Extending JuMP and MathOptInterface

Benoît Legat (UCLouvain)

August 7, 2019

Extending MathOptInterface

Dualization extension

Extending JuMP

Polyhedra extension

Sum-of-Squares extension

Set Programming extension

ParameterJuMP extension

StructJuMP extension

MomentOpt extension

Extending MathOptInterface

MathOptInterface (MOI)

- `add_variable(model)`, `add_constrained_variable(model, set)`.
- `add_constraint(model, func, set)`, e.g. $2x + 3y = 1 \rightarrow$ `(2*x + 3*y)-in-EqualTo(1.0)`.
- `set`, `get` attributes, e.g., `ObjectiveFunction`.
- `submit` submittables, e.g., `HeuristicSolution`.

Extensible framework:

- Generic on attribute, function and set types. New ones can be defined independently.
- Solver-specific features easily exposed to JuMP/MOI users through custom attributes, e.g. solver specific callbacks.
- Expose specialized problem structure easily through custom functions, sets (e.g. Sum-of-Squares variables/constraints).

Semidefinite programming

$$\begin{array}{lll} \underset{Q \in \mathcal{S}^n}{\text{minimize}} & \langle C, Q \rangle & \underset{y \in \mathbb{R}^n}{\text{maximize}} \quad \langle b, y \rangle \\ \text{subject to} & \langle A_i, Q \rangle = b_i & \text{subject to} \quad \sum_i A_i y_i \preceq C \\ & Q \succeq 0 & \end{array}$$

