Recent Advances in Mixed Integer Programming Modeling and Computation

Juan Pablo Vielma

Massachusetts Institute of Technology

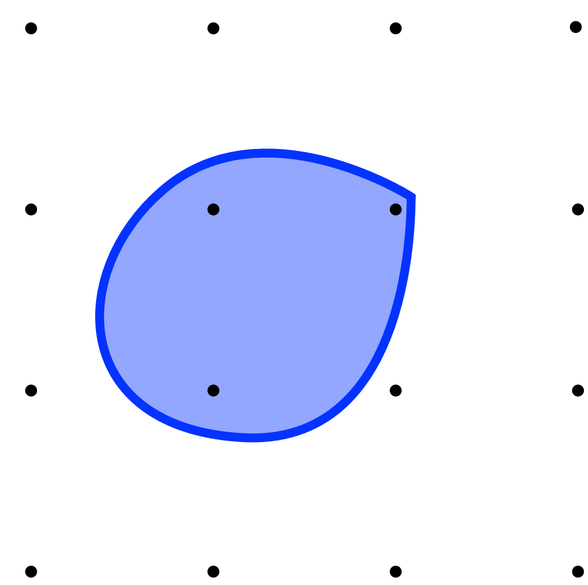
Departamento de Ingeniería Industrial y de Sistemas

Pontificia Universidad Católica de Chile.

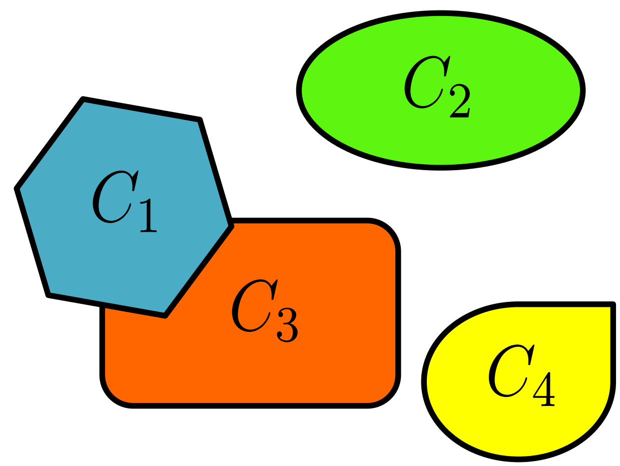
Santiago, Chile, Agosto, 2017.

(Nonlinear) Mixed Integer Programming (MIP)

$$\begin{aligned} \min \quad & f(x) \\ s.t. \quad & \\ & x \in C \\ & x_i \in \mathbb{Z} \quad i \in I \end{aligned}$$



Mostly convex f and C.



NP-hard = Challenge Accepted!

50+ Years of MIP = Significant Solver Speedups

- Algorithmic Improvements (**Machine Independent**):
  - CPLEX → ILOG → IBM
    - v1.2 (1991) – v11 (2007): **29,000 x** speedup
  - GUROBI OPTIMIZATION
    - v1 (2009) – v6.5 (2015): **48.7 x** speedup

≈ **1.9 x / year**

- Also **convex nonlinear**:
  - GUROBI OPTIMIZATION
    - v6.0 (2014) – v6.5 (2015) quadratic: **4.43 x**
      (**V.**, Dunning, Huchette, Lubin, 2015)

Widespread Use of Linear/Quadratic MIP Solvers

http://www.gurobi.com/company/example-customers

State of MIP Solvers

- Mature: Linear and Quadratic (Conic Quadratic/SOCP)
  - Commercial:

GUROBI OPTIMIZATION — IBM CPLEX — optimized by Xpress MP FICO

  - "Open Source"

SCIP — COIN|OR CBC — GNU GLPK

- Emerging: Convex Nonlinear (e.g. SDP)
  - Open-Source + Commercial linear MIP Solver > Commercial

KNITRO < COIN|OR Bonmin < 

Accessing MIP Solvers = Modelling Languages

- User-friendly algebraic modelling languages (AML):

GAMS
AMPL

PYOMO ↔ python™
CVX RESEARCH ↔ MATLAB®

Standalone and Fast — Based on General Language and Versatile

- Fast and Versatile, but complicated
  - Proprietary low-level C/C++ solver interphases.
  - C/C++ Coin-OR interphases and frameworks
- 21st Century AMLs: JuMP ↔ julia

Outline

- JuMP ↔ julia overview.
- Advanced MIP formulations.
- Convex nonlinear MIP solvers.
- Optimal Control with Julia, JuMP and Pajarito.
- Other applications if time permits.

JuMP ⟷ julia

Why julia and JuMP ?

- julia http://julialang.org
  - 21^{st} century programming language
  - MIT licensed (and developed): free and open source
  - (Almost) as fast as C (LLVM JIT) and as easy as Matlab
    - “Floats like python/matlab, stings like C/Fortran”
  - Easy to use and wide library ecosystem (specialized and frontend)
  - Only language besides C/C++/Fortran to scale to 1 Petaflop!
    - 10^{15} floating point operations per second on NERSC Cori Phase II (9,300 nodes and 650,000 cores)

Why julia and JuMP?

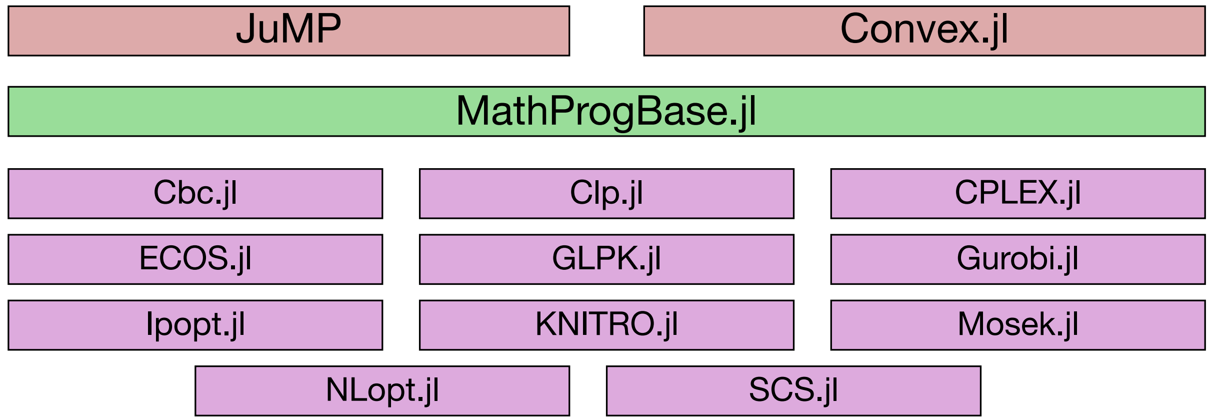
- JuMP https://github.com/JuliaOpt/JuMP.jl
  - Also open-source and free.
  - Julia-based algebraic modelling language for optimization
  - Easy and natural syntax for linear, quadratic and conic (e.g. SDP) mixed-integer optimization.
  - Modular, extensible, easy to embed (e.g. simulation, visualization, etc.) and FAST.
  - Solver-independent access to advanced MIP features (e.g cutting plane callbacks)

