hj_reachability

May 19, 2024

[1]:
```
"""
Starter code for the problem "Hamilton-Jacobi reachability".

Autonomous Systems Lab (ASL), Stanford University
"""

import os

import jax
import jax.numpy as jnp
import numpy as np

import hj_reachability as hj

import matplotlib.pyplot as plt
from animations import animate_planar_quad
```

[2]:
```
9.807 * 2.5 * 0.75
```

[2]: 18.388125000000002

[3]:
```
# Define problem ingredients (exercise parts (a), (b), (c)).


class PlanarQuadrotor:

    def __init__(self):
        # Dynamics constants
        # yapf: disable
        self.g = 9.807         # gravity (m / s**2)
        self.m = 2.5           # mass (kg)
        self.l = 1.0           # half-length (m)
        self.Iyy = 1.0         # moment of inertia about the out-of-plane axis␣
 ↪(kg * m**2)
        self.Cd_v = 0.25       # translational drag coefficient
        self.Cd_phi = 0.02255  # rotational drag coefficient
        # yapf: enable
```

```python
        # Control constraints
        self.max_thrust_per_prop = (
            0.75 * self.m * self.g
        )  # total thrust-to-weight ratio = 1.5
        self.min_thrust_per_prop = (
            0  # at least until variable-pitch quadrotors become mainstream :D
        )

    def full_dynamics(self, full_state, control):
        """Continuous-time dynamics of a planar quadrotor expressed as an ODE.
↪"""
        x, v_x, y, v_y, phi, omega = full_state
        T_1, T_2 = control
        return jnp.array(
            [
                v_x,
                (-(T_1 + T_2) * jnp.sin(phi) - self.Cd_v * v_x) / self.m,
                v_y,
                ((T_1 + T_2) * jnp.cos(phi) - self.Cd_v * v_y) / self.m - self.
↪g,
                omega,
                ((T_2 - T_1) * self.l - self.Cd_phi * omega) / self.Iyy,
            ]
        )

    def dynamics(self, state, control):
        """Reduced (for the purpose of reachable set computation)␣
↪continuous-time dynamics of a planar quadrotor."""
        y, v_y, phi, omega = state
        T_1, T_2 = control
        return jnp.array(
            [
                v_y,
                ((T_1 + T_2) * jnp.cos(phi) - self.Cd_v * v_y) / self.m - self.
↪g,
                omega,
                ((T_2 - T_1) * self.l - self.Cd_phi * omega) / self.Iyy,
            ]
        )

    def optimal_control(self, state, grad_value):
        """Computes the optimal control realized by the HJ PDE Hamiltonian.

        Args:
            state: An unbatched (!) state vector, an array of shape `(4,)`␣
↪containing `[y, v_y, phi, omega]`.
```

```
            grad_value: An array of shape `(4,)` containing the gradient of the␣
↪value function at `state`.

        Returns:
            A vector of optimal controls, an array of shape `(2,)` containing␣
↪`[T_1, T_2]`, that minimizes
            `grad_value @ self.dynamics(state, control)`.
        """
        # PART (a): WRITE YOUR CODE BELOW␣
↪###############################################
        # You may find `jnp.where` to be useful; see corresponding numpy␣
↪docstring:
        # https://numpy.org/doc/stable/reference/generated/numpy.where.html
        control = jnp.where(
            jnp.array(
                [
                    jnp.dot(grad_value,  self.dynamics(state, [self.
↪min_thrust_per_prop, self.min_thrust_per_prop])) < \
                    jnp.dot(grad_value,  self.dynamics(state, [self.
↪max_thrust_per_prop, self.min_thrust_per_prop])),
                    jnp.dot(grad_value,  self.dynamics(state, [self.
↪min_thrust_per_prop, self.min_thrust_per_prop])) < \
                    jnp.dot(grad_value,  self.dynamics(state, [self.
↪min_thrust_per_prop, self.max_thrust_per_prop])),
                ]
            ),
            jnp.array([self.min_thrust_per_prop,self.min_thrust_per_prop]),
            jnp.array([self.max_thrust_per_prop,self.max_thrust_per_prop]),
        )
        return control
       ␣
↪#################################################################################

    def hamiltonian(self, state, time, value, grad_value):
        """Evaluates the HJ PDE Hamiltonian."""
        del time, value  # unused
        control = self.optimal_control(state, grad_value)
        return grad_value @ self.dynamics(state, control)

    def partial_max_magnitudes(self, state, time, value, grad_value_box):
        """Computes the max magnitudes of the Hamiltonian partials over the␣
↪`grad_value_box` in each dimension."""
        del time, value, grad_value_box  # unused
        y, v_y, phi, omega = state
        return jnp.array(
            [
```

```
                jnp.abs(v_y),
                (
                    2 * self.max_thrust_per_prop * jnp.abs(jnp.cos(phi))
                    + self.Cd_v * jnp.abs(v_y)
                )
                / self.m
                + self.g,
                jnp.abs(omega),
                (
                    (self.max_thrust_per_prop - self.min_thrust_per_prop) *␣
↪self.l
                    + self.Cd_phi * jnp.abs(omega)
                )
                / self.Iyy,
            ]
        )
```

