3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools
Oslo, 3 October 2010

Towards a Computer Algebra System with Automatic Differentiation

for use with object-oriented modelling languages

Joel Andersson

and

Moritz Diehl   Boris Houska

Department of Electrical Engineering (ESAT-SCD) &
Optimization in Engineering Center (OPTEC)
Katholieke Universiteit Leuven

OPTEC – Optimization in Engineering

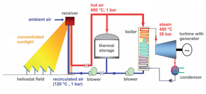
- Interdiciplinary: Mech.Eng. + Elec.Eng. + Civ.Eng. + Comp.Sc.
- Katholieke Universiteit Leuven, Belgium
- 2005-2010, phase II 2010-2017

OPTEC – Optimization in Engineering

- Interdiciplinary: Mech.Eng. + Elec.Eng. + Civ.Eng. + Comp.Sc.
- Katholieke Universiteit Leuven, Belgium
- 2005-2010, phase II 2010-2017

Myself

- M.Sc. Engineering Physics/Mathematics from Chalmers, Gothenburg
- PhD student since Oct 2008 for Prof. Moritz Diehl
- Topic: *Modelling and Derivative Generation for Dynamic Optimization and Application to Large Scale Interconnected DAE Systems*
- Application: Solar thermal power plant

Dynamic optimization

Dynamic optimization problem

We consider dynamic optimization problems of the form (can be generalized further):

$$\begin{array}{ll} \underset{x(\cdot),\, z(\cdot),\, u(\cdot),\, p}{\textbf{minimize:}} & \displaystyle\int_{t=0}^{T} L(x, u, z, p, t)\, dt + E(x(T)) \\ \textbf{subject to:} & \begin{array}{rll} f(\dot{x}(t), x(t), z(t), u(t), p, t) & = 0 & t \in [0, T] \\ h(x(t), z(t), u(t), p, t) & \leq 0 & t \in [0, T] \\ r(x(0), x(T), p) & = 0 & \end{array} \end{array}$$

$x : \mathbb{R}_+ \to \mathbb{R}^{N_x}$ differential states, $z : \mathbb{R}_+ \to \mathbb{R}^{N_z}$ algebraic states,
$u : \mathbb{R}_+ \to \mathbb{R}^{N_u}$ control, $p \in \mathbb{R}^{N_p}$ free parameters

Dynamic optimization problem

We consider dynamic optimization problems of the form (can be generalized further):

$$\begin{array}{rlll} \underset{x(\cdot),\, z(\cdot),\, u(\cdot),\, p}{\textbf{minimize:}} & \displaystyle\int_{t=0}^{T} L(x,\, u,\, z,\, p,\, t)\, dt + E(x(T)) & & \\ \textbf{subject to:} & f(\dot{x}(t),\, x(t),\, z(t),\, u(t),\, p,\, t) & = 0 & t \in [0,\, T] \\ & h(x(t),\, z(t),\, u(t),\, p,\, t) & \leq 0 & t \in [0,\, T] \\ & r(x(0),\, x(T),\, p) & = 0 & \end{array}$$

$x : \mathbb{R}_+ \to \mathbb{R}^{N_x}$ differential states, $z : \mathbb{R}_+ \to \mathbb{R}^{N_z}$ algebraic states,
$u : \mathbb{R}_+ \to \mathbb{R}^{N_u}$ control, $p \in \mathbb{R}^{N_p}$ free parameters

Solution

- Dynamic programming / Hamilton-Jacobi-Bellman equation – for very small problems

Dynamic optimization problem

We consider dynamic optimization problems of the form (can be generalized further):

$$
\begin{array}{rlll}
\underset{x(\cdot),\, z(\cdot),\, u(\cdot),\, p}{\textbf{minimize:}} & \displaystyle\int_{t=0}^{T} L(x, u, z, p, t)\, dt + E(x(T)) & & \\
\textbf{subject to:} & f(\dot{x}(t), x(t), z(t), u(t), p, t) & = 0 & t \in [0, T] \\
& h(x(t), z(t), u(t), p, t) & \leq 0 & t \in [0, T] \\
& r(x(0), x(T), p) & = 0 &
\end{array}
$$

