technische universität
dortmund

Nonlinear Model Predictive Control

A Primer on CasADi

Timm Faulwasser

ie3, TU Dortmund

timm.faulwasser@tu-dortmund.de

Version WS21/22.I

Overview

Basics on Algorithmic Differentiation via CasADi

Solving NLPs via CasADI and IPOPT

Solving Optimal Control Problems

Overview

Basics on Algorithmic Differentiation via CasADi

Solving NLPs via CasADI and IPOPT

Solving Optimal Control Problems

CasADi basics

Recall from lecture: derivatives can be computed

- By hand (**time consuming and error prone**)
- Via symbolic differentiation **(typically slow)**
- Via finite differences **(introduces numerical errors)**
- Via *algorithmic differentiation*

CasADi basics

Recall from lecture: derivatives can be computed

- By hand (**time consuming and error prone**)
- Via symbolic differentiation **(typically slow)**
- Via finite differences **(introduces numerical errors)**
- Via *algorithmic differentiation*

CasADi is an open-source tool for **algorithmic differentiation** in combination with **state-of-the-art NLP solvers** and focus on optimal control

`https://web.casadi.org/`

CasADi basics

Recall from lecture: derivatives can be computed

- By hand (**time consuming and error prone**)
- Via symbolic differentiation **(typically slow)**
- Via finite differences **(introduces numerical errors)**
- Via *algorithmic differentiation*

CasADi is an open-source tool for **algorithmic differentiation** in combination with **state-of-the-art NLP solvers** and focus on optimal control

`https://web.casadi.org/`

Features:

- Usable from C++, Python, MATLAB $\rightarrow$ usable from multiple languages
- A basic symbolic framework
- Optistack interface for easy optimal control
- C-code generation

Installation

- Download CasADi from `https://web.casadi.org/get/`
- Add the extracted folder to your MATLAB path

CasADi for Algorithmic Differentiation

We use MATLAB for our demonstration, python syntax is very similar
Example: Compute the Jacobian of $g : \mathbb{R}^2 \to \mathbb{R}^2$

$$g(x) = \begin{pmatrix} \sin(x_1)\cos(3x_2) \\ e^{-x_1} \end{pmatrix}$$

$$\nabla g(x) = \begin{pmatrix} \cos(x_1)\cos(3x_2) & -3\sin(x_1)\sin(3x_2) \\ -e^{-x_1} & 0 \end{pmatrix}$$

CasADi for Algorithmic Differentiation

We use MATLAB for our demonstration, python syntax is very similar
Example: Compute the Jacobian of $g : \mathbb{R}^2 \to \mathbb{R}^2$

$$g(x) = \begin{pmatrix} \sin(x_1)\cos(3x_2) \\ e^{-x_1} \end{pmatrix}$$

$$\nabla g(x) = \begin{pmatrix} \cos(x_1)\cos(3x_2) & -3\sin(x_1)\sin(3x_2) \\ -e^{-x_1} & 0 \end{pmatrix}$$

Step 1: Create CasADi symbolic variables (SX or MX)

```
x = SX.sym('x',2); % creates a 2-dimension symbolic vector
```

Step 2: Set up your symbolic expressions

```
g = [   sin(x(1))*cos(3*x(2));
        exp(-x(1))             ];
```

CasADi for Algorithmic Differentiation

Step 2: Set up your symbolic expressions

```
g = [   sin(x(1))*cos(3*x(2));
        exp(-x(1))             ];
```

Step 3: Compute sensitivities (gradient(...), jacobian(...), hessian(...))

```
dg = jacobian(g,x)
```

CasADi for Algorithmic Differentiation

Step 2: Set up your symbolic expressions

```
g = [   sin(x(1))*cos(3*x(2));
        exp(-x(1))             ];
```

Step 3: Compute sensitivities (gradient(...), jacobian(...), hessian(...))

```
dg = jacobian(g,x)
```

Output:

```
dg =
@1=3, @2=(@1*x_1),
[[(cos(@2)*cos(x_0)), (-(sin(x_0)*(@1*sin(@2))))],
[(-exp((-x_0))), 00 ]]
```

(special CasADi syntax with @-expressions as placeholders)

How to evaluate symbolic expressions?

