The Complexity of Gradient Descent: CLS = PPAD ∩ PLS

John Fearnley
john.fearnley@liverpool.ac.uk
University of Liverpool
Liverpool, UK

Paul W. Goldberg
paul.goldberg@cs.ox.ac.uk
University of Oxford
Oxford, UK

Alexandros Hollender
alexandros.hollender@cs.ox.ac.uk
University of Oxford
Oxford, UK

Rahul Savani
rahul.savani@liverpool.ac.uk
University of Liverpool
Liverpool, UK

ABSTRACT

We study search problems that can be solved by performing Gradient Descent on a bounded convex polytopal domain and show that this class is equal to the intersection of two well-known classes: PPAD and PLS. As our main underlying technical contribution, we show that computing a Karush-Kuhn-Tucker (KKT) point of a continuously differentiable function over the domain $[0, 1]^2$ is PPAD ∩ PLS-complete. This is the first natural problem to be shown complete for this class. Our results also imply that the class CLS (Continuous Local Search) – which was defined by Daskalakis and Papadimitriou as a more "natural" counterpart to PPAD ∩ PLS and contains many interesting problems – is itself equal to PPAD ∩ PLS.

CCS CONCEPTS

• **Theory of computation → Problems, reductions and completeness**; • **Mathematics of computing → Continuous functions**.

KEYWORDS

TFNP, computational complexity, continuous optimization

1 INTRODUCTION

It is hard to overstate the importance of Gradient Descent. As noted by Jin et al. [26], "Machine learning algorithms generally arise via formulations as optimization problems, and, despite a massive classical toolbox of sophisticated optimization algorithms and a major modern effort to further develop that toolbox, the simplest algorithms—gradient descent, which dates to the 1840s [6] and stochastic gradient descent, which dates to the 1950s [35]—reign supreme in machine learning." Jin et al. [26] continue by highlighting the simplicity of Gradient Descent as a key selling-point, and the importance of theoretical analysis in understanding its efficacy in non-convex optimization.

In its simplest form, which we consider in this paper, Gradient Descent attempts to find a minimum of a continuously differentiable function f over some domain D, by starting at some point x_0 and iterating according to the update rule

$$x_{k+1} \leftarrow x_k - \eta \nabla f(x_k)$$

where η is some fixed step size. The algorithm is based on the fundamental fact that for any point x the term $-\nabla f(x)$ points in the direction of steepest descent in some sufficiently small neighbourhood of x. However, in the unconstrained setting—where the domain is the whole space—it is easy to see that Gradient Descent can at best find a stationary point. Indeed, if the gradient is zero at some point, then there is no escape. Note that a stationary point might be a local minimum, but it could also be a saddle point or even a local maximum. Similarly, in the constrained setting—where the domain D is no longer the whole space—Gradient Descent can at best find a point x that satisfies the Karush-Kuhn-Tucker (KKT) optimality conditions. Roughly, the KKT conditions say that the gradient of f is zero at x, or if not, x is on the boundary of D and any further local improvement would take us outside D.

In this paper we investigate the complexity of finding a point where Gradient Descent terminates—or equivalently, as we will see, a KKT point—when the domain is *bounded*. It is known that a global or even a local minimum cannot be found in polynomial time unless P = NP [2, 32]. Indeed, even deciding whether a point is a local minimum is already co-NP-hard [32]. In contrast, it is easy to check whether a point satisfies the KKT conditions. In general, finding a KKT point is hard, since even deciding whether a KKT point exists is NP-hard in the unconstrained setting [1]. However, when the domain is bounded, a KKT point is guaranteed to exist! This means that in our case, we are looking for something that can be verified efficiently and that necessarily exists. Intuitively, it seems that this problem should be more tractable. This intuition can be made formal by noting that these two properties place the problem in the complexity class TFNP of *total* search problems in NP: any instance has at least one solution, and a solution can be checked in polynomial time. A key feature of such problems is that they cannot be NP-hard unless NP = co-NP [29]. TFNP problems have been classified via certain "syntactic subclasses" of TFNP, of which PPAD and PLS are two of the most important ones.

1.1 NP Total Search Classes: PPAD, PLS, CLS

As discussed by Papadimitriou [34], TFNP is unlikely to have complete problems, and various *syntactic* subclasses have been used to classify the many diverse problems that belong to it. Among them, the classes PPAD and PLS (introduced by Papadimitriou [34] and Johnson et al. [27] respectively) have been hugely successful in this regard. Each of these classes has a corresponding *computationally*

inefficient existence proof principle, one that when applied in a general context, does not yield a polynomial-time algorithm.[1] In the case of PPAD this is the *parity argument on a directed graph*, equivalent to the existence guarantee of *Brouwer fixpoints*: a Brouwer function is a continuous function $f : D \to D$ where D is a convex compact domain, and Brouwer's fixed point theorem guarantees a point x for which $f(x) = x$. PPAD has been widely used to classify problems of computing game-theoretic equilibria (a long line of work on Nash equilibrium computation beginning with Daskalakis et al. [12], Chen et al. [10], and market equilibria, e.g., Chen et al. [8]). PPAD also captures diverse problems in combinatorics and cooperative game theory [28].

PLS, for "Polynomial Local Search", captures problems of finding a local minimum of an objective function f, in contexts where any candidate solution x has a local neighbourhood within which we can readily check for the existence of some other point having a lower value of f. Many diverse local optimization problems have been shown complete for PLS, attesting to its importance. Examples include searching for a local optimum of the TSP according to the Lin-Kernighan heuristic [33], and finding pure Nash equilibria in many-player congestion games [17].

The complexity class CLS ("Continuous Local Search") was introduced by Daskalakis and Papadimitriou [13] to classify various important problems that lie in both PPAD and PLS. PPAD and PLS are believed to be strictly incomparable—one is not a subset of the other—a belief supported by oracle separations [4]. It follows from this that problems belonging to both classes cannot be complete for either one of them. CLS is seen as a strong candidate for capturing the complexity of some of those important problems, but, prior to this work, only two problems related to general versions of Banach's fixed point theorem were known to be CLS-complete [15, 19]. An important result—supporting the claim that CLS-complete problems are hard to solve—is that the hardness of CLS can be based on the cryptographic assumption of indistinguishability obfuscation [24]. Prior to the present paper, it was generally believed that CLS is a proper subset of PPAD ∩ PLS, as conjectured by Daskalakis and Papadimitriou [13].

1.2 Our Contribution and Its Significance

Our main result is to show that finding a point where Gradient Descent on a continuously differentiable function terminates—or equivalently a KKT point—is PPAD ∩ PLS-complete, when the domain is a bounded convex polytope. This continues to hold even when the domain is as simple as the unit square $[0, 1]^2$. The PPAD ∩ PLS-completeness result applies to the "white box" model, where functions are represented as arithmetic circuits.

Computational Hardness. As an immediate consequence, our result provides convincing evidence that the problem is computationally hard. First of all, there are reasons to believe that PPAD ∩ PLS is hard simply because PPAD and PLS are believed to be hard. Indeed, if PPAD ∩ PLS could be solved in polynomial time, then, given an instance of a PPAD-complete problem and an instance of a PLS-complete problem, we would be able to solve at least one of the two instances in polynomial time. Furthermore, since CLS ⊆ PPAD ∩ PLS, the above-mentioned cryptographic hardness of CLS applies automatically to PPAD ∩ PLS, and thus to our problem of interest.

Continuous Local Search. Since Gradient Descent is just a special case of continuous local search, our hardness result implies that

$$\text{CLS} = \text{PPAD} \cap \text{PLS}$$

which disproves the widely believed conjecture by Daskalakis and Papadimitriou [13] that the containment is strict. Our result also allows us to resolve an ambiguity in the original definition of CLS by showing that the high-dimensional version of the class reduces to the 2-dimensional version of the class (the 1-dimensional version is computationally tractable, so no further progress is to be made). Equality to PPAD ∩ PLS also applies to a linear version of CLS analogous to the class Linear-FIXP of Etessami et al. [16].

PPAD ∩ PLS. Perhaps more importantly, our result establishes PPAD ∩ PLS as an important complexity class that captures the complexity of interesting problems. It was previously known that one can construct a problem complete for PPAD ∩ PLS by gluing together two problems, one for each class (see Section 2.2), but the resulting problem is highly artificial. In contrast, the Gradient Descent problem we consider is clearly natural and of separate interest.

Some TFNP classes can be characterized as the set of all problems solved by some type of algorithm. For instance, PPAD is the class of all problems that can be solved by the Lemke-Howson algorithm. PLS is the class of all problems that can be solved by general local search methods. Analogously, one can define the class GD containing all problems that can be solved by the Gradient Descent algorithm on a bounded domain, i.e., that reduce to our Gradient Descent problem in polynomial time. Our result shows that GD = PPAD ∩ PLS. In other words, the class PPAD ∩ PLS, which is obtained by combining PPAD and PLS in a completely artificial way, turns out to have a very natural characterization:

> PPAD ∩ PLS *is the class of all problems that can be solved by performing Gradient Descent on a bounded domain.*

Our new characterization has already been very useful in the context of Algorithmic Game Theory, where it was recently used by Babichenko and Rubinstein [3], to show PPAD ∩ PLS-completeness of computing mixed Nash equilibria of congestion games.

1.3 Further Related Work

Following the definition of CLS by Daskalakis and Papadimitriou [13], two CLS-complete problems were identified: Banach [15] and MetametricContraction [19]. Banach is a computational presentation of Banach's fixed point theorem in which the metric is presented as part of the input (and could be complicated). Banach fixpoints are unique, but CLS problems do not in general have unique solutions, and the problem Banach circumvents that obstacle by allowing certain "violation" solutions, such as a pair of points witnessing that f is not a contraction map. MetametricContraction is a generalisation of Banach, where the metric is replaced by a slightly relaxed notion called a meta-metric.

[1] The other well-known such classes, less relevant to the present paper, are PPA and PPP; it is known that PPAD is a subset of PPA and also of PPP. These set-theoretic containments correspond directly to the strength, or generality, of the corresponding proof principles.

Chatziafratis et al. [7] showed that online gradient descent can encode general PSPACE-computations. In contrast, our result provides evidence that the problem itself (which gradient descent attempts to solve) is hard. The distinction between these two types of statements is most clearly apparent in the case of linear programming, where the simplex method can encode arbitrary PSPACE computations [21], while the problem itself can be solved in polynomial time.

