the supports is accounted for. Thus, the resulting impact distribution should yield 75% of impacts within $\boldsymbol{U_1}$ and 25% of impacts with $\boldsymbol{U_2}$. Case 10 yielded 166/250 (66.4%) impacts in $\boldsymbol{U_1}$ and 59/250 (23.6%) impacts in $\boldsymbol{U_2}$. The remaining impacts fell outside of both distributions. Again, the Monte Carlo process is repeated using a fixed transition altitude of $\boldsymbol{z_t}$ = 773m resulting in 127 (50.8%) impacts in $\boldsymbol{U_1}$, 48 (19.2%) impacts in $\boldsymbol{U_2}$, and 30% outside both distributions. It worthwhile to note that the number of impacts outside of both regions is reduced from 30% to 10% when the optimized transition altitude is used. This can potentially be reduced further by increasing $\boldsymbol{N}$ and/or increasing the binning resolution.

Complex Drop Zones

The final case study involves a complex scenario designed to emulate a real-world mission. The scenario is based on a road network and includes a 3D spatially-varying wind field generated from the Weather Research and Forecasting (WRF) tool [29]. Additionally, a desired impact point is specified on the road network, as shown in Figure 5.14.

The 3D wind field varies with respect to $\boldsymbol{x}$, $\boldsymbol{y}$, and altitude. The wind data used for this example was generated for the Salinas Valley in Northern California and has a resolution of 333m in both the $\boldsymbol{x}$ and $\boldsymbol{y}$ directions and variable grid spacing in the altitude direction. For the drop locations selected in this example, horizontal winds of 15+ m/s are experienced over $\boldsymbol{z_{t,max}}$ during drogue descent, and winds below $\boldsymbol{z_{t,max}}$ are nominally in the 7-8 m/s range with highly variable direction. An example line of control, denoting the various achievable impact locations from varying $\boldsymbol{z_t}$, is shown in Figure 5.14 in red.

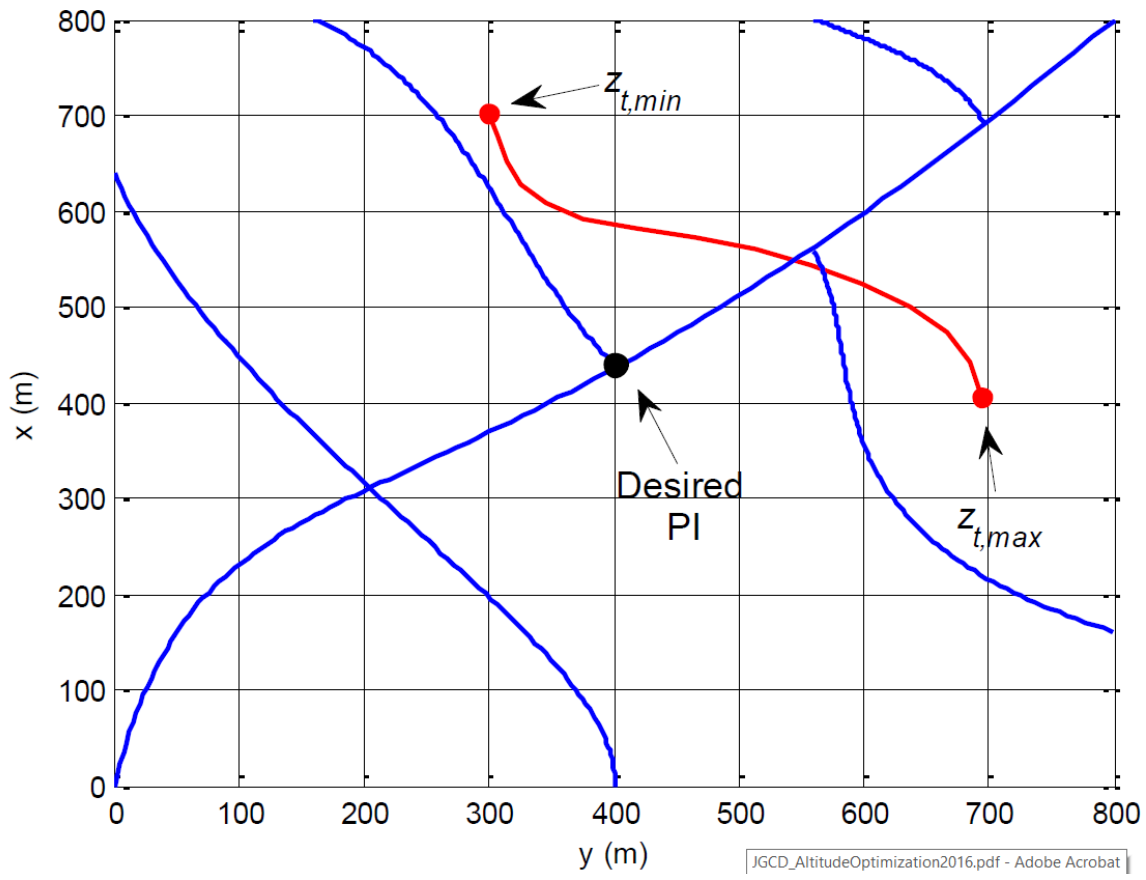


Figure 5.14: Road network and line of control for complex drop zone example

In the practical implementation of such a scenario, if the packages cannot land at the desired impact point, then they should at least land along a road for easier retrieval. However, if the package cannot land at the desired impact point but can land at multiple locations along the road network, it should land at the point with minimum geodesic distance to the desired impact point. Thus, the inverse cost map should be high along roads, decrease as the distance to a road increases, and decrease as the geodesic distance along the road to the desired impact point increases. Additionally, points off of the road network should decrease relative to the distance to the closest point on the road network. This inverse cost map is visualized in Figure 5.15, where the inverse cost is maximum at the desired impact point.