- convert into function objects

```
dgFun = Function('dg',{x},{dg})
```

How to evaluate symbolic expressions?

- convert into function objects

```
dgFun = Function('dg',{x},{dg})
```

Output:

```
dgFun = dg:(i0[2])->(o0[2x2,3nz]) SXFunction
```

(support/range dimension and #structural zeros)

How to evaluate symbolic expressions?

- convert into function objects

```
dgFun = Function('dg',{x},{dg})
```

Output:

```
dgFun = dg:(i0[2])->(o0[2x2,3nz]) SXFunction
```

(support/range dimension and #structural zeros)

Get $\nabla g((1,\ 2)^\top)$:

```
dgFun([1 2]')
```

Output:

```
ans = [[0.518782, 0.70536], [-0.367879, 00]]
```

Overview

Basics on Algorithmic Differentiation via CasADi

Solving NLPs via CasADI and IPOPT

Solving Optimal Control Problems

Solving NLPs

CasADi NLP interface:

$$\begin{aligned} \underset{x}{\text{minimize:}} \quad & f(x,p) \\ \text{subject to:} \quad & x_{\text{lb}} \leq x \leq x_{\text{ub}} \\ & g_{\text{lb}} \leq g(x,p) \leq g_{\text{ub}} \end{aligned}$$

x : decision vector
p : parameters

Why extra bounds x_{lb}, x_{ub}? $\rightarrow$ more efficient numerical handling

Example: Solve

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & (1-x_1)^2 + 0.2(x_2 - x_1^2)^2 \\ \text{subject to} \quad & \frac{p^2}{4} \leq (x_1+0.5)^2 + x_2^2 \leq p^2 \\ & x_1 \geq 0 \end{aligned}$$

Solving NLPs

Step 1: Set up your decision variables (+ parameters)

```
x = SX.sym('x',2); p = SX.sym('p',1);
```

Step 2: Set up objective f, constraints g, bounds $x_{\mathrm{lb}}, x_{\mathrm{ub}}, g_{\mathrm{lb}}, g_{\mathrm{ub}}$, ...

```
f = (1-x(1))^2 + 0.2*(x(2)-x(1)^2)^2;
g = [ p^2/4 - (x(1)+0.5)^2 + x(2)^2;
      (x(1)+0.5)^2 + x(2)^2 - p^2 ];

lbx = [0; -inf];
ubx = inf;
lbg = -inf;
ubg = 0;
```

Solving NLPs

Step 3: Set up a NLP solver and solve (with zero initial conditions)

```
nlp = struct('x',x, 'f',f, 'g',g, 'p',p);
S = nlpsol('S', 'ipopt', nlp)
sol = S('x0', zeros(2,1),...
        'lam_g0', zeros(2,1),...
        'lam_x0', zeros(2,1),...
        'p',      1.25,...
        'lbx',    lbx,...
        'ubx',    ubx,...
        'lbg',    lbg, ...
        'ubg',    ubg);
```

Solving NLPs

Output:

```
This is Ipopt version 3.12.3, running with linear solver mumps.
NOTE: Other linear solvers might be more efficient (see Ipopt
documentation).

Number of nonzeros in equality constraint Jacobian...:        0
Number of nonzeros in inequality constraint Jacobian.:        4
Number of nonzeros in Lagrangian Hessian.............:        3

Total number of variables............................:        2
variables with only lower bounds:                             1
variables with lower and upper bounds:                        0
variables with only upper bounds:                             0
Total number of equality constraints.................:        0
Total number of inequality constraints...............:        2
inequality constraints with only lower bounds:                0
inequality constraints with lower and upper bounds:           0
inequality constraints with only upper bounds:                2
```

