A SOFTWARE FRAMEWORK FOR NEURO-CONTROL OF DYNAMICAL SYSTEMS

Christopher Iliffe Sprague

ABSTRACT

Optimal trajectory profiles for spacecraft guidance are typically precomputed and supplied as nominal paths for simplified control systems to follow. These control systems rely on linearised dynamics about such nominal paths, and thus are subject to considerable error when deviations occur. In this work, a real-time implementable control system, reliant on spatial sensory feedback, is developed through the avenue of machine learning, eliminating the need to follow a precomputed path. In particular, artificial neural networks of various architectures are used to develop functions mapping the state of a dynamical system to a permissible control by learning state-control pairs from a large data set of mass-optimal control trajectories, generated through nonlinear programming and exploiting topological continuation of dynamics. Focus is placed upon investigating the merit of using networks with many layers, otherwise known as deep learning. It is shown that the resulting neuro-controllers are capable of controlling a dynamical system from an arbitrary initial state to an arbitrary final state in a near-optimal manner. A novel software framework is presented, allowing the automated implementation of the described intelligent control paradigm.

***Index Terms*—** trajectory optimisation, optimal control, artificial intelligence, deep learning, autonomous systems

1. INTRODUCTION

Determining optimally controlled trajectories for dynamical aerospace systems typically entails computationally expensive numerical optimisation techniques, which are infeasible for real-time implementation. In practice, a simplified reactive control system is implemented in order to enforce the dynamical system to follow such trajectories. Such simplified control systems often rely upon linearised dynamics about a nominal optimal control trajectory, and hence are only valid within the vicinity of the trajectory [1].

The use of artificial neural networks in machine learning has shown remarkable success in a variety of applications, ranging from playing board games [2] to transferring the artistic style of paintings [3]. Now with computational resources being more developed than ever before, training of networks with architectures having multiple hidden layers has become feasible. Particularly, with the method of so-called *deep learning*, potential has been shown in control tasks [4]. In literature, the application of deep artificial neural networks to control tasks has been largely limited to systems of simple state space dynamics and low-dimensional discrete action space; however promising strides have been made to approximate the *Hamilton-Jacobi-Bellman* equation in the context of nonlinear dynamical systems [5], and thus such methods may indeed prove successful in real-world systems.

In this work it is sought to automate the process of generically developing neuro-controllers for user defined nonlinear dynamical systems. In particular, direct methods of trajectory optimisation are used to solve the two-point boundary value problem resulting from the task of transferring a dynamical system from an arbitrary initial state to an arbitrary final state while maximising some measure of performance, in most aerospace cases, either trajectory duration or propellant expenditures. Various initial system states are considered in solving the transcribed deterministic continuous time trajectory optimisation problem in order to generate large databases of example mass-optimal control trajectories. Because the process of direct trajectory optimisation culminates in the form of a computationally expensive and high dimensional nonlinear programming problem, the method of homotopy is used to exploit the topological continuation in the dynamics of the system in order to generate such a database in an expedient manner.

A large set of initial spacecraft states is generated in sequence of similarity through *random walks* within predefined initial condition boundaries. The *topological continuation* of the system's dynamics between subsequent initial states is exploited to rapidly optimise landing trajectories in order to generate a large database of optimal trajectories. The state-control pairs along each trajectory are then compiled into a single unified database, from which artificial neural networks of various architectures are trained to regress. After sufficient training, each neural network effectively maps the state of the particular dynamical system to an appropriate control at every time step in order to optimally meet its boundary conditions. That is, with the implementation of a trained neural network, such an intelligently controlled dynamical system should be able to control itself to its specified final state, while maximising its objective along its trajectory.

Using the state-control pairs along each generated trajectory and tasking a neural network of a chosen architecture to learn the optimal mapping of the state of a system to an appropriate action, the learning process is conducted through non-

```
class Dynamical_Model(object):
  def __init__(self, si, st, slb, sub, clb, cub,
      tlb, tub):
    ...
  # For Direct Methods
  def EOM_State(self, state, control):
    ...
  def EOM_State_Jac(self, state, control):
    ...
  # For Indirect Methods
  def EOM_Fullstate(self, fullstate, control):
    ...
  def EOM_Fullstate_Jac(self, fullstate, control):
    ...
  def Hamiltonian(self, fullstate):
    ...
  def Pontryagin(self, fullstate):
  # For Machine Learning
  def Brain(self, state):
    ...
```

Listing 1. Dynamical Model Base Class

linear regression of the database for a number of training iterations until a satisfactory measurement of error is achieved. The implemented neural networks accomplishes nonlinear regression through the optimisation of their parameters. After the implemented artificial neural network model has been trained sufficiently, its parameters are saved for later use. In such a model's implementation, the state of the dynamical system is fed through at every time step to the neural network model, from which it computes an output of the prediction, in this case an appropriate control resembling the *optimal* control for the system to take at a particular time to arrive at its target destination while minimising the chosen metric of path performance.

2. DYNAMICAL MODEL

In this software framework, users are enabled to specify a *dynamical model* of their own choosing. Dynamical models are instantiated with an object orientated approach, inheriting from the `Dynamical_Model` base class shown in Listing 1.

In this framework, the instantiation of a dynamical system requires that one must specify the initial state of the system `si` and its target state `st`, describing the *two-point boundary value problem*. Additionally, it is also required that the trajectory optimisation subproblem is rectangular box bounded with respect to the state and control profile of the system. Hence the parameters `slb, sub, clb, cub`, describing the upper and lower bounds of the system's state and control, must be specified a priori. Similarly, because the problem of *direct trajectory optimisation* through *nonlinear programming* also requires the duration of the trajectory t_f as a parameter, the lower and upper bounds of the trajectory duration, `tlb, tub`, must be specified as well.

To further elaborate on this methodology, consider a simple two-dimensional planetary lander, modelled as a point mass capable of thrusting in any direction, given by the system of first order ordinary differential equations, Simple Lander, in Equation 1,

$$\text{Simple Lander} \begin{cases} v_x \\ v_y \\ \frac{Tu}{m}\hat{u}_x \\ \frac{Tu}{m}\hat{u}_y - g \\ -\frac{T}{I_{sp}g_0} \end{cases}, \tag{1}$$

where the set of parameters $\{T, I_{sp}, g_0, g\}$, maximum thrust, specific impulse, Earth's sea-level gravity, and the environmental gravity, are inherent to this specific model. This dynamical model in particular has a state space, $\mathcal{S}$ or `sdim`, dimensionality of five given by the vector

$$\boldsymbol{s}^\top = [x, y, v_x, v_y, m] \in \mathcal{S}, \tag{2}$$

and a control space, $\mathcal{U}$ or `cdim`, dimensionality of three given by the vector

$$\boldsymbol{c}^\top = [u, \hat{u}_x, \hat{u}_y] \in \mathcal{U}. \tag{3}$$

In an object orientated manner, inheriting from the base class `Dynamical_Model`, such a model is instantiated as shown in Listing 2. Using this work flow, the process of generating several optimal control trajectories from various initial states becomes easier.

Aside from the parameters of the system, at the very least one must also define the equations of motion to which the system's dynamics obey. The instance method, `EOM_State`, seen in Listing 3, is used to map the state and control of the system at any moment in time to its time rate of change, as outlined for this particular model in Equation 1.

Optionally, in order to improve the accuracy of numerical integration, one may also supply the Jacobian of the system's equations of motion, `EOM_State_Jac`. In order to improve the speed and accuracy of gradient based optimisation methods and numerical integration, one can optionally also define the square Jacobian matrix $\frac{\partial \dot{\boldsymbol{s}}}{\partial \boldsymbol{s}}$ for the equations of motion, which for the planetary lander in Equation 1 is

$$\frac{\partial \dot{\boldsymbol{s}}}{\partial \boldsymbol{s}} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & -\frac{T\hat{u}_x}{m^2}u \\ 0 & 0 & 0 & 0 & -\frac{T\hat{u}_y}{m^2}u \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \tag{4}$$

which is implemented with the instance method shown in Listing 4.

Just defining a model with the instance method `EOM_State` alone is enough to perform trajectory optimisation using *direct methods*; however, if one wishes to use *indirect methods* to solve a trajectory optimisation problem, it is necessary to define the instance methods `EOM_Fullstate`, `Hamiltonian`,

```
class Lander(Dynamical_Model):
    def __init__(
        self,
        si  = [10, 1000, 20, -5, 9500],
        st  = [0, 0, 0, 0, 8000],
        Isp = 311,
        g   = 1.6229,
        T   = 44000
    ):

        # Problem parameters
        self.Isp  = float(Isp)
        self.g    = float(g)
        self.T    = float(T)
        self.g0   = float(9.802)

        # Instantiate as a Dynamical_Model
        Dynamical_Model.__init__(
            self,
            si,
            st,
            [-1000, 0, -500, -500, 0],
            [1000, 2000, 500, 500, 10000],
            [0, -1, -1],
            [1, 1, 1],
            1,
            200
        )
```

Listing 2. Lander Dynamical Model

```
def EOM_State(self, state, control):
    x, y, vx, vy, m = state
    u, ux, uy       = control
    x0 = self.T*u/m
    return array([
        vx,
        vy,
        ux*x0,
        uy*x0 - self.g,
        -self.T*u/(self.Isp*self.g0)
    ], float)
```

Listing 3. Lander Equations of Motion

```
def EOM_State_Jac(self, state, control):
    x, y, vx, vy, m = state
    u, st, ct       = control
    x0              = self.T*u/m**2
    return array([
        [0, 0, 1, 0,        0],
        [0, 0, 0, 1,        0],
        [0, 0, 0, 0, -ux*x0/m],
        [0, 0, 0, 0, -uy*x0/m],
        [0, 0, 0, 0,        0]
    ], float)
```

Listing 4. Lander Jacobian

```
def Hamiltonian(self, fullstate, control):
    x, y, vx, vy, m, lx, ly, lvx, lvy, lm =
        fullstate
    u, ux, uy = control
    # Get the model parametres
    T, Isp, g0, g = self.T, self.Isp, self.g0,
        self.g
    # Common sub expression elimination
    x0 = T*u/m
    x1 = 1/(Isp*g0)
    # Dot the costates with the states
    H  = lx*vx + ly*vy + lvx*ux*x0 + lvy*(uy*x0 -
        g) - T*lvm*u*x1
    # Add the Lagrangian or cost functional
    H += T*u
    return H
```

Listing 5. Point Lander Hamiltonian Instance Method

and `Pontryagin`, which require analytical derivation through optimal control theory.

