# GPU Acceleration in Optimization and Optimal Control

Pre-Congress Workshop, 23rd IFAC World Congress

Sungho Shin
shin.mit.edu

Massachusetts Institute of Technology

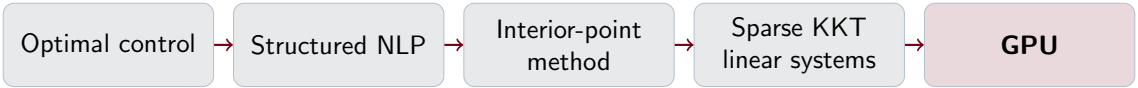
23 August 2026 BEXCO, Busan

# Logistics

| Time | Part | Topic |
|---|---|---|
| 08:35–10:20 | Lecture | Part I: nonlinear optimization<br>Part II: optimal control as a structured NLP<br>Part III: GPU acceleration |
| *10:20–10:40* | *coffee break* | |
| 10:40–11:15 | Hands-on 1 | GPU computing in Julia |
| 11:15–11:55 | Hands-on 2 | Optimal control and parameter estimation on GPUs |

- Slides and notebooks: `https://madsuite.org/ifac2026/`

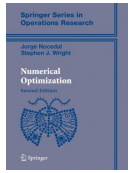
# Scope and Roadmap

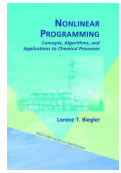



- Part I: nonlinear optimization; optimality conditions and the interior-point method.
- Part II: optimal control as a structured nonlinear program; transcription and collocation.
- Part III: GPU architecture, sparse linear algebra on GPUs, condensed-space interior-point methods.
- In the hands-on session after the break you run the whole pipeline yourself.

# Table of Contents

# References




- Nocedal and Wright (2006): the standard reference; Newton (Ch. 3), AD (Ch. 8), optimality (Ch. 12), interior point (Ch. 19).
- Biegler (2010): optimal control and process systems; Ch. 10 is the source for Part II.
- Wächter and Biegler (2006): the Ipopt paper; Part I's algorithm in detail.
- Hands-on software: ExaModels.jl, MadNLP.jl (Shin, 2025; Shin, Pacaud, et al., 2025).

Part I

# Nonlinear Optimization

Optimality conditions, interior-point methods, the software stack

# Outline

- **Part I: Nonlinear Optimization**
  - Nonlinear Programs
  - Optimality Conditions
  - Interior-Point Methods
  - The Software Stack
  - Summary
- Part II: Optimal Control as a Structured NLP
- Part III: GPU Acceleration
- Part IV: Hands-On Sessions

# Nonlinear Program (NLP) = Nonlinear Optimization

In this workshop, we solve problems of the form

$$\begin{aligned} \min_{\mathbf{x}\in\mathbb{R}^n} \; & f(\mathbf{x}) \\ \text{s.t. } & \mathbf{h}(\mathbf{x}) = \mathbf{0} && \text{(equality constraints)} \\ & \mathbf{g}(\mathbf{x}) \le \mathbf{0} && \text{(inequality constraints)} \end{aligned}$$

with $f : \mathbb{R}^n \to \mathbb{R}$, $\mathbf{h} : \mathbb{R}^n \to \mathbb{R}^{m_E}$, $\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^{m_I}$ twice continuously differentiable.

- Nonconvex, and we look only for a local solution: a feasible point no nearby feasible point improves on. Global optimality is not our concern here.
- Large, but structured. Many real-world problems have this character: discretized optimal control, parameter estimation, power grids, supply chains. $n$ is big, but what comes out is sparse and repetitive, and exploiting that structure is what makes it computationally tractable.

# Outline

# From Local Solutions to Optimality Conditions

## What we are looking for

$\mathbf{x}^*$ is a local solution if it is feasible and no feasible point near it is better:

$$\exists\, \varepsilon > 0 : \quad f(\mathbf{x}^*) \leq f(\mathbf{x}) \quad \text{for every feasible } \mathbf{x} \text{ with } \|\mathbf{x} - \mathbf{x}^*\| < \varepsilon.$$

- An algorithm cannot check that. It quantifies over every nearby feasible point, while a solver holds one point and its derivatives.
- Optimality conditions replace it with equations and inequalities in $f$, $\mathbf{h}$, $\mathbf{g}$ and their derivatives at a single point, which is checkable and something a solver can drive to zero.

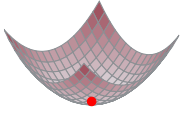
## Plan

unconstrained $\rightarrow$ equality constrained $\rightarrow$ inequality constrained $\rightarrow$ KKT

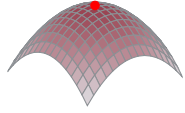
# Unconstrained Problems

At a local solution $\mathbf{x}^*$ of $\min_{\mathbf{x}} f(\mathbf{x})$, from the Taylor expansion:

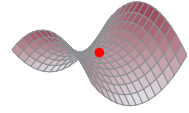
- First-order necessary: $\nabla f(\mathbf{x}^*) = \mathbf{0}$.
- Second-order necessary: $\nabla^2 f(\mathbf{x}^*) \succeq \mathbf{0}$.
- Second-order sufficient: $\nabla f(\mathbf{x}^*) = \mathbf{0}$ and $\nabla^2 f(\mathbf{x}^*) \succ \mathbf{0}$ *imply* a local solution.



local min



local max

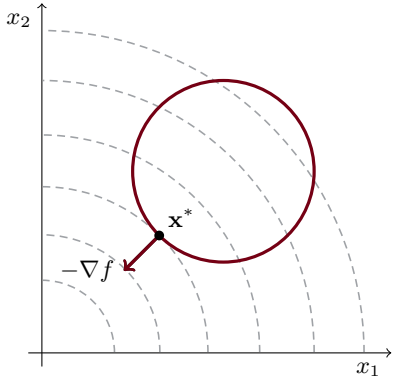


saddle point

- The second-order condition is computationally difficult to check, so most algorithms aim for a stationary point, $\nabla f(\mathbf{x}^*) = \mathbf{0}$, and rely on globalization strategies that reduce the objective at each step.

# Equality, Active, and Inactive Constraints

$$\min_{x_1,x_2} \; f(x_1,x_2) \quad \text{s.t. } h(x_1,x_2)=0$$

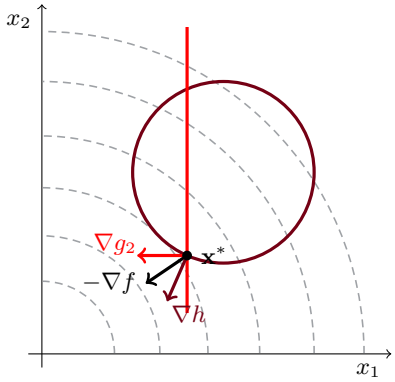



- Equality: feasible moves stay on the curve, so at a solution $-\nabla f$ lies along $\nabla h$.

$$-\nabla f = \lambda \nabla h, \quad \lambda \text{ free}$$

# Equality, Active, and Inactive Constraints

$$\min_{x_1,x_2} \; f(x_1,x_2) \quad \text{s.t. } h(x_1,x_2)=0,\; g_2(x_1,x_2)\le 0$$

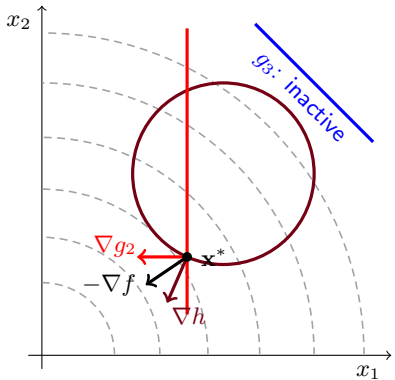



- Equality: feasible moves stay on the curve, so at a solution $-\nabla f$ lies along $\nabla h$.
- Active $g_2$, meaning $g_2(\mathbf{x}^*)=0$: it binds, the solution moves, and $\nabla g_2$ joins, with one sign only, since $g_2$ can push back but not pull.

$$-\nabla f = \lambda \nabla h + \nu_2 \nabla g_2, \quad \nu_2 \ge 0$$

# Equality, Active, and Inactive Constraints

$$\min_{x_1,x_2} \; f(x_1,x_2) \quad \text{s.t. } h(x_1,x_2)=0,\; g_2(x_1,x_2)\le 0,\; g_3(x_1,x_2)\le 0$$




- Equality: feasible moves stay on the curve, so at a solution $-\nabla f$ lies along $\nabla h$.
- Active $g_2$, meaning $g_2(\mathbf{x}^*)=0$: it binds, the solution moves, and $\nabla g_2$ joins, with one sign only, since $g_2$ can push back but not pull.
- Inactive $g_3$ ($g_3(\mathbf{x}^*)<0$): no effect, and which constraints are active is not known in advance.

$$-\nabla f = \lambda \nabla h + \nu_2 \nabla g_2 + \nu_3 \nabla g_3$$

$$\nu_2, \nu_3 \ge 0, \qquad \nu_i \, g_i(\mathbf{x}^*) = 0 \;\Rightarrow\; \nu_3 = 0$$

# The Lagrangian and the Karush–Kuhn–Tucker Conditions

$$\min_{\mathbf{x}} f(\mathbf{x}) \quad \text{s.t. } \mathbf{h}(\mathbf{x}) = \mathbf{0},\ \mathbf{g}(\mathbf{x}) \le \mathbf{0}$$

Both enter the *Lagrangian* $\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\nu}) = f + \boldsymbol{\lambda}^\top \mathbf{h} + \boldsymbol{\nu}^\top \mathbf{g}$; only the sign of $\boldsymbol{\nu}$ differs.