Output (cont'd):

```
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du
0  9.8010002e-001 1.31e-001 9.87e-001  -1.0 0.00e+000    -  0.00e+0
1  6.6794608e-001 0.00e+000 1.01e+001  -1.0 1.76e-001    -  1.19e-0
2  5.7298195e-001 0.00e+000 1.89e+000  -1.0 1.13e-001    -  4.64e-0
3  5.6860369e-001 0.00e+000 8.02e-001  -1.0 8.02e-003   2.0 1.00e+0
4  4.0551835e-001 0.00e+000 5.56e-001  -1.7 1.77e-001    -  5.29e-0
5  4.0688356e-001 0.00e+000 4.28e-001  -1.7 1.28e-002   1.5 1.00e+0
6  1.1751232e-001 0.00e+000 3.91e-001  -1.7 5.45e-001    -  6.60e-0
7  7.8213074e-002 7.65e-002 1.84e-001  -1.7 5.18e-001    -  1.00e+0
8  9.3051090e-002 0.00e+000 1.21e-002  -2.5 7.02e-002    -  1.00e+0
9  9.0600802e-002 0.00e+000 8.76e-005  -3.8 1.87e-002    -  1.00e+0
10 9.0512237e-002 0.00e+000 4.34e-007  -5.7 8.62e-004    -  1.00e+0
11 9.0510219e-002 0.00e+000 1.15e-010  -8.6 1.19e-005    -  1.00e+0

Number of Iterations....: 11
```

Solving NLPs

Output (cont'd):

```
                                   (scaled)                 (unscaled)
Objective...............:  9.051021857953676e-002   9.051021857953
Dual infeasibility......:  1.147774236764931e-010   1.147774236764
Constraint violation....:  0.000000000000000e+000   0.000000000000
Complementarity.........:  2.566280058381725e-009   2.566280058381
Overall NLP error.......:  2.566280058381725e-009   2.566280058381


Number of objective function evaluations             = 12
Number of objective gradient evaluations             = 12
Number of equality constraint evaluations            = 0
Number of inequality constraint evaluations          = 12
Number of equality constraint Jacobian evaluations   = 0
Number of inequality constraint Jacobian evaluations = 12
Number of Lagrangian Hessian evaluations             = 11
Total CPU secs in IPOPT (w/o function evaluations)   = 0.024
Total CPU secs in NLP function evaluations           = 0.000
```

Solving NLPs

Output (cont'd):

```
EXIT: Optimal Solution Found.
S            :   t_proc      (avg)   t_wall      (avg)    n_eval
nlp_f        |        0 (       0)        0 (       0)        12
nlp_g        |        0 (       0)        0 (       0)        12
nlp_grad     |        0 (       0)        0 (       0)         1
nlp_grad_f   |   1.00ms ( 76.92us) 999.00us ( 76.85us)        13
nlp_hess_l   |        0 (       0)        0 (       0)        11
nlp_jac_g    |        0 (       0)        0 (       0)        13
total        |  31.00ms ( 31.00ms)  31.43ms ( 31.43ms)         1
```

- NLP solved successfully
- Solved within milliseconds
- Time spent in sensitivity evaluations negligible (due to AD)
- Most time spent in factoring the KKT system

Solving NLPs

Pitfalls: If converged to a point of local infeasibility or slow convergence

- Try different guess x_0
- Reformulate objective/constraints (e.g. eleminate $\sqrt{\cdot}$)
- Re-scale problem

Output: Solution struct

```
sol =
struct with fields:
f:      [1x1 casadi.DM]
g:      [2x1 casadi.DM]
lam_g:  [2x1 casadi.DM]
lam_p:  [1x1 casadi.DM]
lam_x:  [2x1 casadi.DM]
x:      [2x1 casadi.DM]
```

