A Parallelized, Adam-Based Solver for Reserve and Security Constrained AC Unit Commitment

Samuel Chevalier
Department of Electrical Engineering
University of Vermont
Burlington, Vermont, USA
schevali@uvm.edu

***Abstract*—Power system optimization problems which include the nonlinear AC power flow equations require powerful and robust numerical solution algorithms. Within this sub-field of nonlinear optimization, interior point methods have come to dominate the solver landscape. Over the last decade, however, a number of efficient numerical optimizers have emerged from the field of Machine Learning (ML). One algorithm in particular, Adam, has become the optimizer-of-choice for a massive percentage of ML training problems (including, e.g., the training of GPT-3), solving some of the largest unconstrained optimization problems ever conceived of. Inspired by such progress, this paper designs a parallelized Adam-based numerical solver to overcome one of the most challenging power system optimization problems: security and reserve constrained AC Unit Commitment. The resulting solver, termed `QuasiGrad`, recently competed in the third ARPA-E Grid Optimization (GO3) competition. In the day-ahead market clearing category (with systems ranging from 3 to 23,643 buses over 48 time periods), `QuasiGrad`'s aggregated market surplus scores were within 5% of the winningest market surplus scores. The `QuasiGrad` solver is now released as an open-source Julia package: `QuasiGrad.jl`. The internal gradient-based solver (Adam) can easily be substituted for other ML-inspired solvers (e.g., AdaGrad, AdaDelta, RMSProp, etc.). Test results from large experiments are provided.**

***Index Terms*—AC unit commitment, Adam, optimal power flow, market surplus, mixed-integer, security constraints**

I. Introduction

The first two Advanced Research Projects Agency–Energy (ARPA-E) Grid Optimization competitions (GO1, GO2) focused on various flavors of the Security Constrained Optimal Power Flow (SCOPF) problems [1]–[3]. Within these competitions, the most successful numerical solution techniques leveraged interior point solvers [1]. The winningest approach in GO2, for example, used Ipopt via the Gravity modeling framework [4]. The third Grid Optimization competition (GO3), which recently concluded, focused on multi-period dynamic markets. More specifically, the GO3 market clearing problem incorporated security and reserve constrained AC unit commitment (ACUC) with topology optimization, and it asked competitors to maximize a social surplus function within the context of real-time, day-ahead, and week-look-ahead markets.

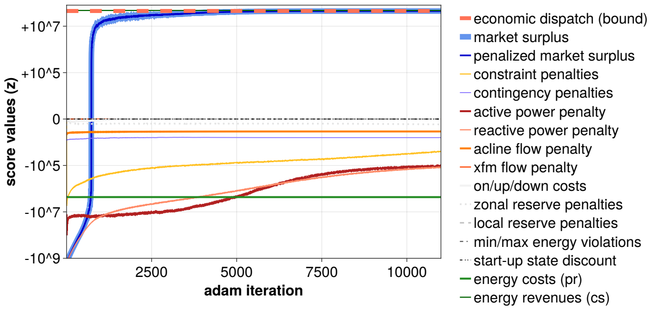

Fig. 1: Illustrated is an Adam solve on a 617-bus, 18 time period, real-time market clearing test case (integers relaxed); this system is initialized with a copper pate economic dispatch solution (LP), whose upper bound is given as the orange dashed line. Within several thousand iterations, Adam finds an AC network solution to within 1% of this global bound. A single back-propagation (i.e., gradient calculation) through this entire system, include all 18 × 562 contingencies, takes ∼24ms when parallelized on 6 CPU threads.

As designed, the GO3 test-cases are generally intractable: the largest case, containing 23,643, buses, has 26,870 separate contingencies and 33,739 switchable lines/transformers *at each time step*. Reliably solving DC unit commitment (DCUC) on such a system is challenging in itself, much less ACUC. In order to overcome such levels of computational intractability, this paper, and the `QuasiGrad` solver which it proposes, leverages techniques designed to solve large scale optimization problems from another community: machine learning (ML). The popularized GPT-3 model, for example, contains 175 billion tunable model parameters (i.e., optimizable *decision variables*) [5]. A plethora of gradient-based optimization tools have come from the ML community for solving such problems [6], but Adam has emerged as the clear dominant solver. GPT-3 was trained with Adam [5], as were other large commercial models.

While Adam has been a successful tool for solving large-scale nonlinear programming (NLP) problems, it has also recently been used to solve sub-problems in massive Mixed-Integer Linear Programming (MILP) Branch-and-Bound (BaB) problems. The α, β-CROWN solver [7], which has won the most recent International Verification of Neural Networks Competitions (VNN-COMPs) [8], uses a GPU-accelerated Adam solver to verify the performance of ex-

Submitted to the 23rd Power Systems Computation Conference (PSCC 2024). This work was supported by the HORIZON-MSCA-2021 Postdoc Fellowship Program, Project #101066991 – TRUST-ML.

tremely large neural networks (whose verification problems often reformulate directly into MILPs). Notably, α,β-CROWN was able to win these highly competitive competitions by carefully leveraging the computational might of modern GPUs along with the inherent *parallelizability* of Adam. Like other gradient-based solvers, Adam requires a massive number of derivative computations – each of these "backpropagations" can be efficiently computed in parallel, greatly accelerating the iterations.

Building on these successes, this paper introduces the `QuasiGrad` solver for solving large scale power system optimization problems (specifically, the one formulated for GO3). The core numerical workhorse under `QuasiGrad`'s hood is Adam. Despite its prowess, Adam needs help: to get this help, `QuasiGrad` leverages a number of other innovations to aid in solving the large-scale reserve and security constrained ACUC market clearing problem. Four of these contributions are summarized before:

1) We reformulate, and explicitly backpropagate through, the GO3 reserve + security constrained ACUC problem.
2) Using a preconditioned conjugate gradient (Kyrlov subspace) method, we stochastically select, numerically solve, and backpropagate through, parallelized security constraints. The computed gradients are passed to Adam.
3) We design a series of parallelized projection methodologies, including a guaranteed feasible ramp-constrained power flow solver, in order to exploit parallel computational resources and accelerate the convergence of Adam.
4) We release `QuasiGrad.jl`, an open-source package available in the Julia ecosystem [9].

In Sec. II, we introduce the Mixed-Integer NLP (MINLP) proposed in GO3, and we transform the problem to make it amenable for gradient-based solvers. In Sec III, we propose the full `QuasiGrad` solver, and in Sec. IV, we provide simulated test results. Conclusions are presented in Sec. V.

II. Problem Formulation

This section introduces the MINLP designed by the GO3 planning team. This MINLP is then transformed into a formulation which is amenable for gradient based solvers (i.e., Adam) to directly interact with. Regarding notation, lower case variables are scalars (e.g., x), bold lower case variables are vectors (e.g., $\boldsymbol{x}$), and upper case variables are generally matrices (e.g., X). In all cases, we closely follow the notation prescribed in the official GO3 problem formulation [10].

A. Motivation for Reformulation

Training problems in ML are often formulated as

$$\min_{\boldsymbol{x}} \ \mathcal{L}(\boldsymbol{x}), \tag{1}$$

where $\mathcal{L}(\boldsymbol{x})$ is the canonical "loss" function relating input/output data mappings, and $\boldsymbol{x}$ is an unconstrained vector of model parameters. The `QuasiGrad` solver is designed around the idea of transforming the GO3 MINLP problem into a form which approximates (1). As formulated in [10], however, The GO3 MINLP contains three challenging complications:

1) equality constraints,
2) inequality constraints,
3) integer variables.

While directly penalizing these constraints/integrality requirements and pushing them into the loss function may seem to be an obvious solution, care must be taken to ensure the penalization procedure does not introduce unnecessary, and potentially intractable (in the case of contingency violations), loss error. For example, consider the following NLP:

$$\min_{\boldsymbol{x},\boldsymbol{y}} \quad f(\boldsymbol{x},\boldsymbol{y}) \tag{2a}$$

$$\text{s.t.} \quad A\boldsymbol{x} = \boldsymbol{y}. \tag{2b}$$

The *best* reformulation substitutes the equality $\boldsymbol{y} \leftarrow A\boldsymbol{x}$ directly, such that $f(\boldsymbol{x}, A\boldsymbol{x})$ is minimized independently. Naive constraint penalization on the other hand, via $\mathcal{L} = f(\boldsymbol{x},\boldsymbol{y}) + \lambda \left\| A\boldsymbol{x} - \boldsymbol{y} \right\|$, is problematic for three reasons:

1) the "$\boldsymbol{y}$" variable is unnecessarily retained;
2) the optimizer must expend implicit computational resource in approximating $A\boldsymbol{x} \approx \boldsymbol{y}$;
3) finally, and most subtly, assume a given numerical solution $\boldsymbol{x}^*$ to (2) is evaluated by computing $f(\boldsymbol{x}^*, A\boldsymbol{x}^*)$. In this case, any effort spent by the optimizer to minimize the penalty term $\lambda \left\| A\boldsymbol{x} - \boldsymbol{y} \right\|$ was a "numerical distraction", since it had no explicit effect on solution quality. In summary, a successful reformulation will "find $\boldsymbol{x}$ and compute $\boldsymbol{y}$" rather than "find $\boldsymbol{x}$ and find $\boldsymbol{y}$".

In light of these observations, the following subsections rewrite the GO3 MINLP into a form which is similar to (1) and, thus, amenable for gradient based solvers (i.e., Adam). In doing so, we eliminate all unnecessary intermediate variables (i.e., "$\boldsymbol{y}$" terms, which we call "auxiliary" variables), and we ensure the gradient solver expends its computational resource computing numerical variable values which actually influence solution quality (i.e., "$\boldsymbol{x}$" terms, which we call "basis" variables).