$x : \mathbb{R}_+ \to \mathbb{R}^{N_x}$ differential states, $z : \mathbb{R}_+ \to \mathbb{R}^{N_z}$ algebraic states,
$u : \mathbb{R}_+ \to \mathbb{R}^{N_u}$ control, $p \in \mathbb{R}^{N_p}$ free parameters

Solution

- Dynamic programming / Hamilton-Jacobi-Bellman equation – for very small problems
- Pontryagin's Maximum Principle – for problems without inequality constraints

Dynamic optimization problem

We consider dynamic optimization problems of the form (can be generalized further):

$$\begin{array}{ll} \underset{x(\cdot),\, z(\cdot),\, u(\cdot),\, p}{\textbf{minimize:}} & \displaystyle\int_{t=0}^{T} L(x, u, z, p, t)\, dt + E(x(T)) \\ \textbf{subject to:} & \begin{array}{rll} f(\dot{x}(t), x(t), z(t), u(t), p, t) & = 0 & t \in [0, T] \\ h(x(t), z(t), u(t), p, t) & \leq 0 & t \in [0, T] \\ r(x(0), x(T), p) & = 0 & \end{array} \end{array}$$

$x : \mathbb{R}_+ \to \mathbb{R}^{N_x}$ differential states, $z : \mathbb{R}_+ \to \mathbb{R}^{N_z}$ algebraic states,
$u : \mathbb{R}_+ \to \mathbb{R}^{N_u}$ control, $p \in \mathbb{R}^{N_p}$ free parameters

Solution

- Dynamic programming / Hamilton-Jacobi-Bellman equation – for very small problems
- Pontryagin's Maximum Principle – for problems without inequality constraints
- **Direct methods**: Parametrize controls and possibly state to form a Nonlinear Program (NLP)

Dynamic optimization problem

We consider dynamic optimization problems of the form (can be generalized further):

$$\begin{array}{ll} \underset{x(\cdot),\, z(\cdot),\, u(\cdot),\, p}{\textbf{minimize:}} & \displaystyle\int_{t=0}^{T} L(x, u, z, p, t)\, dt + E(x(T)) \\ \textbf{subject to:} & \begin{array}{rll} f(\dot{x}(t), x(t), z(t), u(t), p, t) & = 0 & t \in [0, T] \\ h(x(t), z(t), u(t), p, t) & \leq 0 & t \in [0, T] \\ r(x(0), x(T), p) & = 0 & \end{array} \end{array}$$

$x : \mathbb{R}_+ \rightarrow \mathbb{R}^{N_x}$ differential states, $z : \mathbb{R}_+ \rightarrow \mathbb{R}^{N_z}$ algebraic states,
$u : \mathbb{R}_+ \rightarrow \mathbb{R}^{N_u}$ control, $p \in \mathbb{R}^{N_p}$ free parameters

Solution

- Dynamic programming / Hamilton-Jacobi-Bellman equation – for very small problems
- Pontryagin's Maximum Principle – for problems without inequality constraints
- **Direct methods**: Parametrize controls and possibly state to form a Nonlinear Program (NLP)
  - Collocation: Parametrize state to form a large, but sparse NLP

Dynamic optimization problem

We consider dynamic optimization problems of the form (can be generalized further):

$$\begin{array}{ll} \underset{x(\cdot),\, z(\cdot),\, u(\cdot),\, p}{\textbf{minimize:}} & \displaystyle\int_{t=0}^{T} L(x, u, z, p, t)\, dt + E(x(T)) \\ \textbf{subject to:} & \begin{array}{rll} f(\dot{x}(t), x(t), z(t), u(t), p, t) & = 0 & t \in [0, T] \\ h(x(t), z(t), u(t), p, t) & \leq 0 & t \in [0, T] \\ r(x(0), x(T), p) & = 0 & \end{array} \end{array}$$

$x : \mathbb{R}_+ \to \mathbb{R}^{N_x}$ differential states, $z : \mathbb{R}_+ \to \mathbb{R}^{N_z}$ algebraic states,
$u : \mathbb{R}_+ \to \mathbb{R}^{N_u}$ control, $p \in \mathbb{R}^{N_p}$ free parameters

