Differentiable Trajectory Optimization Under Uncertainty

Joan Creus-Costa   John Dean

December 1, 2018

Contents

Nomenclature

h altitude
v vertical velocity
ℓ net lift
w_{ℓ} disturbance on lift
w_v disturbance on the vertical velocity of the balloon (wind)

1 Altitude control

1.1 System dynamics

High Altitude Balloons utilize a lifting gas that is lighter than air to produce a buoyant force to counteract the force of gravity on the balloon membrane and the payload. The buoyant force on the balloon minus the force of gravity on the balloon is refereed to as net lift, ℓ. ValBal uses a valve system to vent lifting gas and a ballast system to drop small mass pellets, to change the net lift of the balloon. This net lift produces an acceleration on the system, causing the balloon to accelerate upwards or downwards until the force of drag on the system equals the net lift, at terminal velocity. Because the balloon accelerates to terminal velocity quickly after a change in net lift, the balloon's vertical velocity, v, can be approximately modeled as always traveling at terminal velocity.

While the force of drag on the balloon is nonlinear with respect to velocity, for the purposed of modeling and control, it can be linearized within a range of reasonable velocities. After doing so, the system dynamics can be simplified to:

$$v(t) = k_d \ell \qquad \dot{h}(t) = v(t)$$

With k_d being the linearized coefficient relating net lift to velocity. The valve and ballast system of can be seen as changes to the derivative of the net lift of the balloon, $\dot{\ell}$. For example, when the valve is open, $\dot{\ell}$ becomes negative as the balloon looses lift. In addition, the atmosphere is full of disturbances, so in the system dynamics we consider 2 other terms: $w_{\dot{\ell}}$ as the atmospheric disturbance on the lift rate, and w_v, as the atmospheric disturbance of velocity. For example, at night the balloon cools and looses lift, which is modeled as a negative $w_{\dot{\ell}}$, and vertical winds are modeled as a nonzero w_v. With these additions, the system dynamics become

$$\dot{v}(t) = k_d(\dot{\ell}(t) + w_{\dot{\ell}}(t)) \qquad \dot{h}(t) = v(t) + w_v(t)$$

Using the following substitution we can write the a state-space representation of the system.

$$x = \begin{bmatrix} h \\ v \end{bmatrix} \qquad u = \dot{\ell}$$

$$\dot{x} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} x + \begin{bmatrix} 0 \\ k_d \end{bmatrix} u + \begin{bmatrix} w_v \\ k_d w_{\dot{\ell}} \end{bmatrix}$$
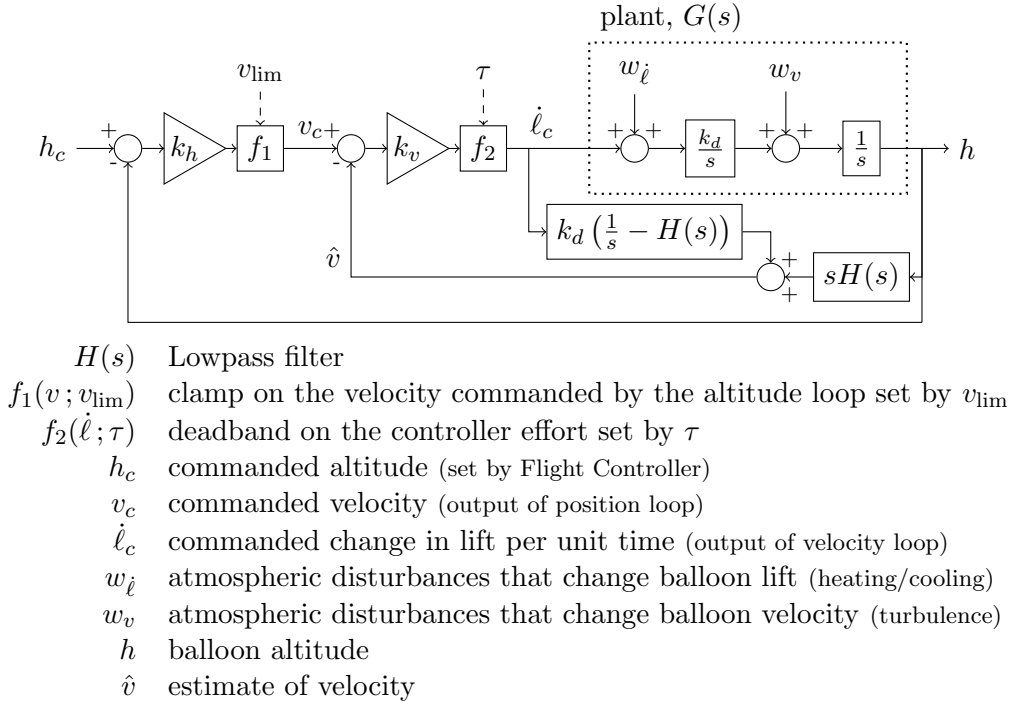
1.2 Control system

