ABSTRACT

BIRGE, BRIAN K. A Computational Intelligence Approach to the Mars Precision Landing Problem. (Under the direction of Dr. Gerald D. Walberg.)

Various proposed Mars missions, such as the Mars Sample Return Mission (MRSR) and the Mars Smart Lander (MSL), require precise re-entry terminal position and velocity states. This is to achieve mission objectives including rendezvous with a previous landed mission, or reaching a particular geographic landmark. The current state of the art footprint is in the magnitude of kilometers. For this research a Mars Precision Landing is achieved with a landed footprint of no more than 100 meters, for a set of initial entry conditions representing worst guess dispersions.

Obstacles to reducing the landed footprint include trajectory dispersions due to initial atmospheric entry conditions (entry angle, parachute deployment height, etc.), environment (wind, atmospheric density, etc.), parachute deployment dynamics, unavoidable injection error (propagated error from launch on), etc. Weather and atmospheric models have been developed.

Three descent scenarios have been examined. First, terminal re-entry is achieved via a ballistic parachute with concurrent thrusting events while on the parachute, followed by a gravity turn. Second, terminal re-entry is achieved via a ballistic parachute followed by gravity turn to hover and then thrust vector to desired location. Third, a guided parafoil approach followed by vectored thrusting to reach terminal velocity is examined. The guided parafoil is determined to be the best architecture.

The purpose of this study is to examine the feasibility of using a computational intelligence strategy to facilitate precision planetary re-entry, specifically to take an approach that is somewhat more intuitive and less rigid, and see where it leads. The test problems used for all research are variations on proposed mars landing mission scenarios developed by NASA.

A relatively recent method of evolutionary computation is Particle Swarm Optimization (PSO), which can be considered to be in the same general class as Genetic Algorithms. An improvement over the regular PSO algorithm, allowing tracking of non-stationary error functions is detailed. Continued refinement of PSO in the larger research community comes from attempts to understand human-human social interaction as well as analysis of the emergent behavior.

Using PSO and the parafoil scenario, optimized reference trajectories are created for an initial condition set of 76 states, representing the convex hull of 2001 states from an early Monte Carlo analysis. The controls are a set series of bank angles followed by a set series of 3DOF thrust vectoring. The reference trajectories are used to train an Artificial Neural Network Reference Trajectory Generator (ANNTraG), with the (marginal) ability to generalize a trajectory from initial conditions it has never been presented. The controls here allow continuous change in bank angle as well as thrust vector. The optimized reference trajectories represent the best achievable trajectory given the initial condition. Steps toward a closed loop neural controller with online learning updates are examined.

The inner loop of the simulation employs the Program to Optimize Simulated Trajectories (POST) as the basic model, containing baseline dynamics and state generation.

This is controlled from a MATLAB shell that directs the optimization, learning, and control strategy.

Using mainly bank angle guidance coupled with CI strategies, the set of achievable reference trajectories are shown to be 88% under 10 meters, a significant improvement in the state of the art. Further, the automatic real-time generation of realistic reference trajectories in the presence of unknown initial conditions is shown to have promise. The closed loop CI guidance strategy is outlined. An unexpected advance came from the effort to optimize the optimization, where the PSO algorithm was improved with the capability for tracking a changing error environment.

A Computational Intelligence Approach to the Mars Precision Landing Problem

by
Brian Kent Birge III

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Aerospace Engineering

Raleigh, North Carolina

2008

APPROVED BY:

_____	_____
Dr. Larry Silverberg	Dr. Paul Ro
_____	_____
Dr. Mark White	Dr. Gerald Walberg Chair of Advisory Commitee

DEDICATION

For my mother, Margaret Rose Birge.

BIOGRAPHY

Brian "Chip" Birge was born on July 31st, 1969 in Town & Country, MO, USA. He graduated from Indiana's Purdue University with a Bachelor's in Physics as well as a Bachelor's in Electrical Engineering, with a minor in Mathematics. He served two terms as IEEE Student Chapter President.

In 1998 he was accepted into the Aerospace Engineering PhD program at North Carolina State University, and became a graduate research assistant with the Mars Mission Research Center.

He has published research having to do with imaging, Particle Swarm Optimization, space situational awareness, and space object identification and characterization. He has been on research teams studying planetary science, satellite dynamics, nonlinear optimization, robotics, and optics.

Currently he resides in Houston, Texas with his wife Kwan and daughter Rachel and is employed as a principal systems engineer NASA's Johnson Space Center working on various manned and robotic space exploration projects. His research activities continue to focus on applied and theoretical computational intelligence, high performance computing, and simulation & modeling.

.

ACKNOWLEDGEMENTS

There are several people over the years that provided me with the help and inspiration to complete this research. Some know it, others do not.

My advisor Dr. Gerald Walberg wins the patience of Job award. Thanks Jerry for your tremendous insight into everything from orbital mechanics to good beer, a font of wisdom and kindness throughout my years of research.

Dr. Russell Eberhart, co-inventor of PSO and my Master's advisor whose infectious enthusiasm for the subject got me excited about computationally emergent behavior and research in general.

Astronomers, the late Dr. John Africano who made pure science fun and Dr. Doyle Hall, whose rigorous approach to real problems is a model I try to follow.

Finally, I especially appreciate the staff of the MAE department and graduate school for their excellent support throughout my time at NCSU.

.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

alppc1……first coefficient of alpha equation (POST)

alppc2……second coefficient of alpha equation (POST)

betpc1……..first coefficient of beta equation (POST)

betpc2…….second coefficient of beta equation (POST)

critr……...criterion used to activate events (POST)

depph…….event at which targeting is to be satisfied (POST)

deptl……..tolerances on dependent variables (POST)

depval……..targets of dependent variables (POST)

depvr……names of dependent variables (POST)

diamp1…….diameter of parachute #1 (POST)

dprng1……...dot product range to reference long, lat (POST)

etapc1……..throttling parameter polynomial coefficient one (POST)

gdalt……...vertical altitude above oblate planet (POST)

gdlat……..geodetic latitude (POST)

idepvr……...type of constraint desired for dependent variables (POST)

ifdeg……allows degrees to be used in targeting (POST)

iguid(1)…….guidance desired (POST)

iguid(10)…….separate channel option for pitch (POST)

iguid(11)……separate channel option for bank angle (POST)

iguid(13)……..relative yaw angle reference option flag (POST)

iguid(2)……selects either independent or identical channel steering (POST)

iguid(9)……separate channel option for yaw (POST)

indph……event that starts perturbing independent variables (POST)

indvr…….names of independent variables (POST)

isp……...specific impulse

ispv……..vacuum specific impulse (POST)

long……longitude, degrees or meters (POST)

MarsGRAM..……..program modeling mars atmosphere

Matlab……high level programming language

neng……...number of engines (POST)

npc(7)……acceleration limit option flag (POST)

opt……optimization flag (POST)

optph……optimize by this event (POST)

optvar……...optimization variables (POST)

phi……..angle between due North and target (deg)

POST……..Program to Optimize Simulated Trajectories 3D version

rn……..……nose radius (m)

sref……...aerodynamic surface area (m^2)

theta……..angle between due North and velocity vector

ur…….first component of lander horizontal velocity planet relative (POST)

velr……...vehicle velocity relative to rotating planet (POST)

vr……second component of lander horizontal velocity planet relative (POST)

wgtsg……..vehicle gross weight (POST)

wjett…….jettisoned weight (POST)

wpropi……...initial propellant weight (POST)

wr……...vertical velocity planet relative (POST)

INTRODUCTION

There is a renewed interest by the public in space exploration, sparked by such successes as the Mars Pathfinder and Orbiter missions. As engineers, each success asks us to push the boundaries a little more on the next project. It is a truism that to find the boundaries of a thing that thing must be broken. We inch forward passing incremental boundaries, each progress building on the previous. Seemingly endless pre-analysis and post-analysis by teams of qualified people dedicated to each project give us high confidence of mission achievement but even so, events happen that deviate from prediction. What can we do when a spacecraft encounters a scenario not envisioned by the mission planners? There exists a real need for adaptive control in a newer sense, not only should the system adapt to changing parameters but it should be able to modify its behavior.

The purpose of this study is to examine the feasibility of using a computational intelligence approach to facilitate precision planetary re-entry from the beginning of the Entry, Descent, Landing (EDL) phase through touchdown. The test problems used for all research are variations on proposed mars landing mission scenarios developed by NASA.

One such scenario is The Mars Rover Sample Return Mission (MRSR). The MRSR requirements make the development of an accurate descent control system challenging. This mission which was cancelled in 1990 nevertheless remains a very good representation of future Mars missions. This new class of Mars missions includes concepts such as hazard

avoidance, precision landings, active downrange and crossrange guidance, re-tasking, and resistance to atmospheric and environmental dispersion.

A baseline MRSR style mission would consist of a two tiered approach. First, a ship is sent to Mars and drops a Lander on the surface. This Lander performs some basic science tasks, among them collecting geologic samples. Rather than send a whole robotic geology laboratory to Mars to analyze the samples (a current impossibility), a second spacecraft is then sent to pick up the sample. This scenario requires the second Lander to precisely match the location of the first Lander to within some engineering capable tolerances. To make the scenario conceptually even simpler, imagine a single spacecraft sent, but this time the science team is keen on examining a very specific feature on the planet, perhaps in the middle of some very rough terrain. The technologies that would make such a Precision Landing possible now seem very desirable.

Given the current state of robotic rovers, a Mars Precision Landing requirement is desired with a landed footprint of no more than 100 meters from a target [2]. In contrast, the Viking mission had a landing footprint of over 100 kilometers (excellent for that particular mission requirement). The current state of the art's footprint is still in the magnitude of kilometers [4].

Obstacles to reducing the landed footprint include trajectory dispersions due to initial atmospheric entry conditions (entry angle, parachute deployment height, etc.), environment (wind, atmospheric density, etc.), parachute deployment dynamics, unavoidable injection error (propagated error from launch on), etc. It is the goal of this

research to reduce the simulated landing footprint to target to 100m and significantly closer in some cases.

In recent years, a number of computational techniques that lend themselves particularly well to control problems have become available. In broad categories they are Fuzzy Systems (FS), Artificial Neural Networks (ANN), and Evolutionary Computation (EC). Fuzzy Systems are based on fuzzy logic which is in turn based on the way our mind deals with incomplete and/or inaccurate information. Neural Nets are modeled after the spatial structure of the brain and allow 'connectionist' learning properties. Evolutionary computation paradigms such as Genetic Algorithms (GA) and Particle Swarm Optimization (PSO) are loosely modeled after biological evolution and optimization. These techniques have been shown individually to work very well in solving a large number of problems in linear and nonlinear system identification, modeling, and control [20].

Until recently FS, ANN, and EC were totally separate fields with very little interaction. A growing number of researchers and practical engineers are discovering that a combination of two or more of these methods offers advantages that a single one lacks.

Using an ANN in a control system we add fault tolerance, distributed (connectionist) representation properties, and the ability to learn optimal responses to new input. If we add an FS 'shell' we include high level rule/decision abilities as well as comparative reasoning. Combining techniques this way is called Computational Intelligence (CI).

This dissertation describes research into using CI strategies to solve the Mars Precision Landing problem.

BACKGROUND

Motivation for the current work springs from several areas. Historically, touchdown requirements for missions to Mars have been 'loose'. That is, the designers and mission planners were more interested in getting to Mars with a functioning vehicle than in reaching a particular geographic location. As long as the terminal velocity conditions were met and the rather large dispersion footprint was adhered to then the design was a success.

Out of the five successfully delivered robotic Mars exploration missions, all had masses below 600kg and landed at geographic positions below -1km Mars Orbiter Laser Altimeter (MOLA) average elevation in order that the entry, descent, and landing system could perform with enough atmospheric density. Also, they all had landed footprints on the order of 100km [4].

Future mission in the planning stages now will require landed payloads up to 80t (80,000 kg) carrying advanced scientific analysis packages and support technologies and located as high as +2km MOLA elevation. It is also required that these landed payloads touchdown within a close distance to pre-positioned robotic systems, within 100 meters. So far, no credible Mars EDL scheme can safely put these large payloads at high elevations in close proximity to the various terrains of interest. This is mainly due to reliance on Viking-era technology architectures, understandable as no one wants to send multi-million dollar missions out on non-flight tested strategies. There is a reasonable degree of predictability with past mission architecture. For the purposes of this study, terminal

descent phase refers to the the period starting from initial parachute deployment to first touchdown.

Viking-era EDL

The first landing standard was that of the Viking Planetary Lander. The mission was planned with two separate landed vehicles, each with autonomous navigation, guidance and control. Each of the two missions consisted of an Orbiter and a Lander. The Orbiter scanned the surface from orbit, looking for suitable places for touchdown. It also scanned for surface moisture and temperature readings as part of the search for life. Once a suitable area was found, the Lander was powered on and received a state update from the orbiter. About 30 hours later (to give time to check out the descent instrumentation), the Lander separated from the orbiter.

After separation, the Lander was rotated into position for the deorbit maneuver. After deorbit, the Lander went into a coast mode until atmospheric entry at 224km above mean surface level. This was determined by a set wait time, rather than any atmospheric sensing. It was designed such that there was approximately 7 to 10 minutes before touchdown on the surface. At the start of this 7-10 minute block, the Lander's flight computer fully powered up from sleep mode and the Lander was oriented by the Reaction Control System (RCS) such that the heat shield was positioned to meet the atmosphere. The Lander's re-entry speed at this time was about 4.6 km/s [18].

As the heat shield ablated, the Lander's velocity decreased and at about 5.79km altitude parachutes were deployed. These were ballistic parachutes, strictly for speed reduction. Seven seconds afterwards the aeroshell was discarded and landing leg extension

followed eight seconds later. 45 seconds after that at approximately 1.49km altitude and 66m/s velocity magnitude, the parachute cover was discarded. At this time, the retro rockets were ignited and individually throttled to affect pitch, yaw and terminal velocity. Engine shut off happened when the legs touched the surface via shock sensor. The nominal velocity at this time was designed to be approximately 2.44 m/s.

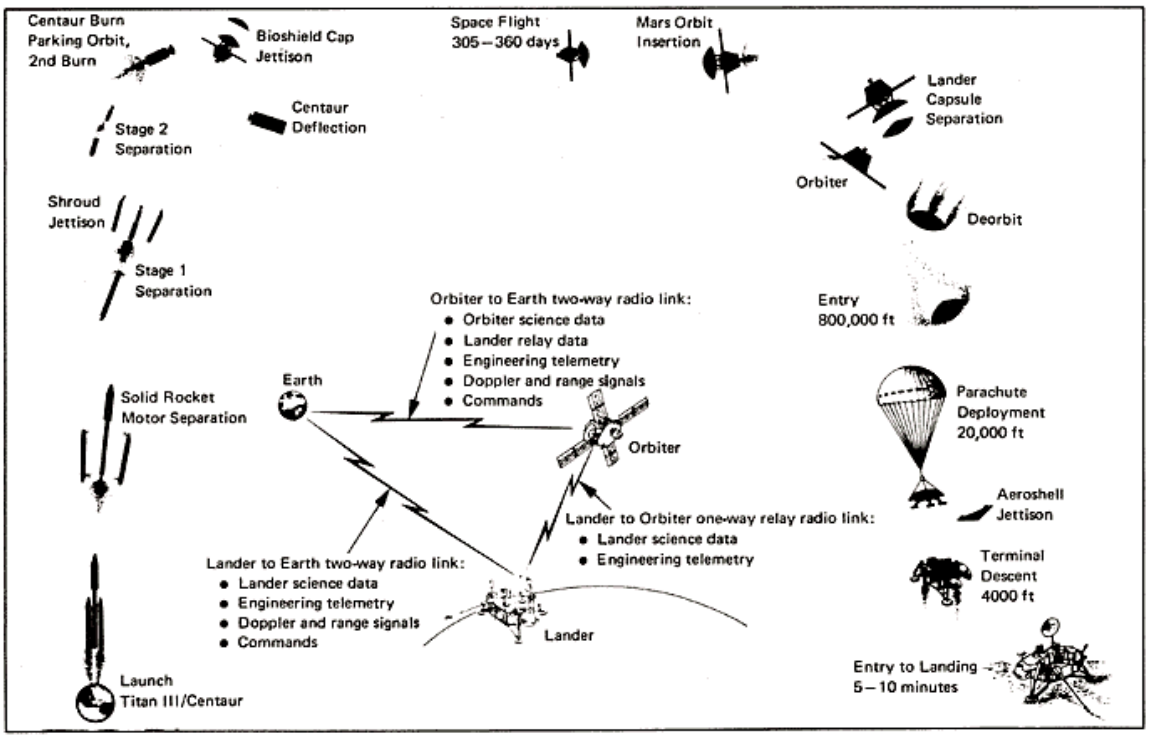

Figure 1. Viking Orbiter/Lander Mission Scheme

All the above was carried out automatically since the delay times for radio commands were about 40 minutes. In other words, once the green ‘go’ button was pressed, everyone crossed their fingers and hoped it worked out. As history shows, the Landers both performed their missions to design goals.

On June 19, 1976 Viking 1 touched down at Chryse Planitia (22.48° N, 49.97° W). Two months later on August 7, 1976 Viking 2 touched down at Utopia Planitia (47.97° N, 225.74° W). The Viking Landers sent back images of the surface, dug surface samples analyzing them for signs of life, installed seismometers, and perhaps most important for this study, they examined atmospheric composition and meteorology. It is the advances in Martian meteorology made by scientists analyzing Viking data that allow a much clearer picture of the regime in which future missions must perform. The Viking 2 Lander ended communications on April 11, 1980, and the Viking 1 Lander followed suit on November 13, 1982.

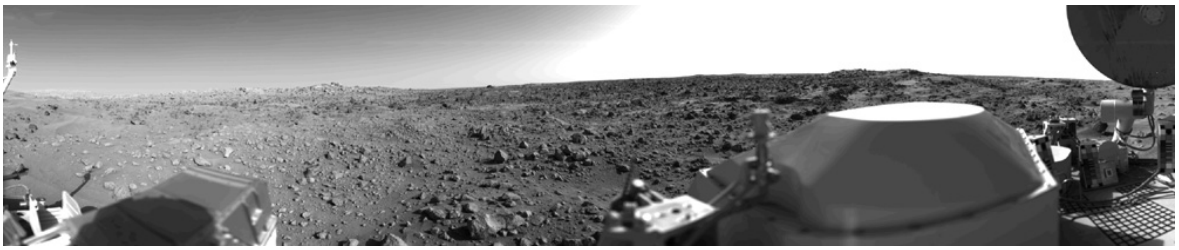

Figure 2. First panoramic view returned of Mars, Viking 1

The Viking design strategy came at a cost of a considerable amount of modeling and guesswork (albeit reasoned). Because the atmosphere was not well known, the mission had to perform to specifications within five different hypothetical Martian atmospheres. Also, because of the technological limitations and lack of knowledge about the environment, the mission was very rigid. If there had been conditions outside of the simulated nominal then the missions would have surely failed at the EDL phase. That it did not fail is a testament to the genius of the Viking engineers and scientists but is hardly a ringing endorsement of the open loop guidance system as a Precision Landing tool.

The spaceship landed basically where the scientists wanted it (give or take a 100 km), nothing was broken. However, if the Lander had encountered a dust devil (not known at the time) or a dust/sand storm (a real concern to the scientists) during re-entry then things may have gone very differently. The Lander could not re-task itself in the face of unexpected input. Additionally, the landing sites were chosen because they were expected to be free from extensive obstacles, at the trade off of being less geologically interesting than other proposed sites. With the ability to re-task during descent theoretically obstacles can be avoided, surprises can be mitigated, more interesting sites can be explored, and the Precision Landing problem can be tackled. Studies have shown that as successful as the Viking missions were, the EDL strategy will be too limiting for future Precision Landing requirements [3].

Pathfinder/MER EDL

The next advances in Mars EDL guidance came with the Mars Pathfinder Lander (MPL) and Mars Exploration Rover (MER) missions. Both Pathfinder and the MER mission had many legacy elements to the terminal descent phase, borrowing heavily from Viking based technology.

The Pathfinder was a successful example of the "better, cheaper, faster" culture at NASA. It was originally developed to be a proof of concept project to land instruments and a robotic explorer on the surface of Mars. It used a similar parachute concept to Viking to slow the descent once the atmosphere had been sensed. For the terminal landing, a Russian concept from 1971 was used, that of the airbag. Part of the Pathfinder design was

motivated by a need for cost savings when compared to the earlier Viking mission but it also took concepts from previous lunar landers and US Army payload delivery systems [28].

The Pathfinder EDL, like Viking was an autonomous rigid series of maneuvers that was triggered by a start command from mission control on Earth. The Pathfinder had direct uncontrolled (except for stability) ballistic entry that was followed by a ballistic parachute descent and an airbag bounce landing that borrowed from an earlier Russian post-Apollo era concept [9].

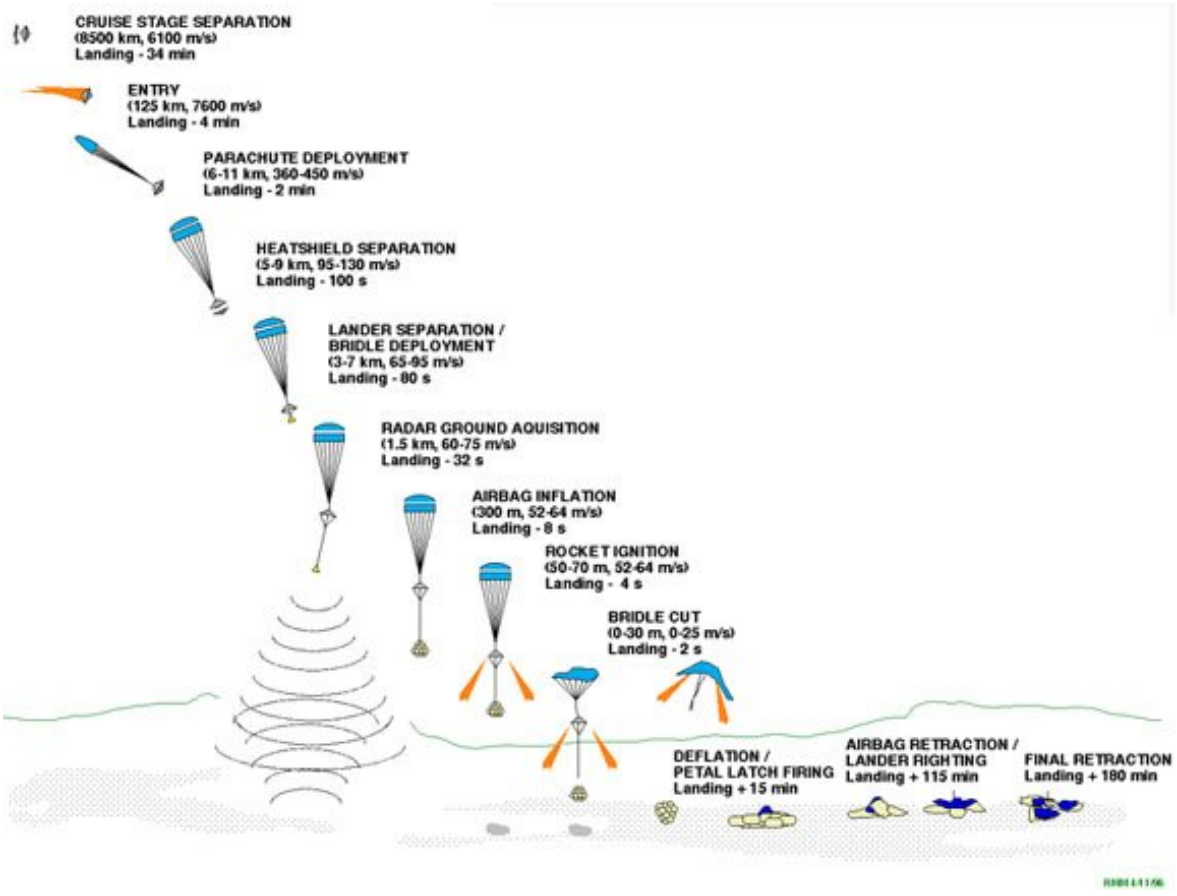

Figure 3. MPF EDL Strategy

After a spin controlled ballistic entry by a Viking style aeroshell allowed the martian atmosphere to decelerate the lander from 7.5 km/s to 400 m/s. At that time the EDL phase began. A 12.5m diameter non-maneuverable parachute was designed to be deployed at between 5 and 11 km altitude to slow the vehicle even further. 20 seconds later the heatshield was jettisoned and the lander began to separate from the backshell by sliding down 20m worth of bridle (metal tape), looking at this point much like a flying pendulum. Then the radar altimeter was activated and acquired the surface about 32 seconds before touchdown at an altitude of 1.5km. 8 seconds and 300m above the surface, the airbags inflated. At 100m above the surface, the backshell ignited a quick burst of retro rockets to slow the velocity down to near zero. At this point the lander was hanging onto the bridle with airbags inflated and was about 12m above the ground. The bridle was cut and the lander protected by airbags, fell to the surface, bounced a bit and that was that.

On July 4, 1997 Pathfinder made its descent to Mars with a 23km touchdown from nominal, well within the designed 300km by 100km touchdown ellipse science requirement.

In contrast with Pathfinder, the MER missions A and B were developed in the atmosphere of change created by the failure of the 1999 Mars missions. The missions were designed to take advantage of the favorable 2003 launch opportunity and therefore the time from development to launch was a mere 35 months. This meant that the schedule was a significant challenge.

The MER A, B missions were basically upgrades to the MPL EDL design. Initially there were no plans to change the EDL design from the MPL. However, due to

unexpectedly high winds during the actual MPL descent a novel imaging descent system was piggy-backed onto the existing technology. A new system of inertial sensors and small retro rockets was also fitted to the backshell. The imaging descent system, named DIMES (descent image motion estimation system), helped the control system correct for off nominal winds. It was used to take some snapshots of the surface, identify some features and from that determine a more accurate ground relative velocity state which the retro rockets then countered, to reduce horizontal velocity as much as possible before the bridle cut and subsequent drop to the surface. During the landing at Gusev Crater, the DIMES was able to correct for a potentially catastrophic wind condition. Because of the heavier payload, the airbags were also strengthened with a more puncture resistant material.

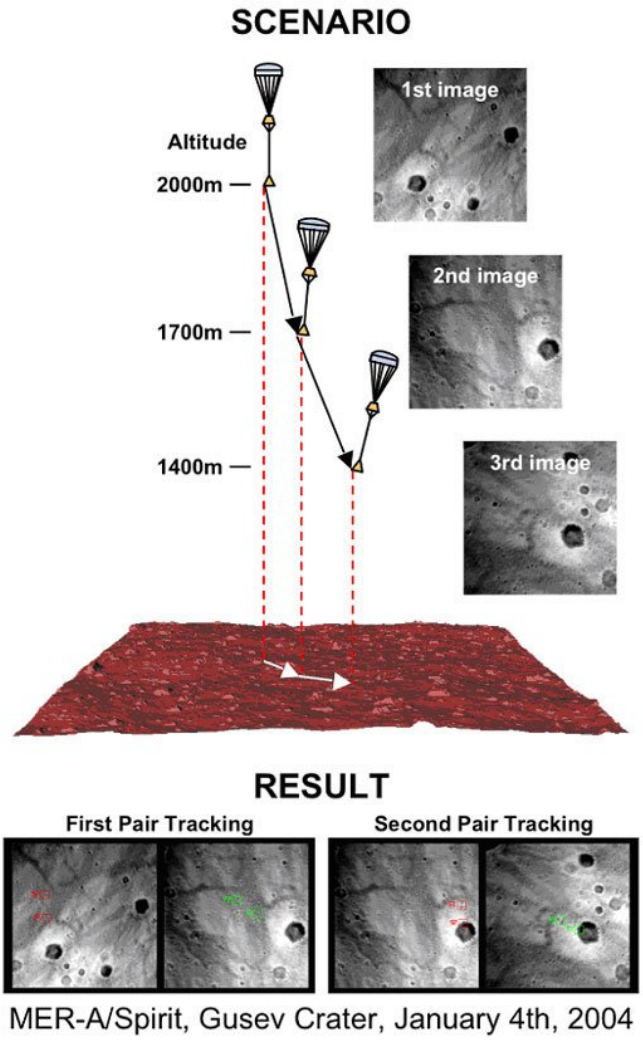

Figure 4. DIMES in action

In early 2004, both MER missions successfully landed the robotic explorers Spirit and Opportunity.

Mars Science Laboratory EDL

At the present, there has been no Mars EDL system that used a real-time guidance algorithm to automatically adjust flight within the Martian environment. The Mars Science

Laboratory (MSL) incorporates a much more sophisticated design than the previous landed missions.

Like Viking, the MLS entry vehicle will fly at an angle-of-attack so that it will have a small lift. Another big change is that MSL will be the first lander to have an autonomous guidance system that can detect and adjust for hazards, allowing a richer set of landing position candidates. The MSL entry vehicle has an offset center of mass designed to allow it to hit atmosphere at a 16 degree angle of attack and have a 19 degree angle of attack at the point of parachute deployment [4]. The angle of attack creates lift and during the initial atmospheric entry, the vehicle will be bank angle adjusted (steered) to drive down entry dispersion errors and reach the desired parachute deployment point. A Reaction Control System (RCS) with retro rockets will adjust the bank angle during this phase.

A disk-gap-band parachute will be deployed when a fixed planet-relative speed of up to Mach 2.5 is determined by integration of sensor data. The parachute is drag only, used for velocity reduction. At deployment, the heat shield will be dropped and an internal balance mass will be jettisoned in order to shift the center of mass to the center of the lander. Then the terminal descent sensor is activated. The terminal descent sensor will scan for velocity and altitude and triggers the powered portion approximately 1100m above the surface at which point the backshell is jettisoned along with the parachute.

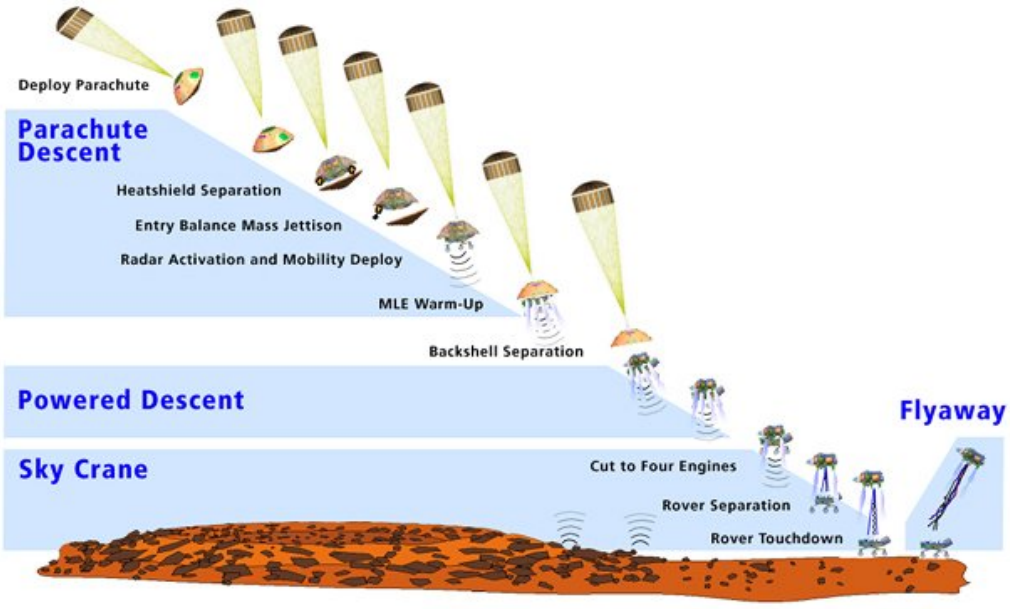

Figure 5. MSL EDL

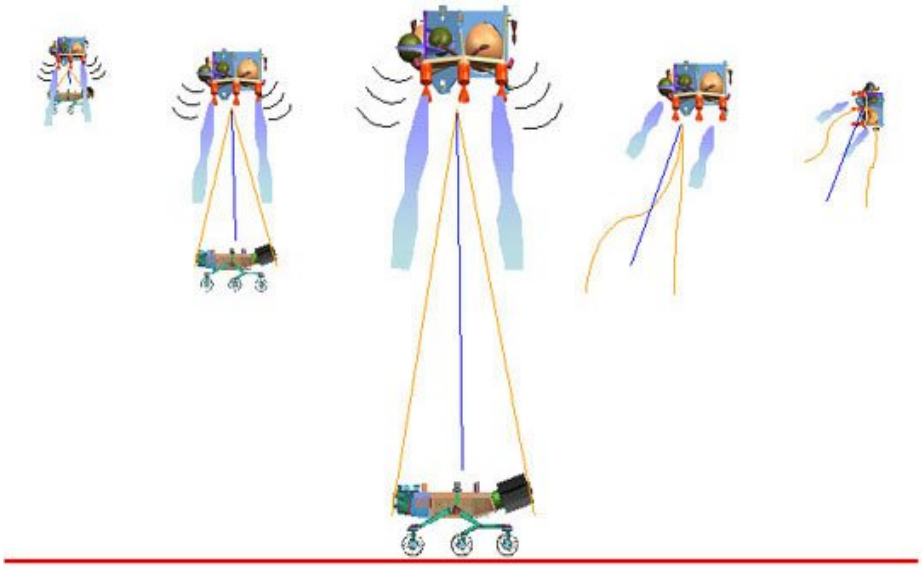

Figure 6. MSL Sky Crane Maneuver

Because of the expected tight altitude window for this mission the engines will throttle up within 2 seconds of parachute jettison. The rockets are throttled to provide gentle vertical descent of 0.75 m/s. During this phase, the lander is lowered via 3 bridles in a procedure known as the Sky Crane maneuver. This two body system allows a closed loop control all the way to touchdown. When the touchdown is sensed, the upper stage cuts loose the lander and fires lateral rockets to avoid coincident impact.

