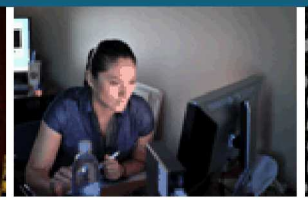
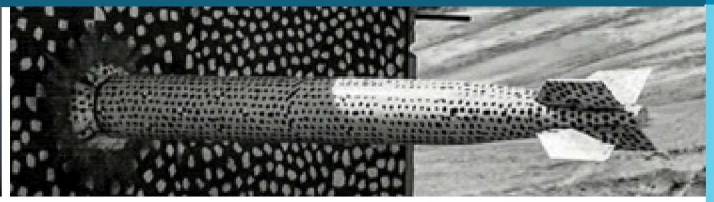
SAND2019-0453C

Rethinking the C++ / Python Boundary in Modeling and Optimization Tools





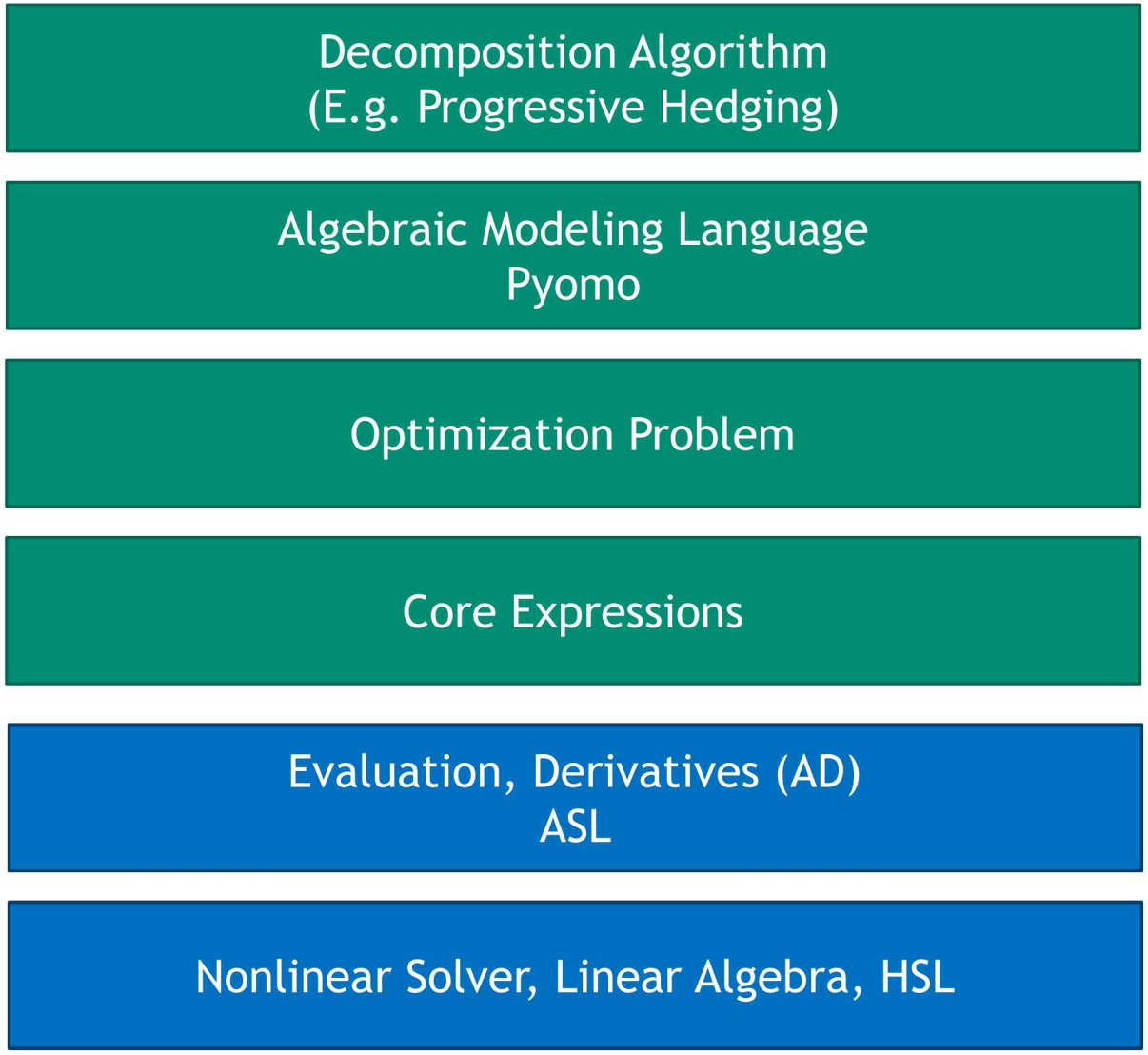
William E. Hart, Jose S. Rodriguez, John Siirola, Carl Laird

Purdue University
Sandia National Laboratories

wehart@sandia.gov

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Rethinking the Software Stack within Pyomo



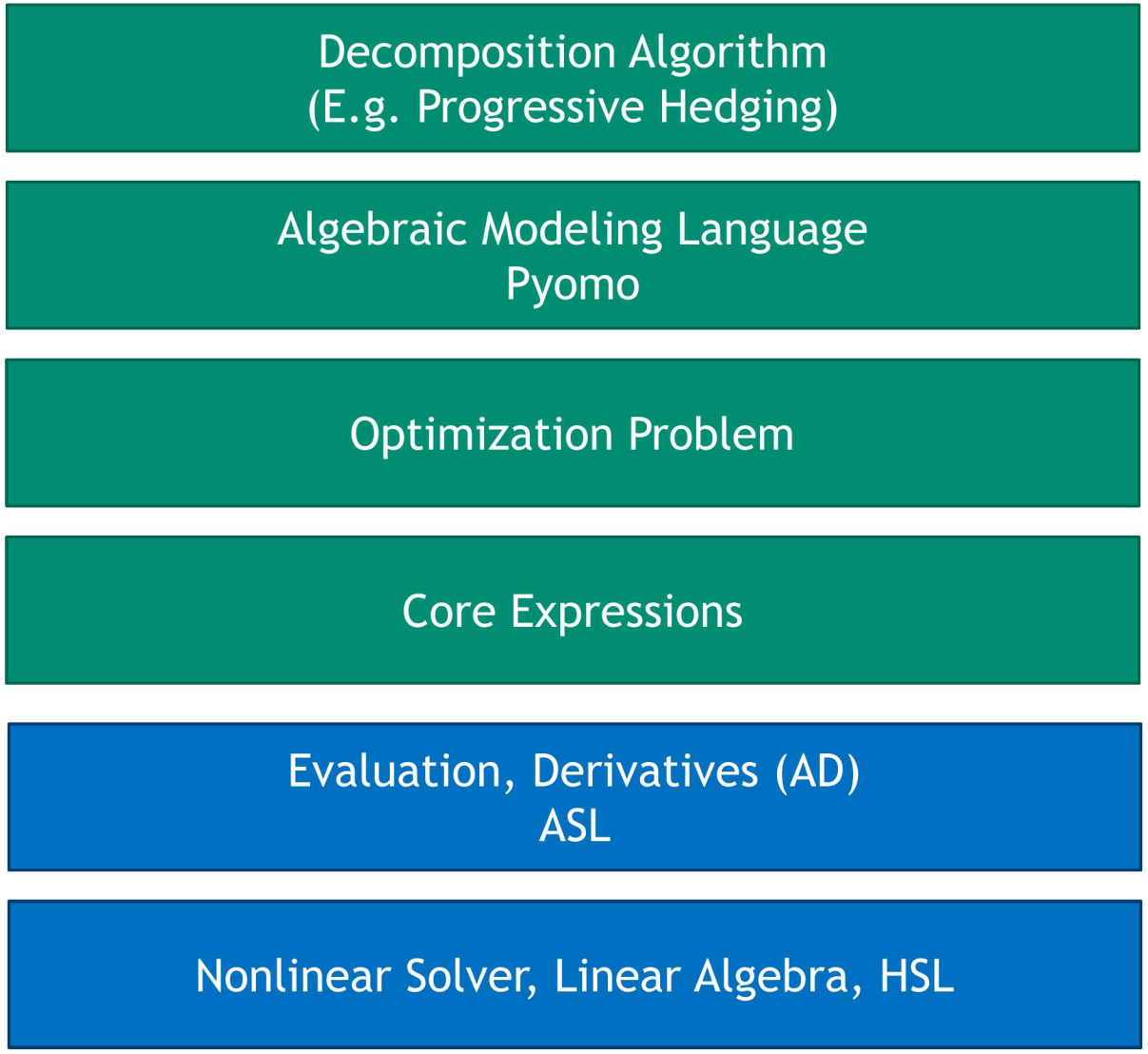
Have we struck the right balance?

