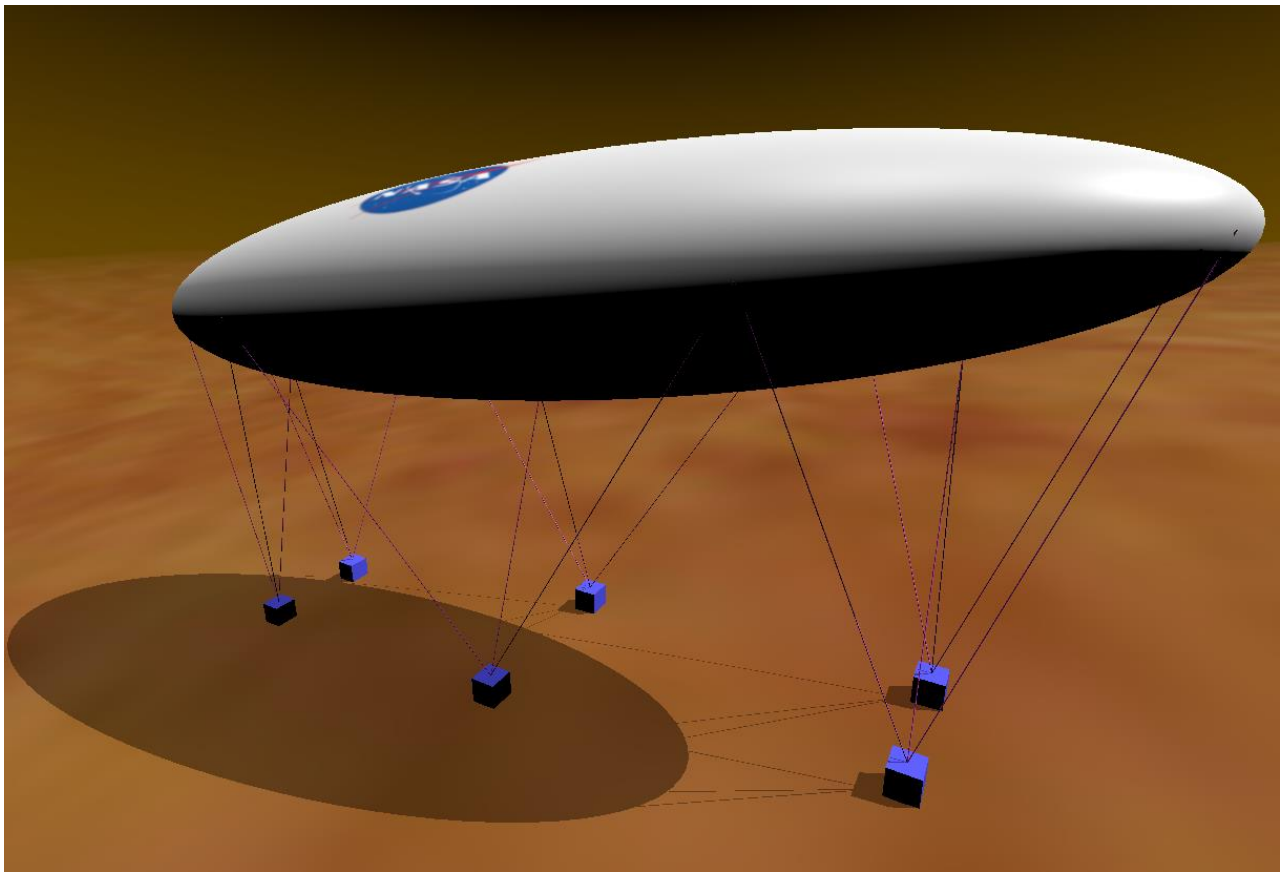
BALLET

Balloon Locomotion for Extreme Terrain

NASA Innovative Advanced Concepts (NIAC) Phase I Final Report



Principal Investigator: Hari Nayar
Co-Investigators: Michael Pauken
Morgan Cable
Research Technologist: Michael Hans

Date: March 4, 2019

Jet Propulsion Laboratory
California Institute of Technology

Table of Contents

1. Introduction 3
   1.1. Concept Description 3
   1.2. Motivation 4
   1.3. Phase 1 Study Overview 4
2. Science Objectives 5
   2.1. Mars Recurring Slope Lineae 5
   2.2. Titan Shorelines 6
   2.3. Titan Dunes 7
   2.4. Titan Cryovolcanic regions 8
3. Mission Formulation 11
   3.1. Spacecraft and Deployment 11
   3.2. Mechanical Design and Materials 17
4. Concept Evaluation 18
   4.1. Analyses 18
   4.2. Results 23
      4.2.1. Titan 23
      4.2.2. Mars 29
      4.2.3. Earth 34
5. Locomotion 38
   5.1. Obstacle Avoidance Motion Planning 39
   5.2. Path Planning and Foot Trajectory Control 39
   5.3. BALLET Model and 3D Visualization 42
6. Conclusions 44

Acknowledgements 47

References 47

Appendix A: Analysis Software Listing A.1

Appendix B: OpenFOAM Aerodynamics Software Listing B.1

Appendix C: Locomotion and Visualization Software Listing C.1

1. Introduction

1.1. Concept Description

BALLET is a limbed robot that uses a balloon for its structure and has its payload in its feet. Science and engineering sub-systems on BALLET including instruments, electronics, power and control systems, and energy storage are evenly distributed into six modular feet. Each foot is connected to the balloon using three cables (Figure 1) -- the minimum needed to control the foot position in 3-D. Cable lengths are controlled using three winches within each foot. Coordinated control of cable lengths places each foot at desired locations on the ground.

To locomote BALLET lifts one foot at a time, places it at a new location on the ground, then re-positions the balloon with respect to the new feet positions by re-adjusting all cable lengths. This procedure is repeated in sequence for the other feet. The balloon is small relative to the total payload mass because the buoyancy required is only needed to lift one foot, i.e. one-sixth of the total payload. BALLET is stable because it is effectively anchored to the ground with its CG close to ground level. An additional advantage BALLET offers is

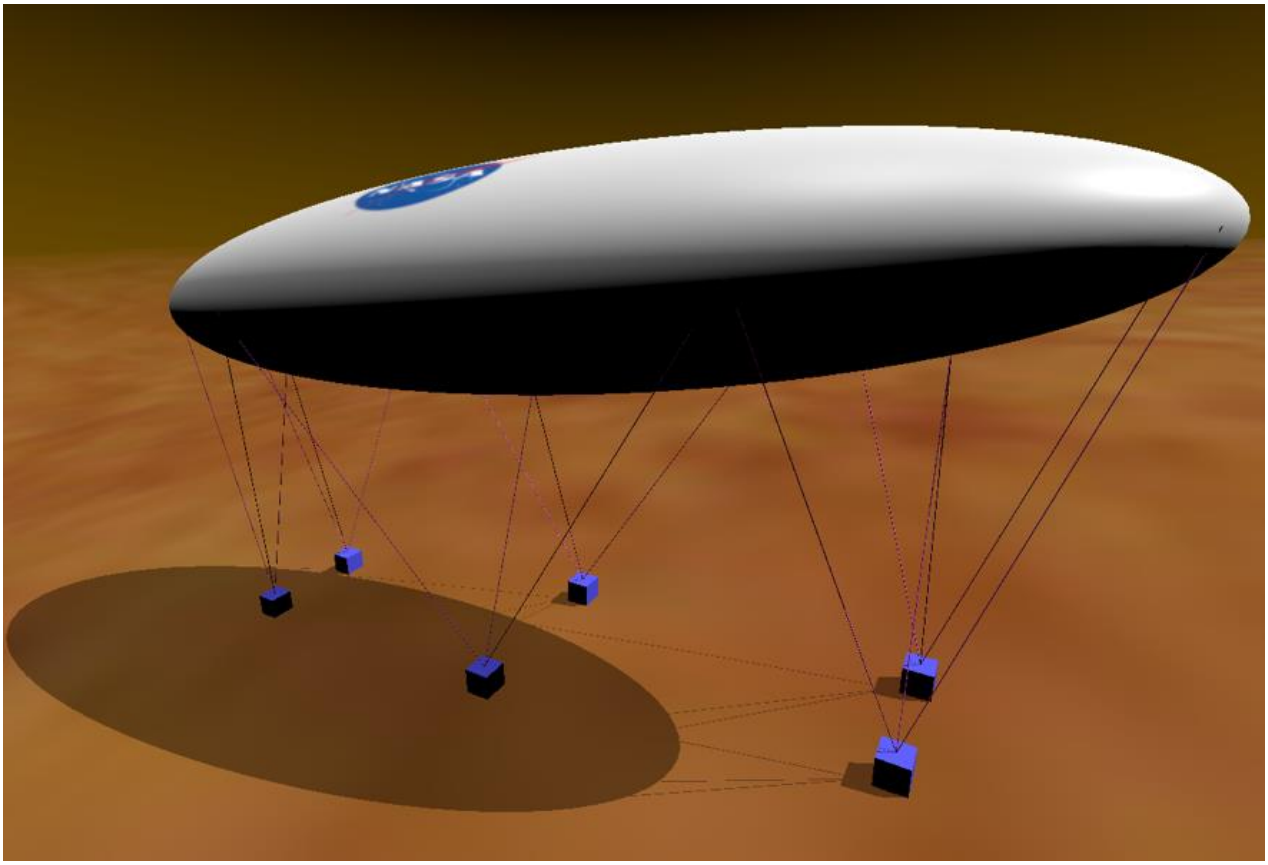


Figure 1 Visualization of the BALLET concept with balloon and 6 suspended payloads serving as feet. Each payload is connected to the balloon by 3 cables that allow positioning of the payload with respect to the balloon.

the potential to add a secondary mission by jettisoning five of its payloads and performing atmospheric exploration as a conventional balloon. While the physics of BALLET will apply on Venus, the environmental conditions and available component technology limit our consideration to Mars and Titan.

A thorough review of prior published research on surface mobility systems and on balloons for Earth and planetary exploration applications was performed to gather background information on BALLET. Survey papers, for example, on mobility [Seeni 2010] and balloons [Cutts 1995, Elfes 2003, Elfes 2008] do not describe this new hybrid concept. A range of options have been considered [Backes 2008, Nesnas 2012, Seeni 2010, Wilcox 2007] for access to rugged terrain on planetary surfaces. Some unusual surface mobility concepts with light-weight or buoyant components have been reported. For example, rovers with inflatable wheels and wind-driven tumbleweed rovers [Hajos 2005]. Underwater walking robots [Schue 1993] have been proposed and developed that use the physics principals of BALLET although none put their payload in their feet.

1.2. Motivation

Safe and stable in-situ access to steep and rugged terrain has the potential for enormous science value in understanding geology, surface and subsurface chemistry, hydrology and potentially prebiotic processes on Titan and Mars. Exploration of these destinations are prioritized in the 2013 Decadal Survey [Space Studies Board, 2013].

Wheeled vehicles are used for surface exploration missions because they are relatively simple and highly efficient in traversing over benign terrain. Operational constraints for Mars (and likely for other planetary surfaces) limit their traverse over obstacles to less than the wheel height and slopes less than 20o. As a consequence, sites chosen for Mars' missions trade-off science against mobility. Conventional legged vehicles handle more difficult terrain but with greater mass and complexity, and reduced stability and safety.

1.3. Phase 1 Study Overview

This report documents the work performed in our investigation into the BALLET concept. We focused on four areas in this Phase I effort. They were 1) identifying the science targets and objectives with the corresponding requisite instrumentation and operational capabilities that could be achieved with a BALLET mission, 2) developing an architecture for the deployment and operation of this concept for a future mission to a planetary body, 3) analyzing a parametric physical model of BALLET under the environmental conditions of Mars, Titan and Earth to determine its feasibility, and 4) developing and demonstrating

coordinated control of the BALLET mobility system to enable locomotion over rugged terrain. The results of our investigations in these focus areas are documented in the following sections. A paper summarizing the preliminary results from this study has been accepted for publication and presentation at the 2019 IEEE Aerospace Conference [Nayar, 2019].

2. Science Objectives

2.1. Mars Recurring Slope Lineae

Recurring slope lineae (RSL) are one of the primary targets for understanding the hydrologic cycle and possibility of extant life on Mars. These features are narrow, dark markings on steep slopes that appear and incrementally lengthen during warm seasons (Figure 2). RSL fade in cooler seasons and recur over multiple Mars years. They are associated with hydrated salts (Ojha et al. 2015) and are believed to be formed by intermittent flow of briny water (McEwen et al. 2014). These briny environments could be host to life such as halophilic microorganisms (Oren et al. 2014). However, access to these tantalizing features is a challenge as they only occur on slopes of 25-40°.

Unambiguous life detection in RSL would be significantly challenging without *in situ* sampling. Several techniques exist that can discern biosignatures (amino acids, fatty acids,

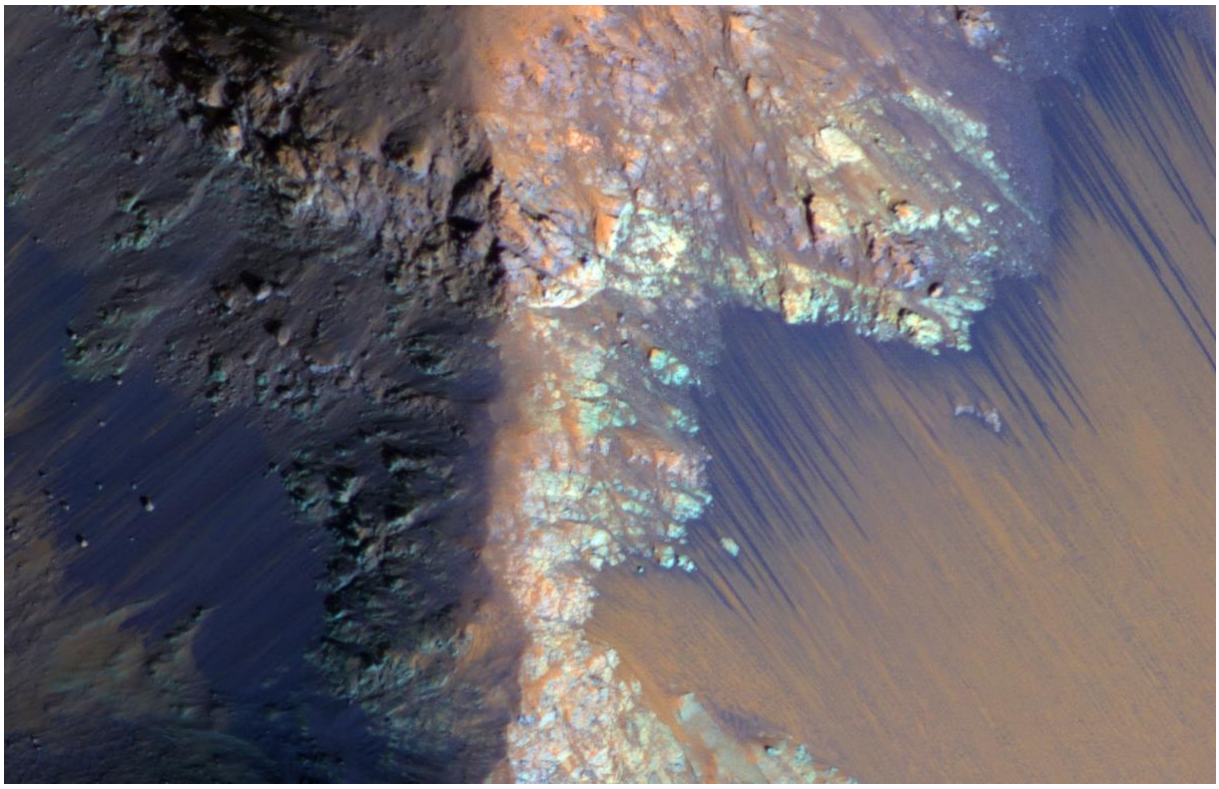


Figure 2 Recurring slope lineae (RSL) on the steep slopes of ancient bedrock in Coprates Chasma.

nucleobases, etc.) *in situ*, even in the presence of significant salt concentrations, and techniques such as sublimation or supercritical water extraction can be used to separate biomolecules from salty matrices for analysis downstream. Sampling a transect and/or a depth profile would strengthen the credibility of any positive biosignature detection. A payload suite containing biosignature detection instrumentation and probes to monitor soil properties (Table 1) would provide life detection capability placed in context important for interpretation of those measurements.

2.2. Titan Shorelines

Titan, the largest moon of Saturn, has many challenging regions that could be accessed via the BALLET platform. Titan is the only other body aside from Earth with standing liquid on its surface. However, due to its low surface temperature (94 K), this liquid is not water but hydrocarbons – primarily methane and ethane, which pool in lakes at the

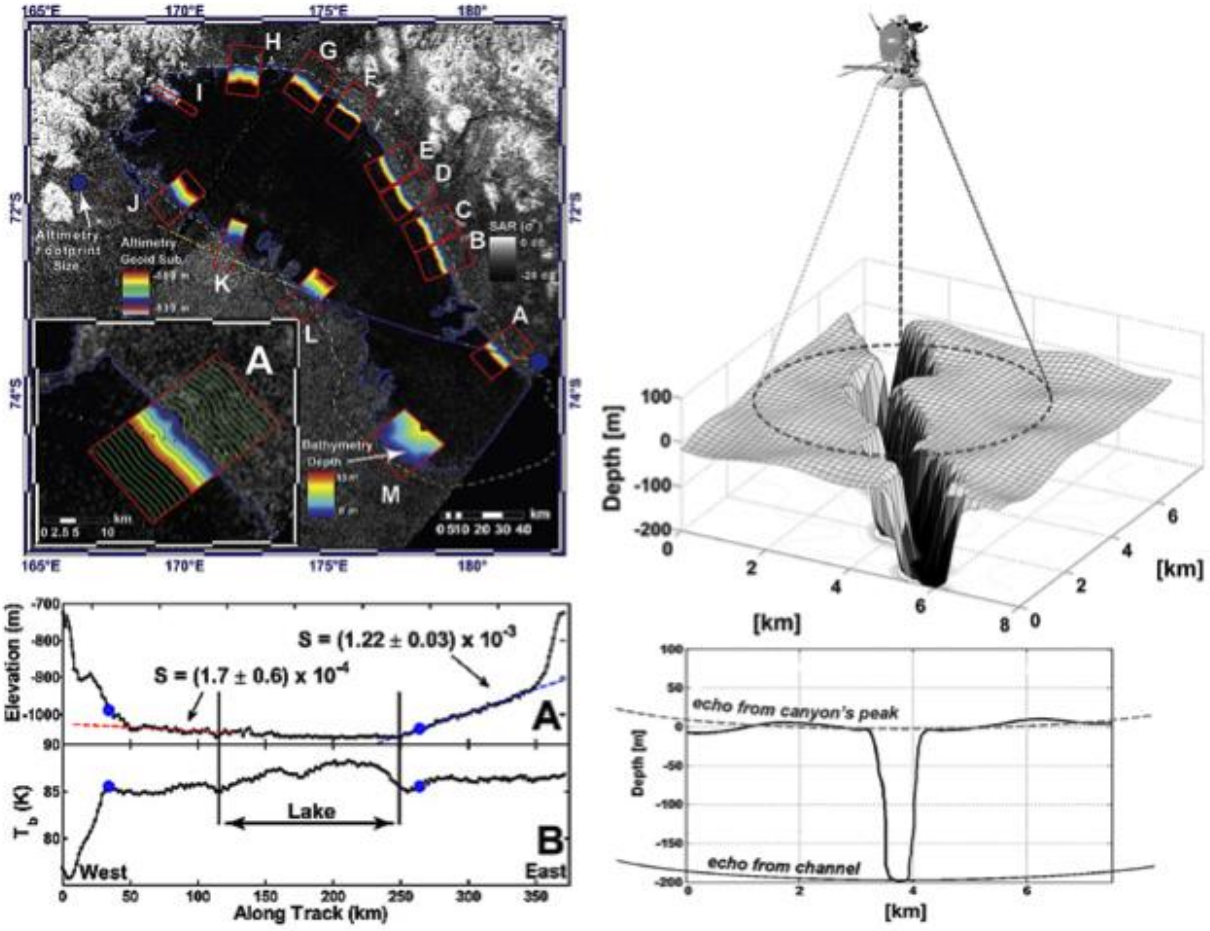


Figure 3 (Left) Bathymetric profile of Ontario Lacus, a lake in the south polar region of Titan, from Hayes et al. 2010. (Right) Cassini radar altimeter data for Vid Flumina, a methane-filled canyon in the northern hemisphere flowing into Ligeia Mare, Titan's second-largest sea, from Poggiali et al. 2016. Both have edges too steep for a traditional rover to access.

poles (Stofan et al. 2007). Due to the absorption and scattering of methane and haze particles in Titan's atmosphere, respectively, determination of surface composition by remote sensing is extremely challenging. Observations through the methane windows in the NIR only allows rough slopes to be estimated; no spectral assignments can be made to identify species. In situ sampling, or spectroscopy at the surface, avoids these issues. Further, in situ missions have much greater spatial resolution, and are able to discern trends invisible to orbit or flyby missions.

Recent work provides fairly rigorous constraints on the composition of the lake liquid (Mitchell et al. 2015); however, the composition of the evaporite region around existing lakes and of dry lakebeds (Cordier et al. 2013 and references therein) is still a mystery. Though many lake landers and submersibles have been proposed (Stofan et al. 2010, Oleson et al. 2015), it is questionable whether such a platform could navigate to safely sample the edge of the lake where the evaporite resides, especially considering that most of these depressions either have steep walls (Hayes et al. 2010, Poggiali et al. 2016) or are surrounded by topographically high areas on the order of 1 km over distances of 50-100 km (Lopes et al. 2007a) (see Figure 3). Any platform would certainly benefit from being able to move along the evaporite, as the composition likely changes with radial distance (less soluble species will precipitate first, and should reside in an outer ring around the lake, while more soluble species will precipitate last and be concentrated closer to the center). Several instruments (Table 1) would help with identification of key molecules and their chemical environments (co-crystal, clathrate, etc.).

2.3. Titan Dunes

The dunes in the equatorial region of Titan (Figure 4) are another primary target of exploration. These are found mainly within ±30° of the equator in dark regions (in the visible, NIR and radar), and cover approximately 20% of Titan's surface (Radebaugh et al. 2008, Lorenz and Radebaugh 2009). Though the fact that they are dark in most wavelengths suggests they are comprised of a significant proportion of organics, we still do not know the composition of these dunes, or how they formed or may be changing. The dunes appear to be approximately 100 m in height, with slopes ranging from steep (20:1 to 50:1) to shallow (200:1), though higher slopes could be present below the resolution of Cassini radar. We note that the steepest slope attempted by any rover on Mars to date is 32°, and slippage was so great in this case that the course was abandoned (Webster et al. 2016). Slopes greater than 20° are considered steep for rover traversal; this becomes significantly more challenging on terrains with loose material, as unconsolidated dune inclines most likely would have.

2.4. Titan Cryovolcanic regions

Several areas of Titan's surface, such as Sotra Patera and Hotei Regio (Figure 5), have features that have been identified as putative cryovolcanoes (Lopes et al. 2013). Cryovolcanism may be an important resurfacing process on Titan, and may also be a major contributor to atmospheric methane (Lopes et al. 2007b). Importantly, these regions may be the only places on Titan where material from the global, subsurface water ocean is being expressed on the surface. Any mission seeking to understand the habitability of this subsurface ocean would find areas of cryovolcanism very attractive sampling sites. As these regions exhibit some of the greatest elevation change on Titan's surface, only a mission architecture capable of traversing/sampling steep slopes can reach these areas to confirm their composition and origin.

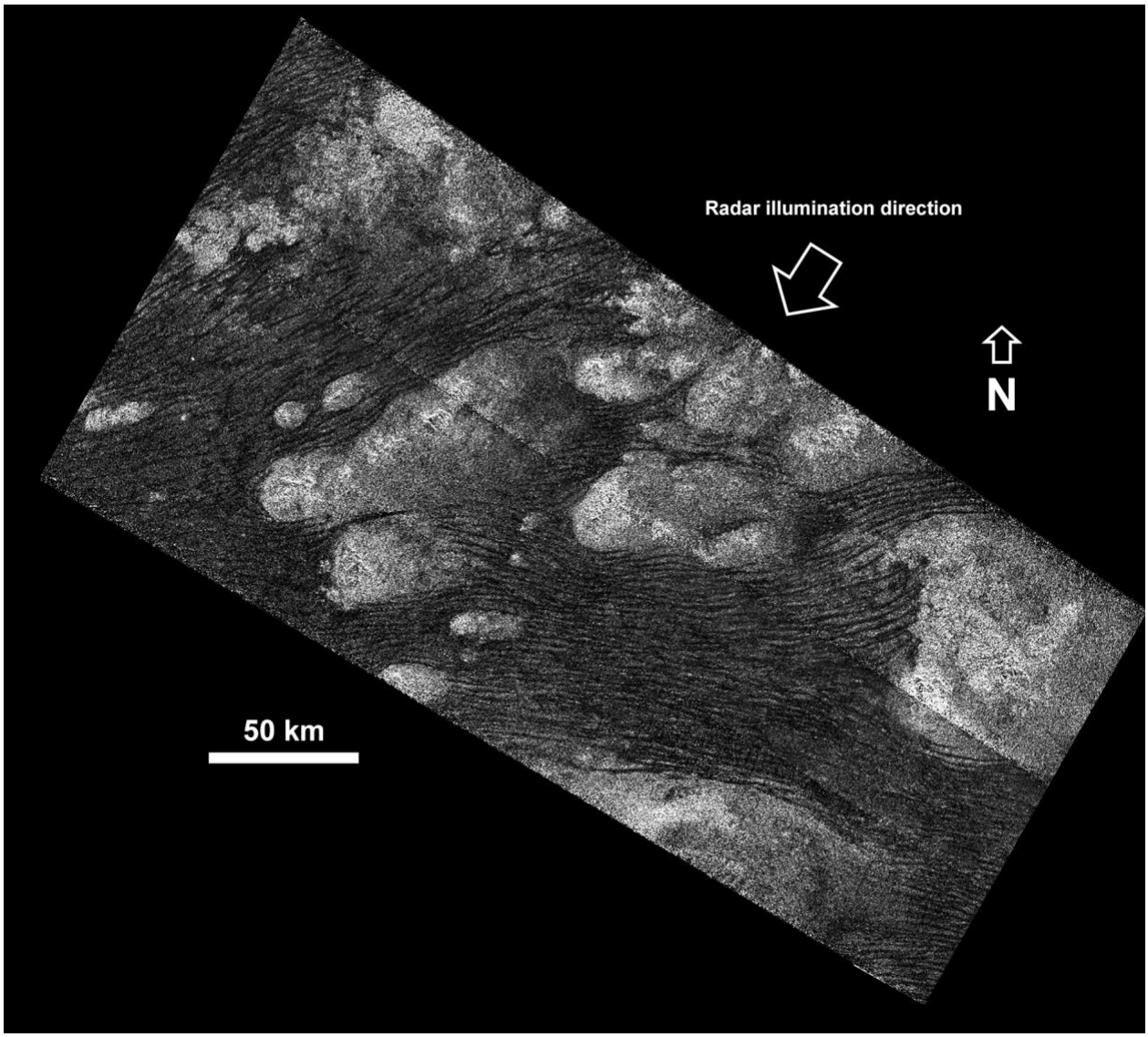


Figure 4 Cassini SAR image of dunes in Shangri-La, Titan. Image credit: NASA/JPL-Caltech/ASI.

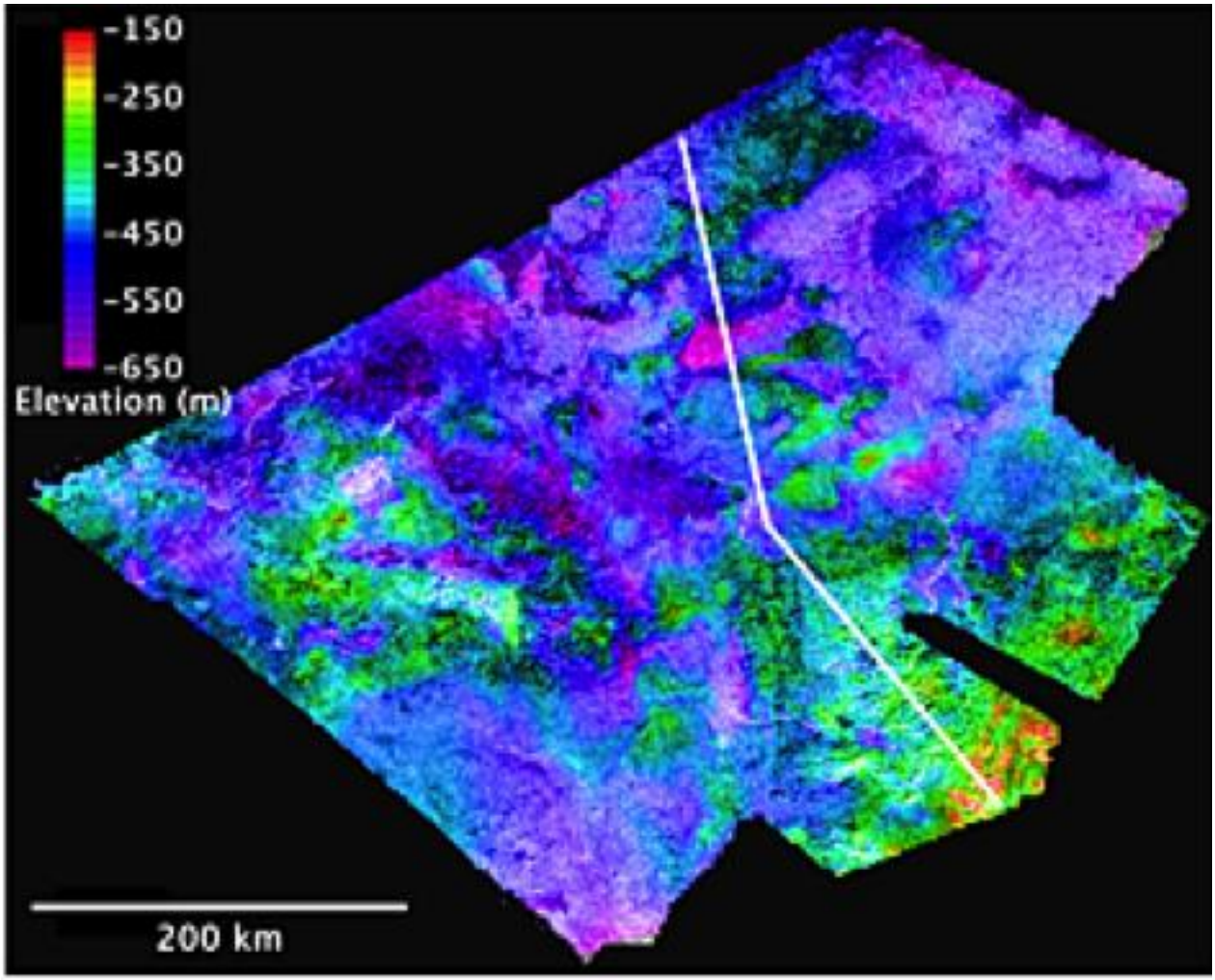


Figure 5 Digital elevation model (DEM) of Hotei Regio, an area of putative cryovolcanism on Titan. From Lopes et al. 2013.

Instrument	Description	Body	Mass	Power
High-resolution mass spec (Orbitrap)	Non-pyrolysis front end (liquid chromatography, MALDI, etc.) – prototypes are under development that may fit within the required mass envelope for BALLET.	Titan	10 kg	50 W
Microfluidics package	Labeling of key functional groups and biochemistry and cold separation with ethanol or a similar solvent. Prototypes at JPL and UC Berkeley/SSL fit well within mass/power/volume requirements.	Mars/Titan	4.0 kg	20 W
Raman microscope (SHERLOC)	Key molecular species on Titan form co-crystals and other structures which are uniquely identified with Raman spectroscopy. The microscope enables mapping of small images to determine grain composition (as opposed to bulk composition) and context. SHERLOC is being designed to fit on the arm of Mars2020, so this instrument should fit within mass/volume constraints.	Titan	4.5 kg	80 W
Seismic package	Geophone or seismometer with 3-axis arrival information. This could help detect cryovolcanic events or 'booming' of dunes.	Titan	1.2 kg	0.05 W
Gas chromatograph mass spectrometer	Needed to separate biomarkers (i.e., chiral amino acids, peptides, lipids) and enable identification of structural isomers (i.e., glycine and methyl carbamate) or branching in long carbon chains.	Mars	2.0 kg	16 W
Vis/NIR imaging spectrometer	To identify hydrated salts and areas where water is concentrated for in situ sampling.	Mars/Titan	3.7 kg	46 W
Environmental sensing (REMS)	Rover Environmental Monitoring Station instrument measures the thermal environment, ultraviolet irradiation and water cycling.	Mars/Titan	1.2 kg*	17 W
Activity/context camera (Mastcam)	Multi-spectral imaging local area for contextual setting.	Mars/Titan	1.3 kg*	13 W
Microscopic camera (RMI)	Remote microscopic imaging of selected site. The foot placement system would be used select site.	Mars/Titan	0.3 kg	<10 W
Near IR Spectrometer (MicrOmega)	Ultra-miniaturized spectral microscope for in situ analysis of samples.	Mars/Titan	1 kg	7 W
Di-electric & soil properties probe (SPARTTA)	Soil shear Properties Assessment, Resistance, Thermal, and Triboelectric Analysis multiTool for shallow subsurface measurements.	Mars/Titan	1 kg	5 W
Digital holographic microscope	Capable of distinguishing between particles and cells via density and motility.	Mars/Titan	10 kg	15 W
Wet Chemistry Laboratory (WCL)	Measurement of soil pH, eH and conductivity, along with ion-selective electrodes for key ions of interest (calcium, magnesium, etc.)	Mars	<10 kg	<15 W

*Table 1 Baseline instrument payload for a BALLET mission. *These instruments exceed the payload limit for a foot on Mars but future lighter-weight versions will likely be available in the timeframe for a BALLET mission.*

3. Mission Formulation

3.1. Spacecraft and Deployment

The BALLET robotic exploration platform could be adapted to investigate either Titan or Mars by making appropriate adaptions for each unique atmospheric environment. At Titan, BALLET could carry 5 kg in each of its six feet and a 45 kg radioisotope thermoelectric power generator (RTG). The Titan balloon envelope would be made from a laminate of polyester fabric and film and would have a volume of approximately 12 m^3. For Mars, BALLET could carry 1 kg in each foot and use solar power. An illustration of the BALLET vehicle concept on Mars is shown in Figure 6. The Mars balloon would be fabricated from a bi-laminate Mylar film and the envelope volume would be about 88 m^3. Laminated film materials are less susceptible to pin hole leaks than single layer films. The science payload and supporting systems for power, telecom, command and data handling would be divided among the feet that anchor the balloon to the surface.

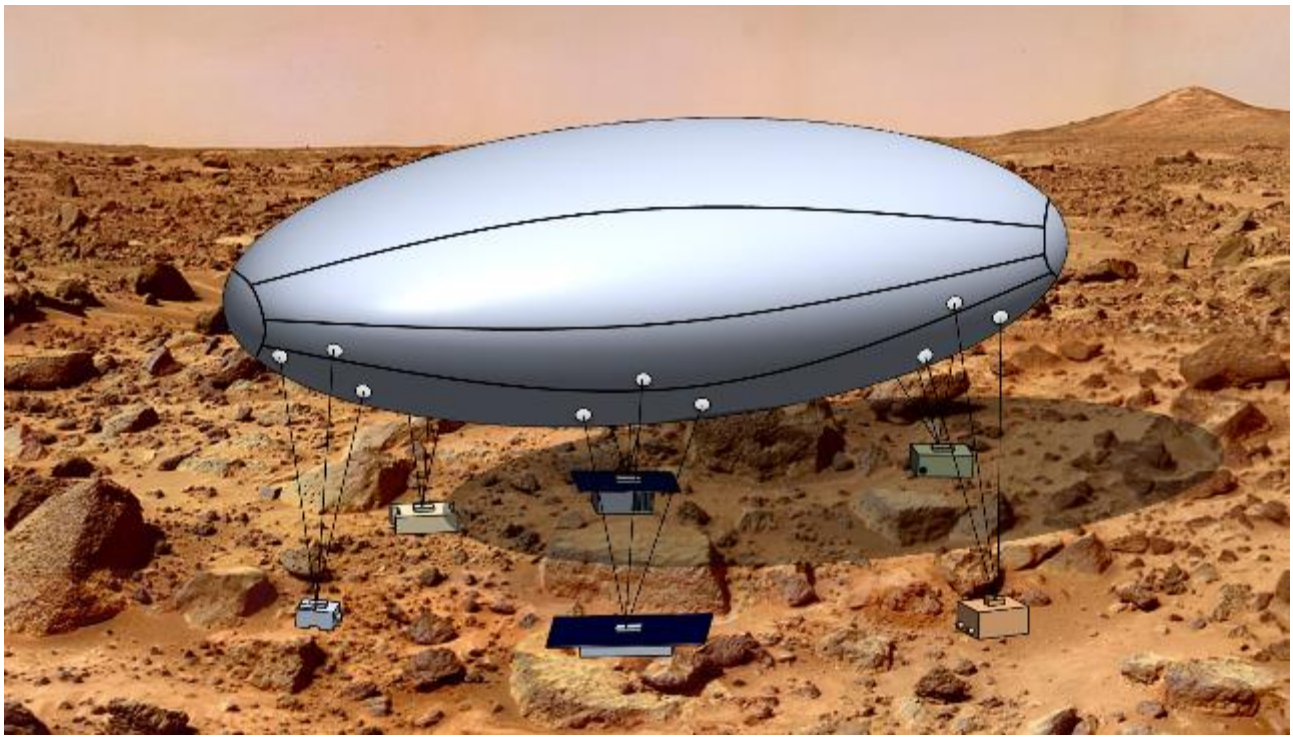


Figure 6 The BALLET vehicle floats above the Martian surface while the payload instruments anchor the balloon to the ground. (Background image courtesy NASA).