[4]:
```
def test_optimal_control(n=10, seed=0):
    planar_quadrotor = PlanarQuadrotor()
    optimal_control = jax.jit(planar_quadrotor.optimal_control)#
    np.random.seed(seed)
    states = 5 * np.random.normal(size=(n, 4))
    grad_values = np.random.normal(size=(n, 4))
    try:
        for state, grad_value in zip(states, grad_values):
            if not jnp.issubdtype(
                optimal_control(state, grad_value).dtype, jnp.floating
            ):
                raise ValueError(
                    "`PlanarQuadrotor.optimal_control` must return a `float`␣
↪array (i.e., not `int`)."
                )
            opt_hamiltonian_value = grad_value @ planar_quadrotor.dynamics(
                state, optimal_control(state, grad_value)
            )
            for T_1 in (
                planar_quadrotor.min_thrust_per_prop,
                planar_quadrotor.max_thrust_per_prop,
            ):
                for T_2 in (
                    planar_quadrotor.min_thrust_per_prop,
                    planar_quadrotor.max_thrust_per_prop,
                ):
                    hamiltonian_value = grad_value @ planar_quadrotor.dynamics(
                        state, np.array([T_1, T_2])
                    )
                    if opt_hamiltonian_value > hamiltonian_value + 1e-4:
```

```
                        raise ValueError(
                            "Check your logic for `PlanarQuadrotor.
↪optimal_control`; with "
                            f"`state` {state} and `grad_value` {grad_value},␣
↪got optimal control"
                            f"{optimal_control(state, grad_value)} with␣
↪corresponding Hamiltonian value "
                            f"{opt_hamiltonian_value:7.4f} but {np.array([T_1,␣
↪T_2])} has a lower corresponding "
                            f"value {hamiltonian_value:7.4f}."
                        )
    except (jax.errors.JAXTypeError, jax.errors.JAXIndexError, AssertionError)␣
↪as e:
        print(
            "`PlanarQuadrotor.optimal_control` must be implemented using only␣
↪`jnp` operations; "
            "`np` may only be used for constants, "
            "and `jnp.where` must be used instead of native python control flow␣
↪(`if`/`else`)."
        )
        raise e
test_optimal_control()
# test_target_set()
# test_envelope_set()
```

```
An NVIDIA GPU may be present on this machine, but a CUDA-enabled jaxlib is not
installed. Falling back to cpu.
```

[34]:

```
def target_set(state):
    """A real-valued function such that the zero-sublevel set is the target set.

    Args:
        state: An unbatched (!) state vector, an array of shape `(4,)`␣
↪containing `[y, v_y, phi, omega]`.

    Returns:
        A scalar, nonpositive iff the state is in the target set.
    """
    # PART (b): WRITE YOUR CODE BELOW␣
↪###############################################
    # raise NotImplementedError
     = [[3, 7], [-1, 1], [-np.pi/12, np.pi/12], [-1, 1]]

    h = jnp.array( [(state[i] -  _s[0])*(state[i] -  _s[1]) for i, _s in␣
↪enumerate( )] )
```

```
    return jnp.max(h)

 ␣
↪##################################################################################
```