Python Code

Boundary – (currently file-based)

Compiled
C / Fortran code

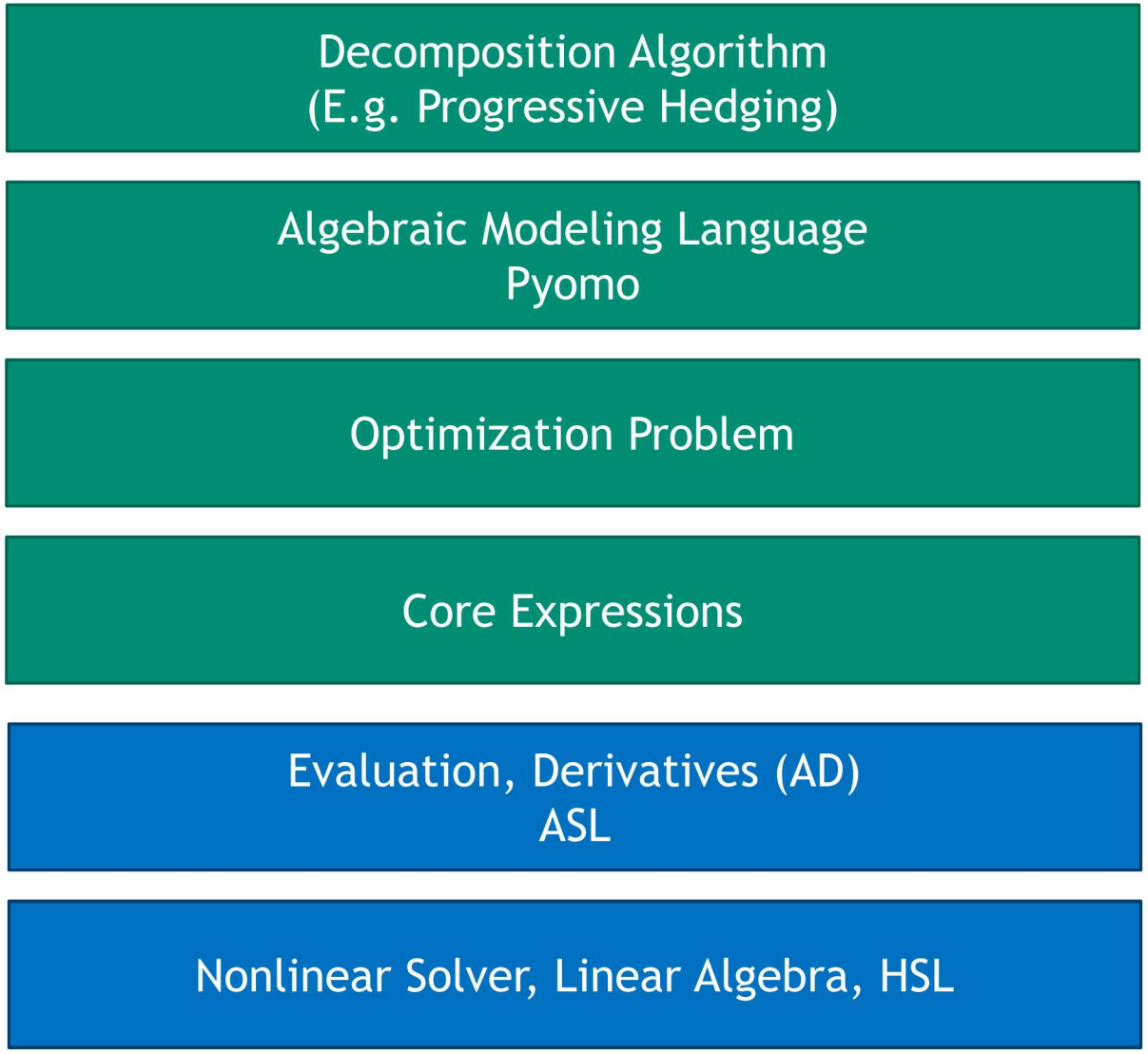
Rethinking the Software Stack within Pyomo



Have we struck the right balance?

- Concerns that Python is slow have motivated alternative AMLs (esp. JuMP).
- Pyomo meta-solvers written in Python have exhibited performance bottlenecks
- Callbacks to C/Fortran solvers cannot be used with file-based solver interfaces

