The Complexity of Gradient Descent: CLS = PPAD ∩ PLS

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

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

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

Rahul Savani
University of Liverpool, United Kingdom
rahul.savani@liverpool.ac.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.

1 Introduction

It is hard to overstate the importance of Gradient Descent. As noted by Jin et al. [2019], "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 [Cauchy, 1847] and stochastic gradient descent, which dates to the 1950s [Robbins and Monro, 1951]—reign supreme in machine learning." Jin et al. [2019] 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 optimisation.

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 [Murty and Kabadi, 1987; Ahmadi and Zhang, 2020a]. Indeed, even deciding whether a point is a local minimum is already co-NP-hard [Murty and Kabadi, 1987]. 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 [Ahmadi and Zhang, 2020b]. 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 [Megiddo and Papadimitriou, 1991]. 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, and CLS

As discussed in Papadimitriou [1994], 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 in Papadimitriou [1994] and Johnson et al. [1988] 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. [2009]; Chen et al. [2009b] and market equilibria, e.g., Chen et al. [2009a]). PPAD also captures diverse problems in combinatorics and cooperative game theory [Kintali et al., 2013].

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 optimisation 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 [Papadimitriou, 1992], and finding pure Nash equilibria in multiplayer congestion games [Fabrikant et al., 2004].

The complexity class CLS ("Continuous Local Search") was introduced by Daskalakis and Papadimitriou [2011] to classify various important problems that lie both in 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 [Beame et al., 1998]. 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 [Daskalakis et al., 2018; Fearnley et al., 2017]. 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 [Hubáček and Yogev, 2017]. Prior to the present paper, it was generally believed that CLS is a proper subset of PPAD ∩ PLS, as conjectured by Daskalakis and Papadimitriou [2011].

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.

[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.

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 [2011] 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. [2020].

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 [2020], to show PPAD ∩ PLS-completeness of computing mixed Nash equilibria of congestion games under randomized reductions.

1.3 Further Related Work

Following the definition of CLS by Daskalakis and Papadimitriou [2011], two CLS-complete problems were identified: Banach [Daskalakis et al., 2018] and MetametricContraction [Fearnley et al., 2017]. 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.

Daskalakis et al. [2020] study nonlinear *min-max* optimisation, a conceptually more complex problem than the purely "min" optimisation 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. [2020] 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 (Lemma C.4). Related work in nonlinear optimisation is covered in Section 3.2.1.

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, and interesting open problems.

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. For a formal definition of these notions see Section 3.2.1.

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 [2011], we consider arithmetic circuits that use the operations $\{+, -, \times, \max, \min, <\}$, as well as rational constants.[3] Another option would be to assume that the functions are given as polynomial-time Turing machines, but this introduces some extra clutter in the formal definitions of the problems. Overall, 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 [2011], i.e. allow as a solution some points that witness 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 in Section 3.2. 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. The formal definitions can be found in Section 3.1, but the high-level descriptions presented here are intended to be sufficient to follow the overview of our main proof in Section 2.3.

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

[2] A very similar version of GD-Fixpoint was also defined by Daskalakis et al. [2020] 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 the $\times$-gates can be used to perform "repeated squaring". To avoid this issue, we restrict ourselves to what we call *well-behaved* arithmetic circuits. See Section 3.1.3 of the preliminaries for more details.

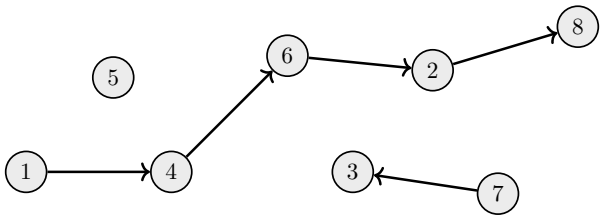


Figure 1: Example of an End-of-Line instance for $n = 3$. The 2^n $(= 8)$ vertices are represented by circular nodes and the directed edges by arrows. Note that the graph is not provided explicitly in the input, but is only represented implicitly by a successor and predecessor circuit. In this example, the End-of-Line solutions are the vertices 3, 7 and 8. In more detail, vertices 3 and 8 are sinks, while vertex 7 is a source. Note that the "trivial" source 1 is not a solution. Finally, the isolated vertex 5 is also not a solution.

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 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 [Morioka, 2001].

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 $C(C(v)) = C(v)$.

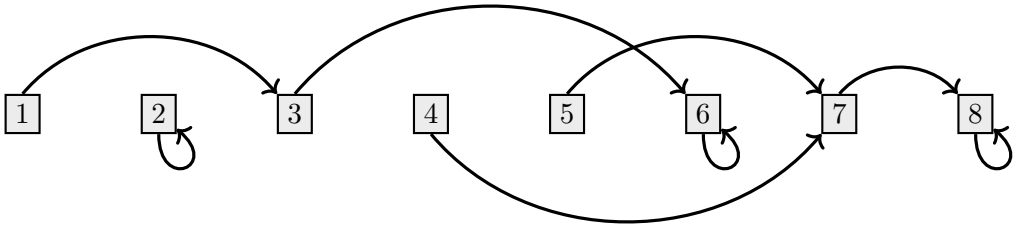


Figure 2: Example of an ITER instance C for $n = 3$. The 2^n $(= 8)$ nodes are represented by squares. The arrows indicate the mapping given by the circuit C. In this example, nodes 2, 6 and 8 are the fixed points of C. Any node that is mapped by C to a fixed point is a solution to the ITER instance. Thus, in this example, the solutions are nodes 3 and 7.

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 of an instance of ITER.

PPAD ∩ PLS. The class PPAD ∩ PLS contains, by definition, all TFNP problems that lie both in 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 [Daskalakis and Papadimitriou, 2011].

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 [2011] 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, showing 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 three problems of interest, establishes the following theorem.

Theorem 5.1. *The problems* KKT, GD-Local-Search, GD-Fixpoint *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.*

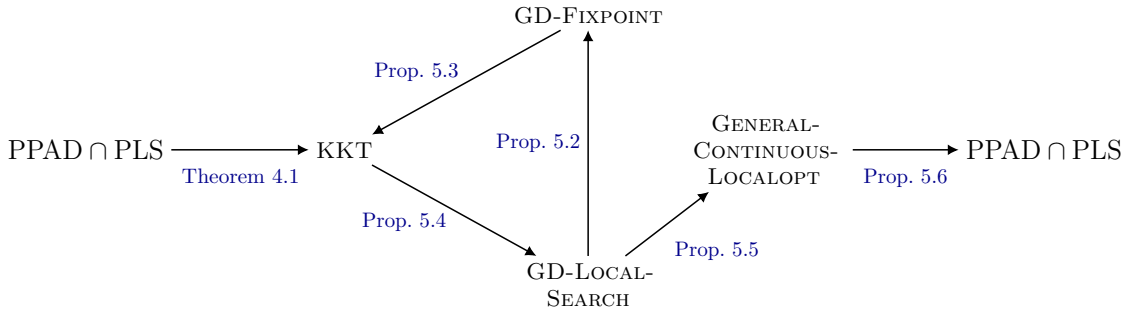


Figure 3: Our reductions. The main one (Theorem 4.1) is on the left; note that the other reductions are all domain- and promise preserving.

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 [Daskalakis et al., 2018; Fearnley et al., 2017] 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 two-dimensional version, a fact that was not previously known. Furthermore, since our results hold even for the promise version of version of General-Continuous-Localopt, 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* (Appendix E), 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 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

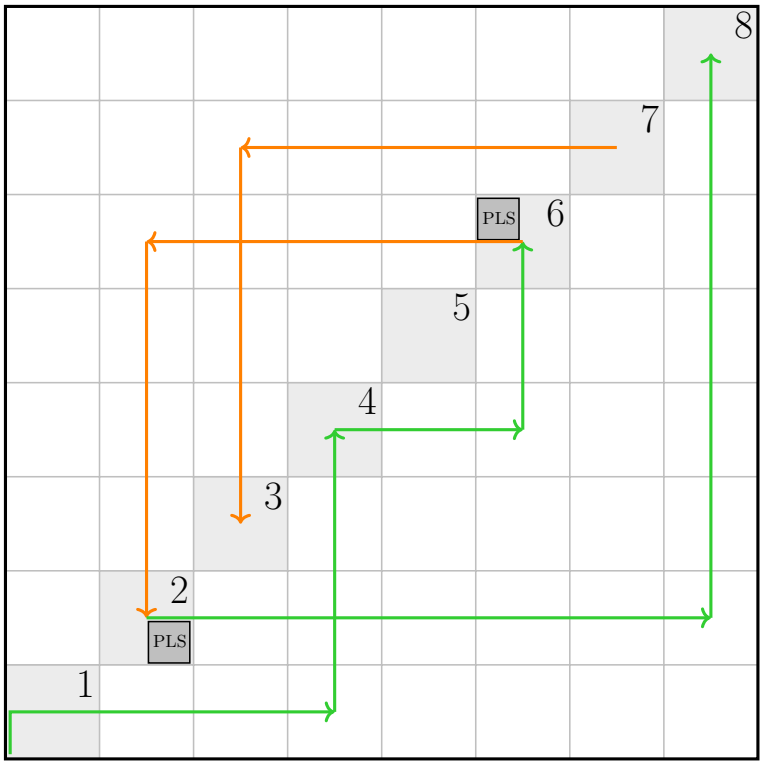


Figure 4: A high-level illustration of our construction. The shaded squares on the diagonal correspond to vertices of the graph represented by I^{EOL}, in this case corresponding to the graph in Figure 1. The green and orange arrows encode the directed edges of the graph. The positions where I^{ITER} is encoded, i.e., the PLS-Labyrinths, are shown as boxes labelled "PLS". They are located at points where the embedding of I^{EOL} would introduce false solutions, and their purpose is to hide those false solutions by co-locating any such solution with a solution to I^{ITER}.

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.

Computer-assisted proof of a required property of 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.

While it would be possible to prove this for all 101 squares by hand, writing such a proof would be error-prone, and verifying it would be extremely tedious. For this reason, we delegate the task to an SMT (Satisfiability modulo theories) solver. We are able to show, for each of our 101 squares, that the statement "bicubic interpolation does not introduce

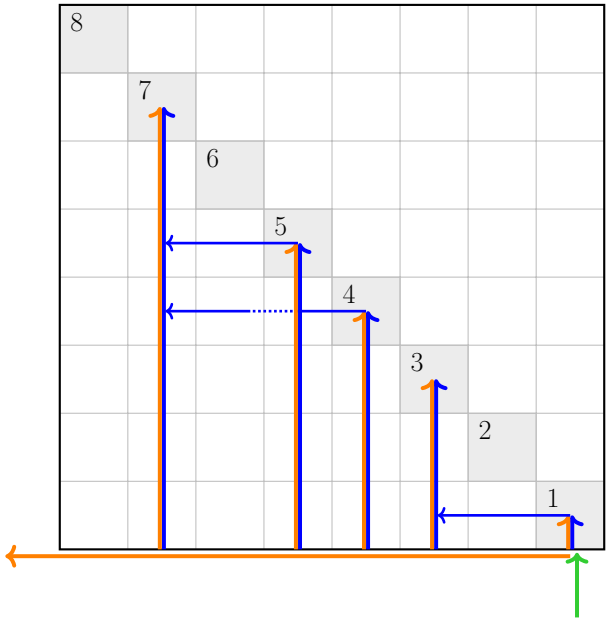


Figure 5: High-level illustration of the PLS-Labyrinth corresponding to the ITER example of Figure 2. Shaded squares on the diagonal correspond to the nodes of ITER. Colours of lines correspond determine how f is constructed at these points. The horizontal blue lines (pointing left) correspond to the 3 edges in Figure 2 that go out from non-solutions, and we do not use similar lines going out from solutions (nodes 3 and 7).

new solutions on this square" can be written down as a formula in the existential theory of the reals, which can then be verified by an SMT solver. When we run the solver, it verifies that bicubic interpolation introduces new solutions only when the small square lies at a solution of I^{ITER} or I^{EOL}.

3 Preliminaries

Let $n \in \mathbb{N}$ be a positive integer. Throughout this paper we use $\|\cdot\|$ to denote the standard Euclidean norm in n-dimensional space, i.e., the ℓ_2-norm in $\mathbb{R}^n$. The maximum-norm, or ℓ_∞-norm, is denoted by $\|\cdot\|_\infty$. For $x, y \in \mathbb{R}^n$, $\langle x, y\rangle := \sum_{i=1}^n x_i y_i$ denotes the inner product. For any non-empty closed convex set $D \subseteq \mathbb{R}^n$, let $\Pi_D : \mathbb{R}^n \to D$ denote the projection onto D with respect to the Euclidean norm. Formally, for any $x \in \mathbb{R}^n$, $\Pi_D(x)$ is the unique point $y \in D$ that minimizes $\|x - y\|$. For $k \in \mathbb{N}$, let $[k] := \{1, 2, \ldots, n\}$.

3.1 Computational Model, Classes and Arithmetic Circuits

We work in the standard Turing machine model. Rational numbers are represented as irreducible fractions, with the numerator and denominator of the irreducible fraction given in binary. Note that given any fraction, it can be made irreducible in polynomial time using the Euclidean algorithm. For a rational number x, we let size(x) denote the number of bits needed to represent x, i.e., the number of bits needed to write down the numerator and denominator (in binary) of the irreducible fraction for x.

3.1.1 NP total search problems and reductions

Search Problems. Let $\{0,1\}^*$ denote the set of all finite length bit-strings and let $|x|$ be the length of $x \in \{0,1\}^*$. A computational search problem is given by a relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$, interpreted as the following problem: given an instance $x \in \{0,1\}^*$, find $y \in \{0,1\}^*$ such that $(x,y) \in R$, or return that no such y exists.

The search problem R is in FNP (*search problems in NP*), if R is polynomial-time computable (i.e., $(x,y) \in R$ can be decided in polynomial time in $|x|+|y|$) and polynomially-balanced (i.e., there exists some polynomial p such that $(x,y) \in R \implies |y| \leq p(|x|)$). Intuitively, FNP contains all search problems where all solutions have size polynomial in the size of the instance and any solution can be checked in polynomial time. The class of all search problems in FNP that can be solved by a polynomial-time algorithm is denoted by FP. The question FP vs. FNP is equivalent to the P vs. NP question.

The class TFNP (*total search problems in NP*) is defined as the set of all FNP problems R that are *total*, i.e., every instance has at least one solution. Formally, R is total, if for every $x \in \{0,1\}^*$ there exists $y \in \{0,1\}^*$ such that $(x,y) \in R$. TFNP lies between FP and FNP.

Note that the totality of TFNP problems does not rely on any promise. Instead, there is a *syntactic* guarantee of totality: for any instance, there is a solution. It is easy to see that a TFNP problem cannot be NP-hard, unless NP = co-NP, Furthermore, it is also believed that no TFNP-complete problem exists. For more details on this, see Megiddo and Papadimitriou [1991].

Reductions between TFNP problems. Let R and S be two TFNP problems. We say that R reduces to S if there exist polynomial-time computable functions $f : \{0,1\}^* \to \{0,1\}^*$ and $g : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ such that, for all $x, y \in \{0,1\}^*$,

$$(f(x), y) \in S \implies (x, g(x,y)) \in R$$

Intuitively, this says that for any instance x of R, if we can find a solution y to instance $f(x)$ of S, then $g(x,y)$ gives us a solution to instance x of R. In particular, note that if S is polynomial-time solvable, then so is R.

3.1.2 The classes PPAD, PLS and PPAD ∩ PLS

Since TFNP problems likely cannot be NP-hard, or TFNP-complete, one instead attempts to classify the problems inside TFNP. Various subclasses of TFNP have been defined and natural problems have been proved complete for these subclasses. In this section we formally define the subclasses PPAD and PLS, which have both been very successful in capturing the complexity of interesting problems.

The most convenient way to define these classes is using problems on Boolean circuits. A Boolean circuit $C : \{0,1\}^n \to \{0,1\}^n$ with n inputs and n outputs, is allowed to use the logic gates $\wedge$ (AND), $\vee$ (OR) and $\neg$ (NOT), where the $\wedge$ and $\vee$ gates have fan-in 2, and the $\neg$ gate has fan-in 1. For ease of notation, we identify $\{0,1\}^n$ with $[2^n]$.

PPAD. The class PPAD is defined as the set of all TFNP problems that reduce to the problem End-of-Line [Papadimitriou, 1994; Daskalakis et al., 2009].

Definition 1. End-of-Line:
Input: Boolean circuits $S, P : [2^n] \to [2^n]$ with $P(1) = 1 \neq S(1)$.
Goal: Find $v \in [2^n]$ such that $P(S(v)) \neq v$ or $S(P(v)) \neq v \neq 1$.

The successor circuit S and the predecessor circuit P implicitly define a directed graph on the vertex set $[2^n]$. There is an edge from v_1 to v_2 if $S(v_1) = v_2$ and $P(v_2) = v_1$. Every vertex has at most one outgoing edge and at most one incoming edge. Since the vertex 1 has one outgoing edge and no incoming edge, it is a source. The goal is to find another end of line, i.e., another source, or a sink of the graph. Note that such a vertex is guaranteed to exist. The condition $P(1) = 1 \neq S(1)$ can be enforced syntactically, so this is indeed a TFNP problem and not a promise problem. See Figure 1 for an example of an End-of-Line instance.

PLS. The class PLS is defined as the set of all TFNP problems that reduce to the problem Localopt [Johnson et al., 1988; Daskalakis and Papadimitriou, 2011].

Definition 2. Localopt:
Input: Boolean circuits $S, V : [2^n] \to [2^n]$.
Goal: Find $v \in [2^n]$ such that $V(S(v)) \geq V(v)$.

This problem embodies local search over the node set $[2^n]$. The output of the circuit V represents a value and ideally we would like to find a node $v \in [2^n]$ that minimises $V(v)$. The circuit S helps us in this task by proposing a possibly improving node $S(v)$ for any v. We stop our search, when we find a v such that $V(S(v)) \geq V(v)$, i.e., S no longer helps us decrease the value of V. This is local search, because the circuit S represents the search for an improving node in some small (polynomial-size) neighbourhood.

In this paper, we also make use of the following PLS-complete problem [Morioka, 2001].

Definition 3. Iter:
Input: Boolean circuit $C : [2^n] \to [2^n]$ with $C(1) > 1$.
Goal: Find v such that either

- $C(v) < v$, or
- $C(v) > v$ and $C(C(v)) = C(v)$.

