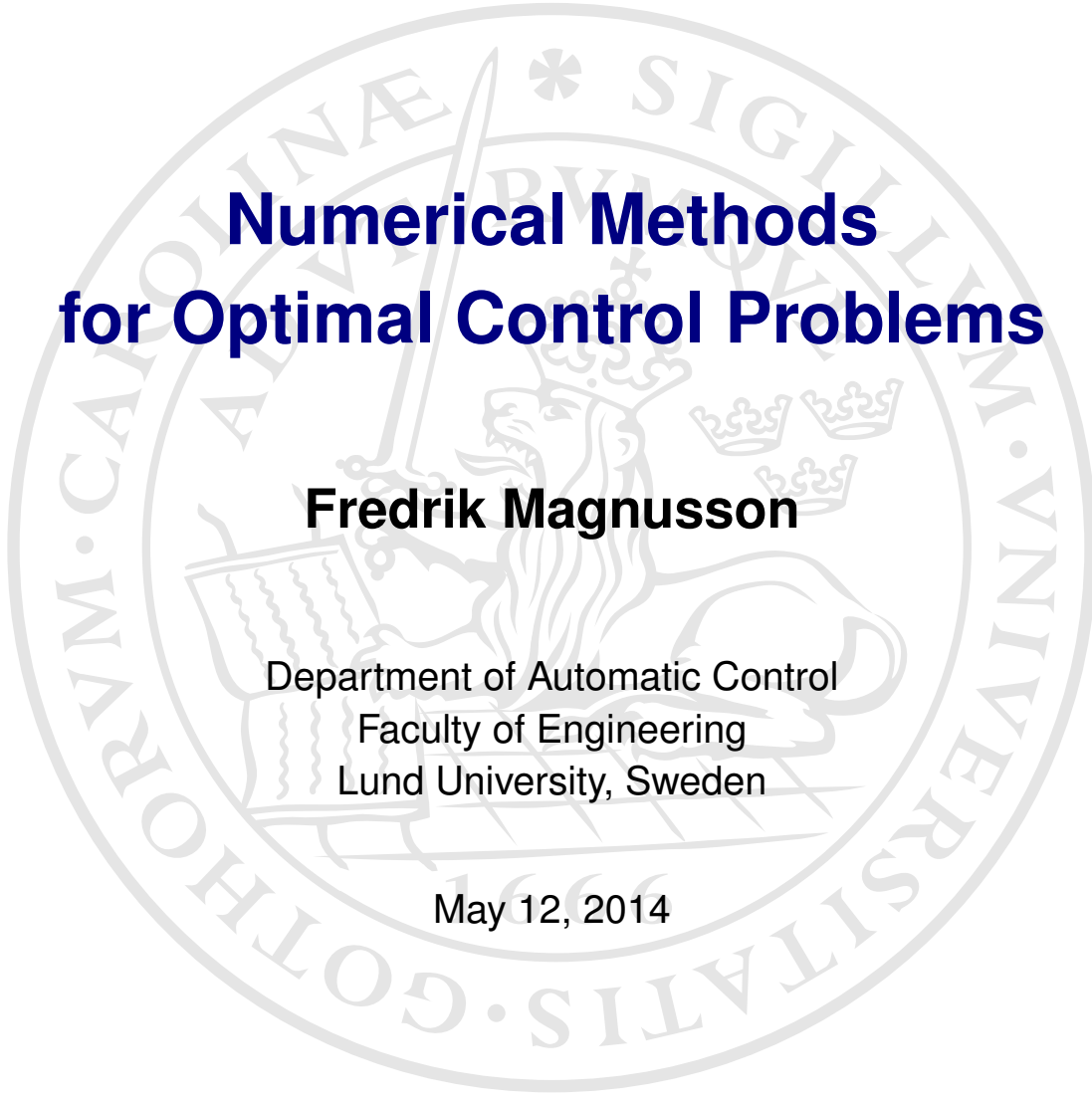


Numerical Methods for Optimal Control Problems

Fredrik Magnusson

Department of Automatic Control
Faculty of Engineering
Lund University, Sweden

May 12, 2014

Using the Numerical Methods in JModelica.org for Optimal Control Problems

Fredrik Magnusson

Department of Automatic Control
Faculty of Engineering
Lund University, Sweden

May 12, 2014

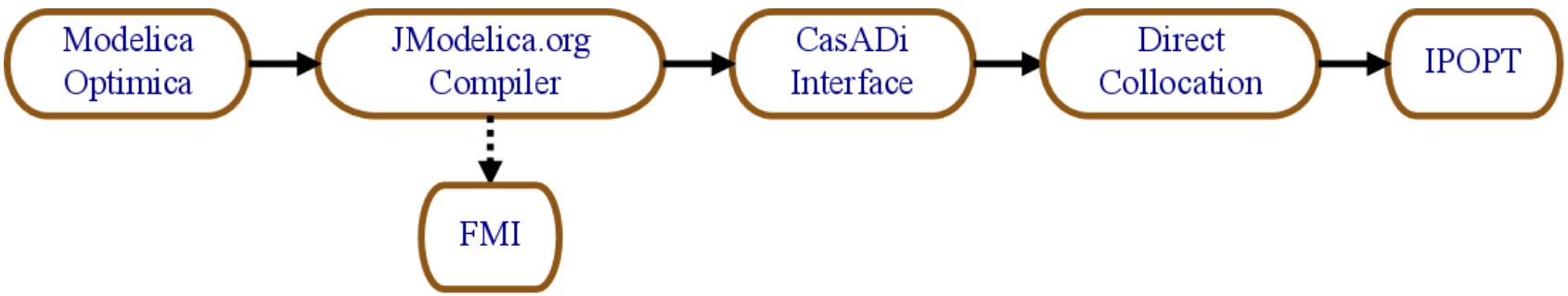
Outline

- Goal: learn how to solve optimal control problems (OCP) numerically using JModelica.org
- Lecture: overview of how JModelica.org works and how to use it
- Exercise: solve some simple problems in Lab C
- Home assignment: solve an interesting OCP of your choosing and send a report to Magnus