B. MINLP Reformulation

We begin by stating the following MINLP, which represents an exact transformation[1] of the GO3 security and reserve constrained ACUC problem [10]. This transformation writes all auxiliary variables $\boldsymbol{y}$ as an explicit function of basis variables $\boldsymbol{x}$. Notably, we have left the contingency constraint function, $\boldsymbol{h}_{\rm ctg}(\cdot)$, in place, since it will receive special consideration:

$$\min_{\boldsymbol{x}_d,\boldsymbol{x}_c} \quad z^{\rm ms}(\boldsymbol{x}_c,\boldsymbol{x}_d,\boldsymbol{y}) + z^{\rm ctg} \tag{3a}$$

$$\text{s.t.} \quad \boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}_c,\boldsymbol{x}_d) \tag{3b}$$

$$\boldsymbol{0} = \boldsymbol{h}_{\rm ctg}(\boldsymbol{x}_c,\boldsymbol{x}_d,\boldsymbol{\theta}_k) \tag{3c}$$

$$A_c\boldsymbol{x}_c + A_d\boldsymbol{x}_d \geq \boldsymbol{0} \tag{3d}$$

$$\underline{\boldsymbol{x}}_c \leq \boldsymbol{x}_c \leq \overline{\boldsymbol{x}}_c \tag{3e}$$

$$\underline{\boldsymbol{x}}_d \leq \boldsymbol{x}_d \leq \overline{\boldsymbol{x}}_d \tag{3f}$$

[1] The only GO3 constraint which is *not* explicitly captured via (3) is the synchronous network connectivity constraint. This constraint specifies that a given line switch cannot induce an electrical island within the AC network. The `QuasiGrad` solver deals with this constraint heuristically.

$$\boldsymbol{x}_d \in \mathbb{Z}^{n_d} \tag{3g}$$

$$\boldsymbol{x}_c \in \mathbb{R}^{n_c}, \tag{3h}$$

where $\boldsymbol{x}_c$ and $\boldsymbol{x}_d$ are vectors of continuous and discrete *basis* variables (we denote $\boldsymbol{x}$ as the more general concatenation of $\boldsymbol{x}_c$ and $\boldsymbol{x}_d$), while $\boldsymbol{y}$ is a vector of auxiliary variables. We use the term "basis" to denote the minimum set of variables needed to uniquely reconstruct a full GO3 solution (e.g., $\boldsymbol{v}$, $\boldsymbol{\theta}$, etc.); these are the same variables which are reported in the *solution.json* file sent to the GO3 solution parser (referred to as "Output data" in [10]). The basis variable sets Ω_c and Ω_d, associated with $\boldsymbol{x}_c$ and $\boldsymbol{x}_d$, are given as

$$\begin{aligned}\Omega_c = \{&v \cup \theta \cup \phi \cup \tau \cup p^{\text{fr,dc}} \cup q^{\text{fr,dc}} \cup q^{\text{to,dc}} \cup p^{\text{on}} \cup \\ &q \cup p^{\text{rgu}} \cup p^{\text{rgd}} \cup p^{\text{scr}} \cup p^{\text{nsc}} \cup p^{\text{rru,on}} \cup \\ &p^{\text{rru,off}} \cup p^{\text{rrd,on}} \cup p^{\text{rrd,off}} \cup q^{\text{rqu}} \cup q^{\text{rqd}}\}\end{aligned} \tag{4}$$

$$\Omega_d = \{u^{\text{sh}} \cup u^{\text{on}}\}. \tag{5}$$

See [10] for definitions. The auxiliaries in $\boldsymbol{y}$, on the other hand, represent the large set of variables which can be directly eliminated from the problem formulation. In the reformulation (3), we have carefully eliminated all such auxiliary variables. Our elimination uses the $\max(\cdot, 0)$, or ReLU, operator extensively; the ReLU is one of the foundational nonlinear activation functions used in modern ML, and gradient-based solvers have demonstrated a remarkable ability to optimize over functions which use it (to the apparent surprise of mathematicians [11]).

Next, we present a series of four representative auxiliary variable reformulation examples; i.e., where we reformulate optimization constraints into to explicit functions à la (3b).

- *Example 1 (slack variable elimination):* The apparent power flow on a given line s_{jt} is penalized if its magnitude is larger than flow limit $s_j^{\max}$. In [10], this penalty z_{jt}^{s} is formulated using slack variable s_{jt}^{+} ("fr" and "to" sides are neglected for notational clarity):

$$0 \leq s_{jt}^{+} \tag{6a}$$

$$z_{jt}^{\text{s}} = d_t c^{\text{s}} s_{jt}^{+} \tag{6b}$$

$$\sqrt{p_{jt}^2 + q_{jt}^2} \leq s_j^{\max} + s_{jt}^{+}. \tag{6c}$$

Since line flows may be written as direct nonlinear functions of the basis variables, and since the slack value may be captured using a $\max$ operator, the penalty may be computed as an explicit function of basis variables $\boldsymbol{x}$:

$$z_{jt}^{\text{s}} = d_t c^{\text{s}} \max((p_{jt}^2(\boldsymbol{x}) + q_{jt}^2(\boldsymbol{x}))^{1/2} - s_j^{\max}, 0). \tag{7}$$

All slack variables are transformed in this way. □

- *Example 2 (startup and shutdown variable elimination):* Startup, shutdown, and on-off variables are linked via evolution equations [10]:

$$u_{jt}^{\text{su}} + u_{jt}^{\text{sd}} \leq 1 \tag{8a}$$

$$u_{jt}^{\text{on}} - u_{j,t-1}^{\text{on}} = u_{jt}^{\text{su}} - u_{jt}^{\text{sd}}. \tag{8b}$$

The startup and shutdown variables, however, are uniquely defined for a given on-off variable u_{jt}^{on} sequence. Therefore, (8) may be captured via the following explicit definitions for auxiliary startup and shutdown variables:

$$u_{jt}^{\text{su}} \triangleq +\max\left(u_{jt}^{\text{on}} - u_{j,t-1}^{\text{on}}, 0\right) \tag{9}$$

$$u_{jt}^{\text{sd}} \triangleq -\min\left(u_{jt}^{\text{on}} - u_{j,t-1}^{\text{on}}, 0\right). \tag{10}$$

These auxiliary variables are then plugged in for a variety of uses (e.g., startup state calculations, shutdown costs, etc). □

- *Example 3 (device cost curves):* Time-dependent device costs z_{jt}^{en} are modeled in [10] via piecewise linear convex (or concave) cost (or value) functions:

$$0 \leq p_{jtm} \leq p_{jtm}^{\max},\ \forall t \in T, j \in J^{\text{pr,cs}}, m \in M_{jt} \tag{11a}$$

$$p_{jt} = \sum_{m \in M_{jt}} p_{jtm},\ \forall t \in T, j \in J^{\text{pr,cs}} \tag{11b}$$

$$z_{jt}^{\text{en}} = d_t \sum_{m \in M_{jt}} c_{jtm}^{\text{en}} p_{jtm},\ \forall t \in T, j \in J^{\text{pr,cs}}. \tag{11c}$$

By instead defining a cumulative block size $p_{jtm_L}^{\text{cum,max}}$ as

$$p_{jtm_L}^{\text{cum,max}} = \sum_{l=1}^{L} p_{jtm_l}^{\max}, \tag{12}$$

energy cost may be explicitly computed as

$$z_{jt}^{\text{en}} = d_t \sum_{l=1}^{|M_{jt}|} c_{jtm_l}^{\text{en}} \max\Big(\min\left(p_{jt} - p_{jtm_{L=l-1}}^{\text{cum,max}}, p_{jtm_l}\right), 0\Big), \tag{13}$$

where $p_{jtm_{L=0}}^{\text{cum,max}} = 0$. The $\max(\min(\cdot))$ formulation sums the length of a bid block, times its marginal cost, until a bid block component exceeds the power production value p_{jt} (at which point, 0 is added thereafter). □

- *Example 4 (power balance):* Active and reactive power imbalance expressions are computed in [10] as

$$p_{it} = \sum_{j \in J_i^{\text{cs}}} p_{jt} + \sum_{j \in J_i^{\text{sh}}} p_{jt} + \sum_{j \in J_i^{\text{fr}}} p_{jt}^{\text{fr}} + \sum_{j \in J_i^{\text{to}}} p_{jt}^{\text{to}} - \sum_{j \in J_i^{\text{pr}}} p_{jt} \tag{14}$$

$$q_{it} = \sum_{j \in J_i^{\text{cs}}} q_{jt} + \sum_{j \in J_i^{\text{sh}}} q_{jt} + \sum_{j \in J_i^{\text{fr}}} q_{jt}^{\text{fr}} + \sum_{j \in J_i^{\text{to}}} q_{jt}^{\text{to}} - \sum_{j \in J_i^{\text{pr}}} q_{jt}. \tag{15}$$

Mismatch penalties, $\forall t \in T, i \in I$, are then computed via $z_{it}^{\text{p}} = d_t c^{\text{p}} p_{it}^{+}$ and $z_{it}^{\text{q}} = d_t c^{\text{q}} q_{it}^{+}$, where slack inequalities $p_{it}^{+} \geq p_{it}$, $p_{it}^{+} \geq -p_{it}$, $q_{it}^{+} \geq q_{it}$, and $q_{it}^{+} \geq q_{it}$ are additionally enforced. To transform this expression, we take absolute value:

$$z_{it}^{\text{p}} = d_t c^{\text{p}} |p_{it}| \tag{16}$$

$$z_{it}^{\text{q}} = d_t c^{\text{q}} |q_{it}|, \tag{17}$$

where p_{it} and q_{it} come directly from the mismatch equations defined above. Since c^{p} and c^{q} are very large penalization constants, a tightening "soft-abs" function will be used in the final `QuasiGrad` formulation, rather than an abs function. □

Through these transformations, along with a series of other similar ones (see SI material for this paper: [12]), we are able to write a market surplus function which is an explicit function of the 21 basis variables identified in (4)-(5). This process was implemented manually, and to the author's knowledge, no generalized code base or software libraries exist which can automatically perform said transformation. Recent tools, e.g., in [13], have used ReLU operators to penalize constraint violations, but the direct application of ReLU-based penalizations will not, e.g., eliminate unnecessary slack variables, or efficiently capture block-bid transformations as in (13).