Daskalakis et al. [14] study nonlinear *min-max* optimization, a conceptually more complex problem than the purely "min" optimization studied here. The PPAD-completeness they obtain reflects the extra structure present in such problems. An important point is that our hardness result requires inverse-exponential parameters, whereas Daskalakis et al. [14] achieve hardness with inverse-polynomial parameters—for us the inverse-exponential parameters are a necessary evil, since the problem can otherwise be solved in polynomial time, even in high dimension (simply by running Gradient Descent). Finally, note that in contrast to our hardness result, in the special case of *convex* optimization our problem can be solved efficiently, even in high dimension and with inverse-exponential precision.

2 OVERVIEW

In this section we give a condensed and informal overview of the concepts, ideas, and techniques of this paper. We begin by providing informal definitions of the problems of interest and the complexity classes. We then present an overview of our results, along with the high-level ideas of our main reduction.

2.1 The Problems of Interest

The motivation for the problems we study stems from the ultimate goal of minimizing a continuously differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ over some domain D. As mentioned in the introduction, this problem is known to be intractable, and so we instead consider relaxations where we are looking for a point where Gradient Descent terminates, or for a KKT point. Our investigation is restricted to bounded domains, namely we consider the setting where the domain D is a bounded convex polytope defined by a collection of linear inequalities. Furthermore, we also assume that the function f and its gradient ∇f are Lipschitz-continuous over D, for some Lipschitz constant L provided in the input. Let $C^1_L(D, \mathbb{R})$ denote the set of continuously differentiable functions f from D to $\mathbb{R}$, such that f and ∇f are L-Lipschitz.

In order to define our Gradient Descent problem, we need to specify what we mean by "a point where Gradient Descent terminates". We consider the following two stopping criteria for Gradient Descent: (a) stop when we find a point such that the next iterate does not improve the objective function value, or (b) stop when we find a point such that the next iterate is the same point. In practice, of course, Gradient Descent is performed with some underlying precision parameter $\varepsilon > 0$. Thus, the appropriate stopping criteria are: (a) stop when we find a point such that the next iterate improves the objective function value by less than ε, or (b) stop when we find a point such that the next iterate is at most ε away. Importantly, note that, given a point, both criteria can be checked efficiently. This ensures that the resulting computational problems lie in TFNP. The totality of the problems follows from the simple fact that a local minimum must exist (since the domain is bounded) and any local minimum satisfies the stopping criteria. The first stopping criterion has a local search flavour and so we call the corresponding problem GD-Local-Search. The second stopping criterion is essentially asking for an approximate fixed point of the Gradient Descent dynamics, and yields the GD-Fixpoint problem.

Since we are performing Gradient Descent on a bounded domain, we have to ensure that the next iterate indeed lies in the domain D. The standard way to achieve this is to use so-called Projected Gradient Descent, which computes the next iterate as usual and then projects it onto the domain. Define Π_D to be the projection operator, that maps any point in D to itself, and any point outside D to its closest point in D (under the Euclidean norm). The two Gradient Descent problems are defined as follows.

GD-Local-Search and GD-Fixpoint *(informal)*

Input: $\varepsilon > 0$, step size $\eta > 0$, domain D, $f \in C^1_L(D, \mathbb{R})$ and its gradient ∇f.

Goal: Compute any point where (projected) gradient descent for f on D terminates. Namely, find $x \in D$ such that x and its next iterate $x' = \Pi_D(x - \eta \nabla f(x))$ satisfy:

- for GD-Local-Search: $f(x') \geq f(x) - \varepsilon$, *($f$ decreases by at most ε)*
- for GD-Fixpoint: $\|x - x'\| \leq \varepsilon$. *($x'$ is ε-close to x)*

In a certain sense, GD-Local-Search is a PLS-style version of Gradient Descent, while GD-Fixpoint is a PPAD-style version.[2] We show that these two versions are computationally equivalent by a triangle of reductions (see Figure 3). The other problem in that triangle of equivalent problems is the KKT problem, defined below.

KKT *(informal)*

Input: $\varepsilon > 0$, domain D, $f \in C^1_L(D, \mathbb{R})$ and its gradient ∇f.

Goal: Compute any ε-KKT point of the minimization problem for f on domain D.

A point x is a KKT point if x is feasible (it belongs to the domain D), and x is either a zero-gradient point of f, or alternatively x is on the boundary of D and the boundary constraints prevent local improvement of f. "ε-KKT" relaxes the KKT condition so as to allow inexact KKT solutions with limited numerical precision.

Representation of f and ∇f. We consider these computational problems in the "white box" model, where some computational device computing f and ∇f is provided in the input. In our case, we assume that f and ∇f are presented as arithmetic circuits. In more detail, following Daskalakis and Papadimitriou [13], we consider arithmetic circuits that use the operations $\{+, -, \times, \max, \min, <\}$, and rational constants.[3] Another option would be to assume that

[2] A very similar version of GD-Fixpoint was also defined by Daskalakis et al. [14] and shown to be equivalent to finding an *approximate* local minimum (which is essentially the same as a KKT point).

[3] A subtle issue is that it might not always be possible to evaluate such a circuit efficiently, because of "repeated squaring". To avoid this issue, we restrict ourselves to what we call *well-behaved* arithmetic circuits. See Section 3 for more details.

the functions are given as polynomial-time Turing machines, but this introduces some extra clutter in the formal definitions of the problems. The definition with arithmetic circuits is cleaner, and, in any case, the complexity of the problems is the same in both cases.

Promise-version and total-version. Given an arithmetic circuit for f and one for ∇f, we know of no easy way of checking that the circuit for ∇f indeed computes the gradient of f, and that the two functions are indeed L-Lipschitz. There are two ways to handle this issue: (a) consider the promise version of the problem, where we restrict our attention to instances that satisfy these conditions, or (b) introduce "violation" solutions in the spirit of Daskalakis and Papadimitriou [13], i.e., allow as a solution a witness of the fact that one of the conditions is not satisfied. The first option is more natural, but the second option ensures that the problem is formally in TFNP. Thus, we use the second option for the formal definitions of our problems. However, we note that our "promise-preserving" reductions ensure that *our hardness results also hold for the promise versions of the problems.*

2.2 Complexity Classes

In this section we provide informal definitions of the relevant complexity classes, and discuss their key features. Formal definitions can be found in the full version [18] of this paper (and many others), but the high-level descriptions presented here are intended to be sufficient to follow the overview of our main proof.

PPAD. The complexity class PPAD is defined as the set of TFNP problems that reduce in polynomial time to problem End-of-Line.

> End-of-Line *(informal)*
>
> **Input**: A directed graph on the vertex set $[2^n]$, such that every vertex has in- and out-degree at most 1, and such that vertex 1 is a source.
>
> **Goal**: Find a sink of the graph, or any other source.

Importantly, the graph is not provided explicitly in the input, but instead we are given Boolean circuits that efficiently compute the successor and predecessor of each vertex. This means that the size of the graph can be exponential with respect to its description length. A problem is *complete* for PPAD if it belongs to PPAD and if End-of-Line reduces in polynomial time to that problem. Many variants of the search for a fixed point of a Brouwer function turn out to be PPAD-complete. This is essentially the reason why GD-Fixpoint, and thus the other two equivalent problems, lie in PPAD. See Figure 1 for an example of an instance of End-of-Line.

PLS. The complexity class PLS is defined as the set of TFNP problems that reduce in polynomial time to the problem Localopt.

> Localopt *(informal)*
>
> **Input**: Functions $V : [2^n] \to \mathbb{R}$ and $S : [2^n] \to [2^n]$.
>
> **Goal**: Find $v \in [2^n]$ such that $V(S(v)) \geq V(v)$.

The functions are given as Boolean circuits. A problem is *complete* for PLS if it belongs to PLS and if Localopt reduces in polynomial time to that problem. PLS embodies general local search

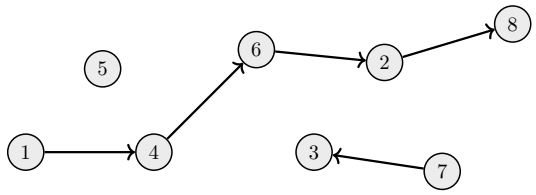

Figure 1: Example of an End-of-Line instance for $n = 3$. Here, the solutions are the vertices 3, 7 and 8.

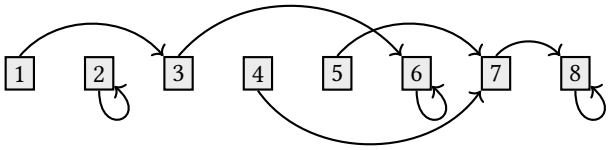

Figure 2: Example of an Iter instance for $n = 3$. Nodes 2, 6 and 8 are the fixed points and the solutions are nodes 3 and 7.

methods where one attempts to optimize some objective function by considering local improving moves. Our problem GD-Local-Search is essentially a special case of local search, and thus lies in PLS. In this paper we make use of the problem Iter, defined below, which is known to be PLS-complete [31].

> Iter *(informal)*
>
> **Input**: A function $C : [2^n] \to [2^n]$ such that $C(v) \geq v$ for all $v \in [2^n]$, and $C(1) > 1$.
>
> **Goal**: Find v such that $C(v) > v$ and $C(C(v)) = C(v)$.

For this problem, it is convenient to think of the nodes in $[2^n]$ as lying on a line, in increasing order. Then, any node is either a fixed point of C, or it is mapped to some node further to the right. We are looking for any node that is not a fixed point, but is mapped to a fixed point. It is easy to see that the condition $C(1) > 1$ ensures that such a solution must exist. See Figure 2 for an example.

PPAD ∩ PLS. The class PPAD ∩ PLS contains, by definition, all TFNP problems that lie in both PPAD and in PLS. Prior to our work, the only known way to obtain PPAD ∩ PLS-complete problems was to combine a PPAD-complete problem A and a PLS-complete problem B as follows [13].

> Either-Solution(A,B)
>
> **Input**: An instance I_A of A and an instance I_B of B.
>
> **Goal**: Find a solution of I_A or a solution of I_B.

In particular, the problem Either-Solution(End-of-Line,Iter) is PPAD ∩ PLS-complete, and this is the problem we reduce from to obtain our results.

CLS. Noting that all known PPAD ∩ PLS-complete problems looked very artificial, Daskalakis and Papadimitriou [13] defined the class

CLS ⊆ PPAD ∩ PLS, which combines PPAD and PLS in a more natural way. The class CLS is defined as the set of TFNP problems that reduce to the problem 3D-Continuous-Localopt.