The BALLET vehicle would be packaged inside a nested flight system for delivery to Titan or Mars. The major flight system components include a carrier vehicle, atmospheric entry system, lander platform and the BALLET vehicle. The carrier vehicle depicted in Figure 7 would be powered with its own RTG system for Titan and would also serve as an orbiting

communication relay. As an orbiter, it could have its own science mission that could compliment the BALLET mission. For a Mars mission the carrier vehicle would use solar arrays for power and it is assumed communications can be accommodated with existing orbiting assets. Therefore, the Mars carrier vehicle would not serve any other purpose than to deliver BALLET to Mars. This is similar to a cruise stage used to deliver Mars rovers and landers.

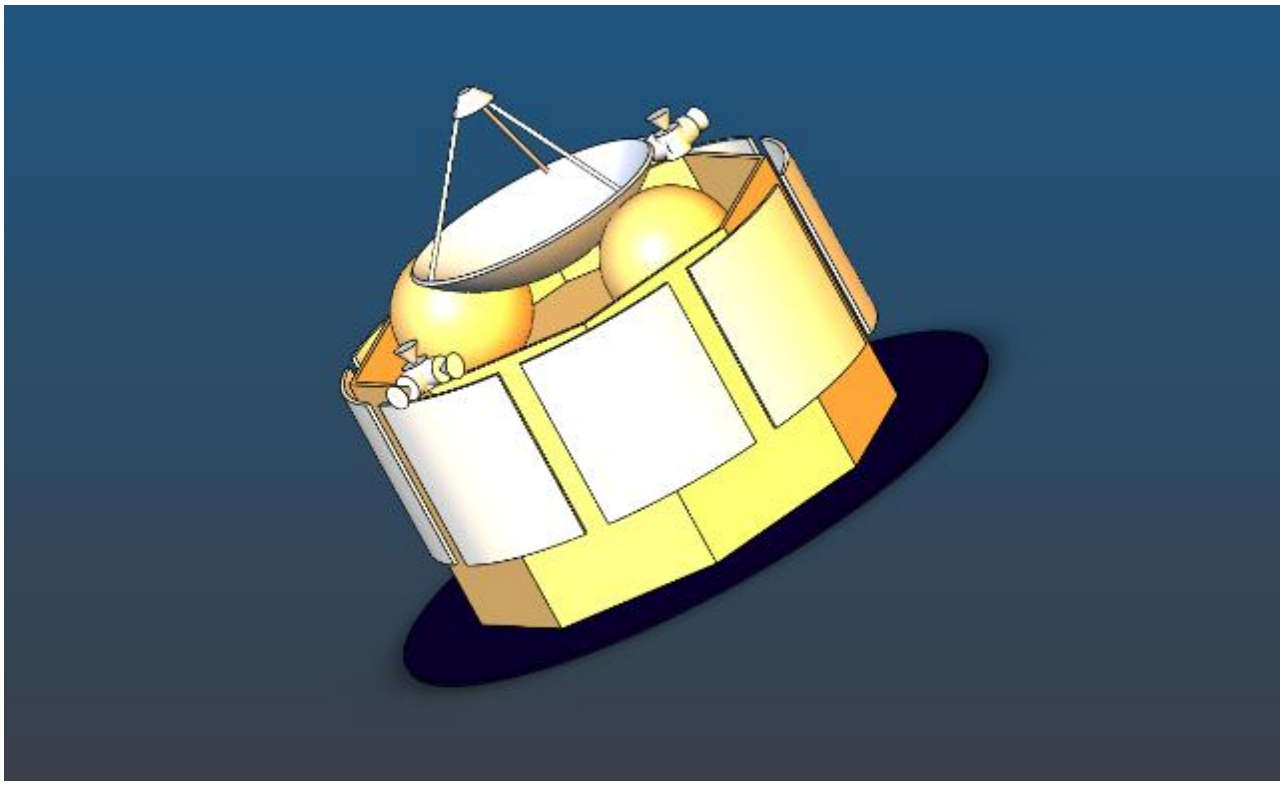


Figure 7. The BALLET vehicle is packaged within a Cruise Vehicle transport to Mars or Titan.

After launch, the cruise vehicle would separate from the launch vehicle and provide for all thermal, power and communication needs for BALLET. Health checks and software uploads would be typical interaction with the vehicle during cruise to either Mars or Titan. The cruise vehicle would have propulsion needed to provide trajectory correction maneuvers and spin capabilities for inertial guidance. The Titan cruise vehicle would also need to reject RTG waste heat from both the cruise and the BALLET vehicles. The Mars cruise vehicle does not need this capability.

As the spacecraft approaches Mars or Titan the atmospheric entry vehicle, as illustrated in Figure 8, would separate from the cruise vehicle. For a Mars mission, the cruise vehicle

would stop spinning and separate from the entry vehicle about 20 minutes prior to atmospheric entry. The cruise vehicle would then be diverted to a separate trajectory into the atmosphere to avoid collision with the entry vehicle. Ultimately the cruise vehicle would impact the Mars surface. For a Titan mission, the cruise vehicle would enter into an orbit around Titan and become an orbiter. The entry vehicle would be kicked off the orbiter which would track the progress of the entry into the atmosphere. The orbiter continues to circle Titan and perform relay functions for the BALLET vehicle during entry, deployment and mission operations.

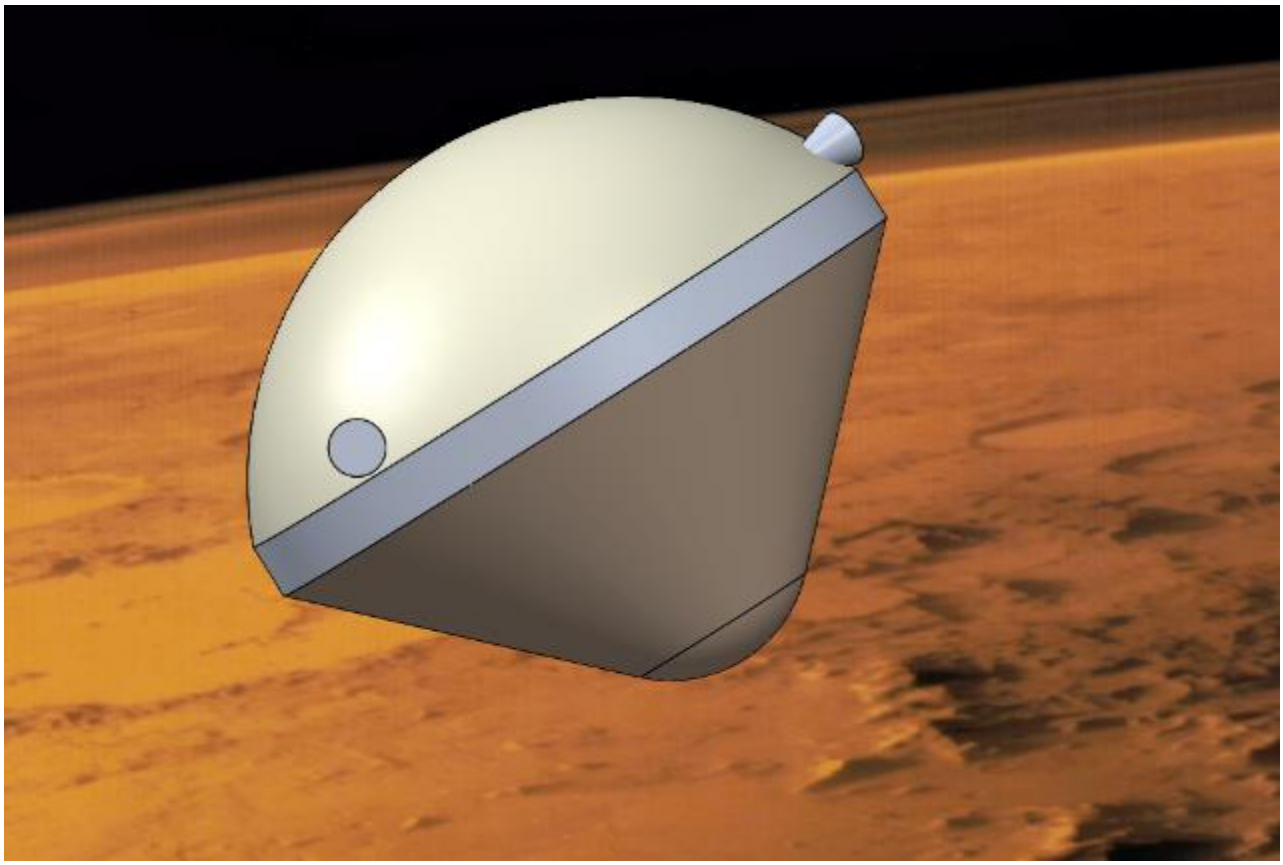


Figure 8 The BALLET vehicle is protected from atmospheric entry by an aeroshell. (Background image credit NASA Solar Systems Directorate).

The sequence of events between atmospheric entry and landing the BALLET vehicle is shown in Figure 9. Upon atmospheric entry, shown in panel A, the heat shield removes a significant amount of energy from vehicle slowing it down until a parachute can be deployed. The shape of the aeroshell and location of the center of gravity could be designed such that

it could be used as a lifting body and provide guided entry (panel B) to reduce landing ellipse error for the target destination. Backshell thrusters would be used to provide navigational guidance during entry. Deployment of the parachute uses a drogue chute first to pull out the main parachute (panel C). Once the parachute deployment has stabilized, the heat shield would be jettisoned (panel D) and fall to the ground and out of the way of the vehicle. As the BALLET lander approaches the surface (panel E), the lander legs would be deployed and it would be dropped from the backshell. Then descent thrusters and guidance navigation would slow the lander until touchdown on the surface. Lander rocket thrusters would also perform a lateral maneuver to avoid collision between the falling backshell/parachute and the lander vehicle during descent.

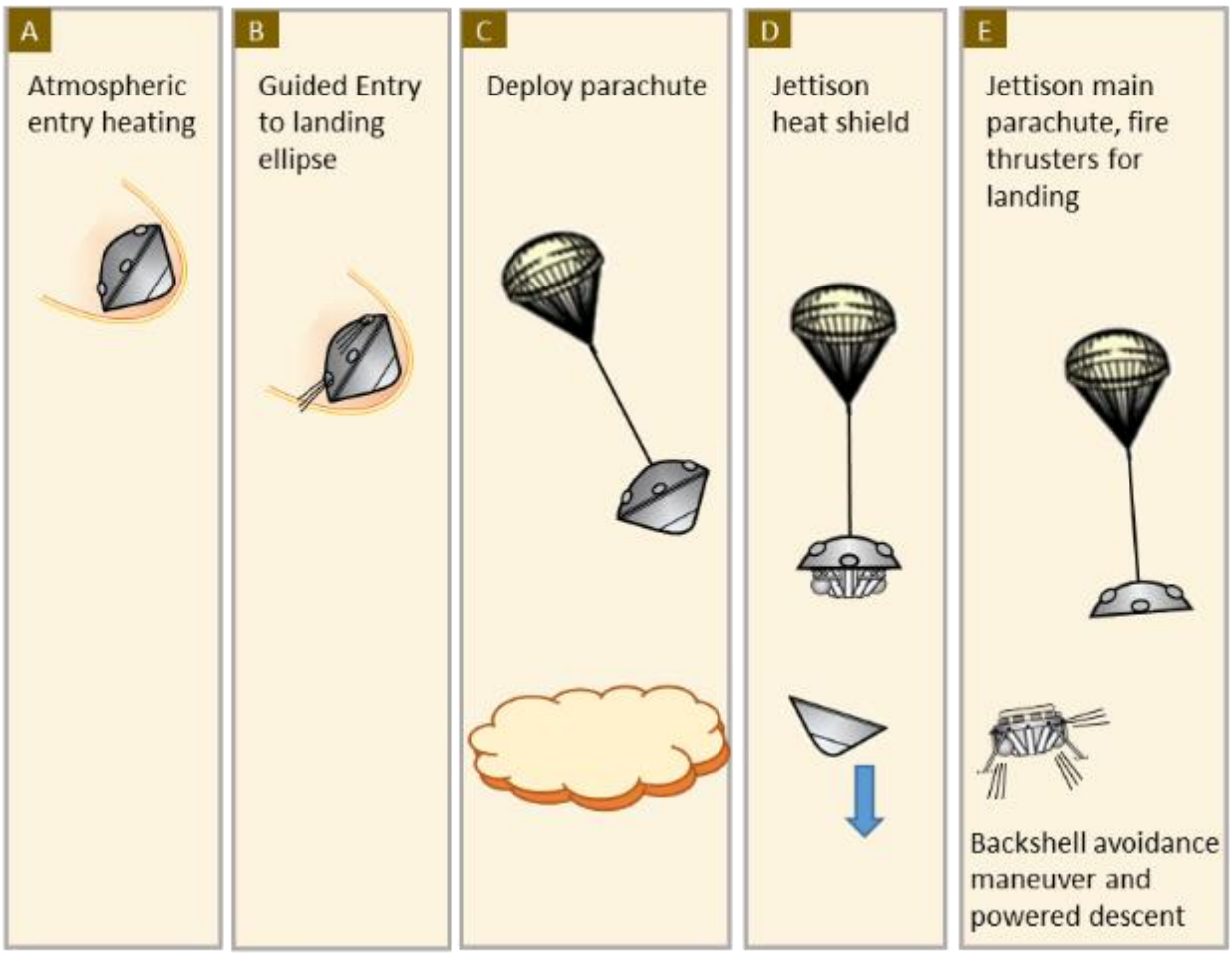


Figure 9 Atmospheric entry sequence for landing the BALLET vehicle. Panel A shows entry into the atmosphere and heating. Panel B shows the guided entry phase. Panel C shows the deployment of the parachute to slow the spacecraft down. Panel D shows the ejection of the heat shield and the descent of the spacecraft. Panel E shows the parachute jettisoned and the use of thrusters for power landing on the surface.

After the lander pallet has reached the surface as shown in Figure 10, the BALLET balloon would be inflated as shown in Figure 11. The balloon sits on the top of the lander pallet and is released from a constraint envelop which holds the folded balloon in place until ready for inflation. Compressed helium gas would be stored on board the lander for balloon inflation. The science payload would be packaged on the top surface of the lander but underneath the balloon. An inflation hose connected to the balloon would be cut after the inflation is completed and a valve on the balloon would be closed to seal the balloon. After the inflation hose is cut, the balloon would be raised up from the lander by extending the instrument tethers. Once the balloon is stable, the BALLET vehicle would move the feet using the tethers to walk off the lander and move to a target destination and begin its science mission as shown in Figure 12.

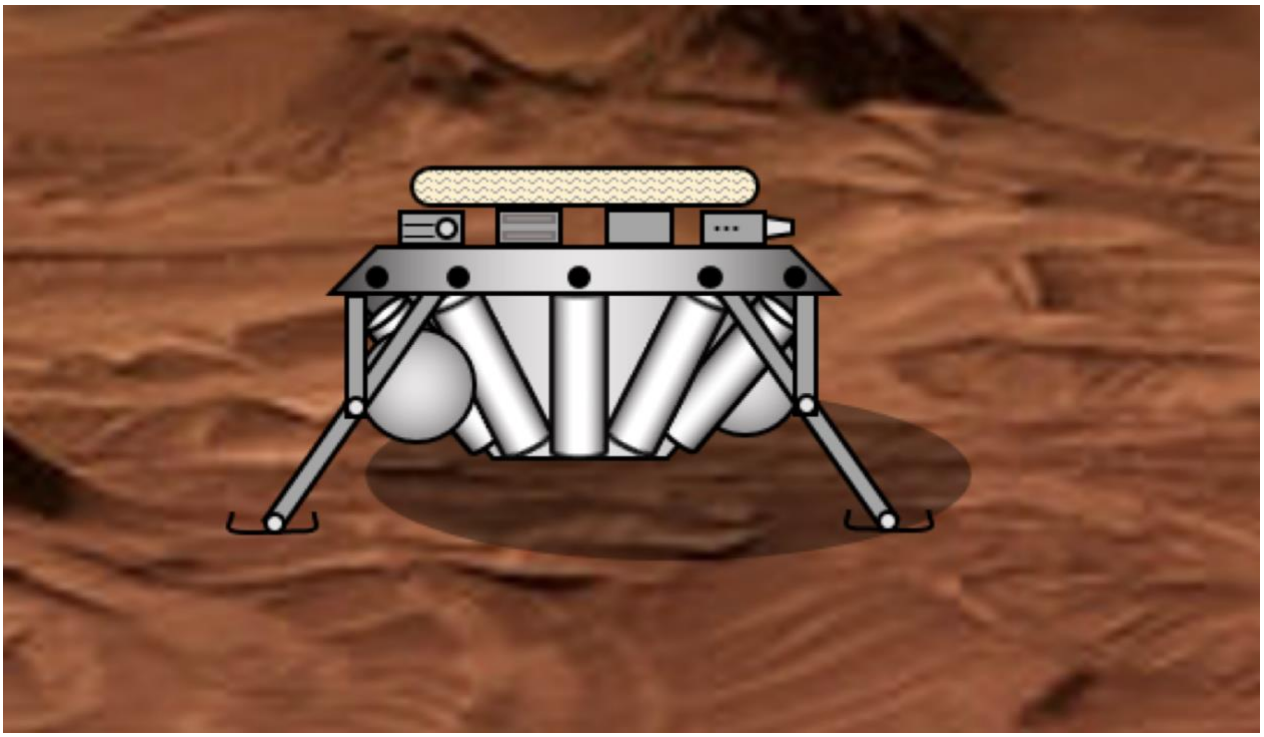


Figure 10 The BALLET Lander pallet configuration deployed on the surface of Mars or Titan.

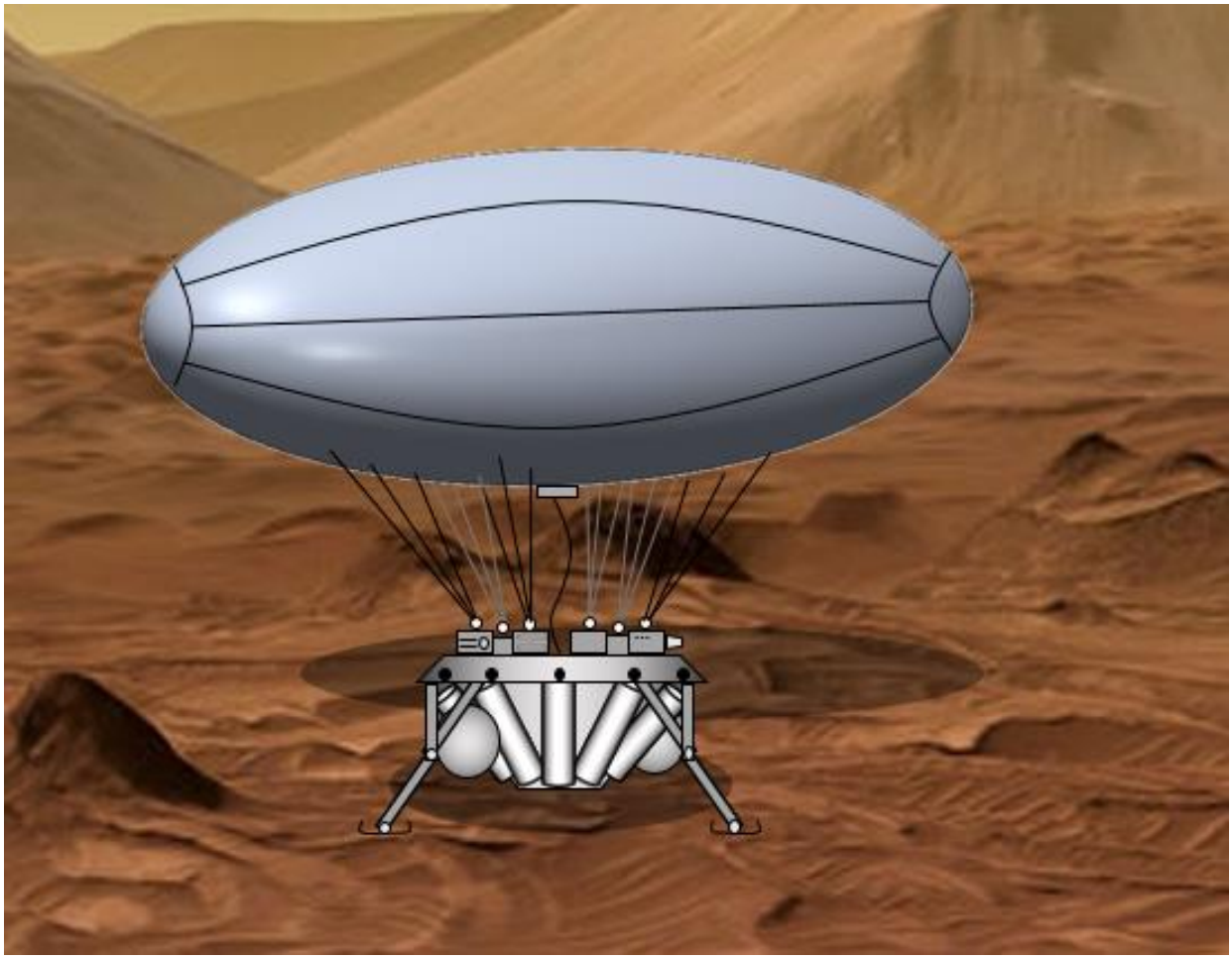


Figure 11 The BALLET balloon is inflated from the top of the lander pallet while the science payload remains in place.

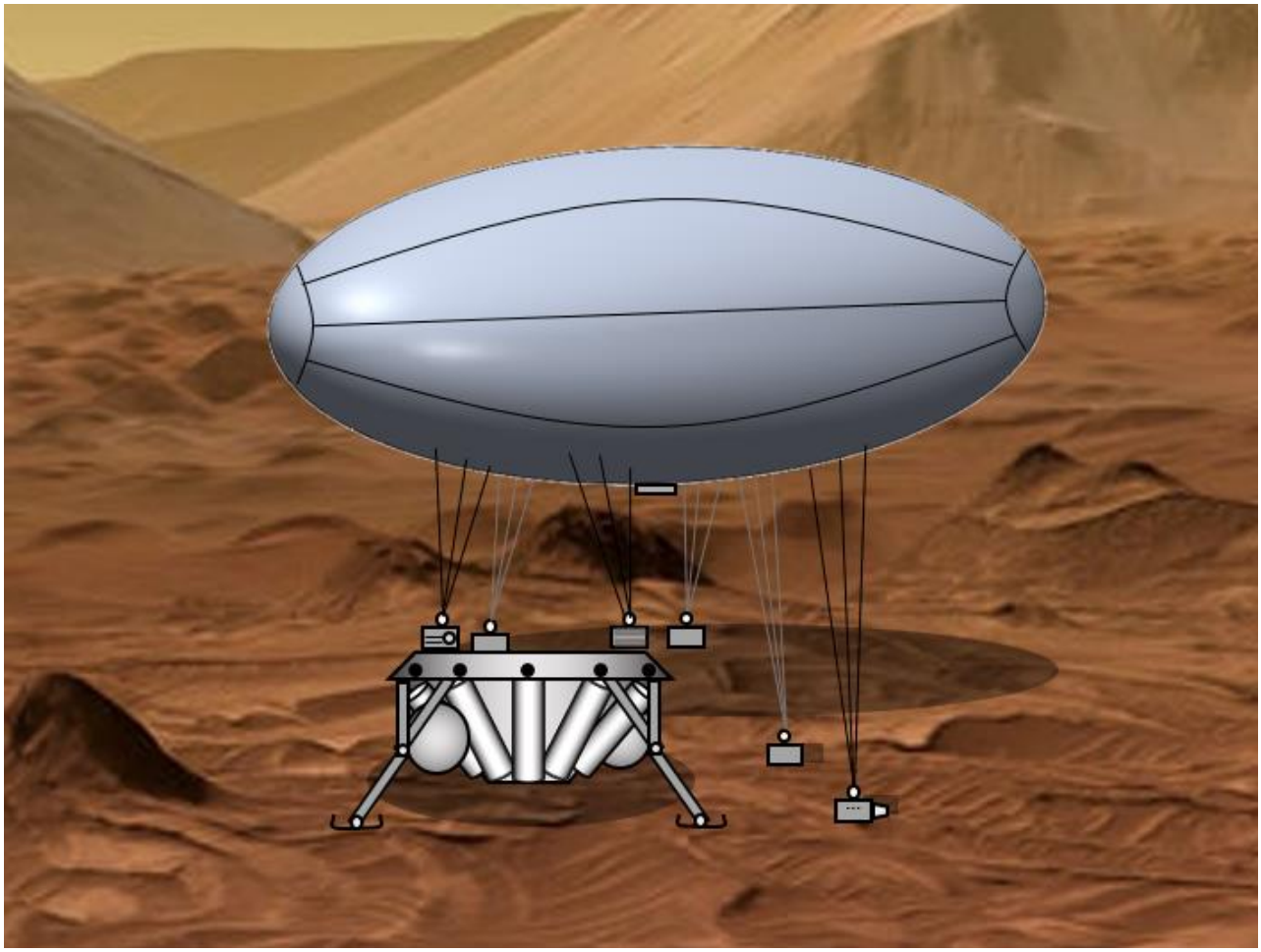


Figure 12 The BALLET vehicle walks off the Lander Pallet to begin the mobile science mission.

3.2. Mechanical Design and Materials

The balloon envelop would be constructed from gores that are connected together using pressure and heat sealed overlapping seams. JPL technology development programs in the 2000s for Mars and Titan balloons used these kinds of seals on envelope gores. The end fitting there the inflation tube is connected has a circular doubler layer to reduce stress since this region is a stress concentration point. Anchor points for the tethers would also be attached using a pressure and heat sealed patch which incorporate a loop for securing a tether line to the balloon envelope. The patches would be sized to accommodate the tensile stresses induced by the loads on the tethers during motion and wind drag. The ends of the balloon are oval or circular caps that are sealed to each of the longitudinal gores. The gores and end caps are evident in the illustrations shown in Figures 6, 11 and 12.

Packaging the balloon for stowage on the lander pallet involves folding the balloon along the length of the gores and laying the folded gores on top of each other. Thin sheets of packing material would be placed between the gores. The packing material falls away from the balloon during inflation. This method provides sufficient curvature radius in the center of each gore to prevent pinholes from being formed in the envelope. After the balloon is folded by gores, it is carefully rolled up from each end towards the center. During inflation, the balloon would simultaneously unroll and unfold. The folded balloon is not tightly packed like parachutes normally are. The packing density of the balloon needs to be comparatively low to prevent folding pinholes into double folded corners that often arise in packing balloons. A restraint cover is placed over the balloon to secure it to the lander pallet during transit. The restraint cover is removed prior to inflation on the planetary surface.

4. Concept Evaluation

4.1. Analyses

Analyses were performed with the goal of characterizing the stability of the BALLET balloon in the environments of Earth, Mars, and Titan. A representative balloon size was chosen for these analyses as defined by Equation (1).

$$a = 2b = 4c \tag{1}$$

Where a, b, and c are the lengths of the semi-major axes in the x, y, and z directions respectively.

Due to symmetry, the buoyant force of the balloon is assumed to act at the center of the ellipsoid. According the Archimedes' principle, the buoyant force is equal to the weight of the air displaced by the balloon. In this analysis, the buoyant force is considered to be the total upward force after subtracting the weight of mass added to the balloon. Added mass includes the mass of the balloon material, as well as the RTG proposed for a mission to Titan. The buoyant force F_b and balloon volume V are given by:

$$F_b = (\rho_{atm} - \rho_{He})Vg - m_{add}g \tag{2}$$

$$V = \frac{4}{3}\pi abc \tag{3}$$

ρ_{atm} is the atmospheric density, ρ_{He} is the density of helium at the planet's surface, g is the gravitational acceleration, $m_{add}$is mass added to the balloon, and a, b, and c are the semimajor axis given in Equation (1). Note that the density of the atmosphere and helium can vary cyclically with days and seasons on Earth, Mars, and Titan causing the buoyant

force to fluctuate according to Equation (2). Initial estimates of required balloon volume for testing on Earth and proposed missions to Titan and Mars are shown in Table 2 with added mass and buoyant force. Table 2 also shows the semi-major axis lengths resulting from the defined balloon volumes, following the relationship defined in Equation (1).

Planet	Volume [m^3]	Semi-major Axis Lengths (a, b, c) [m]	Added Mass [kg]	Buoyant Force [N]
Earth	1.925	(1.54, 0.77, 0.39)	0.5	14.72
Titan	11.534	(2.80, 1.40, 0.70)	45	10.14
Mars	88.134	(5.52, 2.76, 1.38)	0.1	5.51

Table 2 Balloon sizes and shapes on Earth, Titan, and Mars. The table considers the three proposed locations (rows) and four parameters of interest (columns) and provides information for each of the parameters specific to the location that the balloon would be deployed.

Each limb of BALLET is made up of three cables connecting the balloon and a payload. In this analysis each limb is simplified as a single cable connecting the payload center to the average position of the three connection points on the balloon.

Stability Analysis

The method used here to determine the stability of the BALLET balloon is to quantify the upper and lower bounds of the mass of the feet. If the foot mass is too low, the balloon is at risk of sliding or being lifted off the ground with gusts of wind. With a foot mass that is too great, the balloon may tilt or become unstable when lifting a leg. Finding the acceptable range of the mass of the foot will help maintain mission safety while providing requirements for the scientific instruments that can be chosen.

Figure 13 depicts the static force and moment balance analysis that is performed in this paper. The simplification of the limbs as single, vertical cables reduces the number of static balance equations to three:

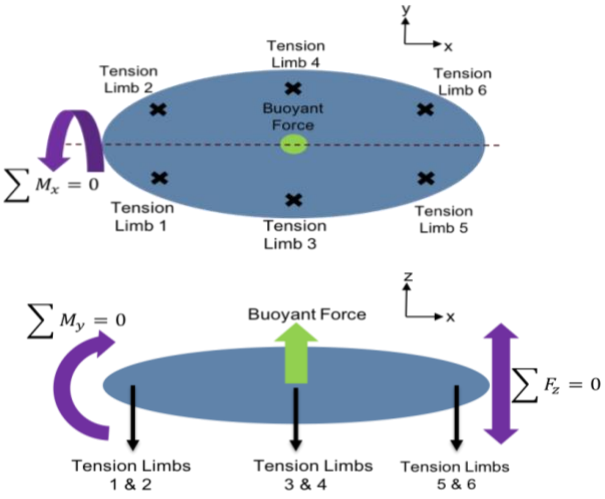


Figure 13 Top (top) and side (bottom) views of free body diagram of BALLET. The simplified model, where each payload is treated as a single vertical force at the average cable connection position is depicted. Given this configuration, only forces in the z axis and moments about the x and y axes are relevant. The buoyant force is treated as a single vertical force acting at the geometric center of the balloon.

$$\sum M_x = 0 \tag{4}$$

$$\sum M_y = 0 \tag{5}$$

$$\sum F_z = 0 \tag{6}$$

where M_x are moments about the x axis, M_y are moments about the y axis, and F_z are forces in the z axis.

Minimum Foot Mass

The minimum mass of the feet can be found with Equation (6). The buoyant force must be completely counteracted by the weight of the feet. As such, the sum of the weight of all feet must be equal to or greater than the buoyant force. Assuming all feet will have the same mass, Equation (7) defines the minimum mass of a single foot $\mathrm{m_{min}}$ as:

$$m_{min} = \frac{F_b}{6g} \tag{7}$$

Where $\mathrm{F_b}$ is the buoyant force and g is the acceleration due to gravity. Equation (7) remains true for both the single and dual limb locomotion techniques.

Maximum Foot Mass

The maximum mass of an individual foot is limited by the moment imparted on the balloon when lifting feet. At the maximum mass, one or more cables will go to zero tension. If any additional mass was added, the cable would buckle due to its inability to resist compressive loads, and the balloon would tilt. In order to solve for the maximum mass, SciPy's Sequential Least SQuares Programming (SLSQP) minimization capability was used in Python. The details of this implementation can be seen in Appendix A. The minimization problem is defined by Equations (8) and (9) below:

$$minimize(-x) \tag{8}$$

$$xg \geq T(x,n) \geq 0; n = \{1,2,3,4,5,6\} \tag{9}$$

where *x* is the mass of a foot, *g* is the acceleration due to gravity, and *T(x,n)* is the tension in the n^{th} cable when the foot mass is *x.*

The function *T(x,n)* can be obtained through the static force and moment analysis of Equations (4), (5), and (6). A diagram of this analysis can be seen in Figure 4.1.1. When

lifting one or two feet, the analysis yields an underdetermined system with infinite solutions. In each case this system has three static balance equations. When lifting one foot, there are five unknown cable tensions. When lifting two feet, there are four unknown cable tensions. In order to find a solution to this system, a least squares method was used. This method finds the solution where the magnitude of the solution vector is a minimum, while still satisfying the system of equations.

Aerodynamic Force Analysis

Aerodynamic forces will affect BALLET on Earth, Mars, and Titan. Drag will introduce transverse forces on the balloon which can cause the feet to slide or otherwise effect the balloon's stability. Lift can also be a concern if the balloon begins to tilt relative to the wind direction. Two methods were used to quantify the effects of wind on BALLET. The first method estimates drag force F_D as:

$$F_D = \frac{1}{2}\rho u^2 C_D A \tag{10}$$

where u is the flow velocity, C_D is the drag coefficient, ρ is the air density, and A is the reference area. For these estimates, a C_D of 0.5 is used.

The second method uses OpenFOAM, an open source software for computational fluid dynamics. OpenFOAM's PisoFoam solver was used, which finds the transient behavior of incompressible turbulent flow. To simplify this analysis, no turbulence models were considered. It is likely that this simplification also results in the worst case aerodynamic effects due to pressure drag dominating skin friction drag for bluff body shapes like the BALLET balloon.

OpenFOAM's blockMesh and snappyHexMesh tools were used with an STL model of the balloon to create a mesh for the simulation. For simulations measuring drag, symmetry was used on two planes to reduce the problem's complexity. Simulations measuring lift used symmetry on one plane, allowing for the balloon to tilt. All lift simulations were performed at an angle of attack of 10 degrees. Simulation flow inlets were given freestream velocity and zero gradient pressure boundary conditions. Flow outlets were given zero gradient velocity and zero pressure boundary conditions. Note that for incompressible flow, the pressure differential drives flow, not the pressure value. These boundary conditions result in a steady flow at the desired velocity. Flow in both the positive x and positive y axes were simulated in order to understand how the angle of incoming wind effects BALLET's stability. The boundary conditions of the balloon are no-slip velocity and zero gradient pressure, allowing

for a boundary layer to form on the balloons surface. Images of typical drag and lift simulations are depicted in Figure 14.

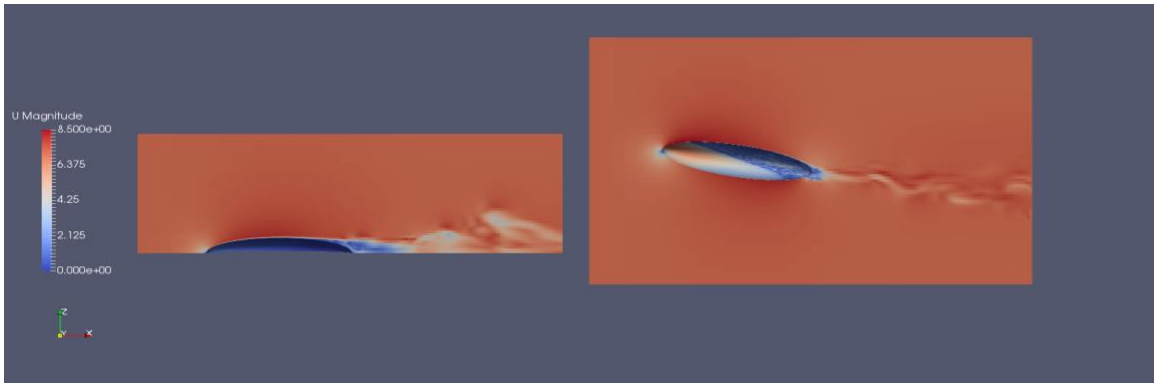


Figure 14 Velocity field for drag (left) and lift (right) simulations in OpenFOAM. Flow approaches an ellipsoid cross section from left to right in both images. The lift simulation uses an angle of attack of 10 degrees. Flow above, below, and in front of the balloon looks very stable. Flow behind the balloon shows vortices shedding from the rear tip of the ellipsoid. This indicates that a transient simulation is necessary to find the aerodynamic forces, as these forces will be cyclic rather than approach a steady state.

Earth Proof of Concept Analysis

The final analysis of this report provides a more detailed estimate of a proof-of-concept BALLET constructed on Earth. The goal of this analysis is to find the range of stable balloon volumes for the proof-of-concept given the proposed balloon material and payload mass. Similar to the stable foot mass analysis, SciPy's SLSQP minimizer was used to solve the following problem:

$$minimize(V) \tag{11}$$

$$m_p g \geq T(V,n) \geq 0; n = \{1,2,3,4,5,6\} \tag{12}$$

Where *V* is the balloon volume, m_p is the payload mass, *g* is the acceleration due to gravity, and *T(V,n)* is the tension in cable n at volume V. The buoyant force required when calculating *T(V,n)* is obtained through Equation (2), with the additional mass m_{add} calculated as:

$$m_{add} = \frac{11}{10} S \rho_b \tag{13}$$

where ρ_b is the area density of the balloon material, and *S* is the ellipsoid surface area. The coefficient of $\frac{11}{10}$ is introduced to account for seams and attachment features represented by a 10% increase in balloon mass.

4.2. Results

4.2.1. Titan

Flat Ground

Figure 15 and Figure 16 depict the maximum stable foot mass on Titan on flat ground for the balloon size given in Table 4.1.1. These results show the tension in each cable with the specified leg lifted off the ground. Due to the symmetry of the balloon, there are only four unique cases.

