Optimization Services and Nonlinear Programming

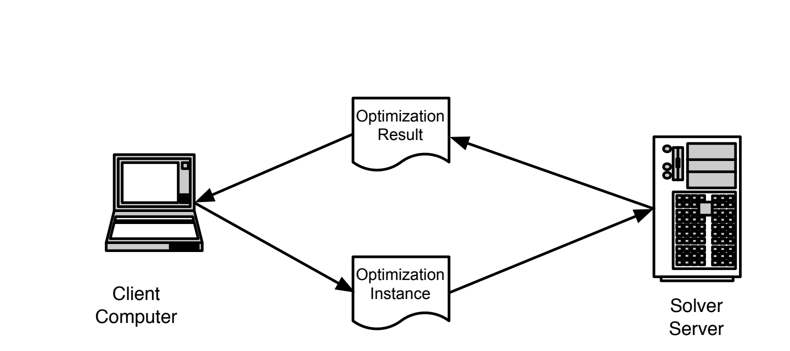
Robert Fourer
Northwestern University
Jun Ma
Northwestern University
Kipp Martin
University of Chicago

November 6, 2007

Outline

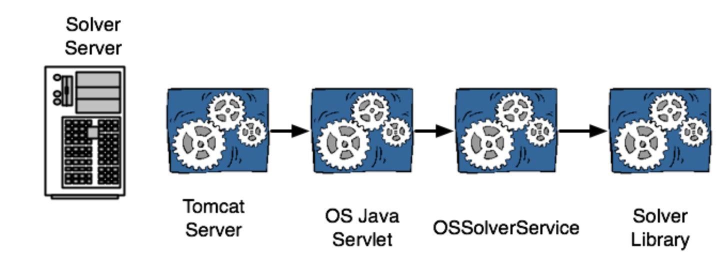
Motivation

Here is the context.



Motivation

Here is what is happening on the solver.



The Solver Side

On the solver end there is an executable **OSSolverService** that communicates with specific solver. Here is what it does.

Step 1: Create an **OSiLReader** object.

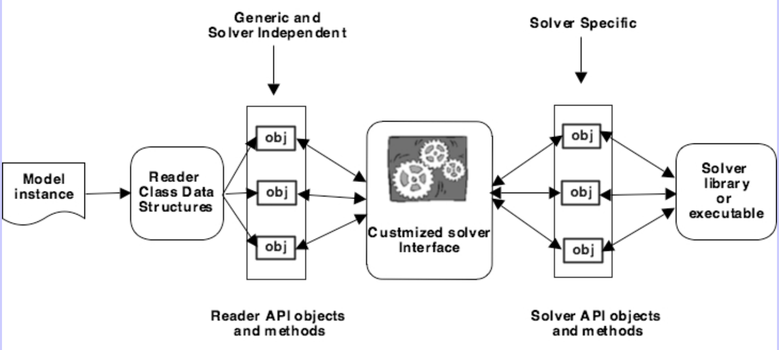
Step 2: Have the **OSiLReader** object parse the OSiL XML instance and create an in-memory **OSInstance** object.

Step 3: Create an object in the specific solver class (e.g. create an IpoptSolver, LindoSolver, or KnitroSolver).

Step 4: Have the solver specific object **work with** the a generic **OSInstance** and the solver library to solve the problem.

Motivation

Here is another view of solver side



OS Customized Solver Interfaces

CoinSolver

Ipopt Solver

Knitro Solver

Lindo Solver

OS Supported Solvers

- Clp – using CoinSolver
- Cbc – using CoinSolver
- Cplex – using CoinSolver
- Dylp – using CoinSolver
- Glpk – using CoinSolver
- **Ipopt – using IpoptSolver**
- **Knitro – using KnitroSolver**
- **Lindo – using KnitroSolver**
- SYMPHONY – using CoinSolver
- Vol – using CoinSolver

Motivation

Objective: Create an API that is as flexible as possible and can deal with numerous solver APIs

In this talk we focus on the nonlinear aspects of the API.

The OS API

The **OSlib** provides in-memory representation of optimization instance, **OSInstance**. It is an API that has three types of methods:

- **get() methods:** a set of methods to get information about the problem instance including the problem in a postfix or prefix format
- **set() methods:** a set of methods to create/modify a problem instance
- **calculate() methods:** a set of methods for performing Algorithmic Differentiation (based upon the COIN-OR CppAD by Brad Bell).

Algorithmic Differentiation

The OS API has both *high level* and *low level* calls for algorithmic differentiation.

Both the high level and low level calls are public methods but **the high level calls**

- are meant to be *user friendly* – the user does not need to know anything about forward sweeps or reverse sweeps or any other aspect of algorithmic differentiation
- the high level calls are similar in nature to calls that the nonlinear solver codes use

Algorithmic Differentiation – Sparsity Patterns

Many derivative-based solvers want to know the sparsity pattern of the constraint Jacobian.

```
sparseJac = osinstance->getJacobianSparsityPattern();
```

- store non zero elements by row in sparse format
- the first **conVals** correspond to variables with constant derivative
- **variables with a constant derivative are never sent to gradient and Hessian calculators**

Algorithmic Differentiation – Sparsity Patterns

Similarly for Hessian of Lagrangian

```
osinstance->getLagrangianHessianSparsityPattern( );
```

Here our code is not real smart smart at this point. For example,

$$x_1^2 + x_2^2 + \cdots + x_n^2$$

will generate a dense Hessian.

