Linköping, June 2011

CasADi

Joel Andersson   Moritz Diehl

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

Outline

Motivation – Large-scale Optimal Control Problems (OCP)

$$
\begin{aligned}
&\min_{x,u,p} \int_0^T l(t,x,u,p)\,dt + E(x(T),p) \\
&\text{subj. to} \\
&\qquad \dot{x} = f(t,x,u,p) = 0 \quad t \in [0,T] \\
&\qquad h(t,x,u,p) \le 0 \quad t \in [0,T] \\
&\qquad x(0) = x_0 \\
&\qquad x_{\min} \le x \le x_{\max} \quad t \in [0,T] \\
&\qquad u_{\min} \le u \le u_{\max} \quad t \in [0,T] \\
&\qquad p_{\min} \le p \le p_{\max}
\end{aligned}
\tag{1}
$$

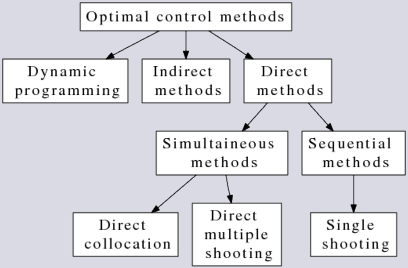
Here $x(\cdot) \in \mathbf{R}^{N_x}$ (differential) states, $u(\cdot) \in \mathbf{R}^{N_u}$ free control signals and $p \in \mathbf{R}^{N_p}$ free parameters.

Motivation

Solving optimal control problems

Methods for solving OCP's:

- *Dynamic programming/HJB*
  "Smart exhaustive search"
- *Indirect methods*
  "Solve necessary conditions for optimality"
- **Direct methods**
  "Reformulate as a nonlinear programming problem (NLP)"

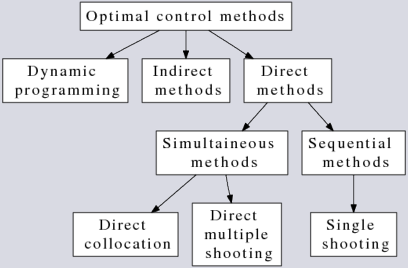


Motivation

Solving optimal control problems

Methods for solving OCP's:

- *Dynamic programming/HJB*
  "Smart exhaustive search"
- *Indirect methods*
  "Solve necessary conditions for optimality"
- **Direct methods**
  "Reformulate as a nonlinear programming problem (NLP)"



Direct methods

- Single shooting: parametrize only controls, eliminate state with ODE/DAE integrators
- Simultaneous methods: parametrize controls *and* state
  - Direct collocation: Fine grid – **interpolate** between gridpoints
  - Direct multiple shooting: Coarse grid – **integrate** between gridpoints
- Solve NLP with (structure exploiting) SQP or IP method

Automatic differentiation, AD

Efficient procedure to automatically calculate derivatives:

$$F(x) : \mathbb{R}^n \rightarrow \mathbb{R}^m \qquad \Rightarrow \qquad J(x) = \frac{\partial F}{\partial x}(x) : \mathbb{R}^n \rightarrow \mathbb{R}^{m \times n} \tag{2}$$

Automatic differentiation, AD

Efficient procedure to automatically calculate derivatives:

$$F(x) : \mathbb{R}^n \rightarrow \mathbb{R}^m \qquad \Rightarrow \qquad J(x) = \frac{\partial F}{\partial x}(x) : \mathbb{R}^n \rightarrow \mathbb{R}^{m \times n} \tag{2}$$

How AD works

- Write F as a sequence of elementary operations:

$$\begin{aligned}
y_{i-n} &= x_i, & i &\in \{1, \ldots, n\} & &\text{independent inputs} \quad (3)\\
y_i &= f_i(y_{j_i}, y_{k_i}), \quad j_i < i,\ k_i < i & i &\in \{1, \ldots, p\} & &\text{intermediate calculations} \quad (4)\\
z_j &= y_{i_j}, & j &\in \{1, \ldots, m\} & &\text{function outputs} \quad (5)
\end{aligned}$$

- Consider the equality (with $y = [y_1, \ldots, y_p]$):

$$\begin{bmatrix} y \\ z \end{bmatrix} = \tilde{F}(x, y) \tag{6}$$

Now differentiate the extended equation

$$\begin{bmatrix} y \\ z \end{bmatrix} = \tilde{F}(x, y) \Rightarrow \begin{bmatrix} \dot{y} \\ \dot{z} \end{bmatrix} = \begin{bmatrix} A & L \\ B & M \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} \tag{7}$$

