HASKELL SPACEFLIGHT WORKSHOP

Jonathan Merritt, Luke Clifton

YOW! LambdaJam 2019, May 13–15

Version timestamp: 2019-04-30T15:42:51+0000

Table of Contents

1

Introduction

This workshop consists of these published notes and a set of problems contained in the associated GitHub Haskell project.

How much of the notes it will be necessary to read depends on the background of an individual participant. Those who are already familiar with numerical methods might be able to skip most of the notes and simply work on the problems directly. However, we have tried to make the notes fairly comprehensive for newcomers, so that people unfamiliar with numerical methods at least have a starting point.

2

ODE Integration and Initial Value Problems

The models we use for a spacecraft depend upon a set of variables that represent its state at an instant in time. These state variables typically include:

- Position
- Velocity
- Mass

They may be scalar quantities or vectors, as appropriate to the problem.

Our simulations are all examples of "Initial Value Problems". In an initial value problem, we know the starting state of the spacecraft, and we have a set of first-order ordinary differential equations (ODEs), which describe how its state evolves with time. We will integrate these ODEs to predict the state at future times. Using this approach, we can compute the time history of state variables that are critial to mission or maneuver planning. For example, we might find the trajectory of a spacecraft (its position as a function of time), and check whether it places the spacecraft in a desired orbit.

The motion of a spacecraft depends on multiple forces that might be acting on it. For example:

- Gravity
- Atmospheric drag
- Rocket thrust

Thrust from a rocket engine may be controlled, and control inputs can be modeled easily in our system. Testing the behavior of a control system, particularly under conditions of real-world variations, is a modern practical use of the methods we cover (eg. [1, 2]).

2.1 1D Euler's Method

We can write a set of coupled, first-order ODEs as:

$$\frac{d\mathbf{x}}{dt} = \dot{\mathbf{x}} = f(t, \mathbf{x}) \tag{2.1}$$

Here, $\mathbf{x}$ is the state vector, t is time, and f is some function. In Euler's method, we approximate a step forward in time by adding the product of the gradient, $\dot{\mathbf{x}}$, and the time step, h, to the current state, $\mathbf{x}$:

$$\mathbf{x}(t+h) \approx \mathbf{x}(t) + \dot{\mathbf{x}}\, h \tag{2.2}$$

$$\approx \mathbf{x}(t) + f\left(t, \mathbf{x}(t)\right)\, h \tag{2.3}$$

2.1.1 Radioactive Decay

We will begin implementing Euler's method with a 1D state, specialized to `Double`, using the process of radioactive decay as an example. Radioactive decay has an analytical solution, thus providing a ground truth against which the numerical result can be compared. It only involves a single state variable, N, which can be represented as a `Double`. Specializing to `Double` gives us a simple starting point that is close to the approach used in many other programming languages.

In radioactive decay, the rate of decay, $\dot{N}$, is proportional to the number of moles of radioactive particles that remain at any instant in time, N:

$$\dot{N} = -\lambda N \tag{2.4}$$

where λ is called the decay constant. This equation can be solved by knowing in advance that an exponential function happens to fit exactly the expected equation:

$$N = N_0 \exp\left(-\lambda t\right) \tag{2.5}$$

So that:

$$\dot{N} = -\lambda \left(N_0 \exp\left(-\lambda t\right)\right) \tag{2.6}$$

$$= -\lambda N \tag{2.7}$$

as required. Conventionally, λ is specified in terms of the half-life of an isotope, $t_{(1/2)}$:

$$\text{at } t = 0,\ N = N_0 \tag{2.8}$$

$$\text{at } t = t_{(1/2)},\ N = \frac{N_0}{2} \tag{2.9}$$

thus:

$$\frac{N_0}{2} = N_0 \exp\left(-\lambda t_{(1/2)}\right) \tag{2.10}$$

$$\ln\left(\tfrac{1}{2}\right) = -\lambda t_{(1/2)} \tag{2.11}$$

$$\lambda = \frac{\ln 2}{t_{(1/2)}} \tag{2.12}$$

As an example, consider the isotope Plutonium-238 (^{238}Pu), which has been used in radioisotope thermoelectric generators (RTGs) for spacecraft such as the Voyager 1 and 2 probes. This isotope has a half-life of approximately 87.7 years.