In this problem, it is convenient to think of the nodes in $[2^n]$ as lying on a line from left to right. Then, we are looking for any node v that is mapped to the left by C, or any node v that is mapped to the right and such that $C(v)$ is a fixed point of C. Since $C(1) > 1$, i.e., node 1 is mapped to the right, it is easy to see that such a solution must exist (apply C repeatedly on node 1). Note that the condition $C(1) > 1$ can be enforced syntactically, so this indeed a TFNP problem and is not a promise problem. See Figure 2 for an example of an Iter instance.

PPAD ∩ PLS. The class PPAD ∩ PLS is the set of all TFNP problems that lie both in PPAD and in PLS. A problem in PPAD ∩ PLS cannot be PPAD- or PLS-complete, unless PPAD ⊆ PLS or PLS ⊆ PPAD. Neither of these two containments is believed to hold, and this is supported by oracle separations between the classes [Beame et al., 1998]. It is easy to construct "artificial" PPAD ∩ PLS-complete problems from PPAD- and PLS-complete problems.

Proposition 3.1 (Daskalakis and Papadimitriou [2011])**.** *For any* TFNP *problems A and B, let* Either-Solution(A,B) *denote the problem: given an instance I_A of A and an instance I_B of B, find a solution of I_A or a solution of I_B. If A is* PPAD*-complete and B is* PLS*-complete, then* Either-Solution(A,B) *is* PPAD ∩ PLS*-complete.*

As a result, we obtain the following corollary, which we will use to show our main PPAD ∩ PLS-hardness result.

Corollary 3.2. EITHER-SOLUTION(END-OF-LINE,ITER) *is* PPAD ∩ PLS*-complete.*

Prior to our work, the problems EITHER-SOLUTION(A,B), where A is PPAD-complete and B is PLS-complete, were the only known PPAD ∩ PLS-complete problems.

3.1.3 Arithmetic circuits and the class CLS

Noting that PPAD ∩ PLS only seemed to have artificial complete problems, Daskalakis and Papadimitriou [2011] defined a subclass of PPAD ∩ PLS with a more natural definition, that combines PPAD and PLS nicely in a single problem. Unlike PPAD and PLS, CLS is defined using arithmetic circuits.

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. For an arithmetic circuit f, we let size(f) denote the size of the circuit, i.e., the number of bits needed to describe the circuit, including the rational constants used therein. Obviously, there are various different ways of defining arithmetic circuits, depending on which gates we allow. The definition we use here is the same as the one used by Daskalakis and Papadimitriou [2011] 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 polynomial time 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. Their modification consists in having an additional input K (in unary) provided as part of the input such that the evaluation of the arithmetic circuit—purportedly—only involves numbers of bit-size at most $K \cdot \text{size}(x)$ on input x. Any point x where the arithmetic circuit fails to satisfy this property is accepted as a solution.

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.

It is easy to see that given an arithmetic circuit f, we can check in polynomial time whether f is well-behaved. Furthermore, these circuits can always be efficiently evaluated.

Lemma 3.3. *Let f be a well-behaved arithmetic circuit with n inputs. Then, for any rational $x \in \mathbb{R}^n$, $f(x)$ can be computed in time* $\text{poly}(\text{size}(f), \text{size}(x))$.

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

We provide a proof of this Lemma in Appendix A.

Using well-behaved arithmetic circuits, instead of the solution proposed by Daskalakis and Papadimitriou, has the advantage that we do not need to add any additional inputs, or any additional violation solutions to our problems. Indeed, the restriction to well-behaved circuits can be enforced syntactically. Furthermore, we note that our problems defined with well-behaved circuits easily reduce to the versions using the solution proposed by Daskalakis and Papadimitriou (see Remark 1 below). Thus, this restriction only makes our hardness results stronger. In fact, for CLS we show that restricting the circuits even further to only use gates $\{+, -, \max, \min, \times\zeta\}$ and rational constants (where $\times\zeta$ is multiplication by a constant), so-called linear arithmetic circuits, does not make the class any weaker (see Section 6.2).

For the problems we consider, it is quite convenient to use arithmetic circuits instead of, say, polynomial-time Turing machines to represent the functions involved. Indeed, the problems could also be defined with polynomial-time Turing machines, but that would introduce some technical subtleties in the definitions (the polynomial used as an upper bound on the running time of the machines would have to be fixed). The important thing to note is that the Turing machine variants of the problems would continue to lie in PPAD $\cap$ PLS. Thus, using arithmetic circuits just makes our hardness results stronger. Note also that in the hard instances we construct, the arithmetic circuits only perform a constant number of true multiplications (see the proof of Lemma 4.2).

Remark 1. The proof of Lemma 3.3 (in Appendix A) shows that if we evaluate a well-behaved arithmetic circuit f on some input x, then, the value $v(g)$ at any gate g of the circuit will satisfy $\text{size}(v(g)) \leq 6 \cdot \text{size}(f)^3 \cdot \text{size}(x)$. As a result, it immediately follows that problems with well-behaved arithmetic circuits can be reduced to the versions of the problems with the modification proposed by Daskalakis and Papadimitriou in the corrigendum of the CLS paper. Indeed, it suffices to let $K = 6 \cdot \text{size}(f)^3$, which can be written down in unary. In particular, this holds for the definition of CLS.

Remark 2. Our definition of well-behaved circuits is robust in the following sense. For any $k \in \mathbb{N}$, say that a circuit f is k-well-behaved if, on any path that leads to an output, there are at most $k \cdot \log(\text{size}(f))$ true multiplication gates. In particular, a circuit is well-behaved if it is 1-well-behaved. It is easy to see that for any fixed $k \in \mathbb{N}$, if we are given a circuit f that is k-well-behaved, we can construct in time $\text{poly}(\text{size}(f))$ a circuit f' that is well-behaved and computes the same function as f. This can be achieved by adding $(\text{size}(f))^k$ dummy gates to the circuit f, i.e., gates that do not alter the output of the circuit. For example, we can add gates that repeatedly add 0 to the output of the circuit.

Lipschitz-continuity. Note that even well-behaved arithmetic circuits might not yield continuous functions, because of the comparison gate. Some of our problems require continuity of the function, and the most convenient type of continuity for computational purposes is Lipschitz-continuity. A function $f : \mathbb{R}^n \to \mathbb{R}^m$ is Lipschitz-continuous on the domain $D \subseteq \mathbb{R}^n$ with Lipschitz-constant L, if for all $x, y \in D$

$$\|f(x) - f(y)\| \leq L \cdot \|x - y\|.$$

Violations and promise-preserving reductions. There is no known way of syntactically enforcing that an arithmetic circuit be Lipschitz-continuous. Thus, to ensure that our problems indeed lie in TFNP, we allow any well-behaved circuit in the input, together

with a purported Lipschitz-constant L, and also accept a pair (x, y) witnessing a violation of L-Lipschitz-continuity as a solution. This "trick" was also used by Daskalakis and Papadimitriou [2011] for the definition of CLS.

One might wonder whether defining a problem in this way, with violations, makes it harder than the (more natural) promise version, where we only consider inputs that satisfy the promise (namely, L-Lipschitz-continuity). We show that for our problems, the promise versions are just as hard. Indeed, the hard instances we construct for the KKT problem satisfy the promises and we then obtain this for the other problems "for free", because all of our reductions are *promise-preserving*, as defined in [Fearnley et al., 2020, Definition 7]. A reduction (f, g) from problem R to problem S is promise-preserving, if for any instance x of R, for any violation solution y of instance $f(x)$ of S, it holds that $g(x, y)$ is a violation solution of instance x of R. Informally: any violation solution of S is mapped back to a violation solution of R.

CLS. The class CLS is defined as the set of all TFNP problems that reduce to 3D-Continuous-Localopt.

Definition 4. Continuous-Localopt:
Input:

- precision/stopping parameter $\varepsilon > 0$,
- well-behaved arithmetic circuits $p : [0, 1]^n \to [0, 1]$ and $g : [0, 1]^n \to [0, 1]^n$,
- Lipschitz constant $L > 0$.

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

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

Alternatively, we also accept one of the following violations as a solution:

- (p is not L-Lipschitz) $x, y \in [0, 1]^n$ such that $|p(x) - p(y)| > L\|x - y\|$,
- (g is not L-Lipschitz) $x, y \in [0, 1]^n$ such that $\|g(x) - g(y)\| > L\|x - y\|$.

For $k \in \mathbb{N}$, we let kD-Continuous-Localopt denote the problem Continuous-Localopt where n is fixed to be equal to k.

Continuous-Localopt is similar to Localopt, in the sense that we are looking for a minimum of p over the domain $[0, 1]^n$ using the help of a function g. The membership of the problem in PLS and in PPAD is easy to show [Daskalakis and Papadimitriou, 2011]. The membership in PPAD follows from the observation that g is a Brouwer function and that every (approximate) fixed point of g also yields a solution to the Continuous-Localopt instance.

Note that the original definition of Continuous-Localopt in [Daskalakis and Papadimitriou, 2011] uses arithmetic circuits without the "well-behaved" restriction. As argued above, these circuits cannot always be evaluated efficiently, and so we instead use well-behaved arithmetic circuits, to ensure that the problem lies in TFNP. The interesting problems shown to lie in CLS by Daskalakis and Papadimitriou [2011] still reduce to Continuous-Localopt even with this restriction on the circuits. It also turns out that this restriction does not make the class any weaker, since we show that 2D-Continuous-Localopt with well-behaved arithmetic circuits is PPAD $\cap$ PLS-hard.

In this paper, we consider more general domains than just $[0,1]^n$ and so we also define a more general version of Continuous-Localopt.

Definition 5. General-Continuous-Localopt:
Input:

- precision/stopping parameter $\varepsilon > 0$,
- $(A, b) \in \mathbb{R}^{m \times n} \times \mathbb{R}^m$ defining a bounded non-empty domain $D = \{x \in \mathbb{R}^n : Ax \leq b\}$,
- well-behaved arithmetic circuits $p : \mathbb{R}^n \to \mathbb{R}$ and $g : \mathbb{R}^n \to \mathbb{R}^n$,
- Lipschitz constant $L > 0$.

Goal: Compute an approximate local optimum of p with respect to g on domain D. Formally, find $x \in D$ such that

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

Alternatively, we also accept one of the following violations as a solution:

- (p is not L-Lipschitz) $x, y \in D$ such that $|p(x) - p(y)| > L\|x - y\|$,
- (g is not L-Lipschitz) $x, y \in D$ such that $\|g(x) - g(y)\| > L\|x - y\|$.

Note that given $(A, b) \in \mathbb{R}^{m \times n} \times \mathbb{R}^m$, it is easy to check whether the domain $D = \{x \in \mathbb{R}^n : Ax \leq b\}$ is bounded and non-empty by using linear programming.

We use the projection Π_D in this definition, because it is not clear whether there is some syntactic way of ensuring that $g(x) \in D$. Note that Π_D can be computed efficiently by using convex quadratic programming, but it is unclear whether it can be computed by our arithmetic circuits. When $D = [0,1]^n$, the projection Π_D can easily be computed by arithmetic circuits, so Π_D is not needed in the definition of Continuous-Localopt. Indeed, when $D = [0,1]^n$, we have $[\Pi_D(x)]_i = \min\{1, \max\{0, x_i\}\}$ for all $i \in [n]$ and $x \in \mathbb{R}^n$.

The definition of CLS using 3D-Continuous-Localopt, instead of 2D-Continuous-Localopt, Continuous-Localopt, or General-Continuous-Localopt, leaves open various questions about whether all these different ways of defining it are equivalent. We prove that this is indeed the case. We discuss this, as well as the robustness of the definition of CLS with respect to other modifications in Section 6.

3.2 Computational Problems from Nonlinear Optimization

In this section we formally define our three problems of interest. We begin by a brief introduction to nonlinear optimization.

3.2.1 Background on Nonlinear Optimization

The standard problem of nonlinear optimization (also called nonlinear programming) can be formulated as follows:

$$\begin{aligned} &\min_{x \in \mathbb{R}^n} f(x) \\ &\text{s.t.} \quad g_i(x) \leq 0 \quad \forall i \in [m] \end{aligned} \tag{1}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is the objective function to be minimised, and $g_1, \ldots, g_m : \mathbb{R}^n \to \mathbb{R}$ are the inequality constraint functions. It is assumed that f, g_i are C^1, i.e., continuously differentiable. Throughout this paper we consider the minimisation problem, but our results also apply to the maximisation problem, since we consider function classes that are closed under negation.

Global minimum. Unfortunately, solving the optimisation problem (1), namely computing a *global* minimum, is intractable, even for relatively simple objective functions and constraints ([Murty and Kabadi, 1987] in the context of quadratic programming, [Blum and Rivest, 1992] in the context of neural networks).

Local minima. The most natural way to relax the requirement of a global minimum, is to look for a *local* minimum instead. A point $x \in \mathbb{R}^n$ is a local minimum of (1), if it satisfies all the constraints, namely $x \in D$, where $D = \{y \in \mathbb{R}^n \mid g_i(x) \leq 0 \, \forall i \in [m]\}$, and if there exists $\epsilon > 0$ such that

$$f(x) \leq f(y) \qquad \forall y \in D \cap B_\epsilon(x) \tag{2}$$

where $B_\epsilon(x) = \{y \in \mathbb{R}^n \mid \|y - x\| \leq \epsilon\}$.

However, while the notion of a local minimum is very natural, an important issue arises when the problem is considered from the computational perspective. Looking at expression (2), it not clear how to efficiently check whether a given point x is a local minimum or not. Indeed, it turns out that deciding whether a given point is a local minimum is co-NP-hard, even for simple objective and constraint functions [Murty and Kabadi, 1987]. Furthermore, it was recently shown that computing a local minimum, even when it is guaranteed to exist, cannot be done in polynomial time unless P = NP [Ahmadi and Zhang, 2020a], even for quadratic functions where the domain is a polytope.

Necessary optimality conditions. In order to avoid this issue, one can instead look for a point satisfying some so-called *necessary optimality conditions*. As the name suggests, these are conditions that must be satisfied for any local minimum, but might also be satisfied for points that are not local minima. Importantly, these conditions can usually be checked in polynomial time. For this reason, algorithms attempting to solve (1), usually try to find a point that satisfies some necessary optimality conditions instead.

KKT points. The most famous and simplest necessary optimality conditions are the Karush-Kuhn-Tucker (KKT) conditions. The KKT conditions are first-order conditions in the sense that they only involve the first derivatives (i.e., the gradients) of the functions in the problem statement. Formally, a point $x \in \mathbb{R}^n$ satisfies the KKT conditions if it is feasible, i.e., $x \in D = \{y \in \mathbb{R}^n \mid g_i(x) \leq 0 \, \forall i \in [m]\}$ and if there exist $\mu_1, \ldots, \mu_m \geq 0$ such that

$$\nabla f(x) + \sum_{i=1}^{m} \mu_i \nabla g_i(x) = 0$$

and $\mu_i g_i(x) = 0$ for all $i \in [m]$. This last condition ensures that $\mu_i > 0$ can only occur if $g_i(x) = 0$, i.e., if the ith constraint is tight. In particular, if no constraint is tight at x, then x is a KKT point if $\nabla f(x) = 0$ (in other words, if it is a stationary point). A point x that satisfies the KKT conditions is also called a KKT point. Note that given access to $\nabla f(x)$, $g_i(x)$ and $\nabla g_i(x)$, one can check in polynomial time whether x is a KKT point, since this reduces to checking the feasibility of a linear program.

Every local minimum of (1) must satisfy the KKT conditions, as long as the problem satisfies some so-called regularity conditions or constraint qualifications. In this paper,

we restrict our attention to linear constraints (i.e., $g_i(x) = \langle a_i, x\rangle - b_i$). In this case, it is known that every local minimum is indeed a KKT point.

ε-KKT points. In practice, but also when studying the computational complexity in the standard Turing model (because of issues of representation), it is unreasonable to expect to find a point that exactly satisfies the KKT conditions. Instead, one looks for an *approximate* KKT point. Given $\varepsilon \geq 0$, we say that $x \in \mathbb{R}^n$ is an ε-KKT point if $x \in D$ and if there exist $\mu_1, \ldots, \mu_m \geq 0$ such that

$$\left\| \nabla f(x) + \sum_{i=1}^{m} \mu_i \nabla g_i(x) \right\| \leq \varepsilon$$

and $\mu_i g_i(x) = 0$ for all $i \in [m]$. In particular, if no constraint is tight at x, then x is an ε-KKT point if $\|\nabla f(x)\| \leq \varepsilon$. Since $\|\cdot\|$ denotes the ℓ_2-norm, we can check whether a point is an ε-KKT point in polynomial time by solving a convex quadratic program. If we instead use the ℓ_∞-norm or the ℓ_1-norm in the definition of ε-KKT point, then we can check whether a point is an ε-KKT point in polynomial time by solving a linear program.

Since we focus on the case where $D = \{y \in \mathbb{R}^n \mid Ay \leq b\}$, $(A, b) \in \mathbb{R}^{m\times n} \times \mathbb{R}^m$, we can rewrite the KKT conditions as follows. A point $x \in \mathbb{R}^n$ is an ε-KKT point if $x \in D$ and if there exist $\mu_1, \ldots, \mu_m \geq 0$ such that

$$\left\| \nabla f(x) + A^T \mu \right\| \leq \varepsilon$$

and $\langle \mu, Ax - b\rangle = 0$. Note that this exactly corresponds to the earlier definition adapted to this case. In particular, the condition "$\mu_i [Ax - b]_i = 0$ for all $i \in [m]$" is equivalent to $\langle \mu, Ax - b\rangle = 0$, since $\mu_i \geq 0$ and $[Ax - b]_i \leq 0$ for all $i \in [m]$.

It is known that if there are no constraints, then it is NP-hard to decide whether a KKT point exists [Ahmadi and Zhang, 2020b]. This implies that, in general, unless P = NP, there is no polynomial-time algorithm that computes a KKT point of (1). However, this hardness result does not say anything about one very important special case, namely when the feasible region D is a compact set (in particular, when it is a bounded polytope defined by linear constraints). Indeed, in that case, a KKT point is guaranteed to exist—since a local minimum is guaranteed to exist—and easy to verify, and thus finding a KKT point is a total search problem in the class TFNP. In particular, this means that, for compact D, the problem of computing a KKT point cannot be NP-hard, unless NP = co-NP [Megiddo and Papadimitriou, 1991]. In this paper, we provide strong evidence that the problem remains hard for such bounded domains, and, in fact, even when the feasible region is as simple as $D = [0, 1]^2$.

