CMM22 a6: Trajectory Optimization

Responsable TA: Miguel Zamora (mimora@ethz.ch)
Assignment credits to James Bern

In this assignment we'll learn the basics of trajectory optimization.

- Please answer written questions in the README.
- **The first parts of this assignment take place in `shooting-app.cpp`, and should be completed first.**
- The following two sections will be implemented in `transcription-app.cpp` and `challenge-app.cpp` respectively.

warm-up

coding tips

1. `pow(10., 3)` and `1e3` are both equal to 10^3
2. Access the last element of a `std::vector` using `xK.back()`
3. In Eigen, $|\boldsymbol{u}|^2$ is just `u.squaredNorm()`
4. Access the front and back of Eigen vectors using

```
C.head(2) = ... // First two elements of C
C.tail(2) = ... //  Last two elements of C
```

5. **Not mandatory:** Set up your code to break if a `NaN` is ever created. E.g. on Windows (VS):

```
#include <float.h>
// BEG Put me at beginning of app's constructor
_clearfp();
_controlfp(_controlfp(0,0) &
    ~(_EM_INVALID | _EM_ZERODIVIDE | _EM_OVERFLOW),
    _MCW_EM);
// END
```

explicit Euler particle dynamics

Consider a particle with mass m and position $\boldsymbol{x}$. Call $\boldsymbol{F}$ the force acting on a particle (or spaceship), and $\boldsymbol{a}$ its acceleration. Newton's second law has

$$\boldsymbol{F} = m\boldsymbol{a},$$

which is the same thing as

$$\boldsymbol{F} = m\frac{d^2\boldsymbol{x}}{dt^2},$$

which all the cool kids write like

$$\boldsymbol{F} = m\ddot{\boldsymbol{x}}.$$

Out loud you say $\ddot{\boldsymbol{x}}$ as "$\boldsymbol{x}$ double dot." We can rearrange this into the second order ODE of our dynamics

$$\ddot{\boldsymbol{x}} = \boldsymbol{F}/m,$$

where I like the shorthand $\boldsymbol{F}/m = \frac{1}{m}\boldsymbol{F}$.

Now to integrate forward in time, we first need to transform these dynamics into an equivalent first order ODE. Introduce the *state vector*

$$\boldsymbol{\xi} = \begin{bmatrix}\boldsymbol{x}\\ \boldsymbol{v}\end{bmatrix},$$

where $\boldsymbol{v} = \dot{\boldsymbol{x}}$ is the particle's velocity. Our equivalent first order ODE is

$$\dot{\boldsymbol{\xi}} = \begin{bmatrix}\dot{\boldsymbol{x}}\\ \dot{\boldsymbol{v}}\end{bmatrix} = \begin{bmatrix}\boldsymbol{v}\\ \boldsymbol{F}/m\end{bmatrix}.$$

We discretize according to explicit Euler (sure it tends to explode, but it's also really simple, so let's just not worry about it), which has the update rule

$$\boldsymbol{\xi}_{k+1} = \boldsymbol{\xi}_k + h\dot{\boldsymbol{\xi}}_k. \tag{1}$$

Note we are using the notation $\boldsymbol{\xi}_k = \boldsymbol{\xi}(t_k)$ to mean "$\boldsymbol{\xi}$ at the k-th timestep". Substituting into Equation (1) we have the update rule for our particle

$$\begin{bmatrix}\boldsymbol{x}_{k+1}\\ \boldsymbol{v}_{k+1}\end{bmatrix} = \begin{bmatrix}\boldsymbol{x}_k\\ \boldsymbol{v}_k\end{bmatrix} + h\begin{bmatrix}\boldsymbol{v}_k\\ \boldsymbol{F}_k/m\end{bmatrix}. \tag{2}$$

1. Implement the above equation in `stepPhysicsExplicitEuler(...)`
   NOTE: When you are done, if you press 's' on your keyboard (or tick the `SIMULATE` checkbox) you should see a little yellow square spaceship fly off to the right forever at its initial velocity $\boldsymbol{v}_0$.

We can rearrange Equation (2) into

$$\begin{bmatrix}\boldsymbol{v}_k\\ \boldsymbol{F}_k/m\end{bmatrix} = \frac{1}{h}\begin{bmatrix}\boldsymbol{x}_{k+1} - \boldsymbol{x}_k\\ \boldsymbol{v}_{k+1} - \boldsymbol{v}_k\end{bmatrix}. \tag{3}$$

2. What is $\boldsymbol{a}_k$ as a function of $\boldsymbol{x}_k, \boldsymbol{x}_{k+1}$ and $\boldsymbol{x}_{k+2}$?
   Note we would call $\boldsymbol{a}_k$ "the acceleration discretized according to explicit Euler."
   HINT: $\boldsymbol{F}_k = m\boldsymbol{a}_k$

spaceship

Let's go ahead and call that particle a spaceship. We'll work in 2D, so its position is

$$\boldsymbol{x} = \begin{bmatrix} x \\ y \end{bmatrix}.$$

Please note that we do *not* concern ourselves with its orientation (I draw the spaceship as a square to make this visually obvious).