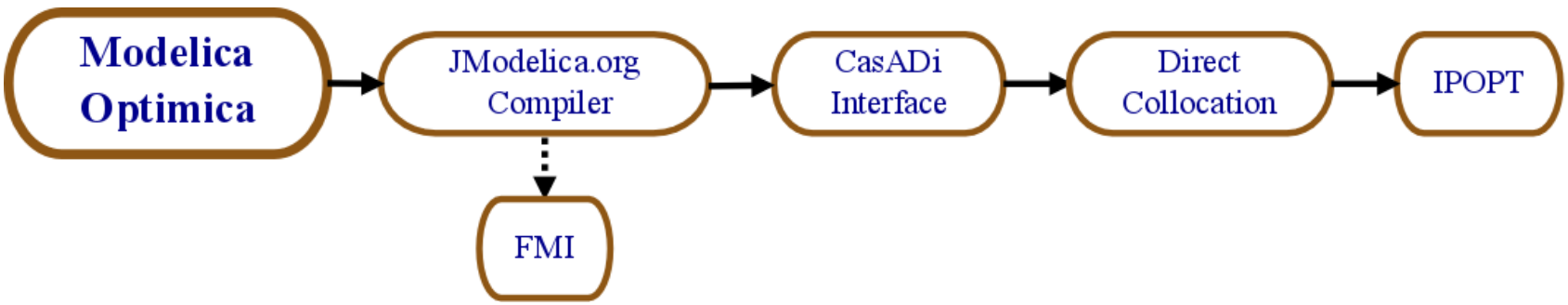
JModelica.org

- Origin is Johan Åkesson's thesis (2007)
- Today mainly developed by Modelon AB, in collaboration with Lund University

The complete JModelica.org toolchain



The complete JModelica.org toolchain



Modelica

- Textual language for modeling dynamic systems
- Differential-algebraic equation (DAE) paradigm
- Object-oriented
- Open standard, supported by $\approx$ 15 different tools
- Most famous tool is Dymola
- Today's tool is JModelica.org

Differential-algebraic equation systems

A general form of DAE systems is

$$F(\dot{x}, x, y, u) = 0, \tag{1}$$

where
x is vector of differentiated variables,
y is vector of algebraic variables (not (necessarily) outputs),
u is vector of control variables.

This is called an implicit DAE. Contains both differential and algebraic equations.

DAE system index

- The system *index* is an important notion for DAE systems.
- Measure of the distance between the DAE system and a corresponding ODE.
- Easier to discuss for the semi-explicit DAE form

$$\dot{x} = f(x, y, u), \tag{2a}$$
$$0 = g(x, y, u). \tag{2b}$$

- If y and g are absent, this is an explicit ODE. Index zero.
- If g_y is non-singular, we can solve (2b) for y and differentiate with respect to time to get an explicit ODE in x and y (implicit function theorem). Index one.
- If g_y is singular, the system is high-index. Index ≥ 2.

DAE system index cont.

- If differentiating the algebraic equation (2b) n times yields an explicit ODE, the system index is n.
- This is called the *differentiation index* of the system. Many other non-equiavlent notions of index exist.
- A sufficient and necessary condition for the implicit system (1) to be low index (≤ 1) is that $\begin{bmatrix} F_{\dot{x}} & F_y \end{bmatrix}$ is non-singular.
- High index is bad; differentation index n means that the system depends on (up to) the n:th-order derivative of u.
- High index numerically challenging (hot topic in numerical analysis in the 90s).

DAE system initial conditions

- Advanced symbolic and numerical algorithms means you do not need to know or worry about system index.
- One exception: initial conditions
- The needed initial conditions depend on the dimension of the state
- For low-index DAE systems, state is the differentiated variable x
- Typically $x(t_0) = x_0$
- I do not know how to compute the dimension of the state for a high-index system...

Modelica syntax

To demonstrate Modelica's syntax, consider the double integrator

$$\dot{x}_1 = x_2,$$
$$\dot{x}_2 = u,$$
$$x_1(0) = 0, \quad x_2(0) = 0.$$

```
model DoubleIntegrator

    Real x1(start=0, fixed=true);
    Real x2(start=0, fixed=true);
    input Real u;

equation

    der(x1) = x2;
    der(x2) = u;

end DoubleIntegrator;
```

Modelica syntax cont.

```
model DoubleIntegrator
    Real x1(start=0, fixed=true);
    Real x2(start=0, fixed=true);
    input Real u;
equation
    der(x1) = x2;
    der(x2) = u;
end DoubleIntegrator;
```

- Declares the class `DoubleIntegrator` of type `model`
- Declares the real-valued variables `x1`, `x2`, and `u`
- The DAE system is defined in the `equation` section (declarative!)
- `der` means derivative with respect to time. No higher-order or partial derivatives!
- Number of equations must be the same as the total number of declared non-input variables

Modelica syntax cont.

```
model DoubleIntegrator
    Real x1(start=0, fixed=true);
    Real x2(start=0, fixed=true);
    input Real u;
equation
    der(x1) = x2;
    der(x2) = u;
end DoubleIntegrator;
```

- `u` is declared as an input
- `x1` and `x2` will be either differentiated or algebraic variables (in this case they are both differentiated)
- Variables have *attributes*. The attributes `start` and `fixed` of `x1` and `x2` have been modified.

Modelica syntax cont.

```
model DoubleIntegrator
    Real x1(start=0, fixed=true);
    Real x2(start=0, fixed=true);
    input Real u;
equation
    der(x1) = x2;
    der(x2) = u;
end DoubleIntegrator;
```

- `start` is the initial value for that variable
- `fixed` indicates that the `start` value is fixed, rather than just a guess for the initial value (default false!)
- Initial value guesses are important when solving DAE initialization problems numerically. As long as you provide initial conditions specifying the state, you do not need to worry about that (if you have a high-index system, you may be in trouble).