Problem 1: Euler integration specialized to `Double`.

In the file `ODE.hs`,

- implement `eulerStepDouble`, which takes a single step of Euler integration
- implement `integrateEulerDouble`, which takes multiple steps

In `ODEExamples.hs`,

- run `plotEulerDoubleExpDecay Screen`, to view a plot of Euler integration applied to the radioactive decay example

Figure 2.1 shows the result of applying Euler integration to the radioactive decay example. In this figure, it is evident that when smaller time-steps are taken, the Euler method more closely approximates the analytical solution. This is usually the case practically with numerical integration, although there is a limit beyond which smaller time steps will begin to diverge from the correct solution due to accruing floating-point errors. We will see later that raising the polynomial order of the integration approximation can improve accuracy with greater computational efficiency than taking smaller time steps.

2.2 Euler's Method for an `AffineSpace` State

We will now generalize Euler's method using the abstractions available in the `vector-space` package. The necessary constraints are captured in Listing 1. Don't panic if this seems a lot to take in, since we'll see a few concrete examples.

The first concept we introduce is the difference between an `AffineSpace` and its associated `VectorSpace`. In the present context, points in the `AffineSpace` are points belonging to the state space of the problem, of type `state` (eg. position, velocity, etc). Vectors of the associated `VectorSpace`, of type `diff`, represent deltas or differences between the points (eg. an offset of position, a delta in velocity, etc). We can add a vector to a point to obtain a new point, but we don't sum points directly. Similarly, we can multiply a vector by a scalar, but we cannot multiply a point by a scalar. Most widely-used frameworks for numerical integration do not make this distinction.