Mars Precision Lander (MPL) EDL

The proposed technology of this research is the Mars Precision Lander (MPL). This will carry on the tradition of using Viking and later derived concepts and adding a few new twists.

The MPL will employ aero-maneuvering to reduce the dispersions at parachute deployment. The aeroshell will have a Viking-like configuration flying at an angle of attack, Pathfinder parachute size, and close to MSL payload mass. At a velocity of approximately 515 m/s and an altitude anywhere from 9 to 14 km the lander will deploy a purely ballistic parachute. At some later time the parachute is turned into a lifting body by snapping opposite sets of tie lines, effectively turning the ballistic parachute into a large parafoil. The parafoil will adjust range to target by bank angle and rate commands. At 1000m above the surface the parafoil is jettisoned and a 3 axis rocket system takes over for the final descent. The rockets will null vertical and horizontal velocities to zero by the time touchdown is reached. The descent follows a neural network based reference trajectory generator trained from simulations as well as realtime sensor updates modifying the static

data and retraining the neural net. The rest of this paper will explain the MPL EDL strategy in more detail.

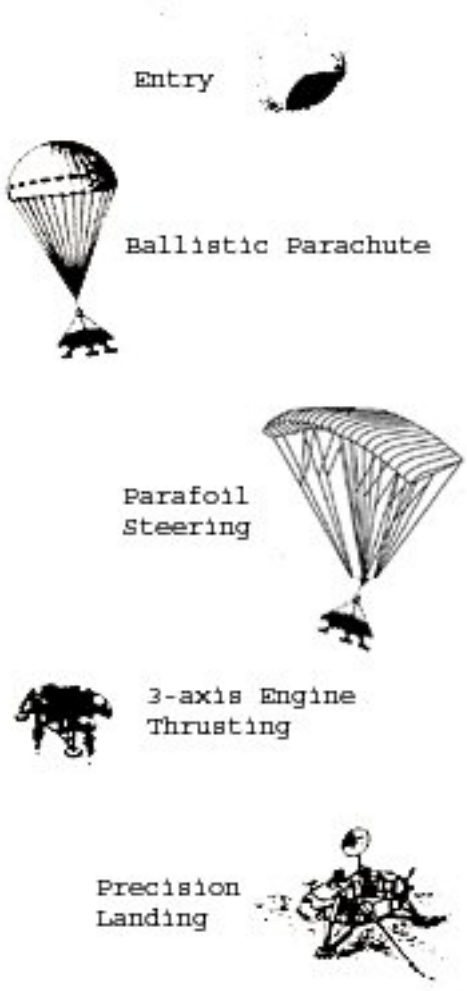

Figure 7. MPL EDL

Summary of Past Missions and Current State of the Art

The following table describes the main differences between several Mars missions, past and proposed. Included are Viking 1, Pathfinder (MPF), MER A, and Mars Science Lab (MSL). MPL is the mission architecture used in the research simulations.

Table 1. Mission Details for Past and Proposed Mars Missions

Landing Year: / **Mission:**	**1976** / **Viking 1**	**1997** / **MPF**	**2004** / **MER A**	**2010** / **MSL plan**	**MPL sim**
Entry Mass kg	992	584	827	2800	2200
Lift to Drag ratio	0.18	0	0	0.22	1*
Parachute Diam. m	16	12.5	14	19.7	13
Drag Coefficient	0.67	0.4	0.4	0.67	0.4
Chute deploy Mach	1.1	1.57	1.77	2	1.77
Chute Dyn. Pres. Pa	350	585	725	750	585
Chute deploy Alt. km	5.79	9.4	7.4	6.5	9.4
Attitude Control	Roll Rate	None	None	Roll Rate	Parafoil Roll
Descent Vel. Cntrl.	Throttle	Sep Cut	Sep Cut	Throttle	Chute & Throttle
Horiz. Vel. Cntrl.	Throttle Pitch	Passive	Lateral SRMs	Throttle Pitch	Throttle 3 axis
End Vert. Vel. m/s	2.4	12.5	8	0.75	< 1
End Horiz. Vel. m/s	< 1	< 20	11.5	< 0.5	< 1
End Mass kg	590	360	539	1541	1500
Major Axis km	280	200	80	20	0.1
Minor Axis (km)	100	100	12	20	0.1

* includes the parafoil effects

TOOLS

Matlab Introduction

The majority of the algorithmic work for this research was done with the Matlab programming language. Matlab is a matrix based computer language developed by the Mathworks company. It is interpreted, meaning there is no separate compile step before run. The user can type commands in a terminal window and have them executed immediately. The syntax is very simple to learn yet the language is very powerful, chiefly due to the large amount of toolboxes, or user contributed code extensions. Matlab is used in many contexts, within algorithm development, data analysis, graphical visualization, simulation, engineering and scientific computation, and application development.

The basic data element in Matlab is an array that does not require dimensioning. This allows vectorized code to be developed as well as being intrinsically suited to matrix operations. Unsurprisingly, Matlab stands for Matrix Laboratory and was originally written to provide easier user access to the powerful LINPACK and EISPACK mathematical libraries. The modern versions of Matlab use the state of the art LAPACK and BLAS libraries.

Matlab is used extensively in both universities and industry allowing for a nice iterative improvement cycle where countless users have added to Matlab's functionality. The toolbox developed for the optimization tasks in this research is one example. The Particle Swarm Optimization toolbox has had over 10,000 unique user downloads since

being uploaded to the Mathworks website and is the most downloaded user contributed optimization code [1].

Program to Optimize Simulated Trajectories (POST)

The state generator environment for this research was the Program to Optimize Simulated Trajectories (POST) [27], PC and Unix versions. POST, as used here, is a three degree of freedom, rigid body, point mass simulation program. The program was originally developed to help design the ascent and descent mission designs of the early Space Shuttle project and has been added to and improved upon continuously since. It is used to simulate atmospheric flight mechanics and orbital transfer problems, has an extensive set of weather models, allows powered or unpowered flight simulation, and has reasonably accurate planetary science built-ins. POST was written in Fortran 77 and later modules continue to be written in C. POST II exists with more capability but was not used in this research. Mostly the PC version (basically a beta of POST II) was used. To set up a simulation, POST uses an input deck, a Fortran-like namelist input procedure to define the problem. To help, POST has an integrated set of Flight Control System (FCS) modules as well as discrete parameter targeting and optimization to both equality and inequality constraints.

The simulation is built around separating the trajectory into a sequence of simulation segments or events. By treating the problem this way, each sub problem can be treated in the best way for that particular sub task. There are five basic functional elements that can be utilized in the simulation. These are the planet module, the vehicle module, the trajectory simulation module, the trajectory auxiliary calculations module, and the targeting/optimization module.

The planet module defines the environment in which our vehicle will operate. The module consists of an oblate spheroid model, a gravitational model, an atmosphere model, and a winds model.

The vehicle module has typical vehicle properties such as mass properties, propulsion, aerodynamics & heating, airframe characteristics, navigation and guidance models, and a flight control system model.

The trajectory simulation module controls the program cycling by parsing the event sequences. It has table interpolation functions and standard integration techniques which are used to sole the translational and rotational equations of motion.

The trajectory auxiliary calculations module takes care of computed output values, such as conic parameters, ranging, tracking data, and many more.

The targeting/optimization module allows the user to select an optimization variable, dependent variables, and independent variables to solve optimization problems. There are a few optimization methods built in, two gradient descent based and one derivative based, all requiring good first guesses for the chosen variables [20].

For this research, POST was used as the state generator environment while Matlab was used as a control shell.

MarsGRAM

MarsGRAM [19] is a Mars weather environment that can be used as a plug in for POST or as a standalone tool to generate tables of wind and atmosphere profiles. MarsGRAM stands for The Mars Global Reference Atmospheric Model. It is based on

surface and atmospheric temperature data collected during the Viking and Mariner missions. It also has pressure data collected from the Viking lander during EDL. It includes mean density as well as mountain-perturbed for any location including altitude and for any seasonal time. It includes code to simulate local scale and global scale dust storms and density perturbations. Other atmospheric components include temperature, pressure, and wind profiles. At the beginning of this research, a version of MarsGRAM was used that had incorrect wind information. It was correct from the Viking data but when the more detailed Pathfinder data became available it was incorporated in a later release. For the earlier studies, tables of wind modifiers were used to make them agree with the latest data. Further improvements included features such as dust devils and random gusting.

MISSION DEVELOPMENT

Scenario Study

The scenario study part of the research was done early on to decide what kind of EDL mission profile would be best to examine from a Computational Intelligence standpoint. Several ideas were postulated and tested within a stock POST Mars environment state space. Minimal optimization was done with the POST included routines.

For future planetary exploration missions, either robotic or manned, it is desirable to precisely target a lander's touchdown point. Perhaps there has been a previous robotic mission that requires a follow-up robotic mission in order to retrieve collected samples and return them to earth. Perhaps there are specific geographical features that require close-up study. Regardless, a need for precision landing within 100 meters of a specified geographic location exists. Current state of the art can only achieve positioning to within approximately 10 kilometers [4]. This study examines some proposed mission profiles for controlling a lander as it touches down on a specific point on the Mars landscape. All model parameters and constants are taken from and designed to be compatible with the M2001 specifications that were available at the time.

Four separate Mars Lander touchdown scenarios are considered (one of them is a baseline) with the goal of minimizing the landed distance to a specified location on the Mars surface. This study considers a set of points from parachute handoff to touchdown on the surface. The initial conditions were provided by Langley Research Center as part of their Mars 2001 Monte Carlo atmospheric entry dispersion study [3].

Baseline

The baseline scenario is the simplest. It takes a subset of initial conditions provided from a Monte Carlo study for the Mars 2001 mission [3] and propagates them through the state integrator until touchdown on the surface. This takes the initial conditions, and pops open the parachute to slow down. There is no thrust applied while the parachute is deployed. Once the parachute is jettisoned, the control system kicks in and performs a gravity turn to touchdown. The lander's touch down latitude and longitude is not targeted but ending velocity and altitude conditions are targeted with the built in optimization of POST. This provides the landed footprint ellipse that all other scenarios will try to minimize.

These desired end conditions are:

$$0.1 <= ur <= 2.1 \text{ m/s}$$

$$0.1 <= vr <= 2.1 \text{ m/s}$$

$$wr = 2.0 \text{ m/s}$$

$$2499 <= gdalt <= 2500\text{m}$$

The relative ground velocity components are ur (N/S), vr (E/W), and wr (up/down). Gdalt is the geodetic altitude. 2500 meters is the surface level above the mean oblate planet spheroid at the mean landed latitude and longitude. As has been previously mentioned, the POST input deck process treats each problem as a series of events.

For the baseline case, Event 1 is the initial setup and parachute deploy. The atmosphere is input as a table lookup. Marsgram winds are input as tables. The initial

position and velocity components are input in an inertial coordinate frame from Mars 2001 Monte Carlo analysis. The gravity model is an oblate planet using spherical harmonics from j2 through j6. Guidance is strictly used for gravity turn (velocity null) and uses atmospheric relative aerodynamic angles. There is one engine, pointed out the X body axis.

Event 2 sets the parachute deployment. The parachute is limited to 13 meters diameter and the weight is adjusted for dropping the heatshield.

Event 3 triggers when the parachute is fully deployed. At this point the lander's surface area is increased to represent the full area of the parachute at 132.73 m^2. The drag of the lander is overshadowed by the drag of the parachute so it is ignored. This event simulates the effect of a ballistic parachute.

Event 4 triggers when the lander is 1000m above the surface (+3500m MOLA). At this time the parachute is jettisoned, the vehicle parameters are returned to reflect lander characteristics and the engines are turned on. The internal POST targeting adjusts the thrust and vehicle angle to meet the constraints using relative aerodynamic angles.

Event 5 triggers when the lander is 500m above the surface (+3000m MOLA). This is another opportunity for the POST targeting functions to adjust the controls.

Event 6 is the last task and occurs at surface touchdown. The engine is turned off and the problem is ended.

The controls used for the gravity turn are engine thrust. POST adjusts the controls in order to meet the final velocity and altitude conditions. For this scenario the control system was given two opportunities to adjust the engine thrust and angle, in events 4 and 5.

The following figure shows the initial parachute handoff conditions as blue circles and the final touchdown points as green x's. Both initial and final condition extrema map out to an approximate 20km x 10km ellipse.

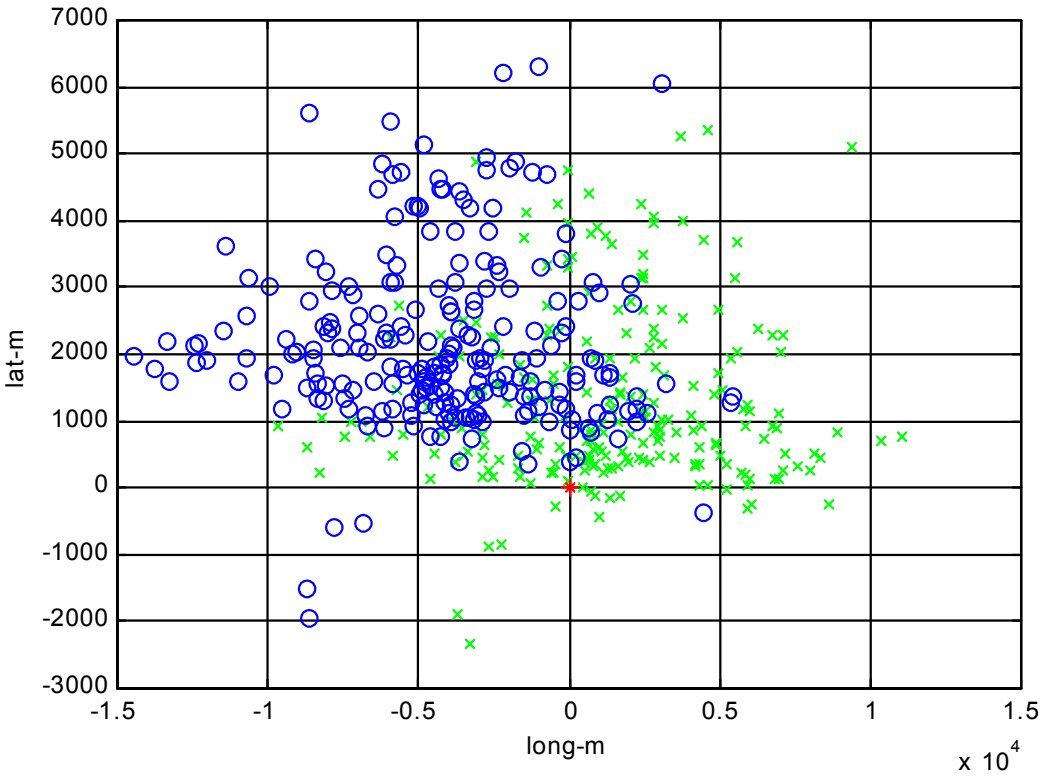

Figure 8. Handoff condition subset with Baseline landed ellipse

The amount of propellant used was relatively constant as there was no real maneuvering performed, and averaged out to about 210 Newtons. Propellant usage is a large problem because the more propellant that is needed then the less of a science package can be delivered to the touchdown site. As can be seen from the below graph, the fuel usage was similar for all initial conditions. There are a few anomalies using very little fuel or in one case, negative amounts, these are cases where POST was unable to target to

desired conditions and returned garbage. There will be more on the reasons for that in the section about First Guesses.

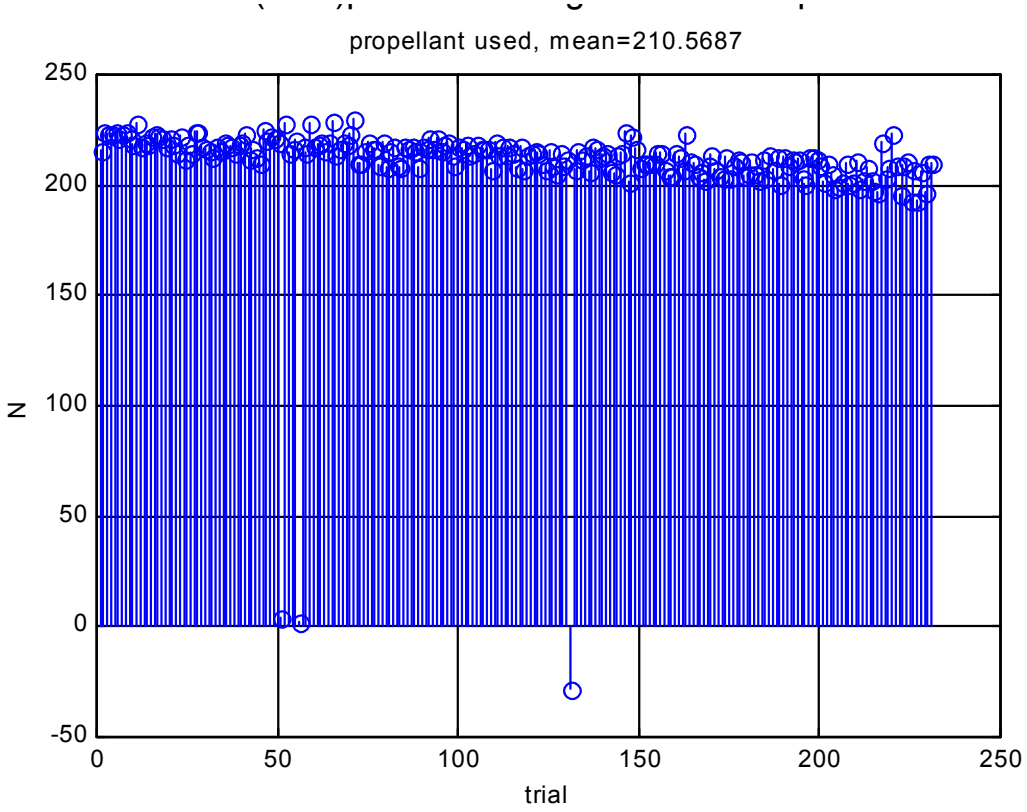

Figure 9. Baseline Scenario Propellant Used

Thrust on a Ballistic Parachute

The first candidate scenario examines the effect of thrust vectoring while the parachute is deployed and includes an algorithm for determining targeting initial guesses which will be discussed in another section.

This scenario builds on the baseline by adding thrust vectoring during the ballistic parachute event. The engines are turned on and used to minimize range to the desired landing point. At an altitude of +1000m above the surface, a gravity turn is performed

exactly as in the baseline scenario. Targeting is used as in the baseline case but optimization is added on. The range to target is minimized by using the same controls as before but used in slightly different times. There is another event added right after the parachute has fully deployed where the engines turn on and get a thrust command. So, for this case there are three opportunities for optimization of the range. Initial guesses here are starting to get a bit trickier, so fewer initial conditions were able to target. This is a trend that will continue all the way through this section of the research and the solution will be discussed later.

This is now a 15 dimension optimization problem (for each initial condition) where there are 3 events that can receive control commands. The controls are thrust, pitch, pitch rate, yaw, and yaw rate. Or in POST terms (variables) etapc1, pitpc1, pitpc2, yawpc1, and yawpc2. Using the gradient descent POST based targeting and optimization routines 216 out of 235 initial conditions managed to converge to solutions.

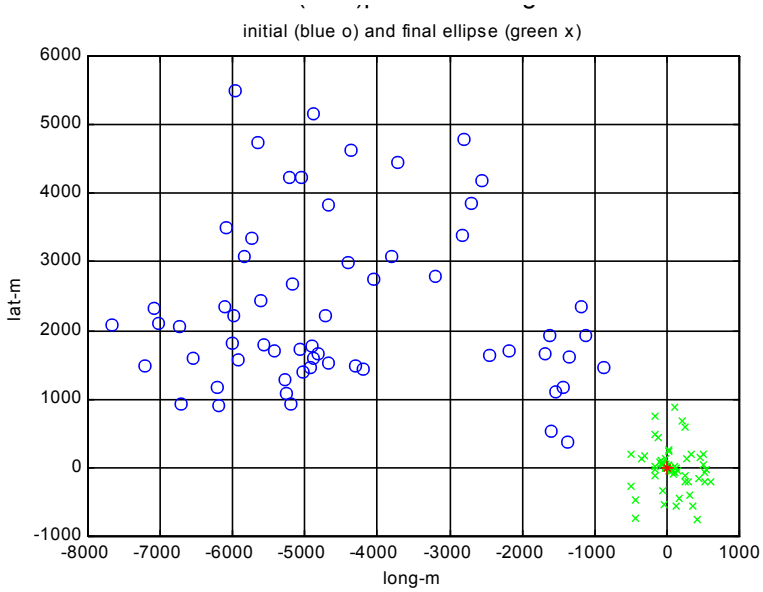

Figure 10. Initial and Final ellipses for the Thrust on the Parachute case

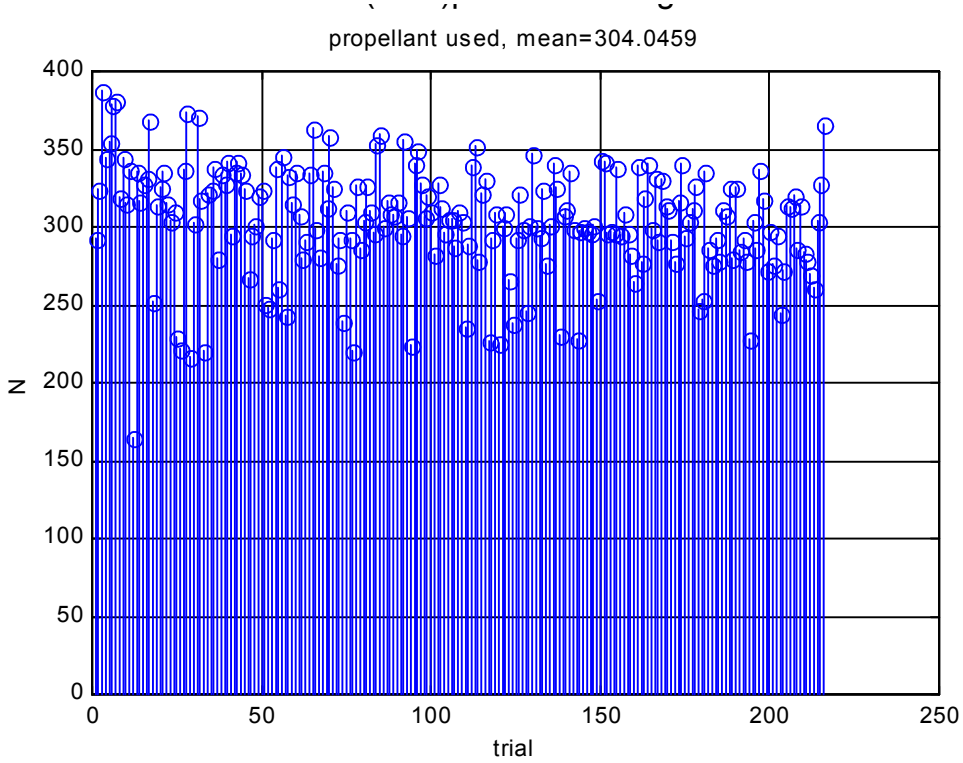

Figure 11. Thrust on the Parachute Fuel Consumption

Hover and Thrust Laterally

The second candidate scenario considers a gravity turn to a hover condition 500 meters above the surface and then uses lateral thrusting to minimize the range to target. This scenario was split into two separate POST input decks to help the targeting and optimization algorithms converge. The first input deck simply did a gravity turn from the handoff condition to 500m above the surface of the landing. The targeting and optimization was set to bring all velocity components to zero and to minimize range to the target. There is no thrusting on the parachute. This first input deck is very similar to the baseline case.

The second part of the scenario handles from the Hover condition (end of the first deck) down to the surface, taking care of the terminal descent phase. The terminal states of the previous case are used as initial conditions for the next input deck. This scenario is different from the 'thrust on parachute' scenario in that it targets directly to the reference latitude and longitude rather than just trying to minimize the landed distance. There is no first guess algorithm for the optimization. The targeting first guesses assume the reference point is somewhat south of the initial conditions (which indeed it is for most of the points tested). The targeting also starts immediately.

Convergence became an issue with this study. As in the other cases, 235 initial conditions were taken from the full 2000 Monte Carlo point data set, in a random uniform sampling. For the first part of the EDL, the gravity turn, convergence occurred 234 times. For the second part, from hover to touchdown, only 180 out of those remaining 234 converged. The POST included optimization had a very difficult time with this scenario.

Table 2. Convergence Success for Hover and Thrust Laterally

Part 1: gravity turn to a hover condition	99% converged
Part 2: horizontal thrust & hover to target	76% converged

It should be noted that these cases did not use an algorithm for choosing first guesses for the targeting algorithm in POST. After this portion of the research it was postulated that a first guess algorithm would better targeting results.

Here is a graph showing the initial handoff ellipse, then the initial hover point ellipse, and last the final ellipse.

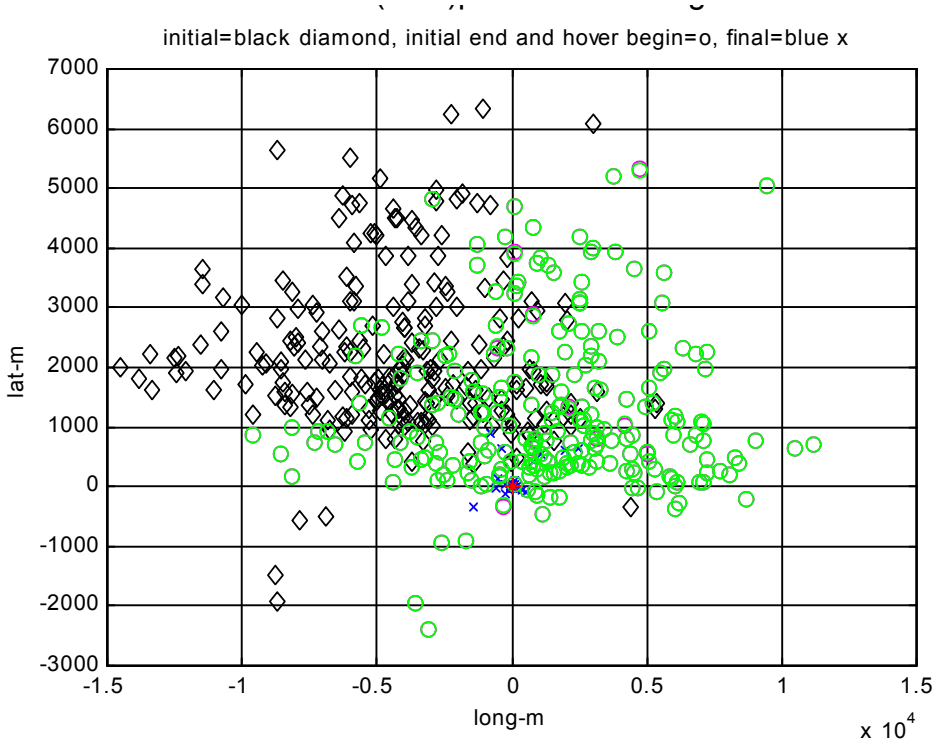

Figure 12. Handoff to Hover Ellipse

The final ellipse is so small (centered at 0 reference) that it is hard to see. Most of the points targeted to within 100 meters. Also, the initial parachute handoff ellipse and the initial hover point ellipse are similar in magnitude with an offset of position. This is expected.

Bringing the lander to a hover 500m above the surface before maneuvering to the reference point is fairly easily accomplished. The amount of horizontal maneuvering impacts fuel used. It seems we can target to pinpoint precision as long as we have enough fuel. This is unrealistic as we do not have unlimited fuel resourced. In addition, more fuel

implies less of a science package so a scenario needs to be found that allows large payloads without needing large amounts of fuel.

The graphs below show the focus on the landed ellipse followed by the fuel usage.

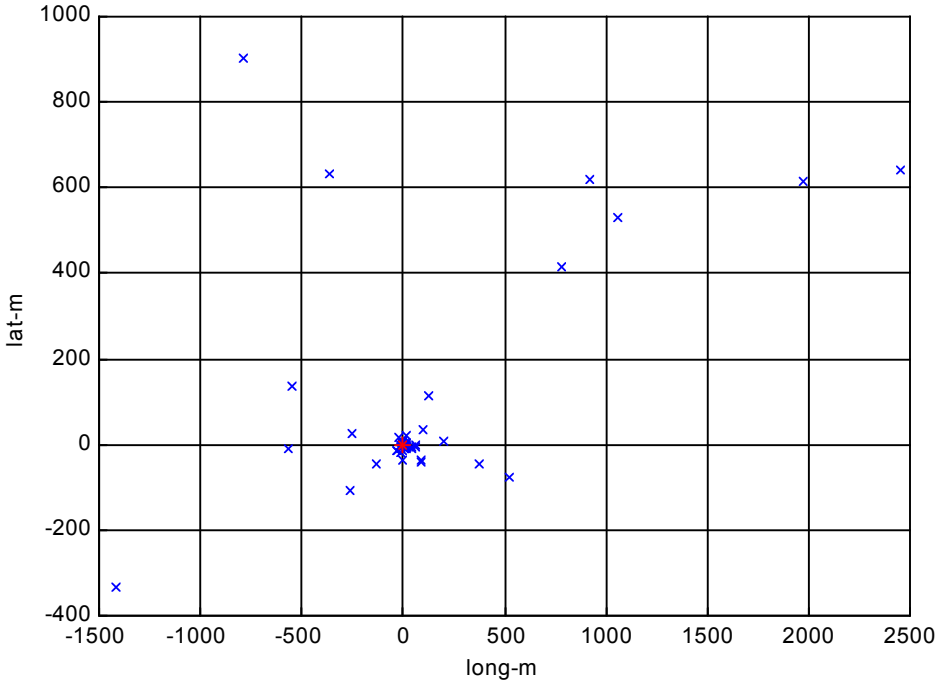

Figure 13. Hover and Thrust Laterally, Landed Ellipse

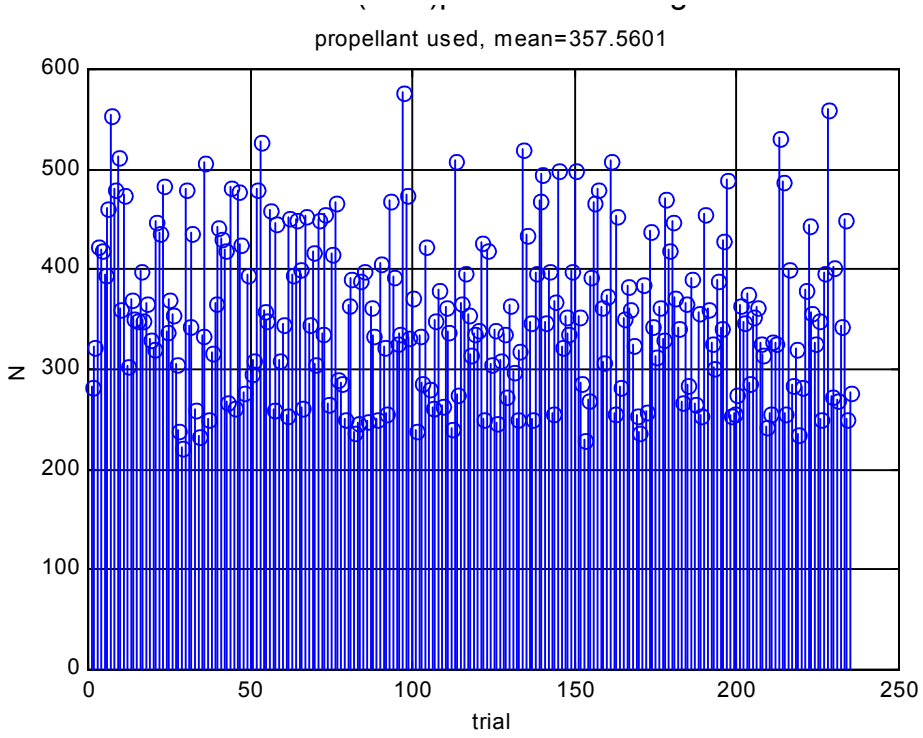

Figure 14. Hover and Thrust Laterally Propellant Usage

Guided, Lifting Parachute (MPL)

The third candidate scenario is the guided, lifting parachute mission (parafoil). This is the scenario that was eventually chosen and has precedent in the X-38 crew escape vehicle project [23]. The POST deck that is described here is slightly different than the one used in the later research. However, for simplicity, they are both referred to as the Mars Precision Lander (MPL) scenario. The main features of the MPL scenario include ballistic entry, ballistic parachute which transforms into a lifting body, then thrusters during the last 1000m meters before touchdown. The chief difference between the MPL incarnations is in the amount of controls and different parachute lift scenarios.

For the initial Scenario Study, the input deck used a single commanded bank angle turn during the lift phase. Then at some optimized time later, the bank angle was nulled for

straight flight and the lander glided straight into the target. The terminal descent phase was the same gravity turn as the other scenarios. This approach was able to optimize range to target in similar manner to the hover and thrust scenario but has the advantage of using much less fuel. Again similar to the hover and thrust scenario, the MPL scenario required good first guesses for the optimization algorithm to converge to a solution. Therefore the number of initial conditions that targeted was less than the full set.

The graphs below show the landed ellipse as well as fuel used. The set shown is the set that successfully targeted to the end velocity conditions. A single bank angle turn followed by a glide does not seem to be enough to achieve a full precision landing, however the strategy shows promise. The landed footprint was reduced from +/-15km to +/-1km. It is interesting to note that the final conditions line up more or less along the mean initial velocity vector.