1.2.1 Overview

The altitude control system has an unconventional objective for a feedback control system. On every ValBal flight, there is an essential trade-off between tracking closely to a target and system ballast use, as ballast is typically the endurance-limiting factor on a flight. Furthermore, this trade-off can not be usually be conveniently represented as a quadratic relationship between state errors and control inputs to allow for an LQG-style optimal control design. In addition, a more complex SMPC-based optimal control approach would not be feasible due to the constraints that the control law run on a low power embedded processor. Further more, the control objectives are typically given (or well-approximated by) a range of equally equally appealing altitudes, with a lower and upper bound. For instance, on an endurance flight with no location objectives, the lower bound is set by aircraft altitudes and the upper bound is set by latex degradation balloon latex degradation.

As such a linear control law that is of the form $u = K\hat{x}$ for some $K \in \mathbf{R}^{1\times2}$, and a state estimate $\hat{x}$, would not be sufficient. Instead, a nested feedback loop structure is used, where an altitude loop commands a velocity loop which commands a change in lift, which allows for the easy addition of two key nonlinearities. The full control system structure is show in figure 1.

Figure 1: Altitude control system block diagram

$H(s)$	Lowpass filter
$f_1(v\,;v_{\text{lim}})$	clamp on the velocity commanded by the altitude loop set by v_{lim}
$f_2(\dot{\ell}\,;\tau)$	deadband on the controller effort set by τ
h_c	commanded altitude (set by Flight Controller)
v_c	commanded velocity (output of position loop)
$\dot{\ell}_c$	commanded change in lift per unit time (output of velocity loop)
$w_{\dot{\ell}}$	atmospheric disturbances that change balloon lift (heating/cooling)
w_v	atmospheric disturbances that change balloon velocity (turbulence)
h	balloon altitude
$\hat{v}$	estimate of velocity

3

1.2.2 Nonlinearities

The two non-linearities in the controller are given by f_1 and f_2. The first function $f_1(v\,;v_{\text{lim}})$ is simply a clamping function that limits the velocity commanded by the altitude loop to $\pm v_{\text{lim}}$. This prevents the controller fromm overreacting when it is far from it's command. The second non-linearity $f_2(\dot{\ell}\,;\tau)$ is a deadband on the output of the controller. It is defined by