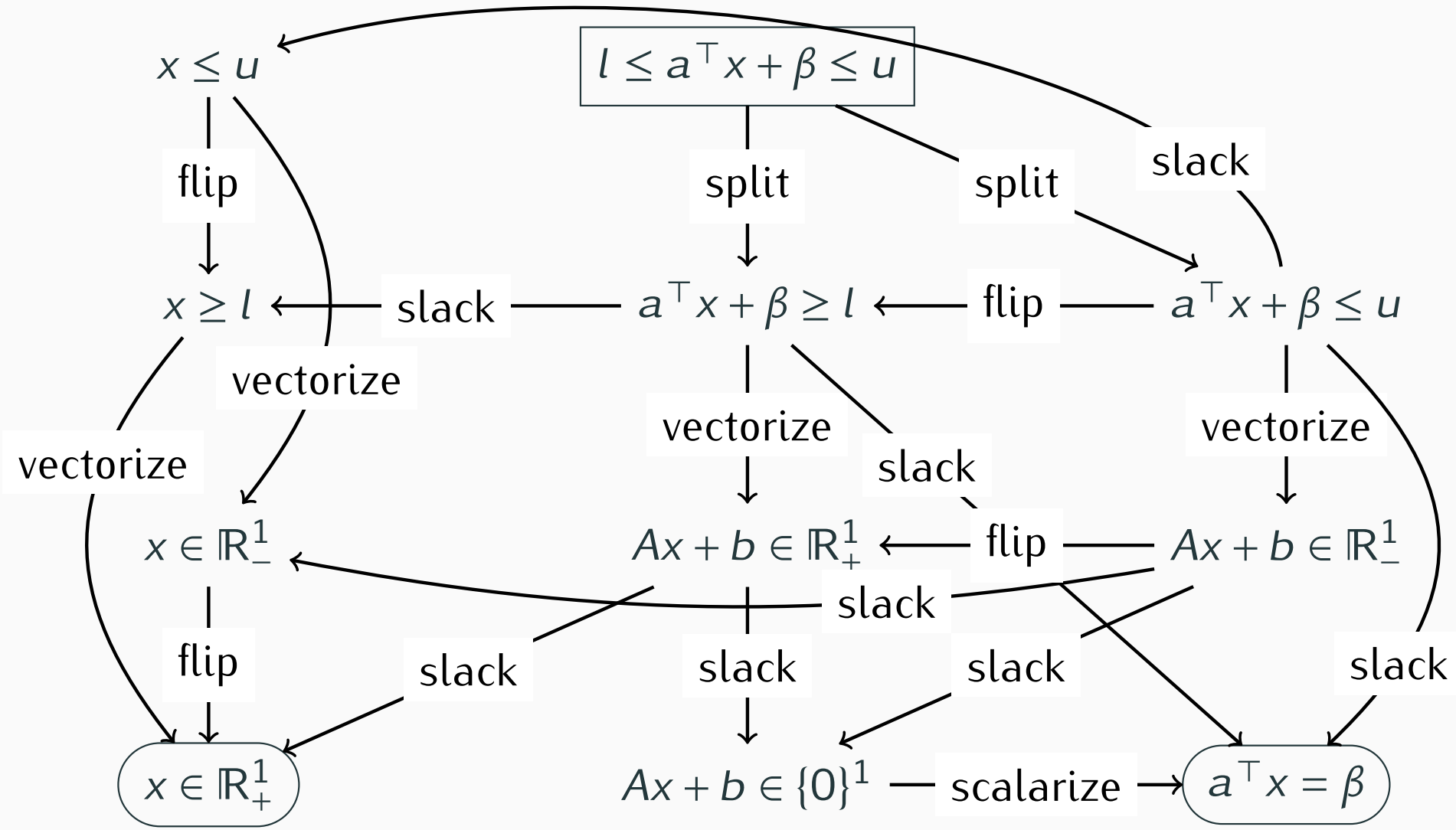
File format: SDPA

Solvers: CSDP, SDPA, DSDP, SDPLR, ...

Variables: Q block diagonal, nonnegative scalar variables (1×1 blocks) or SDP matrices.

Constraints: Affine equations.

Interval constraint for SDPA



Bridges directed hypergraph

Nodes
Node for each set S (variable-in-S).

Node for each constraint F-in-S.

Node for each objective function F.

Edges
For each edge and ingoing node: outgoing nodes are

- variable-in-S created.
- constraints F-in-S created.
- objective F set.

Bridges selection

Multiple dispatch ?
Can we use multiple dispatch where node types are method signatures ?

No. Several bridge can transform same types. Dispatching on more specialized is not accurate.

Need to guide choice to supported types.

Just in Time (JIT) bridge selection
Types F and S are not limited to those defined in MOI $\rightarrow$ infinitely many nodes.

Each bridge defined possible infinitely many edges.

Need to be lazy and compute bridge selection JIT when type first used.

Bridge selection algorithm

Data: Set of bridges B

Invariant: $d_{\mathcal{V}}$, $d_{\mathcal{C}}$, $d_{\mathcal{O}}$ are correct when defined.

Given new type: compute types T where d should be defined with Depth First Search (DFS) from the type: $\mathcal{O}(|B|)$.

Compute d at the types using Bellman–Ford algorithm: $\mathcal{O}(|T| \cdot |B|)$.

Dualization extension

Extends MOI by creating a new layer.

Problem given to GLPK is the dual of problem given to optimizer.

```
optimizer = Dualization.DualOptimizer(GLPK.Optimizer())
# or
model = Model(with_optimizer(() ->
    Dualization.DualOptimizer(GLPK.Optimizer())))
```

Primal (resp. dual) results of optimizer are dual (resp. primal) results of GLPK.

```
struct DualOptimizer{T, OT <: MOI.ModelLike} <:
        MOI.AbstractOptimizer
    dual_problem::DualProblem{T, OT}
end
struct DualProblem{T, OT <: MOI.ModelLike}
    dual_model::OT
    primal_dual_map::PrimalDualMap{T}
end
function MOI.copy_to(dest::DualOptimizer,
                     src::MOI.ModelLike; kws...)
    dualize(src, dest.dual_problem)
    # ...
end
```

Extending JuMP

Extending JuMP macros

```
@constraint(model, [x + 1, x - y] in MOI.Zeros())
```

Implementation:

```
function build_constraint(
    _error::Function,
    func::Vector, set::MOI.AbstractVectorSet)
    return VectorConstraint(x, set)
end
function add_constraint(model, c::AbstractConstraint,
                        name)
    index = MOI.add_constraint(backend(model),
        moi_function(c), moi_set(c))
    ... # set name and shape
    return ConstraintRef(model, index, shape(c))
end
```

Extending JuMP macros: Custom set

```
@constraint(model, [x + 1, x - y] in SecondOrderCone())
```

Implementation:

```
function build_constraint(_error::Function,
                          f::AbstractVector,
                          s::AbstractVectorSet)
    set = moi_set(s, length(f))
    return build_constraint(_error, f, set)
end
function moi_set(::SecondOrderCone, dim::Int)
  return MOI.SecondOrderCone(dim)
end
```

Extending JuMP macros: PSD cone

```
using LinearAlgebra # For Symmetric
@constraint(model, Symmetric([x + 1 x - y
                              x - y y]) in PSDCone())
```