OPERATIONS RESEARCH CENTER

MIT MANAGEMENT SLOAN SCHOOL

Extensive Stack of Modelling and Solver Packages

JuliaOpt



Solvers

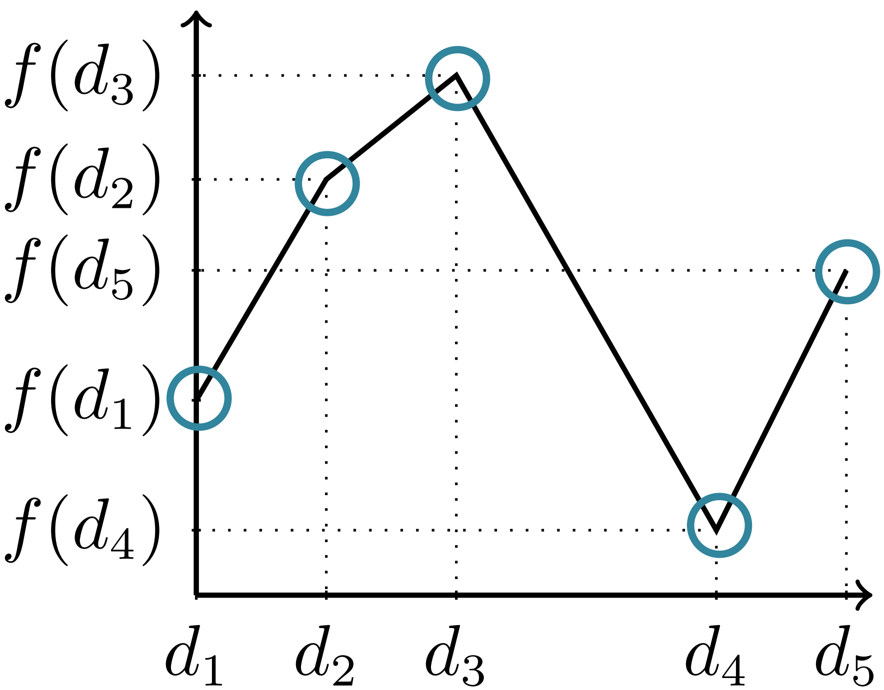
http://www.juliaopt.org

- JuMP extensions for: block stochastic optimization, robust optimization, chance constraints, piecewise linear optimization, polynomial optimization, multi-objective optimization, discrete time stochastic optimal control, sum of squares optimization, etc.
- Useful Julia Packages: Multivariate Polynomials, etc.

Advanced MIP Formulations

Simple Formulation for Univariate Functions

$$z = f(x)$$



$$\text{Size} = O(\#\text{ of segments})$$

Non-Ideal: Fractional Extreme Points

$$\begin{pmatrix} x \\ z \end{pmatrix} = \sum_{j=1}^{5} \begin{pmatrix} d_j \\ f(d_j) \end{pmatrix} \lambda_j$$

$$1 = \sum_{j=1}^{5} \lambda_j, \quad \lambda_j \geq 0$$

$$y \in \{0,1\}^4, \quad \sum_{i=1}^{4} y_i = 1$$

$$0 \leq \lambda_1 \leq y_1$$

$$0 \leq \lambda_2 \leq y_1 + y_2$$

$$0 \leq \lambda_3 \leq y_2 + y_3$$

$$0 \leq \lambda_4 \leq y_3 + y_4$$

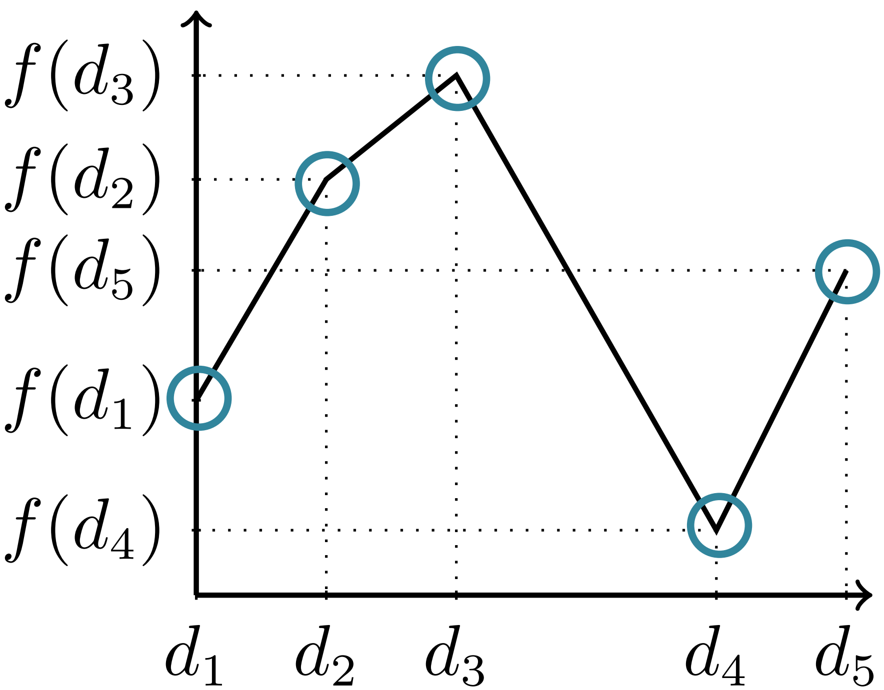
$$0 \leq \lambda_5 \leq y_4$$

Advanced Formulation for Univariate Functions

$$z = f(x)$$

$$\begin{pmatrix} x \\ z \end{pmatrix} = \sum_{j=1}^{5} \begin{pmatrix} d_j \\ f(d_j) \end{pmatrix} \lambda_j$$

$$1 = \sum_{j=1}^{5} \lambda_j, \quad \lambda_j \geq 0$$



$$y \in \{0,1\}^2$$

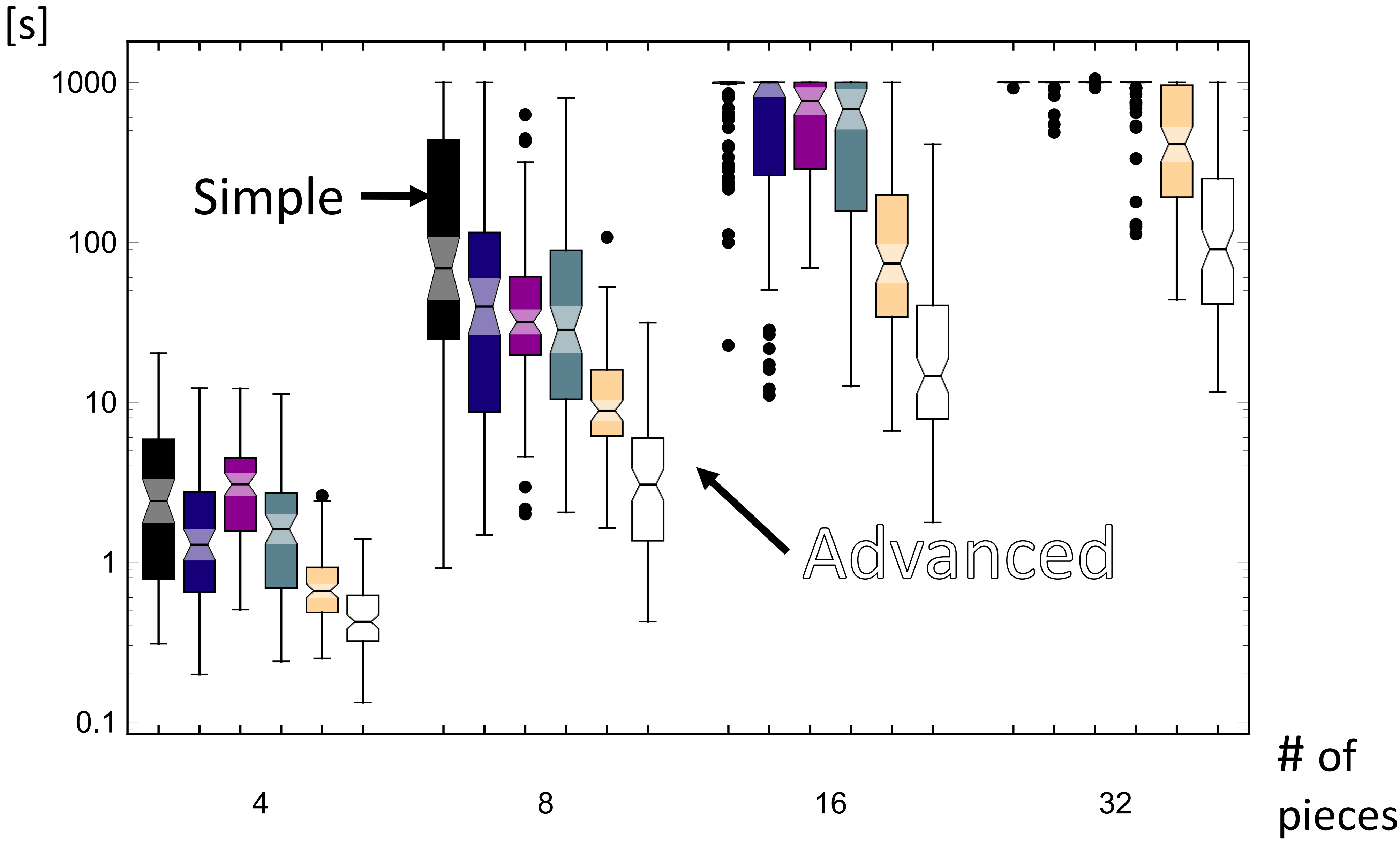
$$0 \leq \lambda_1 + \lambda_5 \leq 1 - y_1$$
$$0 \leq \lambda_3 \leq y_1$$
$$0 \leq \lambda_4 + \lambda_5 \leq 1 - y_2$$
$$0 \leq \lambda_1 + \lambda_2 \leq y_2$$