3D-Continuous-Localopt *(informal)*

Input: $\varepsilon > 0$, L-Lipschitz functions $p : [0,1]^3 \to [0,1]$ and $g : [0,1]^3 \to [0,1]^3$.

Goal: Compute any approximate local optimum of p with respect to g. Namely, find $x \in [0,1]^3$ such that

$$p(g(x)) \geq p(x) - \varepsilon.$$

This problem is essentially a special case of the Localopt problem, where we perform local search over a continuous domain and where the functions are continuous. The formal definition of 3D-Continuous-Localopt includes violation solutions for the Lipschitz-continuity of the functions. We also consider a more general version of this problem, which we call General-Continuous-Localopt, where we allow any bounded convex polytope as the domain.

2.3 Results

The main technical contribution of this work is Theorem 4.1, which shows that the KKT problem is PPAD ∩ PLS-hard, even when the domain is the unit square $[0,1]^2$. The hardness also holds for the promise version of the problem, because the hard instances that we construct always satisfy the promises. We present the main ideas needed for this result in the next section, but we first briefly present the consequences of this reduction here.

A chain of reductions, presented in Section 5 and shown in Figure 3, which includes the "triangle" between the three problems of interest, establishes the following theorem.

Theorem 5.1. *The problems* GD-Local-Search, GD-Fixpoint, KKT *and* General-Continuous-Localopt *are* PPAD ∩ PLS*-complete, even when the domain is fixed to be the unit square* $[0,1]^2$*. This hardness result continues to hold even if one considers the promise-versions of these problems, i.e., only instances without violations.*

These reductions are domain-preserving—which means that they leave the domain D unchanged—and promise-preserving—which means that they are also valid reductions between the promise versions of the problems. As a result, the other problems "inherit" the hardness result for KKT, including the fact that it holds for $D = [0,1]^2$ and even for the promise versions.

Consequences for CLS. The PPAD ∩ PLS-hardness of General-Continuous-Localopt on domain $[0,1]^2$, and thus also on domain $[0,1]^3$, immediately implies the following surprising collapse.

Theorem 6.1. CLS = PPAD ∩ PLS.

As a result, it also immediately follows that the two known CLS-complete problems [15, 19] are in fact PPAD ∩ PLS-complete.

Theorem 6.2. Banach *and* MetametricContraction *are* PPAD ∩ PLS*-complete.*

The fact that our hardness result holds on domain $[0,1]^2$ implies that the n-dimensional variant of CLS is equal to the 2-dimensional version, a fact that was not previously known. Furthermore, since our results hold even when General-Continuous-Localopt is considered as a promise problem, this implies that the definition of CLS is robust with respect to the removal of violations (promise-CLS = CLS). Finally, we also show that restricting the circuits to be linear arithmetic circuits (that compute piecewise-linear functions) does not yield a weaker class, i.e., 2D-Linear-CLS = CLS. This result is obtained by showing that linear circuits can be used to efficiently approximate any Lipschitz-continuous function with *arbitrary precision* (proved in the appendix of the full version), which might be of independent interest. All the consequences for CLS are discussed in detail in Section 6.

2.4 Proof Overview for Theorem 4.1

In this section we provide a brief overview of our reduction from the (PPAD ∩ PLS-complete) problem Either-Solution(End-of-Line, Iter) to the KKT problem on domain $[0,1]^2$.

Given an instance I^{EOL} of End-of-Line and an instance I^{ITER} of Iter, we construct an instance $I^{\text{KKT}} = (\varepsilon, f, \nabla f, L)$ of the KKT problem on domain $[0,1]^2$ such that from any ε-KKT point of f, we can efficiently obtain a solution to either I^{EOL} or I^{ITER}. The function f and its gradient ∇f are first defined on an exponentially small grid on $[0,1]^2$, and then extended within every small square of the grid by using bicubic interpolation. This ensures that the function is continuously differentiable on the whole domain. The most interesting part of the reduction is how the function is defined on the grid points, by using information from I^{EOL}, and then, where necessary, also from I^{ITER}.

Embedding I^{EOL}. The domain is first subdivided into $2^n \times 2^n$ *big squares*, where $[2^n]$ is the set of vertices in I^{EOL}. The big squares on the diagonal (shaded in Figure 4) represent the vertices of I^{EOL} and the function f is constructed so as to embed the directed edges in the graph of I^{EOL}. If the edge (v_1, v_2) in I^{EOL} is a forward edge, i.e, $v_1 < v_2$, then there will be a "green path" going from the big square of v_1 to the big square of v_2. On the other hand, if the edge (v_1, v_2) in I^{EOL} is a backward edge, i.e., $v_1 > v_2$, then there will be an "orange path" going from the big square of v_1 to the big square of v_2. These paths are shown in Figure 4 for the corresponding example instance of Figure 1.

The function f is constructed such that when we move along a green path the value of f decreases. Conversely, when we move along an orange path the value of f increases. Outside the paths, f is defined so as to decrease towards the origin $(0,0) \in [0,1]^2$, where the green path corresponding to the source of I^{EOL} starts. As a result, we show that an ε-KKT point can only occur in a big square corresponding to a vertex v of I^{EOL} such that (a) v is a solution of I^{EOL}, or (b) v is *not* a solution of I^{EOL}, but its two neighbours (in the I^{EOL} graph) are both greater than v, or alternatively both less than v. Case (b) exactly corresponds to the case where a green path "meets" an orange path. In that case, it is easy to see that an ε-KKT point is unavoidable.

The PLS-Labyrinth. In order to resolve the issue with case (b) above, we use the following idea: hide the (unavoidable) ε-KKT point in such a way that locating it requires solving I^{ITER}! This is implemented by introducing a gadget, that we call the PLS-Labyrinth, at the point where the green and orange paths meet (within some

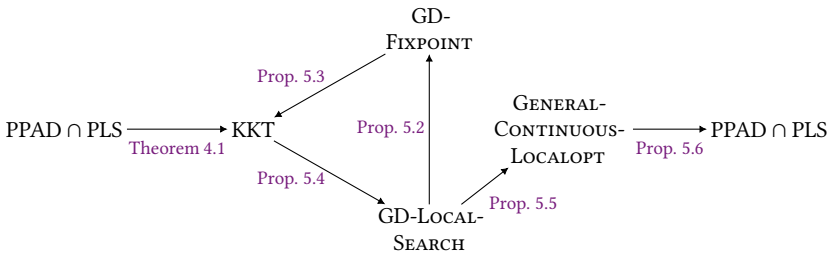

Figure 3: Our reductions. The main one is Theorem 4.1; note that the other reductions are all domain- and promise-preserving.

big square). An important point is that the PLS-Labyrinth only works properly when it is positioned at such a meeting point. If it is positioned elsewhere, then it will either just introduce additional unneeded ε-KKT points, or even introduce ε-KKT points that are easy to locate. Indeed, if we were able to position the PLS-Labyrinth wherever we wanted, this would presumably allow us to show PLS-hardness, which as we noted earlier we do not expect. In Figure 4, the positions where a PLS-Labyrinth is introduced are shown as grey boxes labelled "PLS".

Every PLS-Labyrinth is subdivided into exponentially many *medium squares* such that the medium squares on the diagonal (shaded in Figure 5) correspond to the nodes of I^{ITER}. The point where the green and orange paths meet, which lies just outside the PLS-Labyrinth, creates an "orange-blue path" which then makes its way to the centre of the medium square for node 1 of I^{ITER}. Similarly, for every node u of I^{ITER} that is a candidate to be a solution (i.e., with $C(u) > u$), there is an orange-blue path starting from the orange path (which runs along the PLS-Labyrinth) and going to the centre of the medium square corresponding to u. Sinks of orange-blue paths introduce ε-KKT points, and so for those u that are not solutions of I^{ITER}, the orange-blue path of u turns into a "blue path" that goes and merges into the orange-blue path of $C(u)$. This ensures that sinks of orange-blue paths (that do not turn into blue paths) exactly correspond to the solutions of I^{ITER}. An interesting point to note is that sources of blue paths do *not* introduce ε-KKT points. This allows us to handle crossings between paths in a straightforward manner. Figure 5 shows an overview of the PLS-Labyrinth that encodes the Iter example of Figure 2.

Bicubic interpolation. Within our construction, we specify how the objective function f behaves within the "small squares" of $[0,1]^2$. At this stage, we have values of f and ∇f at the corners of the small squares, and we then need to smoothly interpolate within the interior of the square. We use bicubic interpolation to do this. It constructs a smooth polynomial over the small square given values for f and ∇f at the square's corners.

We must prove that using bicubic interpolation does not introduce any ε-KKT points within any small square, unless that small square corresponds to a solution of I^{ITER} or I^{EOL}. Each individual small square leads to a different class of polynomials, based on the color-coding of the grid point, and the direction of the gradient at each grid point. Our construction uses 101 distinct small squares, and we must prove that no unwanted solutions are introduced in any of them. By making use of various symmetries we are able to group these 101 squares into just four different cases for which we

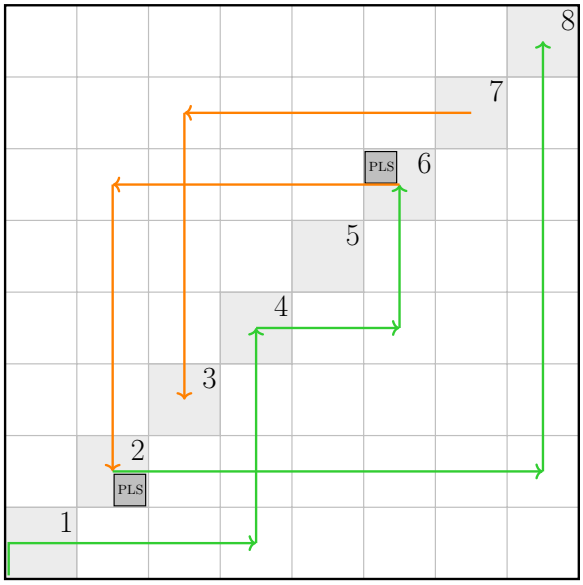

Figure 4: A high-level illustration of our construction. Shaded squares on the diagonal correspond to vertices of the graph represented by I^{EOL}, in this case corresponding to the graph in Figure 1. The green and orange arrows encode the directed edges of the graph. The positions where I^{ITER} is encoded, i.e., the PLS-Labyrinths, are shown as boxes labelled "PLS".

can directly verify that the desired statement holds: an ε-KKT point can only appear in a small square that yields a solution of I^{ITER} or I^{EOL}.