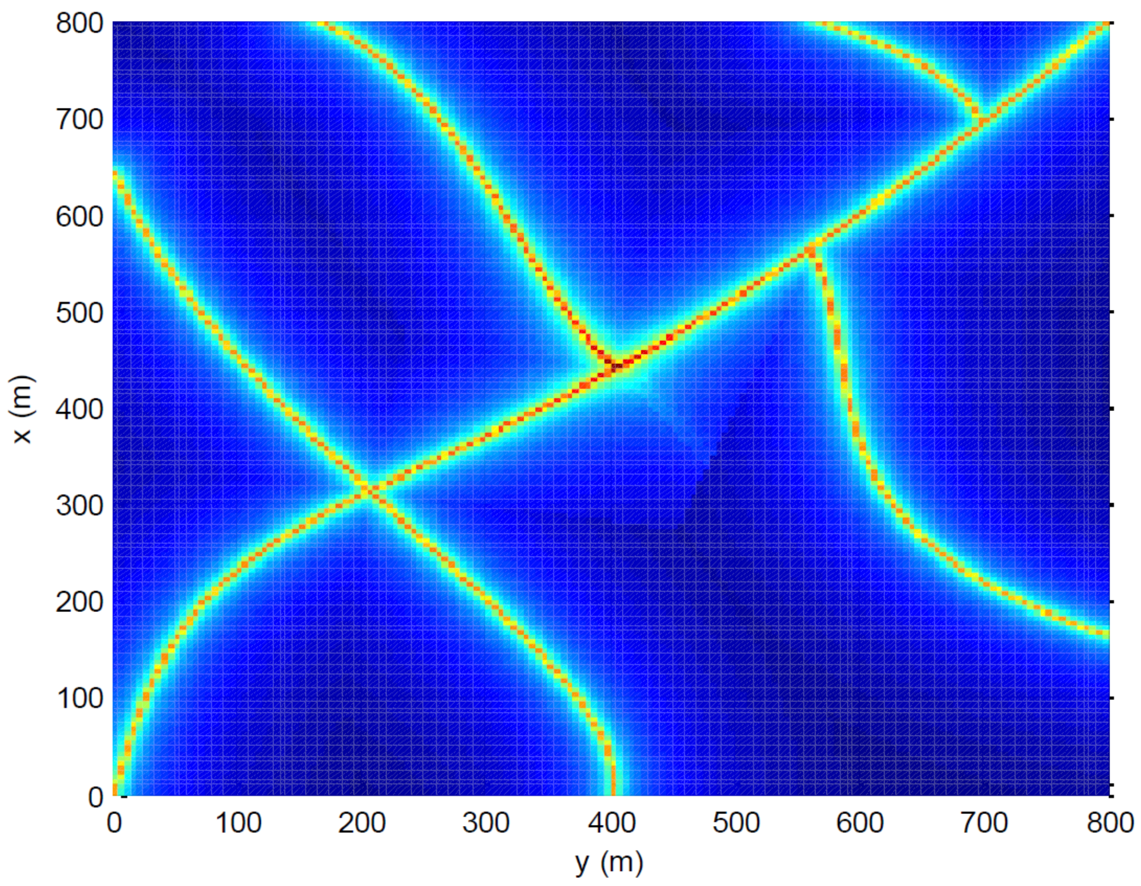


Figure 5.15: Inverse cost map for complex drop zone example

Two Monte Carlo cases were performed: one using a fixed transition altitude of $\boldsymbol{z_t}$ = 625m, and one using the optimized transition altitude as computed by Eq. (35). The uncertainty propagation and marginalization algorithms for these cases used $\boldsymbol{N}$= 2,000,000, and the binning resolution was increased to $\boldsymbol{p} = \boldsymbol{q} = 120$ to maintain adequate resolution of the narrow road features. Low binning resolution makes these road features highly diffused when the marginal density is computed, and thus resolution was increased. For each run, the drop location was randomized within a 500m × 500m area and random $\boldsymbol{C_d}$, wind magnitude, and wind direction perturbations were selected according to the distributions in Appendix B. 300 runs were computed in each Monte Carlo simulation.

The results of the fixed transition altitude case are shown in Figure 5.16, while the results of the optimized transition altitude case are shown in Figure 5.17. Impact statistics for these cases are also given in Table 5.2, including the mean and median distances to the closest point on a road. These distances are determined by computing the distance from each impact location to its respective closest point on a road. It can be seen that the optimized transition altitude algorithm effectively reshapes the impact distribution around the road network. The median distance from the packages to the road network is reduced by 71%. Similarly, the maximum distance from any package to a road is reduced by 57%.

It is worthwhile to note that many of the impacts in Figure 5.17 are skewed to the north, or upwind, side of the road network even though these trajectories could have impacted the road network farther south. This is the result of the diffusion of probabilities that occurs at higher altitudes, i.e. the PDF value along each road is lower at higher altitudes and higher at lower altitudes. This produces the exactly desired behavior from a probabilistic standpoint: in the presence of uncertainty, it is safer to transition at the lowest possible altitude to reduce the time under the main parachute dynamics where the system is exposed to uncertain winds. This behavior is further evidence of the proposed algorithm's proper treatment of uncertainty in the unguided airdrop problem and emerges directly from the probabilistic formulation of the algorithm.