Now differentiate the extended equation

$$\begin{bmatrix} y \\ z \end{bmatrix} = \tilde{F}(x, y) \Rightarrow \begin{bmatrix} \dot{y} \\ \dot{z} \end{bmatrix} = \begin{bmatrix} A & L \\ B & M \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} \tag{7}$$

Eliminate $\dot{y}$

$$\dot{z} = \left[B + M\,(I - L)^{-1}\,A\right]\,\dot{x} = J\,\dot{x} \tag{8}$$

Now differentiate the extended equation

$$\begin{bmatrix} y \\ z \end{bmatrix} = \tilde{F}(x, y) \Rightarrow \begin{bmatrix} \dot{y} \\ \dot{z} \end{bmatrix} = \begin{bmatrix} A & L \\ B & M \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} \tag{7}$$

Eliminate $\dot{y}$

$$\dot{z} = \left[B + M\,(I - L)^{-1}\,A\right]\,\dot{x} = J\,\dot{x} \tag{8}$$

Forward and adjoint mode AD

- Cheap to multiply J with a vector (A, B, L, M sparse, $I - L$ lower triangular):

Now differentiate the extended equation

$$\begin{bmatrix} y \\ z \end{bmatrix} = \tilde{F}(x, y) \Rightarrow \begin{bmatrix} \dot{y} \\ \dot{z} \end{bmatrix} = \begin{bmatrix} A & L \\ B & M \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} \tag{7}$$

Eliminate $\dot{y}$

$$\dot{z} = \left[B + M\,(I - L)^{-1}\,A\right]\,\dot{x} = J\,\dot{x} \tag{8}$$

Forward and adjoint mode AD

- Cheap to multiply J with a vector (A, B, L, M sparse, $I - L$ lower triangular):
  - From the right (*forward mode*): $J\,v = B\,v + M\,(I - L)^{-1}\,A\,v$
  - From the left (*adjoint mode*): $v^\mathrm{T}\,J = v^\mathrm{T}\,B + v^\mathrm{T}\,M\,(I - L)^{-1}\,A$

Background

Now differentiate the extended equation

$$\left[\begin{array}{c} y \\ z \end{array}\right] = \tilde{F}(x, y) \Rightarrow \left[\begin{array}{c} \dot{y} \\ \dot{z} \end{array}\right] = \left[\begin{array}{cc} A & L \\ B & M \end{array}\right] \left[\begin{array}{c} \dot{x} \\ \dot{y} \end{array}\right] \tag{7}$$

Eliminate $\dot{y}$

$$\dot{z} = \left[B + M\,(I - L)^{-1}\,A\right]\,\dot{x} = J\,\dot{x} \tag{8}$$

Forward and adjoint mode AD

- Cheap to multiply J with a vector (A, B, L, M sparse, $I - L$ lower triangular):
  - From the right (*forward mode*): $J\,v = B\,v + M\,(I - L)^{-1}\,A\,v$
  - From the left (*adjoint mode*): $v^{\mathrm{T}}\,J = v^{\mathrm{T}}\,B + v^{\mathrm{T}}\,M\,(I - L)^{-1}\,A$
- To calculate the full Jacobian, multiply by several forward and/or adjoint directions
- NP-hard optimization problem to find the least number of directions
- AD software tools: ADOL-C, CppAD, OpenAD, ...

Outline

Existing tools for OCP

- Tools exist that accept OCP:s in a standard form and solves the problem...
    - Shooting methods (e.g. MUSCOD-II, ACADO Toolkit)
    - Direct collocation (e.g. DIRCOL)
    - ...

Existing tools for OCP

- Tools exist that accept OCP:s in a standard form and solves the problem...
  - Shooting methods (e.g. MUSCOD-II, ACADO Toolkit)
  - Direct collocation (e.g. DIRCOL)
  - ...
- ... but advanced users often prefer to take the "NLP approach" (using e.g. AMPL)

Existing tools for OCP

- Tools exist that accept OCP:s in a standard form and solves the problem...
  - Shooting methods (e.g. MUSCOD-II, ACADO Toolkit)
  - Direct collocation (e.g. DIRCOL)
  - ...
- ... but advanced users often prefer to take the "NLP approach" (using e.g. AMPL)
  - The user reformulates the OCP as an NLP
  - Derivative information is generated automatically and passed to the NLP solver