3 PRELIMINARIES

The full preliminaries, including the formal definitions of the computational problems and important notions from nonlinear optimization, can be found in the full version [18]. Here we only briefly discuss the representation of functions by arithmetic circuits.

An arithmetic circuit representing a function $f : \mathbb{R}^n \to \mathbb{R}^m$, is a circuit with n inputs and m outputs, and every internal node is a binary gate performing an operation in $\{+, -, \times, \max, \min, >\}$ or a rational constant (modelled as 0-ary gate). The comparison gate $>$, on input $a, b \in \mathbb{R}$, outputs 1 if $a > b$, and 0 otherwise. We let $\text{size}(f)$ denote the size of the circuit. The definition we use here is

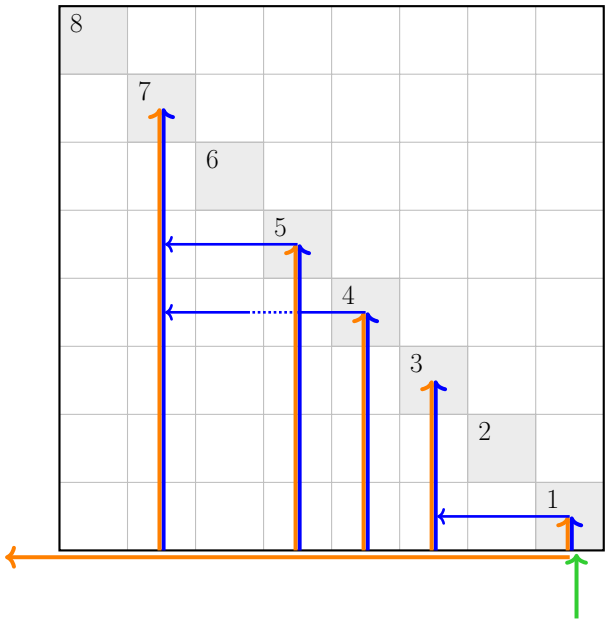

Figure 5: High-level illustration of the PLS-Labyrinth corresponding to the ITER example of Figure 2. Shaded squares on the diagonal correspond to the nodes of ITER.

the same as the one used by Daskalakis and Papadimitriou [13] in their original definition of CLS.

These circuits are very natural, but they suffer from a subtle issue that seems to have been overlooked in prior work. Using the multiplication gate, such an arithmetic circuit can perform repeated squaring to construct numbers that have exponential representation size with respect to the size of the circuit and the input to the circuit. In other words, the circuit can construct numbers that are *doubly* exponential (or the inverse thereof). Thus, in some cases, it might not be possible to evaluate the circuit on some input efficiently, i.e., in time polynomial in the size of the circuit and the given input.

This subtle issue was recently also noticed by Daskalakis and Papadimitriou, who proposed a way to fix it in a corrigendum[4] to the definition of CLS. In this paper, we use an alternative way to resolve the issue. We restrict our attention to what we call *well-behaved* arithmetic circuits. An arithmetic circuit f is well-behaved if, on any directed path that leads to an output, there are at most $\log(\text{size}(f))$ *true* multiplication gates. A true multiplication gate is one where both inputs are non-constant nodes of the circuit. In particular, note that we allow our circuits to perform multiplication by a constant as often as needed without any restriction. Indeed, these operations cannot be used to do repeated squaring. In the full version, we prove that well-behaved circuits can be evaluated in polynomial time.

4 KKT IS PPAD ∩ PLS-HARD

In this section, we prove our main technical result.

[4] http://people.csail.mit.edu/costis/CLS-corrigendum.pdf

THEOREM 4.1. KKT *is* PPAD ∩ PLS*-hard, even when the domain is fixed to be the unit square* $[0,1]^2$*. The hardness continues to hold even if one considers the promise-version of the problem, i.e., only instances without violations.*

In order to show this we provide a polynomial-time many-one reduction from EITHER-SOLUTION(END-OF-LINE,ITER) to KKT on the unit square.

Overview. Consider any instance of END-OF-LINE with 2^n vertices and any instance of ITER with 2^m nodes. We construct a function f for the KKT problem as follows. We first work on the domain $[0,N]^2$ with a grid $G = \{0,1,2,\ldots,N\}^2$, where $N = 2^n \cdot 2^{m+4}$. In the conceptually most interesting part of the reduction, we carefully specify the value of the function f and the direction of $-\nabla f$ (*the direction of steepest descent*) at all the points of the grid G. Then, in the second part of the reduction, we show how to extend f within every square of the grid, so as to obtain a continuously differentiable function on $[0,N]^2$. Finally, we scale down the domain to $[0,1]^2$. We show that any ε-KKT point of f (for some sufficiently small ε) must yield a solution to the END-OF-LINE instance or a solution to the ITER instance.

4.1 Defining the Function on the Grid

Overview of the embedding. We divide the domain $[0,N]^2$ into $2^n \times 2^n$ big squares. For any $v_1, v_2 \in [2^n]$, let $B(v_1,v_2)$ denote the big square

$$\left[(v_1-1)\frac{N}{2^n}, v_1\frac{N}{2^n}\right] \times \left[(v_2-1)\frac{N}{2^n}, v_2\frac{N}{2^n}\right].$$

We use the following interpretation: the vertex $v \in [2^n]$ of the END-OF-LINE instance is embedded at the centre of the big square $B(v,v)$. Thus, the vertices are arranged along the main diagonal of the domain. In particular, the trivial source $1 \in [2^n]$ is located at the centre of the big square that lies in the bottom-left corner of the domain and contains the origin.

We seek to embed the edges of the END-OF-LINE instance in our construction. For every directed edge (v_1,v_2) of the END-OF-LINE instance, we are going to embed a directed path in the grid G that goes from the centre of $B(v_1,v_1)$ to the centre of $B(v_2,v_2)$. The type of path used and the route taken by the path will depend on whether the edge (v_1,v_2) is a "forward" edge or a "backward" edge. In more detail:

- if $v_1 < v_2$ ("forward" edge), then we will use a so-called *green* path that can only travel to the right and upwards. The path starts at the centre of $B(v_1,v_1)$ and moves to the right until it reaches the centre of $B(v_2,v_1)$. Then, it moves upwards until it reaches its destination: the centre of $B(v_2,v_2)$.
- if $v_1 > v_2$ ("backward" edge), then we will use a so-called *orange* path that can only travel to the left and downwards. The path starts at the centre of $B(v_1,v_1)$ and moves to the left until it reaches the centre of $B(v_2,v_1)$. Then, it moves downwards until it reaches its destination: the centre of $B(v_2,v_2)$.

Figure 4 illustrates the high-level idea of the embedding.

For points of the grid G that are part of the "environment", namely that do not lie on a path, the function f will simply be defined by $(x,y) \mapsto x+y$. Thus, if there are no paths at all, the only

local minimum of f will be at the origin. However, a green path starts at the origin and this will ensure that there is no minimum there. This green path will correspond to the outgoing edge of the trivial source $1 \in [2^n]$ of the End-of-Line instance.

The green paths will be constructed such that if one moves along a green path the value of f decreases, which means that we are improving the objective function value. Furthermore, the value of f at any point on a green path will be below the value of f at any point in the environment. Conversely, the orange paths will be constructed such that if one moves along an orange path the value of f increases, so the objective function value becomes worse. Additionally, the value of f at any point on an orange path will be above the value of f at any point in the environment.

As a result, if any path starts or ends in the environment, there will be a local minimum or maximum at that point (and thus a KKT point). The only exception is the path corresponding to the outgoing edge of the trivial vertex $1 \in [2^n]$. The start of that path will not create a local minimum or maximum. Thus, in the example of Figure 4, there will certainly be KKT points in $B(3,3)$, $B(7,7)$ and $B(8,8)$, but not in $B(1,1)$.

Recall that every vertex $v \in [2^n]$ has at most one incoming edge and at most one outgoing edge. Thus, for any vertex $v \neq 1$, one of the following cases occurs:

- v is an isolated vertex. In this case, the big square $B(v,v)$ will not contain any path and will fully be in the environment, thus not containing any KKT point. Example: vertex 5 in Figure 4.
- v has one outgoing edge and no incoming edge. In this case, the big square $B(v,v)$ will contain the start of a green or orange path. There will be a KKT point at the start of the path, which is fine, since v is a (non-trivial) source of the End-of-Line instance. Example: vertex 7 in Figure 4.
- v has one incoming edge and no outgoing edge. In this case, the big square $B(v,v)$ will contain the end of a green or orange path. There will be a KKT point at the end of the path, which is again fine, since v is a sink of the End-of-Line instance. Example: vertices 3 and 8 in Figure 4.
- v has one outgoing and one incoming edge. In this case, there are two sub-cases:
  - If both edges yield paths of the same colour, then we will be able to "connect" the two paths at the centre of $B(v,v)$ and avoid introducing a KKT point there. Example: vertex 4 in Figure 4.
  - If one of the paths is green and the other one is orange, then there will be a local maximum or minimum in $B(v,v)$ (and thus a KKT point). It is not too hard to see that this is in fact unavoidable. This is where we use the main new "trick" of our reduction: we "hide" the exact location of the KKT point inside $B(v,v)$ in such a way, that finding it requires solving a PLS-complete problem, namely the Iter instance. This is achieved by introducing a new gadget at the point where the two paths meet. We call this the PLS-Labyrinth gadget.

The construction of the green and orange paths is described in detail in Section 4.1.2. The PLS-Labyrinth gadget is described in detail in Section 4.1.3.

4.1.1 The Value Regimes. Recall that we want to specify the value of f and $-\nabla f$ (*the direction of steepest descent*) at all points on the grid $G = \{0,1,2,\ldots,N\}^2$, where $N = 2^n \cdot 2^{m+4}$. In order to specify the value of f, it is convenient to define *value regimes*. Namely, if a point $(x,y) \in G$ is in:

- the red value regime, then $f(x,y) := x - y + 4N + 20$.
- the orange value regime, then $f(x,y) := -x - y + 4N + 10$.
- the black value regime, then $f(x,y) := x + y$.
- the green value regime, then $f(x,y) := -x - y - 10$.
- the blue value regime, then $f(x,y) := x - y - 2N - 20$.