Optimization and Modelica

- System modeling: check. But what about optimization?
- Modelica lacks inherent support for optimization formulations
- Language extension Optimica enables optimization
- Johan Åkesson's PhD thesis (2007)
- Not part of official Modelica language, but supported by a few different tools

Optimica syntax

To demonstrate Optimica's syntax, consider (finite horizon LQR)

$$
\begin{aligned}
\text{minimize} \quad & \int_0^{10} \left(x_1^2(t) + x_2^2(t) + u^2(t) \right) \mathrm{d}t, \\
\text{subject to} \quad & \dot{x}_1 = x_2, \\
& \dot{x}_2 = u, \\
& x_1(0) = 1, \quad x_2(0) = 0.
\end{aligned}
$$

```
optimization DILQR(finalTime=10,
                   objectiveIntegrand=x1^2 + x2^2 + u^2)

    extends DoubleIntegrator(x1(start=1));

end DILQR;
```

Optimica syntax cont.

```
optimization DILQR(finalTime=10,
                   objectiveIntegrand=x1^2 + x2^2 + u^2)

    extends DoubleIntegrator(x1(start=1));

end DILQR;
```

- Class `optimization` instead of `model`
- Parameters `finalTime` and `startTime` to specify time horizon
- `objectiveIntegrand` and `objective` specify the running and terminal cost, respectively

Optimica syntax cont.

```
optimization DILQR(finalTime=10,
                   objectiveIntegrand=x1^2 + x2^2 + u^2)

    extends DoubleIntegrator(x1(start=1));

end DILQR;
```

- Can add variables and equations also in `optimization` class
- It is preferable to separate the model and optimization formulation, and then use the Modelica keyword `extends` to inherit the system model.
- Can modify variable attributes when extending

Optimica syntax cont.

Consider instead the minimum time problem

$$
\begin{aligned}
\text{minimize} \quad & t_f, \\
\text{subject to} \quad & \dot{x}_1 = x_2, \quad \dot{x}_2 = u, \\
& -1 \leq u \leq 1, \\
& x_1(0) = 1, \quad x_2(0) = 0, \\
& x_1(t_f) = 0, \quad x_2(t_f) = 0.
\end{aligned}
$$

```
optimization DIMinTime(finalTime(free=true, min=startTime),
                       objective=finalTime)

    extends DoubleIntegrator(x1(start=1), u(min=-1, max=1));

constraint

    x1(finalTime) = 0;
    x2(finalTime) = 0;

end DIMinTime;
```

Optimica syntax cont.

```
optimization DIMinTime(finalTime(free=true, min=startTime),
                       objective=finalTime)
    extends DoubleIntegrator(x1(start=1), u(min=-1, max=1));
constraint
    x1(finalTime) = 0;
    x2(finalTime) = 0;
end DIMinTime;
```

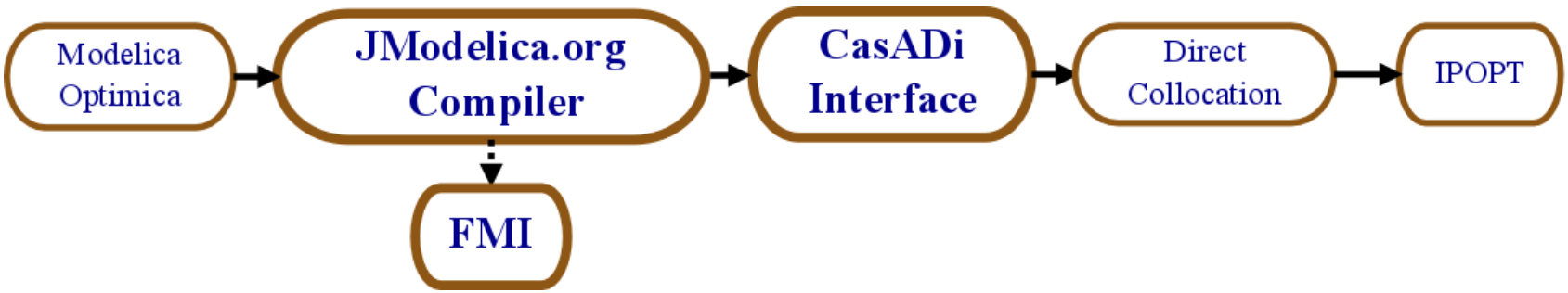
- Objective is now instead to minimize `finalTime`, which has been set to `free`
- Bounds set on u using attribues `min` and `max`
- Can also set bounds on states!

Optimica syntax cont.

```
optimization DIMinTime(finalTime(free=true, min=startTime),
                       objective=finalTime)
    extends DoubleIntegrator(x1(start=1), u(min=-1, max=1));
constraint
    x1(finalTime) = 0;
    x2(finalTime) = 0;
end DIMinTime;
```

- Additional `constraint` section
- Constraints can be *point constraints*, i.e. enforced at time instants such as t_f
- Constraints can also be *path constraints*, i.e. enforced during entire time horizon
- Path constraints generalize `min` and `max`. Use `min` and `max` when possible!

The complete JModelica.org toolchain

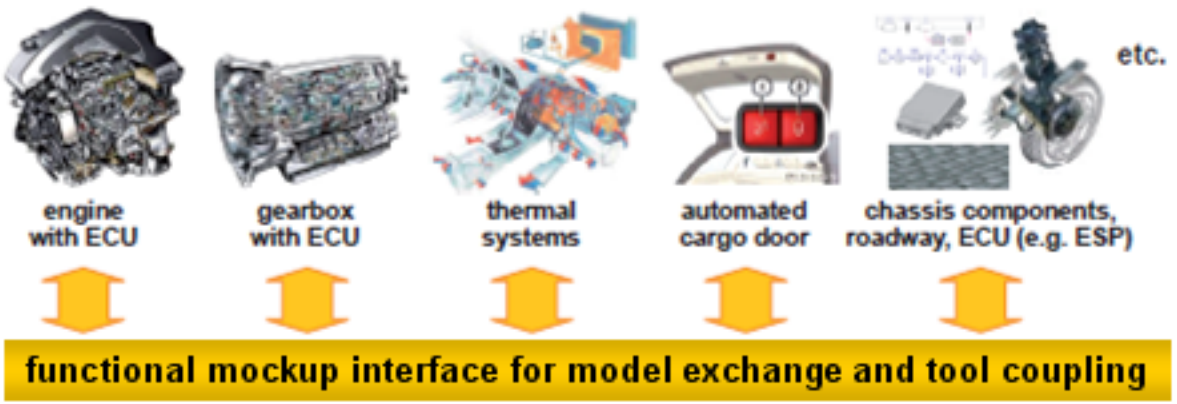


JModelica.org compiler

- Modelica and Optimica code is not executable
- The JModelica.org compiler creates an Abstract Syntax Tree (AST) representation of the Modelica code in Java, hence **J**Modelica.org
- Symbolic transformations, e.g. index reduction, are performed on the AST
- The transformed AST is then used to generate code that can be used for computations