Let us break the forces acting on the spaceship down into two pieces

$$\boldsymbol{F} = \boldsymbol{F}_{\text{thrusters}} + \boldsymbol{F}_{\text{gravity}},$$

where $\boldsymbol{F}_{\text{thrusters}}$ is the propulsive force from the ship's thrusters, and $\boldsymbol{F}_{\text{gravity}}$ is the gravitational force from nearby planets. Assume the spaceship has a perfect, omnidirectional thruster, capable of exerting any force at any moment in time. This is just a fancy way of saying

$$\boldsymbol{F}_{\text{thrusters}} = \boldsymbol{u},$$

with no constraints on $\boldsymbol{u}$[1].

3. Implement `get_Fk(...)`, neglecting gravity for now.
   NOTE: I made a nonzero initial guess for $\boldsymbol{u}$, so the spaceship flies up out the top of the screen when you simulate. Feel free to use a zero initial guess instead.

[1] Note that a more sophisticated system would also consider the orientation θ of the spaceship (in addition to spatial position x, y), and e.g. only exert forces pointing out the back. In this case $\boldsymbol{F}_{\text{thrusters}}$ would be a function of both $\boldsymbol{u}$ and θ.

trajectory optimization

Here we'll look at two approaches to trajectory optimization, direct single shooting and direct transcription. There is no one right approach to trajectory optimization. There isn't even one right approach to e.g. direct transcription.

direct single shooting

Direct shooting optimizes over the control only. In our case it will take the form

$$\boldsymbol{u}^* = \arg\min_{\boldsymbol{u}} \mathcal{O}(\boldsymbol{u}, \boldsymbol{x}(\boldsymbol{u}), \boldsymbol{v}(\boldsymbol{u})),$$

where $\boldsymbol{u} = \begin{bmatrix}\boldsymbol{u}_0 & \boldsymbol{u}_1 & \cdots & \boldsymbol{u}_{K-1}\end{bmatrix}^T$. To find $\boldsymbol{x}(\boldsymbol{u})$ and $\boldsymbol{v}(\boldsymbol{u})$ we integrate forward in time from $(\boldsymbol{x}_0, \boldsymbol{v}_0)$ according to explicit euler.

4. Implement $\mathcal{O}(\boldsymbol{u}, \boldsymbol{x}(\boldsymbol{u}), \boldsymbol{v}(\boldsymbol{u}))$ in `ShootingObjective::evaluate(...)` such that $\boldsymbol{x}_K(\boldsymbol{u}) = \boldsymbol{x}'$ and $\boldsymbol{v}_K(\boldsymbol{u}) = \boldsymbol{0}$. Make sure your objective is successfully optimized by both gradient descent `SECOND_ORDER_SOLVER = false;` and Newton's method `SECOND_ORDER_SOLVER = true;`
   NOTE: The target position $\boldsymbol{x}'$ is drawn as a purple dot that you can drag around.
   HINT: Newton's method likes to have a small regularizer on $|\boldsymbol{u}|^2$.
   HINT: Don't be afraid to spend a little time tuning the weights of your objective. Your solution should work really, really well.

5. Try optimizing for a few different targets with Newton's method. How many steps does Newton's method usually take to converge for this problem? Why?

Now imagine there's a planet at $\boldsymbol{x}_{\text{planet}}$. Call $\boldsymbol{r} = \boldsymbol{x}_{\text{planet}} - \boldsymbol{x}$. The gravitational force acting on the spaceship is probably something like

$$\boldsymbol{F}_{\text{gravity}} = \frac{GMm}{|\boldsymbol{r}|^2}\hat{\boldsymbol{r}},$$

but please go ahead and assume $GMm = 1$.

6. Finish implementing `get_Fk(...)`, being sure to guard the gravitational force inside an `if (PLANET) {...}` block.
   NOTE: Check the box for `PLANET` to turn it on. It should be pretty clear visually whether you got it right.

7. Optimize again for the spaceship to reach the target, now taking into account the planet (I recommend gradient descent). Drag the planet somewhere nice, and take a screenshot of a successful trajectory optimization. Add it to your README using Markdown: `![Image description](link-to-image)`

direct transcription

To start, please copy over your solution to `stepPhysicsExplicitEuler(...)`. Direct transcription will not use this update (since it doesn't really integrate physics forward in time), but I wrote you some nice visualization code that uses it.