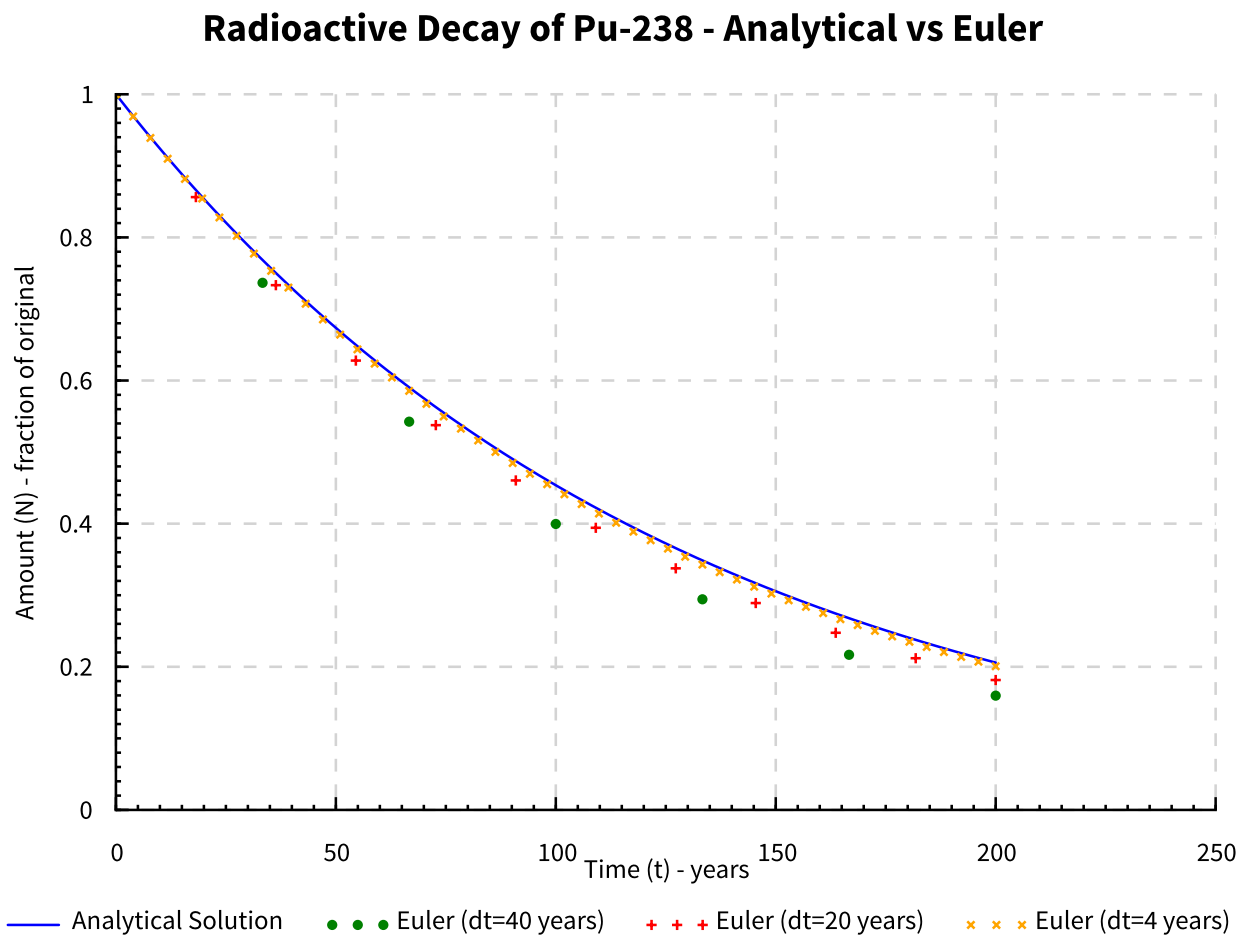

Figure 2.1: Comparison of Euler integration with the analytical result for radioactive decay of the isotope ^{238}Pu. The solid line shows the analytical solution while the points demonstrate the Euler approximation for different time steps.

```
eulerStep
  :: ( AffineSpace state
     , diff ~ Diff state, VectorSpace diff
     , HasBasis time, HasTrie (Basis time)
     , s ~ Scalar diff, s ~ Scalar time )
  => time                                -- ^ Step size @dt@
  -> ((time, state) -> time :-* diff)    -- ^ Gradient function @f (x, t)@
  -> (time, state)                       -- ^ Before the step @(t, x)@
  -> (time, state)                       -- ^ After the step @(t, x)@
```

Listing 1: Constraints for Euler's method generalized by `vector-space`.

Next is the concept of a linear map representing the derivative: `time :-* diff`. In our case, where we deal with finite differences, this constructor can be considered analogous to a (linear) function of type `time -> diff`. What this function represents is the delta, `diff`, which arises from taking a time-step, h, of type `time`. A very simple illustration of the linear map is shown in Listing 2, with a version for vectors shown in Listing 3. The `linear` function assumes that the function it has been provided is linear, and it memoizes the values of that function along each basis vector of the vector space.

```
> :set -XFlexibleContexts
> import Data.LinearMap ((:-*), linear, lapply)
> f = (*) 5 :: Double -> Double
> lm = linear f
> :t lm
lm :: Double :-* Double
> :force lm
lm = Data.LinearMap.LMap (Just 5.0)
> lapply lm 1.0
5.0
> lapply lm 2.0
10.0
```

Listing 2: A scalar linear map. Once the map has been defined (by the `linear` function), the `lapply` function multiplies the input vector (a `Double`) by the memoized value along the unit basis vector.

```
> :set -XFlexibleContexts
> import Data.LinearMap ((:-*), linear, lapply)
> :{
| f :: (Double, Double) -> (Double, Double, Double)
| f (x, y) = (2*x + y, 3*x - y, 4*y)
| :}
> lm = linear f
> :force lm
lm = Data.LinearMap.LMap
     (Just
       (Data.MemoTrie.EitherTrie ((,,) 2.0 3.0 0.0) ((,,) 1.0 -1.0 4.0)))
> lapply lm (5, 6)
(16.0,9.0,24.0)
```

Listing 3: A vector linear map, using tuples for vectors. This provides a better view of the memoisation that is occurring under the hood. The construction of a matrix-like representation (but with automatic dimension checking) is evident.

Finally, we need to describe the operations that can be used on instances of `AffineSpace` and its associated `VectorSpace`:

`lapply m h` applies linear map `m` to vector `h`

`p .+^ v` adds vector `v` to point `p`
`a ^+^ b` adds vector `a` to vector `b`

These operations are sufficient to implement the generalized form of Euler's method.

2.2.1 Simple Harmonic Motion

To motivate the generalized form of Euler's method, let's consider simple harmonic motion (SHM), with two scalar state variables:

- Position, r
- Velocity, v

The data type `State` in `ODEExamples.hs` describes this state, which is the `AffineSpace` of the problem. It introduces statically type-checked units from the `units` package for length and velocity. The data type `DState` is the correspondng `VectorSpace` of the problem, representing deltas in the state.