Functional Mock-Up Interface

- For simulation purposes, most Modelica tools (including JModelica.org) generate C code
- C code is generated according to the Functional Mock-Up Interface (FMI) standard
- FMI defines a C code standard for simulating ODE systems, by creating Functional Mock-Up Units (FMU)
- Emphasis is tool interoperability; e.g. possible to generate an FMU in JModelica.org and use the FMU as a block in Simulink



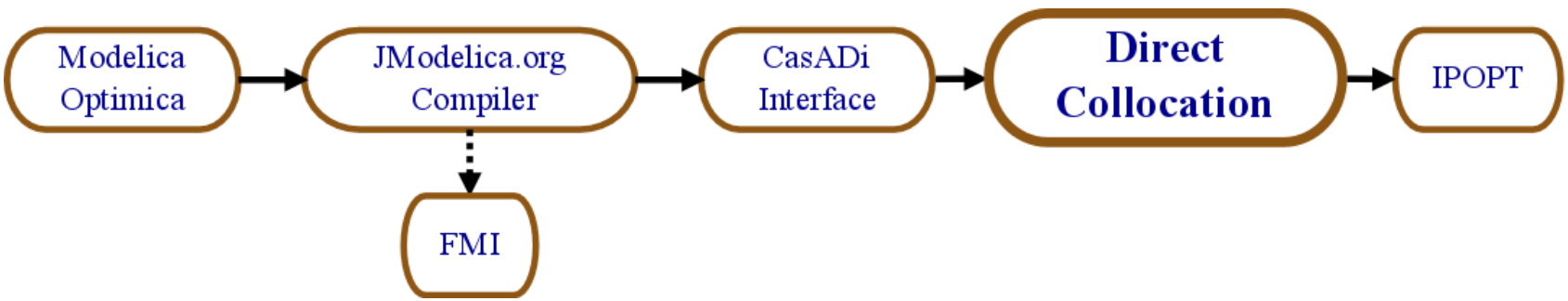
CasADi Interface

- For optimization purposes, the AST is instead used to generate CasADi code
- CasADi (**C**omputer **a**lgebra **s**ystem with **A**lgorithmic **Di**fferentiation) is a tool for computing derivatives using algorithmic differentation (AD)
- AD combines the accuracy of symbolic differentation and speed of numerical differentiation
- CasADi Interface provides an interface to a symbolic representation of OCPs
- CasADi then provides the derivatives needed to solve the optimal control problem numerically using gradient-based methods

Python

- All the components of JModelica.org are invoked from Python
- Python is a free, high-level, dynamically typed programming language
- Mature libraries for scientific computations inspired by MATLAB: NumPy, SciPy, matplotlib

The complete JModelica.org toolchain



Numerical methods for optimal control

- Now we have a representation of optimal control problems suitable for numerical methods. How to proceed?
- The problem is infinite-dimensional. Need to discretize.
- Two choices to make: when to discretize, and how to discretize

Numerical methods for optimal control cont.

When:

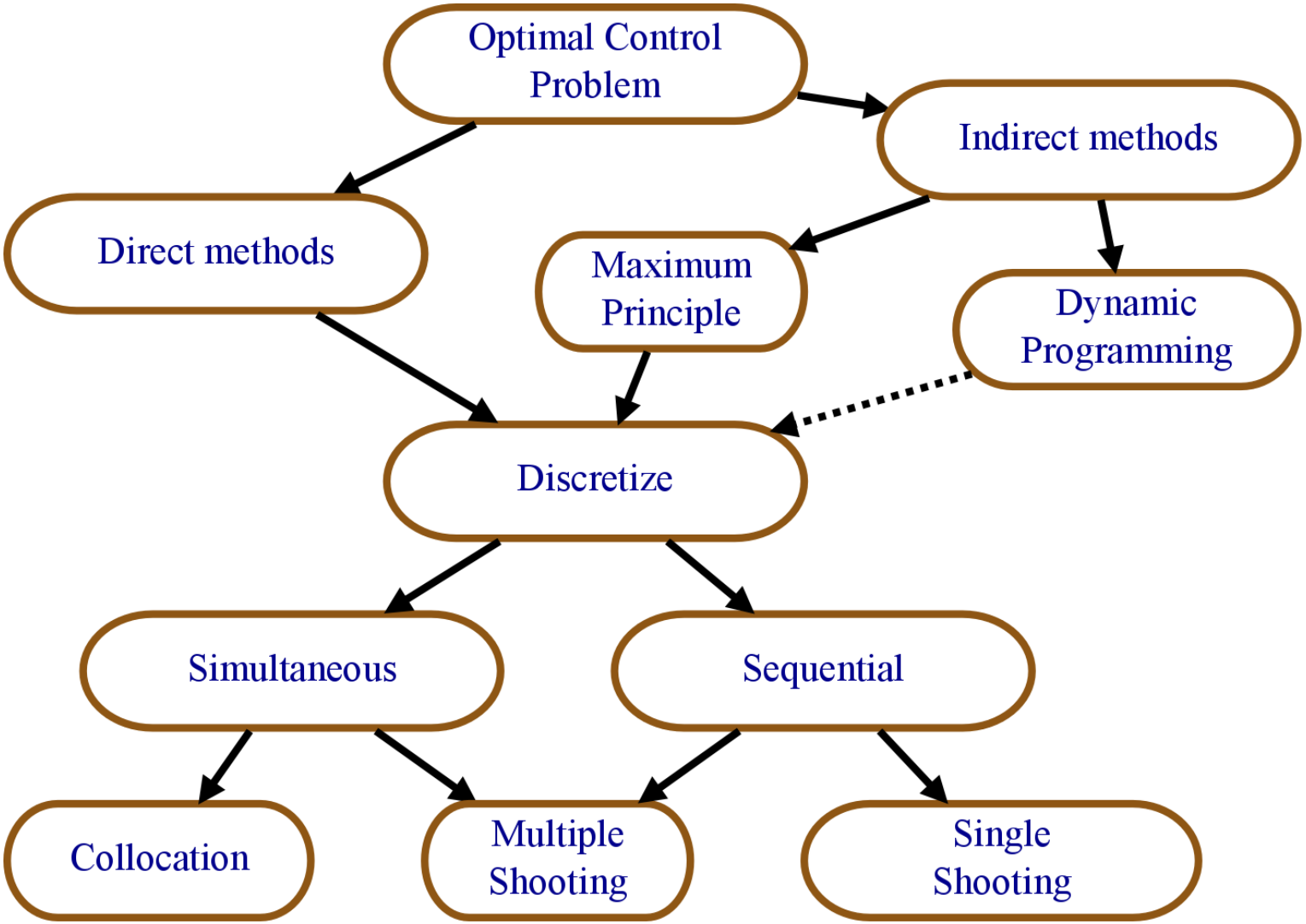
- Indirect methods (optimize, then discretize): establish optimality conditions using maximum principle or dynamic programming and solve the differential equations numerically
- Direct methods (discretize, then optimize): discretize to a mathematical program, and then solve the Karush-Kuhn-Tucker (KKT) conditions numerically