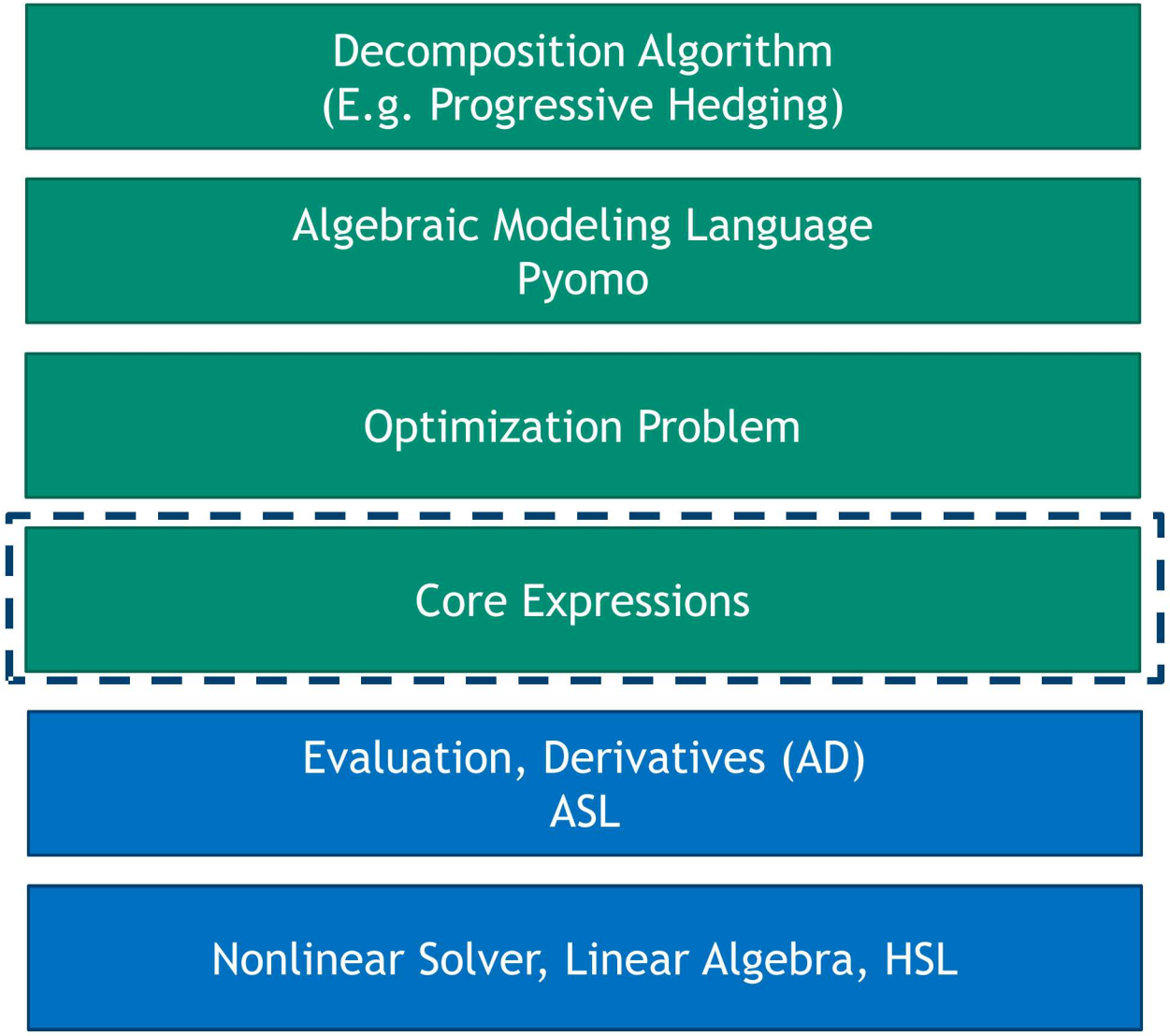
Rethinking the Software Stack within Pyomo



Three Recent Efforts

1. Reimplementation of Pyomo expressions to support PyPy
2. PyNumero: Python framework for efficient numerical algorithms
3. Poek/Coek: Lightweight Python expressions that enable direct solver interfaces

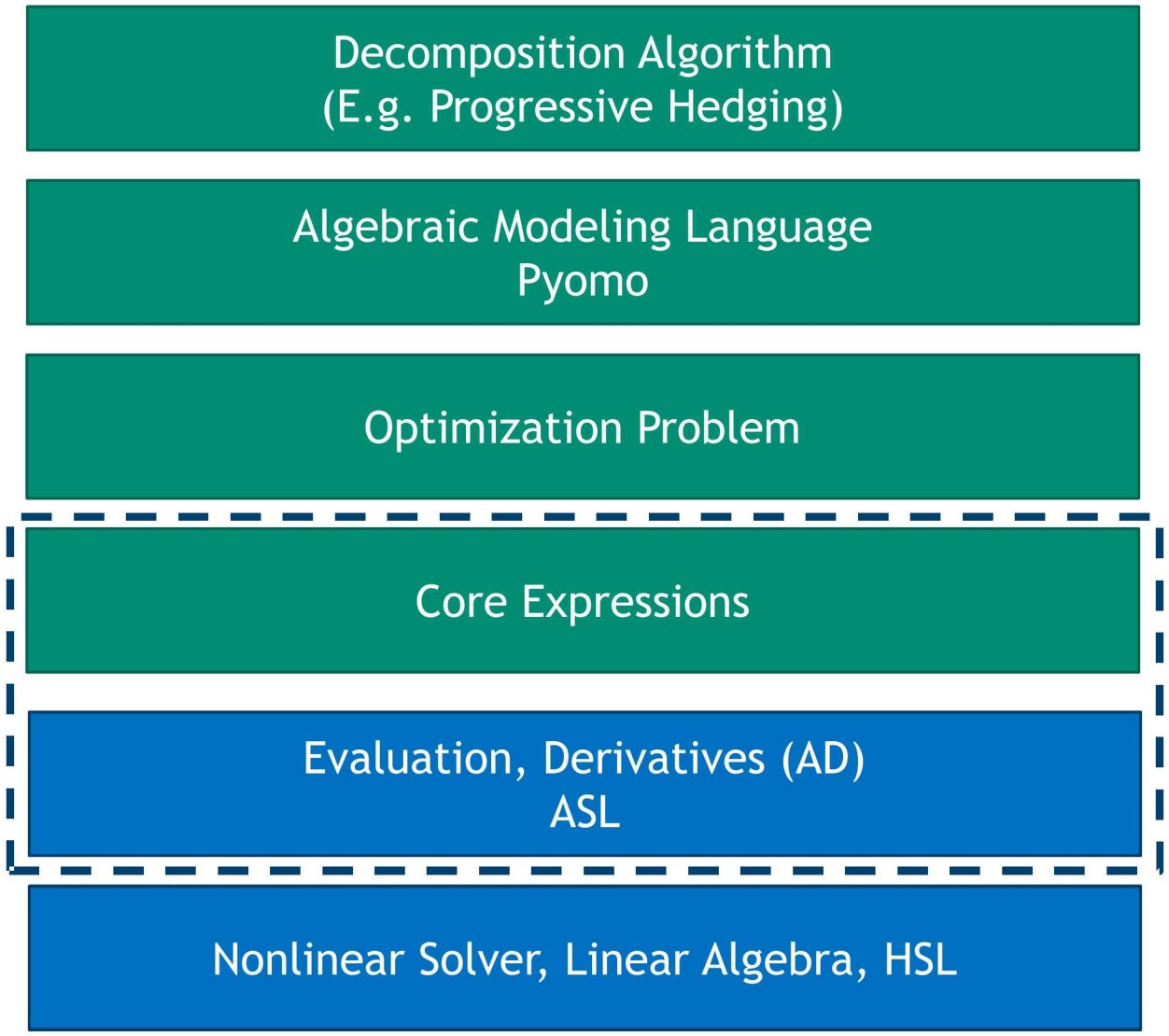
Rethinking the Software Stack within Pyomo



Three Recent Efforts

1. Reimplementation of Pyomo expressions to support PyPy
2. PyNumero: Python framework for efficient numerical algorithms
3. Poek/Coek: Lightweight Python expressions that enable direct solver interfaces