The problem of finding an ε-KKT point has also been studied in the "black box" model, where we only have oracle access to the function and its gradient, and count the number of oracle calls needed to solve the problem. Vavasis [1993] proved that at least $\Omega(\sqrt{L/\varepsilon})$ calls are needed to find an ε-KKT point of a continuously differentiable function $f : [0, 1]^2 \to \mathbb{R}$ with L-Lipschitz gradient. It was recently shown by Bubeck and Mikulincer [2020] that this bound is tight up to a logarithmic factor. For the high-dimensional case, Carmon et al. [2019] showed a tight bound of $\Theta(1/\varepsilon^2)$, when the Lipschitz constant is fixed.

3.2.2 The KKT problem

Given the definition of ε-KKT points in the previous section, we can formally define a computational problem where the goal is to compute such a point. Our formalisation of this

problem assumes that f and ∇f are provided in the input as arithmetic circuits. However, it is unclear if, given a circuit f, we can efficiently determine whether it corresponds to a continuously differentiable function, and whether the circuit for ∇f indeed computes its gradient. Thus, one has to either consider the promise version of the problem (where this is guaranteed to hold for the input), or add violation solutions like in the definition of Continuous-Localopt. In order to ensure that our problem is in TFNP, we formally define it with violation solutions. However, we note that our hardness results also hold for the promise versions.

The type of violation solution that we introduce to ensure that ∇f is indeed the gradient of f is based on the following version of Taylor's theorem, which is proved in Appendix B.2.

Lemma 3.4 (Taylor's theorem)**.** *Let $f : \mathbb{R}^n \to \mathbb{R}$ be continuously differentiable and let $D \subseteq \mathbb{R}^n$ be convex. If ∇f is L-Lipschitz-continuous (w.r.t. the ℓ_2-norm) on D, then for all $x, y \in D$ we have*

$$\left|f(y) - f(x) - \langle \nabla f(x), y - x\rangle\right| \leq \frac{L}{2}\|y - x\|^2.$$

We are now ready to formally define our KKT problem.

Definition 6. KKT:
Input:

- precision parameter $\varepsilon > 0$,
- $(A, b) \in \mathbb{R}^{m\times n} \times \mathbb{R}^m$ defining a bounded non-empty domain $D = \{x \in \mathbb{R}^n : Ax \leq b\}$,
- well-behaved arithmetic circuits $f : \mathbb{R}^n \to \mathbb{R}$ and $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$,
- Lipschitz constant $L > 0$.

Goal: Compute an ε-KKT point for the minimization problem of f on domain D. Formally, find $x \in D$ such that there exist $\mu_1, \ldots, \mu_m \geq 0$ such that

$$\left\|\nabla f(x) + A^T\mu\right\| \leq \varepsilon$$

and $\langle \mu, Ax - b\rangle = 0$.
Alternatively, we also accept one of the following violations as a solution:

- (f or ∇f is not L-Lipschitz) $x, y \in D$ such that

$$|f(x) - f(y)| > L\|x - y\| \qquad \text{or} \qquad \|\nabla f(x) - \nabla f(y)\| > L\|x - y\|,$$

- (∇f is not the gradient of f) $x, y \in D$ that contradict Taylor's theorem (Lemma 3.4), i.e.,

$$\left|f(y) - f(x) - \langle \nabla f(x), y - x\rangle\right| \leq \frac{L}{2}\|y - x\|^2.$$

Note that all conditions on the input of the KKT problem can be checked in polynomial time. In particular, we can use linear programming to check that the domain is bounded and non-empty. With regards to a solution $x \in D$, there is no need to include the values

$\mu_1, \ldots, \mu_m$ as part of a solution. Indeed, given $x \in D$, we can check in polynomial time whether there exist such $\mu_1, \ldots, \mu_m$ by solving the following convex quadratic program:

$$\begin{aligned} \min_{\mu \in \mathbb{R}^m} \quad & \left\| \nabla f(x) + A^T \mu \right\|^2 \\ \text{s.t.} \quad & \langle \mu, Ax - b \rangle = 0 \\ & \mu \geq 0 \end{aligned}$$

If the optimal value of this program is strictly larger than ε^2, then x is not an ε-KKT point. Otherwise, it is an ε-KKT point and the optimal $\mu_1, \ldots, \mu_m$ certify this. If we use the ℓ_∞-norm or the ℓ_1-norm instead of the ℓ_2-norm for the definition of ε-KKT points, then we can check whether a point is an ε-KKT point using the same approach (except that we do not take the square of the norm, and we simply obtain a linear program). Whether we use the ℓ_2-norm, the ℓ_∞-norm or the ℓ_1-norm for the definition of ε-KKT points has no impact on the complexity of the KKT problem defined above. Indeed, is is easy to reduce the various versions to each other.

Note that ε and L are provided in binary representation in the input. This is important, since our hardness result in Theorem 4.1 relies on at least one of those two parameters being exponential in the size of the input. If both parameters are provided in unary, then the problem can be solved in polynomial time on domain $[0,1]^n$ (see Lemma C.4).

3.2.3 Gradient Descent problems

In this section we formally define our two versions of the Gradient Descent problem. Since we consider Gradient Descent on bounded domains D, we need to ensure that the next iterate indeed lies in D. The standard way to handle this is by using so-called *Projected* Gradient Descent, where the next iterate is computed using a standard Gradient Descent step and then projected onto D using Π_D. Formally,

$$x^{(k+1)} \leftarrow \Pi_D \left(x^{(k)} - \eta \nabla f\big(x^{(k)}\big) \right)$$

where $\eta > 0$ is the step size. Throughout, we only consider the case where the step size is fixed, i.e., the same in all iterations. Our first version of the problem considers the stopping criterion is: stop if the next iterate improves the objective function value by less than ε.

Definition 7. GD-Local-Search:
Input:

- precision/stopping parameter $\varepsilon > 0$,
- step size $\eta > 0$,
- $(A, b) \in \mathbb{R}^{m \times n} \times \mathbb{R}^m$ defining a bounded non-empty domain $D = \{x \in \mathbb{R}^n : Ax \leq b\}$,
- well-behaved arithmetic circuits $f : \mathbb{R}^n \to \mathbb{R}$ and $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$,
- Lipschitz constant $L > 0$.

Goal: Compute any point where (projected) gradient descent for f on domain D with fixed step size η terminates. Formally, find $x \in D$ such that

$$f\Big(\Pi_D\big(x - \eta \nabla f(x)\big)\Big) \geq f(x) - \varepsilon.$$

Alternatively, we also accept one of the following violations as a solution:

- (f or ∇f is not L-Lipschitz) $x, y \in D$ such that

$$|f(x) - f(y)| > L\|x - y\| \qquad \text{or} \qquad \|\nabla f(x) - \nabla f(y)\| > L\|x - y\|,$$

- (∇f is not the gradient of f) $x, y \in D$ that contradict Taylor's theorem (Lemma 3.4), i.e.,

$$\left|f(y) - f(x) - \langle \nabla f(x), y - x\rangle\right| \leq \frac{L}{2}\|y - x\|^2.$$

Our second version of the problem considers the stopping criterion: stop if the next iterate is ε-close to the current iterate.

Definition 8. GD-Fixpoint:
Input:

- precision/stopping parameter $\varepsilon > 0$,
- step size $\eta > 0$,
- $(A, b) \in \mathbb{R}^{m\times n} \times \mathbb{R}^m$ defining a bounded non-empty domain $D = \{x \in \mathbb{R}^n : Ax \leq b\}$,
- well-behaved arithmetic circuits $f : \mathbb{R}^n \to \mathbb{R}$ and $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$,
- Lipschitz constant $L > 0$.

Goal: Compute any point that is an ε-approximate fixed point of (projected) gradient descent for f on domain D with fixed step size η. Formally, find $x \in D$ such that

$$\left\|x - \Pi_D\big(x - \eta\nabla f(x)\big)\right\| \leq \varepsilon.$$

Alternatively, we also accept one of the following violations as a solution:

- (f or ∇f is not L-Lipschitz) $x, y \in D$ such that

$$|f(x) - f(y)| > L\|x - y\| \qquad \text{or} \qquad \|\nabla f(x) - \nabla f(y)\| > L\|x - y\|,$$

- (∇f is not the gradient of f) $x, y \in D$ that contradict Taylor's theorem (Lemma 3.4), i.e.,

$$\left|f(y) - f(x) - \langle \nabla f(x), y - x\rangle\right| \leq \frac{L}{2}\|y - x\|^2.$$

The comments made about the KKT problem in the previous section also apply to these two problems. In particular, we show that even the promise versions of the two Gradient Descent problems remain PPAD $\cap$ PLS-hard. In other words, the hard instances we construct have no violations.

4 KKT is PPAD ∩ PLS-hard

In this section, we prove our main technical result.

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 6 illustrates the high-level idea of the embedding with an example.

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 6, there will 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 6.

- 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 6.

- 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 6.

- 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 6.
  - 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 4.1.3. The PLS-Labyrinth gadget is described in detail in Section 4.1.4.

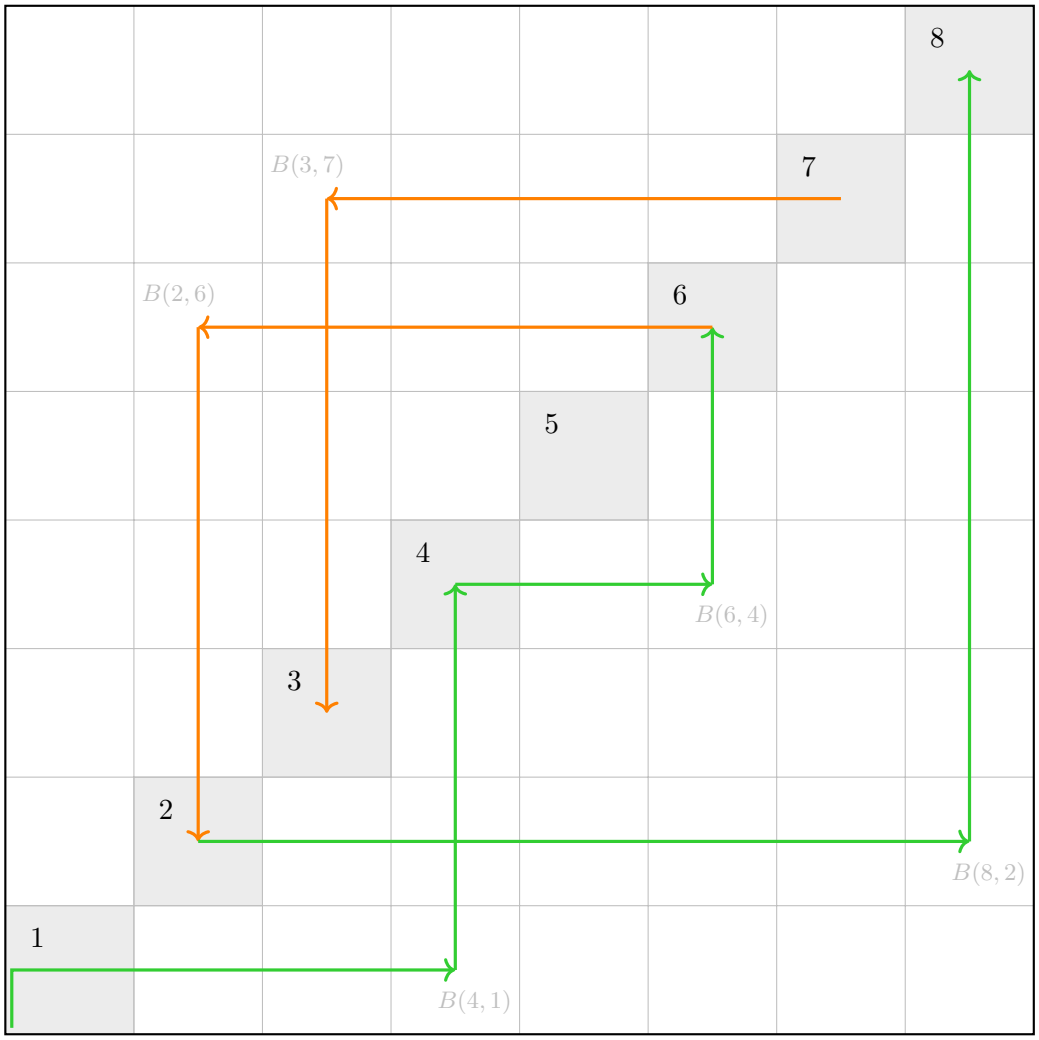


Figure 6: Example of the high-level idea for the embedding of an End-of-Line instance in the domain. In this example we are embedding an End-of-Line instance with the set of vertices $[8]$ (i.e., $n = 3$) and the directed edges: $(1, 4)$, $(2, 8)$, $(4, 6)$, $(6, 2)$ and $(7, 3)$ (see Figure 1). The domain is divided into 8×8 big squares, and the big squares corresponding to the vertices of the End-of-Line graph are coloured in grey. The solutions of this End-of-Line instance are the vertices 3, 7 and 8.

4.1.1 Pre-processing

Consider any instance $((S, P), C)$ of Either-Solution(End-of-Line,Iter), i.e., $S, P : [2^n] \to [2^n]$ is an instance of End-of-Line and $C : [2^m] \to [2^m]$ is an instance of Iter. Without loss of generality, we can assume that these instances satisfy the following:

1. The successor and predecessor circuits S, P agree on all edges. Formally, for all $v \in [2^n]$, it holds that

   - if $S(v) \neq v$, then $P(S(v)) = v$, and
   - if $P(v) \neq v$, then $S(P(v)) = v$.

   This property can be ensured by a simple pre-processing step. We modify the circuit S, so that before outputting $S(v)$, it first checks whether $(S(v) \neq v) \wedge (P(S(v)) \neq v)$, and, if this holds, outputs v instead of $S(v)$. It is easy to see that this new circuit for S can be constructed in polynomial time in the size of S and P. We also perform the analogous modification for P. It is easy to check that this does not introduce any new solutions.

2. For all $u \in [2^m]$ we have $C(u) \geq u$. We can ensure that this holds by modifying the circuit C, so that before outputting $C(u)$, it checks whether $C(u) < u$, and, if this is the case, outputs u instead of $C(u)$. Again, the modification can be done in polynomial time and does not introduce new solutions, nor does it stop the problem from being total.

4.1.2 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 7 illustrates the main properties of the value regimes.

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., $x = 0$).

[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.

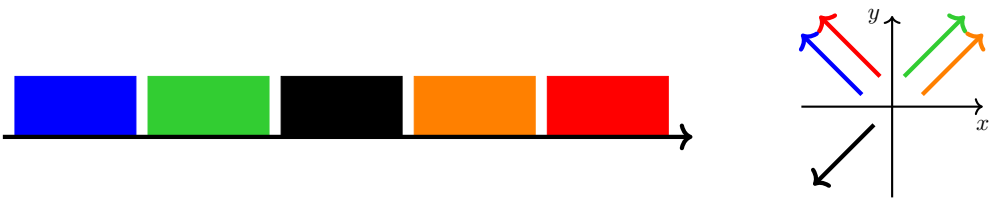


Figure 7: The value regimes. On the left, the colours are ordered according to increasing value, from left to right. On the right, we indicate for each value regime, the direction in which it improves, i.e., decreases, in the x-y-plane.

4.1.3 Embedding the End-of-Line Instance: The Green and Orange Paths

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.

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. Figure 8(a) shows a big square that is traversed by a green path from left to right. Such a big square is said to be of type G1. The black arrows indicate the direction of $-\nabla f$ at every grid point.

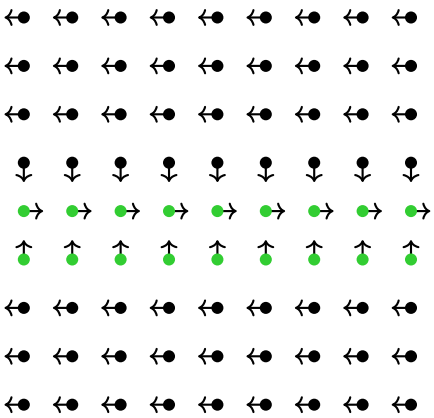
When a green path moves upwards, the two right-most grid points will be coloured in green, and the grid point on the left will be in black. Figure 8(b) shows a big square of type G2, namely one that is traversed by a green path from the bottom to the top.

Recall that a green path implementing 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 8(c) shows how this turn is implemented. The big square $B(v_2, v_1)$ is said to be of type G3.

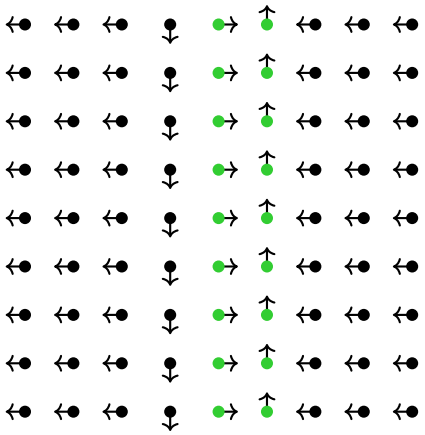
If a vertex $v \in [2^n]$ has one incoming edge (v_1, v) and one outgoing edge (v, v_2) such that $v_1 < v < v_2$, then both edges will be implemented by green paths. The green path corresponding to (v_1, v) will enter $B(v, v)$ from the bottom and stop at the centre of $B(v, v)$. The green path corresponding 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, as shown in Figure 8(d). The big square $B(v, v)$ is said to be of type G4.

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, as shown in Figure 8(e). The big square $B(v, v)$ is said to be of type G5 in that case. As we will show later, 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 $B(v, v)$, as shown in Figure 8(f). The big square $B(v, v)$ is said to be of type G6 in that case. As we will show later, there will be a KKT point at the sink of that green path.

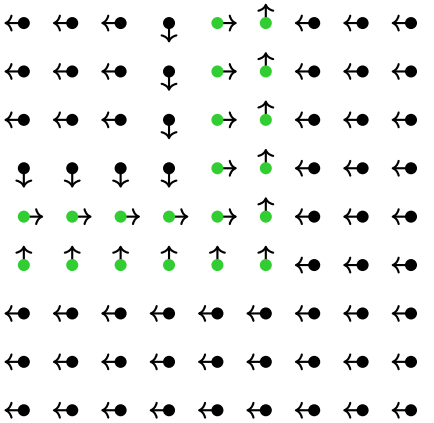
Orange paths. The structure of orange paths is, in a certain sense, symmetric to the structure of green paths. When an orange path moves to the left, the two upper grid points will be coloured in orange, and the grid point at the bottom will be in black. Figure 9(a) shows a big square that is traversed by an orange path from right to left. Such a big square



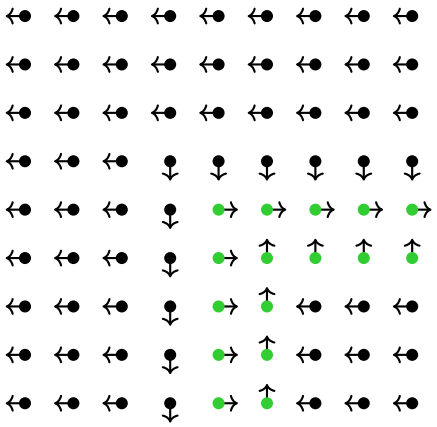
(a) [G1] Green path traversing big square from left to right.



