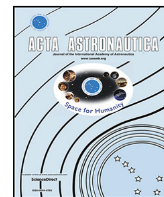
Acta Astronautica 199 (2022) 374–385

Contents lists available at ScienceDirect

Acta Astronautica

journal homepage: www.elsevier.com/locate/actaastro



Enabling intelligent onboard guidance, navigation, and control using reinforcement learning on near-term flight hardware

Callum Wilson *, Annalisa Riccardi

Department of Mechanical and Aerospace Engineering, University of Strathclyde, 75 Montrose Street, Glasgow, G1 1XJ, United Kingdom

ARTICLE INFO

Keywords:
Intelligent control
Reinforcement learning
Edge artificial intelligence

ABSTRACT

Future space missions require technological advances to meet more stringent requirements. Next generation guidance, navigation, and control systems must safely operate autonomously in hazardous and uncertain environments. While these developments often focus on flight software, spacecraft hardware also creates computational limitations for onboard algorithms. Intelligent control methods combine theories from automatic control, artificial intelligence, and operations research to derive control systems capable of handling large uncertainties. While this can be beneficial for spacecraft control, such control systems often require substantial computational power. Recent improvements in single board computers have created physically lighter and less power-intensive processors that are suitable for spaceflight and purpose built for machine learning. In this study, we implement a reinforcement learning based controller on NVIDIA Jetson Nano hardware and apply this controller to a simulated Mars powered descent problem. The proposed approach uses optimal trajectories and guidance laws under nominal environment conditions to initialise a reinforcement learning agent. This agent learns a control policy to cope with environmental uncertainties and updates its control policy online using a novel update mechanism called Extreme Q-Learning Machine. We show that this control system performs well on flight suitable hardware, which demonstrates the potential for intelligent control onboard spacecraft.

1. Introduction

Designing effective spacecraft Guidance Navigation and Control (GNC) systems poses several challenges. These control systems must deal with substantial uncertainties inherent to space missions which is difficult for many conventional methods to handle. One class of methods which can be used for controlling uncertain environments is intelligent control. These methods use theories and architectures from the field of Artificial Intelligence (AI), combined with automatic control and operations research, to devise autonomous control systems that handle uncertainties in a system's environment, goals, or even in the control system itself [1]. This has clear benefits for spacecraft GNC, however, one factor which limits the use of intelligent control onboard spacecraft is its computational cost. Computing systems onboard spacecraft face stringent power budgets, while many AI architectures are computationally intensive.

New advances in edge hardware for AI have increased the possibility for spacecraft onboard intelligence. Several manufacturers now produce high performance Graphics Processing Units (GPUs) that are optimised for running AI algorithms while remaining computationally and physically lightweight. For example, companies such as NVIDIA [2] and Intel [3] are investing substantial resources in developing edge AI capabilities. Due to the availability of this hardware, applications of onboard intelligence are now growing, with most being related to image processing. One recent example uses flight tested hardware to create segmentation maps of floods with the goal of spacecraft being able to downlink these maps in real time to enable quicker disaster response [4]. Applications of onboard intelligence to GNC are more limited and so here we aim to use onboard intelligence to directly train a controller.

Reinforcement Learning (RL) is a class of AI methods that has gained interest for many different applications. These methods learn how to control a system through interaction and optimise their behaviour with respect to a reward function specified by the designer [5]. Recent breakthroughs in AI applied to games have used RL methods, which demonstrates their capabilities on a variety of difficult problems. Most notably, work by DeepMind produced AlphaGo and AlphaZero which overcame the task – previously thought impossible – of beating the world's best humans at the game of Go [6]. More recently, DeepMind have now demonstrated the application of RL for solving the highly complex problem of controlling a nuclear fusion reaction using a real

* Corresponding author.
E-mail address: callum.j.wilson@strath.ac.uk (C. Wilson).

https://doi.org/10.1016/j.actaastro.2022.07.013
Received 31 March 2022; Received in revised form 4 June 2022; Accepted 7 July 2022
Available online 15 July 2022
0094-5765/© 2022 The Authors. Published by Elsevier Ltd on behalf of IAA. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

reactor [7]. In most cases, including those listed previously, a RL agent learns how to control a system through repeated, simulated interactions and then does not update its control policy in operation. While this approach is still effective, it does not fully exploit the adaptive or learning capabilities of intelligent control methods [8]. Our work aims to demonstrate the possibility for including online updates with a RL agent.

Spacecraft landing on extra-terrestrial bodies is an important area of research as further missions to the Moon and Mars are planned [9]. These will have stricter performance requirements than previous missions which motivates the use of novel technologies such as intelligent control. Existing approaches to powered descent guidance use a variety of methods spanning conventional control theories, optimal control, and Machine Learning (ML) [10]. These can broadly be categorised as 'offline', where the control system is designed or trained offline and does not update, and 'online', where the controller updates its control policy online in real time. The earliest examples of automated powered descent guidance algorithms come from the Apollo missions [11]. They used polynomial guidance algorithms that find analytical expressions for the spacecraft's acceleration in time that are used to control its thrusters. Another class of algorithms referred to as Zero Effort Miss (ZEM) and Zero Effort Velocity (ZEV) uses a similar set of assumptions to apollo guidance and calculates analytical expressions for thrust commands. Such methods have been applied to Mars powered descent [12] and other more general spacecraft control problems including asteroid intercept [13]. These conventional control approaches can be considered offline as their guidance laws do not update online.

In cases requiring better performance than that provided by conventional control methods, theories from optimal control are used. This is where a controller aims to maximise or minimise a certain characteristic of a system. In the case of powered descent, the optimality criterion is often to minimise fuel consumption. There are various optimisation methods that can solve the minimum fuel powered descent problem offline for specific initial conditions. When designing control systems that are to be optimised online, there are additional challenges. As with intelligent control methods, computational resources onboard spacecraft are a significant limitation. In addition, some optimisation methods, such as Non-Linear Programming (NLP) approaches, are not guaranteed to converge on a solution, which is necessary for mission critical operations [10]. One class of approaches for optimal powered descent guidance uses convex programming methods [14]. These approaches solve the optimal control problem online by convexifying any nonconvex constraints, such as thrust and glideslope requirements [15,16]. To account for more general nonconvex problems, such as the non-linear relationships between aerodynamic disturbances and the spacecraft state, methods of successive convexification are also used. In this case, the optimisation algorithm iteratively generates solutions to convex approximations of the nonconvex problem using linearisation [17,18].

ML methods can also be used in conjunction with the theoretical guarantees of optimal control to provide faster solutions that are more robust to uncertainties. For example, in [19] the authors train a Neural Network (NN) to approximate optimal control actions for planetary landing without the need to run online optimisations. Similarly, in [20] they reduce the optimal control problem to defining a small set of parameters that vary based on the initial conditions. Again, NNs are trained to learn the mapping from initial conditions to optimal control policy. As well as providing the mapping from states to optimal actions, in [21] NNs are also used to model the gravity dynamics of an asteroid for generating optimal trajectories. Since the performance of many optimisation algorithms is sensitive to the initial guess, NNs can also learn to generate suitable initial guesses. This is the approach employed in [22] for generating optimal asteroid landing trajectories.

Due to its performance in other domains, RL has also gained a lot of interest as an approach to spacecraft GNC problems. The most commonly used RL method for these applications is Proximal Policy Optimisation (PPO) [23]. This algorithm performs well in high-dimensional tasks with continuous actions but requires longer training times than similar discrete-action methods. RL has been used to solve a variety of trajectory optimisation problems, including interplanetary trajectories and orbit transfers. These often use RL as a means for providing greater robustness to uncertainties, such as in [24] where the authors include noise in various aspects of a low-thrust Earth–Mars trajectory problem. Other works also investigate using RL for low-thrust manoeuvres applied to transfers between Earth–Moon Lyapunov orbits [25,26]. In all these applications PPO is used to train the NN control policy. This is also the case in [27] where PPO is compared to a direct behavioural cloning approach for a rendezvous problem. In a behavioural cloning framework, the agent learns to take actions based on expert demonstrations; similar to the ML and optimal control approaches detailed above. This approach can also be used to create an initial policy for training. The authors in [28] use solutions to a deterministic optimal control problem to initialise a RL agent in a supervised manner prior to training. Here we employ a similar method for using optimal control solutions as demonstrations.

RL has also been successfully applied to planetary powered descent guidance in other previous works. The OpenAI Gym set of benchmark environments even has a highly simplified 2D lunar lander environment that only starts from a small range of initial conditions and incorporates limited uncertainties [29]. For more realistic powered descent problems, previous works also used optimal control demonstrations for initialising an agent [30] as well as during training [21]. When considering a larger problem space, training agents from scratch becomes more challenging as they might never reach the desired final state. This makes careful tuning of the reward function necessary as demonstrated in [31] where the authors train an agent on a 6-DOF problem using PPO. Other approaches to training agents for powered descent also combine RL with previously described analytical methods such as ZEM/ZEV guidance. In this framework, RL has been used to adapt parameter values [32] and also to guide the spacecraft between determined waypoints [33].