Numerical methods for optimal control cont.

How:

- Sequential methods: discretize control, simulate dynamic system using embedded numerical integrator, and update control iteratively based on sensitivities
- Simultaneous methods: discretize all system variables and solve the resulting equation system

Numerical methods for optimal control cont.



Numerical methods for optimal control cont.

We essentially end up with 6 different methods (of course, more exist):

- Indirect single shooting
- Indirect multiple shooting
- Indirect collocation (Bo)
- Direct single shooting
- Direct multiple shooting
- Direct collocation (JModelica.org)

Method properties

- $\pm$ Indirect methods rely on calculus of variations
- $-$ Indirect methods very sensitive to initial guesses of costates and switching structure of inequalities
- $-$ Single shooting can be slow and can not handle open-loop unstable systems
- $+$ Direct multiple shooting and direct collocation are the most powerful

Direct multiple shooting vs. collocation

- Multiple shooting is more memory efficient
- Multiple shooting has adaptive discretization (from embedded numerical integrators)
- Collocation is faster
- Not clear which is more robust

A closer look at direct collocation

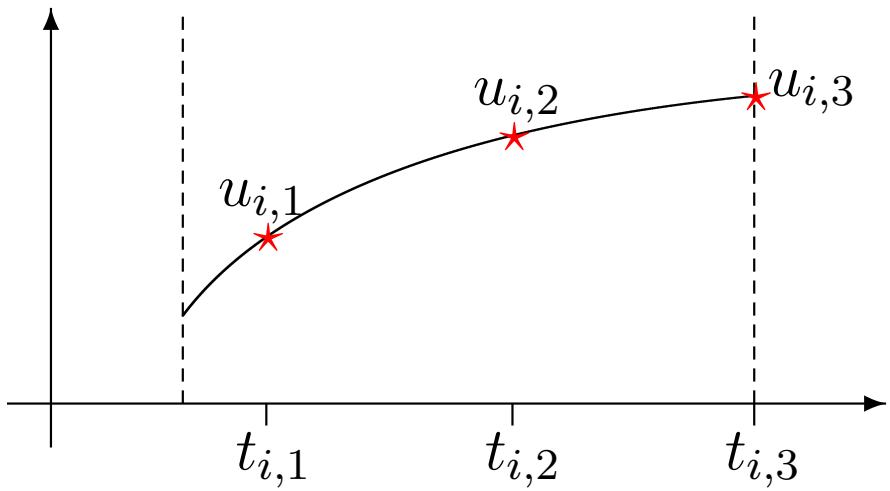
Main idea is to approximate system trajectories by polynomials:

- Divide the time horizon $[t_0, t_f]$ into n_e elements
- Approximate system variables in each element by a *collocation polynomial*
- Force this polynomial to satisfy all the constraints in n_c *collocation points*
- This uniquely determines a polynomial of degree $n_c - 1$ by interpolation
- The discretization is not adaptive. YOU have to choose n_e and n_c!

Collocation polynomials

Let $t_{i,k}$ denote collocation point number k in element i and let

$$u_{i,k} := u(t_{i,k}).$$



Polynomials for x and y are constructed in a similar manner.

The choice of collocation points $t_{i,k}$ define the collocation method.

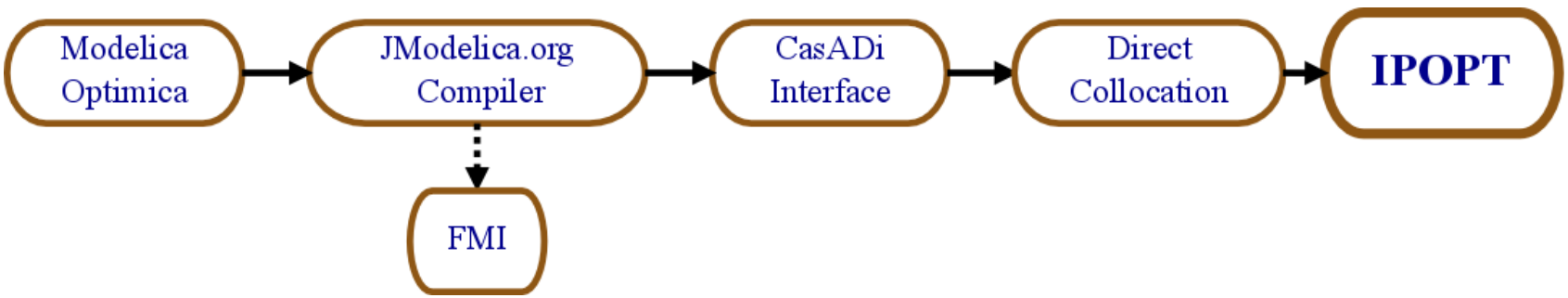
Nonlinear program

- Applying a direct collocation discretization procedure results in a finite-dimensional nonlinear program (NLP) of the form

$$
\begin{aligned}
&\text{minimize} && f(x), \\
&\text{with respect to} && x \in \mathbb{R}^n, \\
&\text{subject to} && x_L \leq x \leq x_U, \\
& && g(x) = 0, \\
& && h(x) \leq 0.
\end{aligned}
$$