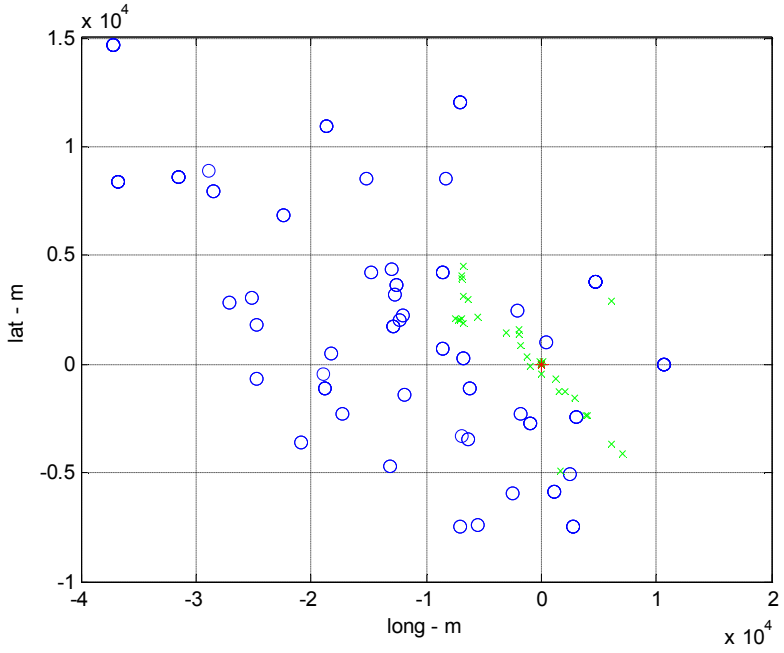

Figure 15. MPL Landed Ellipse

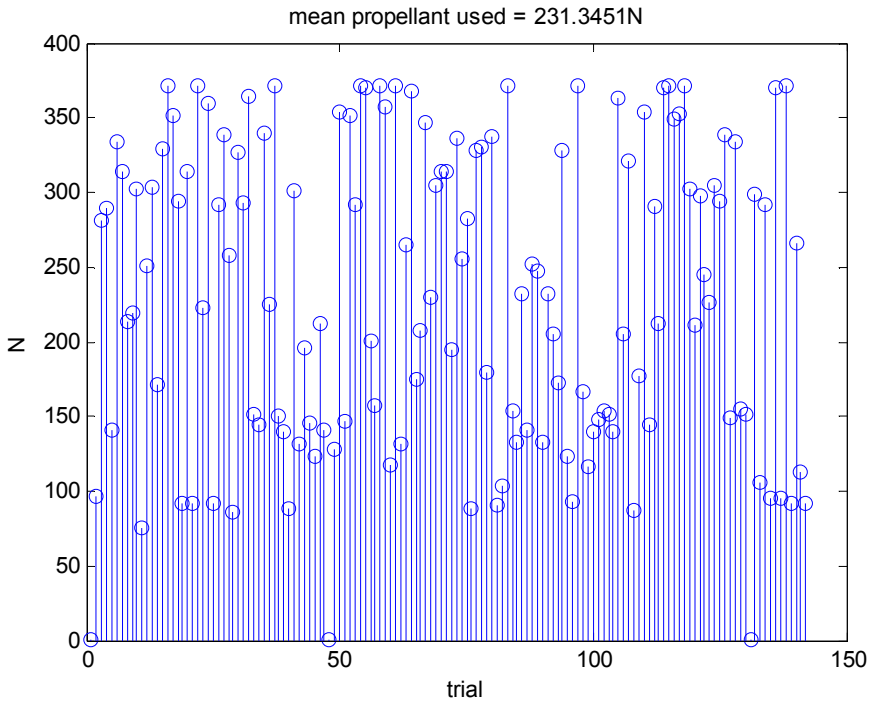

Figure 16. MPL Propellant Usage

Summary of Scenario Study

Four scenarios were studied, a baseline case followed by thrust on a ballistic parachute, hover and thrust laterally, and the guided lifting parachute with one turn command. Despite optimization convergence problems on the last two scenarios, they both showed promise as candidate precision landing mission architectures.

The lifting parachute wins as the scenario to further investigate as it allows potentially more maneuverability with significantly less fuel cost.

Environment

There are several Mars environment models that were developed as part of this research. The environment is a tricky area for the MPL. The goal is to design a system capable of handling wide dispersions in various atmospheric quantities. However, for a real-world representative simulation, a set of accurate environment models are needed. The initial Mars model that was included with POST was a simple low fidelity table lookup with inaccurate wind data. In an effort at improvement, several more detailed models were examined.

Standalone Models

The standalone models are an attempt to develop higher fidelity Mars environmental models for the POST MPL simulation. Atmosphere, gravity, and a Topography Sensor model (height field) were developed. The standalone models were also used in a Boeing sponsored Mars Precision Landing subcontract as part of the (proprietary) control system validation scheme [15]. The standalone models, as the name implies, can be used individually to investigate Mars environment properties, or can be linked together for a full engineering style simulation.

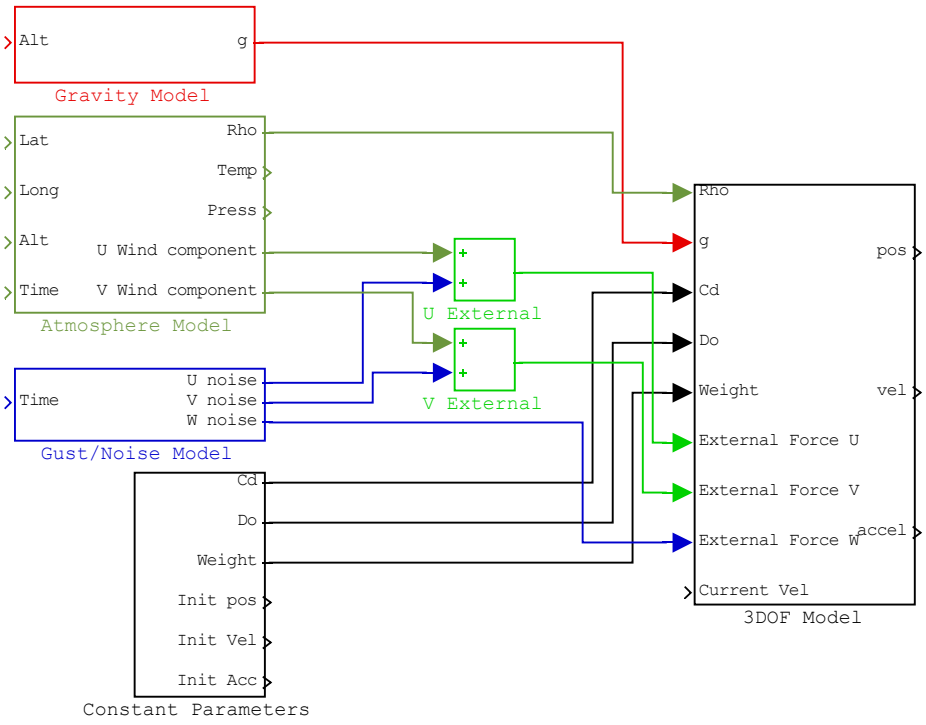

Figure 17. Standalone Mars Environment Model Integration

Atmosphere

The atmosphere model was initially based off of the simple height based model from Glenn Research Center [12]. It is not particularly accurate but it is very simple to code and runs very fast in an engineering simulation.

In addition to the Glenn Research Center model, calls to MarsGRAM can be made. This is the most accurate global atmosphere model available, when using the latest version of MarsGRAM. However, it is slow functionalized this way due to the need to write to a file. For MarsGRAM, it was found it is best to use it from within POST instead of standalone mode.

A high fidelity model covering Helles Basin, Valles Marinares, and the Mars North Pole was developed with data provided by Oregon State University [31]. The OSU data is several gigabytes worth and had never been formulated for use in a real-time simulation before. When they run their models, it takes several days on a large parallel computing cluster to get the results. As of necessity, that data has been pared down considerably for this simulation but the subset still provides the highest fidelity model over the covered local regions.

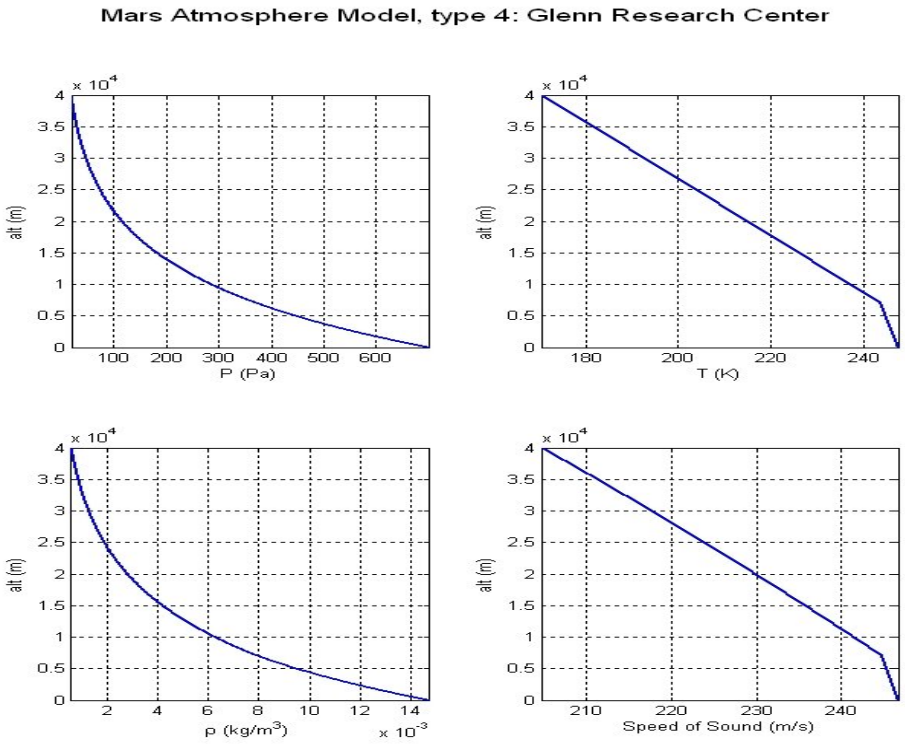

Figure 18. Mars Atmosphere Model, Glenn Research Center type

The atmosphere model is functionalized with inputs of longitude, latitude, altitude, and model type. Optional inputs are elapsed time and initial time. The outputs are pressure

in pascals, temperature in Kelvin, atmospheric density in kg/m^3, and speed of sound in meters per second.

There is also a simplified call that will use the appropriate Mars model based on the state, OSU for the three localized regions and either Glenn or MarsGRAM for the non-localized data, depending on user input.

Winds & Gusting

Similar to the atmosphere model, a winds model was created. The model was developed from the OSU data [31] and is strictly valid only for those localized regions. It is a dynamic wind model, covering the changing winds in hour increments for a period of 20 days. For static wind profiles, the latest version of MarsGRAM can also be used. Originally wind table modifiers were used in conjunction with MarsGRAM to correct for model inaccuracies but that has been superseded by newer releases. The dynamic wind profiles, though only valid for North Pole, Helles Basin, and Valles Marinares regions, are used regardless of the latitude and longitude of the vehicle during the simulation. This is because these are the only dynamic wind models in existence and for validation of an engineering model, the interest is in the changing wind profile envelope and how it affects control system performance, as opposed to completely sub-meter accurate location weather dynamics.

The inputs to the model are longitude, latitude, altitude, elapsed time, and model type (including dynamic winds from Antarctica for engineering validation). The outputs are Northerly and Easterly wind components in meters per second.

In addition to the wind models, there is a simple gusting model that is set as a perturbation of the winds. It is user configurable to specify a gust profile in time and/or altitude or to have random gusts of a maximum strength. The below figures show the dynamic wind profile in 3 hour increments over a single 24 hour period.

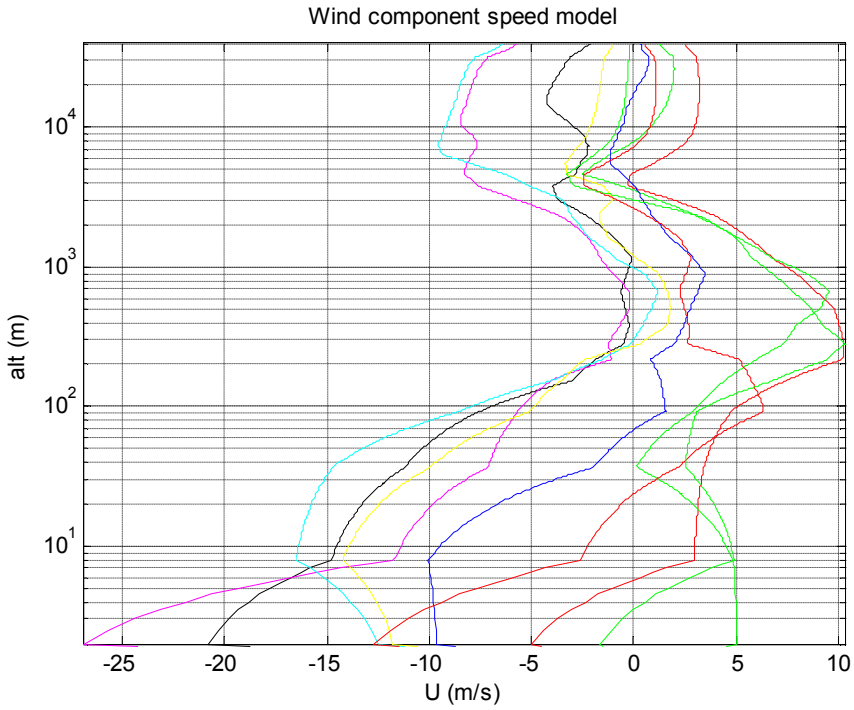

Figure 19. Mars Northerly Winds as a function of Altitude & Time

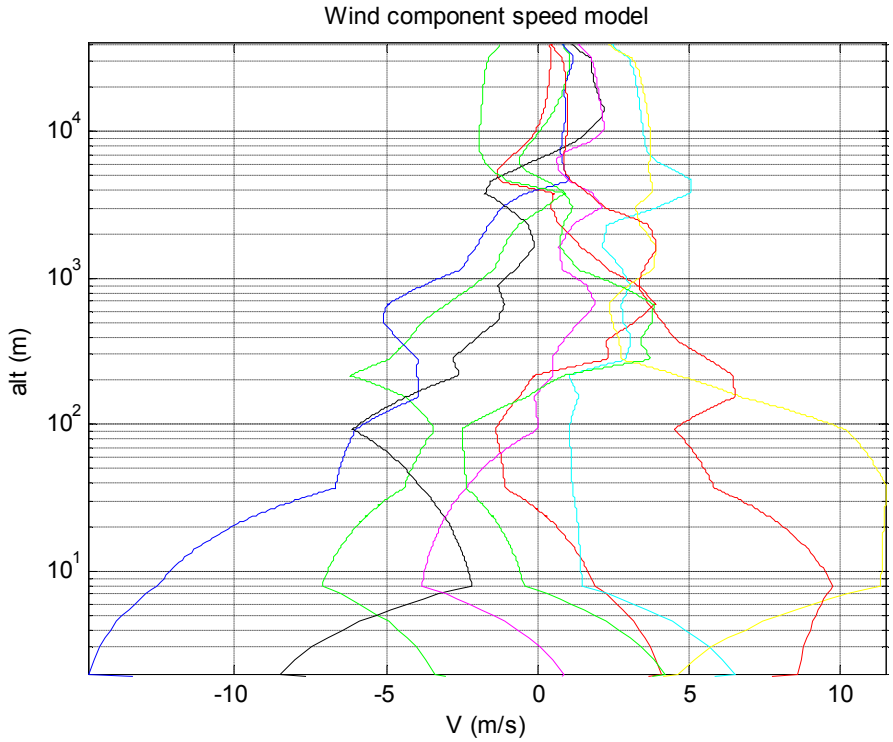

Figure 20. Mars Easterly Winds as a function of Altitude & Time

Because of the timescales during the EDL phase of MPL, the winds can be taken as static to a first approximation. For speed the MarsGRAM POST static wind model was used for the initial runs and the dynamic winds used for higher fidelity simulation.

Gravity

The gravity model returns the acceleration due to gravity for mars (or earth or any custom planet). The model is user configurable to treat the planet either as a spheroid or oblate. The mars model has spherical harmonics up through the 8th term and provides very good fidelity [27]. Gravity waves and topographical effects were not included in the model

as they would add considerable complexity, computation time, and not be very useful for the purposes of the terminal descent phase.

The inputs to the function are longitude, latitude (only meaningful for an oblate planet), altitude, and model type whether oblate or spheroid mars or earth or custom. If it is a custom planet then radius of the equator, radius at pole, mass factor of planet in earth multiples, and 2^{nd} up to 8^{th} gravitational harmonics. The output is the gravitational acceleration in the 'down' direction in meters per second squared. Optional outputs are the gravitational acceleration in inertial coordinate components, the gravitational potential, and gravitational acceleration in spherical coordinate components (deg/s^2).

Topography Sensor

A topographic sensor with zero noise was modeled as a function to return the surface height at any location. The data used comes from the MOLA instrument and represents the best fidelity at the time, since superseded by later measurements. The function uses a table lookup with resolution in half degree increments. The function has been optimized for speed and only loads the dataset once. The graph below shows how surface height changes as a function of location and time for three representative MPL trajectories from parachute handoff to touchdown. The point being that they are all different and this is important when trying to determine when touchdown should occur.

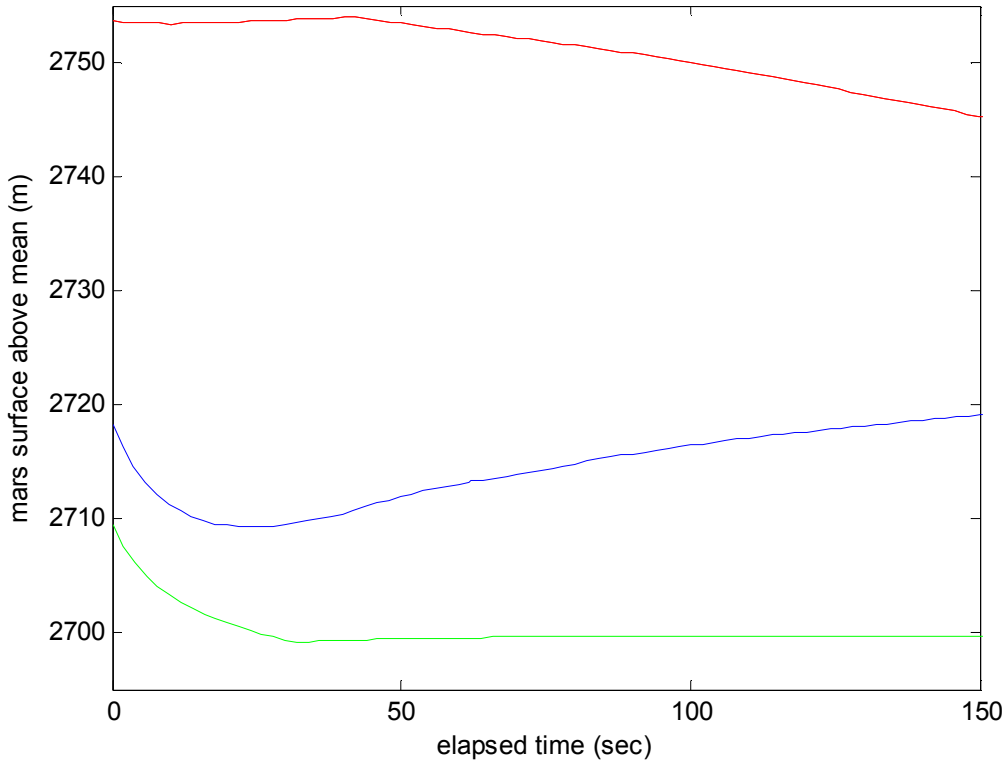

Figure 21. Deviation from mean geoid surface height along 3 trajectories

POST Models

The POST included models cover all the same ground as the standalone models with the exception of the Topography Sensor and dynamic winds. Though lower fidelity in most cases, the POST models offer the advantage of being fully integrated with the rest of the state generation and run relatively quickly. For validation runs the POST models were mainly used. For more in depth simulation, the standalone atmosphere, dynamic wind, and topography sensor models were integrated into the architecture.

MPL POST Input Deck

This section describes the architecture of the Mars Precision Lander (MPL) POST input deck, as it differs slightly from what was used in the Scenario Study, chiefly in the number of controls. Briefly, the scenario consists of ballistic entry, followed by ballistic parachute to reduce velocity. Then the parachute is converted to a parafoil by severing some tie lines and the vehicle is bank angle steered toward the target. There are 3 steering opportunities. Then the retro rockets are turned on, parachute is jettisoned, and the MPL is controlled for terminal descent with 3 vectored thrusting events.

Event 1, sets initial conditions. Wind table modifiers are input (for the early versions of MarsGRAM). Initial conditions are passed in from a Matlab shell that iterates through a list. The vehicle weight at parachute deploy is 585.479 kg. Propellant weight starts off at 100kg.

Event 2, deploys the ballistic parachute. The parachute dynamics are set such that it is fully deployed to a 13m diameter in about 3 seconds. The parachute drag coefficient is 0.41.

Event 3, converts the parachute to a lifting body at some elapsed time. The lift to drag (L/D) ratio was experimented with values from 0.5 to 2.0. It was found that a low L/D value prevented the lander from reaching the target due to a lack of glide time. The high side of L/D values caused the parafoil to stall out and crash the lander in many cases. After this study the L/D value was chosen to equal 1.0 as a good tradeoff between glide time and stall. The following figure shows an example of a steerable parachute subsystem test. By

increasing or decreasing tension on either of the 2 sets of tie lines, the system is able to command bank angle turns. The photo is from the X-38 project [23].

Figure 22. Autonomously Steered Parafoil drop test

Events 4-6, glide on the parafoil using bank angle commands to minimize distance to target. The timing of these events is also an optimization variable.

Event 8, turns on engine when lander is 1000m above MOLA derived surface.

Events 9-11, use vectorized thrusting to minimize both range to target and touchdown velocity. Timing of each of the three events is also an optimization variable.

Event 12, touchdown and end program.

Events 3-6, and 9-11 are when the control system is active and must be optimized. The controls are 7 time, 6 bank angle, 3 bank rate, 3 angle of attack, 3 sideslip, and 3 thrust.

This is a 25 dimension problem, there are 25 unknowns that must be adjusted in order to minimize the range to target as well as make sure the lander touches down with as close to zero velocity as possible.

TRAJECTORY OPTIMIZATION

The idea behind trajectory optimization is to find the set of controls that will optimize the range to our target as well as meet the terminal conditions of minimal velocity. The terminal condition has to be met within tolerance; otherwise the MPL will be damaged by impact. Also, the range minimization must be successful; otherwise the goal of a Precision Lander will not have been met. Two conditions of the MPL architecture make this extraordinarily difficult for POST. First, the optimization problem as presented is highly nonlinear and can be discontinuous over some regions. Secondly, the problem for most of the scenarios under consideration was of very high dimension. The POST included algorithms are 2 gradient descent based methods and the NPSOL algorithm [14]. Though they can work in nonlinear environments, they do not handle high dimensionality and discontinuities very well. Nonetheless, an attempt to use POST optimization was made.

The basic targeting and optimization technique used in POST is the Projected Gradient Algorithm (PGA). It is an iterative algorithm designed to solve general nonlinear programming problems and is an attempt to treat multi-dimensional problems as a series of single dimension problems. At the beginning of the optimization, the algorithm primarily seeks to adhere to problem specific constraints and at the end of the optimization the emphasis has switched to a cost function reduction approach.

The first attempt at trajectory optimization included letting POST do the work during the scenario studies. Physically reasonable but otherwise random first guesses at the solution were presented to POST for each initial condition and POST had the job of taking

those guesses and refining them into a valid converged solution. Except for the most simple of cases, that of finding the elapsed time to deploy a ballistic parachute for the baseline case, a single variable, this proved too difficult for POST optimization to converge to a solution. The NPSOL algorithm was not attempted because it requires a smooth function with derivatives that exist to the 2^{nd} order [14]. This is not guaranteed with the cost function developed later.

The next attempt was to come up with some First Guess heuristics to help the POST optimization algorithm converge. Better first guesses lead to more converged solutions. This was successful for the Thrust on a Parachute scenario.

First Guesses with POST

As previously noted, the optimization routines that are included with POST require reasonably good first guesses to converge to a solution. Without good guesses, with 'good' being highly dependent on the problem's error topology, convergence will not happen. The algorithm will search in non-optimal regions or get stuck in local neighborhood extrema.

To get first guesses for ground relative yaw and pitch (yawr and pitr POST variables) used while thrusting on parachute we look at several cases. First we determine the initial conditions of the lander at parachute hand-off, then compare to the targeted reference point on the ground. In general, the initial conditions indicate that the lander is moving easterly and will overshoot the target. The initial conditions examined consisted of 2000 points supplied by LaRC from a Monte Carlo analysis studying the position ellipse at parachute handoff.

To determine the various conditions, the problem was split into two sub-problems. First it was considered how to correctly yaw the craft to achieve targeting and second it was considered how to correctly pitch the craft for same. After examining the geometry for both sub-problems, some heuristics were developed. The following graph shows the geometry from which the first guesses were developed.

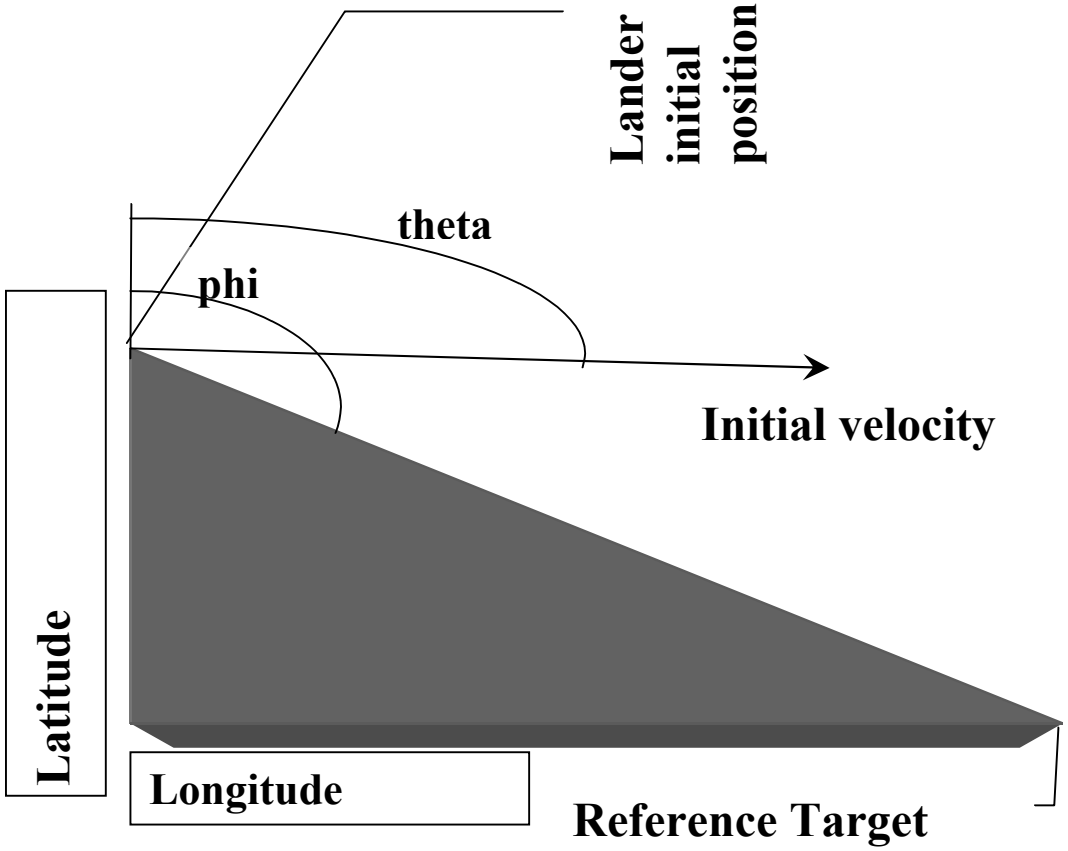

Figure 23. First Guess Geometry

Results of First Guess Study

Despite having some promise on the Thrust on a Parachute case, developing a set of heuristics via inspection for the other scenarios was abandoned for further study. The procedure is highly problem dependent and does not approach the level of convergence required for a robust trajectory generation algorithm. The heuristic approach could be improved as well but being problem dependent it is limited to a very specific class of problems where it would be useful. In addition, the other mission scenarios are significantly more complex with multiple steering and thrusting events.

There is another way to optimize that is portable to many classes of problems, simple to understand, requires no heuristics, and is very resistant to the problems that plague gradient descent methods, i.e. discontinuity and dimensionality. This is the Particle Swarm Optimization method (PSO).

PSO Primer

PSO is a method used to optimize n-dimensional problems, requires no rigid first guess algorithms, explores the majority of problem space, has little problem with being stuck in local minima, and is both uncomplicated to code and uncomplicated to understand in its most basic form.

PSO is a stochastic population based optimization strategy with a simple memory component. It is considered a subset of Evolutionary Computation, of which Genetic Algorithms (GA) also is a member. In fact PSO has similar or better results than GA [17, 20].

The mechanics of PSO start off by seeding the problem space with candidate solutions randomly positioned across the search range, with an initial random velocity. Each solution is expressed as a position within the dimension space of the problem. For the 25D MPL EDL problem, that means that each candidate solution has 25 values, such as time, bank angle, thrust level, etc.

Each candidate solution hereafter referred to as particle, will have a single rating value or cost associated with it. If the candidate solution does not meet the objectives very well, the cost will be high. If it meets the objectives, the cost will be low. The rating value is expressed as the output of a cost function whose inputs will be each component of each particle.

Each particle will have its cost calculated. Then each particle will update its memory with the location of the global best particle as well as its own personal best valued position. Then based on these values, the particles have their positions and velocities updated.

The velocity equation is the heart of the PSO algorithm and expresses each particle's velocity as a balance between attraction to its own personal best position and the current global best position among all particles. This is the difference between local and global searching and is one of the reasons the algorithm is so resistant to getting stuck in local minima [10].

$$vel_i(k+1) = inert(k)*vel_i(k) + 2*rand() *(pbestx_i - currentx_i(k)) + 2*rand() *(gbestx - currentx_i(k))$$

$$pos_i(k+1) = pos_i(k) + vel_i(k+1)$$

Where $vel_i(k+1)$ is particle i's velocity at iteration (k+1). Inert(k) is a linearly decreasing inertia term with respect to iteration. $Pbestx_i$ is particle i's personal best position in hyperspace. $Currentx_i(k)$ is particle i's current position. Gbestx is the global best valued position among all particles. The rand terms impart uniform random impulse over the range [0,1]. And of course, $pos_i(k+1)$ is particle i's position after updating with the new velocity. A much lengthier discussion of the mechanics behind PSO and some of the variations is in the first appendix.

PSO for Changing Environments

For this research, the PSO algorithm was improved with the capability for tracking a changing error environment. Motivation stems from the need to be able to retrain a neural network controller 'on the fly' to state data that will change the error metric value. Other researchers have tackled the changing error topology problem [6] and come up with the idea of a sentry particle that polls the error space for changes. The improvement made here over the typical sentry method is to assign this task to the global best particle rather than a random particle or to a specially created particle that does nothing for the convergence

properties of the algorithm. Using the global best particle as a sentry has a couple advantages. One, less computation time than the extra particle method. Two, feedback on the current best solution is immediate and does not require interpretation. In the end, for this problem, we only really care about the global best. Status monitoring of the global best particle then makes sense. The current improved strategy polls the global best position to a user defined tolerance to see if it has changed. Instead of every iteration, the polling takes place every 5 iterations. If the tolerance has been exceeded then a changing environment has been detected.

When a changed environment has been detected the particle's velocities are shaken up by increasing them a single order of magnitude. Additionally, the personal best of each particle is reset to the current position. The global best is not reset. There is no need because as far as PSO knows it is still the true best position. This strategy has proven very successful for test cases involving nonlinear dynamic functions of several dimensions. The success is dependent on the relation of time (or iteration) to the test function as well as the rest between change detection polls. 5 iterations was found to be a good choice for the poll interval with all the test cases but it is expected this parameter can be tuned further with study. This improvement to the PSO algorithm adds the capability to track the global best solution in a changing error environment. An expected immediate application is in the training and re-training of online neural networks that are given unknown data.

Training Set Order Reduction

Being a population based algorithm, PSO is only as fast as the objective cost function it must evaluate. Because each iteration requires multiple calls to the POST state generator this time can quickly become prohibitive. For each initial condition under the MPL mission scenario architecture, a PSO optimization run could take anywhere from several minutes to several hours. Because of this, there was a need to reduce the amount of initial conditions used, but still represent as best as possible the full range set of conditions the MPL is expected to encounter during EDL. The full range is needed because a neural network will be trained to generalize from these trajectories.

The full 2000 point initial data set was reduced to 76 initial conditions by performing two separate three dimensional convex hull operations. The convex hull represents all the points that exist already in the data set, describing the full boundaries of the data set.

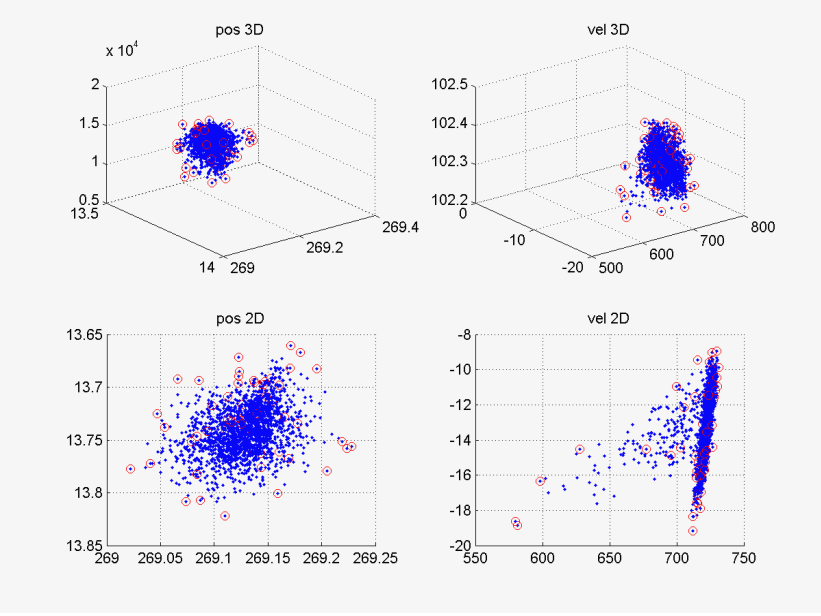

Figure 24. Reduction of Initial Conditions

One convex hull operation was done in position space and a second was done in velocity space. The pared down data set from each convex hull operation was combined to yield 75 initial conditions. A 76^{th} initial condition was artificially created by taking the mean of the 75 boundary points. The above figure shows the full 2000 point data set in position (longitude, latitude, altitude) and velocity space (surface relative components) with 2D and 3D views in order to inspect the shape for exploitation. The circled points represent the combined convex hull reduced set in both velocity and position.