Solution

- Dynamic programming / Hamilton-Jacobi-Bellman equation – for very small problems
- Pontryagin's Maximum Principle – for problems without inequality constraints
- **Direct methods**: Parametrize controls and possibly state to form a Nonlinear Program (NLP)
  - Collocation: Parametrize state to form a large, but sparse NLP
  - Single-shooting: Eliminate the state with an DAE integrator to form a small, but nonlinear NLP

Dynamic optimization problem

We consider dynamic optimization problems of the form (can be generalized further):

$$\begin{array}{ll} \underset{x(\cdot),\, z(\cdot),\, u(\cdot),\, p}{\textbf{minimize:}} & \displaystyle\int_{t=0}^{T} L(x, u, z, p, t)\, dt + E(x(T)) \\ \textbf{subject to:} & \begin{array}{rll} f(\dot{x}(t), x(t), z(t), u(t), p, t) & = 0 & t \in [0, T] \\ h(x(t), z(t), u(t), p, t) & \leq 0 & t \in [0, T] \\ r(x(0), x(T), p) & = 0 & \end{array} \end{array}$$

$x : \mathbb{R}_+ \rightarrow \mathbb{R}^{N_x}$ differential states, $z : \mathbb{R}_+ \rightarrow \mathbb{R}^{N_z}$ algebraic states,
$u : \mathbb{R}_+ \rightarrow \mathbb{R}^{N_u}$ control, $p \in \mathbb{R}^{N_p}$ free parameters

Solution

- Dynamic programming / Hamilton-Jacobi-Bellman equation – for very small problems
- Pontryagin's Maximum Principle – for problems without inequality constraints
- **Direct methods**: Parametrize controls and possibly state to form a Nonlinear Program (NLP)
  - Collocation: Parametrize state to form a large, but sparse NLP
  - Single-shooting: Eliminate the state with an DAE integrator to form a small, but nonlinear NLP
  - **Multiple-shooting**: Parametrize state at some times and use single shooting in between

Good reference: L. Biegler *Nonlinear Programming*, SIAM 2010

Dynamic optimization

Direct Multiple Shooting (Bock, 1984)

- Subdivide time horizon: $0 = t_0 \leq \ldots \leq T_N$
- Parametrize control: $u(t) = u_i, \quad t \in [t_i, t_{i+1}]$
- Parametrize state: $s_{x,i} = x(t_i)$

Dynamic optimization

Direct Multiple Shooting (Bock, 1984)

- Subdivide time horizon: $0 = t_0 \leq \ldots \leq T_N$
- Parametrize control: $u(t) = u_i, \quad t \in [t_i, t_{i+1}]$
- Parametrize state: $s_{x,i} = x(t_i)$
- Nonlinear Program (NLP):

$$\underset{s_{x,i},\, u_i,\, p}{\textbf{minimize:}} \quad \sum_{i=0}^{N-1} L_i(s_{x,i}, u_i, p) + E(s_{x,N})$$

subject to:

$$\begin{aligned} s_{x,i+1} &= F_i(s_{x,i}, u_i, p), \quad \forall i \\ 0 &\geq h_i(s_{x,i}, u_i, p), \quad \forall i \\ 0 &= r(s_{x,0}, s_{x,N}, p) \end{aligned}$$

- F_i: Call to an DAE integrator

Dynamic optimization

Direct Multiple Shooting (Bock, 1984)

- Subdivide time horizon: $0 = t_0 \leq \ldots \leq T_N$
- Parametrize control: $u(t) = u_i, \quad t \in [t_i, t_{i+1}]$
- Parametrize state: $s_{x,i} = x(t_i)$
- Nonlinear Program (NLP):

$$\underset{s_{x,i},\, u_i,\, p}{\textbf{minimize:}} \quad \sum_{i=0}^{N-1} L_i(s_{x,i}, u_i, p) + E(s_{x,N})$$

subject to:

$$\begin{aligned} s_{x,i+1} &= F_i(s_{x,i}, u_i, p), \quad \forall i \\ 0 &\geq h_i(s_{x,i}, u_i, p), \quad \forall i \\ 0 &= r(s_{x,0}, s_{x,N}, p) \end{aligned}$$

- F_i: Call to an DAE integrator
- Solve with e.g. structure-exploiting SQP method
- Software: ACADO Toolkit, MUSCOD-II

Dynamic optimization

Direct Multiple Shooting (Bock, 1984)