Through the formal processes of optimal control theory, one first defines the vector of *costate variables*, $\boldsymbol{\lambda}$ or `l`, corresponding to the system's state, such that its dimension matches that of the systems state. In the particular case of the lander described in Equation 1, the costate variables are

$$\boldsymbol{\lambda}^\top = [\lambda_x, \lambda_y, \lambda_{v_x}, \lambda_{v_y}, \lambda_m], \tag{5}$$

from which the *Hamiltonian* is derived through

$$\mathcal{H} = \boldsymbol{\lambda}^\top \dot{\boldsymbol{s}} + \mathcal{L}, \tag{6}$$

where the system's *Lagrangian* or cost functional is defined as

$$\mathcal{L} = Tu. \tag{7}$$

For this model, the Lagrangian in Equation 7 basically says that the optimal trajectory should minimise the use of the control, directly restricting the corresponding fuel expenditure. By computing Equation 6, the Hamiltonian of the system in Equation 1 becomes

$$\begin{aligned}\mathcal{H} = \lambda_x v_x + \lambda_y v_y + \\ \lambda_{v_x}\left(\frac{Tu}{m}\hat{u}_x\right) + \lambda_{v_x}\left(\frac{Tu}{m}\hat{u}_y - g\right) + \\ \lambda_m\left(-\frac{T}{I_{sp}g_0}\right) + Tu,\end{aligned} \tag{8}$$

which is then defined with the instance method in Listing 5. After the Hamiltonian is derived, one then derives the costate equations of motion, which through optimal control theory is found by computing

$$\dot{\boldsymbol{\lambda}} = -\frac{\partial H}{\partial \boldsymbol{s}}, \tag{9}$$

```
def EOM_Fullstate(self, fullstate, control):
    x, y, vx, vy, m, lx, ly, lvx, lvy, lm =
        fullstate
    u, ux, uy      = control
    T, Isp, g0, g = self.T, self.Isp, self.g0,
        self.g
    x0             = T*u/m
    x1             = T*u/m**2
    return array([
        [                    vx],
        [                    vy],
        [                ux*x0],
        [            uy*x0 - g],
        [       -T*u/(Isp*g0)],
        [                    0],
        [                    0],
        [                -lx],
        [                -ly],
        [x1*(lvx*ux + lvy*uy)]
    ], float)
```

Listing 6. Lander Full State Equations of Motion

which for the lander model of Equation 1 becomes

$$\dot{\boldsymbol{\lambda}} = \begin{bmatrix} 0 \\ 0 \\ -\lambda_x \\ -\lambda_y \\ \frac{Tu}{m^2}\left(\hat{u}_x\lambda_{v_x} + \hat{u}_y\lambda_{v_y}\right) \end{bmatrix}, \tag{10}$$

allowing the instance method `EOM_Fullstate` to be defined as shown in Listing 6. Similarly to that shown in Equation 4, one can increase the performance of optimisation and numerical integration by defining the Jacobian matrix of the full state equations of motion, as shown in Equation 11 and in the instance method shown in Listing 7.

$$\frac{\partial(\dot{\boldsymbol{s}} \cup \dot{\boldsymbol{\lambda}})}{\partial(\boldsymbol{s} \cup \boldsymbol{\lambda})} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\frac{T\hat{u}_x}{m^2}u & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\frac{T\hat{u}_y}{m^2}u & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\frac{2Tu}{m^3}\left(\hat{u}_y\lambda_{v_y}+\lambda_{v_x}\hat{u}_x\right) & 0 & 0 & \frac{T\hat{u}_x}{m^2}u & \frac{T\hat{u}_y}{m^2}u & 0 \end{bmatrix} \tag{11}$$

The only necessary step left to be taken in order to implement indirect methods of trajectory optimisation is to define the instance method `Pontryagin`, named after the famous optimal control theory principle, Pontryagin's maximum principle [6], which takes as its parameters the system's full state $\boldsymbol{s} \cup \boldsymbol{\lambda}$. Through maximising the Hamiltonian over the set of all possible controls, that is

$$H(\boldsymbol{s}_k^\star, \boldsymbol{u}_k^\star, \lambda_k^\star) \leq H(\boldsymbol{s}_k^\star, \boldsymbol{u}, \lambda_k^\star)\ \forall \boldsymbol{u} \in \mathcal{U}, \tag{12}$$

```
def EOM_Fullstate_Jac(self, fullstate, control):
    x, y, vx, vy, m, lx, ly, lvx, lvy, lm =
        fullstate
    u, ux, uy = control
    T         = self.T
    x0        = T*u/m**2
    x1        = ux*x0
    x2        = uy*x0
    x3        = 2*T*u/m**3
    return array([
        [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, -x1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, -x2, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, -1, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, -1, 0, 0, 0],
        [0, 0, 0, 0, -uy*lvy*x3 - lvx*ux*x3,
            0, 0, x1, x2, 0]
    ], float)
```

Listing 7. Lander Full State Jacobian

an on board optimal control policy is found that maps the system's full state to an appropriate control, that is $\boldsymbol{s}_k \mapsto \boldsymbol{u}_k \forall k$. In the specific case of the planetary lander of Equation 1, Equation 12 results in the optimal thrust direction being

$$\hat{u}_x = -\frac{\lambda_{v_x}}{\sqrt{\lambda_{v_x}^2 + \lambda_{v_y}^2}} \tag{13}$$

$$\hat{u}_y = -\frac{\lambda_{v_y}}{\sqrt{\lambda_{v_x}^2 + \lambda_{v_y}^2}} \tag{14}$$

and the optimal thrust throttle being

$$u = \begin{cases} 1 \text{ if } S < 0 \\ 0 \text{ if } S > 0 \end{cases}, \tag{15}$$

where the switching function is

$$S = \frac{I_{sp} g_0 \sqrt{\lambda_{v_x}^2 + \lambda_{v_y}^2}}{m} - \lambda_m. \tag{16}$$

Through such analytical derivations, one can finally define the instance method `Pontryagin` as shown in Listing 8.

After having properly defined the `Dynamical_Model` instance methods for a user's particular model, either direct or indirect methods of trajectory optimisation, outlined in Sections 3 and 4, can be used to compute optimal control trajectories between arbitrary initial and final states. The user can use methods of direct trajectory optimisation if they have defined, at the least, the `EOM_State` instance method. The user can use indirect methods if they have defined the instance methods `EOM_Fullstate`, `Hamiltonian`, and `Pontryagin`. It is noted,

```
def Pontryagin(self, fullstate):
    x, y, vx, vy, m, lx, ly, lvx, lvy, lm =
        fullstate
    lv = sqrt(abs(lvx)**2 + abs(lvy)**2)
    ux = -lvx/lv
    uy = -lvy/lv
    S  = 1 - self.Isp*self.g0*lv/m - lm
    if S < 0:
      u = 1
    elif S => 0:
      u = 0
    return u, ux, uy
```

Listing 8. Lander Pontryagin Maximum Principle

however, that the implementation of indirect trajectory optimisation methods is rather burdensome in comparison that of direct trajectory optimisation methods. In indirect methods, not only does the user need to define a greater number of instance methods, but the resulting methods also require analytical derivation, which in many systems proves to be quite difficult when the dynamics are sufficiently complicated.

3. INDIRECT TRAJECTORY OPTIMISATION

In the regime of indirect trajectory optimisation, the optimal control at any moment in time is readily determined from Pontryagin's maximum principle, shown in Equation 12. Although this principle supplies the proper controls, these outputs are only considered optimal with respect to the instantaneous state of the system; the controls at every moment in time will only guide the system to the proper final state if the initial costate variables $\boldsymbol{\lambda}_0$ are determined. These initial costate variables govern how the control of the system flows within its domain. Ultimately, indirect methods result in a two-point boundary value problem, in which optimality is guaranteed by Pontryagin's maximum principle, and hence only the boundary conditions of the particular scenario need to be satisfied.