Cost Function

The PSO algorithm used here operates by evaluating a multi-input single-output cost function. The cost function places values on each particle for ranking within the PSO algorithm. The MPL optimization cost function is cast as a function with 25 input parameters and a single output parameter, representing the cost.

The inputs to the function are 7 time variables, 6 bank angles, 3 bank angle rates, 3 angles of attack, 3 sideslip angles, and 3 constant thrust throttling parameters. These inputs represent the control for a single trajectory over its entire run.

cost = f(t1, t2, t3, t4, t5, t6, t7, b1, b2, b3, b4, b5, b6, br1, br2, br3, aoa1, aoa2, aoa3, ss1, ss2, ss3, thr1, thr2, thr3)

Where:

cost = sum of weighed sub-costs (n/a)

t1 = time to deploy ballistic parachute (s)

t2 = time to convert parachute to parafoil and perform 1st steering event (s)

t3 = time to perform 2nd steering event (s)

t4 = time to perform 3rd steering event (s)

t5 = time to drop chute, activate engines, and perform 1st thrust event (s)

t6 = time to perform 2nd thrust event (s)

t7 = time to perform 3rd thrust event (s)

b1 – b3 = bank angles associated with parafoil steering (deg)

b4 – b6 = bank angles associated with engine vector thrust (deg)

br1 – br3 = bank angle rates associated with engine vector thrust (deg/s)

aoa1 – aoa3 = angles of attack associated with engine vector thrust (deg)

ss1 – ss3 = sideslip associated with engine vector thrust (deg)

thr1 – thr3 = throttling parameter, controls thrust magnitude (n/a)

The cost term on the left hand side of the equation is scalar value but represents a weighted sum of many individual hidden cost terms. These hidden cost terms are where the function is able to meet the objectives of minimizing distance to target and reducing end velocity, among others.

The cost function takes the inputs and generates a POST input deck, then calls POST to run the deck, collates the POST output, converts to the Matlab format, and uses the output to calculate various costs.

This is a description of a multi-objective optimization problem (MOP).

Multi-objective Optimization

Mathematically, a MOP is expressed as:

$$\min_{x \in C} F(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_n(x) \end{bmatrix} \qquad \ldots(MOP)$$

$$C = \{x : h(x) = 0,\ g(x) \leq 0,\ a \leq x \leq b\}$$

The goal is to optimize a set of functions F(x) subject to a set of constraints C. For this research, x represents the optimal control. For a multi-objective problem there exists the concept of pareto optimal sets. That is, this is a set of x for which no further improvement of any of the f1(x) … fn(x) in F(x) can be done without making one or more other f1(x) … fn(x) worse. One objective may optimize at the expense of another objective. Finding the balance where no more global improvement occurs is the pareto front and it can be a set of solutions.

There are several ways of dealing with the difficulty of answering what improvement even means in a MOP. One of the simplest and definitely the most used [11]

is the concept of the weighted sum. This is where the multi-input multi-output (MIMO) problem is reduced to a multi-input single-output (MISO) problem. Mathematically this is expressed as:

$$\sum_{i=1}^{n} \alpha_i f_i(x), \qquad \alpha_i > 0, \ \ i = 1, 2, \ldots, n.$$

Each function within F(x) is multiplied by a weight alpha greater than zero and summed over the size of F(x).

The MPL problem is a MIMO but can be recast into a MISO using the standard weighed sum of objectives. The next section describes the cost terms as well as the weighting strategy in detail for the MPL problem.

Terms of the Cost Function

The most important objectives are making sure the terminal state velocity is below 1 m/s, the range to target is 100m or less, and fuel usage is minimized. There are several other cost function objectives that were parameterized and the next few paragraphs will go through each.

Rate limits: this checks the POST output to see if either the bank angle or thrust exceeded design limits. If so, the cost function immediately returned a total cost of 1e99.

Range at termination: this checks the range to the target at the end of the trajectory. Individual sum squared errors are calculated for altitude, longitude, and latitude.

Everything is converted to meters and a total distance cost is calculated as the square root of the sums of the three components. Additionally, if the vehicle goes below minimum altitude (calculated from the Topography Sensor model) at any point that is considered a crash landing and the cost for the altitude component is multiplied by 1e6.

Range at engine start: this is similar to range at termination but calculates the cost based on how close to target the lander is when the engines turn on for the gravity turn. If the engines never turn on, this cost is set artificially large.

Terminal speed: this is a sum square root error of the velocity components in the ground relative frame. Higher speed equals higher cost.

Heading angle: this calculates a geometric difference between actual heading and the straight line heading to the target from any position along the trajectory. This cost term penalizes velocity headings that do not lead toward the target. It has a similar rule strategy to the original First Guess Algorithm.

Pitch angle: this is the 2nd part of the heading angle cost, exactly as the First Guess Algorithm specifies.

Glide time: this cost term attempts to maximize glide time by using a cost term that is the inverse of elapsed time.

Altitude drop: this cost term calculates the altitude change rate and assigns a larger cost to a larger loss rate. This will minimize altitude drop speed to help with altitude bleeding during steep bank angle commands.

Propellant weight: this cost simply penalizes the cost function if the lander runs out of fuel before the terminal conditions are met.

All the above cost terms are multiplied by weights and summed to yield the total cost function result. A large amount of mixing and matching of weights was performed. In fact, the literature states that there is no sure way to choose the weights needed to transform the problem from MIMO to MISO [11]. So by lots of iteration and inspection a set of final weights was arrived at that encouraged but did not dictate PSO to find optimal trajectories for each of the 76 initial conditions. The final configuration heavily weights the distance and velocity criteria, with a distant third magnitude weight for the propellant weight constraint. Though some remnants of the other cost functions remain in the weightings, their influence is minimal at best when compared with the main three criteria.

The cost function code is attached as an appendix where the weights are detailed.

Results

The results of the Trajectory Optimization with PSO are exciting. In 88% of the initial conditions, PSO was able to find an optimum trajectory under 10m from the target reference. This is an order of magnitude improvement over the research goals, 2 orders of magnitude over the Mars Science Laboratory (future flight) strategy, and 3 orders of magnitude better than the Viking derived strategies.

The figure below shows the optimization of a particular trajectory. The PSO algorithm calls the cost function with a set of candidate solutions at each epoch (iteration) and the best solution is shown below. As the PSO iterates the graphical output is dynamically changed to track several parameters of interest such as speed, ground track, propellant weight, control action, etc. This trajectory took about 600 iterations to converge, equaling several hours of serial computer time.

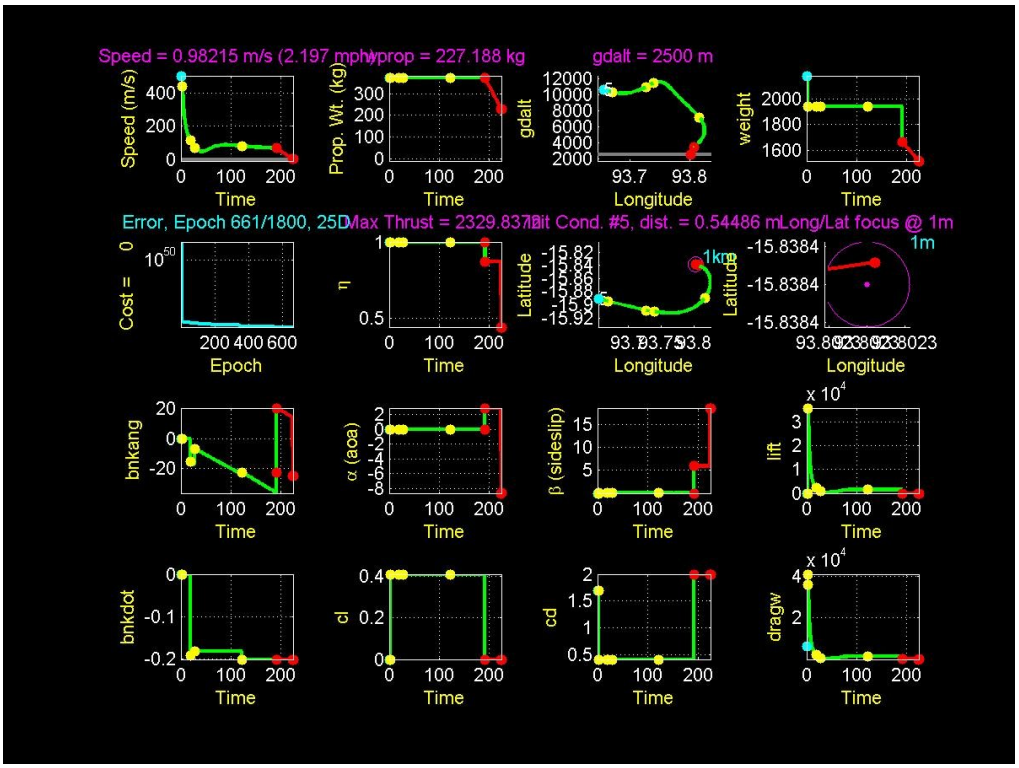

Figure 25. PSO Trajectory Optimization Run

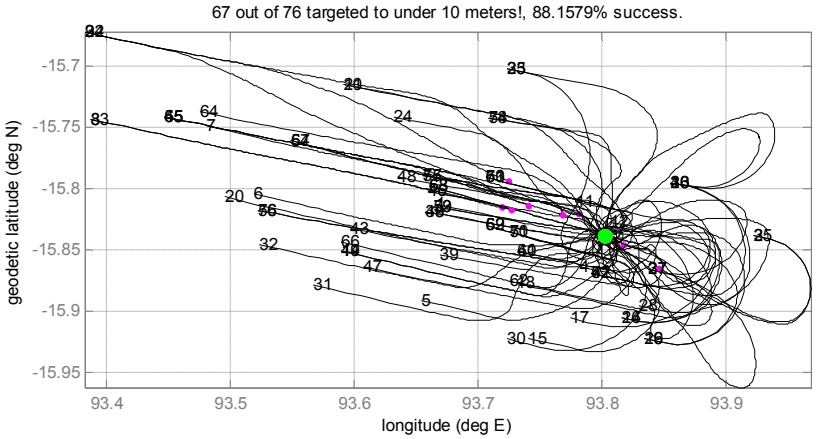

Figure 26. Trajectory Optimization Results

The above figure shows each initial condition of the reduced set and its trajectory. There are some initial conditions that failed to meet the targeting objectives. In all cases this was because there just wasn't enough lift, altitude, or fuel to reach the desired goal. The initial conditions were simply too far away from the target when using the MPL scenario and the associated constraints. There were also many cases that upon multiple runs showed different solutions that nonetheless achieved the targeting goal. This is a qualitative approach to finding the pareto front. It is also a good data set for the next step, that of training a neural net based reference trajectory generator that can generalize from inputs it has not seen before.

Table 3. Distance Breakdown for Trajectory Optimization

Distance to Target	% of trajectories meeting criteria
<= 10m	86.8
<=100m	89.5
<=1km	93.4
>1km	6.6

The Trajectory Optimization Study has proven that a mission with the MPL scenario, namely a bank angle commanded steerable parachute followed by terminal descent on thrusters can reach the research goal of under 100m distance to the target 89% of the time.

ARTIFICIAL NEURAL NETWORK TRAJECTORY GENERATOR

This part of the research details studies done in an effort to leverage from the PSO Trajectory Optimization study and build a static Artificial Neural Network Trajectory Generator (ANNTraG) capable of generating physically realizable reference trajectories for the whole 2000 Monte Carlo initial condition point set. The goal is to build a black box that will take as input the current state of the MPL and return a reference trajectory for it to follow.

Neural Network Basics

Artificial Neural Networks (also Neural Networks or ANN for short) architectures are the most widely recognized Computational Intelligence (CI) algorithms and are based on the way our brains neurons were thought to interconnect and transfer information. They simulate massively parallel operations and are very good at non-linear function approximation. In fact, it is a proven mathematical theorem that any single layer ANN can resolve any arbitrary set of input and output targets [16]. Of course for very complicated high dimensional problems you have to be very careful how you build up the ANN if you want it to work. ANN's are 'trained' to data that is presented to them. They have the ability to learn and recognize new I/O patterns without any a-priori knowledge, unlike the early versions of expert systems. The classic way to present data and update the neural network is via backpropagation.

The following figure shows the salient points of an ANN. It is essentially a highly nonlinear combination of inputs, weights, biases, and functions. The input is transformed to the output via weighted summations fed into a series of (possibly) nonlinear functions. For this research the only details that are needed are that ANN's can be used to approximate functions, they must be trained with presented target data, they are modified by changing the weighting between connections, and they are often good at generalizing. Generalization is the ability to provide reasonable output to input it has never seen before. This works best if the input is in the range of the training data.

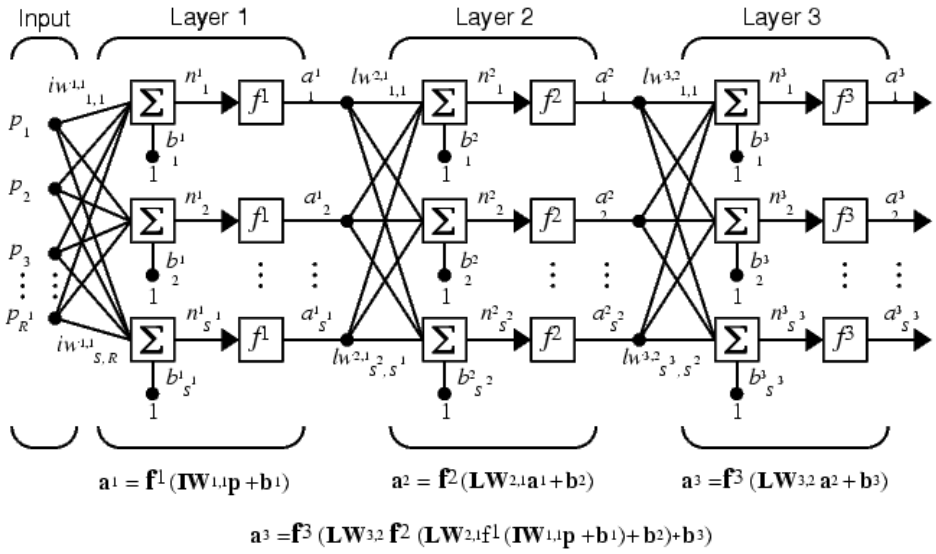

Figure 27. Generalized Schematic of an ANN

Validation Study

A study to determine if the neural network open loop controller and trajectory generator had promise was performed. In this validation study, a set of neural nets have

been trained as inverse controllers on a set of simplified problems based on the Mars re-entry problem for a hypothetical Mars Sample Return Mission.

Two sub problems were looked at. The first consisted of a single fixed hand off point with a random target pulled from a set of previously determined reachable targets. The goal was to have the controller choose the proper time to turn the ballistic parachute into a lifting parachute in order to reach the desired target point precisely.

The second problem is a more complicated version of the first. The initial conditions were now perturbed to reflect the range of possible hand off conditions for the MPL problem.

Validation Problem 1

The first step was to run the post deck 'runVal1.inp' several times to get data to train a neural net with. Briefly, the input deck 'runVal1.inp' is set up as follows:

This deck is set for no POST-centric targeting or optimization. Event 1: Problem is set up with fixed initial conditions in inertial coordinates including 'timeo' (all from M2001 parachute handoff conditions). Mars atmosphere input as tables with Braun wind modifiers. Parachute is set up as ballistic and immediately starts to open. Event 22: Ballistic parachute is fully deployed to a diameter of 13 meters. Also, weight drops to reflect loss of hear shield. Event 25: This event happens at some 'tdurp' time after the previous event, it is our only control for this problem. At this event the ballistic parachute is 'firewired' into a lifting steer-able parachute of L/D=1. There is no steering done, we

just go on a glide in same direction until we hit the ground. Event 80: Impact on 'surface' is chosen when altitude is at 2500m. Event 500: This is the end problem event.

So in summary, this input deck takes the lander initial conditions, immediately pops the parachute open, then at some time turns it into a lifting parachute, and then glides to the surface.

To get the data needed for training a neural net the value of 'tdurp' at event 25 was changed by trial and error. This gave a reachable ground track also. The magnitude of 'tdurp' was bound by the 2 conditions:

1. Immediately switch to lifting parachute (tdurp = 0 sec)
2. Never switch to lifting (tdurp = 250 sec, this was found from doing a few experimental runs)

Values in between were chosen to give a reasonably complete description of the range of impact values (i.e., tdurp chosen for a smooth set of end values for latitude and longitude).

The training dataset is a 3 variable set of data describing what value of 'tdurp' it takes to reach a certain longitude and latitude impact position.

This set of end values of the impact position was fit to a 4^{th} degree polynomial to get an approximation of the ground track such that:

$$gdlat = f(long)$$

Where 'gdlat' is geodetic latitude and 'long' is longitude (bound by minimum and maximum found from POST runs). This equation is used in testing out the neural net.

For training, the original 3 variable set was normalized using built in functions of the Matlab Neural Network Toolbox. Then this normalized data was fed to a 2x2x1 neural net. The hidden layer's activation function was 'tansig' and the output neuron was linear. This is the standard function approximation set up [16]. The training method used was 'trainlm' because of its speed and resistance to local minima.

Choosing the data set to train with became a big problem. Initially a set of 5 pts was chosen, corresponding to tdurp = [300,100,50,10,0]. This trained with no problems but the testing showed poor generalization. Then a batch of 30 points was trained. This did better but the minimum ranges of latitude and longitude were wrong, the net still did not generalize well. This process went on for awhile, choosing various data sets of various sizes and strategies. One was chosen for symmetric tdurp values, another was chosen for symmetric longitude, latitude values, etc. Eventually it was found by clustering the data at the beginning, middle, and end of the impact range the net could train with 10 data points and have pretty good generalization. In other words, a few points were taken for what could be determined as 'very little time on lifting chute', another set for 'almost all the time on lifting chute', and the last for the midrange between. The training set eventually used looks like the following figure. The values for tdurp are shown by the ground impact points.

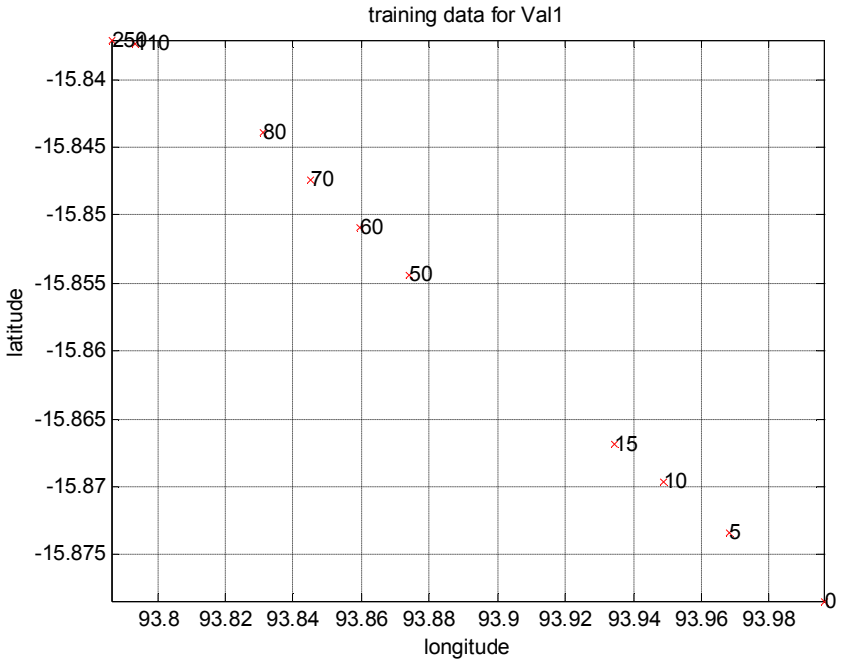

Figure 28. Neural Training Set for Validation Problem 1

After the training set was used to train the neural net, a batch of 21 POST runs was performed for testing. The data is all different from that used for training except at the minimum and maximum bounds. Results are in the next section.

Validation Problem 1 Results

Table 4. Validation Problem 1, Target Errors

err long (m)	err gdlat (m)	distance (m)
0	7.438288	7.438288
67.66433	19.29925	70.36282
137.1557	44.22152	144.1084
100.3266	38.87651	107.5956
43.98979	20.82451	48.66993
5.451491	2.322613	5.925646
15.4648	7.624232	17.24207
24.77082	11.83018	27.45074
25.11847	12.3513	27.99093
18.09923	5.54562	18.92976
3.06505	8.774742	9.294446
20.81025	23.35628	31.28236
47.27843	33.82272	58.13108
65.90805	36.90438	75.53678
64.97097	30.58729	71.81092
36.27505	14.86489	39.2026
18.64732	7.754283	20.19534
81.99757	30.18847	87.37817
121.5377	42.68663	128.816
94.18551	32.50739	99.63757
41.49087	12.76286	43.40948

Looking at the above table an immediate quantitative judgment can be reached. Since the original goal is to hit the target within 100m and the data only has 3 points outside of this range, it can be seen this method achieves the goal 85.7% of the time. With better training a higher percentage will reach the goal. The tables show the out of tolerance results in red. The 21 point data spread is shown below graphically with '+' values the actual or POST data and the 'x' values are what the controller predicted.

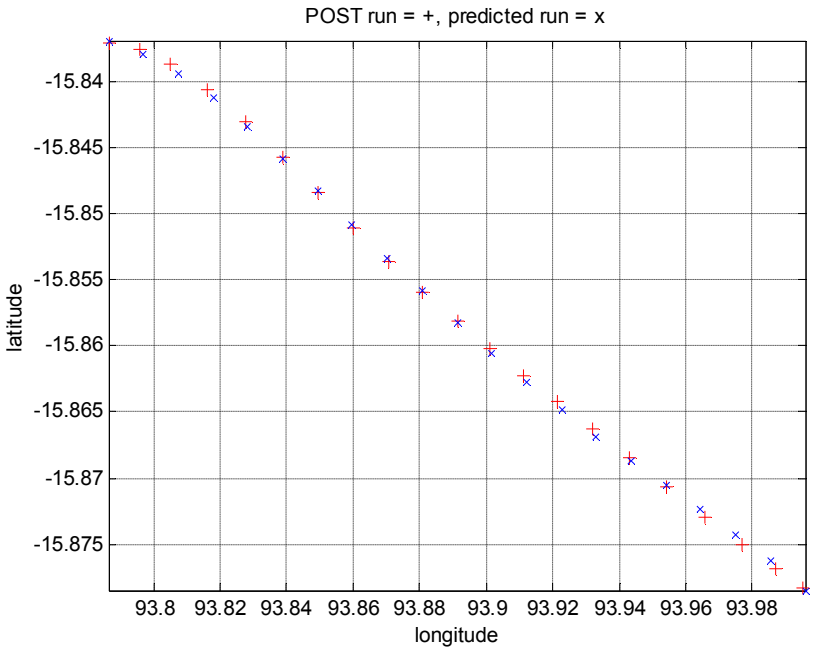

Figure 29. Validation Problem 1, Impact Points

Validation Problem 2

This problem was set up exactly the same way as the first problem except one degree of freedom for the initial conditions was perturbed. This was the geodetic altitude (altitude normal to surface). Since the initial conditions from the first problem were in inertial coordinates (xi, yi, zi, vxi, vyi, vzi) it was necessary to convert to spherical coordinates in order to isolate just the initial altitude. The spherical coordinates used were the planet relative kind: long, gdlat, gdalt, velr, azvelr, and gammar. The actual numbers used were the same as the first problem except that now they are in spherical coordinates and the altitude is perturbed. The static initial conditions used are shown in the table below.

Table 5. Initial Conditions in Planet Relative Coordinates

GdLat (deg)	Long (deg)	Velr (m/s)	AzVelr (deg)	GammaR (deg)
-15.812887	93.666346	501.01721	102.79617	-21.07023

The training set for Tdurp was the same as last problem initially:

Tdurp = [0,5,10,15,50,60,70,80,110,250] in seconds.

The training set for GdAlt was initially chosen as the maximum, minimum, and the middle point of the total range from the M2001 handoff conditions. These were:

GdAlt = [16950.476, 13247.23925, 9544.0025] in meters.

In order to improve training results, the following Tdurp and GdAlt were used (from trial and error):

Tdurp = [0,5,10,15,50,60,70,80,100,110,250,-0.1]

Gdalt = [17000, 13250, 9500]

The -0.1 in Tdurp was there to prevent the net from returning negative numbers. That is, extending the data set out past the normal range helps training for the actual range. When implementing this there is a need to remember which range is physically valid. The same idea holds with the modification for GdAlt.

The three data sets are shown in the following graph with the reachable impact points for the max altitude indicated by ‘o’, the points for minimum altitude shown by ‘+’, and the midpoint altitude’s reachable impact points shown by ‘x’. The lines connecting the data are not meant to imply a linear interpolation; they are merely to make it easier to group the data visually.

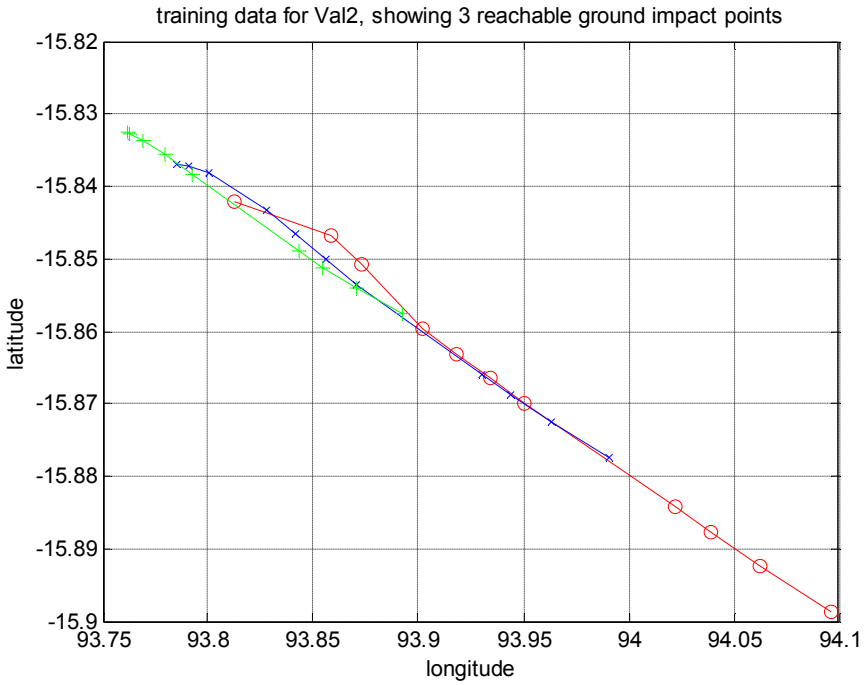

Figure 30. Reachable Ground Impact points for Validation Problem 2

For each of these 3 impact sets, a polynomial was found of the form:

gdlat = f(long)

This is the same concept as in the first section. Here there is a more complicated problem. We have data for 3 altitudes (and associated impact set functions).

The function for GdAlt not equal to the 3 given is not defined. From looking at the above graph it can be seen the problem is not as simple as just extending or limiting the minimum and maximum longitudes. The actual shape of the impact point function changes with altitude, probably partly due to the Mars atmosphere winds since the bulge gets more pronounced with more time on the lifting parachute. Because of this observation, a neural net was trained with the following architecture:

Input:

1. Altitude, GdAlt (initial)

Outputs:

1. maximum longitude we can reach, maxlong
2. minimum longitude we can reach, minlong
3. Polynomial coefficient 1, PL(1)
4. Polynomial coefficient 2, PL(2)
5. Polynomial coefficient 3, PL(3)
6. Polynomial coefficient 4, PL(4)
7. Polynomial coefficient 5, PL(5)

The training data was initially the previously mentioned 3 sets of impact data. The fitting polynomial function was set to 4^{th} order from trial and error (5 coefficients). The neural net trained in 69 epochs with 5 hidden neurons. Eventually a group of 6 altitudes was used for training:

Gdalt = [17000, 13250, 9500, 16950.476, 13247.23925, 9544.0025]

A few of the training runs for the neural net are shown below. As can be seen the higher the initial altitude, the more complicated the resulting set of impact points. This leads to more uncertainty in our prediction of reachable impact points for higher altitudes. The lower altitudes fit perfectly. This can be viewed as part of a propagation error, that is,

the predicted impact point will not be exactly correct because the function that chooses the impact points is not perfect. This error will propagate over to prediction of the correct Tdurp value to reach the predicted impact points. The below results were deemed close enough for this study. Further training of this neural net with better/different data or more neurons will yield more accurate results.

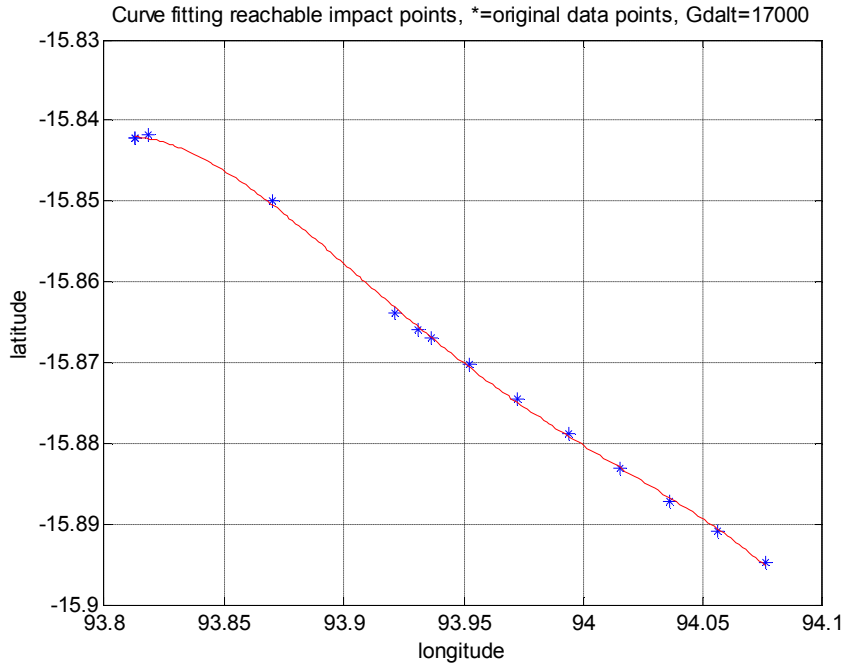

Figure 31. Curve fit of Reachable Impact Points, 17km Altitude

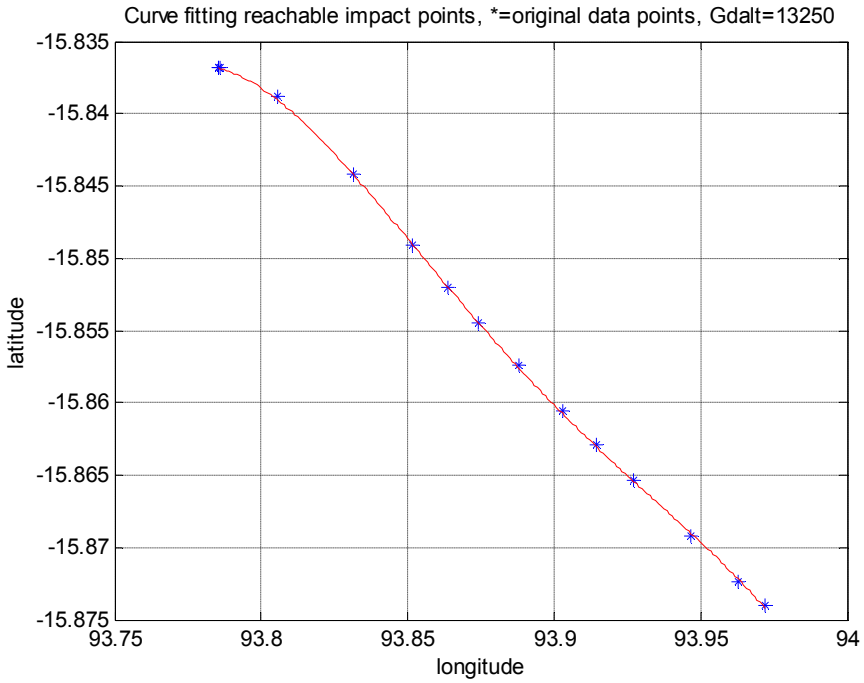

Figure 32. Curve fit of Reachable Impact Points, 13km Altitude

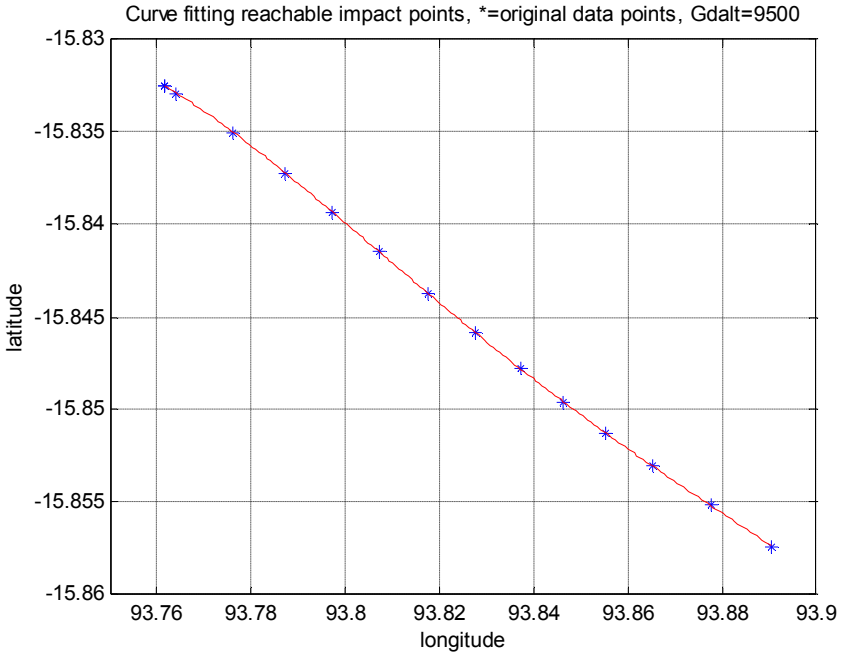

Figure 33. Curve fit of Reachable Impact Points, 9.5km Altitude

After getting a neural net approximating the reachable impact points as a function of initial altitude it was time for training the controller. The same data sets derived for the impact point study were used for developing the neural net controller. It has the architecture:

Inputs:

1. Longitude, Long (desired)
2. Latitude, GdLat (desired)
3. Altitude, GdAlt (initial)

Output:

1. Tdurp @ event 25 (predicted)

The training data used consisted of 150 states originally but with some problems training, that was pared down to 141 points by getting rid of obviously out of bounds values.