Note that at any point on the grid, the value regimes are ordered: red > orange > black > green > blue. Furthermore, it is easy to check that the gap between any two regimes at any point is at least 10. Figure 6 illustrates the main properties of the value regimes.

Figure 6: The value regimes. On the left, the colours are ordered according to increasing value, from left to right. On the right, we indicate for each value regime, the direction in which it improves, i.e., decreases, in the x-y-plane.

The black value regime will be used for the environment. Thus, unless stated otherwise, every grid point is coloured in black, i.e., belongs to the black value regime. Furthermore, unless stated otherwise, at every black grid point (x,y), the direction of steepest descent, i.e., $-\nabla f(x,y)$, will point to the left.[5] The only exceptions to this are grid points that lie in paths, or grid points that lie on the left boundary of the domain (i.e., $x = 0$).

4.1.2 Embedding the End-of-Line Instance: The Green and Orange Paths. Our construction specifies for each grid point a colour (which represents the value of f at that point) and an arrow that represents the direction of $-\nabla f$ at that point. A general "rule" that we follow throughout our construction is that the function values should be consistent with the arrows. For example, if some grid point has an arrow pointing to the right, then the adjacent grid point to the right should have a lower function value, while the adjacent grid point to the left should have a higher function value. This rule is not completely sufficient by itself to avoid KKT points, but it is already a very useful guide.

Recall that the grid $G = \{0,1,2,\ldots,N\}^2$ subdivides every big square $B(v_1,v_2)$ into $2^{m+4} \times 2^{m+4}$ small squares. The width of the paths we construct will be two small squares. This corresponds to a width of three grid points. We refer to the full version of the paper for illustrations of every gadget in the construction.

Green paths. When a green path moves to the right, the two lower grid points will be coloured in green, and the grid point at the top will be in black. When a green path moves upwards, the two right-most grid points will be coloured in green, and the grid point on the left will be in black. Recall that a green path implementing

[5] Notice that that is not exactly the same as the negative gradient of the "black regime function" $(x,y) \mapsto x + y$, which would point south-west. Nevertheless, as we show later, this is enough to ensure that the bicubic interpolation that we use does not introduce any points with zero gradient in a region of the environment.

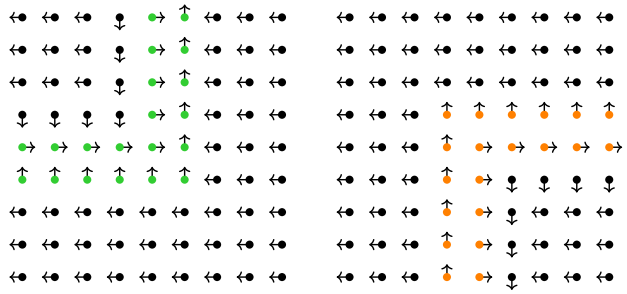

Figure 7: Left: Green path turn. Right: Orange path turn.

an edge (v_1, v_2) (where $v_1 < v_2$) comes into the big square $B(v_2, v_1)$ from the left and leaves at the top. Thus, the path has to "turn". Figure 7 shows how this turn is implemented. The black arrows indicate the direction of $-\nabla f$ at every grid point.

If a vertex $v \in [2^n]$ has one incoming edge (v_1, v) and one outgoing edge (v, v_2) such that $v_1 < v < v_2$, then both edges will be implemented by green paths. The green path corresponding to (v_1, v) will enter $B(v, v)$ from the bottom and stop at the centre of $B(v, v)$. The green path corresponding to (v, v_2) will start at the centre of $B(v, v)$ and leave the big square on the right. In order to avoid introducing any KKT points in $B(v, v)$ (since v is not a solution of the End-of-Line instance), we will connect the two paths at the centre of $B(v, v)$. This will be achieved by a simple turn, similar to the one shown in Figure 7.

If a vertex $v \in [2^n] \setminus \{1\}$ has one outgoing edge (v, v_2) such that $v < v_2$, and no incoming edge, then this will yield a green path starting at the centre of $B(v, v)$ and going to the right. It is not hard to see that there will be a KKT point at the source of that green path. On the other hand, if a vertex $v \in [2^n] \setminus \{1\}$ has one incoming edge (v_1, v) such that $v_1 < v$, and no outgoing edge, then this will yield a green path coming from the bottom and ending at the centre of $B(v, v)$. Again, the sink of that green path will yield a KKT point.

Orange paths. The structure of orange paths is, in a certain sense, symmetric to the structure of green paths. When an orange path moves to the left, the two upper grid points will be coloured in orange, and the grid point at the bottom will be in black. When an orange path moves downwards, the two left-most grid points will be coloured in orange, and the grid point on the right will be in black. The construction of the gadgets for the "turns" is completely analogous to the one for the green paths. See Figure 7. Note that the arrows on an orange path essentially point in the *opposite* direction compared to the direction of the path. This is because we want the value to increase (i.e., worsen) when we follow an orange path.

Crossings. Note that, by construction, green paths only exist below the diagonal, and orange paths only exist above the diagonal. Thus, there is no point where an orange path crosses a green path. However, there might exist points where green paths cross, or orange paths cross. This problem always occurs when one tries to embed an End-of-Line instance in a two-dimensional domain. Chen and Deng [9] proposed a simple, yet ingenious, trick to resolve this issue. The idea is to locally re-route the two paths so that they no longer cross. This modification has the following two crucial properties: a) it is completely local, and b) it does not introduce any new solution (in our case a KKT point).

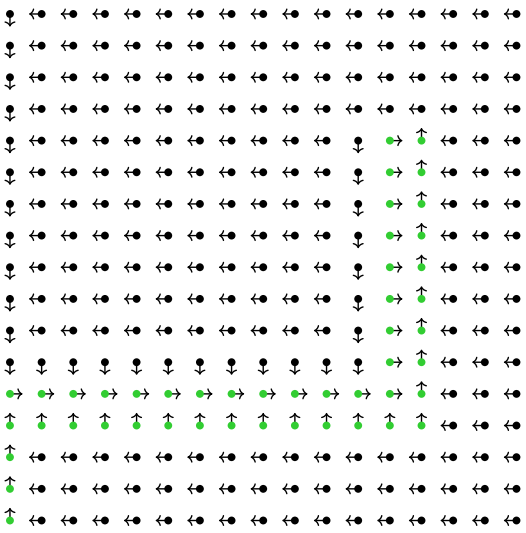

Figure 8: Full construction for a small example, in particular showing the whole boundary and construction for $B(1, 1)$.

Boundary and origin squares. The big square $B(1, 1)$ is special. Since it corresponds to the trivial source of the End-of-Line instance, it has one outgoing edge (which necessarily corresponds to a green path) and no incoming edge. Normally, this would induce a KKT point at the centre of $B(1, 1)$. Furthermore, recall that, by the definition of the black value regime, there must also be a KKT point at the origin, if it is coloured in black. By a careful construction (which is very similar to the one used by Hubáček and Yogev [24] for Continuous-Localopt) we can ensure that these two KKT points neutralise each other. In other words, instead of two KKT points, there is no KKT point at all in $B(1, 1)$. Figure 8 shows a complete instance (including the special construction for $B(1, 1)$) for a small example where $n = 1$ and big squares have size 8×8.

Green and orange paths meeting. Our description of the construction is almost complete, but there is one crucial piece missing. Indeed, consider any vertex v that has one incoming edge (v_1, v) and one outgoing edge (v, v_2) such that: A) $v_1 < v$ and $v_2 < v$, or B) $v_1 > v$ and $v_2 > v$. As it stands, a green path and an orange path meet at the centre of $B(v, v)$ which means that there is a local minimum or maximum at the centre of $B(v, v)$, and thus a KKT point. However, v is not a solution to the End-of-Line instance. Even though we cannot avoid having a KKT point in $B(v, v)$, we can "hide" it, so that finding it requires solving the Iter instance. This is implemented by constructing a PLS-Labyrinth gadget at the point where the green and orange paths meet. Figure 4 shows where this PLS-Labyrinth gadget is positioned inside the big square for both cases A and B. The PLS-Labyrinth gadget can only be positioned at a point where a green path and an orange path meet. In particular, it cannot be used to "hide" a KKT point occurring at a source or sink of a green or orange path, i.e., at a solution of the End-of-Line instance.

4.1.3 *Embedding the* Iter *Instance: The* PLS*-Labyrinth.*

PLS*-Labyrinth.* We begin by describing the PLS-Labyrinth gadget for case A, i.e., v has one incoming edge (v_1, v) and one outgoing edge (v, v_2) such that $v_1 < v$ and $v_2 < v$. The PLS-Labyrinth gadget comprises $2^{m+2} \times 2^{m+2}$ small squares and is positioned in the big square $B(v, v)$ as shown in Figure 4 for vertex 6. Note, in particular, that the bottom side of the gadget is adjacent to the orange path, and the bottom-right corner of the gadget lies just above the point where the green and orange paths intersect (which occurs at the centre of $B(v, v)$). Finally, observe that since $B(v, v)$ has $2^{m+4} \times 2^{m+4}$ small squares, there is enough space for the PLS-Labyrinth gadget.

For convenience, we subdivide the PLS-Labyrinth gadget into $2^m \times 2^m$ medium squares. Thus, every medium square is made out of 4×4 small squares. We index the medium squares as follows: for $u_1, u_2 \in [2^m]$, let $M(u_1, u_2)$ denote the medium square that is the u_2th from the bottom and the u_1th from the *right*. Thus, $M(1, 1)$ corresponds to the medium square that lies at the bottom-right of the gadget (and is just above the intersection of the paths). Our construction will create the following paths inside the PLS-Labyrinth gadget:

- For every $u \in [2^m]$ such that $C(u) > u$, there is an orange-blue path starting at $M(u, 1)$ and moving upwards until it reaches $M(u, u)$.
- For every $u \in [2^m]$ such that $C(u) > u$ and $C(C(u)) > C(u)$, there is a blue path starting at $M(u, u)$ and moving to the left until it reaches $M(C(u), u)$.