KKT conditions: at a local solution $\mathbf{x}^*$, *under a constraint qualification (e.g., LICQ)*, there exist $\boldsymbol{\lambda}^*, \boldsymbol{\nu}^*$ with

$$\begin{aligned}
\nabla f(\mathbf{x}^*) + \nabla \mathbf{h}(\mathbf{x}^*)\boldsymbol{\lambda}^* + \nabla \mathbf{g}(\mathbf{x}^*)\boldsymbol{\nu}^* &= \mathbf{0} && \text{(stationarity)} \\
\mathbf{h}(\mathbf{x}^*) = \mathbf{0}, \quad \mathbf{g}(\mathbf{x}^*) &\le \mathbf{0} && \text{(primal feasibility)} \\
\boldsymbol{\nu}^* &\ge \mathbf{0} && \text{(dual feasibility)} \\
\nu_i^*\, g_i(\mathbf{x}^*) &= 0 && \text{(complementarity)}
\end{aligned}$$

- Second order: $\mathbf{w}^\top \nabla^2_{\mathbf{xx}} \mathcal{L}\, \mathbf{w} > 0$ on the tangent space of the active constraints.
- These conditions are checkable at a single point, which is what makes them usable by an algorithm. The next section drives them to zero.

# The Combinatorial Complexity of Inequality Constraints

$$\min_{\mathbf{x}} f(\mathbf{x}) \quad \text{s.t. } \mathbf{h}(\mathbf{x}) = \mathbf{0},\ \mathbf{g}(\mathbf{x}) \leq \mathbf{0}$$

- Which inequalities are active is not known until the problem is solved: at $\mathbf{x}^*$ each $g_i$ is tight or slack, and $2^{m_I}$ active sets are possible.
- Active-set methods guess the active set and solve an equality-constrained subproblem. Excellent when small, and they warm-start well; the number of updates grows badly with size.
- Interior-point methods *replace* complementarity by a smooth approximation, and solve a sequence of smooth problems. The method of choice at these sizes, and the one that maps onto a GPU.

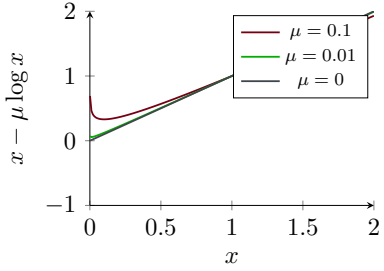
# The Barrier Subproblom

$$\min_{\mathbf{x}} \; f(\mathbf{x}) \quad \text{s.t. } \mathbf{h}(\mathbf{x}) = \mathbf{0},\ \mathbf{x} \geq \mathbf{0}$$

$\Downarrow$ replace the bound by a barrier

$$\min_{\mathbf{x}} \; f(\mathbf{x}) - \mu \textstyle\sum_i \log x_i \quad \text{s.t. } \mathbf{h}(\mathbf{x}) = \mathbf{0}$$

the scalar case $\min_x \; x \text{ s.t. } x \geq 0$




- The barrier $-\mu \log x_i$ blows up at the boundary, so any iterate with a finite objective has $\mathbf{x} > \mathbf{0}$ automatically.
- The inequality is gone: this is an equality-constrained problem, whose optimality conditions we already have.
- The minimizer of $x - \mu \log x$ over $x > 0$ is $x = \mu$, so as $\mu \downarrow 0$ it slides to the boundary and the barrier solution approaches a solution of the original problem.

# The Homotopy Interpretation

Writing the first-order conditions of the barrier subproblem and introducing $\mathbf{z} = \mu \mathbf{X}^{-1}\mathbf{1}$ as the bound multipliers gives the perturbed KKT system:

$$\begin{aligned} \nabla f(\mathbf{x}) + \nabla \mathbf{h}(\mathbf{x})\boldsymbol{\lambda} - \mathbf{z} &= \mathbf{0} \\ \mathbf{h}(\mathbf{x}) &= \mathbf{0} \\ \mathbf{X}\mathbf{Z}\mathbf{1} &= \mu \mathbf{1} \qquad (\leftarrow \text{this is the smoothing}) \end{aligned}$$

where $\mathbf{X} = \operatorname{diag}(\mathbf{x})$, $\mathbf{Z} = \operatorname{diag}(\mathbf{z})$.

- At $\mu = 0$ these are the KKT conditions. For $\mu > 0$ it is a *smooth, square* nonlinear system with a unique solution $(\mathbf{x}(\mu), \boldsymbol{\lambda}(\mu), \mathbf{z}(\mu))$ under standard assumptions.
- The set $\{(\mathbf{x}(\mu), \boldsymbol{\lambda}(\mu), \mathbf{z}(\mu)) : \mu > 0\}$ is the central path.
- The algorithm: follow the central path with Newton steps, driving $\mu$ down to a very small value ($\mu \approx 10^{-9}$) and never to $0$: $\mu = 0$ is degenerate (no interior, dependent constraint gradients).

# The Newton Step

Newton's method on the perturbed KKT system of the previous slide, at the iterate $(\mathbf{x}_k, \boldsymbol{\lambda}_k, \mathbf{z}_k)$:

$$\begin{bmatrix} \nabla^2_{\mathbf{xx}} \mathcal{L}(\mathbf{x}_k, \boldsymbol{\lambda}_k) & \nabla \mathbf{h}(\mathbf{x}_k) & -\mathbf{I} \\ \nabla \mathbf{h}(\mathbf{x}_k)^\top & \mathbf{0} & \mathbf{0} \\ \mathbf{Z}_k & \mathbf{0} & \mathbf{X}_k \end{bmatrix} \begin{bmatrix} \mathbf{d}^{\mathbf{x}}_k \\ \mathbf{d}^{\boldsymbol{\lambda}}_k \\ \mathbf{d}^{\mathbf{z}}_k \end{bmatrix} = - \begin{bmatrix} \nabla f(\mathbf{x}_k) + \nabla \mathbf{h}(\mathbf{x}_k) \boldsymbol{\lambda}_k - \mathbf{z}_k \\ \mathbf{h}(\mathbf{x}_k) \\ \mathbf{X}_k \mathbf{Z}_k \mathbf{1} - \mu_k \mathbf{1} \end{bmatrix}$$

- Three equations, three unknowns: one Newton system per iteration.
- The last row is diagonal, so $\mathbf{d}^{\mathbf{z}}_k$ comes out exactly:

$$\mathbf{d}^{\mathbf{z}}_k = -\mathbf{z}_k + \mu_k \mathbf{X}_k^{-1} \mathbf{1} - \mathbf{X}_k^{-1} \mathbf{Z}_k \, \mathbf{d}^{\mathbf{x}}_k.$$

- Substituting it into the first row leaves a smaller system in $(\mathbf{d}^{\mathbf{x}}_k, \mathbf{d}^{\boldsymbol{\lambda}}_k)$ alone, treated in the next slide.

# The KKT Matrix

With $\mathbf{d}^{\mathbf{z}}$ eliminated, this is what is solved at every iteration:

$$\underbrace{\begin{bmatrix} \nabla^2_{\mathbf{xx}}\mathcal{L}(\mathbf{x}_k, \boldsymbol{\lambda}_k) + \mathbf{X}_k^{-1}\mathbf{Z}_k & \nabla \mathbf{h}(\mathbf{x}_k) \\ \nabla \mathbf{h}(\mathbf{x}_k)^\top & \mathbf{0} \end{bmatrix}}_{\mathbf{K}} \begin{bmatrix} \mathbf{d}_k^{\mathbf{x}} \\ \mathbf{d}_k^{\boldsymbol{\lambda}} \end{bmatrix} = \begin{bmatrix} -\nabla f(\mathbf{x}_k) - \nabla \mathbf{h}(\mathbf{x}_k)\boldsymbol{\lambda}_k + \mu_k \mathbf{X}_k^{-1}\mathbf{1} \\ -\mathbf{h}(\mathbf{x}_k) \end{bmatrix}$$

- $\mathbf{K}$ is large, sparse, and symmetric indefinite; the sparsity pattern is fixed across iterations, only the values change.
- It is also ill-conditioned by construction: as $\mu \to 0$ the barrier entries $\mathbf{X}^{-1}\mathbf{Z}$ split into $O(\mu)$ and $O(1/\mu)$.
- Ill-conditioning is what makes a direct factorization essentially the only reliable option; iterative solvers struggle on this system.
- Symmetric indefinite means that factorization is $\mathbf{LBL}^\top$, with $1 \times 1$ and $2 \times 2$ pivots, not Cholesky. Its pivoting is detailed in Part III.

The Ipopt formulation (Wächter and Biegler, 2006); Nocedal and Wright (2006, Ch. 19).

# Inertia Correction

The factorization is trustworthy only if $\mathbf{K}$ has the right inertia. Two parameters regularize it:

$$\mathbf{K}(\delta_k, \gamma_k) = \begin{bmatrix} \nabla^2_{\mathbf{xx}} \mathcal{L}(\mathbf{x}_k, \boldsymbol{\lambda}_k) + \mathbf{X}_k^{-1}\mathbf{Z}_k + \delta_k \mathbf{I} & \nabla \mathbf{h}(\mathbf{x}_k) \\ \nabla \mathbf{h}(\mathbf{x}_k)^\top & -\gamma_k \mathbf{I} \end{bmatrix}$$

- The Newton direction is productive only when $\mathrm{Inertia}(\mathbf{K}) = (n,\ m_E,\ 0)$: $n$ positive, $m_E$ negative, none zero.
- The procedure: from $\delta_k = \gamma_k = 0$, factorize; if the inertia is right, take the step; otherwise raise the primal $\delta_k$, set the dual $\gamma_k > 0$, and factorize again.
- One factorization per trial, every iteration, and the $\mathbf{LBL}^\top$ factorization reports the inertia for free.