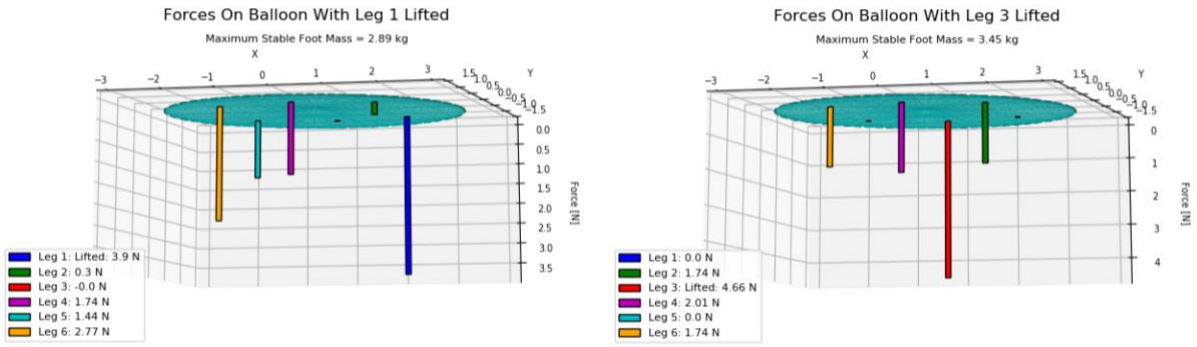


Figure 15 Titan cable tensions with one payload lifted. The two possible configurations are show with a vertical bar at each tension point. The length of these bars is related to the tension the corresponding cable. When leg 1 is lifted, the maximum stable foot mass is 2.89 kg, and the tension in leg 3 goes to zero. With leg 3 lifted, this mass is 3.45 kg, and the tension in legs 1 and 5 go to zero.

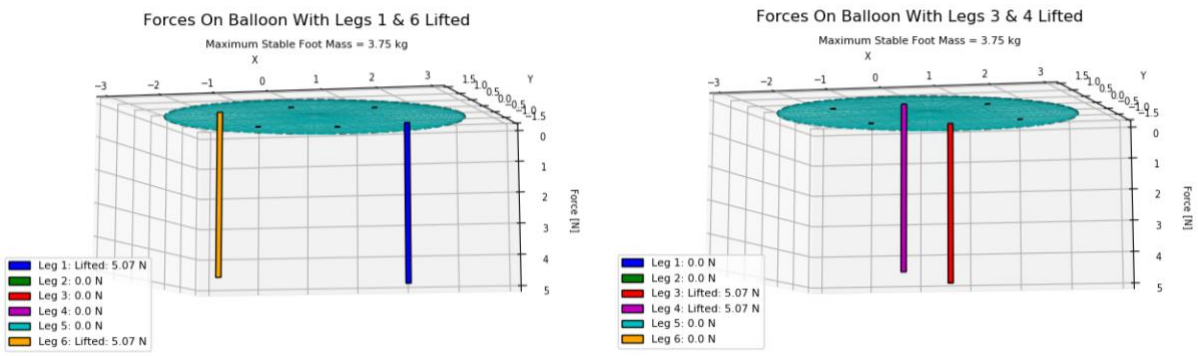


Figure 16 Titan cable tensions with two payloads lifted. The two possible configurations are show with a vertical bar at each tension point. The length of these bars is related to the tension the corresponding cable. When legs 1 and 6 are lifted, the maximum stable foot mass is 3.75 kg, and all other tensions go to zero. With legs 3 and 4 lifted, this mass is also 3.75 kg, and all other tensions go to zero.

As seen in the above figures, at the maximum foot mass one or more tensions go to zero in all cases. When a single leg is lifted, the moment imparted on the balloon limits the foot mass. The maximum stable mass in these cases is dependent on balloon shape and volume. When two opposing legs are lifted as in Figure 16, they cancel out the moment imparted by a single lifted payload, resulting in no moment on the balloon. The weight of each payload at the maximum stable mass is equal to half of the buoyant force in this case. When lifting two opposing payloads simultaneously, only the balloon volume effects the maximum stable foot mass. This result shows that a two-legged gait provides the most stable configuration. If maneuvers can be limited to two-legged gaits exclusively, this would allow more scientific instruments to be stored in the payload or reduce the necessary size

of the balloon when compared to single-legged gaits. For either case, the minimum stable foot mass is 1.25kg according to Equation (7).

Figure 17 demonstrates the benefits of two-legged gaits more clearly. This figure shows the tension in the cables while in a two-legged gait, when the payload mass is equal to the maximum stable mass of a single-legged gait found in Figure 15 In this case, all cable tensions are greater than zero, indicating significantly greater stability than the single-legged gait at the same payload mass.

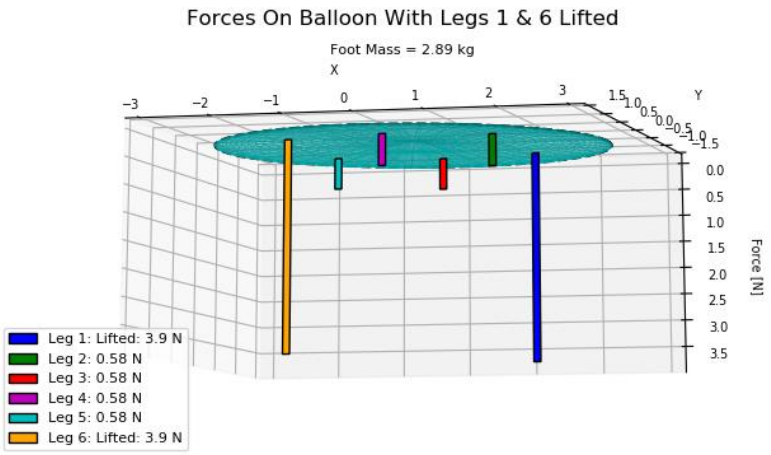


Figure 17 Titan cable tensions for a two-legged gait with the maximum foot mass of single-legged gait of 2.89 kg. Legs 1 and 6 are lifted, and all other cables have a tension of 0.58 N, showing the greater stability of two-legged gaits when compared to a single-legged gait with the same foot mass.

Slope

It may be desirable to pitch the BALLET balloon when traversing a slope for multiple reasons. On a steep slope, it is possible that the front or back of the balloon could come into contact with the slope if a pitch isn't applied. Also, if winds are flowing along a slope, a pitch can be applied to reduce the balloon's angle of attack, preventing lift and drag from overwhelming the balloon. When the BALLET balloon is pitched, the moment arms that determine balloon stability are altered as depicted in Figure 18. Additionally, the moment arms are not equally affected on opposite sides of the balloon due to the connection points lying outside of the x-y plane. This results in the cable tensions shifting toward the front or back of the balloon in both the single and two-legged gait patterns. These results are shown in Figure 19 and Figure 20.

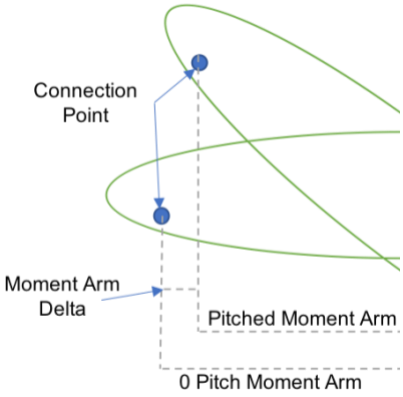


Figure 18 Effect of pitch on the moment arms created by the cables. Two ellipsoidal balloons with differing pitch are superimposed. The pitch causes the moment arms to differ between the two cases, such that the greater pitch results in a smaller moment.

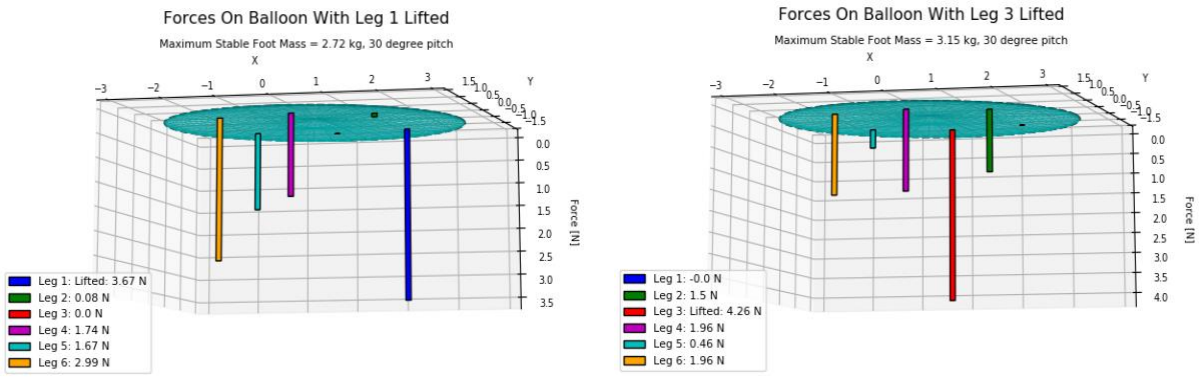


Figure 19 Titan cable tensions with one payload lifted and a 30 degree pitch. The two possible configurations are show with a vertical bar at each tension point. The length of these bars is related to the tension the corresponding cable. When leg 1 is lifted, the maximum stable foot mass is 2.72 kg, and the tension in leg 3 goes to zero. With leg 3 lifted, this mass is 3.15 kg, and the tension in leg 1 goes to zero.

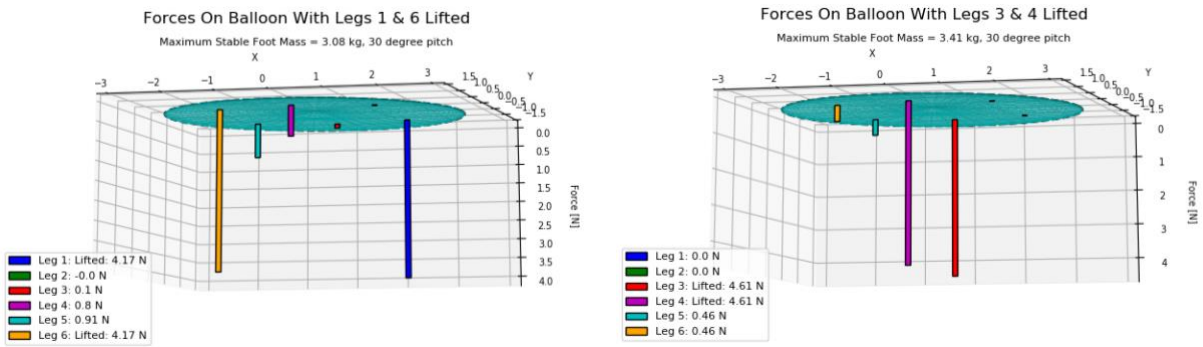


Figure 20 Titan cable tensions with two payloads lifted and a 30-degree pitch. The two possible configurations are show with a vertical bar at each tension point. The length of these bars is related to the tension the corresponding cable. When legs 1 and 6 are lifted, the maximum stable foot mass is 3.08 kg, and the tension in leg 2 goes to zero. With legs 3 and 4 lifted, this mass is 3.41 kg, and the tension in legs 1 and 2 go to zero.

All results show at least one cable tension going to zero, indicating a local maximum value was found. All maximum stable payload masses are also less than their zero pitch counterparts. Figure 20 also demonstrates the uneven shift in the moment arms. When at zero pitch, four cable tensions go to zero for two-legged gaits, but at a 30 degree pitch, this is not the case. This is due to the uneven change in moment arms between the front and back of the balloon, as described above. For both the zero and 30 degree pitch cases, the minimum stable payload mass remains at 1.25kg, as defined in Equation (7).

Figure 21 shows the maximum stable payload mass at varying values of pitch, for specific payloads being lifted. In both of these graphs, the lower line indicates the maximum mass

available for the corresponding gait pattern. At low values of pitch the dual leg locomotion technique shows far more stability than the single legged technique. At high values of pitch, the advantage of the two-legged gait is greatly minimized, to the point that the single legged gait is equally stable.

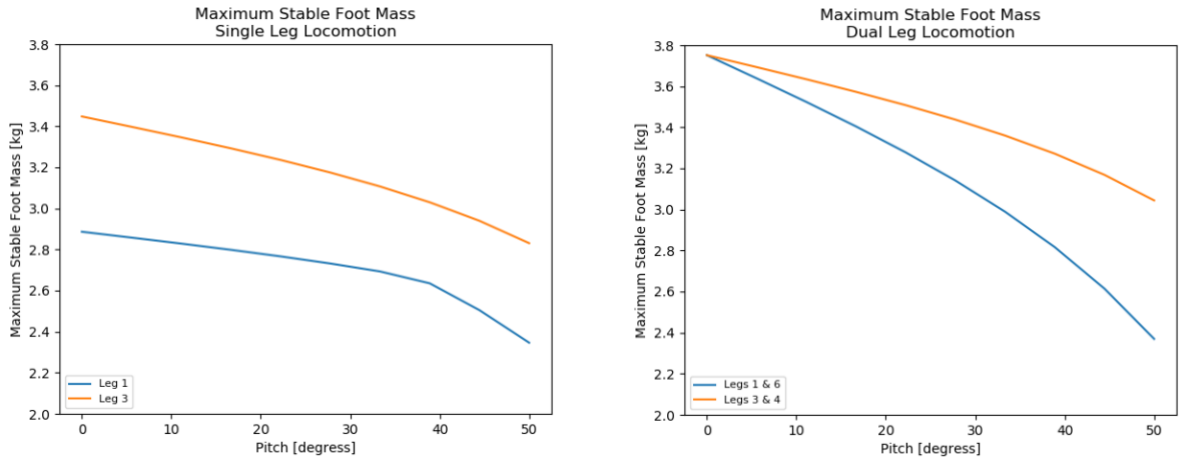


Figure 21 Maximum stable payload mass on Titan at varying pitch for single and dual-legged locomotion techniques. As pitch increases the maximum stable foot mass decreases in both cases, with the dual-legged foot mass decreasing at a greater rate. At a pitch value of near 40 degrees the stability advantages of dual-legged locomotion are lost.

Buoyancy Changes with Atmospheric Conditions

The temperature on Titan's surface is not known to change drastically over the course of a day or season (Cottini et al. 2012). This leads to little variation in BALLET's buoyancy force over time. Effects of changes in atmospheric conditions were not considered on Titan due to its fairly stable climate.

Aerodynamic Forces

A first estimate of drag forces on the BALLET balloon using Equation (10) is calculated. A worst case drag coefficient estimate of 0.5 is used. The resulting drag forces are as follows: 4.073 N drag when flow is in the x direction, and 8.147 N drag when flow in the y direction.

An open-source computational fluid dynamics package, OpenFOAM, was used to simulate and analyze the aerodynamic forces on BALLET. The simulation parameters used for Titan are a freestream velocity of 1 m/s, air density of 5.280 kg/m^3, and kinematic viscosity of 1.246e-6 m^2/s. Kinematic viscosity of Titan's atmosphere was estimated as that of pure Nitrogen gas at Titan's average surface temperature due to the belief that Titan's

atmosphere is greater than 95% Nitrogen. Results of these simulations are presented in Figure 22

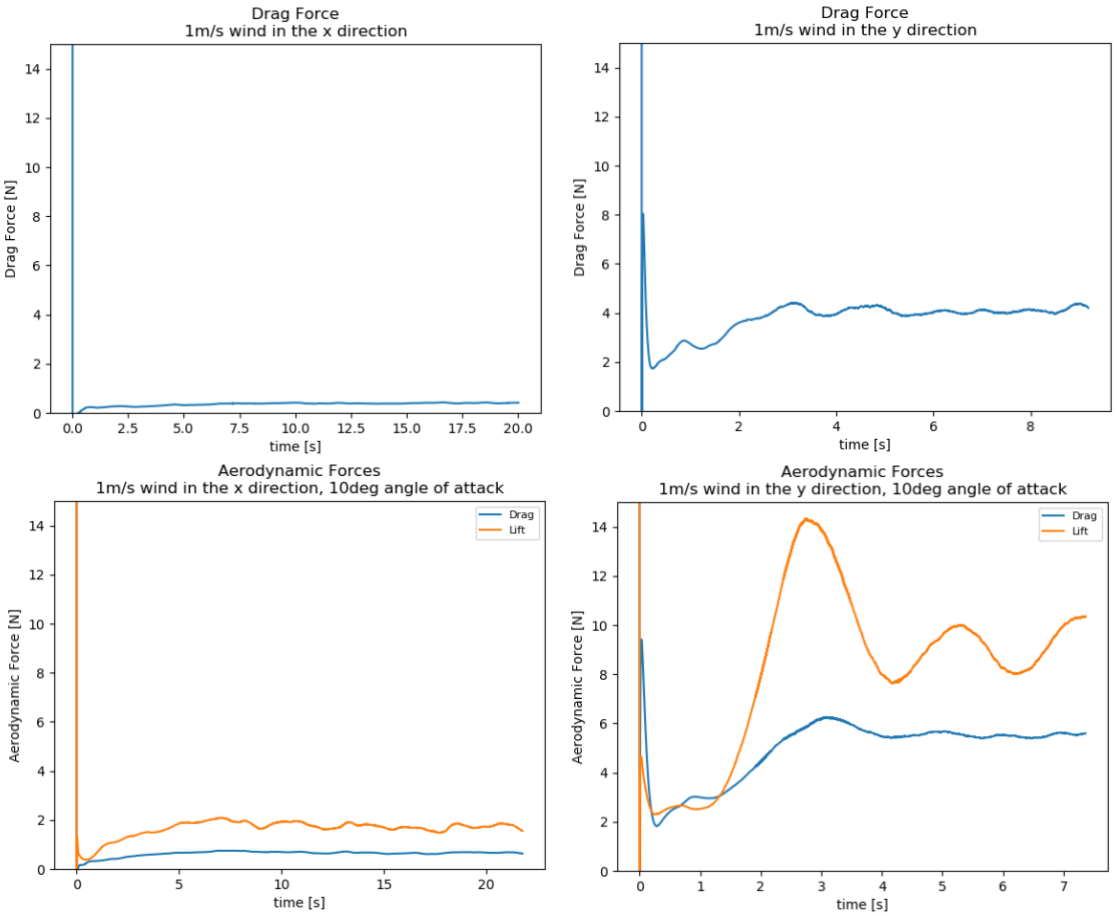


Figure 22 Aerodynamic forces on Titan at nominal wind speed of 1 m/s. Forces were recorded with air flowing in the x and y directions. Simulations show initial perturbations in force before reaching a cyclic steady state. Steady state average values for x direction flow are 0.41 N drag at zero pitch, 0.65 N drag at 10 degree pitch, and 1.66 N lift at 10 degree pitch. Steady state average values for y direction flow are 4.05 N drag at zero pitch, 5.55 N drag at 10 degree pitch, and 9.05 N lift at 10 degree pitch.

OpenFOAM results show lower drag than the Equation (10) estimate, which is expected given the large drag coefficient used in that calculation. The simulations also show that it is preferable to face BALLET into the wind, such that the flow is perpendicular to the balloon's smallest cross sectional area. Facing this way will result in the lowest possible drag and lift forces. The drag forces found in the simulation are significant when compared to the buoyancy of the proposed titan balloon of 10.14N. In the case that BALLET sees flow from its y direction, the magnitude of the drag force will about 40% of the buoyant force. The effect of these forces on stability will need to be analyzed further. In the event that the flow

incidents the balloon at an angle of attack of 10 degrees, the lift forces can be large enough to carry the balloon away or push it into the ground. Note that the proposed size of the Titan balloon is large enough to accommodate a 45kg generator. One way to minimize the expected lift and drag forces is to reconsider the size of this generator, allowing for a significantly smaller balloon. More analysis can be done to determine the relationship between lift and angle of attack for this balloon shape, allowing for a maximum acceptable tilt value to be defined.

4.2.2. Mars

Given the similarities between the Titan, Mars, and Earth analyses, many of the comments made on the Titan data in the previous section pertain to Mars and Earth as well. These comments will not be repeated, but the results analysis specific to Mars will be shown.

Flat Ground

The size and shape of the balloon used for this analysis is given in Table 2. The ratio of semi-major axes remains the same as the Titan analyses, but with increased volume to create more buoyancy in the thin atmosphere of Mars. The results are displayed on Figures 23 and 24.

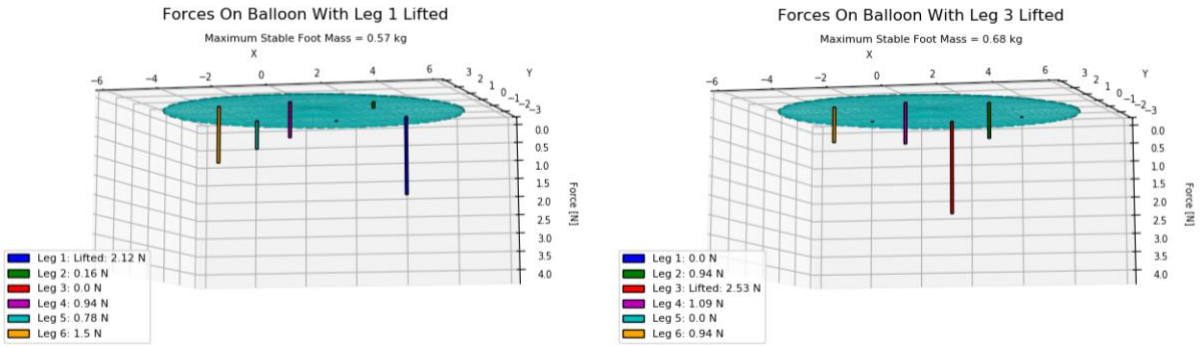


Figure 23 Mars cable tensions with one payload lifted. The two possible configurations are show with a vertical bar at each tension point. The length of these bars is related to the tension the corresponding cable. When leg 1 is lifted, the maximum stable foot mass is 0.57 kg, and the tension in leg 3 goes to zero. With leg 3 lifted, this mass is 0.68 kg, and the tension in legs 1 and 5 go to zero.

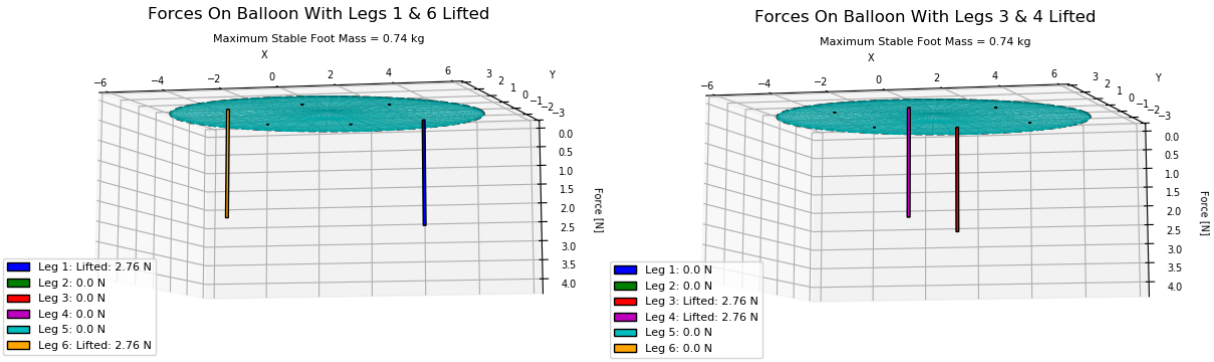


Figure 24 Mars cable tensions with two payloads lifted. The two possible configurations are show with a vertical bar at each tension point. The length of these bars is related to the tension the corresponding cable. When legs 1 and 6 are lifted, the maximum stable foot mass is 0.74 kg, and all other tensions go to zero. With legs 3 and 4 lifted, this mass is also 0.74 kg, and all other tensions go to zero.

The shape of the resulting force distribution on the balloon is identical to that of the Titan analysis due to the similar geometry. According to Equation (7) the minimum payload mass is 0.248 kg.

Slope

The corresponding results for slopes on Earth are shown on Figures 25 and 26. And Figure 27 shows the maximum stable payload mass at varying values of pitch.

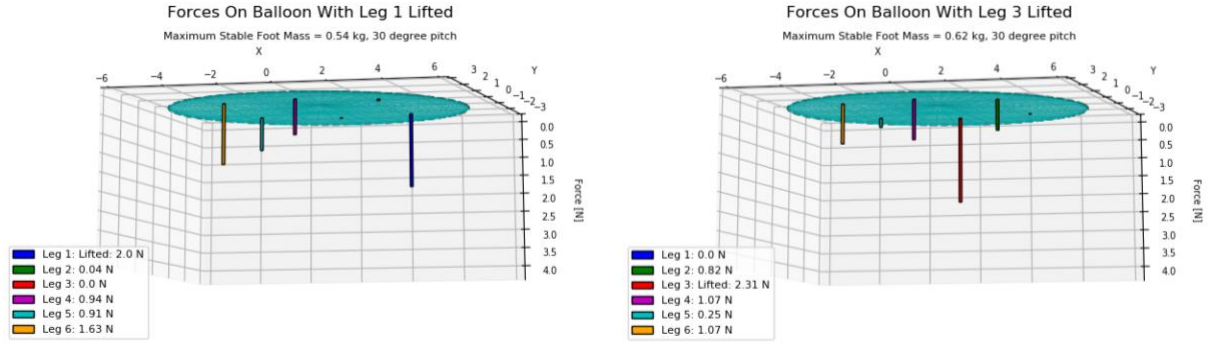


Figure 25 Mars cable tensions with one payload lifted and a 30 degree pitch. The two possible configurations are show with a vertical bar at each tension point. The length of these bars is related to the tension the corresponding cable. When leg 1 is lifted, the maximum stable foot mass is 0.54 kg, and the tension in leg 3 goes to zero. With leg 3 lifted, this mass is 0.62 kg, and the tension in leg 1 goes to zero.

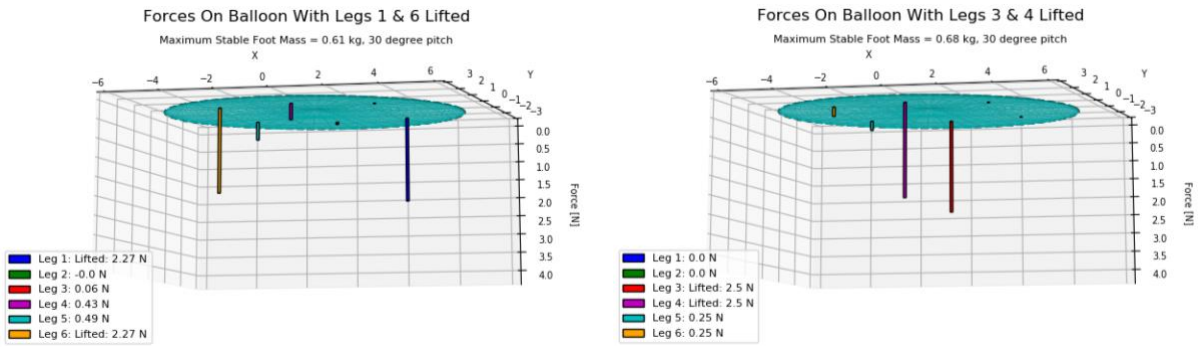


Figure 26 Mars cable tensions with two payloads lifted and a 30 degree pitch. The two possible configurations are show with a vertical bar at each tension point. The length of these bars is related to the tension the corresponding cable. When legs 1 and 6 are lifted, the maximum stable foot mass is 0.61 kg, and the tension in leg 2 goes to zero. With legs 3 and 4 lifted, this mass is 0.68 kg, and the tension in legs 1 and 2 go to zero.

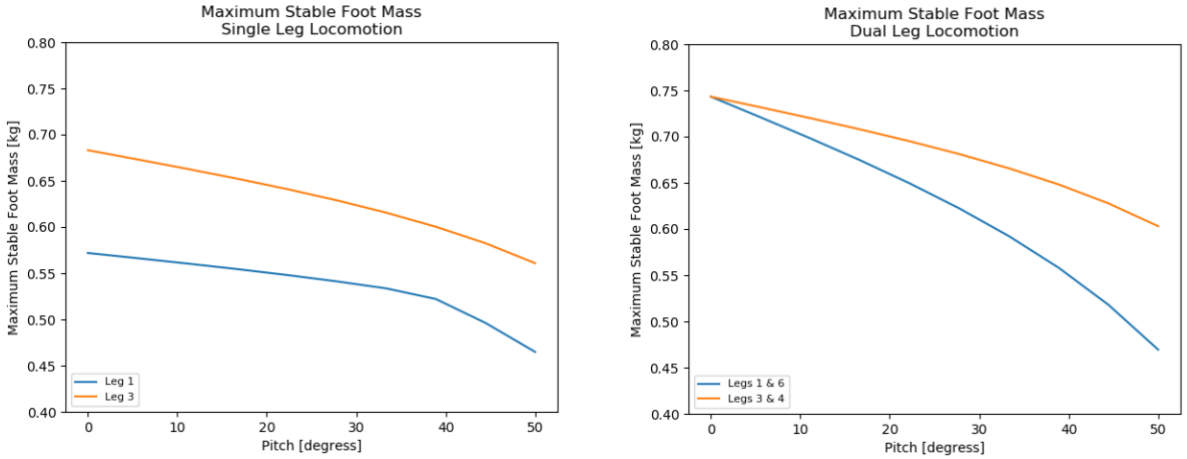


Figure 27 Maximum stable payload mass on Mars at varying pitch for single and dual-legged locomotion techniques. As pitch increases the maximum stable foot mass decreases in both cases, with the dual-legged foot mass decreasing at a greater rate. At a pitch value of near 40 degrees the stability advantages of dual-legged locomotion are lost.

Buoyancy changes with Atmospheric Conditions

Unlike Titan, the temperature of Mars fluctuates greatly on a daily and seasonal basis. This temperature change will have an impact on the buoyancy of the balloon. The payload mass must be such that BALLET remains stable for all possible buoyancy values that will be encountered on Mars. Data from the Viking landers is used to quantify what conditions can be expected at two different latitudes on Mars.

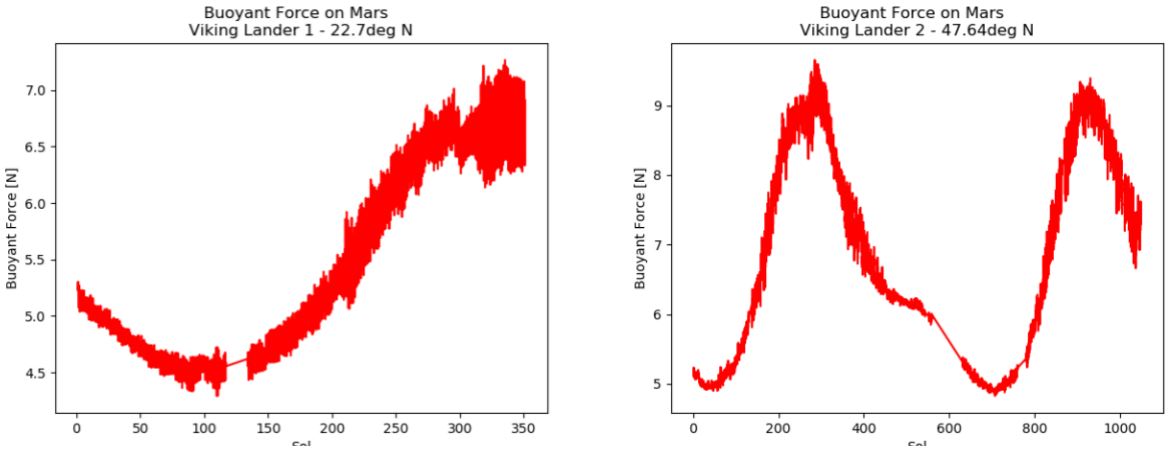


Figure 28 Change in buoyancy over time based on Viking lander data. At 22.7 degrees N latitude the buoyancy of the proposed balloon can vary between about 4.5 and 7.0 N. At 47.64 degrees N latitude, the buoyancy can vary between about 5 and 9.5 N.

Figure 28 shows the magnitude of oscillations in buoyancy on a seasonal scale at 22.7 and 47.64 degrees North latitude. Based on this data, Table 3 shows the range of stable payload masses for dual and single-legged gaits for a long term mission to Mars with a balloon of the defined volume and shape. These values are at zero pitch and would vary similarly to Figure 4.2.13 with changes in pitch. At both latitudes the stable minimum mass for a long term mission is larger than the 0.248 kg estimated at nominal conditions. Similarly, the maximum mass has decreased in all cases.

	Single-Legged Gait		Two-Legged Gait	
Latitude [deg N]	**Minimum Mass [kg]**	**Maximum Mass [kg]**	**Minimum Mass [kg]**	**Maximum Mass [kg]**
22.27	0.32308	0.44518345	0.32308	0.57855969
47.64	0.42383	0.50727619	0.42383	0.65925532

Table 3 Stable payload mass range at Viking lander locations. The table consists of two latitudes (rows) and four columns of minimum and maximum masses for both gait types.

Aerodynamic Forces

A first estimate of drag forces on the BALLET balloon using Equation (10) is calculated. A worst case drag coefficient estimate of 0.5 is used. The resulting drag forces are as follows: 6.501 N drag when flow is in the x direction, and 13.001 N drag when flow in the y direction.

OpenFOAM simulations were performed as previously described. The simulation parameters used for Mars are a freestream velocity of 10 m/s, air density of 0.020 kg/m^3, and kinematic viscosity of 6.54e-4 m^2/s. Kinematic viscosity was estimated as that of pure

carbon dioxide due to it making up about 95% of Mars' atmosphere. Results of these simulations are presented in Figure 29.

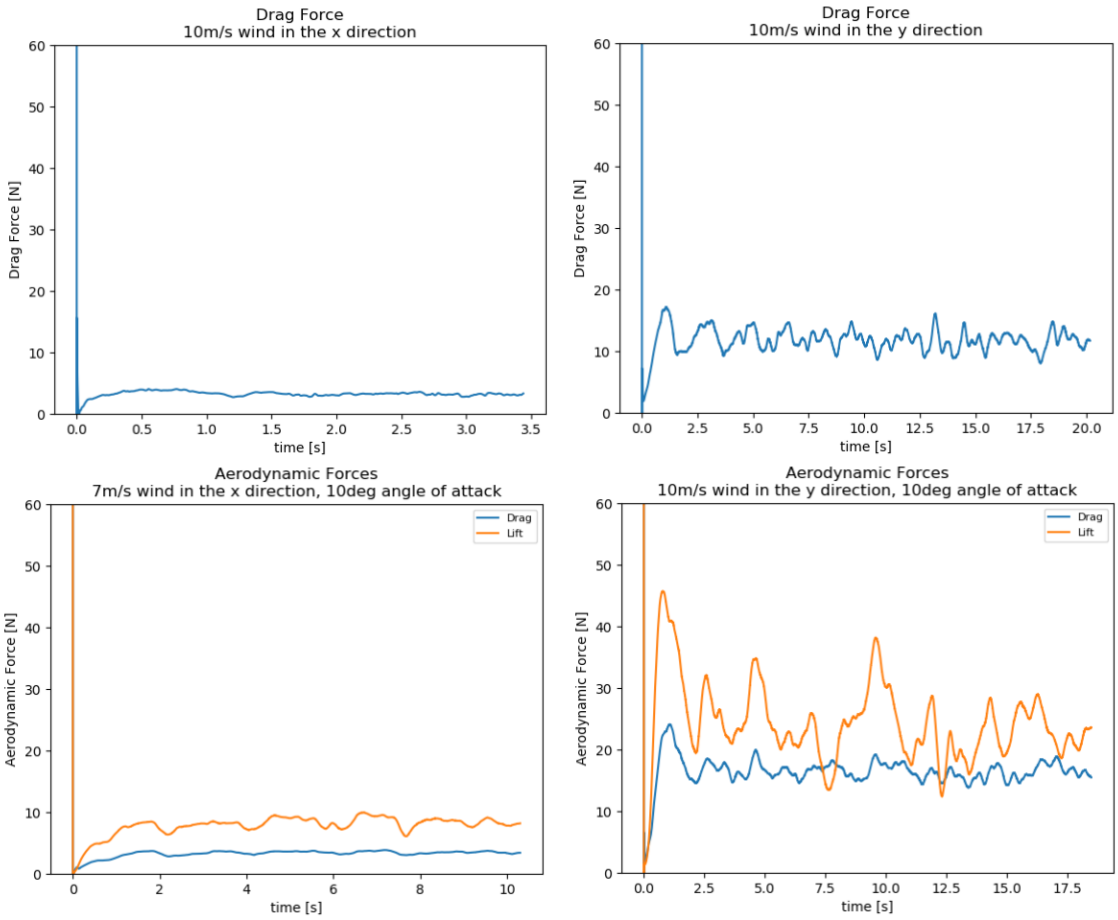


Figure 29 Aerodynamic forces on Mars at nominal wind speed of 10 m/s. Forces were recorded with air flowing in the x and y directions. Simulations show initial perturbations in force before reaching a cyclic steady state. Steady state average values for x direction flow are 3.16 N drag at zero pitch, 3.28 N drag at 10 degree pitch, and 8.31 N lift at 10 degree pitch. Steady state average values for y direction flow are 11.41 N drag at zero pitch, 15.43 N drag at 10 degree pitch, and 22.41 N lift at 10 degree pitch.

Simulation results are larger than expected based on the initial estimate of Equation (10). This could indicate a need to better tune the expected kinematic viscosity on Mars or to attempt more simulations with varying viscosity values. Mars' atmosphere changes significantly throughout a day and year, which would affect kinematic viscosity and thus the drag and lift forces. Similar to the Titan results, these show that facing BALLET such that the wind flows along its x-axis is preferable. Both the Equation (10) estimate and the OpenFOAM simulations show that aerodynamic forces will be a large problem on Mars. The required size of the balloon is much larger than on Titan due to the thin atmosphere,

causing these forces to be significant. Additionally, average windspeed on Mars is much greater than on Titan, exacerbating the issue. These results show that without serious consideration on how to mitigate these aerodynamic effects, a BALLET balloon on Mars may not be possible.

4.2.3. Earth

Flat Ground

The corresponding results for flat ground on Earth are shown on Figures 30 and 31.

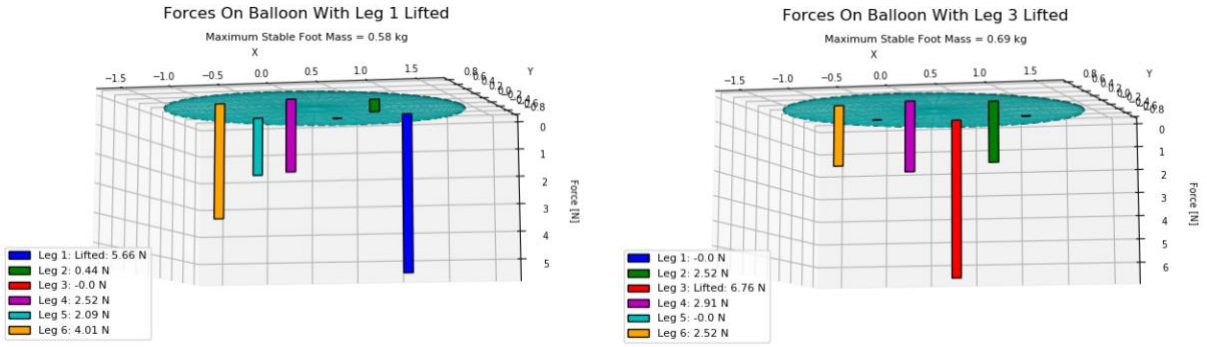


Figure 30 Earth cable tensions with one payload lifted. The two possible configurations are show with a vertical bar at each tension point. The length of these bars is related to the tension the corresponding cable. When leg 1 is lifted, the maximum stable foot mass is 0.58 kg, and the tension in leg 3 goes to zero. With leg 3 lifted, this mass is 0.69 kg, and the tension in legs 1 and 5 go to zero.

