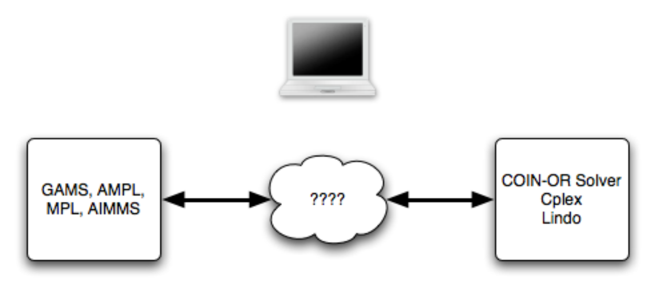
Meeting the Challenges of Solver Independence

Steven Dirkse
GAMS
Lou Hafer Simon
Fraser University
Kipp Martin
University of Chicago
Ted Ralphs
Lehigh University

November 6, 2007

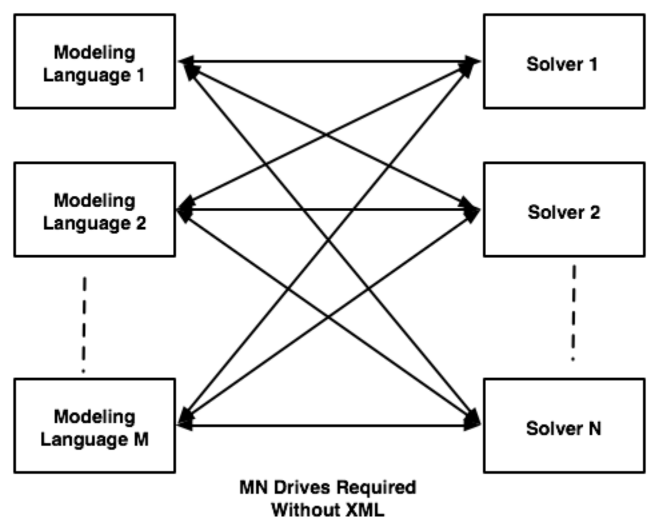
Motivation

The context: a loosely coupled modeling language and a solver.



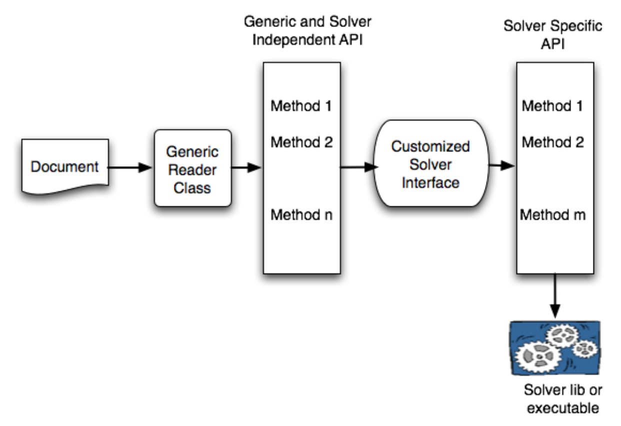
Motivation

The problem:



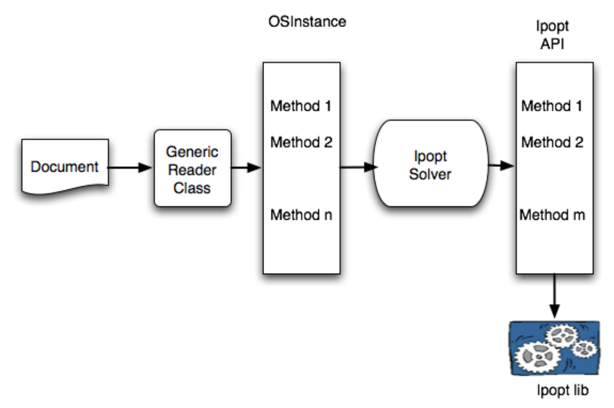
Motivation

Here is another view of solver side



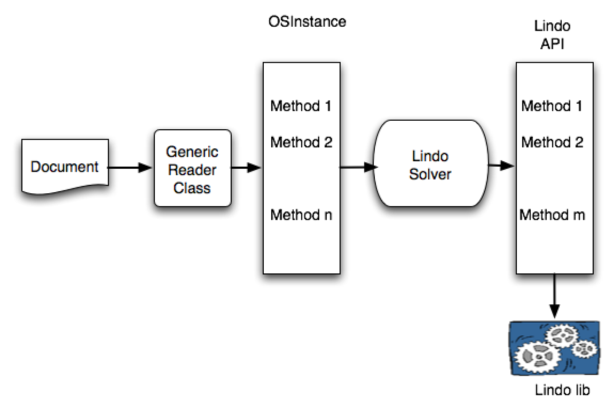
Motivation

Here is another view of solver side – with Ipopt



Motivation

Here is another view of solver side – with Lindo



Motivation

An ideal world – no solver interface!

If the instance interface and option interface are robust enough then we have:

```
load(problem_instance);
solve(options_list);
```

It is up to the solver to implement the problem_instance and interpret what is in the options_list

COIN-OR Open Solver Interface

Problem 1: Only works for mixed-integer linear.

Extensions

- General nonlinear
- Cone
- Stochastic
- Constraint

COIN-OR Open Solver Interface

Problem 2: Does not distinguish between instance, solver options, instance result, and instance modification.

Important Concept

It is important to distinguish between:

- The model instance
- Solver options
- Solver results
- Model modifications

Important Concept

The Optimization Services project is designed to help provide this framework.

- The model instance – **OSInstance class**
- Solver options – **OSOptions class**
- Solver results – **OSResult class**
- Model modifications – to be completed

COIN-OR Open Solver Interface

Problem modification: borrow from the Gang of Four *Design Patters*

Use the **command class** for storing modifications.

COIN-OR Open Solver Interface

Two key classes:

- OsiXFormMgr
- OsiXForm

OsiXForm

Has pure virtual functions:

- virtual void record() = 0;
- virtual void modify() = 0;

Example

class ModifyConstraintBounds : public OsiXForm

- virtual void record() = 0;
- virtual void modify() = 0;

Algorithmic Differentiation – some motivation

Key Idea: The API of nonlinear solvers not really setup to maximize the efficient use of AD.

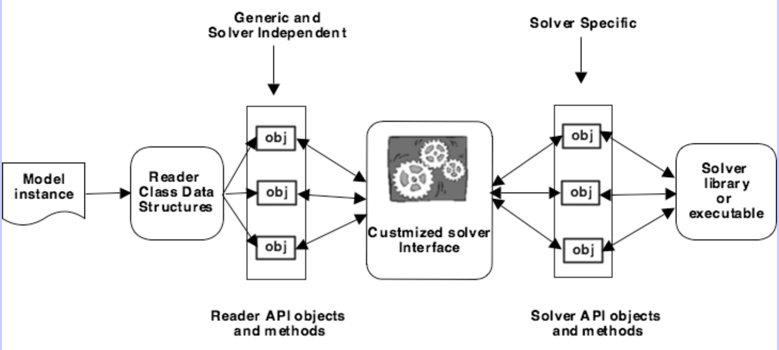
A typical API will have methods such as:

- get an objective function value
- get constraint values
- get objective gradient
- get constraint Jacobian
- get Hessian of Lagrangian

An Issue: from an AD perspective, when, for example, calculating first derivatives it would be nice to know if a second derivative is also required.

Motivation

Here is another view of solver side



Algorithmic Differentiation – some motivation

The Problem: Many nonlinear algorithms, such as interior point methods **do not** calculate all orders of derivatives for the current iterate.

For example, they may do a simple line search and not use any second derivative information.

In the API method calls for function evaluations they **do not communicate if a higher order derivative is required for the current iterate.**

Algorithmic Differentiation – gradient calculation

calculateAllConstraintFunctionGradients()

The method arguments are:

- **double* x**
- **double* objLambda**
- **double* conLambda**
- **bool new_x**
- **int highestOrder**

Algorithmic Differentiation – Hessian calculation

In our implementation, there are **exactly two** conditions that require a new function or derivative evaluation. A new evaluation is required if and only if

1. The value of `new_x` is true

–OR–

2. For the callback function the value of the input parameter `highestOrder` is strictly greater than the current value of `m_iHhighestOrderEvaluated`.

In the code we keep track of the highest order derivative calculation that has been made.