The Ipopt formulation (Wächter and Biegler, 2006); Nocedal and Wright (2006, Ch. 19).

# The Interior-Point Algorithm

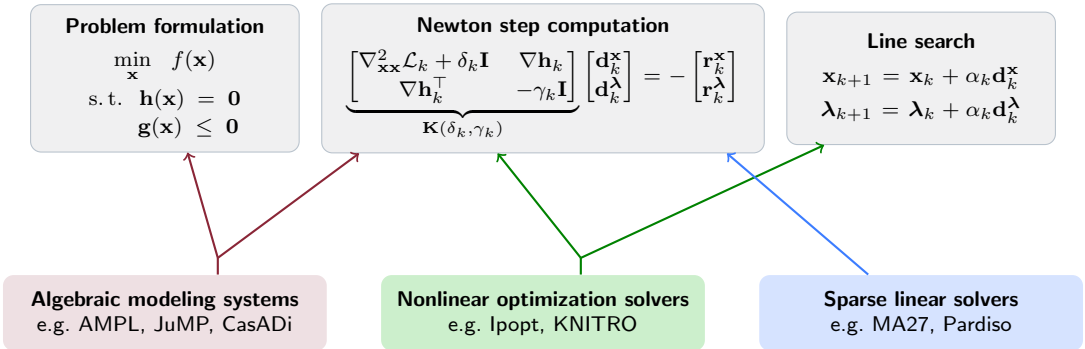
## Primal–dual interior-point method

Given $\mathbf{x}_0 > \mathbf{0}$, $\mathbf{z}_0 > \mathbf{0}$, $\mu_0 > 0$:

1. Evaluate $f, \mathbf{h}$ and their derivatives; assemble the KKT matrix.
2. Factorize $\mathbf{K}(\delta_k, \gamma_k)$, correcting the inertia with $\delta_k$, and solve with those factors for the step: $\mathbf{K}(\delta_k, \gamma_k)\,[\mathbf{d}_k^{\mathbf{x}}; \mathbf{d}_k^{\boldsymbol{\lambda}}] = -\mathbf{r}_k$. The linear solver does both.
3. Line search on a merit function or a *filter* for a step length $\alpha_k$: $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k^{\mathbf{x}}$, $\boldsymbol{\lambda}_{k+1} = \boldsymbol{\lambda}_k + \alpha_k \mathbf{d}_k^{\boldsymbol{\lambda}}$.
4. Once the barrier problem is solved to $\varepsilon_\mu$, decrease $\mu$ and repeat.

Wächter and Biegler (2006).

# The Software Stack



# Modeling Layers

| | **Interface** | **Derivatives** |
|---|---|---|
| AMPL | standalone language | AMPL Solver Library |
| Pyomo | Python | via `.nl`/ASL, or its own |
| JuMP | Julia | expression-graph reverse mode |
| CasADi | Python/MATLAB/C++ | symbolic + code generation |
| JAX | Python | dense AD; sparse $\nabla \mathbf{g}$, $\nabla^2 \mathcal{L}$ |
| PyTorch | Python | via coloring + matrix–vector products |
| **ExaModels** | Julia | **GPU-focused: one kernel/pattern** |

AMPL (Fourer, n.d.); Pyomo (Hart et al., 2017); JuMP (Lubin et al., 2023); CasADi (Andersson et al., 2019); ExaModels (Shin, 2025).

# Solvers and Linear Solvers

| **NLP solvers** | | |
|---|---|---|
| Ipopt | interior point | filter line search |
| KNITRO | interior point | trust region, and an active set |
| **MadNLP** | interior point | **GPU-capable** linear algebra |

| **Sparse symmetric indefinite solvers** | | |
|---|---|---|
| MA27 / MA57 / MA97 | CPU, serial (HSL) | $\mathbf{LBL}^\top$, $1 \times 1$ and $2 \times 2$ pivots |
| Pardiso, MUMPS | CPU, multicore | the same, at larger scale |
| **cuDSS** | GPU | **no** $2 \times 2$ **pivots**: Cholesky, $\mathbf{LDL}^\top$, LU |

The CPU solvers pivot, and get the inertia as a by-product. The GPU solver does not. **Closing that gap is Part III.**

Wächter and Biegler (2006); Byrd et al. (2006); Shin, Pacaud, et al. (2025); Iain S. Duff (2004); NVIDIA (2024).

# Summary of Part I

1. Large-scale nonlinear programs are solved by interior-point methods: the KKT conditions characterize a solution, and the barrier smooths the complementarity condition that makes them hard.
2. Each iteration decomposes into three computations: the evaluation of the NLP functions and their derivatives, the solution of the KKT system (one sparse symmetric indefinite factorization, with an inertia check that conventionally needs numerical pivoting), and the solver's internal computations.

**Part III** moves all three onto the device. Among them, the direct solution of the ill-conditioned KKT system is the main challenge.

Part II

# Optimal Control as a Structured NLP

Direct transcription, orthogonal collocation, structured NLPs

# Outline

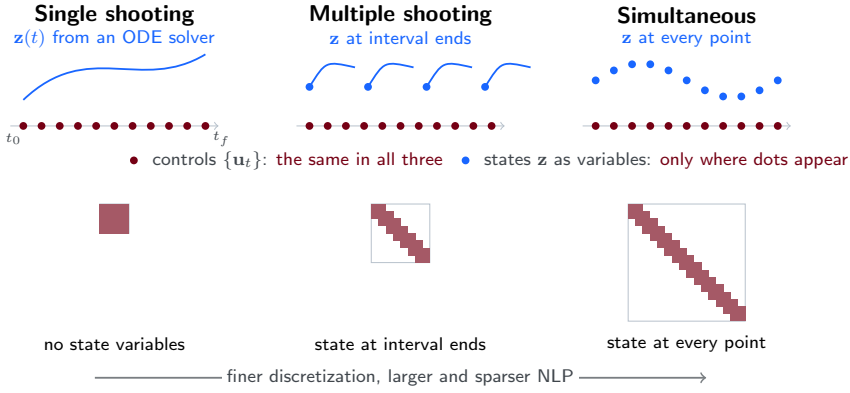
# The Continuous-Time Problem

$$
\begin{aligned}
\min_{\mathbf{z}(\cdot),\,\mathbf{u}(\cdot)} \quad & \varphi(\mathbf{z}(t_f)) + \int_{t_0}^{t_f} \psi(\mathbf{z}(t), \mathbf{u}(t))\, dt \\
\text{s.t.} \quad & \dot{\mathbf{z}}(t) = \mathbf{f}(\mathbf{z}(t), \mathbf{u}(t)), \quad t \in [t_0, t_f] \\
& \mathbf{z}(t_0) = \mathbf{z}_0 \\
& \mathbf{g}(\mathbf{z}(t), \mathbf{u}(t)) \leq \mathbf{0}, \quad t \in [t_0, t_f]
\end{aligned}
$$

- The decision variables are *functions*: an infinite-dimensional problem.
- Two routes to a computable problem:
  - Indirect: "optimize then discretize". Pontryagin gives a two-point boundary value problem.
  - Direct: "discretize then optimize". Finitely many decision variables give a finite-dimensional NLP.
- This workshop is entirely about the direct route: what all the software is built around.

Biegler (2010); Betts (2010).

# Three Direct Approaches




- The simultaneous form suits a GPU because the sequential ODE solve is gone, replaced by algebraic constraints that evaluate in parallel, not because of the sparsity.
- What is given up is adaptive time stepping, which the shooting methods keep.

# Running Example: The Goddard Rocket Problem

A rocket ascends vertically against drag and gravity; maximize the final altitude.

$$\begin{aligned} \max_{h,v,m,\tau} \quad & h(t_f) \\ \text{s.t.} \quad & \dot{h} = v, \qquad \dot{v} = \frac{\tau - D(h,v)}{m} - g(h), \qquad \dot{m} = -\tau/c \\ & h(0) = h^0, \quad v(0) = 0, \quad m(0) = m^0, \quad m(t_f) = m^f \end{aligned}$$

- States: altitude $h$, velocity $v$, mass $m$; control: thrust $\tau$.
- Drag $D(h,v) = D_c v^2 e^{-h_c(h-h^0)/h^0}$ grows with speed; gravity $g(h) = g_0(h^0/h)^2$ decays with altitude.

A classical benchmark from the COPS collection (Dolan and More, 2001); see also Betts (2010).

# The Rocket, Transcribed

The simplest transcription: the trapezoidal rule on a uniform grid $t = 0, \ldots, T$.

$$
\begin{aligned}
\max_{h_t, v_t, m_t, \tau_t, \Delta t} \quad & h_T \\
\text{s.t.} \quad & h_t = h_{t-1} + \tfrac{\Delta t}{2}(v_t + v_{t-1}) \\
& v_t = v_{t-1} + \tfrac{\Delta t}{2}\left( \tfrac{\tau_t - D(h_t, v_t)}{m_t} - g(h_t) + \tfrac{\tau_{t-1} - D(h_{t-1}, v_{t-1})}{m_{t-1}} - g(h_{t-1}) \right) \\
& m_t = m_{t-1} - \tfrac{\Delta t}{2c}(\tau_t + \tau_{t-1}) \\
& h_0 = h^0, \quad v_0 = 0, \quad m_0 = m^0, \quad m_T = m^f
\end{aligned}
$$

- A finite-dimensional NLP with $4(T+1)+1$ variables and $3T+4$ equality constraints.
- Refining the discretization means increasing $T$: the problem grows, but the algebra does not change. Five distinct algebraic patterns, for any $T$.