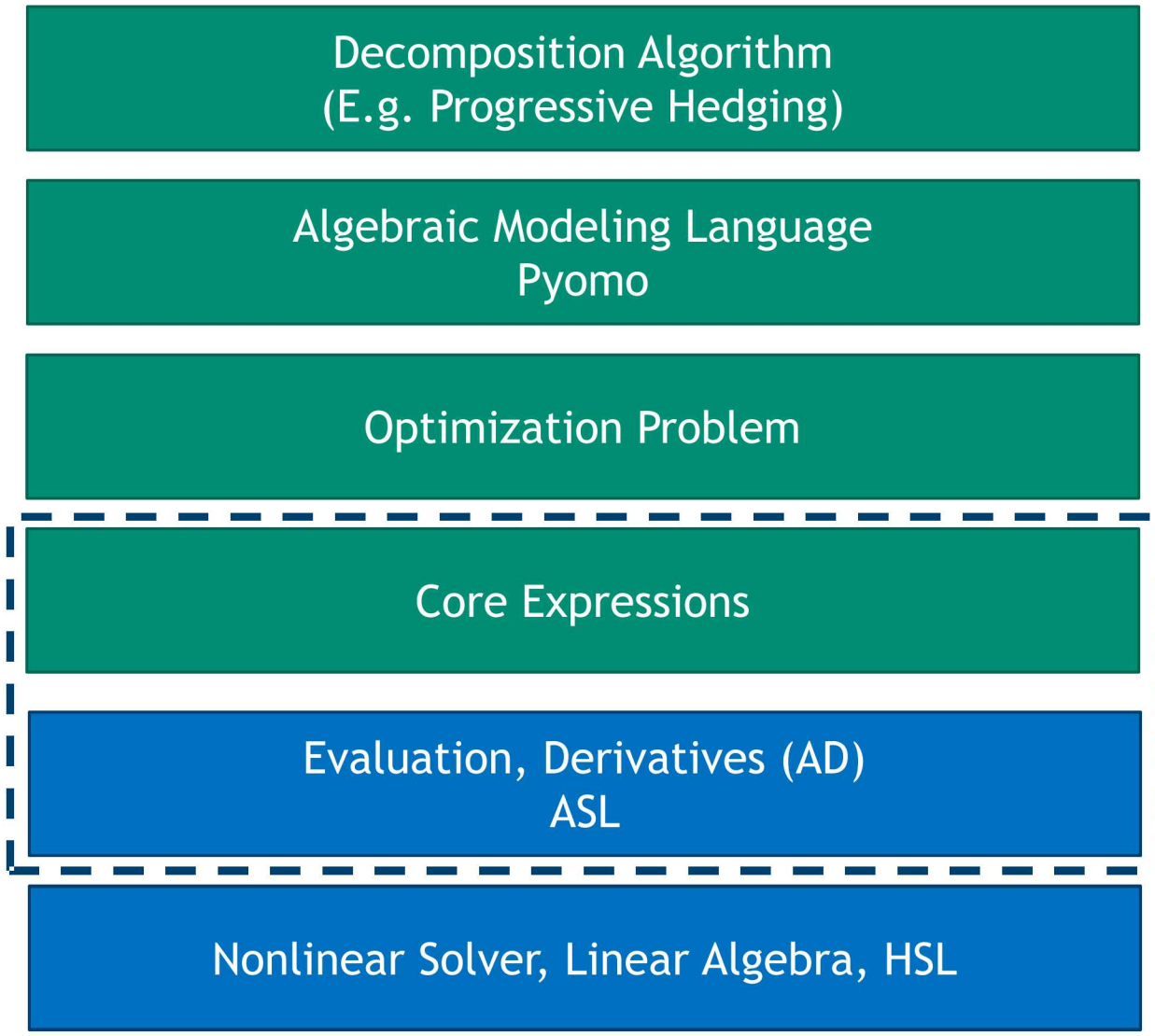
Rethinking the Software Stack within Pyomo



Three Recent Efforts

1. Reimplementation of Pyomo expressions to support PyPy
2. PyNumero: Python framework for efficient numerical algorithms
3. Poek/Coek: Lightweight Python expressions that enable direct solver interfaces

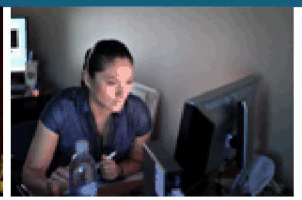
Rethinking the Software Stack within Pyomo

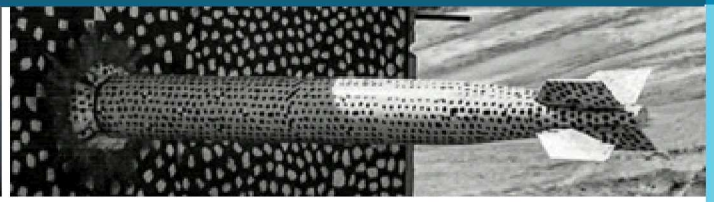


Three Recent Efforts

1. Reimplementation of Pyomo expressions to support PyPy
2. PyNumero: Python framework for efficient numerical algorithms
3. Poek/Coek: Lightweight Python expressions that enable direct solver interfaces

Support for the PyPy Python Interpreter

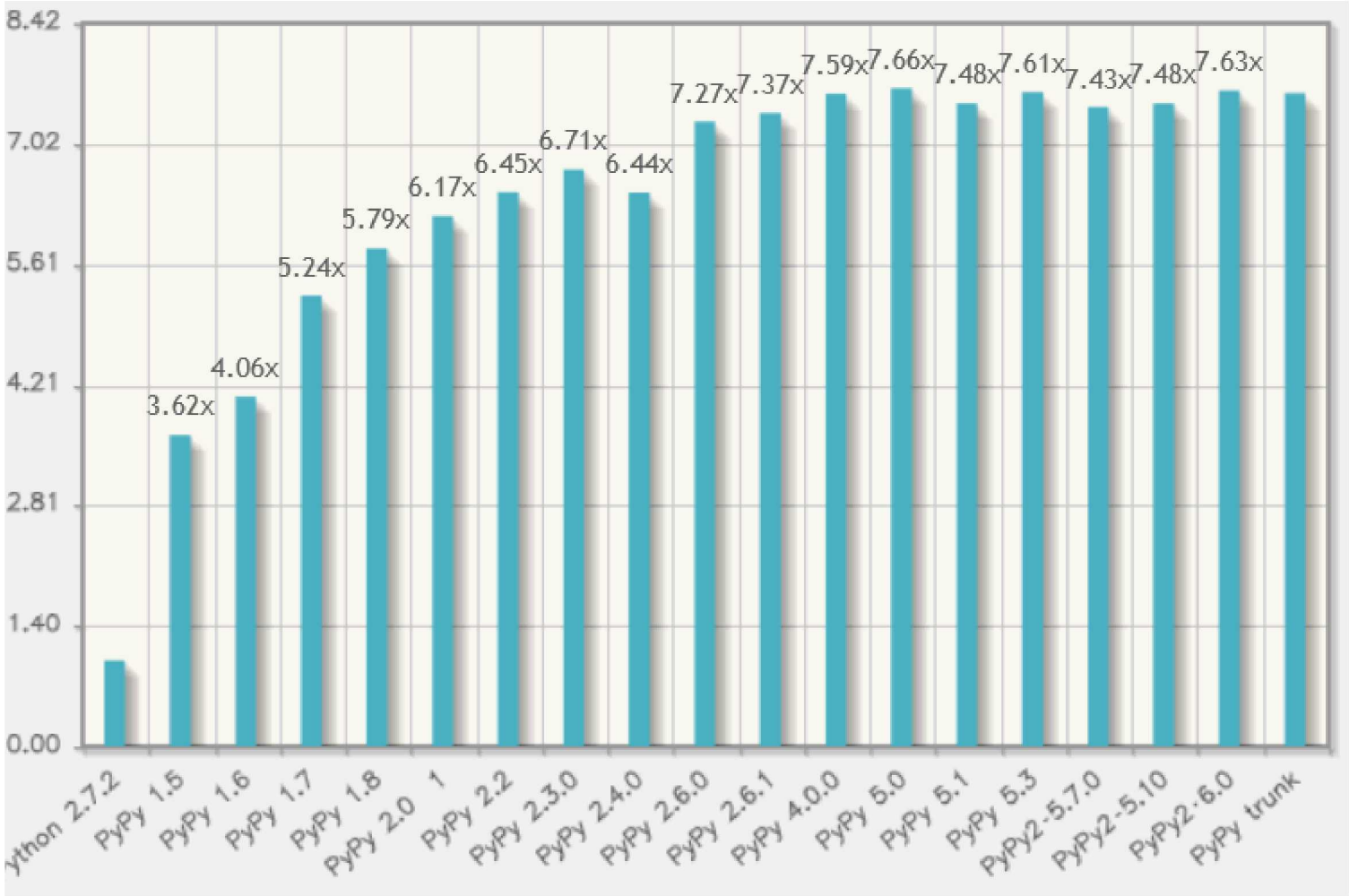




Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Motivations

1. PyPy is a faster Python implementation



Motivations

2. Pyomo 5.5 leverages recursion in many operations, including expression cloning.
   - Even simple non-linear expressions can result in deep expression trees where these recursive operations fail because Python runs out of stack space.