Training finally yielded this graph where the net outputs are the asterisks and the training data are the dots. Training was performed to mean square error of less than $2x10^{-5}$.

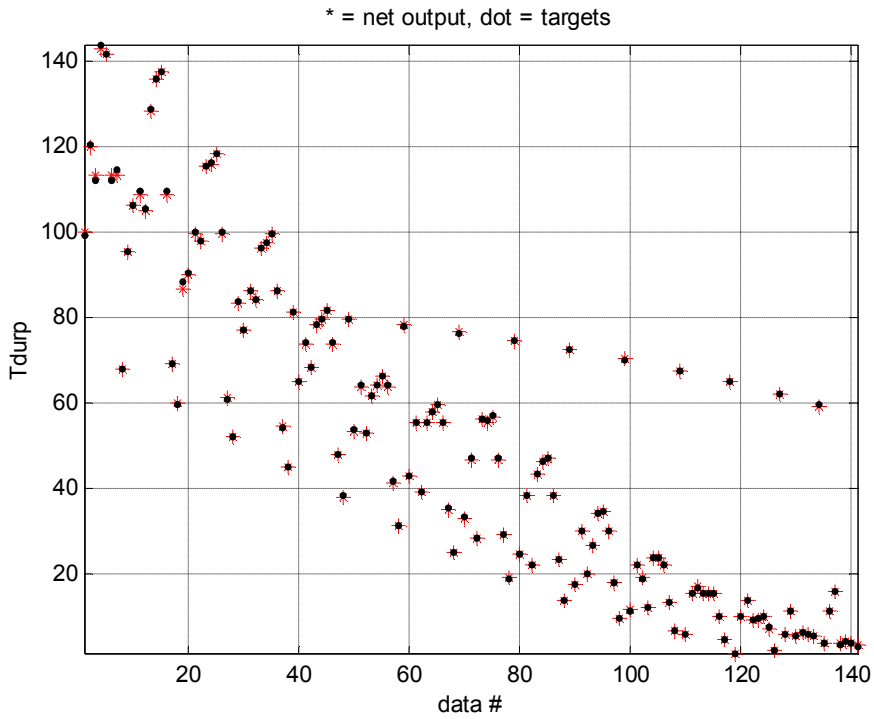

Figure 34. Training Data overlayed with Neural Net derived Data

Validation Problem 2 Results

28 runs were performed for the testing of this problem. There were 7 different altitude choices with 4 impact points each. The performance was significantly lower than in the first problem. 9 of the 28 failed to achieve the goal of within 100m of target. That gives performance of this problem a 67.9% success rating.

Table 6. Validation Problem 2, Target Errors

err long (m)	err gdlat (m)	straight dist (m)
0	3.35526638	3.35526638
3.673703	3.63951834	5.1709072
3.40036763	4.05865202	5.29519239
101.648521	0.139754249	101.648617
3.50223143	4.78604743	5.93058808
2089.44938	167.27395	2096.13437
41.2530775	27.625707	49.6486877
32.0647538	11.4000411	34.0310061
10.8164733	3.21617008	11.2844957
27.0132346	15.5076846	31.1480838
8.75245501	8.70703738	12.3457672
4.64564648	6.03990635	7.61987533
20.3774922	13.8218294	24.6229377
16.4426085	61.1955434	63.3660312
30.6981906	0.905711047	30.7116122
17.9593726	8.79452955	19.9971678
243.641113	1.7365555	243.647302
17.96046	1.09330348	17.9937055
31.1640694	15.7036769	34.8970584
20.1908859	8.0413968	21.733291
181.643796	14.0459625	182.186053
288.185507	35.8130999	290.402247
16.6242951	3.54050226	16.9972424
10378.1032	2313.24524	10632.7852
148.165909	18.7948422	149.353228
824.300454	113.745038	832.111271
727.921374	95.108696	734.108433
14956.5651	3108.39292	15276.1551

Next is the graph of the 28 test runs. The ‘x’s or predicted points were mostly in line with what was expected, however some of them are considerably out of range to what is known possible with the used input deck. The reason for these errors have to do with the net that trained to the reachable impact points. The 4^{th} order approximation to the reachable impact points for each altitude is a good one but there are 2 problems with that net as it stands. Firstly, the limits of max and min longitude are not represented very well. The

curve of the polynomial that approximates the impact points does very well on the middle points but tends to 'curve out' at the limits. Secondly, the net does not respond well to altitudes it has not seen, the functions become unpredictable in some cases. Both of these problems can be theoretically solved by spending some more time on training the impact point approximation net.

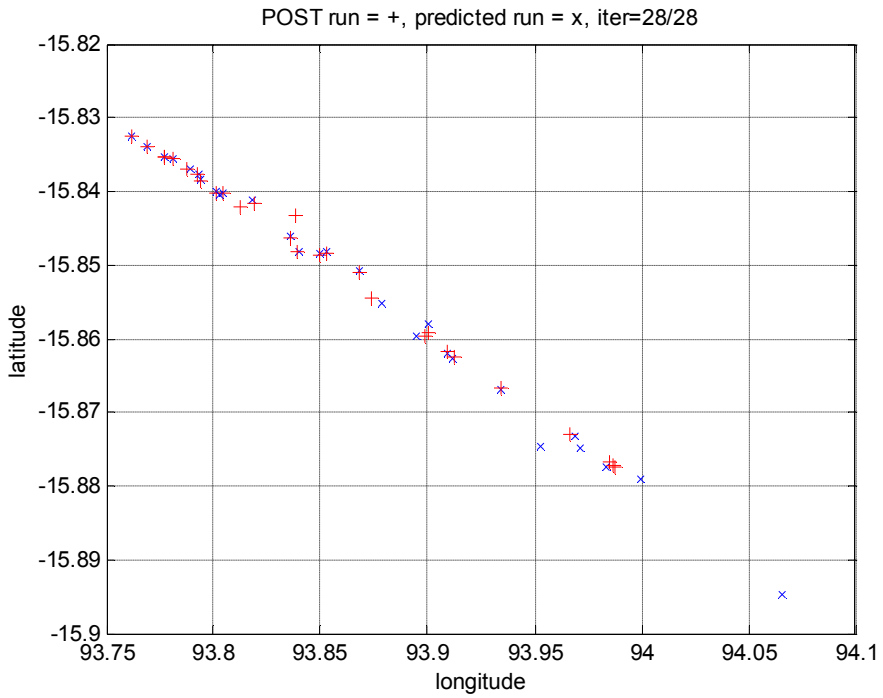

Figure 35. Validation Problem 2, Impact Points

Validation Study Conclusion

The results for both of the problems investigated are mildly encouraging. As is expected, the main problems occur in preparing the data for training and dealing with the boundary conditions on the data. Since the eventual goal is to have a controller put the lander down wherever mission planners wish (within the reachable space) the final problem will end up with more controls and hence complexity than the ones shown here.

That fact points to the need for a methodical approach to dealing with the inevitable exponential growth in the data sets used for training.

The neural net approach appears very robust in terms of predicting the correct 'Tdurp' for any longitude and latitude pair when the altitude is not perturbed and the boundary conditions are not examined too closely.

When the impact point neural net is fed altitudes it has not seen before it does a poor job of generalization. The validation study for the neural net inverse controller approach indicates there is promise in this approach but also pitfalls. The conclusion reached is to go ahead with the attempt on the full neural net trajectory reference generator with the PSO optimized reduced set.

ANNTraG

ANNTraG stands for Artificial Neural Network Trajectory Generator. Three versions of a static neural network trajectory generator were attempted. These were a discrete version, a continuous version, and an interpolated version.

Discrete

The discrete version of the trajectory generator trains a neural network with 12 inputs and 26 outputs. The inputs are initial position and velocity components and terminal position and velocity components. The outputs are the 25 parameters used in the Trajectory Optimization strategy, 7 time variables, 6 bank angle variables, 3 bank angle rate variables, 3 angle of attack variables, 3 sideslip variables, and 3 thrust throttle parameter variables,

plus a ‘class’ parameter. Except for class, these represent the controls used during the discrete time events as outlined in the MPL POST input deck. Class is simply the index of the initial condition used (out of the 76 state reduced set). The class parameter is an indication of pattern matching of the input to the closest similar known initial condition. This is a useful parameter for heuristic building as well as clustering investigations and fuzzy rule firing.

The idea behind this neural net is to be able to generate open loop control estimate for specific events throughout the trajectory, i.e. the 3 timed parafoil steering events and the 3 timed thrusting events. Using the terminal state from known trajectories as an input during training will be replaced by desired terminal states during testing and actual runs. In fact, the idea is that even poor trajectories at to the training knowledge base of the ANN thereby increasing successful generalization. The output of the network, the control estimates, are fed directly to a program that generates a POST input deck with those control parameters. Then the whole deck is run and the result is fed back into the ANN for further training.

The neural net architecture consisted of the classic function approximation ANN with 2 hidden layers of 200 and 100 neurons. The activation functions for the hidden layers were the tansig function, a sigmoid function that ranges from [-1,1], as opposed to a normal sigmoid that ranges from [0,1]. The output layer was activated with a pure linear function. There were two methods attempted for training. First the Levenberg-Marquardt optimization was used but as nodes were added and training data set size increased from simple single test cases to the full 76 state reduced set, the memory requirements grew too

large. Because of this, a modified gradient descent algorithm was used for weight and bias updates during training. This version had a momentum term to help 'push' the solution past local extrema. The performance metric function was toggled between simple sum squared error, mean squared error, and mean squared error with regularization. The network was trained for 1000 epochs to an unconstrained minimization goal.

The discrete ANNTraG performed well on the training data and was relatively easy to train. However, when presented with input it had not seen before, either within or outside the expected range, the generated trajectories did not achieve the desired end state.

Continuous

As an attempt to improve the generalization on the discrete case, the continuous ANNTraG was developed. This ANN uses all the data for each trajectory and predicts the desired control that will achieve the next step in the trajectory.

The continuous version of the trajectory generator trains a neural network with 14 inputs and 7 outputs. The inputs include initial positions and velocities, current positions and velocities, current propellant weight, and current elapsed time. The outputs are the controls: commanded bank angle, commanded bank angle rate, commanded angle of attack, commanded sideslip, commanded thrust throttle parameter, current vehicle configuration, and current class, where class is the same as described for the discrete architecture.

The current vehicle configuration output is a calculated variable that determines whether the vehicle is in freefall, on a ballistic parachute, on a parafoil, or has engine thrusting activated. The pseudo code logic behind these choices is as follows:

```
if ( engine_on == true )
        current_vehicle_state = ENGINE_THRUSTING_ACTIVATED
else if ( parachute_diameter > 0 )
        current_vehicle_state = BALLISTIC_PARACHUTE
else if ( drag_coefficient == 0.41 AND lift_coefficient == 0.41 )
        current_vehicle_state = PARAFOIL_STEERING
else
        current_vehicle_state = FREEFALL
end
```

Another slight variant on this replaces the current_vehicle_state single variable with 4 on/off flags representing the same concept.

The continuous case was designed to take the entire state dataset from the known trajectories (not just initial and final conditions) and match them up with the appropriate control parameter at the same time step.

Using lessons learned from the discrete case, the training metric sum squared error was not used. The mean squared error with regularization was the preferred choice but plain mean squared error was used also when training error did not initially achieve goals.

The training algorithm was also able to restart itself with added nodes in alternating hidden layers if the change in error rate was not large enough over a few hundred iterations. This ANN was trained for 25000 iterations on a much larger dataset than the discrete case. The number of nodes was 450 for each of the hidden layers with radial basis functions used for activation. The output layer used the tansig function instead of linear. The training function used was gradient descent with the momentum term.

This ANN does not benefit from bad trajectories in the same way the discrete case does. There is no desired terminal state used as input, it is implicityly built in with the training data. The data from the reduced set PSO optimizer runs was pared down to only include the data that reached desired the terminal state conditions. This means that 66 states were used instead of the 76 that describe the convex hull of the full 2000 state Monte Carlo derived set.

As with the discrete case, the training set fared better than unknown sets. With the continuous case though, the training set had a very difficult time converging to acceptable error values. As a substudy and troubleshooting exercise, rather than have a single ANN providing the output, a series of ANNs were developed each with a single output, focusing on one of the outputs from the original continuous architecture. It was found that all the training output controls tracked very well except for the current_vehicle_state output. This was thought to be due to the step nature of that variable. Investigating that hypothesis, a subnet was was created that used 4 flags to describe the vehicle state rather than a single variable. Though this improved somewhat, it was still very difficult to train. The conclusion was that more nodes and more time would be necessary to train that particular

variable down to a tight error tolerance. Nevertheless, the original continuous ANNTraG performed well enough to develp the open loop control system as a base brain. That is discussed more in the Open Loop section.

Because of generalization problems, several runs were done using PSO as the training function. Comparison found no real difference between this method and the gradient descent in terms of generalization, contradicting the literature [20]. This could indicate a need for a richer training data set, with a few samplings across the range rather than a strict convex hull, in order to fill in some of the gaps.

Interpolation

The interpolated reference trajectory generator uses a different strategy than the other two. Because of severe generalization problems with both the discrete and continuous cases, an interpolated approach was attempted. This was an experiment as much in pattern matching as in function approximation. The network has 6 inputs and 76 outputs. The inputs are initial pos and velocity. Each output represents how similar the input initial condition is to each of the 76 initial conditions.

The goal was to take this output and create a weighted superposition of the 76 successful initial condition trajectories. This weighted superposition or interpolation would be the new reference trajectory. The results were not successful. Interesting trajectories were generated, however they invariably failed to line up with the initial conditions and/or desired terminal condition. This is no doubt partly due to the large number of network outputs when compared with inputs, as well as a lack of data that fully describes the region

under consideration (again using the 66 pared down set from the continuous case). There is also a scaling factor that needs further investigation. The training was only able to reach about 0.02 error using the mean squared metric. Using regularized mean squared error made the training error worse at 0.04. These results were relatively invariant to the addition of more nodes. Since the training did not converge to the point where the training data matched the simulated data, no generalization runs were attempted.

Open Loop Control

Using the Continuous ANNTraG, open loop control was attempted using known trajectories. The open loop control was placed inside a POST iteration. At each time step, the current state was fed into the ANNTraG and the reference trajectory for the next time step was output, including the predicted controls that would match. The following graph shows the results of that effort for a representative case. The known good reference trajectory is shown as a magenta line while the blue and red point data represents the neural net output (for the controls) and the POST generated reaction to those controls (for the states). This looks like it is on the right track but there are significant errors in the commanded controls that propagate through the trajectory. This is expected as the error will be carried through from the continuous ANNTraG training and the design is open loop.

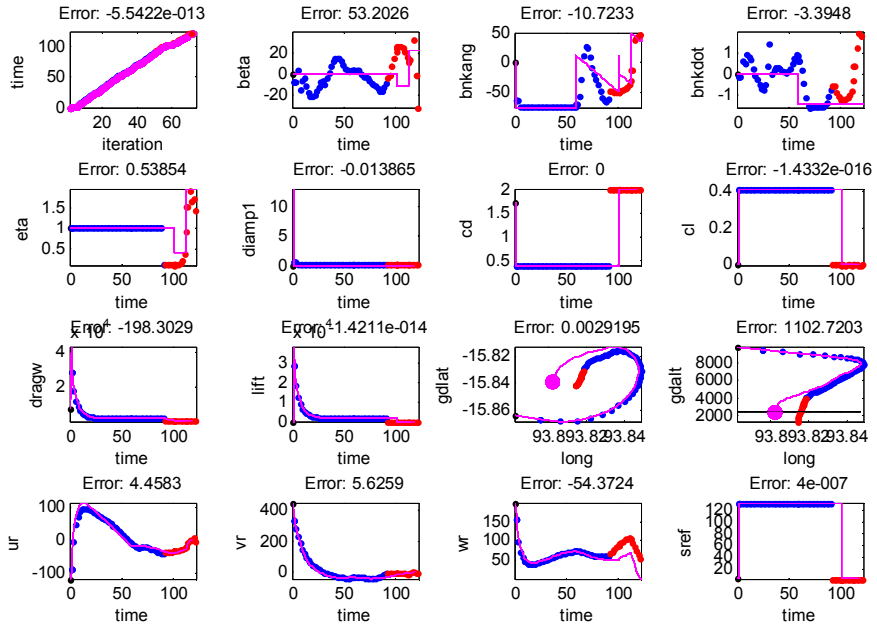

Figure 36. Open Loop Control on Known Trajectory

From the above graph, it can be seen that the major trending of the reference trajectories is correct but the tracking is not exact. This is partially due to several reasons. One, the ANNTraG had a reduced set of data to work with. This can possible be solved by iteratively starting some more runs of the PSO Trajectory Optimization strategy until at least all 76 reduced data set initial conditions converge. The difficulty there is that based on that same study and with the proposed MPL scenario, that some of the initial conditions simply will never be able to converge, whether due to lack of fuel or altitude or lift. Two other solutions present themselves and are both worth investigating further. Firstly, recast the continuous ANNTraG in a similar fashion to the discrete case, where there are added inputs to the ANN describing the desired end state of the vehicle. This would allow the ANNTraG to use 'bad' as well as 'good' trajectories to improve prediction. Secondly, a

different data set can be used. Instead of a strict convex hull of the Monte Carlo initial condition dataset, a uniform sampling across its range can be used as well. This will yield a richer data set with some guarantees of convergence.

The following figure shows another different representative trajectory to that above. This case tracked similarly but shows how generalization is supposed to work. Although not a successful run in a strict sense, the terminal states are very close to the desired (with the exception of altitude). The controller starts off tracking the latitude, longitude, altitude, and velocity references very well, despite not tracking the controller reference for bank angle very well at all. The system is able to compensate for this somewhat and puts the lander on the ground at the correct latitude and longitude with a near correct velocity. The color coding of the figure gives insight into the controller actions along with the vehicle state. Again, the magenta line is the known trajectory that the neural controller should follow. The dotted data is the output from the continuous ANNTraG in the Open Loop Controller configuration. Each graph shows error between known and ANNTraG numerically in the title. The black dots represent the vehicle in a ballistic parachute condition. The blue dots represent the parafoil steering phase of EDL while the cyan area represents freefall followed by red dots indicating active engine thrust. If we treat the figure as a 4x4 matrix and look at entries (2,3) and (2,4) it can be seen that the parafoil steering phase went on longer than in the known case. The bank angle and bank angle rate controls are the culprits when it comes to initially propagating error as the tracking has ringing. This can be tracked down to the difficulty of training the ANNTraG to the current vehicle state output control variable as discussed in the continuous section.

Again, a richer dataset with perhaps a slight reformulation of the ANN's inputs will help alleviate this problem.

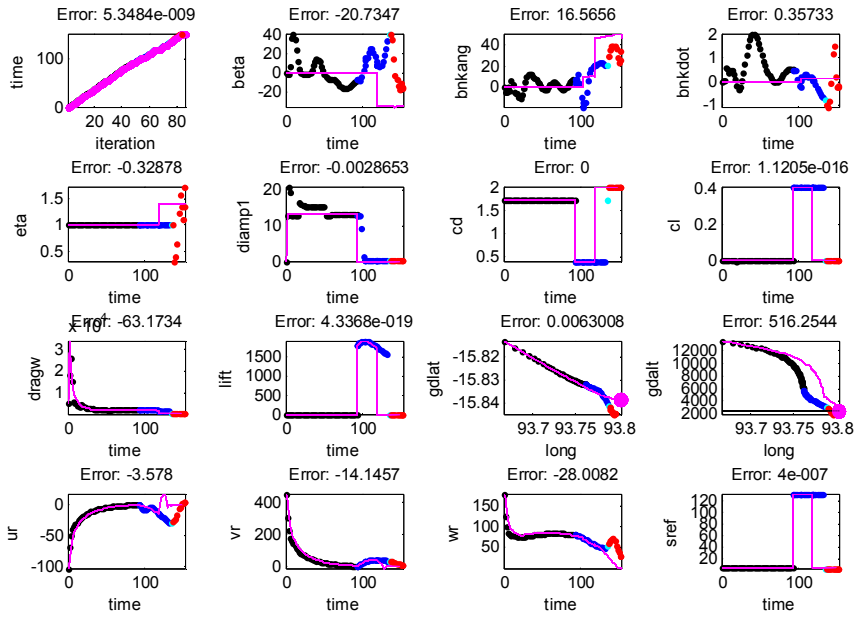

Figure 37. Open Loop Control on Known Trajectory #2

CONCLUDING REMARKS

The proposed Mars Precision Lander (MPL) requires precise re-entry terminal position and velocity states. This Precise Landing is to achieve mission objectives such as rendezvous with a previous landed mission, or reaching a particular geographic landmark. The current accepted state of the art footprint is in the magnitude of kilometers. Reference trajectories have been developed that indicate a reachable footprint can be on the order of meters. This is a significant advance in the state of the art.

As part of this effort, high fidelity and real-time weather and atmospheric models have been developed.

Three descent scenarios have been examined. First, terminal re-entry is achieved via a ballistic parachute with concurrent thrusting events while on the parachute, followed by a gravity turn. Second, terminal re-entry is achieved via a ballistic parachute followed by gravity turn to hover and then thrust vector to desired location. Third, a guided parafoil approach followed by vectored thrusting to reach terminal velocity is examined. The guided parafoil was determined to be the best architecture.

The feasibility of using a computational intelligence strategy to facilitate precision planetary re-entry has been established if not conclusively proven by using Particle Swarm Optimization (PSO) and Artificial Neural Networks (ANN).

Using PSO and the parafoil scenario, optimized reference trajectories are created for an initial condition set of 76 states, representing the convex hull of 2001 states from an early Monte Carlo analysis. The controls are a set series of bank angles followed by a set

series of 3DOF thrust vectoring. The reference trajectories are used to train an Artificial Neural Network Reference Trajectory Generator (ANNTraG), with the (marginal) ability to generalize a trajectory from initial conditions it has never been presented. The optimized reference trajectories represent the best achievable trajectory given the initial condition. Steps toward a closed loop neural controller with online learning updates were examined.

Using mainly bank angle guidance coupled with PSO, the set of achievable reference trajectories are shown to be 88% under 10 meters, a significant improvement in the state of the art. Further, the automatic real-time generation of realistic reference trajectories in the presence of unknown initial conditions is shown to be worthy of more investigation. The closed loop CI guidance strategy is outlined. An unexpected advance came from the effort to optimize the optimization, where the general PSO algorithm was improved with the capability for tracking a changing environment.

RECOMMENDATIONS FOR FUTURE WORK

Closing the Loop

The most pressing need for future work would be to close the loop on the open loop controller. The ANNTraG provides the reference trajectory, there needs to be in place a closed loop system that detects errors between the ANNTraG and the POST generated states, and drives the errors to zero. Tracking of the errors is also useful in the next recommendation.

Online Update of the Neural Network Controller

Once the ANNTraG is up and running correctly as a closed loop system, at least on the training data then machine learning based updates can take place. Any data returned from POST and the Closed Loop controller that disagrees with the ANNTraG generated data will be incorporated into the ANNTraG for future reference. This updating requires an online re-training of the neural network with a memory component. Some standard Machine Learning candidates have been proposed as well as using PSO for the updates. There is some literature indication that PSO works well in this capacity of re-training in real-time [10, 20]. The following figure is a possible architecture to pursue.

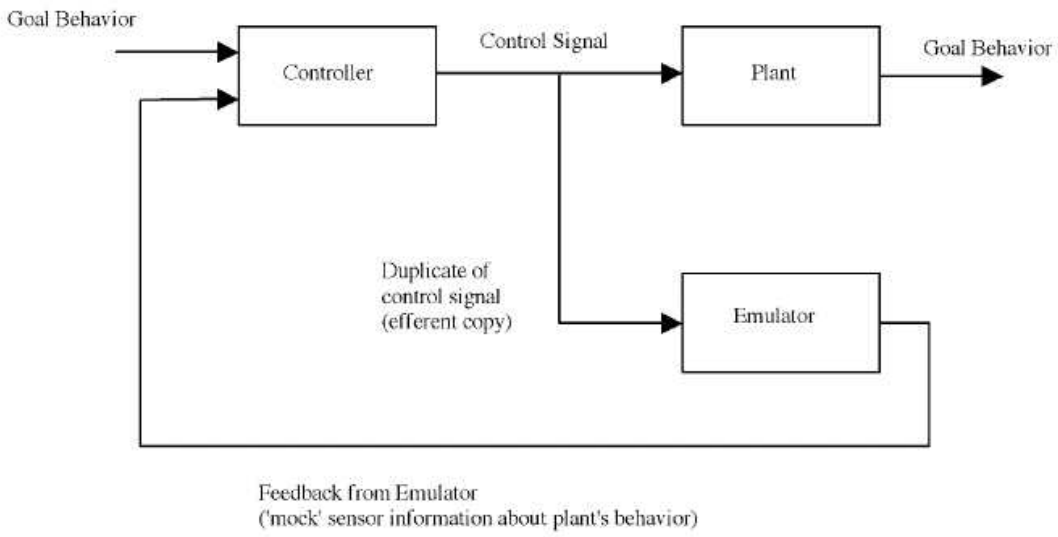

Figure 38. Conceptual closed loop approach

Hazard Avoidance

Some work on hazard avoidance has been done. The Topography Sensor has been integrated into the Matlab controlled POST loop and dynamically updates the desired reference surface. More work needs to be done getting a higher fidelity topographic model. This should use the latest laser altimeter measurements. Also, a smaller scale 'rocks & boulders' model needs to be integrated into the simulation. Currently there is a model coded as a small random perturbation of the surface height but no logic has been added for active avoidance.

The following figure shows an optimization run superimposed on a Topographic Sensor dataset. The various shades from dark to light represent surface height from low to high relative to the mean. The blurry aspect reflects that the current best data is of insufficient resolution to see small features.

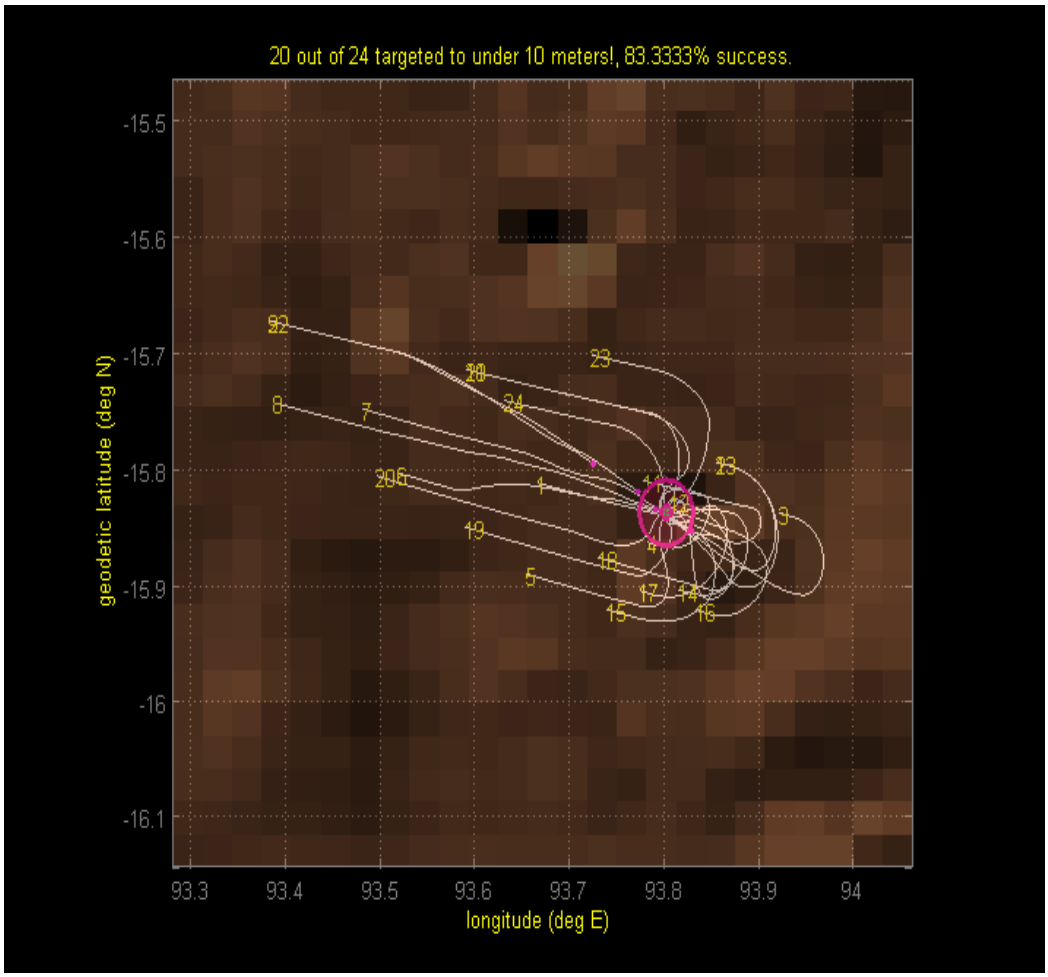

Figure 39. Fidelity of Topographic Sensor with currently available Data

REFERENCE MATERIALS

1. Birge, B., "PSOt - A Particle Swarm Optimization Toolbox for Use With Matlab" IEEE 2003 Swarm Intelligence Symposium, Indianapolis, IN, USA, pp. 182-186, (2003)

2. Brand, T., Fuhrman, K., Geller, D., Hattis, P., Paschall, S., Tao, Y., "GN&C Technology Needed to Achieve Pinpoint Landing Accuracy at Mars", AIAA/AAS Astrondynamics Specialist Conference, Providence, Rhode Island, Aug. 16-19, AIAA-2004-4748, (2004)

3. Braun, Robert D. et. al., "The Mars Surveyor 2001 Lander: A First Step Toward Precision Landing", IAF-98-Q.3.03, 49th IAF Congress, Melbourne Australia, Sept 28-Oct 2, (1998)

4. Braun, R.D., Manning, R.M., "Mars Exploration Entry, Descent and Landing Challenges", IEEEAC paper #0076, FINAL, Updated December 9, (2005)

5. Calhoun, P. and Queen, E., "Entry Vehicle Control System Design for the Mars Science Laboratory", Journal of Spacecraft and Rockets, Vol 43, No. 2, March-April (2006)

6. Carlisle, A. and Dozier, G., "Tracking Changing Extrema with Adaptive Particle Swarm Optimizer", WAC 2002 Proceedings, Orlando, FL, June 9-13, (2002)

7. Carman, G., Ives, D., Geller, D., "Apollo-Derived Mars Precision Lander Guidance", AIAA Atmospheric Flight Mechanics Conference, August 10-12, 1998, Boson, MA, AIAA 98-4570, (1998)

8. Carter, P., Smith, R., "Mars Rover Sample Return – Lander Performance", AIAA 89-0633, Aerospace Sciences Meeting, 27th, Reno, NV, Jan 9-12, (1989)

9. Carroll, M., "Mars: The Russians are Going!", Astronomy, Vol. 21, No. 10, p. 1, 4, 26-33, ARI, (1993)

10. Clerc, M., "Particle Swarm Optimization", ISTE, ISBN: 1-905209-04-5, (2006)

11. Deb, K., "Multi-Objective Optimization using Evolutionary Algorithms", John Wiley & Sons, ISBN 0 471 87339 X, (2001)

12. Donadee, J., Hiltner, D., "Mars Atmosphere Model – Metric Units", (1999), <http://www.grc.nasa.gov/WWW/K-12/airplane/atmosmrm.html>

13. Fogiel, Dr. M., Staff of Research and Education Association, "Handbook of Mathematical, Scientific, and Engineering Formulas, Tables, Functions, Graphs, Transforms", REA, (1994)

14. Gill, P., Murray, W., Saunders, M., Wright, M., "User's Guide for NPSOL 5.0: A Fortran Package for Nonlinear Programming", Techical Report SOL 86-6*, (2001), <http://www.cam.ucsd.edu/~peg/papers/npdoc.pdf>

15. Golightly, G., "Boeing to Design Guidance Parachute Technology for Mars Missions", News Release, (2004), <http://www.boeing.com/ids/news/2004/q3/nr_040816n.html>

16. Haykin, Simon, "Neural Networks, A Comprehensive Foundation second edition", Prentice Hall, Inc., ISBN 0-13-273350-1, (1999)

17. Hassan, R., Cohanim, B., de Weck, O., "A Comparison of Particle Swarm Optimization and the Genetic Algorithm", 46th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics & Materials Conference, 18-21 April 2005, Austin, TX, AIAA 2005-1897, (2005)

18. Ingoldby, R.N., "Guidance and Control System Design of the Viking Planetary Lander", J. Guidance and Control, Vol. 1, No. 3, May-June 1978, pp. 189-196, (1978)

19. Justus, C.G., James, B.F., Johnson, D.L., "Mars Global Reference Atmospheric Model (Mars-GRAM 3.34): Programmer's Guide", NASA Technical Memorandum 108509, (1996)

20. Kennedy, J., Eberhart, R., "Swarm Intelligence", Academic Press, ISBN 1-55860-595-9, (2001)

21. Knacke, T.W., "Parachute Recovery Systems Design Manual", Para Publishing, PO Box 4232, Santa Barbara, CA, 93140-4232, USA, ISBN 0-915516-85-3, (1992)

22. Lin, Chin-Teng, George Lee, C.S., "Neural Fuzzy Systems, A Neuro-Fuzzy Synergism to Intelligent Systems", Prentice Hall P T R, (1996)

23. Machin, R., Stein, J., and Muratore, J., "An Overview of the X-38 Prototype Crew Return Vehicle Development and Test Program", AIAA-99-1703, (1999)

24. McEneaney, W.M., Mease, K.D., "Error Analysis for a Guided Mars Landing", J. Astronautical Sciences, Vol. 39, No. 4, Oct-Dec, pp. 423-445, (1991)

25. Murrow, Harold N. and McFall, John C. Jr., “Summary of Experimental Results Obtained From The NASA Planetary Entry Parachute Program”, American Inst of Aeronautics and Astronautics, Aerodynamic Deceleration Systems Conference, El Centro, CA, Sep. 23-25, AIAA-1968-934, (1968)

26. Patel, S., Hackett, N.R., Jorgensen, D.S., “Qualification of the Guided Parafoil Air Delivery System – Light (GPADS-Light)”, AIAA Aerodynamic Decelerator Systems Technology Conference, 14th, San Francisco, CA, June 3-5, AIAA-1997-1493, (1997)

27. Powell, R.W., Striepe, S.A., Desai, P.N., Braun, R.D.,Brauer, G.L., Cornick, D.E., Olson, D.W., Petersen, F.M., Stevenson, R., "Volume II, Utilization Manual, Program To Optimize Simulated Trajectories (POST)", Version 5.2, NTRS 2005-08-25, NASA-CR-132690, (1997)

28. Spencer, D.A., Thurman, S.W., Peng, Chia-Yen, Kallemeyn, P.H., Blanchard, R.C., Braun, R.D., "Mars Pathfinder Atmospheric Entry Reconstruction", AAS 98-146, AAS/AIAA, (1998)

29. Trelea, I., “The particle swarm optimization algorithm: convergence analysis and parameter selection”, Information Processing Letters, Vol. 85, no. 6, pp. 317-325, March, (2003)

30. Tuckness, D.G., "Analysis of a Terminal Landing on Mars", J.Spacecraft and Rockets, Vol.32 No.1, pp.142-148, (1995)

31. Tyler, D., Barnes, J., “Development of the Oregon State University Mars MM5 and Description of our Initial Results”, Mars atmosphere modeling and observations Workshop, Granada, Spain, (2003), <http://www-mars.lmd.jussieu.fr/granada2003/abstract/tyler_model.pdf >

32. Walberg, G. and Birge, B., “Terminal Guidance Techniques for a Mars Precision Lander”, AIAA Space 2000 Conference & Exposition 19-21 September, AIAA 2000-5342, (2000)

APPENDICES

Appendix A - Particle Swarm Optimization (PSO)

Overview

Particle Swarm Optimization (PSO) can be considered a subset of Evolutionary Computation (EC) [20]. EC also includes Genetic Algorithms, Genetic Programming, Evolutionary Programming, and Evolutionary Strategies. Emergence can be paraphrased as 'complex behavior from simple rules'. The game of 'Life' has defined this idea but it is at the heart of the PSO algorithm also.