C. Further refinements: constraint penalization, variable clipping, and integer relaxation

With the isolation of auxiliary variables, the MINLP of (3) can have its (i) auxiliary variables eliminated, (ii) linear constraints penalized with a soft-ReLU[2], (iii) integers relaxed, and (iv) basis variables "clipped" into a rectangular bounding box $\mathcal{B}$ defined from constraints (3g)-(3h). The updated NLP formulation is given as:

$$\begin{aligned} \min_{\boldsymbol{x}_d,\boldsymbol{x}_c\in\mathcal{B}} \quad & z^{\mathrm{ms}}(\boldsymbol{x}_c,\boldsymbol{x}_d,\boldsymbol{f}(\boldsymbol{x}_c,\boldsymbol{x}_d))+z^{\mathrm{ctg}} \\ & \qquad +\rho\cdot\sigma_s\left(A_c\boldsymbol{x}_c+A_d\boldsymbol{x}_d\right) && \text{(18a)} \\ \text{s.t.} \quad & \boldsymbol{0}=\boldsymbol{h}_{\mathrm{ctg}}(\boldsymbol{x}_c,\boldsymbol{x}_d,\boldsymbol{\theta}_k). && \text{(18b)} \end{aligned}$$

Integer variables: The general strategy of the `QuasiGrad` solver for dealing with integers goes as follows: (i) solve NLP (18) to some degree of accuracy, (ii) project the relaxed integers into the feasible space (see next subsection), (iii) permanently fix a subset of the integers whose projected values were closest to their relaxed values, adding them to set $\mathcal{F}$, and (iv) repeat until all integers are fixed to feasible values. This sort of *batch rounding* procedure is a highly useful heuristic, but it can easily be replaced with a more systematic integer search (as with α,β-CROWN, which performs a complete BaB search routine with Adam as the subproblem solver). This procedure is generally inspired by the Iterative Batch Rounding (IBR) routine used by the GravityX team in GO1 and GO2 [2]. Our general strategy is outlined in Alg. 1, where n_b is the total number of binaries. As the Adam solver iterates and the wall clock time increases, the soft-ReLU function in (18a) increasingly penalizes constraint violations more strongly, similar to the soft-abs tightening in Fig. 4.

Algorithm 1 `QuasiGrad` Process for Fixing Integers

```
$\mathcal{F} \leftarrow \emptyset$ (no fixed binaries)
while $|\mathcal{F}| < n_b$ (some binaries are not fixed) do
    Solve continuous NLP (18) with binaries in $\mathcal{F}$ fixed
    Project device binaries via Proj. 1 using parallelized solves
    Add a fraction of the projected binaries to $\mathcal{F}$
end
```

D. Integer projection

Adam does not enforce integrality constraints itself. Rather, in our approach, we rely on the successive, highly-parallelized projection of all integer variables. The associated projection is given in Proj. 1. This MILP projection enforces all ramp, reserve, headroom, and limit constraints for a given device – producer (generator) or consumer (load). Notably, these projections are executed in parallel via multi-threading (i.e., on each CPU thread, Gurobi is explicitly given a MILP to solve, each associated with a single device). The objective function tries to keep the decision variables $\boldsymbol{x}_c$, $\boldsymbol{x}_d$ as close to the continuous NLP solution $\boldsymbol{x}_c^0$, $\boldsymbol{x}_d^0$ as possible via penalizing deviations. The matrices $D_c^{\mathrm{g}_i}$, $D_d^{\mathrm{g}_i}$ are diagonal matrices which simply select the decision variables associated with device i. Finally, integer variables in $\mathcal{F}$ are fixed to their previously projected values and eliminated from a given projection. The parallel nature of device projections is illustrated by the parallel vertical red lines in Fig. 2. Notably, a single device projection over, e.g., 18 time periods typically solves to ~ 0 optimality gap via Branch-and-Bound search routine in less than 50 ms (often, even faster).

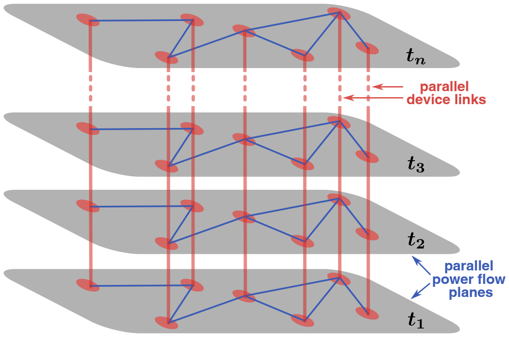

Fig. 2: The parallel nature of devices constraints and power flow constraints are portrayed. Devices can be projected feasible in parallel (via Proj. 1), and power flow solves can be performed in parallel (via Proj. 3).

Projection 1: Optimal Device Binary Projection [MILP]
⋆ *parallelizable across each device*

$$\begin{array}{rll} \min\limits_{\boldsymbol{x}_c\in\mathbb{R},\boldsymbol{x}_d\in\mathbb{Z}} & \left\|D_c^{\mathrm{g}_i}\left(\boldsymbol{x}_c-\boldsymbol{x}_c^0\right)\right\|_1+\left\|D_d^{\mathrm{g}_i}\left(\boldsymbol{x}_d-\boldsymbol{x}_d^0\right)\right\|_1 & \\ \text{s.t.} & \boldsymbol{x}_{d,i}=\boldsymbol{x}_{d,i}^0,\ i\in\mathcal{F} & \textbf{\textit{(fixed binaries)}} \\ & \text{[10, eqs. (48)-(58)]} & \text{(binary constraints)} \\ & \text{[10, eqs. (68)-(74)]} & \text{(ramp limits)} \\ & \text{[10, eq. (98)-(108)]} & \text{(reserve constraints)} \\ & \text{[10, eq. (109)-(118)]} & \text{(producer limits)} \\ & \text{[10, eq. (119)-(128)]} & \text{(consumer limits)} \end{array}$$

E. Contingency gradients

In subsection II-B, we solved for and eliminated all auxiliary variables, but we left the contingency expression (18b). Contingencies in GO3 are modeled via DC power flow solutions (in conjunction with nonlinear apparent power flow calculations). In order to eliminate a DC power flow expression $\boldsymbol{p}_t^{\mathrm{inj}} = Y_k\boldsymbol{\theta}_{tk}$, we would need to (i) solve it directly via $\boldsymbol{\theta}_{tk} = Y_k^{-1}\boldsymbol{p}_t^{\mathrm{inj}}$, (ii) use the phase angle solution to compute flow violations, and then (iii) push those violations up into the objective function (18a). This approach is non-scalable, however, for two reasons:

- Y_k^{-1} is generally a dense matrix, and it would consume very large amounts of memory to compute and store such matrices for each contingency/time;

[2] The soft-abs function applied to scalar x is defined via $|x|_{\mathrm{s}} \triangleq \sqrt{x^2+\epsilon^2}$. The soft-ReLU function applied to scalar x is $\sigma_{\mathrm{s}}(x) \triangleq \sqrt{\max(x,0)^2+\epsilon^2}$.

- every Adam iteration would require $n_t \times n_{\mathrm{ctg}}$ dense matrix-vector products, which would be untenable.

Instead, the approach we take can be summarized in three steps: at each Adam iteration, we (i) solve a subset of contingencies, (ii) backpropagate through the ones with the worst violations, and then (iii) pass these computed gradients to Adam. Thus, Adam does not *solve* contingencies, but it does feel the pressure from their efficiently computed *gradients*. In the following, we propose efficient methods for solving a subset of contingencies at each Adam iteration and then backpropagating through them.

1) Contingency evaluation: As in [10], GO3 contingency k line flows are computed via

$$\begin{aligned} p_{jtk} &= -b^{\mathrm{sr}}_j u^{\mathrm{on}}_{jt}\left(\theta_{itk} - \theta_{i'tk} - \phi_{jt}\right) && \text{(19a)} \\ &= \underbrace{-b^{\mathrm{sr}}_j u^{\mathrm{on}}_{jt}\left(\theta_{itk} - \theta_{i'tk}\right)}_{f_{jtk}} + \underbrace{b^{\mathrm{sr}}_j u^{\mathrm{on}}_{jt}\phi_{jt}}_{b_{jt}}. && \text{(19b)} \end{aligned}$$

We vectorize f_{jtk}, b_{jt} across all lines and transformers into $\boldsymbol{f}_{tk}$, $\boldsymbol{b}_t$. We then use a signed incidence matrix E to compute nodal injections (which are know by device injections):

$$\begin{aligned} \boldsymbol{p}^{\mathrm{inj}}_t &= E^T(\boldsymbol{f}_{tk} + \boldsymbol{b}_t) && \text{(20a)} \\ &= E^T Y_x E \boldsymbol{\theta}_{tk} + E^T \boldsymbol{b}_t && \text{(20b)} \\ &= Y_b \boldsymbol{\theta}_{tk} + E^T \boldsymbol{b}_t, && \text{(20c)} \end{aligned}$$

where Y_x is a diagonal matrix of inverse line reactances. Deleting the reference bus (hat notation), reduced nodal angles $\hat{\boldsymbol{\theta}}_{tk}$ and contingency branch flows $\boldsymbol{p}_{tk}$ may be computed:

$$\hat{\boldsymbol{\theta}}_{tk} = \hat{Y}_b^{-1}\left(\hat{\boldsymbol{p}}^{\mathrm{inj}}_t - \hat{E}^T \boldsymbol{b}_t\right) \qquad (21)$$

$$\boldsymbol{p}_{tk} = Y_x \hat{E} \hat{\boldsymbol{\theta}}_{tk}. \qquad (22)$$

Once computed, $\boldsymbol{p}_{tk}$ is used in conjunction with reactive power line flows to compute apparent power branch overloads. The main computational task in this process is solving the linear system in (21). Here, we exploit a preconditioned conjugate gradient (`pcg`) solver [14]. This solver is a Krylov subspace method which *iteratively* (rather than recursively, in the case of Gaussian elimination) approximates a linear system solution. Since the base-case DC admittance matrix $\hat{Y}_b$ is static for a given set of line switches, we use a constant Limited memory LDL (`LLDL`) factorization as a preconditioner $\hat{P}$. This greatly accelerates `pcg` convergence:

$$\hat{P} \leftarrow \mathtt{LLDL}\left(\hat{E}^T Y_x \hat{E}\right) \qquad (23)$$

$$\hat{\boldsymbol{\theta}}_{tb} \approx \mathtt{pcg}(\hat{\boldsymbol{p}}^{\mathrm{inj}}_t - \hat{E}^T \boldsymbol{b}_t, \hat{Y}_b, \hat{P}, \epsilon_{\mathrm{pcg}}), \qquad (24)$$

where the `pcg` function approximates the solution of (21) with preconditioner $\hat{P}$; it terminates when the provided error metric ϵ_{pcg} is satisfied. Notably, we only solve (24) for the base-case at time t (i.e., no contingency branches removed from the network yet). Using a low-rank update procedure recently pioneered in [15], we then *rank-1 correct* to solve for each contingency solution. This is motivated by the fact that the admittance matrix of a given contingency is only "rank-1 away" from the base-case admittance matrix. To show this, let Y_k be an almost-empty matrix with a single, nonzero entry; this entry is placed on the diagonal element associated with the single line that is removed in a contingency, and its value is the negative admittance of that line. Then, the relationship between the base-case admittance $\hat{Y}_b$ and a given contingency admittance matrix $\hat{Y}_{b,k}$ is

$$\begin{aligned} \text{base-case admittance:} \quad & \hat{Y}_b = \hat{E}^T Y_x \hat{E} \\ \text{contingency admittance:} \quad & \hat{Y}_{b,k} = \hat{E}^T Y_x \hat{E} + \underbrace{\hat{E}^T Y_k \hat{E}}_{\boldsymbol{v}_k \boldsymbol{v}_k^T \text{ (rank-1)}}. \end{aligned}$$

