Predictive Control

Table of Contents

MuJoCo MPC (MJPC) is a tool for the design and experimentation of predictive control algorithms. Such algorithms solve planning problems of the form:

$$\begin{aligned} \underset{x_{1:T}, u_{1:T}}{\text{minimize}} \quad & \sum_{t=0}^{T} c(x_t, u_t), \\ \text{subject to} \quad & x_{t+1} = f(x_t, u_t), \\ & (x_0, \text{given}), \end{aligned}$$

where the goal is to optimize state, $x \in \mathbf{R}^n$, action $u \in \mathbf{R}^m$ trajectories subject to a model $f : \mathbf{R}^n \times \mathbf{R}^m \to \mathbf{R}^n$ specifying the transition dynamics of the system. The minimization objective is the total return, which is the sum of step-wise values $c : \mathbf{R}^n \times \mathbf{R}^m \to \mathbf{R}_+$ along the trajectory from the current time until the horizon T.

Cost Function

Behaviors are designed by specifying a per-timestep cost l of the form:

$$l(x, u) = \sum_{i}^{M} w_i \cdot \text{n}_i\Big(\mathbf{r}_i(x, u)\Big)$$

The cost is a sum of M terms, each comprising:

- A nonnegative scalar weight $w \geq 0$ that determines the relative importance of this term.
- A norm function $\text{n}(\cdot)$ taking a vector and returning a non-negative scalar that achieves its minimum at $\mathbf{0}$.
- The residual $\mathbf{r}$ is a vector of elements that is "small when the task is solved". Residuals are implemented as custom MuJoCo sensors.

Risk (in)sensitive costs

We additionally provide a convenient family of exponential transformations, $c(x, u) = \rho(l(x, u); R)$, that can be applied to the cost function to model risk-seeking and risk-averse behaviors. This family is parameterized by a scalar $R \in \mathbf{R}$ (our default is $R = 0$) and is given by:

$$\rho(l, R) = \frac{e^{R \cdot l} - 1}{R}.$$

The mapping, $\rho : \mathbf{R}_+ \times \mathbf{R} \to \mathbf{R}_+$, has the following properties:

- It is defined and smooth for any R
- At $R = 0$, it reduces to the identity $\rho(l; 0) = l$
- 0 is a fixed point of the function: $\rho(0; R) = 0$
- It is nonnegative: $\rho(l; R) \geq 0$ if $l \geq 0$
- It is strictly monotonic: $\rho(l; R) > \rho(q; R)$ if $l > q$
- The value of $\rho(l; R)$ has the same unit as l

$R = 0$ can be interpreted as risk neutral, $R > 0$ as risk-averse, and $R < 0$ as risk-seeking. See Whittle et. al, 1981 for more details on risk sensitive control. Furthermore, when $R < 0$, this transformation creates cost functions that are equivalent to the rewards commonly used in the Reinforcement Learning literature. For example using the quadratic norm $\text{n}(\mathbf{r}) = \mathbf{r}^T Q \mathbf{r}$ for some symmetric positive definite (SPD) matrix $Q = \Sigma^{-1}$ and a risk parameter $R = -1$ leads to an inverted-Gaussian cost $1 - e^{-\mathbf{r}^T \Sigma^{-1} \mathbf{r}}$, whose minimisation is equivalent to performing maximum likelihood estimation for the Gaussian.

Cost Derivatives

Many planners (e.g., Gradient Descent and iLQG) utilize derivative information in order to optimize a policy.

Gradients are computed exactly:

$$\frac{\partial l}{\partial x} = \sum_i w_i \cdot \frac{\partial \text{n}_i}{\partial \mathbf{r}} \frac{\partial \mathbf{r}_i}{\partial x},$$

$$\frac{\partial l}{\partial u} = \sum_i w_i \cdot \frac{\partial \text{n}_i}{\partial \mathbf{r}} \frac{\partial \mathbf{r}_i}{\partial u}.$$

Hessians are approximated:

$$\frac{\partial^2 l}{\partial x^2} \approx \sum_i w_i \cdot \left(\frac{\partial \mathbf{r}_i}{\partial x}\right)^T \frac{\partial^2 \mathrm{n}_i}{\partial \mathbf{r}^2} \frac{\partial \mathbf{r}_i}{\partial x},$$

$$\frac{\partial^2 l}{\partial u^2} \approx \sum_i w_i \cdot \left(\frac{\partial \mathbf{r}_i}{\partial u}\right)^T \frac{\partial^2 \mathrm{n}_i}{\partial \mathbf{r}^2} \frac{\partial \mathbf{r}_i}{\partial u},$$

$$\frac{\partial^2 l}{\partial x, \partial u} \approx \sum_i w_i \cdot \left(\frac{\partial \mathbf{r}_i}{\partial x}\right)^T \frac{\partial^2 \mathrm{n}_i}{\partial \mathbf{r}^2} \frac{\partial \mathbf{r}_i}{\partial u},$$

via Gauss-Newton.