$$\text{Size} = O(\log_2 \# \text{ of segments})$$

Ideal: Integral Extreme Points

- V. and Nemhauser 2011.

Formulation Improvements can be Significant



Transportation Problems with Concave Costs using GUROBI

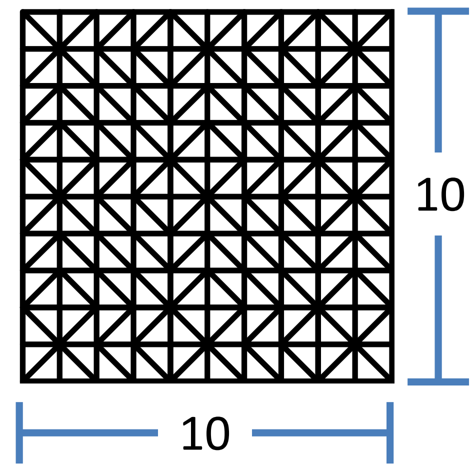
All Easily Accessible Through JuMP Extensions

- PiecewiseLinearOpt.jl (Huchette and V. 2017)

$$\begin{array}{ll} \min & \exp(x+y) \\ s.t. & \\ & x, y \in [0,1] \end{array}$$

Automatically select Δ

Automatically construct formulation (easily chosen)




```
using JuMP, PiecewiseLinearOpt
m = Model()
@variable(m, x)
@variable(m, y)

z = piecewiselinear(m, x, y, 0:0.1:1, 0:0.1:1, (u,v) -> exp(u+v))
@objective(m, Min, z)
```

Convex Nonlinear MIP Solvers

Nonlinear MIP B&B Algorithms

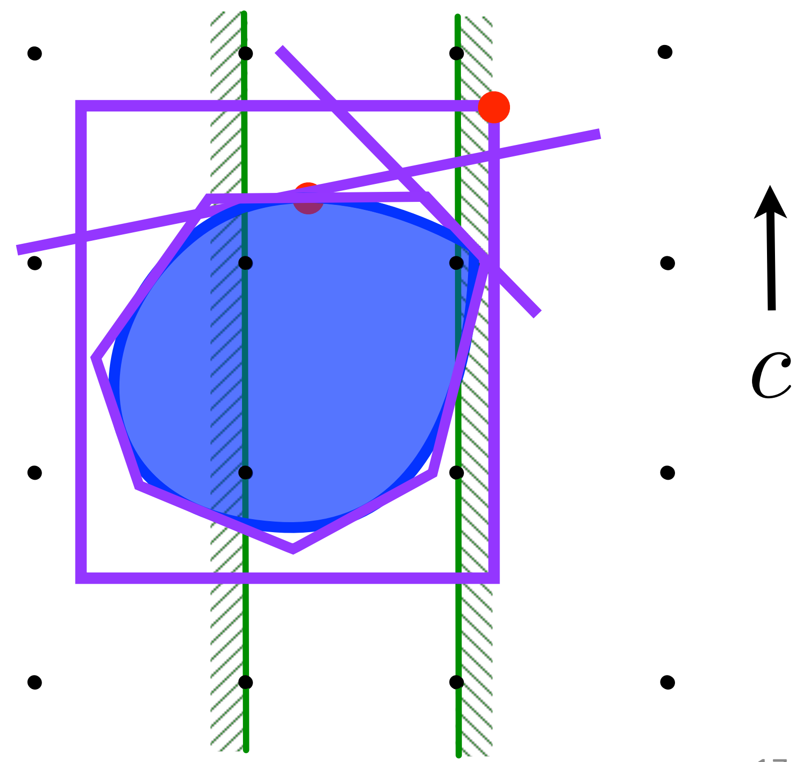
- NLP (QCP) Based B&B
- (Dynamic) LP Based B&B
  - Few cuts = high speed.
  - Possible slow convergence.
- Lifted LP B&B
  - Extended or Lifted relaxation.
  - Static relaxation
    - Mimic NLP B&B.
  - Dynamic relaxation
    - Standard LP B&B