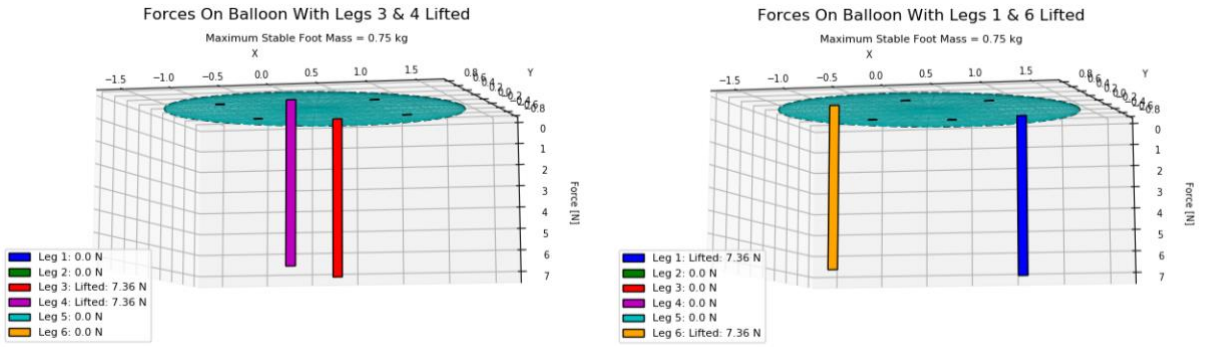


Figure 31 Earth cable tensions with two payloads lifted. The two possible configurations are show with a vertical bar at each tension point. The length of these bars is related to the tension the corresponding cable. When legs 1 and 6 are lifted, the maximum stable foot mass is 0.75 kg, and all other tensions go to zero. With legs 3 and 4 lifted, this mass is also 0.75 kg, and all other tensions go to zero.

Slope

On slopes, the results on Earth are shown on Figures 32 and 33. Figure 34 shows the maximum stable payload mass at varying values of pitch.

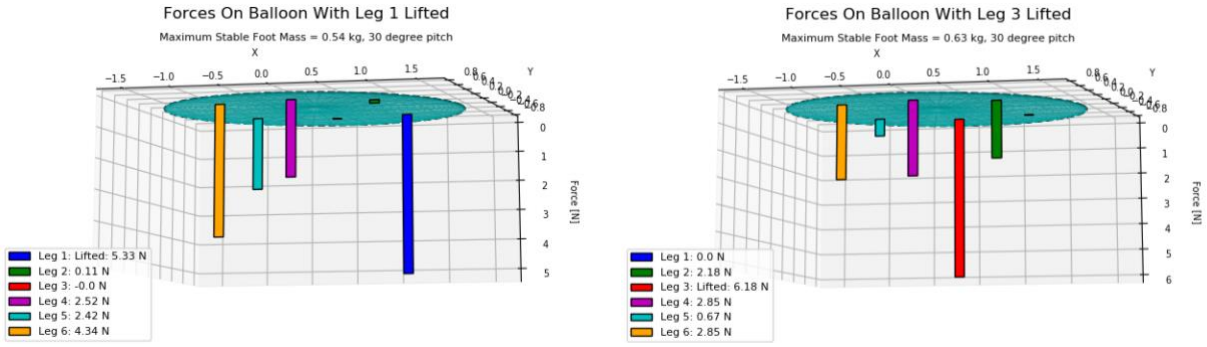


Figure 32 Earth cable tensions with one payload lifted and a 30 degree pitch. The two possible configurations are show with a vertical bar at each tension point. The length of these bars is related to the tension the corresponding cable. When leg 1 is lifted, the maximum stable foot mass is 0.54 kg, and the tension in leg 3 goes to zero. With leg 3 lifted, this mass is 0.63 kg, and the tension in leg 1 goes to zero.

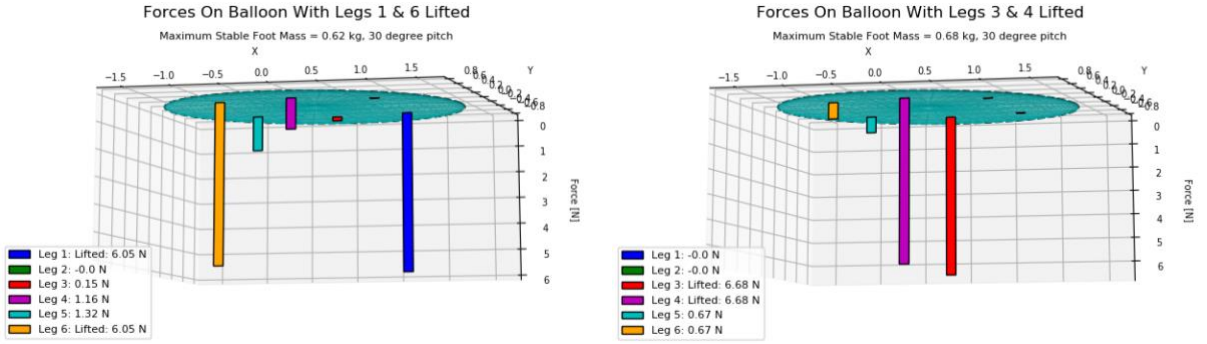


Figure 33 Earth cable tensions with two payloads lifted and a 30 degree pitch. The two possible configurations are show with a vertical bar at each tension point. The length of these bars is related to the tension the corresponding cable. When legs 1 and 6 are lifted, the maximum stable foot mass is 0.62 kg, and the tension in leg 2 goes to zero. With legs 3 and 4 lifted, this mass is 0.68 kg, and the tension in legs 1 and 2 go to zero.

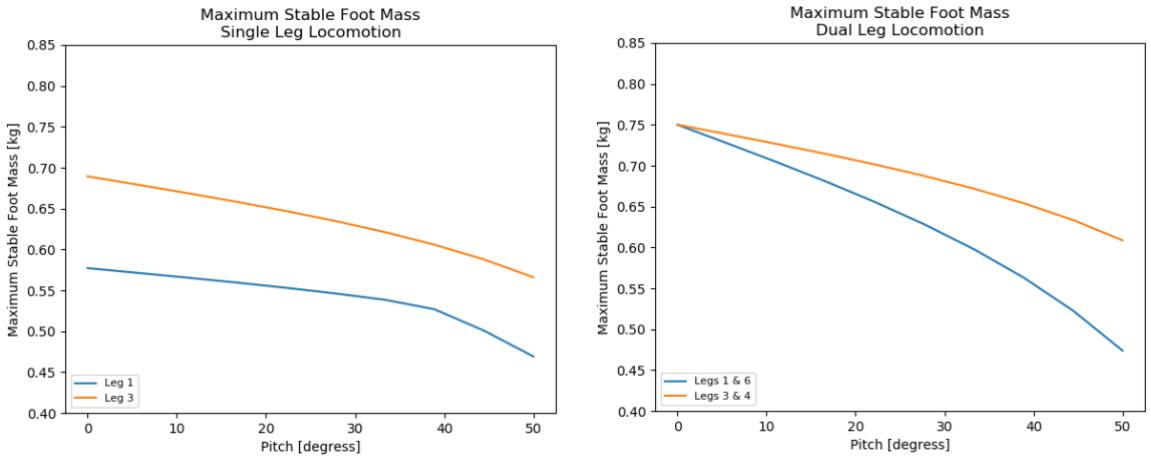


Figure 34 Maximum stable payload mass on Earth at varying pitch for single and dual-legged locomotion techniques. As pitch increases the maximum stable foot mass decreases in both cases, with the dual-legged foot mass decreasing at a greater rate. At a pitch value of near 40 degrees the stability advantages of dual-legged locomotion are lost.

Buoyancy changes with Atmospheric Conditions

While the atmospheric conditions on Earth vary greatly with latitude, time of day, and season, an Earth balloon would be built as a proof-of-concept. As such, it would be tested largely indoors where atmospheric conditions like temperature and wind can be controlled to avoid large differences in buoyancy and stability as might be seen on Mars. This study assumes average conditions that would be found indoors, and does not attempt to define the stability of BALLET for all climates expected on Earth.

Aerodynamic Forces

A first estimate of drag forces on the BALLET balloon using Equation (10) is calculated. A worst case drag coefficient estimate of 0.5 is used. The resulting drag forces are as follows: 13.946 N drag when flow is in the x direction, and 27.891 N drag when flow in the y direction.

OpenFOAM simulations were performed as previously described. The simulation parameters used for Earth are a freestream velocity of 7 m/s, air density of 1.217 kg/m^3, and kinematic viscosity of 1.5e-5 m^2/s. Results of these simulations are presented in Figure 35.

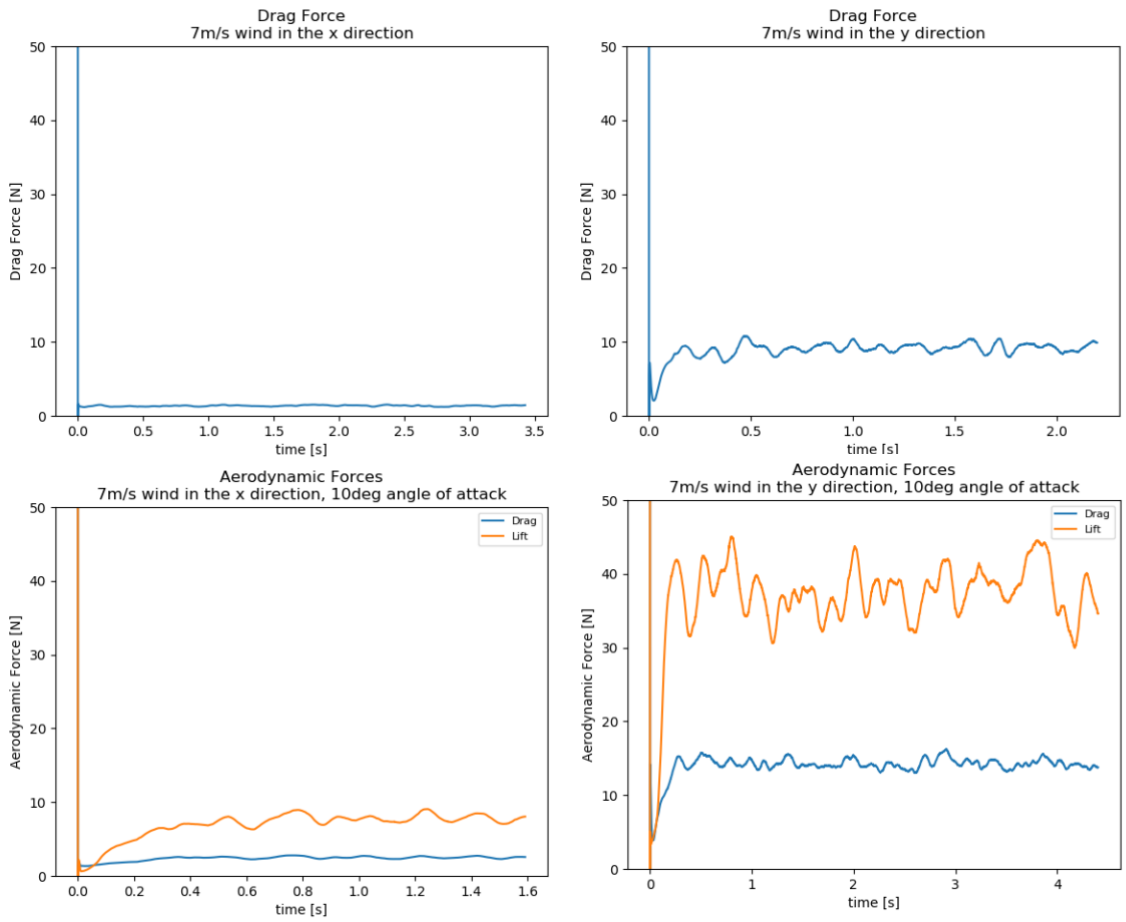


Figure 35 Aerodynamic forces on Earth at nominal wind speed of 7 m/s. Forces were recorded with air flowing in the x and y directions. Simulations show initial perturbations in force before reaching a cyclic steady state. Steady state average values for x direction flow are 1.34 N drag at zero pitch, 2.34 N drag at 10 degree pitch, and 7.94 N lift at 10 degree pitch. Steady state average values for y direction flow are 9.38 N drag at zero pitch, 13.93 N drag at 10 degree pitch, and 36.85 N lift at 10 degree pitch.

These results are very similar to that of Titan and Mars. While the magnitudes of aerodynamic forces on Earth, Mars, and Titan vary greatly, the general trend remains the same. Lift and drag are significantly smaller when wind is flowing along the balloon's x-axis. In the event that the balloon tilts, and/or wind flows along the balloon's y-axis, there is a much greater chance of instability occurring. The forces found by this analysis show that winds on Earth would be an issue. As stated earlier, an Earth balloon would be largely used indoors, where stability issues due to wind will be minimized. Given the similarity of these results to the other planets, it would be beneficial to test a physical proof-of-concept balloon on Earth under these simulated conditions. This could lend confidence to the simulations and allow for extrapolation to behavior on other planets.

Finally, a proof-of-concept design is considered. The purpose of this analysis is to properly size the balloon for this proof-of-concept so that it will maintain stability through a range of testing. Two balloon shapes were considered in this study. Shape 1 is defined by Equation (1). Shape 2 is defined by Equation (14) below:

$$a = \frac{10}{3}b = \frac{10}{2}c \tag{14}$$

where a, b, and c are the semimajor axis of the ellipsoid. Equation (13) is used to determine the balloon mass, where ρ_b is 0.127 kg/m^2.

Figure 36 shows the range of stable balloon volumes that result from this analysis. For a given payload mass, the balloon volume must be between the upper and lower lines to maintain stability during testing. The two balloon shapes show very similar results, indicating that these results are not very sensitive to balloon shapes near the ones analyzed. For a proposed payload mass of 2 kg, the balloon volume would need to be near 10 m^3.

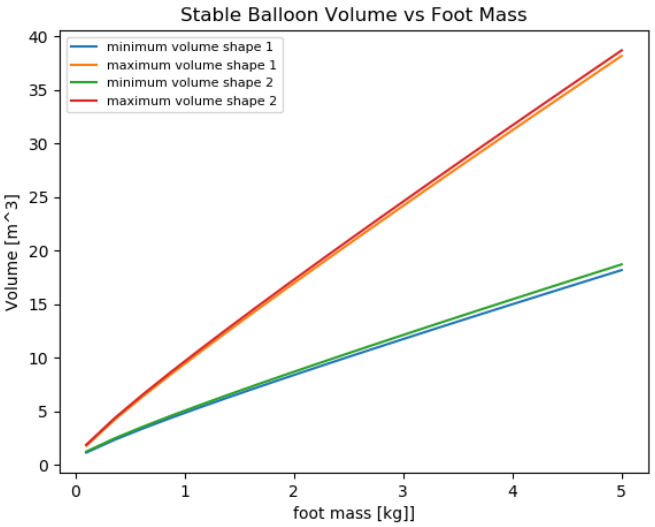


Figure 36 Stable balloon volume at varying payload mass for an Earth proof-of-concept. As foot mass increases, maximum and minimum balloon volumes increase. Maximum balloon volume increases at a faster rate, making the range of stable volume increase with foot mass. Both balloon shapes show nearly identical results.

5. Locomotion

The approach chosen for BALLET to locomote in rugged terrain is described in this section. BALLET is a novel robotic surface mobility system. An investigation into how it locomotes is an important element of the development of the concept. A survey of prior related research was conducted to help inform the development of BALLET's locomotion algorithms. BALLET is a legged robotic system and mobility for legged robotic systems have been investigated for many decades. The approach we propose is to leverage the algorithms developed in prior research performed for mobility for legged robots [Waldron, 1986; Kajita & Espiau, 2008]. As a conservative, simple and low-energy approach, statically stable walking is chosen. There are several levels of software control needed to implement locomotion on BALLET. At the top level is the generation of a path to the desired destination

while negotiating around or stepping over the obstacles and hazards. This element is described in Section 5.1. The output from this element is a set of waypoints that define the path to the destination. The curved paths between waypoints and foot trajectories are calculated to execute the locomotion along the curved segments is described in Section 5.2. Section 5.3 describes the software developed to model BALLET, demonstrate the algorithms for its locomotion and its 3D visualization.

5.1. Obstacle Avoidance Motion Planning

The first step in motion planning is to construct a map of the environment, identify the destination and the obstacles in the field. For BALLET, the maximum step size determines size of obstacle that can be stepped over. Larger obstacles are designated as hazards that have to be avoided. This process is used to identify hazards in the field – if obstacles are smaller than the step size, they do not pose a problem for motion planning but need to be considered in foot placement. The motion planning problem is decoupled into two parts. The first is vehicle motion planning with hazards designated as no-go regions. Sample-based algorithms are widely used in the literature and they can be used to determine a route to the desired destination to generate a motion plan for BALLET. For example the review paper by Karaman & Frazzoli [2011] describe the RRT (left) and RRT* (right) algorithms for optimal motion planning around obstacles.

Given an optimal route defined by a set of waypoints from the motion planner, a foot placement optimizer is then used to plan the steps to be taken to step over or around the obstacles within constraints of placement area available for each foot. Given the waypoints, destination position and map of field designated safe-step regions, based on grade, roughness, terramechanics, etc, a path is constructed to allow stepping through the waypoints to the destination.

5.2. Path Planning and Foot Trajectory Control

A path-planning algorithm was developed to sequence the motion of each foot to traverse along paths generated by the motion planning algorithm. From the overall motion plan generated using the approach described in the previous sub-section, path segments are generated. Each path segment to a local destination will consist of arc motions over the planetary surface. For any local locomotion from an initial position to a destination, an arc of a circle can be constructed, as is illustrated on Figure 37, with a corresponding radius and arc angle. The arc is sub-divided into step-sized segments.

To demonstrate locomotion with BALLET, a simple statically stable gait was identified and implemented on a geometric model. In this procedure, to locomote to a new desired position, a circular arc projected on to a horizontal plane from the current balloon centroid to the new position is constructed (see Figure 37). This arc defines the path the balloon must take. Similarly, arcs are constructed for each foot defining the path each foot takes while maintaining its relative position with respect to the balloon. For any locomotion destination, the foot that has the longest path determines the number of steps to be taken to complete the path using a predetermined maximum step length. The remaining paths are then discretized to have the same number of steps.

Following this initialization procedure, the first foot is lifted vertically a set height, moved horizontally to the same height position above its next step position then lowered down until the foot is on the surface. Foot motion is accomplished by varying the three cable lengths that suspend the foot from the balloon as is illustrated on Figure 38. The foot is also rotated during the step to appropriately match the curvature of the path. The balloon is then moved one-sixth of its step and rotated appropriately by varying all the cables that attach it to the ground to follow the curvature of the path and to follow the slope of the ground beneath. The second foot is then moved, followed again by the balloon and so on until all six feet have taken a step and the balloon has moved a full step.

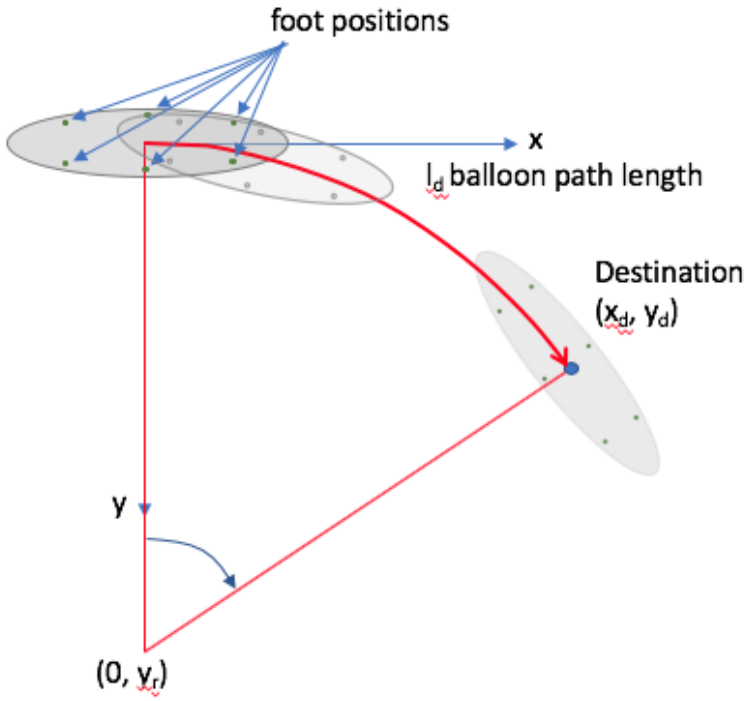


Figure 37 Paths are generated by constructing arcs of circles between the start and destination positions to locomote to a desired destination. For the arc, the length of the path and the rate of turn is determined.

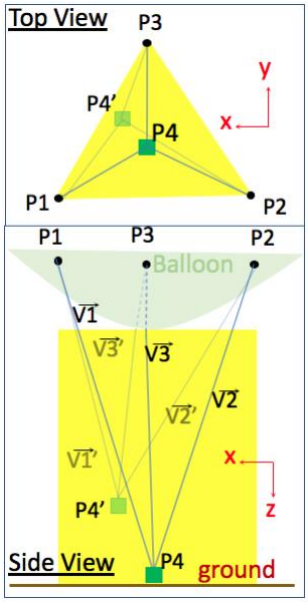


Figure 38 Top and side views of the cables of a foot are shown on this figure. The space that a foot can occupy is shaded in yellow. The foot is positioned a location in that space by differentially controlling the lengths of the foot cables..

This procedure is repeated until the balloon reaches the desired position and orientation. The algorithm accommodates undulating terrain by always positioning the foot a set height above the target step position before being lowered to the ground. A finite-state-machine (FSM) shown on Figure 39 was implemented to control the locomotion algorithm. Having demonstrated lifting one foot at a time, the algorithm was modified to lift two feet at a time as recommended by the force and moment analysis in the previous section. This accomplished by modifying the FSM to move two feet at a time and moving the balloon one-third of a step between the feet motion.

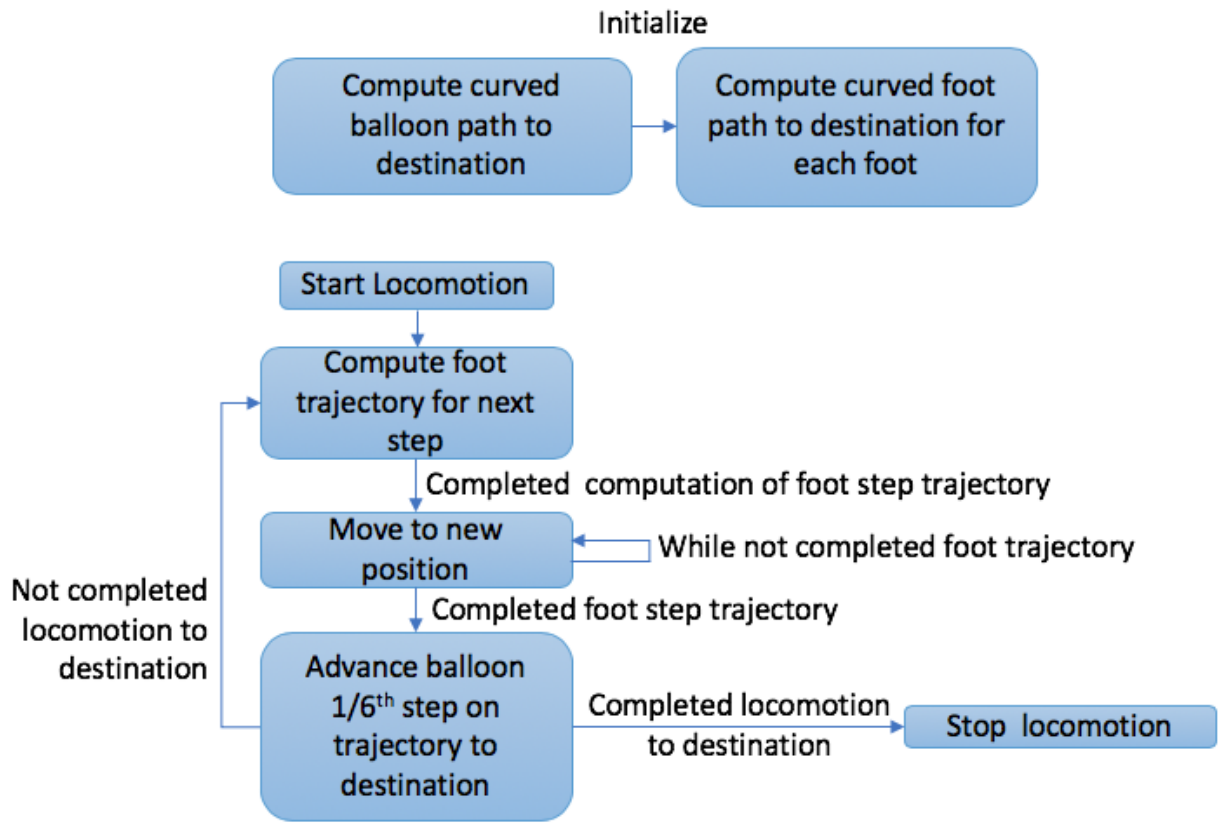


Figure 39 Software control of the locomotion process is achieved by transitioning between functions in the locomotion software algorithm. The transitions are orchestrated by a Finite-state-machine that specifies the conditions for transition and the states to transition to.

A more sophisticated stepping algorithm is possible to optimize the foot placement to step over local and small hazards for example, using the algorithm by Chen, Kumar & Luo [1999]. The foot placement is chosen to maintain stability and step size is adaptively chosen to approach close to then step over hazards that are smaller than the maximum step possible.

5.3. BALLET Model and 3D Visualization

3D computer graphic model and visualization of BALLET and its locomotion was implemented to illustrate its mobility using the open-source Blender visualization engine. The model and visualization software were developed using the Python programming language. The complete source-code for the implementation is listed in Appendix B of this report. The object-oriented software implementation consisted of two parts.

The first part is a parametric kinematic model *Ballet* consisting of a *Balloon*, 6 *Limb*s, each with 3 *LimbCable*s and 1 *Foot*, and a model of the *Terrain*. The BALLET object also contains the finite state machine mobility algorithms for locomoting over the surface. The unified

modeling language (UML) object diagram of this part of the software is illustrated on Figure 40. The second part is the BalletVisualization software to display Ballet and its environment in 3D and orchestrate locomotion, lighting and camera motion to render images in order to create animations of the locomotion. The *ModalTimerOperator* object assists with triggering the refreshing of the 3D rendering of all the objects in the scene during the creation of animation sequences.

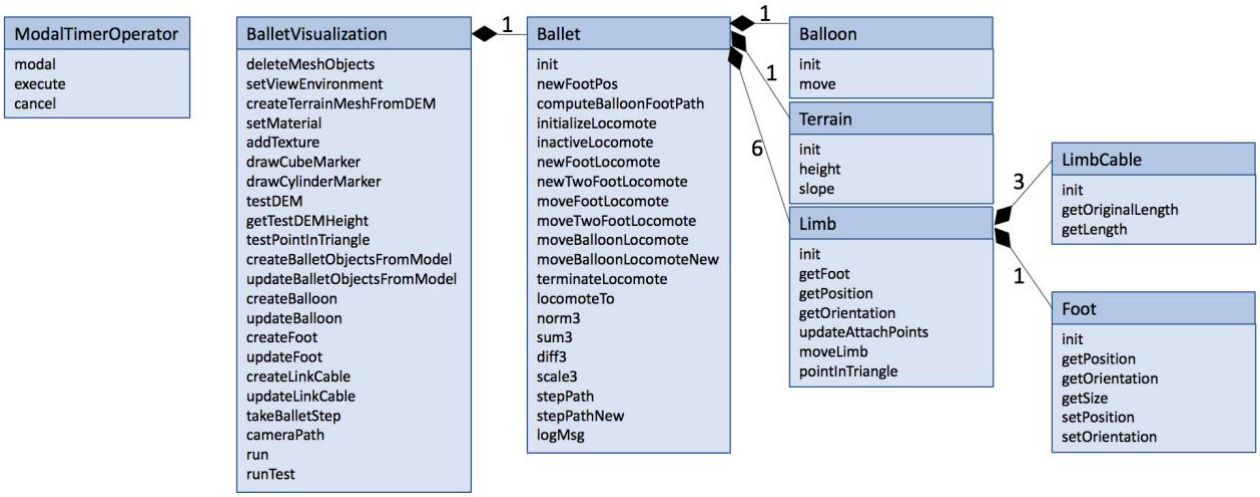


Figure 40 The Unified Modeling Language (UML) diagram for the software implementing the modeling and visualization of the BALLET simulation. Each block in the diagram represents a software object in the object-oriented architecture of the software package.

An example of the 3D visualization for single-step locomotion is shown on Figure 41. The sequence of feet taking steps is front-right, front-left, middle-right, middle-left, back-right and finally back-left.

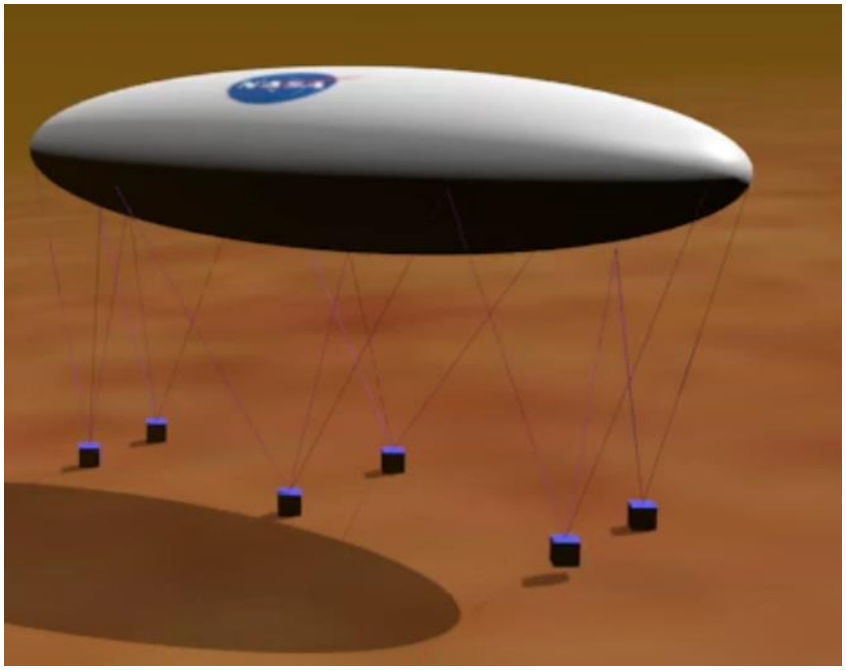


Figure 41 Screenshot from animation of BALLET of single-step locomotion with front-right foot taking a step.

Figure 42 shows a visualization of two-step locomotion. The sequence of feet taking steps is front-right and back-left, middle-right and middle-left, and finally back-right and front-left.

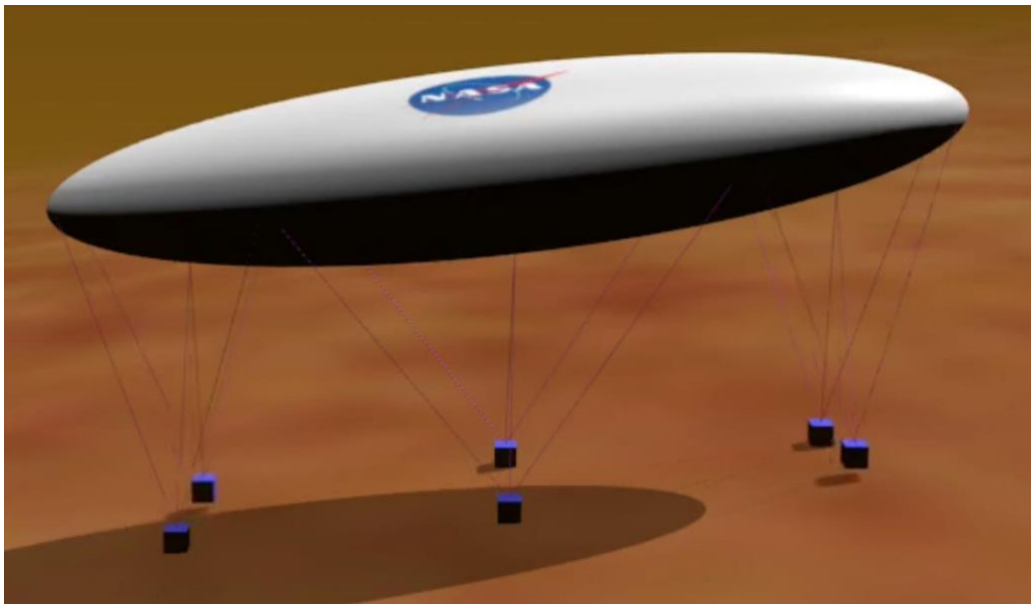


Figure 42 Screenshot from animation of two-step locomotion with front-right and back-left feet taking a step simultaneously.

6. Conclusions

The accomplishments from the Phase I investigation of BALLET are:

- Identification of target planetary bodies, potential science objectives and the instrument suites needed to accomplish the objectives
- Development of the BALLET mission context including the entry, descent and landing on the target planetary bodies and the balloon deployment scenario for getting BALLET into an operational state.
- Analysis of the performance of BALLET under a range of atmospheric and terrain conditions on Mars, Titan and Earth (where testing would occur).
- Development of locomotion algorithms using coordinated limb motions to enable traverse over a range of terrain types.
- Visualization of the operation of BALLET in three dimensions.
- Documentation of the work performed in reports and presentations and publication of a paper to be presented at the 2019 IEEE Aerospace Conference.

The summary results are:

- Compelling science targets for a BALLET mission are RSLs on Mars, lake-shores on Titan and cryo-volcanos on Titan. Instrument suites tailored for these respective science targets have been identified and are feasible for deployment on a BALLET-based mission.
- An entry, descent and landing architecture was developed for BALLET on Mars and Titan. The design of a deployment system for BALLET from the lander was also developed. Power and communications for operations have also been investigated showing a feasible mission architecture for these bodies.
- Of the planetary bodies studied, Titan has the most favorable conditions for BALLET. The combination of a dense atmosphere, low gravity and low surface wind speeds allow use of a RTG power system combined with a science instrument package with a total mass up to 15kg.
- Conditions on Mars are less favorable. With the thin atmosphere, a larger balloon is needed and, with nominal wind speeds of 5 to 10 m/s, drag forces on the balloon are less than the weight of the payloads. However, special precautions to actively anchor BALLET have to be taken under high-wind conditions where wind speeds can reach 26 m/s. Furthermore, the power system for a Mars mission relies on thin-film photovoltaics on the top surface of the balloon. This is currently low-TRL technology that will have to be sufficiently matured for the expected 2030s timeframe of a BALLET mission.

- A two legged locomotion technique is more stable than single legged locomotion. When lifting one payload at a time, a moment is imparted on the balloon that becomes the limiting factor for stability. Lifting opposing legs results in zero moment applied to the balloon. In this case, stability is only limited by the balance of vertical forces due to buoyancy and payload weight. The range of stable payload mass is greater for two-legged locomotion because of this affect.
- When traversing a slope, tilting the balloon with the slope will result in a narrower stable payload mass range. At very high slopes, two-legged locomotion loses its stability benefit over single-legged locomotion.
- Buoyancy will change significantly with atmospheric conditions on Mars. This narrows the range of stable payload mass when compared to a steady climate but does not prohibit a long-term mission.
- Aerodynamic forces will be a major factor in balloon stability. Due to the wind speeds of Mars, this planet may not be feasible for BALLET, with lift and drag forces possibly exceeding the weight of the system. Titan's low wind speed and high atmospheric density make it the most favorable option in dealing with aerodynamic forces. In all cases, facing BALLET such that its smallest cross sectional area is perpendicular to the flow will result in the smallest lift and drag forces possible for the proposed balloon shape.
- A proof-of-concept BALLET on Earth is possible with a moderately sized balloon. For payload masses of 2kg an approximate stable balloon size of $10m^3$ would be necessary.
- Algorithms for motion planning and navigation over rough terrain from prior research of legged robotics systems can be leveraged for BALLET. Coordinated control of the cable system and feet placement for locomotion, a problem unique to BALLET, has been shown to be algorithmically feasible.

Contact science on targets in rugged terrain with BALLET enables direct measurement of water and salt content, enables local temporal and spatial coverage, provides options for multiple measurements with alternative instruments, and potentially enables shallow subsurface sampling. BALLET provides an alternative means to access these sites, expands the range of surface mobility and favorably expands the trade between mobility and science. Cameras placed at multiple locations on the balloon and on the feet, for video logging of BALLET's operations to stream in outreach efforts, will provide a fascinating display for public engagement. Our Phase I investigation showed that this concept has compelling advantages for science exploration at lake-shore and cryo-volcano sites on Titan that remain inaccessible to other surface mobility approaches.

This effort has verified basic principles and formulated the BALLET mission concept and it has led to a new set of critical questions to address in the progression of this concept into a mission. A NASA NIAC Phase II proposal is being submitted to address these questions.

Acknowledgements

The research described in this publication was carried out at the Jet Propulsion Laboratory of California Institute of Technology under contract from the National Aeronautics and Space Administration. This work was supported by the NASA Innovative Advanced Concepts (NIAC) Program.

References

Backes, P., Zimmerman, W., Jones, J., & Gritters, C. (2008, March). Harpoon-based sampling for planetary applications. In *Aerospace Conference, 2008 IEEE* (pp. 1-10). IEEE.

Chen, C. H., Kumar, V., & Luo, Y. C. (1999). Motion planning of walking robots in environments with uncertainty. *Journal of Robotic Systems*, *16*(10), 527-545.