In SHM, a linear spring force, F, is proportional to the position, r, with a spring constant, k:

$$F = -kr \tag{2.13}$$

This is combined with the equations of motion of a point mass to produce the following governing ODE:

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{r} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} v \\ -kr/m \end{bmatrix} \tag{2.14}$$

where v is the velocity and m is the mass. If the initial conditions of the problem at $t = 0$ are $r = r_0$ and $v = 0$ then the analytical solution is:

$$\mathbf{x}(t) = \begin{bmatrix} r_0 \cos(\omega t) \\ -\omega r_0 \sin(\omega t) \end{bmatrix} \tag{2.15}$$

in which ω is the angular velocity, given by:

$$\omega = \sqrt{\frac{k}{m}} \tag{2.16}$$

This analytical solution can be differentiated twice manually to confirm that it satisfies the governing ODE, and values substituted to confirm that it satisfies the initial conditions.

Problem 2: Generalized Euler integration.

In the file `ODE.hs`,

- implement `eulerStep`
- implement `integrate` and `integrateWithDiff`

In `ODEExamples.hs`,

- run `plotEulerSHM Screen`, to view a plot of the position state variable, computed by Euler integration of SHM equations

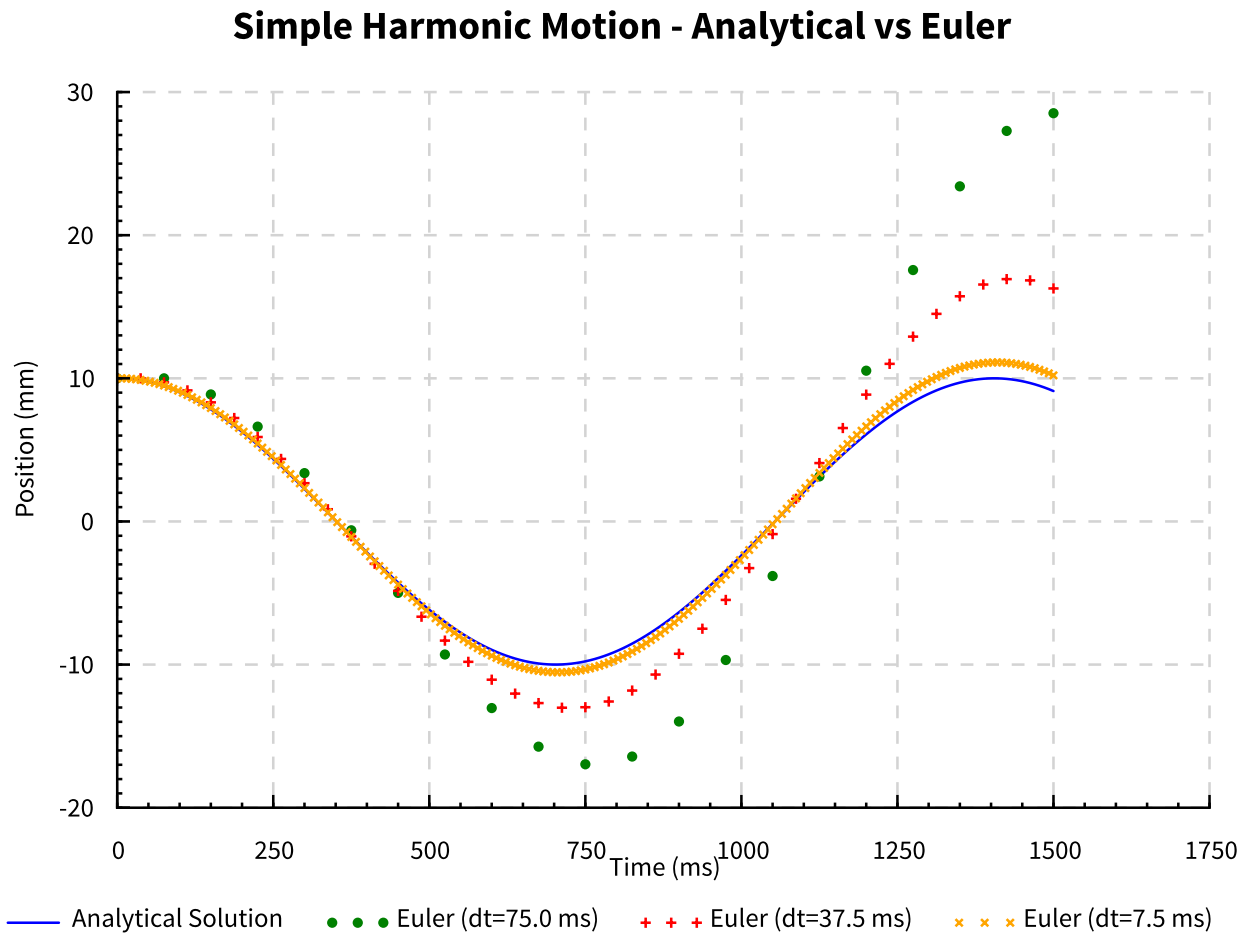

Figure 2.2: Comparison of Euler integration with the analytical result for the position variable of a simple harmonic oscillator. The solid line shows the analytical solution while the points demonstrate the Euler approximations for different time steps.

Figure 2.2 shows the position variable of a simple harmonic oscillator example. Once again, the smaller the time step, the more closely the result tracks the analytical solution.

2.3 4th-Order Runge-Kutta Integration

We have used Euler integration so far because it introduced the concepts we rely on from `vector-space`. However, there is a more common default for numerical integration of ODEs: the 4th-Order Runge-Kutta method (RK4). RK4 is the default in Matlab and SciPy (in those packages it is typically used with an embedded 5th-order approximation for step-size control; something we won't implement here). RK4 is usually a much better choice than the Euler method in terms of the accuracy/performance tradeoff.