$$\max \quad \sum_{i=1}^{n} c_i x_i$$

$$s.t. \quad Ax + Dz \leq b,$$

$$g_i(x) \leq 0,\ i \in I, \quad x \in \mathbb{Z}^n$$

$$x \in \mathbb{Z}^{n_1} \times \mathbb{R}^{n_2}$$



Second Order Conic or Conic Quadratic Problems

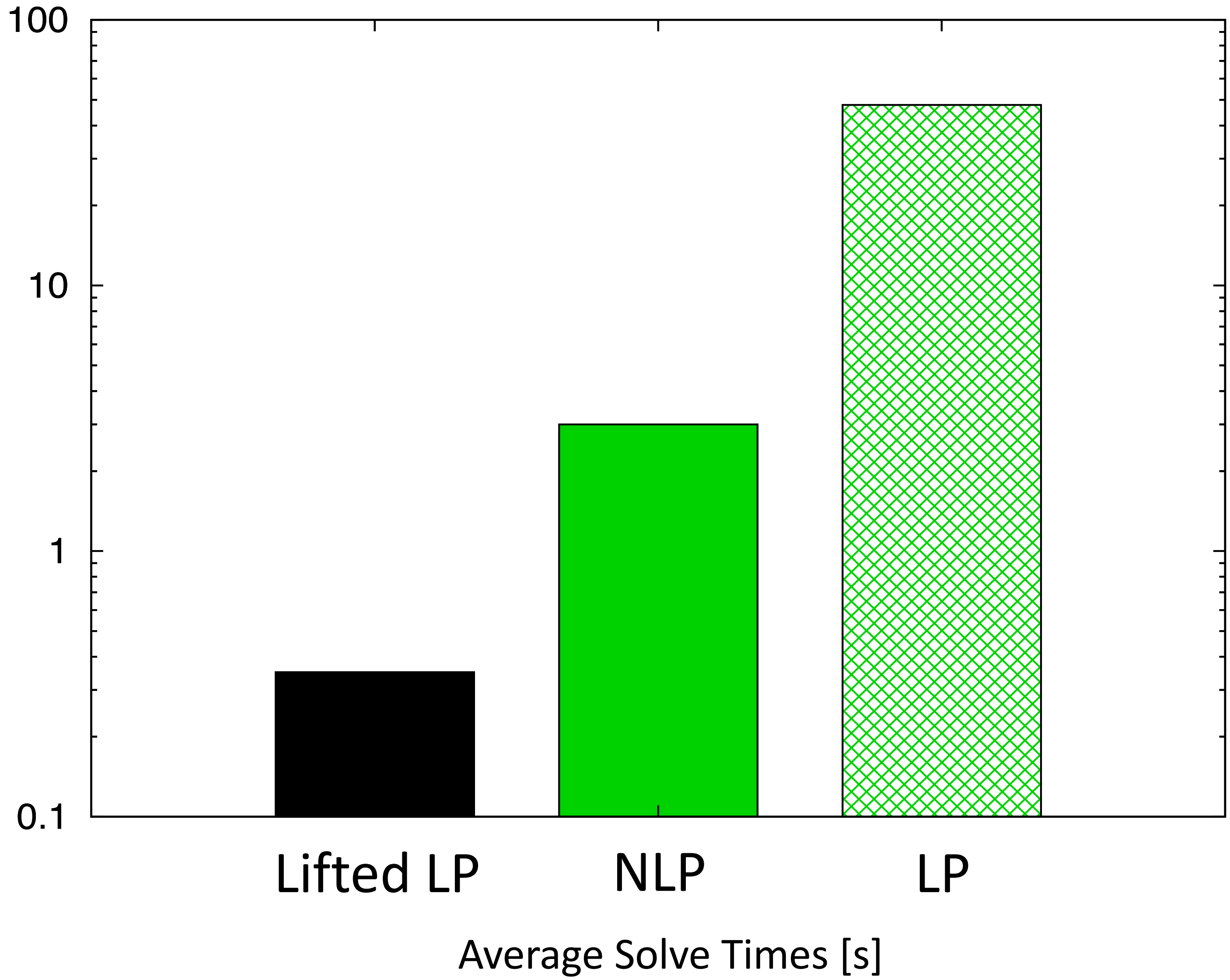
- Problems using Euclidean norm:
  - e.g. Portfolio Optimization Problems

$$
\begin{aligned}
\max \quad & \bar{a}x \\
s.t. \quad & \\
& \left\| Q^{1/2} x \right\|_2 \leq \sigma \\
& \sum_{j=1}^{n} x_j = 1, \quad x \in \mathbb{R}_+^n \\
& x_j \leq z_j \quad \forall j \in [n] \\
& \sum_{j=1}^{n} z_j \leq K, \quad z \in \{0,1\}^n
\end{aligned}
$$

- $\bar{a}$ expected returns.
- $Q^{1/2}$ square root of covariance matrix.
- K maximum number of assets.
- σ maximum risk.

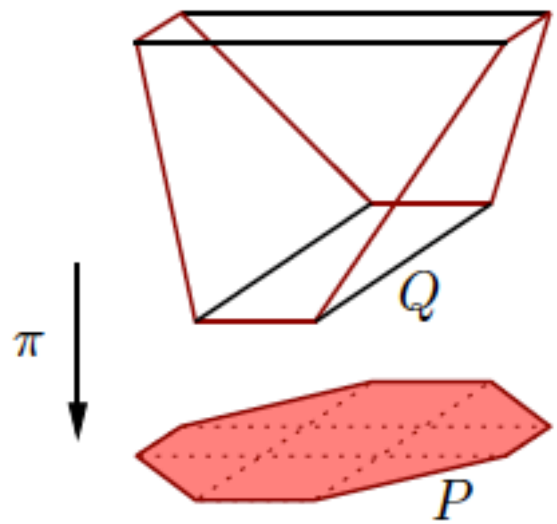
LP v/s NLP B&B for CPLEX v11 for n = 20 and 30

- Results from V., Ahmed and Nemhauser 2008.



Lifted or Extended Approximations

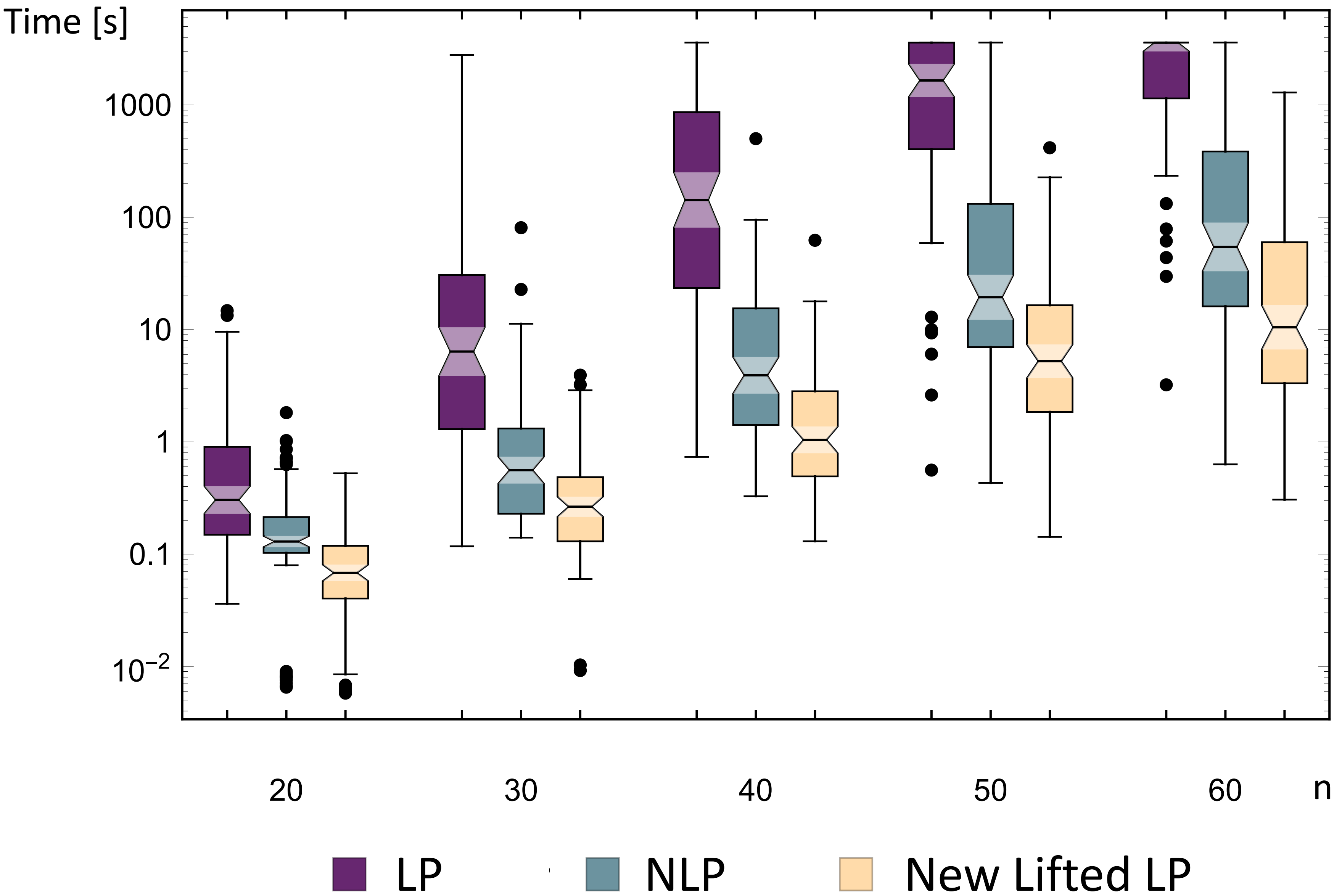
- Projection = multiply constraints.
- V., A. and N. 2008:
  - Extremely accurate, but static and complex approximation by Ben-Tal and Nemirovski
- V., Dunning, Huchette and Lubin 2016: Simple, dynamic and good approximation:



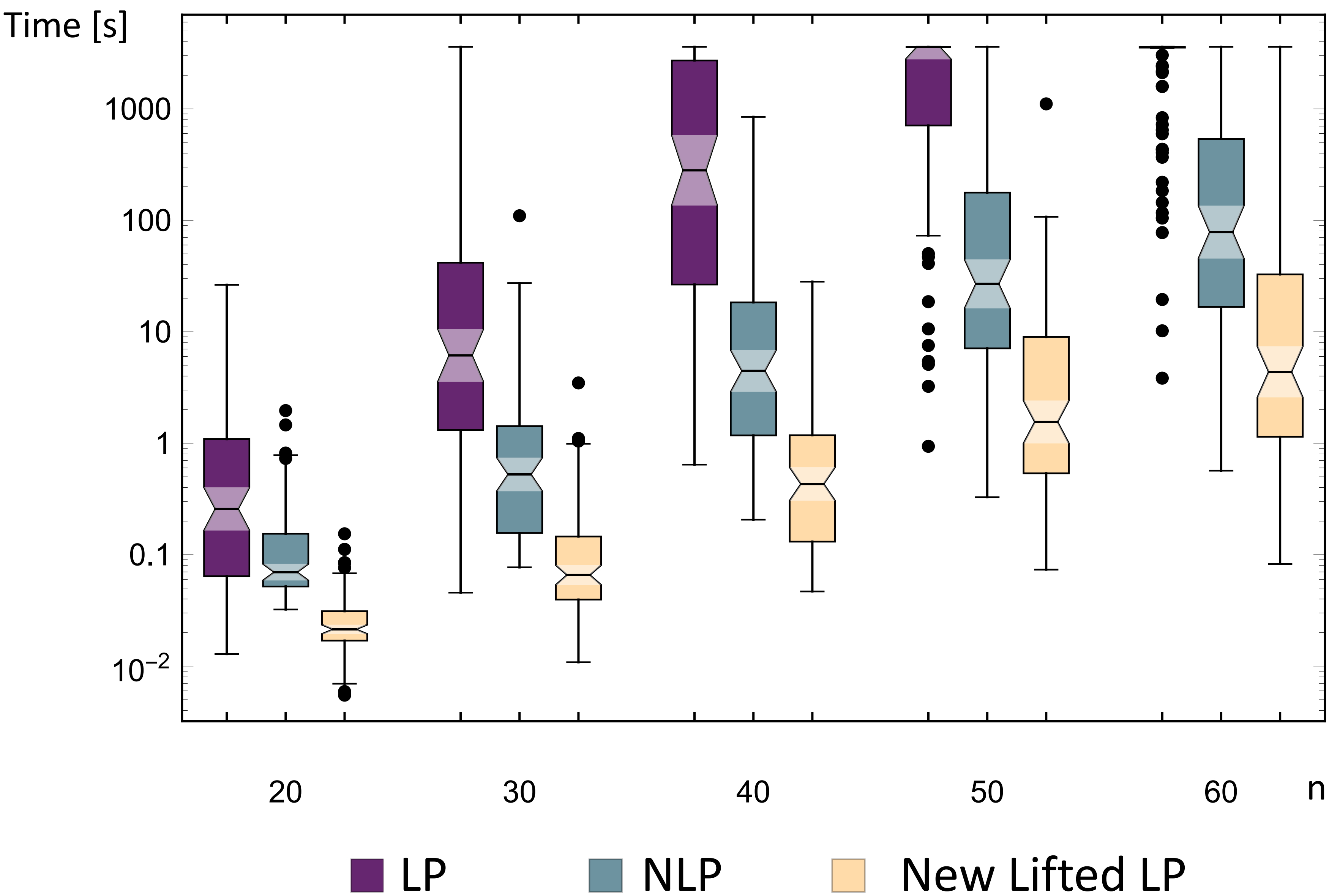
$$\|y\|_2 \leq y_0 \quad \longrightarrow \quad \begin{array}{l} y_i^2 \leq z_i \cdot y_0 \quad \forall i \in [n] \\ \sum_{i=1}^n z_i \leq y_0 \end{array}$$

Image from Lipton and Regan, https://rjlipton.wordpress.com

CPLEX v12.6 for n = 20, 30, 40, 50 and 60



Gurobi v5.6.3 for n = 20, 30, 40, 50 and 60



All Major Solvers Now Implement Lifted LP

- First Talks: julia / JuMP
  - SIAM Optimization (SIOPT), May 2014 ≈ two weeks coding.
  - IBM Thomas J. Watson Research Center, December 2014.
- Paper in arxive, May 28, 2015.

Two weeks!

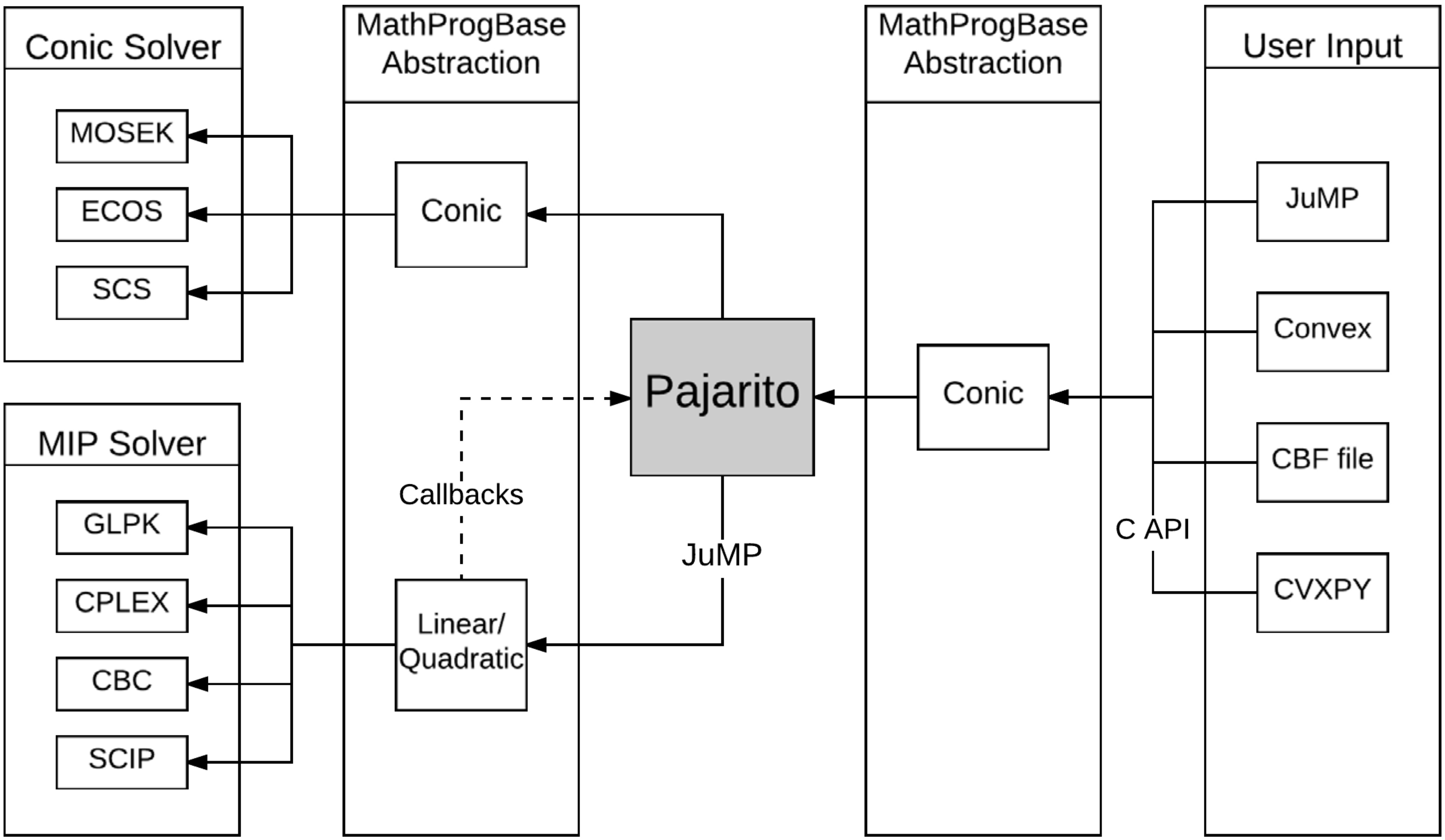
- IBM CPLEX v12.6.2, June 12, 2015.
- GUROBI OPTIMIZATION v6.5, October 2015.
- optimized by Xpress MP FICO v8.0, May 2016.
- SCIP v4.0, March 2017.