Although in both the cases of indirect and direct trajectory optimisation methods, the dynamical model needs to satisfy the boundary conditions of arriving at the desired final state, indirect trajectory optimisation requires also the satisfaction of extra non physical boundary conditions, inherent to optimal control theory. If any individual elements of the final state of the system are free, for example total propellant use, their corresponding costate variables must be zero at the final time. Additionally, if the duration of the system's trajectory is free, the system's Hamiltonian must be zero at the final time, forming what is known as an infinite time-horizon problem as frequently encountered in mass-optimal control problems. In summary, the problem of indirect trajectory optimisation follows the following steps:

1. Form the Hamiltonian, i.e. Equation 6.
2. Find the Euler-Lagrange equations, e.g. Equation 7.
3. Solve the adjoint equations through Pontryagin's maximum principle for the optimal control policy, i.e. Equation 12.
4. Find the switching function, e.g. Equation 16.
5. Apply appropriate boundary and transversality conditions.
   - $\mathcal{H}(t_f) = 0$ if t_f is free.
   - $\lambda_i(t_f) = 0$ if x_i is free.
6. Solve the resulting two-point boundary value problem, i.e. find the initial costate variables $\boldsymbol{\lambda_0} = \boldsymbol{\lambda}(t_0)$.

```
from Trajectory import Point_Lander
from Optimisation import Indirect_Shooting,
    Indirect_Multiple_Shooting
from PyGMO import *

# Instantiate the dynamical model
model = Point_Lander()
# Instantiate the single shooting problem
problem1 = Indirect_Shooting(model)
problem2 = Indirect_Multiple_Shooting(model)
# Use sequential least squares quadratic
    programming
algo = algorithm.scipy_slsqp()
# Generate a random initial guesses
pop1 = population(problem1, 1)
pop2 = population(problem2, 1)
# Optimise problems to default error tolerance
pop1 = algo.evolve(pop1)
pop2 = algo.evolve(pop2)
# Extract the final solution
z1 = pop1.champion.x
z2 = pop2.champion.x
```

Listing 9. Indirect Trajectory Optimisation

It should be noted that this method of trajectory optimisation is rather difficult to solve, not only due to the fact that the user needs to supply analytical derivations, but also because the resulting initial costate variables are technically unbounded non-physical quantities, and thus are rather unintuitive to supply an initial guess to in order to initialise the two point boundary value problem solver algorithm.

With the use of the open-source software library, Python Parallel Global Multi-objective Optimiser (PyGMO), of the European Space Agency's Advanced Concepts Team [7], the user can solve the resulting two point boundary value problem using either of the indirect trajectory optimisation problems, `Indirect_Shooting` or `Indirect_Multiple_Shooting`, to setup and solve the finite parameter problem. With the intuitive work flow of this framework, the user may simply solve such problems as shown in Listing 9. This approach is taken with all optimisation methods within this framework.

4. DIRECT TRAJECTORY OPTIMISATION

A nonlinear program (NLP) is described by a by a *finite* set of variables $\boldsymbol{x}$ and constraints $\boldsymbol{c}$. The goal of a transcription method is to transform an *infinite-dimension* optimal control problem into a finite-dimension nonlinear program. Transcribing such a problem into a finite set of variables and constraints allows one to use iterative Newton-based methods in solving the resulting NLP. Transcribing a continuous optimal control problem into a finite-dimension parameter optimisation problem involves the following:

1. Characterise the dynamic system's trajectory with a finite set of variables.

2. Use a parameter optimisation method to solve the resulting finite-dimensional problem.

Direct methods operate by discretising the trajectory optimisation problem directly, in order to formulate a constrained parameter optimisation problem. Direct transcription methods, in general, have several advantages over indirect transcription methods. The necessity to analytically determine initial costate variables is circumvented, and the optimisation problem size is reduced by a factor of two [8]. One of the most significant merits is that, one does not need to specify the structure of the problem, a priori. In general, direct transcription methods are very robust and are able to converge to optimal solutions from poor initial guesses.

For general dynamical systems, the problem of trajectory optimisation is formulated as

$$\begin{aligned} \underset{\boldsymbol{u}(t)}{\text{minimise}} \quad & \phi(t_s, t_f, \boldsymbol{x_s}, \boldsymbol{x_f}) + \int_{t_s}^{t_f} \mathcal{L}(\boldsymbol{x}(t), \boldsymbol{u}(t), t) dt \\ \text{subject to} \quad & \dot{\boldsymbol{x}} = \boldsymbol{a}(\boldsymbol{x}, t) + \boldsymbol{u}(t) \text{ dynamic constraints} \\ & \mathcal{G}(t_s, t_f, \boldsymbol{x}_s, \boldsymbol{x}_f) \leq 0 \text{ boundary conditions} \\ & \boldsymbol{u} \in \mathcal{U}(\boldsymbol{x}, t) \text{ propulsion constraints} \end{aligned}$$

Essentially, this formulation asks for the control at every moment in time $\boldsymbol{u}(t)$ in order to minimise the initial and terminal costs $\phi(t_s, t_f, \boldsymbol{x}_s, \boldsymbol{x}_f)$ and the path costs $\int_{t_s}^{t_f} \mathcal{L}\left(\boldsymbol{x}(t), \boldsymbol{u}(t), t\right) dt$, all while abiding to dynamic constraints and boundary constraints, which are all dictated by the state of the dynamical system $\boldsymbol{x}(t)$. Following the common notation of optimal control theory, the trajectory of dynamical system can be formulated as a system of time varying variables.

$$\boldsymbol{z} = \begin{bmatrix} \boldsymbol{x}(t) \\ \boldsymbol{u}(t) \end{bmatrix}, \tag{17}$$

consisting of the system's *state variables* $\boldsymbol{x}(t)$ and the *control variables* $\boldsymbol{u}(t)$.

The way in which the dynamical system evolves over time is described by the system dynamics. These dynamics, for the scope of this paper's discussion, are described by a system of ordinary differential equations, more commonly known as *state equations*. The state equations, in their generic form, are written as

$$\dot{\boldsymbol{x}} = \frac{d\boldsymbol{x}}{dt} = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t), t). \tag{18}$$

Note that this description, in its generic form, describes a non-autonomous system, hence the right hand side's explicit dependence on time t. In the case of many aerospace applications, the system of ordinary differential equations governing a system's dynamics, does not explicitly depend on time. For example, in the case of a spacecraft on an interplanetary trajectory, its dynamics do not depend on time, but rather the inverse square of its distance relative to surrounding celestial bodies.

In the problem of trajectory optimisation, two conditions must be strictly satisfied, that is the system's initial dynamic variables

$$\boldsymbol{\psi}_0 = \boldsymbol{\psi}(\boldsymbol{x}(t_0), \boldsymbol{u}(t_0), t_0), \tag{19}$$

at the initial time t_0, and the system's terminal dynamic variables,

$$\boldsymbol{\psi}_f = \boldsymbol{\psi}(\boldsymbol{x}(t_f), \boldsymbol{u}(t_f), t_f), \tag{20}$$

at the terminal time t_f define the *boundary conditions* of the trajectory optimisation problem.

In many dynamical systems, the problem of trajectory optimisation leads itself to path constraints. These path constraints can range from obstacle avoidance in quad-copters to trajectory energy in spacecraft inserting into Lagrange points. These path constraints can be formally stated as

$$\boldsymbol{g}(\boldsymbol{x}(t), \boldsymbol{u}(t), t) = 0. \tag{21}$$

Note, that this generic description of a trajectory optimisation problem's path constraints forms a strong analogue to equality constraints in design optimisation.

Of course, for many reasons it is necessary to bound a trajectory optimisation problem, including, but not limited to: hardware constraints and simulation constraints. For example, one would bound the angular velocity of their fictional spacecraft so that it is not spinning into eternity. Additionally, one may want to place rectangular bounds on the environment of their spacecraft's simulation, for example: a spacecraft's position should be within the solar system. Aside from state constraints, there are also control constraints. The bounding on the system's state is formulated as

$$\boldsymbol{x}_l \leq \boldsymbol{x}(t) \leq \boldsymbol{x}_u \tag{22}$$

and on the system's controls as

$$\boldsymbol{u}_l \leq \boldsymbol{u}(t) \leq \boldsymbol{u}_u. \tag{23}$$

The whole point of trajectory optimisation is select a control policy $\boldsymbol{u}(t)$ to minimise or maximise some measure of performance. In some cases, this measure of performance

might be the time to travel from a starting location to an ending location in a dynamical system such as a quad copter. In many aerospace cases, the measure of performance is, of course, propellant expenditure. For a general dynamical system, the path cost or objective function is defined as

$$J = \phi(\boldsymbol{x}(t_0), \boldsymbol{x}(t_f), t_0, t_f) \tag{24}$$

In the case of a landing that reduces fuel usage, otherwise know as a mass optimal landing, the final mass is used explicitly to determine the measure of performance.