$$
f_2(\dot{\ell}\,;\tau) = \begin{cases} 0 & |\dot{\ell}| < \tau \\ \mathbf{sign}(\dot{\ell})(|\dot{\ell}| - \tau) & |\dot{\ell}| > \tau \end{cases}
$$
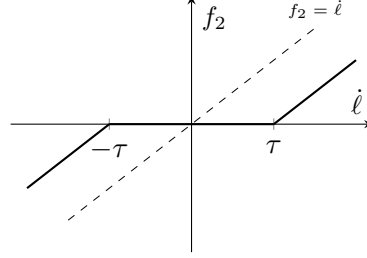
The parameter τ is set by $\tau = k_h k_v e_{\text{tol}}$, where e_{tol} is the steady-state error in altitude allowable by the controller. This means, that with $v = 0$, the system can be at an altitude anywhere in the range $h = h_{\text{cmd}} \pm e_{\text{tol}}$, without the controller acting. The controller can then be thought of as having two bounds, $h_{\text{lower}}, h_{\text{upper}}$ that are given by $h_{\text{lower}} = h_{\text{cmd}} - e_{\text{tol}}$ and $h_{\text{upper}} = h_{\text{cmd}} + e_{\text{tol}}$. As such, controller input commands can either be given in the form $(h_{\text{cmd}}, e_{\text{tol}})$ or $(h_{\text{lower}}, h_{\text{upper}})$. The values are either set by the human flight controller commanding the system, or a trajectory planning algorithm (described in section 2).

1.2.3 Velocity Estimator

For control, it is important that we have a reasonable estimate of the vertical velocity of the balloon, $\hat{v}$. However, the atmosphere has a large amount of turbulence and waves, which have a dramatic impact on the short-term velocity of the balloon. For control on the altitude ranges that valbal flies in, we do not care about these effects. What we actually care about is the component the velocity of the balloon that is due to the net lift. For example, it is possible for the balloon to be rising at 1m/s, and still be neutrally buoyant, as it is simply floating atop a wave in the atmosphere. In this case, we would not want the controller to act to this in the same way it would to the balloon rising steadily at 1m/s due to a positive net lift.

So, an estimator must be designed to calculate $\hat{v}$. A first order estimator such as a Kalman filter would suffice for this, if the noise on the velocity was Gaussian and uncorrelated in time. However, previous flight data and atmospheric literature cite shows that this noise is dominated by large-amplitude waves at specific frequencies. So, a 2nd order lowpass filter, $H(s)$, is used to achieve better roll-off and reject the high frequency waves. In addition, actions of the controller are be integrated through the plan model to update the $\hat{v}$ estimate immediately after controller actions. The corner frequency of $H(s)$ is tpyically set around $(15 \text{ minutes})^{-1}$ and is determined by looking at the spectral content of atmospheric waves in prior flight data.

1.2.4 Choosing Gains

1.2.5 Simulations
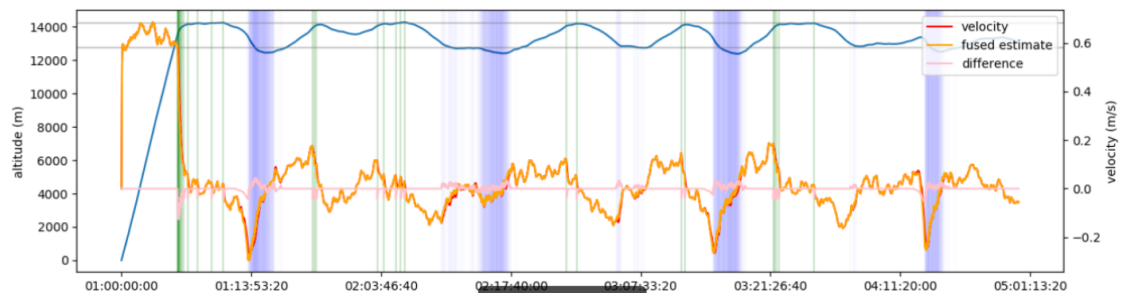
2 Trajectory planning

2.1 Introduction

ValBal has no direct control over it's lateral movement around the globe, but rather is always drifting with the air currents of earth's atmosphere. However, by changing the altitude that it flies at, it can choose different wind currents moving in a range of directions in a given column of air. The high-level goal of trajectory planning is to come up with these altitudes that the balloon should fly at in order to achieve some mission outcome in the lateral trajectory of the balloon.