The applications of RL described thus far involved entirely offline training of the control policy. In an intelligent control framework, the controller can also update online, as is the goal of our work. Online adaptive methods of RL for powered descent are less common, however there are some examples that employ meta-learning methods. In this paradigm, the agent's policy is paramaterised as a recurrent NN that learns how to perform well on new tasks with limited training examples [34]. Recent works used such an approach, combined with PPO, for near asteroid GNC applications [35,36] and lunar landing [37]. Here we employ a different approach to online updating by adapting the NN output weights using conventional RL update rules. Online updates can be desirable for highly uncertain environments, but have certain limitations. Updating a control scheme online creates an additional computational burden for the spacecraft compared to ML methods that only perform inference online. In addition, it can be difficult to ensure the stability of ML based control methods such that a spacecraft will not enter a dangerous state. This is more challenging for methods that adapt online. Other works have addressed the problem of 'safe' RL using Lyapunov-based updates [38]. Furthermore, Lyapunov-based control laws derived using RL have been applied to spacecraft transfer problems [39].

We apply RL to a three degree-of-freedom Martian spacecraft powered descent problem with environmental uncertainties. This builds on previous work that improved the training time for this problem using discrete action spaces [40] and by incorporating optimal control demonstrations [41]. As shown above, there are many traditional control approaches used for spacecraft powered descent. However, these often rely on assumptions on the system. For example, assuming uncertainties are bounded or that the model of the system is sufficiently accurate. The aim of intelligent control is to control systems where

these assumptions are no longer valid. In extra-terrestrial powered descent problems, there are very few examples of successful landings which leaves a great deal of uncertainty in future missions. In this work, we did not investigate all possible uncertainties in this application, but aim to make progress towards more intelligent controllers that can be implemented onboard spacecraft.

The primary contribution of this work is the demonstration of online updates of a pretrained RL agent performed on near-term flight hardware. Online updates use a novel update mechanism called Extreme Q-Learning Machine (EQLM) [42]. This uses theories from Extreme Learning Machines (ELMs) [43] to update NN parameters without using conventional gradient-based methods. EQLM is capable of training a NN without iterative tuning making it suitable for an online learning environment. To demonstrate the feasibility of using this approach onboard spacecraft, we ran our agent on the NVIDIA Jetson Nano 2 GB developer kit [44]. This small, single board computer incorporates a NVIDIA Maxwell GPU that is designed for edge AI applications. The NVIDIA Jetson system has existing flight heritage onboard a recent Lockheed mission that demonstrated various onboard AI applications.[1] At the problem scale considered here, the use of a GPU is not necessarily beneficial for computation time. Nevertheless, use of this hardware allows the agent to scale to larger networks as required, for example, in RL performing end-to-end training with image inputs. With this work, we aim to show that intelligent control is feasible onboard near-term flight hardware for spacecraft GNC.

2. Problem statement

This section details the spacecraft lander powered descent problem to which we apply the RL agent. Here we introduce the mathematical models and parameters describing the environment and present the problem statements for optimal control and RL.

2.1. Environment

The environment we consider is a three degree-of-freedom powered descent for Mars. For this case with no rotations, the spacecraft is considered to have thrusters aligned with its body-frame axes in an orthogonal configuration as opposed to thrust vectoring. The equations of motion of the spacecraft are shown in Eqs. (1) to (3):

$$\frac{d}{dt}(\mathbf{x}) = \dot{\mathbf{x}} \tag{1}$$

$$\frac{d}{dt}(\dot{\mathbf{x}}) = \frac{\mathbf{F}_{thrust} + \mathbf{F}_{env}}{m} + \mathbf{g} \tag{2}$$

$$\frac{d}{dt}(m) = -\frac{\Sigma_{m=i,j,k}|F_m|}{I_{sp} \cdot g_0} \tag{3}$$

where $\mathbf{x} = \{x_i, x_j, x_k\}$ m is the spacecraft's position with respect to the desired landing location, $\mathbf{F}_{thrust} = \{F_i, F_j, F_k\}$ N is the force exerted by the thrusters on the spacecraft, $\mathbf{F}_{env}$ is the disturbance forces from the environment, m is the spacecraft total mass, $\mathbf{g} = \{0, 0, -3.72\}$ N/kg is the acceleration due to gravity for Mars, $I_{sp} = 210$ s is the specific thrust of each thruster, and $g_0 = 9.81$ N/kg is the reference acceleration due to gravity. Within the defined coordinate system, the desired landing position is at $\mathbf{x} = \{0, 0, 0\}$ and the unit vectors $\vec{i}$, $\vec{j}$, and $\vec{k}$ are positioned in the downrange, crossrange, and altitude directions respectively.

Possible thrusts are discretised within the ranges $-F_i^{max} \le F_i \le F_i^{max}$, $-F_j^{max} \le F_j \le F_j^{max}$, and $0 \le F_k \le F_k^{max}$. Maximum thrust magnitudes are $F_i^{max} = F_j^{max} = 10$ kN and $F_k^{max} = 13$ kN. The number of discrete actions in each direction is 7 in both the i- and j-directions and 3 in the k-direction, giving an action space size of 147 discrete actions. Values for maximum thrusts and number of discrete actions are selected based on results of previous studies [41]. Environmental forces are randomly sampled every 5 timesteps from a normal distribution with mean 0 N and standard deviation 100 N and linearly interpolated between samples. The control system's sampling time is 0.2 s.

2.2. Optimal control problem

When generating optimal trajectories, we only consider 'nominal' conditions where the environmental disturbances, $\mathbf{F}_{env}$ are zero. These disturbance forces are then included when training and testing the RL agent. Eq. (4) states the optimisation problem and relevant constraints:

$$\begin{aligned}
\text{minimise} \quad & \sum_{t=0}^{t_f} \left(\sum_{m=i,j,k} F_m^2 \right) \\
\text{subject to} \quad & \mathbf{x}|_{t=t_f} = 0, \ \dot{\mathbf{x}}|_{t=t_f} = 0, \\
& \mathbf{x}|_{t=0} = \mathbf{x}_0, \ \dot{\mathbf{x}}|_{t=0} = \dot{\mathbf{x}}_0, \ m|_{t=0} = m_0, \\
& \mathbf{F}_{thrust}^{min} \le \mathbf{F}_{thrust} \le \mathbf{F}_{thrust}^{max}.
\end{aligned} \tag{4}$$

where t_f is the time of flight. This parameter and the thrusts, $\mathbf{F}_{thrust}$ at each control point are the control variables to be optimised. In the spacecraft mass model used here (Eq. (3)), the minimum energy formulation given in Eq. (4) is equivalent to the minimum fuel problem.

2.3. Reinforcement learning problem

In a general RL problem, an agent seeks to maximise its cumulative discounted reward G_t, which is also referred to as return. The return following timestep t is defined by Eq. (5).

$$\begin{aligned}
G_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots \\
&= \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}
\end{aligned} \tag{5}$$

The parameter γ is the discount factor which takes a value in the range $0 \le \gamma \le 1$. This value controls the extent to which long term rewards affect the return. For $\gamma = 0$, only the reward received at the next timestep is considered and as $\gamma \to 1$ more rewards at future timesteps become significant.

The reward function defining the reward r given at each timestep is chosen by the control system designer to reflect the objective of the system. Appropriate definition of the reward function is crucial to ensure the agent learns a desirable policy. This problem benefits from a shaped reward function which effectively guides the agent towards more optimal actions since it is otherwise difficult for the agent to learn to find the landing site. The reward function we use here is the same as from previous work [41] which is adapted from the reward function used by Gaudet et al. [31]. This function is shown in Eq. (6):

$$\begin{aligned}
r =& \alpha \left\| \dot{\mathbf{x}} - \dot{\mathbf{x}}_{targ} \right\| + \beta \left\| \frac{\mathbf{F}_{thrust}}{\mathbf{F}_{thrust}^{max}} \right\| + \eta \\
&+ \kappa \left(r_z < 0.01 \ \text{ and } \ \|\mathbf{x}\| < x_{lim} \ \text{ and } \ \|\dot{\mathbf{x}}\| < \dot{x}_{lim} \right)
\end{aligned} \tag{6}$$

$\dot{\mathbf{x}}_{targ}$ is a target velocity that the agent is rewarded for following. The shape of this target velocity is defined by Eqs. (7) to (12):

$$\dot{\mathbf{x}}_{targ} = -v_0 \left(\frac{\hat{\mathbf{r}}}{\|\hat{\mathbf{r}}\|} \right) \left(1 - exp\left(-\frac{t_{go}}{\tau} \right) \right) \tag{7}$$

$$v_0 = 70 \tag{8}$$

$$t_{go} = \frac{\|\hat{\mathbf{r}}\|}{\|\hat{\mathbf{v}}\|} \tag{9}$$

$$\hat{\mathbf{r}} = \begin{cases} \mathbf{x} - [0 \ 0 \ 15], & \text{if } x_3 > 15 \\ [0 \ 0 \ x_3], & \text{otherwise} \end{cases} \tag{10}$$

$$\hat{\mathbf{v}} = \begin{cases} \dot{\mathbf{x}} - [0 \ 0 \ -2], & \text{if } x_3 > 15 \\ \dot{\mathbf{x}} - [0 \ 0 \ -1], & \text{otherwise} \end{cases} \tag{11}$$

[1] Lockheed Martin and University of Southern California Build Smart CubeSats, La Jument, Media - Lockheed Martin (Aug. 2020). URL https://news.lockheedmartin.com/news-releases?item=128962 (accessed 29/9/2021).