[35]:
```
def test_target_set():
    try:
        in_states = [
            np.array([5.0, 0.0, 0.0, 0.0]),
            np.array([6.0, 0.1, 0.1, 0.1]),
            # feel free to add test cases
        ]
        out_states = [
            np.array([2.0, 0.0, 0.0, 0.0]),
            np.array([5.0, 2.0, 0.0, 0.0]),
            np.array([5.0, 0.0, 2.0, 0.0]),
            np.array([5.0, 0.0, 0.0, 2.0]),
            # feel free to add test cases
        ]
        for x in in_states:
            if not jnp.issubdtype(target_set(x).dtype, jnp.floating):
                raise ValueError(
                    "`target_set` must return a `float` scalar (i.e., not␣
↪`int`)."
                )
            if target_set(x) > 0:
                raise ValueError(
                    f"Check your logic for `target_set`; for `state` {x} (in)␣
↪you have target_set(state) = "
                    f"{target_set(x)}."
                )
        for x in out_states:
            if target_set(x) <= 0:
                raise ValueError(
                    f"Check your logic for `target_set`; for `state` {x} (out)␣
↪you have target_set(state) = "
                    f"{target_set(x)}."
                )
    except (jax.errors.JAXTypeError, jax.errors.JAXIndexError, AssertionError)␣
↪as e:
        print(
            "`target_set` must be implemented using only `jnp` operations, "
            "`np` may only be used for constants, "
            "and `jnp.where` must be used instead of native python control flow␣
↪(`if`/`else`)."
        )
        raise e
```

```
test_target_set()
```

[36]:
```
def envelope_set(state):
    """A real-valued function such that the zero-sublevel set is the␣
↪operational envelope.

    Args:
        state: An unbatched (!) state vector, an array of shape `(4,)`␣
↪containing `[y, v_y, phi, omega]`.

    Returns:
        A scalar, nonpositive iff the state is in the operational envelope.
    """
    # PART (c): WRITE YOUR CODE BELOW␣
↪###############################################
      = [[1, 9], [-6, 6], [-np.inf, np.inf], [-8, 8]]

    e = jnp.array( [(state[i] - _s[0])*(state[i] - _s[1]) for i, _s in␣
↪enumerate( )] )

    e = jnp.where(jnp.array([True, True, False, True]), e, jnp.array([0., 0.,␣
↪-10., 0.]))

    return jnp.max(e)
  ␣
↪################################################################################
```

[37]:
```
def test_envelope_set():
    try:
        in_states = [
            np.array([5.0, 0.0, 0.0, 0.0]),
            np.array([7.0, 5.0, 100.0, 6.0]),
            # feel free to add test cases
        ]
        out_states = [
            np.array([0.0, 0.0, 0.0, 0.0]),
            np.array([5.0, 8.0, 0.0, 0.0]),
            np.array([5.0, 0.0, 0.0, 10.0]),
            # feel free to add test cases
        ]
        for x in in_states:
            if not jnp.issubdtype(envelope_set(x).dtype, jnp.floating):
                raise ValueError(
                    "`envelope_set` must return a `float` scalar (i.e., not␣
↪`int`)."
                )
```

```
            if envelope_set(x) > 0:
                raise ValueError(
                    f"Check your logic for `envelope_set`; for `state` {x} (in)␣
↪you have envelope_set(state) = "
                    f"{envelope_set(x)}."
                )
        for x in out_states:
            if envelope_set(x) <= 0:
                raise ValueError(
                    f"Check your logic for `envelope_set`; for `state` {x}␣
↪(out) you have envelope_set(state) = "
                    f"{envelope_set(x)}."
                )
    except (jax.errors.JAXTypeError, jax.errors.JAXIndexError, AssertionError)␣
↪as e:
        print(
            "`envelope_set` must be implemented using only `jnp` operations, "
            "`np` may only be used for constants, "
            "and `jnp.where` must be used instead of native python control flow␣
↪(`if`/`else`)."
        )
        raise e
test_envelope_set()
```