- Solving the NLP gives an approximate solution to OCP
- Dual variables of NLP correspond to costate of OCP

The complete JModelica.org toolchain



IPOPT

- IPOPT (**I**nterior **P**oint **OPT**imizer) iteratively solves NLPs using a primal-dual interior-point method with filter-based line search
- Nonlinear dynamics $\implies$ nonlinear equality constraints $\implies$ nonconvexity
- Any local minimum is considered a solution

NLP initialization

- A decent initial guess of the solution is important to find a decent local optimum
- For large nonlinear systems, a decent initial guess is important to find anything at all
- For NLPs arising from direct collocation, the decision variables include all of the system variables (input, differentiated and algebraic variables) **at all collocation points**
- An initial guess is decent if it is close to a decent local optimum. Does not have to be feasible!

NLP initialization cont.

- Constant (w.r.t. time) initial guesses can be provided with the Optimica variable attribute `initialGuess`
- For non-trivial problems, constant initial guesses are usually insufficient
- Usually better to guess an optimal input, and then simulate the system using the guessed input
- Alternatively, solve a simpler but related optimization problem and use that as guess

Regularity conditions

- In each iteration, IPOPT basically solves the linearized KKT conditions (KKT system)
- The KKT conditions involve the gradient of the objective and the Jacobian of the constraints (first-order derivatives)
- The linearization thus involves second-order derivatives of the objective and the constraints (Hessian of the augmented Lagrangian)
- F, L, and K thus need to be C^2! (Although not w.r.t. time)
- CasADi is used to obtain these derivatives

Solving the KKT system

- Solving the KKT system in each iteration is the most time consuming step of the entire toolchain
- Important that this linear system of equations can be solved efficiently
- This is a challenge - recall my last Friday seminar!
- Another reason why a decent initial guess is important, since it affects the KKT conditioning

Choice of linear solver

- IPOPT supports many different linear solvers, but only one (MUMPS) is freely available
- MUMPS is often insufficient, but personal academic licenses available for better ones
- Unfortunately, using these means that you have to install JModelica.org yourself

Complete double integrator example

To demonstrate, simulate the solution to problem 2 on home assignment 3, i.e.

$$
\begin{aligned}
\text{minimize} \quad & t_f, \\
\text{subject to} \quad & \dot{x}_1 = x_2, \quad \dot{x}_2 = u, \\
& -1 \leq u \leq 1, \\
& x_1(0) = 0, \quad x_2(0)) = 0, \\
& x_1(t_f) = 2, \quad x_2(t_f)) = 2.
\end{aligned}
$$

Solution is $u^* \equiv 1$, $t_f = 2$.

```
model DoubleIntegrator
    Real x1(start=0, fixed=true);
    Real x2(start=0, fixed=true);
    input Real u;
equation
    der(x1) = x2;
    der(x2) = u;
end DoubleIntegrator;
```

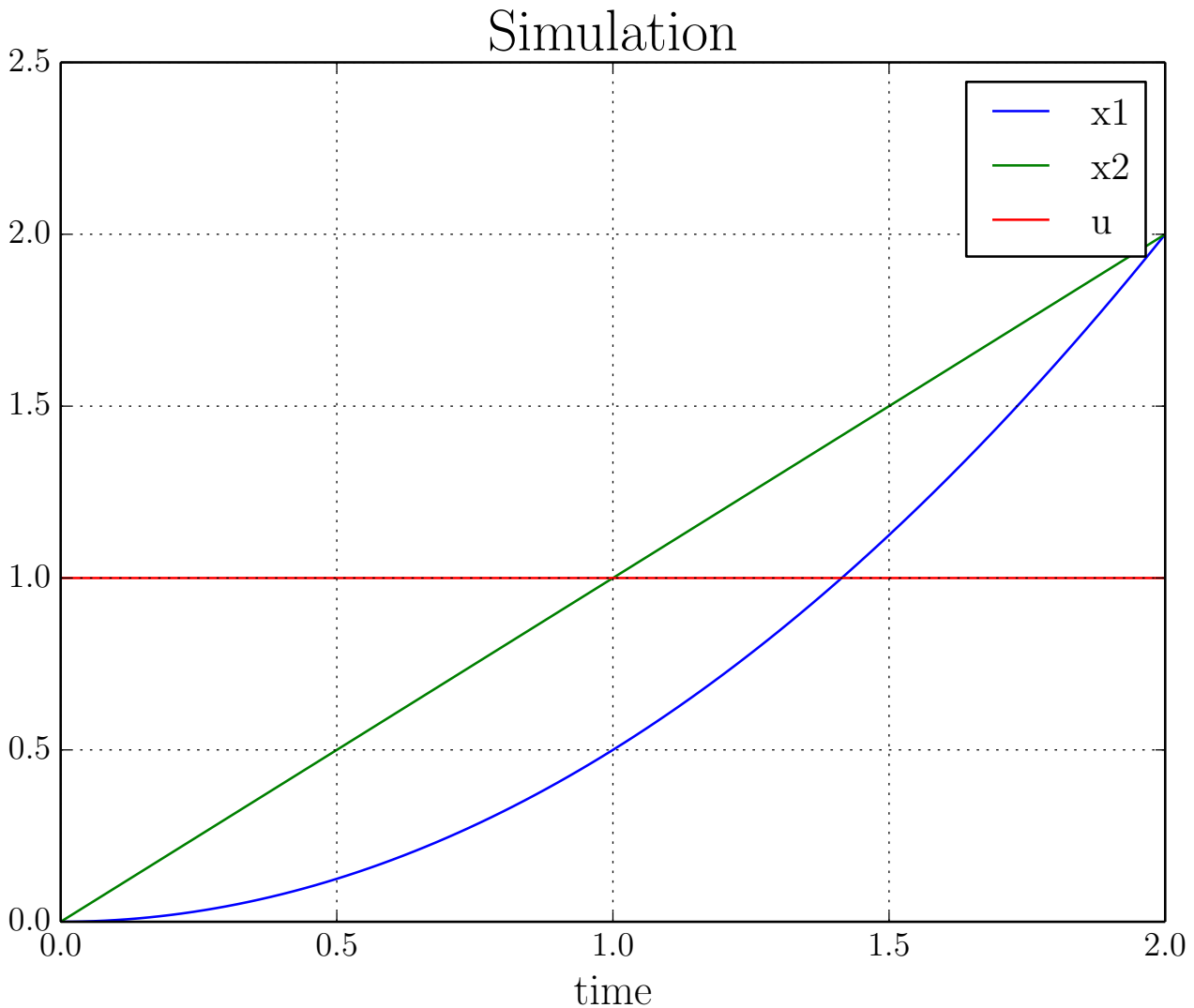
Complete double integrator example cont.

```
# Import JModelica.org modules
from pymodelica import compile_fmu
from pyfmi import load_fmu
from pyjmi import transfer_optimization_problem

# Import useful Python modules
import matplotlib.pyplot as plt
import numpy as np

# Compile simulation model
model = load_fmu(compile_fmu(
    'DoubleIntegrator', 'double_integrator.mop'))

# Create system input
t0 = 0.
tf = 2.
u_values = np.array([[t0, 1.], [tf, 1.]]) # Constantly 1
```