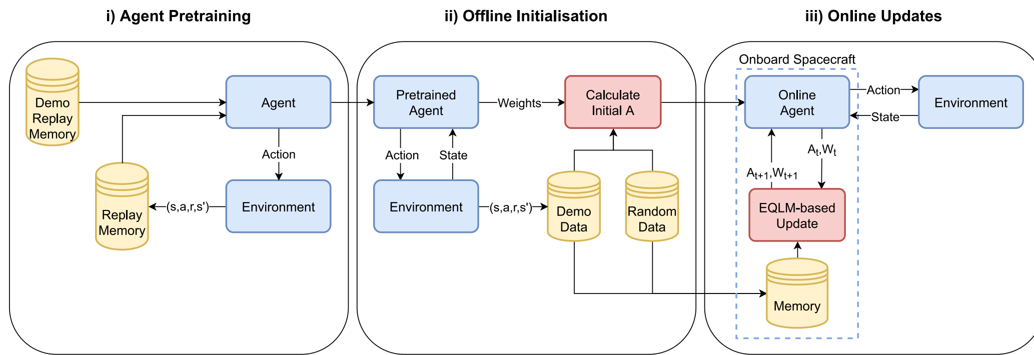


Fig. 1. Schematic illustration of the three main phases in our proposed method.

$$\tau = \begin{cases} 20, & \text{if } x_3 > 15 \\ 100, & \text{otherwise} \end{cases} \tag{12}$$

This motivates the agent to follow a velocity pointing towards 15 m above the desired landing zone which decreases as it approaches this point. Then over the final 15 m of descent the target velocity is entirely normal to the surface in the k-direction. The constants α, β, η, and κ in Eq. (6) weight different parts of the reward function. The α term is a negative reward that increases as the difference between the spacecraft's velocity and target velocity increases. The β term is also a negative reward for the control effort normalised with respect to the maximum thrust. η is a positive constant that motivates the agent to keep advancing in the environment. Finally, κ is the reward gained for a successful landing within the limits of $x_{lim} = 5$ m and $\dot{x}_{lim} = 2$ m/s. The values of the constant coefficients in the reward function are $\alpha = -0.01$, $\beta = -0.05$, $\eta = 0.01$, and $\kappa = 10$.

Here we only considered constraints on the terminal state and thrust magnitudes in both the optimal control and RL problems. In the RL problem, the constraint on the terminal state is only given implicitly to the agent as a positive reward for a successful landing (the κ term above). Additional path and state constraints are typically incorporated as negative rewards for a RL agent. This was the approached used in [31]. Other methods for enforcing constraints on an agent's policy have been proposed [45,46], but are not investigated in this work.

As well as defining the reward function, a state representation must be chosen for the agent. We use the spacecraft's position, velocity, and mass. These values are scaled by a constant factor $\mathbf{s}_{scale}$ to avoid having excessive values input to the agent. The state, $\mathbf{s}$ is then defined by Eq. (13):

$$\mathbf{s} = \{\mathbf{x},\ \dot{\mathbf{x}},\ m\} \cdot \mathbf{s}_{scale} \tag{13}$$

with the values for scaling each variable defined as follows:

$$\mathbf{s}_{scale} = \{0.01,\ 0.01,\ 0.01,\ 0.1,\ 0.1,\ 0.1,\ 0.005\} \tag{14}$$

3. Methods

Fig. 1 gives an overview of our proposed approach, which can be divided into three main parts. The first two parts occur offline for initialising the agent and the final part occurs online. First an agent is trained using conventional gradient descent methods to determine initial weights for the Q-network. We use the conventional gradient-based updates for this part instead of EQLM since the Q-networks have multiple layers, but ELM updates only single layers. The Q-network weights learned from this part are then used to initialise offline the matrix used for EQLM updates. Finally, the agent updates its output weights online.

3.1. Offline pretraining

In this part of the method we train an agent using the Deep Q-Networks (DQN) algorithm with optimal control demonstrations as described in previous work [41]. One of the key features of the DQN algorithm is experience replay [47], where the agent stores its observations in a replay memory and samples from this when updating Q-network parameters. Since updates are independent of the policy being followed, this allows the use of demonstration data for updates. These demonstrations come from solving the optimal control problem defined by Eq. (4) for different initial conditions. When performing updates, the agent uses experiences from both this 'demo' replay memory and the agent's own replay memory.

There are several suitable methods for solving the optimal control problem and here we use a Sequential Least Squares Programming (SLSQP) solver [48]. The optimal solutions give demonstrations of successful landings without environmental disturbances, but they use a continuous action space. The DQN algorithm requires discrete actions and so the optimal control demonstrations must be transformed into a discrete action space before they can be used for training. The process for discretising the actions is described in [41].

Items in each of the replay memories consist of a state, action, observed reward, and observed state, denoted $(s_j, a_j, r_{j+1}, s_{j+1})$. The agent computes estimates of the action-value function Q using a neural network with parameters θ. In the DQN algorithm, an additional target network with parameters θ^T computes action-value targets for updating. These target action-values, denoted $\mathbf{t}_j$, are calculated as shown in Eq. (15). At each update step, targets are calculated for a minibatch of k experiences, where $k = k_{replay} + k_{demo}$ and k_{replay} and k_{demo} are the number of experiences sampled from the agent's replay memory and the demonstration memory, respectively.

$$\mathbf{t}_j = \begin{cases} r_j, & \text{if } s_{j+1} \text{ is terminal} \\ r_j + \gamma \max_a Q_{\theta^T}(s_{j+1}, a), & \text{otherwise} \end{cases} \tag{15}$$

The temporal-difference error, $\mathbf{e}_j$ can then simply be calculated as the difference between the estimated action value and target action value:

$$\mathbf{e}_j = Q_\theta(s_j, a_j) - \mathbf{t}_j \tag{16}$$

This error is then used to update the parameters of the Q-network via gradient descent. For the problem scale we consider here, a multilayer Q-Network is required to sufficiently approximate the action-value function. EQLM is used to update single layer Q-networks with random initial weights and biases. Training a Q-Network initially using gradient descent allows EQLM updates to be performed on just the output layer and instead of random initial weights, the other network parameters are tuned to give a useful representation to the final layer. Following pretraining, all the Q-Network parameters are fixed except for the output weights which will be updated online.

3.2. Offline EQLM initialisation

Prior to applying EQLM online, it is first necessary to initialise variables for performing updates. Here we describe the process for calculating initial values for the matrix $A^{\dagger}$ which is used for performing online updates with EQLM. The input to the Q-network is the environment state, denoted $\mathbf{s}$. The output of the ith hidden layer, denoted $\mathbf{y}_i$, is:

$$\mathbf{y}_i = \left(f_i \circ f_{i-1} \circ \cdots \circ f_1\right)(\mathbf{s}) \tag{17}$$

where f_i is a nonlinear function of the previous layer's output resulting in each output being a composite function of all previous layers. For an initial minibatch of k inputs $(\mathbf{s}_1, \ldots, \mathbf{s}_k)$, we define the matrix $\mathbf{H}$ as the output of the final hidden layer (denoted $\mathbf{y}$ for simplicity) for each of the inputs as shown:

$$\mathbf{H} = \begin{bmatrix} \mathbf{y}^{\top}|_{\mathbf{s}=\mathbf{s}_1} \\ \vdots \\ \mathbf{y}^{\top}|_{\mathbf{s}=\mathbf{s}_k} \end{bmatrix}_{k\times\tilde{N}} \tag{18}$$

where $\tilde{N}$ denotes the number of nodes in the final hidden layer. Each row of $\mathbf{H}$ contains the final hidden layer output $\mathbf{y}$ for each of the inputs $(\mathbf{s}_1, \ldots, \mathbf{s}_k)$. The output weights of the network are denoted with β. Matrix $\mathbf{T}$ is then defined as the corresponding target action-values for the inputs $(\mathbf{s}_1, \ldots, \mathbf{s}_k)$ as shown in Eq. (19). These targets are calculated using Eq. (15).

$$\mathbf{T} = \begin{bmatrix} \mathbf{t}^{\top}|_{\mathbf{s}=\mathbf{s}_1} \\ \vdots \\ \mathbf{t}^{\top}|_{\mathbf{s}=\mathbf{s}_k} \end{bmatrix}_{k\times m} \tag{19}$$

where m denotes the number of discrete actions. For a set of output weights which perfectly represents the targets $\mathbf{T}$ for a given $\mathbf{H}$, the Q-network model can be written as shown in Eq. (20):

$$\mathbf{H}\beta = \mathbf{T} \tag{20}$$

As stated previously, following pretraining only the output weights β are updated. In the case where EQLM is used without pretraining, network parameters are randomly assigned and the output weights are initialised as shown in Eq. (21):

$$\beta_{t=0} = A^{\dagger}_{t=0}\mathbf{H}^{\top}\mathbf{T} \tag{21}$$

where the matrix $A^{\dagger}$ that is used for online updates can be initialised using Eq. (22):

$$A^{\dagger}_{t=0} = \left[\frac{I}{\tilde{\gamma}} + \mathbf{H}^{\top}\mathbf{H}\right]^{\dagger} \tag{22}$$

However, following pretraining we can instead use the optimised set of output weights to initialise $A^{\dagger}$ by rearranging Eq. (21) as shown:

$$A^{\dagger}_{t=0} = \beta_{t=0}\left[\mathbf{H}^{\top}\mathbf{T}\right]^{\dagger} \tag{23}$$

where $\beta_{t=0}$ are the optimised output weights. $\mathbf{H}$ and $\mathbf{T}$ are both calculated for a large batch of experiences to perform this step. Similarly to the pretraining, these experiences are sampled from two replay memories: one obtained using the pretrained agent's policy and one using a random policy. The total minibatch size for initialisation is then $k = k_{agent} + k_{random}$ where k_{agent} and k_{random} are the number of experiences sampled from the agent's replay memory and random replay memory, respectively.