Implementation:

```
function build_constraint(_error::Function,
                          Q::Symmetric,
                          ::PSDCone)
    n = LinearAlgebra.checksquare(Q)
    func = [Q[i, j] for j in 1:n for i in 1:j]
    set = MOI.PositiveSemidefiniteConeTriangle(n)
    VectorConstraint(func, set,
                     SymmetricMatrixShape(n))
end
```

Adding bridges

```
struct BridgeableConstraint{C, B} <: AbstractConstraint
    constraint::C
    bridge_type::B
end

function add_constraint(model::Model,
                        c::BridgeableConstraint,
                        name::String="")
    add_bridge(model, c.bridge_type)
    return add_constraint(model, c.constraint, name)
end
```

Polyhedra extension

Polyhedra example

Creating the hypercube from JuMP:

```
model = Model()
@variable(model, 0 <= x[1:n] <= 1)

poly = polyhedron(model, CDDLib.Library(:exact))
```

Polyhedra constraint:

```
simplex = intersect(
    HalfSpace([-1, 0], 0), HalfSpace([0, -1], 0),
    HyperPlane([1, 1], 1))
model = Model()
@variable(model, x[1:2])
@constraint(model, A * x in simplex)
```

JuMP model to polyhedron

```
MOI.Utilities.@model(_MOIModel,
    (), (MOI.EqualTo, MOI.LessThan,), (), (),
    (), (MOI.ScalarAffineFunction,), (), ())
mutable struct LPHRep{T} <: HRepresentation{T}
    model::_MOIModel{T}
    # Cached list of indices
    hyperplane_indices::Union{
        Nothing, Vector{MOI.ConstraintIndex{
          MOI.ScalarAffineFunction{T}, MOI.EqualTo{T}}}}
    hyperplane_indices::Union{
        Nothing, Vector{MOI.ConstraintIndex{
          MOI.ScalarAffineFunction{T}, MOI.LessThan{T}}}}
end
```

Polyhedra constraint: Extending MOI

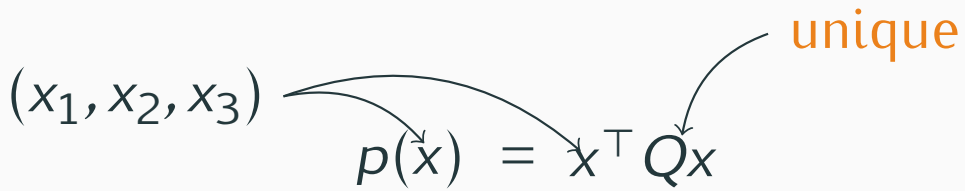
```
struct PolyhedraOptSet{T, RepT <: Rep{T}} <:
    MOI.AbstractVectorSet
    rep::RepT
end
struct PolyhedraToLPBridge{T, F} <:
    MOI.Bridges.AbstractBridge
    hyperplanes::Vector{MOI.ConstraintIndex{
        F, MOI.EqualTo{T}}}
    halfspaces::Vector{MOI.ConstraintIndex{
        F, MOI.LessThan{T}}}
end
```

Polyhedra constraint: Extending JuMP

```
function JuMP.build_constraint(
    error_fun::Function, func, set::Polyhedra.Rep)
    return JuMP.BridgeableConstraint(
        JuMP.build_constraint(
            error_fun, func, PolyhedraOptSet(set)),
        PolyhedraToLPBridge)
end
```

Sum-of-Squares extension

Nonnegative quadratic forms into sum of squares



$$x_1^2 + 2x_1x_2 + 5x_2^2 + 4x_2x_3 + x_3^2 = x^\top \begin{pmatrix} 1 & 1 & 0 \\ 1 & 5 & 2 \\ 0 & 2 & 1 \end{pmatrix} x$$

$$p(x) \geq 0 \ \forall x \iff Q \succeq 0 \qquad \Big\downarrow \text{ cholesky}$$

$$(x_1 + x_2)^2 + (2x_2 + x_3)^2 \longleftarrow x^\top \begin{pmatrix} 1 & 1 & 0 \\ 0 & 2 & 1 \end{pmatrix}^\top \begin{pmatrix} 1 & 1 & 0 \\ 0 & 2 & 1 \end{pmatrix} x$$