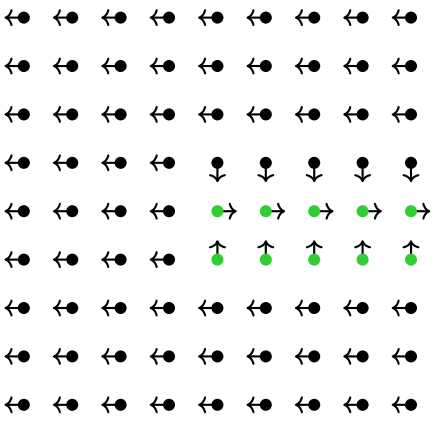
(b) [G2] Green path traversing big square from bottom to top.



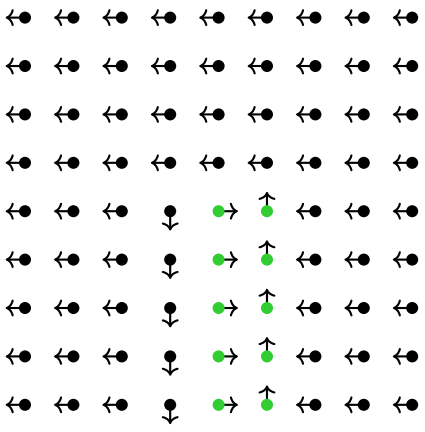
(c) [G3] Green path entering big square from the left, turning, and leaving at the top.



(d) [G4] Green path entering big square from the bottom, turning, and leaving on the right.



(e) [G5] Source: green path starting at the centre of big square and leaving on the right.



(f) [G6] Sink: green path entering big square from the bottom and ending at the centre.

Figure 8: Construction of the green paths. The figures show various types of big squares containing different portions of green paths. In these illustrations, the big squares are assumed to have size 8×8 instead of $2^{m+4} \times 2^{m+4}$.

is said to be of type O1.

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. Figure 9(b) shows a big square of type O2, namely one that is traversed by an orange path from top to bottom.

An orange path implementing an edge (v_1, v_2) (where $v_1 > v_2$) comes into the big square $B(v_2, v_1)$ from the right and leaves at the bottom. This turn is implemented as shown in Figure 9(c). The big square $B(v_2, v_1)$ is said to be of type O3.

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 orange paths. The orange path corresponding to (v_1, v) will enter $B(v, v)$ from the top and stop at the centre of $B(v, v)$. The orange path corresponding to (v, v_2) will start at the centre of $B(v, v)$ and leave the big square on the left. As above, we avoid introducing a KKT point by connecting the two paths at the centre of $B(v, v)$. This is achieved by the turn shown in Figure 9(d). The big square $B(v, v)$ is said to be of type O4.

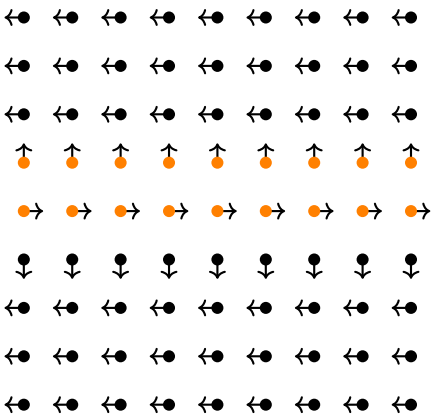
If a vertex $v \in [2^n] \setminus \{1\}$ has one outgoing edge (v, v_2) such that $v > v_2$, and no incoming edge, then this will yield an orange path starting at the centre of $B(v, v)$ and going to the left, as shown in Figure 9(e). The big square $B(v, v)$ is said to be of type O5 in that case. As we will show later, there will be a KKT point at the source of that orange 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 an orange path coming from the top and ending at the centre of $B(v, v)$, as shown in Figure 9(f). The big square $B(v, v)$ is said to be of type O6 in that case. As we will show later, there will be a KKT point at the sink of that 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. First of all, note that it is impossible to have more than two paths traversing a big square, and thus any crossing involves exactly two paths. Furthermore, no crossing can occur in big squares where a "turn" occurs, since, in that case, the turn connects the two paths.

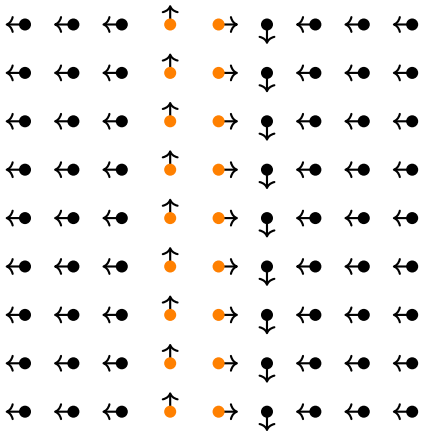
The only way for two green paths to cross is the case where a green path traverses a big square from left to right, and a second green path traverses the same big square from bottom to top. In that case, we say that the big square is of type G7. This problem always occurs when one tries to embed an End-of-Line instance in a two-dimensional domain. Chen and Deng [2009] 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 10(a) shows how this modification is implemented for crossing green paths, i.e., what our construction does for big squares of type G7.

The same issue might arise for orange paths. By the same arguments as above, this can only happen when an orange path traverses a big square from right to left, and a second orange path traverses the same big square from top to bottom. In that case, we say that the big square is of type O7. Figure 10(b) shows how the issue is locally resolved in that case, i.e., what our construction does for big squares of type O7.

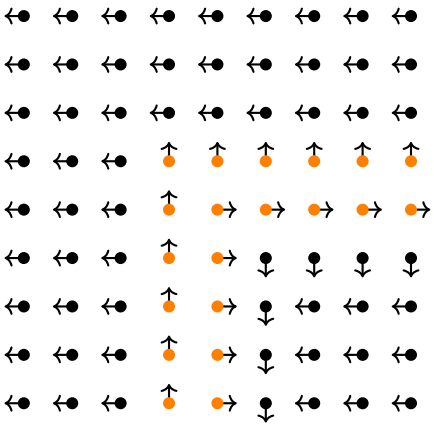
Boundary and origin squares. Any big square that is not traversed by any path (including all big squares $B(v, v)$ where v is an isolated vertex of the End-of-Line



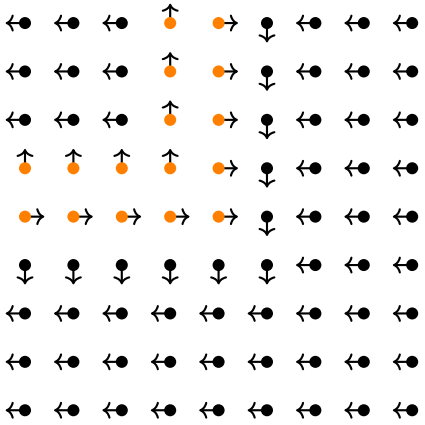
(a) [O1] Orange path traversing big square from right to left.



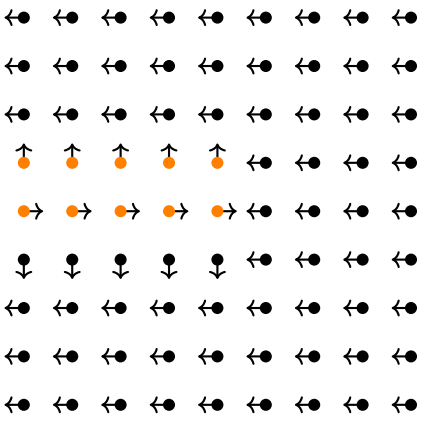
(b) [O2] Orange path traversing big square from top to bottom.



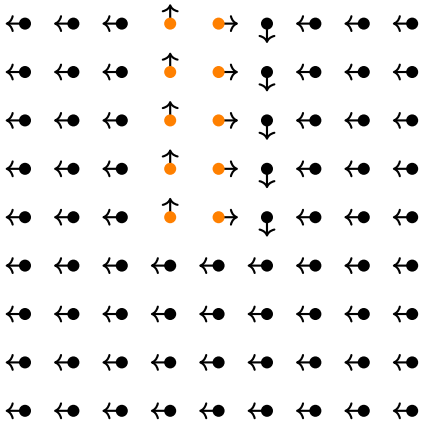
(c) [O3] Orange path entering big square from the right, turning, and leaving at the bottom.



(d) [O4] Orange path entering big square from the top, turning, and leaving on the left.

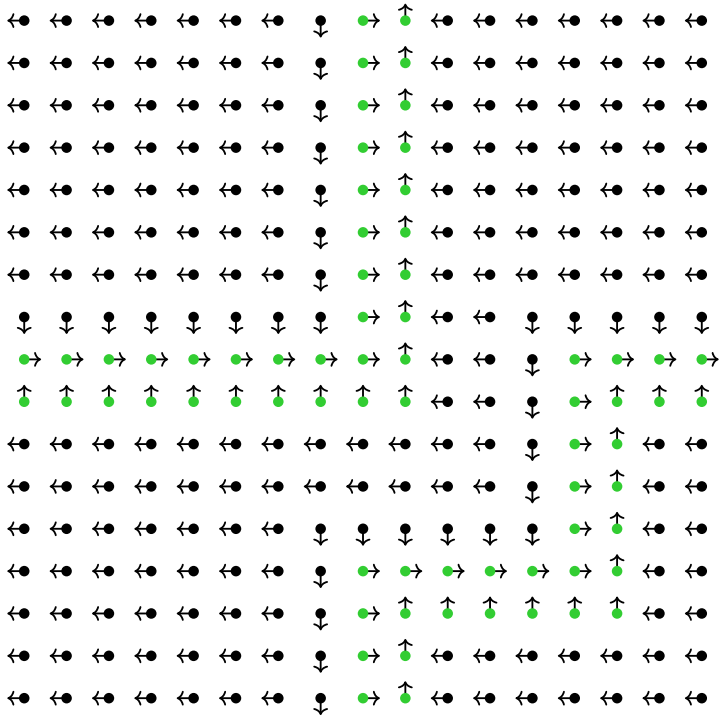


(e) [O5] Source: orange path starting at the centre of big square and leaving on the left.

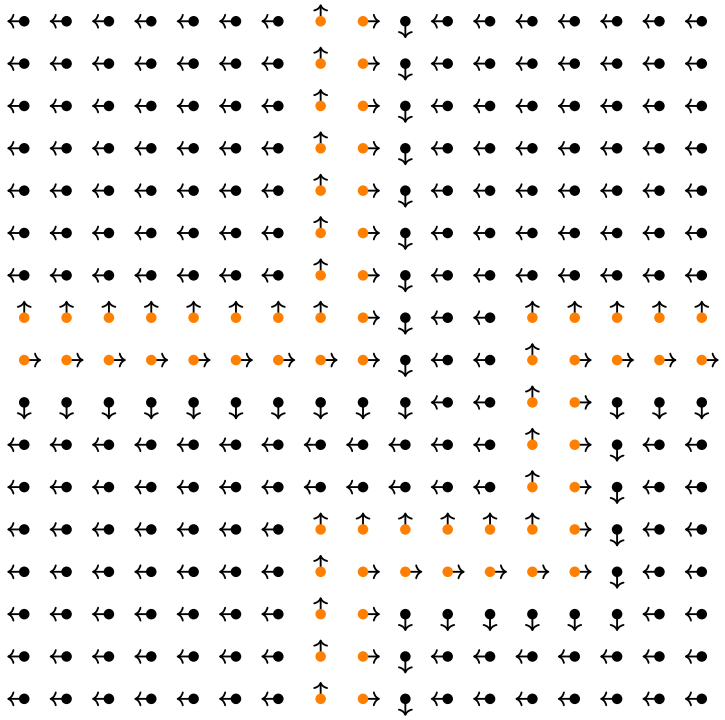


(f) [O6] Sink: orange path entering big square from the top and ending at the centre.

Figure 9: Construction of the orange paths. The figures show various types of big squares containing different portions of orange paths. In these illustrations, the big squares are assumed to have size 8×8 instead of $2^{m+4} \times 2^{m+4}$.



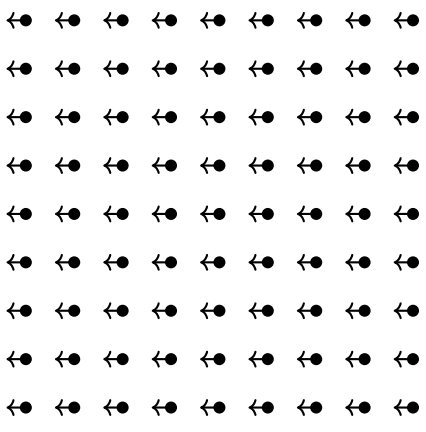
(a) [G7] Crossing of green paths.



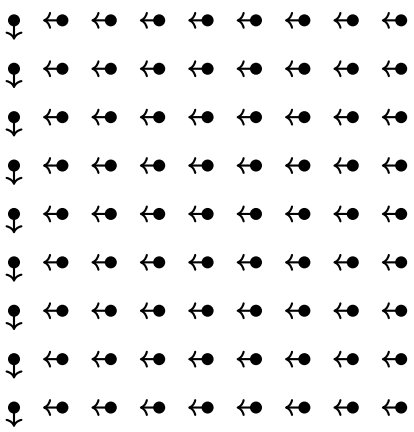
(b) [O7] Crossing of orange paths.

Figure 10: Crossing gadgets for green and orange paths. In these two illustrations, the big squares are assumed to have size 16×16 instead of $2^{m+4} \times 2^{m+4}$.

instance), will have all its grid points coloured in black, and $-\nabla f$ pointing to the left. These big squares, which are said to be of type E1, are as represented in Figure 11(a). The only exceptions to this rule are the big squares $B(1, v)$ for all $v \in [2^n] \setminus \{1\}$. In those big squares, which are said to be of type E2, the grid points on the left boundary have $-\nabla f$ pointing downwards, instead of to the left. The rest of the grid points have $-\nabla f$ pointing to the left as before. Note that none of these big squares is ever traversed by a path, so they are always as shown in Figure 11(b).



(a) [E1] Big square not traversed by any path.



(b) [E2] Big square on left boundary of domain.

Figure 11: Big squares not traversed by any path. In these two illustrations, the big squares are assumed to have size 8×8 instead of $2^{m+4} \times 2^{m+4}$.

The big square $B(1, 1)$ is special and we say that it is of type S. 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 $B(1, 1)$ (as in Figure 8(e)). 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 [2017] 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 $B(1, 1)$. The construction for $B(1, 1)$ is shown in Figure 12.

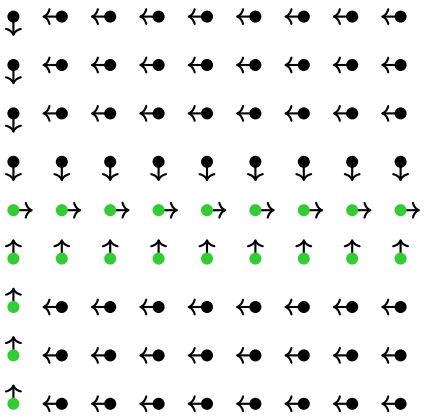


Figure 12: [S] Construction for big square $B(1, 1)$ (for size 8×8 instead of $2^{m+4} \times 2^{m+4}$).

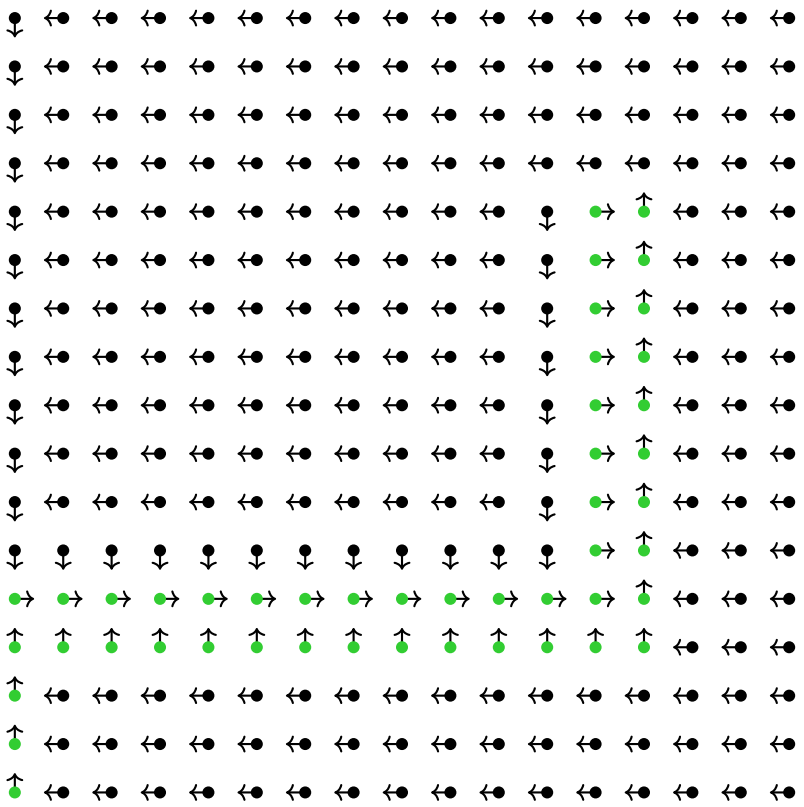
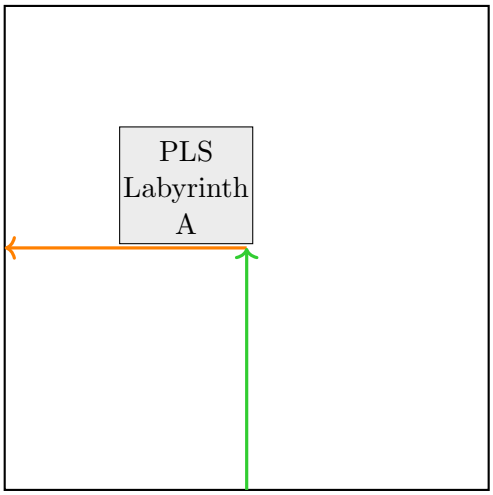


Figure 13: [X] Full construction for a small example, in particular showing the whole boundary. Here $n = 1$ and big squares have size 8×8 (instead of $2^{m+4} \times 2^{m+4}$).