Cordier, D., Barnes, J. W. and Ferriera, A. G. (2013) On the chemical composition of Titan's dry lakebed evaporites. *Icarus*, 226 (2), 1431-1437.

Cottini, V., Nixon, C. A., Jennings, D. E., de Kok, R., Teanby, N. A., Irwin, P. G., & Flasar, F. M. (2012). Spatial and temporal variations in Titan's surface temperatures from Cassini CIRS observations.*Planetary and Space Science*, *60*(1), 62-71.

Cutts, J. A., Nock, K. T., Jones, J. A., Rodriguez, G., Balaram, J., Powell, G. E., & Synott, S. P. (1995). Aerovehicles for planetary exploration.

Elfes, A., Bueno, S. S., Bergerman, M., De Paiva, E. C., Ramos, J. G., & Azinheira, J. R. (2003). Robotic airships for exploration of planetary bodies with an atmosphere: Autonomy challenges. *Autonomous Robots*, *14*(2), 147-164.

Elfes, Alberto, Jeffery L. Hall, Eric A. Kulczycki, Daniel S. Clouse, Ami C. Morfopoulos, James F. Montgomery, Jonathan M. Cameron, Adnan Ansar, and Richard J. Machuzak. "Autonomy architecture for aerobot exploration of Saturnian moon Titan." *IEEE Aerospace and Electronic Systems Magazine* 23, no. 7 (2008).

Hajos, Gregory A., et al. "An overview of wind-driven rovers for planetary exploration." *Proceedings of the 43rd AIAA Aerospace Sciences Meeting and Exhibit, Reno, NV, Jan*. 2005.

Hayes, A. G., Wolf, A. S., Aharonson, O., Zebker, H., Lorenz, R., Kirk, R. L., Paillou, P. et al. (2010) Bathymetry and absorptivity of Titan's Ontario Lacus. *J. Geophys Res. Planets*, 115 (E9), 1-11.

Kajita, S., and B. Espiau. "Legged robots." *Springer handbook of robotics*. Springer, Berlin, Heidelberg, 2008. 361-389.

Karaman, S., & Frazzoli, E. (2011). Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, *30*(7), 846-894.

Lopes, R. M .C., Kirk. R. L., Mitchell, K. L., LeGall, A., Barnes, J. W. et al. (2013) Cryovolcanism on Titan: New results from Cassini RADAR and VIMS. *J. Geophys. Res. Planets*, 118(3), 416-435.

Lopes, R. M .C., Mitchell, K. L., Stofan, E. R., Lunine, J. I., Lorenz, R. et al. (2007b) Cryovolcanic features on Titan's surface as revealed by the Cassini Titan Radar Mapper. *Icarus*, 186(2), 395-412.

Lopes, R. M .C., Mitchell, K. L., Wall. S. D., Mitri, G., Janssen, M. et al. (2007a) The lakes and seas of Titan. *EOS Trans. AGU*, 88 (51), 569-570.

Lorenz R. D. and Zimbelman J. R. (2014) Booming or Singing Dunes. In: Dune Worlds. Springer Praxis Books. Springer, Berlin, Heidelberg.

Lorenz, R. D. and Radebaugh, J. (2009) Global pattern of Titan's dunes: Radar survey from the Cassiniprime mission. *Geophys. Res. Lett.*, 36, L03202, 1-4.

McEwen, A. S., Dundas, C. M., Mattson, S. S., Toigo, A. D. et al. (2014) Recurring slope lineae in equatorial regions of Mars. *Nature Geoscience*, 7, 53-58.

Mitchell, K. L., Barmatz, M. B., Jamieson, C. S., Lorenz, R. D. and Lunine, J. I. (2015) Laboratory measurements of cryogenic liquid alkane microwave absorptivity and implications for the composition of Ligeia Mare, Titan. *Geophys. Res. Lett.*, 42 (5), 1340-1345.

Nayar, H., M. Pauken, M. Cable, M. Hans, "Balloon-based concept vehicle for extreme terrain mobility", IEEE Aerospace Conf., Big Sky, MT, 2019

Nesnas, I. A., Matthews, J. B., Abad-Manterola, P., Burdick, J. W., Edlund, J. A., Morrison, J. C., ... & Anderson, R. C. (2012). Axel and DuAxel rovers for the sustainable exploration of extreme terrains. *Journal of Field Robotics*, *29*(4), 663-685.

Ojha, L., Wilhelm, M. B., Murchie, S. L., McEwen, A. F. et al. (2015) Spectral evidence for hydrated salts in recurring slope lineae on Mars. *Nature Geoscience*, 8, 829-832.

Oleson, S. R., Lorenz, R. D. and Paul, M. V. (2015) Titan Submarine: Exploring The Depths of Kraken Mare. AIAA SPACE Forum, 31 Aug-2 Sep 2015, Pasadena, California. DOI: 10.2514/6.2015-4445.

Oren, A., Bardavid, R. E. and Mana, L. (2014) Perchlorate and halophilic prokaryotes: Implications for possible halophilic life on Mars. *Extremophiles*, 18(1), 75-80.

Poggiali, V., Mastrogiuseppe, M., Hayes, A. G., Seu, R., Birch, S. P. D., Lorenz, R., Grima, C. and Hofgartner, J. D. (2016) Liquid-filled canyons on Titan. *Geophys. Res. Lett.*, 43 (15), 7887-7894.

Radebaugh, J., Lorenz, R. D., Lunine, J. I., Wall, S. D., Boubin, G., Reffet, E., et al. (2008) Dunes on Titan observed by Cassini RADAR. *Icarus*, 194(2), 690-703.

Seeni, Aravind, Bernd Schäfer, and Gerd Hirzinger. (2010). "Robot mobility systems for planetary surface exploration–state-of-the-art and future outlook: a literature survey." *Aerospace Technologies Advancements*. InTech, 2010.

Space Studies Board, Vision and Voyages for Planetary Science in the Decade 2013-2022. National Academies Press, 2012.

Stofan, E. R., Elachi, C., Lunine, J. I., Lorenze, R. D. et al. (2007) The lakes of Titan. *Nature*, 445, 61-64.

Stofan, E. R., Lorenz, R. D., Lunine, J. I., Aharonson, O., Bierhaus, B. et al. (2010) Titan Mare Explorer (TiME): First In Situ Exploration of an Extraterrestrial Sea. Astrobiology Science Conference, abstract no. 5270.

Waldron KE. Force and motion management in legged locomotion. IEEE Journal on Robotics and Automation. 1986 Dec;2(4):214-20.

Webster, G., Brown, D., and Cantillo, L. (2016) "Rover Takes on Steepest Slope Ever Tried on Mars" URL: https://www.nasa.gov/feature/jpl/rover-takes-on-steepest-slope-ever-tried-on-mars

Wilcox, Brian H., Todd Litwin, Jeff Biesiadecki, Jaret Matthews, Matt Heverly, Jack Morrison, Julie Townsend, Norman Ahmad, Allen Sirota, and Brian Cooper. "ATHLETE: A cargo handling and manipulation robot for the moon." *Journal of Field Robotics* 24, no. 5 (2007): 421-434.

Appendix A: BALLET Stability Analysis On Titan

The method used here to determine the feasibility and stability of the BALLET balloon is to quantify the upper and lower bounds of the mass of the feet. If the foot mass is too low, the balloon is at risk of sliding or being lifted off the ground with gusts of wind. With a foot mass that is too great, the balloon may tilt or become unstable when lifting a leg. Finding the acceptable range of the mass of the foot will help maintain mission safety while providing requirements for the scientific instruments that can be chosen.

Initialize Analysis

Here an additional notebook is loaded. This additional notebook contains the implementation of the functions used in this analysis, as well as initializing constants.

Some of the constants initialized are shown in the table below:

Property	**Earth**	**Mars**	**Titan**
Gravitational accel ($\frac{m}{s^2}$)	9.807	3.71	1.352
Surface atm density ($\frac{kg}{m^3}$)	1.217	0.020	5.280
Helium surface density ($\frac{kg}{m^3}$)	0.178	0.002	0.728
Nominal wind speed ($\frac{m}{s}$)	7.000	10.000	1.000
Drag coeff	0.500	0.500	0.500
Foot mass (kg)	1.0	1.0	1.0
Added mass on balloon (kg)	0.5	0.1	45.000
Needed balloon buoyancy force (N)	19.614	5.936	70.980
Balloon volume (m^3)	1.925	88.134	11.534
Balloon diameter (m)	1.543	5.521	2.803
Balloon x-section area (m^2)	1.871	23.943	6.172
Nominal wind drag force (N)	27.891	11.972	8.147
Gravity anchoring force (N)	44.132	16.695	30.420

	Earth	**Mars**	**Titan**	**He**
Atmospheric Molecular Weight	28.97	43.34	29.0	4.0

In [1]:
```
%run BALLET_Functions.ipynb
```

Choose Balloon Geometry

Here we define the balloon geometry that will be used for this analysis. We do this based on the desired volume, and assumed ratios of semimajor axis. In this case a=2**b=4**c

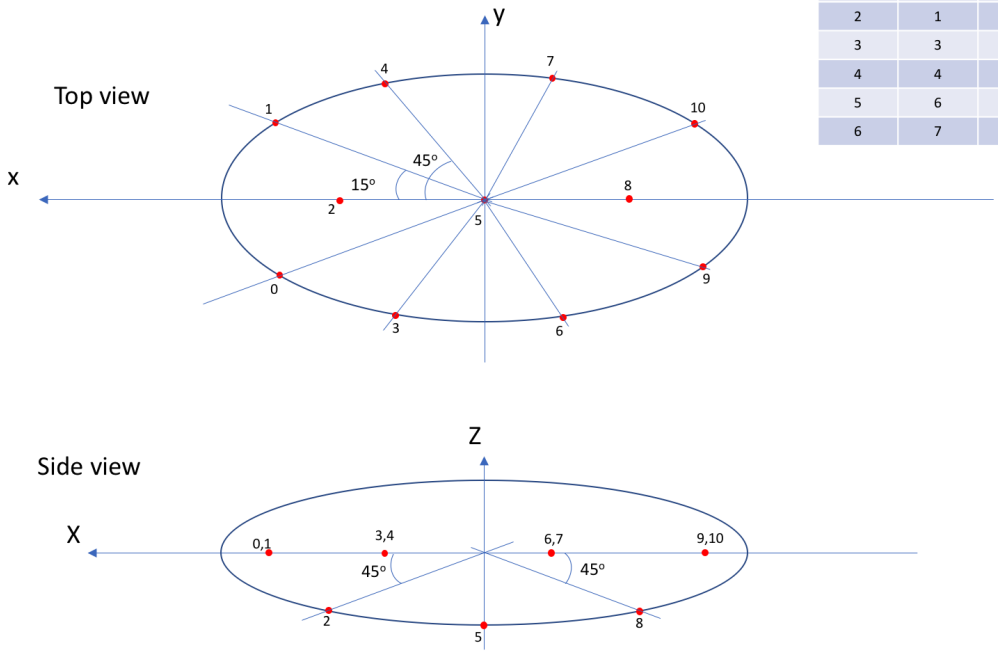
In [2]:
```
temp = (vol_titan*(3/4)/np.pi)
a = (temp*8.0) ** (1.0/3.0)
b = temp ** (1.0/3.0)
c = (temp/8.0) ** (1.0/3.0)
balloon_height = 7.0
print("a = "+str(a)+" m")
print("b = "+str(b)+" m")
print("c = "+str(c)+" m")
```

```
a = 2.803241032880424 m
b = 1.4016205164402122 m
c = 0.7008102582201061 m
```

Find Connection Points and Foot Locations

Here we find the cable connection points on the balloon, based on the given diagram of their placement:

BALLET Cable attachment points



Payload attach points

Limb	Attach 1	Attach 2	Attach3
1	0	2	3
2	1	4	2
3	3	5	6
4	4	7	5
5	6	8	9
6	7	10	8

This analysis finds 3 of the connection points and then uses symetry to find the others

In [3]:
```
# Find balloon connection points based on geometry
p1 = find_position_on_ellipsoid_z_0(15, a, b, c)
p0 = p1[:]
p0[1] *= -1.0
p9 = p0[:]
p9[0] *= -1.0
p10 = p1[:]
p10[0] *= -1.0

p4 = find_position_on_ellipsoid_z_0(45, a, b, c)
p3 = p4[:]
p3[1] *= -1.0
p6 = p3[:]
p6[0] *= -1.0
p7 = p4[:]
p7[0] *= -1.0

# p2 & p8
p2 = find_position_on_ellipsoid_y_0(45, a, b, c)
p2[2] *= -1.0
p8 = p2[:]
p8[0] *= -1.0

# p5 is on the bottom center
p5 = [0.0, 0.0, -c]

# Assemble the points into groups by leg
points = [p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, p10]
legs = [[p0, p2, p3], [p1, p4, p2], [p3, p5, p6], [p4, p7, p5], [p6, p8, p9], [p7, p10, p8]]

# Find the foot location as the average of point locations
feet = [get_foot_x_y(x) for x in legs]

# Plot the geometry
plot_balloon_geometry(points, feet, legs, a, b, c)
plt.xlabel("m")
plt.ylabel("m");
```

Finding Buoyant Force

The buoyant force of the balloon is assumed to be at the center of the ellipsoid. The force is equal to the weight of the air displaced by the balloon. In this analysis, the boyant force is considered to be the total upward force after subtracting the weight of mass added to the balloon

$F_b = (\rho_{atm} - \rho_{He})Vg - m_{add}g$

$V = \frac{4}{3}\pi abc$

ρ_{atm} is the atmospheric density, ρ_{He} is the density of helium at the planet's surface, g is the gravitational acceleration, m_{add} is mass added to the balloon, V is the volume of the balloon, and a, b, and c are the semimajor axis lengths of the ellipsoid.

In [4]:
```
F_b = (rho_titan-rho_helium_titan)*vol_titan*g_titan - added_mass_titan*g_titan # Boyant force (on tit
an)

print("Volume = "+str(vol_titan)+" m^3")
print("Boyant Force = "+str(F_b)+" N")
```

```
Volume = 11.534 m^3
Boyant Force = 10.143742336000017 N
```

Bounding The Foot Mass

In order to find the bounds of the foot mass, force and moment balance equations must be used. To simplify the process, each 'leg' is treated as a single cable, rather than three. The single cable location is taken to be the average position of the three connection points of that leg.

In [5]:
```
# Assume legs have one connection at the center of all 3 cables
# Find the leg connection points
L1 = (np.array(points[0])+np.array(points[2])+np.array(points[3]))/3
L2 = (np.array(points[1])+np.array(points[4])+np.array(points[2]))/3
L3 = (np.array(points[3])+np.array(points[5])+np.array(points[6]))/3
L4 = (np.array(points[4])+np.array(points[7])+np.array(points[5]))/3
L5 = (np.array(points[6])+np.array(points[8])+np.array(points[9]))/3
L6 = (np.array(points[7])+np.array(points[10])+np.array(points[8]))/3
connection_points = [L1, L2, L3, L4, L5, L6]
```

The foot is assumed to be directly below this position, such that all force vectors along the cables are parallel to the z-axis. The boyant force of the balloon is assumed to be at the center of the ellipsoid pointing along the z-axis. The weight of mass added to the balloon is assumed to be at the center of the ellipsoid, directly counteracting the boyant force.

Minimum Foot Mass

The minimum mass of the feet can be found through performing a force balance in the z direction. The boyant force must be completely counteracted by the weight of the feet. As such, the sum of the weight of all feet must be equal or greater to the boyant force. This analysis assumes all feet will be the same mass, so the following equation defines the minimum mass of a single foot.

$m_{min} = \frac{F_b}{6g}$

In [6]:
```
m_min = F_b/(6*g_titan)
print("Minimum Foot Mass = "+ str(m_min)+" kg")
```

```
Minimum Foot Mass = 1.2504613333333354 kg
```

Maximum Foot Mass

The maximum mass of an individual foot is limited by the balance of moments when lifting one or two feet. At the maximum mass, one or more cables will go to 0 N tension. If any additional mass was added, the cable would buckle due to its inability to resist compressive loads, and the balloon would tilt. As such, in order to solve for the maximum mass an optimization techinique is used, with the constraints that all cables remain in tension, and the tension in the cables remains less than or equal to the weight of a single foot.

Before optimizing for maximum mass, force and moment balance equations are used to find the force required at the connection points to remain balloon stability. Given the geometry of the problem, the sum of forces in the x and y directions do not provide any information. Similarly, the sum of moment about the z-axis is redundant. This leaves three equations:

$\sum F_z = 0$

$\sum M_x = 0$

$\sum M_y = 0$

If one foot is lifted, this leaves 3 equations, and 5 unknowns. Similarly, if two feet are lifted, there are 3 equations and 4 unknowns. This defines an underdetermined system with infinite solutions, requiring linear programming techniques to find a solution. Given the nature of the problem, a least squares solution is chosen. This method finds the solution where the sum of forces at the connection points is at a minimum, while still satisfying the constraints. For details on this implementation, see the accompanying notebook.

Lifting One Foot

Due to symetry, two cases must be tested. First, lifting leg 1:

In [7]:
```
lift_leg_1_max_mass = find_maximum_foot_mass_legs_lifted(connection_points, F_b, g_titan, [0])
lift_leg_1_forces = lift_legs(connection_points, F_b, lift_leg_1_max_mass*g_titan, [0])
plot_forces_bar_graph(
    connection_points,
    lift_leg_1_forces,
    [0],
    lift_leg_1_max_mass,
    a, b, c, g_titan,
    "Forces On Balloon With Leg 1 Lifted",
    "Maximum Stable Foot Mass = "+str(round(lift_leg_1_max_mass[0],2))+" kg"
)
```

Now lifting leg 3:

In [8]:
```
lift_leg_3_max_mass = find_maximum_foot_mass_legs_lifted(connection_points, F_b, g_titan, [2])
lift_leg_3_forces = lift_legs(connection_points, F_b, lift_leg_3_max_mass*g_titan, [2])
plot_forces_bar_graph(
    connection_points,
    lift_leg_3_forces,
    [2],
    lift_leg_3_max_mass,
    a, b, c, g_titan,
    "Forces On Balloon With Leg 3 Lifted",
    "Maximum Stable Foot Mass = "+str(round(lift_leg_3_max_mass[0],2))+" kg"
)
```

Now that the two cases have been solved, the maximum allowable foot mass is the minimum of the two found maximums

In [9]:
```
lift_one_leg_maximum_mass = min([lift_leg_3_max_mass, lift_leg_1_max_mass])[0]
print("Foot Mass Bounds - 1 Leg Lifted at a Time")
print("Minimum Mass = "+str(m_min)+" kg")
print("Maximum Mass = "+str(lift_one_leg_maximum_mass)+" kg")
```

```
Foot Mass Bounds - 1 Leg Lifted at a Time
Minimum Mass = 1.2504613333333354 kg
Maximum Mass = 2.886571822296406 kg
```

Lifting Two Feet

Due to symetry, two cases must be tested. First, lifting legs 1 and 6:

In [10]:
```
lift_legs_1_6_max_mass = find_maximum_foot_mass_legs_lifted(connection_points, F_b, g_titan, [0,5])
lift_legs_1_6_forces = lift_legs(connection_points, F_b, lift_legs_1_6_max_mass*g_titan, [0,5])
plot_forces_bar_graph(
    connection_points,
    lift_legs_1_6_forces,
    [0,5],
    lift_legs_1_6_max_mass,
    a, b, c, g_titan,
    "Forces On Balloon With Legs 1 & 6 Lifted",
    "Maximum Stable Foot Mass = "+str(round(lift_legs_1_6_max_mass[0],2))+" kg"
)
```

Now lifting legs 3 and 4:

In [11]:
```
lift_legs_3_4_max_mass = find_maximum_foot_mass_legs_lifted(connection_points, F_b, g_titan, [2,3])
lift_legs_3_4_forces = lift_legs(connection_points, F_b, lift_legs_3_4_max_mass*g_titan, [2,3])

plot_forces_bar_graph(
    connection_points,
    lift_legs_3_4_forces,
    [2,3],
    lift_legs_3_4_max_mass,
    a, b, c, g_titan,
    "Forces On Balloon With Legs 3 & 4 Lifted",
    "Maximum Stable Foot Mass = "+str(round(lift_legs_3_4_max_mass[0],2))+" kg",
    1.0
)
```

Now that the two cases have been solved, the maximum allowable foot mass is the minimum of the two found maximums

In [12]:
```
lift_two_legs_maximum_mass = min([lift_legs_1_6_max_mass, lift_legs_3_4_max_mass])[0]
print("Foot Mass Bounds - 2 Legs Lifted at a Time")
print("Minimum Mass = "+str(m_min)+" kg")
print("Maximum Mass = "+str(lift_two_legs_maximum_mass)+" kg")
```

```
Foot Mass Bounds - 2 Legs Lifted at a Time
Minimum Mass = 1.2504613333333354 kg
Maximum Mass = 3.7513840000000047 kg
```

As seen in the above results, when opposite legs are lifted, they completely cancel out the moment acting on the balloon. This means that for this case, only the vertical force balance equations come into play. When using a foot mass equivalent to that of the maximum allowable foot mass when lifting a single leg, it can be seen that the balloon is actually more stable. This is becauses none of the cables go slack, despite twice the mass being lifted simultaneously.

In [13]:
```
lift_two_legs_with_one_leg_mass_forces = lift_legs(connection_points, F_b, [lift_one_leg_maximum_mass*
g_titan], [0,5])
plot_forces_bar_graph(
    connection_points,
    lift_two_legs_with_one_leg_mass_forces,
    [0,5],
    [lift_one_leg_maximum_mass],
    a, b, c, g_titan,
    "Forces On Balloon With Legs 1 & 6 Lifted",
    "Foot Mass = "+str(round(lift_one_leg_maximum_mass,2))+" kg"
)
```

Lifting Feet on a slope

When moving on a slope it is possible that the pitch of the balloon will change with the slope of the ground it is climbing. First, it is demonstrated that a change in the pitch of the balloon does not effect its center of bouyancy, bouyant force, or create a moment on the balloon. This is done in 2D due to the symmetry of the ellipsoid.

In [14]:
```
#Choose a pitch of 30 degrees
pitch = np.pi/6.0

#Choose a number of points to outline the ellipse
samples = 1000

#Create a set of points outlining an ellipse rotated by the chosen pitch
t = np.linspace(0, 2*np.pi, samples)
Ell = np.array([a*np.cos(t) , c*np.sin(t)])
nCk = np.array([[np.cos(pitch) , -np.sin(pitch)],[np.sin(pitch) , np.cos(pitch)]])
Ell_rot = np.zeros((2,Ell.shape[1]))
for i in range(Ell.shape[1]):
    Ell_rot[:,i] = np.dot(nCk,Ell[:,i])

#Loop through all sets of two points
#Find the force and moment about the center of the ellipse due to the area between the two points
#Sum these forces and moments to find total boyant force and moment
max_height = np.max(Ell_rot[1,:])
Fb2 = np.array([0.0,0.0])
Moment = 0.0
for i in range(Ell_rot.shape[1]):

    #Get vector from point 0 to point 1
    #Overflow to the first point when the last point is reached
    p1 = np.array([Ell_rot[0][i], Ell_rot[1][i]])
    if(i == Ell_rot.shape[1]-1):
        p2 = np.array([Ell_rot[0][0], Ell_rot[1][0]])
    else:
        p2 = np.array([Ell_rot[0][i+1], Ell_rot[1][i+1]])

    #Vector between points
    p1_p2 = p2-p1

    #Point between p1 and p2
    center = p1+(p1_p2/2.0)

    #Distance between points
    dist = np.linalg.norm(p1_p2)

    #Perpendicular unit vector
    perp = np.array([-1.0*p1_p2[1], p1_p2[0]])/dist

    #Force magnitude
    Fmag = np.abs(rho_titan*g_titan*(center[1]-max_height)*dist)

    #Force vector
    Fvec = perp*Fmag

    #Add to Force tally
    Fb2 = Fb2+Fvec

    #Now find moment due to this force
    center3d = np.array([center[0], 0.0, center[1]])
    F3d = np.array([Fvec[0], 0.0, Fvec[1]])
    Torque = np.cross(center3d, F3d)
    Moment = Moment+Torque[1]

#Find the area of the ellipse
A = a*c*np.pi

#Subtract the mass of the air in the balloon
Fb2 = Fb2-np.array([0.0, rho_helium_titan*A*g_titan])

#Find bouyant force per meter for 2d ellipse
Fb1 = (rho_titan-rho_helium_titan)*A*g_titan

#Print out the total force and moment
print("Integrated Bouyant Force With Pitch = ["+str(Fb2[0])+", 0.0,"+str(Fb2[1])+"] N/m")
print("Bouyant Force From Archimedes Principle = "+str(Fb1)+" N/m")
print("Integrated Bouyant Torque With Pitch = "+str(Moment)+" Nm/m")
print("Bouyant Torque From Archimemdes Principle = 0.0 Nm/m")
```

```
Integrated Bouyant Force With Pitch = [2.6344531755729293e-14, 0.0,37.98274854134601] N/m
Bouyant Force From Archimedes Principle = 37.983039009168884 N/m
Integrated Bouyant Torque With Pitch = -2.8325540147513044e-14 Nm/m
Bouyant Torque From Archimemdes Principle = 0.0 Nm/m
```

As shown above, a change in pitch does not produce any forces out of line with the z-axis. The force in the z-axis is identical to that found with Archimedes principle. Also, there is no additional moment to be accounted for in the analysis.

Although this change in pitch does not effect the bouyant force, it will effect both the maximum stable foot mass and which cables go slack at this maximum mass. If all cable connection points were on the x-y plane, the change in pitch would not effect which cables go slack. Due to an offset of the connection points in the z-direction, the change in pitch biases the moment arm lengths, changing the cables that go slack. Below are plots of each of the previous lifted-foot analyses with a pitch of 30 degrees.

In [15]:
```
pitch = 30.0*np.pi/180.0

lift_leg_1_max_mass_pitch = find_maximum_foot_mass_legs_lifted(connection_points, F_b, g_titan, [0], p
itch)
lift_leg_1_forces_pitch = lift_legs(connection_points, F_b, lift_leg_1_max_mass_pitch*g_titan, [0], pi
tch)
plot_forces_bar_graph(
    connection_points,
    lift_leg_1_forces_pitch,
    [0],
    lift_leg_1_max_mass_pitch,
    a, b, c, g_titan,
    "Forces On Balloon With Leg 1 Lifted",
    "Maximum Stable Foot Mass = "+str(round(lift_leg_1_max_mass_pitch[0],2))+" kg, 30 degree pitch"
)

lift_leg_3_max_mass_pitch = find_maximum_foot_mass_legs_lifted(connection_points, F_b, g_titan, [2], p
itch)
lift_leg_3_forces_pitch = lift_legs(connection_points, F_b, lift_leg_3_max_mass_pitch*g_titan, [2], pi
tch)
plot_forces_bar_graph(
    connection_points,
    lift_leg_3_forces_pitch,
    [2],
    lift_leg_3_max_mass_pitch,
    a, b, c, g_titan,
    "Forces On Balloon With Leg 3 Lifted",
    "Maximum Stable Foot Mass = "+str(round(lift_leg_3_max_mass_pitch[0],2))+" kg, 30 degree pitch"
)

lift_legs_1_6_max_mass_pitch = find_maximum_foot_mass_legs_lifted(connection_points, F_b, g_titan, [0,
5], pitch)
lift_legs_1_6_forces_pitch = lift_legs(connection_points, F_b, lift_legs_1_6_max_mass_pitch*g_titan, [
0,5], pitch)
plot_forces_bar_graph(
    connection_points,
    lift_legs_1_6_forces_pitch,
    [0,5],
    lift_legs_1_6_max_mass_pitch,
    a, b, c, g_titan,
    "Forces On Balloon With Legs 1 & 6 Lifted",
    "Maximum Stable Foot Mass = "+str(round(lift_legs_1_6_max_mass_pitch[0],2))+" kg, 30 degree pitch"
)

lift_legs_3_4_max_mass_pitch = find_maximum_foot_mass_legs_lifted(connection_points, F_b, g_titan, [2,
3], pitch)
lift_legs_3_4_forces_pitch = lift_legs(connection_points, F_b, lift_legs_3_4_max_mass_pitch*g_titan, [
2,3], pitch)
plot_forces_bar_graph(
    connection_points,
    lift_legs_3_4_forces_pitch,
    [2,3],
    lift_legs_3_4_max_mass_pitch,
    a, b, c, g_titan,
    "Forces On Balloon With Legs 3 & 4 Lifted",
    "Maximum Stable Foot Mass = "+str(round(lift_legs_3_4_max_mass_pitch[0],2))+" kg, 30 degree pitch"
,
)
```

Below are graphs of the maximum stable foot mass over pitch angles ranging from 0 to 50 degrees for locomotion with both one and two feet

In [16]:
```
#Number of pitches to plot
samples = 10

#Pitches
pitches = np.linspace(0,50.0*np.pi/180.0, samples)

#Loop through all pitches, finding max stable mass at each pitch and configuration
leg_1_max_mass_lifted_pitch = []
leg_3_max_mass_lifted_pitch = []
legs_1_6_max_mass_lifted_pitch = []
legs_3_4_max_mass_lifted_pitch = []
for ii in range(10):
    leg_1_max_mass_lifted_pitch.append(find_maximum_foot_mass_legs_lifted(connection_points, F_b, g_ti
tan, [0], pitches[ii]))
    leg_3_max_mass_lifted_pitch.append(find_maximum_foot_mass_legs_lifted(connection_points, F_b, g_ti
tan, [2], pitches[ii]))
    legs_1_6_max_mass_lifted_pitch.append(find_maximum_foot_mass_legs_lifted(connection_points, F_b, g
_titan, [0,5], pitches[ii]))
    legs_3_4_max_mass_lifted_pitch.append(find_maximum_foot_mass_legs_lifted(connection_points, F_b, g
_titan, [2,3], pitches[ii]))

plot_max_mass_versus_pitch(
    pitches,
    leg_1_max_mass_lifted_pitch,
    leg_3_max_mass_lifted_pitch,
    "Leg 1",
    "Leg 3",
    "Maximum Stable Foot Mass",
    "Single Leg Locomotion",
    2.0,
    3.8
)

plot_max_mass_versus_pitch(
    pitches,
    legs_1_6_max_mass_lifted_pitch,
    legs_3_4_max_mass_lifted_pitch,
    "Legs 1 & 6",
    "Legs 3 & 4",
    "Maximum Stable Foot Mass",
    "Dual Leg Locomotion",
    2.0,
    3.8
)
```

As seen in the above graphs, for both mobility modes as the slope of the balloon increases, the maximum stable foot mass decreases. At very high angles, the dual leg gait gives very little advantage over the single leg gait in terms of stability.

Drag Force Analysis

Finding Reynolds Number

This will tell us if there will be turbulent or laminar flow over the ellipsoid. A Reynolds number less than 2000 is laminar flow. Reynolds number greater than 4000 is turbulent flow. Between 2000 and 4000 is transition flow.

$$Re = \frac{\rho V L}{\mu}$$

Where ρ is the density of the fluid, V is the velocity of the fluid, L is the characterstic length of the flow, and μ is the dynamic viscosity of the fluid.

Dynamic viscosity is the most difficult parameter to obtain. To find this parameter, Sutherland's Law is used with two coefficients. Sutherland's Law requires only temperture to estimate dynamic viscosity, which is 94.15 K on Titan on average.

In [17]:
```
wind_speed_titan = 1.0
mu = dynamic_viscosity(94.15)
Re = (rho_titan * wind_speed_titan * max([a, b, c])*2.0)/mu
print("Re = "+str(Re))
```

```
Re = 4546067.0108338045
```

Given the large reynolds number, turbulent flow is expected.

Estimating Drag Force

Drag force is given as

$F_d = \frac{1}{2} C_d \rho A V^2$

Where C_d is the drag coefficient, A is the cross-sectional aread, ρ is the density of the fluid, and V the fluid velocity. A nominal coefficient of drag for an ellipsoid is given as 0.5.

In [18]:
```
Cd = 0.5

#Frontal area
A_x = np.pi*b*c

#Side area
A_y = np.pi*a*c

Fd_x = (1.0/2.0)*Cd*rho_titan*A_x*wind_speed_titan**2
Fd_y = (1.0/2.0)*Cd*rho_titan*A_y*wind_speed_titan**2

print("Nominal Drag coefficient:")
print("Fd_x = "+str(Fd_x)+" N")
print("Fd_y = "+str(Fd_y)+" N")
```

```
Nominal Drag coefficient:
Fd_x = 4.0733778744195055 N
Fd_y = 8.14675574883901 N
```

Estimating Aerodynamic Forces in OpenFoam

The open source computational fluid dynamics program OpenFoam was used to estimate lift and drag forces on the BALLET balloon. OpenFoam's PisoFoam solver was used, which finds the transient behavior of incompressible turbulent flow. To simplify this analysis, no turbulence models were considered. It is likely that this simplification also results in the worst case aerodynamic effects due to pressure drag dominating skin friction drag for bluff body shapes like the BALLET balloon. For flow on titan, the following initial conditions were used: Freestream Velocity = 1 m/s, rho = 5.28 kg/m^3, and kinematic viscocity = 0.000001246212121 m^2/s. Kinematic viscosity was estimated as that of nitrogen at titan surface temperatures. This assumption is made due to titan's atmosphere which is predicted to be 95-97% nitrogen.

OpenFoam's blockMesh and snappyHexMesh tools were used with an STL model of the balloon to create a mesh for the simulation. For simulations measuring drag, symmetry was used on two planes to reduce the problem's complexity. Simulations measuring lift used symmetry on one plane, allowing for the balloon to tilt. Simulation flow inlets were given freestream velocity and zero gradient pressure boundary conditions. Flow outlets were given zero gradient velocity and zero pressure boundary conditions. Note that for incompressible flow, the pressure differential drives flow, not the pressure value. These boundary conditions result in a steady flow at the desired velocity. The boundary conditions of the balloon are no-slip velocity and zero gradient pressure, allowing for a boundary layer to form on the balloons surface. The results are below:

In [19]:
```
# Drag facing wind
dragFrontFile = "data/titan_drag_front.dat"
dragSideFile = "data/titan_drag_side.dat"
liftFrontFile = "data/titan_lift_front.dat"
liftSideFile = "data/titan_lift_side.dat"

plot_openfoam_drag(dragFrontFile, "1m/s wind in the x direction", "Drag Force", 0, 15)
plot_openfoam_drag(dragSideFile, "1m/s wind in the y direction", "Drag Force", 0, 15)
plot_openfoam_lift(liftFrontFile, "1m/s wind in the x direction, 10deg angle of attack", "Aerodynamic
 Forces", 0, 15)
plot_openfoam_lift(liftSideFile, "1m/s wind in the y direction, 10deg angle of attack", "Aerodynamic F
orces", 0, 15)
```

BALLET Analysis Functions

This notebook houses the functions used in the BALLET Analysis notebook. It is meant to separate the technical details from the results of the analysis.

Geometry

The functions in this section concern the geometry of the balloon and its legs/feet.

Find Position On Ellipsoid Z = 0

This function finds the location of a point on an ellipsoid at z=0 given an angle from the x axis as in the diagram

In [12]:
```
def find_position_on_ellipsoid_z_0(angle, a, b, c):
    y = np.sqrt(1.0 / ((1.0/((np.tan(np.radians(angle))**2)*(a**2))) + 1.0/b**2))
    return [y/np.tan(np.radians(angle)), y, 0.0]
```

Find Position On Ellipsoid Y = 0

This function finds the location of a point on an ellipsoid at y=0 given an angle from the x axis as in the diagram

In [13]:
```
def find_position_on_ellipsoid_y_0(angle, a, b, c):
    z = np.sqrt(1.0 / ((1.0/((np.tan(np.radians(angle))**2)*(a**2))) + 1.0/c**2))
    return [z/np.tan(np.radians(angle)), 0.0, z]
```

Get Foot X Y

Gets the x,y location of a foot given the three cable connection points. This always assumes the foot is at the center of this triangle by averaging the x,y locations of the cable connections.

In [14]:
```
def get_foot_x_y(points):
    return [ np.mean(x) for x in [ [points[0][y], points[1][y], points[2][y]] for y in range(0,2)]]
```

Static Analysis

Functions used for force and moment balancing.

Get Foot Mass Bounds

Find bounds of the foot mass based purely on a sum of moments in the z-axis. This informs the initial guess of the optimization technique.