Derivatives of n, i.e., $\frac{\partial \mathrm{n}}{\partial \mathbf{r}}$, $\frac{\partial^2 \mathrm{n}}{\partial \mathbf{r}^2}$, are computed analytically.

The residual Jacobians, i.e., $\frac{\partial \mathbf{r}}{\partial x}$, $\frac{\partial \mathbf{r}}{\partial u}$, are computed efficiently using MuJoCo's efficient finite-difference utility.

Risk (in)sensitive derivatives

Likewise, we also provide derivaties for the risk (in)sensitive cost functions.

Gradients are computed exactly:

$$\frac{\partial c}{\partial x} = e^{R \cdot l} \cdot \frac{\partial l}{\partial x},$$

$$\frac{\partial c}{\partial u} = e^{R \cdot l} \cdot \frac{\partial l}{\partial u}.$$

Hessians are approximated:

$$\frac{\partial^2 c}{\partial x^2} \approx e^{R \cdot l} \cdot \frac{\partial^2 l}{\partial x^2} + R \cdot e^{R \cdot l} \cdot \left(\frac{\partial l}{\partial x}\right)^T \frac{\partial l}{\partial x},$$

$$\frac{\partial^2 c}{\partial u^2} \approx e^{R \cdot l} \cdot \frac{\partial^2 l}{\partial u^2} + R \cdot e^{R \cdot l} \cdot \left(\frac{\partial l}{\partial u}\right)^T \frac{\partial l}{\partial u},$$

$$\frac{\partial^2 c}{\partial x, \partial u} \approx e^{R \cdot l} \cdot \frac{\partial^2 l}{\partial x \partial u} + R \cdot e^{R \cdot l} \cdot \left(\frac{\partial l}{\partial x}\right)^T \frac{\partial l}{\partial u},$$

as before, via Gauss-Newton.

Task Specification

In order to create a new task, at least two files are needed:

1. An MJCF file that describes the simulated model and the task parameters.
2. A C++ file that implements the computation of the residual.

It's considered good practice to separate the physics model and task description into two XML files, and include the model file into the task file.

Settings

Configuration parameters for the `Planner` and `Agent` are specified using MJCF custom numeric elements.

```
<custom>
    <numeric
        name="[setting_name]"
        data="[setting_value(s)...]"
    />
</custom>
```

Values in brackets should be replaced by the designer.

- `setting_name` : String specifying setting name
- `setting_value(s)` : Real(s) specifying setting value(s)

`Agent` settings can be specified by prepending `agent_` for corresponding class members.

`Planner` settings can similarly be specified by prepending the corresponding optimizer name, (e.g., `sampling_` , `gradient_` , `ilqg_`).

It is also possible to create GUI elements for parameters that are passed to the residual function. These are specified by the prefix `residual_` , when the suffix will be the display name of the slider:

```
<custom>
    <numeric
        name="residual_[name]"
        data="[value] [lower_bound] [upper_bound]"
    />
</custom>
```

- `value` : Real number for the initial value of the parameter.
- `lower_bound` : Real specifying lower bound of GUI slider.
- `upper_bound` : Real specifying upper bound of GUI slider.

Residual Specification

As mentioned above, the cost is a sum of terms, each computed as a (scalar) norm of a (vector) residual. Each term is defined as a user sensor. The sensor values constitute the residual vector (implemented by the residual function, see below). The norm for each term is defined by the `user` attribute of the respective user sensor, according to the following format:

```
<sensor>
    <user
        name="[term_name]"
        dim="[residual_dimension]"
        user="
            [norm_type]
            [weight]
            [weight_lower_bound]
            [weight_upper_bound]
            [norm_parameters...]"
    />
</sensor>
```

- `term_name` : String describing term name (to appear in GUI).
- `residual_dimension` : number of elements in residual.
- `norm_type` : Integer corresponding to the NormType enum, see norms.
- `weight` : Real number specifying weight value of the cost.
- `weight_lower_bound` , `weight_upper_bound` : Real numbers specifying lower and upper bounds on weight, used for configuring the GUI slider.
- `parameters` : Optional real numbers specifying norm-specific parameters, see norm implementation.