In a nutshell, PSO is a method used to optimize n-dimensional problems, requires no rigid first guess algorithms, explores the majority of problem space, has little problem with being stuck in local minima, and is both simple to code and simple to understand in it's most basic form.

PSO is a stochastic population based optimization strategy with a simple memory component. The method was invented through iterative simulation of simplified social models. The social aspect has implications for the cutting edge of current research but for engineering applications PSO can be considered as a pure optimization algorithm without the metaphors.

It was initially based on social interaction research among birds and then discovered to be useful for optimization. PSO conceptually works with any dimension problem and has been shown to work well at finding the optimum for single objective and multi-objective functions, nonlinear and linear. PSO is similar to Genetic Algorithms due to the stochastic population based nature but easier to implement with the same or better

convergence performance [17]. It is a good candidate for high performance cluster computing.

Having roots in both traditional Artificial Life research as well as Evolutionary Computation (a much more applied engineering area) it is an almost intuitive concept yet can be cast with just a couple simple mathematical statements. Early on, the algorithm was used with training neural network weights and has been shown to perform very well on traditional genetic algorithm test functions [20]. Other researcher's, most notably Maurice Clerc [10] have rigorously analyzed the algorithm's particle dynamics and treated it as a closed loop control system of its own.

PSO is particularly suited to training backpropagation neural networks with a goal towards good generalization performance.

PSO Algorithm Details

Particle Swarm Optimization (PSO) is an algorithm originally designed for the optimization of continuous nonlinear functions. PSO has been extended by various researchers over the years to include versions with memory, clustering and anti-clustering, discrete problem capability, distributed computing versions, and a whole host of variants and designs for specialty applications. At the algorithm's core is the basic concept of position and velocity. The original PSO algorithm (now referred to as the Common PSO) is presented here.

The problem space is seeded with a population of particles over the range of interest. Each particle's position in hyperspace represents a candidate problem solution.

For example, in 2D space each particle would have two positional components, x1 and x2. The initial seed can be random or fixed. The positions are given random velocities and evaluated for fitness via a cost function, cost = $f(x_1,x_2)$ in our 2D example. Each particle knows its own best position over all k iterations and the overall best position found by all the particles. Its trajectory over iterations is controlled by a balance between attraction to its own personal best and the global best.

For each particle at iteration k:

$$v_k = a\ v_{k-1} + b(p - x_{k-1})$$

$$x_k = x_{k-1} + v_k$$

where:

v_k = velocity at iteration k

x_k = position at iteration k

p = balance between personal best position (exploration) and global best position (social) with stochastic components.

a = momentum factor

b = attraction balance coefficient

Traditional optimization techniques often depend on gradient descent type of strategies as well as requiring good first guesses in which to converge. PSO does not require a good first guess (though it helps). All that is needed is the region of search which can be very large. Because of the momentum term combined with the spread of particles,

the algorithm is very resistant to local minima, it is unlikely to get stuck in a hyper dimensional trough.

In addition, it has been shown that the number of particles chosen for the swarm has little to do with the ability to solve the problem once you get past the minimum needed, which is a near constant [29]. The number of particles needed for a certain n-dimension problem does not scale linearly as n increases. By convention, originally from inspection, the number of particles for any dimension problem is generally set between 24 and 30. The error space does not appear to be searched any more efficiently for extremely large numbers of particles.

For the work in this dissertation the Trelea 'set 1' convention is used [29], where [a, b] is set as a constant to [0.6, 1.7].

A standard computational example to test Numerical Computation algorithms is the Schaffer f6 function. This can be visualized in three dimensional space as a still frame picture of a water drop falling into a puddle, with the associated waves dying out the farther from the center we go.

Mathematically, the function can be expressed as

$$f(x) = 0.5 + \frac{\sin^2 \sqrt{x^2 + y^2} - 0.5}{\left(1 + 0.001 \times (x^2 + y^2)\right)^2}$$

with the global minimum f(x,y) = 0 at x = 0 and y = 0 concurrently.

The f6 function is often used as a test function because it is difficult to impossible for traditional optimization techniques to solve. It has many local minima, a single global minimum at (0,0) and a set of global maxima all equal, at the first wave surrounding the minimum. PSO will find the minimum anywhere from 50 to 300 iterations and similarly for the maxima [1].

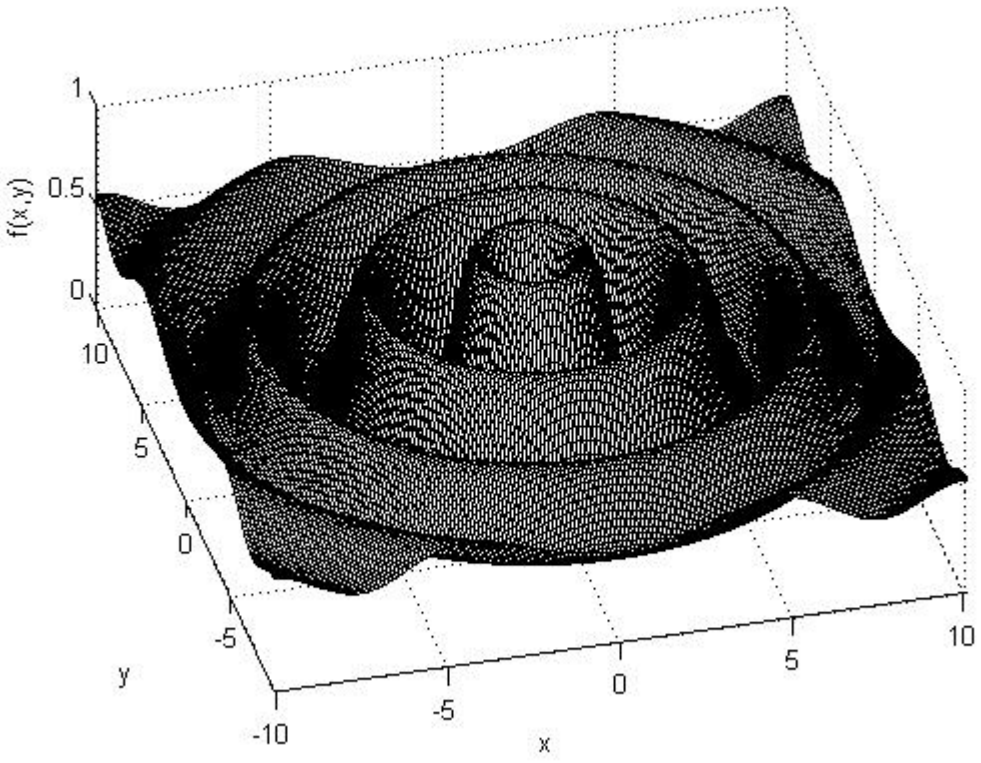

Figure 40. Schaffer f6 function

PSO Matlab Toolbox Usage

For this research, a PSO toolbox was developed in the Matlab format. It has several advantages over other canned routines. It is easy to use, freely available from Mathworks, and both comprehensive and robust.

It adds several features not in any other PSO implementation. It has boundary control, the user can specify how out of bounds positions are treated. The out of bounds particle can be allowed to extend beyond range (classic method). The advantage of the classic method is PSO will really explore unknown areas, making it unnecessary even to have a good idea of the search range of each dimension. The disadvantage is that the algorithm may never converge, especially if the velocity damping term (inertia) is set incorrectly. Secondly, the particle can 'saturate' or 'stick' to the boundary, eventually pulled back into the range by action of other particles. This is a good method when you want to guarantee convergence but you are not sure of the search range you should use. Thirdly, each particle can 'wrap' around hyperdimensionally to the other side of the range. This encourages exploration of the search range. Or, fourthly, as is used in all research presented here, the particle will bounce back with opposite velocity upon hitting a boundary. Again this encourages exploration within a particular search range and anecdotally seems to perform better than the wrap method. The last two methods are presented nowhere else but in this research to the author's knowledge.

The code is optimized for vector operations and there are sub-functions that will take advantage of parallelism. At one point in the dissertation research it was attempted to run the trajectory optimization in a parallel fashion and to this end both a parallel MPI (Message Passing Interface) PSO was developed as well as a parallel job manager for the overall trajectory problem. They were tested on the Air Force Research Laboratory's High Performance clusters at Maui. The PSO portion will also work with out of the box Matlab that has the Distributed Computing Toolbox installed.

The user has extensive control over the running of an optimization task. Number of particles, acceleration constants and inertia weights (for Common PSO), error goal, whether to seed with an initial position or set of particle positions or all random, various plotting and display options, iterations to train, minimization strategy, are all some of the parameters the user can adjust.

Additionally there are five PSO models included in the toolbox. The Common PSO with linearly decreasing inertia term, Trelea types 1 and 2 [29], Maurice Clerc's Constricted PSO type 1" [10], and the Hassan, Cohanim, Weck type [17] which is a variation on the Common PSO.

Not immediately accessible to the user but easily modified is the flag that controls adaptation to changing environments. Motivation for this capability stemmed from wanting to modify a set of neural net weights based on changing state space and unexpected input. This is a step improvement on the Carlisle & Dozier [6] method of using a sentry. For change detection in this PSO, the global best is polled every 5 iterations to see if the position has remained the same but the value has changed by at least some delta. If so, then a dynamically changing environment has been detected. When this happens, all the particles' personal bests are reset to equal current positions and their velocities are 'agitated'. The agitation portion is ongoing research but currently uses a strategy of multiplying the velocity of all particles by an order of magnitude. This acts to initiate a more aggressive exploration of the problem space. This method has been successful in tracking 4 test problems, an f6 function that changes position linearly in (x,y) space, an f6 function that circles out dynamically from the origin over iterations following the track of

a fermat spiral, the ‘bubbles’ f6 function which is 2 f6 functions offset in the error space and amplitude modulated out of phase from each other over iterations, and a similar scheme where the 2 f6 functions circled each other as they are amplitude modulated.

Lastly, the newest feature set to the toolbox includes built in neural net training, well integrated into Matlab’s neural net toolbox, and the PSO also now allows optimization of continuous, discrete, and hybrid functions.

The toolbox is freely available from the Mathworks website in the User Contributed Code section under Optimization. The full link as of this writing is http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=7506&objectType=file .

A short explanation follows on how to use the PSO toolbox. For this example the Schaffer f6 function traditional version will be optimized for the minimum. The range of the search will default to [-100,100] for each dimension. This illustrates robustness in the face of poor initial guesses. Looking at the graph below shows how flat the search topology is at the outer end of the range. Even so, there are many local minima. It should be noted that even with a specific initial seed of particles all clustered at the edges of the range, the solution is still found. However, for this problem, the typical uniform random over the whole range seed is used.

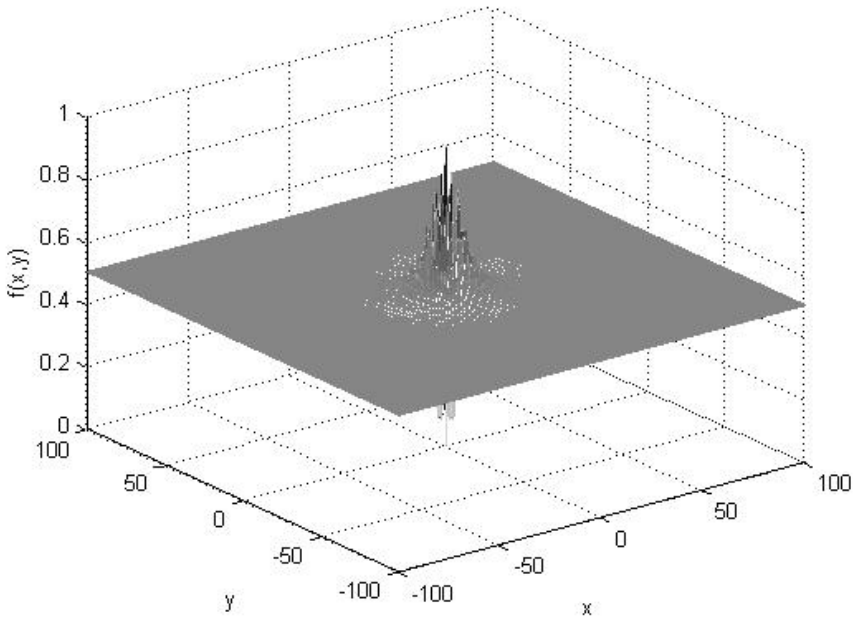

Figure 41. Schaffer f6 over the range [-100,100]

We already know the minimum value is f(x,y) = 0 for location x = 0 and y =0 concurrently. To use all default calling parameters (number of particles, minimization as opposed to maximization, a standard search range over [-100,100] for each dimension, etc), the optimization routine only needs to be called with the name of the function to optimize and the number of dimensions to search.

In this case 'f6' is the name of the Schaffer function as implemented in the toolbox and it has two dimensions to search over.

out = pso('f6',2)

This will provide output such as:

PSO: 1/2000 iterations, GBest = 0.49842388798423815.
PSO: 100/2000 iterations, GBest = 0.0006587693638556158.
PSO: 200/2000 iterations, GBest = 0.00052738785184652803.

```
PSO: 300/2000 iterations, GBest = 6.343611730352805e-008.
PSO: 400/2000 iterations, GBest = 8.90550416743352e-009.
PSO: 500/2000 iterations, GBest = 6.0507154842071031e-013.
PSO: 600/2000 iterations, GBest = 6.6613381477509392e-016.
PSO: 700/2000 iterations, GBest =                  0.
PSO: 800/2000 iterations, GBest =                  0.
PSO: 861/2000 iterations, GBest =                  0.

--> Solution likely, GBest hasn't changed by at least 1e-025 for 250 epochs.
out =
  1.0e-008 *
    0.1780
    0.0335
         0
```

The run took approximately 0.5 seconds to find the minimum The default graphical output will look like the following figure.

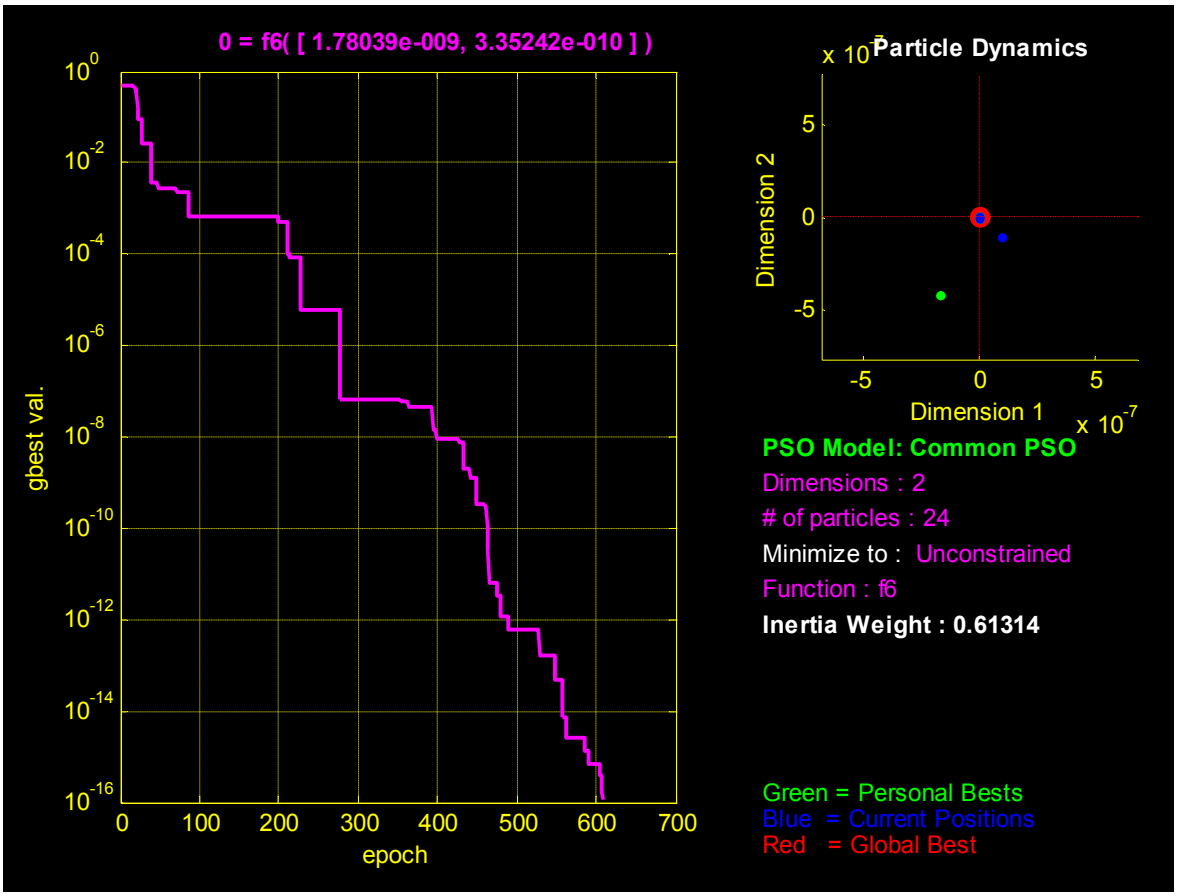

Figure 42. PSO applied to the Schaffer F6 function

Of course, the user can specify much more control over the optimization run. The code itself, in a following appendix has detailed comments on each user adjustable parameter. The code has been used for a wide variety of optimization problems. In various revisions it has been downloaded over 10,000 times and used by researchers across the world from the United States, to China, Southeast Asia, to the Middle East, Europe, and Australia. The toolbox constantly evolves with the pace of PSO research.

The toolbox also includes capability for interface into Matlab's Neural Network Toolbox. The following figure shows a toolbox demonstration run of a PSO trained ANN. The ANN is being trained to approximate a sine wave.

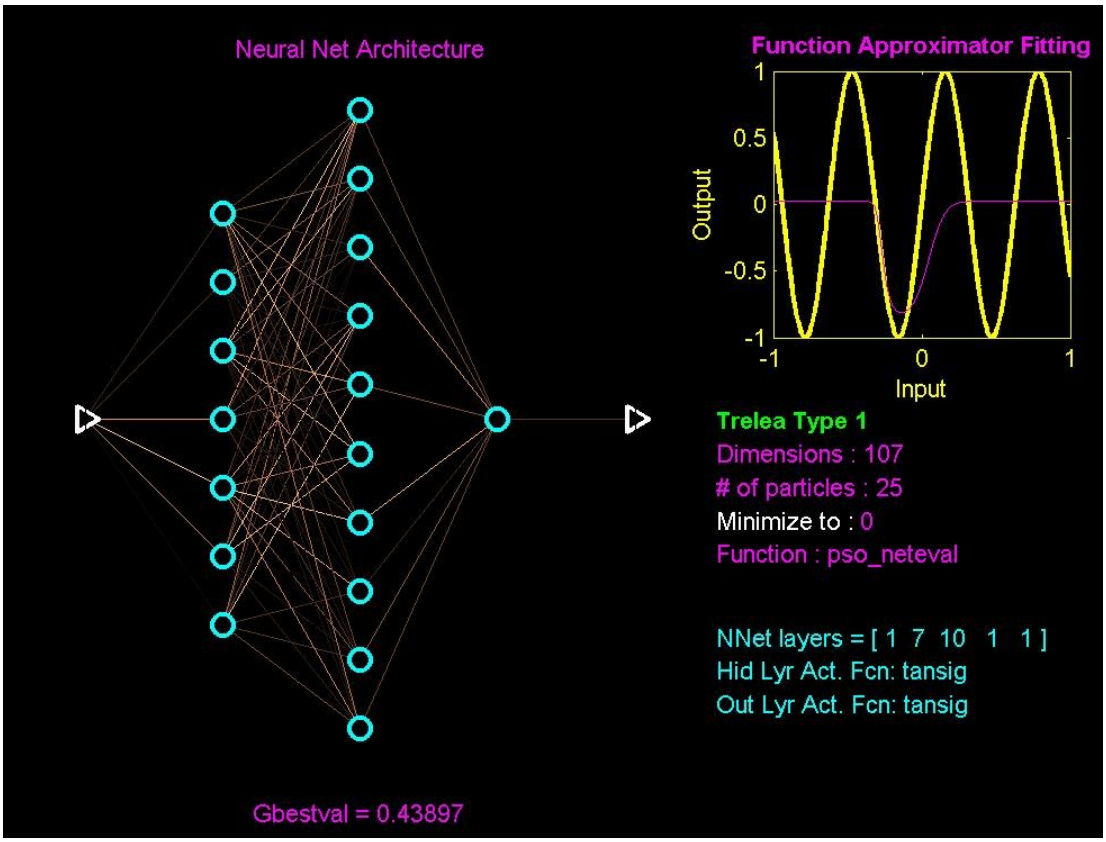

Figure 43. PSO trained Artificial Neural Network

Both directly as a result of research into PSO ability and indirectly from needs arising from the Mars Precision Landing Problem research new capability for tracking changing environments and hybridized continuous/discrete function optimization has been added to the toolbox. These new capabilities open up PSO to tackle more and varied problems such as target capture and online neural network updating.

Appendix B - MPL POST Optimization Input Deck

```
l$search
c*************************************************************************
c          runVal4.inp
c            used for batch runs (include file in event 25)
c            include file in first event also for init conditions (gdalt)
c            goes with the trnVal4.* and runVal4.* matlab progs/data
c
c    Mars Lander from LaRC hand-off to 2500 m
c    no optimization, bank on steerable chute controlled by matlab shell
c
c    Parachute diameter = 13.0 m
c
c    Marsgram atmosphere - Feb. 3, 2002; 00 hr, 00 min, 0.0 s
c    ln(pres) and atem input as tables
c    Marsgram winds*Braun multipliers input as tables
c
c    Parachute is lifting with variable steering events made by post_pso_fcn.m
c
c    Last modified: 01/22/04
c
c*************************************************************************
    ioflag    = 3,      / metric input, metric output
    ipro      = -1,     / trajectory print flag
    maxitr    = -1,      / maximum number of iteration
    opt       = 0,       / NO OPTIMIZATION
    srchm     = 0,       / projected gradient=4, no targeting=0
 $
c*************************************************************************
c     Trajectory Simulation Inputs
c*************************************************************************
l$gendat
    event       = 1,0.0,      / first event number (primary event)
    fesn        = 500,        / final event number
c - NUMERICAL INTEGRATION METHODS p. 6.a.15-1
    npc(2) =1,  / integration method (using RK) [flag]
    dltmax =1,  / max step size when using variable steps [s]
    dltmin =0.05,  / min step size  "     "      "       "   [s]
    dt=0.1,/ integration time step [s]
    kstpmx=5,/ max # of integration steps for each integration
    npinc=5,/ # of integration steps on each cycle
c*************************************************************************
c  Initial Event Conditions/Setup
```

```
c
c - INITIAL POSITION AND VELOCITY p. 6.a.12-1
    npc(3)=4,/ initial vel in planet rel coords
    npc(4)=2,/ initial pos in spherical coords
    npc(40)=3,/ reference plane for azimuth and FPA [flag] 3=?
c - ---- ---- ---- ---- ---- ----
c - initial conditions using nominal calculated
c - from mean of chutestatesSASnew.dat w/gdalt perturbed
c - holds gdalt, azvelr, gammar, gdlat, long, velr
*include 'runval4init.dat',
c
c - ---- ---- ---- ---- ---- ----
c
c - RANGE CALCULATIONS p. 6.a.19-1
    npc(12)=2,/ cross/down range option [flag]
    lonref   = 93.8023,   / developed by looking at averages of various lift
    latrefgd = -15.8384,  / chute non-optimized cases (showellipsebanks2.m)
c
c - PARACHUTE MODEL p. 6.a.28-1
    npc(32)=2,
    diamp(1)=0.0,/ init val of chute diam, unfurl to 13m
    drgpk(1)=1,
    idrgp(1)=0,
    parif(1)=70.0,
c
c - AERODYNAMIC INPUTS p. 6.a.1-1
    npc(8)=3,/aerodynamic coefficient [flag]
    sref=4.5238934,/aerodynamic reference area [m^2] (from M98)
c
c - AEROHEATING CALCULATIONS p. 6.a.2-1
    npc(15)=1,/ calculate aeroheating rate & tot. heat using Chapman
    npc(26)=0,/ no special aeroheating calculations
    rn= 0.6638,/ nose radius for Chapman heating (M98nom.inp)
c
c - ATMOSPHERE PARAMETERS p. 6.a.4-1
    npc(5)= 6,/ 2/3/02, 0 hr Marsgram atmosphere input as tables
    npc(6)= 2,/ Marsgram winds* Braun multipliers input as tables
    atmosk(1)=241.0,
    atmosk(2)=5.335e-03,
c
c - CONIC CALCULATION OPTION p. 6.a.5-1
    npc(1)= 3,/Keplerian conic option [flag]
    mre=1hu,/value of mean radius to be used [m](1hu = [re+rp]/2)
c
```

```
c - GRAVITY MODEL p. 6.a.10-1
    npc(16)=0,/ spherical or oblate model (oblate) [flag]
    j2=0.1958616e-02,/ spherical harmonics of gravity potential function
    j3=0.3144926e-04,
    j4=-0.1889437e-04,
    j5=0.2669248e-05,
    j6=-0.1340757e-05,
    j7=0.0d0,
    j8=0.0d0,
    re=3393940.0,/ equatorial radius [m]
    rp=3376780.0,/ polar radius [m]
    mu=4.28282868534e+13,/ gravitational constant (mars) [m^3/s^2]
    omega=7.088218e-05,/ rate of rotation of planet [rad/s]
    go=3.718,/ weight to mass factor (Mars surface)
c - VELOCITY LOSSES p. 6.a.25-1
    npc(25)=2,/ velocity loss calculation
c*************************************************************************
c    Initial Guidance Inputs
c*************************************************************************
    iguid(1)=0,0,1,/ atm.rel. aero angle guidance
    alppc(1)=0.0,/ initial alpha
    maxtim=2000.,/ maximum time
    altmax=550000.,/ maximum altitude
    altmin=-3000.0,/ minimum altitude
c*************************************************************************
c  Vehicle Model
c*************************************************************************
    wgtsg=2176.811,       / veh. wt. at parachute deploy, N (585.479 kg)
    wpropi=372.0,         / initial propellant weight, N (100 kg)
    npc(30)=0,            / enhanced (component) weight model
    npc(9)=1,             / rocket engine
    npc(27) = 1,          / integrate flow rate of specified engines
    npc(22)=2,            / input all four coef's in throttling parameter
    neng=1,               / 1 engine
    ispv(1)=553.9,553.9,  / Mars Isp (Earth Isp = 210 sec)
    iwdf(1)=2,            / flow rate = vac. thrust/ispv
    iwpf(1)= 0,
    iengmf(1)=0,          / engine off initially
    iengt(1)=0,           / fixed engine angles (in tables) w.r.t body
c*************************************************************************
c  Print Block
c - PRINT VARIABLE REQUESTS p. 6.a.16-1 --->>
    npvl=5,/ # of print variables per line [flag]
    pinc=2,/ print interval
```

```
    prnc=2,/ make binary profile for plotting
    prnca=2, / make ascii profile for plotting
    prnt(97)='diamp1','cdp1','diarp1','dragp1','lonref','latrefgd',
    prnt(103)='uw','vw','ww','totwv','ur','vr','wr','pstop',

c234567890123456789012345678901234567890123456789012345678901234567890123
4567890
    title=0h*runVal4.inp*,
 $
l$tblmlt
 vwum = 1.0,
 vwvm = 1.0,
 $
l$tab
 table = 'denkt',1,'gdalt',3,1,1,1,
 0.0,1.0,30000,1.0,130000,1.0,
 $
l$tab
 table = 'prest',1,'gdalt',6,1,1,1,
 0,6.52553,1859,6.36383,4217,6.15610,6387,5.96265,8515,5.77084,
 8816,5.74359,
 $
l$tab
 table = 'atemt',1,'gdalt',6,1,1,1,
 0,226.1419,1859,223.766,4217,220.502,6387,217.854,8515,215.292,
 8816,214.936,
 $
l$tab
 table = 'vwut',1,'gdalt',6,1,1,1,
 0,-0.1248,1859,0.02672,4217,0.19915,6387,0.55803,8515,0.86006,
 8816,0.85601,
 $
l$tab
 table = 'vwvt',1,'gdalt',6,1,1,1,
 0,0.200574,1859,0.42458,4217,0.68911,6387,1.5575,8515,2.2659,
 8816,2.2508,
 $
l$tab
 table = 'vwwt',0,0.0,
 $
l$tab
 table = 'tvc1t',0,2800.0, / engine #1 coef of thrust
 $
l$tab
```

```
 table = 'ae1t',0,0.2,     / engine #1 nozzle exit area
 $
l$tab
 table = 'pi1t',0,-90.0,   / engine #1 gimbal pitch angle (pointing 'down')
 $
l$tab
 table = 'yi1t',0,0.0,     / engine #1 gimbal yaw angle
 $
l$tab
 table='cdt',0,1.7,
 $
l$tab
 table='clt',0,0.0,
 $
l$tab
 table='cdp1t',0,0.41,
 endphs=1,
 $
c
c Parachute fully deployed
l$gendat
 event=22.,0.0, / primary event
 critr='diamp1',
 value=13.0,
 parif(1)=0.0,
 wgtsg = 1937.297,
 endphs=1,
 $
c - - - Convert parachute to 'airplane', simulates lifting/steerable chute
l$gendat
  event=25,0.0, / primary event
  critr='tdurp',
*include 'runval4fw.dat',  / matlab generated time to blow chute
  sref=132.73, / surface area of chute about 13m (pi*r^2)
  npc(32)=0,      / don't calculate parachute drag
  iguid(1)=0,     / aero-guidance
  iguid(2)=1,     / individual component steering
  iguid(6)=0,     / alpha carried over
  iguid(7)=0,     / beta carried over
  iguid(8)=0,     / bnkang carried over (this would be used for steering)
 $
l$tblmlt
 $
l$tab
```

```
 table='cdt',0,0.41,
 $
l$tab
 table='clt',0,.41, /lift coeff includes parachute, i.e. =1*0.41
 endphs = 1,
 $
cc
cc
cc
cc - steering events, time they happen and bank commanded are controls
*include 'runval4bc.dat', / this may be multiple steering events
cc
ccc start thrusting engine
c*include 'runval4tc.dat', / this may be multiple thrusting events
ccc
ccc
ccc
cc - marks 2500 meter mark (0m above surface)
l$gendat
 event=350,0.0, / primary event
 critr='gdalt',
 value = 2500,
 iengmf(1) =  0,0,
 iwpf(1) = 0,0,
 endphs=1,
 $
c**************************************************************************
c     This event terminates sim
c**************************************************************************
l$gendat
 event=500,0.0, / primary event
 critr='tdurp',
 value=0,
 endphs=1,
 endjob=1,
 endprb=1,
 $
```