- Subdivide time horizon: $0 = t_0 \leq \ldots \leq T_N$
- Parametrize control: $u(t) = u_i, \quad t \in [t_i, t_{i+1}]$
- Parametrize state: $s_{x,i} = x(t_i)$
- Nonlinear Program (NLP):

$$\underset{s_{x,i},\, u_i,\, p}{\textbf{minimize:}} \quad \sum_{i=0}^{N-1} L_i(s_{x,i}, u_i, p) + E(s_{x,N})$$

subject to:

$$\begin{aligned} s_{x,i+1} &= F_i(s_{x,i}, u_i, p), \quad \forall i \\ 0 &\geq h_i(s_{x,i}, u_i, p), \quad \forall i \\ 0 &= r(s_{x,0}, s_{x,N}, p) \end{aligned}$$

- F_i: Call to an DAE integrator
- Solve with e.g. structure-exploiting SQP method
- Software: ACADO Toolkit, MUSCOD-II

Notes

- The problem formulation can be generalized (e.g. free end time, multiple model stages, hybrid)
- Large scale NLP solvers require derivative information, at least first order

Automatic differentiation

Automatic differentiation[a], AD, is able to cheaply and accurately providing derivative evaluation of a function $f = f(x)$ by applying the **chain rule to the algorithm**. Two "modes":

- Forward mode: $\frac{\partial f}{\partial x}\, r$
- Reverse (adjoint) mode: $r^{\mathsf{T}}\, \frac{\partial f}{\partial x}$

[a] See e.g. Griewank & Walther: Evaluating Derivatives, 2008

Automatic differentiation

Automatic differentiation

Automatic differentiation[a], AD, is able to cheaply and accurately providing derivative evaluation of a function $f = f(x)$ by applying the **chain rule to the algorithm**. Two "modes":

- Forward mode: $\frac{\partial f}{\partial x}\, r$
- Reverse (adjoint) mode: $r^{\mathrm{T}}\, \frac{\partial f}{\partial x}$

[a] See e.g. Griewank & Walther: Evaluating Derivatives, 2008

Implementations

- Operator overloading (OO)
  - Idea: Use operator overloading in e.g. C++, to record calculations
  - Easy to implement, expecially in forward mode, needs only a C++- compiler
  - Disadvantage: Effective implementation is based on template meta programming
  - Tools: ADOL-C, CppAD, ...
- Source code transformation (SCT)
  - Implements AD inside a compiler (think GCC)
  - Advantage: Efficient code
  - Disadvantage: Hard to implement, less mature than OO
  - Tool: OpenAD

Can be applied to "black-box" C or Fortran functions!

Motivation

Observations

- To apply an existing AD tool, we first need to generate C-code
- The AD tool will *parses* the code to obtain the graph we already had!
- More sensible approach: Apply AD to the graph directly

- Benifits:
  - No compiler in the loop
  - No information losses (e.g. for systems with switches)
  - Convex reformulation (cf. CVX)
  - Structure exploatation

Motivation

Observations

- To apply an existing AD tool, we first need to generate C-code
- The AD tool will *parses* the code to obtain the graph we already had!
- More sensible approach: Apply AD to the graph directly
- Benifits:
  - No compiler in the loop
  - No information losses (e.g. for systems with switches)
  - Convex reformulation (cf. CVX)
  - Structure exploatation

Structure exploatation: The Lifted Newton method

- Albersmeyer & Diehl, SIAM 2010
- "Lifts" non-linear root-finding problems to a higher dimension
- Speeds up convergence, increases region of attraction
- Requires that the functions are given as an *algorithm*
- Example: A single-shooting algorithm can be lifted to a multiple-shooting algorithm with condensing

CasADi

What is CasADi?

- A minimalistic Computer Algebra System (CAS) written in self-contained C++

CasADi

What is CasADi?

- A minimalistic Computer Algebra System (CAS) written in self-contained C++
- Implements AD by a hybrid OO/SCT approach

CasADi

What is CasADi?

- A minimalistic Computer Algebra System (CAS) written in self-contained C++
- Implements AD by a hybrid OO/SCT approach
- **Enables step-by-step symbolic reformulation of a dynamic optimization problem into an equivalent NLP**

CasADi

What is CasADi?