RK4 is definitely not the final word though! The popular Numerical Recipes textbook recommends an 8th-order method (Dopr853) for general production use in non-stiff systems [3]. Specialized integrators may also be used for particular problems. The Apollo Guidance Computer used Nyström's Method to perform integration for efficiency and because of the dominant effect of a single, central gravitational force in most situations [4].[1] Long-duration astrodynamics problems, such as those concerning solar-system formation, or the behaviour of orbits over thousands of years, may have to use symplectic integrators to achieve reasonable accuracy (eg. [5]). We don't investigate these methods here because of time limitations, and because RK4 is both easy to implement and entirely sufficient for the examples.

We will only supply the equations for RK4 here and refer readers elsewhere (eg. [3, 6]) for a complete derivation:

$$\mathbf{k_1} = h\, f(t, \mathbf{x}) \tag{2.17}$$
$$\mathbf{k_2} = h\, f(t + \tfrac{1}{2}h, \mathbf{x} + \tfrac{1}{2}\mathbf{k_1}) \tag{2.18}$$
$$\mathbf{k_3} = h\, f(t + \tfrac{1}{2}h, \mathbf{x} + \tfrac{1}{2}\mathbf{k_2}) \tag{2.19}$$
$$\mathbf{k_4} = h\, f(t + h, \mathbf{x} + \mathbf{k_3}) \tag{2.20}$$
$$\mathbf{x}(t + h) \approx \mathbf{x} + \tfrac{1}{6}\mathbf{k_1} + \tfrac{1}{3}\mathbf{k_2} + \tfrac{1}{3}\mathbf{k_3} + \tfrac{1}{6}\mathbf{k_4} \tag{2.21}$$

The vectors $\mathbf{k_1} \ldots \mathbf{k_4}$ can be treated as stages of the computation, and are good candidates for let-floating. However, be aware that these equations do not directly represent the Haskell code. Instead, when implementing them, some care must be taken to consider what components are computed by `lapply`, what operations are adding vectors, and what operations are offsetting a point by a vector. Determining these are left as part of the exercise.

[1] The Draper Lab Apollo documents do refer to RK4 though, as "The usual fourth-order Runge-Kutta integration". Apparently it has a long history as the "go to" approach!

Problem 3: Runge-Kutta Integration.

In the file `ODE.hs`,

- implement `rk4Step`

In `ODEExamples.hs`,

- run `plotSHMComparison Screen`, to view a plot of Euler vs RK4 for the SHM example, using the same number of function evaluations

Figure 2.3 shows the comparison of Euler's method and RK4, for the same number of function evaluations. It is clear that RK4 much more closely approximates the analytical result.