Thus, using the Sherman-Morrison-Woodbury (SMW) formula [15], [16], we may rank-1 correct a base-case nodal phase angle solution. Setting $\boldsymbol{c}_t \triangleq \hat{\boldsymbol{p}}^{\mathrm{inj}}_t - \hat{E}^T \boldsymbol{b}_t$, we have

$$\begin{aligned} (\hat{Y}_b + \boldsymbol{v}_k \boldsymbol{v}_k^T)\hat{\boldsymbol{\theta}}_{tk} &= \boldsymbol{c}_t && \text{(25a)} \\ \hat{\boldsymbol{\theta}}_{tk} &= \left(\hat{Y}_b^{-1} - \frac{\hat{Y}_b^{-1}\boldsymbol{v}_k \boldsymbol{v}_k^T \hat{Y}_b^{-1}}{1 + \boldsymbol{v}_k^T \hat{Y}_b^{-1} \boldsymbol{v}_k}\right)\boldsymbol{c}_t && \text{(25b)} \\ &= \hat{\boldsymbol{\theta}}_{tb} - \underbrace{\hat{Y}_b^{-1}\boldsymbol{v}_k}_{\boldsymbol{u}_k} \underbrace{\frac{\boldsymbol{v}_k^T \hat{Y}_b^{-1}}{1 + \boldsymbol{v}_k^T \hat{Y}_b^{-1} \boldsymbol{v}_k}}_{\boldsymbol{w}_k} \boldsymbol{c}_t && \text{(25c)} \\ &= \hat{\boldsymbol{\theta}}_{tb} - \boldsymbol{u}_k(\boldsymbol{w}_k^T \boldsymbol{c}_t), && \text{(25d)} \end{aligned}$$

since $\hat{\boldsymbol{\theta}}_{tb} \approx \hat{Y}_b^{-1}\boldsymbol{c}_t$ via (24). Thus, the DC power flow solution to a given contingency can be computed as the rank-1 correction to a single `pcg` solve. This rank-1 correction is quickly computed with one vector-vector inner product, once vector-scalar product, and one vector-vector subtraction. After calculating $\hat{\boldsymbol{\theta}}_{tk}$, contingency active power flows are computed via (22), and then directional line penalties are computed via

$$\boldsymbol{s}^{\mathrm{fr},+}_{tk} = \max\{(\boldsymbol{p}^2_{tk} + (\boldsymbol{q}^{\mathrm{fr}}_{tk})^2)^{\frac{1}{2}} - \boldsymbol{s}^{\max,\mathrm{ctg}}, 0\} \qquad (26)$$

$$\boldsymbol{s}^{\mathrm{to},+}_{tk} = \max\{(\boldsymbol{p}^2_{tk} + (\boldsymbol{q}^{\mathrm{to}}_{tk})^2)^{\frac{1}{2}} - \boldsymbol{s}^{\max,\mathrm{ctg}}, 0\} \qquad (27)$$

$$z^{\mathrm{ctg}}_{tk} = \boldsymbol{1}^T\left(d_t c^{\mathrm{s}} \max\left(\boldsymbol{s}^{\mathrm{fr},+}_{tk}, \boldsymbol{s}^{\mathrm{to},+}_{tk}, 0\right)\right). \qquad (28)$$

2) Contingency backpropagation: Impactful contingencies with nonzero penalties are backpropagated through, i.e., we take the gradient of aggregated penalty scalar z^{ctg}_{tk} with respect to all relevant basis variables. Taking the gradients with respect to reactive power flows, which are direct functions of nodal voltage variables, is fairly straightforward and explained in the SI [12]. Active power injection gradients are less trivial and involve three steps: differentiate z^{ctg}_{tk} with respect to active power flows $\boldsymbol{p}_{tk}$, differentiate $\boldsymbol{p}_{tk}$ with respect to injections, and then differentiate injections with respect to basis variables.

We begin this process with an example: assume some scalar $z = f(\boldsymbol{y})$ has gradient $\nabla_{\boldsymbol{y}} z = \boldsymbol{d}$. If $\boldsymbol{y} = A\boldsymbol{x}$, then

$$\left.\begin{aligned} \nabla_{\boldsymbol{y}} z = \boldsymbol{d} \\ \nabla_{\boldsymbol{x}}[y_1, y_2, ..., y_n] = A^T \end{aligned}\right\} \Rightarrow \nabla_{\boldsymbol{x}} z = A^T \boldsymbol{d} \qquad (29)$$

is a well known result. We exploit this in the following way: we write the contingency penalty as a function of branch flows: $z^{\mathrm{ctg}}_{tk} = f(\boldsymbol{p}_{tk})$ whose gradient is $\nabla_{\boldsymbol{p}_{tk}} z^{\mathrm{ctg}}_{tk} = \boldsymbol{d}_k$ (this gradient can be written by inspection of (26)-(28)). Next, we write the linear mapping between flows, injections, and phase shifters:

$$\boldsymbol{p}_{tk} = Y_{x,k}\hat{E}\hat{Y}_k^{-1}\hat{\boldsymbol{p}}_t^{\text{inj}} - Y_{x,k}\hat{E}\hat{Y}_k^{-1}\hat{E}^T\boldsymbol{b}_t. \tag{30}$$

Fully analogous to (29), the gradient mappings between contingency penalties, nodal injections, and phase shifters are

$$\begin{aligned}\nabla_{\hat{\boldsymbol{p}}_t^{\text{inj}}} z_{tk}^{\text{ctg}} &= \left(Y_{x,k}\hat{E}\hat{Y}_k^{-1}\right)^T \boldsymbol{d}_k &\text{(31a)}\\ &= \hat{Y}_k^{-1}\hat{E}^T Y_{x,k}\boldsymbol{d}_k &\text{(31b)}\\ \nabla_{\boldsymbol{b}_t} z_{tk}^{\text{ctg}} &= -\left(Y_{x,k}\hat{E}\hat{Y}_k^{-1}\hat{E}^T\right)^T \boldsymbol{d}_k &\text{(31c)}\\ &= -\hat{E}\left(\nabla_{\hat{\boldsymbol{p}}_t^{\text{inj}}} z_{tk}^{\text{ctg}}\right). &\text{(31d)}\end{aligned}$$

Thus, in (31b), we are required to solve *yet another* linear system. This is to be expected, since the contingency *evaluation* (24) linear system solve incorporated no nonlinearity (it is simply a DC power flow solve), while the second linear system solve backpropagates through (i.e., takes sensitivity to) a number of nonlinear functions in (26)-(28)). To solve (31b), we again use low-rank corrections to a `pcg` base-case solve.

There is one final, non-obvious step in the backpropagation. To take this step, we note that $\nabla_{\hat{\boldsymbol{p}}_t^{\text{inj}}}\boldsymbol{p}_{tk} = (Y_{x,k}E\hat{Y}_k^{-1})^T$ is in fact an approximation – not because of nonlinearity, but because of GO3 slack distribution rules. When taking a gradient, a power perturbation, say, on bus 1 must be *uniformly* redistributed at all other buses according to [10, eq. (162)-(163)]. Thus, a power perturbation Δp_1 at bus 1 shows up like the following smaller perturbations at all other buses:

$$\Delta p_1 \rightarrow \begin{bmatrix} p_1 - \frac{\Delta p_1}{n} + \Delta p_1 \\ p_2 - \frac{\Delta p_1}{n} \\ \vdots \\ p_n - \frac{\Delta p_1}{n}\end{bmatrix}. \tag{32}$$

Thus, in the backpropagation, we need to correct for this effect. To do so, denote $A = Y_{x,k}\hat{E}\hat{Y}_k^{-1}$, where $\boldsymbol{p}_{tk} = A\hat{\boldsymbol{p}}_t^{\text{inj}}$. By (32), each perturbation in power Δp "spreads out" across all powers, giving us the following scalar relation:

$$\begin{aligned}\Delta p_{tk,i} &= A_{i,j}\Delta\hat{p}_{t,j}^{\text{inj}} - \frac{1}{n}\sum_k A_{i,k}\Delta\hat{p}_{t,j}^{\text{inj}},\ \forall i,j &\text{(33a)}\\ &= \left(A_{i,j} - \frac{1}{n}A_i\boldsymbol{1}\right)\Delta\hat{p}_{t,j}^{\text{inj}},\ \forall i,j. &\text{(33b)}\end{aligned}$$

Since these expressions hold $\forall i,j$, the matrix A' which relates perturbations in injections and flows is given generally as

$$A' = A - \frac{A\boldsymbol{1}\boldsymbol{1}^T}{n} = A\left(I - \frac{\boldsymbol{1}\boldsymbol{1}^T}{n}\right). \tag{34}$$

Directly updating the gradient in (31b), we have

$$\begin{aligned}\nabla_{\hat{\boldsymbol{p}}_t^{\text{inj}}} z_{tk}^{\text{ctg}} &= \left[Y_{x,k}\hat{E}\hat{Y}_k^{-1}\left(I - \frac{\boldsymbol{1}\boldsymbol{1}^T}{n}\right)\right]^T \boldsymbol{d}_k &\text{(35a)}\\ &= \left(I - \frac{\boldsymbol{1}\boldsymbol{1}^T}{n}\right)^T \underbrace{\hat{Y}_k^{-1}\hat{E}^T Y_{x,k}\boldsymbol{d}_k}_{\boldsymbol{\eta}} &\text{(35b)}\\ &= \boldsymbol{\eta} - \tfrac{1}{n}\sum \boldsymbol{\eta}. &\text{(35c)}\end{aligned}$$

Thus, once we compute (31b), which yields $\boldsymbol{\eta}$, we correct its value via the surprisingly elegant (35c). Copious numerical test results confirmed the validity of this unexpected expression. Similar rank-1 flow corrections (e.g., more explicitly, to the PTDF matrix) have been observed in [17, eq. 9]).