- A minimalistic Computer Algebra System (CAS) written in self-contained C++
- Implements AD by a hybrid OO/SCT approach
- **Enables step-by-step symbolic reformulation of a dynamic optimization problem into an equivalent NLP**
- Tailored for high speed, especially during numerical evaluation

CasADi

What is CasADi?

- A minimalistic Computer Algebra System (CAS) written in self-contained C++
- Implements AD by a hybrid OO/SCT approach
- **Enables step-by-step symbolic reformulation of a dynamic optimization problem into an equivalent NLP**
- Tailored for high speed, especially during numerical evaluation
- Contains interfaces to Sundials (IDAS,CVodes), IPOPT, ACADO Toolkit, JModelica

CasADi

What is CasADi?

- A minimalistic Computer Algebra System (CAS) written in self-contained C++
- Implements AD by a hybrid OO/SCT approach
- **Enables step-by-step symbolic reformulation of a dynamic optimization problem into an equivalent NLP**
- Tailored for high speed, especially during numerical evaluation
- Contains interfaces to Sundials (IDAS,CVodes), IPOPT, ACADO Toolkit, JModelica
- Open-source project: first public release autumn 2010 (`www.casadi.org`)
- Permissive licence, LGPL

Structure of CasADi

Two graph representations used in conjunction

	Scalar expression, SX "maximum speed"	**Matrix expression, MX** "maximum generality"
number of arguments	two	arbitrary
nodes	scalar-valued	vector-valued
operations	built-in	all (e.g. calls to **FX**)
branching/jumps	no	yes
parallelization	no	yes
syntax	`double`	`double, ublas::matrix`

Dynamically created functions

Function expression, FX

- $[y_1, \ldots, y_n] = f(x_1, \ldots, x_m)$, $y_i \in \mathbb{R}^{r_i}$, $x_j \in \mathbb{R}^{q_j}$
- Polymorphic class, derived classes:
  - Function given by a graph of SX nodes
  - Function given by a graph of MX nodes
  - External function (e.g. from DLL)
  - Integrator and "Simulator"

Observation

- SX $\sim$ DAG in AD-tools
- MX $\sim$ DAG in Modelica

The `Integrator` class

- An *integrator* is considered to be a function: $x(t_\mathrm{f}) = F(t_0, t_\mathrm{f}, x(t_0), p)$
- Currently one explicit integrator (CVodes) and one implicit integrator (IDAS)
- AD forward/reverse corresponds to forward/adjoint *sensitivities*

Structure of CasADi

The `Integrator` class

- An *integrator* is considered to be a function: $x(t_f) = F(t_0, t_f, x(t_0), p)$
- Currently one explicit integrator (CVodes) and one implicit integrator (IDAS)
- AD forward/reverse corresponds to forward/adjoint *sensitivities*

The `Simulator` class

- Evaluates an *output function*, $h(t, x, p)$ in a *set of time points*, $[t_1, \ldots, t_n]$, using an arbitrary *integrator*
- Also considered a function: $[y_1, \ldots, y_n, x(t_f)] = G([t_0, t_f, x(t_0)), p, [t_1, \ldots, t_n])$ with $y_i := h(t_i, x(t_i), p)$

Structure of CasADi

The `Integrator` class

- An *integrator* is considered to be a function: $x(t_f) = F(t_0, t_f, x(t_0), p)$
- Currently one explicit integrator (CVodes) and one implicit integrator (IDAS)
- AD forward/reverse corresponds to forward/adjoint *sensitivities*

The `Simulator` class

- Evaluates an *output function*, $h(t, x, p)$ in a *set of time points*, $[t_1, \ldots, t_n]$, using an arbitrary *integrator*
- Also considered a function: $[y_1, \ldots, y_n, x(t_f)] = G([t_0, t_f, x(t_0), p, [t_1, \ldots, t_n])$ with $y_i := h(t_i, x(t_i), p)$

Relation to dynamic optimization

The integrator and simulator classes used e.g. in shooting-methods

Results

Determinant calculation

- AD speed benchmark (ADOL-C, CppAD)
- $f(X) = |X|, \quad X \in \mathbb{R}^{N \times N}$ by minor expansion
- Complexity exponential in N
- Adjoint derivatives
- Intel Core Duo 2.4 GHz, 4 GB RAM, 3072 KB L2 cache, 128 KB L1 cache