Biegler (2010, Ch. 10); Betts (2010, Ch. 4).

# Discretization Order and Truncation Error

| | local error | accumulated |
|---|---|---|
| Euler | $\mathcal{O}(h^2)$ | $\mathcal{O}(h)$ |
| Trapezoidal | $\mathcal{O}(h^3)$ | $\mathcal{O}(h^2)$ |
| Collocation, $K$ points | $\mathcal{O}(h^{2K+1})$ | $\mathcal{O}(h^{2K})$ |

- What we want from a discretization: accuracy, stability on stiff dynamics, and a form that is *purely algebraic*.
- Low order decays slowly: halving $h$ doubles $N$, so accuracy is bought with discretization points and the NLP is large-scale by construction.
- A fixed low-order rule is also hard to keep stable on stiff dynamics.
- So the standard practice is to buy accuracy from the scheme instead: high order, and specifically orthogonal collocation.

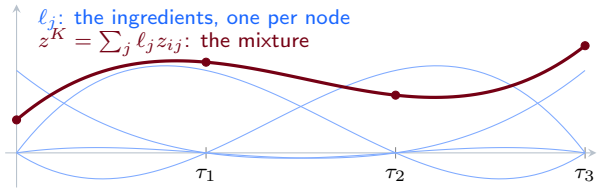
Chapter 10 of Biegler (2010).

# Collocation: Polynomial Representation of the State

Represent the state on an element as a linear combination of polynomials:

$$z(t) \overset{\text{want}}{\approx} z^K(t) = \sum_{j=0}^{K} \ell_j(\tau)\, z_{ij}, \qquad \text{the } z_{ij} \text{ are the unknowns.}$$

- We cannot make $\dot{z}^K = \mathbf{f}(z^K, u)$ hold at every $t$ with finitely many unknowns.
- We therefore impose it at $K$ chosen points, the collocation points, which defines the method.



# Collocation on One Element

On element $i$ of length $h_i$, with $\tau \in [0,1]$, the state is a degree-$K$ Lagrange polynomial through its values at $\tau_0 = 0 < \tau_1 < \cdots < \tau_K$:

$$z^K(t) = \sum_{j=0}^{K} \ell_j(\tau)\, z_{ij}, \qquad \ell_j(\tau) = \prod_{\substack{k=0 \\ k \neq j}}^{K} \frac{\tau - \tau_k}{\tau_j - \tau_k}.$$

Since $\ell_j(\tau_k) = \delta_{jk}$, the $z_{ij}$ *are* the state values at the collocation points. Enforcing the ODE there gives the collocation equations

$$\sum_{j=0}^{K} D_{kj}\, z_{ij} \;=\; h_i\, \mathbf{f}(z_{ik},\, u_{ik}), \qquad D_{kj} = \frac{d\ell_j}{d\tau}(\tau_k).$$

**precomputed constants** **decision variables**

- $D$ is fixed, built once from the $\tau_k$: each equation is a linear combination of unknowns equal to $h_i\, \mathbf{f}(\cdot)$, which is what an NLP solver takes.

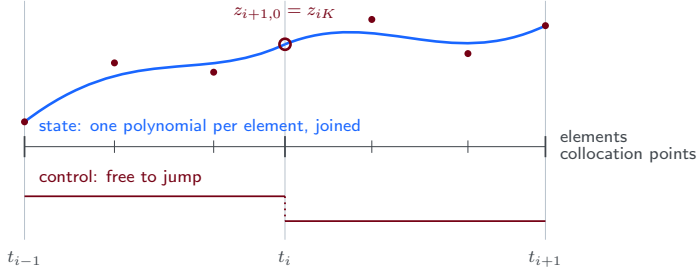
# Connecting the Elements

## Orthogonal collocation on finite elements

Split $[t_0, t_f]$ into $N$ elements. On each, represent the state by a *polynomial* that satisfies the ODE *exactly* at $K$ interior collocation points, and connect the polynomials across elements.

Each element is collocated independently, so nothing yet ties the pieces together:

$$z_{i+1,0} = \sum_{j=0}^{K} \ell_j(1)\, z_{ij}, \quad i = 1, \ldots, N-1, \qquad z_{1,0} = z_0.$$



# The Resulting NLP

Collecting everything, the simultaneous formulation is

$$
\begin{aligned}
\min_{z_{ij},\, u_{ik},\, h_i} \quad & \varphi(z_f) + \sum_{i=1}^{N} h_i \sum_{k=1}^{K} \omega_k \, \psi(z_{ik}, u_{ik}) \\
\text{s.t.} \quad & \textstyle\sum_{j=0}^{K} z_{ij}\, \dot{\ell}_j(\tau_k) = h_i\, \mathbf{f}(z_{ik}, u_{ik}), && i = 1, \ldots, N,\ k = 1, \ldots, K && \text{(collocation)} \\
& z_{i+1,0} = \textstyle\sum_{j=0}^{K} \ell_j(1) z_{ij}, \qquad z_{1,0} = z_0 && i = 1, \ldots, N-1 && \text{(continuity)} \\
& \mathbf{g}(z_{ik}, u_{ik}) \le \mathbf{0} && i = 1, \ldots, N,\ k = 1, \ldots, K && \text{(path)}
\end{aligned}
$$

**precomputed constants** **decision variables**

- Size: $O((n_z K + n_u K)N)$ variables. Refining $N$ or $K$ makes it bigger.
- Path constraints are enforced *pointwise at the collocation points*.

Biegler (2010, Eq. (10.19)).

# Choosing the Collocation Points

Put the $\tau_k$ at the roots of an orthogonal polynomial: the same idea as Gaussian quadrature. The three standard choices differ only in how many element endpoints they include. For $K = 3$:


- All three are stable on stiff dynamics; Radau damps the stiffest modes the hardest, which is why it is the usual default.
- Its right endpoint at $\tau_K = 1$ (circled) also makes continuity trivial: consecutive elements simply share the node.

Biegler (2010, Ch. 10).

# Summary of Part II

1. Direct transcription turns an optimal control problem into an NLP, and orthogonal collocation on finite elements does it at high order, stably for stiff systems, and purely algebraically.
2. What comes out is large, sparse, block-banded, and above all repetitive: a fixed number of algebraic patterns evaluated over many data points.

**Part III**: the repetition buys us parallel derivative evaluation; the regular sparsity buys us GPU factorization, once we solve the pivoting problem.

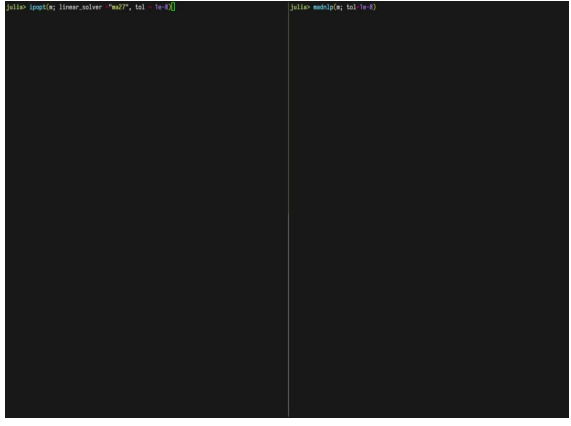
Part III

# GPU Acceleration

SIMD abstraction, pivoting-free linear algebra, condensed-space IPM

# MadNLP Converging on a GPU

AC optimal power flow, PGLIB-OPF case13659_pegase (13 659 buses), to $10^{-8}$: Ipopt + MA27 on the CPU (left) against MadNLP on the GPU (right)



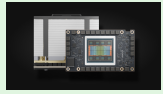

Live solver logs during the solve.

# Outline

# Motivation: Optimization on Accelerated Hardware

**CPU node** 2 ×
AMD EPYC 9965
**45** TFLOP/s FP64
**1.2** TB/s memory

VS.



**GPU node** 8 ×
AMD Instinct MI325X
**654** TFLOP/s FP64
**48** TB/s memory

$\approx 14\times$ compute
$\approx 39\times$ memory
per node

## Why this matters for optimization

A GPU node has $14\times$ the compute and $39\times$ the memory bandwidth. That headroom is what puts larger optimization problems in reach.

Currently the fastest CPU and GPU chips by peak **double-precision** (FP64) throughput, per `waredb.com/ranking/vector/fp64/all`. A node is taken as 2 sockets or 8 accelerators.

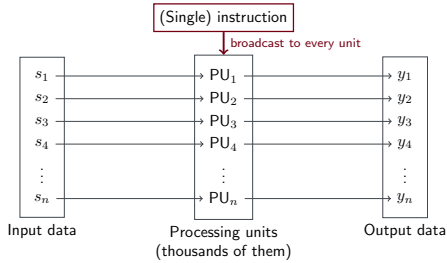
# The Current Landscape

- Factorization-free methods have been the dominant approach on GPUs:
  - PDLP: first-order method for LPs (Applegate et al., 2022; Lu et al., 2025)
  - preconditioned CG within ADMM for convex QPs (Schubiger et al., 2020)
  - augmented Lagrangian + IPM + CG for nonlinear problems (Cao et al., 2016)
- They use only GPU-friendly kernels (`spmv`, `axpy`, `map`, `mapreduce`), which is why they worked first. They scale, but high precision remains hard, and dynamic optimization usually wants it.

**Can we run second-order methods with direct linear algebra on GPUs?**

Applegate et al. (2022); Lu et al. (2025); Schubiger et al. (2020); Cao et al. (2016).

# The Execution Model: SIMD




- A CPU core executes a complicated sequence of instructions serially, and does so very efficiently. Its whole design is spent on finishing one such sequence quickly.
- A GPU keeps thousands of simple sequences running at once, all on the same instruction over different data: SIMD. Each is slow; there are very many.