In [15]:
```
#Bounds the mass of a foot based on forces in z direction only
#No torque is taken into account
#This is to get initial guesses and bounds for maximum weight of a foot
def get_foot_mass_bounds(boyant_force, g, num_lifted_feet):

    #Divide it evenly by the number of feet to get min weight
    FFootMin = boyant_force/6.0

    #Calculate minimum mass from Weight of foot
    FMassMin = FFootMin/g

    #Divide by number of lifted feet for max weight
    FFootMax = boyant_force/num_lifted_feet

    #Calculate maximum mass from Weight of foot
    FMassMax = FFootMax/g

    #Return the foot mass
    return [FFootMin, FFootMax]
```

Lift Legs

This is the least squares solution of the forces on the cables when at least one foot is lifted

In [27]:
```
def lift_legs(positions, boyant_force, footWeight, legs_lifted, pitch = 0.0):

    #footWeight must be given as an array for the optimizer
    foot_weight = footWeight[0]

    #Create rotation matrix from pitch (pitch is about y axis)
    nCk = np.array([
           [np.cos(pitch) , 0.0, -1.0*np.sin(pitch)],
           [0.0           , 1.0, 0.0          ],
           [np.sin(pitch), 0.0, np.cos(pitch)]])

    #Rotate the positions
    rotated_positions = []
    for ii in range(len(positions)):
        rotated_positions.append(np.dot(nCk, positions[ii]))

    #Define A from Sum of forces and Sum of torques = 0
    #Row 1 is sum of forces
    #Row 2 and 3 are sum of torques in x and y respectively
    row1 = []
    row2 = []
    row3 = []
    for ii in range(0,6):
        if ii in legs_lifted:
            continue

        row1.append(1.0)
        row2.append(rotated_positions[ii][1])
        row3.append(rotated_positions[ii][0])

    #Create the A matrix
    A = np.matrix([row1, row2, row3], dtype=float)

    #Define b
    b = [boyant_force-len(legs_lifted)*foot_weight, 0.0, 0.0]
    for leg_lifted in legs_lifted:
        b[1] += -1.0*foot_weight*rotated_positions[leg_lifted][1]
        b[2] += -1.0*foot_weight*rotated_positions[leg_lifted][0]

    b = np.matrix(b, dtype=float).transpose()

    #Find A*A^T
    AAt = A.dot(A.transpose())

    #Invert it
    AAt_inv = np.linalg.inv(np.matrix(AAt))

    #Find At*AAt_inv
    AtAAt_inv = A.transpose()*AAt_inv

    #Find the solution x = A((A*A^T)^-1)*b
    solution = AtAAt_inv*b

    #Check that solution meets constraints
    sum_forces = len(legs_lifted)*foot_weight + np.sum(solution) - boyant_force
    sum_torque_x = 0
    sum_torque_y = 0
    for leg_lifted in legs_lifted:
        sum_torque_x += foot_weight*rotated_positions[leg_lifted][1]
        sum_torque_y += foot_weight*rotated_positions[leg_lifted][0]

    index_offset = 0
    for ii in range(0,6):
        if ii in legs_lifted:
            index_offset += 1
            continue
        sum_torque_x += solution[ii-index_offset]*rotated_positions[ii][1]
        sum_torque_y += solution[ii-index_offset]*rotated_positions[ii][0]

    constraint_error = abs(np.sum([sum_forces, sum_torque_x, sum_torque_y]))

    if constraint_error > 1e-10:
        print("Warning! Sum of constraint violations = "+str(constraint_error))

    #Return the solution
    return solution
```

Find Maximum Foot Mass Legs Lifted

Find the maximum mass of a single foot when particular legs are lifted

In [19]:
```
def find_maximum_foot_mass_legs_lifted(positions, boyant_force, gravity, legs_lifted, pitch = 0.0):

    # Get max and min foot mass values for this size balloon (based only on vertical force balance)
    # Use this to bound the maximum foot weight and make initial guesses
    foot_bounds = get_foot_mass_bounds(boyant_force, gravity, len(legs_lifted))

    # Setup constraint functions
    # This constraint says the force on a cable must be greater than 0
    def constraint_function_1(x, index):
        output = lift_legs(positions, boyant_force, x, legs_lifted, pitch)
        return output.tolist()[index]

    # This constraint says the force on a cable must be less than the weight of a foot
    def constraint_function_2(x, index):
        output = lift_legs(positions, boyant_force, x, legs_lifted, pitch)
        return x-output.tolist()[index]

    # Setup constraints
    # Constraint states that the resulting forces must be positive
    # and that the forces must be less than the weight of the foot
    cons = []
    for ii in range(0,6-len(legs_lifted)):
        cons.append({'type': 'ineq', 'fun': lambda x, ii=ii: constraint_function_1(x, ii)})
        cons.append({'type': 'ineq', 'fun': lambda x, ii=ii: constraint_function_2(x, ii)})


    # Setup function to minimize
    # In this case we are maximizing foot weight
    fun = lambda x: x*-1.0

    # Run optimization to find greatest foot mass that still balances the balloon
    bounds = [foot_bounds]
    result = minimize(fun, [np.mean(foot_bounds)], method='SLSQP', bounds=bounds, constraints=cons)
    return result['x']/gravity
```

Find Min and Max Volume when legs are lifted with a particular foot mass

Find the maximum mass of a single foot when particular legs are lifted

In [ ]:
```
def find_volume_range_legs_lifted(area_density, additional_mass_percentage, foot_mass, gravity, legs_lifted, pitch = 0.0):

    # Get max and min Volume for the analysis
    volume_bounds = [0, 10000]

    # Setup constraint functions
    # This constraint says the force on a cable must be greater than 0
    def constraint_function_1(x, index):
        axes = get_axis_lengths(x[0])
        Fb = get_buoyant_force(axes[0], axes[1], axes[2], area_density, gravity, additional_mass_percentage)
        output = lift_legs(get_ave_connection_points(axes[0], axes[1], axes[2]), Fb, [foot_mass*gravity], legs_lifted, pitch)
        return output.tolist()[index][0]

    # This constraint says the force on a cable must be less than the weight of a foot
    def constraint_function_2(x, index):
        axes = get_axis_lengths(x[0])
        Fb = get_buoyant_force(axes[0], axes[1], axes[2], area_density, gravity, additional_mass_percentage)
        output = lift_legs(get_ave_connection_points(axes[0], axes[1], axes[2]), Fb, [foot_mass*gravity], legs_lifted, pitch)
        return foot_mass*gravity-output.tolist()[index][0]

    # Setup constraints
    # Constraint states that the resulting forces must be positive
    # and that the forces must be less than the weight of the foot
    cons = []
    for ii in range(0,6-len(legs_lifted)):
        cons.append({'type': 'ineq', 'fun': lambda x, ii=ii: constraint_function_1(x, ii)})
        cons.append({'type': 'ineq', 'fun': lambda x, ii=ii: constraint_function_2(x, ii)})


    # Setup function to minimize
    # In this case we are minimizing volume
    fun_min = lambda x: x
    fun_max = lambda x: -x

    # Run optimization to find greatest foot mass that still balances the balloon
    bounds = [volume_bounds]
    result_min = minimize(fun_min, [np.mean(volume_bounds)], method='SLSQP', bounds=bounds, constraints=cons)
    result_max = minimize(fun_max, [np.mean(volume_bounds)], method='SLSQP', bounds=bounds, constraints=cons)
    return [result_min['x'][0], result_max['x'][0]]
```

Find Min and Max Volume when legs are lifted with a particular foot mass

Same as above but with the second axis ratio

In [ ]:
```
def find_volume_range_legs_lifted2(area_density, additional_mass_percentage, foot_mass, gravity, legs_
lifted, pitch = 0.0):

    # Get max and min Volume for the analysis
    volume_bounds = [0, 10000]

    # Setup constraint functions
    # This constraint says the force on a cable must be greater than 0
    def constraint_function_1(x, index):
        axes = get_axis_lengths2(x[0])
        Fb = get_buoyant_force(axes[0], axes[1], axes[2], area_density, gravity, additional_mass_perce
ntage)
        output = lift_legs(get_ave_connection_points(axes[0], axes[1], axes[2]), Fb, [foot_mass*gravit
y], legs_lifted, pitch)
        return output.tolist()[index][0]

    # This constraint says the force on a cable must be less than the weight of a foot
    def constraint_function_2(x, index):
        axes = get_axis_lengths2(x[0])
        Fb = get_buoyant_force(axes[0], axes[1], axes[2], area_density, gravity, additional_mass_perce
ntage)
        output = lift_legs(get_ave_connection_points(axes[0], axes[1], axes[2]), Fb, [foot_mass*gravit
y], legs_lifted, pitch)
        return foot_mass*gravity-output.tolist()[index][0]

    # Setup constraints
    # Constraint states that the resulting forces must be positive
    # and that the forces must be less than the weight of the foot
    cons = []
    for ii in range(0,6-len(legs_lifted)):
        cons.append({'type': 'ineq', 'fun': lambda x, ii=ii: constraint_function_1(x, ii)})
        cons.append({'type': 'ineq', 'fun': lambda x, ii=ii: constraint_function_2(x, ii)})


    # Setup function to minimize
    # In this case we are minimizing volume
    fun_min = lambda x: x
    fun_max = lambda x: -x

    # Run optimization to find greatest foot mass that still balances the balloon
    bounds = [volume_bounds]
    result_min = minimize(fun_min, [np.mean(volume_bounds)], method='SLSQP', bounds=bounds, constraint
s=cons)
    result_max = minimize(fun_max, [np.mean(volume_bounds)], method='SLSQP', bounds=bounds, constraint
s=cons)
    return [result_min['x'][0], result_max['x'][0]]
```

Helper functions to deal with balloon geometry

In [ ]:
```
def get_connection_points(a,b,c):
    # Find balloon connection points based on geometry
    p1 = find_position_on_ellipsoid_z_0(15, a, b, c)
    p0 = p1[:]
    p0[1] *= -1.0
    p9 = p0[:]
    p9[0] *= -1.0
    p10 = p1[:]
    p10[0] *= -1.0

    p4 = find_position_on_ellipsoid_z_0(45, a, b, c)
    p3 = p4[:]
    p3[1] *= -1.0
    p6 = p3[:]
    p6[0] *= -1.0
    p7 = p4[:]
    p7[0] *= -1.0

    # p2 & p8
    p2 = find_position_on_ellipsoid_y_0(45, a, b, c)
    p2[2] *= -1.0
    p8 = p2[:]
    p8[0] *= -1.0

    # p5 is on the bottom center
    p5 = [0.0, 0.0, -c]

    # Assemble the points into groups by leg
    return [p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, p10]

#get the connection points for the current volume
def get_ave_connection_points(a,b,c):
    points = get_connection_points(a,b,c)
    L1 = (np.array(points[0])+np.array(points[2])+np.array(points[3]))/3
    L2 = (np.array(points[1])+np.array(points[4])+np.array(points[2]))/3
    L3 = (np.array(points[3])+np.array(points[5])+np.array(points[6]))/3
    L4 = (np.array(points[4])+np.array(points[7])+np.array(points[5]))/3
    L5 = (np.array(points[6])+np.array(points[8])+np.array(points[9]))/3
    L6 = (np.array(points[7])+np.array(points[10])+np.array(points[8]))/3
    return [L1, L2, L3, L4, L5, L6]

def get_legs(cp):
    return [[cp[0], cp[2], cp[3]], [cp[1], cp[4], cp[2]], [cp[3], cp[5], cp[6]], [cp[4], cp[7], cp[5
]], [cp[6], cp[8], cp[9]], [cp[7], cp[10], cp[8]]]

def get_feet(legs):
    return [get_foot_x_y(x) for x in legs]
```

In [ ]:
```
# approximate surface area of ellipsoid
def surface_area_ellipsoid(a,b,c):
    return 4*np.pi*((((a*b)**1.6)+((a*c)**1.6)+((b*c)**1.6))/3.0)**(1/1.6)
```

In [ ]:
```
# balloon mass with extra mass added
def balloon_mass(a,b,c, rho, added_mass):
    return surface_area_ellipsoid(a,b,c)*rho*(1.0 + added_mass)
```

In [2]:
```
# Get the buoyant force of the proof of concept with
def get_buoyant_force(a, b, c, rho, gravity, added_mass):
    weight = balloon_mass(a,b,c,rho,added_mass)*gravity
    V = (4/3)*np.pi *a*b*c
    Lift = (rho_earth-rho_helium_earth)*V*gravity
    return Lift - weight
```

In [3]:
```
#a=2b=4c
def get_axis_lengths(V):
    temp = (V*(3/4)/np.pi)
    a = (temp*8.0) ** (1.0/3.0)
    b = temp ** (1.0/3.0)
    c = (temp/8.0) ** (1.0/3.0)
    return [a,b,c]

#a*0.3=b, a*0.2=c
def get_axis_lengths2(V):
    temp = (V*(3/4)/np.pi)/(0.2*0.3)
    a = (temp) ** (1.0/3.0)
    b = a*0.3
    c = a*0.2
    return [a,b,c]
```

Utility

Utility functions to accomplish simple mathematic operations

Normalized

Returns a normalized vector

In [20]:
```
def normalized(vector):
    x = np.linalg.norm(vector)
    if x==0:
        return vector

    return vector/x
```

Plotting

Simple functions for various plots

Plot Balloon Geometry

Plot the ballon, its cables, and feet

In [21]:
```
def plot_balloon_geometry(points, feet, legs, a, b, c):

    # Create Figure
    fig = plt.figure()
    ax = Axes3D(fig)
    ax.set_aspect('equal')

    # Plot connection points to balloon
    for point in points:
        ax.scatter(point[0], point[1], point[2], c='r', marker='o')

    # Plot feet
    for foot in feet:
        ax.scatter(foot[0], foot[1], -1.0*balloon_height, c='b', marker='^')

    # Plot cables
    for idx, leg in enumerate(legs):
        ax.plot([feet[idx][0], leg[0][0]], [feet[idx][1], leg[0][1]], [-1.0*balloon_height, leg[0][2
]], c='g')
        ax.plot([feet[idx][0], leg[1][0]], [feet[idx][1], leg[1][1]], [-1.0*balloon_height, leg[1][2
]], c='g')
        ax.plot([feet[idx][0], leg[2][0]], [feet[idx][1], leg[2][1]], [-1.0*balloon_height, leg[2][2
]], c='g')

    # Create ellipsoid
    phi = np.linspace(0,2*np.pi, 100).reshape(100, 1) # the angle of the projection in the xy-plane
    theta = np.linspace(0, np.pi, 100).reshape(-1, 100) # the angle from the polar axis, ie the polar
 angle

    # Transformation formulae for a spherical coordinate system.
    X = a*np.sin(theta)*np.cos(phi)
    Y = b*np.sin(theta)*np.sin(phi)
    Z = c*np.cos(theta)
    ax.plot_surface(X, Y, Z, color='c', alpha=0.5)

    # Create cubic bounding box to simulate equal aspect ratio
    # Matplotlib can't do axis equal properly in 3d
    max_range = points[0][0]
    Xb = 0.5*max_range*np.mgrid[-1:2:2,-1:2:2,-1:2:2][0].flatten()
    Yb = 0.5*max_range*np.mgrid[-1:2:2,-1:2:2,-1:2:2][1].flatten()
    Zb = 0.5*max_range*np.mgrid[-1:2:2,-1:2:2,-1:2:2][2].flatten()
    for xb, yb, zb in zip(Xb, Yb, Zb):
       ax.plot([xb], [yb], [zb], 'w')
```

Plot Forces Bar Graph

Function to plot the forces on each cable as a 3d bar graph

In [1]:
```
def plot_forces_bar_graph(positions, forces, legs_lifted, max_mass, a, b, c, g, suptitle, title, pitch
= 0.0):

    # Now plot the results
    fig = plt.figure()
    ax = Axes3D(fig)
    ax.set_aspect('equal')

    #normalize length of force vectors
    norm_force = normalized(forces)

    #colors
    colors = ['b', 'g','r','m','c','orange']

    lifted_force = max_mass[0]*g
    legend = []
    #legend.append('Balloon Footprint')
    index_offset = 0
    for idx in range(0,6):
        legend_string = 'Leg '+str(idx+1)+': '
        if idx in legs_lifted:
            legend_string += 'Lifted: '+str(round(lifted_force, 2))+" N"
            index_offset += 1
        else:
            legend_string+=str(round(forces.tolist()[idx-index_offset][0], 2))+" N"
        legend.append(legend_string)



    index_offset = 0
    for idx, position in enumerate(positions):
        if idx in legs_lifted:
            index_offset += 1
            ax.bar([position[0]], lifted_force, zs=[position[1]], zdir='y', width=0.1, alpha=1.0, zord
er=200, color=colors[idx], edgecolor='k')
            continue
        ax.bar([position[0]], [forces[idx-index_offset].tolist()[0][0]], zs=[position[1]], zdir='y', w
idth=0.1, alpha=1.0, zorder=200, color=colors[idx], edgecolor='k')

    plt.legend(legend, loc=3, fontsize=8)
    # Create ellipsoid
    phi = np.linspace(0,2*np.pi, 100).reshape(100, 1) # the angle of the projection in the xy-plane
    theta = np.linspace(0, np.pi, 100).reshape(-1, 100) # the angle from the polar axis, ie the polar
 angle
    theta2 = np.linspace(0, 2*np.pi, 100).reshape(100,1) # the angle from the polar axis, ie the polar
angle

    # Transformation formulae for a spherical coordinate system.


    x, y = np.mgrid[-3:3:150j,-3:3:150j]
    z =  3*(1 - x)**2 * np.exp(-x**2 - (y + 1)**2) \
   - 10*(x/5 - x**3 - y**5)*np.exp(-x**2 - y**2) \
   - 1./3*np.exp(-(x + 1)**2 - y**2)
    Xe = a*np.sin(theta)*np.cos(phi)
    Ye = b*np.sin(theta)*np.sin(phi)
    Ze = 0.1*np.cos(theta)
    ax.plot_surface(Xe, Ye, Ze, color='c', alpha=0.3, antialiased=False)


    #phi = np.linspace(0,2*np.pi, 256).reshape(256, 1)
    X1 = a*np.cos(phi)
    Y1 = b*np.sin(phi)
    Z1 = [0.0]*100
    ax.plot(X1, Y1, Z1, zorder=-1, color='k', linestyle='dashed')

    plt.suptitle(suptitle, fontsize=12, y=.90)
    plt.title(title, fontsize=8, y=1.05)
    ax.set_xlabel('X', fontsize=7)
    ax.set_ylabel('Y', fontsize=7)
    ax.set_zlabel('Force [N]', fontsize=7)


    # Create cubic bounding box to simulate equal aspect ratio
    # Matplotlib can't do axis equal properly in 3d
    max_range = points[0][0]
```

```
    Xb = 0.5*(b+0.1)*np.mgrid[-1:2:2,-1:2:2,-1:2:2][0].flatten()
    Yb = 0.5*(b+0.1)*np.mgrid[-1:2:2,-1:2:2,-1:2:2][1].flatten()
    Zb = 0.5*(b+0.1)*np.mgrid[-1:2:2,-1:2:2,-1:2:2][2].flatten()+(b+0.1)
    for xb, yb, zb in zip(Xb, Yb, Zb):
       ax.plot([xb], [yb], [zb], 'w')

    ax.tick_params(axis = 'both', which = 'major', labelsize = 7)
    ax.view_init(170,255)
```

Plot Max Mass Versus Pitch

Function to plot the maximum stable foot mass vs pitch data

In [2]:
```
def plot_max_mass_versus_pitch(pitches, set1, set2, legend1, legend2, suptitle, title, ymin, ymax):
    fig = plt.figure()
    axes = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # left, bottom, width, height (range 0 to 1)
    minimum_set = [np.min([v1,v2]) for v1,v2 in zip(set1,set2)]
    plot_pitches = [pitch*180.0/np.pi for pitch in pitches]
    axes.plot(plot_pitches, set1, plot_pitches, set2)
    axes.set_xlabel('Pitch [degress]')
    axes.set_ylabel('Maximum Stable Foot Mass [kg]')
    plt.suptitle(suptitle)
    plt.title(title)
    plt.legend([legend1, legend2], loc=3, fontsize=8)
    plt.ylim([ymin, ymax])
```

Plot OpenFoam Drag

Function to plot the drag force

In [ ]:
```
def plot_openfoam_drag(fileName, title, suptitle, yMin, yMax):
    time = []
    xForce = []

    #Open the file
    with open(fileName, 'r') as infile:

        #Skip the header
        for _ in range(3):
            next(infile)

        #Loop through every line
        for line in infile:

            #Remove non-numerical characters for easier parsing
            strippedLine = line.replace('(', ' ').replace(')', ' ')
            splitList = strippedLine.split()
            time.append(float(splitList[0]))

            #Multiply by 4 because only 1/4 of the ellipsoid is in the simulation
            xForce.append((float(splitList[1])+float(splitList[4]))*4)

    fig = plt.figure()
    axes = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # left, bottom, width, height (range 0 to 1)
    axes.plot(time, xForce)
    axes.set_xlabel('time [s]')
    axes.set_ylabel('Drag Force [N]')
    plt.suptitle(suptitle)
    plt.title(title)
    plt.ylim(yMin, yMax)
```

Plot OpenFoam Lift

Function to plot the lift and drag forces

In [ ]:
```
def plot_openfoam_lift(fileName, title, suptitle, yMin, yMax):
    time = []
    xForce = []
    zForce = []

    #Open the file
    with open(fileName, 'r') as infile:

        #skip header
        for _ in range(3):
            next(infile)

        #Loop through every line
        for line in infile:

            #replace non-number characters to spaces for easier parsing
            strippedLine = line.replace('(', ' ').replace(')', ' ')
            splitList = strippedLine.split()
            time.append(float(splitList[0]))

            #Multiply by two because only half of the ellipsoid was in the simulation
            zForce.append((float(splitList[3])+float(splitList[6]))*2)
            xForce.append((float(splitList[1])+float(splitList[4]))*2)

    #Plot
    fig = plt.figure()
    axes = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # left, bottom, width, height (range 0 to 1)
    axes.plot(time, xForce, time, zForce)
    plt.legend(['Drag', 'Lift'], loc=1, fontsize=8)
    axes.set_xlabel('time [s]')
    axes.set_ylabel('Aerodynamic Force [N]')
    plt.suptitle(suptitle)
    plt.title(title)
    plt.ylim(yMin, yMax)
```

Atmosphere Analysis

Viking Lander Data Parsing

Functions to parse and filter viking lander data

In [ ]:
```
#Apply a simple filter that averages surrounding temperatures to smooth data
def filter_data(data):
    filter_size = 100
    molarMassMarsAtm = 0.04334
    molarMassHelium = 0.004002602
    size = len(data['temp'])
    data['temp_filt'] = []
    for ii in range(size):
        if(ii < filter_size/2.0):
            numSamples = int(ii+filter_size/2)
            temp_sum = sum(data['temp'][0:numSamples])
            data['temp_filt'].append(float(temp_sum/numSamples))
        elif(ii > size-(filter_size/2.0)):
            numSamples = int((filter_size/2)+(size-ii))
            temp_sum = sum(data['temp'][ii-int(filter_size/2):-1])
            data['temp_filt'].append(float(temp_sum/numSamples))
        else:
            temp_sum = sum(data['temp'][int(ii-filter_size/2) : int(ii+filter_size/2)])
            data['temp_filt'].append(float(temp_sum/filter_size))

        data['rho_atm'].append(GasDensity(molarMassMarsAtm, data['temp_filt'][-1], data['pressure'][ii
])))
        data['rho_helium'].append(GasDensity(molarMassHelium, data['temp_filt'][-1], data['pressure'][
ii]))

# Remove empty points in the data
def remove_zeros(data):
    data['rho_atm_pruned'] = []
    data['rho_he_pruned'] = []
    data['sol_pruned'] = []
    data['temp_pruned'] = []
    data['sol_time_pruned'] = []
    for ii in range(len(data['temp_filt'])):
        if(not(data['rho_atm'][ii] == 0)):
            data['sol_pruned'].append(data['sol'][ii])
            data['rho_atm_pruned'].append(data['rho_atm'][ii])
            data['rho_he_pruned'].append(data['rho_helium'][ii])
            data['temp_pruned'].append(data['temp_filt'][ii])
            data['sol_time_pruned'].append(data['sol_time'][ii])

#Calculate gas density based on molar mass, temperature, and pressure
def GasDensity(MolarMass, Temperature, Pressure):
    R = 8.314
    if(Temperature > 0):
        return (Pressure*MolarMass)/(R*Temperature)
    return 0

#Load and parse viking lander data file
def load_viking_lander_file(filename):
    data = dict()
    data['year'] = []
    data['solar_long'] = []
    data['sol'] = []
    data['wind_speed'] = []
    data['wind_dir'] = []
    data['pressure'] = []
    data['temp'] = []
    data['rho_atm'] = []
    data['rho_helium'] = []
    data['ave_temp_offset'] = []
    data['sol_time'] = []
    with open(filename, 'r') as infile:
        line = ''
        for line in infile:
            line = line.split()
            data['year'].append(float(line[0]))
            data['solar_long'].append(float(line[1]))
            data['sol'].append(float(line[2]))
            data['wind_speed'].append(float(line[3]))
            data['wind_dir'].append(float(line[4]))
            data['pressure'].append(float(line[5])*100)
            data['temp'].append(float(line[7])+ 273.15)
            data['sol_time'].append(data['sol'][-1] % 1)

    filter_data(data)
    remove_zeros(data)
```

```
    data['ave_temp_offset'] = data['temp_pruned'][:]
    data['ave_density_offset'] = data['rho_atm_pruned'][:]
    data['ave_he_offset'] = data['rho_he_pruned'][:]
    oldNum = -1
    solCount = 1
    aveTemp = 0
    aveDensity = 0
    aveHe = 0
    for ii,sol in enumerate(data['sol_pruned']):
        newNum = int(data['sol_pruned'][ii]//1)
        if(newNum > oldNum):
            oldNum = newNum
            aveTemp/=solCount
            aveDensity/=solCount
            aveHe/=solCount
            for jj in range(ii-solCount, ii):
                data['ave_temp_offset'][jj]-=aveTemp
                data['ave_density_offset'][jj]-=aveDensity
                data['ave_he_offset'][jj]-=aveHe
            solCount = 0
            aveTemp=0
            aveDensity = 0
            aveHe = 0
        solCount+=1
        aveTemp+=data['temp_pruned'][ii]
        aveDensity+=data['rho_atm_pruned'][ii]
        aveHe+=data['rho_he_pruned'][ii]
    return data

#Plot atmospheric and helium densities based on viking lander data
def plot_viking_lander_data(data, title):
    fig = plt.figure()
    axes = fig.add_axes([0.15, 0.1, 0.8, 0.8]) # left, bottom, width, height (range 0 to 1)
    axes.plot(data['sol_pruned'], data['rho_atm_pruned'],'r')
    axes.set_xlabel('Sol')
    axes.set_ylabel('Atmospheric Density [kg/m^3]')
    plt.suptitle('Atmospheric Density on Mars')
    plt.title(title)
    fig = plt.figure()
    axes = fig.add_axes([0.15, 0.1, 0.8, 0.8]) # left, bottom, width, height (range 0 to 1)
    axes.plot(data['sol_pruned'], data['rho_he_pruned'],'r')
    axes.set_xlabel('Sol')
    axes.set_ylabel('Helium Density [kg/m^3]')
    plt.suptitle('Helium Density on Mars')
    plt.title(title)

#Plot buoyant force based on viking lander data
def plot_buoyant_force(data, volume, added_mass, gravity, title):
    data['Fb'] = [(rho_mars-rho_he)*volume*gravity - added_mass*gravity for rho_mars, rho_he in zip(da
ta['rho_atm_pruned'], data['rho_he_pruned'])]
    fig = plt.figure()
    axes = fig.add_axes([0.15, 0.1, 0.8, 0.8]) # left, bottom, width, height (range 0 to 1)
    axes.plot(data['sol_pruned'], data['Fb'],'r')
    axes.set_xlabel('Sol')
    axes.set_ylabel('Buoyant Force [N]')
    plt.suptitle('          Buoyant Force on Mars')
    plt.title(title)

#Plot stable foot masses over a martian year based on viking lander data
def plot_foot_masses(data,connection_points,g,volume, title):
    #Number of points to plot
    samples = 1000
    molarMassMarsAtm = 0.04334
    molarMassHelium = 0.004002602

    #indices
    indices = list(map(int, np.linspace(0, len(data['Fb'])-1, samples)))
    leg_1_max_mass_lifted_temp = []
    leg_3_max_mass_lifted_temp = []
    legs_1_6_max_mass_lifted_temp = []
    legs_3_4_max_mass_lifted_temp = []
    min_mass = []
    for index in indices:
        #index = int(indexx)
        leg_1_max_mass_lifted_temp.append(find_maximum_foot_mass_legs_lifted(connection_points, data[
'Fb'][index], g, [0], 0))
```

```
        leg_3_max_mass_lifted_temp.append(find_maximum_foot_mass_legs_lifted(connection_points, data[
'Fb'][index], g, [2], 0))
        legs_1_6_max_mass_lifted_temp.append(find_maximum_foot_mass_legs_lifted(connection_points, dat
a['Fb'][index], g, [0,5], 0))
        legs_3_4_max_mass_lifted_temp.append(find_maximum_foot_mass_legs_lifted(connection_points, dat
a['Fb'][index], g, [2,3], 0))
        min_mass.append(data['Fb'][index]/(6.0*g))

    min_one_leg = min(leg_1_max_mass_lifted_temp+leg_3_max_mass_lifted_temp)
    min_two_leg = min(legs_1_6_max_mass_lifted_temp+legs_3_4_max_mass_lifted_temp)
    max_min_mass = max(min_mass)

    aveDensity_atm = sum(data['rho_atm_pruned'])/len(data['rho_atm_pruned'])
    aveDensity_he = sum(data['rho_he_pruned'])/len(data['rho_he_pruned'])
    maxDensityOffset_he = max(data['ave_he_offset'])
    minDensityOffset_he = min(data['ave_he_offset'])
    maxDensityOffset_atm = max(data['ave_density_offset'])
    minDensityOffset_atm = min(data['ave_density_offset'])
    maxFb = ((aveDensity_atm+maxDensityOffset_atm)-(aveDensity_he+maxDensityOffset_he))*volume*g
    minFb = ((aveDensity_atm+minDensityOffset_atm)-(aveDensity_he+minDensityOffset_he))*volume*g
    FbOffset = (maxFb-minFb)/2.0
    fig = plt.figure()
    axes = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # left, bottom, width, height (range 0 to 1)
    minimum_set = [np.min([v1,v2]) for v1,v2 in zip(leg_1_max_mass_lifted_temp,leg_3_max_mass_lifted_t
emp)]
    plot_sols = [data['sol_pruned'][ii] for ii in indices]
    axes.plot(plot_sols, leg_1_max_mass_lifted_temp, plot_sols, leg_3_max_mass_lifted_temp, plot_sols,
min_mass)
    axes.set_xlabel('sol')
    axes.set_ylabel('Stable Foot Mass [kg]')
    plt.suptitle('Stable Foot Mass over a Martian Year')
    plt.title(title)
    plt.legend(['Max leg 1 lifted', 'Max leg 3 lifted', 'Minimum'], loc=3, fontsize=8)

    fig = plt.figure()
    axes = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # left, bottom, width, height (range 0 to 1)
    minimum_set = [np.min([v1,v2]) for v1,v2 in zip(leg_1_max_mass_lifted_temp,leg_3_max_mass_lifted_t
emp)]
    plot_sols = [data['sol_pruned'][ii] for ii in indices]
    axes.plot(plot_sols, legs_1_6_max_mass_lifted_temp, plot_sols, legs_3_4_max_mass_lifted_temp, plot
_sols, min_mass)
    axes.set_xlabel('sol')
    axes.set_ylabel('Stable Foot Mass [kg]')
    plt.suptitle('Stable Foot Mass over a Martian Year')
    plt.title(title)
    plt.legend(['Max legs 1 & 6 lifted', 'Max legs 3 & 4 lifted', 'Minimum'], loc=3, fontsize=8)
    return([max_min_mass, min_one_leg, max_min_mass, min_two_leg, FbOffset])
```

Finding Dynamic Viscosity

In [24]:
```
def dynamic_viscosity(temp):
    #Sutherland Coefficients
    C1 = 1.458e-6
    C2 = 110.4
    return (C1*(temp**(3/2)))/(temp+C2)
```

Initialization of Analysis

This includes importing useful libraries, as well as defining constants like gravity

In [1]:
```
#Imports and symbol initialization
#%matplotlib notebook
%matplotlib qt
import sympy
from IPython.display import display
from mpl_toolkits.mplot3d import Axes3D
import mpmath as mp
import matplotlib
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from sklearn.preprocessing import normalize
sympy.init_printing(use_latex='mathjax')
from matplotlib.colors import LightSource
from matplotlib.patches import Ellipse


rho_titan = 5.280
rho_earth = 1.217
rho_mars = 0.020
rho_helium_titan = 0.728
rho_helium_earth = 0.178
rho_helium_mars = 0.002
added_mass_titan = 45.0
added_mass_earth = 0.5
added_mass_mars = 0.1
g_titan = 1.352
g_earth = 9.81
g_mars = 3.71
vol_titan = 11.534
vol_earth = 1.925
vol_mars = 88.134
```