Direct transcription simultaneously optimizes over the control and the state (in our case, both position and velocity). In our case it will take the form

$$(\boldsymbol{u}^*, \boldsymbol{x}^*, \boldsymbol{v}^*) = \arg\min_{\boldsymbol{u},\boldsymbol{x},\boldsymbol{v}} \mathcal{O}(\boldsymbol{u}, \boldsymbol{x}, \boldsymbol{v})$$
$$\text{such that } (\boldsymbol{u}^*, \boldsymbol{x}^*, \boldsymbol{v}^*) \text{ satisfies } \boldsymbol{F} = m\boldsymbol{a}.$$

Note that it does not integrate physics forward in time in the typical sense. Rather it enforces physics as a constraint. Our optimization variables are the control, position, and velocity trajectories

$$\boldsymbol{u} = \begin{bmatrix} \boldsymbol{u}_0 \\ \boldsymbol{u}_1 \\ \vdots \\ \boldsymbol{u}_K \end{bmatrix}, \ \boldsymbol{x} = \begin{bmatrix} \tilde{\boldsymbol{x}_0} \\ \boldsymbol{x}_1 \\ \vdots \\ \boldsymbol{x}_K \end{bmatrix}, \ \boldsymbol{v} = \begin{bmatrix} \tilde{\boldsymbol{v}_0} \\ \boldsymbol{v}_1 \\ \vdots \\ \boldsymbol{v}_K \end{bmatrix}.$$

Please note that $\tilde{\boldsymbol{x}_0}$ and $\tilde{\boldsymbol{v}_0}$ here are **not** the same as the known initial conditions of the spaceship $\boldsymbol{x}_0, \boldsymbol{v}_0$. Note also that $\boldsymbol{u}_K$ is totally irrelevant, and will not show up in your objective (except in your regularizer!)

8. Implement $\mathcal{O}(\boldsymbol{u}, \boldsymbol{x}, \boldsymbol{v})$ in `TranscriptionObjective::evaluate(...)`, and optimize using Newton's method.
   NOTE: The final trajectory you get must closely match the dark simulation trajectory shown for `DRAW_REAL_PHYSICS_FOR_COMPARISON = true;`

   HINTS:

   - Your objective should have $\boldsymbol{x}_K = \boldsymbol{x}'$ and $\boldsymbol{v}_K = \boldsymbol{0}$.
   - Add a soft constraint to enforce Equation (3) for $k = 0, ..., K-1$.
   - Add a soft constraint to enforce $\tilde{\boldsymbol{x}_0} = \boldsymbol{x}_0$.
   - Add a soft constraint to enforce $\tilde{\boldsymbol{v}_0} = \boldsymbol{v}_0$.
   - Add a $|\cdot|^2$ regularizer on the vector of optimization variables $\begin{bmatrix} \boldsymbol{u} & \boldsymbol{x} & \boldsymbol{v} \end{bmatrix}^T$.

9. Now try optimizing with gradient descent. Does it work? Why or why not?
   NOTE: A correct answer could be just one or two sentences long.
   HINT: Consider using gradient descent with line search to minimize the toy problem $\mathcal{O}(x, y) = x^2 + y^2 + 10^6(x-1)^2$ starting from the initial guess $(1, 1)$.

challenge problem

Open `challenge-app.cpp`. This is a slightly modified version of `shooting-app.cpp` that uses semi-implicit Euler as the integrator. The goal is to put the spaceship into a desired orbit, with the catch that you have only K frames to do it (let's pretend you run out of fuel or something). For all time $t \geq t_K$ you will no longer be able to supply thrust ($\boldsymbol{u}_k = \mathbf{0}$ for all $k \geq K$).

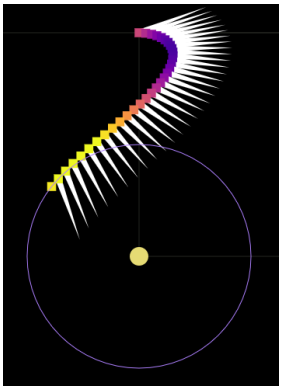
The code minimizes $\mathcal{O}(\boldsymbol{u})$ using gradient descent, where $\boldsymbol{u} = \begin{bmatrix} \boldsymbol{u}_0 & \boldsymbol{u}_1 & \cdots & \boldsymbol{u}_{K-1} \end{bmatrix}^T$. The code simulates (`SIMULATE = true`) by sending

$$\boldsymbol{u}_{\text{sim}} = (\boldsymbol{u}_0, ..., \boldsymbol{u}_{K-1}, \mathbf{0}, \mathbf{0}, ...)$$

to the spaceship, where $\boldsymbol{u}$ is the current best guess stored in `u_curr`.

10. Implement an objective that brings the spaceship into orbit along the purple line.

The optimal orbit entering trajectory $\boldsymbol{x}(\boldsymbol{u}^*)$ might be something like this:



Note that if you simulate for a few minutes (or longer), you should still see the spaceship tracing the purple line (thank you semi-implicit Euler).