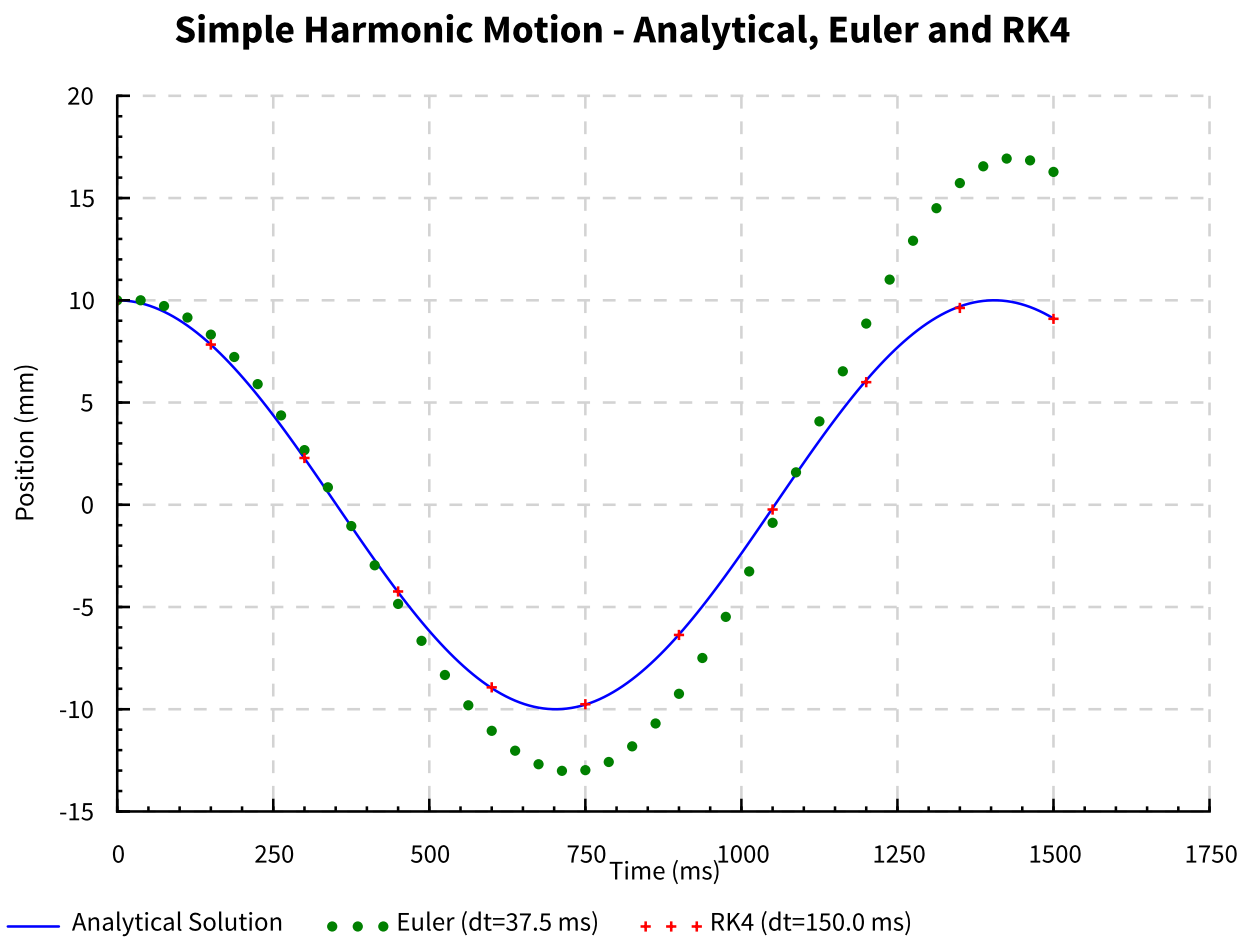

Figure 2.3: Comparison of Euler and RK4 integrations, for the same number of function evaluations, with the analytical result of a simple harmonic oscillator.

2.4 Simulating Apollo Lunar Ascent

Now that we have a working integrator, it's possible to look at an example of a somewhat realistic simulation: the algorithm used for Lunar Ascent Guidance during the Apollo missions.