To solve an optimal control problem through direct trajectory optimisation methods, one considers the problem of choosing a dynamical system's control inputs $\boldsymbol{u}(t)$ for every time t to minimise the *cost functional*

$$J = \phi\left[\boldsymbol{x}(t_F), t_F\right], \tag{25}$$

while obeying the *state equations*

$$\dot{\boldsymbol{x}} = \boldsymbol{f}\left[\boldsymbol{x}(t), \boldsymbol{u}(t)\right]. \tag{26}$$

Under the assumption that the initial and final times $\{t_I, t_F\}$ are chosen *a priori*, one can define the NLP *decision vector* as

$$\boldsymbol{z}^\top = \left[\boldsymbol{u}_1, \boldsymbol{y}_2, \boldsymbol{u}_2, \ldots, \boldsymbol{y}_M, \boldsymbol{u}_M\right], \tag{27}$$

where $\boldsymbol{x}_k$ and $\boldsymbol{u}_k$ are the system's state and control variables respectively, evaluated at all times $t_1, \ldots, t_M$. In the regime of direct trajectory optimisation, it is required to enforce the continuity of the dynamical system's state dynamics. From the perspective of *nonlinear programming*, if one supposes that

$$\dot{\boldsymbol{y}} = \frac{\boldsymbol{y}_{k+1} + \boldsymbol{y}_k}{h} \tag{28}$$

$$h = \frac{t_F}{M} \tag{29}$$

this is conveniently enforced through *equality constraints* or *defects*

$$\boldsymbol{c}_k = \boldsymbol{\zeta}_k = \boldsymbol{y}_{k+1} - \boldsymbol{y}_k - h\boldsymbol{f}\left(\boldsymbol{y}_k, \boldsymbol{u}_k\right) = \boldsymbol{0} \quad \forall k \in \{1, \ldots, M-1\}. \tag{30}$$

Under the NLP notation, the *cost functional* then becomes the *objective function* to be minimised

$$F(\boldsymbol{x}) = \phi(\boldsymbol{y}_M). \tag{31}$$

In direct transcription methods, the continuous state $\boldsymbol{x}(t)$ and control $\boldsymbol{u}(t)$ are discretized into a finite dimensional description. Through this technique, the original optimal control problem is transformed into a nonlinear programming problem (NLP). For the sake of intuition, transforming the nonlinear programming problem back to an optimal control problem would be akin to having an infinite quantity of states, controls, and constraints.

The first step in transforming a continuous optimal control problem into a finite dimension nonlinear programming problem is to define the resolution of the discretisation n, or the number of nodes. Including the statement of the dynamical system's simulation starting time t_I and final time t_F. From these statements, one can formulate the time grid

$$[t_1, t_2., \ldots, t_n], \tag{32}$$

where one should note that $t_I = t_1$ and $t_F = t_n$, which form part of the system's boundary conditions. Intuitively, it is followed that the states are discretised as

$$\boldsymbol{x}_i = \boldsymbol{x}(t_i) \tag{33}$$

and the controls as

$$\boldsymbol{u}_i = \boldsymbol{u}(t_i). \tag{34}$$

Once the system's original optimal control description has been transcribed through direct transcription, the values of the system's states $\boldsymbol{x}_i$ and controls $\boldsymbol{u}_i$ are represented by a (possibly) large set of NLP variables. The aforementioned bounds on both the system's state and control are simply instantiated at each node n in the discretisation. Perhaps the most crucial part of this direct trajectory optimisation process is enforcing the system's inherent dynamics at each node. That is, an analytical quadrature procedure is used to compare the system's states at collocated nodes, in order to asses the validity of the optimiser's placement of each of the system's states.

As previously noted, under the direct transcription process, in which the system's continuous state and control descriptions are transformed to a finite dimensional parameter constrained problem, a single NLP *decision* vector is dealt with. This decision vector is formulated as

$$\boldsymbol{z}^\top = [t_0, t_f, \boldsymbol{x}_0, \boldsymbol{u}_0, \boldsymbol{x}_1, \boldsymbol{u}_1 \ldots, \boldsymbol{x}_n, \boldsymbol{u}_n], \tag{35}$$

where t_0 and t_f are regarded as auxiliary variables. The auxiliary variables, in this case the time grid boundaries, determine the behaviour of the optimiser's manipulation of the state and control variables, through the time step size passed to the analytical quadrature method.

In order to compute the defects at each node in the grid space, a method to analytically propagate the system's dynamics is used. In this paper, herein the method of trapezoidal quadrature will be used. Within the grid space, at node n, the defect is defined as

$$\boldsymbol{\zeta}_n = \boldsymbol{x}_n - \boldsymbol{x}_{n-1} - \frac{h_n}{2}\left(\boldsymbol{f}_n + \boldsymbol{f}_{n-1}\right). \tag{36}$$

Note that h_n is the collocation step size, defined by

$$h_n = t_n - t_{n-1}, \tag{37}$$

and $\boldsymbol{f}_n$ is described by the system's equations of motion at node n as

$$\boldsymbol{f}_n = \boldsymbol{f}\left(\boldsymbol{x}(t_n), \boldsymbol{u}(t_n), t_n\right). \tag{38}$$

The resulting characterisation of the defect forms the equality constraints of the dynamical system's trajectory optimisation problem. If these equality constraints are satisfied at each node, then the dynamics of the optimised trajectory will be feasible according to trapezoidal quadrature.

While the trapezoidal transcription method for trajectory optimisation is sufficient for many applications, one can find higher fidelity results and better performance from the Hermite-Simpson transcription method, especially in its separated form. Many trajectory discretisation methods do not fully exploit the right hand side sparsity of the system's differential equations. The discretisation separability of the *Hermite-Simpson Method* [8] can be exploited to increase *sparsity* in the calculation of the NLP *Jacobian* and *Hessian*, thus saving significant time in computations. The so called defects, the equality constraints enforcing the system's dynamics, are characterised by the *Hermite interpolant* for the state at the interval midpoint

$$\mathbf{0} = \bar{\boldsymbol{x}}_{k+1} - \frac{\boldsymbol{x}_{k+1} + \boldsymbol{x}_k}{2} - \frac{h_k}{8}(\boldsymbol{f}_k - \boldsymbol{f}_{k+1}) \forall k \in \mathcal{N} \quad (39)$$

and the *Simpson quadrature* over the interval

$$\mathbf{0} = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k - \frac{h_k}{6}(\boldsymbol{f}_{k+1} + 4\bar{\boldsymbol{f}}_{k+1} + \boldsymbol{f}_k) \forall k \in \mathcal{N}, \quad (40)$$

where system's state $\boldsymbol{x}_k$, control $\boldsymbol{u}_k$, dynamics $\boldsymbol{f}_k$, and time step h_k are indexed by the set of trajectory *nodes* $\mathcal{N} = [1, \ldots, M]$ and *segments* $\mathcal{N} = [1, \ldots, M-1]$. The resulting NLP *decision vector*

$$\boldsymbol{z}^\top = [t_f, \boldsymbol{x}_1, \boldsymbol{u}_1, \bar{\boldsymbol{x}}_2, \bar{\boldsymbol{u}}_2, \boldsymbol{x}_2, \boldsymbol{u}_2, \ldots, \bar{\boldsymbol{x}}_M, \bar{\boldsymbol{u}}_M, \boldsymbol{x}_M, \boldsymbol{u}_M] \quad (41)$$

is then manipulated through *sequential least squares quadratic programming* until the dynamics constraints, Equations 39 and 40, and boundary conditions,

$$\boldsymbol{y}_1 = \boldsymbol{y}_{initial} \quad (42)$$

$$\boldsymbol{y}_M = \boldsymbol{y}_{target}, \quad (43)$$

are satisfied within an error tolerance. Similarly to the process shown in Listing 9, the problem of directly optimising the trajectory of a dynamical system is outlined in Listing 10, where the *Hermite-Simpson-Separated* transcription `HSS` is used for example.

In the process of trajectory optimisation, often the most severe bottleneck in performance manifests in supplying an initial guess. One of the simplest ways to supply an initial guess is to initialise the nonlinear programme decision vector $\boldsymbol{z}$ randomly within the state and control space, $\mathcal{S}$ and $\mathcal{U}$ respectively, as done in Listing 10. At first this may seem like a sufficient idea; however, if one brings their attention to the problem's dynamic constraints, this will certainly be ineffective, as the randomly initialised variable will not make sense.

```
from Trajectory import Point_Lander
from Optimisation import HSS
from PyGMO import *

# Instantiate the dynamical model
model = Point_Lander()
# Use 20 transcription segments
nsegs = 20
# Use Hermite-Simpson-Separated transcription
problem = HSS(model, nsegs)
# Use sequential least squares quadratic
    programming
algo = algorithm.scipy_slsqp()
# Use also monotonic basin hopping
algo = algorithm.mbh(algo)
# Generate an 2 initial guesses
pop = population(problem, 2)
# Solve the optimisation problem
pop = algo.evolve(pop)
# Extract the solution
z = pop.champion.x
```

Listing 10. Direct Trajectory Optimisation

Examining a two dimensional planetary lander, whose dynamics are governed by the first order system of ordinary differential equations shown in Equation 1, it is quite unsurprising that a random distribution of state nodes will likely prevent the implemented optimiser from converging to a feasible solution. A smarter way to provide a guess that looks more natural, with respect to the system's dynamics, is to provide a ballistic guess, otherwise known as an uncontrolled trajectory, through the use of the `Guess.Ballistic` optimisation instance method.

To supply an initial guess for the direct trajectory optimisation of a dynamical system through a ballistic guess, one simply propagates or numerically integrates the system's dynamics from an initial state to an arbitrary final time t_f, and then samples M evenly distributed nodes along the trajectory to subsequently input to the optimiser. However, one must ensure that all these nodes of the system's state are within the problem's rectangular state space bounds as discussed in Section 1, that is $\boldsymbol{s}_k \forall k \in \{1, \ldots, M\} \in \mathcal{S}$. Using this methodology, by altering Listing 10 slightly, as shown in Listing 11, the convergence of the direct trajectory optimisation problem is improved significantly.