3.3. Online updating

With the weights and matrix for updating weights initialised, the agent can update its weights online. As with the initialisation part, the online updates use data from agent demonstrations and random actions. In addition to these replay memories, the experiences observed online by the agent also make up the data used to update. These experiences are all fed to the agent in minibatches, whose size is again denoted k. If the agent has already updated from N experiences and samples a new batch of k experiences $(\mathbf{s}_N, \ldots, \mathbf{s}_{N+k})$ for updating, the new incremental matrices $\mathbf{H}_{IC}$ and $\mathbf{T}_{IC}$ can be defined as follows:

$$\mathbf{H}_{IC} = \begin{bmatrix} \mathbf{y}^{\top}|_{\mathbf{s}=\mathbf{s}_N} \\ \vdots \\ \mathbf{y}^{\top}|_{\mathbf{s}=\mathbf{s}_{N+k}} \end{bmatrix}_{k\times\tilde{N}} \tag{24}$$

$$\mathbf{T}_{IC} = \begin{bmatrix} \mathbf{t}^{\top}|_{\mathbf{s}=\mathbf{s}_N} \\ \vdots \\ \mathbf{t}^{\top}|_{\mathbf{s}=\mathbf{s}_{N+k}} \end{bmatrix}_{k\times m} \tag{25}$$

Using these matrices, the online EQLM updates are performed with the following equations:

$$K_t = I - A^{\dagger}_t\mathbf{H}^{\top}_{IC}\left(\mathbf{H}_{IC}A^{\dagger}_t\mathbf{H}^{\top}_{IC} + I_{k\times k}\right)^{\dagger}\mathbf{H}_{IC} \tag{26}$$

$$\beta_{t+1} = K_t\beta_t + K_tA^{\dagger}_t\mathbf{H}^{\top}_{IC}\mathbf{T}_{IC} \tag{27}$$

$$A^{\dagger}_{t+1} = K_tA^{\dagger}_t \tag{28}$$

where K_t is another matrix used to calculate weight updates. These updates occur after a certain number of timesteps, n_{step} in the environment. At each update, the agent's previous n_{step} experiences are all used to update. This gives a total minibatch size for online updates of $k = n_{step} + k_{agent} + k_{random}$. In practice, these online updates would occur onboard the spacecraft. As an additional step prior to deployment of the agent and following initialisation, the agent carries out further training episodes with online updates. These update the output weights β and matrix $A^{\dagger}$ before they are used on the hardware with the goal of improving the online agent's performance.

4. Simulation and results

Here we describe the simulation setups for training and testing agents and present results from offline experiments and online testing. The offline results consist of a hyperparameter study and offline testing of various random seeds to obtain the best agent. The hardware used for testing online updates is the NVIDIA Jetson Nano 2 GB developer kit. It is equipped with a 64-bit Quad-core ARM A57 CPU at 1.43 GHz and a 128-core NVIDIA Maxwell GPU. The onboard memory for the developer kit is 2 GB 64-bit LPDDR4. The methods detailed previously were implemented in the Tensorflow Python library which makes it possible to use GPU acceleration.

4.1. Simulation setup

Table 1 shows the range of initial conditions used in simulations, where each state variable is uniformly sampled from the range shown at the start of every episode. The initial phase of agent pretraining using gradient descent methods and optimal demonstrations ran for 10,000 episodes. Table 2 shows the hyperparameters that were fixed for all pretraining runs. The network has three hidden layers with *tanh* activation functions and initial weight updates used the RMSProp algorithm. The parameter n_{ϵ} defines the number of episodes over which the rate of exploration ϵ decreases. This value of exploration probability is in the range $0 \leq \epsilon \leq 1$ and defines the probability of the agent selecting a random action at each timestep. γ is the discount factor (shown in Eq. (15)), which has a different value for offline pretraining and online updates. n_T is the number of timesteps between target network updates.

Following pretraining, the next phase initialised the agent for online EQLM updates. The initial minibatch size k used to initialise $A^{\dagger}_{t=0}$ was fixed at 2000 for all experiments with $k_{agent} = 1000$ experiences sampled from the agent demonstrations and $k_{random} = 1000$ from random actions. In addition to the fixed hyperparameters in Table 2, the value of γ for

Table 1
Range of initial conditions for training and testing agents.

State variable	Min.	Max.
i-position	0.4 km	1.1 km
j-position	−1.1 km	1.1 km
k-position	2.4 km	2.6 km
i-velocity	−75 $\mathrm{ms^{-1}}$	−5 $\mathrm{ms^{-1}}$
j-velocity	−35 $\mathrm{ms^{-1}}$	35 $\mathrm{ms^{-1}}$
k-velocity	−95 $\mathrm{ms^{-1}}$	−65 $\mathrm{ms^{-1}}$
Mass	2000 kg	2000 kg

Table 2
Fixed hyperparameters for agent pretraining (from [41]).

Parameter	Value
Hidden units	(300,200,300)
Learning rate	1×10^{-5}
n_ϵ	4000
γ (offline)	0.926
n_T	65

Table 3
Values of hyperparameters evaluated in the hyperparameter study.

Parameter	Values
ϵ_0	0.4, 0.6, 0.8
k (offline)	100, 120, 150
n_{step}	4, 8, 16
k (online)	40, 80, 120

Table 4
Pareto-optimal hyperparameter configurations. Config. numbers are arbitrary to refer to each configuration.

Config.	ϵ_0	k (offline)	n_{step}	k (online)
1	0.6	120	8	40
2	0.8	100	4	40
3	0.8	150	8	80

Table 5
Summary statistics of maximum terminal position and velocity across 8 runs for the three pareto-optimal hyperparameter configurations. Config. numbers refer to the set of hyperparameters from Table 4.

Config.	Position (m)				Velocity (m/s)			
	mean	s.t.d.	min.	max.	mean	s.t.d.	min.	max.
1	371.17	627.72	22.48	1933.12	32.46	26.87	13.55	89.41
2	700.17	1580.57	23.20	4871.42	28.52	26.08	5.75	92.46
3	213.12	246.30	43.25	742.18	34.01	23.75	7.09	78.19

Table 6
Summary statistics of terminal position and velocity over 500 episodes for the three pareto-optimal pretraining runs with the best selected hyperparameter configuration. Run numbers are arbitrary to refer to each of the pareto-optimal points.

Run	Position (m)				Velocity (m/s)			
	mean	s.t.d.	min.	max.	mean	s.t.d.	min.	max.
1	11.77	3.18	4.80	23.20	5.45	4.87	0.25	15.07
2	10.59	5.73	0.97	29.44	3.54	1.23	0.75	9.00
3	19.66	4.67	7.28	33.30	2.06	0.99	0.31	5.75

EQLM updates was fixed at $\gamma = 0.9$. The range of initial conditions used for both offline and online EQLM training were the same as in pretraining, which are shown in Table 1. Offline EQLM updates ran for 500 episodes prior to testing the online updating agent.

4.2. Hyperparameter study

Since the performance of the learned policy is sensitive to the hyperparameters used in training, we initially ran a study on a subset of the hyperparameters to find those that gave the best performance. Values of the fixed hyperparameters shown in Table 2 were selected based on previous hyperparameter studies [40,41]. In this study we tested the values of 4 different hyperparameters: initial exploration probability ϵ_0, minibatch size k for offline updates, number of steps between online updates n_{step}, and minibatch size k for online updates. The offline minibatch size comprises experiences from optimal control demonstrations and the agent's replay memory with the same number of experiences sampled from each, i.e. $k_{replay} = k_{demo} = \frac{1}{2}k$. Similarly, for the online minibatch size the same number of experiences were sampled from agent demonstrations and random memory such that $k_{agent} = k_{random} = \frac{1}{2}(k - n_{step})$.

Table 3 shows the values of hyperparameters tested in this study. We performed a grid search over the 3 different values of each hyperparameter giving a total of 81 different configurations. Each configuration ran 8 times with different random seeds. Fig. 2 shows the pretraining learning curves for all values of ϵ_0 and offline k. All the pairs of hyperparameters converged to a similar mean cumulative reward by the end of the training period, but the most notable differences can be seen in the earliest episodes of training. While lower values of ϵ_0 showed a steeper initial increase in rewards, these also had lower mean rewards over the first few episodes than for higher values of ϵ_0. This suggests that taking more greedy actions was harmful to the agent's performance at the start of training. Varying the offline minibatch size over the range shown here gave little difference in the learning curves.

Each configuration of hyperparameters performed 500 offline updating episodes and 500 test episodes with online updates. We assessed the performance of each agent based on their maximum terminal position, $\left\|\mathbf{x}|_{t=t_f}\right\|$ and maximum terminal velocity, $\left\|\dot{\mathbf{x}}|_{t=t_f}\right\|$ across the 500 test episodes. These values were averaged across the 8 random seeds to obtain the measures of performance for each configuration of hyperparameters. Using two cost functions to assess agent performance results in a set of pareto-optimal points with associated optimal hyperparameters. In this case, 3 points were non-dominated with respect to the costs described above. The hyperparameters associated with these points are listed in Table 4.