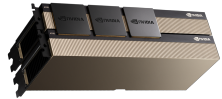
# Strengths and Weaknesses of the Hardware

## Suitable

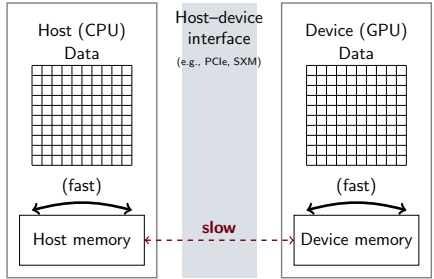
- Uniform work: one operation over many elements (`map`, `axpy`, `mapreduce`, `spmv`).
- Regular access: neighboring threads, neighboring addresses.
- Independent work: dense linear algebra, batched small problems.

## Unsuitable

- Divergent work: different paths in a group are serialized; decisions wait on the numbers.
- Irregular access: indirect, scattered addresses.
- Sequential dependencies: step $k+1$ waiting on step $k$.



# Host and Device Memory

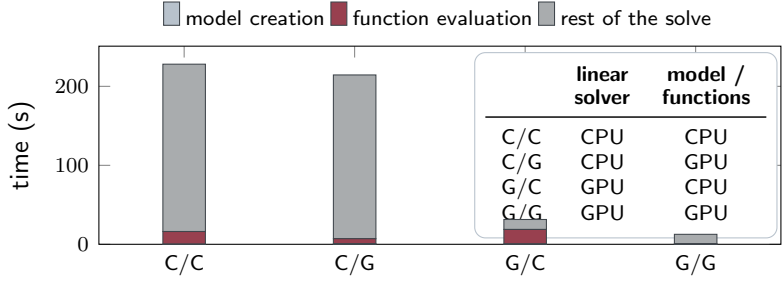



- Device memory has very high bandwidth to the device's own cores. The link to the host is narrow and high-latency by comparison.
- One round trip per iteration cancels the gain, which is why partial porting fails.

**the data must live on the device, and so must the operations**

# The Cost of Partial Porting

AC optimal power flow, PGLIB-OPF 78,478 buses: model and solver on CPU or GPU




- Solver to the GPU: 228 s → 31 s. Model as well: 31 s → 13 s.
- With the model left on the host (G/C), evaluation costs more than in the all-CPU solve: every callback pays a transfer and a barrier.

Reproduced from Shin, Schanen, et al. (2026).

# Algorithmic Redesign for GPUs

- A good CPU solver does not become a GPU solver by recompilation: the parts that make it good on a CPU (adaptive, sequential, data-dependent decisions) are the parts that do not run.
- So the useful question is not "how do I parallelize this loop?" but "what equivalent problem can I solve instead, with no sequential decisions in it?"

**Adapting CPU code into GPU code is not merely a matter of software engineering.**

# The Three Components to Port to the GPU

The same three layers from Part I. Every one of them has to move:

**Problem formulation**

$$\begin{aligned} \min_{\mathbf{x}} \quad & f(\mathbf{x}) \\ \text{s.t.} \quad & \mathbf{h}(\mathbf{x}) = \mathbf{0} \\ & \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \end{aligned}$$

**Newton step computation**

$$\underbrace{\begin{bmatrix} \nabla^2_{\mathbf{xx}}\mathcal{L}_k + \delta_k \mathbf{I} & \nabla \mathbf{h}_k \\ \nabla \mathbf{h}_k^\top & -\gamma_k \mathbf{I} \end{bmatrix}}_{\mathbf{K}(\delta_k, \gamma_k)} \begin{bmatrix} \mathbf{d}_k^{\mathbf{x}} \\ \mathbf{d}_k^{\boldsymbol{\lambda}} \end{bmatrix} = - \begin{bmatrix} \mathbf{r}_k^{\mathbf{x}} \\ \mathbf{r}_k^{\boldsymbol{\lambda}} \end{bmatrix}$$

**Line search**

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{x}_k + \alpha_k \mathbf{d}_k^{\mathbf{x}} \\ \boldsymbol{\lambda}_{k+1} &= \boldsymbol{\lambda}_k + \alpha_k \mathbf{d}_k^{\boldsymbol{\lambda}} \end{aligned}$$

**Algebraic modeling systems**
e.g. AMPL, JuMP, CasADi

**Nonlinear optimization solvers**
e.g. Ipopt, KNITRO

**Sparse linear solvers**
e.g. MA27, Pardiso

# The Three Components to Port to the GPU

The same three layers from Part I. Every one of them has to move:

**Problem formulation**

$$\begin{aligned} \min_{\mathbf{x}} \quad & f(\mathbf{x}) \\ \text{s.t.} \quad & \mathbf{h}(\mathbf{x}) = \mathbf{0} \\ & \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \end{aligned}$$

**Newton step computation**

$$\underbrace{\begin{bmatrix} \nabla^2_{\mathbf{xx}}\mathcal{L}_k + \delta_k \mathbf{I} & \nabla \mathbf{h}_k \\ \nabla \mathbf{h}_k^\top & -\gamma_k \mathbf{I} \end{bmatrix}}_{\mathbf{K}(\delta_k, \gamma_k)} \begin{bmatrix} \mathbf{d}_k^{\mathbf{x}} \\ \mathbf{d}_k^{\boldsymbol{\lambda}} \end{bmatrix} = - \begin{bmatrix} \mathbf{r}_k^{\mathbf{x}} \\ \mathbf{r}_k^{\boldsymbol{\lambda}} \end{bmatrix}$$

**Line search**

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{x}_k + \alpha_k \mathbf{d}_k^{\mathbf{x}} \\ \boldsymbol{\lambda}_{k+1} &= \boldsymbol{\lambda}_k + \alpha_k \mathbf{d}_k^{\boldsymbol{\lambda}} \end{aligned}$$

**Algebraic modeling systems**
e.g. AMPL, JuMP, CasADi

**Nonlinear optimization solvers**
e.g. Ipopt, KNITRO

**Sparse linear solvers**
e.g. MA27, Pardiso

no drop-in for MA27

# The Three Components to Port to the GPU

The same three layers from Part I. Every one of them has to move:

**Problem formulation**

$$\begin{aligned} \min_{\mathbf{x}} \quad & f(\mathbf{x}) \\ \text{s.t.} \quad & \mathbf{h}(\mathbf{x}) = \mathbf{0} \\ & \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \end{aligned}$$

**Newton step computation**

$$\underbrace{\begin{bmatrix} \nabla^2_{\mathbf{xx}}\mathcal{L}_k + \delta_k \mathbf{I} & \nabla \mathbf{h}_k \\ \nabla \mathbf{h}_k^\top & -\gamma_k \mathbf{I} \end{bmatrix}}_{\mathbf{K}(\delta_k, \gamma_k)} \begin{bmatrix} \mathbf{d}_k^{\mathbf{x}} \\ \mathbf{d}_k^{\boldsymbol{\lambda}} \end{bmatrix} = - \begin{bmatrix} \mathbf{r}_k^{\mathbf{x}} \\ \mathbf{r}_k^{\boldsymbol{\lambda}} \end{bmatrix}$$

**Line search**

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{x}_k + \alpha_k \mathbf{d}_k^{\mathbf{x}} \\ \boldsymbol{\lambda}_{k+1} &= \boldsymbol{\lambda}_k + \alpha_k \mathbf{d}_k^{\boldsymbol{\lambda}} \end{aligned}$$

**Algebraic modeling systems**
e.g. AMPL, JuMP, CasADi

**Nonlinear optimization solvers**
e.g. Ipopt, KNITRO

so the algorithm must change

**Sparse linear solvers**
e.g. MA27, Pardiso

no drop-in for MA27

# The Three Components to Port to the GPU

The same three layers from Part I. Every one of them has to move:

**Problem formulation**

$$\begin{aligned} \min_{\mathbf{x}} \quad & f(\mathbf{x}) \\ \text{s.t.} \quad & \mathbf{h}(\mathbf{x}) = \mathbf{0} \\ & \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \end{aligned}$$

**Newton step computation**

$$\underbrace{\begin{bmatrix} \nabla^2_{\mathbf{xx}}\mathcal{L}_k + \delta_k \mathbf{I} & \nabla \mathbf{h}_k \\ \nabla \mathbf{h}_k^\top & -\gamma_k \mathbf{I} \end{bmatrix}}_{\mathbf{K}(\delta_k, \gamma_k)} \begin{bmatrix} \mathbf{d}_k^{\mathbf{x}} \\ \mathbf{d}_k^{\boldsymbol{\lambda}} \end{bmatrix} = - \begin{bmatrix} \mathbf{r}_k^{\mathbf{x}} \\ \mathbf{r}_k^{\boldsymbol{\lambda}} \end{bmatrix}$$

**Line search**

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{x}_k + \alpha_k \mathbf{d}_k^{\mathbf{x}} \\ \boldsymbol{\lambda}_{k+1} &= \boldsymbol{\lambda}_k + \alpha_k \mathbf{d}_k^{\boldsymbol{\lambda}} \end{aligned}$$

**Algebraic modeling systems**
e.g. AMPL, JuMP, CasADi

and only if the structure survives can it be parallel

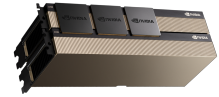
**Nonlinear optimization solvers**
e.g. Ipopt, KNITRO

so the algorithm must change

**Sparse linear solvers**
e.g. MA27, Pardiso

no drop-in for MA27

# The Three Components to Port to the GPU



The same three layers from Part I. Every one of them has to move:

**Problem formulation**

$$\begin{aligned} \min_{\mathbf{x}} \quad & f(\mathbf{x}) \\ \text{s.t.} \quad & \mathbf{h}(\mathbf{x}) = \mathbf{0} \\ & \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \end{aligned}$$