The user sensors defining the residuals should be declared first. Therefore, all sensors should be defined in the task file. After the cost terms are specified, more sensors can be defined to specify the task, e.g., when the residual implementation relies on reading a sensor value. If a sensor named `trace%i` exists, the GUI will use it to draw the traces of the tentative and nominal trajectories.

Computing the residual vectors is specified via a C++ function with the following interface:

```
void residual(
    double* residual,
    const double* parameters,
    const mjModel* model,
    const mjData* data)
```

- `residual` : The function's output. Each element in the double array should be set by this function, corresponding to the ordering and sizes provided by the user sensors in the task XML. See example below.
- `parameters` : Array of doubles, this input is comprised of elements specified using `residual_`
- `model` : the task's `mjModel` .
- `data` : the task's `mjData` . Marked `const` , because this function should not change mjData values.

NOTE: User sensors corresponding to cost terms should be specified before any other sensors.

Examples

As an example, consider these snippets specifying the Swimmer task:

```
<sensor>
  <user name="Control" dim="5" user="0 0.1 0 0.1" />
  <user name="Distance" dim="2" user="2 30 0 100 0.04" />
  <framepos name="trace0" objtype="geom" objname="nose"/>
  <framepos name="nose" objtype="geom" objname="nose"/>
  <framepos name="target" objtype="body" objname="target"/>
</sensor>
```

```
void Swimmer::Residual(double* residual, const double* parameters,
                        const mjModel* model, const mjData* data) {
  mju_copy(residual, data->ctrl, model->nu);
  double* target = mjpc::SensorByName(model, data, "target");
  double* nose = mjpc::SensorByName(model, data, "nose");
  mju_sub(residual + model->nu, nose, target, 2);
}
```

The swimmer's cost has two terms:

1. The first user sensor declares a quadratic norm (index 0) on a vector of 5 entries with weight 0.1. The first line of the `Residual` function copies the control into this memory space.
2. The second user sensor declares an L2 norm (index 2) with weight 30 and one parameters: 0.04. The `Residual` function computes the XY coordinates of the target's position relative to the nose frame, and writes these two numbers to the remaining 2 entries of the residual vector.

The repository includes additional example tasks:

- Humanoid Stand | Walk
- Swimmer
- Walker
- Cart-Pole
- Acrobot
- Particle
- Quadruped Hill | Flat
- In-Hand Manipulation
- Quadrotor
- Panda Arm Manipulation

Task Transitions

The `task` class supports *transitions* for subgoal-reaching tasks. The transition function is called at every step, and is meant to allow the task specification to adapt to the agent's behavior (e.g., moving the target whenever the agent reaches it). This function should test whether a transition condition is fulfilled, and at that event it should mutate the task specification. It has the following interface:

```
int Swimmer::Transition(
  const mjModel* model,
  mjData* data,
  Task* task)
```

- `state` : an integer marking the current task mode. This is useful when the sub-goals are enumerable (e.g., when cycling between keyframes) If the user uses this feature, the function should return a new state upon transition.
- `model` : the task's `mjModel` . It is marked `const` because changes to it here will only affect the GUI and will not affect the planner's copies.
- `data` : the task's `mjData` . A transition should mutate fields of `mjData` , see below.
- `task` : task object.

Because we only sync the `mjData` state between the GUI and the agent's planner, tasks that need transitions should use `mocap` fields or `userdata` to specify goals. For code examples, see the `Transition` functions in these example tasks: Swimmer (relocating the target), Quadruped (iterating along a fixed set of targets) and Hand (recovering when the cube is dropped).

Additionally, custom labeled buttons can be added to the GUI by specifying a string of labels delimited with a pipe character: `|` . For example:

```
<custom>
  <text name="task_transition" data="Label (1)|Label (2)|Label(3)" />
</custom>
```

Planners

The purpose of `Planner` is to find improved policies using numerical optimization.

This library includes three planners that use different techniques to perform this search:

- **Predictive Sampling**
  - random search
  - derivative free
  - spline representation for controls
- **Gradient Descent**
  - requires gradients
  - spline representation for controls
- **Iterative Linear Quadratic Gausian (iLQG)**
  - requires gradients and Hessians
  - direct representation for controls

State

Rollouts are performed after setting both the simulation and initialization states.

Simulation State

This includes:

- configuration: `data->qpos`
- velocity: `data->qvel`
- acceleration: `data->qacc`

Initialization State

This includes:

- mocap: `data->mocap`
- userdata: `data->userdata`
- time: `data->time`

Give feedback