Table 5 describes the performance of each of these points. The statistics shown describe the distribution of maximum position and velocity across the 8 different runs for each configuration, where the mean values are the relevant costs. The larger values of maximum terminal position and velocity come from agents that fail to learn a successful landing and terminate the episode after the maximum number of steps. Configuration 2 has a particularly large variation in terminal position across random seeds, however, it also has the lowest minimum and mean values for terminal velocity. For this reason, we selected this configuration of hyperparameters, which are the following: $\epsilon_0 = 0.8$, $k(\text{offline}) = 100$, $n_{step} = 4$, $k(\text{online}) = 40$. All further experiments use these hyperparameters and those specified in Table 2.

4.3. Pretraining and initialisation

After performing the hyperparameter study, we selected the best performing set of pretrained weights. We assessed the performance similarly to the hyperparameter study using the worst-case position and velocity across the 500 test episodes for each random seed. Table 6 summarises the performance of the 3 pareto-optimal pretrained weights. These clearly show the trade-off between the position and velocity costs, where an agent with lower mean terminal velocities has higher mean terminal positions and vice versa. While testing agent initialisations, we used all three of the pareto-optimal sets of pretrained weights to consider this trade-off further.

In addition to the set of pretrained weights, both the initialisation of the matrix $A^{\dagger}_{t=0}$ and the offline EQLM updates involve a stochastic process that affects the final performance of the agent. Therefore, to obtain the best performing online updating agent, we tested a number of agents using the same set of pretrained weights but with different

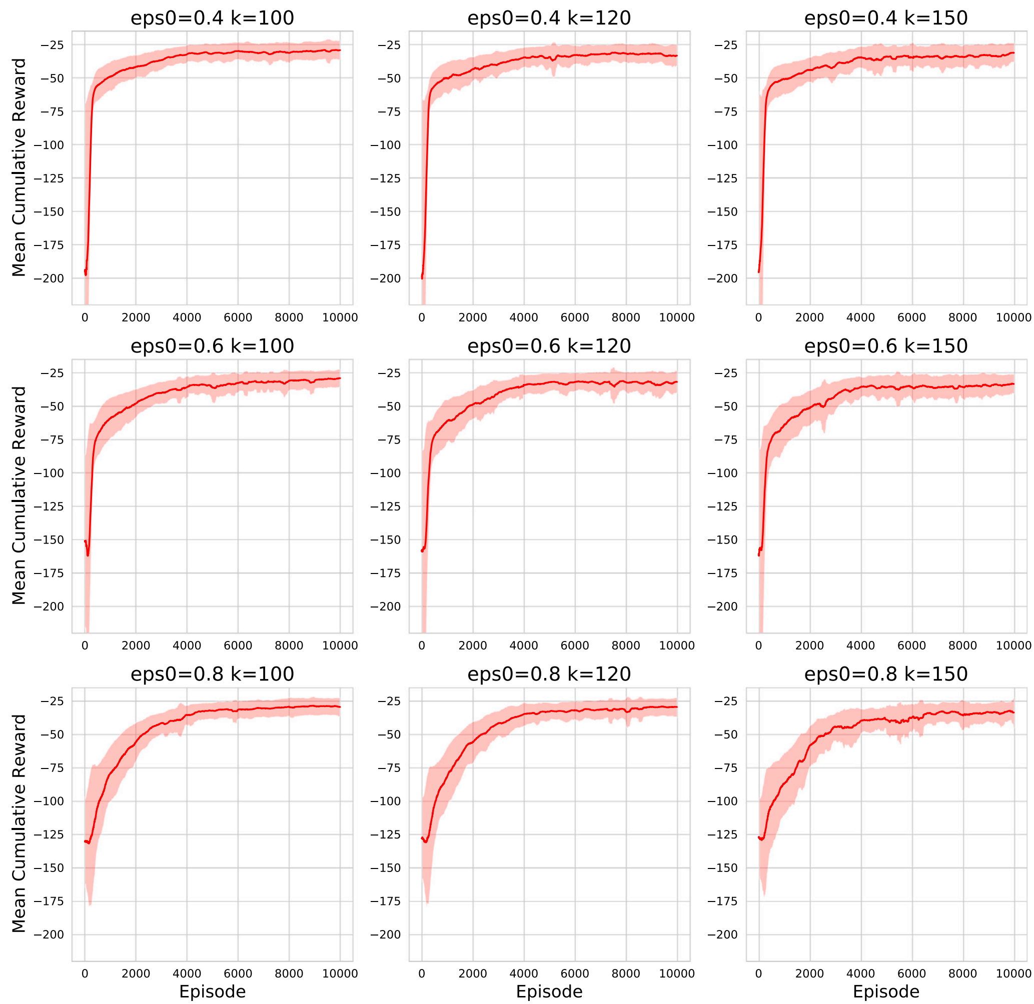


Fig. 2. Mean learning curves across 8 random seeds for each configuration of pretraining hyperparameters. Shaded region indicates ±1 standard deviation.

Table 7
Summary statistics of terminal position and velocity for the pareto optimal initialisation seeds with the best selected hyperparameter configuration and pretrained weights. Pre. run numbers refer to pretraining runs from Table 6 and init. run numbers are arbitrary to refer to each of the pareto-optimal points.

Pre.	Init.	Position (m)				Velocity (m/s)			
run	run	mean	s.t.d.	min.	max.	mean	s.t.d.	min.	max.
1	1	10.71	3.88	1.43	24.52	4.84	3.89	0.02	14.16
1	2	10.72	4.12	0.95	19.42	10.07	5.15	0.44	19.41
1	3	11.51	4.16	1.26	20.17	6.71	5.47	0.19	18.61
3	1	18.79	5.35	10.69	40.49	1.72	0.71	0.18	4.44
3	2	17.98	4.08	9.06	30.79	1.75	0.84	0.34	5.56
3	3	18.43	3.58	6.85	32.33	2.02	0.87	0.22	4.97

random seeds for initialisation and offline updates. We ran 32 different random seeds for each set of pretrained weights. As previously, we assessed the performance based on worst-case terminal position and velocity. Table 7 summarises the performance at the pareto-optimal points.

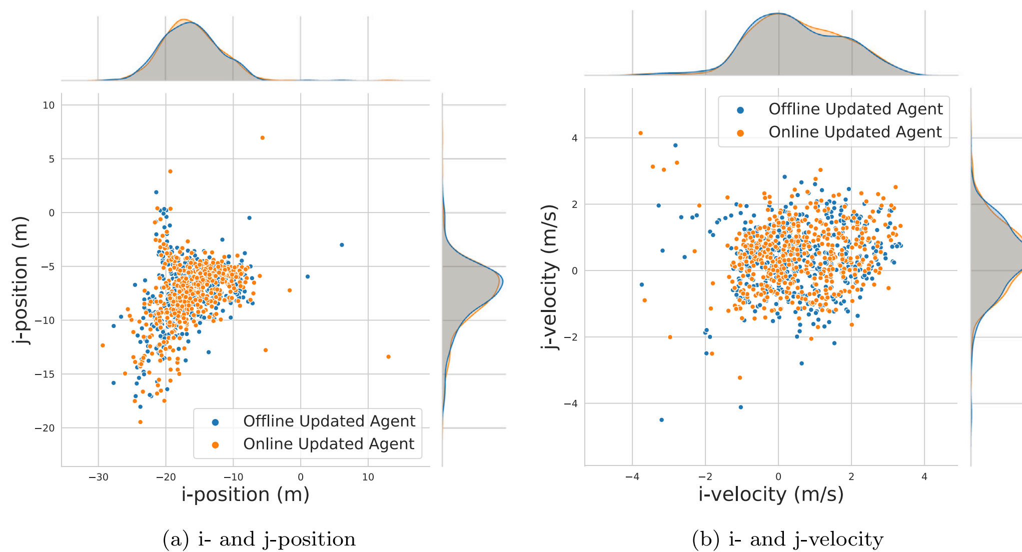
Only pretraining runs 1 and 3 had pareto optimal points. Here we will refer to each run by the tuple of its pretraining run and initialisation run, e.g. (3,1) refers to the fourth row in Table 7. Similarly to the results in Table 6, we see that pretraining run 1 performs better in terms of position and run 3 performs better in terms of velocity. This difference in performance is clearest in the maximum terminal velocity, with a lowest value of 4.44 m/s for run (3,1) and highest value of 19.41 m/s for run (1,2). When choosing which seed to use for online testing, we deemed the terminal velocity to be the most important cost to minimise. Given the initialisation runs from pretraining run 1 performed poorly with respect to this cost, we only considered those from pretraining run 3. In terms of mean terminal velocity, these three agents performed very similarly. While run (3,2) had the highest maximum terminal velocity of the three, its mean terminal velocity was only 0.03 m/s higher than that of run (3,1). In addition, run (3,2) gave the lowest values of mean and maximum terminal position out of these runs. For these reasons, we selected run (3,2) for online tests.

4.4. Online updates

The remainder of the results come from experiments run on the Jetson Nano hardware. Three different agents were run on the hardware to test their performance:

- *Pretrained agent* - using the fixed weights of the pretrained agent without any online updating.
- *Offline updated agent* - using the pretrained agent plus 500 episodes of offline updates without any online updating.
- *Online updated agent* - using the pretrained agent plus 500 episodes of offline updates and with online updating.