5. DATA GENERATION

In order to arrive at the final objective of being able to implement a neuro-controller to optimally control a user specified dynamical system, under this methodology a large database of optimal control trajectories must first be generated. As mentioned in Section 1, the topological continuation of a system's dynamics may be exploited to enable rapid generation of many optimal control trajectories, in a what is formally known as homotopy, relating two continuous functions from

```
from Trajectory import Point_Lander
from Optimisation import HSS
from PyGMO import *

# Instantiate the dynamical model
model = Point_Lander
# Use 20 transcription segements
nsegs = 20
# Instantiate the problem
problem = HSS(model, nsegs=20)
# Generate ballistic guess from initial state
zguess = prob.Guess.Ballistic(model.si, tf=30)
# Use sequential least squares quadratic
    programming
algo = algorithm.scipy_slsqp()
# Use also monotonic basin hopping
algo = algorithm.mbh(algo)
# Create an empty solution space
pop = population(prob)
# Append the ballistic guess
pop.push_back(zguess)
# Optimise the ballistic trajectory into an
    optimal one
pop = algo.evolve(pop)
# Extract the solution
z = pop.champion.x
```

Listing 11. Ballistic Guess

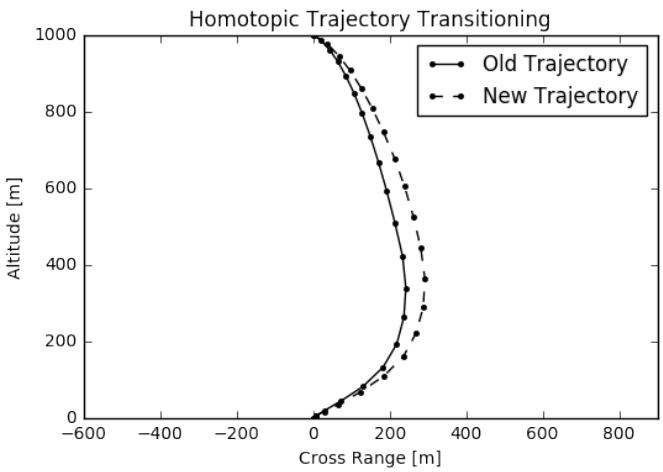

Fig. 1. Homotopic Trajectory Transitioning

topological space to another.

As optimising trajectories of complicated nonlinear systems is rather computationally arduous, the topological continuation of a system's dynamics is exploited in order to render each individual optimisation process expedient. If one generates an optimal trajectory from a particular initial state and then uses the resulting solution to optimise a trajectory from a similar initial state, it would be seen that the optimiser converges incredibly quickly and that the solution is very similar, as shown in Figure 1 for the system of Equation 1.

In practice, in order to exploit the topological continuation of the system's dynamics, *random walks* are performed within user specified boundaries of a dynamical system's possible initial states, `silb, siub`. The random walks are done with a step size with respect to a percentage of each state element's boundary region size, as shown in Listing 12. The random walks of initial states, within the position and velocity boundaries of the lander described in Equation 1, are shown for example in Figure 2 and 3.

From the generated sequence of initial states, generated in order of similarity through the random walk within bounds, the methods of trajectory optimisation discussed in Section 3 and 4 may be employed from each initial state. The optimisation process quickly converges from each subsequent initial state due to their similarities as explained by homotopy. Through this process, a very large database of optimal trajectories can be generated from various initial states.

Now consider for example a planetary lander encounter-

```
def Random_Initial_States(model, mxstep=10.,
    nstates=5000.):
    # The model's initial state boundaries
    silb    = model.silb
    siub    = model.siub
    # The size of the boundary space
    sispace = siub - silb
    # Convert the percent input to number
    mxstep  = mxstep*1e-2*sispace
    # Make array
    states = zeros((nstates, model.sdim))
    # First state in between everything
    states[0] = silb + 0.5*sispace
    for i in arange(1, nstates):
        perturb   = random.randn(model.sdim)*
            mxstep
        # Reverse and decrease the steps of
            violating elements
        states[i] = states[i-1] + perturb
        badj = states[i] < silb
        states[i, badj] = states[i-1, badj] -
            0.005*perturb[badj]
        badj = states[i] > siub
        states[i, badj] = states[i-1, badj] -
            0.005*perturb[badj]
    return states
```

Listing 12. Random Walks

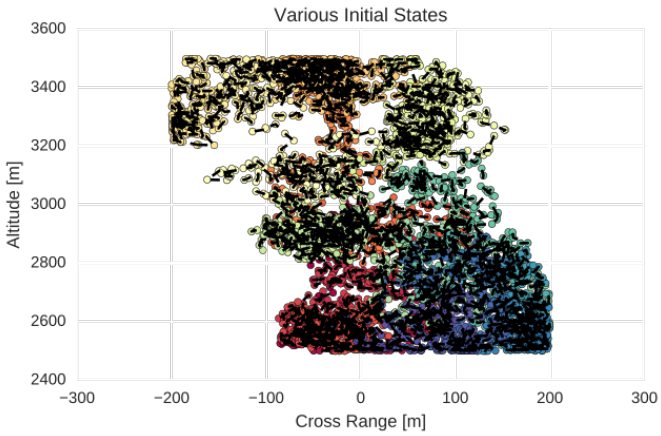

Fig. 2. Initial Positions

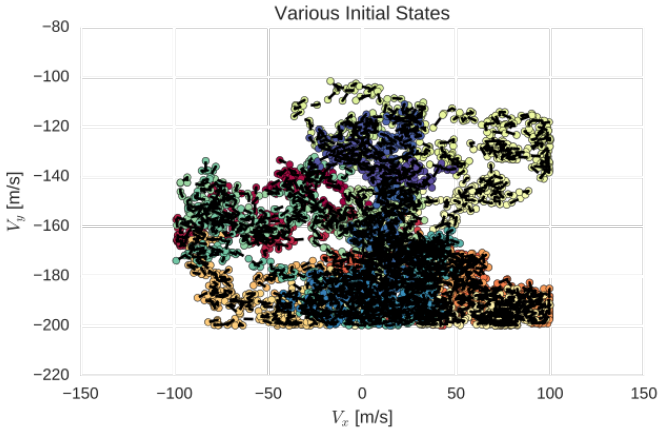

Fig. 3. Initial Velocities

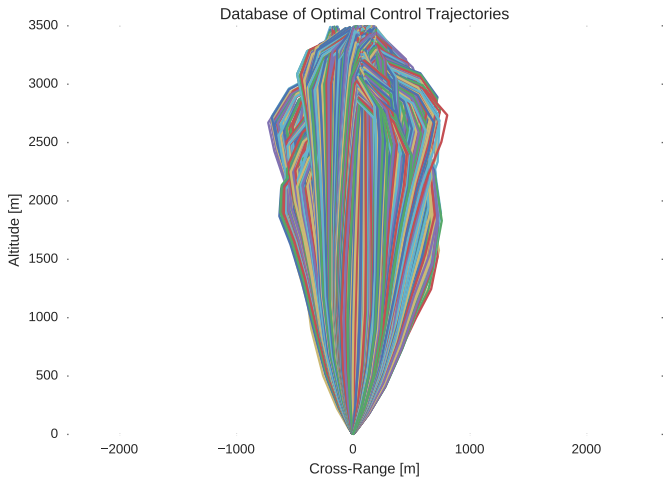

Fig. 4. Database of Optimal Control Trajectories

ing the Martian atmosphere

$$\text{Mars Lander} \begin{cases} \dot{x} = vx \\ \dot{y} = vy \\ \dot{v}_x = \frac{c_1 u}{m} \cos(\theta) - \frac{c_3 v x}{m} v \\ \dot{v}_y = \frac{c_1 u}{m} \sin(\theta) - \frac{c_3 v y}{m} v - g \\ \dot{m} = -\frac{c_1 u}{c_2} \end{cases}, \tag{44}$$

where the system's control is characterised by $u \in [0, 1]$ and $\theta \in [0, \pi]$, the thrust throttle level and direction respectively, and the thrust direction θ is restricted to only point upward, as would be expected from a landing trajectory optimisation solution. The system's set of constant parameters are outlined as

$$\text{Parameters} = \begin{cases} c_1 = T_{max} \\ c_2 = I_{sp} g_0 \\ c_3 = \frac{\rho C_D A}{2} \end{cases}. \tag{45}$$

It is noted that this system's control actions are similar to that of the simple system in Equation 1, and that the system's dynamics prove to be sufficiently difficult in the analytical derivations required for indirect trajectory optimisation methods. As such, using the advantage of the topological continuation of the system's dynamics, the Hermite-Simpson-Separated method is used to generate a large database of optimal control trajectories as shown in Figure 4, with eight thousand generated optimal control trajectories, each with twenty transcription segments enforced by the Hermite-Simpson quadrature and interpolation constraints in Equations 39 and 40. The state-control pairs along each of the generated trajectories are compiled into a single unified dataset, which in the example of Figure 4 forms 176000 samples, well over enough to sufficiently train artificial neural networks of especially deep architectures as seen in Section 6.