3) Contingency backpropagation summary: As Adam iterates, we maintain a running list of the most severe contingencies. At each Adam iteration, we evaluate contingencies in the top $X\%$ percentage of this list, along with a stochastic selection from the bottom percentage. All contingencies that have a z_{tk}^{ctg} score (28) higher than a certain numerical threshold get backpropagated through; their gradients then get included in (36) and sent to Adam. Alg. 2 summarizes this procedure.

Algorithm 2 Contingency solver

```
Require: Set of worst contingencies $\mathcal{K}_t$, set of stochastically selected contingencies $\mathcal{S}_t$, pcg tolerance $\epsilon_{\text{pcg}}$, backprop threshold $\zeta$
for $t \in T$ do                    ▷ parallel loop over ACUC time periods
    pcg solve all base-case DC power flows via (24)
end
for $t \in T$ do
    for $k \in \mathcal{K}_t \cup \mathcal{S}_t$ do        ▷ parallel loop over contingencies
        Rank-1 correct base-case solutions via SMW: (25d)
        Score ctg via (28)
        if $z_{tk}^{\text{ctg}} > \zeta$ threshold then
            Solve backpropagation (31b)-(31d) via pcg + SMW
        end
    end
end
Update contingency sets $\mathcal{K}_t$ and $\mathcal{S}_t$
return Approximated contingency gradients $\nabla_{\boldsymbol{x}}\boldsymbol{h}_{\text{ctg}}$
```

F. Implementation of the Adam solver

We may now sum the contingency gradients $\nabla_{\boldsymbol{x}}\boldsymbol{h}_{\text{ctg}}$ with the NLP objective function (18a) gradients:

$$\boxed{\boldsymbol{g} = \nabla_{\boldsymbol{x}}\left(z^{\text{ms}}(\boldsymbol{x}) + \rho\cdot\sigma_s\left(A\boldsymbol{x}\right)\right) + \nabla_{\boldsymbol{x}}\boldsymbol{h}_{\text{ctg}}, \quad (36)}$$

where the shorthand $\boldsymbol{x}$ has been used to represent all basis variables. Eq. (36) is the result of a *backpropagation*. Notably, all gradient in (36) are *manually* computed in the `QuasiGrad` solver source code, which required a fairly substantial effort. Backpropagation generates a cascade of derivatives which, via chain rule, connect the sensitivity of a loss function (or market surplus function) to a basis variable. For instance, backpropagation from basis variables which influence line flows "x_{lf}" to the market surplus function is given by

$$\begin{aligned}\nabla_x z^{\text{ms}} =& \nabla_{z^{\text{base}}} z^{\text{ms}} \cdot \nabla_{z_t^{\text{t}}} z^{\text{base}} \cdot \nabla_{z_{jt}^{\text{s}}} z_t^{\text{t}} \cdot \nabla_{s_{jt}^{+}} z_{jt}^{\text{s}}\\ &\cdot \nabla_{s_{jt}^{\text{fr/to},+}} s_{jt}^{+} \cdot \nabla_{p/q_{jt}^{\text{fr/to},+}} s_{jt}^{\text{fr/to},+} \cdot \nabla_{x_{\text{lf}}} p/q_{jt}^{\text{fr/to},+},\\ & x_{\text{lf}} \in \{v_{it}, v_{i't}, \theta_{it}, \theta_{i't}, \tau_{jt}, \phi_{jt}, u_{jt}^{\text{on}}\}.\end{aligned}$$

Further details on gradient reformulation and backpropagation are provided in the SI [12]. Importantly, computation of this gradient can exploit multi-threaded parallelism, which is the general key to Adam's success. Depending on the type of gradient needed, the `QuasiGrad` solver parallelizes over ACUC time instance, network devices, or contingencies. One of the main tools for achieving such parallelism is Julia's `Threads.@threads` macro, which assigns the computational workload associated with a loop onto different available CPU threads. As a specific example, consider the gradient of a "fr" line active power flow with respect to the "to" side voltage (where $\boldsymbol{\delta} = \boldsymbol{\theta}_t^{\rm fr} - \boldsymbol{\theta}_t^{\rm to} - \boldsymbol{\phi}_t$):

```
for $t \in T$
    $\nabla_{\boldsymbol{v}_t^{\rm to}} \boldsymbol{p}_t^{\rm fr} = \left(-\boldsymbol{g}^{\rm sr} \cos\left(\boldsymbol{\delta}\right) - \boldsymbol{b}^{\rm sr} \sin\left(\boldsymbol{\delta}\right)\right) \boldsymbol{v}_t^{\rm fr} / \boldsymbol{\tau}_t$
end
```

Generally, a CPU will execute these gradient solves in series (one for each ACUC time instance). However, we may instruct the compiler to compute these gradients in parallel via

```
Threads.@threads for $t \in T$
    $\nabla_{\boldsymbol{v}_t^{\rm to}} \boldsymbol{p}_t^{\rm fr} = \left(-\boldsymbol{g}^{\rm sr} \cos\left(\boldsymbol{\delta}\right) - \boldsymbol{b}^{\rm sr} \sin\left(\boldsymbol{\delta}\right)\right) \boldsymbol{v}_t^{\rm fr} / \boldsymbol{\tau}_t$
end
```

Each doubling of CPU threads generally halves computational time. We note that there is no single "right way" to multi-thread: an infinite variety of effective options exist.

After efficiently computing the gradients in (36), we pass these gradients to a modified Adam solver [18]. Adam has been written about ad nauseam in the ML literature, so we provide little discussion of the Adam solver itself. At a high level, however, Adam uses first and second order moment estimates to implicitly track the curvature of a loss function landscape. Adam step sizes adaptively react to observed changes in the curvature, thus accelerating convergence towards some local minimum. Since Adam only needs gradient information to make decisions about step size and direction, variables can be updated in parallel. Furthermore, gradient calculations can computed concurrently, making the approach highly amenable to parallel computation.

We feed the gradients from (36) into the modified Adam solver of Alg. 3, which loops over basis variables and ACUC time instances (in parallel). After updating the Adam and basis variables states, basis variables are "clipped" (i.e., projected) back into their feasible domain, as stated in Line 6. For example, a binary state u is clipped into the range between 0 and 1 via $u \leftarrow \min(\max(u, 0), 1)$. A voltage state v may be similarly clipped via $v \leftarrow \min(\max(v, \underline{v}), \overline{v})$.

Adam step size: Practically speaking, one of the most important aspects of Alg. 3 is setting the gradient descent step size α. Since the GO3 clearing problems are time-limited (10, 120, and 240 minute limits for the real time, day-ahead, and week-ahead problems, respectively), we use wall-clock time to set the Adam step size: $\alpha_\omega(⏱ = t_w)$, which is a function of basis variable ω. Initially, Adam takes relatively large steps, but as time depletes, the steps sizes decay to very small values (in order to "clean up" the solution). We use a reflected sigmoid function to set step size magnitude:

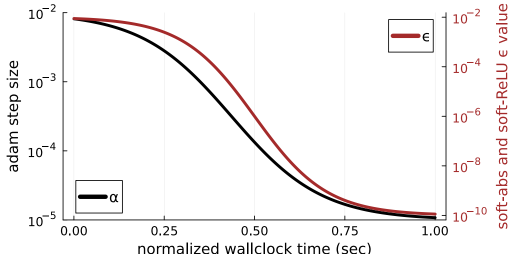

Fig. 3: Adam step size decay (α, left) and soft-abs/soft-ReLU tightening (ϵ, right). Step size decay "leads" soft-abs/soft-ReLU tightening.

$$\text{normalize time: } \hat{t}_w = 2\frac{t_w - t_0}{t_f - t_0} - 1 \tag{37}$$

$$\text{magnitude scale: } \beta = \frac{e^{4\hat{t}_w}}{0.6 + e^{4\hat{t}_w}} \tag{38}$$

$$\text{actual step size: } \alpha = \alpha_0 10^{\beta \cdot \log 10(\frac{\alpha_f}{\alpha_0})}. \tag{39}$$

A representative step size decay curve, which exactly plots (37)-(39), is given by the black curve in Fig. 3.

Algorithm 3 Modified Adam Solver (Original Adam: [18])

```
Require: Adam decay parameters $\beta_1, \beta_2$, adam iteration index $i$, basis variable gradients $\boldsymbol{g}_{t,\omega}$, step size function $\alpha_\omega(⏱)$
1: for $\omega \in \Omega$ do                      ▷ loop over basis variables
2:     for $t \in T$ do                            ▷ parallel loop over ACUC time periods
3:         $\boldsymbol{m}_{t,\omega} \leftarrow \beta_1 \cdot \boldsymbol{m}_{t,\omega} + (1-\beta_1) \cdot \boldsymbol{g}_{t,\omega}$
4:         $\boldsymbol{v}_{t,\omega} \leftarrow \beta_2 \cdot \boldsymbol{v}_{t,\omega} + (1-\beta_2) \cdot \boldsymbol{g}_{t,\omega}^2$
5:         $\boldsymbol{x}_{t,\omega} \leftarrow \boldsymbol{x}_{t,\omega} - \alpha_\omega(⏱) \cdot \left(\frac{\boldsymbol{m}_{t,\omega}}{1-\beta_1^i} + \epsilon\right) \Big/ \left(\sqrt{\frac{\boldsymbol{v}_{t,\omega}}{1-\beta_2^i}} + \epsilon\right)$
6:         $\boldsymbol{x}_{t,\omega} \leftarrow \min(\max(\boldsymbol{x}_{t,\omega}, \underline{\boldsymbol{x}}_{t,\omega}), \overline{\boldsymbol{x}}_{t,\omega})$   ▷ clip all states!
    end
end
```

Homotopic constraint penalization: Certain gradients in (36) dominate other gradients by up to several orders of magnitude, essentially drowning out their contributions. In order to overcome this challenge, we first loosen these constraint penalties, and then we use a homotopy procedure to monotonically increase the penalization of these constraints as wall clock time increases. For example, we use a scaled, soft-abs function to penalize power balance error: $|x|_s = \beta \cdot \sqrt{x^2 + \epsilon^2}$. The ϵ^2 term decays in the same fashion as Adam step size (37)-(39), but over more orders of magnitude – see the red curve in Fig. 3, while β increases linearly from 0.1 to 1.0. The effect of this homotopic penalization is demonstrated in Fig. 4. Note that Adam is a first order method, so the upward shifting of these curves away from the origin is immaterial: only the *gradient* of these curves is relevant. Homotopic constraint penalization is applied to power balance, branch flow, contingency, and "penalized constraint" (see (18a)) penalties. Advanced homotopy methods have been highly successful in previous GO competitions [19].