Figure 5 shows a high-level overview of how the Iter instance is embedded in the PLS-Labyrinth. Note that if $C(u) > u$ and $C(C(u)) > C(u)$, then the blue path starting at $M(u, u)$ will move to the left until $M(C(u), u)$ where it will reach the orange-blue path moving up from $M(C(u), 1)$ to $M(C(u), C(u))$ (which exists since $C(C(u)) > C(u)$). Thus, every blue path will always "merge" into some orange-blue path. On the other hand, some orange-blue paths will stop in the environment without merging into any other path. Consider any $u \in [2^m]$ such that $C(u) > u$. The orange-blue path for u stops at $M(u, u)$. If $C(C(u)) > C(u)$, then there is a blue path starting there, so the orange-blue path "merges" into the blue path. However, if $C(C(u)) \leq C(u)$, i.e., $C(C(u)) = C(u)$, there is no blue path starting at $M(u, u)$ and the orange-blue path just stops in the environment. Thus, the only place in the PLS-Labyrinth where a path can stop in the environment is in a medium square $M(u, u)$ such that $C(u) > u$ and $C(C(u)) = C(u)$. This corresponds exactly to the solutions of the Iter instance C. In our construction, we will ensure that KKT points can indeed only occur at points where a path stops without merging into any other path.

Orange-blue paths. An orange-blue path moves from $M(u, 1)$ upwards to $M(u, u)$ (for some $u \in [2^m]$ such that $C(u) > u$) and has a width of two small squares. The left-most point is coloured in orange and the two points on the right are blue. Figure 9 shows a medium square that is being traversed by the orange-blue path, i.e., a medium square $M(u, w)$ where $w < u$. When the orange-blue path reaches $M(u, u)$, it either "turns" to the left and creates the beginning of a blue path, or it just stops there. The case where the orange-blue path just stops, occurs when there is no blue path starting at $M(u, u)$. Note that, in that case, u is a solution of the Iter instance, and so KKT points are acceptable in $M(u, u)$.

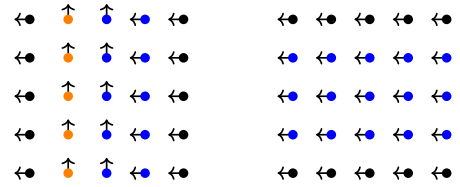

Figure 9: Left: An orange-blue path. Right: A blue path.

The orange-blue path begins in $M(u, 1)$, which lies just above the orange path. In fact, the beginning of the orange-blue path is adjacent to the orange path. This is needed, since if the orange-blue path started in the environment, the point coloured orange would yield a local maximum and thus a KKT point. The beginning of the orange-blue path for $u = 1$ is special, since, in a certain sense, this path is created by the intersection of the green and orange paths.

Blue paths. A blue path starts in $M(u, u)$ for some $u \in [2^m]$ such that $C(u) > u$ and $C(C(u)) > C(u)$. It moves from right to left and has a width of two small squares. All three points on the path are coloured blue and the direction of steepest descent points to the left. Figure 9 shows a medium square traversed by a blue path. As mentioned above, the blue path starts at $M(u, u)$, which corresponds to a "turn". When the blue path reaches $M(C(u), u)$, it merges into the orange-blue path going from $M(C(u), 1)$ to $M(C(u), C(u))$. This merging is implemented in a straightforward way. Crossings between blue and orange-blue paths can be easily handled here, because blue paths can *start* in the environment.

The PLS-Labyrinth gadget for case B is essentially symmetric to the one presented above. It can be obtained by rotating the gadget by 180 degrees and applying the following simple transformation to the colours: swap green and orange, and replace blue by red.

4.2 Extending the Function to the Rest of the Domain

Up to this point we have defined the function f and the direction of its gradient at all grid points of G. In order to extend f to the whole domain $[0, N]^2$, we use *bicubic interpolation* (see e.g. [36] or the corresponding Wikipedia article[6]). Note that the more standard and simpler *bilinear* interpolation (used in particular by Hubáček and Yogev [24]) yields a continuous function, but not necessarily a continuously differentiable function. On the other hand, bicubic interpolation ensures that the function will indeed be continuously differentiable over the whole domain $[0, N]^2$.

We use bicubic interpolation in every small square of the grid G. Consider any small square and let $(x, y) \in [0, 1]^2$ denote the local coordinates of a point inside the square. Then, the bicubic interpolation inside this square will be a polynomial of the form:

$$f(x, y) = \sum_{i=0}^{3} \sum_{j=0}^{3} a_{ij} x^i y^j$$

[6] https://en.wikipedia.org/wiki/Bicubic_interpolation

where the coefficients a_{ij} are computed as follows

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -3 & 3 & -2 & -1 \\ 2 & -2 & 1 & 1 \end{bmatrix} \cdot$$
$$\begin{bmatrix} f(0,0) & f(0,1) & f_y(0,0) & f_y(0,1) \\ f(1,0) & f(1,1) & f_y(1,0) & f_y(1,1) \\ f_x(0,0) & f_x(0,1) & f_{xy}(0,0) & f_{xy}(0,1) \\ f_x(1,0) & f_x(1,1) & f_{xy}(1,0) & f_{xy}(1,1) \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -3 & 2 \\ 0 & 0 & 3 & -2 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & -1 & 1 \end{bmatrix}$$

Here f_x and f_y denote the partial derivatives with respect to x and y respectively. Similarly, f_{xy} denotes the second order partial derivative with respect to x and y. It remains to explain how we set the values of f, f_x, f_y and f_{xy} at the four corners of the square:

- The values $f(0,0)$, $f(0,1)$, $f(1,0)$ and $f(1,1)$ are set according to the value regimes in our construction.
- The values of $f_x(0,0)$, $f_x(0,1)$, $f_x(1,0)$, $f_x(1,1)$, $f_y(0,0)$, $f_y(0,1)$, $f_y(1,0)$ and $f_y(1,1)$ are set based on the direction of steepest descent $(-\nabla f)$ in our construction, with a length multiplier of $\delta = 1/2$. For example, if the arrow of steepest descent at $(0,1)$ is pointing to the left, then we set $f_x(0,1) = \delta$ and $f_y(0,1) = 0$. If it is pointing up, then we set $f_x(0,1) = 0$ and $f_y(0,1) = -\delta$.
- We always set $f_{xy}(0,0) = f_{xy}(0,1) = f_{xy}(1,0) = f_{xy}(1,1) = 0$.

By using this interpolation procedure in each small square, we obtain a function $f : [0,N]^2 \to \mathbb{R}$. We then have the following Lemma, which is proved in the full version.

Lemma 4.2. *The function $f : [0,N]^2 \to \mathbb{R}$ we obtain by bicubic interpolation has the following properties:*

- *It is continuously differentiable;*
- *f and its gradient ∇f are Lipschitz-continuous on $[0,N]^2$ with Lipschitz-constant $L = 2^{18}N$;*
- *Well-behaved arithmetic circuits computing f and ∇f can be constructed in polynomial time.*

4.3 Correctness

To show the correctness of the construction, we prove the following lemma, which states that 0.01-KKT points of f only lie at solutions for the End-of-Line instance or the Iter instance.

Lemma 4.3. *Let $\varepsilon = 0.01$. We have that (x,y) is an ε-KKT point of f on the domain $[0,N]^2$ only if (x,y) lies in a "solution region", namely:*

- *(x,y) lies in a big square $B(v,v)$, such that $v \in [2^n] \setminus \{1\}$ is a source or sink of the* End-of-Line *instance, or*
- *(x,y) lies in a medium square $M(u,u)$ of some* PLS*-Labyrinth gadget, such that $u \in [2^m]$ is a solution to the* Iter *instance C, i.e., $C(u) > u$ and $C(C(u)) = C(u)$.*

Proof Sketch. The behaviour of the function in a given small square depends on the information we have about the four corners, namely the colours and arrows at the four corners, but also on the position of the square in our instance, since the value defined by a colour depends on the position. For our proof, it is convenient to consider a square with the (colour and arrow) information about its four corners, but without any information about its position. Indeed, if we can show that a square does not contain any ε-KKT point using only this information, then this will always hold, wherever the square is positioned. As a result, we obtain a finite number of squares (with colour and arrow information) that we need to check.

However, this still leaves us with 101 distinct small squares to verify. Fortunately, using various symmetries, these squares can be grouped into just four distinct cases that we can check individually. The details can be found in the full version. □

4.4 Re-scaling

The last step of the reduction is to re-scale the function f so that it is defined on $[0,1]^2$ instead of $[0,N]^2$. As a result, the final function and its gradient are $\widehat{L}$-Lipschitz-continuous with $\widehat{L} = N \cdot L = 2^{18}N^2 = 2^{2n+2m+26}$. Note that we can re-scale the instance depending on the parameter regime we are interested in. For example, we immediately obtain hard instances with Lipschitz-constant 1, and with inversely exponential precision parameter.

5 GRADIENT DESCENT AND KKT ARE PPAD ∩ PLS-COMPLETE

In this section, we explain how the PPAD ∩ PLS-hardness of KKT (Theorem 4.1) implies that our other problems of interest, including our Gradient Descent problems, are PPAD ∩ PLS-complete.

Theorem 5.1. *The problems* GD-Local-Search, GD-Fixpoint, KKT *and* General-Continuous-Localopt *are* PPAD ∩ PLS*-complete, even when the domain is fixed to be the unit square $[0,1]^2$. This hardness result continues to hold even if one considers the promise-versions of these problems, i.e., only instances without violations.*

The hardness results in this theorem are the "best possible", in the following sense:

- Promise-problem: as mentioned in the theorem, the hardness holds even for the promise-versions of these problems. In other words, the hard instances that we construct are not pathological: they satisfy all the conditions that we would expect from the input, e.g., ∇f is indeed the gradient of f, ∇f and f are indeed L-Lipschitz-continuous, etc.
- Domain: the problems remain hard even if we fix the domain to be the unit square $[0,1]^2$, which is arguably the simplest two-dimensional bounded domain. All the problems become polynomial-time solvable if the domain is one-dimensional.
- Exponential parameters: in all of our problems, the parameters, such as ε and L, are provided in the input in *binary* representation. This means that the parameters are allowed to be exponentially small or large with respect to the length of the input. Our hardness results make use of this, since the proof of Theorem 4.1 constructs an instance of KKT where ε is some constant, but L is exponential in the input length. By a simple transformation (essentially, a re-scaling), this instance can be transformed into one where ε is exponentially small and L is constant. It is easy to see that at least one of ε or L must be exponentially large/small, for the problem to be hard on the domain $[0,1]^2$. However, this continues to hold even in high dimension, i.e., when the domain is $[0,1]^n$. In other words, if the parameters are given in unary, the problem is easy, even in high dimension. This is in contrast with the problem of finding a Brouwer fixed point, where moving to

domain $[0, 1]^n$ makes it possible to prove PPAD-hardness even when the parameters are given in unary.