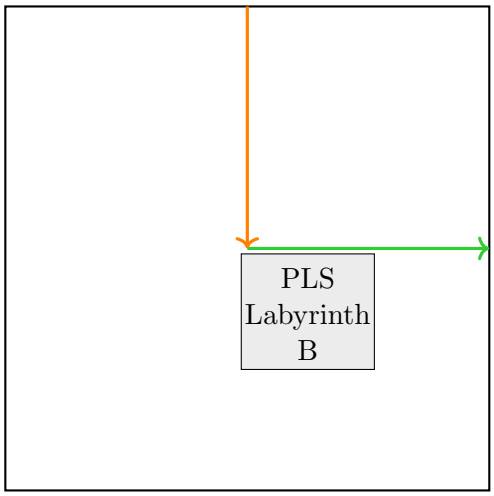
Figure 13 shows the whole construction for a small example where $n = 1$ and big squares have size 8×8 (instead of $2^{m+4} \times 2^{m+4}$).

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. Figures 14(a) and 14(b) show where this PLS-Labyrinth gadget is positioned inside a big square of type LA (namely when case A above occurs) and a big square of type LB (namely when case B above occurs) respectively. 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.

In our construction, every big square is of type G1, G2, ..., G7, O1, O2, ..., O7, E1, E2, S, LA or LB. Note that we can efficiently determine the type of a given big square, if we have access to the End-of-Line circuits S and P.



(a) [LA] Position of PLS-Labyrinth gadget in big square of type LA.



(b) [LB] Position of PLS-Labyrinth gadget in big square of type LB.

Figure 14: Position of PLS-Labyrinth gadget in big squares of type LA and LB.

4.1.4 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 $v_2 < v$. In particular, $B(v, v)$ is of type LA. The PLS-Labyrinth gadget has size $2^{m+2} \times 2^{m+2}$ small squares and is positioned in the big square $B(v, v)$ as shown in Figure 14(a). 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 15 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

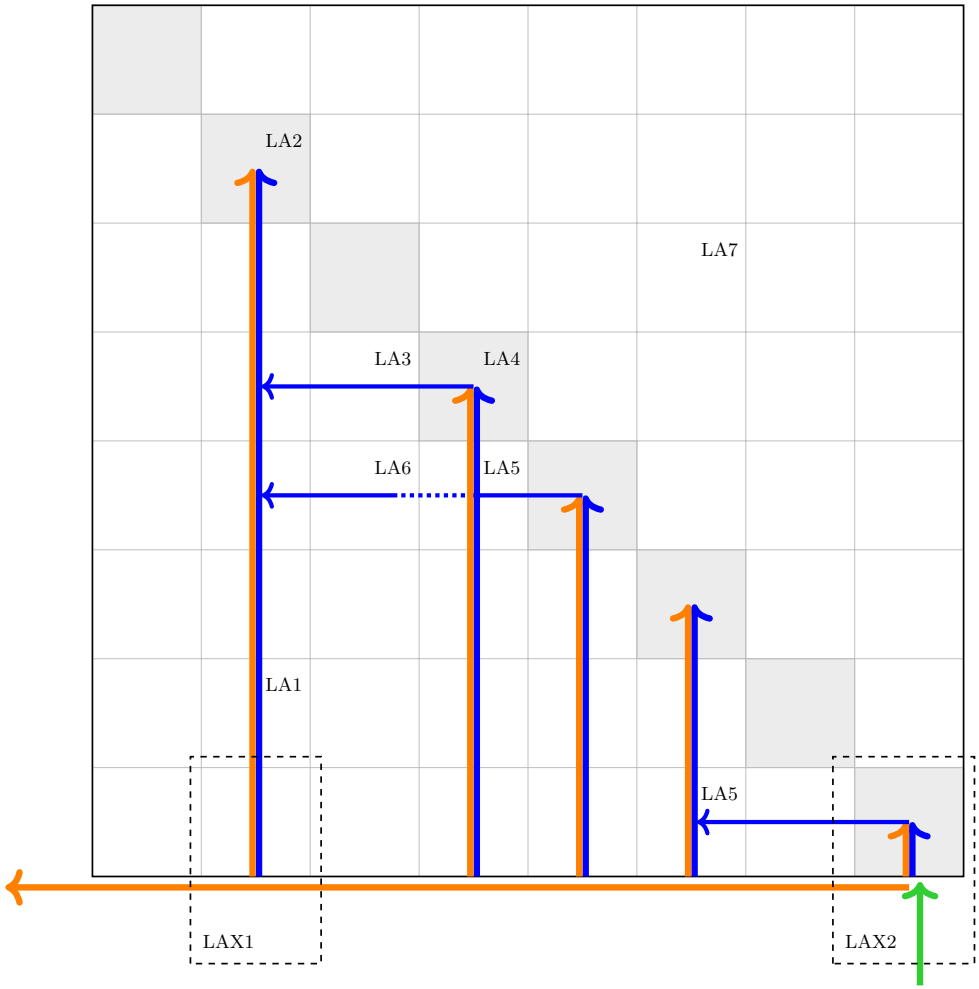


Figure 15: Map of the PLS-Labyrinth for case A corresponding to the ITER example of Figure 2. Shaded squares are the medium squares corresponding to the nodes of ITER. The horizontal blue lines (pointing left) correspond to the 3 edges in Figure 2 that go out from non-solutions, and we do not use similar lines going out from solutions (nodes 3 and 7). We have also indicated the parts LA1-LA6, and LAX1-LAX2, that are constructed in Figure 16.

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 16(a) shows a medium square that is being traversed by the orange-blue path, i.e., a medium square $M(u,w)$ where $w < u$. We say that such a medium square $M(u,w)$ is of type LA1. When the orange-blue path reaches $M(u,u)$, it either "turns" to the left and creates the beginning of a blue path (medium square of type LA4, Figure 16(d)), or it just stops there (medium square of type LA2, Figure 16(b)). 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 it is acceptable for a medium square of type LA2 to contain a KKT point.

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 as shown in Figure 16(g). 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. Figure 16(h) shows how the intersection is implemented and how exactly it is adjacent to $M(1, 1)$. Note that $M(1, 1)$ is just a standard "turn", i.e., a medium square of type LA4.

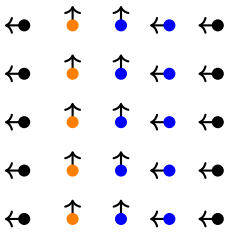
Blue paths. A blue path starts in $M(u, u)$ for some $u \in [2^m]$ such that $C(u) > u$ and $C(C(u)) > C(u)$. It moves from right to left and has a width of two small squares. All three points on the path are coloured blue and the direction of steepest descent points to the left. Figure 16(c) shows a medium square traversed by a blue path. Such a medium square is said to be of type LA3. As mentioned above, the blue path starts at $M(u, u)$ which of type LA4 (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 straightforward and is implemented as shown in Figure 16(f). The medium square $M(C(u), u)$ is then said to be of type LA5.

Crossings. Note that two orange-blue paths cannot cross, and similarly two blue paths can also not cross. However, a blue path going from $M(u, u)$ to $M(C(u), u)$ can cross many other orange-blue paths, before it reaches and merges into its intended orange-blue path. Fortunately, these crossings are much easier to resolve than earlier. Indeed, when a blue path is supposed to cross an orange-blue path, it can simply merge into it and restart on the other side. The important thing to note here is that, while a blue path cannot stop in the environment (without creating a KKT point), it can *start* in the environment. Figure 16(e) shows how this is implemented. In particular, we use a medium square of type LA5 for the merging, and a medium square of type LA6 for the re-start of the blue path.

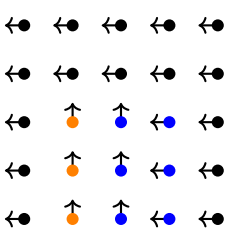
Note that if the blue path has to cross more than one orange-blue path in immediate succession, then it will simply merge into the first one it meets, and restart after the last one (i.e., as soon as it reaches a medium square that is not traversed by an orange-blue path).

Finally, we say that a medium square is of type LA7, if it does not contain any path at all. Medium squares of type LA7 are like the environment, i.e., all the grid points are coloured black and the arrows of steepest descent point to the left. In our construction, every medium square in the PLS-Labyrinth gadget is of type LA1, LA2, ..., LA6, or LA7. It is easy to check that the type of a given medium square can be determined efficiently, given access to the ITER circuit C.

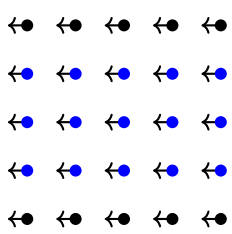
The PLS-Labyrinth gadget for case B is, in a certain sense, symmetric to the one presented above. Indeed, it suffices to perform a point reflection (in other words, a rotation by 180 degrees) with respect to the centre of $B(v, v)$, and a very simple transformation of the colours. With regards to the final interpolated function, this corresponds to rotating $B(v, v)$ by 180 degrees around its centre and multiplying the output of the function by -1. Let $\phi : B(v, v) \to B(v, v)$ denote rotation by 180 degrees around the centre of $B(v, v)$. Then, the direction of steepest descent at some grid point $(x, y) \in B(v, v)$ in case B is



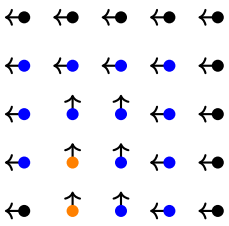
(a) [LA1] Orange-blue path traversing medium square.



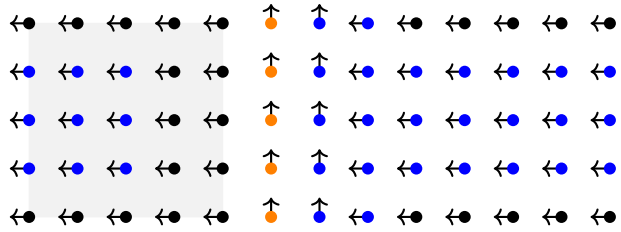
(b) [LA2] Orange-blue path ending in medium square.



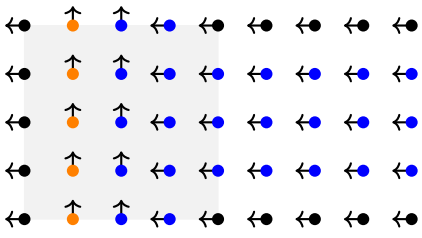
(c) [LA3] Blue path traversing medium square.



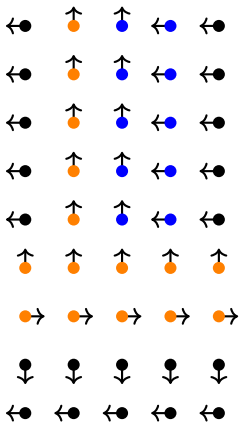
(d) [LA4] Orange-blue path turning in medium square and creating start of blue path.



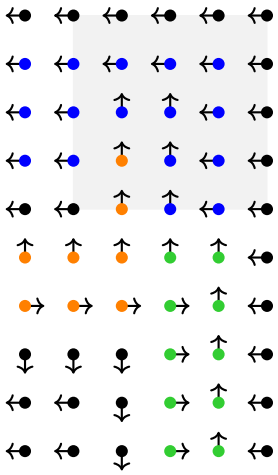
(e) [LA6] Blue path crossing over orange-blue path. Medium square of type LA6 indicated in grey.



(f) [LA5] Blue path merging into orange-blue path. Medium square of type LA5 indicated in grey.



(g) [LAX1] Start of orange-blue path, adjacent to orange path.



(h) [LAX2] Start of orange-blue path for $u = 1$, adjacent to the intersection of green and orange path. $M(1, 1)$ is indicated in grey.

Figure 16: Construction of blue and orange-blue paths in the PLS-Labyrinth gadget inside a big square of type LA.

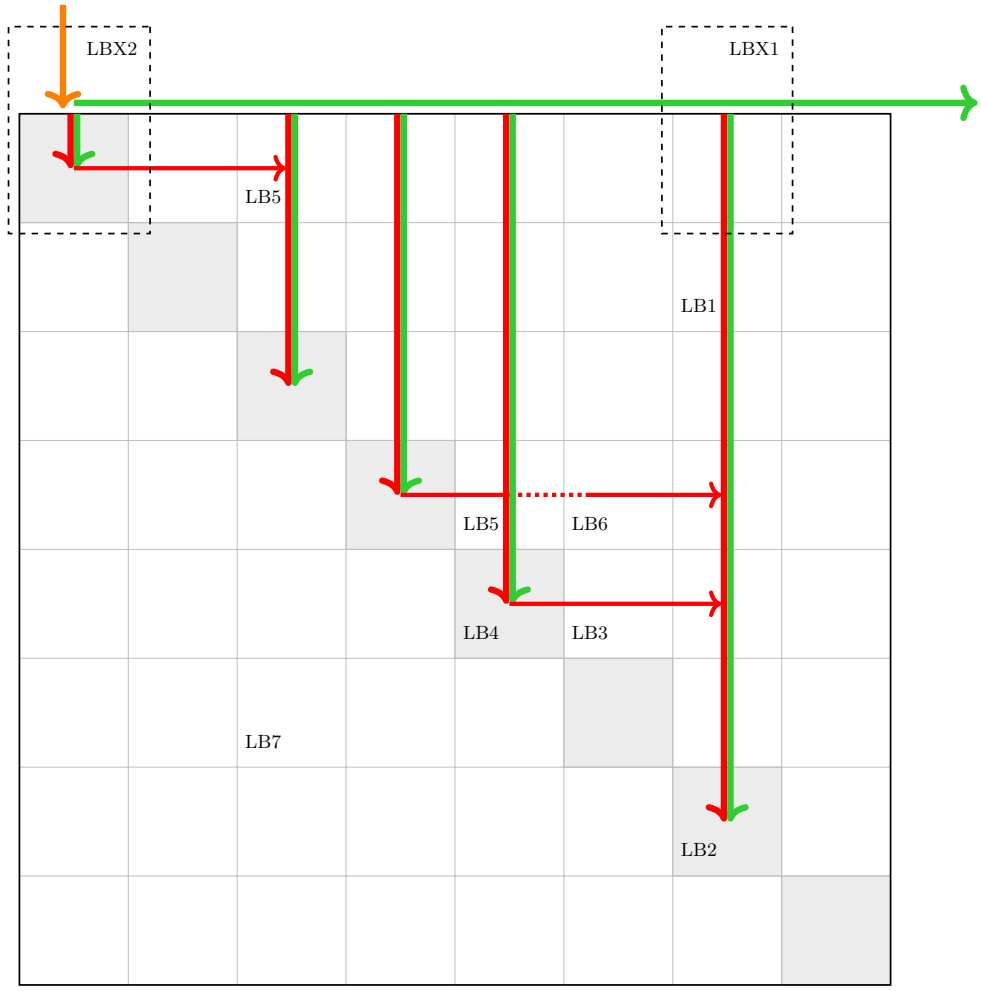


Figure 17: Map of the PLS-Labyrinth for case B corresponding to the ITER example of Figure 2. Shaded squares are the medium squares corresponding to the nodes of ITER. We have also indicated the parts LB1-LB6, and LBX1-LBX2, that are constructed in Figure 18.

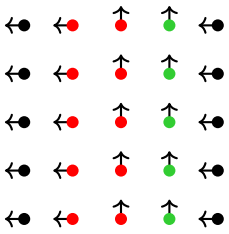
simply the same as the direction of steepest descent at $\phi(x, y)$ in case A. The colour of (x, y) in case B is obtained from the colour of $\phi(x, y)$ in case A as follows:

- black remains black,
- green becomes orange, and vice-versa,
- blue becomes red, and vice-versa.

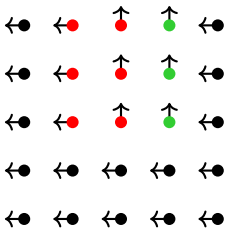
Figure 17 shows a high-level overview of the PLS-Labyrinth gadget for case B. We obtain corresponding medium squares of type LB1, LB2, ..., LB7. The analogous illustrations for case B are shown in Figure 18.

4.2 Extending the Function to the Rest of the Domain

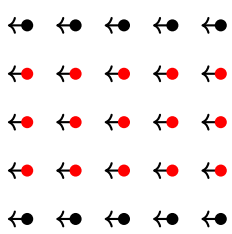
Up to this point we have defined the function f and the direction of its gradient at all grid points of G. In order to extend f to the whole domain $[0, N]^2$, we use *bicubic*



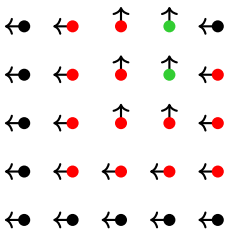
(a) [LB1] Red-green path traversing medium square.



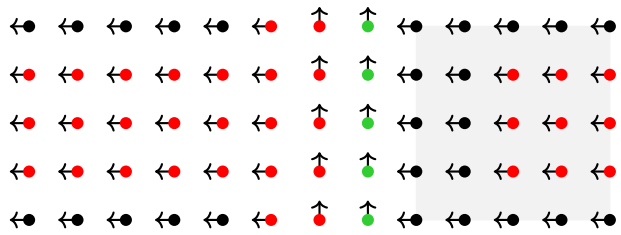
(b) [LB2] Red-green path ending in medium square.



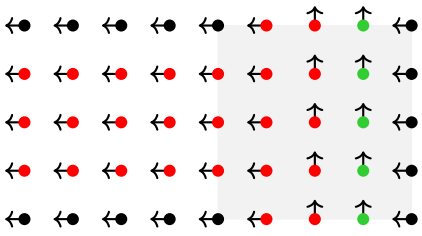
(c) [LB3] Red path traversing medium square.



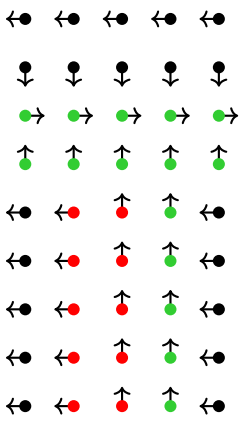
(d) [LB4] Red-green path turning in medium square and creating start of red path.



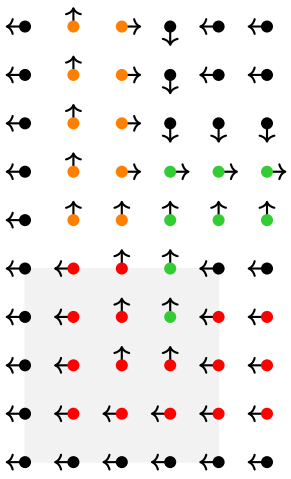
(e) [LB6] Red path crossing over red-green path. Medium square of type LB6 indicated in grey.