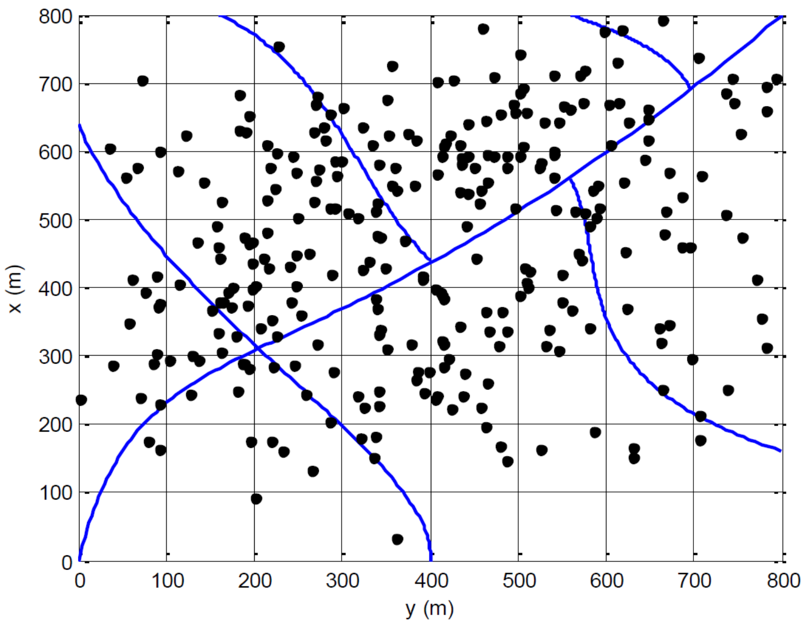


Figure 5.16: Monte Carlo impact locations, fixed transition altitude case

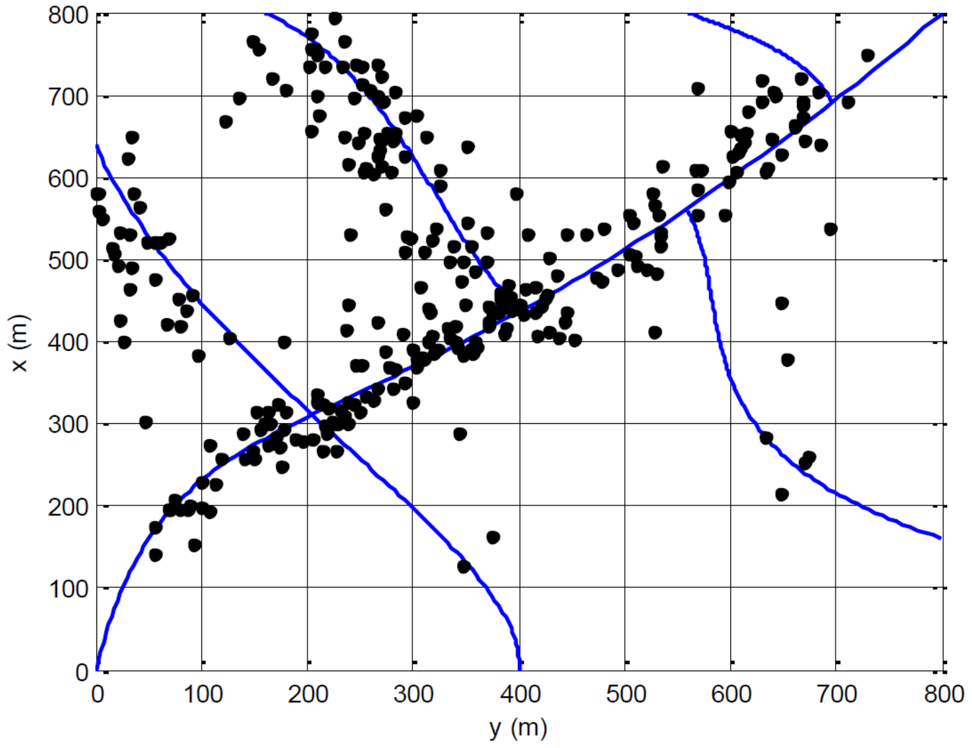


Figure 5.17: Monte Carlo impact locations, optimized transition altitude case

Table 5.2: Monte Carlo impact statistics for complex drop zone example

	Fixed Transition Altitude	**Optimized Transition Altitude**
Mean distance from impacts to road	64 m	24 m
Median distance from impact to road	56 m	16 m
Maximum distance from impact to road	305 m	131 m

CHAPTER 6

CONCLUSION

A transition altitude optimization algorithm for HALO ballistic airdrop has been presented. The proposed algorithm has been shown to be quite successful in shaping the impact dispersion pattern of unguided airdrops. The use of uncertainty as an input to a mission planner and the analysis of the uncertainty propagation proved critical in achieving these results. The results are generated through a detailed simulation framework that includes the airdrop dynamic model, atmospheric and wind model, and parachute inflation model. The uncertainty propagation is helped in part by a comprehensive study of two boundary value problem solvers used for trajectory generation. Finally, a scenario creation tool was presented as a complimentary aspect of the current mission planning procedure, allowing the robust generation of complex input distributions.

Future Work

Further improvements to the algorithm can still be made, including terrain implementation and formalization of an all-encompassing mission planner. A terrain model for the both the mission planner and desired impact distribution definition does not currently exist. Currently, all terrain is assumed to be flat. While this works well for test cases, a larger number of scenarios could be explored with the implementation of a robust terrain model. Additionally, many of the components of the processes detailed in this thesis are separate software structures. It would be beneficial to future development to have these pieces folded into a main software base.