[38]:

```
# Set up problem for use with PDE solver.
planar_quadrotor = PlanarQuadrotor()
state_domain = hj.sets.Box(
    lo=np.array([0.0, -8.0, -np.pi, -10.0]), hi=np.array([10.0, 8.0, np.pi, 10.
↪0])
)
grid_resolution = (
    25,
    25,
    30,
    25,
)  # can/should be increased if running on GPU, or if extra patient
grid = hj.Grid.from_lattice_parameters_and_boundary_conditions(
    state_domain, grid_resolution, periodic_dims=2
)

target_values = hj.utils.multivmap(target_set, np.arange(4))(grid.states)
envelope_values = hj.utils.multivmap(envelope_set, np.arange(4))(grid.states)
terminal_values = np.maximum(target_values, envelope_values)

solver_settings = hj.SolverSettings.with_accuracy(
    "medium",  # can/should be changed to "very_high" if running on GPU, or if␣
↪extra patient
```

```
    hamiltonian_postprocessor=lambda x: jnp.minimum(x, 0),
    value_postprocessor=lambda t, x: jnp.maximum(x, envelope_values),
)

# Propagate the HJ PDE _backwards_ in time.
initial_time = 0.0
final_time = -5.0
yn = None
if os.path.exists("hj_reachability_values.npz"):
    yn = (
        input(
            "Existing hj_reachability_values.npz file found from a previous␣
↪solve; use it (Y/n)? "
        )
        .lower()
        .strip()
    )
if yn is None or yn == "n":
    print("Computing the value function by solving the HJ PDE.")
    values = hj.step(
        solver_settings,
        planar_quadrotor,
        grid,
        initial_time,
        terminal_values,
        final_time,
    ).block_until_ready()
    print("Saving the value function to hj_reachability_values.npz.")
    np.savez("hj_reachability_values.npz", values=values)
else:
    print("Loading previously computed value function from␣
↪hj_reachability_values.npz.")
    values = np.load("hj_reachability_values.npz")["values"]
grad_values = grid.grad_values(values)
```

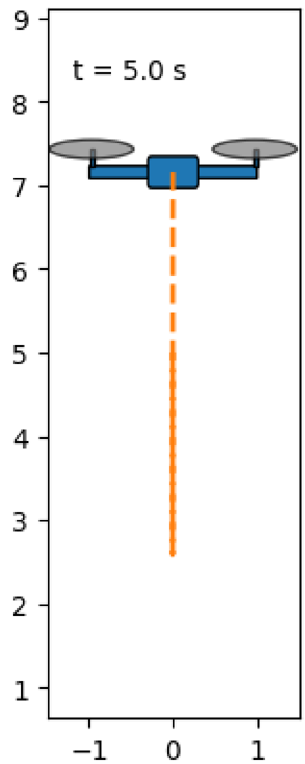
```
Computing the value function by solving the HJ PDE.

100%|##################################|  5.0000/5.0 [21:33<00:00, 258.74s/sim_s]

Saving the value function to hj_reachability_values.npz.
```

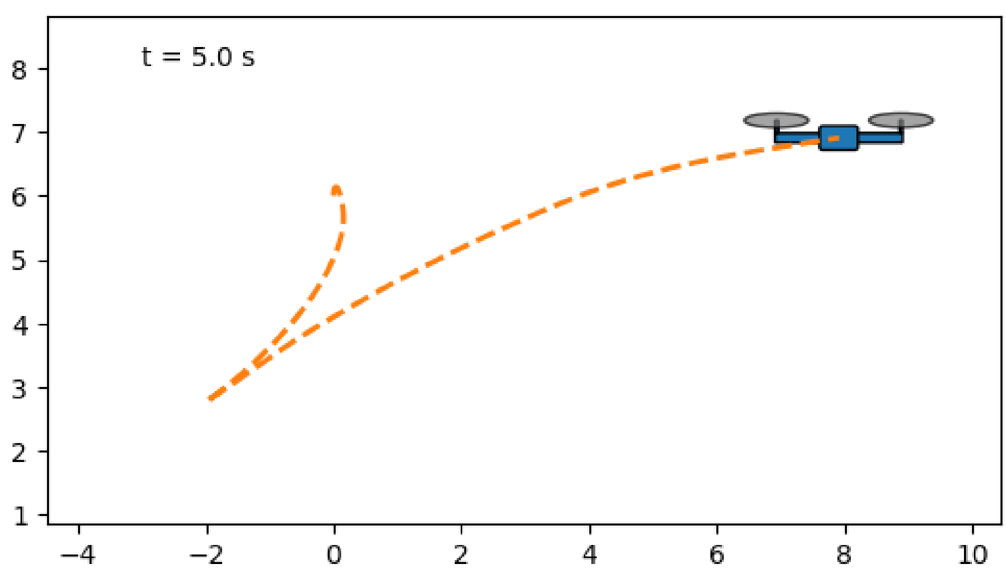
[45]:
```
# Utilities for rolling out the optimal controls and visualizing.


@jax.jit
def optimal_step(full_state, dt):
    state = full_state[2:]
    grad_value = grid.interpolate(grad_values, state)
```