This algorithm is described in summary by a NASA technical report [7] and in much greater detail by an MIT Draper Lab document [4], which contains sufficient detail to implement the algorithm.

This section contains a "pre-baked" scenario that uses the code written in this chapter to run a prepared simulation of the Apollo Lunar Ascent. We do not expect participants to be able to cover all the details in the available time, so much of the information in this section is provided only to give a high-level overview.

The approach we use for simulation involves checking-in with our version of the Apollo Guidance Computer (AGC) every 2 s of simulation time, matching the polling that was used in the original AGC. The ascent guidance returns a commanded thrust angle and an optional engine shutoff time. We take the commanded thrust angle and compute an angular acceleration that will point the ascent stage at that angle by the end of the 2 s period. This approximates the behaviour of the original Digital Autopilot, which achieved the same thing by firing the Reaction Control System (RCS) thrusters. We then integrate the equations of motion forward for 2 s before polling the AGC once again. This swapping between calls to the AGC and forward integration continues until we reach the commanded engine shutoff time, at which point we stop the thrust from the Ascent Propulsion System (APS) engine. Following the burn phase, we take its final state as the initial conditions of a new integration, and integrate the equations of motion for a further 10 000 s to numerically compute the coasting trajectory.

A rare feature of this simulation is that it incorporates statically-checked units, from the `units` package. Whether or not that was a worthwhile exercise is debatable, as the overhead and extra code complexity is somewhat overwhelming. However, we do have static confirmation that almost all aspects of the algorithm use consistent units.

The ascent guidance uses the following parameters as a target:

- target velocity (a 2D vector)
- target radius (a scalar distance measured from the center of the moon)

During the Apollo missions, the desired insertion orbit of the ascent stage was computed by Misson Control in advance of starting the ascent. That orbit determined a final radius and velocity that should be reached at the end of the burn, specifying the required target parameters. In our simulation, we did not perform any of those calculations, but instead used the target parameters for a "quick, early takeoff", which were programmed by default into the AGC, and would typically be overridden by the astronauts under nominal conditions [4]. The phase of the orbit relative to the Command Service Module (CSM) would be set approximately, by knowing the nominal ascent duration and timing the takeoff accordingly [4]. Rendezvous with the CSM was an entirely different manouvre, performed after the ascent, and not part of the ascent guidance [4].

The inner workings of the guidance algorithm are somewhat outside the scope of this work-

shop, and in fact are not derived in the Draper Lab document [4]. We can roughly describe our inferred understanding of the approach based upon other texts. A detailed derivation of multiple types of guidance is provided by Battin [6], who was a core member of the Draper Lab during the Apollo era. A freely-available derivation that quite closely matches the Apollo algorithm is also provided by Townsend et al. [8], although those authors do not specifically mention Apollo.

Given the ascent target parameters, the guidance begins each polling loop by computing a velocity-to-be-gained. This is the difference between the target velocity and the current velocity, corrected for any velocity losses during the burn from gravitational acceleration. The change in velocity due to gravity was given by an expression obtained by integrating the gravitational acceleration analytically. The upper limit of that integration required an approximation of the remaining burn time (time-to-go estimate). The initial time-to-go estimate was a hard-coded value, but subsequent iterations obtained an estimate from a Taylor Series expansion of the well-known Tsiolkovsky Rocket Equation, using the velocity-to-be-gained as the delta-V. Finally, a control law, referred to as the "linear guidance concept" [4], was used to choose the commanded thrust direction, so that the velocity-to-be-gained would fall to zero at the end of the burn, thus matching the target velocity, and the target radius would also be achieved. In addition, the algorithm contained several, somewhat more ad-hoc tweaks, such as prioritising radial thrust and performing bang-bang directional control of radial acceleration if the target radius was reached early.