Nonnegative polynomial into sum of squares

(x_1, x_2, x_3) (x_1, x_1x_2, x_2) *not* unique

$$p(x) = X^\top Q X$$

$$x_1^2 + 2x_1^2x_2 + 5x_1^2x_2^2 + 4x_1x_2^2 + x_2^2 = X^\top \begin{pmatrix} 1 & 1 & 0 \\ 1 & 5 & 2 \\ 0 & 2 & 1 \end{pmatrix} X$$

$$p(x) \geq 0 \ \forall x \impliedby Q \succeq 0 \quad \Big\downarrow \text{cholesky}$$

$$(x_1 + x_1x_2)^2 + (2x_1x_2 + x_2)^2 \longleftarrow X^\top \begin{pmatrix} 1 & 1 & 0 \\ 0 & 2 & 1 \end{pmatrix}^\top \begin{pmatrix} 1 & 1 & 0 \\ 0 & 2 & 1 \end{pmatrix} X$$

When is nonnegativity equivalent to sum of squares ?

Determining whether a polynomial is nonnegative is NP-hard.

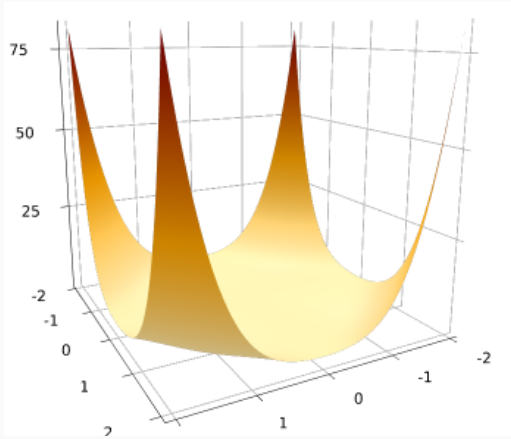
Hilbert 1888
Nonnegativity of $p(x)$ of n variables and degree $2d$ is equivalent to sum of squares in the following three cases:

- $n = 1$: Univariate polynomials
- $2d = 2$: Quadratic polynomials
- $n = 2$, $2d = 4$: Bivariate quartics

Motzkin 1967
First explicit example:

$$x_1^4 x_2^2 + x_1^2 x_2^4 + 1 - 3x_1^2 x_2^2 \geq 0 \quad \forall x$$

but is not a sum of squares.



Theory behind bridges

$$x \in S_1 \Leftrightarrow Ax \in S_2 \qquad AS_1 = S_2$$
$$A^* y \in S_1^* \Leftrightarrow y \in S_2^* \qquad S_1^* = A^* S_2^*$$

In Lagrangian:

$$\langle Ax, y \rangle_2 = \langle x, A^* y \rangle_1$$

Transformation of variable-in-S_2 to variable-in-S_1.

Primal Transform value v to Av.

Dual Transform dual y to $A^{-*}y$.

Transformation of f-in-S_1 constraint to Af-in-S_2 constraint.

Primal Transform value v of Af to $A^{-1}v$ of f.

Dual Transform dual y of $A^* y$.

Sum-of-Squares bridges

Polynomial $p \in \Sigma$ (p is SOS) iff $p = X^\top Q X$ with $Q \in \mathbb{S}_+$ (Q is PSD). Hence $\Sigma = A\mathbb{S}_+$.

SOSPolnomialBridge: Transformation of variable-in-Σ to variable-in-$\mathbb{S}_+$.

Transformation of contraint F-in-Σ: SlackBridge + SOSPolnomialBridge.

Result transformations

Constraint Attribute
Examples: ConstraintPrimal, ConstraintDual, ConstraintFunction, ConstraintSet, ...

Redirected to bridge when constraint is bridged.

New attributes:

- GramMatrixAttribute: Gram matrix Q indexed by X.
- MomentMatrixAttribute: Moment matrix index by X, dual of constraint $Q \in \mathbb{S}_+$.
- MomentsAttribute: Vector of moments, dual of constraint $p = X^\top Q X$.