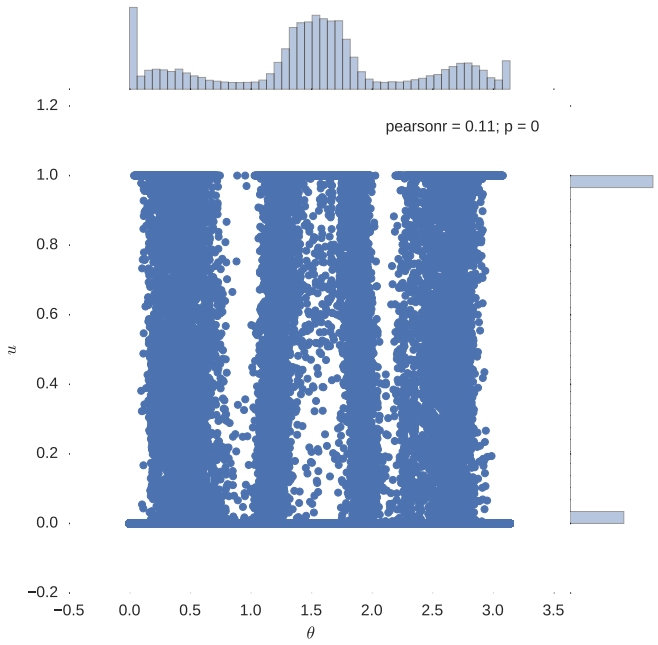

Fig. 5. Joint Distribution of Control Outputs

6. MACHINE LEARNING

Artificial neural networks are a modelling regime motivated by the structure and behaviour of the central nervous system in animals. Neural networks are popularly used for nonlinear modelling, forecasting, regression, and classification, and they quite often surpass the performance of advanced modelling and forecasting methods of other regimes.

The structure of neural networks resemble a horde of connected artificial neurons that serve to emulate local memory which, from each neuron, is disseminated to other neurons. While the exact mechanism by which biological neural networks are *trained* is relatively unknown, in artificial neural networks there typically exists a *training rule* whereby *weights* of connections are iteratively manipulated through experience or by evidence in data. While the mechanism by which these manipulations are done more closely resembles that of numerical optimisation, the process conceptually represents that of which is seen in nature, for example a child learning to categorise the sentiment of their parents' facial expressions. One of the most exciting merits of using artificial neural networks is their ability to generalise outside the domain in which they were trained. That is, a sufficiently trained neural network should be able to successfully infer correct results from unseen cases.

Artificial neural networks have shown a great deal of success in such tasks as classification, regression, and clustering. In this work focus is placed on supervised learning through regression of the type of database described in Section 5. In the process of supervised learning, a *training pattern*, consisting of corresponding input and output data, is attempted to be modelled by a function mapping inputs to outputs. Such a mapping function is represented, in this case, by a neural network, a nonlinear combination of parameters. The successful mapping of input data to output data is typical possible given enough data and computing resources, but especially powerful approximations models are fostered when the architecture of the implemented neural network includes at least one hidden layer.

```
'''
This net has 3 hidden layers.
Layer 1 has 10 nodes,
Layer 2 has 25 nodes,
Layer 3 has 5  nodes.
'''

layers = [10, 25, 5]
...
Neural_Net = MLP(path)
Neural_Net.build(data, iin, iout, layers)
...
```

Listing 13. Building a Neural Network

Artificial neural networks are noted to be among the most effective methods for nonlinear regression. The learning process of neural networks, by which they iteratively learn the correct policy for mapping inputs to outputs, is characterised by the objective of minimising an error function with respect to a set of free parameters. One of such error functions is the commonly used *mean squared error*. In recognising the merit of using deep artificial neural networks for nonlinear regression, in this software framework it has been made convenient to construct neural networks with one or more hidden layers, which find themselves amidst the burgeoning field of *deep learning*. One can easily instantiate a deep artificial neural network, again in a convenient object orientated approach, as shown in Listing 13, with three hidden layers, with each layer having a user specified number of nodes. For example, the visual architecture of a neural network with two hidden layers, each with twenty nodes (20×2), can be seen in Figure 6; and one with four hidden layers, each also with twenty nodes (20×4) in Figure 7.

Although it has long been pointed out that artificial neural networks with hidden layers form powerful models of nonlinear functional relationships, an algorithm to train such a network was, for quite some time, nonexistent. The most significant challenge that prevented the finding of an appropriate training algorithm was that there was no way to judge the error produced within hidden layers; however, after many years the *back propagation* algorithm was spawned.

However, in this work, rather than the widely used back propagation algorithm, the *Adam* method of stochastic optimisation [9] is used. The Adam method is a first-order gradient-based optimisation algorithm of stochastic objec-

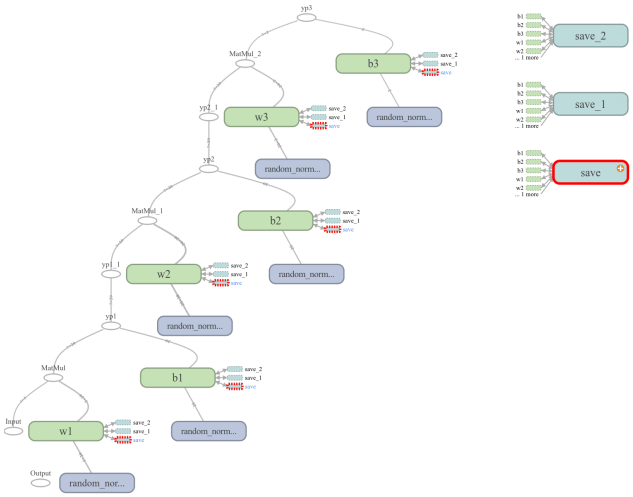

Fig. 6. 20×2 Multilayer Perceptron

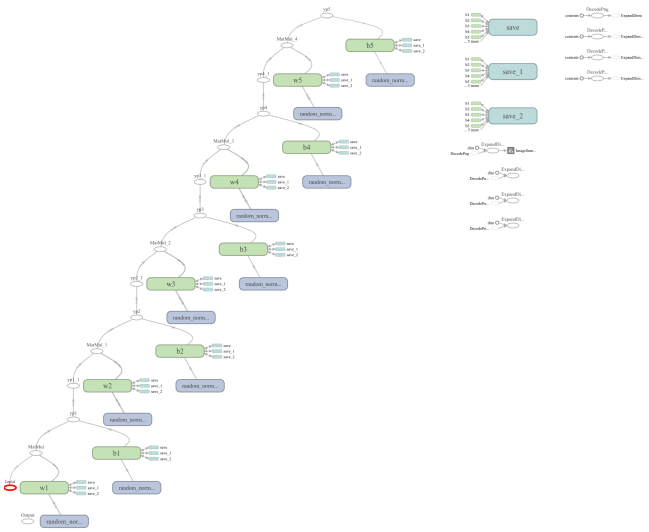

Fig. 7. 20×4 Multilayer Perceptron

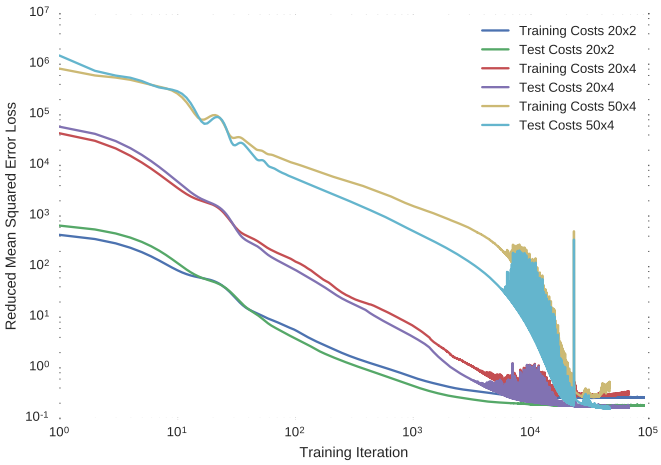

Fig. 8. Training Loss with $\alpha = 1 \times 10^{-2}$

tive functions, based on adaptive estimates of lower-order moments. This method is noted to be easy to use, computationally efficient, have little memory requirements, is invariant to diagonal rescaling of gradients, and is suited well for problems that entail large datasets and/or parameter spaces. Additionally, the need to worry about the behaviour of the *learning rate* of the model is circumvented, as the learning rate is theoretically guaranteed to decay in accordance to

$$\alpha_t = \frac{\alpha}{\sqrt{t}}, \tag{46}$$

where α is the nominal user specified learning rate and t is the training iteration.

This algorithm is often regarded as state of the art in comparison to the back propagation algorithm, and hence its use is justified. Although the decay of the learning rate is taken care of, the nominal learning rate must be selected carefully. In the case of the optimal control trajectory database of Figure 4, the training progress of minimising the mean squared error to the model approximations is shown for multilayer perceptrons of architectures[1] having

- two twenty node hidden layers (20×20)
- four twenty node hidden layers ($20 \times 20 \times 20 \times 20$)
- four fifty node hidden layers ($50 \times 50 \times 50 \times 50$)

at a nominal learning rate of $\alpha = 1 \times 10^{-2}$ in Figure 8 and $\alpha = 1 \times 10^{-4}$ in Figure 9.