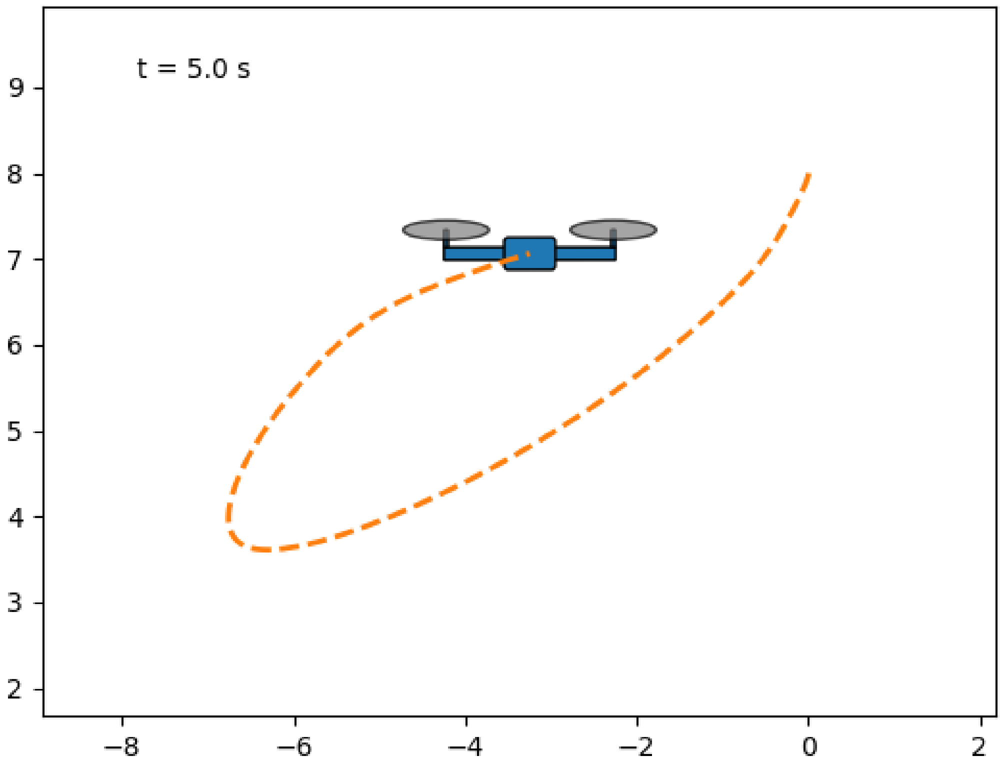
```python
    control = planar_quadrotor.optimal_control(state, grad_value)
    return full_state + dt * planar_quadrotor.full_dynamics(full_state, control)


def optimal_trajectory(full_state, dt=1 / 100, T=5):
    full_states = [full_state]
    t = np.arange(T / dt) * dt
    for _ in t:
        full_states.append(optimal_step(full_states[-1], dt))
    return t, np.array(full_states)


def animate_optimal_trajectory(full_state, dt=1 / 100, T=5,␣
 ↪display_in_notebook=False):
    t, full_states = optimal_trajectory(full_state, dt, T)
    value = grid.interpolate(values, full_state[2:])
    fig, anim = animate_planar_quad(
        t,
        full_states[:, 0],
        full_states[:, 2],
        full_states[:, 4],
        # f"V = {value:7.4f}",
        # display_in_notebook=display_in_notebook,
    )
    return fig, anim
```