(f) [LB5] Red path merging into red-green path. Medium square of type LB5 indicated in grey.



(g) [LBX1] Start of red-green path, adjacent to green path.



(h) [LBX2] Start of red-green path for $u = 1$, adjacent to the intersection of orange and green path. $M(1, 1)$ is indicated in grey.

Figure 18: Construction of red and red-green paths in the PLS-Labyrinth gadget inside a big square of type LB.

interpolation (see e.g. [Russell, 1995] or the corresponding Wikipedia article[6]). Note that the more standard and simpler *bilinear* interpolation (used in particular by Hubáček and Yogev [2017]) 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:

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

where the coefficients a_{ij} are computed as follows

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \tag{4}$$
$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -3 & 3 & -2 & -1 \\ 2 & -2 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} f(0,0) & f(0,1) & f_y(0,0) & f_y(0,1) \\ f(1,0) & f(1,1) & f_y(1,0) & f_y(1,1) \\ f_x(0,0) & f_x(0,1) & f_{xy}(0,0) & f_{xy}(0,1) \\ f_x(1,0) & f_x(1,1) & f_{xy}(1,0) & f_{xy}(1,1) \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -3 & 2 \\ 0 & 0 & 3 & -2 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & -1 & 1 \end{bmatrix}$$

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 $f : [0, N]^2 \to \mathbb{R}$. In fact, we can even extend the function to points $(x, y) \in \mathbb{R}^2 \setminus [0, N]^2$ by simply using the interpolated polynomial obtained for the small square that is closest to (x, y). This will be done automatically by our construction of the arithmetic circuit computing f and it will ensure that the gradient is well-defined even on the boundary of $[0, N]^2$.

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

- *it is continuously differentiable on $\mathbb{R}^2$,*
- *f and its gradient ∇f are Lipschitz-continuous on $[0, N]^2$ with Lipschitz-constant $L = 2^{18} N$,*

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

- *well-behaved arithmetic circuits computing f and ∇f can be constructed in polynomial time (in the size of the circuits S, P and C).*

Proof. Regarding the first point, see, e.g., [Russell, 1995].

Lipschitz-continuity. In order to prove the second point, we first show that f and ∇f are L-Lipschitz-continuous in every small square of the grid. Consider any small square. In our construction, the values of f, f_x, f_y, f_{xy} used in the computation of the coefficients a_{ij} are clearly all upper bounded by $2^3 N$ in absolute value. Thus, using Equation (4), it is easy to check that $|a_{ij}| \leq 2^{10} N$ for all $i, j \in \{0, 1, 2, 3\}$. Furthermore, note that the partial derivatives of f inside the small square can be expressed as:

$$\frac{\partial f}{\partial x}(x, y) = \sum_{i=1}^{3} \sum_{j=0}^{3} i \cdot a_{ij} x^{i-1} y^j \qquad \frac{\partial f}{\partial y}(x, y) = \sum_{i=0}^{3} \sum_{j=1}^{3} j \cdot a_{ij} x^i y^{j-1} \tag{5}$$

using the local coordinates $(x, y) \in [0, 1]^2$ inside the small square. Finally, it is easy to check that the monomials $x^i y^j$, $i, j \in \{0, 1, 2, 3\}$, are all 6-Lipschitz continuous over $[0, 1]^2$. Putting everything together and using Equation (3) and Equation (5), it follows that f and ∇f are Lipschitz-continuous (w.r.t. the ℓ_2-norm) with Lipschitz constant $L = 2^{18} N$ inside the small square. Note that the change from local coordinates to standard coordinates is just a very simple translation that does not impact the Lipschitzness of the functions.

Since f and ∇f are L-Lipschitz-continuous inside every small square and continuous over all of $[0, N]^2$, it follows that they are in fact L-Lipschitz-continuous over the whole domain $[0, N]^2$. Indeed, consider any points $z_1, z_2 \in [0, N]^2$. Then, there exists $\ell \in \mathbb{N}$ such that the segment $z_1 z_2$ can be subdivided into $z_1 w_1 w_2 \dots w_\ell z_2$ so that each of the segments $z_1 w_1$, $w_1 w_2, \dots$, $w_{\ell-1} w_\ell$, $w_\ell z_2$ lies within a small square. For ease of notation, we let $w_0 := z_1$ and $w_{\ell+1} := z_2$. Then, we can write

$$\|\nabla f(z_1) - \nabla f(z_2)\| \leq \sum_{i=0}^{\ell} \|\nabla f(w_i) - \nabla f(w_{i+1})\| \leq L \sum_{i=0}^{\ell} \|w_i - w_{i+1}\| = L \|z_1 - z_2\|$$

where we used the fact that $\sum_{i=0}^{\ell} \|w_i - w_{i+1}\| = \|z_1 - z_2\|$, since $w_0 w_1 w_2 \dots w_\ell w_{\ell+1}$ is just a partition of the segment $z_1 z_2$. The exact same argument also works for f.

Arithmetic circuits. Before showing how to construct the arithmetic circuits for f and ∇f, we first construct a Boolean circuit B that will be used as a sub-routine. The Boolean circuit B receives as input a point (x, y) on the grid $G = \{0, 1, \dots, N\}^2$ and outputs the colour (i.e., value regime) and steepest descent arrow at that point. It is not too hard to see that the circuit B can be constructed in time that is polynomial in the sizes of the circuits S, P and C. In more detail, it performs the following operations:

1. Compute End-of-Line-vertices $v_1, v_2 \in [2^n]$ such that (x, y) lies in the big square $B(v_1, v_2)$.

2. Using the End-of-Line circuits S and P determine the exact type of $B(v_1, v_2)$, namely one of the following: G1-G7, O1-O7, E1, E2, S, LA or LB.

3. If the type of $B(v_1, v_2)$ is not LA or LB, then we know the exact structure of $B(v_1, v_2)$ and can easily return the colour and arrow at (x, y).

4. If the type of $B(v_1, v_2)$ is LA or LB, then first determine whether (x, y) lies in the PLS-Labyrinth inside $B(v_1, v_2)$ or not.

5. If (x, y) does not lie in the PLS-Labyrinth, then we can easily determine the colour and arrow at (x, y), since we know the exact structure of $B(v_1, v_2)$ except the inside of the PLS-Labyrinth.

6. If (x, y) lies in the PLS-Labyrinth, then we can compute ITER-vertices $u_1, u_2 \in [2^m]$ such that (x, y) lies in the medium square $M(u_1, u_2)$ of the PLS-Labyrinth inside $B(v_1, v_2)$.

7. Using the ITER circuit C determine the type of $M(u_1, u_2)$, namely one of the following: LA1-LA7, LB1-LB7. Given the type of $M(u_1, u_2)$, we then know the exact structure of $M(u_1, u_2)$ and can in particular determine the colour and arrow at (x, y).

The arithmetic circuits for f and ∇f are then constructed to perform the following operations on input $(x, y) \in [0, N]^2$:

1. Using the comparison gate $<$ and binary search, compute the bits representing $(\widehat{x}, \widehat{y}) \in \{0, 1, \ldots, N-1\}^2$: a grid point such that (x, y) lies in the small square that has $(\widehat{x}, \widehat{y})$ as its bottom left corner.

2. Simulate the Boolean circuit B using arithmetic gates to compute (a bit representation) of the colour and arrow at the four corners of the small square, namely $(\widehat{x}, \widehat{y})$, $(\widehat{x}+1, \widehat{y})$,$(\widehat{x}, \widehat{y}+1)$ and $(\widehat{x}+1, \widehat{y}+1)$.

3. Using this information and the formulas for the value regimes, compute the 16 terms for f, f_x, f_y and f_{xy} needed to determine the bicubic interpolation. Then, compute the coefficients a_{ij} by performing the matrix multiplication in Equation (4).

4. In the arithmetic circuit for f, apply Equation (3) to compute the value of $f(x, y)$. In the arithmetic circuit for ∇f, apply Equation (5) to compute the value of $\nabla f(x, y)$. Note that in the interpolation Equations (3) and (5), we have to use the local coordinates $(x - \widehat{x}, y - \widehat{y}) \in [0, 1]^2$ instead of (x, y).

The two arithmetic circuits can be computed in polynomial time in n, m and in the sizes of S, P, C. Since n and m are upper bounded by the sizes of S and C respectively, they can be constructed in polynomial time in the sizes of S, P, C. Furthermore, note that the two circuits are well-behaved. In fact, they only use a *constant* number of true multiplication gates. To see this, note that true multiplication gates are only used for the matrix multiplication in step 3 and for step 4. In particular, steps 1 and 2 do not need to use any true multiplication gates at all (see, e.g., [Daskalakis et al., 2009; Chen et al., 2009b]). □

4.3 Correctness

To show the correctness of the construction, we need to show 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 a 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 S, P, 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 $C(C(u)) = C(u)$.*

Note that we have defined ε-KKT points in Section 3.2.1 with respect to the ℓ_2-norm, but here it is more convenient to consider the ℓ_∞-norm instead. Note that any ε-KKT point w.r.t. the ℓ_2-norm is also an ε-KKT point w.r.t. the ℓ_∞-norm. Thus, if Lemma 4.3 holds for ε-KKT points w.r.t. the ℓ_∞-norm, then it automatically also holds for ε-KKT points w.r.t. the ℓ_2-norm.

For the domain $[0, N]^2$, it is easy to see that a point $x \in [0, N]^2$ is an ε-KKT point (with respect to the ℓ_∞-norm) if and only if

- for all $i \in \{1, 2\}$ with $x_i \neq 0$: $[\nabla f(x)]_i \leq \varepsilon$
- for all $i \in \{1, 2\}$ with $x_i \neq N$: $-[\nabla f(x)]_i \leq \varepsilon$.

Intuitively, these conditions state that if x is not on the boundary of $[0, N]^2$, then it must hold that $\|\nabla f(x)\|_\infty \leq \varepsilon$. If x is on the boundary of $[0, 1]^2$, then "$-\nabla f(x)$ must point straight outside the domain, up to an error of ε".

In order to prove Lemma 4.3, we will show that any small square that does not lie in a solution region, does not contain any ε-KKT point. 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. Conceptually, this is a straightforward task: for each small square we get a set of cubic polynomials that could be generated by bicubic interpolation for that square, and we must prove that no polynomial in that set has an ε-KKT point within the square.

Unfortunately, there are 101 distinct small squares[7] used by our construction. So, while we could write a proof for every single case, this would likely be an error-prone task, and the result would be tens of pages of straightforward proofs that would be tedious to verify.

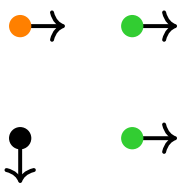
To avoid this, we instead delegate the proof to an SMT solver. For each square we write down an SMT formula that encodes "there exists an ε-KKT point within this square". The SMT solver then checks this formula for satisfiability over the algebraic real numbers[8] and if the formula is found to be unsatisfiable, then Lemma 4.3 has been proven for that square. We applied the Z3 SMT solver [de Moura and Bjørner, 2008] to all 101 squares in our construction[9], and we found that the formula is unsatisfiable for every square that does not lie directly at the end of a line of the End-of-Line instance, or at a solution of the Iter instance, which proves Lemma 4.3.

The SMT formula. We now give some further details on how the SMT formula is constructed. The full annotated source code for building the SMT formula can be found in Appendix F of the appendix. As a running example, we will use the following square, which appears in the gadget shown in Figure 16(h).

[7] This number could likely be cut down by considering symmetries, but there will still be a large number of squares remaining even after accounting for this.

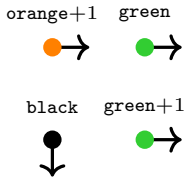
[8] SMT solvers are capable of deciding satisfiability for the existential theory of the reals. There are no rounding issues or floating point errors to worry about.

[9] The full output is available at https://github.com/jfearnley/PPADPLS/blob/main/report/report_static.pdf



As described in our construction, the value of f at each point is determined by the color of that point. Since this square could appear in multiple places, we do not know the absolute values for f at any of the points, but we do have information about the relative values. For example, we know that for green points, the value increases as we move down and to the left. So we can introduce a symbolic variable `green` to represent the value of the green function at the top-right point of the square, and then we know that the value of f at the top-right point is `green` and the value of f at the bottom-right point is $\texttt{green} + 1$.

The orange function also increases as we go down and left, so we can introduce a new symbolic variable `orange` to represent the value of the orange function at the top-right point, and then the value of f at the top-left point is $\texttt{orange} + 1$. The black function increases as we go up or right, so for this color we introduce a symbolic variable `black` to represent the value of the function at the bottom-left point of the square, and so the value of f at the bottom-left point of the square is simply `black`. To summarize, we obtain the following symbolic values for f on our square.



We also know from our construction that at any given grid point, the orange value regime has a higher value than the black regime, which has a higher value than the green regime. Furthermore, by construction it is guaranteed that the gap between any two distinct regimes at any given point is at least 10. Since any regime can change by at most 2 over the square, we get the following inequalities[10] $\texttt{orange} > \texttt{black} + 4$ and $\texttt{black} > \texttt{green} + 4$.

Given these symbolic values for the corners of the square, we can then introduce two symbolic values `x` and `y`, and then carry out bicubic interpolation symbolically using the formula given in (3), which yields a polynomial for f inside the square in terms of `orange`, `black`, `green`, `x`, and `y`. We can then symbolically differentiate this polynomial to get polynomials for f_x and f_y.

Finally, we write down the following formula

$$\begin{aligned}\forall\ \texttt{orange}, \texttt{black}, \texttt{green}, \texttt{x}, \texttt{y}\ \cdot\ & \\ & \texttt{orange} > \texttt{black} + 4 \\ \wedge\ & \texttt{black} > \texttt{green} + 4 \\ \wedge\ & \texttt{x} \in [0,1]\ \wedge\ \texttt{y} \in [0,1] \\ \Rightarrow\ & f_x(x,y) \notin [-\varepsilon, \varepsilon]\ \vee\ f_y(x,y) \notin [-\varepsilon, \varepsilon],\end{aligned}$$

which states that there is no ε-stationary point in the square, by insisting that either f_x or f_y is not ε-close to zero. Note that this is equivalent to saying that there is no ε-KKT

[10] In fact we get $\texttt{orange} > \texttt{black} + 7$ and so on, but the solver is able to verify the lemma for these weaker inequalities

point in the square, if this square is not positioned on the boundary of the instance. Since we universally quantify over `orange`, `black`, and `green`, the formula states that no ε-KKT points exist no matter what particular values f takes at the corners. So, if we ask the SMT solver to satisfy the negation of this formula, and if it declares the negation to be unsatisfiable, then Lemma 4.3 has been proven for the square.

For other squares, we likewise introduce symbolic variables for the colors red and blue, and we follow a similar approach. See Appendix F in the appendix for the annotated source code of the proof.

Boundary squares. For squares that lie on the boundary of the instance, there is an additional property to check to ensure that they do not contain any ε-KKT points: the direction of steepest descent for a point on the boundary should not point straight outside the domain. An example of the boundary of the instance is shown in Figure 13.

There are 12 distinct small squares that can lie on the boundary of the instance. Using the same machinery that we developed for the main proof, we can also write down an SMT formula to verify the squares on the boundary of the instance.

If our example square lay on the left boundary of the instance, we would use the following formula

$$\begin{aligned}\forall\ \texttt{orange}, \texttt{black}, \texttt{green}, \texttt{x}, \texttt{y}\ \cdot\ & \\ & \texttt{orange} > \texttt{black} + 4 \\ \wedge\ & \texttt{black} > \texttt{green} + 4 \\ \wedge\ & \texttt{x} = 0\ \wedge\ \texttt{y} \in [0,1] \\ \Rightarrow\ & f_x(x,y) < -\varepsilon\ \vee\ f_y(x,y) \notin [-\varepsilon, \varepsilon],\end{aligned}$$

which is similar to the main formula, but with two changes. Firstly we only care about the case where $x = 0$ since this corresponds to the left-hand border of the square, and secondly we insist that either f_y is not ε-close to 0, or that $f_x(x,y) < -\varepsilon$, so either the y-gradient ensures that we are not at an ε-stationary point with respect to y, or the x-gradient ensures that the direction of steepest descent according to the x-gradient points into the instance. These conditions exactly check that the point is not an ε-KKT point.

We write similar formulas for the other boundaries, and then check each square on the boundary using these formulas. Further details can be found in Appendix F of the appendix. Note that we do not need to use the SMT solver to check whether any of the four corners of the domain is a solution. These four points are grid points and thus the gradient there is directly given by the corresponding arrow in our construction. It is straightforward to directly check that none of the four corners of the instance is an ε-KKT point, because the arrows point inside the domain and have length $\delta = 1/2 > \varepsilon$. The SMT solver verified that no ε-KKT points lie anywhere else on the boundary, which completes the proof of Lemma 4.3.

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 $[0,N]^2$. Thus, the final function, which we denote $\widehat{f}$ here, is defined by

$$\widehat{f}(x,y) = (1/N) \cdot f(N \cdot x, N \cdot y).$$

The properties of f proved in Lemma 4.2 naturally also hold for $\widehat{f}$, in the following sense. Clearly, $\widehat{f}$ is also continuously differentiable. Furthermore, it holds that $\nabla\widehat{f}(x,y) =$

$\nabla f(N \cdot x, N \cdot y)$. Thus, we can easily construct well-behaved arithmetic circuits for $\widehat{f}$ and $\nabla \widehat{f}$ in polynomial time given well-behaved circuits for f and ∇f, which, in turn, can be efficiently constructed according to Lemma 4.2. Furthermore, since ∇f is L-Lipschitz-continuous, it is easy to see that $\nabla \widehat{f}$ is $\widehat{L}$-Lipschitz-continuous with $\widehat{L} = N \cdot L = 2^{18} N^2 = 2^{2n+2m+26}$. Finally, note that since f is L-Lipschitz-continuous, $\widehat{f}$ is too, and in particular it is also $\widehat{L}$-Lipschitz-continuous.

All these properties imply that the instance of KKT we construct does not admit any violation solutions. In other words, it satisfies all the expected promises. Finally, note that any ε-KKT point of $\widehat{f}$ on $[0,1]^2$ immediately yields an ε-KKT point of f on $[0,N]^2$. Thus, the correctness of the reduction follows from Lemma 4.3.