However... We Can Sill Beat CPLEX!

- julia / JuMP-based solver Pajarito
  - Lubin, Yamangil, Bent and V. '16 and Coey, Lubin and V. '17.

solver	conv	wrong	not conv	limit	time(s)
	termination status counts				
SCIP	78	1	0	41	43.36
CPLEX	96	3	5	16	14.30
Paj-iter	96	1	0	23	38.70
Paj-MSD	101	0	0	19	18.12

Flexible Architecture Thanks to Julia-Opt Stack

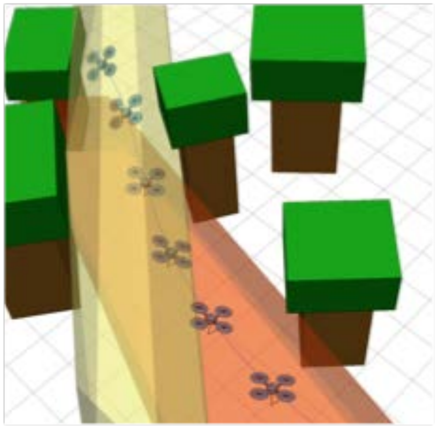


- Fastest Open Source MISOCP Solver!
- Pajarito can also solve MISDPs and MI-"EXP"

Optimal Control with Julia, JuMP and Pajarito

Joey Huchette ≈ two weeks for SIOPT '17

Trajectory Planning with Collision Avoidance



- Motivating: Steering a quadcopter through obstacles [Deits/Tedrake:2015]
- Position described by polynomials:
  - $(p^x(t), p^y(t))_{t\in[0,1]}$
  - avoid obstacles
  - initial/terminal conditions
  - minimize "jerk" of path
- Solution approach:
  - split domain into "safe" polyhedrons + discretize time into intervals
  - "smooth" piecewise polynomial trajectories in each interval, which chose polyhedron

variables = polynomials

↓

Mixed-Integer Polynomial Programming

Disjunctive *Polynomial* Optimization Formulation

Variables = Polynomials : $\{p_i : [T_i, T_{i+1}] \to \mathbb{R}^2\}_{i=1}^N$

$$\min_{\boxed{p}} \quad \sum_{i=1}^N ||p_i'''(t)||^2$$

s.t.

$$p_1(0) = X_0,\ p'(0) = X_0',\ p''(0) = X_0''$$

$$p_N(1) = X_f,\ p_N'(1) = X_f',\ p_N''(1) = X_f''$$

Initial/Terminal Conditions

$$p_i(T_{i+1}) = p_{i+1}(T_{i+1}) \quad \forall i \in \{1, \ldots, N-1\}$$

$$p_i'(T_{i+1}) = p_{i+1}'(T_{i+1}) \quad \forall i \in \{1, \ldots, N-1\}$$

$$p_i''(T_{i+1}) = p_{i+1}''(T_{i+1}) \quad \forall i \in \{1, \ldots, N-1\}$$

Interstitial Smoothing Conditions

$$\bigvee_{r=1}^R [A^r p_i(t) \leq b^r] \ \text{ for } t \in [T_i, T_{i+1}] \quad \forall i \in \{1, \ldots, N-1\}$$

Avoid Collision = Remain in Safe Regions

~~Disjunctive~~ *Polynomial* Optimization Formulation
Mixed-Integer

Variables = Polynomials : $\{p_i : [T_i, T_{i+1}] \to \mathbb{R}^2\}_{i=1}^N$

$$\min_{\boxed{p}} \quad \sum_{i=1}^N ||p_i'''(t)||^2$$

s.t. $p_1(0) = X_0,\ p'(0) = X_0',\ p''(0) = X_0''$

$p_N(1) = X_f,\ p_N'(1) = X_f',\ p_N''(1) = X_f''$

Initial/Terminal Conditions

$p_i(T_{i+1}) = p_{i+1}(T_{i+1}) \quad \forall i \in \{1, \ldots, N-1\}$

$p_i'(T_{i+1}) = p_{i+1}'(T_{i+1}) \quad \forall i \in \{1, \ldots, N-1\}$

$p_i''(T_{i+1}) = p_{i+1}''(T_{i+1}) \quad \forall i \in \{1, \ldots, N-1\}$

Interstitial Smoothing Conditions

$b_j^r + M_j^r(1 - z_{i,r}) - A_j^r p_i(t) \geq 0 \quad$ for $t \in [T_i, T_{i+1}] \quad \forall i, j, r$

$\sum_{r=1}^R z_{i,r} = 1 \quad \forall i, z \in \{0, 1\}^{N \times R}$

Avoid Collision = Remain in Safe Regions

~~Disjunctive~~ ~~*Polynomial*~~ Optimization Formulation
Mixed-Integer Sum-of-Squares

Variables = Polynomials : $\{p_i : [T_i, T_{i+1}] \to \mathbb{R}^2\}_{i=1}^N$

$$\min_{\boxed{p}} \quad \sum_{i=1}^{N} ||p_i'''(t)||^2$$

s.t.

$$p_1(0) = X_0,\ p'(0) = X_0',\ p''(0) = X_0''$$
$$p_N(1) = X_f,\ p_N'(1) = X_f',\ p_N''(1) = X_f''$$

Initial/Terminal Conditions

$$p_i(T_{i+1}) = p_{i+1}(T_{i+1}) \quad \forall i \in \{1, \dots, N-1\}$$
$$p_i'(T_{i+1}) = p_{i+1}'(T_{i+1}) \quad \forall i \in \{1, \dots, N-1\}$$
$$p_i''(T_{i+1}) = p_{i+1}''(T_{i+1}) \quad \forall i \in \{1, \dots, N-1\}$$

Interstitial Smoothing Conditions

$$b_j^r + M_j^r(1 - z_{i,r}) - A_j^r p_i(t) \ \text{ is SOS for } t \in [T_i, T_{i+1}] \quad \forall i, j, r$$