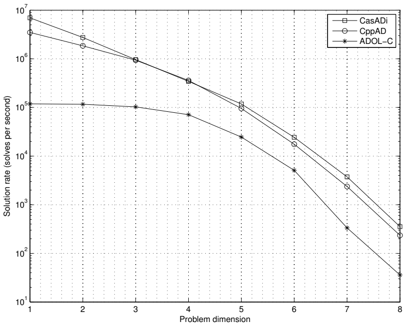
Results

Determinant calculation

- AD speed benchmark (ADOL-C, CppAD)
- $f(X) = |X|, \quad X \in \mathbb{R}^{N \times N}$ by minor expansion
- Complexity exponential in N
- Adjoint derivatives
- Intel Core Duo 2.4 GHz, 4 GB RAM, 3072 KB L2 cache, 128 KB L1 cache

Results

- Outperforms ADOL-C, keeps up with CppAD up to 8-by-8 ($\approx$ 100.000 operations)
- $\sim$ 20 ns per operation (= speed of cache)
- Optimized c-code (generated), not much faster



Results

Minimal fuel rocket flight

$$
\begin{array}{ll}
\underset{u,\,s,\,v,\,m}{\textbf{minimize:}} & -m(T) \\
\textbf{subject to:} & \begin{array}{l} \dot{s} = v \\ \dot{v} = (u - \alpha\, v^2)/m \\ \dot{m} = -\beta\, u^2 \\ s(0) = 0, \quad s(T) = 10 \\ v(0) = 0, \quad v(T) = 0 \\ m(0) = 1,\ T = 10 \\ -10 \leq u \leq 10 \end{array}
\end{array} \tag{1}
$$

Euler forward integrator

```
SX s_0("s_0"), v_0("v_0"), m_0("m_0");
std::vector<SX> x0 = {s_0,v_0,m_0};
SX u("u");
SX s = s_0, v = v_0, m = m_0;
double dt = 10.0/1000;
for(int j=0; j<1000; ++j){
  s += dt*v;
  v += dt / m * (u - alpha * v*v);
  m += -dt * beta*u*u;
}
std::vector<SX> x = {s,v,m};
FX integrator = SXFunction({x0,u},x);
```

Single shooting

```
std::vector<double> X0 = {0,0,1}; // X at t=0
MX X = X0; // state vector
MX U("U",1000); // control vector
for(int k=0; k<1000; ++k){
  X = integrator.evaluate({X,U[k]});
}
MX s_T = X[0], v_T = X[1], m_T = X[2];
...
```

Results

Minimal fuel rocket flight

$$
\begin{array}{lll}
\underset{u,\, s,\, v,\, m}{\textbf{minimize:}} & -m(T) & \\
 & \dot{s} = v & \\
 & \dot{v} = (u - \alpha\, v^2)/m & \\
 & \dot{m} = -\beta\, u^2 & (1) \\
\textbf{subject to:} & s(0) = 0, \quad s(T) = 10 & \\
 & v(0) = 0, \quad v(T) = 0 & \\
 & m(0) = 1,\ T = 10 & \\
 & -10 \leq u \leq 10 &
\end{array}
$$

Solution

- 1000 controls intervals, 1000 steps per interval

Euler forward integrator

```
SX s_0("s_0"), v_0("v_0"), m_0("m_0");
std::vector<SX> x0 = {s_0,v_0,m_0};
SX u("u");
SX s = s_0, v = v_0, m = m_0;
double dt = 10.0/1000;
for(int j=0; j<1000; ++j){
  s += dt*v;
  v += dt / m * (u - alpha * v*v);
  m += -dt * beta*u*u;
}
std::vector<SX> x = {s,v,m};
FX integrator = SXFunction({x0,u},x);
```

Single shooting

```
std::vector<double> X0 = {0,0,1}; // X at t=0
MX X = X0; // state vector
MX U("U",1000); // control vector
for(int k=0; k<1000; ++k){
  X = integrator.evaluate({X,U[k]});
}
MX s_T = X[0], v_T = X[1], m_T = X[2];
...
```