Complete double integrator example cont.

```
# Simulate with created input
sim_res = model.simulate(start_time=t0, final_time=tf,
                         input=('u', u_values),
                         options={'ncp': 100})

# Obtain simulation result
sim_time = sim_res['time']
sim_x1 = sim_res['x1']
sim_x2 = sim_res['x2']
sim_u = sim_res['u']

# Plot simulation result
plt.close(1); plt.figure(1)
plt.plot(sim_time, sim_x1)
plt.plot(sim_time, sim_x2)
plt.plot(sim_time, sim_u)
plt.grid(True)
plt.xlabel('time')
plt.legend(['x1', 'x2', 'u'])
plt.title('Simulation')
plt.show()
```

Double integrator simulation



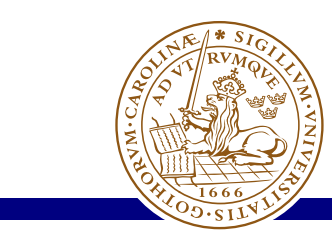
Double integrator LQR

Consider the optimization problem from before:

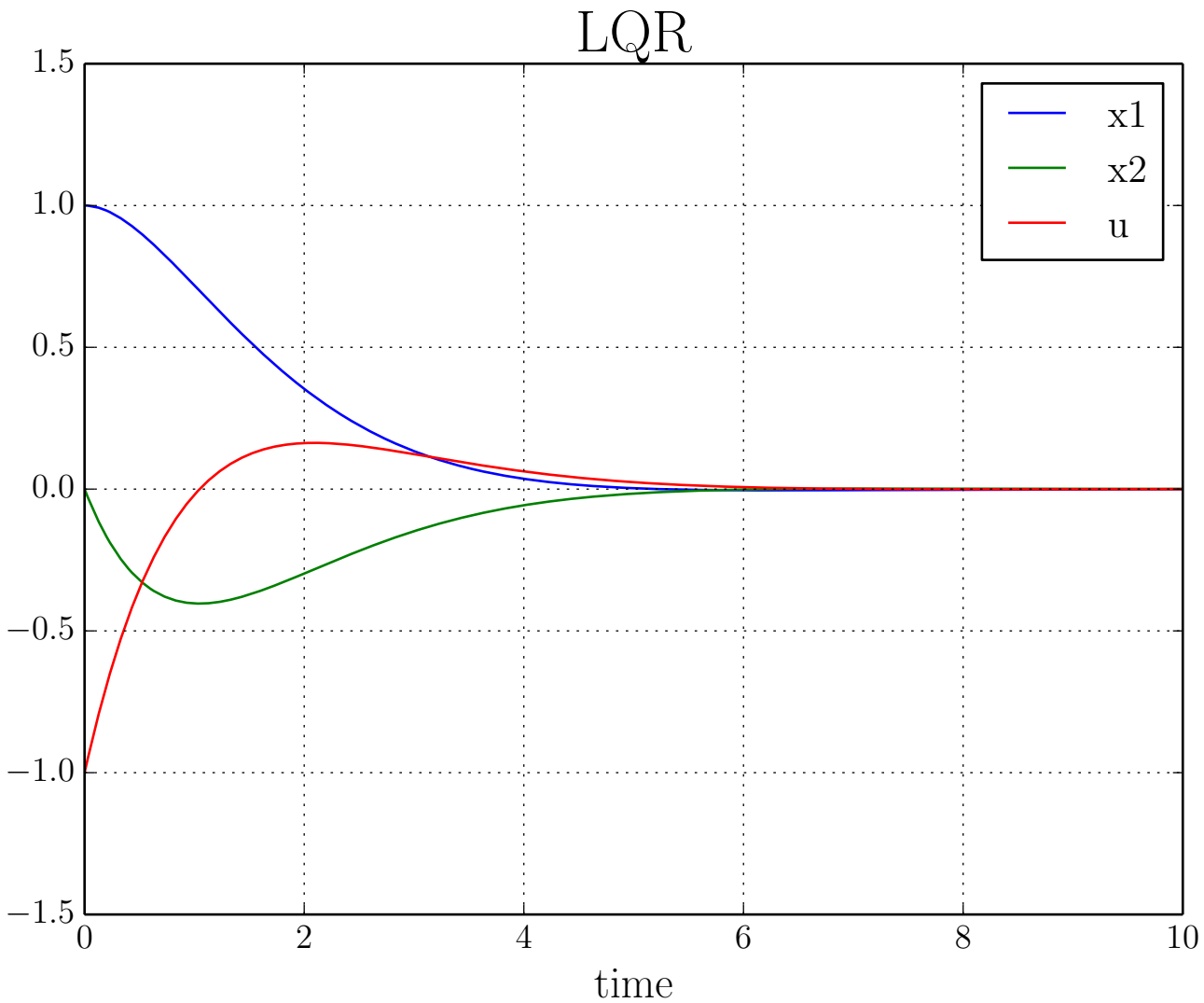
```
optimization DILQR(finalTime=10,
                   objectiveIntegrand=x1^2 + x2^2 + u^2)

    extends DoubleIntegrator(x1(start=1));

end DILQR;
```

Double integrator LQR cont.

```
# Compile LQR problem
op_lqr = transfer_optimization_problem(
    'DILQR', 'double_integrator.mop')

# Solve optimization problem
lqr_res = op_lqr.optimize()

# Obtain optimization result
lqr_time = lqr_res['time']
lqr_x1 = lqr_res['x1']
lqr_x2 = lqr_res['x2']
lqr_u = lqr_res['u']

# Plot LQR result
```



Double integrator LQR cont.