APPENDIX A

MODIFIED SIMPLE SHOOTING METHOD CODE IMPLEMENTATION

The following code is the implementation of the Modified Simple Shooting Method for BVP solving in the mission planner C++ code used for the transition altitude optimization algorithm. Any variables used that are not explicitly declared in the function are assumed to be declared as public in the header. The custom variable `state` is a `typedef` of `vector<double>`. Many functions used in the main `BVPsolveMSSM` function are provided so that the interested reader can follow the path of execution. Functions not included are deemed unimportant to execution of the solver.

```
int DynamicSystem::BVPsolveMSSM(bool showText)
{
    showState = false;

    // calculate density at current altitude for vterm calculation
    double dropAltitude = 1000.0;
    double rho = 1.22566578494891 * pow(1.00000000-0.0000225696709*(dropAltitude),4.258);
    vterm = sqrt(2*m*g/(rho*Cd*S));

    // set the initial state vector. x and y guesses are calculated. x0[4] and x0[5] correspond
    // to velocity in x and y, these are set equal to the wind at the point (x,y) at the start
    // of the solver loop.
    double guess[2];
    CalculateGuess(guess);
    double xInitial = guess[0];
    double yInitial = guess[1];
    double zInitial = -1000.0;
    double xDotInitial = 0.0;
    double yDotInitial = 0.0;
    double zDotInitial = vterm;

    vector<double> initialState = { 0.0, xInitial, yInitial, zInitial,
                                    xDotInitial, yDotInitial, zDotInitial, 0.0, joint_ground };

    GetSysWind(initialState[1], initialState[2], -initialState[3]);
    initialState[4] = syswind[3]*cos(syswind[4]);
    initialState[5] = syswind[3]*sin(syswind[4]);

    // Initialize time and state vectors and counters
    vector<double> myTimes;
    myTimes.push_back(0.0);

    vector<vector<double>> myStates;
    myStates.push_back(initialState);

    int n = 0;
    int numShots = 1;
    int k = 0;
    int totalIterations = 0;
    int allowableIterations = 100;

    // initialize public variables from header
    dxdt.assign(9, 0.00);
    nextState.assign(9, 0.00);
    nextTime = 0.0;
    refPathValAtTime.assign(2, 0.0);
    boundsValAtTime.assign(4, 0.0);

    // Initialize shot guess variables
    vector<double> tempGuess = {xInitial, yInitial};
    shotGuess.clear();
    shotGuess.push_back(tempGuess);

    double finalTolerance = 0.0;

    // ---------- FOR DEBUG -----------
    // vector<double> tempMyTimes;
    // vector<vector<double>> tempMyStates;

    // Set desired impact points
    desiredImpactX = xtar;
    desiredImpactY = ytar;

    // Diagnostic print out
    if(showText)
    {
        cout << "Starting MSSM" << endl;
    }

    // ---------- FOR DEBUG -----------
    // cout << "Cd = " << Cd << "\twm_perturb = " << wm_perturb << "\twd_perturb = " << wd_perturb << endl;

    // Solve for a drop location that keeps the package within bounds until ground impact
    while(myStates.at(n).at(3) <= 0.0)
    {
        // Check if x and y states are in bounds
        FuncRefPath(myStates.at(n).at(3));

        // ---------- FOR DEBUG: Printint out bounds -----------
        // cout << myStates.at(n).at(3) << "\t|\t"
        //      << boundsValAtTime[0] << " > " << myStates.at(n).at(1) << " > " << boundsValAtTime[1] <<
"\t|\t"
        //      << boundsValAtTime[2] << " > " << myStates.at(n).at(2) << " > " << boundsValAtTime[3] <<
endl;

        // ---------- FOR DEBUG: Printing out state history -----------
        // cout << myTimes.at(n) << "\t\t"
        //      << myStates.at(n).at(1) << "\t\t"
        //      << myStates.at(n).at(2) << "\t\t"
        //      << myStates.at(n).at(3) << "\t\t"
        //      << myStates.at(n).at(4) << "\t\t"
        //      << myStates.at(n).at(5) << "\t\t"
        //      << myStates.at(n).at(6) << endl;

        // If x or y states go out of bounds, correct the guess
        if ((myStates.at(n).at(1) > boundsValAtTime[0] || myStates.at(n).at(1) < boundsValAtTime[1])
            || (myStates.at(n).at(2) > boundsValAtTime[2] || myStates.at(n).at(2) < boundsValAtTime[3]))
        {
            cout << numShots << "\t" << myStates.at(n).at(3) << endl;

            // Diagnostic print outs
            if(showText)
            {
                cout << "===== Shot " << numShots << " went out of bounds at altitude "
                     << myStates.at(n).at(3) << " meteres ====="
                     << "\n--Drop = ( " << myStates.at(0).at(1) << ", " << myStates.at(0).at(2)
                     << ")\tImpact = ( " << myStates.at(n).at(1) << ", " << myStates.at(n).at(2)
                     << ")\tTarget = ( " << xtar << ", " << ytar << ")" << endl;

                cout << "-- [ " << myTimes.at(n) << "\t\t"
                     << myStates.at(n).at(1) << "\t\t"
                     << myStates.at(n).at(2) << "\t\t"
                     << myStates.at(n).at(3) << "\t\t"
                     << myStates.at(n).at(4) << "\t\t"
                     << myStates.at(n).at(5) << "\t\t"
                     << myStates.at(n).at(6) << "]" << endl;
            }

            int correctionShots = ModifiedNewtonMethod(initialState, myStates.at(n), true, false, showText);

            // Reset initial state
            initialState[0] = 0.0;
            initialState[1] = shotGuess.back().at(0);
            initialState[2] = shotGuess.back().at(1);
            initialState[3] = zInitial;
            initialState[4] = xDotInitial;
            initialState[5] = yDotInitial;
            initialState[6] = zDotInitial;
            initialState[7] = 0.0;
            initialState[8] = joint_ground;

            GetSysWind(initialState[1], initialState[2], -initialState[3]);
            initialState[4] = syswind[3]*cos(syswind[4]);
            initialState[5] = syswind[3]*sin(syswind[4]);

            // Reset state variables
            myTimes.clear();
            myTimes.push_back(0.0);
            myStates.clear();
            myStates.push_back(initialState);
            n = 0;

            // Sum up the number of shots
            totalIterations = totalIterations + 1 + correctionShots;

            if(showText)
            {
                cout << "Total Iterations = " << totalIterations << endl;
            }

            // Increment counter for next shot
            numShots++;

            // Diagnostic print outs
            if(showText)
            {
                cout << "\t-- Found new guess ( " << shotGuess.back().at(0) << " , " <<
shotGuess.back().at(1) << " )" << endl;
            }
        } // end if - bounds check

        // If we exceed 50 total iterations, break out of the solver