**Newton step computation**

$$\underbrace{\begin{bmatrix} \nabla^2_{\mathbf{xx}}\mathcal{L}_k + \delta_k \mathbf{I} & \nabla \mathbf{h}_k \\ \nabla \mathbf{h}_k^\top & -\gamma_k \mathbf{I} \end{bmatrix}}_{\mathbf{K}(\delta_k, \gamma_k)} \begin{bmatrix} \mathbf{d}_k^{\mathbf{x}} \\ \mathbf{d}_k^{\boldsymbol{\lambda}} \end{bmatrix} = - \begin{bmatrix} \mathbf{r}_k^{\mathbf{x}} \\ \mathbf{r}_k^{\boldsymbol{\lambda}} \end{bmatrix}$$

**Line search**

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{x}_k + \alpha_k \mathbf{d}_k^{\mathbf{x}} \\ \boldsymbol{\lambda}_{k+1} &= \boldsymbol{\lambda}_k + \alpha_k \mathbf{d}_k^{\boldsymbol{\lambda}} \end{aligned}$$

**Algebraic modeling systems**
**ExaModels.jl**

**Nonlinear optimization solvers**
**MadNLP.jl**

**Sparse linear solvers**
**NVIDIA cuDSS**

# The SIMD Abstraction

Many large NLPs are a small number of algebraic patterns, instantiated many times.

$$
\begin{aligned}
\max_{h_t, v_t, m_t, \tau_t, \Delta t} \quad & h_T \\
\text{s.t.} \quad & h_t = h_{t-1} + \tfrac{\Delta t}{2}(v_t + v_{t-1}), && t = 1, \dots, T \\
& v_t = v_{t-1} + \tfrac{\Delta t}{2}\left( \tfrac{\tau_t - D(h_t, v_t)}{m_t} - g(h_t) + \tfrac{\tau_{t-1} - D(h_{t-1}, v_{t-1})}{m_{t-1}} - g(h_{t-1}) \right), && t = 1, \dots, T \\
& m_t = m_{t-1} - \tfrac{\Delta t}{2c}(\tau_t + \tau_{t-1}), && t = 1, \dots, T \\
& h_0 = h^0, \quad v_0 = 0, \quad m_0 = m^0, \quad m_T = m^f
\end{aligned}
$$

Shin, Anitescu, et al. (2024).

# The SIMD Abstraction

Many large NLPs are a small number of algebraic patterns, instantiated many times.

$$\begin{aligned} \min_{\mathbf{x}^\flat \le \mathbf{x} \le \mathbf{x}^\sharp} \quad & \sum_{l \in [L]} \sum_{i \in [I_l]} f^{(l)}\left(\mathbf{x};\, p_i^{(l)}\right) \\ \text{s.t.} \quad & g^{(m)}\left(\mathbf{x};\, q_j^{(m)}\right) = 0, \qquad \forall m \in [M],\ j \in [J_m] \end{aligned}$$

Shin, Anitescu, et al. (2024).

# The SIMD Abstraction

Many large NLPs are a small number of algebraic patterns, instantiated many times.

$$\begin{aligned} \min_{\mathbf{x}^\flat \le \mathbf{x} \le \mathbf{x}^\sharp} \quad & \sum_{l \in [L]} \sum_{i \in [I_l]} f^{(l)}\big(\mathbf{x};\, p_i^{(l)}\big) \\ \text{s.t.} \quad & g^{(m)}\big(\mathbf{x};\, q_j^{(m)}\big) = 0, \qquad \forall m \in [M],\ j \in [J_m] \end{aligned}$$

- $f^{(l)}, g^{(m)}$: the patterns, a handful, known at compile time. $p_i^{(l)}, q_j^{(m)}$: the data, very many. A finer mesh adds data, no new expression.
- One tree walk and one compiled kernel per pattern, run over all the data in parallel. Sparsity from the same tree: no coloring.
- With enough threads, evaluating all the NLP functions is effectively $O(1)$ in wall-clock, nearly independent of problem size.

Shin, Anitescu, et al. (2024).

# The Rocket in ExaModels

```julia
using ExaModels
backend = nothing # or CUDABackend() for the GPU
core = ExaCore(; backend, minimize = false)
@add_var(core, h, 0:T; lvar = h0, start = 1.0)
@add_var(core, v, 0:T; lvar = 0.0, start = 0.0)
@add_var(core, m, 0:T; lvar = mf, uvar = m0, start = m0)
@add_var(core, tau, 0:T; lvar = 0.0, uvar = tau_max, start = tau_max)
@add_var(core, dt, 1; lvar = 0.0, start = 1/T)

@add_obj(core, h[t] for t = T:T)
@add_con(core, h[t] - h[t-1] - 0.5*dt[1]*(v[t] + v[t-1]) for t = 1:T)
@add_con(core, m[t] - m[t-1] + 0.5*dt[1]/c*(tau[t] + tau[t-1]) for t = 1:T)
# ... velocity dynamics, initial and terminal conditions
model = ExaModel(core)
```

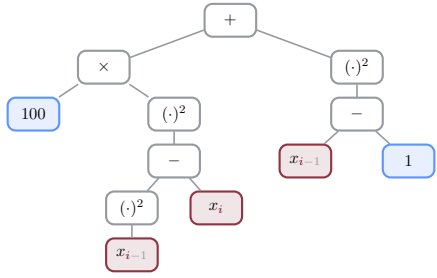
- The `for` stays inside the call. What comes before it is the pattern, what comes after is the iterator; they are stored separately and never flattened.
- A `for` loop *around* the call emits $T$ unrelated objects, and the repetition is lost.

Shin, Schanen, et al. (2026); ExaModels.jl (Shin, 2025); abstraction from Shin, Anitescu, et al. (2024).

# The Parametrized Expression Tree

One constraint or objective term of the model, as ExaModels stores it:

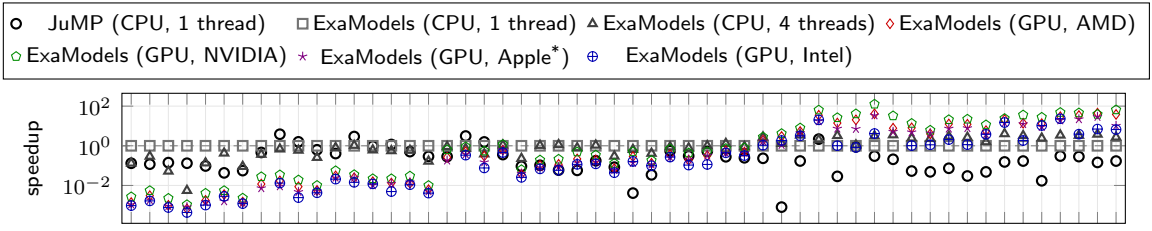
$$100\left(x_{i-1}^2 - x_i\right)^2 + \left(x_{i-1} - 1\right)^2, \qquad i = 2, \ldots, N$$




- $i$ is a placeholder: one tree for all $N - 1$ terms.
- Derivative kernel and sparsity, once per tree: evaluation is a parallel loop.

Shin, Schanen, et al. (2026); ExaModels.jl (Shin, 2025); abstraction from Shin, Anitescu, et al. (2024).

# Performance of the Modeling Layer




Speedup over a single CPU thread on an AMD EPYC 9474F; COPS suite, per-instance evaluation times. *the Apple runs are single precision. (Shin, Schanen, et al., 2026)

# The Fixed Cost of a Device Call

| Hessian evaluation | Small (16) $\mathrm{nnz} < 10^3$ | Medium (18) $10^3 \le \mathrm{nnz} < 10^5$ | Large (20) $10^5 \le \mathrm{nnz}$ |
|---|---|---|---|
| CPU, 1 thread (EPYC 9474F) | 1.9 $\mu$s | 50.2 $\mu$s | 2.83 ms |
| GPU (NVIDIA B200) | 71.9 $\mu$s | 75.6 $\mu$s | 92.9 $\mu$s |
| GPU speedup | $0.0\times$ | $0.7\times$ | $30\times$ |

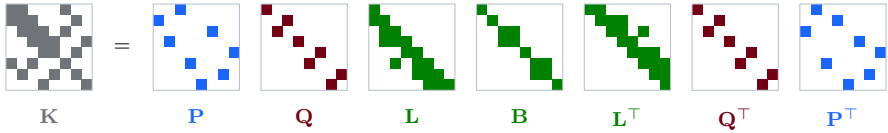
**With enough threads the GPU evaluates the whole NLP as essentially one operation: the time barely moves as the problem grows a thousandfold ($72 \rightarrow 93\,\mu$s), while the CPU scales with the work. The fixed per-call latency is the price, so make few, large device calls.**

Data from Shin, Schanen, et al. (2026): ExaModels `hess_coord!`, COPS suite, shifted geometric mean per size class ($\sigma = 10^{-5}$ s).

# Numerical Pivoting

From Part I: every iteration solves the KKT system, and that solve is where the time goes. It means factorizing $\mathbf{K}$, which in the sparse case carries two permutations:

$$\mathbf{K} = \mathbf{P}\,\mathbf{Q}\,\mathbf{L}\mathbf{B}\mathbf{L}^{\top}\,\mathbf{Q}^{\top}\mathbf{P}^{\top}$$

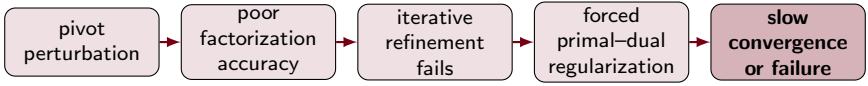



- $\mathbf{P}$: fill-reducing reordering, from the sparsity pattern, computed once and reused.
- $\mathbf{Q}$: numerical pivoting, chosen *on the fly*. The obstacle: irregular access, no parallelism.
- $\mathbf{B}$ is block diagonal, $1 \times 1$ and $2 \times 2$: where the inertia is read off.