Appendix B: BALLET Aerodynamics Analysis

```
/*--------------------------------*- C++ -
*----------------------------------*\
| =========                 |
|
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
|
|  \\    /   O peration     | Version:  4.1
|
|   \\  /    A nd           | Web:      www.OpenFOAM.org
|
|    \\/     M anipulation  |
|
\*---------------------------------------------------------------------
-------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * //

application     pisoFoam;

startFrom       startTime;

startTime       7.2;

stopAt          endTime;

endTime         50000;

deltaT          0.0002;

writeControl    timeStep;

writeInterval   1000;

purgeWrite      1;

writeFormat     ascii;

writePrecision  6;

writeCompression off;

timeFormat      general;
```

```
timePrecision   6;

runTimeModifiable true;

functions
{
    #include "forces"
}

//
*********************************************************************
*** //
```

```
/*--------------------------------*- C++ -
*----------------------------------*\
| =========                 |
|
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
|
|  \\    /   O peration     | Version:  4.1
|
|   \\  /    A nd           | Web:      www.OpenFOAM.org
|
|    \\/     M anipulation  |
|
\*---------------------------------------------------------------------
-------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSchemes;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * //

ddtSchemes
{
    default         Euler;
}

gradSchemes
{
    default         Gauss linear;
}


divSchemes
{
    default         none;
    div(phi,U)      Gauss LUST grad(U);
    div((nuEff*dev2(T(grad(U))))) Gauss linear;
}

laplacianSchemes
{
    default         Gauss linear corrected;
}

interpolationSchemes
{
```

```
    default         linear;
}

snGradSchemes
{
    default         corrected;
}


//
*********************************************************************
*** //
```

```
/*--------------------------------*- C++ -
*----------------------------------*\
| =========                 |
|
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
|
|  \\    /   O peration     | Version:  4.1
|
|   \\  /    A nd           | Web:      www.OpenFOAM.org
|
|    \\/     M anipulation  |
|
\*---------------------------------------------------------------------
-------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSolution;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * //

solvers
{
    p
    {
        solver          GAMG;
        tolerance       1e-06;
        relTol          0.1;
        smoother        GaussSeidel;
    }

    pFinal
    {
        $p;
        tolerance       1e-06;
        relTol          0;
    }

    "(U|k|epsilon|omega|R|nuTilda)"
    {
        solver          smoothSolver;
        smoother        GaussSeidel;
        tolerance       1e-05;
        relTol          0;
    }
}
```

```
PISO
{
    nCorrectors     2;
    nNonOrthogonalCorrectors 0;
    pRefCell        0;
    pRefValue       0;
}



// ************************************************************************* //
```

```
/*--------------------------------*- C++ -
*----------------------------------*\
| =========                 |
|
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
|
|  \\    /   O peration     | Version:  4.1
|
|   \\  /    A nd           | Web:      www.OpenFOAM.org
|
|    \\/     M anipulation  |
|
\*---------------------------------------------------------------------
-------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      blockMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * //

convertToMeters 1;

vertices
(
    (-4 0 0)
    (-4 0  3)
    (-4 5 3)
    (-4 5 0)
    (8 0 0)
    (8 0 3)
    (8 5 3)
    (8 5 0)
);

blocks
(
    hex (0 4 7 3 1 5 6 2) (36 15 9) simpleGrading (1 1 1)
);

edges
(
);

boundary
(
    inletWall
```

```
    {
        type patch;
        faces
        (
            (0 1 2 3)
            (5 6 2 1)
            (7 3 2 6)
        );
    }
    sym
    {
        type symmetry;
        faces
        (
            (7 4 0 3)
            (4 5 1 0)
        );
    }
    outletWalls
    {
        type patch;
        faces
        (
            (7 6 5 4)
        );
    }
);

mergePatchPairs
(
);

// ************************************************************************* //
```

```
/*--------------------------------*- C++ -
*----------------------------------*\
| =========                 |
|
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
|
|  \\    /   O peration     | Version:  4.1
|
|   \\  /    A nd           | Web:      www.OpenFOAM.org
|
|    \\/     M anipulation  |
|
\*---------------------------------------------------------------------
-------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      snappyHexMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * //

// Which of the steps to run
castellatedMesh true;
snap            true;
addLayers       true;


// Geometry. Definition of all surfaces. All surfaces are of class
// searchableSurface.
// Surfaces are used
// - to specify refinement for any mesh cell intersecting it
// - to specify refinement for any mesh cell inside/outside/near
// - to 'snap' the mesh boundary to the surface
geometry
{
    titan_balloon.stl
    {
        type triSurfaceMesh;
        scale 0.01;
        name titan_balloon;
    }

    refinementBox
    {
        type searchableBox;
        min (-1.5 -0.5 -0.5);
        max (8 3.0 2.0);
```

```
    }
};



// Settings for the castellatedMesh generation.
castellatedMeshControls
{

    // Refinement parameters
    // ~~~~~~~~~~~~~~~~~~~~~

    // If local number of cells is >= maxLocalCells on any processor
    // switches from from refinement followed by balancing
    // (current method) to (weighted) balancing before refinement.
    maxLocalCells 100000;

    // Overall cell limit (approximately). Refinement will stop
immediately
    // upon reaching this number so a refinement level might not
complete.
    // Note that this is the number of cells before removing the part
which
    // is not 'visible' from the keepPoint. The final number of cells
might
    // actually be a lot less.
    maxGlobalCells 2000000000;

    // The surface refinement loop might spend lots of iterations
refining just a
    // few cells. This setting will cause refinement to stop if <=
minimumRefine
    // are selected for refinement. Note: it will at least do one
iteration
    // (unless the number of cells to refine is 0)
    minRefinementCells 10;

    // Allow a certain level of imbalance during refining
    // (since balancing is quite expensive)
    // Expressed as fraction of perfect balance (= overall number of
cells /
    // nProcs). 0=balance always.
    maxLoadUnbalance 0.10;


    // Number of buffer layers between different levels.
    // 1 means normal 2:1 refinement restriction, larger means slower
    // refinement.
    nCellsBetweenLevels 4;
```

```
    // Explicit feature edge refinement
    // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    // Specifies a level for any cell intersected by its edges.
    // This is a featureEdgeMesh, read from constant/triSurface for
now.
    features
    (
    );



    // Surface based refinement
    // ~~~~~~~~~~~~~~~~~~~~~~~~

    // Specifies two levels for every surface. The first is the
minimum level,
    // every cell intersecting a surface gets refined up to the
minimum level.
    // The second level is the maximum level. Cells that 'see'
multiple
    // intersections where the intersections make an
    // angle > resolveFeatureAngle get refined up to the maximum
level.

    refinementSurfaces
    {
        titan_balloon
        {
            // Surface-wise min and max refinement level
            level (4 4);
        }
    }

    // Resolve sharp angles
    resolveFeatureAngle 30;


    // Region-wise refinement
    // ~~~~~~~~~~~~~~~~~~~~~~

    // Specifies refinement level for cells in relation to a surface.
One of
    // three modes
    // - distance. 'levels' specifies per distance to the surface the
    //   wanted refinement level. The distances need to be specified
in
    //   descending order.
```

```
    // - inside. 'levels' is only one entry and only the level is
used. All
    //   cells inside the surface get refined up to the level. The
surface
    //   needs to be closed for this to be possible.
    // - outside. Same but cells outside.

    refinementRegions
    {
        refinementBox
        {
            mode inside;
            levels ((1.0 2));
        }
    }


    // Mesh selection
    // ~~~~~~~~~~~~~~

    // After refinement patches get added for all refinementSurfaces
and
    // all cells intersecting the surfaces get put into these patches.
The
    // section reachable from the locationInMesh is kept.
    // NOTE: This point should never be on a face, always inside a
cell, even
    // after refinement.
    locationInMesh (4.9 2.9 1.9);


    // Whether any faceZones (as specified in the refinementSurfaces)
    // are only on the boundary of corresponding cellZones or also
allow
    // free-standing zone faces. Not used if there are no faceZones.
    allowFreeStandingZoneFaces true;
}



// Settings for the snapping.
snapControls
{
    //- Number of patch smoothing iterations before finding
correspondence
    //  to surface
    nSmoothPatch 5;

    //- Relative distance for points to be attracted by surface
feature point
```

```
    //  or edge. True distance is this factor times local
    //  maximum edge length.
    tolerance 4.0;

    //- Number of mesh displacement relaxation iterations.
    nSolveIter 0;

    //- Maximum number of snapping relaxation iterations. Should stop
    //  before upon reaching a correct mesh.
    nRelaxIter 5;

    // Feature snapping

        //- Number of feature edge snapping iterations.
        //  Leave out altogether to disable.
        //nFeatureSnapIter 10;

        //- Detect (geometric only) features by sampling the surface
        //  (default=false).
        //implicitFeatureSnap false;

        //- Use castellatedMeshControls::features (default = true)
        //explicitFeatureSnap true;

        //- Detect points on multiple surfaces (only for
explicitFeatureSnap)
        //multiRegionFeatureSnap false;
}



// Settings for the layer addition.
addLayersControls
{
    // Are the thickness parameters below relative to the undistorted
    // size of the refined cell outside layer (true) or absolute sizes
(false).
    relativeSizes false;

    // Per final patch (so not geometry!) the layer information
    layers
    {
        "titan_balloon.*"
        {
            nSurfaceLayers 10;
        }
    }

    // Expansion factor for layer mesh
    expansionRatio 1.2;
```

```
    // Wanted thickness of final added cell layer. If multiple layers
    // is the thickness of the layer furthest away from the wall.
    // Relative to undistorted size of cell outside layer.
    // See relativeSizes parameter.
    finalLayerThickness 0.01;

    // Minimum thickness of cell layer. If for any reason layer
    // cannot be above minThickness do not add layer.
    // Relative to undistorted size of cell outside layer.
    minThickness 0.0001;

    // If points get not extruded do nGrow layers of connected faces
that are
    // also not grown. This helps convergence of the layer addition
process
    // close to features.
    // Note: changed(corrected) w.r.t 17x! (didn't do anything in 17x)
    nGrow 0;

    // Advanced settings

    // When not to extrude surface. 0 is flat surface, 90 is when two
faces
    // are perpendicular
    featureAngle 30;

    // At non-patched sides allow mesh to slip if extrusion direction
makes
    // angle larger than slipFeatureAngle.
    slipFeatureAngle 30;

    // Maximum number of snapping relaxation iterations. Should stop
    // before upon reaching a correct mesh.
    nRelaxIter 3;

    // Number of smoothing iterations of surface normals
    nSmoothSurfaceNormals 3;

    // Number of smoothing iterations of interior mesh movement
direction
    nSmoothNormals 3;

    // Smooth layer thickness over surface patches
    nSmoothThickness 10;

    // Stop layer growth on highly warped cells
    maxFaceThicknessRatio 0.5;

    // Reduce layer growth where ratio thickness to medial
```

```
    // distance is large
    maxThicknessToMedialRatio 0.3;

    // Angle used to pick up medial axis points
    // Note: changed(corrected) w.r.t 17x! 90 degrees corresponds to
130 in 17x.
    minMedianAxisAngle 90;


    // Create buffer region for new layer terminations
    nBufferCellsNoExtrude 0;


    // Overall max number of layer addition iterations. The mesher
will exit
    // if it reaches this number of iterations; possibly with an
illegal
    // mesh.
    nLayerIter 5000;
}



// Generic mesh quality settings. At any undoable phase these
determine
// where to undo.
meshQualityControls
{
    #include "meshQualityDict"


    // Advanced

    //- Number of error distribution iterations
    nSmoothScale 4;
    //- Amount to scale back displacement at error points
    errorReduction 0.75;
}


// Advanced

// Write flags
writeFlags
(
    scalarLevels
    layerSets
    layerFields     // write volScalarField for layer coverage
);
```

```
// Merge tolerance. Is fraction of overall bounding box of initial
mesh.
// Note: the write tolerance needs to be higher than this.
mergeTolerance 1e-6;


//
*********************************************************************
*** //
```

```
/*--------------------------------*- C++ -
*----------------------------------*\
| =========                 |
|
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
|
|  \\    /   O peration     | Version:  4.1
|
|   \\  /    A nd           | Web:      www.OpenFOAM.org
|
|    \\/     M anipulation  |
|
\*---------------------------------------------------------------------
-------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      transportProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * //

transportModel  Newtonian;

nu              [0 2 -1 0 0 0 0] 0.000001246212121;

//
*************************************************************************
*** //
```

```
/*--------------------------------*- C++ -
*----------------------------------*\
| =========                 |
|
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
|
|  \\    /   O peration     | Version:  4.1
|
|   \\  /    A nd           | Web:      www.OpenFOAM.org
|
|    \\/     M anipulation  |
|
\*---------------------------------------------------------------------
-------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      turbulenceProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * //

simulationType laminar;


//
************************************************************************
*** //
```

```
/*--------------------------------*- C++ -
*----------------------------------*\
| =========                 |
|
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
|
|  \\    /   O peration     | Version:  4.1
|
|   \\  /    A nd           | Web:      www.OpenFOAM.org
|
|    \\/     M anipulation  |
|
\*---------------------------------------------------------------------
-------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    object      U;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * //

dimensions      [0 1 -1 0 0 0 0];

internalField   uniform (1 0 0);

boundaryField
{
    inletWall
    {
        type            freestream;
        freestreamValue $internalField;
    }

    outletWalls
    {
        type            zeroGradient;
    }

    "titan_balloon*"
    {
        type            noSlip;
    }

    sym
    {
        type            symmetry;
    }
```

```
}

//
**********************************************************************
*** //
```

```
/*--------------------------------*- C++ -
*----------------------------------*\
| =========                 |
|
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
|
|  \\    /   O peration     | Version:  4.1
|
|   \\  /    A nd           | Web:      www.OpenFOAM.org
|
|    \\/     M anipulation  |
|
\*---------------------------------------------------------------------
-------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * //

dimensions      [0 2 -2 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    inletWall
    {
        type            zeroGradient;
    }

    outletWalls
    {
        type            fixedValue;
        value           $internalField;
    }

    "titan_balloon*"
    {
        type            zeroGradient;
    }

    sym
    {
        type            symmetry;
    }
```

```
}

//
**********************************************************************
*** //
```

Appendix C: Locomotion and Visualization Software

The following is the listing of the Ballet.py file for creating the Ballet model and the algorithms for its mobility

```
"""
Ballet.py contains the interface and utilities to support
driving the BALLET model
in Blender.

Hari Nayar
BALLET project
"""
import bpy
import os
import math
import numpy as np
import imageio
from mathutils import Vector
from mathutils import Euler
import logging


TERRAIN_PATH = '/Users/hdnayar/Documents/projects/BALLET/simulation/terrain/'
SOURCE_PATH = '/Users/hdnayar/Documents/projects/BALLET/simulation/software/BALLET/'

logging.basicConfig(filename=SOURCE_PATH+'BALLET_debug.log',level=logging.DEBUG)

#############################################################
## Ballet is the top level class that has all the elements of the
## mobility system. The mobility system is initialized with an
## initial position (of the balloon centroid) and heading direction
## (measured clockwise from the X-axis). A world coordinate system is
## defined with z down and the (0,0,0) position is in the center of
## of the terrain model.
## The Ballet system consists of a balloon and 6 limbs.
## The balloon has 8 attach points.
## Each limb consists of a foot and 3 cables that connect the foot
## to the balloon at designated attach points.
##
BALLET_LIMBS = 6                    # Ballet has 6 legs
BALLET_ATTACH_POINTS = 11           # There are 11 cable attachment points on the ballooon
BALLOON_DIMENSIONS = [2.0,0.6,0.4]  # Ellipsoid dimensions (meters) in x, y and z
FOOT_SIZE = [0.1, 0.1, 0.1]         # Each foot a 0.1 meter cube

BALLOON_ALTITUDE = 1.5              # Balloon height off the terrain
TERRAIN_SIZE_X = 100
TERRAIN_SIZE_Y = 100

CABLES_FOOT = 3                     # 3 cables per foot
CABLE_DIAMETER = 0.003              # Cable diameter in meters - 3 mm

FOOT_POSITION_INCREMENT = 0.002     # 2 mm
BALLOON_POSITION_INCREMENT = 0.002  # 2mm

DISTANCE_EPSILON = 0.001            # Set position precision to 1mm
ORIENTATION_EPSILON = 0.01          # Set orientation precition to 0.01 radians

OBSTACLE_HEIGHT = 0.05              # Height of an obstacle on the ground
                                    # that BALLET needs to step over

STEP_MOTION_DISCREZTIZATION = 0.01  # The distance between each point in the
                                    # foot step trajectory

PATH_MOTION_DISCREZTIZATION = 0.2   # The distance between each point in the
                                    # foot step trajectory

# These are the attach locations as indexed in the ATTACH_LOCATIONS array for the
# cables for each limb. The top view of the locations on the balloon is as shown
# here. The Z axis points down into the ground and Z=0 at the grouhnd plane with
# zero height. So the balloon z value is generally negative.
#                     ^
#                     Y
#               4       7
#         1                       10
#    <X         2      5      8
#         0                       9
#               3       6
#
LIMB_ATTACHMENTS = [[0, 2, 3], [1, 4, 2], [3, 5, 6], [4, 7, 5],
                                                [6, 8, 9], [7, 10, 8]]

# This array specifies the location around the appropriate ellipse for the location
# The 1st value is the coordinate axis: 0 is the X axis
# The 2nd value is the next coordinate axis: 1 is the Y axis, 2 is the Z axis
# The 3rd value is the index in increments of pi/12 around the ellipse
ATTACH_LOCATIONS = [[0,1,23], [0,1,1], [0,2,21], [0,1,20], [0,1,4], [0,2,18],
                            [0,1,16], [0,1,8], [0,2,15], [0,1,13], [0,1,11]]
```

```
class Ballet():
    class __impl:
        def __init__(self, x_pos, y_pos, z_pos, heading_angle, terrain):
            """Overall BALLET class to generate the objects of Ballet"""

            if terrain is not None:
                # Initialize the position of Ballet - the centroid of the balloon
                self.position = np.array([x_pos, y_pos, z_pos])
                self.heading  = heading_angle
                self.terrain = terrain
                self.pitch = self.terrain.slope(x_pos, y_pos, heading_angle, \
                                        BALLOON_DIMENSIONS[0]/2)
                self.balloon_attach_points = np.zeros([BALLET_ATTACH_POINTS, 3])
                self.limbs = [0]*BALLET_LIMBS
                self.cable_diameter = CABLE_DIAMETER
                self.locomoteDataDict = {}
                self.orientation = np.array([self.heading, self.pitch, 0.0])
                self.step_portion = 0.0

                # Set up the Balloon
                self.balloon = Balloon(self.position, self.orientation,
                                                    self.balloon_attach_points)

                # Set up the Limbs
                for index in range(0, BALLET_LIMBS):
                    limb_attach_point_0 = self.balloon_attach_points[
                                                    LIMB_ATTACHMENTS[index][0]]
                    limb_attach_point_1 = self.balloon_attach_points[
                                                    LIMB_ATTACHMENTS[index][1]]
                    limb_attach_point_2 = self.balloon_attach_points[
                                                    LIMB_ATTACHMENTS[index][2]]
                    limb_attach_points = np.vstack(
                        [limb_attach_point_0, limb_attach_point_1, limb_attach_point_2])

                    # The initial position of the foot is in the
                    # centroid of the projection of the 3 attach
                    # points in the horizontal plane
                    x_init = (limb_attach_point_0[0] + \
                            limb_attach_point_1[0] + limb_attach_point_2[0])/3.0
                    y_init = (limb_attach_point_0[1] + \
                            limb_attach_point_1[1] + limb_attach_point_2[1])/3.0

                    self.limbs[index] = Limb(index, x_init, y_init, heading_angle,
                        self.terrain, limb_attach_points)

                    # The structure for locomotion data
                    # States are:
                    # 0: inactiveLocomote,
                    # 1: newFootLocomote,
                    # 2: moveFootLocomote,
                    # 3: moveBalloonLocomote,
                    # 4: terminateLocomote
                    # 5: newTwoFootLocomote,
                    # 6: moveTwoFootLocomote,
                    # 7: moveBalloonLocomoteNew,

                    self.locomoteDataDict = {'State': 0,
                                             'Destination': [0.0, 0.0],
                                             'BalletPath': list(),
                                             'BalletPathIndex': 0,
                                             'FootIndex': -1,
                                             'FootStepTraj': list(),
                                             'FootStepTraj1': list(),
                                             'FootStepTraj2': list(),
                                             'FootStepTrajIndex': 0
                                             }
                    self.debug_variable1 = 0
                    self.debug_variable2 = 0

                print("Initiated ballet")
        # Singleton - return the existing version if it exists

        # A utility function called from computeFootPath to perform the foot
        # rotation to the new position
        def newFootPos(self, angle_val, sti, cti, foot_offset, y_rad):
            fox = foot_offset[0]
            foy = foot_offset[1]
            new_fx = cti * fox - sti * (foy - y_rad)
            new_fy = sti * fox + cti * (foy - y_rad) + y_rad
            new_fz = self.terrain.height(new_fx, new_fy)
            return [new_fx, new_fy, new_fz, angle_val]
```

```
# For a given destination position, compute the array of balloon
# path positions and orientation to traverse to get to the destination
# Assumes that the start position is defined as the position [0,0]
# This function returns the array path_position with the paths
# for the balloon, and each foot.
def computeBalloonFootPath(self, foot_offset, destination):
    y_rad = 0.0
    if destination[1] != 0:
        y_rad = (destination[0]*destination[0] + \
                destination[1]*destination[1])/(2.0*destination[1])
    else:
        y_rad = 1e10
    path_len = 0
    theta = 2.0 * math.atan2(destination[1], destination[0])

    if destination[1] != 0:
        path_len = y_rad * theta
    else:
        path_len = destination[0]

    path_position = []
    for i in range(7):
        path_position.append([])

    # if path curves to the right
    if destination[1] > 0:
        theta_increment = 0.0
        delta_theta = PATH_MOTION_DISCREZTIZATION * theta / path_len
        while theta_increment < theta:
            si = math.sin(theta_increment)
            ci = math.cos(theta_increment)
            # The balloon position in the trajectory
            x_pos = y_rad * si
            y_pos = y_rad * (1.0 - ci)
            z_pos = self.terrain.height(x_pos, y_pos) + BALLOON_ALTITUDE
            path_position[0].append([x_pos, y_pos, z_pos, \
                    theta_increment])
            # For each foot
            for i in range(6):
                path_position[i+1].append(self.newFootPos( \
                    theta_increment, si, ci, foot_offset[i], y_rad))
            theta_increment = theta_increment + delta_theta
        si = math.sin(theta)
        ci = math.cos(theta)
        # The last balloon position
        x_pos = y_rad * si
        y_pos = y_rad * (1.0 - ci)
        z_pos = self.terrain.height(x_pos, y_pos) + BALLOON_ALTITUDE
        path_position[0].append([x_pos, y_pos, z_pos, theta])
        # The last set of feet positions
        for i in range(6):
            path_position[i+1].append(self.newFootPos(theta, si, ci, \
                    foot_offset[i], y_rad))

    # if path curves to the left
    elif destination[1] < 0:
        theta_increment = 0.0
        delta_theta = PATH_MOTION_DISCREZTIZATION * theta / path_len
        while theta_increment > theta:
            si = math.sin(theta_increment)
            ci = math.cos(theta_increment)
            # The balloon position in the trajectory
            x_pos = y_rad * si
            y_pos = y_rad * (1.0 - ci)
            z_pos = self.terrain.height(x_pos, y_pos) + BALLOON_ALTITUDE
            path_position[0].append([x_pos, y_pos, z_pos, \
                                    theta_increment])
            # For each foot
            for i in range(6):
                path_position[i+1].append(self.newFootPos( \
                    theta_increment, si, ci, foot_offset[i], y_rad))
            theta_increment = theta_increment + delta_theta
        si = math.sin(theta)
        ci = math.cos(theta)
        # The last balloon position
        x_pos = y_rad * si
        y_pos = y_rad * (1.0 - ci)
        z_pos = self.terrain.height(x_pos, y_pos) + BALLOON_ALTITUDE
        path_position[0].append([x_pos, y_pos, z_pos, theta])
        # The last set of feet positions
        for i in range(6):
            path_position[i+1].append(self.newFootPos(theta, si, ci, \
                    foot_offset[i], y_rad))
```

```
        else:
            path_increment = 0.0
            while path_increment < path_len:
                z_pos = self.terrain.height(path_increment, 0.0) + \
                        BALLOON_ALTITUDE
                path_position[0].append([path_increment, 0.0, z_pos, 0.0])
                for i in range(6):
                    fox = path_increment + foot_offset[i][0]
                    foy = foot_offset[i][1]
                    foz = self.terrain.height(fox, foy)
                    path_position[i+1].append([fox, foy, foz, 0.0])
                path_increment = path_increment + PATH_MOTION_DISCRETIZATION
            z_pos = self.terrain.height(path_len, 0.0) + BALLOON_ALTITUDE
            path_position[0].append([path_len, 0.0, z_pos, 0.0])
            for i in range(6):
                fox = path_len + foot_offset[i][0]
                foy = foot_offset[i][1]
                foz = self.terrain.height(fox, foy)
                path_position[i+1].append([fox, foy, foz, 0.0])
        return path_position

    def initializeLocomote(self, destination):
        # Use current foot x,y offset from the balloon center frame
        foot_offset = []
        for i in range(6):
            foot_offset.append(self.limbs[i].getPosition())
        #   Generate balloon trajectory to the destination
        #   Generate feet trajectories
        self.locomoteDataDict['BalletPath'] = \
                self.computeBalloonFootPath(foot_offset, destination)
        #   Set balloon  and feet trajectory position to zero
        self.locomoteDataDict['BalletPathIndex'] = 0
        # Start with the first foot in the list
        self.locomoteDataDict['FootIndex'] = 0
        self.locomoteDataDict['FootStepTrajIndex'] = 0

        # If single foot stepping use this
        # self.locomoteDataDict['State'] = 1

        # If dual foot stepping use this
        self.locomoteDataDict['State'] = 5
        return 0

    def inactiveLocomote(self):
        self.locomoteDataDict['State'] = 0
        return 0

    def newFootLocomote(self):
        #   Generate current foot step trajectory
        #   Set foot setp trajectory position to zero
        startPosition = self.locomoteDataDict['BalletPath'] \
            [self.locomoteDataDict['FootIndex']+1] \
            [self.locomoteDataDict['BalletPathIndex']]
        endPosition = self.locomoteDataDict['BalletPath'] \
            [self.locomoteDataDict['FootIndex']+1] \
            [self.locomoteDataDict['BalletPathIndex'] + 1]
        self.locomoteDataDict['FootStepTraj'] = \
                        self.stepPath(startPosition, endPosition)

        # Set state for foot Lift
        self.locomoteDataDict['State'] = 2
        return 0

    # New to handle moving 2 feet at a time
    def newTwoFootLocomote(self):
        #   Generate current foot step trajectory
        #   Set foot setp trajectory position to zero
        startPosition1 = self.locomoteDataDict['BalletPath'] \
            [self.locomoteDataDict['FootIndex']+1] \
            [self.locomoteDataDict['BalletPathIndex']]
        endPosition1 = self.locomoteDataDict['BalletPath'] \
            [self.locomoteDataDict['FootIndex']+1] \
            [self.locomoteDataDict['BalletPathIndex'] + 1]

        startPosition2 = self.locomoteDataDict['BalletPath'] \
            [BALLET_LIMBS - self.locomoteDataDict['FootIndex']] \
            [self.locomoteDataDict['BalletPathIndex']]
        endPosition2 = self.locomoteDataDict['BalletPath'] \
            [BALLET_LIMBS - self.locomoteDataDict['FootIndex']] \
            [self.locomoteDataDict['BalletPathIndex'] + 1]

        stepDist1 = self.norm3(self.diff3(endPosition1, startPosition1))
        stepDist2 = self.norm3(self.diff3(endPosition2, startPosition2))
```

```
        # Need to scale the step discretization for the foot with the
        # shorter path to take smaller steps in the same time
        if stepDist1 > stepDist2:
            stepDiscretization1 = 1.0
            stepDiscretization2 = stepDist2/stepDist1
        else:
            stepDiscretization2 = 1.0
            stepDiscretization1 = stepDist1/stepDist2

        # Compute the trajectories for the 2 feet
        self.locomoteDataDict['FootStepTraj1'] = \
                        self.stepPathNew(startPosition1, endPosition1, \
                                         stepDiscretization1)
        self.locomoteDataDict['FootStepTraj2'] = \
                        self.stepPathNew(startPosition2, endPosition2, \
                                         stepDiscretization2)


        # Set state for foot motion for 2 feet at the same time
        self.locomoteDataDict['State'] = 6
        return 0

    def moveFootLocomote(self):
        #   update this foot's and cables' positions to next position and
        # orientation
        footPosition = self.locomoteDataDict['FootStepTraj'][ \
                            self.locomoteDataDict['FootStepTrajIndex']][:3]
        footOrientation = self.locomoteDataDict['FootStepTraj'][ \
                            self.locomoteDataDict['FootStepTrajIndex']][3]

        ############################CHANGED HERE
        self.limbs[self.locomoteDataDict['FootIndex'] \
                   ].moveLimb(footPosition, [0.0, 0.0, footOrientation])

        # update locomoteDataDict to go to the next point in the step
        # trajectory
        self.locomoteDataDict['FootStepTrajIndex'] = \
                    self.locomoteDataDict['FootStepTrajIndex'] + 1
        # if step counter is at array end, set state to move Balloon
        # otherwise continue to move foot
        if ((self.locomoteDataDict['FootStepTrajIndex']) == \
                            len(self.locomoteDataDict['FootStepTraj'])):
            # Set state for balloon move
            self.locomoteDataDict['State'] = 3

        return 0

    def moveTwoFootLocomote(self):
        #   update this foot's and cables' positions to next position and
        # orientation
        footPosition1 = self.locomoteDataDict['FootStepTraj1'][ \
                            self.locomoteDataDict['FootStepTrajIndex']][:3]
        footOrientation1 = self.locomoteDataDict['FootStepTraj1'][ \
                            self.locomoteDataDict['FootStepTrajIndex']][3]

        ############################CHANGED HERE
        self.limbs[self.locomoteDataDict['FootIndex'] \
                   ].moveLimb(footPosition1, [0.0, 0.0, footOrientation1])

        footPosition2 = self.locomoteDataDict['FootStepTraj2'][ \
                            self.locomoteDataDict['FootStepTrajIndex']][:3]
        footOrientation2 = self.locomoteDataDict['FootStepTraj2'][ \
                            self.locomoteDataDict['FootStepTrajIndex']][3]

        ############################CHANGED HERE
        self.limbs[BALLET_LIMBS - self.locomoteDataDict['FootIndex'] - 1 \
                   ].moveLimb(footPosition2, [0.0, 0.0, footOrientation2])

        # update locomoteDataDict to go to the next point in the step
        # trajectory
        self.locomoteDataDict['FootStepTrajIndex'] = \
                    self.locomoteDataDict['FootStepTrajIndex'] + 1
        # if step counter is at array end, set state to move Balloon
        # otherwise continue to move foot
        if ((self.locomoteDataDict['FootStepTrajIndex']) == \
                            len(self.locomoteDataDict['FootStepTraj1'])):
            # Set state for balloon move
            self.locomoteDataDict['State'] = 7

        return 0
```

```
def moveBalloonLocomote(self):
    #   update balloon to next position - 1/6 of step
    currentBalloonPosition = self.locomoteDataDict['BalletPath'][0] \
        [self.locomoteDataDict['BalletPathIndex']][:3]
    nextBalloonPosition = self.locomoteDataDict['BalletPath'][0] \
        [self.locomoteDataDict['BalletPathIndex'] + 1][:3]

    balloon_distance = self.norm3(self.diff3(nextBalloonPosition, \
                            currentBalloonPosition))
    self.step_portion = self.step_portion + \
                    STEP_MOTION_DISCREZTIZATION/balloon_distance
    footIndex = self.locomoteDataDict['FootIndex'] + 1
    if self.step_portion > footIndex/BALLET_LIMBS:
        self.step_portion = footIndex/BALLET_LIMBS

    current_yaw = self.locomoteDataDict['BalletPath'][0] \
        [self.locomoteDataDict['BalletPathIndex']][3]
    next_yaw = self.locomoteDataDict['BalletPath'][0] \
        [self.locomoteDataDict['BalletPathIndex'] + 1][3]

    currentTerrainSlope = self.terrain.slope(currentBalloonPosition[0], \
            currentBalloonPosition[1], current_yaw,
            BALLOON_POSITION_INCREMENT)
    nextTerrainSlope = self.terrain.slope(nextBalloonPosition[0], \
            nextBalloonPosition[1], next_yaw,
            BALLOON_POSITION_INCREMENT)

    ############################CHANGED HERE
    currentOrientation = [0.0, currentTerrainSlope, \
        self.locomoteDataDict['BalletPath'][0] \
        [self.locomoteDataDict['BalletPathIndex']][3]]

    ############################CHANGED HERE
    nextOrientation = [0.0, nextTerrainSlope, \
        self.locomoteDataDict['BalletPath'][0] \
            [self.locomoteDataDict['BalletPathIndex'] + 1][3]]

    newBalloonPosition = self.sum3(self.scale3(self.step_portion,
            self.diff3( nextBalloonPosition, currentBalloonPosition)), \
            currentBalloonPosition)
    newBalloonOrientation = self.sum3(self.scale3(self.step_portion,
            self.diff3( nextOrientation, currentOrientation)), \
            currentOrientation)

    # This function should also update all BALLET cable lengths for
    # new balloon position and orientation
    self.position = newBalloonPosition
    self.orientation = newBalloonOrientation
    self.balloon.move(newBalloonPosition, newBalloonOrientation)

    # Update ballet attach points with data from balloon
    for index in range(0, BALLET_LIMBS):
        limb_attach_point_0 = self.balloon.attach_points[
                                LIMB_ATTACHMENTS[index][0]]
        limb_attach_point_1 = self.balloon.attach_points[
                                LIMB_ATTACHMENTS[index][1]]
        limb_attach_point_2 = self.balloon.attach_points[
                                LIMB_ATTACHMENTS[index][2]]
        limb_attach_points = np.vstack(
            [limb_attach_point_0, limb_attach_point_1, limb_attach_point_2])

        self.limbs[index].updateAttachPoints(limb_attach_points)

    # Update the attach points in Ballet
    for i in range(BALLET_ATTACH_POINTS):
        for j in range(3):
            self.balloon_attach_points[i][j] = \
                        self.balloon.attach_points[i][j]

    # if not completed the balloon motion for this step
    # continue moving the balloon
    self.locomoteDataDict['FootStepTrajIndex'] = 0
    self.debug_variable1 = self.step_portion
    self.debug_variable2 = footIndex/BALLET_LIMBS
    if (self.step_portion < footIndex/BALLET_LIMBS):
        self.locomoteDataDict['State'] = 3
    # update current_foot pointer to next foot
    elif footIndex < BALLET_LIMBS:
        self.locomoteDataDict['FootIndex'] = footIndex
        self.locomoteDataDict['State'] = 1
    # set foot pointer to first foot
    else:
        self.step_portion = 0.0
        self.locomoteDataDict['FootIndex'] = 0
        # if at end of trajectory, terminate
        if (self.locomoteDataDict['BalletPathIndex'] + 2) == \
            len(self.locomoteDataDict['BalletPath'][0]):
            self.locomoteDataDict['State'] = 4
```

```
            else:
                # else, move the first foot
                self.locomoteDataDict['State'] = 1
                # update the trajectory pointer
                self.locomoteDataDict['BalletPathIndex'] = \
                    self.locomoteDataDict['BalletPathIndex'] + 1
        return 0

    def moveBalloonLocomoteNew(self):
        #   update balloon to next position - 1/6 of step
        currentBalloonPosition = self.locomoteDataDict['BalletPath'][0] \
            [self.locomoteDataDict['BalletPathIndex']][:3]
        nextBalloonPosition = self.locomoteDataDict['BalletPath'][0] \
            [self.locomoteDataDict['BalletPathIndex'] + 1][:3]

        balloon_distance = self.norm3(self.diff3(nextBalloonPosition, \
                                currentBalloonPosition))
        self.step_portion = self.step_portion + \
                        STEP_MOTION_DISCREZTIZATION/balloon_distance
        footIndex = self.locomoteDataDict['FootIndex'] + 1
        if self.step_portion > 2.0*footIndex/BALLET_LIMBS:
            self.step_portion = 2.0*footIndex/BALLET_LIMBS

        current_yaw = self.locomoteDataDict['BalletPath'][0] \
            [self.locomoteDataDict['BalletPathIndex']][3]
        next_yaw = self.locomoteDataDict['BalletPath'][0] \
            [self.locomoteDataDict['BalletPathIndex'] + 1][3]

        currentTerrainSlope = self.terrain.slope(currentBalloonPosition[0], \
                currentBalloonPosition[1], current_yaw,
                BALLOON_POSITION_INCREMENT)
        nextTerrainSlope = self.terrain.slope(nextBalloonPosition[0], \
                nextBalloonPosition[1], next_yaw,
                BALLOON_POSITION_INCREMENT)

        ############################CHANGED HERE
        currentOrientation = [0.0, currentTerrainSlope, \
            self.locomoteDataDict['BalletPath'][0] \
            [self.locomoteDataDict['BalletPathIndex']][3]]

        ############################CHANGED HERE
        nextOrientation = [0.0, nextTerrainSlope, \
            self.locomoteDataDict['BalletPath'][0] \
                [self.locomoteDataDict['BalletPathIndex'] + 1][3]]

        newBalloonPosition = self.sum3(self.scale3(self.step_portion,
                self.diff3( nextBalloonPosition, currentBalloonPosition)), \
                currentBalloonPosition)
        newBalloonOrientation = self.sum3(self.scale3(self.step_portion,
                self.diff3( nextOrientation, currentOrientation)), \
                currentOrientation)

        # This function should also update all BALLET cable lengths for
        # new balloon position and orientation
        self.position = newBalloonPosition
        self.orientation = newBalloonOrientation
        self.balloon.move(newBalloonPosition, newBalloonOrientation)

        # Update ballet attach points with data from balloon
        for index in range(0, BALLET_LIMBS):
            limb_attach_point_0 = self.balloon.attach_points[
                                        LIMB_ATTACHMENTS[index][0]]
            limb_attach_point_1 = self.balloon.attach_points[
                                        LIMB_ATTACHMENTS[index][1]]
            limb_attach_point_2 = self.balloon.attach_points[
                                        LIMB_ATTACHMENTS[index][2]]
            limb_attach_points = np.vstack(
                [limb_attach_point_0, limb_attach_point_1, limb_attach_point_2])

            self.limbs[index].updateAttachPoints(limb_attach_points)

        # Update the attach points in Ballet
        for i in range(BALLET_ATTACH_POINTS):
            for j in range(3):
                self.balloon_attach_points[i][j] = \
                                    self.balloon.attach_points[i][j]
```

```
        # if not completed the balloon motion for this step
        # continue moving the balloon
        self.locomoteDataDict['FootStepTrajIndex'] = 0
        self.debug_variable1 = self.step_portion
        self.debug_variable2 = footIndex/BALLET_LIMBS
        if (self.step_portion < footIndex/(BALLET_LIMBS/2)):
            self.locomoteDataDict['State'] = 7
        # update current_foot pointer to next foot
        elif footIndex < BALLET_LIMBS/2:
            self.locomoteDataDict['FootIndex'] = footIndex
            self.locomoteDataDict['State'] = 5
        # set foot pointer to first foot
        else:
            self.step_portion = 0.0
            self.locomoteDataDict['FootIndex'] = 0
            # if at end of trajectory, terminate
            if (self.locomoteDataDict['BalletPathIndex'] + 2) == \
                len(self.locomoteDataDict['BalletPath'][0]):
                self.locomoteDataDict['State'] = 4
            else:
                # else, move the first foot
                self.locomoteDataDict['State'] = 5
                # update the trajectory pointer
                self.locomoteDataDict['BalletPathIndex'] = \
                    self.locomoteDataDict['BalletPathIndex'] + 1
        return 0

    def terminateLocomote(self):
        self.locomoteDataDict['State'] = 0
        self.locomoteDataDict['Destination'] = [0.0, 0.0]
        self.locomoteDataDict['BalletPath'] = list()
        self.locomoteDataDict['BalletPathIndex'] = 0
        self.locomoteDataDict['FootIndex'] = -1
        self.locomoteDataDict['FootStepTraj'] = list()
        self.locomoteDataDict['FootStepTrajIndex'] = 0

        return 0

    def locomoteTo(self):
        # This is called repeatedly until BALLET has completed
        # locomoting through its path
        # In each call, the current phase of the motion is updated
        # Phases are the following:
        # inactiveLocomote: not running Locomote
        # newFootLocomote: compute the foot step trajectory
        # moveFootLocomote: move the foot along its step trajectory
        # moveBalloonLocomote: move the balloon along its trajectory
        # terminateLocomote: terminate locomotion
        state = self.locomoteDataDict['State']
        switcher = {
            0: self.inactiveLocomote,
            1: self.newFootLocomote,
            2: self.moveFootLocomote,
            3: self.moveBalloonLocomote,
            4: self.terminateLocomote,
            5: self.newTwoFootLocomote,
            6: self.moveTwoFootLocomote,
            7: self.moveBalloonLocomoteNew,
            }
        switcher.get(state, lambda: "Invalid locomote state")()

    # Returns the norm of the specified 3-D vector
    def norm3(self, a):
        return math.sqrt(a[0]*a[0] + a[1]*a[1] + a[2]*a[2])

    # Returns the vector difference between the 3D vectors a and b
    def diff3(self, a, b):
        return([a[0]-b[0], a[1]-b[1], a[2]-b[2]])

    # Returns the sum of the 3-D vectors
    def sum3(self, a, b):
        return([a[0]+b[0], a[1]+b[1], a[2]+b[2]])

    # Returns the factor scaled vector
    def scale3(self, factor, vector):
        return(factor * vector[0], factor * vector[1], factor * vector[2])
```

```
    # Computes the path in 3D space for a foot to take a step from
    # a start position to an end position. The path is assumed to be
    # 1) a vertical lift of the foot to a specified height
    # 2) a horizontal traverse to the point above the end position
    # 3) a vertical drop to the end position on the ground
    # NOTE: Check for start and end positions on the ground should be done
    # outside this function
    def stepPath(self, start_position, end_position):
        # Given a start x,y,z position and an end x,y,z position,
        # compute the intermediate x, y, z positions in taking a step
        step_trajectory = []
        step_trajectory.append([start_position[0], start_position[1], \
            start_position[2], start_position[3]])

        # Lift up part of trajectory, x, y and orientation don't change
        new_z = start_position[2] + STEP_MOTION_DISCREZTIZATION
        while (new_z < start_position[2] + OBSTACLE_HEIGHT):
            step_trajectory.append([start_position[0], start_position[1],
                    new_z, start_position[3]])
            new_z = new_z + STEP_MOTION_DISCREZTIZATION
        last_position = [start_position[0], start_position[1], \
                start_position[2] + OBSTACLE_HEIGHT, start_position[3]]
        step_trajectory.append(last_position)

        # Straight-line lateral path part of trajectory
        lateral_distance = self.norm3(self.diff3(end_position,start_position))
        lateral_step = [STEP_MOTION_DISCREZTIZATION/lateral_distance * \
            (end_position[0]-start_position[0]), \
            STEP_MOTION_DISCREZTIZATION/lateral_distance * \
                (end_position[1]-start_position[1]), \
            STEP_MOTION_DISCREZTIZATION/lateral_distance * \
                (end_position[2]-start_position[2]), \
            STEP_MOTION_DISCREZTIZATION/lateral_distance * \
                (end_position[3]-start_position[3])]
        new_position = last_position
        end_top_position = [end_position[0], end_position[1], \
            end_position[2] + OBSTACLE_HEIGHT, end_position[3]]
        while (self.norm3(self.diff3(end_top_position, new_position)) > \
            STEP_MOTION_DISCREZTIZATION):
            last_position = [new_position[0]+lateral_step[0], \
                             new_position[1]+lateral_step[1], \
                             new_position[2]+lateral_step[2],
                             new_position[3]+lateral_step[3]]
            step_trajectory.append(last_position)
            new_position = last_position
        step_trajectory.append([end_position[0], end_position[1], \
            end_position[2] + OBSTACLE_HEIGHT, end_position[3]])

        # Put down part of trajectory
        new_z = end_position[2] + OBSTACLE_HEIGHT - STEP_MOTION_DISCREZTIZATION
        while (new_z > end_position[2]):
            step_trajectory.append([end_position[0], end_position[1], new_z,
                        end_position[3]])
            new_z = new_z - STEP_MOTION_DISCREZTIZATION
        step_trajectory.append([end_position[0], end_position[1],\
                    end_position[2], end_position[3]])
        return step_trajectory

    def stepPathNew(self, start_position, end_position, stepDiscretization):
        # Given a start x,y,z position and an end x,y,z position,
        # compute the intermediate x, y, z positions in taking a step
        step_trajectory = []
        step_trajectory.append([start_position[0], start_position[1], \
            start_position[2], start_position[3]])

        # Lift up part of trajectory, x, y and orientation don't change
        new_z = start_position[2] + stepDiscretization * \
                        STEP_MOTION_DISCREZTIZATION
        while (new_z < start_position[2] + OBSTACLE_HEIGHT):
            step_trajectory.append([start_position[0], start_position[1],
                    new_z, start_position[3]])
            new_z = new_z + stepDiscretization * \
                            STEP_MOTION_DISCREZTIZATION
        last_position = [start_position[0], start_position[1], \
                start_position[2] + OBSTACLE_HEIGHT, start_position[3]]
        step_trajectory.append(last_position)
```

```
        # Straight-line lateral path part of trajectory
        lateral_distance = self.norm3(self.diff3(end_position,start_position))
        lateral_step = [stepDiscretization * \
                        STEP_MOTION_DISCREZTIZATION/lateral_distance * \
            (end_position[0]-start_position[0]), stepDiscretization * \
            STEP_MOTION_DISCREZTIZATION/lateral_distance * \
                (end_position[1]-start_position[1]), stepDiscretization * \
            STEP_MOTION_DISCREZTIZATION/lateral_distance * \
                (end_position[2]-start_position[2]), stepDiscretization * \
            STEP_MOTION_DISCREZTIZATION/lateral_distance * \
                (end_position[3]-start_position[3])]
        new_position = last_position
        end_top_position = [end_position[0], end_position[1], \
            end_position[2] + OBSTACLE_HEIGHT, end_position[3]]
        while (self.norm3(self.diff3(end_top_position, new_position)) > \
            STEP_MOTION_DISCREZTIZATION):
            last_position = [new_position[0]+lateral_step[0], \
                             new_position[1]+lateral_step[1], \
                             new_position[2]+lateral_step[2], \
                             new_position[3]+lateral_step[3]]
            step_trajectory.append(last_position)
            new_position = last_position
        step_trajectory.append([end_position[0], end_position[1], \
            end_position[2] + OBSTACLE_HEIGHT, end_position[3]])

        # Put down part of trajectory
        new_z = end_position[2] + OBSTACLE_HEIGHT - stepDiscretization * \
                                  STEP_MOTION_DISCREZTIZATION
        while (new_z > end_position[2]):
            step_trajectory.append([end_position[0], end_position[1], new_z,
                        end_position[3]])
            new_z = new_z - stepDiscretization * STEP_MOTION_DISCREZTIZATION
        step_trajectory.append([end_position[0], end_position[1],\
                    end_position[2], end_position[3]])
        return step_trajectory

    # Function to print a message for debugging
    def logMsg(self):
        # For debugging
        logging.debug("PrintMessage:")
        logging.debug("PrintData: ")
        logging.debug(self.cable_diameter)
        return


# storage for the instance reference
__instance = None

def __init__(self, x_pos = 0, y_pos = 0, z_pos = 0, heading_angle = 0, \
                                                    terrain = None):
    """ Create singleton instance """
    # Check whether we already have an instance
    if Ballet.__instance is None:
        # Create and remember instance
        Ballet.__instance = \
            Ballet.__impl(x_pos, y_pos, z_pos, heading_angle, terrain)
        print("Creating Ballet object")
    else:
        print("Using previously created Ballet object")

    # Store instance reference as the only member in the handle
    self.__dict__['_Ballet__instance'] = Ballet.__instance

def __getattr__(self, attr):
    """ Delegate access to implementation """
    return getattr(self.__instance, attr)

def __setattr__(self, attr, value):
    """ Delegate access to implementation """
    return setattr(self.__instance, attr, value)
```

```
class Balloon():
    def __init__ (self, position, orientation, attach_points):
        # Initialize the balloon in the designated position and heading
        # (yaw) with initial pitch and roll set to zero.
        self.position = position
        self.orientation = orientation
        self.attach_points = attach_points
        self.dimensions = BALLOON_DIMENSIONS
        self.attach_point_offset = np.zeros([BALLET_ATTACH_POINTS, 3])

        # Specify the attach points in the local balloon coordinate frame
        for index in range(0,BALLET_ATTACH_POINTS):
            # The first coordinate of the attach point - always the x coordinate
            # Save the attach point positions as the offset. These will
            # be used to compute the new positions based on the orientation
            # of the balloon
            self.attach_point_offset[index,0] = BALLOON_DIMENSIONS[0] * \
                math.cos(ATTACH_LOCATIONS[index][2]*math.pi/12)

            self.attach_points[index,0] = self.attach_point_offset[index,0] + \
                position[0]

            # The second coordinate of the attach point
            # - either the y or the z coordinate
            # Note that the remaining coordinate is initialized to zero
            # ATTACH_LOCATIONS[index][1] is equal to 1
            if ATTACH_LOCATIONS[index][1] == 1:
                self.attach_point_offset[index,1] = BALLOON_DIMENSIONS[1] * \
                    math.sin(ATTACH_LOCATIONS[index][2]*math.pi/12)

                self.attach_points[index, 1] = self.attach_point_offset[index,1] \
                    + position[1]

                self.attach_point_offset[index,2] = 0.0
                self.attach_points[index, 2] = position[2]

            # ATTACH_LOCATIONS[index][1] is equal to 2
            else:
                self.attach_point_offset[index,2] = \
                    BALLOON_DIMENSIONS[2] * \
                    math.sin(ATTACH_LOCATIONS[index][2]*math.pi/12)
                self.attach_points[index, 2] = self.attach_point_offset[index,2] \
                    + position[2]

                self.attach_point_offset[index,1] = 0.0
                self.attach_points[index, 1] = position[1]

        self.move(position, orientation)

    def move(self, position, orientation):
        # function moves balloon position (x, y, z) and orientation
        # (yaw and pitch), updates cable attach points
        # and updates limbs/cables for new balloon position - this does
        # not change the foot positions - assumed fixed on the ground
        self.position = position
        self.orientation = orientation
        rotation = Euler(orientation, 'ZYX')
        rotation_matrix = rotation.to_matrix()
        for index in range(0,BALLET_ATTACH_POINTS):
            point = Vector(self.attach_point_offset[index])
            point.rotate(rotation_matrix)
            self.attach_points[index, 0] = point[0] + position[0]
            self.attach_points[index, 1] = point[1] + position[1]
            self.attach_points[index, 2] = point[2] + position[2]
        return 0
```

```
# A Limb consists of a Foot and three LimbCable objects.
# The Limb moves for Ballet to locomote. The Limb can take a
# step to a new position.
class Limb():
    def __init__(self, index, x_init, y_init, heading_init,
                                                terrain, attach_points):
        # Initialize the limb foot position and the cables
        self.index = index
        self.terrain = terrain
        z_init = terrain.height(x_init, y_init)
        self.position = np.array([x_init, y_init, z_init])
        self.attach_points = attach_points

        # Initialize to orient with heading
        self.orientation = np.array([heading_init, 0.0, 0.0])
        self.foot = Foot(index, self.position, self.orientation)

        # Set up the 3 cables of the limb
        self.cable = [0]*CABLES_FOOT
        for index in range(0,CABLES_FOOT):
            self.cable[index] = LimbCable(index,
                        self.position, attach_points[index])

    # Return the foot
    def getFoot(self):
        return self.foot

    # Return the limb i.e. foot position
    def getPosition(self):
        return self.position

    # Return the limb i.e. foot orientation
    def getOrientation(self):
        return self.orientation

    def updateAttachPoints(self, attach_points):
        # Set new cable balloon attach positions
        for index in range(0,CABLES_FOOT):
            self.cable[index].balloon_end_position = attach_points[index]
            self.attach_points[index] = attach_points[index]

        return 0

    def moveLimb(self, new_position, new_orientation):
        # Set new cable positions and foot_position

        # foot_top_position = new_position
        # foot_top_position[2] = new_position[2] + FOOT_SIZE[2]
        foot_top_position = np.array([new_position[0], new_position[1], \
                new_position[2] + FOOT_SIZE[2]])

        for index in range(0,CABLES_FOOT):
            self.cable[index].foot_end_position = foot_top_position
            self.cable[index].balloon_end_position = self.attach_points[index]
        self.foot.position = new_position
        self.foot.orientation = new_orientation

        return 0

    # Return True if pt is in the triangle formed by A, B and C, else False
    def pointInTriangle(self, pt, tria, trib, tric):
        as_x = pt[0]-tria[0]
        as_y = pt[1]-tria[1]
        bs_x = pt[0]-trib[0]
        bs_y = pt[1]-trib[1]
        cs_x = pt[0]-tric[0]
        cs_y = pt[1]-tric[1]

        s_ab = (trib[0]-tria[0])*as_y - (trib[1]-tria[1])*as_x
        s_bc = (tric[0]-trib[0])*bs_y - (tric[1]-trib[1])*bs_x
        s_ca = (tria[0]-tric[0])*cs_y - (tria[1]-tric[1])*cs_x

        if (s_ab>0 and s_bc>0 and s_ca>0) or (s_ab<0 and s_bc<0 and s_ca<0):
            return True
        else:
            return False
```

```
# The Foot is moved to locomote. The Foot has a position and
# orientation.
class Foot():
    def __init__(self, index, position, orientation):
        # Set up the foot to hold the foot data
        self.index = index
        self.position = position
        self.orientation = orientation
        self.size = FOOT_SIZE

    # Return the position of the foot
    def getPosition(self):
        return self.position

    # Return the orientation of the foot
    def getOrientation(self):
        return self.orientation

    def getSize(self):
        return self.size

    def setPosition(self, position):
        self.position = position

    def setOrientation(self, orientation):
        self.orientation = orientation

# There are 3 LimbCable for each Foot. The LimbCable has a foot
# position and a balloon position that define its ends. The
# vector difference between these points give the lengths
# of the cable.
class LimbCable():
    def __init__(self, index, foot_position, balloon_attach_position):
        self.index = index
        foot_top_position = np.array([foot_position[0], foot_position[1], \
                    foot_position[2] + FOOT_SIZE[2]])
        self.foot_end_position = foot_top_position
        self.balloon_end_position = balloon_attach_position
        dx = self.balloon_end_position[0] - self.foot_end_position[0]
        dy = self.balloon_end_position[1] - self.foot_end_position[1]
        dz = self.balloon_end_position[2] - self.foot_end_position[2]
        self.original_length = math.sqrt(dx**2 + dy**2 + dz**2)

    def getOriginalLength(self):
        return(self.original_length)

    def getLength(self):
        dx = np.asscalar(self.balloon_end_position[0] - self.foot_end_position[0])
        dy = np.asscalar(self.balloon_end_position[1] - self.foot_end_position[1])
        dz = np.asscalar(self.balloon_end_position[2] - self.foot_end_position[2])
        return(math.sqrt(dx**2 + dy**2 + dz**2))
```

```
# This class contains the digital elevation model
# for the terrain. Set up a flat zero-height terrain initially.
class Terrain():
    def __init__(self, x_size=100, y_size=100, x_min=-50, y_min=-50,
                 x_max=50, y_max=50, path = '', filename = '',
                 height_scale=1.0, position_offset = Vector((0,0,0))):
        # Initially create a flat terrain at zero height
        self.bounds = np.array([x_min, y_min, x_max, y_max])
        self.x_length = x_max - x_min
        self.y_length = y_max - y_min
        self.x_resolution = x_size
        self.y_resolution = y_size
        self.x_cellsize = self.x_length/(x_size - 1)
        self.y_cellsize = self.y_length/(y_size - 1)
        self.offset = position_offset

        print('Filename: ', filename)

        if (filename == ''):
            self.dem = np.zeros((x_size, y_size))
        else:
            image = imageio.imread(os.path.join(path, filename))
            self.x_resolution  = image.shape[0]
            self.y_resolution = image.shape[1]
            print("image shape: ", image.shape)
            self.dem = np.zeros((self.x_resolution, self.y_resolution))
            for i in range(0, self.x_resolution):
                for j in range(0, self.y_resolution):
                    self.dem[i,j] = self.offset[2] + (65536.0 - image[i,j]) \
                                    * height_scale / 255.0


    # Return the height of the terrain at the x,y position specified - use bi-linear
    # interpolation to determine the height.
    def height(self, x, y):
        x_local = x - self.offset[0]
        y_local = y - self.offset[1]
        b_index_x = int((x_local - self.bounds[0])*(self.x_resolution-1)/(
                                    self.bounds[2] - self.bounds[0]))
        b_index_y = int((y_local - self.bounds[1])*(self.y_resolution-1)/(
                                    self.bounds[3] - self.bounds[1]))

        height_0_0 = self.dem[b_index_x, b_index_y]
        height_0_1 = self.dem[b_index_x, b_index_y + 1]
        height_1_0 = self.dem[b_index_x + 1, b_index_y]
        height_1_1 = self.dem[b_index_x + 1, b_index_y + 1]
        x_portion = (x_local - (b_index_x*self.x_cellsize + \
                                    self.bounds[0])) / self.x_cellsize
        y_portion = (y_local - (b_index_y*self.y_cellsize + \
                                    self.bounds[0])) / self.y_cellsize

        return (height_0_0 * (1-x_portion) * (1-y_portion) + \
            height_1_0 * (x_portion) * (1-y_portion) + \
            height_0_1 * (1-x_portion) * (y_portion) + \
            height_1_1 * (x_portion) * (y_portion))

    # Return the slope of the terrain at the x,y position
    # and along the specified orientation angle over the extent specified
    def slope(self, x, y, yaw, extent):
        # Get height at front and back extent positions along the angle
        # specified
        h1 = self.height(x + extent * math.cos(yaw), y + extent * math.sin(yaw))
        h2 = self.height(x - extent * math.cos(yaw), y - extent * math.sin(yaw))
        # Compute and return the slope
        return(math.atan2(h2-h1, 2.0*extent))
```