The pretrained agent uses the weights from pretraining run 3 (Table 6) of the agent with the best hyperparameters. Both the offline and online updated agents use the weights following 500 episodes of offline updates from run (3,2) (Table 7). In the case of the offline updated agent, these weights are fixed during testing whereas the online



(a) i- and j-position

(b) i- and j-velocity

Fig. 3. Distributions of terminal states for offline- and online-updated agents over 500 episodes.

Table 8
Statistics of terminal states from 500 episodes run on Jetson Nano hardware for three different configurations of agent.

Agent	Position (m)				Velocity (m/s)			
	mean	s.t.d.	min.	max.	mean	s.t.d.	min.	max.
Pretrained	49.22	182.40	6.99	1509.36	3.57	5.47	0.22	42.43
Offline updated	18.07	4.32	6.04	31.95	1.81	0.86	0.35	5.77
Online updated	18.07	4.26	7.41	31.85	1.79	0.85	0.21	5.84

Table 9
Statistics of fuel consumption from 500 episodes run on Jetson Nano hardware for three different configurations of agent. For the pretrained agent episodes, results are split between the episodes that did land and those that did not and terminated after 500 steps.

Agent	Fuel consumption (kg)			
	mean	s.t.d.	min.	max.
Pretrained (no landing)	770.8	12.1	753.9	791.5
Pretrained	498.5	26.7	436.1	688.4
Offline updated	486.7	24.9	434.1	641.2
Online updated	486.7	25.1	438.9	659.2

updated agent also performs EQLM updates during testing. Table 8 summarises each agent's performance over the 500 episodes.

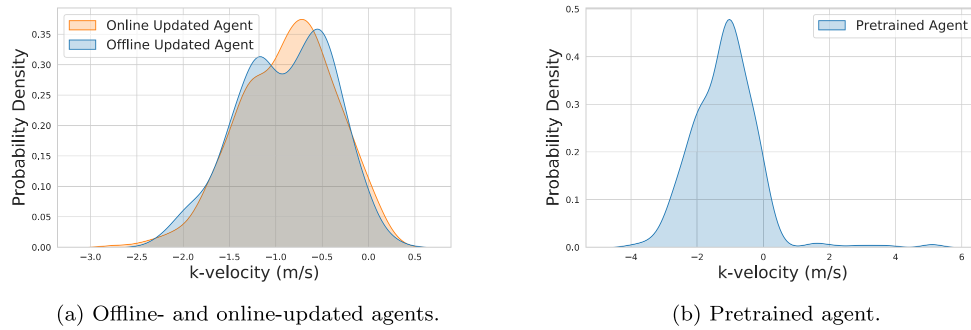
The distributions of final states for each of the offline and online updated agents are shown in Figs. 3(a), 3(b), and 4(a). From Fig. 3(b), we see the terminal velocities in the i- and j-directions of both agents are clustered close to the desired terminal velocity of 0 m/s. Some episodes still have terminal velocities outwith the desired range, particularly towards the negative i-direction. The distributions of velocities for both agents seen here are very similar. Considering the terminal positions in Fig. 3(a), we see that the agent tends to land off-centre towards the negative i- and j-directions. This is likely a consequence of selecting the random seeds that compromised performance in terms of terminal position in favour of lower terminal velocities. As with the terminal velocities, both agents have similar distributions of terminal position. Fig. 4(a) shows the distribution of velocities in the normal direction for both agents. In most cases, the terminal velocity in this direction is less than 2 m/s. Again we see a similar distribution in both agents, however the offline updated agent has an extra peak in its probability density at around -1.2 m/s.

Looking at the summary statistics of each agent's performance in Table 8, this highlights the similar performance of the offline and online updated agents. Both agents have the same mean terminal position and only 0.02 m/s difference in mean terminal velocity. Worst case performance for both position and velocity is also very similar with a difference of only 0.07 m/s.

Compared to the agents updated using EQLM, the pretrained agent performed very poorly. The distributions of final states for this agent are shown in Figs. 5(a), 5(b), and 4(b). In 9 of the 500 testing episodes, this agent failed to land and terminated the episode after 500 steps. These are highlighted in the cluster of points in Fig. 5(a) that are far from the desired landing site. In the episodes where this agent did successfully land, its terminal velocities were still on average significantly larger than the EQLM updated agents as shown in Table 8. This highlights the benefit of updating the output weights using EQLM as it can greatly improve the performance of a pretrained agent. It should also be noted that the selection of pretrained weights was evaluated using the performance of the online updated agent. Therefore, it is likely that these pretrained weights are not those that perform best without online updates. Nevertheless, in this case the EQLM updates gave a substantial improvement in performance with respect to the pinpoint soft landing goal.

Table 9 shows the performance of each agent with respect to the goal of minimising fuel consumption. In the case of the pretrained agent, we distinguish between the 9 episodes where it did not land and the rest of the episodes. Clearly, the largest fuel consumption comes from these episodes where the pretrained agent did not land. From the rest of the agents, both the offline and online updated agents have the same mean fuel consumption of 486.7 kg. The offline updated agent has a slightly lower minimum and maximum fuel consumption compared to the online updated agent, but only by 1.1% and 2.8% respectively.

Fig. 6 shows one example trajectory obtained using the online updated agent. This agent starts from an initial position of $\mathbf{x} = \{0.79, -0.06, 2.43\}$ km and initial velocity of $\dot{\mathbf{x}} = \{-68.4, 31.0, -91.0\}$ m/s. Its trajectory lasts 65.6 s and its terminal position is 14.88 m from the desired location with a terminal velocity of 1.62 m/s. Over this episode, the fuel consumption was 469.7 kg. In addition to the trajectory and thrust profile, Fig. 6(d) shows how the magnitude of the output weights vary over this episode. The overall trend shows the difference in weights from the initial weights gradually increased during online updates. The total change at the end of the episode was approximately 4×10^{-7}. This is significantly less than the average weight magnitude of approximately 6×10^{-2}, which shows that on average most weights remained at similar values. On the other hand, the maximum change in weights across this episode was approximately 1.6×10^{-3}, which



(a) Offline- and online-updated agents. (b) Pretrained agent.

Fig. 4. Distributions of terminal k-velocities for each of the agents over 500 episodes.

(a) i- and j-position. (b) i- and j-velocity.

Fig. 5. Distributions of terminal states for the pretrained agent over 500 episodes.

suggests some weights varied more significantly. A notable feature of the control profile is the jitter in thrust commands shown in Fig. 6(c). Assuming a minimum thrust duration of 0.01 s [49], this profile could be realised by the thrusters. However, regular switching of thrust commands could affect the system's performance and dynamic effects of the thrusters are more likely to be significant. This simplified model still shows desirable performance that motivates further analysis with more realistic models of the overall system and thrusters.

4.5. Update times

The final results we show are the time taken to update the agent when running on the Jetson Nano to get an indication of how feasible this approach is on spacecraft hardware. Fig. 7 shows histograms of the time taken to select an action and perform EQLM updates over 500 steps. Action selection always takes less than 0.01 s. Weight updates take longer and are mostly in the range 0.04 – 0.07 s. On a few occasions these updates last slightly longer but always less than 0.1 s. Since the sampling time in this environment is 0.2 s, this shows that updates can be performed online sufficiently quickly using near-term flight hardware. These update times could also be reduced by further optimising the code for updating weights, if necessary.

To explore the update times on flight hardware further, we analysed the effect of different hardware configurations on the update times and compared these to pure software-in-the-loop simulations on a PC. The PC hardware used here ran Ubuntu 18.04 with a 3.6 GHz Intel i7-4790 CPU and 8 GB RAM. We also restricted the Jetson Nano to only using the CPU to see the effect of using the GPU. Table 10 shows the results for each hardware configuration. Clearly the PC times are fastest and on average approximately 10× faster than either configuration of the Jetson Nano. It is interesting to note that in this case the use of a GPU results in slower updates. This is likely caused by the communication time between CPU and GPU, which is more significant for this relatively small size of network.

Table 10
Timings for weight updates across different hardware configurations.

Device	Update duration (ms)			
	mean	s.t.d.	min.	max.
PC	5.06	2.58	3.03	22.86
CPU	47.5	6.95	40.92	88.6
CPU+GPU	51.16	5.04	44.86	93.06

As shown, at the scale we considered here GPU acceleration did not improve the speed of this control system. Use of a GPU becomes more beneficial for larger network sizes. Therefore, we tested the update times of NNs with the same number of layers but different numbers of hidden nodes - $(N, \frac{2}{3}N, N)$ in each layer, where $N = 300$ is the size used in previous experiments. Fig. 8 shows the variation in update times for different values of N. At the smaller network sizes, the CPU updated slightly faster on average without the GPU. For networks with more than 450 output weights, GPU acceleration did provide a speed-up in update times in this example.