`x:`	$x^\star$	`lam_g:`	multipliers for g
`f:`	$f(x^\star)$	`lam_p:`	multipliers for p
`g:`	$g(x^\star)$	`lam_x:`	multipliers for $x_{\mathrm{lb}}, x_{\mathrm{ub}}$

Solving NLPs

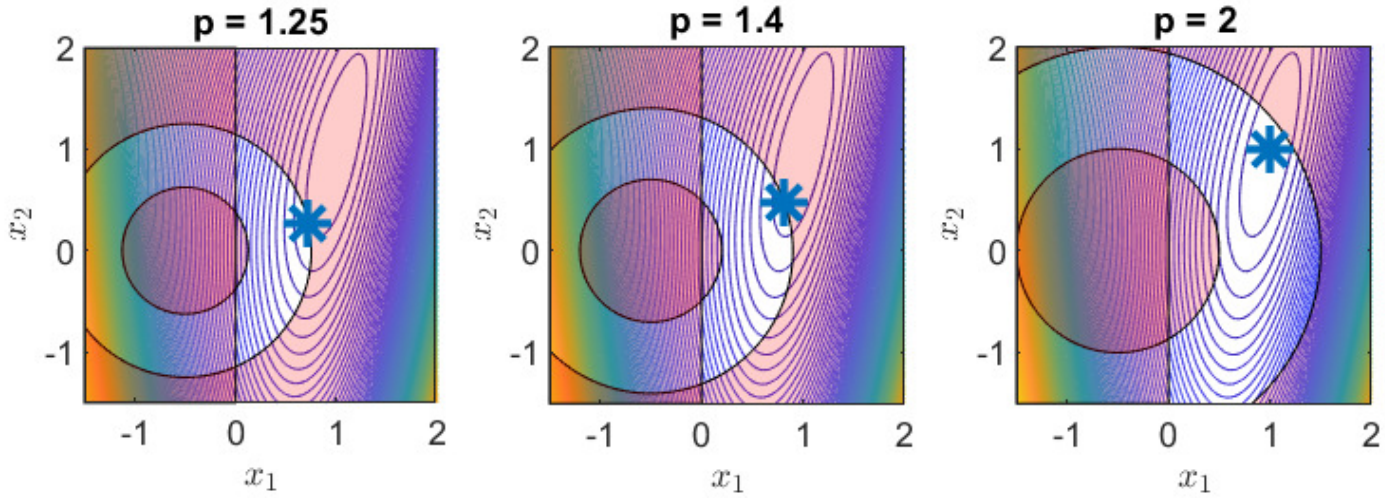
Why parameterized NLPs?

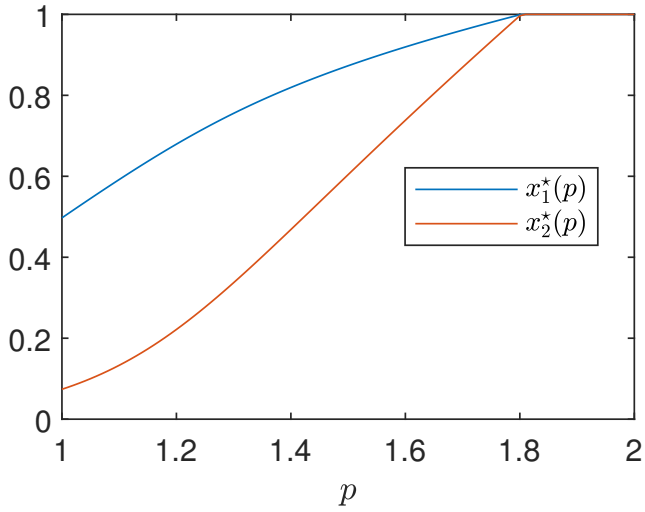
- NLP construction only once
- very relevant for MPC (changing $x(0)$)