Appendix C - Generic POST Input Deck used in Control System

```
% genericdeck_individualnets.m
% creates a POST3D input deck with a single event
%
% designed as a drop in block for the neural net controller, to provide the
% simulation environment for bankangled firewire/thrust case
%
% writes a file called gendeck.inp (text) in current directory
%
% usage: genericdeck(state,controls,[Dlong,Dlat],status_reset)
%
% state is a 7 element row vector:
%   {gdalt,azvelr,gammar,gdlat,long,velr,propellant weight}
% controls is a 10 element row vector:
%   {bnkang,bnkdot,alpha,alpdot,beta,betdot,eta,diamp,lift,thrust}
% where class is pattern classifier number
%       status is chute/eng on/off stuff
%
% Dlong,Dlat are optional and are desired lat/long, used for range calcs
% within post, default to

% Brian Birge
% Rev 2.0 - 2/21/07
% Rev 3.0 - 10/21/07 - version individualnets (misnomer)
%--------------------------------------------------------------------------
function out = genericdeck_individualnets(state,controls,varargin)
persistent set_status
if isempty(set_status)
   set_status=0;
end
%%
% error checking & input parse
% optional input for range calc
 if    nargin == 5
   Dlong    = varargin{2}(1);
   Dgdlat   = varargin{2}(2);
   int_time = varargin{1}; % integration time
   set_status = varargin{3}; % used to reset status between runs
 elseif nargin == 4
   Dlong    = varargin{2}(1);
   Dgdlat   = varargin{2}(2);
   int_time = varargin{1}; % integration time
 elseif nargin == 3
```

```
  Dlong    = 93.8023;
  Dgdlat   = -15.8384;
  int_time = varargin{1}; % integration time
 elseif nargin == 2
  Dlong    = 93.8023;
  Dgdlat   = -15.8384;
  int_time = 0.1; % integration time default to 10hz
 else
  error('Too many input args');
 end
% integration time = int_time

% current (initial) state ('real' values)
 Igdalt   = state(1);
 Iazvelr  = state(2);
 Igammar  = state(3);
 Igdlat   = state(4);
 Ilong    = state(5);
 Ivelr    = state(6);
 Ipropwt  = state(7);
 Itime    = state(8);

% current control flags (from neural controller)
 %  1) bankangle
 %  2) bnkdot
 %  3) alpha
 %  4) alpdot (usually set = 0)
 %  5) beta
 %  6) betdot (usually set = 0)
 %  7) eta (1st thrust throttle coeff)
 %  8) parachute diameter
 %  9) commanded drag coeff (used to determine when chute/engine sstates)
 % 10) thrust
 Cbnkang  = controls(1 );
 Cbnkdot  = controls(2 );
 Calpha   = controls(3 );
 Cbeta    = controls(4 );
 Ceta     = controls(5 );
 Cctfg    = controls(6 );
 Cclass   = controls(7 );
%-------------------------------------------------------------------------
%%
% write the file...
```

```
fid1=fopen('gendeck.inp','w');

% deck description goes here
fprintf(fid1,'l$search\n');
fprintf(fid1,'c*****************************************************************
***************\n');
fprintf(fid1,'c  gendeck.inp\n');
fprintf(fid1,'c      generic input deck with one event for interfacing with matlab \n');
fprintf(fid1,'c\n');

% don't change this
fprintf(fid1,'c*****************************************************************
***************\n');
fprintf(fid1,'     ioflag   = 3,  / metric input, metric output\n');
fprintf(fid1,'     ipro     = -1, / trajectory print flag\n');
fprintf(fid1,'     maxitr   = -1, / maximum number of iteration\n');
fprintf(fid1,'     opt      = 0,  / NO OPTIMIZATION\n');
fprintf(fid1,'     srchm    = 0,  / projected gradient=4, no targeting=0\n');
fprintf(fid1,' $\n');

% don't change this
fprintf(fid1,'c*****************************************************************
***************\n');
fprintf(fid1,'c     Trajectory Simulation Inputs\n');
fprintf(fid1,'c*****************************************************************
***************\n');
fprintf(fid1,'l$gendat\n');
fprintf(fid1,'     event  = 1,0.0, / first event number (primary event)\n');
fprintf(fid1,'     fesn   = 500,   / final event number\n');
fprintf(fid1,'c - NUMERICAL INTEGRATION METHODS p. 6.a.15-1\n');
fprintf(fid1,'     npc(2) = 1,     / integration method (using RK) [flag]\n');
fprintf(fid1,'     dltmax = .1,    / max step size when using variable steps [s]\n');
fprintf(fid1,'     dltmin = 0.0001,/ min step size  "     "      "       "   [s] \n');
fprintf(fid1,'     dt     = 0.001,   / integration time step [s]\n');
fprintf(fid1,'     kstpmx = 5,     / max # of integration steps for each integration\n');
fprintf(fid1,'     npinc  = 5,     / # of integration steps on each cycle\n');
fprintf(fid1,'c\n');

% these are loaded from matlab workspace
fprintf(fid1,'c*****************************************************************
***************\n');
fprintf(fid1,'c - INITIAL POSITION AND VELOCITY p. 6.a.12-1\n');
fprintf(fid1,'     npc(3)  = 4, / initial vel in planet rel coords\n');
fprintf(fid1,'     npc(4)  = 2, / initial pos in spherical coords\n');
```

```
fprintf(fid1,'     npc(40) = 3, / reference plane for azimuth and FPA [flag] 3=?\n');
fprintf(fid1,'c - ---- ---- ---- ---- ---- ----\n');
fprintf(fid1,'c - initial conditions using nominal calculated \n');
fprintf(fid1,'c - from mean of chutestatesSASnew.dat w/gdalt perturbed\n');
fprintf(fid1,'c - holds gdalt, azvelr, gammar, gdlat, long, velr\n');
fprintf(fid1,'     gdalt  = %12.7G,\n',Igdalt);
fprintf(fid1,'     azvelr = %12.7G,\n',Iazvelr);
fprintf(fid1,'     gammar = %12.7G,\n',Igammar);
fprintf(fid1,'     gdlat  = %12.7G,\n',Igdlat);
fprintf(fid1,'     long   = %12.7G,\n',Ilong);
fprintf(fid1,'     velr   = %12.7G,\n',Ivelr);
%fprintf(fid1,'     timref = %12.7G,\n',Itime);
fprintf(fid1,'     time   = %12.7G,\n',Itime);

% Dlong, and Dgdlat goes here
fprintf(fid1,'c************************************************************
**********\n');
fprintf(fid1,'c - RANGE CALCULATIONS p. 6.a.19-1\n');
fprintf(fid1,'     npc(12)  = 2,/ cross/down range option [flag]\n');
fprintf(fid1,'          lonref     = %12.7f, / developed by looking at averages of various
lift\n',Dlong);
fprintf(fid1,'                     latrefgd   =   %12.7f,   /   chute   non-optimized   cases
(showellipsebanks2.m)\n',Dgdlat);
fprintf(fid1,'c\n');


% parachute and drag calcs
 if Cctfg == 0 % freefall
    comment    = 'freefall';
    Cdiamp = 0;
    sref       = 4.5238934;
    wgtsg      = 2176.811;
    cdt        = 1.7;
    clt        = 0;
    cdp1t      = 0.41;
    npc32      = 0;
    parif      = 0;
    set_status = 0;

 elseif Cctfg == 1 % ballistic chute
    comment    = 'ballistic chute';
    Cdiamp = 13;
    wgtsg      = 1937.297;
    parif      = 0;
```

```
  npc32      = 2;
  sref       = 4.5239;
  cdt        = 1.7;
  clt        = 0;
  cdp1t      = 0.41;
  set_status = 1;

elseif Cctfg == 2       % parafoil
  comment    = 'parafoil steering';
  Cdiamp = 0;
  sref       = 132.73; % 13 meters diameter (same as for ballistic)
  wgtsg      = 1937.297;
  cdt        = 0.41;
  clt        = 0.41;
  cdp1t      = 0.0;
  npc32      = 0;
  parif      = 0;
  set_status = 2;

elseif Cctfg == 3
  comment    = 'engine on and/or jettisoned parafoil';
  Cdiamp = 0;
  sref       = 4.5238934;
  wgtsg      = 1660.595; % includes jettisoned parafoil
  cdt        = 2;
  clt        = 0;
  cdp1t      = 0.41;
  npc32      = 0;
  parif      = 0;
  set_status = 3;

else
  comment    = 'unknown state, vals set to freefall';
  Cdiamp = 0;
  sref       = 4.5238934;
  wgtsg      = 2176.811;
  cdt        = 1.7;
  clt        = 0;
  cdp1t      = 0.41;
  npc32      = 0;
  parif      = 0;
  set_status = 0;

end
```

```
disp(['class = ',num2str(Cclass),', set_status = ',num2str(set_status),', ',comment]);
%disp([num2str(Cdiamp),' ',num2str(Cthrust),' ',num2str(Clift)]);
disp('**********************************************************************
**********');



fprintf(fid1,'c**************************************************************
*****************\n');
 fprintf(fid1,'c - PARACHUTE MODEL p. 6.a.28-1\n');
 fprintf(fid1,'     npc(32)  = %d, / chute drag flag\n',round(npc32));
 fprintf(fid1,'     diamp(1) = %12.7G, / init val of chute diam, unfurl to 13m\n',Cdiamp);
 fprintf(fid1,'     drgpk(1) = 1,\n');
 fprintf(fid1,'     idrgp(1) = 0,\n');
 fprintf(fid1,'c %s\n',comment);
 fprintf(fid1,'     sref     = %12.7G, / surface area of chute about 13m (pi*r^2)\n',sref);
 fprintf(fid1,'     parif(1) = %12.7G,\n',parif);
 fprintf(fid1,'          wgtsg       = %12.7G, / veh. wt. at parachute deploy, N (585.479
kg)\n',wgtsg);

fprintf(fid1,'c\n');

%%
% don't change
fprintf(fid1,'c**************************************************************
*****************\n');
fprintf(fid1,'c - AERODYNAMIC INPUTS p. 6.a.1-1\n');
fprintf(fid1,'     npc(8) = 3,         /aerodynamic coefficient [flag]\n');
fprintf(fid1,'c\n');
%%
% don't change
fprintf(fid1,'c**************************************************************
*****************\n');
fprintf(fid1,'c - AEROHEATING CALCULATIONS p. 6.a.2-1\n');
fprintf(fid1,'     npc(15) = 1,     / calculate aeroheating rate & tot. heat using Chapman\n');
fprintf(fid1,'     npc(26) = 0,     / no special aeroheating calculations\n');
fprintf(fid1,'     rn      = 0.6638,/ nose radius for Chapman heating (M98nom.inp)\n');
fprintf(fid1,'c\n');
%%
% don't change
fprintf(fid1,'c**************************************************************
*****************\n');
fprintf(fid1,'c - ATMOSPHERE PARAMETERS p. 6.a.4-1\n');
fprintf(fid1,'     npc(5)    = 6,    / 2/3/02, 0 hr Marsgram atmosphere input as tables\n');
```

```
fprintf(fid1,'    npc(6)    = 2,    / Marsgram winds* Braun multipliers input as tables\n');
fprintf(fid1,'    atmosk(1) = 241.0,\n');
fprintf(fid1,'    atmosk(2) = 5.335e-03,\n');
fprintf(fid1,'c\n');
%%
% don't change
fprintf(fid1,'c************************************************************
**********\n');
fprintf(fid1,'c - CONIC CALCULATION OPTION p. 6.a.5-1\n');
fprintf(fid1,'    npc(1) = 3,   / Keplerian conic option [flag]\n');
fprintf(fid1,'    mre    = 1hu, / value of mean radius to be used [m](1hu = [re+rp]/2)\n');
fprintf(fid1,'c\n');
%%
% don't change
fprintf(fid1,'c************************************************************
**********\n');
fprintf(fid1,'c - GRAVITY MODEL p. 6.a.10-1\n');
fprintf(fid1,'    npc(16) =  0,              / spherical or oblate model (oblate) [flag]\n');
fprintf(fid1,'    j2      =  0.1958616e-02,   / spherical harmonics of gravity potl funct\n');
fprintf(fid1,'    j3      =  0.3144926e-04,\n');
fprintf(fid1,'    j4      = -0.1889437e-04,\n');
fprintf(fid1,'    j5      =  0.2669248e-05,\n');
fprintf(fid1,'    j6      = -0.1340757e-05,\n');
fprintf(fid1,'    j7      =  0.0d0,\n');
fprintf(fid1,'    j8      =  0.0d0,\n');
fprintf(fid1,'    re      = 3393940.0,        / equatorial radius [m]\n');
fprintf(fid1,'    rp      = 3376780.0,        / polar radius [m]\n');
fprintf(fid1,'    mu      = 4.28282868534e+13, / gravitational constant (mars) [m^3/s^2]\n');
fprintf(fid1,'    omega   = 7.088218e-05,     / rate of rotation of planet [rad/s]\n');
fprintf(fid1,'    go      = 3.718,            / weight to mass factor (Mars surface)\n');
fprintf(fid1,'c\n');
%%
% don't change
fprintf(fid1,'c************************************************************
**********\n');
fprintf(fid1,'c - VELOCITY LOSSES p. 6.a.25-1\n');
fprintf(fid1,'    npc(25) = 2,               / velocity loss calculation\n');
fprintf(fid1,'c\n');


%%
% Guidance goes here
fprintf(fid1,'c************************************************************
**********\n');
```

```
fprintf(fid1,'c    Initial Guidance Inputs\n');
fprintf(fid1,'c*****************************************************************
*****************\n');
fprintf(fid1,'     iguid(1) =  0,0,1,       / atm.rel. aero angle guidance\n');
fprintf(fid1,'     iguid(6) =  1,1,1,0,0,0, / aoa,ss,bnk input deg\n');
fprintf(fid1,'     bnkpc(1) =  %12.7G,  / bankangle\n',...
                    Cbnkang);
fprintf(fid1,'     bnkpc(2) =  %12.7G,  / bankangle vel\n',...
                    Cbnkdot);
fprintf(fid1,'     alppc(1) =  %12.7G,  / alpha (angle of attack)\n',...
                    Calpha);
fprintf(fid1,'     alppc(2) =  %12.7G,  / alpha (angle of attack) vel\n',...
                    0);
fprintf(fid1,'     betpc(1) =  %12.7G,  / beta (sideslip)\n',...
                    Cbeta);
fprintf(fid1,'     betpc(2) =  %12.7G,  / beta (sideslip) vel\n',...
                    0);
fprintf(fid1,'     maxtim   =  2000.,   / maximum time\n');
fprintf(fid1,'     altmax   =  550000., / maximum altitude\n');
fprintf(fid1,'     altmin   =  -3000.,  / minimum altitude     \n');
fprintf(fid1,'c\n');
%%
% put in weight of propellant here (wprop) and engine on/off commands
fprintf(fid1,'c*****************************************************************
*****************\n');
fprintf(fid1,'c  Vehicle Model\n');
fprintf(fid1,'c*****************************************************************
*****************\n');
fprintf(fid1,'     wpropi    = %12.7G,      / initial propellant weight, N (100 kg)\n',Ipropwt);
fprintf(fid1,'     npc(30)   = 0,           / enhanced (component) weight model\n');
fprintf(fid1,'     npc(9)    = 1,           / rocket engine\n');
fprintf(fid1,'     npc(27)   = 1,           / integrate flow rate of specified engines\n');
fprintf(fid1,'     npc(22)   = 2,           / input all four coef''s in throttling parameter\n');
fprintf(fid1,'     neng      = 1,           / 1 engine\n');
fprintf(fid1,'     ispv(1)   = 553.9,553.9, / Mars Isp (Earth Isp = 210 sec)\n');
fprintf(fid1,'     iwdf(1)   = 2,           / flow rate = vac. thrust/ispv\n');
fprintf(fid1,'     iengt(1)  = 0,           / fixed engine angles (in tables) w.r.t body\n');
if Cctfg == 3
  fprintf(fid1,'     iengmf(1) = 1,      / engine on\n');
  fprintf(fid1,'     iwpf(1)   = 1,\n');
  fprintf(fid1,'     etapc(1)  = %12.7G, / throttle engine (eta)\n',Ceta);
else
  fprintf(fid1,'     iengmf(1) = 0,      / engine off\n');
  fprintf(fid1,'     iwpf(1)   = 0,\n');
```

```
end
fprintf(fid1,'c\n');

%%
% don't change this
fprintf(fid1,'c**************************************************************
**********\n');
fprintf(fid1,'c  Print Block\n');
fprintf(fid1,'c - PRINT VARIABLE REQUESTS p. 6.a.16-1 --->>\n');
fprintf(fid1,'     npvl      = 5, / # of print variables per line [flag]\n');
fprintf(fid1,'     pinc      = 2, / print interval\n');
fprintf(fid1,'     prnc      = 0, / make binary profile for plotting\n');
fprintf(fid1,'     prnca     = 0, / make ascii profile for plotting\n');
fprintf(fid1,'     prnt(97)  = "diamp1","cdp1","diarp1","dragp1","lonref","latrefgd",\n');
fprintf(fid1,'     prnt(103) = "uw","vw","ww","totwv","ur","vr","wr","ttime",\n');
fprintf(fid1,'     prnt(111) = "cd","sref","pstop",\n');
fprintf(fid1,'c\n');
%%
% tables
fprintf(fid1,'     title=0h*gendeck.inp*,\n');
fprintf(fid1,' $\n');
fprintf(fid1,'l$tblmlt\n');
fprintf(fid1,' vwum = 1.0,\n');
fprintf(fid1,' vwvm = 1.0,\n');
fprintf(fid1,' $\n');
fprintf(fid1,'l$tab\n');
fprintf(fid1,' table = "denkt",1,"gdalt",3,1,1,1,\n');
fprintf(fid1,' 0.0,1.0,30000,1.0,130000,1.0,\n');
fprintf(fid1,' $\n');
fprintf(fid1,'l$tab\n');
fprintf(fid1,' table = "prest",1,"gdalt",6,1,1,1,\n');
fprintf(fid1,' 0,6.52553,1859,6.36383,4217,6.15610,6387,5.96265,8515,5.77084,\n');
fprintf(fid1,' 8816,5.74359,\n');
fprintf(fid1,' $\n');
fprintf(fid1,'l$tab\n');
fprintf(fid1,' table = "atemt",1,"gdalt",6,1,1,1,\n');
fprintf(fid1,' 0,226.1419,1859,223.766,4217,220.502,6387,217.854,8515,215.292,\n');
fprintf(fid1,' 8816,214.936,\n');
fprintf(fid1,' $\n');
fprintf(fid1,'l$tab\n');
fprintf(fid1,' table = "vwut",1,"gdalt",6,1,1,1,\n');
fprintf(fid1,' 0,-0.1248,1859,0.02672,4217,0.19915,6387,0.55803,8515,0.86006,\n');
fprintf(fid1,' 8816,0.85601,\n');
fprintf(fid1,' $\n');
```

```
fprintf(fid1,'l$tab\n');
fprintf(fid1,' table = "vwvt",1,"gdalt",6,1,1,1,\n');
fprintf(fid1,' 0,0.200574,1859,0.42458,4217,0.68911,6387,1.5575,8515,2.2659,\n');
fprintf(fid1,' 8816,2.2508,\n');
fprintf(fid1,' $\n');
fprintf(fid1,'l$tab\n');
fprintf(fid1,' table = "vwwt",0,0.0,\n');
fprintf(fid1,' $\n');
fprintf(fid1,'l$tab\n');
fprintf(fid1,' table = "tvc1t",0,2800.0, / engine #1 coef of thrust\n');
fprintf(fid1,' $\n');
fprintf(fid1,'l$tab\n');
fprintf(fid1,' table = "ae1t",0,0.2,     / engine #1 nozzle exit area\n');
fprintf(fid1,' $\n');
fprintf(fid1,'l$tab\n');
fprintf(fid1,' table = "pi1t",0,180.0,   / engine #1 gimbal pitch angle (pointing "down")\n');
fprintf(fid1,' $\n');
fprintf(fid1,'l$tab\n');
fprintf(fid1,' table = "yi1t",0,0.0,     / engine #1 gimbal yaw angle\n');
fprintf(fid1,' $\n');

% vehicle drag, vehicle lift, and parachute drag coefficients
fprintf(fid1,'l$tab  \n');
fprintf(fid1,' table="cdt",0,%12.7G,\n',cdt);
fprintf(fid1,' $\n');
fprintf(fid1,'l$tab  \n');
fprintf(fid1,' table="clt",0,%12.7G,\n',clt);
fprintf(fid1,' $\n');
fprintf(fid1,'l$tab\n');
fprintf(fid1,' table="cdp1t",0,%12.7G,\n',cdp1t);
fprintf(fid1,' endphs=1,\n');
fprintf(fid1,' $\n');
fprintf(fid1,'c\n');
% end it all
fprintf(fid1,'c**********************************************************
******************\n');
fprintf(fid1,'l$gendat\n');
fprintf(fid1,' event  = 400,0.0, / primary event\n');
fprintf(fid1,' critr  = "tdurp",\n');
fprintf(fid1,' value  = %12.7G,\n',int_time);  % user input control, poll matlab every 1/10
sec
fprintf(fid1,' endphs = 1,\n');
fprintf(fid1,' $\n');
```

```
fprintf(fid1,'c*****************************************************************
*****************\n');
fprintf(fid1,'c     This event terminates sim      \n');
fprintf(fid1,'c*****************************************************************
*****************\n');
fprintf(fid1,'l$gendat\n');
fprintf(fid1,' event  = 500,0.0, / primary event\n');
fprintf(fid1,' critr  = "tdurp",\n');
fprintf(fid1,' value  = 0.0,\n');
fprintf(fid1,' endphs = 1,\n');
fprintf(fid1,' endjob = 1,\n');
fprintf(fid1,' endprb = 1,\n');
fprintf(fid1,' $\n');

fclose(fid1);
out = set_status;
return
```

Appendix D - Cost Function used in Trajectory Optimization Runs

```
% runVal4.m
% runs runVal4.inp, saves the data
%
% getinitcond.mat must be loaded into the workspace for this function to
% work
%
% usage: out=runval4(in)
%      in(1) = time to firewire
%      in(2),in(3) - in(end-1),in(end) = tdurp,bnk pairs, each pair
%                        is one event, can be as many events as you want
%      out = scalar value of cost function

% Brian Birge
% 1/22/04
% Rev 3.1
function [out]=runVal4_vectorized(in)
global Dgdalt Dlong Dgdlat fw_min fw_max t_min t_max bnk_min bnk_max
global bnkrt_min bnkrt_max thr_min thr_max thr_bnk_min thr_bnk_max
global thr_pit_min thr_pit_max thr_yaw_min thr_yaw_max

% set of 76 initial conditions to optimize
 Igdalt  = evalin('base','Igdalt;');
 Iazvelr = evalin('base','Iazvelr;');
 Igammar = evalin('base','Igammar;');

 Igdlat  = evalin('base','Igdlat;');
 Ilong   = evalin('base','Ilong;');
 Ivelr   = evalin('base','Ivelr;');

% desired end conditions
 Dgdalt  = evalin('base','Dgdalt');
 Dlong   = evalin('base','Dlong');
 Dgdlat  = evalin('base','Dgdlat');
 Dur     = evalin('base','Dur');
 Dvr     = evalin('base','Dvr');
 Dwr     = evalin('base','Dwr');

 rmars=(3393940+3376780/2); %meters, radius of mars average
%-------------------------------------------------------------------------
%
 for jj=1:length(in(:,1)) % # of particles loop
 deckparams = [Igdalt,Iazvelr,Igammar,Igdlat,Ilong,Ivelr,in(jj,:)];
```

```
%if length(deckparams) == 34
    post_pso_fcn3x3_usethis(deckparams);
%else
%    post_pso_fcn6x3(deckparams);
%end

 % this sometimes bombs I think because the file is created yet
 % hence the pause
 try
  runpost('runval4b'); % has 2 vectored thrusting events at end
 catch
   try
    pause(1)
    runpost('runval4b');
   catch
    pause(1)
    runpost('runval4b');
   end
 end

 %runpost('runval4'); % no thrusting, bank angle gliding parachute only
%-------------------------------------------------------------------------
%%
 % this cost function:
 % 1) minimizes delta between gdalt(end), long(end), gdlat(end), and target
 % 2) minimizes delta between ur(end), vr(end), wr(end), and target
  gdalt  = evalin('base','gdalt');
  wprop  = evalin('base','wprop');
  long   = evalin('base','long');
  gdlat  = evalin('base','gdlat');
  ur     = evalin('base','ur');
  vr     = evalin('base','vr');
  wr     = evalin('base','wr');
  azvelr = evalin('base','azvelr');
  gammar = evalin('base','gammar');
  time   = evalin('base','time');
  thrust = evalin('base','thrust');

  bnkang = evalin('base','bnkang');
  alpha  = evalin('base','alpha');
  beta   = evalin('base','beta');
  eta    = evalin('base','eta');
```

```
 lenny = length(gdalt);
 lenny = floor(lenny/6);

%-------------------------------------------------------------------------
%%
 % this module takes care of when angle rates cause the angles to
 % go over limits (thr_max is really eta_max, not thrust)
 if   max(bnkang) > bnk_max || min(bnkang) < bnk_min
  out(jj,1) = 1e99;
  disp('Constraints violated: bnkang');
 elseif max(eta)  > thr_max || min(eta)    < thr_min
  out(jj,1) = 1e99;
  disp('Constraints violated: eta');
 else

 %-------------------------------------------------------------------------
 %%
  cost_alt   = sum((Dgdalt*ones(lenny,1)-gdalt(end-lenny+1:end)).^2);
 %-------------------------------------------------------------------------
 %%
  cost_long  = sum(d2m((Dlong*ones(lenny,1)-...
                    long(end-lenny+1:end)),rmars).^2);
 %-------------------------------------------------------------------------
 %%
  cost_gdlat = sum(d2m((Dgdlat*ones(lenny,1)-...
                    gdlat(end-lenny+1:end)),rmars).^2);
 %-------------------------------------------------------------------------
 %%
  % distance to target at end of run
  cost_endalt = (Dgdalt-gdalt(end));
  if cost_endalt<0
     cost_endalt = cost_endalt*1e6;
  end
  cost_endalt   = cost_endalt^2;
  cost_endgdlat = d2m(Dgdlat-gdlat(end),rmars).^2;
  cost_endlong  = d2m(Dlong-long(end),rmars).^2;

  cost_dist     = sqrt(cost_endalt + cost_endgdlat + cost_endlong);
 %-------------------------------------------------------------------------
 %%
 % distance to target at engine start
 % engine starts 1000m above surface
  % find index when engines start
   tmpind=find(thrust>0);
```

```
  if length(tmpind)>0
    engstartind=tmpind(1);
    cost_enginegdalt = (Dgdalt+1000-gdalt(engstartind)).^2;
    cost_enginegdlat = (Dgdlat-gdlat(engstartind)).^2;
    cost_enginelong  = (Dlong-long(engstartind)).^2;

    %cost_distengine = ...
    %    sqrt(cost_enginegdalt + cost_enginegdlat + cost_enginelong);

    cost_ur_engine = (Dur-ur(engstartind)).^2;
    cost_vr_engine = (Dvr-vr(engstartind)).^2;
    cost_wr_engine = (Dwr-wr(engstartind)).^2;
  else
    cost_enginegdalt = 1e9;
    cost_enginegdlat = 1e9;
    cost_enginelong  = 1e9;

    %cost_distengine = 1e9; % penalty for never turning on engine

    cost_ur_engine = 1e9;
    cost_vr_engine = 1e9;
    cost_wr_engine = 1e9;

  end
%-------------------------------------------------------------------------
%%
% cost associated with ending velocity
 cost_ur=(Dur-ur(end)).^2;
 cost_vr=(Dvr-vr(end)).^2;
 cost_wr=(Dwr-wr(end)).^2;

 speed = sqrt(cost_ur + cost_vr + cost_wr);
 cost_speed = speed;
%-------------------------------------------------------------------------
%%
 % cost associated with matching a straight line from initial condition to target
 % 1st build a straight line of same length as tdurp from initial lat,long,alt to target

 % 2nd take difference at each time step (should this be based on
 % velocity?)

 % 3rd sum up for cost term
  cost_line=0;
%-------------------------------------------------------------------------
```

```
%%
 % cost associated with making sure we are always heading in the right
 % direction (azvelr stuff)
  % 1st find out what desired heading should be at each time instant
   east=d2m(long-Dlong*ones(size(long)),rmars);
   north=d2m(gdlat-Dgdlat*ones(size(gdlat)),rmars);
   dist=sqrt((east.^2)+(north.^2));
   ang=r2d(acos(((east.^2)+(dist.^2)-(north.^2))./(2*east.*dist)));
   clear deshdg
   for i=1:length(azvelr)
      if long(i)<Dlong && gdlat(i)>Dgdlat
         deshdg(i,1)=90+ang(i);
      elseif long(i)<Dlong && gdlat(i)<Dgdlat
         deshdg(i,1)=-ang(i);
      elseif long(i)>Dlong && gdlat(i)>Dgdlat
         deshdg(i,1)=180+ang(i);
      elseif long(i)>Dlong && gdlat(i)<Dgdlat
         deshdg(i,1)=270+ang(i);
      elseif long(i)==Dlong && gdlat(i)>Dgdlat
         deshdg(i,1)=180;
      elseif long(i)==Dlong && gdlat(i)<Dgdlat
         deshdg(i,1)=0;
      elseif long(i)<Dlong && gdlat(i)==Dgdlat
         deshdg(i,1)=90;
      elseif long(i)>Dlong && gdlat(i)==Dgdlat
         deshdg(i,1)=270;
      else
         deshdg(i,1)=azvelr(i);
      end
   end
  % 2nd find out difference between desired heading and actual
  %[size(azvelr),size(deshdg)]

   sqr_err=(azvelr-deshdg).^2;
  % 3rd sum 'em up for a cost term
   %cost_head = 0;
   cost_head = sqrt(sum(sqr_err));
%-------------------------------------------------------------------------
%%
% cost associated with making sure we are pitched correctly (gammar stuff)
% basically a continuation of above but including altitude now
% if travelling away (determined by diff(grnd) ) then gammar should be
% positive, if travelling towards target then gammar should be negative, if
% perpendicular then gammar should be 0 deg for level holding flight
```

```
 grnd     = dist;
 dgrndsgn = [0;sign(diff(grnd))];
 totdist  = sqrt(grnd.^2);
 desgam   = r2d(acos( ((grnd.^2)+(totdist.^2)-(gdalt.^2))./(2.*grnd.*totdist) ));
 desgam   = desgam.*dgrndsgn;  % applying correct sign to desired gammar
 sqr_err  = (gammar-desgam).^2;
 %cost_gammar=0;
 cost_gammar = sqrt(sum(sqr_err));
%-------------------------------------------------------------------------
%%
% cost associated with always getting closer to target than previous time step
 %cost_delta_alt=diff(((Dgdalt.*ones(size(long)))-(gdalt)).^2);
 cost_delta_gdlat=diff(d2m(((Dgdlat.*ones(size(long)))-gdlat).^2,rmars));
 cost_delta_long=diff(d2m(((Dlong.*ones(size(long)))-long).^2,rmars));

% cost_delta = sum((cost_delta_alt  >0).*cost_delta_alt...
%               +(cost_delta_gdlat>0).*cost_delta_gdlat...
%               +(cost_delta_long >0).*cost_delta_long);
cost_delta = sum( (cost_delta_gdlat>0).*cost_delta_gdlat...
             +(cost_delta_long >0).*cost_delta_long);
%-------------------------------------------------------------------------
%%
% cost associated with maximizing glide time
 cost_glide   = (1/time(end));
%-------------------------------------------------------------------------
%%
% cost associated with minimizing altitude drop speed
 cost_altdiff = max(diff(gdalt));
%-------------------------------------------------------------------------
%%
% cost associated with achieving propellant weight goals
 wpropgoal = 0;
 %if wprop(end)<0
 %    cost_wprop= 1e9;
 %else
    %cost_wprop = (wprop(end)-wpropgoal);
    cost_wprop = 0;
    if wprop(end)<0
       cost_wprop = -wprop(end)*1e99;
       %cost_wprop = 1e20;
    end
 %end
%-------------------------------------------------------------------------
%%
```

```
  % switch emphasis to slowing down when we are with
  % 10 meters of target, otherwise emphasis is on distance
%   if dist(end)<10
%       distweight  = 250;
%       speedweight = 1000;
%   else
      distweight  = 1000;
      speedweight = 250;
%   end
 %-------------------------------------------------------------------------
 %%
 % cost terms (1st column) and their weights (2nd column)
    cost_terms=[...
        cost_endalt,         10;...
        cost_endgdlat,       10;...
        cost_endlong,        10;...
        cost_alt,            0;...
        cost_long,           0;...
        cost_gdlat,          0;...
        cost_ur,             10;...
        cost_vr,             10;...
        cost_wr,             30;...
        cost_line,           0;...
        cost_head,           0;...
        cost_gammar,         0;...
        cost_glide,          0;...
        cost_delta,          0;...
        cost_altdiff,        0;...
        cost_dist,         distweight;...
        cost_speed,        speedweight;...
        cost_enginegdalt,    10;...
        cost_enginegdlat,    10;...
        cost_enginelong,     10;...
        cost_ur_engine,      0;...
        cost_vr_engine,      0;...
        cost_wr_engine,      0;...
        cost_wprop,          10;...
    ];
  costfcn=(sum(cost_terms(:,1).*cost_terms(:,2)));

 %  cost_terms = [cost_dist(end),1;cost_speed(end),1]; % simple speed/dist criterion

 %-------------------------------------------------------------------------
 %%
```

```
% if we hit physical goal, that's good enough (10m or less to target)
% also, provide some convexity to draw other particles to good region
 dist=sqrt(sum([cost_endalt,cost_endgdlat,cost_endlong]));

  if dist(end)<=1 && speed(end)<=1 && wprop(end)>=0
     costfcn = 0;
     %disp('END WITHIN 1m of Target');
  elseif dist(end)<=10 && speed(end)<=5 && wprop(end)>=0
     costfcn = costfcn/10;
     disp('END WITHIN 10m of Target');
  elseif dist(end)<=1000
 %    costfcn=costfcn/3;
     %disp('End within 1km of target');
  elseif dist(end)<=5000
%     costfcn=costfcn/1.5;
     %disp('End within 5km of target');
  elseif dist(end)>dist(1)
     costfcn=costfcn*1e6; % big penalty for heading wrong direction
     disp(['Got lost! End dist ',num2str(dist(end)),' > Start dist ',...
        num2str(dist(1))]);
  end
  if min(wprop)<0 % big penalty for running out of fuel
     costfcn=costfcn*1e15;
     disp([' Out of Fuel, wprop(end) = ',num2str(wprop(end))]);
  end
  if min(gdalt)<Dgdalt-.1   % big penalty for hitting ground
     costfcn=1e55*costfcn;
     disp(['Crash, min(gdalt) = ',num2str(min(gdalt)),' ( < ',...
        num2str(Dgdalt),' )']);
  end

%--------------------------------------------------------------------------

  out(jj,1) = costfcn;

  clear cost_endalt cost_endgdlat cost_endlong cost_alt cost_long cost_gdlat
  clear cost_ur cost_vr cost_wr cost_line cost_head cost_gammar cost_glide
  clear cost_delta cost_altdiff cost_dist cost_speed cost_enginegdalt
  clear cost_enginegdlat cost_enginelong cost_ur_engine cost_vr_engine
  clear cost_wr_engine cost_wprop

 end % end if constraints violated
end % end jj (particle loop)
return
```