Existing tools for OCP

- Tools exist that accept OCP:s in a standard form and solves the problem...
  - Shooting methods (e.g. MUSCOD-II, ACADO Toolkit)
  - Direct collocation (e.g. DIRCOL)
  - ...
- ... but advanced users often prefer to take the "NLP approach" (using e.g. AMPL)
  - The user reformulates the OCP as an NLP
  - Derivative information is generated automatically and passed to the NLP solver
  - Advantages:
    - Can formulate arbitrarily complex non-standard OCP:s
    - User gets a better insight
  - Drawback:
    - Until now only for collocation methods

CasADi

What is CasADi?

CasADi = Computer algebra system for Automatic Differentiation. An open-source (LGPL) symbolic framework for quick, yet efficient, implementation of derivative based algorithms for dynamic optimization

CasADi

What is CasADi?

CasADi = Computer algebra system for Automatic Differentiation. An open-source (LGPL) symbolic framework for quick, yet efficient, implementation of derivative based algorithms for dynamic optimization

Takes the NLP approach to solving optimal control problems and extends it to shooting methods (multiple shooting method in 30–50 lines)

CasADi

What is CasADi?

CasADi = Computer algebra system for Automatic Differentiation. An open-source (LGPL) symbolic framework for quick, yet efficient, implementation of derivative based algorithms for dynamic optimization

Takes the NLP approach to solving optimal control problems and extends it to shooting methods (multiple shooting method in 30–50 lines)

`www.casadi.org`

The core of CasADi

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

The core of CasADi

- A minimalistic Computer Algebra System (CAS) written in self-contained C++
  - Matlab-like syntax "everything is a matrix"

The core of CasADi

- A minimalistic Computer Algebra System (CAS) written in self-contained C++
    - Matlab-like syntax "everything is a matrix"
    - Use from C++ or Python (soon also Octave)

The core of CasADi

- A minimalistic Computer Algebra System (CAS) written in self-contained C++
    - Matlab-like syntax “everything is a matrix”
    - Use from C++ or Python (soon also Octave)
    - 8 flavors of automatic differentiation

The core of CasADi

- A minimalistic Computer Algebra System (CAS) written in self-contained C++
  - Matlab-like syntax “everything is a matrix”
  - Use from C++ or Python (soon also Octave)
  - 8 flavors of automatic differentiation
    - Forward or adjoint mode

The core of CasADi

- A minimalistic Computer Algebra System (CAS) written in self-contained C++
  - Matlab-like syntax “everything is a matrix”
  - Use from C++ or Python (soon also Octave)
  - 8 flavors of automatic differentiation
    - Forward or adjoint mode
    - Directional derivatives or source-to-source transformation with new graph for Jacobian

The core of CasADi

- A minimalistic Computer Algebra System (CAS) written in self-contained C++
  - Matlab-like syntax "everything is a matrix"
  - Use from C++ or Python (soon also Octave)
  - 8 flavors of automatic differentiation
    - Forward or adjoint mode
    - Directional derivatives or source-to-source transformation with new graph for Jacobian
    - **Scalar or matrix graph representation**

The core of CasADi

- A minimalistic Computer Algebra System (CAS) written in self-contained C++
  - Matlab-like syntax "everything is a matrix"
  - Use from C++ or Python (soon also Octave)
  - 8 flavors of automatic differentiation
    - Forward or adjoint mode
    - Directional derivatives or source-to-source transformation with new graph for Jacobian
    - **Scalar or matrix graph representation**

Automatic differentiation in CasADi

- CasADi applies AD to a sequence of vector/matrix valued operations
  - Conventional AD-tools support only unary or binary elementary operations (+, -, *, sin, sqrt, ...)
  - Matrix operations are *expanded* into a large set of scalar operations
- CasADi allows multiple matrix-valued input, multiple-matrix valued output: $[x, p] \rightarrow [f, \frac{\partial f}{\partial x}, \frac{\partial f}{\partial p}]$
- Also e.g. matrix multiplication, implicitly defined functions (solve linear or nonlinear system), etc.

How to get it to work efficiently?

- Matrix sparsity always fixed, sparsity shared between nodes
- Delay generation of Jacobians of function evaluations until the whole graph is known
- Use two different computational graph representations (scalar / matrix)
  - Avoids that the extra generality comes at the expense of memory use and speed

How to get it to work efficiently?