Note that we can re-scale the instance depending on the parameter regime we are interested in. The instance $(\varepsilon, \widehat{f}, \nabla \widehat{f}, \widehat{L})$ we have constructed, is clearly equivalent to the instance $(\alpha\varepsilon, \alpha\widehat{f}, \nabla(\alpha\widehat{f}), \alpha\widehat{L})$ for any $\alpha > 0$. For example, by letting $\alpha = 1/\widehat{L}$, we 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 show how the PPAD ∩ PLS-hardness of KKT (Theorem 4.1) implies that our problems of interest, including our Gradient Descent problems, are PPAD ∩ PLS-complete. Namely, we prove:

Theorem 5.1. *The problems* KKT, GD-Local-Search, GD-Fixpoint *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 (Lemma C.3).

- 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, this instance can be transformed into one where ε is exponentially small and L is constant (see Section 4.4). 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 $[0,1]^n$ (Lemma C.4). 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, proved in Section 4, and a set of domain- and promise-preserving reductions as pictured in Figure 3, which are presented in the rest of this section as follows. In Section 5.1 we show that the problems KKT, GD-Local-Search and GD-Fixpoint are equivalent. Then, in Section 5.2 we reduce GD-Local-Search to General-Continuous-Localopt, and finally we show that General-Continuous-Localopt lies in PPAD ∩ PLS.

5.1 KKT and the Gradient Descent problems are equivalent

The equivalence between KKT, GD-Local-Search and GD-Fixpoint is proved by providing a "triangle" of reductions as shown in Figure 3. Namely, we show that GD-Local-Search reduces to GD-Fixpoint (Proposition 5.2), GD-Fixpoint reduces to KKT (Proposition 5.3), and KKT reduces to GD-Local-Search (Proposition 5.4). All the reductions are domain- and promise-preserving.

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

Proof. Let $(\varepsilon, \eta, A, b, f, \nabla f, L)$ be an instance of GD-Local-Search. The reduction simply constructs the instance $(\varepsilon', \eta, A, b, f, \nabla f, L)$ of GD-Fixpoint, where $\varepsilon' = \varepsilon/L$. This reduction is trivially domain-preserving and it is also promise-preserving, because any violation of the constructed instance is immediately also a violation of the original instance. Clearly, the reduction can be computed in polynomial time, so it remains to show that any (non-violation) solution of the constructed instance can be mapped back to a solution or violation of the original instance.

Consider any solution $x \in D$ of the GD-Fixpoint instance, i.e.,

$$\|x - y\| = \|x - \Pi_D(x - \eta \nabla f(x))\| \leq \varepsilon'.$$

where $y = \Pi_D(x - \eta \nabla f(x))$. If x, y do not satisfy the L-Lipschitzness of f, then we have obtained a violation. Otherwise, it must be that

$$|f(x) - f(y)| \leq L\|x - y\| \leq L\varepsilon' = \varepsilon.$$

In particular, it follows that

$$f(y) \geq f(x) - \varepsilon$$

which means that x is a solution of the original GD-Local-Search instance. □

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

Proof. Let $(\varepsilon, \eta, A, b, f, \nabla f, L)$ be an instance of GD-Fixpoint. The reduction simply constructs the instance $(\varepsilon', A, b, f, \nabla f, L)$ of KKT, where $\varepsilon' = \varepsilon/\eta$. This reduction is trivially domain-preserving and it is also promise-preserving, because any violation of the constructed instance is immediately also a violation of the original instance. Clearly, the reduction can be computed in polynomial time, so it remains to show that any (non-violation) solution of the constructed instance can be mapped back to a solution or violation of the original instance.

In more detail, we will show that any ε'-KKT point must be an ε-approximate fixed point of the gradient descent dynamics. Consider any ε'-KKT point of the KKT instance, i.e., a point $x \in D$ such that there exists $\mu \geq 0$ with $\langle \mu, Ax - b \rangle = 0$ and $\|\nabla f(x) + A^T \mu\| \leq \varepsilon'$.

Let $y = \Pi_D(x - \eta \nabla f(x))$. We want to show that $\|x - y\| \leq \varepsilon$. Since y is the projection of $x - \eta \nabla f(x)$ onto D, by Lemma B.1 it follows that for all $z \in D$

$$\langle x - \eta \nabla f(x) - y, z - y \rangle \leq 0.$$

Letting $z := x$, this implies that

$$\begin{aligned} \|x - y\|^2 \leq \eta \langle \nabla f(x), x - y \rangle &= \eta \langle \nabla f(x) + A^T \mu, x - y \rangle - \eta \langle A^T \mu, x - y \rangle \\ &\leq \eta \langle \nabla f(x) + A^T \mu, x - y \rangle \\ &\leq \eta \|\nabla f(x) + A^T \mu\| \cdot \|x - y\| \end{aligned}$$

where we used the Cauchy-Schwarz inequality and the fact that $\langle A^T \mu, x - y \rangle \geq 0$, which follows from

$$\langle A^T \mu, x - y \rangle = \langle \mu, A(x - y) \rangle = \langle \mu, Ax - b \rangle - \langle \mu, Ay - b \rangle \geq 0$$

since $\langle \mu, Ax - b \rangle = 0$, $\mu \geq 0$ and $Ay - b \leq 0$ (because $y \in D$).

We can now show that $\|x - y\| \leq \varepsilon$. If $\|x - y\| = 0$, this trivially holds. Otherwise, divide both sides of the inequality obtained above by $\|x - y\|$, which yields

$$\|x - y\| \leq \eta \|\nabla f(x) + A^T \mu\| \leq \eta \cdot \varepsilon' = \varepsilon.$$

□

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

Proof. Let $(\varepsilon, A, b, f, \nabla f, L)$ be an instance of KKT. The reduction simply constructs the instance $(\varepsilon', \eta, A, b, f, \nabla f, L)$ of GD-Local-Search, where $\varepsilon' = \frac{\varepsilon^2}{8L}$ and $\eta = \frac{1}{L}$. This reduction is trivially domain-preserving and it is also promise-preserving, because any violation of the constructed instance is immediately also a violation of the original instance. Clearly, the reduction can be computed in polynomial time, so it remains to show that any (non-violation) solution of the constructed instance can be mapped back to a solution or violation of the original instance.

Consider any $x \in D$ that is a solution of the GD-Local-Search instance and let $y = \Pi_D(x - \eta \nabla f(x))$. Then, it must be that $f(y) \geq f(x) - \varepsilon'$. We begin by showing that this implies that $\|x - y\| \leq \frac{\varepsilon}{2L}$, or we can find a violation of the KKT instance.

Step 1: Bounding $\|x - y\|$. If x and y do not satisfy Taylor's theorem (Lemma 3.4), then we immediately obtain a violation. If they do satisfy Taylor's theorem, it holds that

$$\langle \nabla f(x), x - y \rangle - \frac{L}{2} \|y - x\|^2 \leq f(x) - f(y) \leq \varepsilon'.$$

Now, since y is the projection of $x - \eta \nabla f(x)$ onto D, by Lemma B.1 it follows that $\langle x - \eta \nabla f(x) - y, z - y \rangle \leq 0$ for all $z \in D$. In particular, by letting $z := x$, we obtain that

$$\langle \nabla f(x), x - y \rangle \geq \frac{1}{\eta} \langle x - y, x - y \rangle = L \|y - x\|^2$$

where we used the fact that $\eta = 1/L$. Putting the two expressions together we obtain that

$$\frac{L}{2} \|y - x\|^2 = L \|y - x\|^2 - \frac{L}{2} \|y - x\|^2 \leq \varepsilon'$$

which yields that $\|x - y\| \leq \sqrt{2\varepsilon'/L} = \frac{\varepsilon}{2L}$.

Step 2: Obtaining an ε-KKT point. Next, we show how to obtain an ε-KKT point or a violation of the KKT instance. Note that if $y - x = -\eta\nabla f(x)$, then we immediately have that $\|\nabla f(x)\| = \|x - y\|/\eta \leq \varepsilon/2$, i.e., x is an ε-KKT point. However, because of the projection Π_D used in the computation of y, in general we might not have that $y - x = -\eta\nabla f(x)$ and, most importantly, x might not be an ε-KKT point. Nevertheless, we show that y will necessarily be an ε-KKT point.

Since y is the projection of $x - \eta\nabla f(x)$ onto D, by Lemma B.1 it follows that for all $z \in D$

$$\langle x - \eta\nabla f(x) - y, z - y\rangle \leq 0.$$

From this it follows that for all $z \in D$

$$\langle -\nabla f(x), z - y\rangle \leq \frac{1}{\eta}\langle y - x, z - y\rangle \leq \frac{1}{\eta}\|x - y\| \cdot \|z - y\| \leq \frac{\varepsilon}{2}\|z - y\|$$

where we used the Cauchy-Schwarz inequality, $\eta = 1/L$ and $\|x-y\| \leq \varepsilon/2L$. Next, unless x and y yield a violation to the L-Lipschitzness of ∇f, it must hold that $\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\| \leq \varepsilon/2$. Thus, we obtain that for all $z \in D$

$$\begin{aligned}\langle -\nabla f(y), z - y\rangle &= \langle -\nabla f(x), z - y\rangle + \langle \nabla f(x) - \nabla f(y), z - y\rangle \\ &\leq \frac{\varepsilon}{2}\|z - y\| + \|\nabla f(x) - \nabla f(y)\| \cdot \|z - y\| \\ &\leq \varepsilon\|z - y\|\end{aligned} \tag{6}$$

where we used the Cauchy-Schwarz inequality.

Let $I = \{i \in [m] : [Ay - b]_i = 0\}$, i.e., the indices of the constraints that are tight at y. Denote by $A_I \in \mathbb{R}^{(m-|I|)\times n}$ the matrix obtained by only keeping the rows of A that correspond to indices in I. Consider any $p \in \mathbb{R}^n$ such that $A_I p \leq 0$. Then, there exists a sufficiently small $\alpha > 0$ such that $z = y + \alpha p \in D$. Indeed, note that $[Az - b]_i = [Ay - b]_i + \alpha[Ap]_i$ and thus

- for $i \in I$, we get that $[Az - b]_i \leq 0$, since $[Ay - b]_i = 0$ and $[Ap]_i \leq 0$,
- for $i \notin I$, we have that $[Ay - b]_i < 0$. If $[Ap]_i \leq 0$, then we obtain $[Az - b]_i \leq 0$ as above. If $[Ap]_i > 0$, then it also holds that $[Az - b]_i \leq 0$, as long as $\alpha \leq -\frac{[Ay-b]_i}{[Ap]_i}$.

Thus, it suffices to pick $\alpha = \min\left\{-\frac{[Ay-b]_i}{[Ap]_i} : i \notin I, [Ap]_i > 0\right\}$. Note that this indeed ensures that $\alpha > 0$.

Since $z = y + \alpha p \in D$, using (6) we get that

$$\langle -\nabla f(y), p\rangle = \frac{1}{\alpha}\langle -\nabla f(y), z - y\rangle \leq \frac{\varepsilon}{\alpha}\|z - y\| = \varepsilon\|p\|.$$

As a result, we have shown that the statement

$$\exists p \in \mathbb{R}^n : A_I p \leq 0, \langle -\nabla f(y), p\rangle > \varepsilon\|p\|$$

does not hold. By a stronger version of Farkas' Lemma, which we prove in the appendix (Lemma B.3), it follows that there exists $\nu \in \mathbb{R}^{|I|}_{\geq 0}$ such that $\|A_I^T\nu + \nabla f(y)\| \leq \varepsilon$. Let $\mu \in \mathbb{R}^m_{\geq 0}$ be such that μ agrees with ν on indices $i \in I$, i.e., $\mu_I = \nu$, and $\mu_i = 0$ for $i \notin I$. Then we immediately obtain that $A_I^T\nu = A^T\mu$ and thus $\|A^T\mu + \nabla f(y)\| \leq \varepsilon$. Since we also have that $\langle \mu, Ay - b\rangle = \langle \mu_I, [Ay - b]_I\rangle = 0$ (because $[Ay - b]_I = 0$), it follows that y indeed is an ε-KKT point of f on domain D. □

5.2 From GD-Local-Search to PPAD ∩ PLS

In this section we show that GD-Local-Search reduces to General-Continuous-Localopt (Proposition 5.5), and then that General-Continuous-Localopt lies in PPAD ∩ PLS (Proposition 5.6).

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

Proof. This essentially follows from the fact that the local search version of Gradient Descent is a special case of continuous local search, which is captured by the General-Continuous-Localopt problem. Let $(\varepsilon, A, b, \eta, f, \nabla f, L)$ be an instance of GD-Local-Search. The reduction simply constructs the instance $(\varepsilon, A, b, p, g, L')$ of General-Continuous-Localopt, where $p(x) = f(x)$, $g(x) = x - \eta \nabla f(x)$ and $L' = \max\{\eta L + 1, L\}$. We can easily construct an arithmetic circuit computing g, given the arithmetic circuit computing ∇f. It follows that the reduction can be computed in polynomial time. In particular, since we extend ∇f by using only the gates $-$ and $\times\zeta$, the circuit for g is also well-behaved.

Let us now show that any solution to the General-Continuous-Localopt instance yields a solution to the GD-Local-Search instance. First of all, by construction of g, it immediately follows that any local optimum solution of the General-Continuous-Localopt instance is also a non-violation solution to the GD-Local-Search instance.

Next, we show that any pair of points $x, y \in D$ that violate the $(\eta L + 1)$-Lipschitzness of g, also violate the L-Lipschitzness of ∇f. Indeed, if x, y do not violate the L-Lipschitzness of ∇f, then

$$\|g(x) - g(y)\| \leq \|x - y\| + \eta \|\nabla f(x) - \nabla f(y)\| \leq (\eta L + 1)\|x - y\|.$$

In particular, any violation to the L'-Lipschitzness of g yields a violation to the L-Lipschitzness of ∇f.

Finally, any violation to the L'-Lipschitzness of p immediately yields a violation to the L-Lipschitzness of f. Since any violation to General-Continuous-Localopt yields a violation to GD-Local-Search, the reduction is also promise-preserving. □

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

Proof. This essentially follows by the same arguments that were used by Daskalakis and Papadimitriou [2011] to show that CLS lies in PPAD ∩ PLS. The only difference is that here the domain is allowed to be more general. Consider any instance $(\varepsilon, A, b, p, g, L)$ of General-Continuous-Localopt.

The containment of General-Continuous-Localopt in PPAD follows from a reduction to the problem of finding a fixed point guaranteed by Brouwer's fixed point theorem, which is notoriously PPAD-complete. Indeed, let $x^* \in D$ be any ε/L-approximate fixed point of the function $x \mapsto \Pi_D(g(x))$, i.e., such that $\|\Pi_D(g(x^*)) - x^*\| \leq \varepsilon/L$. Then, unless x^* and $\Pi_D(g(x^*))$ yield a violation of L-Lipschitzness of p, it follows that $p(\Pi_D(g(x^*))) \geq p(x^*) - \varepsilon$, i.e., x^* is a solution of the General-Continuous-Localopt instance. Formally, the reduction works by constructing the instance $(\varepsilon', A, b, g, L)$ of General-Brouwer, where $\varepsilon' = \varepsilon/L$. The formal definition of General-Brouwer can be found in Appendix D, where it is also proved that the problem lies in PPAD.

The containment of CLS in PLS was proved by Daskalakis and Papadimitriou [2011] by reducing Continuous-Localopt to a problem called Real-Localopt, which they show to lie in PLS. Real-Localopt is defined exactly as Continuous-Localopt, except

that the function g is not required to be continuous. In order to show the containment of General-Continuous-Localopt in PLS, we reduce to the appropriate generalisation of Real-Localopt, which we simply call General-Real-Localopt. Formally, the reduction is completely trivial, since any instance of General-Continuous-Localopt is also an instance of General-Real-Localopt, and solutions can be mapped back as is. The formal definition of General-Real-Localopt can be found in Appendix D, where it is also proved that the problem lies in PLS. □

6 Consequences for Continuous Local Search

In this section, we explore the consequences of Theorem 4.1 (and Theorem 5.1) for the class CLS, defined by Daskalakis and Papadimitriou [2011] to capture problems that can be solved by "continuous local search" methods. In Section 6.2 we also consider a seemingly weaker version of CLS, which we call Linear-CLS, and show that it is in fact the same as CLS. Finally, we also define a Gradient Descent problem where we do not have access to the gradient of the function (which might, in fact, not even be differentiable) and instead use "finite differences" to compute an approximate gradient. We show that this problem remains PPAD ∩ PLS-complete.

6.1 Consequences for CLS

The class CLS was defined by Daskalakis and Papadimitriou [2011] as a more natural counterpart to PPAD ∩ PLS. Indeed, Daskalakis and Papadimitriou noted that all the 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, which is 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, CLS is actually equal to PPAD ∩ PLS.

Theorem 6.1. CLS = PPAD ∩ PLS.

Recall that in Theorem 5.1, we have shown that 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. Before proving this in Proposition 6.3 below, we explore some further consequences of our results for CLS.

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

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

For the definitions of these problems, which are computational versions of Banach's fixed point theorem, see [Daskalakis et al., 2018] and [Fearnley et al., 2017], respectively.

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 [2011] 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 (Lemma C.1), it is unclear how to construct a reduction in the other direction. Indeed, similar reductions exists 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 $\subset$ 3D-CLS $\subset$ 4D-CLS .... We show that even the two-dimensional version is PPAD $\cap$ 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 (Lemma C.3), 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 hypercube $[0,1]^n$. 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 $\cap$ 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 Continuous-Localopt is shown to be PPAD $\cap$ PLS-hard, even on domain $[0,1]^2$ (Theorem 5.1).

- Circuits: CLS is defined using the problem Continuous-Localopt 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 arithmetic circuits. In Section 6.2 we show that Linear-CLS = CLS.

Before moving on to Section 6.2 and Linear-CLS, we provide the last reduction in the chain of reductions proving Theorem 6.1.

Proposition 6.3. General-Continuous-Localopt *with fixed domain* $[0,1]^2$ *reduces to* 2D-Continuous-Localopt *using a promise-preserving reduction. In particular, the problem lies in* CLS.

Proof. Given an instance (ε, p, g, L) of General-Continuous-Localopt with fixed domain $[0,1]^2$, we construct the instance (ε, p, g', L) of 2D-Continuous-Localopt, where $g'(x) = \Pi_D(g(x))$. Note that since $D = [0,1]^2$, the projection Π_D can easily be computed as $[\Pi_D(x)]_i = \min\{1, \max\{0, x_i\}\}$ for all $x \in \mathbb{R}^2$ and $i \in [2]$. In particular, since we extend

g by using only the gates $-$, $\times\zeta$, min, max and rational constants, the circuit for g' is also well-behaved.