        if(totalIterations > allowableIterations)
        {
            cout << "ERROR: Exceeded allowed number of total iterations." << endl;
            break;
        }

        // Integrate system
        DoRK4(myTimes.at(n), myStates.at(n));
        myTimes.insert(myTimes.end(), nextTime);
        myStates.insert(myStates.end(), nextState);

        n++;
    } // end while

    // Correct final guess if we're less than 50 total iterations
    if(totalIterations < allowableIterations)
    {
        // Simulate final guess for verification
        x0[0] = myStates.at(0).at(0);   // time
        x0[1] = myStates.at(0).at(1);   // X
        x0[2] = myStates.at(0).at(2);   // Y
        x0[3] = myStates.at(0).at(3);   // Z
        x0[4] = myStates.at(0).at(4);   // Vx
        x0[5] = myStates.at(0).at(5);   // Vy
        x0[6] = myStates.at(0).at(6);   // Vz
        x0[7] = myStates.at(0).at(7);   // Trace(f)
        x0[8] = myStates.at(0).at(8);   // joint pdf

        GetSysWind(x0[1], x0[2], -x0[3]);
        x0[4] = syswind[3]*cos(syswind[4]);
        x0[5] = syswind[3]*sin(syswind[4]);

        IntegrateToZ(0.0);
        myStates.at(n) = x_impact;

        // Diagnostic print outs
        if(showText)
        {
            cout << "===== Shot " << numShots << " stayed in bounds all the way to the ground" << endl;
            cout << "\t-- Starting Point = ( " << myStates.at(0).at(1) << " , " << myStates.at(0).at(2) << "
)" << endl;
            cout << "\t-- Impact Point = ( " << myStates.at(n).at(1) << " , " << myStates.at(n).at(2) << " ,
" << myStates.at(n).at(3) << " )" << endl;

            cout << "\nCorrecting guess for tolerance" << endl;
        }

        myStates.at(n).at(3) = 0.0;
        int correctionShots = ModifiedNewtonMethod(initialState, myStates.at(n), true, true, showText);

        // Done with main shots, so only add correction shots
        // NOTE: The plus 1 is because the final successful shot is not
        // counted elsewhere, so we count it here
        totalIterations = totalIterations + correctionShots + 1;
    }

    // Simulate final guess for verification
    // showState = true;
    x0[0] = 0.00;                   // time
    x0[1] = shotGuess.back().at(0);  // X
    x0[2] = shotGuess.back().at(1);  // Y
    x0[3] = zInitial;               // Z
    x0[4] = xDotInitial;            // Vx
    x0[5] = yDotInitial;            // Vy
    x0[6] = zDotInitial;            // Vz
    x0[7] = 0.00;                   // Trace(f)
    x0[8] = joint_ground;           // joint pdf

    GetSysWind(x0[1], x0[2], -x0[3]);
    x0[4] = syswind[3]*cos(syswind[4]);
    x0[5] = syswind[3]*sin(syswind[4]);

    IntegrateToZ(0.0);//myStates.at(n).at(3));

    if(showText)
    {
        myTimes.clear();
        myStates.clear();
        myTimes.push_back(0.0);
        myStates.push_back(x0);
        n = 0;
        cout << "===================================Final State Sim" << endl;
        while(myStates.at(n).at(3) <= 0.0)
        {
            FuncRefPath(myStates.at(n).at(3));

            // ---------- FOR DEBUG: Printint out bounds -----------
            // cout << myStates.at(n).at(3) << "\t|\t"
            //      << boundsValAtTime[0] << " > " << myStates.at(n).at(1) << " > " << boundsValAtTime[1] <<
"\t|\t"
            //      << boundsValAtTime[2] << " > " << myStates.at(n).at(2) << " > " << boundsValAtTime[3] <<
endl;

             // ---------- FOR DEBUG: Printing out state history -----------
            // cout << myTimes.at(n) << "\t\t"
            //      << myStates.at(n).at(1) << "\t\t"
            //      << myStates.at(n).at(2) << "\t\t"
            //      << myStates.at(n).at(3) << "\t\t"
            //      << myStates.at(n).at(4) << "\t\t"
            //      << myStates.at(n).at(5) << "\t\t"
            //      << myStates.at(n).at(6) << endl;

             // If x or y states go out of bounds, correct the guess
            if ((myStates.at(n).at(1) > boundsValAtTime[0] || myStates.at(n).at(1) < boundsValAtTime[1])
                || (myStates.at(n).at(2) > boundsValAtTime[2] || myStates.at(n).at(2) < boundsValAtTime[3]))
            {
                cout << "WENT OUT OF BOUDNS" << endl;
            }

            // Integrate system
            DoRK4(myTimes.at(n), myStates.at(n));
            myTimes.insert(myTimes.end(), nextTime);
            myStates.insert(myStates.end(), nextState);

            n++;
        } // end while
    } // end if
    // Diagnostic Print out
    if(showText)
    {
        cout << "\t-- Final starting Point = ( " << x0[1] << " , " << x0[2] << " )" << endl;
        cout << "\t-- Final Impact Point = ( " << x_impact[1] << " , " << x_impact[2] << " , " << x_impact[3]
<< " )" << endl;
    }

    return totalIterations;
}
```

```
int DynamicSystem::ModifiedNewtonMethod(state initialState, state finalState, bool doJacobian, bool
finalCorrection, bool showText)
{
    // Initialize shot calculation variables
    // -- Jacobian
    vector<double> DF;
    DF.assign(4, 0.0);
    double x_xplus, x_xminus, x_yplus, x_yminus;
    double y_xplus, y_xminus, y_yplus, y_yminus;
    double dropPerturbation = 10.0;

    // -- F
    double f1, f2;

    // -- lambda
    double lambda = 1.0;//0.3;//1.0;
    double error;
    double errorprev;

    // Initialize counter
    int mNMCount = 0;
    int k = shotGuess.size()-1;

    // Initialize dummy guess
    vector<double> tempGuess;
    tempGuess.assign(2, 0.0);

    // Initialize initial state values
    double xInitial = initialState[1];
    double yInitial = initialState[2];
    double zInitial = initialState[3];
    double xDotInitial = initialState[4];
    double yDotInitial = initialState[5];
    double zDotInitial = initialState[6];

    // Initialize final state values
    double xFinal = finalState[1];
    double yFinal = finalState[2];
    double zFinal = finalState[3];

    // Initialize allowable number of iterations
    int totalAllowedNewtonIterations = 0;
    if(finalCorrection)
    {
        totalAllowedNewtonIterations = 10;
    }
    else
    {
        totalAllowedNewtonIterations = 10;
    }
    bool haveNotHitCheck = true;
    int maxAllowedIterations = 50;

    // Correct the guess until final tolerance is less than the target tolerance
    double finalTolerance = sqrt( pow( refPathValAtTime[0]-xFinal , 2.0) + pow( refPathValAtTime[1]-yFinal ,
2.0) );
    while(finalTolerance > tar_tol)
    {
        // If we exceed the max number of allowed iterations, break out of the solver
        if(mNMCount > maxAllowedIterations)
        {
            cout << "ERROR: Exceeded allowed number of Newton iterations." << endl;
            break;
        }
```

```
// Increment Newton iteration counter