Sum-of-Squares constraint macro

```
@constraint(model, p in SOSCone())
```

Implementation:

```
function JuMP.build_constraint(_error::Function, p,
                                cone::SOSCone; kws...)
    coefs = coefficients(p)
    monos = monomials(p)
    set = JuMP.moi_set(cone, monos; kws...)
    shape = PolyJuMP.PolynomialShape(monos)
    return PolyJuMP.bridgeable(
        JuMP.VectorConstraint(coefs, set, shape),
        JuMP.moi_function_type(typeof(coefs)),
        typeof(set)
    )
end
```

Set Programming extension

Example with ellipsoids: Model

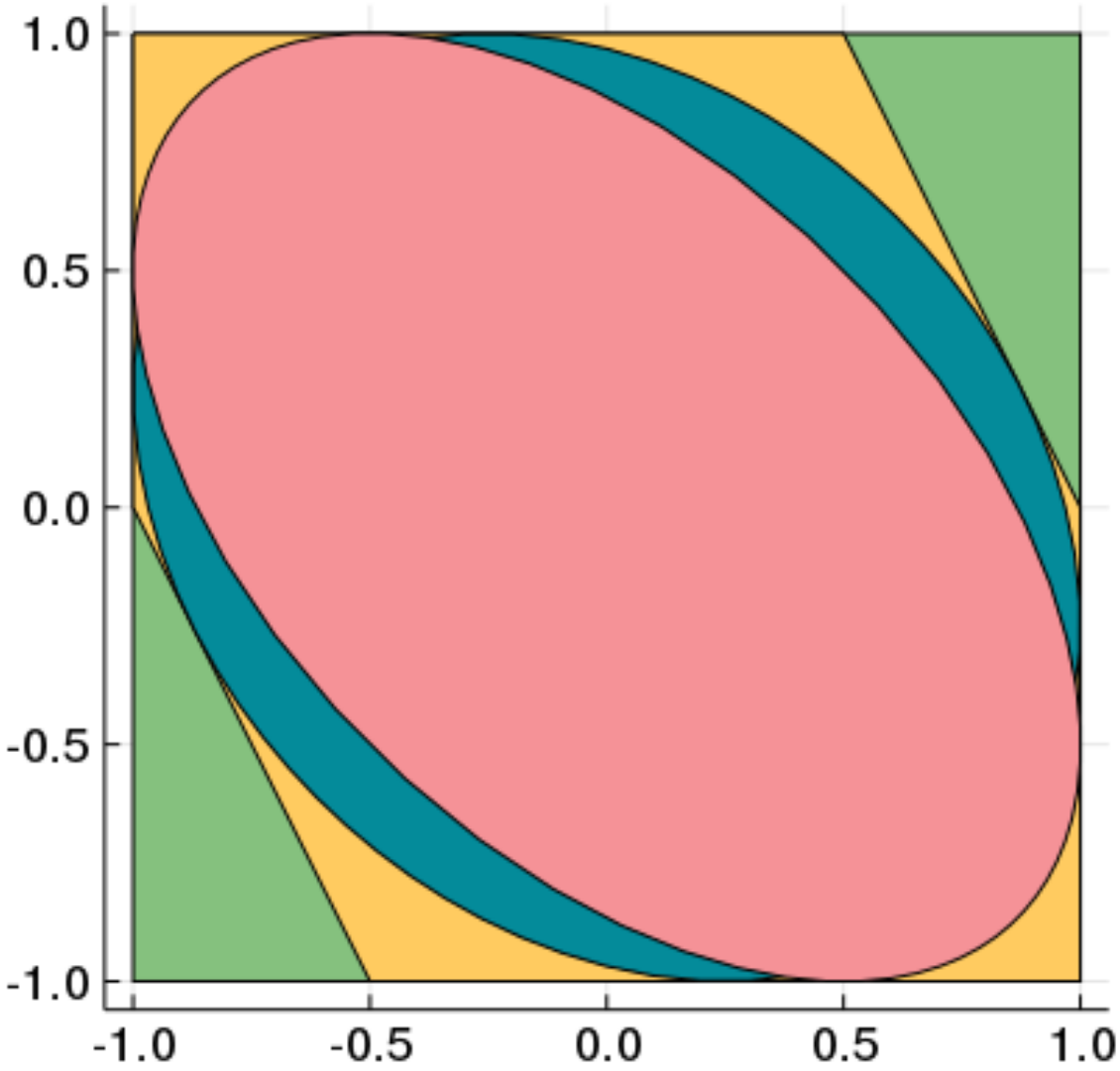
Maximal volume ellipsoid (determinant):

```
model = Model(...)
@variable(model, S, Ellipsoid(symmetric=true))
@constraint(model, S ⊆ □)
@constraint(model, A * S ⊆ E * S)
@objective(model, Max, nth_root(volume(S)))
@time JuMP.optimize!(model)
ell = JuMP.value(S)
```