3. Unexpected performance bottlenecks arise due to implicit expression cloning

```
M = ConcreteModel()
M.x = Var(range(100))

# This loop is fast.
e = 0
for i in range(100):
    e = e + M.x[i]

# This loop is slow (?!?).
e = 0
for i in range(100):
    e = M.x[i] + e
```

```
M = ConcreteModel()
M.p = Param(initialize=3)
M.q = 1/M.p
M.x = Var(range(100))

# M.q is cloned every time it is used.
e = 0
for i in range(100):
    e = e + M.x[i]*M.q
```

Expression Immutability

Pyomo 5.5 expressions are mutable, so they can be modified in place.

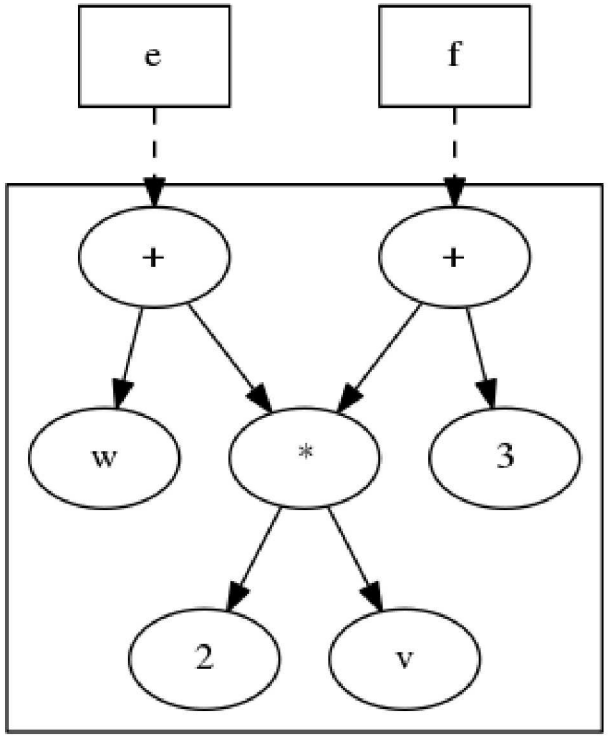
- Shared subtrees between expressions can lead to unexpected models.
- Pyomo 5.5 uses cloning and reference counts to detangle expressions.

Pyomo 5.6 ensures that expressions are immutable,
so detangling is not necessary!

```
M = ConcreteModel()
M.v = Var()
M.w = Var()

e = 2*M.v
f = e + 3

e += M.w
```



Testing Performance

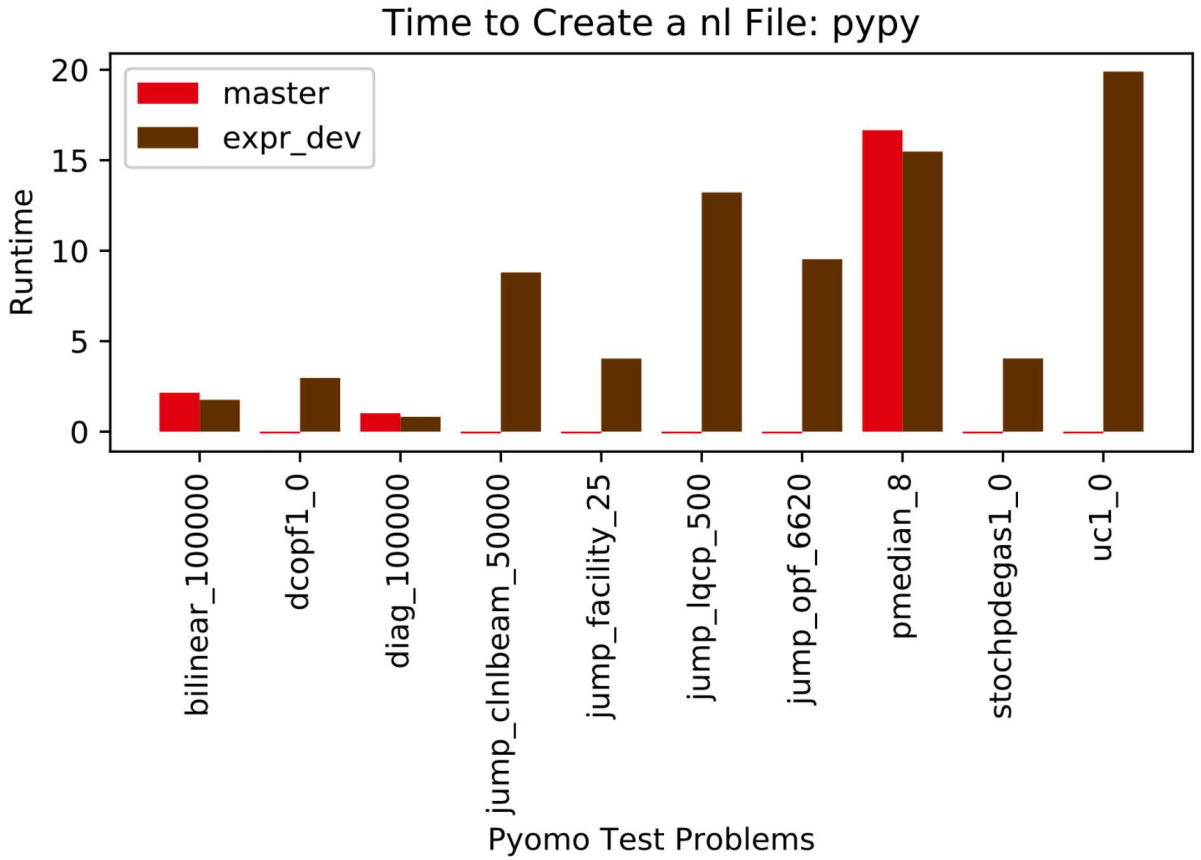
There are at least four elements of Pyomo's performance that could be impacted by the new expression system:

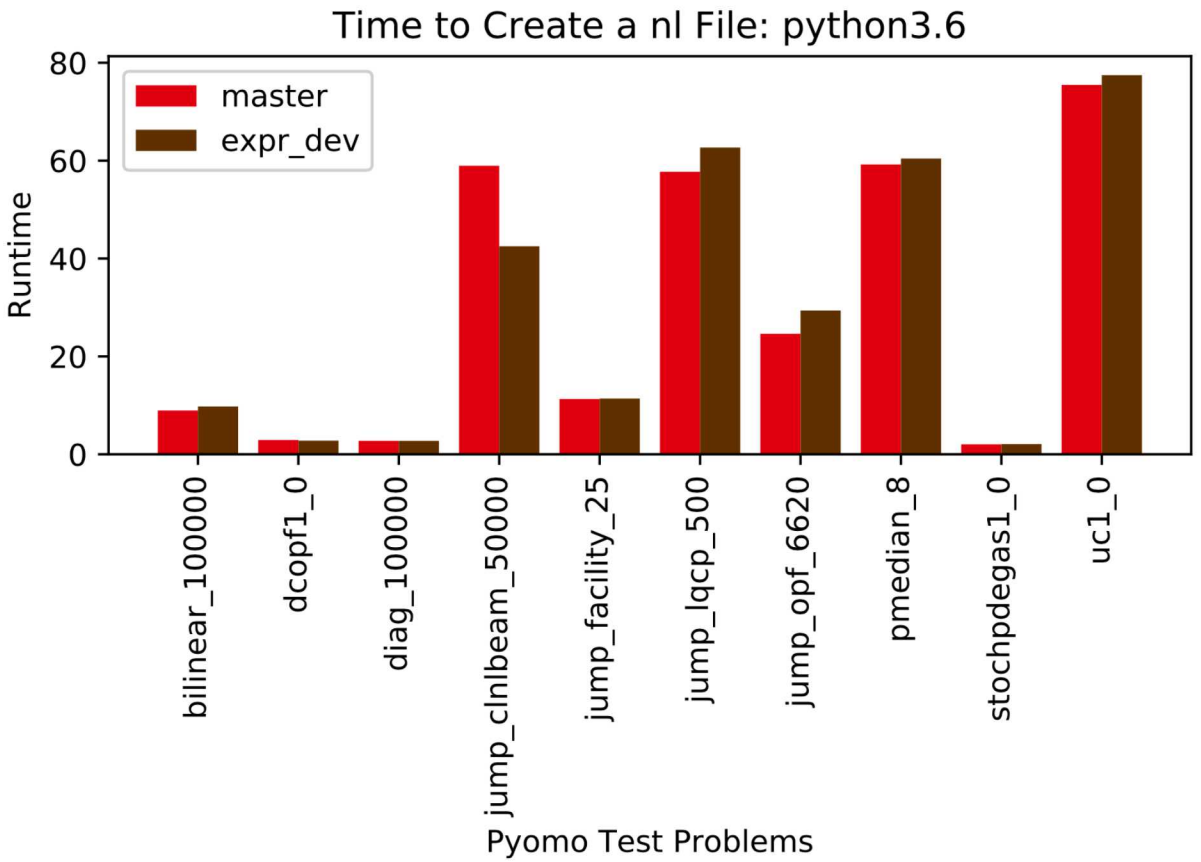
1. Time to generate expressions
2. Time to generate the standard representation used for sending problems to a solver
3. Time to write problem files
4. Time to perform optimization

Our tests measured the time to construct and write models (1+2+3).

We expect that (3) will not be a factor for NL files, since the problem representation is effectively unchanged.

Performance Comparison: Writing NL Files





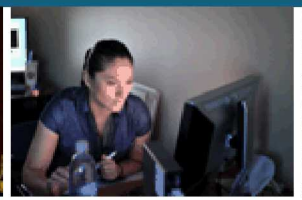
Summary

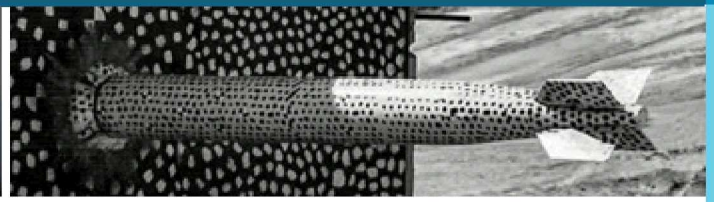
Pyomo now runs using PyPy

The median performance of the new expressions is 4% slower in CPython and 2.5x faster in PyPy (relative to CPython)

All cloning operations were eliminated, which eliminates a key source of “unexpected” performance bottlenecks

PyNumero: Python Numerical Optimization





Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Motivation

Pyomo is a Python-based, open-source optimization modeling language with a diverse set of optimization capabilities

It is built upon Python – a full programming language that allows Pyomo to be used in solutions ranging from simple scripting to high-level domain specific tools and meta-algorithms.

- It is easy to build ***on top*** of Pyomo

Low-level solvers are implemented in compiled languages and interfaced with Pyomo

They are not aware of Pyomo models and structure

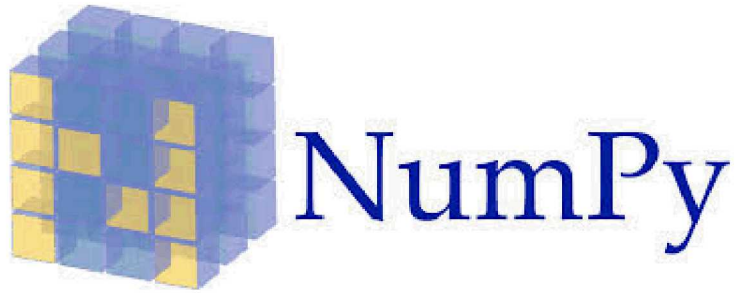
- Maybe with the exception of suffixes

Pyomo has not supported (**until now**) a framework for low-level numerical treatment of Pyomo models

- Provide efficient numerical derivatives in “Pyomo” format (e.g., outer-approximation)
- New numerical methods (e.g., NLP algorithms, sensitivity, trust-region surrogate methods)
- Implementation of decomposition approaches that are natively aware of Pyomo model structure

What is PyNumero?

PyNumero: A high-level python framework focused on compatibility with NumPy/SciPy and Pyomo for rapid development of nonlinear algorithms without large sacrifices on computational performance.



Very similar goals to NLPy (provide high-level interface for building NLP algorithms)

- Focused on compatibility with NumPy/SciPy and Pyomo

Impact:

- Dramatically reduce time required to prototype new serial and parallel numerical methods while minimizing the performance penalty
- Supports projects with novel numerical algorithm needs

Nonlinear Optimization Algorithms

Formulation

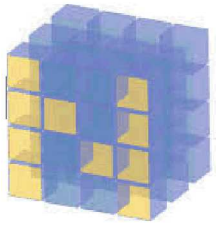
$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & c(x) = 0 \\ & d_L \leq d(x) \leq d_U \\ & x_L \leq x \leq x_U \end{aligned}$$

PYOMO

$f(x)$ Objective
x Variables
$c(x)$ Equalities
$d(x)$ Inequalities

Solution

$$\begin{aligned} \nabla_x \mathcal{L}(x^*, y_c^*, y_d^*, z_l^*, z_u^*, v_l^*, v_u^*) &= 0 \\ c(x^*) &= 0 \\ d_L \leq d(x^*) &\leq d_U \\ \vdots & \end{aligned}$$