[Literature review]

2.2 Formulation

The physical system is represented by continuous states $s \in \mathcal{S}$ defined at discrete times $t_k = t_s + k\Delta t$, for $k = 0, 1, \ldots$ In the case of ValBal, we can let $\mathcal{S} = \mathcal{H} \times \Lambda \times \Phi$, if we're considering altitudes, latitudes, and longitudes; we could extend this to include the ballast levels $\mathcal{B}$, or lifts $\mathcal{L}$. The altitude changes with respect to time according to the setpoint altitudes given to the control algorithm, with some randomness due to atmospheric noise. Latitude and longitude change according to Euler integration of the wind field, evaluated at every given time, altitude, and coordinates. Our goal is to find a policy to maximize some objection function, which might, for instance, be the total horizontal distance travelled.

While traditional methods of finding a policy to optimize a physical system rely on discretizing the state space, we note that this is highly intractable to do in this situation, for various reasons:

- The state space is extremely large, even if we restrict ourselves to the bare minimum and discretize coarsely. Methods such as policy iteration would likely take unbearably long.

- The underlying dynamics are continuous, and in fact very easy to integrate. If we discretize the state space, we would lose resolution in the integration. Furthermore, the noise between discretized time steps is correlated; therefore, unless we take very large time steps, or increase the size of the state space, the Markov assumption will not hold, and the policy will not properly handle uncertainty.

While clearly keeping the state space continuous (only discretizing time) makes integration a lot easier, it makes finding a policy harder: the optimal policy is highly nonlinear due to the characteristics of atmospheric wind patterns, and our desired map $(t, h, \phi, \lambda) \rightarrow (h_s, \Delta_s)$ (from time, altitude, and coordinates, to a setpoint altitude and a tolerance around it) is high dimensional. A key realization is that, while the map is highly nonlinear, it is generally smooth, and doesn't have many local minima: wind patterns are generally spatially large and temporally slow moving.

This results in another advantage that comes from keeping the state space continuous. By picking some fixed set of parameters θ (that might be the setpoint altitudes and

tolerances at various points of the flight) and integrating the trajectory as a function of those parameters, we can take the gradient of the cost function (that is a function of the trajectory) with respect to the parameters, and optimize our objective. We expand upon a formulation in the following section.

2.2.1 Differentiable altitude trajectory

Ultimately, our goal is to come up with some continuous and differentiable function $V(\theta)$ that gives the value of the objective function. However, in many instances, the value of the objective function will be related to the integration of the trajectory—e.g. distance travelled, or final longitude. This means that we need to come up with a formulation of the trajectory that depends on the optimization parameters and is differentiable with respect to them.

The parameters that form θ can, for instance, be a set of altitude waypoints defined at various points of the flight, or altitude tolerances at each of those points. If our state consisted only of altitudes, and we were to simply linearly interpolate between altitude waypoints $\theta_0, \theta_1, \ldots$ (defined at $t_0^w, t_1^w, \ldots$) we would find the parameters relevant to the current timestep t_k such that $t_i^w < t_k < t_{i+1}^w$. Then we simply give:

$$h(t_k; \theta) = \theta_i + (t_k - t_i^w) \frac{\theta_{i+1} - \theta_i}{t_{i+1}^w - t_i^w}$$

which has a very simple derivative with respect to the optimization variables θ. However, this assumes that we follow *exactly* the trajectory defined by the optimization parameters. In reality, the value function $V(\theta)$ will be taken over an expectation ensemble of possible trajectories under noise.