Indeed, it is seen that if the learning rate is set too greedily, the training convergence of the cost minimisation becomes rather noisy as in Figure 8, hence improvement is seen with

[1] In this work, for an artificial neural network with an architecture having L number of hidden layers, each with $n_i \forall i \in \{i, \ldots, L\}$ number of nodes, the network's dimensions are expressed in full notation as $n_i \times \cdots \times n_L$, and in short notation as $N \times L$ if $n_i = N \forall i \in \{i, \ldots, L\}$.

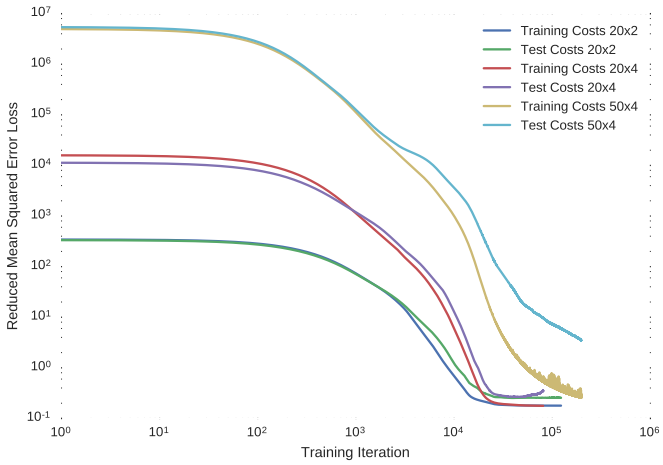

Fig. 9. Training Loss with $\alpha = 1 \times 10^{-4}$

a more conservative learning rate as in Figure 9. Despite this distinction, although a lower learning rate further encourages convergence to the true optimal network parameters, it does however result in a longer training duration, making it infeasible for many domestic computing apparatuses.

It has been shown with multi-layer perceptrons having a single hidden layer that including an excessive number of neurons typically results in *over training*. In the case of over-training, the implemented neural network is unable to generalise sufficiently. Although historically it was commonplace to iteratively manipulate a neural network's number of nodes until a sufficient level of generalisation is achieved, it is now deemed more practical to implement *early stopping*, in which one halts the training of the subject neural network before it begins over-training. Thus at the expense of a more scientifically based method of determining a neural network's structure, a large amount of time is saved by disregarding the neural network's structure above a certain tolerance of sufficiency, determined through trial and error. In this framework, users can interrupt and resume the training of a neural network at any moment without jeopardising the storage of the model's parameters, as such parameters are saved at every training iteration for finalisation or later use. In order to investigate neural networks of various architectures, several multilayer perceptions can be conveniently analysed for the application of optimal control in parallel. Within this software framework, one can train multiple neural network architectures asynchronously, as shown in Listing 14.

The success of using artificial neural networks to learn the optimal control polices of dynamical systems is indeed seen in Figures 8 and 9, with all implemented neural network models plateauing to a mean squared error loss of approximately $e \approx 0.16$ for the system in Equation 44. In particularly deep neural architectures, such as those implemented here, rectified linear units used as nodal activation functions are seen

```
from ML import MLP

def train(width, nlayers):
    ...

    # Instantiate the neural network
    net    = MLP(path)
    # Build the neural network
    net.build(data, iin, iout, layers)
    # Specify the learning rate
    lr     = 1e-4
    # Number of training iterations
    tit    = 200000
    # How often to display cost
    dispr = 300
    # Train the network
    net.train(lr, tit, dispr)
    ...

if __name__ == "__main__":
    nlayers = [1, 2, 3]
    width   = [10, 20]

    # Train asynchronously in parallel
    for args in itertools.product(width, nlayers):
        p = multiprocessing.Process(target=train,
            args=args)
        p.start()
```

Listing 14. Asynchronous Neural Network Training

to indeed avoid the problem of vanishing gradients that come with additional layers, imparted by the saturating tendencies of sigmoidal units. Indeed, it is seen in both Figures 10 and 11, that the expected *bang-bang* policy of optimal control theory is mimicked in accordance to Figure 5, where it is noted that the network having a greater number of hidden layers seems to more coherently replicate the expected policy.

7. NEURO-CONTROL

Assuming that the user's implemented artificial neural network model has been properly trained, it is easy for users to implement their neural network model to directly control their unique dynamical model between a chosen initial and final state. Different neural network architectures are easily examined in their success to meet specified boundary conditions with this framework's object orientated approach. In this section, artificial neural networks of architectures 20×2 and 20×4, are examined in their ability to not only control the user defined dynamical system within the training domain, but also generalise their learnt optimal control policy outside of the region in which they have been trained. The process of implementing trained artificial neural networks to control the user's own dynamical model is outlined in Listing 15.

In verifying the success of each neural network to optimally control the trajectory of a dynamical system, random initial states are typically selected within the 5% dilated box-

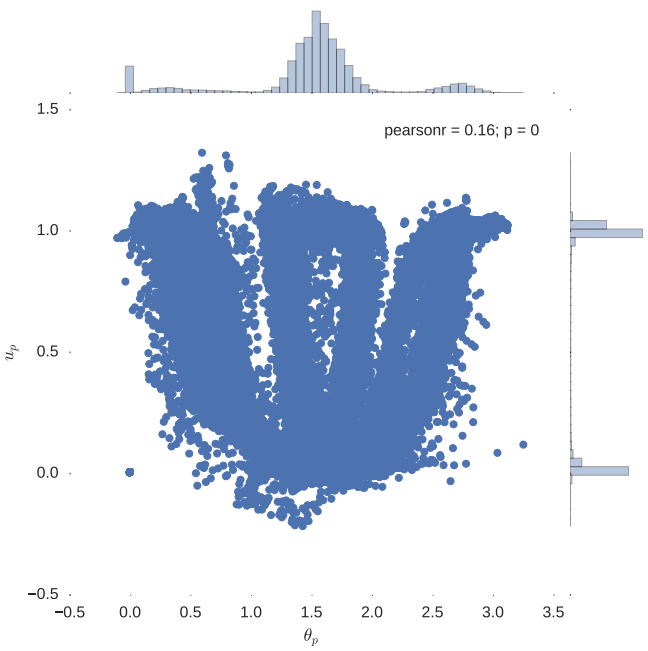

Fig. 10. 20×2 Multilayer Perceptron Predictions

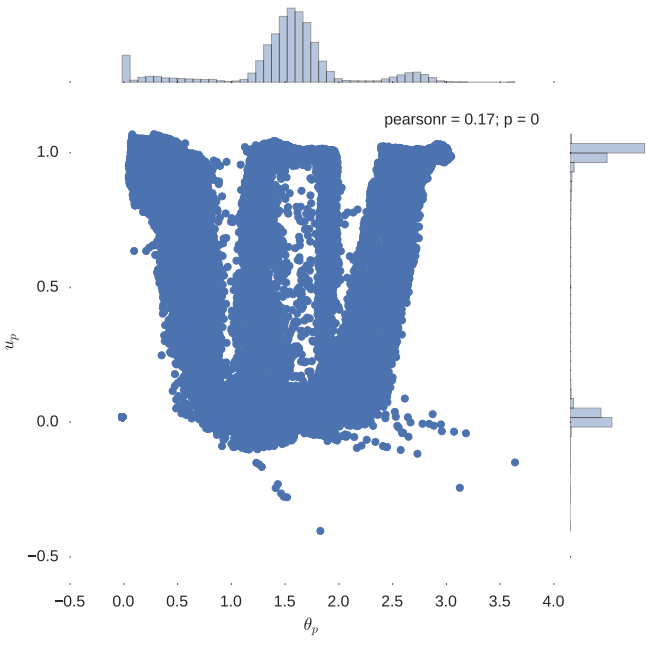

Fig. 11. 20×4 Multilayer Perceptron Predictions

```
# Select randomly a few training trajectories
itraj = np.random.choice(range(ntraj),10)
test_si = data[itraj,0,0:5]
# Instantiate the dynamical system
model = Point_Lander_Drag()
# The name of the trained neural network
net = 'HSS_10_Model'
# Shorthand dimension of the network
dim = (20, 4)
# Assign the neuro-controller to the system
model.controller = Neural(model, net, dim).Control
# Time of simulation
tf = 100
# The resolution of numerical integration
nnodes = 500
# Control the system from several initial states
for si in test_si:
    s, c = model.Propagate.Neural(si, tf, nnodes,
        False)
    t = np.linspace(0,tf, nnodes).reshape(nnodes
        ,1)
    fs = np.hstack((s,c,t))
    fsl.append(fs)
```

Listing 15. Neuro-Control Implementation

bounded region in which the training initial states were generated from, as shown in Listing 15. From the randomly selected initial states the dynamics of the system are then propagated with the control mapping enforced by the implemented neural network. Ten neuro-controlled trajectories are shown for the 20×2 neural network in regards to position in Figure 12 and thrust throttle profile in Figure 13; likewise for the 20×4 network in Figures 14 and 15.

Indeed, it is quite easily seen that the 20×4 network is superior to the 20×2 network in performance. Not only is this evidenced by the network's regression of the training database, but more importantly by its performance with respect to its dynamic implementation as a real-time controller. The difference between the control profile generated from the 20×2 network in Figure 13 and the 20×4 network in Figure 15 is revealing; the 20×4 network is more accurately able to replicate the *bang-bang* optimal control profile that was to be expected in accordance to optimal control theory. This is unsurprising as it is widely indicated in the field of machine learning that the addition of extra hidden layers to an artificial neural network increases significantly its ability to generalise its behaviour to unseen scenarios outside the domain in which it was trained.