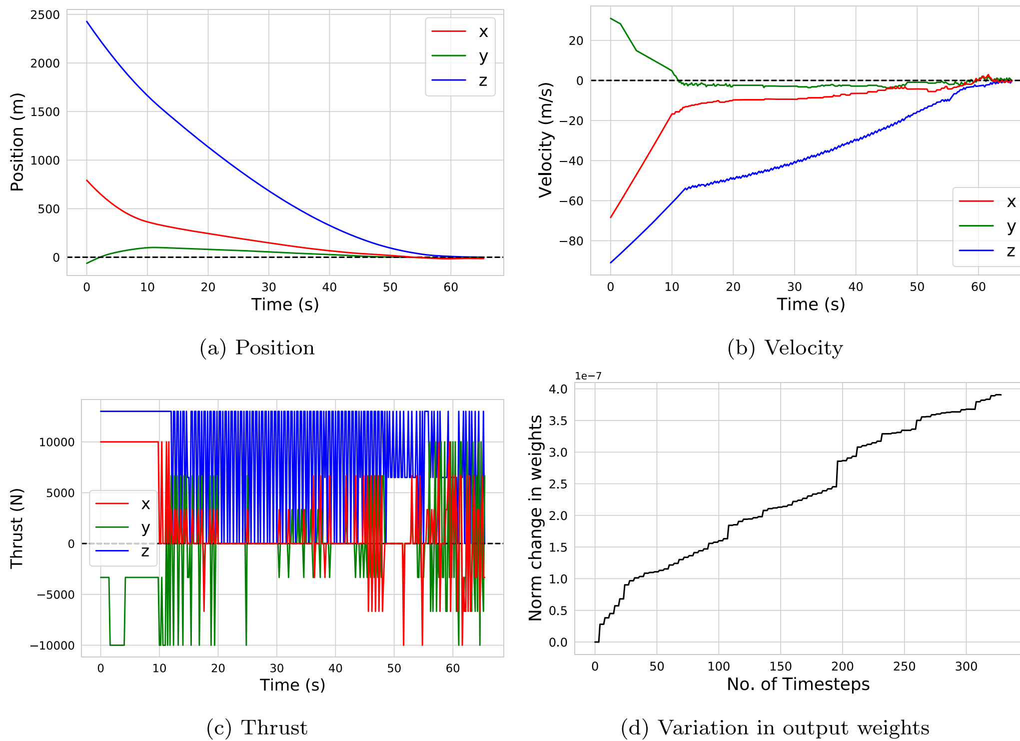


Fig. 6. Example spacecraft trajectory from the online-updated agent. Dashed lines indicate desired final state of position and velocity equal to zero. 'x', 'y', and 'z' refer to i-, j-, and k- directions respectively. Variations in output weights are calculated as norm of the difference in weights from the start of the episode divided by the number of weights.

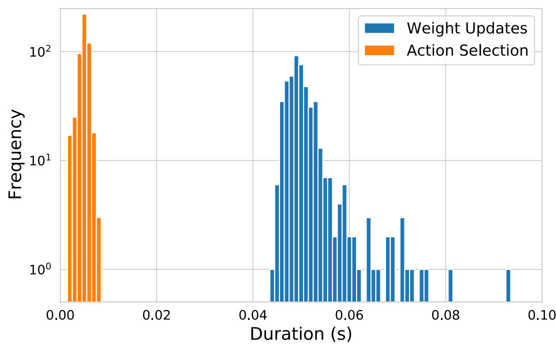


Fig. 7. Histogram of time to update the agent's weights online.

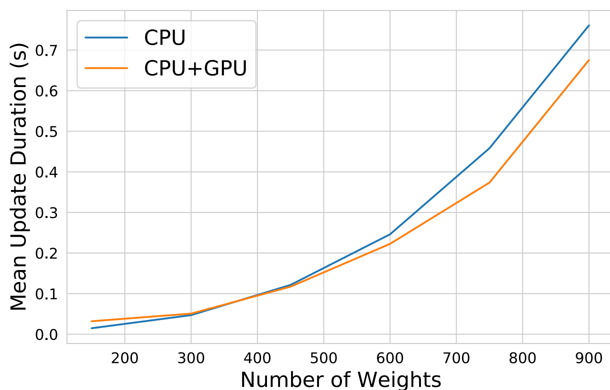


Fig. 8. Variation in update times with network size for each hardware configuration.

5. Conclusions

This work demonstrates the potential for intelligent control methods to be implemented on flight suitable hardware for onboard spacecraft control. Using a RL based approach, we were able to train an agent to solve a powered descent problem subject to environmental uncertainties. This RL training was supplemented with optimal control demonstrations. Once pretrained offline, the agent could also perform weight updates online with new information. The fast update times for this approach as implemented on near-term flight hardware show its suitability for implementation onboard a spacecraft.

When comparing the performance of agents updated using EQLM to the pretrained agent, in this case we observed an improvement in performance from adding the EQLM updates. This improvement mainly came from the offline updates, while the online updated agent had comparable performance to that of the offline updated agent. Given the online updates used the same data collected offline with limited online experiences, this does not give a great deal of potential for improving the agent's performance online. Furthermore, the environment modelled here only incorporated uncertainties in disturbing forces. Future work should investigate greater levels of uncertainty in more of the system's components, as well as uncertainties not modelled in offline training, to determine the merits of this approach.

Here we mainly considered feasibility of the approach in terms of hardware performance. In addition to constraints on the update time of the controller, other factors affect the feasibility of this method for use in a real mission. Space missions undergo strict verification and validation processes which are often difficult to apply to ML methods. For example, the approach used here relied heavily on testing across random seeds to optimise performance. While this can find effective configurations for the agent, in practice this would be an unreliable process that cannot be easily verified without extensive offline simulation. A more formal approach to improving the agent's performance which is agnostic to random seeds would be beneficial. In addition, the limited mass memory onboard spacecraft must be carefully considered.

The approach used here would require offline experiences to be stored onboard the spacecraft, which is undesirable for limiting the agent's memory requirements. These experiences could be replaced with a system model that estimates future states and uses these to update the control system, thereby reducing the need for additional onboard memory and potentially helping the agent adapt better to environmental uncertainties. This approach of online updating RL remains low Technology Readiness Level (TRL), however future developments in these areas could see the eventual use of such methods for space missions.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

Supported by European Space Agency (Contract Number: 4000124916/18/NL/CRS/hh).

References

[1] C. Wilson, F. Marchetti, M. Di Carlo, A. Riccardi, E. Minisci, Classifying intelligence in machines: A taxonomy of intelligent control, Robotics 9 (3) (2020) 64, http://dx.doi.org/10.3390/robotics9030064.

[2] M. Janakiram, NVIDIA ups the ante on edge AI with jetson AGX orin, Forbes (2022).

[3] J. Wu, Edge AI is the future, intel and udacity are teaming up to train developers, Forbes (2020).

[4] G. Mateo-Garcia, J. Veitch-Michaelis, L. Smith, S.V. Oprea, G. Schumann, Y. Gal, A.G. Baydin, D. Backes, Towards global flood mapping onboard low cost satellites with machine learning, Sci. Rep. 11 (1) (2021) 7249, http://dx.doi.org/10.1038/s41598-021-86650-z.

[5] R.S. Sutton, A.G. Barto, Reinforcement Learning: An Introduction, second ed., in: Adaptive Computation and Machine Learning Series, The MIT Press, Cambridge, Massachusetts, 2018.

[6] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, D. Hassabis, Mastering the game of go without human knowledge, Nature 550 (7676) (2017) 354–359, http://dx.doi.org/10.1038/nature24270.

[7] J. Degrave, F. Felici, J. Buchli, M. Neunert, B. Tracey, F. Carpanese, T. Ewalds, R. Hafner, A. Abdolmaleki, D. de las Casas, C. Donner, L. Fritz, C. Galperti, A. Huber, J. Keeling, M. Tsimpoukelli, J. Kay, A. Merle, J.-M. Moret, S. Noury, F. Pesamosca, D. Pfau, O. Sauter, C. Sommariva, S. Coda, B. Duval, A. Fasoli, P. Kohli, K. Kavukcuoglu, D. Hassabis, M. Riedmiller, Magnetic control of tokamak plasmas through deep reinforcement learning, Nature 602 (7897) (2022) 414–419, http://dx.doi.org/10.1038/s41586-021-04301-9.

[8] P.J. Antsaklis, Defining intelligent control, IEEE Control Syst. Mag. 14 (3) (1993).

[9] M.B. Quadrelli, L.J. Wood, J.E. Riedel, M.C. McHenry, M. Aung, L.A. Cangahuala, R.A. Volpe, P.M. Beauchamp, J.A. Cutts, Guidance, navigation, and control technology assessment for future planetary science missions, J. Guid. Control Dyn. 38 (7) (2015) 1165–1186, http://dx.doi.org/10.2514/1.G000525.

[10] Z.-y. Song, C. Wang, S. Theil, D. Seelbinder, M. Sagliano, X.-f. Liu, Z.-j. Shao, Survey of autonomous guidance methods for powered planetary landing, Front. Inf. Technol. Electron. Eng. 21 (5) (2020) 652–674, http://dx.doi.org/10.1631/FITEE.1900458.

[11] A.R. Klumpp, Apollo lunar descent guidance, Automatica 10 (2) (1974) 133–146, http://dx.doi.org/10.1016/0005-1098(74)90019-3.

[12] Y. Guo, M. Hawkins, B. Wie, Waypoint-optimized zero-effort-miss/zero-effort-velocity feedback guidance for mars landing, J. Guid. Control Dyn. 36 (3) (2013) 799–809, http://dx.doi.org/10.2514/1.58098.

[13] Y. Guo, M. Hawkins, B. Wie, Optimal feedback guidance algorithms for planetary landing and asteroid intercept, in: AAS/AIAA Astrodynamics Specialist Conference, AAS, 2011.

[14] S. Boyd, S.P. Boyd, L. Vandenberghe, Convex Optimization, Cambridge University Press, 2004, Google-Books-ID: mYm0bLd3fcoC.

