OpenSim Moco Cheat Sheet for the Matlab Interface

MocoStudy

Access the MocoProblem from the study.
```matlab
problem = study.updProblem();
```

Set the model.
```matlab
problem.setModel(Model('model_file.osim'));
```

Set variable bounds.
Set initial time to 0; final time between 0.5 and 1.5 s.
```matlab
problem.setTimeBounds(MocoInitialBounds(0), MocoFinalBounds(0.5, 1.5));
```

The coordinate value must be between 0 and $\pi$ over the phase, and its initial value is 0 and its final value is $\pi/2$.
```matlab
problem.setStateInfo('/jointset/j8/q8/value', [0, pi], [0, pi/2]);
```

The control for actuator `tau0` must be within [-50, 50] over the phase.
```matlab
problem.setControlInfo('/tau0', [-50, 50]);
```

MocoProblem

Optimize static model properties.
Create parameter `myparam` to optimize the mass of Body `/b0` within [0.1, 0.5].
```matlab
problem.addParameter(MocoParameter('myparam', '/b0', 'mass', MocoBounds(0.1, 0.5)));
```

Add cost terms to the problem.
Control · ControlTracking · FinalTime
StateTracking · MarkerTracking · TranslationTracking
OrientationTracking · JointReaction

Minimize the sum of squared controls with weight 1.5.
```matlab
problem.addCost(MocoControlCost('effort', 1.5));
```

Add path constraints to the problem.
Define time-dependent bounds for controls.
```matlab
pathCon = MocoControlBoundConstraint();
problem.addPathConstraint(pathCon);
```

MocoSolver

Initialize the CasADi or Tropter solver.
```matlab
solver = study.initCasADiSolver();
% alternative: solver = study.initTropterSolver();
```

Settings for Tropter and CasADi solvers.
Solve the problem on a grid of 51 mesh points.
```matlab
solver.set_num_mesh_points(51);
```

Transcribe the optimal control problem with the Hermite-Simpson scheme (alternative: `trapzoidal`).
```matlab
solver.set_transcription_scheme('hermite-simpson');
```

Loosen the convergence and constraint tolerances.
```matlab
solver.set_convergence_tolerance(1e-3); solver.set_constraint_tolerance(1e-3);
```

Stop optimization after 500 iterations.
```matlab
solver.set_max_iterations(500);
```

By default, the Hessian is approximated from first derivatives. Set to 'exact' to use an exact Hessian.
```matlab
solver.set_optim_hessian_approximation('exact');
```

Solve the study and obtain a MocoSolution.
```matlab
solution = study.solve();
```

Visualize the solution.
```matlab
study.visualize(solution);
```

Create a guess, randomize it, then set the guess.
```matlab
guess = solver.createGuess(); guess.randomizeAdd();
solver.setGuess(guess);
```

Set the guess from a MocoTrajectory or MocoSolution file.
```matlab
solver.setGuessFile('previous_solution.sto');
```

Settings for only CasADi solver.
By default, CasADi uses `central` finite differences; `forward` differences are faster but less accurate.
```matlab
solver.set_finite_difference_scheme('forward');
```

Turn off parallel calculations.
```matlab
solver.setParallel(0);
```

Monitor solver progress by writing every 10th iterate to file.
```matlab
solver.set_output_interval(10);
```

Compute outputs from the solution.
```matlab
outputs = StdVectorString(); outputs.add('active_force_length_multiplier');
table = study.analyze(solution, outputs);
```
Create a MocoTrajectory.
```
traj = MocoTrajectory('MocoStudy_solution.sto');
traj = MocoTrajectory.createFromStatesControlsTables(
    states, controls);
```

Get time information.
```
traj.getNumTimes();
traj.getInitialTime(); trajectory.getFinalTime();
traj.getTimeMat();
```

Get names of variables.
```
traj.getStateNames(); traj.getControlNames();
traj.getMultiplierNames(); traj.getParameterNames();
```

Get the trajectory/value for a single variable by name.
```
traj.getStateMat(name); traj.getControlMat(name);
traj.getMultiplierMat(name); traj.getParameter(name);
```

Get the trajectories/values for all variables of a given type.
```
traj.getStatesTrajectoryMat();
traj.getControlsTrajectoryMat();
traj.getMultipliersTrajectoryMat();
traj.getParametersMat();
```

Change the number of times in the trajectory.
```
traj.resampleWithNumTimes(150);
```

Set variable values.
```
traj.setTime(times)
traj.setState(stateTraj); traj.setControl(controlTraj);
traj.setParameter(value);
traj.setStatesTrajectory(statesTraj);
traj.insertStatesTrajectory(subsetStates);
```

Randomize the variable values.
```
traj.randomizeAdd();
```

Export the trajectory.
```
traj.write('mocotrajectory.sto');
traj.exportToStatesTable()
traj.exportToStateTrajectory(mocoProblem)
```

Compare two trajectories.
```
traj.isNumericallyEqual(otherTraj);
traj.compareContinuousVariablesRMS(otherTraj);
traj.compareParametersRMS(otherTraj);
```

Functions on only MocoSolution.
```
solution.success(); solution.getStatus();
solution.getObjective(); solution.getNumIterations();
solution.getSolverDuration();
solution.unseal(); % Access a failed solution.
```

The Moco Optimal Control Problem

```
minimize \sum_j w_j f_j(t_0, t_f, y_0, y_f, x_0, x_f, \lambda_0, \lambda_f, p, \int_{t_0}^{t_f} s_{c,j}(t, y(t), x(t), p) \, dt)
```

subject to

- \dot{q} = u
- \dot{\lambda} = f_{app}(t, y(t), x(t), \lambda(t), p) - f_{aux}(q, u(t), \lambda(t), p)
- 0 = \phi(q)
- 0 = \psi(q, u(t), \lambda(t), p)
- 0 = \alpha(q, u(t), \lambda(t), p)
- g_L \leq g(t, y(t), x(t), \lambda(t), p) \leq g_U
- y_{t, L} \leq y(t) \leq y_{t, U}
- y_{x, L} \leq y_x(t) \leq y_{x, U}
- x_{t, L} \leq x(t) \leq x_{t, U}
- x_{x, L} \leq x_x(t) \leq x_{x, U}
- \lambda_{t, L} \leq \lambda(t) \leq \lambda_{t, U}
- \lambda_{x, L} \leq \lambda_x(t) \leq \lambda_{x, U}
- p \in [p_L, p_U]

with respect to

- t: time
- q(t): generalized coordinates
- u(t): generalized speeds
- z(t): auxiliary states (muscle fiber length and activation)
- y(t) = (q(t), u(t), z(t))
- x(t): controls (including muscle activation)
- p: constant parameters
- \lambda: kinematic constraint multipliers
- w_j: weight for j-th cost
- J_j: the j-th cost
- s_{c,j}: integrand used in the j-th cost
- M: mass matrix
- f_{bias}: centripetal and coriolis forces
- G: kinematic constraint Jacobian
- f_{app}: applied forces (gravity, muscles, etc.)
- f_{aux}: auxiliary (muscle) dynamics
- \phi: position-level (holonomic) kinematic constraints
- \psi: velocity-level (non-holonomic) kinematic constraints
- \alpha: acceleration-level kinematic constraints
- g: path constraints
- K: number of endpoint constraint goals
- subscript U: an upper bound
- subscript L: a lower bound