Appendix E - Routine to Build Partial POST input decks

```
% post_pso_fcn3x3_usethis.m
% different from post_pso_fcn3x3 in that it doesn't use thrust rate
%
% function builds up input deck for use with runval4.m and pso.m
% usage:   post_pso_fcn(in)
%  'in' is a single vector of:
%      in(1)-in(6) = initial conditions: gdalt, azvelr, gammar, gdlat,
%                                    long, velr
%      in(7) = tdurp when ballistic chute is turned into lifting chute
%              (firewire)
%      in(8)-in(end) = interleaved data pts of tdurp, bnkpc(1:2),
%      etapc(1:2)
function post_pso_fcn3x3(in)
  global Dgdalt Dlong Dgdlat Dur Dvr Dwr

 Igdalt  = in(1);
 Iazvelr = in(2);
 Igammar = in(3);
 Igdlat  = in(4);
 Ilong   = in(5);
 Ivelr   = in(6);

 % this is the section that needs a certain format for each deck see
 % batchval4.m
 indx_offset = 6;
 tdurp_fire = in(1+indx_offset);       % these index #'s are defined in
 tdurp_bnk  = in([ 2, 5, 8]+indx_offset); % batchVal4_discrete_3x3events_usethis.m
 bnk        = in([ 3, 6, 9]+indx_offset);
 bnkrt      = in([ 4, 7,10]+indx_offset);
 tdurp_thr  = in([11,16,21]+indx_offset);
 thr        = in([12,17,22]+indx_offset);
 thr_bnk    = in([13,18,23]+indx_offset);
 thr_aoa    = in([14,19,24]+indx_offset);
 thr_bet    = in([15,20,25]+indx_offset);

% write initial conditions for post run
 fid1=fopen('runval4init.dat','w');
   fprintf(fid1,['  gdalt  = ',num2str(Igdalt), ',\n']);
   fprintf(fid1,['  azvelr = ',num2str(Iazvelr),',\n']);
   fprintf(fid1,['  gammar = ',num2str(Igammar),',\n']);
   fprintf(fid1,['  gdlat  = ',num2str(Igdlat), ',\n']);
   fprintf(fid1,['  long   = ',num2str(Ilong),  ',\n']);
```

```
  fprintf(fid1,['  velr   = ',num2str(Ivelr),  ',\n']);
 fclose(fid1);

% write time to firewire for post run
 fid2=fopen('runval4fw.dat','w');
  fprintf(fid2,['  value=',num2str(tdurp_fire),',\n']);
 fclose(fid2);

% write times and bank angles for post run bank angle commands
% this is variable depending on function input
 fid3 = fopen('runval4bc.dat','w');
 for i=1:length(bnk)
   % just numbers the events starting at 30 in increments of 5
    evnt     = 25+i*5;
    fprintf(fid3,'cc-------------------------------------------------------------------\n');
    fprintf(fid3,'cc-------------------------------------------------------------------\n');
    fprintf(fid3,['l$gendat\n']);
    if i ==1
      fprintf(fid3,['  event =',num2str(evnt),...
         ',0.0, / primary event\n']);
    else
      fprintf(fid3,['  event =',num2str(evnt),...
         ',1.0, / secondary event\n']);
    end
    fprintf(fid3,['  critr="tdurp",\n']);
    fprintf(fid3,['  value = ',num2str(tdurp_bnk(i)),',\n']);
    fprintf(fid3,['  iguid(6) =0,0,1,0,0,0,/ bnkang input (from targeting algorithm)\n']);
    fprintf(fid3,['  bnkpc(1) =',num2str(bnk(i)),',\n']);
    fprintf(fid3,['  bnkpc(2) =',num2str(bnkrt(i)),',\n']);
    fprintf(fid3,['  parif(1) = 0.0, /no ball, use foil\n']);
    fprintf(fid3,['  diamp(1) = 0.0, /no ball, use foil\n']);
    fprintf(fid3,['  npc(32)  = 0, /no ball, use foil\n']);
    fprintf(fid3,['  endphs = 1,\n']);
    fprintf(fid3,[' $\n']);
 end
 fclose(fid3);

% write times and thrust values for post run thrust commands
% this is variable depending on function input
 fid4=fopen('runval4tc.dat','w');
 fprintf(fid4,'cc-------------------------------------------------------------------\n');
 fprintf(fid4,'cc-------------------------------------------------------------------\n');

 fprintf(fid4,'cc jetison parachute & go into freefall\n');
```

```
fprintf(fid4,'l$gendat\n');
fprintf(fid4,' event = 250, 0.0,\n');
fprintf(fid4,' critr = "gdalt",\n');
fprintf(fid4,[' value = ',num2str(Dgdalt+1000),',\n']);
fprintf(fid4,' npc(32)=0,              / jettison parachute\n');
fprintf(fid4,' parif(1)= 0.,\n');
fprintf(fid4,' diamp(1)= 0.,\n');
fprintf(fid4,' wjett = 276.702,        / weight to jettison\n');
fprintf(fid4,' sref=4.5238934,         / aerodynamic reference area [m^2] (from M98)\n');
fprintf(fid4,' iengmf(1) = 0,\n');
fprintf(fid4,' iwpf(1) = 0,\n');
fprintf(fid4,' iguid(1) = 0,0,1,\n');
fprintf(fid4,' $\n');
fprintf(fid4,'l$tblmlt\n');
fprintf(fid4,' $\n');
fprintf(fid4,'l$tab\n');
fprintf(fid4,' table="cdt",0,2.0,   / change drag back to lander only\n');
fprintf(fid4,' $\n');
fprintf(fid4,'l$tab\n');
fprintf(fid4,' table="clt",0,0.0,    / no lift\n');
fprintf(fid4,' $\n');
fprintf(fid4,'l$tab\n');
fprintf(fid4,' table="cdp1t",0,0.0,  / no chute drag\n');
fprintf(fid4,' endphs = 1,\n');
fprintf(fid4,' $\n');

for i=1:length(thr)
   evnt=250+i*5; % just numbers the events starting at evnt+5 in increments of 5
   fprintf(fid4,'cc-----------------------------------------------------------------------\n');
   fprintf(fid4,'cc-----------------------------------------------------------------------\n');
   fprintf(fid4,'l$gendat\n');
   if i ==1
     fprintf(fid4,['  event =',num2str(evnt),',0.0, / primary event\n']);
   else
     fprintf(fid4,['  event =',num2str(evnt),',1.0, / secondary event\n']);
   end

   fprintf(fid4,'  critr="tdurp",\n');
   fprintf(fid4,['  value = ',num2str(tdurp_thr(i)),',\n']);
   if i==1
      fprintf(fid4,'c initial engine event, turns engine on\n');
      fprintf(fid4,'  iengmf(1) = 1,\n');
      fprintf(fid4,'  iwpf(1)   = 1,\n');
      fprintf(fid4,'  iguid(1)  = 0,0,1,\n');
```

```
    end
    fprintf(fid4,['  iguid(6) =1,1,1,0,0,0,  / aoa,ss,bnk input deg\n']);
    fprintf(fid4,['  etapc(1) =',num2str(thr(i)),',\n']);
    fprintf(fid4,['  bnkpc(1) =',num2str(thr_bnk(i)),',\n']);
    fprintf(fid4,['  alppc(1) =',num2str(thr_aoa(i)),',\n']);
    fprintf(fid4,['  betpc(1) =',num2str(thr_bet(i)),',\n']);
    fprintf(fid4,['  endphs = 1,\n']);
    fprintf(fid4,[' $\n']);

end
fprintf(fid4,'cc------------------------------------------------------------------\n');
fprintf(fid4,'cc------------------------------------------------------------------\n');
fprintf(fid4,['c - marks ',num2str(Dgdalt),' meter mark (0m above surface)\n']);
fprintf(fid4,'l$gendat\n');
fprintf(fid4,' event=499,0.0, / primary event\n');
fprintf(fid4,' critr="gdalt",\n');
fprintf(fid4,[' value = ',num2str(Dgdalt),',\n']);
fprintf(fid4,' iengmf(1) =  0,0,\n');
fprintf(fid4,' iwpf(1) = 0,0,\n');
fprintf(fid4,' endphs=1,\n');
fprintf(fid4,' $\n');

fclose(fid4);
```

Appendix F – Particle Swarm Optimization (PSO) routine

```
% pso.m
% a generic particle swarm optimizer
% to find the minimum or maximum of any
% MISO matlab function
%
% Implements Common, Trelea type 1 and 2, and Clerc's class 1". It will
% also automatically try to track to a changing environment (with varied
% success - BKB 3/18/05)
%
% This vectorized version removes the for loop associated with particle
% number. It also *requires* that the cost function have a single input
% that represents all dimensions of search (i.e., for a function that has 2
% inputs then make a wrapper that passes a matrix of ps x 2 as a single
% variable)
%
% Usage:
%  [optOUT]=PSO(functname,D)
% or:
%  [optOUT,tr,te]=...
%       PSO(functname,D,mv,VarRange,minmax,PSOparams,plotfcn,PSOseedValue)
%
% Inputs:
%    functname - string of matlab function to optimize
%    D - # of inputs to the function (dimension of problem)
%
% Optional Inputs:
%    mv - max particle velocity, either a scalar or a vector of length D
%          (this allows each component to have it's own max velocity),
%          default = 4, set if not input or input as NaN
%
%    VarRange - matrix of ranges for each input variable,
%      default -100 to 100, of form:
%       [ min1 max1
%         min2 max2
%           ...
%         minD maxD ]
%
%    minmax = 0, funct minimized (default)
%           = 1, funct maximized
%           = 2, funct is targeted to P(12) (minimizes distance to errgoal)
%
%    PSOparams - PSO parameters
```

```
%     P(1) - Epochs between updating display, default = 100. if 0,
%           no display
%     P(2) - Maximum number of iterations (epochs) to train, default = 2000.
%     P(3) - population size, default = 24
%
%     P(4) - acceleration const 1 (local best influence), default = 2
%     P(5) - acceleration const 2 (global best influence), default = 2
%     P(6) - Initial inertia weight, default = 0.9
%     P(7) - Final inertia weight, default = 0.4
%     P(8) - Epoch when inertial weight at final value, default = 1500
%     P(9)- minimum global error gradient,
%               if abs(Gbest(i+1)-Gbest(i)) < gradient over
%               certain length of epochs, terminate run, default = 1e-25
%     P(10)- epochs before error gradient criterion terminates run,
%               default = 150, if the SSE does not change over 250 epochs
%                           then exit
%     P(11)- error goal, if NaN then unconstrained min or max, default=NaN
%     P(12)- type flag (which kind of PSO to use)
%               0   = Common PSO w/intertia (default)
%               1,2 = Trelea types 1,2
%               3   = Clerc's Constricted PSO, Type 1"
%                 4   = Hassan, Cohanim, Weck type (variation of Common, similar to Trelea
#2)
%     P(13)- PSOseed, default=0
%             = 0 for initial positions all random
%             = 1 for initial particles as user input
%
%    plotfcn - optional name of plotting function, default 'goplotpso',
%             make your own and put here
%
%    PSOseedValue - initial particle position, depends on P(13), must be
%                 set if P(13) is 1 or 2, not used for P(13)=0, needs to
%                 be nXm where n<=ps, and m<=D
%                 If n<ps and/or m<D then remaining values are set random
%                 on Varrange
% Outputs:
%    optOUT - optimal inputs and associated min/max output of function, of form:
%        [ bestin1
%          bestin2
%            ...
%          bestinD
%          bestOUT ]
%
% Optional Outputs:
```

```
%    tr    - Gbest at every iteration, traces flight of swarm
%    te    - epochs to train, returned as a vector 1:endepoch
%
% Example:  out=pso('f6',2)

% Brian Birge
% Rev 3.3 - 2/18/06
% Rev 3.4 - 7/ 4/06 - added Hassan type P(12) = 4



function [OUT,varargout]=pso(functname,D,varargin)

rand('state',sum(100*clock));
if nargin < 2
  error('Not enough arguments.');
end

% PSO PARAMETERS
if nargin == 2      % only specified functname and D
  VRmin   = ones(D,1)*-100;
  VRmax   =  ones(D,1)*100;
  VR      = [VRmin,VRmax];
  minmax  = 0;
  P       = [];
  mv      = 4;
  plotfcn ='goplotpso';

elseif nargin == 3  % specified functname, D, and mv
  VRmin   = ones(D,1)*-100;
  VRmax   = ones(D,1)*100;
  VR      = [VRmin,VRmax];
  minmax  = 0;
  mv      = varargin{1};
  if isnan(mv)
     mv  = 4;
  end
  P       = [];
  plotfcn = 'goplotpso';

elseif nargin == 4  % specified functname, D, mv, Varrange
  mv      = varargin{1};
  if isnan(mv)
     mv  = 4;
```

```
  end
  VR      = varargin{2};
  minmax  = 0;
  P       = [];
  plotfcn = 'goplotpso';

elseif nargin == 5  % Functname, D, mv, Varrange, and minmax
  mv      = varargin{1};
  if isnan(mv)
     mv = 4;
  end
  VR      = varargin{2};
  minmax  = varargin{3};
  P       = [];
  plotfcn = 'goplotpso';

elseif nargin == 6  % Functname, D, mv, Varrange, minmax, and psoparams
  mv      = varargin{1};
  if isnan(mv)
     mv  = 4;
  end
  VR      = varargin{2};
  minmax  = varargin{3};
  P       = varargin{4}; % psoparams
  plotfcn = 'goplotpso';

elseif nargin == 7  % Functname, D, mv, Varrange, minmax, and psoparams, plotfcn
  mv      = varargin{1};
  if isnan(mv)
     mv  = 4;
  end
  VR      = varargin{2};
  minmax  = varargin{3};
  P       = varargin{4}; % psoparams
  plotfcn = varargin{5};

elseif nargin == 8  % Functname, D, mv, Varrange, minmax, and psoparams, plotfcn,
PSOseedValue
  mv          = varargin{1};
  if isnan(mv)
     mv       = 4;
  end
  VR          = varargin{2};
  minmax      = varargin{3};
```

```
  P           = varargin{4}; % psoparams
  plotfcn     = varargin{5};
  PSOseedValue = varargin{6};

else
  error('Wrong # of input arguments.');

end

% sets up default pso params
Pdef = [100 2000 24 2 2 0.9 0.4 1500 1e-25 250 NaN 0 0]; % defaults
Plen = length(P);
P    = [P,Pdef(Plen+1:end)]; % replace defaults with user input if it exists

df     = P(1);  % display frequency
me     = P(2);  % max epochs
ps     = P(3);  % # of particles (population size)
ac1    = P(4);  % acceleration constant #1
ac2    = P(5);  % acceleration constant #2
iw1    = P(6);  % initial inertia value
iw2    = P(7);  % final inertia value
iwe    = P(8);  % epoch at which final inertia value reached
ergrd  = P(9);  % minimum error change
ergrdep = P(10); % duration of min err change before stopping
errgoal = P(11); % error goal
trelea  = P(12); % PSO model type
PSOseed = P(13); % initial values of particles

% used with trainpso, for neural net training
% this pulls variables out of the neural net training shell
if strcmp(functname,'pso_neteval')
  net = evalin('caller','net');
   Pd = evalin('caller','Pd');
   Tl = evalin('caller','Tl');
   Ai = evalin('caller','Ai');
    Q = evalin('caller','Q');
   TS = evalin('caller','TS');
end


% error checking
 if ((minmax==2) & isnan(errgoal))
    error('minmax= 2, errgoal= NaN: choose an error goal or set minmax to 0 or 1');
 end
```

```
 if ( (PSOseed==1) & ~exist('PSOseedValue') )
    error('PSOseed flag set but no PSOseedValue was input');
 end

 if exist('PSOseedValue')
    tmpsz=size(PSOseedValue);
    if D < tmpsz(2)
       error('PSOseedValue column size must be D or less');
    end
    if ps < tmpsz(1)
       error('PSOseedValue row length must be # of particles or less');
    end
 end

% set plotting flag
if (P(1))~=0
  plotflg=1;
else
  plotflg=0;
end

% preallocate variables for speed up
 tr = ones(1,me)*NaN;

% take care of setting max velocity and position params here
if length(mv)==1
 velmaskmin = -mv*ones(ps,D);     % min vel, psXD matrix
 velmaskmax = mv*ones(ps,D);      % max vel
elseif length(mv)==D
 velmaskmin = repmat(forcerow(-mv),ps,1); % min vel
 velmaskmax = repmat(forcerow( mv),ps,1); % max vel
else
 error('Max vel must be either a scalar or same length as prob dimension D');
end
posmaskmin  = repmat(VR(1:D,1)',ps,1);  % min pos, psXD matrix
posmaskmax  = repmat(VR(1:D,2)',ps,1);  % max pos
posmaskmeth = 3; % 3=bounce method (see comments below inside epoch loop)

% PLOTTING
 message = sprintf('PSO: %%g/%g iterations, GBest = %%20.20g.\n',me);

% INITIALIZE INITIALIZE INITIALIZE INITIALIZE INITIALIZE INITIALIZE
```

```
% initialize population of particles and their velocities at time zero,
% format of pos= (particle#, dimension)
 % construct random population positions bounded by VR
  pos(1:ps,1:D) = normmat(rand([ps,D]),VR',1);

  if PSOseed == 1        % initial positions user input, see comments above
    tmpsz                    = size(PSOseedValue);
    pos(1:tmpsz(1),1:tmpsz(2)) = PSOseedValue;
  end

 % construct initial random velocities between -mv,mv
  vel(1:ps,1:D) = normmat(rand([ps,D]),...
      [forcecol(-mv),forcecol(mv)]',1);

% initial pbest positions vals
 pbest = pos;

% cost function call
% 7/7/06 - change this s.t. if out = matrix then each row length represents
%          a multiobjective set, used for pareto front optimization
 out = feval(functname,pos);  % returns column of cost values (1 for each particle)
%---------------------------

 pbestval=out;   % initially, pbest is same as pos

% assign initial gbest here also (gbest and gbestval)
 if minmax==1
   % this picks gbestval when we want to maximize the function
    [gbestval,idx1] = max(pbestval);
 elseif minmax==0
   % this works for straight minimization
    [gbestval,idx1] = min(pbestval);
 elseif minmax==2
   % this works when you know target but not direction you need to go
   % good for a cost function that returns distance to target that can be either
   % negative or positive (direction info)
    [temp,idx1] = min((pbestval-ones(size(pbestval))*errgoal).^2);
    gbestval    = pbestval(idx1);
 end

 % preallocate a variable to keep track of gbest for all iters
 bestpos        = zeros(me,D+1)*NaN;
 gbest          = pbest(idx1,:);  % this is gbest position
   % used with trainpso, for neural net training
```

```
  % assign gbest to net at each iteration, these interim assignments
  % are for plotting mostly
   if strcmp(functname,'pso_neteval')
      net=setx(net,gbest);
   end
%tr(1)         = gbestval;       % save for output
bestpos(1,1:D) = gbest;

% this part used for implementing Carlisle and Dozier's APSO idea
% slightly modified, this tracks the global best as the sentry whereas
% their's chooses a different point to act as sentry
% see "Tracking Changing Extremea with Adaptive Particle Swarm Optimizer",
% part of the WAC 2002 Proceedings, June 9-13, http://wacong.com
 sentryval = gbestval;
 sentry    = gbest;

if (trelea == 3)
% calculate Clerc's constriction coefficient chi to use in his form
 kappa   = 1; % standard val = 1, change for more or less constriction
 if ( (ac1+ac2) <=4 )
    chi = kappa;
 else
    psi     = ac1 + ac2;
    chi_den = abs(2-psi-sqrt(psi^2 - 4*psi));
    chi_num = 2*kappa;
    chi     = chi_num/chi_den;
 end
end

% INITIALIZE END INITIALIZE END INITIALIZE END INITIALIZE END
rstflg = 0; % for dynamic environment checking
% start PSO iterative procedures
 cnt    = 0; % counter used for updating display according to df in the options
 cnt2   = 0; % counter used for the stopping subroutine based on error convergence
 iwt(1) = iw1;
for i=1:me  % start epoch loop (iterations)

    out       = feval(functname,[pos;gbest]);
    outbestval = out(end,:);
    out       = out(1:end-1,:);

    tr(i+1)         = gbestval; % keep track of global best val
    te              = i; % returns epoch number to calling program when done
    bestpos(i,1:D+1) = [gbest,gbestval];
```

```
    %assignin('base','bestpos',bestpos(i,1:D+1));
   %--------------------------------------------------------------------
   % this section does the plots during iterations
    if plotflg==1
     if (rem(i,df) == 0 ) | (i==me) | (i==1)
        fprintf(message,i,gbestval);
        cnt = cnt+1; % count how many times we display (useful for movies)

        eval(plotfcn); % defined at top of script

      end  % end update display every df if statement
    end % end plotflg if statement

    % check for an error space that changes wrt time/iter
    % threshold value that determines dynamic environment
    % sees if the value of gbest changes more than some threshold value
    % for the same location
    chkdyn = 1;
    rstflg = 0; % for dynamic environment checking

    if chkdyn==1
     threshld = 0.05;  % percent current best is allowed to change, .05 = 5% etc
     letiter  = 5; % # of iterations before checking environment, leave at least 3 so PSO has
time to converge
     outorng  = abs( 1- (outbestval/gbestval) ) >= threshld;
     samepos  = (max( sentry == gbest ));

     if (outorng & samepos) & rem(i,letiter)==0
        rstflg=1;
       % disp('New Environment: reset pbest, gbest, and vel');
       %% reset pbest and pbestval if warranted
%        outpbestval = feval( functname,[pbest] );
%        Poutorng    = abs( 1-(outpbestval./pbestval) ) > threshld;
%        pbestval    = pbestval.*~Poutorng + outpbestval.*Poutorng;
%        pbest       = pbest.*repmat(~Poutorng,1,D) + pos.*repmat(Poutorng,1,D);

        pbest     = pos; % reset personal bests to current positions
        pbestval  = out;
        vel       = vel*10; % agitate particles a little (or a lot)

       % recalculate best vals
        if minmax == 1
          [gbestval,idx1] = max(pbestval);
```

```
   elseif minmax==0
     [gbestval,idx1] = min(pbestval);
   elseif minmax==2 % this section needs work
     [temp,idx1] = min((pbestval-ones(size(pbestval))*errgoal).^2);
     gbestval    = pbestval(idx1);
   end

   gbest  = pbest(idx1,:);

   % used with trainpso, for neural net training
   % assign gbest to net at each iteration, these interim assignments
   % are for plotting mostly
   if strcmp(functname,'pso_neteval')
     net=setx(net,gbest);
   end
 end  % end if outorng

 sentryval = gbestval;
 sentry    = gbest;

end % end if chkdyn

% find particles where we have new pbest, depending on minmax choice
% then find gbest and gbestval
 %[size(out),size(pbestval)]
if rstflg == 0
 if minmax == 0
   [tempi]           = find(pbestval>=out); % new min pbestvals
   pbestval(tempi,1)  = out(tempi);   % update pbestvals
   pbest(tempi,:)     = pos(tempi,:); % update pbest positions

   [iterbestval,idx1] = min(pbestval);

   if gbestval >= iterbestval
      gbestval = iterbestval;
      gbest    = pbest(idx1,:);
      % used with trainpso, for neural net training
      % assign gbest to net at each iteration, these interim assignments
      % are for plotting mostly
       if strcmp(functname,'pso_neteval')
         net=setx(net,gbest);
       end
   end
 elseif minmax == 1
```

```
    [tempi,dum]       = find(pbestval<=out); % new max pbestvals
    pbestval(tempi,1)  = out(tempi,1); % update pbestvals
    pbest(tempi,:)     = pos(tempi,:); % update pbest positions

    [iterbestval,idx1] = max(pbestval);
    if gbestval <= iterbestval
       gbestval = iterbestval;
       gbest    = pbest(idx1,:);
       % used with trainpso, for neural net training
       % assign gbest to net at each iteration, these interim assignments
       % are for plotting mostly
        if strcmp(functname,'pso_neteval')
          net=setx(net,gbest);
        end
    end
  elseif minmax == 2  % this won't work as it is, fix it later
    egones           = errgoal*ones(ps,1); % vector of errgoals
    sqrerr2          = ((pbestval-egones).^2);
    sqrerr1          = ((out-egones).^2);
    [tempi,dum]      = find(sqerr1 <= sqrerr2); % find particles closest to targ
    pbestval(tempi,1) = out(tempi,1); % update pbestvals
    pbest(tempi,:)    = pos(tempi,:); % update pbest positions

    sqrerr           = ((pbestval-egones).^2); % need to do this to reflect new pbests
    [temp,idx1]      = min(sqrerr);
    iterbestval      = pbestval(idx1);

    if (iterbestval-errgoal)^2 <= (gbestval-errgoal)^2
      gbestval = iterbestval;
      gbest    = pbest(idx1,:);
      % used with trainpso, for neural net training
       % assign gbest to net at each iteration, these interim assignments
       % are for plotting mostly
        if strcmp(functname,'pso_neteval')
          net=setx(net,gbest);
        end
    end
  end
 end


%   % build a simple predictor 10th order, for gbest trajectory
%   if i>500
%    for dimcnt=1:D
```

```
%      pred_coef  = polyfit(i-250:i,(bestpos(i-250:i,dimcnt))',20);
%     % pred_coef  = polyfit(200:i,(bestpos(200:i,dimcnt))',20);
%      gbest_pred(i,dimcnt) = polyval(pred_coef,i+1);
%    end
%    else
%      gbest_pred(i,:) = zeros(size(gbest));
%    end

  %gbest_pred(i,:)=gbest;
  %assignin('base','gbest_pred',gbest_pred);

%   % convert to non-inertial frame
%    gbestoffset = gbest - gbest_pred(i,:);
%    gbest = gbest - gbestoffset;
%    pos   = pos + repmat(gbestoffset,ps,1);
%    pbest = pbest + repmat(gbestoffset,ps,1);


%PSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOP
SOPSOPSOPSOPSOPSO

     % get new velocities, positions (this is the heart of the PSO algorithm)
     % each epoch get new set of random numbers
      rannum1 = rand([ps,D]); % for Trelea and Clerc types
      rannum2 = rand([ps,D]);

      if     trelea == 1
       % from Trelea's paper, parameter set 1
        vel = 0.600.*vel...                         % prev vel
            +1.700.*rannum1.*(pbest-pos)...         % independent
            +1.700.*rannum2.*(repmat(gbest,ps,1)-pos); % social

      elseif trelea == 2
       % from Trelea's paper, parameter set 2
        vel = 0.729.*vel...                         % prev vel
            +1.494.*rannum1.*(pbest-pos)...         % independent
            +1.494.*rannum2.*(repmat(gbest,ps,1)-pos); % social

      elseif trelea ==3
       % Clerc's Type 1" PSO
        vel = chi*(vel...                           % prev vel
            +ac1.*rannum1.*(pbest-pos)...           % independent
            +ac2.*rannum2.*(repmat(gbest,ps,1)-pos)) ; % social
```

```
elseif trelea ==4
 % Hassan, Cohanim, Weck, Venter, "A Comparison of Particle Swarm
 % Optimization and the Genetic Algorithm", American Institute of
 % Aeronautics and Astronautics, 46th AIAA/ASME/ASCE/AHS/ASC Structures,
 % Structural Dynamics & Materials Conference 18-21 April 2005, Austin,
 % Texas
  vel = 0.500.*vel...                          % prev vel
       +1.500.*rannum1.*(pbest-pos)...          % independent
       +1.500.*rannum2.*(repmat(gbest,ps,1)-pos); % social

else
 % common PSO algo with inertia wt
 % get inertia weight, just a linear funct w.r.t. epoch parameter iwe
  if i<=iwe
     iwt(i) = ((iw2-iw1)/(iwe-1))*(i-1)+iw1;
  else
     iwt(i) = iw2;
  end
 % random number including acceleration constants
  ac11 = rannum1.*ac1;    % for common PSO w/inertia
  ac22 = rannum2.*ac2;

  vel = iwt(i).*vel...                         % prev vel
       +ac11.*(pbest-pos)...                    % independent
       +ac22.*(repmat(gbest,ps,1)-pos);         % social

end

% limit velocities here using masking
 vel = ( (vel <= velmaskmin).*velmaskmin ) + ( (vel > velmaskmin).*vel );
 vel = ( (vel >= velmaskmax).*velmaskmax ) + ( (vel < velmaskmax).*vel );

% update new position (PSO algo)
 pos = pos + vel;

% position masking, limits positions to desired search space
% method: 0) no position limiting, 1) saturation at limit,
%         2) wraparound at limit , 3) bounce off limit
 minposmask_throwaway = pos <= posmaskmin;  % these are psXD matrices
 minposmask_keep      = pos >  posmaskmin;
 maxposmask_throwaway = pos >= posmaskmax;
 maxposmask_keep      = pos <  posmaskmax;

 if    posmaskmeth == 1
```

```
      % this is the saturation method
       pos = ( minposmask_throwaway.*posmaskmin ) + ( minposmask_keep.*pos );
       pos = ( maxposmask_throwaway.*posmaskmax ) + ( maxposmask_keep.*pos );
      elseif posmaskmeth == 2
       % this is the wraparound method
       pos = ( minposmask_throwaway.*posmaskmax ) + ( minposmask_keep.*pos );
       pos = ( maxposmask_throwaway.*posmaskmin ) + ( maxposmask_keep.*pos );
      elseif posmaskmeth == 3
       % this is the bounce method, particles bounce off the boundaries with -vel
       pos = ( minposmask_throwaway.*posmaskmin ) + ( minposmask_keep.*pos );
       pos = ( maxposmask_throwaway.*posmaskmax ) + ( maxposmask_keep.*pos );

       vel = (vel.*minposmask_keep) + (-vel.*minposmask_throwaway);
       vel = (vel.*maxposmask_keep) + (-vel.*maxposmask_throwaway);
      else
       % no change, this is the original Eberhart, Kennedy method,
       % it lets the particles grow beyond bounds if psoparams (P)
       % especially Vmax, aren't set correctly, see the literature
      end


%PSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOPSOP
SOPSOPSOPSOPSOPSO
% check for stopping criterion based on speed of convergence to desired
  % error
   tmp1 = abs(tr(i) - gbestval);
   if tmp1 > ergrd
     cnt2 = 0;
   elseif tmp1 <= ergrd
     cnt2 = cnt2+1;
     if cnt2 >= ergrdep
       if plotflg == 1
        fprintf(message,i,gbestval);
        disp(' ');
        disp(['--> Solution likely, GBest hasn''t changed by at least ',...
            num2str(ergrd),' for ',...
                num2str(cnt2),' epochs.']);
        eval(plotfcn);
       end
       break
     end
   end

  % this stops if using constrained optimization and goal is reached
```

```
   if ~isnan(errgoal)
    if ((gbestval<=errgoal) & (minmax==0)) | ((gbestval>=errgoal) & (minmax==1))

        if plotflg == 1
            fprintf(message,i,gbestval);
            disp(' ');
            disp(['--> Error Goal reached, successful termination!']);

            eval(plotfcn);
        end
        break
    end

   % this is stopping criterion for constrained from both sides
    if minmax == 2
      if ((tr(i)<errgoal) & (gbestval>=errgoal)) | ((tr(i)>errgoal) ...
            & (gbestval <= errgoal))
        if plotflg == 1
            fprintf(message,i,gbestval);
            disp(' ');
            disp(['--> Error Goal reached, successful termination!']);

            eval(plotfcn);
        end
        break
      end
    end % end if minmax==2
   end  % end ~isnan if

end  % end epoch loop

% output & return
 OUT=[gbest';gbestval];
 varargout{1}=[1:te];
 varargout{2}=[tr(find(~isnan(tr)))];

 return
```