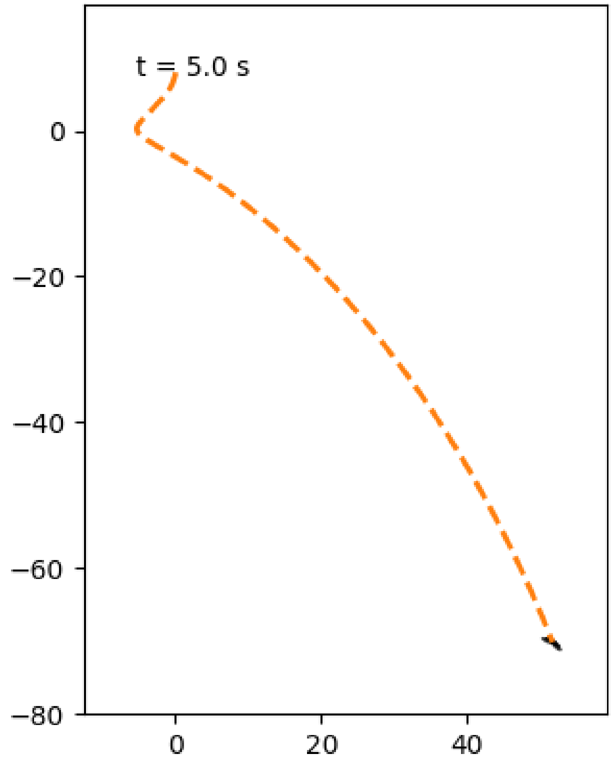
[46]:
```python
# Dropping the quad straight down (v_y = -5, mimicking waiting for a sec after␣
 ↪the drop to turn the props on).
state = [5.0, -5.0, 0.0, 0.0]
fig, ani = animate_optimal_trajectory(np.array([0, 0] + state))
ani.save("planar_quad_1.mp4", writer="ffmpeg")
plt.show()

# Flipping the quad up into the air.
state = [6.0, 2.0, -3 * np.pi / 4, -4.0]
fig, ani = animate_optimal_trajectory(np.array([0, 0] + state))
ani.save("planar_quad_2.mp4", writer="ffmpeg")
plt.show()

# Dropping the quad like a falling leaf.
state = [8.0, -0.8, np.pi / 2, 2.0]
fig, ani = animate_optimal_trajectory(np.array([0, 0] + state))
ani.save("planar_quad_3.mp4", writer="ffmpeg")
plt.show()

# Too much negative vertical velocity to recover before hitting the floor.
state = [8.0, -3.0, np.pi / 2, 2.0]
```

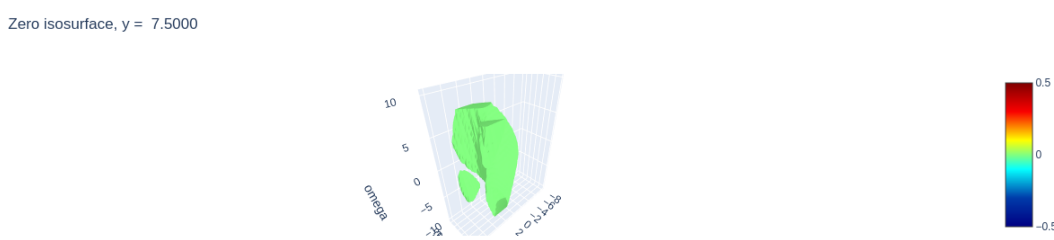
```
fig, ani = animate_optimal_trajectory(np.array([0, 0] + state))
ani.save("planar_quad_4.mp4", writer="ffmpeg")
plt.show()
```