Algorithmic Differentiation – some motivation

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

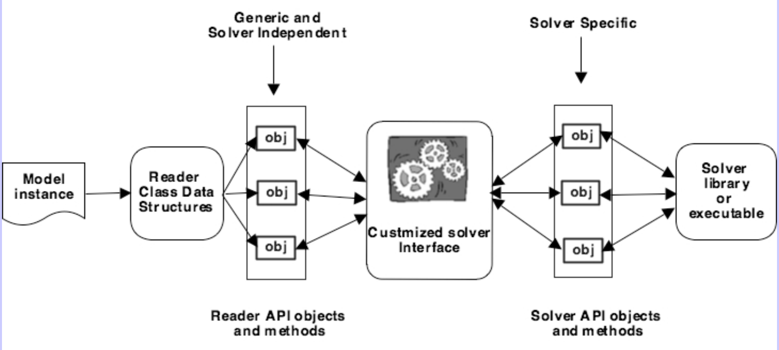
A typical API will have methods such as:

- 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 – gradient calculation

Assume, for example, only first derivative information required.

If a call has been placed to `calculateAllConstraintFunctionValues` with `highestOrder = 0`, then the appropriate call to get gradient evaluations is

```
calculateAllConstraintFunctionGradients( x, NULL, NULL,
false, 1);
```

Note that in this function call `new_x = false`. This prevents a call to `forwardAD()` with order 0 to get the function values.

Algorithmic Differentiation – gradient calculation

If, at the current iterate, the Hessian of the Lagrangian function is also desired then an appropriate call is

```
calculateAllConstraintFunctionGradients(x, objLambda,
    conLambda, false, 2);
```

In this case, if there was a prior call

```
calculateAllConstraintFunctionValues(x, w, z,  true, 0);
```

then only first and second derivatives are calculated, not function values.

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.

Algorithmic Differentiation – Hessian calculation

```
for(index = 0; index < n; index++){
     unit_col_vec[ index] = 1;
     // calculate column i of the Jacobian matrix
     gradVals = f.Forward(1, unit_col_vec);
     unit_col_vec[ index] = 0;
     // get row i of the Lagrangian function!!!
     f.Reverse(2, lagMultipliers);
}
```

In a bad implementation, I could end up doing a forward sweep three times.

The OS API – Postfix

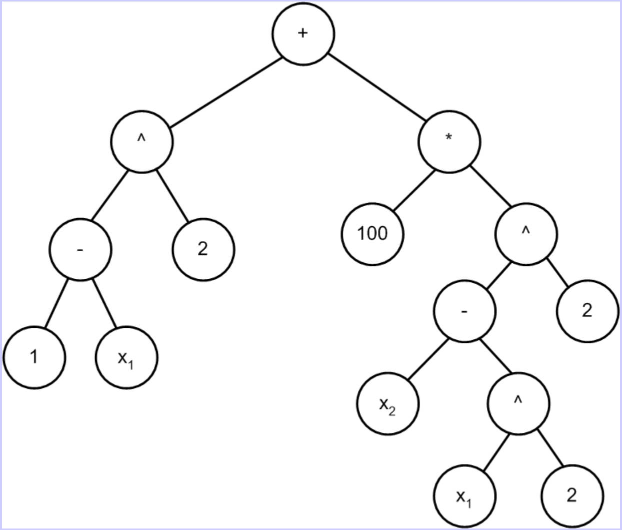
Postfix is an excellent way to represent a wide variety of optimization problems.

By defining enough operators you can model very generic nonlinear optimization.

A good structure for implementing algorithmic differentiation.

As an example, the internal representation of an optimization instance in LINDO is a postfix instruction list.

The OS API – Postfix



The OS API – Postfix

Key Postfix related methods.

In order to get the problem in postfix use the method:

```
getNonlinearExpressionTreeInPostfix( int rowIdx);
```

This returns a vector of pointers to **OSnLNode** objects.

There is an OSnLNode for every operator.

OS Supported Operators

- OSnLNodeVariable
- OSnLNodeTimes
- OSnLNodePlus
- OSnLNodeSum
- OSnLNodeMinus
- OSnLNodeNegate
- OSnLNodeDivide
- OSnLNodePower
- OSnLNodeProduct
- OSnLNodeLn
- OSnLNodeSqrt
- OSnLNodeSquare
- OSnLNodeSin
- OSnLNodeCos
- OSnLNodeExp
- OSnLNodeif
- OSnLNodeAbs
- OSnLNodeMax
- OSnLNodeMin
- OSnLNodeE
- OSnLNodePI
- OSnLNodeAllDiff

Using Lindo

Step 1: For each row and the objective function convert the expression tree into a list of OSnLNodes in postfix.

Step 2: Loop over the list of OSnLNodes and create a Lindo instructions list (map between OS operators and Lindo operator codes).

Step 3: Call the Lindo **LSaddInstruct()** method.

Step 4: Solve!

Motivation

Here is another view of solver side