[15] B. Acikmese, S.R. Ploen, Convex programming approach to powered descent guidance for mars landing, J. Guid. Control Dyn. 30 (5) (2007) 1353–1366, http://dx.doi.org/10.2514/1.27553.

[16] B. Acikmese, J.M. Carson, L. Blackmore, Lossless convexification of nonconvex control bound and pointing constraints of the soft landing optimal control problem, IEEE Trans. Control Syst. Technol. 21 (6) (2013) 2104–2113, http://dx.doi.org/10.1109/TCST.2012.2237346.

[17] Y. Mao, M. Szmuk, B. Açıkmeşe, Successive convexification of non-convex optimal control problems and its convergence properties, in: 2016 IEEE 55th Conference on Decision and Control (CDC), 2016, pp. 3636–3641, http://dx.doi.org/10.1109/CDC.2016.7798816.

[18] M. Szmuk, T.P. Reynolds, B. Açıkmeşe, Successive convexification for real-time six-degree-of-freedom powered descent guidance with state-triggered constraints, J. Guid. Control Dyn. 43 (8) (2020) 1399–1413, http://dx.doi.org/10.2514/1.G004549.

[19] C. Sánchez-Sánchez, D. Izzo, Real-time optimal control via deep neural networks: Study on landing problems, J. Guid. Control Dyn. 41 (5) (2018) 1122–1135, http://dx.doi.org/10.2514/1.G002357.

[20] S. You, C. Wan, R. Dai, J.R. Rea, Learning-based onboard guidance for fuel-optimal powered descent, J. Guid. Control Dyn. 44 (3) (2021) 601–613, http://dx.doi.org/10.2514/1.G004928.

[21] L. Cheng, Z. Wang, Y. Song, Fanghua Jiang, Real-time optimal control for irregular asteroid landings using deep neural networks, Acta Astronaut. 170 (2020) 66–79, http://dx.doi.org/10.1016/j.actaastro.2019.11.039.

[22] L. Cheng, Z. Wang, F. Jiang, Fanghua Jiang, J. Li, Fast generation of optimal asteroid landing trajectories using deep neural networks, IEEE Trans. Aerosp. Electron. Syst. 56 (4) (2020) 2642–2655, http://dx.doi.org/10.1109/taes.2019.2952700.

[23] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, Proximal policy optimization algorithms, 2017, arXiv preprint arXiv:1707.06347 [cs].

[24] A. Zavoli, L. Federici, Reinforcement learning for robust trajectory design of interplanetary missions, J. Guid. Control Dyn. 44 (8) (2021) 1440–1453, http://dx.doi.org/10.2514/1.g005794.

[25] N.B. LaFarge, D.H. Miller, K.C. Howell, R. Linares, Autonomous closed-loop guidance using reinforcement learning in a low-thrust, multi-body dynamical environment, Acta Astronaut. 186 (2021) 1–23, http://dx.doi.org/10.1016/j.actaastro.2021.05.014.

[26] S. Boone, S. Bonasera, J.W. McMahon, N. Bosanac, N.R. Ahmed, Incorporating observation uncertainty into reinforcement learning-based spacecraft guidance schemes, in: AIAA SCITECH 2022 Forum, 2022, http://dx.doi.org/10.2514/6.2022-1765.

[27] L. Federici, B. Benedikter, A. Zavoli, Deep learning techniques for autonomous spacecraft guidance during proximity operations, J. Spacecr. Rockets 58 (6) (2021) 1774–1785, http://dx.doi.org/10.2514/1.a35076.

[28] A. Rubinsztejn, K. Bryan, R. Sood, F.E. Laipert, Using reinforcement learning to design missed thrust resilient trajectories, in: South Lake Tahoe, California, Jet Propulsion Laboratory, National Aeronautics and Space Administration, 2020, URL https://trs.jpl.nasa.gov/handle/2014/54432. Accepted: 2022-03-16T02:52:17Z Publisher: Pasadena, CA: Jet Propulsion Laboratory, National Aeronautics and Space Administration, 2020.

[29] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, W. Zaremba, OpenAI gym, 2016, arXiv preprint arXiv:1606.01540.

[30] B. Gaudet, R. Furfaro, Adaptive pinpoint and fuel efficient mars landing using reinforcement learning, IEEE/CAA J. Autom. Sin. 1 (4) (2014) 397–411, http://dx.doi.org/10.1109/JAS.2014.7004667, Conference Name: IEEE/CAA Journal of Automatica Sinica.

[31] B. Gaudet, R. Linares, R. Furfaro, Deep reinforcement learning for six degree-of-freedom planetary landing, Adv. Space Res. 65 (7) (2020) 1723–1741, http://dx.doi.org/10.1016/j.asr.2019.12.030.

[32] R. Furfaro, A. Scorsoglio, R. Linares, M. Massari, Adaptive generalized ZEM-ZEV feedback guidance for planetary landing via a deep reinforcement learning approach, Acta Astronaut. 171 (2020) 156–171, http://dx.doi.org/10.1016/j.actaastro.2020.02.051.

[33] R. Furfaro, R. Linares, Waypoint-based generalized ZEM/ZEV feedback guidance for planetary landing via a reinforcement learning approach, in: 3rd International Academy of Astronautics Conference on Dynamics and Control of Space Systems, 2017, pp. 401–416.

[34] C. Finn, P. Abbeel, S. Levine, Model-agnostic meta-learning for fast adaptation of deep networks, in: Proceedings of the 34th International Conference on Machine Learning, PMLR, (ISSN: 2640-3498) 2017, pp. 1126–1135.

[35] L. Federici, A. Scorsoglio, L. Ghilardi, A. D'Ambrosio, B. Benedikter, A. Zavoli, R. Furfaro, Image-based meta-reinforcement learning for autonomous terminal guidance of an impactor in a binary asteroid system, in: AIAA SCITECH 2022 Forum, in: AIAA SciTech Forum, American Institute of Aeronautics and Astronautics, San Diego (CA), USA, 2022.

[36] B. Gaudet, R. Linares, R. Furfaro, Terminal adaptive guidance via reinforcement meta-learning: Applications to autonomous asteroid close-proximity operations, Acta Astronaut. 171 (2020) 1–13, http://dx.doi.org/10.1016/j.actaastro.2020.02.036.

[37] A. Scorsoglio, A. D'Ambrosio, L. Ghilardi, B. Gaudet, F. Curti, R. Furfaro, Image-based deep reinforcement meta-learning for autonomous lunar landing, J. Spacecr. Rockets 59 (1) (2022) 153–165.

[38] Y. Chow, O. Nachum, E. Duenez-Guzman, M. Ghavamzadeh, A Lyapunov-based approach to safe reinforcement learning, in: Advances in Neural Information Processing Systems, 31, Curran Associates, Inc., 2018.

[39] H. Holt, R. Armellin, N. Baresi, Y. Hashida, A. Turconi, A. Scorsoglio, R. Furfaro, Optimal Q-laws via reinforcement learning with guaranteed stability, Acta Astronaut. 187 (2021) 511–528, http://dx.doi.org/10.1016/j.actaastro.2021.07.010.

[40] C. Wilson, A. Riccardi, Improving the efficiency of reinforcement learning for a spacecraft powered descent with Q-learning, Opt. Eng. (2021) http://dx.doi.org/10.1007/s11081-021-09687-z.

[41] C. Wilson, A. Riccardi, Leveraging optimal control demonstrations in reinforcement learning for powered descent, in: 2021 8th International Conference on Astrodynamics Tools and Techniques (ICATT), Noordwijk, The Netherlands, 2021.

[42] C. Wilson, A. Riccardi, E. Minisci, A novel update mechanism for Q-networks based on extreme learning machines, in: 2020 International Joint Conference on Neural Networks (IJCNN), IEEE, Glasgow, United Kingdom, 2020, pp. 1–7, http://dx.doi.org/10.1109/IJCNN48605.2020.9207098.

[43] G.-B. Huang, Q.-Y. Zhu, C.-K. Siew, Extreme learning machine: Theory and applications, Neurocomputing 70 (1) (2006) 489–501, http://dx.doi.org/10.1016/j.neucom.2005.12.126.

[44] S. Hariharapura Sheshadri, D. Franklin, Introducing the ultimate starter AI computer, the NVIDIA jetson nano 2GB developer kit, 2020, URL https://developer.nvidia.com/blog/ultimate-starter-ai-computer-jetson-nano-2gb-developer-kit/.

[45] C. Tessler, D.J. Mankowitz, S. Mannor, Reward constrained policy optimization, 2018, arXiv preprint arXiv:1805.11074.

[46] S. Miryoosefi, K. Brantley, H. Daume III, M. Dudik, R.E. Schapire, Reinforcement learning with convex constraints, in: Advances in Neural Information Processing Systems, Vol. 32, Curran Associates, Inc., 2019.

[47] W. Fedus, P. Ramachandran, R. Agarwal, Y. Bengio, H. Larochelle, M. Rowland, W. Dabney, Revisiting fundamentals of experience replay, in: International Conference on Machine Learning, PMLR, 2020, pp. 3061–3071.

[48] D. Kraft, A software package for sequential quadratic programming, 1988.

[49] K.H. Kienitz, J. Bals, Pulse modulation for attitude control with thrusters subject to switching restrictions, Aerosp. Sci. Technol. 9 (7) (2005) 635–640, http://dx.doi.org/10.1016/j.ast.2005.06.006.