- Matrix sparsity always fixed, sparsity shared between nodes
- Delay generation of Jacobians of function evaluations until the whole graph is known
- Use two different computational graph representations (scalar / matrix)
  - Avoids that the extra generality comes at the expense of memory use and speed
  - Highly efficient, scalar graph $\sim$10 ns per elementary operation ($<$ 1 ns with code generation)

How to get it to work efficiently?

- Matrix sparsity always fixed, sparsity shared between nodes
- Delay generation of Jacobians of function evaluations until the whole graph is known
- Use two different computational graph representations (scalar / matrix)
    - Avoids that the extra generality comes at the expense of memory use and speed
    - Highly efficient, scalar graph $\sim$10 ns per elementary operation ($<$ 1 ns with code generation)
    - Less efficient matrix graph, called orders of magnitude more seldom

How to get it to work efficiently?

- Matrix sparsity always fixed, sparsity shared between nodes
- Delay generation of Jacobians of function evaluations until the whole graph is known
- Use two different computational graph representations (scalar / matrix)
  - Avoids that the extra generality comes at the expense of memory use and speed
  - Highly efficient, scalar graph $\sim$10 ns per elementary operation ($<$ 1 ns with code generation)
  - Less efficient matrix graph, called orders of magnitude more seldom
  - User responsible for choosing which one to use

How to get it to work efficiently?

- Matrix sparsity always fixed, sparsity shared between nodes
- Delay generation of Jacobians of function evaluations until the whole graph is known
- Use two different computational graph representations (scalar / matrix)
  - Avoids that the extra generality comes at the expense of memory use and speed
  - Highly efficient, scalar graph $\sim$10 ns per elementary operation ($<$ 1 ns with code generation)
  - Less efficient matrix graph, called orders of magnitude more seldom
  - User responsible for choosing which one to use

Two graph representations

	Scalar-valued nodes	**Sparse, matrix-valued nodes**
design objective	"maximum speed"	"maximum generality"
type of operations	built-in	built-in or user-defined
number of inputs	one or two	arbitrary
number of output	one	arbitrary
branching/jumps	no	yes
parallelization	no	yes

"Smart interfaces" to numerical codes

- NLP solvers: **IPOPT**, **KNITRO**, **LiftOpt**

“Smart interfaces“ to numerical codes

- NLP solvers: **IPOPT**, **KNITRO**, **LiftOpt**
  - Automatic generation of exact, sparse Hessians and Jacobians, “lifting”

“Smart interfaces“ to numerical codes

- NLP solvers: **IPOPT**, **KNITRO**, **LiftOpt**
  - Automatic generation of exact, sparse Hessians and Jacobians, “lifting”
- Integrators: **CVODES**, **IDAS**

“Smart interfaces“ to numerical codes

- NLP solvers: **IPOPT**, **KNITRO**, **LiftOpt**
  - Automatic generation of exact, sparse Hessians and Jacobians, “lifting”
- Integrators: **CVODES**, **IDAS**
  - Automatic formulation of forward and adjoint sensitivitity equations

"Smart interfaces" to numerical codes

- NLP solvers: **IPOPT**, **KNITRO**, **LiftOpt**
  - Automatic generation of exact, sparse Hessians and Jacobians, "lifting"
- Integrators: **CVODES**, **IDAS**
  - Automatic formulation of forward and adjoint sensitivitity equations
  - Generation of Jacobian information, sparse direct linear solvers (CSparse, SuperLU)

"Smart interfaces" to numerical codes

- NLP solvers: **IPOPT**, **KNITRO**, **LiftOpt**
  - Automatic generation of exact, sparse Hessians and Jacobians, "lifting"
- Integrators: **CVODES**, **IDAS**
  - Automatic formulation of forward and adjoint sensitivitity equations
  - Generation of Jacobian information, sparse direct linear solvers (CSparse, SuperLU)
- Other tools: **ACADO Toolkit**, **KINSOL**, **CPLEX**

Other features

- Just-in-time compilation
  - Generate C-code from CasADi computational graphs
  - Compile to a dynamic library
  - Dynamic loading

Other features

- Just-in-time compilation
  - Generate C-code from CasADi computational graphs
  - Compile to a dynamic library
  - Dynamic loading
- Import of DAE-systems formulated in Modelica
  - Modelica – DAE modelling language