In other words, our trajectory integration needs to output a distribution of possible states it might end up, while remaining differentiable. We introduce a parameter λ that can be interpreted as a percentile of the noise distribution that a given trajectory is under. At the beginning of a rollout, a sequence $\lambda_0, \lambda_1, \ldots$ is randomly generated such that the variation results in a noise level consistent with flight data. In the limit where all λ are 0.5, the median trajectory is selected; in a more realistic simulation, the percentile would slowly fluctuate between smaller and larger numbers such that a particular rollout would be flying low or high respectively with respect to a noise-free simulation. For instance, in the scenario given above, we could introduce $\lambda(t)$ as a random walk bounded between -500 m and 500 m, defined between t_0 and t_f. Then our position is given by:

$$h(t_k; \theta) = \theta_i + (t_k - t_i^w) \frac{\theta_{i+1} - \theta_i}{t_{i+1}^w - t_i^w} + \lambda(t_k)$$

This allows us to consider a stochastic evaluation of our value function that remains differentiable.

$$\tilde{V}(\theta) = \sum_{\text{random } \lambda} F(t_f, \lambda; \theta)$$

2.2.2 Differentiable horizontal trajectory

The section above only handled altitude; however, the objective function for ValBal relies on maximizing spatial parameters, and not altitude. The horizontal integration needs to preserve differentiability to be able to compute a full gradient with respect to every optimization parameter. We use atmospheric data provided by NOAA which, in particular, includes the velocity field $w(h, \lambda, \phi, t) \to (u, v)$, where h is the altitude defined on a set of forecast altitudes;[1] λ and ϕ are the coordinates defined on a (not necessarily regular) grid of latitudes and longitudes;[2] t is a forecast time (usually separated by 1 to 3 hours) and (u, v) are the two horizontal components of wind.

For completely regular grids, bilinear interpolation will suffice. For irregular grids, a scheme such interpolation weighted by inverse distance $(\sum^K \alpha_i w_i)/(\sum^K \alpha_i)$ for the closest K points would do the job. The weights of each nearby point are a function of altitude, latitude, and longitude; therefore, when we take the weighted sum, we preserve a differentiability path through the (differentiable) altitude. In other words, the final velocities (u_k, v_k) are themselves a function of θ, which we use to find the new coordinates:

$$x_{k+1} = x_k + u_k \Delta t \quad y_{k+1} = y_k + v_k \Delta t$$

and the conversion to (λ_k, ϕ_k) follows from a simple computation of spherical coordinates.

The viability of this method, as well as the quality of NOAA data can demonstrated by comparing the integration method and wind model to previous balloon flight data.

Words to go here
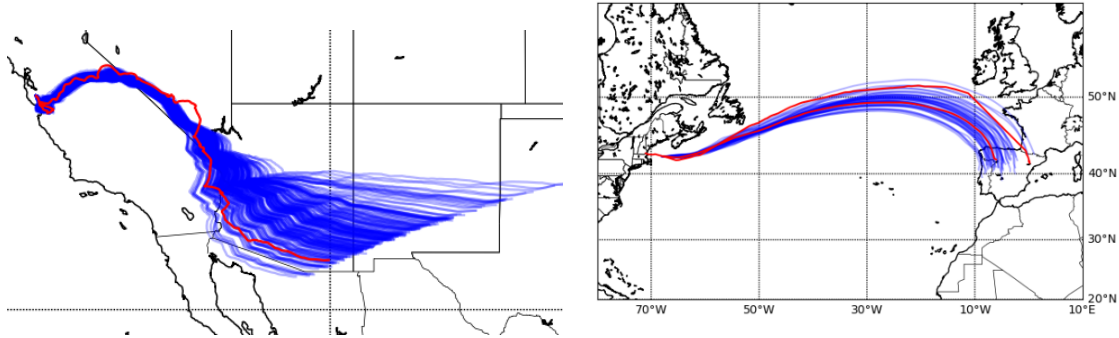
2.2.3 Differentiable value function

We can consider a variety of value functions. If we're trying to maximize the horizontal function, we can use the differentiable longitude computed in Section 2.2.2. For a rollout

---

[1]The data from NOAA is defined at barometric pressures, which can be mapped monotonically to altitude.