Theorem 5.1 follows from Theorem 4.1 and a set of domain- and promise-preserving reductions as pictured in Figure 3, which are stated below and proved in the full version. The first three Propositions establish that the KKT problem and the two versions of the Gradient Descent problem are equivalent. The two remaining Propositions prove the membership of all our problems in PPAD ∩ PLS.

Proposition 5.2. GD-Local-Search *reduces to* GD-Fixpoint *using a domain- and promise-preserving reduction.*

Proposition 5.3. GD-Fixpoint *reduces to* KKT *using a domain- and promise-preserving reduction.*

Proposition 5.4. KKT *reduces to* GD-Local-Search *using a domain- and promise-preserving reduction.*

Proposition 5.5. GD-Local-Search *reduces to* General-Continuous-Localopt *using a domain- and promise-preserving reduction.*

Proposition 5.6. General-Continuous-Localopt *lies in* PPAD ∩ PLS.

6 CONSEQUENCES FOR CONTINUOUS LOCAL SEARCH

In this section, we explore the consequences of Theorem 4.1 for the class CLS. We also consider a Gradient Descent problem where "finite differences" are used to compute an approximate gradient.

6.1 Consequences for CLS

The class CLS was defined by Daskalakis and Papadimitriou [13] as a more natural counterpart to PPAD ∩ PLS. Indeed, they noted that all known PPAD ∩ PLS-complete problems were unnatural, namely uninteresting combinations of a PPAD-complete and a PLS-complete problem. As a result, they defined CLS, a subclass of PPAD ∩ PLS, as a more natural combination of PPAD and PLS, and conjectured that CLS is a *strict* subclass of PPAD ∩ PLS. They were able to prove that various interesting problems lie in CLS, thus further strengthening the conjecture that CLS is a more natural subclass of PPAD ∩ PLS, and more likely to capture the complexity of interesting problems.

It follows from our results that, surprisingly:

Theorem 6.1. CLS = PPAD ∩ PLS.

Recall that by Theorem 5.1, General-Continuous-Localopt with domain $[0, 1]^2$ is PPAD ∩ PLS-complete. Theorem 6.1 follows from the fact that this problem lies in CLS, almost by definition. This is formally proved in the full version. We now explore some further consequences of our results for CLS.

An immediate consequence is that the two previously known CLS-complete problems [15, 19] are in fact PPAD ∩ PLS-complete.

Theorem 6.2. Banach *and* MetametricContraction *are* PPAD ∩ PLS*-complete.*

Furthermore, our results imply that the definition of CLS is "robust" in the following sense:

- Dimension: the class CLS was defined by Daskalakis and Papadimitriou [13] as the set of all TFNP problems that reduce to 3D-Continuous-Localopt, i.e., Continuous-Localopt with $n = 3$. Even though it is easy to see that kD-Continuous-Localopt reduces to $(k + 1)$D-Continuous-Localopt, it is unclear how to construct a reduction in the other direction. Indeed, similar reductions exist for the Brouwer problem, but they require using a discrete equivalent of Brouwer, namely End-of-Line, as an intermediate step. Since no such discrete problem was known for CLS, this left open the possibility of a hierarchy of versions of CLS, depending on the dimension, i.e., 2D-CLS ⊂ 3D-CLS ⊂ 4D-CLS ... . We show that even the two-dimensional version is PPAD ∩ PLS-hard, and thus the definition of CLS is indeed independent of the dimension used. In other words,

$$\text{2D-CLS} = \text{CLS} = n\text{D-CLS}.$$

  Note that this is tight, since 1D-Continuous-Localopt can be solved in polynomial time, i.e., 1D-CLS = FP.
- Domain: some interesting problems can be shown to lie in CLS, but the reduction produces a polytopal domain, instead of the standard domain $[0, 1]^n$. In other words, they reduce to General-Continuous-Localopt, which we have defined as a generalization of Continuous-Localopt. Since General-Continuous-Localopt is PPAD ∩ PLS-complete (Theorem 5.1), it follows that CLS can equivalently be defined as the set of all TFNP problems that reduce to General-Continuous-Localopt.
- Promise: the problem Continuous-Localopt, which defines CLS, is a problem with violation solutions. One can instead consider promise-CLS, which is defined as the set of all TFNP problems that reduce to a promise version of Continuous-Localopt. In the promise version of Continuous-Localopt, we restrict our attention to instances that satisfy the promise, i.e., where the functions p and g are indeed L-Lipschitz-continuous. The class promise-CLS could possibly be weaker than CLS, since the reduction is required to always map to instances of Continuous-Localopt without violations. However, it follows from our results that promise-CLS = CLS, since the promise version of the Continuous-Localopt problem is shown to be PPAD ∩ PLS-hard, even on domain $[0, 1]^2$ (Theorem 5.1).
- Turing reductions: since PPAD and PLS are closed under Turing reductions [5], it is easy to see that this also holds for PPAD ∩ PLS, and thus by our result also for CLS.
- Circuits: CLS is defined using the Continuous-Localopt problem where the functions are represented by general arithmetic circuits. If one restricts the type of arithmetic circuit that is used, this might yield a weaker version of CLS. Linear arithmetic circuits are a natural class of circuits that arise when reducing from various natural problems. We define Linear-CLS as the set of problems that reduce to Continuous-Localopt with linear circuits. In Section 6.2 we show that Linear-CLS = CLS.

6.2 Linear-CLS and Gradient Descent with Finite Differences

The class CLS was defined by Daskalakis and Papadimitriou [13] using the Continuous-Localopt problem which uses arithmetic circuits with gates in $\{+, -, \min, \max, \times, <\}$ and rational constants.

In this section we show that even if we restrict ourselves to linear arithmetic circuits (i.e., only the gates in $\{+, -, \min, \max, \times\zeta\}$ and rational constants are allowed), the Continuous-Localopt problem and CLS remain just as hard as the original versions.

Definition 6.3. Linear-Continuous-Localopt:
Input:
- precision/stopping parameter $\varepsilon > 0$,
- linear arithmetic circuits $p : [0,1]^n \to [0,1]$ and $g : [0,1]^n \to [0,1]^n$.

Goal: Compute an approximate local optimum of p with respect to g. Formally, find $x \in [0,1]^n$ such that

$$p(g(x)) \geq p(x) - \varepsilon.$$

We define the class 2D-Linear-CLS as the set of all TFNP problems that reduce to 2D-Linear-Continuous-Localopt. We show that:

Theorem 6.4. 2D-Linear-CLS = PPAD ∩ PLS.

This theorem mainly relies on a general result which says that any arithmetic circuit can be arbitrarily well approximated by a linear arithmetic circuit on a bounded domain. This approximation theorem is proved (in a more general form) in the appendix of the full version. The proof uses known techniques developed in the study of the complexity of Nash equilibria [10, 12], but replaces the usual averaging step by a median step, which ensures that we obtain the desired accuracy of approximation.

Instead of reducing 2D-Continuous-Localopt to 2D-Linear-Continuous-Localopt, we prove Theorem 6.4 by a different route that also allows us to introduce a problem which might be of independent interest. To capture the cases where the gradient is not available or perhaps too expensive to compute, we consider a version of Gradient Descent where the *finite differences* approach is used to compute an approximate gradient, which is then used as usual to obtain the next iterate. Formally, given a finite difference spacing parameter $h > 0$, the approximate gradient $\widetilde{\nabla}_h f(x)$ at some point $x \in [0,1]^n$ is computed as

$$\left[\widetilde{\nabla}_h f(x)\right]_i = \frac{f(x + h \cdot e_i) - f(x - h \cdot e_i)}{2h}$$

for all $i \in [n]$. The computational problem is defined as follows.

GD-Finite-Diff *(informal)*
Input:
- precision/stopping parameter $\varepsilon > 0$,
- step size $\eta > 0$,
- finite difference spacing parameter $h > 0$,
- linear arithmetic circuit $f : \mathbb{R}^n \to \mathbb{R}$.

Goal: Compute any point where (projected) gradient descent for f on domain $D = [0,1]^n$ using finite differences to approximate the gradient and fixed step size η terminates.

GD-Finite-Diff reduces to Linear-Continuous-Localopt by setting $p := f$ and $g := \Pi_D(x - \eta\widetilde{\nabla}_h f(x))$. Furthermore, this reduction is domain-preserving and thus Theorem 6.4 immediately follows from the following theorem, which is proved in the full version.

Theorem 6.5. GD-Finite-Diff *is* PPAD ∩ PLS*-complete, even with fixed domain* $[0,1]^2$.

This result is interesting by itself, because the problem is arguably quite natural, but also because it is the first problem that is complete for PPAD ∩ PLS (and CLS) that has a *single* arithmetic circuit in the input. Note that our other problems which we prove to be PPAD ∩ PLS-complete, as well as the previously known CLS-complete problems, all have two arithmetic circuits in the input.

7 FUTURE DIRECTIONS

Our results may help to identify the complexity of the following problems that are known to lie in PPAD ∩ PLS:

- Mixed-Congestion: The problem of finding a *mixed* Nash equilibrium of a congestion game. It is known that finding a *pure* Nash equilibrium is PLS-complete [17]. As mentioned in Section 1.2, Babichenko and Rubinstein [3] have recently applied our main result to obtain PPAD ∩ PLS-completeness for Mixed-Congestion. It would be interesting to extend this to *network* congestion games, where the strategies are represented implicitly.
- polynomial-KKT: The special case of the KKT problem where the function is a polynomial, provided explicitly in the input (exponents in unary). A consequence of the above-mentioned reduction by Babichenko and Rubinstein [3] is that the problem is PPAD ∩ PLS-complete for polynomials of degree 5. It remains open to extend this hardness result to lower degree polynomials.
- Contraction: Find a fixed point of a function that is contracting with respect to some ℓ_p-norm.
- Tarski: Find a fixed point of an order-preserving function, as guaranteed by Tarski's theorem [11, 16, 22].
- ColorfulCarathéodory: A problem based on a theorem in convex geometry [30].

The first three problems on this list were known to lie in CLS [13], while the other two were only known to lie in PPAD ∩ PLS.