I. S. Duff and Reid (1983); Wächter and Biegler (2006).

# Pivot Perturbation and Pivoting-Free Classes

- GPU sparse direct solvers, cuDSS among them, cannot do $\mathbf{LBL}^\top$: no adequately robust numerical pivoting for indefinite systems.
- Run a vanilla IPM on one anyway and what you get is pivot perturbation, which is not free:




- Fine on small problems, failing frequently beyond roughly $10^4$ variables.
- Two classes need no pivoting at all: SPD (Cholesky exists and is stable) and SQD (the factorization exists for *any* symmetric permutation).

Świrydowicz et al. (2022); Vanderbei (1995); Montoison, Pacaud, Shin, et al. (2025).

# Condensation

For a problem with inequalities only,

$$\min_{\mathbf{x}} \; f(\mathbf{x}) \quad \text{s.t. } \mathbf{g}(\mathbf{x}) \le \mathbf{0},$$

the KKT system of Part I, written with slacks, has marked blocks that are diagonal, so $\mathbf{d}_s$ and $\mathbf{d}_\lambda$ eliminate by exact substitution:

$$\begin{bmatrix} \nabla^2_{\mathbf{xx}}\mathcal{L} + \delta_p \mathbf{I} & & \nabla \mathbf{g}(\mathbf{x}) \\ & \mathbf{S}^{-1}\boldsymbol{\Lambda} & -\mathbf{I} \\ \nabla \mathbf{g}(\mathbf{x})^\top & -\mathbf{I} & -\delta_d \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{d}_x \\ \mathbf{d}_s \\ -\mathbf{d}_\lambda \end{bmatrix} = - \begin{bmatrix} \nabla f(\mathbf{x}) - \nabla \mathbf{g}(\mathbf{x})\boldsymbol{\lambda} \\ \boldsymbol{\Lambda}\mathbf{1} - \mu \mathbf{S}^{-1}\mathbf{1} \\ \mathbf{g}(\mathbf{x}) - \mathbf{s} \end{bmatrix}$$

$$\iff \quad \underbrace{\left( \nabla^2_{\mathbf{xx}}\mathcal{L} + \delta_p \mathbf{I} + \nabla \mathbf{g}(\mathbf{x}) \left( \delta_d \mathbf{I} + \boldsymbol{\Lambda}^{-1}\mathbf{S} \right)^{-1} \nabla \mathbf{g}(\mathbf{x})^\top \right)}_{\mathbf{K}_{\text{cond}}} \mathbf{d}_x = -\mathbf{r}_{\text{cond}}$$

- Nothing is approximated, and $\mathbf{K}_{\text{cond}}$ is $n \times n$ rather than $(n + 2m) \times (n + 2m)$.
- Positive definite exactly when the original system had the right inertia, so a successful Cholesky is the certificate: no pivoting.

Shin, Anitescu, et al. (2024); Pacaud et al. (2024).

# Equality Constraints and Condensation

Optimal control problems are *made* of equality constraints: every collocation equation, every continuity condition.

$$\begin{aligned} &\min_{\mathbf{x}} f(\mathbf{x}) \\ &\text{s.t. } \mathbf{h}(\mathbf{x}) = \mathbf{0} \\ &\quad \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \end{aligned}$$

$$\begin{bmatrix} \mathbf{K}_{\text{cond}} + \delta_p \mathbf{I} & \nabla \mathbf{h}(\mathbf{x}) \\ \nabla \mathbf{h}(\mathbf{x})^\top & -\delta_d \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{d}_x \\ \mathbf{d}_\lambda \end{bmatrix} = - \begin{bmatrix} \mathbf{r}_x \\ \mathbf{r}_\lambda \end{bmatrix}$$

- $\mathbf{h}(\mathbf{x}) = \mathbf{0}$ has no slack, so no diagonal block and no elimination: condensation leaves the equality block behind and the system stays indefinite.

**Option 1: Lifted KKT**
relax the equalities into narrow inequalities, then condense everything

**Option 2: Hybrid KKT**
keep the equalities, and handle the resulting Schur complement iteratively

# Option 1: Lifted KKT

Step 1: lift and relax, with $\tau$ no larger than the solver tolerance:

$$\mathbf{h}(\mathbf{x}) = \mathbf{0} \qquad \Longrightarrow \qquad \mathbf{h}(\mathbf{x}) + \mathbf{s} = \mathbf{0} \quad \text{and} \quad -\tau \leq \mathbf{s} \leq \tau$$

- *Every* constraint now has a slack, so condensation applies throughout and the condensed matrix is SPD under the usual inertia condition.
- cuDSS factorizes it by Cholesky on a pattern fixed in advance: no pivoting, no excessive regularization.

## What you are paying

The answer solves a relaxed problem: as accurate as the tolerance you asked for, never more accurate than $\tau$, and the condensed matrix grows ill-conditioned as $\mu \to 0$.

Shin, Anitescu, et al. (2024).

# Option 2: Hybrid KKT

$$\begin{bmatrix} \mathbf{K}_{\text{cond}} + \delta_p \mathbf{I} & \nabla \mathbf{h}(\mathbf{x}) \\ \nabla \mathbf{h}(\mathbf{x})^\top & -\delta_d \mathbf{I} \end{bmatrix} \xrightarrow[\text{(solution unchanged)}]{+\rho\, \nabla \mathbf{h} \nabla \mathbf{h}^\top} \begin{bmatrix} \mathbf{K}_{\text{cond}} + \delta_p \mathbf{I} + \rho\, \nabla \mathbf{h}(\mathbf{x}) \nabla \mathbf{h}(\mathbf{x})^\top & \nabla \mathbf{h}(\mathbf{x}) \\ \nabla \mathbf{h}(\mathbf{x})^\top & -\delta_d \mathbf{I} \end{bmatrix}$$

- The $(1,1)$ block, call it $\mathbf{K}_\rho$, is SPD for $\rho$ large enough, iff the inertia is correct, so it is Cholesky-factorizable, once, without pivoting.
- Then solve the Schur complement in $\mathbf{d}_\lambda$ iteratively (CG):

$$\Big( \underbrace{\nabla \mathbf{h}(\mathbf{x})^\top \mathbf{K}_\rho^{-1} \nabla \mathbf{h}(\mathbf{x})}_{\to\ \rho^{-1}\mathbf{I} \text{ as } \rho \to \infty} + \delta_d \mathbf{I} \Big)\, \mathbf{d}_\lambda = \mathbf{r}'_\lambda$$

- The limit is the mechanism: large $\rho$ drives the Schur complement toward $\rho^{-1}\mathbf{I}$, so its spectrum clusters and CG converges in very few iterations.
- No relaxation, no accuracy ceiling; the price is an inner CG loop and a parameter $\rho$.

Golub and Greif (2003); Regev et al. (2023); Pacaud et al. (2024).

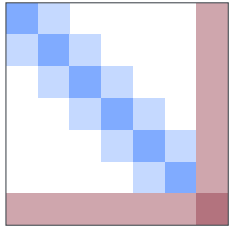
# Lifted KKT and Hybrid KKT Compared

| | **Lifted KKT** | **Hybrid KKT** |
|---|---|---|
| Equalities | relaxed to $\pm\tau$ | kept exactly |
| System solved | one SPD system, $n \times n$ | SPD system + CG on Schur complement |
| Linear solve | purely direct | direct + iterative |
| Accuracy | limited by $\tau$ | not limited |
| Extra parameter | $\tau$ | $\rho$ |
| Main cost | ill-conditioning as $\mu \downarrow 0$ | ill-conditioning as $\rho \uparrow$; CG iterations |

- Both are implemented in MadNLP.jl, and both sit on cuDSS. The hands-on uses Lifted KKT, MadNLP's default on the GPU.
- Neither is uniformly better. Lifted KKT is the simpler and usually the faster at moderate tolerance; Hybrid KKT is what you reach for when you need tight convergence.

# Block Tridiagonal Structure of the KKT Matrix

Transcribing a horizon couples stage $t$ only to $t \pm 1$, so the KKT matrix is block tridiagonal (bordered, if there are variables shared across the horizon such as $\Delta t$):



blue: one block per stage, plus its couplings

red: the border, from horizon-wide variables

- A sequential sweep down the diagonal is the Thomas algorithm: $N$ steps, no parallelism. Cyclic reduction eliminates every other stage and recurses: $\log N$ levels of independent block work.
- Each level is then many identical small dense factorizations, dispatched as a single batched call, which suits the hardware.

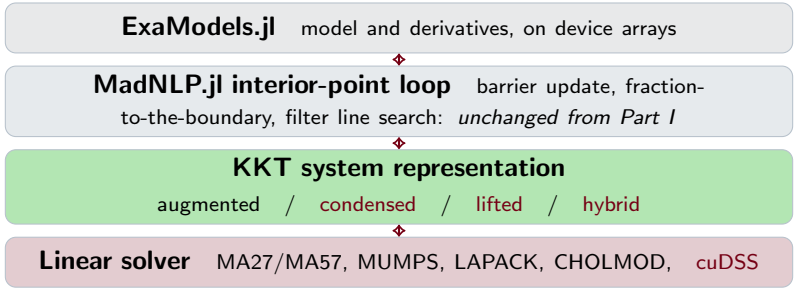
Jin et al. (2026); Schwan et al. (2026).