Example: Approximate the mapping $x^\star(p)$ numerically for $p \in [1,\ 2]$

```
X = [];
pVec = 1:0.01:2;
for p = pVec
      sol = S('x0', zeros(2,1),...
              'lam_g0', zeros(2,1),...
              'lam_x0', zeros(2,1),...
              'p',      p,...
              'lbx',    lbx,...
              'ubx',    ubx,...
              'lbg',    lbg, ...
              'ubg',    ubg);
      X = [X full(sol.x)];
end
```

Solving NLPs





Source: https://web.casadi.org/blog/nlp_sens/

Overview

Basics on Algorithmic Differentiation via CasADi

Solving NLPs via CasADI and IPOPT

Solving Optimal Control Problems

Solving OCPs

Two possibilities in CasADi:

- Formulate and solve OCP via the (low-level) symbolic framework introduced above
- Formulate and solve OCP via the `Opti` syntax $\rightarrow$ simplified problem setup (now)

Example: A simple minimum-time problem

- Very simple 1d model of a car

$$\begin{pmatrix} \dot{p} \\ \dot{v} \end{pmatrix} = \begin{pmatrix} v \\ u - v \end{pmatrix}$$

- Throttle is control input u
- Arrive at destination as fast as possible
- Time-varying speed-limit $L(p) = 1 - \sin(2\pi p)/2$

Source: https://web.casadi.org/blog/ocp/

Solving OCPs

Formulate the corresponding OCP

$$
\begin{aligned}
&\underset{x(\cdot),\,u(\cdot),\,T}{\text{minimize}} && T \\
&\text{subject to} && \dot{x}(t) = f(x(t), u(t)) \quad t \in [0, T], && \text{dynamic constraints} \\
& && p(0) = 0, && \text{start at position 0} \\
& && v(0) = 0, && \text{start with zero speed} \\
& && p(T) = 1, && \text{the finish line is at position 1} \\
& && 0 \leq u(t) \leq 1, && \text{path constraint: throttle is limited} \\
& && v(t) \leq L(p(t)). && \text{speed limit varying along the track}
\end{aligned}
$$

Step 1: Construct `opti` variables:

```
opti = casadi.Opti();          % optimization problem
X    = opti.variable(2,N+1);   % state trajectory
U    = opti.variable(1,N);     % control trajectory (throttle)
T    = opti.variable();        % final time
```

Source: https://web.casadi.org/blog/ocp/

Solving OCPs

Step 2: Construct objective and integrate dynamics and RK4 integrator (direct multiple shooting):

```
opti.minimize(T);                  % minimize time

f = @(x,u) [x(2);u-x(2)];          % dx/dt = f(x,u)

dt = T/N;
for k=1:N                          % loop over control intervals
      k1 = f(X(:,k),          U(:,k));
      k2 = f(X(:,k)+dt/2*k1, U(:,k));
      k3 = f(X(:,k)+dt/2*k2, U(:,k));
      k4 = f(X(:,k)+dt*k3,   U(:,k));
      x_next = X(:,k) + dt/6*(k1+2*k2+2*k3+k4);

      opti.subject_to(X(:,k+1)==x_next);  % close the gaps
end
```

Source: https://web.casadi.org/blog/ocp/

Solving OCPs

Step 3: Add path and boundary constraints:

```
pos   = X(1,:);
speed = X(2,:);

% -- path constraints ------
limit = @(pos) 1-sin(2*pi*pos)/2;
opti.subject_to(speed<=limit(pos)); % track speed limit
opti.subject_to(0<=U<=1);           % control is limited

% -- boundary conditions ----
opti.subject_to(pos(1)==0);   % start at position 0 ...
opti.subject_to(speed(1)==0); % ... from stand-still
opti.subject_to(pos(N+1)==1); % finish line at position 1

% -- misc. constraints  -----
opti.subject_to(T>=0);        % Time must be positive
```

Source: https://web.casadi.org/blog/ocp/

Solving OCPs

Step 3: Initialize solver and solve OCP:

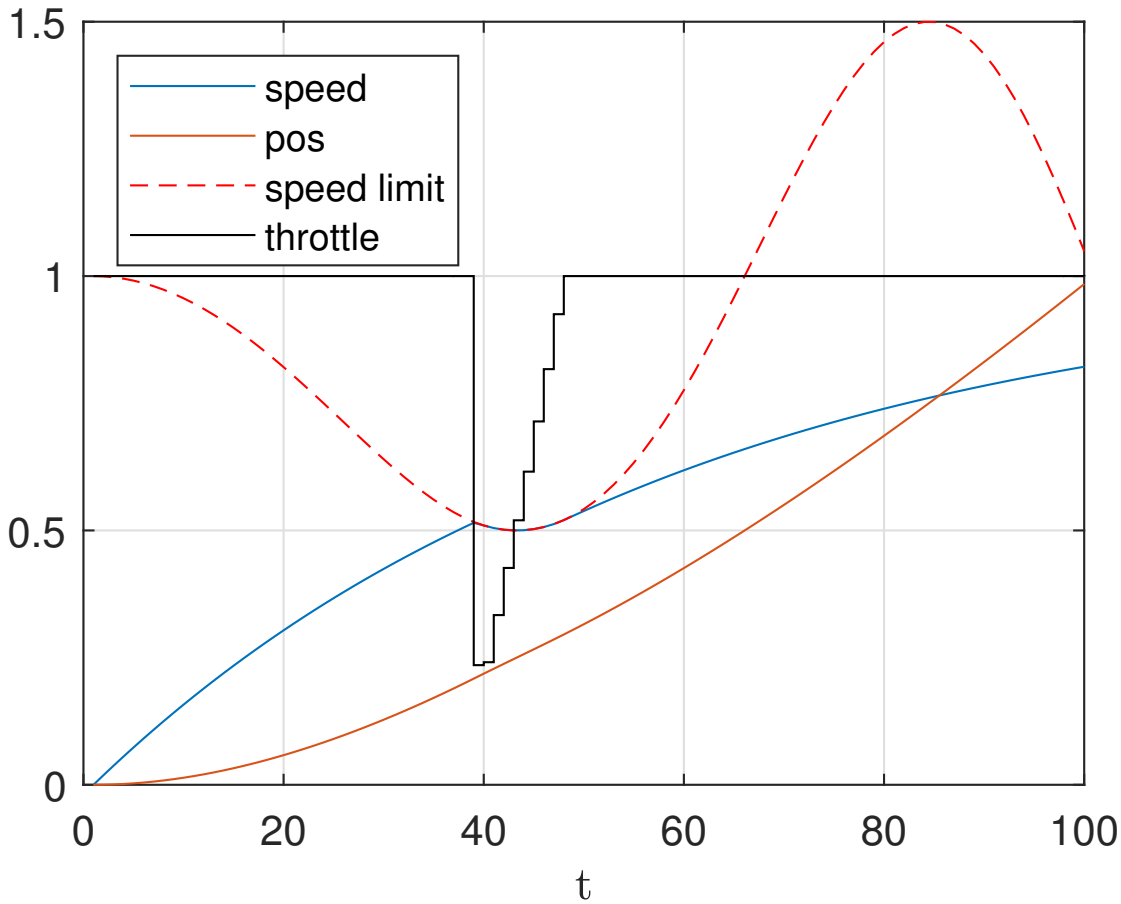
```
% -- initial values for solver --
opti.set_initial(speed, 1);
opti.set_initial(T, 1);

% -- solve NLP                  ---
opti.solver('ipopt'); % set numerical backend
sol = opti.solve();   % actual solve
```

Output:

- IPOPT requires 42 iterations and 216ms (Intel Corei5 10.Gen)

Source: https://web.casadi.org/blog/ocp/

Solving OCPs



For MPC:

- Put the above OCP into an MPC loop
- Change initial condition in each sampling instance

Source: https://web.casadi.org/blog/ocp/