mNMCount++;

// If Newton iteration counter exceeds 10 iterations, turn off the Jacobian computation
// if we're at the final correction or break out of the solver if we're at other corrections
if((mNMCount > totalAllowedNewtonIterations) && (haveNotHitCheck == true))
{
    haveNotHitCheck = false;
    if(finalCorrection)
    {
        doJacobian = false;
    }
    else
    {
        mNMCount = totalAllowedNewtonIterations;
        break;
    }
}

// Diagnostic print out
if(showText)
{
    cout << "\t-- Starting Newton Method Iteration " << mNMCount << endl;
}

// Compute Jacobian
if(doJacobian)
{
    // --- X plus h ---
    x0[0] = 0.00;                                       // time
    x0[1] = shotGuess.at(k).at(0) + dropPerturbation;   // X
    x0[2] = shotGuess.at(k).at(1);                      // Y
    x0[3] = zInitial;                                   // Z
    x0[4] = xDotInitial;                                // Vx
    x0[5] = yDotInitial;                                // Vy
    x0[6] = zDotInitial;                                // Vz
    x0[7] = 0.00;                                       // Trace(f)
    x0[8] = joint_ground;                               // joint pdf

    GetSysWind(x0[1], x0[2], -x0[3]);
    x0[4] = syswind[3]*cos(syswind[4]);
    x0[5] = syswind[3]*sin(syswind[4]);

    IntegrateToZ(-zFinal);
    x_xplus = x_impact[1];
    y_xplus = x_impact[2];

    // --- X minus h ---
    x0[0] = 0.00;                                       // time
    x0[1] = shotGuess.at(k).at(0) - dropPerturbation;   // X
    x0[2] = shotGuess.at(k).at(1);                      // Y
    x0[3] = zInitial;                                   // Z
    x0[4] = xDotInitial;                                // Vx
    x0[5] = yDotInitial;                                // Vy
    x0[6] = zDotInitial;                                // Vz
    x0[7] = 0.00;                                       // Trace(f)
    x0[8] = joint_ground;                               // joint pdf

    GetSysWind(x0[1], x0[2], -x0[3]);
    x0[4] = syswind[3]*cos(syswind[4]);
    x0[5] = syswind[3]*sin(syswind[4]);

    IntegrateToZ(-zFinal);
    x_xminus = x_impact[1];
    y_xminus = x_impact[2];

    // --- Y plus h ---
    x0[0] = 0.00;                                       // time
    x0[1] = shotGuess.at(k).at(0);                      // X
    x0[2] = shotGuess.at(k).at(1) + dropPerturbation;   // Y
    x0[3] = zInitial;                                   // Z
    x0[4] = xDotInitial;                                // Vx
    x0[5] = yDotInitial;                                // Vy
    x0[6] = zDotInitial;                                // Vz
    x0[7] = 0.00;                                       // Trace(f)
    x0[8] = joint_ground;                               // joint pdf
```

```
            GetSysWind(x0[1], x0[2], -x0[3]);
            x0[4] = syswind[3]*cos(syswind[4]);
            x0[5] = syswind[3]*sin(syswind[4]);

            IntegrateToZ(-zFinal);
            x_yplus = x_impact[1];
            y_yplus = x_impact[2];

            // --- Y minus h ---
            x0[0] = 0.00;                                      // time
            x0[1] = shotGuess.at(k).at(0);                     // X
            x0[2] = shotGuess.at(k).at(1) - dropPerturbation;   // Y
            x0[3] = zInitial;                                  // Z
            x0[4] = xDotInitial;                               // Vx
            x0[5] = yDotInitial;                               // Vy
            x0[6] = zDotInitial;                               // Vz
            x0[7] = 0.00;                                      // Trace(f)
            x0[8] = joint_ground;                              // joint pdf

            GetSysWind(x0[1], x0[2], -x0[3]);
            x0[4] = syswind[3]*cos(syswind[4]);
            x0[5] = syswind[3]*sin(syswind[4]);

            IntegrateToZ(-zFinal);
            x_yminus = x_impact[1];
            y_yminus = x_impact[2];

            DF[0] = (1 / (2*dropPerturbation))*(x_xplus - x_xminus);
            DF[1] = (1 / (2*dropPerturbation))*(x_yplus - x_yminus);
            DF[2] = (1 / (2*dropPerturbation))*(y_xplus - y_xminus);
            DF[3] = (1 / (2*dropPerturbation))*(y_yplus - y_yminus);
        }
        else
        {
            DF[0] = 1;
            DF[1] = 0;
            DF[2] = 0;
            DF[3] = 1;
        }

        // -- Diagnostic print out
        if(showText)
        {
            cout << "\t\tDF = [[ " << DF[0] << "\t" << DF[1] << "\n\t\t" << DF[2] << "\t" << DF[3] << " ]]"
<< endl;
        }

        // Compute F
        f1 = xFinal - refPathValAtTime[0];
        f2 = yFinal - refPathValAtTime[1];

        // -- Diagnostic print out
        if(showText)
        {
            cout << "\t\tF = [ " << f1 << "\t" << f2 << " ] "<< endl;
        }

        // Compute lambda
        error = f1*f1 + f2*f2;
        if (mNMCount != 1)
        {
            if (error > errorprev) {lambda /= 1.25;}       // error increased, decrease step
            else if (error < errorprev) {lambda *= 1.05;}  // error decreased, increase step
            if (lambda > 2) {lambda = 2;}
        }
        errorprev = error;

        // -- Diagnostic print out
        if(showText)
        {
            cout << "\t\tlambda = " << lambda << endl;
        }

        // Compute new shot: shotNew = shotOld - inv(DF)*F
        tempGuess[0] = shotGuess.at(k).at(0) - ( (lambda * (DF[3]*f1 - DF[1]*f2) ) / (DF[0]*DF[3] -
DF[1]*DF[2]) );
        tempGuess[1] = shotGuess.at(k).at(1) - ( (lambda * (DF[0]*f2 - DF[2]*f1) ) / (DF[0]*DF[3] -
DF[1]*DF[2]) );
```

```
        shotGuess.insert(shotGuess.end(), tempGuess);

        // Increment shot counter
        k++;

        // -- Diagnostic print out
        if(showText)
        {
            cout << "\t\t-- updated guess = [ " << shotGuess.at(k).at(0) << "\t" << shotGuess.at(k).at(1) <<
" ] "<< endl;
        }

        // Simulate new guess
        showState = false;
        x0[0] = 0.00;                   // time
        x0[1] = shotGuess.at(k).at(0);  // X
        x0[2] = shotGuess.at(k).at(1);  // Y
        x0[3] = zInitial;               // Z
        x0[4] = xDotInitial;            // Vx
        x0[5] = yDotInitial;            // Vy
        x0[6] = zDotInitial;            // Vz
        x0[7] = 0.00;                   // Trace(f)
        x0[8] = joint_ground;           // joint pdf

        GetSysWind(x0[1], x0[2], -x0[3]);
        x0[4] = syswind[3]*cos(syswind[4]);
        x0[5] = syswind[3]*sin(syswind[4]);

        IntegrateToZ(-zFinal);
        xFinal = x_impact[1];
        yFinal = x_impact[2];

        // -- Diagnostic print out
        if(showText)