Other features

- Just-in-time compilation
  - Generate C-code from CasADi computational graphs
  - Compile to a dynamic library
  - Dynamic loading
- Import of DAE-systems formulated in Modelica
  - Modelica – DAE modelling language
- Symbolic reformulation of DAE:s
  - Sorting/scaling of variables and equations
  - Elimination of some or all algebraic states *symbolically*

Outline

Direct collocation

- NLP approach: User writes NLP objective and constraint functions

Direct collocation

- NLP approach: User writes NLP objective and constraint functions
- Matrix algebra keeps size of constructed graphs small

Direct collocation

- NLP approach: User writes NLP objective and constraint functions
- Matrix algebra keeps size of constructed graphs small
- Automatic calculation of Jacobian and Hessian information

Direct collocation

- NLP approach: User writes NLP objective and constraint functions
- Matrix algebra keeps size of constructed graphs small
- Automatic calculation of Jacobian and Hessian information
- Solve with an interfaced NLP solver (IPOPT, KNITRO)

Direct collocation

- NLP approach: User writes NLP objective and constraint functions
- Matrix algebra keeps size of constructed graphs small
- Automatic calculation of Jacobian and Hessian information
- Solve with an interfaced NLP solver (IPOPT, KNITRO)

Shooting methods

- Everything above is valid

Optimal control using CasADi

Direct collocation

- NLP approach: User writes NLP objective and constraint functions
- Matrix algebra keeps size of constructed graphs small
- Automatic calculation of Jacobian and Hessian information
- Solve with an interfaced NLP solver (IPOPT, KNITRO)

Shooting methods

- Everything above is valid
- Add ODE/DAE integrator calls to the computational graph

Optimal control using CasADi

Direct collocation

- NLP approach: User writes NLP objective and constraint functions
- Matrix algebra keeps size of constructed graphs small
- Automatic calculation of Jacobian and Hessian information
- Solve with an interfaced NLP solver (IPOPT, KNITRO)

Shooting methods

- Everything above is valid
- Add ODE/DAE integrator calls to the computational graph
- Automatic formulation of ODE/DAE sensitivity equations

Optimal control using CasADi

Direct collocation

- NLP approach: User writes NLP objective and constraint functions
- Matrix algebra keeps size of constructed graphs small
- Automatic calculation of Jacobian and Hessian information
- Solve with an interfaced NLP solver (IPOPT, KNITRO)

Shooting methods

- Everything above is valid
- Add ODE/DAE integrator calls to the computational graph
- Automatic formulation of ODE/DAE sensitivity equations
- Automatic ODE/DAE Jacobian information, preconditioning

Optimal control using CasADi

Code example: Complete single-shooting method in 30 lines of code!

```
from casadi import *

# Declare variables (use simple, efficient DAG)
t = SX("t") # time
x=SX("x"); y=SX("y"); u=SX("u"); L=SX("cost")

# ODE right hand side function
f = [(1 - y*y)*x - y + u, x, x*x + y*y + u*u]
rhs = SXFunction([[t],[x,y,L],[u]],[f])

# Create an integrator (CVodes)
I = CVodesIntegrator(rhs)
I.setOption("abstol",1e-10) # abs. tolerance
I.setOption("reltol",1e-10) # rel. tolerance
I.setOption("steps_per_checkpoint",100)
I.init()

# All controls (use complex, general DAG)
NU = 20; U = MX("U",NU)

# The initial state (x=0, y=1, L=0)
X  = MX([0,1,0])

# Time horizon
T0 = MX(0);  TF = MX(20.0/NU)

# State derivative, algebraic state (not used)
XP = MX(); Z = MX()

# Build up a graph of integrator calls
for k in range(NU):
  [X,XP,Z] = I.call([T0,TF,X,U[k],XP,Z])

# Objective function: L(T)
F = MXFunction([U],[X[2]])

# Terminal constraints: 0<=[x(T);y(T)]<=0
G = MXFunction([U],[X[0:2]])

# Create NLP solver
solver = IpoptSolver(F,G)
solver.setOption("tol",1e-5)
solver.setOption("hessian_approximation", \
                 "limited-memory")
solver.setOption("max_iter",1000)
solver.init()

# Set bounds and initial guess
solver.setInput(NU*[-0.75], NLP_LBX)
solver.setInput(NU*[1.0],NLP_UBX)
solver.setInput(NU*[0.0],NLP_X_INIT)
solver.setInput([0,0],NLP_LBG)
solver.setInput([0,0],NLP_UBG)

# Solve the problem
solver.solve()
```