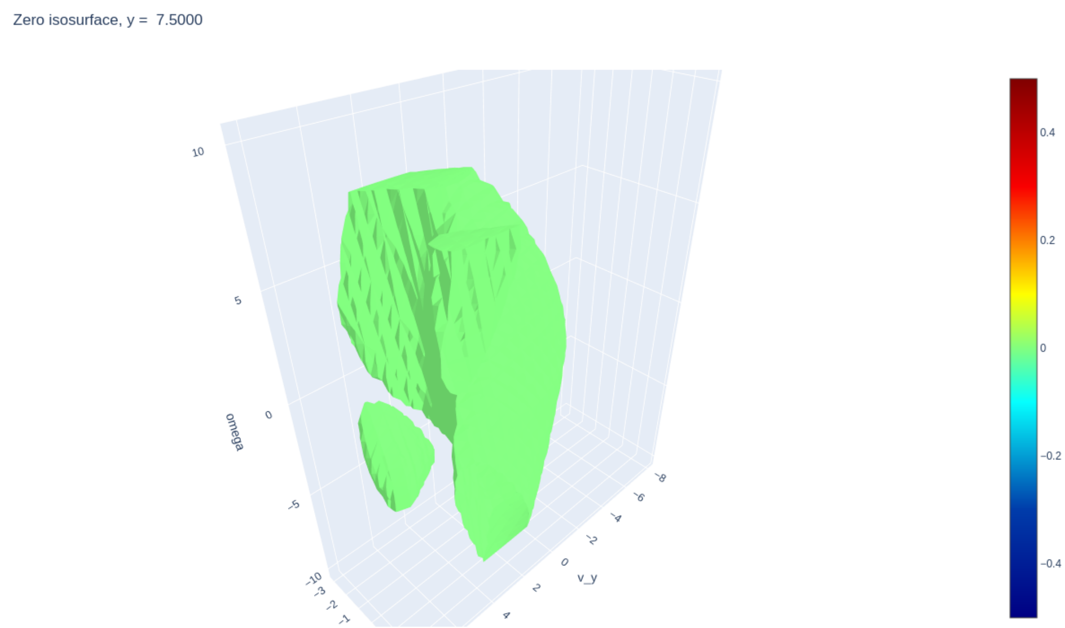

t = 5.0 s
9
8
7
6
5
4
3
2
−8
−6
−4
−2
0
2



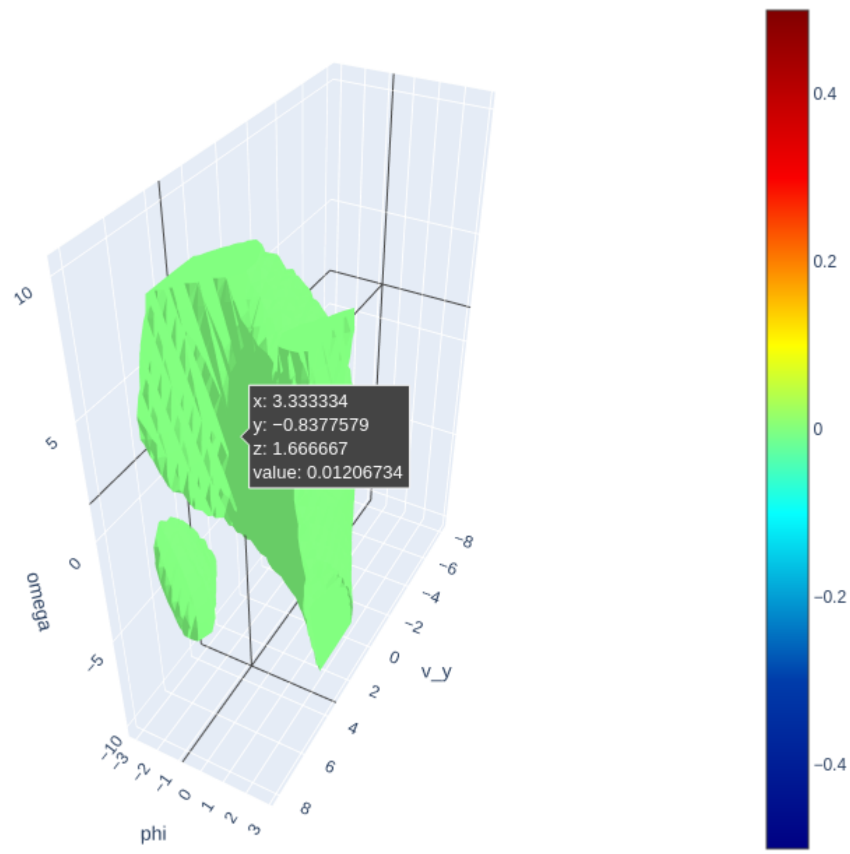


[53]:
```
# Examining an isosurface (exercise part (d)).
import plotly.graph_objects as go

i_y = 18
fig = go.Figure(
    data=go.Isosurface(
        x=grid.states[i_y, ..., 1].ravel(),
        y=grid.states[i_y, ..., 2].ravel(),
        z=grid.states[i_y, ..., 3].ravel(),
        value=values[i_y].ravel(),
```

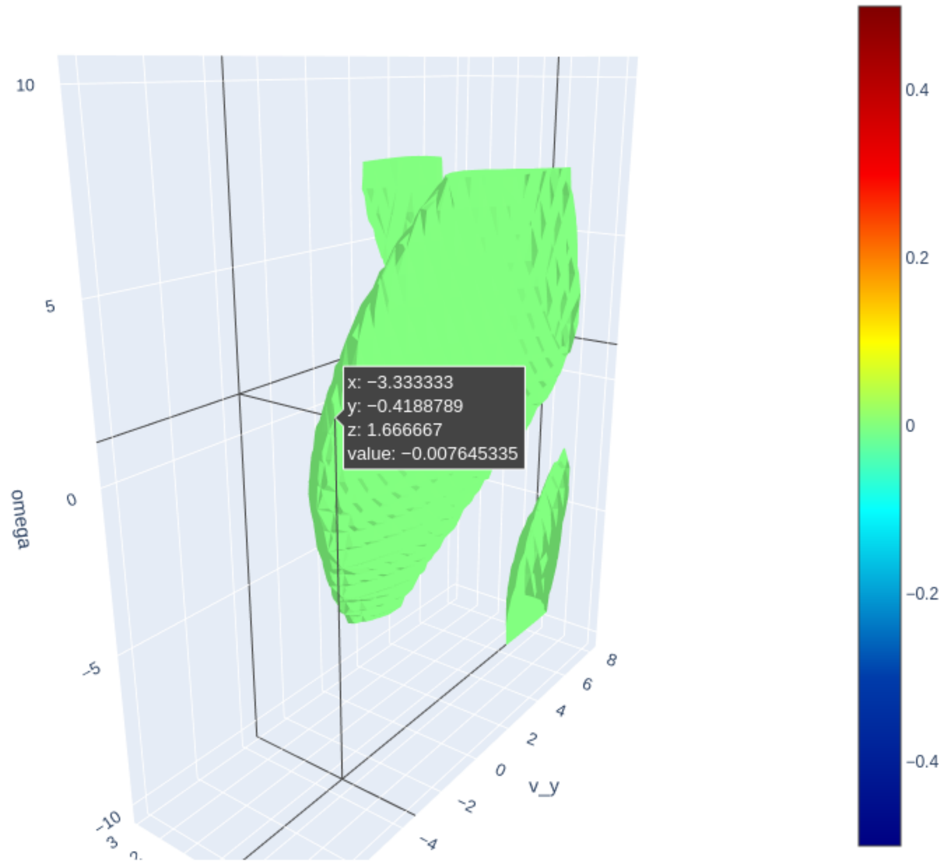
```
        colorscale="jet",
        isomin=0,
        surface_count=1,
        isomax=0,
    ),
    layout_title=f"Zero isosurface, y = {grid.coordinate_vectors[0][i_y]:7.4f}",
    layout_scene_xaxis_title="v_y",
    layout_scene_yaxis_title="phi",
    layout_scene_zaxis_title="omega",
)
fig.update_layout(
    autosize=False,
    width=800,
    height=800,
)
fig.show()
```



Zero isosurface, y = 7.5000



Zero isosurface, y = 7.5000




```
[ ]:
```