The collapse between CLS and PPAD ∩ PLS raises the question of whether the class EOPL (for End of Potential Line), a subclass of CLS, is also equal to PPAD ∩ PLS. The class EOPL, or more precisely its subclass UEOPL (with U for unique), is known to contain various problems of interest that have unique solutions such as Unique Sink Orientation (USO), the P-matrix Linear Complementarity Problem (P-LCP), Simple Stochastic Games (SSG) and Parity Games [20]. We conjecture that EOPL ≠ PPAD ∩ PLS. Unlike CLS, EOPL has a more standard combinatorial definition that is simultaneously a special case of End-of-Line and Localopt. While PPAD ∩ PLS captures problems that have a PPAD-type proof of existence and a PLS-type proof of existence, EOPL seems to capture problems that have a single proof of existence which is simultaneously of PPAD- and PLS-type. The first step towards confirming this conjecture would be to provide an oracle separation between EOPL and PPAD ∩ PLS, in the sense of Beame et al. [4].

Ishizuka [25] has recently shown that the definition of EOPL is robust with respect to some modifications (similarly to PPAD [23]), and has provided a somewhat artificial problem that is complete for PPA ∩ PLS. This raises the interesting question of whether PPA ∩ PLS, and other intersections of well-studied classes, also

admit natural complete problems, or if PPAD ∩ PLS is in fact an isolated case.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for suggestions that helped improve the presentation of the paper. Alexandros Hollender was supported by an EPSRC doctoral studentship (Reference 1892947).

REFERENCES

[1] Amir Ali Ahmadi and Jeffrey Zhang. 2020. Complexity aspects of local minima and related notions. arXiv:2008.06148

[2] Amir Ali Ahmadi and Jeffrey Zhang. 2020. On the complexity of finding a local minimizer of a quadratic function over a polytope. arXiv:2008.05558

[3] Yakov Babichenko and Aviad Rubinstein. 2021. Settling the complexity of Nash equilibrium in congestion games. In *Proceedings of the 53rd ACM Symposium on Theory of Computing (STOC)*. arXiv:2012.04327

[4] Paul Beame, Stephen Cook, Jeff Edmonds, Russell Impagliazzo, and Toniann Pitassi. 1998. The Relative Complexity of NP Search Problems. *J. Comput. System Sci.* 57, 1 (1998), 3–19. https://doi.org/10.1145/225058.225147

[5] Samuel R. Buss and Alan S. Johnson. 2012. Propositional proofs and reductions between NP search problems. *Annals of Pure and Applied Logic* 163, 9 (2012), 1163–1182. https://doi.org/10.1016/j.apal.2012.01.015

[6] Augustin-Louis Cauchy. 1847. Méthode générale pour la résolution des systèmes d'équations simultanées. *C. R. Acad. Sci. Paris* 25 (1847), 536–538.

[7] Vaggos Chatziafratis, Tim Roughgarden, and Joshua R. Wang. 2019. On the Computational Power of Online Gradient Descent. In *Proceedings of the 32nd Conference on Learning Theory (COLT)*. 624–662. http://proceedings.mlr.press/v99/chatziafratis19a.html

[8] Xi Chen, Decheng Dai, Ye Du, and Shang-Hua Teng. 2009. Settling the Complexity of Arrow-Debreu Equilibria in Markets with Additively Separable Utilities. In *Proceedings of the 50th IEEE Symposium on Foundations of Computer Science (FOCS)*. 273–282. https://doi.org/10.1109/FOCS.2009.29

[9] Xi Chen and Xiaotie Deng. 2009. On the complexity of 2D discrete fixed point problem. *Theoretical Computer Science* 410, 44 (2009), 4448 – 4456. https://doi.org/10.1016/j.tcs.2009.07.052

[10] Xi Chen, Xiaotie Deng, and Shang-Hua Teng. 2009. Settling the complexity of computing two-player Nash equilibria. *J. ACM* 56, 3 (2009), 14:1–14:57. https://doi.org/10.1145/1516512.1516516

[11] Chuangyin Dang, Qi Qi, and Yinyu Ye. 2020. Computations and Complexities of Tarski's Fixed Points and Supermodular Games. arXiv:2005.09836

[12] Constantinos Daskalakis, Paul W. Goldberg, and Christos H. Papadimitriou. 2009. The complexity of computing a Nash equilibrium. *SIAM J. Comput.* 39, 1 (2009), 195–259. https://doi.org/10.1137/070699652

[13] Constantinos Daskalakis and Christos Papadimitriou. 2011. Continuous local search. In *Proceedings of the 22nd ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 790–804. https://doi.org/10.1137/1.9781611973082.62

[14] Constantinos Daskalakis, Stratis Skoulakis, and Manolis Zampetakis. 2021. The Complexity of Constrained Min-Max Optimization. In *Proceedings of the 53rd ACM Symposium on Theory of Computing (STOC)*. arXiv:2009.09623

[15] Constantinos Daskalakis, Christos Tzamos, and Manolis Zampetakis. 2018. A converse to Banach's fixed point theorem and its CLS-completeness. In *Proceedings of the 50th ACM Symposium on Theory of Computing (STOC)*. 44–50. https://doi.org/10.1145/3188745.3188968

[16] Kousha Etessami, Christos Papadimitriou, Aviad Rubinstein, and Mihalis Yannakakis. 2020. TarskiâĂŹs Theorem, Supermodular Games, and the Complexity of Equilibria. In *Proceedings of the 11th Innovations in Theoretical Computer Science Conference (ITCS)*. 18:1–18:19. https://doi.org/10.4230/LIPIcs.ITCS.2020.18

[17] Alex Fabrikant, Christos Papadimitriou, and Kunal Talwar. 2004. The complexity of pure Nash equilibria. In *Proceedings of the 36th ACM Symposium on Theory of Computing (STOC)*. 604–612. https://doi.org/10.1145/1007352.1007445

[18] John Fearnley, Paul W. Goldberg, Alexandros Hollender, and Rahul Savani. 2020. The Complexity of Gradient Descent: CLS = PPAD ∩ PLS. arXiv:2011.01929

[19] John Fearnley, Spencer Gordon, Ruta Mehta, and Rahul Savani. 2017. CLS: New Problems and Completeness. arXiv:1702.06017

[20] John Fearnley, Spencer Gordon, Ruta Mehta, and Rahul Savani. 2020. Unique End of Potential Line. *J. Comput. System Sci.* 114 (2020), 1–35. https://doi.org/10.1016/j.jcss.2020.05.007

[21] John Fearnley and Rahul Savani. 2015. The Complexity of the Simplex Method. In *Proceedings of the 47th ACM Symposium on Theory of Computing (STOC)*. 201–208. https://doi.org/10.1145/2746539.2746558

[22] John Fearnley and Rahul Savani. 2021. A faster algorithm for finding Tarski fixed points. In *38th International Symposium on Theoretical Aspects of Computer Science (STACS)*. 29:1–29:16. https://doi.org/10.4230/LIPIcs.STACS.2021.29

[23] Paul W. Goldberg and Alexandros Hollender. 2019. The Hairy Ball Problem is PPAD-Complete. In *Proceedings of the 46th International Colloquium on Automata, Languages, and Programming (ICALP)*. 65:1–65:14. https://doi.org/10.4230/LIPIcs.ICALP.2019.65

[24] Pavel Hubáček and Eylon Yogev. 2017. Hardness of continuous local search: Query complexity and cryptographic lower bounds. In *Proceedings of the 28th ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 1352–1371. https://doi.org/10.1137/1.9781611974782.88

[25] Takashi Ishizuka. 2021. The complexity of the parity argument with potential. *J. Comput. System Sci.* 120 (2021), 14–41. https://doi.org/10.1016/j.jcss.2021.03.004

[26] Chi Jin, Praneeth Netrapalli, Rong Ge, Sham M. Kakade, and Michael I. Jordan. 2021. On Nonconvex Optimization for Machine Learning: Gradients, Stochasticity, and Saddle Points. *J. ACM* 68, 2 (2021), 11:1–11:29. https://doi.org/10.1145/3418526

[27] David S. Johnson, Christos H. Papadimitriou, and Mihalis Yannakakis. 1988. How easy is local search? *J. Comput. System Sci.* 37, 1 (1988), 79–100. https://doi.org/10.1016/0022-0000(88)90046-3

[28] Shiva Kintali, Laura J. Poplawski, Rajmohan Rajaraman, Ravi Sundaram, and Shang-Hua Teng. 2013. Reducibility among Fractional Stability Problems. *SIAM J. Comput.* 42, 6 (2013), 2063–2113. https://doi.org/10.1137/120874655

[29] Nimrod Megiddo and Christos H. Papadimitriou. 1991. On total functions, existence theorems and computational complexity. *Theoretical Computer Science* 81, 2 (1991), 317–324. https://doi.org/10.1016/0304-3975(91)90200-L

[30] Frédéric Meunier, Wolfgang Mulzer, Pauline Sarrabezolles, and Yannik Stein. 2017. The rainbow at the end of the lineâĂŤa PPAD formulation of the colorful Carathéodory theorem with applications. In *Proceedings of the 28th ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 1342–1351. https://doi.org/10.1137/1.9781611974782.87

[31] Tsuyoshi Morioka. 2001. *Classification of search problems and their definability in bounded arithmetic.* Master's thesis. University of Toronto. https://www.collectionscanada.ca/obj/s4/f2/dsk3/ftp04/MQ58775.pdf

[32] Katta G. Murty and Santosh N. Kabadi. 1987. Some NP-complete problems in quadratic and nonlinear programming. *Mathematical Programming* 39, 2 (1987), 117–129. https://doi.org/10.1007/BF02592948

[33] Christos H. Papadimitriou. 1992. The complexity of the Lin-Kernighan heuristic for the traveling salesman problem. *SIAM J. Comput.* 21, 3 (1992), 450–465. https://doi.org/10.1137/0221030

[34] Christos H. Papadimitriou. 1994. On the complexity of the parity argument and other inefficient proofs of existence. *J. Comput. System Sci.* 48, 3 (1994), 498–532. https://doi.org/10.1016/S0022-0000(05)80063-7

[35] Herbert Robbins and Sutton Monro. 1951. A stochastic approximation method. *Annals of Mathematical Statistics* (1951), 400–407.

[36] William S. Russell. 1995. Polynomial interpolation schemes for internal derivative distributions on structured grids. *Applied Numerical Mathematics* 17, 2 (1995), 129 – 171. https://doi.org/10.1016/0168-9274(95)00014-L