Indeed, by simulating the controlled trajectories imparted by the 20×2 and 20×4 networks, the advantage of additional layers is exemplified. Although the optimal control data generated through nonlinear programming is accurate, it is not entirely precise, as there exists so called *chattering effects* due to the discretisation of the system's state-time dynamics. Despite this, while it is not guaranteed that such a neural network's control will prove to be capable of mass-

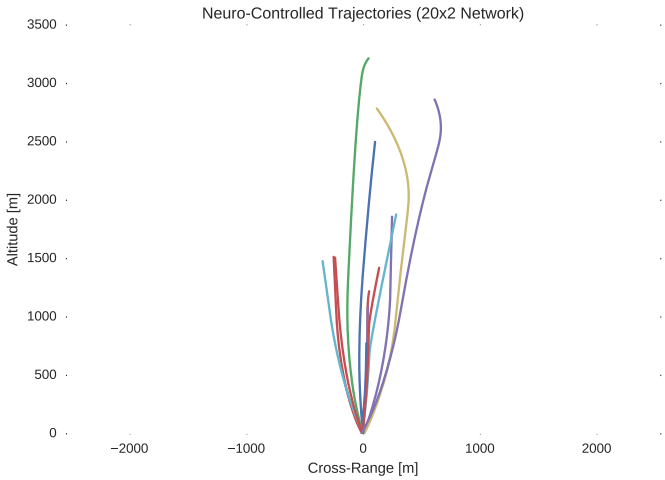

Fig. 12. 20×2 Neuro-Controlled Trajectory

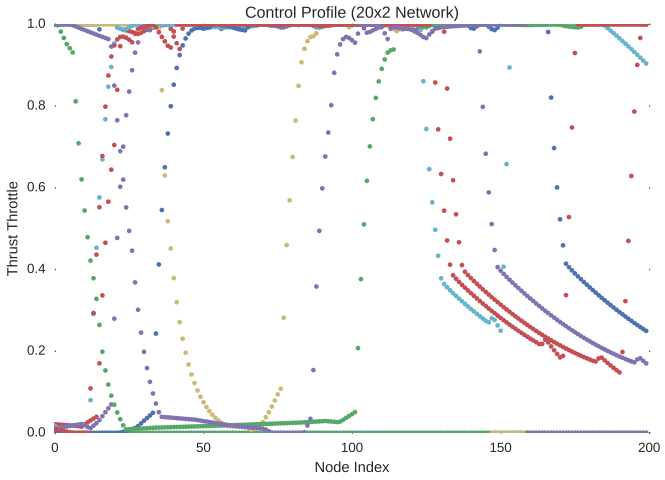

Fig. 13. 20×2 Neuro-Controlled Control Profile

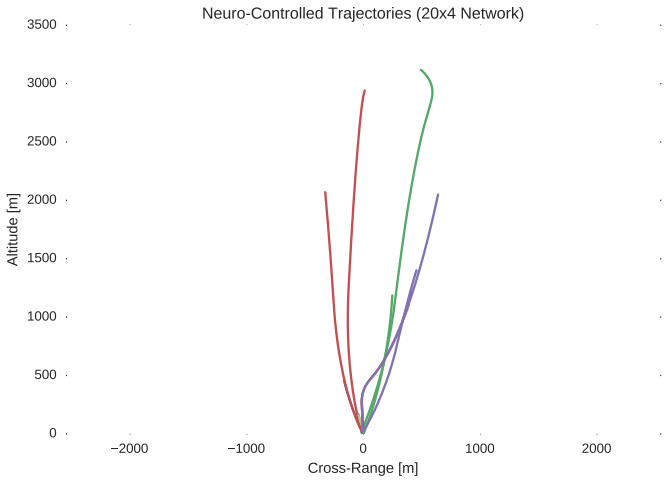

Fig. 14. 20×4 Neuro-Controlled Trajectory

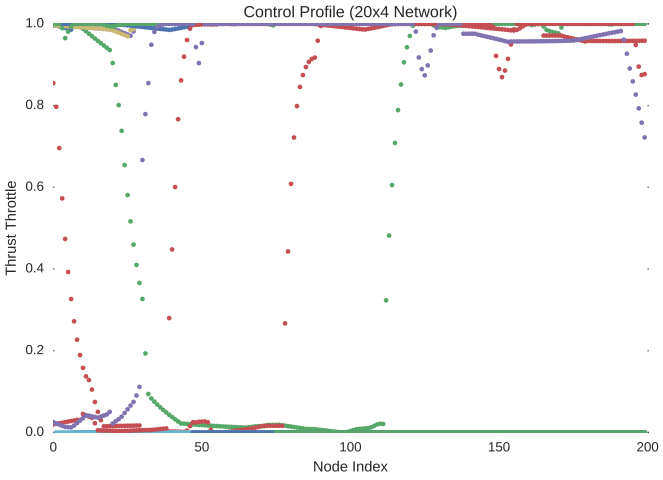

Fig. 15. 20×4 Neuro-Controlled Control Profile

optimal control, it is shown that they would indeed be able to control such a system with sufficient feasibility. That is, although the neuro-controlled dynamical system may not perfectly exploit its environmental dynamics to maximise its savings with respect to fuel expenditure, as would be expected from global trajectory optimisation, it should indeed be able to safely land. Thus, in this respect, such neuro-controllers as those explained in this text, present the significant capability to **autonomously** control such dynamical systems in **real-time**.

8. CONCLUSIONS AND FUTURE WORK

It is shown that neural networks trained on optimal state-control pairs enables the *real-time* implementation of optimal control. It is shown that these *intelligent* controllers may pose a substitution for the popular usage of controllers that used linear dynamics about a nominal trajectory. Using neural networks as controllers enables appropriate control outside of a nominal trajectory, thus making them more robust to anomalies than the currently implemented controllers.

It has been shown in this work that supervised learning poses merit for use in real-time optimal control. Comparing this method to reinforcement learning, it is noted that a significant advantage is that the user does not have to manually assign reward to an environment, which for the application of crucial aerospace control, seems rather arbitrary. Supervised learning requires data, which in new scenarios may not be readily available. However, considering the plethora of data in this now data-driven world, supervised learning proposes a great deal of merit. Considering these points, a strategy of exploiting each regimes merits is sought.

A potential direction of future work is to combine the merits of both reinforcement and supervised learning in what is known as guided policy search [10]. This method essentially

employs reinforcement learning in regular practice; However, instead of manually assigning rewards to an environment, optimal examples are given through trajectory optimisation, hence the benefits of both reinforcement learning and supervised learning are combined, with their drawback being eliminated.

9. REFERENCES

[1] B. Acikmese and S. R. Ploen, "Convex programming approach to powered descent guidance for mars landing," *Journal of Guidance, Control, and Dynamics*, vol. 30, no. 5, pp. 1353–1366, Sep 2007, Accessed on: Apr 20, 2017. [Online]. Available: https://doi.org/10.2514%2F1.27553

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015, Accessed on: Apr 20, 2017. [Online]. Available: http://dx.doi.org/10.1038/nature14236

[3] L. A. Gatys, A. S. Ecker, and M. Bethge, "A neural algorithm of artistic style," *CoRR*, vol. abs/1508.06576, 2015, Accessed on: Apr 20, 2017. [Online]. Available: http://arxiv.org/abs/1508.06576

[4] S. Levine, "Exploring deep and recurrent architectures for optimal control," *CoRR*, vol. abs/1311.1761, 2013, Accessed on: Apr 20, 2017. [Online]. Available: http://arxiv.org/abs/1311.1761

[5] C. Sánchez-Sánchez and D. Izzo, "Real-time optimal control via deep neural networks: study on landing problems," *CoRR*, vol. abs/1610.08668, 2016, Accessed on: Apr 20, 2017. [Online]. Available: http://arxiv.org/abs/1610.08668

[6] L. S. Pontryagin, V. G. Boltyanshii, R. V. Gamkrelidze, and E. F. Mishenko, *The Mathematical Theory of Optimal Processes*. New York: John Wiley and Sons, 1962.

[7] D. Izzo, D. Hennes, L. F. Simões, and M. Märtens, "Designing complex interplanetary trajectories for the global trajectory optimization competitions," in *Space Engineering: Modeling and Optimization with Case Studies*, G. Fasano and J. D. Pintér, Eds. Cham: Springer International Publishing, 2016, pp. 151–176, Accessed on: Apr 20, 2017. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-41508-6_6

[8] J. Betts, *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming*, 2nd ed. Society for Industrial and Applied Mathematics, 2010, Accessed on: Apr, 2017. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/1.9780898718577

[9] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014, Accessed on: Apr 20, 2017. [Online]. Available: http://arxiv.org/abs/1412.6980

[10] S. Levine and V. Koltun, "Guided policy search," in *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, ser. JMLR Workshop and Conference Proceedings, vol. 28. JMLR.org, 2013, pp. 1–9, Accessed on: Apr 20, 2017. [Online]. Available: http://jmlr.org/proceedings/papers/v28/levine13.html