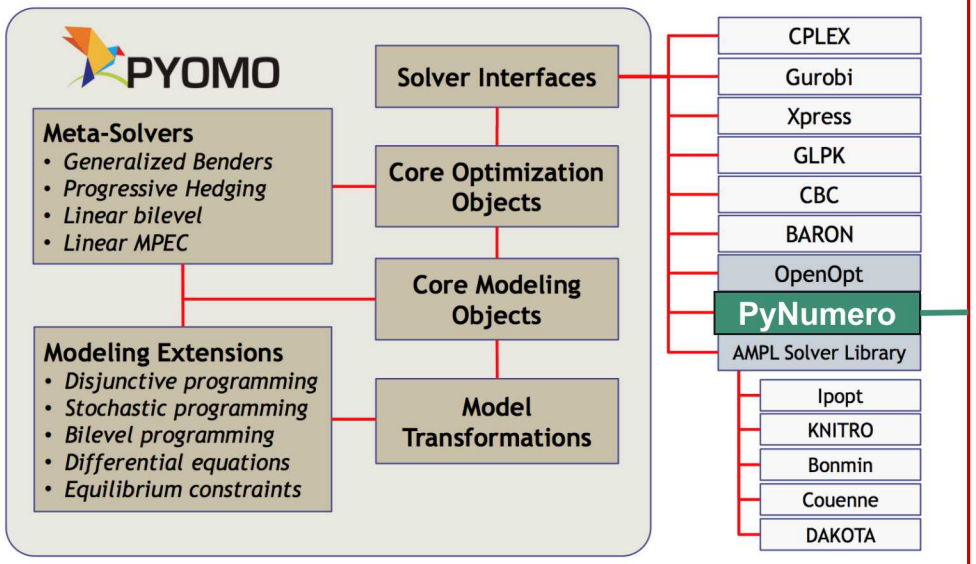


NumPy ASL

Derivatives $\nabla_x f(x), J_c(x) \ldots$
Sparse Algebra $\nabla^2_{xx}\mathcal{L}$
Solution of sparse systems

PyNumero and Pyomo

Idea: PyNumero provides a new solver interface that can interact with models very efficiently




```
from pyomo.contrib.pynumero.interfaces import PyomoNLP
import pyomo.environ as aml

m = aml.ConcreteModel()
m.x = aml.Var([1, 2, 3], bounds=(0.0, None))
m.phys = aml.Constraint(expr=m.x[3]**2 + m.x[1] == 25)
m.rsrc = aml.Constraint(expr=m.x[2]**2 + m.x[1] <= 18.0)
m.obj = aml.Objective(expr=m.x[1]**4-m.x[3]*m.x[2]**3)

def my_algorithm(model):
    nlp = PyomoNLP(model)
    x = nlp.create_vector_x()
    c = nlp.evaluate_c(x)
    Jc = nlp.jacobian_c(x)
    …
```

PYOMO

NumPy

Structure and Decomposition

Optimization with inherent structure is ubiquitous in engineering applications

- Stochastic programming
- Dynamic optimization
- Network optimization problems
- PDE optimization

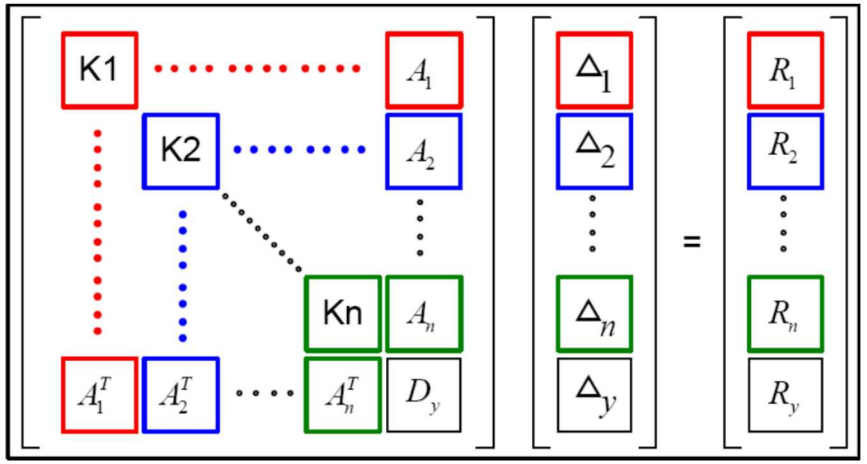
$$\min_{x_i, z} \quad \sum_{i \in \mathcal{P}} f_i(x_i)$$

$$\text{s.t.} \quad c_i(x_i) \geq 0, \ i \in \mathcal{P}$$

$$A_i x_i + B_i z = 0, \ (y_i) \quad i \in \mathcal{P}$$

Decomposition approaches allow for parallelization

- Internal decomposition
  - Schur-complement decomposition
  - Cyclic reduction
- External decomposition
  - Alternating direction method of multipliers (ADMM)
  - Progressive Hedging (PH)



PyNumero provides data structures to develop external and internal decomposition approaches

- BlockMatrix, BlockVector
- mpi4py

Summary

PyNumero is a flexible framework for prototyping and developing NLP algorithms in Python.

- Basic interior-point implemented in 3 weeks solved 200 Cuter tests.

PyNumero exploits the Numpy ecosystem and C++ python extensions to achieve good performance.

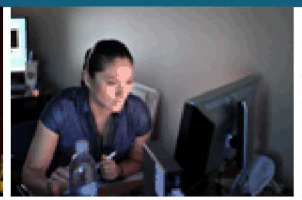
- Python interior-point implementation only 1.5 times slower than ipopt on a 100K variables/constraints problem
- Supports python calls to efficient linear solvers (e.g. MA27 and Mumps)
- Access to ASL from python
- Efficient interface to CyIpopt

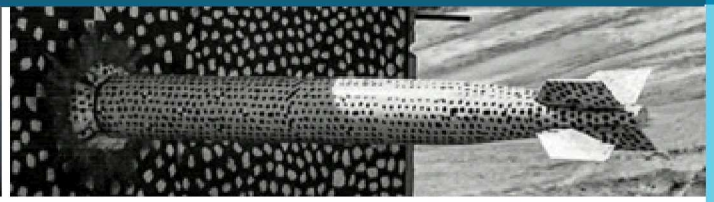
PyNumero facilitates research of decomposition algorithms.

- Schur-complement decomposition
- Alternating direction method of multipliers
- Progressive hedging

PyNumero is distributed with Pyomo and conda-forge.

POEK: A Python Optimize Expression Kernel





Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

IDEA/Motivation

Idea: Pyomo expression kernels could be executed in C++

- COEK – C++ expression library that interfaces to AD
- POEK – Python expression library that interfaces to COEK
  - Overload operators on expressions objects: variables, constants, expressions
  - Each operator call results in a Python-C call to create a new expression