The following is the listing of the BalletVisualization.py file for generating the 3D display and animation of BALLET

```
import bpy
import bmesh
import math
import mathutils
import random
from mathutils import Vector
from mathutils import Euler
from math import radians
import numpy as np
import Ballet
import logging

D = bpy.data
C = bpy.context

debug = False

markerLocations = np.zeros((100,6))
markerIndex = 0

TERRAIN_PATH = '/Users/hdnayar/Documents/projects/BALLET/simulation/terrain/'
GRAPHICS_PATH = '/Users/hdnayar/Documents/projects/BALLET/simulation/graphics/'
SOURCE_PATH = '/Users/hdnayar/Documents/projects/BALLET/simulation/software/BALLET/'
OUTPUT_PATH = '/Users/hdnayar/Documents/projects/BALLET/simulation/software/output/'

logging.basicConfig(filename=SOURCE_PATH+'BALLET_debug.log',level=logging.DEBUG)

BalletVisualObjects = {}

# Clear display of objects
# clear mesh and object
def deleteMeshObjects():
    for item in bpy.context.scene.objects:
        if item.type == 'MESH':
            bpy.context.scene.objects.unlink(item)
    for item in bpy.data.objects:
        if item.type == 'MESH':
            bpy.data.objects.remove(item)
    for item in bpy.data.meshes:
        bpy.data.meshes.remove(item)
    for item in bpy.data.materials:
        bpy.data.materials.remove(item)


# Set up the environment for viewing
def setViewEnvironment(shade = 'MATERIAL', lamp = 'SUN', energy = 1.0):
    scene = bpy.context.scene

    # Set up lamp to be sun with energy level
    if len(bpy.data.lamps) > 0:
        lamp = bpy.data.lamps[0]         # get the lamp data.
        lamp.type = 'SUN'
        lamp.energy = 1.0

    # Set lamp position
    for item in bpy.data.objects:
        if item.type == 'LAMP':
            item.location = (0,0,10)

    # Add a new diffuse lamp
    # Create new lamp datablock
    lamp_data = bpy.data.lamps.new(name="Diffuse_Lamp", type='SUN')
    lamp_object = bpy.data.objects.new(name="Diffuse Lamp", object_data=lamp_data)
    lamp_data.energy = 0.4
    lamp_object.location = (5,5,10)

    # Link lamp object to the scene so it'll appear in this scene
    scene.objects.link(lamp_object)

    # And finally select it make active
    lamp_object.select = True
    scene.objects.active = lamp_object

    for area in bpy.context.screen.areas:
        if area.type == 'VIEW_3D' :
            show_only_render = True
            viewport_shade = shade
            area.spaces[0].show_only_render = True
            area.spaces[0].viewport_shade = shade

    if scene.world is None:
        # create a new world
        new_world = bpy.data.worlds.new("New World")
        scene.world = new_world
```

```
    # Verify that this is useful
    # bpy.ops.texture.new()
    # bpy.data.textures["Texture"].type = 'CLOUDS'

    scene.world.zenith_color = (0.0, 0.0, 0.0)
    scene.world.horizon_color = (0.4, 0.2, 0.0)
    scene.world.use_sky_blend = True
    scene.world.mist_settings.use_mist = True


# Create terrain
def createTerrainMeshFromDEM(name, origin, dem, extent, mesh_name):
    """
    This function creates a blender terrain mesh object using
    data in the DEM centered at the origin - middle of the DEM
    dem is a 2D array of z-values corresponding to x and y coordinates
    extent contains the x and y edge lengths of the DEM.
    mesh_name is the string name of the DEM mesh
    """

    global BalletVisualObjects

    if debug:
        global markerLocations
        global markerIndex
        markerIndex = 0

    # mesh arrays
    verts = []
    faces = []

    # get the size of the DEM
    numX = dem.shape[0]
    numY = dem.shape[1]
    scaleX = extent[0]
    scaleY = extent[1]


    # fill verts array
    for i in range(0, numX):
        for j in range(0, numY):
            # nomalize range (u and v range from -0.5 to 0.5)
            u = i/(numX-1)-1/2
            v = j/(numY-1)-1/2

            x = scaleX*u + origin[0]
            y = scaleY*v + origin[1]
            z = dem[i, j]

            vert = (x, y, z)
            verts.append(vert)

            if debug:
                if i%100 == 0 and j%100 == 0:
                    drawCubeMarker((0.1, 0.1, 0.1), (x, y, z), [1.0, 0.0, 0.0])
                    markerLocations[markerIndex, 0] = x
                    markerLocations[markerIndex, 1] = y
                    markerLocations[markerIndex, 2] = z
                    markerIndex = markerIndex + 1

    # fill faces array
    count = 0
    for i in range(0, numY * (numX-1)):
        if count < numY-1:
            A = i
            B = i+1
            C = (i+numY)+1
            D = (i+numY)
            face = (A, B, C, D)
            faces.append(face)
            count = count + 1
        else:
            count = 0

    # create mesh and object
    mesh = bpy.data.meshes.new(mesh_name)
    terrainMeshObject = bpy.data.objects.new(mesh_name, mesh)

    # set mesh location
    terrainMeshObject.location = bpy.context.scene.cursor_location
    bpy.context.scene.objects.link(terrainMeshObject)
```

```
    # create mesh from python data
    mesh.from_pydata(verts, [], faces)
    mesh.update(calc_edges=True)
    m   = mesh.materials
    if(len(m) < 1):
        mat = bpy.data.materials.new('terrainMaterial')
        m.append(mat)
    addTexture(m[0], TERRAIN_PATH, "c002.png", 'ORCO')
    setMaterial(m[0], [1.0, 0.5, 0.0], [1.0, 0.5, 0.0], 0.5)

    BalletVisualObjects.update({'Terrain': terrainMeshObject})

    return terrainMeshObject

def setMaterial(bv_mat, diffuse, specular, alpha):
    bv_mat.diffuse_color = diffuse
    bv_mat.diffuse_shader = 'LAMBERT'
    bv_mat.diffuse_intensity = 1.0
    bv_mat.specular_color = specular
    bv_mat.specular_shader = 'COOKTORR'
    bv_mat.specular_intensity = 0.5
    bv_mat.alpha = alpha
    bv_mat.ambient = 1
    return bv_mat

def addTexture(bv_mat, path, filename, tex_coords):
    # Load image file from url.
    try:
        #make a temp filename that is valid on your machine
        tmp_filename = path + filename
        #fetch the image in this file
        img = bpy.data.images.load(tmp_filename)
        #pack the image in the blender file so...
        img.pack()
    except Exception as e:
        raise NameError("Cannot load image: {0}".format(e))

    # Create image texture from image
    new_texture = bpy.data.textures.new('NewTexture', type='IMAGE')
    new_texture.image = img

    # Add texture slot for color texture
    mtex = bv_mat.texture_slots.add()
    mtex.texture = new_texture
    mtex.texture_coords = tex_coords

def drawCubeMarker(size, location, m_color):
    # print('Location: ', location)
    markerMesh = bpy.ops.mesh.primitive_cube_add(location=location)
    solid = bpy.context.active_object

    # scale dimensions for balloon dimensions
    solid.scale = size
    solid.name = 'marker'

    m   = solid.data.materials
    if(len(m) < 1):
        mat = bpy.data.materials.new('markerMaterial')
        m.append(mat)

    setMaterial(m[0], m_color, m_color, 0.5)

    # Blender.Redraw()

    return markerMesh

def drawCylinderMarker(size, location, m_color):
    # print('Location: ', location)
    markerMesh = bpy.ops.mesh.primitive_cylinder_add(location=location)
    solid = bpy.context.active_object

    # scale dimensions for balloon dimensions
    solid.scale = size
    solid.name = 'marker'

    m   = solid.data.materials
    if(len(m) < 1):
        mat = bpy.data.materials.new('markerMaterial')
        m.append(mat)

    setMaterial(m[0], m_color, m_color, 0.5)

    # Blender.Redraw()

    return markerMesh
```

```
def testDEM():
    # mesh arrays

    extent = 100
    max_elev = 10.0
    verts = []
    faces = []
    test_dem = np.zeros((extent, extent))
    global markerLocations
    global markerIndex

    for i in range (0, extent):
        for j in range (0,extent):
            test_dem[i,j] = max_elev * math.sin(i*math.pi/extent) \
                                * math.sin(j*math.pi/extent)

    for i in range(0, extent):
        for j in range(0, extent):
            # nomalize range (u and v range from -0.5 to 0.5)
            u = float(i)/float(extent-1)-float(1/2)
            v = float(j)/float(extent-1)-float(1/2)

            x = extent*u
            y = extent*v
            z = test_dem[i, j]

            vert = (x, y, z)
            verts.append(vert)

            if i%10 == 0 and j%10 == 0:
                # drawCubeMarker((0.1, 0.1, 0.1), (x, y, z), [1.0, 0.0, 0.0])
                markerLocations[markerIndex, 0] = x
                markerLocations[markerIndex, 1] = y
                markerLocations[markerIndex, 2] = z
                markerIndex = markerIndex + 1


    # fill faces array
    count = 0
    for i in range(0, extent * (extent-1)):
        if count < extent-1:
            A = i
            B = i+1
            C = (i+extent)+1
            D = (i+extent)
            face = (A, B, C, D)
            faces.append(face)
            count = count + 1
        else:
            count = 0

    # create mesh and object
    mesh = bpy.data.meshes.new('test_dem')
    terrainMeshObject = bpy.data.objects.new('test_dem', mesh)

    # set mesh location
    terrainMeshObject.location = bpy.context.scene.cursor_location
    bpy.context.scene.objects.link(terrainMeshObject)

    # create mesh from python data
    mesh.from_pydata(verts, [], faces)
    mesh.update(calc_edges=True)

    m   = mesh.materials
    if(len(m) < 1):
        mat = bpy.data.materials.new('terrainMaterial')
        m.append(mat)

    addTexture(m[0], TERRAIN_PATH, "c002.png", 'ORCO')
    setMaterial(m[0], [0.75, 0.5, 0.2], [0.75, 0.5, 0.2], 0.5)

    return test_dem

def getTestDemHeight(test_dem, x, y):
    b_index_x = int(x) + 50
    b_index_y = int(y) + 50
    cellsize = 100.0/99.0

    height_0_0 = test_dem[b_index_x, b_index_y]
    height_0_1 = test_dem[b_index_x, b_index_y + 1]
    height_1_0 = test_dem[b_index_x + 1, b_index_y]
    height_1_1 = test_dem[b_index_x + 1, b_index_y + 1]
    x_portion = (50 + x - (b_index_x*cellsize)) / cellsize
    y_portion = (50 + y - (b_index_y*cellsize)) / cellsize
```

```
    return (height_0_0 * (1-x_portion) * (1-y_portion) + \
            height_1_0 * (x_portion) * (1-y_portion) + \
            height_0_1 * (1-x_portion) * (y_portion) + \
            height_1_1 * (x_portion) * (y_portion))

# Test point in triangle
def testPointInTriangle(pt, tria, trib, tric):
    as_x = pt[0]-tria[0]
    as_y = pt[1]-tria[1]
    bs_x = pt[0]-trib[0]
    bs_y = pt[1]-trib[1]
    cs_x = pt[0]-tric[0]
    cs_y = pt[1]-tric[1]

    s_ab = (trib[0]-tria[0])*as_y - (trib[1]-tria[1])*as_x
    s_bc = (tric[0]-trib[0])*bs_y - (tric[1]-trib[1])*bs_x
    s_ca = (tria[0]-tric[0])*cs_y - (tria[1]-tric[1])*cs_x

    if (s_ab>0 and s_bc>0 and s_ca>0) or (s_ab<0 and s_bc<0 and s_ca<0):
        return True
    else:
        return False

# Create Ballet elements in Blender
def createBalletObjectsFromModel(ballet):
    """
    Draw the model - assumes that the model has already been created
    and is passed in
    """

    global BalletVisualObjects

    # Create Balloon Object
    balloonMesh = createBalloon(ballet.balloon.dimensions, ballet.position)

    BalletVisualObjects.update({'Balloon': balloonMesh})

    # Create Limb Objects
    FootObjects = {}
    for i in range(0,len(ballet.limbs)):
        foot = ballet.limbs[i].foot
        footMesh = createFoot(foot.getSize(), foot.getPosition(), i)
        FootCables = {}
        for j in range(0, len(ballet.limbs[i].cable)):
            cable = ballet.limbs[i].cable[j]
            FootCables.update({j:createLinkCable(cable.foot_end_position,
                cable.balloon_end_position, ballet.cable_diameter/2, i, j)})
        FootObjects.update({i:{'foot':footMesh, 'cables':FootCables}})

    BalletVisualObjects.update(FootObjects)
    return balloonMesh

# Update the 3-D objects of Ballet using the values in the ballet objects
def updateBalletObjectsFromModel(ballet):
    """
    Update the locations and orientations of the objects that form BALLET
    - assumes that the model has already been created and is passed in
    """

    global BalletVisualObjects

    # Create Balloon Object
    updateBalloon(BalletVisualObjects['Balloon'], \
                  ballet.position, ballet.orientation)
    cable_radius = ballet.cable_diameter/2

    # Update Limb Objects
    for i in range(0,len(ballet.limbs)):
        foot = ballet.limbs[i].foot
        updateFoot(BalletVisualObjects[i]['foot'], foot.getSize(), \
                   foot.getPosition(), foot.getOrientation())
        for j in range(0, len(ballet.limbs[i].cable)):
            cable = ballet.limbs[i].cable[j]
            cable_length = cable.getOriginalLength()
            updateLinkCable(BalletVisualObjects[i]['cables'][j],
                cable.foot_end_position, cable.balloon_end_position, \
                        cable_radius, cable_length)


def createBalloon(size, location):
    balloonMesh = bpy.ops.mesh.primitive_ico_sphere_add(
        subdivisions=5, location=location)
    solid = bpy.context.active_object
```

```
    # scale dimensions for balloon dimensions
    solid.scale = size
    solid.name = 'balloon'
    bpy.ops.object.shade_smooth()

    m   = solid.data.materials
    if(len(m) < 1):
        mat = bpy.data.materials.new('balloonMaterial')
        m.append(mat)

    setMaterial(m[0], [0.8, 0.8, 0.8], [0.8, 0.8, 0.8], 0.5)
    addTexture(m[0], GRAPHICS_PATH, "NASA6.png", 'ORCO') #'GLOBAL')

    # Blender.Redraw()
    return bpy.context.object

def updateBalloon(balloon_mesh, position, orientation):
    balloon_mesh.location = position
    balloon_mesh.rotation_euler = Euler(orientation, 'ZYX')
    return 0


def createFoot(size, location, index):
    # input loc is the location of the ground so we need to
    # determine the cube centroid position
    # from the offset of the cube height
    # NOTE: Could use terrain surface normal to orient foot in the future
    cubePosition = [location[0], location[1], location[2]+size[2]/2]
    footMesh = bpy.ops.mesh.primitive_cube_add(location=cubePosition, radius=0.5)
    solid = bpy.context.active_object
    solid.scale = size
    solid.name = 'foot_' + str(index)

    m   = solid.data.materials
    if(len(m) < 1):
        mat = bpy.data.materials.new('footMaterial')
        m.append(mat)

    setMaterial(m[0], [0.1, 0.1, 1.0], [0.1, 0.1, 1.0], 0.5)

    return bpy.context.object

def updateFoot(foot_mesh, size, position, orientation):
    cubePosition = [position[0], position[1], position[2]+size[2]/2]
    cubeOrientation = [orientation[0], 0.0, 0.0]
    foot_mesh.location=Vector(cubePosition)
    foot_mesh.rotation_euler = Euler(cubeOrientation, 'ZYX')
    return 0

def createLinkCable(footTopPos, balloonAttachPos, radius, foot_index, cable_index):

    # Save the positions of the ends of the cable
    x1, y1, z1 = footTopPos
    x2, y2, z2 = balloonAttachPos

    # Compute the length of the cable
    dx = x2 - x1
    dy = y2 - y1
    dz = z2 - z1
    length = math.sqrt(dx**2 + dy**2 + dz**2)

    # Create the cable at the appropriate mid-point location
    cableMesh = bpy.ops.mesh.primitive_cylinder_add(
        radius=radius,
        depth=length,
        location=(dx/2 + x1, dy/2 + y1, dz/2 + z1)
        )

    # Orient the cable
    phi = math.atan2(dy, dx)
    if length > 0.0:
        theta = math.acos(dz/length)
    else:
        theta = 0

    bpy.context.object.rotation_euler[1] = theta
    bpy.context.object.rotation_euler[2] = phi

    solid = bpy.context.active_object
    solid.name = 'cable_' + str(foot_index) + '_' + str(cable_index)

    m   = solid.data.materials
    if(len(m) < 1):
        mat = bpy.data.materials.new('cableMaterial')
        m.append(mat)
```

```
    return bpy.context.object

def updateLinkCable(cable_mesh, foot_top_pos, balloon_attach_pos, radius, \
                    cable_length):
    # Save the positions of the ends of the cable
    x1, y1, z1 = foot_top_pos
    x2, y2, z2 = balloon_attach_pos

    # Compute the length of the cable
    dx = x2 - x1
    dy = y2 - y1
    dz = z2 - z1
    new_length = math.sqrt(dx**2 + dy**2 + dz**2)
    scale_length = new_length/cable_length

    # Create the cable at the appropriate mid-point location
    cable_mesh.location = Vector([dx/2 + x1, dy/2 + y1, dz/2 + z1])
    cable_mesh.scale = (1, 1, scale_length)

    # Orient the cable
    phi = math.atan2(dy, dx)
    if new_length > 0.0:
        theta = math.acos(dz/new_length)
    else:
        theta = 0

    cable_mesh.rotation_euler[1] = theta
    cable_mesh.rotation_euler[2] = phi
    return 0


def takeBalletStep():

    # Use modal to update display
    # Use global to access ballet

    return {'PrintMessage'}

def cameraPath():

    # For straight line motion
    # camera_position =  np.array([[32.0, 0.0, 9.0],
    #                     [12.0, 0.0, 9.0],
    #                     [2.0, -10.0, 10.0],
    #                     [2.0, -10.0, 10.0]
    #                     ])
    # camera_orientation = np.array([[radians(90.0), radians(0.0), radians(90.0)],
    #                     [radians(90.0), radians(0.0), radians(90.0)],
    #                     [radians(80.0), radians(0.0), radians(0.0)],
    #                     [radians(80.0), radians(0.0), radians(0.0)]])

    # For curved path motion
    #camera_position =  np.array([[0.0, -10.0, 10.0],
    #                    [0.0, -10.0, 10.0],
    #                    [0.0, -10.0, 10.0],
    #                    [0.0, -10.0, 10.0]
    #                    ])
    #camera_orientation = np.array([[radians(80.0), radians(0.0), radians(0.0)],
    #                    [radians(80.0), radians(0.0), radians(0.0)],
    #                    [radians(80.0), radians(0.0), radians(0.0)],
    #                    [radians(80.0), radians(0.0), radians(0.0)]
    #                    ])
    # For 2 feet motion
    camera_position =  np.array([[0.0, -8.0, 10.0],
                        [0.0, -8.0, 10.0],
                        [0.0, -8.0, 10.0],
                        [0.0, -8.0, 10.0]
                        ])
    camera_orientation = np.array([[radians(80.0), radians(0.0), radians(0.0)],
                        [radians(80.0), radians(0.0), radians(0.0)],
                        [radians(80.0), radians(0.0), radians(0.0)],
                        [radians(80.0), radians(0.0), radians(0.0)]
                        ])

    total_steps = 2464
    pause_steps = 900
    camera_waypoints = np.array([0, pause_steps, total_steps-pause_steps, total_steps])
    move_steps_factor = 0.0
    camera_step_pos = np.array([camera_position[0]])
    camera_step_rot = np.array([camera_orientation[0]])
    new_camera_position = np.zeros(3)
    new_camera_orientation = np.zeros(3)
    delta_camera_position = np.zeros(3)
    delta_camera_orientation = np.zeros(3)
    save_pos = np.zeros(3)
    save_rot = np.zeros(3)
```

```
    for i in range(len(camera_waypoints)-1):
        #print(len(camera_waypoints)-1)
        move_steps_factor = camera_waypoints[i+1]-camera_waypoints[i]
        new_camera_position[:] = camera_position[i]
        new_camera_orientation[:] = camera_orientation[i]
        delta_camera_position[:] = (camera_position[i+1] - \
                        camera_position[i]) / move_steps_factor
        delta_camera_orientation[:] = (camera_orientation[i+1] - \
                        camera_orientation[i]) / move_steps_factor
        while len(camera_step_pos) < camera_waypoints[i+1]:
            save_pos[:] = new_camera_position
            save_rot[:] = new_camera_orientation
            new_camera_position[:] =  save_pos + delta_camera_position
            new_camera_orientation[:] =  save_rot + delta_camera_orientation
            #print(new_camera_orientation)
            camera_step_pos = np.append(camera_step_pos, \
                            np.array([new_camera_position]), axis=0)
            camera_step_rot = np.append(camera_step_rot, \
                            np.array([new_camera_orientation]), axis=0)
            #print(camera_step_pos)
            #print(camera_step_rot)
        camera_step_pos = np.append(camera_step_pos, \
                        np.array([camera_position[i+1]]), axis=0)
        camera_step_rot = np.append(camera_step_rot, \
                        np.array([camera_orientation[i+1]]), axis=0)

    return camera_step_pos, camera_step_rot


# Execute the script
def run(origo):
    origin = Vector(origo)

    # Delete any existing mesh objects
    deleteMeshesObjects()

    # Create terrain
    terrain = Ballet.Terrain(100, 100, -50, -50, 50, 50)

    # Draw Terrain
    createTerrainMeshFromDEM('test', origin, terrain.dem,
        [terrain.x_size, terrain.y_size], 'test_terrain')

    # Create Ballet
    ballet = Ballet.Ballet(0, 0, 4, 0, terrain)

    # Draw Ballet

    return

def runtest(choice):
    global markerIndex
    global markerLocations
    origin = Vector((0,0,0))
    if choice:
        td = testDEM()
    else:
        terrain = Ballet.Terrain(1000, 1000, -50, -50, 50, 50, \
            TERRAIN_PATH, 't10.png', 0.03, origin)
        createTerrainMeshFromDEM('test', origin, terrain.dem, \
                                [terrain.x_length, terrain.y_length], 'test_terrain')

    markerIndex = 0
    for i in range(0, 999):
        for j in range(0, 999):
            if i%100 == 0 and j%100 == 0:
                u = 100.0 * (float(i)/float(99)-float(1/2))
                v = 100.0 * (float(j)/float(99)-float(1/2))
                if choice:
                    tdHeight = getTestDemHeight(td, u, v)
                else:
                    tdHeight = terrain.height(u, v)
                drawCylinderMarker((0.1, 0.1, 0.1), (u, v, tdHeight), [0.0, 1.0, 1.0])
                markerLocations[markerIndex, 3] = u
                markerLocations[markerIndex, 4] = v
                markerLocations[markerIndex, 5] = tdHeight
                markerIndex = markerIndex + 1

    print(markerLocations)
```

```
class ModalTimerOperator(bpy.types.Operator):
    """Operator which runs its self from a timer"""
    bl_idname = "wm.modal_timer_operator"
    bl_label = "Modal Timer Operator"

    limits = bpy.props.IntProperty(default=0)
    _timer = None

    def modal(self, context, event):

        if event.type in {'RIGHTMOUSE', 'ESC'} or self.limits > 700: #2464:
            self.limits = 0
            self.cancel(context)
            return {'FINISHED'}

        if event.type == 'TIMER':
            ##########################################################
            # Replace this with the code to execute BALLET locomotion
            ballet = Ballet.Ballet()

            # Move to new position
            ballet.locomoteTo()

            # Update camera location and orientation
            self.cam.location = Vector((self.cam_pos[self.limits][0], \
                    self.cam_pos[self.limits][1],self.cam_pos[self.limits][2]))
            self.cam.rotation_euler = (self.cam_rot[self.limits][0], \
                    self.cam_rot[self.limits][1],self.cam_rot[self.limits][2])

            # Update display
            updateBalletObjectsFromModel(ballet)
            ballet.debug_variable1 = self.limits
            # Save rendered image to file
            bpy.data.scenes["Scene"].render.filepath = OUTPUT_PATH + \
                                'Ballet_Sim_{:05d}.png'.format(self.limits)
            bpy.ops.render.render( write_still=True )

            self.limits += 1
            ##########################################################

        return {'PASS_THROUGH'}

    def execute(self, context):
        wm = context.window_manager
        self._timer = wm.event_timer_add(time_step=0.02, window=context.window)
        wm.modal_handler_add(self)
        self.cam = bpy.data.objects['Camera']
        self.cam_pos, self.cam_rot = cameraPath()
        return {'RUNNING_MODAL'}

    def cancel(self, context):
        wm = context.window_manager
        wm.event_timer_remove(self._timer)

def register():
    bpy.utils.register_class(ModalTimerOperator)


def unregister():
    bpy.utils.unregister_class(ModalTimerOperator)

if __name__ == "__main__":
    # register()
    run((0, 0, 0))
```