[2]Some models are more regular than others; for instance, NAM is defined on a rather annoying Lambert conformal grid.

of length K, the total value is the horizontal distance:

$$V = \sum_{k=0}^{K-1} u_k \Delta t \tag{1}$$

where Δt is a constant that does not affect the result. Other valid objective functions could include the distance to a particular point:

$$V = \sum_{k=0}^{K-1} \|r_k - r_{\text{target}}\|$$

2.3 Preliminary Results

Certainty Equivalent

Before running the full stochastic gradient decent on the monte-carlo problem, we developed and tested the algorithm on certainty-equivalent simulation with no noise. In this experiment, we optimized a single starting trajectory, assuming that we had complete control over 20 altitude way points, with the balloon linearly interpolating between them. The results of such a test are shown in figure 2.
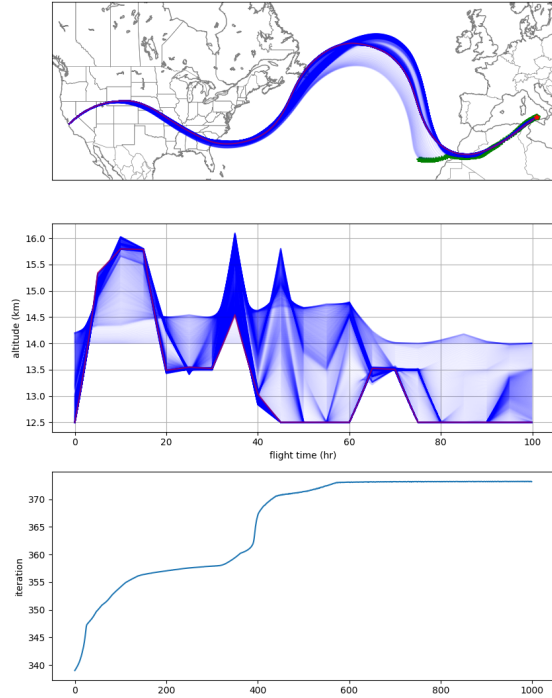

Figure 2: Certainty equivalent optimization from a single trajectory of the total longitude traveled. Trajectories are depicted both in latitude vs longitude over a Mercator projection (top), and altitude vs time (middle). The launch site simulated was Hollister, California and flight endpoints are designated with a green star. Intermediate trajectories are show in blue, with the final optimized result in red. A convergence plot of the objective function vs iteration number is shown (bottom)

We can see from this figure, that the optimization behaves as expected. The final longitude slowly moves forward each iteration, until it is at reaches a local extremum. The path of green points over the map–the endpoints of each intermediate trajectory, always moves to the right as iteration number increases. Running the 1000 iterations of certainty equivalent optimization on a 100 hour flight such as this take well under a second on our hardware.

Monte-Carlo

To run our first test of this formulation, we we opted to utilize the traditional Model Predictive Control approach, in which we optimize an open-loop policy over a finite horizon. In flight, this optimization would be ran again and again as the balloon flies, using the most recently observed system state to optimize new trajectories, forming a feedback control law.

Results of this method on a sample of NOAA analysis wind data is show in figure 3.