III. QuasiGrad

The previous section reformulated the GO3 MINLP in to form which Adam can interact with. While Adam is a powerful

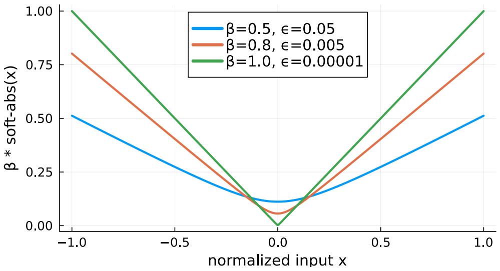

Fig. 4: Illustrated is the successive tightening of the soft-abs function $\beta\sqrt{x^2+\epsilon^2}$ as wall clock time increases. The value of epsilon is decreased in concordance with the red curve illustrated in Fig. 3.

optimization tool, copious testing has found that Adam's effectiveness can be greatly enhanced if it is combined with other well-established, hyper-efficient numerical techniques (e.g., parallelized LP solvers). In this section, we review each of these tools, and we then we put them all together into a single, coherent solver (`QuasiGrad`) with Adam at its center.

A. Copper plate economic dispatch with LP relaxed binaries

Since Adam is a local, gradient-based solver, it is highly influenced by its initialization. To initialize the `QuasiGrad` solver, we first pose and solve a copper plate economic dispatch problem, where all integers are LP relaxed, and all contingency penalties and network variables are neglected. This LP[3], which we classify as a projection, is given in Model 2. Notably, the solution provides an excellent initialization for Adam, but the market surplus value also acts as a global upper bound on the MINLP – this is useful for testing and benchmarking, as seen by the dashed orange line in Fig. 1.

Projection 2: Copper Plate Economic Dispatch [LP]
⋆ ***optionally*** *parallelizable across time instances*

$$\max_{\boldsymbol{x}_c,\boldsymbol{x}_d} \quad z^{\mathrm{ms}}$$

s.t. [10, eqs. (1)-(163)] (nominal GO3 formulation)

neglect:

- shunts, contingencies, integers (LP relax)
- all network variables $(\boldsymbol{v},\boldsymbol{\theta},\boldsymbol{\tau},\boldsymbol{\phi})$ and flow limits

impose:

- $\displaystyle\sum_{j\in J^{\mathrm{pr}}} p_{jt} = \sum_{j\in J^{\mathrm{cs}}} p_{jt} + \sum_{j\in J^{\mathrm{dc}}} p_{jt}^{\mathrm{fr/to}},\ \forall t$ (p balance)
- $\displaystyle\sum_{j\in J^{\mathrm{pr}}} q_{jt} = \sum_{j\in J^{\mathrm{cs}}} q_{jt} + \sum_{j\in J^{\mathrm{dc}}} q_{jt}^{\mathrm{fr/to}},\ \forall t$ (q balance)

B. Successively Linearized Power Flow Approximations

Finding an AC power flow solution after solving the copper plate economic dispatch can be very challenging – this is due to the fact that the economic dispatch often dispatches far more power than the network can physically accommodate (i.e., exceeding the maximum P-δ power transfer point of some lines). Adam can solve the resulting power flow, but schlepping large amounts of power across a network with gradient descent can be slower than "hot starting" Adam with several linearized parallel power flow solves. We pose this linearized power flow problem in Proj. 3, where

- $\boldsymbol{J}_{pv}^{\star}$, $\boldsymbol{J}_{p\theta}^{\star}$, $\boldsymbol{J}_{qv}^{\star}$, $\boldsymbol{J}_{q\theta}^{\star}$ are power balance sub-Jacobians;
- $\boldsymbol{J}_{sv}^{\star}$, $\boldsymbol{J}_{s\theta}^{\star}$ are apparent power flow sub-Jacobians;
- $\alpha \geq 1$ is an iteratively tightening flow constraint term;
- the objective function keeps all device injections as close to their initializations as possible via ℓ_2 norm penalty.

This projection also penalizes voltage perturbations (in order to regularize for convergence towards a consistent solution), and it regularizes for costs via γ_5 (i.e., pushing the solver towards cheaper power flow solutions). Notably, we use a quadratic objective function – in testing, this was found to be much faster than a linear, ℓ_1 norm penalizing function. The convex QPs of Proj. 3 are solved with Gurobi in parallel, as motivated by the parallel power flow planes in Fig. 2. These linearized projections are solved, and then iteratively re-solved, at the newly found solutions, which are used as new linearization points. These successively linearized projections are *not* run until convergence (i.e., $\boldsymbol{\Delta v} \approx \boldsymbol{\Delta\theta} \approx \boldsymbol{0}$ is neither achieved nor desired), and a single QP can solve in less than 1 second for a network with several thousand buses.

Projection 3: Regularized Power Balance Projection [QP]
⋆ *parallelizable across each time instance*

$$\min_{\boldsymbol{x}_c} \quad \gamma_1 \left\|\boldsymbol{p}_g - \boldsymbol{p}_g^0\right\|_2 + \gamma_2 \left\|\boldsymbol{q}_g - \boldsymbol{q}_g^0\right\|_2 + \gamma_3 \boldsymbol{\Delta v}^T \boldsymbol{\Delta v}$$

$$+ \gamma_4 \boldsymbol{\Delta\theta}^T \boldsymbol{\Delta\theta} + \gamma_5 \frac{c^T \boldsymbol{p}_g}{c^T \boldsymbol{p}_g^0} + \gamma_6 \cdot \{\text{other regularizers}\}$$

$$\text{s.t.} \quad p = p_0 + \boldsymbol{J}_{pv}^{\star}\Delta v + \boldsymbol{J}_{p\theta}^{\star}\Delta\theta \qquad (p \text{ balance})$$

$$q = q_0 + \boldsymbol{J}_{qv}^{\star}\Delta v + \boldsymbol{J}_{q\theta}^{\star}\Delta\theta \qquad (q \text{ balance})$$

$$s_0 \boldsymbol{J}_{sv}^{\star}\Delta v + \boldsymbol{J}_{s\theta}^{\star}\Delta\theta \leq \alpha \cdot \boldsymbol{s}^{\max} \qquad (\text{flow limits})$$

$$\underline{\boldsymbol{v}} \leq \boldsymbol{v} + \boldsymbol{\Delta v} \leq \overline{\boldsymbol{v}} \qquad (\text{voltage limits})$$

$$-72^\circ \leq E\theta - \phi \leq 72^\circ \qquad (\text{angle limits})$$

$$p_i = \sum_{j\in J_i^{\mathrm{pr}}} p_j - \sum_{j\in J_i^{\mathrm{cs}}} p_j - \sum_{j\in J_i^{\mathrm{dc}}} p_j^{\mathrm{fr/to}} \qquad (p \text{ injection})$$

$$q_i = \sum_{j\in J_i^{\mathrm{pr}}} q_j - \sum_{j\in J_i^{\mathrm{cs}}} q_j - \sum_{j\in J_i^{\mathrm{dc}}} q_j^{\mathrm{fr/to}} \qquad (q \text{ injection})$$

[10, eq. (109)-(118)] (producer limits)
[10, eq. (119)-(128)] (consumer limits)
[10, eq. (152)-(156)] (dc line limits)

C. Reserve variable cleanup

The GO3 clearing problem is filled with reserve variables. In order to further help Adam, we often run a "reserve variable cleanup" LP; this procedure very quickly tunes reserve variables via Proj. 4 in order to minimize very costly reserve shortfalls in the cheapest way possible. These (parallelizable) LPs solve on the order of seconds on very large systems, but they save Adam a significant amount of computational effort.

[3] In some cases, the full LP is too large to be solved all at once (even on the GO3 evaluation platform, with 64 CPU cores and 256 GB of octa-channel DDR4-3200 memory). In these cases, we break the economic dispatch into parallelized sub-problems across shorter time periods; we then LP project device binaries to be feasible across all time (not shown here).

Projection 4: Reserve Product "Cleanup" Projection [LP]

⋆ *parallelizable across each time instance*

$$\begin{aligned}
\max_{\{\text{reserve variables}\}} \quad & -\sum_{j\in J^{\text{pr,cs}}}\left(z^{\text{rgu}}_{jt}+z^{\text{rgd}}_{jt}+z^{\text{scr}}_{jt}+z^{\text{nsc}}_{jt}+z^{\text{rru}}_{jt}\right)\\
& -\sum_{j\in J^{\text{pr,cs}}}\left(z^{\text{rrd}}_{jt}+z^{\text{qru}}_{jt}+z^{\text{qrd}}_{jt}\right)\\
& -\sum_{n\in N^{\text{p}}}\left(z^{\text{rgu}}_{nt}+z^{\text{rgd}}_{nt}+z^{\text{scr}}_{nt}+z^{\text{nsc}}_{nt}+z^{\text{rru}}_{nt}+z^{\text{rrd}}_{nt}\right)\\
& -\sum_{n\in N^{\text{q}}}\left(z^{\text{qru}}_{nt}+z^{\text{qrd}}_{nt}\right)\\
\text{s.t.} \quad & [10,\text{ eq. (20)-(47)}] \qquad \text{(zonal reserve penalties)}\\
& [10,\text{ eq. (80)-(108)}] \quad \text{(device reserve costs \& limits)}\\
& [10,\text{ eq. (109)-(128)}] \qquad \text{(device limits)}
\end{aligned}$$

D. Ramp-constrained power flow

Adam is an excellent power flow solver, but its convergence towards an "ϵ accurate" solution can be very slow, leading to unnecessary penalization in the final solution. In order to overcome this, the final step of the `QuasiGrad` solver performs a "ramp constrained" power flow projection. This projection is necessarily delicate, because it must respect the device limits – if it does not, the device variables (i.e., pq injections) have to be *re-projected* feasible via Proj. 1, necessitating *another* power flow solve; this cycle continues *ad infinitum*, with no reason for convergence. Serial power flow solves which respect all future ramp constraints are extremely slow, both because they must be solved serially, and because they necessarily include linking constraints with all future power flow planes[4].