# Parallel Block Tridiagonal Factorization

- cuDSS is structure-agnostic: given a matrix, not a horizon. A good ordering recovers much of the parallelism, but the schedule is not exposed.
- Exploiting the structure explicitly is a live research direction:
  - Jin et al. (2026): recursive Schur-complement reduction, batched BLAS/LAPACK, NVIDIA and AMD.
  - Schwan et al. (2026): nested dissection, Cholesky from $\mathcal{O}(Nn^3)$ to $\mathcal{O}(\log_2 N\, n^3)$.

**The first of those is in this congress: Wednesday 26 August, 14:30, Room 211, session Numerical Methods for Optimal Control.**

# Implementation in MadNLP.jl




Shin, Pacaud, et al. (2025).

# The COPS Benchmark

| Tol | Solver | Small $\text{nnz} < 2^{18}$ | | Medium $2^{18} \leq \text{nnz} < 2^{20}$ | | Large $2^{20} \leq \text{nnz}$ | | Total | |
|---|---|---|---|---|---|---|---|---|---|
| | | Solved | Time | Solved | Time | Solved | Time | Solved | Time |
| $10^{-4}$ | MadNLP (GPU) | **13** | **0.8665** | **15** | **4.8665** | **16** | **3.8194** | **44** | **3.2314** |
| | Ipopt (CPU) | **13** | 5.2315 | **15** | 15.9701 | 15 | 45.8411 | 43 | 19.2243 |
| $10^{-8}$ | MadNLP (GPU) | **13** | **0.8575** | **16** | **1.5572** | **16** | **8.3549** | **45** | **3.3797** |
| | Ipopt (CPU) | **13** | 5.9413 | **15** | 17.6758 | 15 | 40.8639 | 43 | 19.2999 |

- Benchmark: COPS (Dolan and More, 2001), largely dynamic optimization problems, discretized as in Part II.
- Hardware: NVIDIA GV100 / Intel Xeon Gold 6130. Time: SGM10, 900 s limit.

Table reproduced from Montoison, Pacaud, Shin, et al. (2025).

# Other Routes to a Pivoting-Free Solver

- Lifted KKT and Hybrid KKT are two points in a design space, not the only two.
- MadNCL: an augmented-Lagrangian algorithm. The quadratic penalty term is inherently GPU-friendly; it uses the nonlinearly-constrained Lagrangian formulation together with the condensation ideas above (Montoison, Pacaud, Saunders, et al., 2025).
- MadIPM: for convex quadratic programs; uses regularization to avoid pivoting altogether (Montoison, Pacaud, Shin, et al., 2025).
- Factorizing the augmented system directly, with no pivoting and no condensation, is under active investigation.
- Further out: multi-GPU for problems that exceed device memory, and mixed precision: factorize in single, refine in double.

Montoison, Pacaud, Saunders, et al. (2025); Montoison, Pacaud, Shin, et al. (2025).

# Implications for Optimal Control

- The simultaneous approach suits this hardware: transcribing the whole horizon removes the sequential ODE solve and leaves something large, sparse, block tridiagonal and repetitive.
- On COPS, the largest instances go from 45.8 s to 3.8 s at $10^{-4}$, about $12\times$, and the GPU solves 16 of them where the CPU solves 15.
- The route is not settled: Lifted KKT, Hybrid KKT, null-space and factorization-free methods all trade differently. Which wins for a given problem class remains open.

**The software is installable today: ExaModels.jl, MadNLP.jl, cuDSS. What we would like from this workshop is for large-scale optimal control to start using it.**

Part IV

# Hands-On Sessions

Julia on the GPU; optimal control and parameter estimation end to end

# Outline

- Part I: Nonlinear Optimization
- Part II: Optimal Control as a Structured NLP
- Part III: GPU Acceleration
- **Part IV: Hands-On Sessions**

# References I


Andersson, Joel A. E. et al. (Mar. 2019). "CasADi: A Software Framework for Nonlinear Optimization and Optimal Control". In: *Mathematical Programming Computation* 11.1, pp. 1–36.

Applegate, David et al. (Jan. 2022). *Practical Large-Scale Linear Programming Using Primal-Dual Hybrid Gradient*. arXiv: 2106.04756 [math].

Betts, John T. (Jan. 2010). *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming*. Second. Society for Industrial and Applied Mathematics.

Biegler, Lorenz T. (2010). *Nonlinear Programming: Concepts, Algorithms, and Applications to Chemical Processes*. MOS-SIAM Series on Optimization 10. Philadelphia, Pa: SIAM [u.a.]

Byrd, Richard H. et al. (2006). "Knitro: An Integrated Package for Nonlinear Optimization". In: *Large-Scale Nonlinear Optimization*. Ed. by G. Di Pillo and M. Roma. Boston, MA: Springer US, pp. 35–59.

Cao, Yankai et al. (2016). "An Augmented Lagrangian Interior-Point Approach for Large-Scale NLP Problems on Graphics Processing Units". In: *Computers & Chemical Engineering* 85, pp. 76–83.

Dolan, E. D. and J. J. More (Jan. 2001). *Benchmarking Optimization Software with COPS*. Tech. rep. ANL/MCS-TM-246. Argonne National Lab., IL (US).

Duff, I. S. and J. K. Reid (Sept. 1983). "The Multifrontal Solution of Indefinite Sparse Symmetric Linear". In: *ACM Transactions on Mathematical Software* 9.3, pp. 302–325.

Duff, Iain S. (June 2004). "MA57—a Code for the Solution of Sparse Symmetric Definite and Indefinite Systems". In: *ACM Trans. Math. Softw.* 30.2, pp. 118–144.

Fourer, Robert (n.d.). "AMPL: A Mathematical Programming Language". In: ().

Golub, Gene H. and Chen Greif (Jan. 2003). "On Solving Block-Structured Indefinite Linear Systems". In: *SIAM Journal on Scientific Computing* 24.6, pp. 2076–2092.

Hart, William E. et al. (2017). *Pyomo — Optimization Modeling in Python*. Vol. 67. Springer Optimization and Its Applications. Cham: Springer International Publishing.

Jin, David et al. (Aug. 2026). "Harnessing Batched GPU Kernels for Solutions of Block Tridiagonal Systems". In: *23rd IFAC World Congress*. Paper WeB13.5, session "Numerical Methods for Optimal Control"; preprint arXiv:2509.03015. Busan, Republic of Korea.

Lu, Haihao et al. (July 2025). *cuPDLP+: A Further Enhanced GPU-Based First-Order Solver for Linear Programming*. arXiv: 2507.14051 [math].

Lubin, Miles et al. (Sept. 2023). "JuMP 1.0: Recent Improvements to a Modeling Language for Mathematical Optimization". In: *Mathematical Programming Computation* 15.3, pp. 581–589.

Montoison, Alexis, François Pacaud, Michael Saunders, et al. (Oct. 2025). *MadNCL: A GPU Implementation of Algorithm NCL for Large-Scale, Degenerate Nonlinear Programs*. arXiv: 2510.05885 [math].

Montoison, Alexis, François Pacaud, Sungho Shin, et al. (Nov. 2025). "GPU Implementation of Second-Order Linear and Nonlinear Programming Solvers". In: *NeurIPS Workshop on GPU-Accelerated and Scalable Optimization*.

# References II


Nocedal, Jorge and Stephen J. Wright (2006). *Numerical Optimization*. 2nd ed. Springer Series in Operations Research. New York: Springer.

NVIDIA (2024). *NVIDIA cuDSS (Preview): A High-Performance CUDA Library for Direct Sparse Solvers — NVIDIA cuDSS Documentation*. https://docs.nvidia.com/cuda/cudss/index.html.

Pacaud, François et al. (May 2024). *Condensed Interior-Point Methods for Scalable Nonlinear Programming on GPUs*. To appear in Mathematical Programming Computation. arXiv: 2405.14236 [math].

Regev, Shaked et al. (Mar. 2023). "HyKKT: A Hybrid Direct-Iterative Method for Solving KKT Linear Systems". In: *Optimization Methods and Software* 38.2, pp. 332–355.

Schubiger, Michel et al. (Oct. 2020). "GPU Acceleration of ADMM for Large-Scale Quadratic Programming". In: *Journal of Parallel and Distributed Computing* 144, pp. 55–67.

Schwan, Roland et al. (Jan. 2026). *GPU-Accelerated Cholesky Factorization of Block Tridiagonal Matrices*. arXiv: 2601.03754 [math.OC].

Shin, Sungho (Feb. 2025). *ExaModels.jl*. Exanauts.

Shin, Sungho, Mihai Anitescu, et al. (Nov. 2024). "Accelerating Optimal Power Flow with GPUs: SIMD Abstraction of Nonlinear Programs and Condensed-Space Interior-Point Methods". In: *Electric Power Systems Research* 236, p. 110651.

Shin, Sungho, François Pacaud, et al. (Apr. 2025). *MadNLP.jl*. MadNLP.

Shin, Sungho, Michel Schanen, et al. (Aug. 2026). *ExaModels.jl: An Algebraic Modeling System for Nonlinear Programming on GPUs*. arXiv: 2608.16265 [math.OC].

Świrydowicz, Kasia et al. (July 2022). "Linear Solvers for Power Grid Optimization Problems: A Review of GPU-accelerated Linear Solvers". In: *Parallel Computing* 111, p. 102870.

Vanderbei, Robert J. (Feb. 1995). "Symmetric Quasidefinite Matrices". In: *SIAM Journal on Optimization* 5.1, pp. 100–113.

Wächter, Andreas and Lorenz T. Biegler (Mar. 2006). "On the Implementation of an Interior-Point Filter Line-Search Algorithm for Large-Scale Nonlinear Programming". In: *Mathematical Programming* 106.1, pp. 25–57.