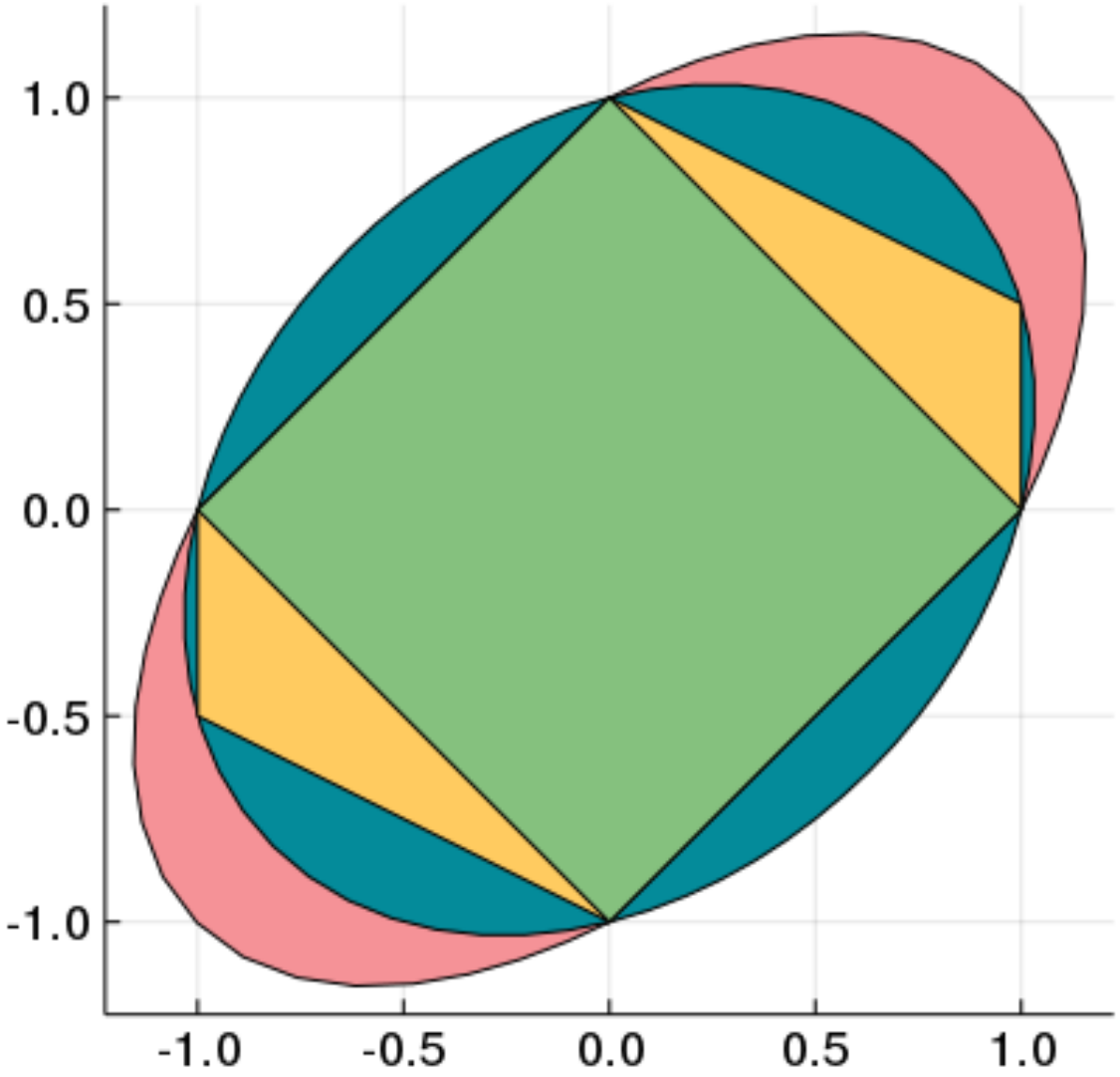
Maximal sum of the squares of the semi-axes of the ellipsoid (trace):

```
@objective(model, Max,
           L1_heuristic(volume(S), [1.0, 1.0]))
```

Example with ellipsoids: Primal Solution



Example with ellipsoids: Polar Solution



Set variables

```
function JuMP.build_variable(
    _error, info::JuMP.VariableInfo,
    set::AbstractSetVariable)
    if info.has_lb || info.has_ub || ...
        _error(...)
    end
    return set
end
function JuMP.add_variable(
    model::JuMP.AbstractModel, set::AbstractSetVariable,
    name::String)
    vref = SetVariableRef(...)
    push!(data(model).variables, vref)
    return vref
end
```