In order to overcome this challenge, we first consider n parallel power flow problems

$$t_1 : \boldsymbol{f}_1(\boldsymbol{p}_1, \boldsymbol{q}_1, \boldsymbol{v}_1, \boldsymbol{\theta}_1) \tag{40a}$$
$$t_2 : \boldsymbol{f}_2(\boldsymbol{p}_1, \boldsymbol{q}_1, \boldsymbol{v}_1, \boldsymbol{\theta}_1) \tag{40b}$$
$$\vdots$$
$$t_n : \boldsymbol{f}_n(\boldsymbol{p}_n, \boldsymbol{q}_n, \boldsymbol{v}_n, \boldsymbol{\theta}_n) \tag{40c}$$

linked via ramp constraints

$$t_1 : \boldsymbol{p}_0 + \boldsymbol{d}^{\text{rd}}_1 \le \boldsymbol{p}_1 \le \boldsymbol{p}_0 + \boldsymbol{d}^{\text{ru}}_1 \tag{41a}$$
$$t_2 : \boldsymbol{p}_1 + \boldsymbol{d}^{\text{rd}}_2 \le \boldsymbol{p}_2 \le \boldsymbol{p}_1 + \boldsymbol{d}^{\text{ru}}_2 \tag{41b}$$
$$\vdots$$
$$t_n : \boldsymbol{p}_{n-1} + \boldsymbol{d}^{\text{rd}}_n \le \boldsymbol{p}_n \le \boldsymbol{p}_{n-1} + \boldsymbol{d}^{\text{ru}}_n, \tag{41c}$$

where $\boldsymbol{p}_0$, $\boldsymbol{d}^{\text{ru}}_i$, and $\boldsymbol{d}^{\text{rd}}_i$ are constants, and (41) represent an exact transformation of the ramp limits [10, eqs. (68)-(74)] (once all binaries are frozen).

In order to tractably solve (40)-(41), we separate all network devices into two groups: the first group, termed group a, has its power injections frozen at t_2, t_4, t_6, etc., and second group, termed group b, has its power injections frozen at t_1, t_3, t_5, etc. Fig 5 demonstrates these alternatively frozen groupings. If a device d power is frozen at time t, we say it belongs to set $\mathcal{F}^t_d$. Using these groupings, we pose the following associated constraints, where bracketed constraints are only enforced for the associated grouping (a or b):

$$\begin{aligned}
t_1 : & \left[\boldsymbol{p}_0 + \boldsymbol{d}^{\text{rd}}_1 \le \boldsymbol{p}_1 \le \boldsymbol{p}_0 + \boldsymbol{d}^{\text{ru}}_1\right]_a, \; \left[\boldsymbol{p}_1 = \boldsymbol{p}^0_1\right]_b,\\
& \left[\boldsymbol{p}_1 + \boldsymbol{d}^{\text{rd}}_2 \le \boldsymbol{p}_2 \le \boldsymbol{p}_1 + \boldsymbol{d}^{\text{ru}}_2\right]_a, \qquad (42\text{a})\\
t_2 : & \left[\boldsymbol{p}_1 + \boldsymbol{d}^{\text{rd}}_2 \le \boldsymbol{p}_2 \le \boldsymbol{p}_1 + \boldsymbol{d}^{\text{ru}}_2\right]_b, \; \left[\boldsymbol{p}_2 = \boldsymbol{p}^0_2\right]_a,\\
& \left[\boldsymbol{p}_2 + \boldsymbol{d}^{\text{rd}}_3 \le \boldsymbol{p}_3 \le \boldsymbol{p}_2 + \boldsymbol{d}^{\text{ru}}_3\right]_b, \qquad (42\text{b})\\
t_3 : & \left[\boldsymbol{p}_2 + \boldsymbol{d}^{\text{rd}}_3 \le \boldsymbol{p}_3 \le \boldsymbol{p}_2 + \boldsymbol{d}^{\text{ru}}_3\right]_a, \; \left[\boldsymbol{p}_3 = \boldsymbol{p}^0_3\right]_b,\\
& \left[\boldsymbol{p}_3 + \boldsymbol{d}^{\text{rd}}_4 \le \boldsymbol{p}_4 \le \boldsymbol{p}_3 + \boldsymbol{d}^{\text{ru}}_4\right]_a, \qquad (42\text{c})\\
& \vdots
\end{aligned}$$

where devices are alternatively frozen and ramp constrained. Using this structure, the following result holds.

Theorem 1. *Assume* (41) *initially holds. By enforcing* (42)*, the power balance problems* (40) *can be solved in parallel while maintaining ramp feasibility* (41) *across all devices.*

Proof. Going sequentially, (42a) directly implies (41a), since the initialized injections are ramp rate feasible. At t_2, (42b) implies (41b). This is because $[\boldsymbol{p}_2]_a$ was chosen in t_1 such that the t_2 ramp constraint for devices in group a would be satisfied (these devices are frozen at t_2). The logic of choosing a device injection such that its ramp rate constraints are feasible at both the given and the following time step, and then freezing the device at the following time step, holds through to t_n. □

The parallelized ramp-constrained power flow projection is given in Proj. 5. As with Proj. 3, this projection is iteratively re-solved as new linearization points are identified, thus driving the AC power balance constraints to satisfaction (the "p balance" and "q balance" constraints are linearized power mismatch expressions). Multi-period energy constraints are not enforced in this routine, since they inherently "break" the parallelizability of the routine. In practice, however, this routine makes very small operational adjustments and rarely had any effect on the multi-period energy score. This projection strikes a balance between speed (it is parallelizable) and solution quality, since at every time, one half of the devices can have their set-points updated in service of finding a minimally invasive power flow solution. This projection offers a useful solution to one of the most challenging problems faced by the author in solving GO3.

E. The `QuasiGrad` solver

We now algorithmically introduce the full `QuasiGrad` solver. At a high level, the solver initializes a solution with an economic dispatch, and then it solves a series of NLPs while sequentially rounding and fixing binaries into feasible positions. The full algorithm is presented in Alg. 4.

[4] For example, if the active power set-point of a device is to be altered at t_1, then we must ensure there exists a ramp-feasible power injection at t_2, but if the power at t_2 is being changed, then we must ensure there exists a ramp-feasible power injection at t_3, etc., until t_f.

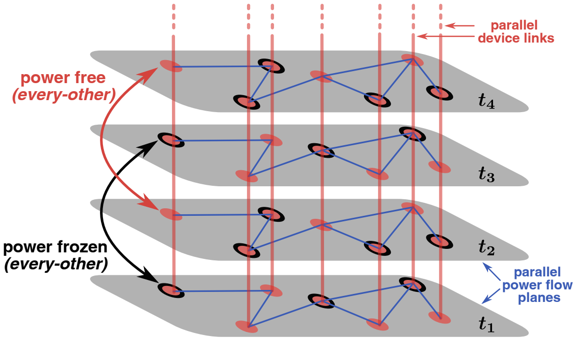

Fig. 5: Ramp-constrained power flow planes.

Projection 5: Ramp-Constrained Power Flow at Time t [QP]

⋆ *parallelizable across each time instance*

$$\begin{aligned}
\min_{\boldsymbol{x}_c}\quad & \gamma_1\left\|\boldsymbol{p}_g-\boldsymbol{p}_g^0\right\|_2+\gamma_2\left\|\boldsymbol{q}_g-\boldsymbol{q}_g^0\right\|_2+\gamma_{3/4}\boldsymbol{\Delta v/\theta}^T\boldsymbol{\Delta v/\theta} & \\
\text{s.t.}\quad & p=p_0+\boldsymbol{J}_{pv}^{\star}\Delta v+\boldsymbol{J}_{p\theta}^{\star}\Delta\theta & (p\text{ balance})\\
& q=q_0+\boldsymbol{J}_{qv}^{\star}\Delta v+\boldsymbol{J}_{q\theta}^{\star}\Delta\theta & (q\text{ balance})\\
& \underline{\boldsymbol{v}}\leq\boldsymbol{v}+\Delta\boldsymbol{v}\leq\overline{\boldsymbol{v}} & (\text{voltage limits})\\
& p_i=\sum_{j\in J_i^{\text{pr}}}p_j-\sum_{j\in J_i^{\text{cs}}}p_j-\sum_{j\in J_i^{\text{dc}}}p_j^{\text{fr/to}} & (p\text{ injection})\\
& q_i=\sum_{j\in J_i^{\text{pr}}}q_j-\sum_{j\in J_i^{\text{cs}}}q_j-\sum_{j\in J_i^{\text{dc}}}q_j^{\text{fr/to}} & (q\text{ injection})\\
& [10,\text{ eqs. (68)-(74)}] & (\textbf{ramp limits})\\
& [10,\text{ eq. (109)-(118)}] & (\text{producer limits})\\
& [10,\text{ eq. (119)-(128)}] & (\text{consumer limits})\\
& [10,\text{ eq. (152)-(156)}] & (\text{dc line limits})\\
& p_j=p_j^0,\ j\in\{\{J^{\text{pr}}\cup J^{\text{cs}}\}\cap\mathcal{F}_d^t\} & (\textbf{\textit{frozen subset via}}\ (42))
\end{aligned}$$

IV. Test Results

In this section, we present simulated test results collected from division 1 (real time market) of the C3E3.1 GO3 testcase library. We the provide brief comments about the official GO3 test results, which are partially available (in a very aggregated form) at the following footnote[5].