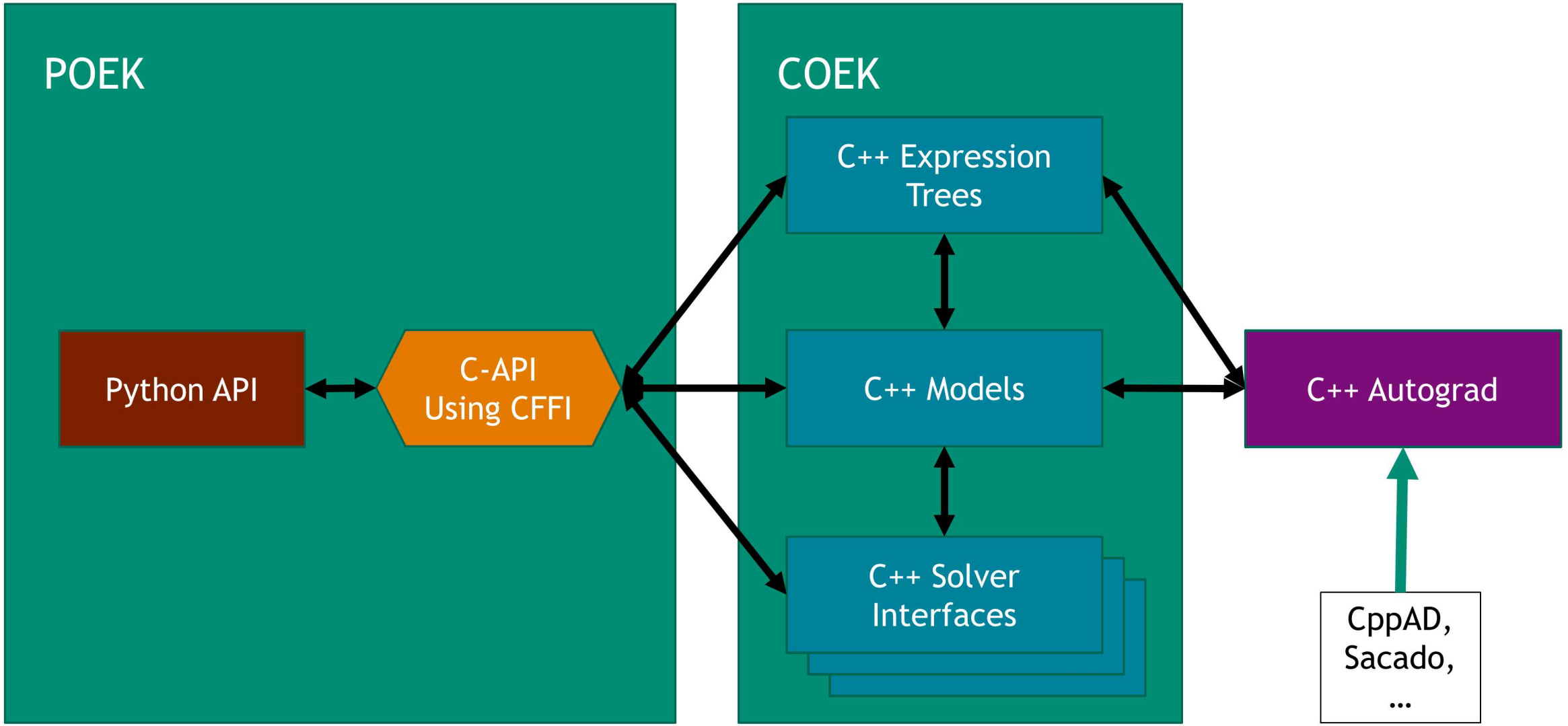
There are several obvious performance wins

- Create fewer Python objects for expressions
- Avoid creating Python expression objects with long lifetimes (which will help with memory)
- Avoid creating canonical expression representations
- Avoid file I/O (including expensive string manipulation for floating point numbers)

Many Performant Python frameworks move compute-intensive kernels into C

- Numpy/Pandas
- Tensorflow

Conceptual Design of POEK and COEK



P-Median Model: POEK

```
x = {}
for n in range(N):
  for m in range(M):
    x[n,m] = variable(lb=0, ub=1, initialize=0)
y = variable(N, lb=0, ub=1, initialize=0)
d = {}
for n in range(N):
  for m in range(M):
    d[n,m] = random.uniform(0.0,1.0)

pmedian = model()

# objective
pmedian.add( sum(d[n,m]*x[n,m] for n in range(N) for m in range(M)) )

# single_x
for m in range(M):
  pmedian.add( sum(x[n,m] for n in range(N)) == 1 )

# bound_y
for n in range(N):
  for m in range(M):
    pmedian.add( x[n,m] - y[n] <= 0 )

# num_facilities
pmedian.add( sum(y[n] for n in range(N)) == P )
```

P-Median Model: Pyomo

```
model = ConcreteModel()

model.N = RangeSet(N)
model.M = RangeSet(M)
model.x = Var(model.N, model.M, bounds=(0,1), initialize=0)
model.y = Var(model.N, bounds=(0,1), initialize=0)
model.d = Param(model.N, model.M,
                initialize=lambda  n, m, model : random.uniform(1.0,2.0))

def rule(model):
        return sum(model.d[n,m]*model.x[n,m] for n in model.N for m in model.M)
model.obj = Objective(rule=rule)

def rule(model, m):
        return sum(model.x[n,m] for n in model.N) ==  1.0
model.single_x = Constraint(model.M, rule=rule)

def rule(model, n,m):
        return model.x[n,m] - model.y[n] <= 0.0
model.bound_y = Constraint(model.N, model.M, rule=rule)

def rule(model):
        return sum(model.y[n] for n in model.N) == P
model.num_facilities = Constraint(rule=rule)
```

Preliminary Performance Results

Time to construct model and setup Ipopt

- P-median: N=M=640, P=1
- Cpython 3.6

	Pyomo	POEK	Speedup
Build Model	22.7	10.3	2.2x
Setup Ipopt	44.6	6.8	6.6x
TOTAL	67.3	17.1	3.9x

Observations & Conclusions

- CFFI interface is fast enough to justify many Python-C calls when constructing expressions
- Eliminating expression translation and file I/O in NL writer is a big win (NL files)
- Matrix/Vector expressions would make model build faster
- C++ expressions can be interrogated from Python using callbacks

FINAL THOUGHTS

Faster Pyomo Expressions

- PyPy provides a nontrivial speedup over Cpython
- However, PyPy is not commonly used
  - NOTE: PyPy can now be easily installed with Conda

PyNumero

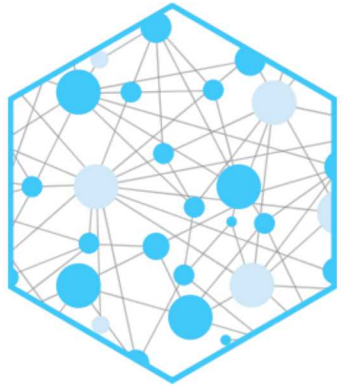
- PyNumero's hybrid strategy is well-established in the Python community
- Is distributed with Pyomo and conda-forge making it easy to install and use
- Extends Pyomo to build hybrid solvers using python and C/C++

Poek/Coek

- This is a proof-of-concept
- But Poek already does a lot of the basics
- NOTE: Coek looks a lot like the AMPL Solver Library

Acknowledgements

- This work was conducted as part of the Institute for the Design of Advanced Energy Systems (IDAES) with funding from the Office of Fossil Energy, Cross-Cutting Research, U.S. Department of Energy



IDAES
Institute for the Design of Advanced Energy Systems

NETL

Berkeley Lab

Sandia National Laboratories

Carnegie Mellon
West Virginia University

Questions?

Test Problems

Pyomo/POEK was tested with the following test problems:

Problem	Description
bilinear_100000	A model with large bilinear expressions
dcopf1_0	A DC OPF power grid model
diag_100000	A large diagonal model
jump_clnlbeam_50000	The JuMP clnlbeam test problem
jump_facility_25	The JuMP facility test problem
jump_lqcp_500	The JuMP lqcp test problem
jump_opf_6620	The JuMP opf test problem
pmedian_8	A large, dense p-median test problem
stochpdegas1_0	A large dynamic optimization problem
uc1_0	A unit commitment model