Results

Minimal fuel rocket flight

$$
\begin{array}{lll}
\textbf{minimize:} & -m(T) & \\
u, s, v, m & & \\
 & \dot{s} = v & \\
 & \dot{v} = (u - \alpha\, v^2)/m & \\
 & \dot{m} = -\beta\, u^2 & (1) \\
\textbf{subject to:} & s(0) = 0, \quad s(T) = 10 & \\
 & v(0) = 0, \quad v(T) = 0 & \\
 & m(0) = 1, \, T = 10 & \\
 & -10 \leq u \leq 10 &
\end{array}
$$

Solution

- 1000 controls intervals, 1000 steps per interval
- Build graph for integrating over a single interval
- This graph defines the function "integrator"
- Build up a graph with calls

Euler forward integrator

```
SX s_0("s_0"), v_0("v_0"), m_0("m_0");
std::vector<SX> x0 = {s_0,v_0,m_0};
SX u("u");
SX s = s_0, v = v_0, m = m_0;
double dt = 10.0/1000;
for(int j=0; j<1000; ++j){
  s += dt*v;
  v += dt / m * (u - alpha * v*v);
  m += -dt * beta*u*u;
}
std::vector<SX> x = {s,v,m};
FX integrator = SXFunction({x0,u},x);
```

Single shooting

```
std::vector<double> X0 = {0,0,1}; // X at t=0
MX X = X0; // state vector
MX U("U",1000); // control vector
for(int k=0; k<1000; ++k){
  X = integrator.evaluate({X,U[k]});
}
MX s_T = X[0], v_T = X[1], m_T = X[2];
...
```

Results

Minimal fuel rocket flight

$$
\begin{array}{ll}
\underset{u,\,s,\,v,\,m}{\textbf{minimize:}} & -m(T) \\
 & \dot{s} = v \\
 & \dot{v} = (u - \alpha\, v^2)/m \\
 & \dot{m} = -\beta\, u^2 \\
\textbf{subject to:} & s(0) = 0, \quad s(T) = 10 \\
 & v(0) = 0, \quad v(T) = 0 \\
 & m(0) = 1, \; T = 10 \\
 & -10 \leq u \leq 10
\end{array} \tag{1}
$$

Solution

- 1000 controls intervals, 1000 steps per interval
- Build graph for integrating over a single interval
- This graph defines the function "integrator"
- Build up a graph with calls
- This graph defines objective and constraint funcs.
- Pass to NLP solver (IPOPT)

Euler forward integrator

```
SX s_0("s_0"), v_0("v_0"), m_0("m_0");
std::vector<SX> x0 = {s_0,v_0,m_0};
SX u("u");
SX s = s_0, v = v_0, m = m_0;
double dt = 10.0/1000;
for(int j=0; j<1000; ++j){
  s += dt*v;
  v += dt / m * (u - alpha * v*v);
  m += -dt * beta*u*u;
}
std::vector<SX> x = {s,v,m};
FX integrator = SXFunction({x0,u},x);
```

Single shooting

```
std::vector<double> X0 = {0,0,1}; // X at t=0
MX X = X0; // state vector
MX U("U",1000); // control vector
for(int k=0; k<1000; ++k){
  X = integrator.evaluate({X,U[k]});
}
MX s_T = X[0], v_T = X[1], m_T = X[2];
...
```

Results

Minimal fuel rocket flight

$$
\begin{array}{ll}
\underset{u,\,s,\,v,\,m}{\textbf{minimize:}} & -m(T) \\
\textbf{subject to:} & \dot{s} = v \\
& \dot{v} = (u - \alpha\, v^2)/m \\
& \dot{m} = -\beta\, u^2 \\
& s(0) = 0, \quad s(T) = 10 \\
& v(0) = 0, \quad v(T) = 0 \\
& m(0) = 1,\; T = 10 \\
& -10 \leq u \leq 10
\end{array} \tag{1}
$$

Solution

- 1000 controls intervals, 1000 steps per interval
- Build graph for integrating over a single interval
- This graph defines the function "integrator"
- Build up a graph with calls
- This graph defines objective and constraint funcs.
- Pass to NLP solver (IPOPT)
- Convergence after 11 iteration, 10.4 s.