The C3E3.1 dataset[6] contains six division 1 (i.e., real time market with 18 time periods) test cases: the 617-, 1576-, 4224-, 6049-, 6717-, 8316-, and 23643-bus systems. We provide local (i.e., laptop simulated) test results for a single QuasiGrad solve of each of these cases, excluding the 23643-bus system, which cannot be solved locally due to memory constraints. For benchmarking, we compare the QuasiGrad solution to the economic dispatch solution of Proj. 2, which is a global upper bound. All tests are run in Julia v1.10.0-beta1 on a Dell XPS with 16.0 GB of RAM. Julia is launched with access to 6 physical CPU threads for parallelization. Each solver terminates in under 10 minutes (600 seconds), as stipulated in GO3. All test results are confirmed feasible by the GO3 C3DataUtilities Python library[7], and all reported scores are crosschecked against the C3DataUtilities solution scores.

Algorithm 4 `QuasiGrad`

```
Require: Total wall clock time ⏱, total number of binaries $n_b$, number of binaries $n_b^+$ to freeze after every NLP (Adam) solve
$\mathcal{F} \leftarrow \emptyset$ (no fixed binaries)
Initialize with economic dispatch via Proj. 2
while $|\mathcal{F}| < n_b$ (some binaries are not fixed) do
    Solve linearized power flow projection via Proj. 3
    Cleanup reserve variables via Proj. 4
    for $t_w \in$ ⏱$_s$ do        ▷ run adam for subset of wall clock time
        Evaluate and backpropagate objective (18b)
        Solve and backpropagate contingencies via Alg. 2
        Feed gradients (36) to Adam via Alg. 3
        Clip all states (Alg. 3, line 6)
    end
    Project device binaries and variables feasible via Proj. 1
    Add $n_b^+$ binaries to frozen binary set $\mathcal{F}$ based on projection
end
Snap shunts
Solve linearized power flow projection via Proj. 3
Cleanup reserve variables via Proj. 4
Run final Adam solve
Project device variables feasible via Proj. 1 (all binaries fixed)
Solve ramp-constrained power flow via Proj. 5
Cleanup reserve variables via Proj. 4
return feasible continuous $\boldsymbol{x}_c$ and discrete $\boldsymbol{x}_d$ solution vectors
```

TABLE I: C3E3.1 Division 1 Test Results

testcase	**617**	**1576**	**4224**	**6049**	**6717**	**8316**
z^{ms}	4.52e7	9.96e7	8.95e7	8.45e7	1.34e8	1.01e9
z^{ed}	4.54e7	1.02e8	9.20e7	1.08e8	1.37e8	1.16e9
gap	**99.8%**	**98.1%**	**97.3%**	**78.3%**	**97.7%**	**87.0%**
z^{base}	4.53e7	1.00e8	8.96e7	8.45e7	1.34e8	1.01e9
z^{t}	4.53e7	1.00e8	8.96e7	8.45e7	1.34e8	1.01e9
	Relevant Penalty Breakdowns (%):					
z^{en}	98.6	62.9	90.8	51.7	66.0	89.8
$z^{\text{on/p/d}}$	0.02	13.7	6.94	46.9	0.11	8.90
z^{ac}	0.08	0.0	0.06	0.20	0.23	0.43
z^{xfm}	0.0	0.0	0.49	0.17	0.0	0.04
z^{pq}	0.0	0.0	0.0	0.0	0.0	0.0
z^{zonal}	0.06	0.0	0.0	0.0	32.6	0.0
$z^{\text{ctg-min}}$	1.18	23.1	1.21	0.56	1.08	0.55
$z^{\text{ctg-avg}}$	0.05	0.32	0.52	0.38	0.01	0.28

Test results are reported in Table I, which reports the *gap* of the QuasiGrad solution relative to the economic dispatch (i.e., $100.0 \times z^{\text{ms}}/z^{\text{ed}}$). The gaps are generally quite high, indicating that the QuasiGrad solver was able to find high quality ACUC solutions within the 10 minute time allowance.

As of the submission of this manuscript, the full GO3 results have not yet been released. Overall, the QuasiGrad solver performed well, but it was not a top performing algorithm. Across the 667 tests, QuasiGrad found aggregated market surplus scores that were within 31%, 5%, and 44%, respectively, of the top performing team in the three market divisions, with 65 scores that were within the top 5 best. While there is significant room for improvement, the results demonstrate

[5] https://gocompetition.energy.gov/challenges/challenge-3/Leaderboards/Event-4

[6] https://gocompetition.energy.gov/challenges/600650/datasets

[7] https://github.com/GOCompetition/C3DataUtilities

both the validity of this new approach, and the potential for it to be competitive with conventional approaches in the future.

V. Conclusion

This paper introduced an Adam-based solver, called `QuasiGrad` (summarized in Alg. 4), capable of solving large-scale, reserve and security ACUC problems. The solver, which is released publicly as the Julia package `QuasiGrad.jl` [9], efficiently parallelizes backpropagation and variable projection processes, making efficient use of parallel computing hardware. The solver is able to find high quality solutions to large-scale problems in short periods of time, and by design, the approach is hyper-scalable. Due to its ability to efficiently parallelize, `QuasiGrad` runs monotonically and predictably faster when it is given monotonically more computational resources. Future work will seek to test the `QuasiGrad` solver on GPU hardware (which was not part of the GO3 competition). Planned follow-on work will provide deeper testing analysis and a more thorough investigation into the specific benefits of the innovations proposed in this paper. Furthermore, future directions should investigate the capacity for `QuasiGrad` to help train physics-informed machine learning models, such as the Lagrange multiplier penalty-based learning models used in, e.g., [20].

VI. Acknowledgements

The author gratefully acknowledges support and helpful feedback from Spyros Chatzivaseiliadis, Daniel Molzahn, Amrit Pandey, Mads Almassalkhi, and the GO3 organizers (especially Steven Elbert, Arun Veeramany, and Jesse Holzer). Finally, the author would like to thank Katherine and Rosemary Chevalier for their continued support throughout the development of QuasiGrad.

Appendix A

`QuasiGrad` was developed in `Julia v1.10.0`. All functions are type stable, and all memory is pre-allocated, thus minimizing the amount of garbage collection. Parallelization is achieved through `Threads.@threads` and `polyester.@batch`. `LoopVectorization.jl`, and its macro `@tturbo`, are used extensively to accelerate computations. The preconditioned conjugate gradient solver, `cg!`, is called from `IterativeSolvers.jl`. The `lldl` function from `Preconditioners.jl` is used to build the limited memory LDL^T preconditioner for contingency solving. `JuMP.jl` and `Gurobi.jl` are used to formulate and solve all optimizations (LPs, MILPs, and QPs). Gurobi 11 (and an associated academic license) was used.

References

[1] I. Aravena, D. K. Molzahn *et al.*, "Recent developments in security-constrained ac optimal power flow: Overview of challenge 1 in the arpa-e grid optimization competition," *Operations Research*, 2023.

[2] F. Safdarian, J. Snodgrass *et al.*, "Grid optimization competition on synthetic and industrial power systems," in *2022 North American Power Symposium (NAPS)*, 2022, pp. 1–6.

[3] J. Holzer, C. Coffrin *et al.*, "Grid optimization competition challenge 2 problem formulation," https://gocompetition.energy.gov/sites/default/files/Challenge2_Problem_Formulation_20210531.pdf.

[4] H. Hijazi, G. Wang, and C. Coffrin, "Gravity: A mathematical modeling language for optimization and machine learning," 2018.

[5] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[6] D. Choi, C. J. Shallue, Z. Nado, J. Lee, C. J. Maddison, and G. E. Dahl, "On empirical comparisons of optimizers for deep learning," *arXiv preprint arXiv:1910.05446*, 2019.

[7] S. Wang, H. Zhang, K. Xu, X. Lin, S. Jana, C.-J. Hsieh, and J. Zico Kolter, "Beta-CROWN: Efficient Bound Propagation with Per-neuron Split Constraints for Complete and Incomplete Neural Network Robustness Verification," *arXiv e-prints*, p. arXiv:2103.06624, Mar. 2021.

[8] M. N. Müller, C. Brix, S. Bak, C. Liu, and T. T. Johnson, "The third international verification of neural networks competition (vnn-comp 2022): Summary and results," 2023.

[9] S. Chevalier, "Quasigrad.jl," Feb. 2024. [Online]. Available: https://github.com/SamChevalier/QuasiGrad.jl

[10] J. Holzer, C. Coffrin *et al.*, "Grid optimization competition challenge 3 problem formulation," https://gocompetition.energy.gov/sites/default/files/Challenge3_Problem_Formulation_20230126.pdf.

[11] G. Strang, *Linear algebra and learning from data*. SIAM, 2019.

[12] S. Chevalier, "Supplementary information for a parallelized, adam-based solver for reserve and security constrained ac unit commitment," https://samchevalier.github.io/docs/SI.pdf.

[13] S. Abhyankar, J. Drgoňa, A. Tuor, and A. August, "Neuro-physical dynamic load modeling using differentiable parametric optimization," in *2023 IEEE Power & Energy Society General Meeting (PESGM)*, 2023, pp. 1–5.

[14] M. R. Hestenes, E. Stiefel *et al.*, "Methods of conjugate gradients for solving linear systems," *Journal of research of the National Bureau of Standards*, vol. 49, no. 6, pp. 409–436, 1952.

[15] J. T. Holzer, Y. Chen, Z. Wu, F. Pan, and A. Veeramany, "Fast simultaneous feasibility test for security constrained unit commitment," *IEEE Transactions on Power Systems*, pp. 1–10, 2023.

[16] R. Horn and C. Johnson, *Matrix Analysis*. Cambridge University Press, 1990.

[17] Y. Huang, T. Ding, C. Mu, X. Zhang, Y. He, and M. Shahidehpour, "Distributed slack-bus based dc optimal power flow with transmission loss: A second-order cone programming approach and sufficient conditions," *IEEE Transactions on Automation Science and Engineering*, pp. 1–13, 2023.

[18] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[19] T. McNamara, A. Pandey, A. Agarwal, and L. Pileggi, "Two-stage homotopy method to incorporate discrete control variables into ac-opf," *Electric Power Systems Research*, vol. 212, p. 108283, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0378779622004722

[20] T. W. K. Mak, M. Chatzos, M. Tanneau, and P. V. Hentenryck, "Learning regionally decentralized ac optimal power flows with admm," *IEEE Transactions on Smart Grid*, vol. 14, no. 6, pp. 4863–4876, 2023.