Parsing set constraints

```
function JuMP.parse_one_operator_constraint(_error::Function, vectorized::Bool,
                                            ::Val{:⊂}, lhs, rhs)
    _error("Unrecognized symbol ⊂ you mean ⊆ ?")
end
function JuMP.parse_one_operator_constraint(_error::Function, vectorized::Bool,
                                            ::Val{:⊆}, lhs, rhs)
    parse_code = :()
    if vectorized
        build_call = :(JuMP.build_constraint.($_error, $(esc(lhs)), $(esc(:(SetProg.PowerSet.($rhs))))))
    else
        build_call = :(JuMP.build_constraint($_error, $(esc(lhs)), $(esc(:(SetProg.PowerSet($rhs))))))
    end
    return parse_code, build_call
end
function JuMP.parse_one_operator_constraint(_error::Function, vectorized::Bool,
                                            ::Val{:⊃}, lhs, rhs)
    _error("Unrecognized symbol ⊃, did you mean ⊇ ?")
end
function JuMP.parse_one_operator_constraint(_error::Function, vectorized::Bool,
                                            ::Val{:⊇}, lhs, rhs)
    parse_one_operator_constraint(_error, vectorized, Val(:⊆), rhs, lhs)
end
```

Store set constraints

```
function JuMP.build_constraint(
    _error::Function, subset, sup_powerset::PowerSet;
    kws...)
    InclusionConstraint(subset, sup_powerset.set, kws)
end
function JuMP.add_constraint(
    model::JuMP.Model, constraint::SetConstraint,
    name::String="")
    d = data(model)
    index = ConstraintIndex(d.last_index += 1)
    d.constraints[index] = constraint
    d.names[index] = name
    return JuMP.ConstraintRef(model, index, SetShape())
end
```

Optimize hook

`create_spaces`: Find dimensions and representation (e.g. polar/dual or not)

```
function optimize_hook(model::JuMP.AbstractModel; kws...)
    d = data(model)
    clear_spaces(d)
    create_spaces(d)
    load(model, d)
    JuMP.optimize!(model, ignore_optimize_hook = true,
                   kws...)
end
```

ParameterJuMP extension

ParameterJuMP example

```
model = ModelWithParams(with_optimizer(...))
@variable(model, x)
add_parameter(model, a, 10)
@constraint(model, x >= a)
optimize!(model)
dual(a)

fix(a, 20)
optimize!(model)
```

Adding parametrized constraint

```
function JuMP.build_constraint(
    _error::Function, aff::PAE, set)
    return JuMP.ScalarConstraint(aff, set) # Simplified
end

function JuMP.add_constraint(model::JuMP.Model, c::PAEC,
                             name::String="")
    c_lin = JuMP.ScalarConstraint(c.func.v, c.set)
    cref = JuMP.add_constraint(model, c_lin, name)
    data = model.ext[:params]::ParameterData
    _get_param_dict(data, S)[cref] = c.func.p
    return cref
end
```

Optimize hook

```
function parameter_optimizehook(model::JuMP.Model;
                                kwargs...)
    data = model.ext[:params]::ParameterData

    sync(data)

    ret = JuMP.optimize!(
        model, ignore_optimize_hook = true, kwargs...)

    if !no_duals(data) && JuMP.has_duals(model)
        _update_duals(data)
    end
    return ret
end
```

StructJuMP extension

Example

```
numScen = 2
m = StructuredModel(num_scenarios=numScen)
@variable(m, x[1:2])
@NLconstraint(m, x[1] + x[2] == 100)
@NLobjective(m, Min, x[1]^2 + x[2]^2 + x[1]*x[2])

for i in 1:numScen
    bl = StructuredModel(parent=m, id=i)
    @variable(bl, y[1:2])
    @NLconstraint(bl, x[1] + y[1] + y[2] >=  0)
    @NLconstraint(bl, x[2] + y[1] + y[2] <= 50)
    @NLobjective(bl, Min, y[1]^2 + y[2]^2 + y[1]*y[2])
end
```