Double integrator minimal time

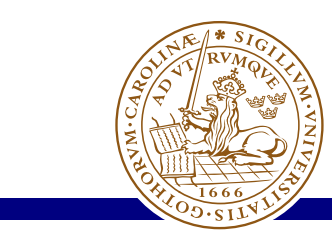
Consider the minimal time problem from before:

```
optimization DIMinTime(finalTime(free=true, min=startTime),
                       objective=finalTime)

    extends DoubleIntegrator(x1(start=1), u(min=-1, max=1));

constraint

    x1(finalTime) = 0;
    x2(finalTime) = 0;

end DIMinTime;
```

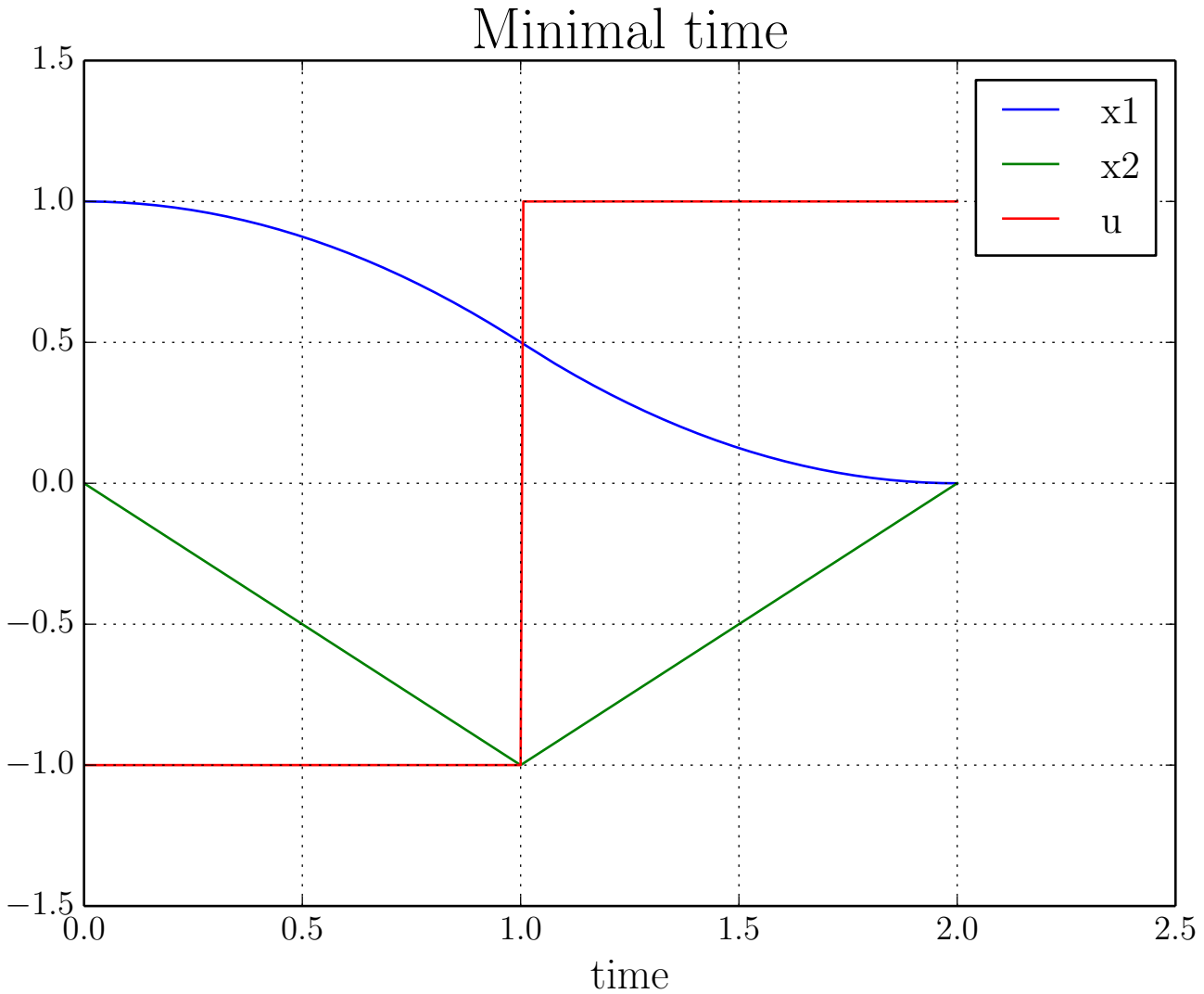
This time, we will use the solution to the LQR problem as an initial guess.

Double integrator minimal time cont.

```
# Compile minimal time problem
mintime_op = transfer_optimization_problem(
    'DIMinTime', 'double_integrator.mop')

# Get optimization options
opts = mintime_op.optimize_options()

# Set LQR solution as initial guess
opts['init_traj'] = lqr_res.result_data
opts['nominal_traj'] = lqr_res.result_data

# Solve optimization problem
mintime_res = mintime_op.optimize(options=opts)

# Obtain optimization result
mintime_time = mintime_res['time']
mintime_x1 = mintime_res['x1']
mintime_x2 = mintime_res['x2']
mintime_u = mintime_res['u']

# Plot time minimization result
```



Double integrator minimal time cont.



Resources

- NumPy for Matlab users:
  `http://wiki.scipy.org/NumPy_for_Matlab_Users`
- JModelica.org User's Guide:
  `http://www.jmodelica.org/page/236`