Euler forward integrator

```
SX s_0("s_0"), v_0("v_0"), m_0("m_0");
std::vector<SX> x0 = {s_0,v_0,m_0};
SX u("u");
SX s = s_0, v = v_0, m = m_0;
double dt = 10.0/1000;
for(int j=0; j<1000; ++j){
  s += dt*v;
  v += dt / m * (u - alpha * v*v);
  m += -dt * beta*u*u;
}
std::vector<SX> x = {s,v,m};
FX integrator = SXFunction({x0,u},x);
```

Single shooting

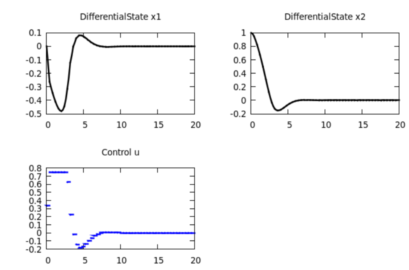
```
std::vector<double> X0 = {0,0,1}; // X at t=0
MX X = X0; // state vector
MX U("U",1000); // control vector
for(int k=0; k<1000; ++k){
  X = integrator.evaluate({X,U[k]});
}
MX s_T = X[0], v_T = X[1], m_T = X[2];
...
```

Results

Van-der-pol oscillator

- From JModelica example collection
- Export OCP as Modelica/Optimica XML
- Parse the XML code in CasADi, reconstruct OCP
- Solve with ACADO Toolkit
  - Open-source dynamic optimization software from OPTEC
  - Houska & Ferreau 2008-present
  - `www.acadotoolkit.org`
  - Spatial discretization with multiple-shooting
  - Non-linear program (NLP) solved with Sequential Quadratic Programming (SQP)
  - Limited memory Hessian approximation
  - Initialized with $u = 0$ for all t
- Convergence after 26 iterations

$$
\begin{array}{ll}
\underset{x_1(\cdot),\, x_2(\cdot),\, u(\cdot)}{\textbf{minimize:}} & \displaystyle\int_0^{20} e^{p_3}\,(x_1^2 + x_2^2 + u^2)\,dt \\
\textbf{subject to:} & \begin{array}{l} \dot{x}_1 = (1 - x_2^2)\,x_1 - x_2 + u \\ \dot{x}_2 = p_1\, x_1 \\ x_1(0) = 0, \quad x_2(0) = 1 \\ u \leq 0.75 \end{array}
\end{array} \tag{2}
$$



Conclusions

Conclusions

- OCPs can be step-by-step symbolically reformulated into a NLP

Conclusions

- OCPs can be step-by-step symbolically reformulated into a NLP
- Benefits: Structure exploiting, no compiler in-the-loop, no retapes for hybrid systems

Conclusions

- OCPs can be step-by-step symbolically reformulated into a NLP
- Benefits: Structure exploiting, no compiler in-the-loop, no retapes for hybrid systems
- By using a combination of two sorts of graphs, we can satisfy demands on speed, memory *and* generality

Conclusions

- OCPs can be step-by-step symbolically reformulated into a NLP
- Benefits: Structure exploiting, no compiler in-the-loop, no retapes for hybrid systems
- By using a combination of two sorts of graphs, we can satisfy demands on speed, memory *and* generality
- Shooting methods can be implemented by introducing ODE/DAE integrators into the directed graphs

Conclusions

- OCPs can be step-by-step symbolically reformulated into a NLP
- Benefits: Structure exploiting, no compiler in-the-loop, no retapes for hybrid systems
- By using a combination of two sorts of graphs, we can satisfy demands on speed, memory *and* generality
- Shooting methods can be implemented by introducing ODE/DAE integrators into the directed graphs
- CasADi: Open-source software project

Outlook

Next steps

- First public release of CasADi (autumn 2010)
- Fully integrate with JModelica, support all features
- Applications: solar power, combined cycle, hydropower valley
- Large-scale (distributed, parallel) SQP (with C. Savorgnan, A. Kozma)

Outlook

Next steps

- First public release of CasADi (autumn 2010)
- Fully integrate with JModelica, support all features
- Applications: solar power, combined cycle, hydropower valley
- Large-scale (distributed, parallel) SQP (with C. Savorgnan, A. Kozma)

Further research

- Dynamic optimization of large-scale *hybrid* systems (reformulate as MINLP)
- PDE constrained optimization

Thank you for listening.