We notice a few things from this result. First, that in this particular scenario, the initial commands resulted in a large variation in the resulting flight path. This was due to large changes in wind direction within the command tolerance over the Atlantic ocean. We see that in the optimized open-loop trajectory, policies are picked such that there is not much variation in the resulting outcome.
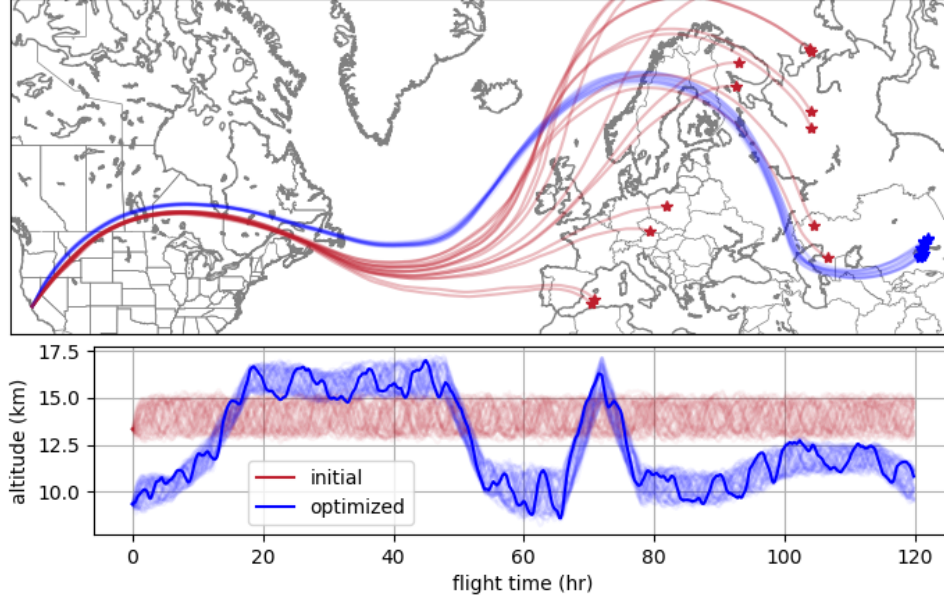
10

Figure 3: Example result of optimizing an open loop trajectory. The initial settings were a 13.5km setpoint and a 0.75km tolerance (red), while the optimized trajectory shows both changing setpoints and tolerances (blue). The dark blue line gives an example possible altitude profile from the many possible altitudes profiles given the command settings

3 Architecture

In this section we describe the architecture behind the simulations and trajectory optimization. It involves a complex pipeline, that involves fetching atmospheric data, a modular and fast simulation engine, and provisions for simulating with real flight code. It is mostly written more than two thousand lines of C++ and Python.

The core of the simulator is written in C++. This allows for extremely fast simulations; when performing direct rollouts, it can compute hundreds of thousands of three-day ValBal trajectories per second on a 6-core machine. The code is designed to be modular; one can easily use different objective functions (horizontal distance, distance to a given target...), different optimization methods (vanilla gradient descent, Adagrad, Adam...), various differentiable controllers (uniform distribution, approximate Lasagna, differentiable Lasagna...), and so on.

Having been designed with optimization in mind, another key feature of the simulation codebase is the formulation in terms of C++ templates, that allow the compiler to, from

a single codebase, generate code that either merely computes rollouts of the trajectories, or keeps track of the gradients leading to the final objective function. The latter uses *automatic differentation*, the same algorithm used in the training of large neural networks. It keeps track of the computation graph and derivative at each node; once the final objective function is computed (say, the final longitude), the gradient is propagated backwards to each parameter, and some first-order optimization method is used to update the parameters. By using the Adept library, the parametrized codebase can be specialized to use Adept's differentiable floating point type, that keeps track of the computation graph. Doing so incurs an overhead, of about 5 to $10\times$, but is able to provide the exact gradient of the objective with respect to the parameters, in a much easier way than direct numeric differentiation, and without having separate code paths for direct evaluation and gradient evaluation.

We also built several Python scripts to fetch the data and process the results. We can retrieve wind data from either the Global Forecast System (GFS) model from NOAA, or the European ECMWF model. We preprocess the data, that comes in GRIB files, into binary files that are loaded directly by C++. This is key to the speed of the C++ codebase, by using the `mmap` system call from Linux, the data will end up cached in the comparatively faster RAM, and wind table lookups require merely the computation of an offset (as opposed to searching through a database, with various levels of indirection). In particular, we only store relevant variables at the altitudes ValBal normally flies, in order to save space and improve cache performance. Several Python scripts are used to plot the trajectories found by the C++ script (saved to a binary file) using the Matplotlib library.

4 Flight Test