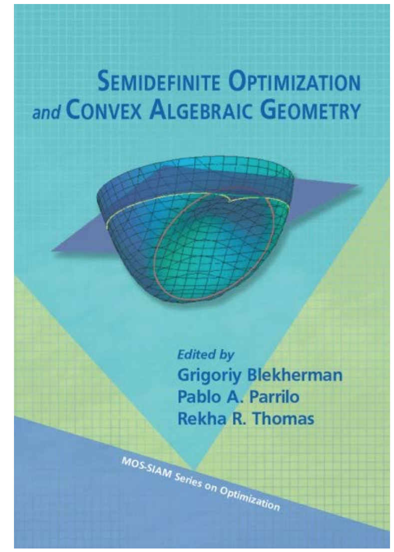
$$\sum_{r=1}^{R} z_{i,r} = 1 \quad \forall i,\ z \in \{0,1\}^{N \times R}$$

Avoid Collision = Remain in Safe Regions

From Sum of Squares to Semidefinite Programming

- Sufficient condition for non-negative polynomial:
  - Sum of Squares : $$f(x) = \sum_i g_i^2(x)$$
  - SDP representable for fixed degree:

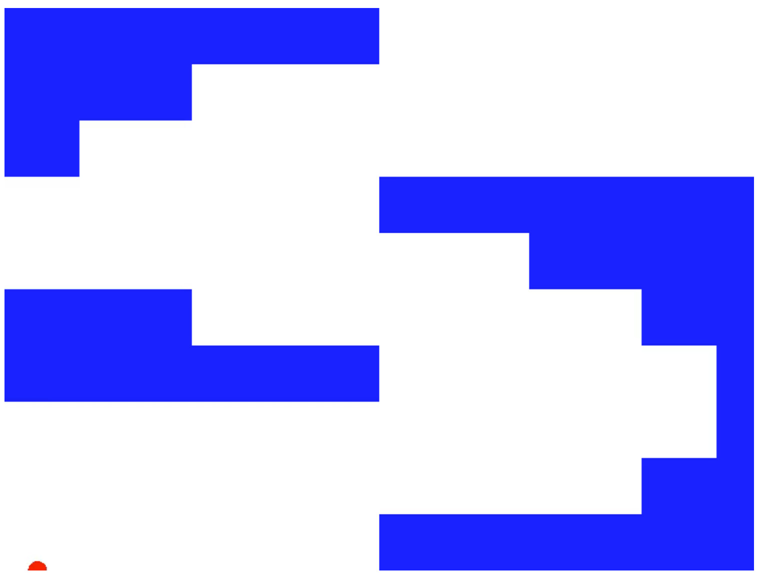
$$\text{degree} \leq k \quad \rightarrow \quad (k-1) \times (k-1) \text{ matrices}$$



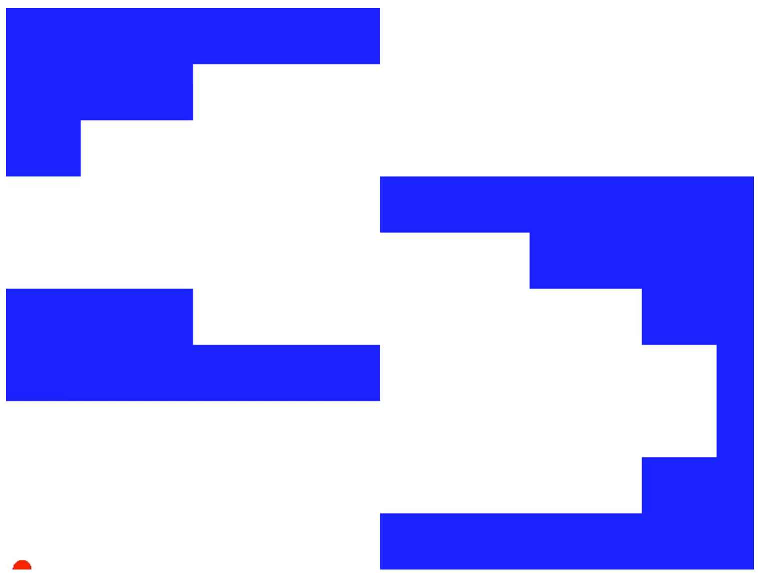
- MI-SOS:
  - Low degree polynomials (≤3):
    - MI-SOCP: solvable by Gurobi/CPLEX
    - Deits/Tedrake:2015
  - Higher degree polynomials:
    - MI-SDP: solvable by Pajarito

Results for 9 Regions and 8 time steps

- Infeasible for degree ≤ 3 (MI-SOCP)
- Pajarito results for degree 5:



First Feasible Solution:
58 seconds



Optimal Solution:
651 seconds

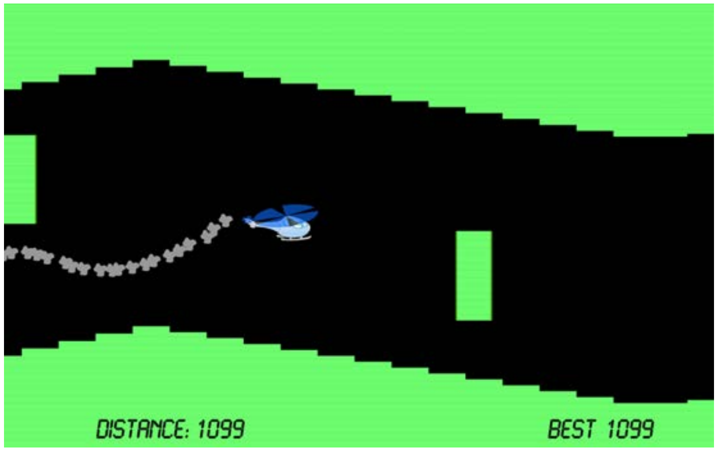
```julia
model = SOSModel(solver=PajaritoSolver())

@polyvar(t)
Z = monomials([t], 0:r)

@variable(model, H[1:N,boxes], Bin)

p = Dict()
for j in 1:N
   @constraint(model, sum(H[j,box] for box in boxes) == 1)
   p[(:x,j)] = @polyvariable(model, _, Z)
   p[(:y,j)] = @polyvariable(model, _, Z)
   for box in boxes
       xl, xu, yl, yu = box.xl, box.xu, box.yl, box.yu
       @polyconstraint(model, p[(:x,j)] >= Mxl + (xl-Mxl)*H[j,box], domain = (t >= T[j] && t <= T[j+1]))
       @polyconstraint(model, p[(:x,j)] <= Mxu + (xu-Mxu)*H[j,box], domain = (t >= T[j] && t <= T[j+1]))
       @polyconstraint(model, p[(:y,j)] >= Myl + (yl-Myl)*H[j,box], domain = (t >= T[j] && t <= T[j+1]))
       @polyconstraint(model, p[(:y,j)] <= Myu + (yu-Myu)*H[j,box], domain = (t >= T[j] && t <= T[j+1]))
   end
end

for ax in (:x,:y)
   @constraint(model,                p[(ax,1)       ]([0], [t]) == X₀[ax])
   @constraint(model, differentiate(p[(ax,1)], t   )([0], [t]) == X₀’[ax])
   @constraint(model, differentiate(p[(ax,1)], t, 2)([0], [t]) == X₀’’[ax])
   for j in 1:N-1
       @constraint(model,                p[(ax,j)     ]([T[j+1]],[t]) ==                p[(ax,j+1)     ]([T[j+1]],[t]))
       @constraint(model, differentiate(p[(ax,j)],t  )([T[j+1]],[t]) == differentiate(p[(ax,j+1)],t  )([T[j+1]],[t]))
       @constraint(model, differentiate(p[(ax,j)],t,2)([T[j+1]],[t]) == differentiate(p[(ax,j+1)],t,2)([T[j+1]],[t]))
   end
   @constraint(model,                p[(ax,N)       ]([1], [t]) == X₁[ax])
   @constraint(model, differentiate(p[(ax,N)], t   )([1], [t]) == X₁’[ax])
   @constraint(model, differentiate(p[(ax,N)], t, 2)([1], [t]) == X₁’’[ax])
end

@variable(model, γ[keys(p)] ≥ 0)
for (key,val) in p
   @constraint(model, γ[key] ≥ norm(differentiate(val, t, 3)))
end
@objective(model, Min, sum(γ))
```