        {
            cout << "\t\t-- final state based on new guess = [ " << xFinal << "\t" << yFinal << "\t" <<
zFinal << " ] "<< endl;
        }

        finalTolerance = sqrt( pow( refPathValAtTime[0]-xFinal , 2.0) +
                               pow( refPathValAtTime[1]-yFinal , 2.0) );

        // -- Diagnostic print out
        if(showText)
        {
            cout << "\t-- Completed Newton Method Iteration " << mNMCount << ", Final Tol = "
                 << finalTolerance << "\tUsing guess = [" << shotGuess.at(k).at(0)
                 << "\t" << shotGuess.at(k).at(1) << "]" << endl;
        }

        showState = false;

        // -- Pause program execution after each new guess so we can examine output
        if(showText)
        {
            cin.get();
        }

    } // end Modified Newton Method while

    return mNMCount;
}
```

```
void DynamicSystem::FuncRefPath(double z)
{
    // Set initial vales for reference path
    double zi = -1000.0;
    double xi = desiredImpactX;
    double yi = desiredImpactY;

    // Set final values for reference path
    double zf = 0.0;
    double xf = desiredImpactX;
    double yf = desiredImpactY;

    // Range around initial x and initial y values
    double a = 5000.0;

    // Range around final x and final y values
    double b = 50.0;

    // Compute x reference path and upper and lower bounds for the given z value
    double phi1 = ((xf-xi)/(zf-zi))*z + xi;
    double bound1Upper = (( b-a )/(zf-zi))*z + (xi+b);
    double bound1Lower = (( a-b )/(zf-zi))*z + (xi-b);

    // Compute y reference path and upper and lower bounds for the given z value
    double phi2 = ((yf-yi)/(zf-zi))*z + yi;
    double bound2Upper = (( b-a )/(zf-zi))*z + (yi+b);
    double bound2Lower = (( a-b )/(zf-zi))*z + (yi-b);

    refPathValAtTime[0] = phi1;
    refPathValAtTime[1] = phi2;

    boundsValAtTime[0] = bound1Upper;
    boundsValAtTime[1] = bound1Lower;
    boundsValAtTime[2] = bound2Upper;
    boundsValAtTime[3] = bound2Lower;
}
```

```
void DynamicSystem::DoRK4(double currentTime, state currentState)
{
    //adsf
    state k1, k2, k3, k4, tempState;
    double h = 0.1;

    tempState.assign(9, 0.00);

    // cout << "Trying FuncODE" << endl;
    k1 = FuncODE(currentTime, currentState);
    // cout << "Executed k1" << endl;

    tempState[0] = currentState[0] + (h/2)*k1[0];
    tempState[1] = currentState[1] + (h/2)*k1[1];
    tempState[2] = currentState[2] + (h/2)*k1[2];
    tempState[3] = currentState[3] + (h/2)*k1[3];
    tempState[4] = currentState[4] + (h/2)*k1[4];
    tempState[5] = currentState[5] + (h/2)*k1[5];
    tempState[6] = currentState[6] + (h/2)*k1[6];
    tempState[7] = currentState[7] + (h/2)*k1[7];
    tempState[8] = currentState[8] + (h/2)*k1[8];
    k2 = FuncODE(currentTime + (h/2), tempState);
    // cout << "Executed k2" << endl;

    tempState[0] = currentState[0] + (h/2)*k2[0];
    tempState[1] = currentState[1] + (h/2)*k2[1];
    tempState[2] = currentState[2] + (h/2)*k2[2];
    tempState[3] = currentState[3] + (h/2)*k2[3];
    tempState[4] = currentState[4] + (h/2)*k2[4];
    tempState[5] = currentState[5] + (h/2)*k2[5];
    tempState[6] = currentState[6] + (h/2)*k2[6];
    tempState[7] = currentState[7] + (h/2)*k2[7];
    tempState[8] = currentState[8] + (h/2)*k2[8];
    k3 = FuncODE(currentTime + (h/2), tempState);
    // cout << "Executed k3" << endl;

    tempState[0] = currentState[0] + h*k3[0];
    tempState[1] = currentState[1] + h*k3[1];
    tempState[2] = currentState[2] + h*k3[2];
    tempState[3] = currentState[3] + h*k3[3];
    tempState[4] = currentState[4] + h*k3[4];
    tempState[5] = currentState[5] + h*k3[5];
    tempState[6] = currentState[6] + h*k3[6];
    tempState[7] = currentState[7] + h*k3[7];
    tempState[8] = currentState[8] + h*k3[8];
    k4 = FuncODE(currentTime + h, tempState);
    // cout << "Executed k4" << endl;

    // cout << "k1 = " << k1[1] << "\t" << k1[2] << "\t" << k1[3]
    //                 << "\t" << k1[4] << "\t" << k1[5] << "\t" << k1[6] << "\t"<< endl;

    nextState[0] = currentState[0] + (h/6)*(k1[0] + 2*k2[0] + 2*k3[0] + k4[0]);
    nextState[1] = currentState[1] + (h/6)*(k1[1] + 2*k2[1] + 2*k3[1] + k4[1]);
    // cout << "x_next = " << nextState[1] << " = " << currentState[1] << " + "
    //                     << (h/6) << " * (" << k1[1] << " + " << 2*k2[1] << " + "
    //                         << 2*k3[1] << " + " << k4[1] << ")" << endl;
    nextState[2] = currentState[2] + (h/6)*(k1[2] + 2*k2[2] + 2*k3[2] + k4[2]);
    // cout << "y_next = " << nextState[2] << " = " << currentState[2] << " + "
    //                     << (h/6) << " * (" << k1[2] << " + " << 2*k2[2] << " + "
    //                         << 2*k3[2] << " + " << k4[2] << ")" << endl;
    nextState[3] = currentState[3] + (h/6)*(k1[3] + 2*k2[3] + 2*k3[3] + k4[3]);
    nextState[4] = currentState[4] + (h/6)*(k1[4] + 2*k2[4] + 2*k3[4] + k4[4]);
    nextState[5] = currentState[5] + (h/6)*(k1[5] + 2*k2[5] + 2*k3[5] + k4[5]);
    nextState[6] = currentState[6] + (h/6)*(k1[6] + 2*k2[6] + 2*k3[6] + k4[6]);
    nextState[7] = currentState[7] + (h/6)*(k1[7] + 2*k2[7] + 2*k3[7] + k4[7]);
    nextState[8] = currentState[8] + (h/6)*(k1[8] + 2*k2[8] + 2*k3[8] + k4[8]);

    nextTime = currentTime + h;

    return;
}
```

```
state DynamicSystem::FuncODE(double t, state x)
{
    // Main parachute equations of motion
    //      x[0] = t
    //      x[1] = x
    //      x[2] = y
    //      x[3] = z
    //      x[4] = xdot
    //      x[5] = ydot
    //      x[6] = zdot
    //      x[7] = tr(J)
    //      x[8] = phi

    // get wind at current location
    GetSysWind(x[1], x[2], -x[3]);

    // calculate density at current altitude
    double rho;
    rho = 1.22566578494891 * pow(1.00000000-0.0000225696709*(-x[3]),4.258);

    // calculate apparent mass
    double m_app;
    m_app = (4/3) * pow(R, 3) * PI * 0.25 * rho;

    // relative velocities with wind
    double xdot_rel, ydot_rel, zdot_rel, Vtotal;
    xdot_rel = x[4] - syswind[3] * cos(syswind[4]);
    ydot_rel = x[5] - syswind[3] * sin(syswind[4]);
    zdot_rel = x[6] - syswind[2];
    Vtotal = sqrt(xdot_rel * xdot_rel + ydot_rel * ydot_rel + zdot_rel * zdot_rel);

    // directional drag
    double Dx, Dy, Dz;
    Dx = -(0.5 * rho * xdot_rel * Vtotal * Cd * S);
    Dy = -(0.5 * rho * ydot_rel * Vtotal * Cd * S);
    Dz = -(0.5 * rho * zdot_rel * Vtotal * Cd * S);

    // incremental state
    dxdt[0] = 1.00;
    dxdt[1] = x[4];
    dxdt[2] = x[5];
    dxdt[3] = x[6];
    dxdt[4] = Dx / (m + m_app);
    dxdt[5] = Dy / (m + m_app);
    dxdt[6] = (Dz + m*g) / (m + m_app);
    dxdt[7] = 0.00;
    dxdt[8] = 0.00;

    return dxdt;
}
```