Given the Draper Lab's prior involvement with ballistic missile projects, such as the Minuteman ICBMs [9], the ascent guidance algorithm seems quite likely to have been influenced by missile systems. It also seems likely that earlier practical tests of those systems and other rockets lent some confidence to the early choice of guidance approaches. However, it is naturally extremely difficult to track down any details of large weapons systems for final confirmation.[2]

In our version of the algorithm, we have made some minor simplifications to suit this workshop, none of which are substantial changes to the core algorithm or guidance concept. The main changes and approximations are:

- We projected the problem into 2D, by removing the parameter specifying the distance from the CSM orbital plane, and removing the associated control parameters. Mathematically, this is identical to launching in the CSM orbital plane and remaining there, so it is simply a special case of the full guidance algorithm.
- We removed the thrust filter computations and associated pre-launch initializations, since we don't model the behaviour and noise characteristics of the inertial guidance unit. Instead, we used the nominal initial value for the thrust, based on stored values for the APS exhaust velocity and mass flow rate.
- We did not allow the RCS to substitute for the APS, and we did not consider modeling

[2] I tried. Hopefully I'm not on any (additional) watch lists now. - J. Merritt.

abort scenarios.

- We did not include lunar rotation in the initial conditions (our initial velocity is zero).
- The Average-G routine is substituted with a simpler version.
- Window Pointing Direction (WDP) is unused and not specified (it always points toward the center of the moon in the original version anyway).
- The Digital Autopilot (not technically part of the Ascent Guidance itself) is not modelled in full, and is instead substituted by a piecewise constant angular acceleration.
- We did not take into account additional thrust produced by +X firing of the RCS for attitude control. Due to the regular angular acceleration, and thus regular RCS burns during ascent, this may result in our slightly underestimating the net thrust.

Although the details above are unlikely to be investigated deeply during the workshop, the larger point is to demonstrate that a relatively involved control algorithm, that was part of one of the most astounding engineering achievements in history, can be simulated fairly easily using the integration approaches developed in this chapter.

Symbols

f	A function.
F	Total force, scalar (N).
h	Time step (s).
k	Spring constant (N/m).
m	Mass (kg).
N	Number of moles of a substance.
N_0	Number of moles of a substance at $t = 0$.
r	Position, scalar (m).
r_0	Position, scalar, at $t = 0$ (m).
t	Time (s).
$t_{(1/2)}$	Radioactive half life (s).
v	Velocity, scalar (m/s).
$\mathbf{x}$	System state vector.
λ	Radioactive decay constant (1/s).
ω	Angular frequency (1/s).

References

[1] J. L. Prince, P. N. Desai, E. M. Queen, and M. R. Grover, "Mars phoenix entry, descent and landing simulation design and modelling analysis," *Journal of Spacecraft and Rockets*, vol. 48, no. 5, pp. 754–764, 2011. [Online]. Available: https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20080033126.pdf

[2] G. L. Brauer, D. E. Cornick, and R. Stevenson, *Capabilities and Applications of the Program to Optimize Simulated Trajectories (POST). Program Summary Document. NASA CR-2770*. NASA, 1977. [Online]. Available: https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19770012832.pdf

[3] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*, 3rd ed. New York, NY, USA: Cambridge University Press, 2007.

[4] G. M. Levine, Ed., *Apollo: Guidance, Navigation and Control*. Charles Stark Draper Laboratory, MIT, 1971, vol. Section 5: Guidance Equations (Rev. 11). [Online]. Available: https://www.ibiblio.org/apollo/Documents/j2-80-R-567-SEC5-REV11_text.pdf

[5] H. Rein and D. Tamayo, "JANUS: a bit-wise reversible integrator for N-body dynamics," *Monthly Notices of the Royal Astronomical Society*, vol. 473, no. 3, pp. 3351–3357, 09 2017. [Online]. Available: https://arxiv.org/abs/1704.07715

[6] R. H. Battin, *An introduction to the Mathematics and Methods of Astrodynamics, Revised Edition*. American Institute of Aeronautics and Astronautics, 1999.

[7] F. V. Bennett, "Apollo Lunar Descent and Ascent Trajectories. NASA TM X-58040," *AIAA 8th Aerospace Sciences Meeting, New York*, 1970. [Online]. Available: https://www.hq.nasa.gov/alsj/nasa58040.pdf

[8] G. E. Townsend, A. S. Abbott, and R. R. Palmer, *Guidance, Flight Mechanics and Trajectory Optimization. NASA CR-1007*. NASA, 1968, vol. VIII – Boost Guidance Equations. [Online]. Available: https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19680010980.pdf

[9] M. A. Dennis, "Encyclopaedia Britannica: Charles Stark Draper." [Online]. Available: https://www.britannica.com/biography/Charles-Stark-Draper