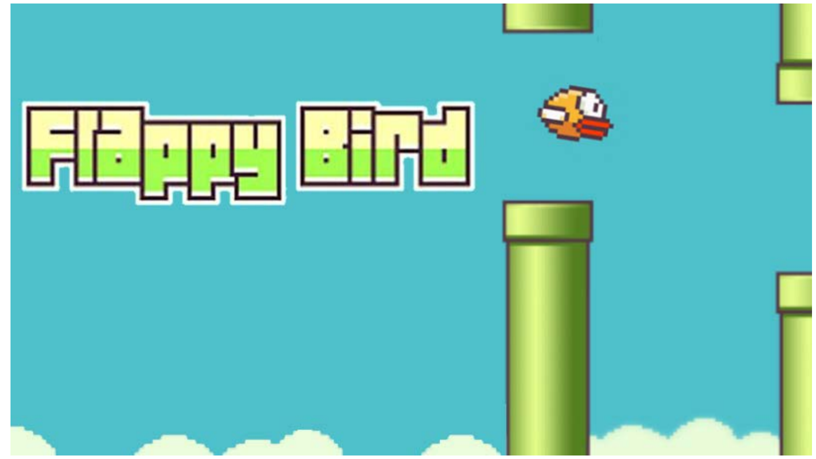
```julia
function eval_poly(r)
    for i in 1:N
        if T[i] <= r <= T[i+1]
            return PP[(:x,i)]([r], [t]), PP[(:y,i)]([r], [t])
            break
        end
    end
end
```

```
using SFML

const window_width = 800
const window_height = 600

window = RenderWindow("Helicopter",
                       window_width, window_height)
event = Event()

rects = RectangleShape[]
for box in boxes
    rect = RectangleShape()
    xl = (window_width/M)*box.xl
    xu = (window_width/M)*box.xu
    yl = window_height*(domain.yu-box.yl)
    yu = window_height*(domain.yu-box.yu)
    set_size(rect, Vector2f(xu-xl, yu-yl))
    set_position(rect, Vector2f(xl, yl))
    set_fillcolor(rect, SFML.white)
    push!(rects, rect)
end

type Helicopter
    shape::CircleShape
    past_path::Vector{Vector2f}
    path_func::Function
end

const radius = 10

heli = Helicopter(CircleShape(),
                  Vector2f[Vector2f(X₀[:x]*window_width,
                       X₀[:y]*window_height)], eval_poly)
set_position(heli.shape, Vector2f(window_width/2,
             window_height/2))
set_radius(heli.shape, radius)
set_fillcolor(heli.shape, SFML.red)
set_origin(heli.shape, Vector2f(radius, radius))
```

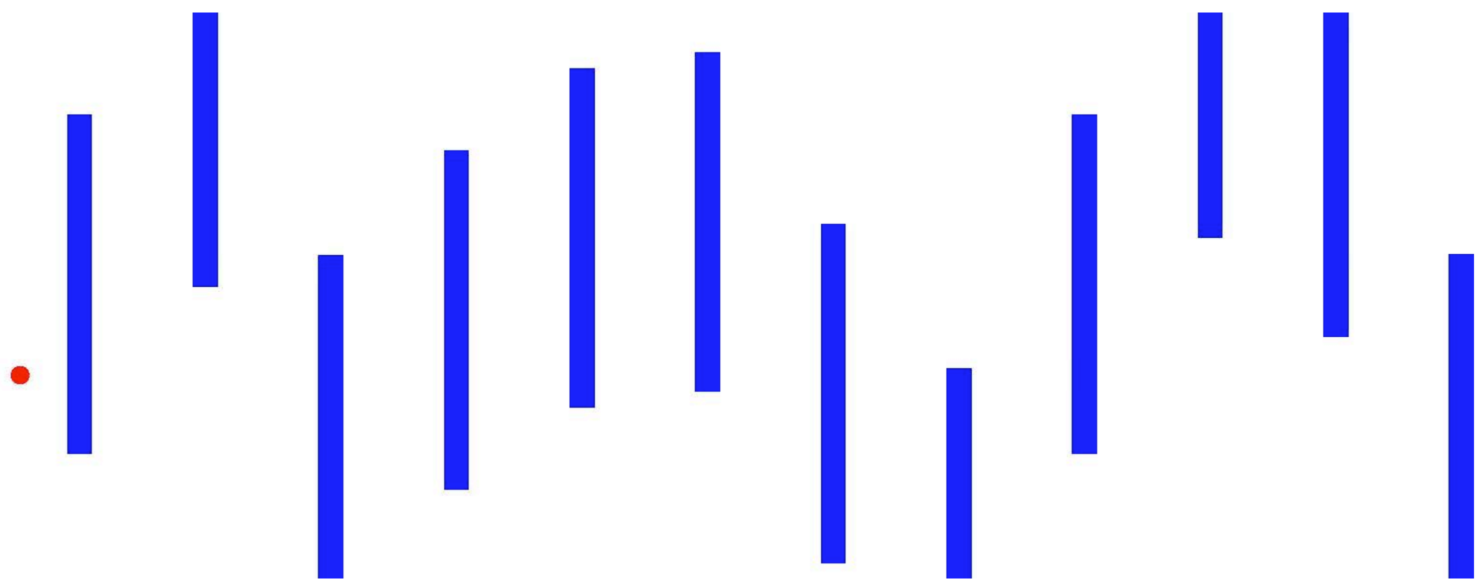
```
function update_heli!(heli::Helicopter, tm)
    (_x,_y) = heli.path_func(tm)
    x = window_width / M * _x
    y = window_height * (1-_y)
    pt = Vector2f(x,y)
    set_position(heli.shape, pt)
    # move(heli.shape, pt-heli.past_path[end])
    push!(heli.past_path, pt)
    get_position(heli.shape)
    nothing
end

const maxtime = 10.0

make_gif(window, window_width, window_height,
         1.05*maxtime, "foobarbat.gif", 0.05)

clock = Clock()
restart(clock)

while isopen(window)
    frametime = as_seconds(get_elapsed_time(clock))
    @show normalizedtime = Tmin +
                (frametime / maxtime)*(Tmax-Tmin)
    (normalizedtime >= Tmax) && break
    while pollevent(window, event)
        if get_type(event) == EventType.CLOSED
            close(window)
        end
    end
    clear(window, SFML.blue)
    for rect in rects
        draw(window, rect)
    end
    update_heli!(heli, normalizedtime)
    draw(window, heli.shape)
    display(window)
end
```

Helicopter Game / Flappy Bird





- 60 horizontal segments, obstacle every 5 = 80 sec. to opt.



Summary

- Advances in MIP
  - Advanced Formulations
  - Advanced Solvers
  - Easy Access Through JuMP
- JuMP extensions
  - Even more domain-specific languages
  - Power Systems:

Los Alamos
NATIONAL LABORATORY