APPENDIX B

DYNAMIC SIMULATION PARAMETERS

The model used in the Results section is based on the descent dynamics of a 4.57 m diameter ring-slot drogue and a G-12E main parachute. Table B1 provides the parameters and uncertainty distributions used in these dynamic simulations.

Table B1: Simulation parameter and uncertainty distribution values

Parameter	Description	Value
m	Package Mass	1,048 kg
$\bar{C}_d$	Main Parachute Nominal Drag Coefficient	1.0487
R	Main Parachute Radius	9.75 m
$C_{d,dr}$	Drogue Parachute Drag Coefficient	0.6
R_{dr}	Drogue Parachute Radius	2.3 m
k_a	Apparent Mass Coefficient	0.25
σ_c	Standard Deviation in Main Parachute Drag Coefficient	0.0422
σ_w	Standard Deviation in Wind Magnitude Scaling Parameter	0.05
σ_ψ	Standard Deviation in Wind Direction	0.0873 rad
$z_{t,min}$	Minimum Allowable Transition Altitude	240 m
$z_{t,max}$	Maximum Allowable Transition Altitude	1,000 m

REFERENCES

[1] Benney, R., Henry, M., Kristen, L., Meloni, A., "DoD New JPADS Programs and NATO Activities," AIAA Paper 2009-2952, May 2009.

[2] Hattis, P., Fill, T., Rubenstein, D., Wright, R., Benney, R., "An Advanced On-Board Airdrop Planner to Facilitate Precision Payload Delivery," AIAA Paper 2000-4307, August 2000.

[3] Wright, R., Benney, R., McHugh, J., "An On-Board 4D Atmospheric Modeling System to Support Precision Airdrop," AIAA Paper 2005-7070, September 2005.

[4] Campbell, D., Fill, T., Hattis, P., Tavan, S., "An On-Board Mission Planning System to Facilitate Precision Airdrop," AIAA Infotech Conference, 26-29 September 2005, Arlington, VA, AIAA Paper 2005-7071.

[5] Munnell, C., "Company Developing Wind Measurement Technology to Improve Cargo Airdrops," National Defense Magazine, September 2014.

[6] Cacan, M., Scheuermann, E., Ward, M., Costello, M., and Slegers, N., "Autonomous Airdrop Systems Employing Ground Wind Measurements for Improved Landing Accuracy," IEEE/ASME Transactions on Mechatronics, Vol. 20, No. 6, 2015, pp. 3060-3070.

[7] Potvin, J., Charles, R., and Desbrais, K., "Comparative DSSA Study of Payload-Container Dynamics Prior to, During and After Parachute Inflation," AIAA Paper 2007-2564, May 2007.

[8] Cuthbert, P. A., "A Software Simulation of Cargo Drop Tests," AIAA Paper 2003-2132, May 2003.

[9] VanderMey, J., Doman, D., Gerlach, A., "Release Point Determination and Dispersion Reduction for Ballistic Airdrops," Journal of Guidance, Control, and Dynamics, Vol. 38, No. 11, 2015, pp. 2227-2235.

[10] Gerlach, A., Manyam, S., and Doman, D., "Precision Airdrop Transition Altitude Optimization via the One-in-a-Set Traveling Salesman Problem," 2016 American Control Conference, Boston, MA, 6-8 July 2016.

[11] Sobczyk, K., Stochastic Differential Equations, Kluwer Academic Publishers, Dordrecht, Germany, 1991.

[12] Klein, B., Rogers, J., "A Probabilistic Approach to Unguided Airdrop," AIAA Paper 2015-2119, April, 2015.

[13] Etkin, B., Reid, L. D., Dynamics of Flight: Stability and Control, John Wiley and Sons, Hoboken, NJ, 1996, pp. 364-367

[14] Department of Defense Handbook, *Flying Qualities of Piloted Aircraft*, MIL-HDBK-1797, December 1997.

[15] Kumar, M., Chakravorty, S., and Junkins, J., "On the Curse of Dimensionality in the Fokker-Planck Equation," Advances in the Astronautical Sciences, Vol. 135, 2009, pp. 1781-1800.

[16] Kumar, M., Chakravorty, S., Singla, P., and Junkins, J., "The Partition of Unity Finite Element Approach with HP-Refinement for the Stationary Fokker-Planck Equation," Journal of Sound and Vibration, Vol. 327, 2009, pp. 144-162.

[17] Petry, G., "Airdrop Error Analysis," Air Force Systems Command Technical Report ASD-TR-75-8, Wright Patterson Air Force Base, Ohio, June 1975.

[18] Guglieri, G., "Parachute-Payload System Flight Dynamics and Trajectory Simulation," International Journal of Aerospace Engineering, Vol. 2012, 2012.

[19] Dunn, W. L., Shultis, J., Exploring Monte Carlo Methods, Elsevier Science and Technology, 2012, pp. 133-166.

[20] Scott, D., Multivariate Density Estimation: Theory, Visualization, and Practice, John Wiley & Sons, New York, pp. 1992.

[21] Halder, A., Bhattacharya, R., "Dispersion Analysis in Hypersonic Flight During Planetary Entry Using Stochastic Liouville Equation," Journal of Guidance, Control, and Dynamics, Vol. 34, No. 2, March-April 2011, pp. 459-474.

[22] Risken, H., The Fokker-Planck Equation: Methods of Solution and Applications, Springer-Verlag, New York, 1996, pp. 63-91.

[23] Leonard, A., Klein, B., Jumonville, C., Rogers, J., Gerlach, A., and Doman, D., "A Probabilistic Algorithm for Ballistic Parachute Transition Altitude Optimization", (Submitted for Publication to Journal of Guidance, Control, and Dynamics)

[24] Holsapple, R., Venkataraman, R., and Doman, D., "A New, Fast Numerical Method for Solving Two-Point Boundary Value Problems", http://www.math.ttu.edu/~rvenkata/papers/jgcdnote.pdf (Accessed June 1, 2016).

[25] Holsapple, R., Venkataraman, R., and Doman, D., "A Modified Simple Shooting Method for Solving Two Point Boundary Value Problems," Proceedings of the IEEE Aerospace Conference, Big Sky, MT, Vol. 6, IEEE, New York, NY 10016-5997, March 2003, pp. 2783–2790.

[26] Gerlach, A., Doman, D., "Analytical Solution for Optimal Drogue-to-Main Parachute Transition Altitude for Precision Ballistic Airdrops," Journal of Guidance, Control, and Dynamics, Submitted for Publication, 2015.

[27] "Mapping Toolbox User's Guide", http://www.mathworks.com/help/pdf_doc/map/map_ug.pdf (Accessed June 1, 2016).

[28] Von Neumann, J., "Various Techniques Used in Connection With Random Digits," National Bureau of Standards Applied Mathematics Serial, No. 12, 1951, pp. 36-38.

[29] National Center for Atmospheric Research, "A Description of the Advanced Research WRF Version 3," NCAR Technical Note TN-475+STR, Boulder, CO, June 2008.

[30] Fields, T., LaCombe, J., Wang, E., "Autonomous Guidance of a Circular Parachute Using Descent Rate Control," *Journal of Guidance, Control, and Dynamics*, Vol. 35, No. 4, 2012, pp. 1367-1370.