Summary

- CasADi
  - Open-source project, `www.casadi.org`

Summary

- CasADi
  - Open-source project, `www.casadi.org`
  - Computer algebra system with AD using two graph representations

Summary

- CasADi
  - Open-source project, `www.casadi.org`
  - Computer algebra system with AD using two graph representations
  - Use from C++, Python or (soon) Octave

Summary

- CasADi
  - Open-source project, `www.casadi.org`
  - Computer algebra system with AD using two graph representations
  - Use from C++, Python or (soon) Octave
  - “Smart interfaces“ to popular numerical codes

Summary

Summary

- CasADi
  - Open-source project, `www.casadi.org`
  - Computer algebra system with AD using two graph representations
  - Use from C++, Python or (soon) Octave
  - “Smart interfaces“ to popular numerical codes
- Dynamic optimization using CasADi
  - “NLP approach”, extended to shooting methods

Summary

Summary

- CasADi
  - Open-source project, `www.casadi.org`
  - Computer algebra system with AD using two graph representations
  - Use from C++, Python or (soon) Octave
  - “Smart interfaces“ to popular numerical codes
- Dynamic optimization using CasADi
  - “NLP approach”, extended to shooting methods
  - Automatic generation of derivative/sensitivity information

Summary

- CasADi
  - Open-source project, `www.casadi.org`
  - Computer algebra system with AD using two graph representations
  - Use from C++, Python or (soon) Octave
  - “Smart interfaces“ to popular numerical codes
- Dynamic optimization using CasADi
  - “NLP approach”, extended to shooting methods
  - Automatic generation of derivative/sensitivity information
  - Automatic generation of sparse NLP Jacobians

Summary

- CasADi
  - Open-source project, `www.casadi.org`
  - Computer algebra system with AD using two graph representations
  - Use from C++, Python or (soon) Octave
  - “Smart interfaces“ to popular numerical codes
- Dynamic optimization using CasADi
  - “NLP approach”, extended to shooting methods
  - Automatic generation of derivative/sensitivity information
  - Automatic generation of sparse NLP Jacobians
  - A number of codes already written (multiple shooting/collocation,..)

Outlook

- More efficient Jacobians using graph coloring
  - Bidirectional (combination of AD forward and adjoint)

Summary

Outlook

- More efficient Jacobians using graph coloring
  - Bidirectional (combination of AD forward and adjoint)
  - Partial colorings (calculate only parts of the Jacobian)

Outlook

- More efficient Jacobians using graph coloring
  - Bidirectional (combination of AD forward and adjoint)
  - Partial colorings (calculate only parts of the Jacobian)
  - Substitution-based methods

Outlook

- More efficient Jacobians using graph coloring
  - Bidirectional (combination of AD forward and adjoint)
  - Partial colorings (calculate only parts of the Jacobian)
  - Substitution-based methods
  - Symmetry exploitation (for Hessians)

Summary

Outlook

- More efficient Jacobians using graph coloring
  - Bidirectional (combination of AD forward and adjoint)
  - Partial colorings (calculate only parts of the Jacobian)
  - Substitution-based methods
  - Symmetry exploitation (for Hessians)

- Push the limits on the speed and maximum problem sizes

Outlook

- More efficient Jacobians using graph coloring
  - Bidirectional (combination of AD forward and adjoint)
  - Partial colorings (calculate only parts of the Jacobian)
  - Substitution-based methods
  - Symmetry exploitation (for Hessians)
- Push the limits on the speed and maximum problem sizes
  - Large-scale DAEs from Modelica
  - PDE constrained optimization?
- New optimization algorithms

Summary

Outlook

- More efficient Jacobians using graph coloring
  - Bidirectional (combination of AD forward and adjoint)
  - Partial colorings (calculate only parts of the Jacobian)
  - Substitution-based methods
  - Symmetry exploitation (for Hessians)
- Push the limits on the speed and maximum problem sizes
  - Large-scale DAEs from Modelica
  - PDE constrained optimization?
- New optimization algorithms
- Applications

Outlook

- More efficient Jacobians using graph coloring
  - Bidirectional (combination of AD forward and adjoint)
  - Partial colorings (calculate only parts of the Jacobian)
  - Substitution-based methods
  - Symmetry exploitation (for Hessians)
- Push the limits on the speed and maximum problem sizes
  - Large-scale DAEs from Modelica
  - PDE constrained optimization?
- New optimization algorithms
- Applications

Thank you for listening!