AbstractModel without any MOI backend

```
mutable struct StructuredModel <: JuMP.AbstractModel
    parent::Union{StructuredModel, Nothing}
    # Graph data ...

    nextvaridx::Int
    variables::Dict{Int, JuMP.AbstractVariable}
    varnames::Dict{Int, String}
    nextconidx::Int
    constraints::Dict{Int, JuMP.AbstractConstraint}
    connames::Dict{Int, String}
    objective_sense::MOI.OptimizationSense
    objective_function::JuMP.AbstractJuMPScalar
end
```

Variables

```
struct StructuredVariableRef <: JuMP.AbstractVariableRef
    model::StructuredModel
    idx::Int
end
function JuMP.add_variable(
    model::StructuredModel, v::JuMP.AbstractVariable,
    name::String="")
    model.nextvaridx += 1
    vref = StructuredVariableRef(
        model, model.nextvaridx)
    model.variables[vref.idx] = v
    JuMP.set_name(vref, name)
    return vref
end
```

Constraint

```
struct StructuredConstraintRef
    model::StructuredModel
    idx::Int
end
function JuMP.add_constraint(
    model::StructuredModel, c::JuMP.AbstractConstraint,
    name::String="")
    model.nextconidx += 1
    cref = StructuredConstraintRef(
        model, model.nextconidx)
    model.constraints[cref.idx] = c
    JuMP.set_name(cref, name)
    return cref
end
```

Objective

```
function JuMP.set_objective_sense(
    model::StructuredModel, sense::MOI.OptimizationSense)
    model.objective_sense = sense
end
function JuMP.set_objective_sense(
    model::StructuredModel, f::JuMP.AbstractJuMPScalar)
    model.objective_function = f
end
```

New attribute

```
struct VariableBlock <: MOI.AbstractVariableAttribute
end
```

- Block 0: parent
- Block i: ith children

```
optimizer = PIPS.Optimizer()
parent_vars = MOI.add_variables(optimizer, n0)
MOI.set(optimizer, VariableBlock(), parent_vars,
        zeros(n0))
child1_vars = MOI.add_variables(optimizer, n1)
MOI.set(optimizer, VariableBlock(), child_vars,
        zeros(n1))
```

MomentOpt extension

Example

Generalized Moment Problem for

$$\min_{|x|\leq 1, |y|\leq 1} x^4y^2 + x^2y^4 - 3x^2y^2 + 1$$

```
@polyvar x y
f =x^4*y^2 + x^2*y^4 -3x^2*y^2 + 1
K = @set -1 <= x && x <= 1 && -1 <= y && y <= 1
gmp = GMPModel()
@measure(gmp, mu, [x, y], support=K)
@objective(gmp, Min, dot(f, mu))
@constraint(gmp, dot(1, mu) == 1)
relax!(gmp, 2, with_optimizer(CSDP.Optimizer))
```

Model

```
mutable struct GMPModel <: JuMP.AbstractModel
    objective::Union{Nothing, MomObj}
    constraints::Vector{MomCon}
    constraint_names::Vector{String}
    measures::Vector{Measure}

    dual::JuMP.Model
    cref::Dict{Measure, JuMP.ConstraintRef{JuMP.Model}}
    dref::Vector{JuMP.VariableRef}
end
```

Constraint

```
function JuMP.build_constraint(
    _error::Function, aff::AffMomExpr,
    set::MOI.AbstractScalarSet)
    return MomCon(aff, set)
end

function JuMP.add_constraint(
    gmp::GMPModel, momcon::MomCon, name::String="")
    push!(gmp.constraints, momcon)
    push!(gmp.constraint_names, name)
    return ConstraintRef(gmp, length(constraints(gmp)),
                                MomConShape())
end
```

Objective

```
function JuMP.set_objective(
    gmp::GMPModel, sense::MOI.OptimizationSense,
    mom::AbstractMomentExpressionLike)
    gmp.objective = MomObj(sense, mom)
end
```

Relax

```
function relax!(gmp::GMPModel, order::Int,
                optimizer::OptimizerFactory)
    gmp.dual = SOSModel(optimizer)

    # ...

    optimize!(gmp.dual)
end
```

Conclusion

Many ways to extend JuMP/MOI:

- Create new MOI attribute/submittable
- Create new MOI set and bridge
- Create new `AbstractVariableRef`
- Extend variable macro with `parse_variable`, `build_variable`, `add_variable`
- Extend constraint macro with `parse_constraint`, `build_constraint`, `add_constraint`
- Extend objective macro with `set_objective_function`
- Use the `ext` field and an optimize hook
- Use a custom `JuMP.AbstractModel`
- ...