Any non-violation solution of the constructed instance is also a solution of the original instance. Any violation of the constructed instance is immediately mapped back to a violation of the original instance. In particular, it holds that $\|g'(x)-g'(y)\| \leq \|g(x)-g(y)\|$ for all $x, y \in [0,1]^2$, since projecting two points cannot increase the distance between them. This implies that any violation of the L-Lipschitzness of g' is also a violation of the L-Lipschitzness of g. Note that by Lemma C.2 we do not need to ensure that the codomain of p is in $[0,1]$. Finally, it is easy to see that 2D-Continuous-Localopt lies in CLS, since it immediately reduces to 3D-Continuous-Localopt (Lemma C.1). □

6.2 Linear-CLS and Gradient Descent with Finite Differences

The class CLS was defined by Daskalakis and Papadimitriou [2011] 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 9. 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

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

For $k \in \mathbb{N}$, we let kD-Linear-Continuous-Localopt denote the problem Linear-Continuous-Localopt where n is fixed to be equal to k. Note that the definition of Linear-Continuous-Localopt does not require violation solutions, since every linear arithmetic circuit is automatically Lipschitz-continuous with a Lipschitz-constant that can be represented with a polynomial number of bits (Lemma A.1). In particular, Linear-Continuous-Localopt reduces to Continuous-Localopt and thus to General-Continuous-Localopt.

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.

Note that, just as for CLS, the one-dimensional version can be solved in polynomial time, i.e., 1D-Linear-CLS = FP. The containment 2D-Linear-CLS ⊆ PPAD ∩ PLS immediately follows from the fact that 2D-Linear-CLS ⊆ CLS ⊆ PPAD ∩ PLS. The other, more interesting, containment in Theorem 6.4 can be proved by directly reducing 2D-Continuous-Localopt to 2D-Linear-Continuous-Localopt. This reduction mainly relies on a more 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 (Theorem E.1) is stated and proved in Appendix E. The proof uses known techniques developed in the study of the complexity of Nash equilibria [Daskalakis et al.,

2009; Chen et al., 2009b], 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. Note that even though we define the problem on the domain $[0,1]^n$, it can be defined on more general domains as in our other problems.

Definition 10. GD-Finite-Diff:
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. Formally, find $x \in [0,1]^n$ such that

$$f(\Pi_D(x - \eta \widetilde{\nabla}_h f(x))) \geq f(x) - \varepsilon$$

where for all $i \in [n]$

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

GD-Finite-Diff immediately reduces to Linear-Continuous-Localopt by setting $p := f$ and $g := \Pi_D(x - \eta \widetilde{\nabla}_h f(x))$. It is easy to construct a linear arithmetic circuit computing g, given a linear arithmetic circuit computing f. Note, in particular, that the projection Π_D can be computed by a linear circuit since $D = [0,1]^n$. Indeed, $[\Pi_D(x)]_i = \min\{1, \max\{0, x_i\}\}$ for all $i \in [n]$ and $x \in \mathbb{R}^n$. Finally, the restriction of the codomain of p to $[0,1]$ can be handled exactly as in the proof of Lemma C.2.

In particular, the reduction from GD-Finite-Diff to Linear-Continuous-Localopt is domain-preserving and thus Theorem 6.4 immediately follows from the following theorem.

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 GD-Finite-Diff 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.

Proof. As explained above, GD-Finite-Diff immediately reduces to Linear-Continuous-Localopt and thus to General-Continuous-Localopt, which lies in PPAD ∩ PLS by Proposition 5.6. Thus, it remains to show that GD-Finite-Diff is PPAD ∩ PLS-hard when we fix $n = 2$. This is achieved by reducing from GD-Local-Search on domain $[0,1]^2$, which is PPAD ∩ PLS-hard by Theorem 5.1. In fact, we can even simplify the reduction by only considering GD-Local-Search instances that have some additional structure, but remain PPAD ∩ PLS-hard. Namely, consider an instance $(\varepsilon, \eta, f, \nabla f, L)$ of GD-Local-Search on domain $D = [0,1]^2$ such that:

- ∇f is the gradient of f,
- f and ∇f are L-Lipschitz-continuous on $[-1,2]^2$.

To see that the problem remains PPAD ∩ PLS-hard even with these restrictions, note that the restrictions are satisfied by the hard instances constructed for the KKT problem in the proof of Theorem 4.1, and that the reduction from KKT to GD-Local-Search in Proposition 5.4 also trivially preserves them. In particular, even though the proof of Theorem 4.1 only mentions that f and ∇f are L-Lipschitz-continuous on $[0,1]^2$, the same arguments also show that they are L-Lipschitz-continuous on $[-1,2]^2$ (where L has been scaled by some fixed constant).

Let us now reduce from the instance $(\varepsilon, \eta, f, \nabla f, L)$ of GD-Local-Search to GD-Finite-Diff. We construct the instance $(\varepsilon', \eta, h, F)$ of GD-Finite-Diff where $\varepsilon' = \varepsilon/4$, $h = \min\{1, \frac{\varepsilon}{8\eta L^2}\}$ and F is a linear arithmetic circuit that is obtained as follows. Let $\delta = \min\{\varepsilon/4, Lh^2/2\}$. By Theorem E.1 and Remark 3, we can construct a linear arithmetic circuit $F : \mathbb{R}^2 \to \mathbb{R}$ in polynomial time in size(f), $\log L$ and $\log(1/\delta)$ such that $|f(x) - F(x)| \le \delta$ for all $x \in [-1,2]^2$. Note that the second possibility in Theorem E.1 cannot occur, since f is guaranteed to be L-Lipschitz-continuous on $[-1,2]^2$.

Consider any solution of that instance of GD-Finite-Diff, i.e., a point $x \in [0,1]^2$ such that $F(\Pi_D(x - \eta\widetilde{\nabla}_h F(x))) \ge F(x) - \varepsilon/4$. Let us show that x is a solution to the original GD-Local-Search instance, i.e., that $f(\Pi_D(x - \eta\nabla f(x))) \ge f(x) - \varepsilon$.

We have that for $i \in \{1,2\}$

$$
\begin{aligned}
&\left|\left[\widetilde{\nabla}_h f(x)\right]_i - \left[\nabla f(x)\right]_i\right| \\
&= \left|\frac{f(x + h\cdot e_i) - f(x - h\cdot e_i)}{2h} - \left[\nabla f(x)\right]_i\right| \\
&\le \frac{1}{2h}\left(\left|f(x + h\cdot e_i) - f(x) - h\left[\nabla f(x)\right]_i\right| + \left|-f(x - h\cdot e_i) + f(x) - h\left[\nabla f(x)\right]_i\right|\right) \\
&= \frac{1}{2h}\left(\left|f(x + h\cdot e_i) - f(x) - \left\langle \nabla f(x), (x + h\cdot e_i) - x\right\rangle\right|\right. \\
&\quad \left. + \left|-f(x - h\cdot e_i) + f(x) + \left\langle \nabla f(x), (x - h\cdot e_i) - x\right\rangle\right|\right) \\
&\le \frac{1}{2h}\left(\frac{L}{2}\left\|h\cdot e_i\right\|^2 + \frac{L}{2}\left\|-h\cdot e_i\right\|^2\right) = \frac{Lh}{2}
\end{aligned}
$$

where we used Taylor's theorem (Lemma 3.4). Note that $x \pm h\cdot e_i \in [-1,2]^2$, since $h \le 1$. Furthermore, it is easy to see that $\left|[\widetilde{\nabla}_h F(x)]_i - [\widetilde{\nabla}_h f(x)]_i\right| \le \delta/h$, since F approximates f up to error δ on all of $[-1,2]^2$. It follows that $\left\|\widetilde{\nabla}_h F(x) - \nabla f(x)\right\| \le \sqrt{2}(\delta/h + Lh/2) \le 2Lh$. From this it follows that

$$
\left|f\Big(\Pi_D\big(x - \eta\nabla f(x)\big)\Big) - f\Big(\Pi_D\big(x - \eta\widetilde{\nabla}_h F(x)\big)\Big)\right|
$$

$$
\begin{aligned}
&\leq L \cdot \left\| \Pi_D\big(x - \eta \nabla f(x)\big) - \Pi_D\big(x - \eta \widetilde{\nabla}_h F(x)\big) \right\| \\
&\leq L \cdot \left\| \big(x - \eta \nabla f(x)\big) - \big(x - \eta \widetilde{\nabla}_h F(x)\big) \right\| \\
&\leq \eta L \cdot \left\| \widetilde{\nabla}_h F(x) - \nabla f(x) \right\| \\
&\leq 2\eta L^2 h.
\end{aligned}
$$

Finally, note that $|f(x) - F(x)| \leq \delta \leq \varepsilon/4$ and

$$
\left| f\Big(\Pi_D\big(x - \eta \widetilde{\nabla}_h F(x)\big)\Big) - F\Big(\Pi_D\big(x - \eta \widetilde{\nabla}_h F(x)\big)\Big) \right| \leq \delta \leq \varepsilon/4.
$$

Thus, since $F\Big(\Pi_D\big(x - \eta \widetilde{\nabla}_h F(x)\big)\Big) \geq F(x) - \varepsilon/4$, it follows that

$$
f\Big(\Pi_D\big(x - \eta \nabla f(x)\big)\Big) \geq f(x) - 3\varepsilon/4 - 2\eta L^2 h
$$

i.e., x is a solution to the original GD-Local-Search instance, since $2\eta L^2 h \leq \varepsilon/4$. □

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 [Fabrikant et al., 2004]. As mentioned in Section 1.2, Babichenko and Rubinstein [2020] have recently applied our main result to obtain PPAD ∩ PLS-completeness under randomized reductions. It would be interesting to show completeness under deterministic reductions.

- 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 randomised reduction by Babichenko and Rubinstein [2020] is that the problem is PPAD ∩ PLS-complete for polynomials of degree 5 under randomized reductions.

- 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 [Etessami et al., 2020; Fearnley and Savani, 2020; Dang et al., 2020].

- ColorfulCarathéodory: A problem based on a theorem in convex geometry [Meunier et al., 2017].

The first three problems on this list were known to lie in CLS [Daskalakis and Papadimitriou, 2011], 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 [Fearnley et al., 2020]. We conjecture that EOPL $\neq$ PPAD $\cap$ PLS. Unlike CLS, EOPL has a more standard combinatorial definition that is simultaneously a special case of End-of-Line and Localopt. While PPAD $\cap$ 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 $\cap$ PLS, in the sense of Beame et al. [1998].

Acknowledgments

Alexandros Hollender is supported by an EPSRC doctoral studentship (Reference 1892947).

References

Amir Ali Ahmadi and Jeffrey Zhang. On the complexity of finding a local minimizer of a quadratic function over a polytope. *arXiv preprint*, 2020a. URL https://arxiv.org/abs/2008.05558.

Amir Ali Ahmadi and Jeffrey Zhang. Complexity aspects of local minima and related notions. *arXiv preprint*, 2020b. URL https://arxiv.org/abs/2008.06148.

Yakov Babichenko and Aviad Rubinstein. Settling the complexity of Nash equilibrium in congestion games. In submission, November 2020.

Paul Beame, Stephen Cook, Jeff Edmonds, Russell Impagliazzo, and Toniann Pitassi. The Relative Complexity of NP Search Problems. *Journal of Computer and System Sciences*, 57(1):3–19, 1998. doi:10.1145/225058.225147.

Dimitri P. Bertsekas. *Nonlinear Programming*. Athena Scientific, 1999.

Avrim Blum and Ronald L. Rivest. Training a 3-node neural network is NP-complete. *Neural Networks*, 5(1):117–127, 1992. doi:10.1016/S0893-6080(05)80010-3.

Sébastien Bubeck and Dan Mikulincer. How to trap a gradient flow. *arXiv preprint*, 2020. URL https://arxiv.org/abs/2001.02968.

Yair Carmon, John C. Duchi, Oliver Hinder, and Aaron Sidford. Lower bounds for finding stationary points I. *Mathematical Programming*, pages 1–50, 2019. doi:10.1007/s10107-019-01406-y.

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

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

Xi Chen, Decheng Dai, Ye Du, and Shang-Hua Teng. Settling the complexity of Arrow-Debreu equilibria in markets with additively separable utilities. In *Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 273–282. IEEE, 2009a. doi:10.1109/FOCS.2009.29.

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

Chuangyin Dang, Qi Qi, and Yinyu Ye. Computations and complexities of Tarski's fixed points and supermodular games. *arXiv preprint*, 2020. URL https://arxiv.org/abs/2005.09836.

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

Constantinos Daskalakis, Paul W. Goldberg, and Christos H. Papadimitriou. The complexity of computing a Nash equilibrium. In *SIAM Journal on Computing*, volume 39, pages 195–259, 2009. doi:10.1137/070699652.

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

Constantinos Daskalakis, Stratis Skoulakis, and Manolis Zampetakis. The complexity of constrained min-max optimization. *arXiv preprint*, 2020. URL http://arxiv.org/abs/2009.09623.

Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

Kousha Etessami and Mihalis Yannakakis. On the complexity of Nash equilibria and other fixed points. *SIAM Journal on Computing*, 39(6):2531–2597, 2010. doi:10.1137/080720826.

Kousha Etessami, Christos Papadimitriou, Aviad Rubinstein, and Mihalis Yannakakis. Tarski's Theorem, Supermodular Games, and the Complexity of Equilibria. In *Proceedings of the 11th Innovations in Theoretical Computer Science Conference (ITCS)*, pages 18:1–18:19. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ITCS.2020.18.

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

John Fearnley and Rahul Savani. A faster algorithm for finding Tarski fixed points. *arXiv preprint*, 2020. URL https://arxiv.org/abs/2010.02618.

John Fearnley, Spencer Gordon, Ruta Mehta, and Rahul Savani. CLS: new problems and completeness. *arXiv preprint*, 2017. URL https://arxiv.org/abs/1702.06017.

John Fearnley, Spencer Gordon, Ruta Mehta, and Rahul Savani. Unique End of Potential Line. *Journal of Computer and System Sciences*, 114:1 – 35, 2020. doi:10.1016/j.jcss.2020.05.007.

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

Chi Jin, Praneeth Netrapalli, Rong Ge, Sham M. Kakade, and Michael I. Jordan. On nonconvex optimization for machine learning: Gradients, stochasticity, and saddle points. *arXiv preprint*, 2019. URL http://arxiv.org/abs/1902.04811.

David S. Johnson, Christos H. Papadimitriou, and Mihalis Yannakakis. How easy is local search? *Journal of Computer and System Sciences*, 37(1):79–100, 1988. doi:10.1016/0022-0000(88)90046-3.

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

Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley Professional, 1998.

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

Ruta Mehta. Constant rank bimatrix games are PPAD-hard. In *Proceedings of the 46th Annual ACM Symposium on the Theory of Computing (STOC)*, pages 545–554, 2014. doi:10.1145/2591796.2591835.

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

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

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

Christos H. Papadimitriou. The complexity of the Lin-Kernighan heuristic for the traveling salesman problem. *SIAM Journal on Computing*, 21(3):450–465, 1992. doi:10.1137/0221030.

Christos H. Papadimitriou. On the complexity of the parity argument and other inefficient proofs of existence. *Journal of Computer and System Sciences*, 48(3):498–532, 1994. doi:10.1016/S0022-0000(05)80063-7.

Herbert Robbins and Sutton Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, pages 400–407, 1951.

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

Stephen A. Vavasis. Black-box complexity of local minimization. *SIAM Journal on Optimization*, 3(1):60–80, 1993. doi:10.1137/0803004.

A More on arithmetic circuits

A.1 Evaluation of well-behaved arithmetic circuits (Proof of Lemma 3.3)

We restate the Lemma here for convenience.

Lemma 3.3. *Let f be a well-behaved arithmetic circuit with n inputs. Then, for any rational $x \in \mathbb{R}^n$, $f(x)$ can be computed in time* $\text{poly}(\text{size}(f), \text{size}(x))$.

Proof. Recall that 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. Without loss of generality, we can assume that the circuit f only contains gates that are used to compute at least one of the outputs.

Let x denote the input to circuit f and for any gate g of f let $v(g)$ denote the value computed by gate g when x is provided as input to the circuit. For any gate g that is not an input gate or a constant gate, let g_1 and g_2 denote the two gates it uses as inputs. Clearly, if g is one of $\{+, -, \times, \max, \min, >\}$, $v(g)$ can be computed in polynomial time in $\text{size}(v(g_1)) + \text{size}(v(g_2))$, including transforming it into an irreducible fraction. Thus, in order to show that the circuit can be evaluated in polynomial time, it suffices to show that for all gates g of f, it holds that $\text{size}(v(g)) \leq p(\text{size}(f) + \text{size}(x))$, where p is some fixed polynomial (independent of f and x). In the rest of this proof, we show that

$$\text{size}(v(g)) \leq 6 \cdot \text{size}(f)^3 \cdot \text{size}(x).$$

It is convenient to partition the gates of the circuit depending on their depth. For any gate g in f, we let $d(g)$ denote the depth of the gate in f. The input gates and the constant gates are at depth 1. For any other gate g, we define its depth inductively as $d(g) = 1 + \max\{d(g_1), d(g_2)\}$, where g_1 and g_2 are the two input gates of g. Note that $d(g) \leq \text{size}(f)$ for all gates g in the circuit.

We also define a notion of "multiplication-depth" $md(g)$. The gates g at depth 1 all have $md(g) = 0$. For the rest of the gates, the multiplication-depth is defined inductively. For a gate g whose inputs are g_1 and g_2, we let $md(g) = 1 + \max\{md(g_1), md(g_2)\}$ if g is a true multiplication gate, and $md(g) = \max\{md(g_1), md(g_2)\}$ otherwise. Since f is well-behaved, it immediately follows that $md(g) \leq \log(\text{size}(f))$ for all gates g of the circuit.

We begin by showing that for any gate g of f, it holds that $|v(g)| \leq 2^{\text{size}(f)^2(\text{size}(x)+\text{size}(f))}$. This follows from the stronger statement that

$$|v(g)| \leq 2^{d(g) \cdot 2^{md(g)} \cdot (\text{size}(x)+\text{size}(f))},$$

which we prove by induction as follows. First of all, note that any gate at depth 1 satisfies the statement, since any input or constant of the circuit is bounded by $2^{\text{size}(x)}$ or $2^{\text{size}(f)}$ respectively. Next, assume that the statement holds for all gates with depth $\leq k-1$ and consider some gate g at depth k. Let g_1 and g_2 denote its two inputs, which must satisfy that $d(g_1) \leq k-1$ and $d(g_2) \leq k-1$. If g is one of $\{\min, \max, <\}$, then the statement immediately also holds for g. If g is an addition or subtraction gate, then $|v(g)| \leq |v(g_1)| + |v(g_2)| \leq 2\max\{|v(g_1)|, |v(g_2)|\}$, which implies that the statement also hold for g, since $d(g_1), d(g_2) \leq k-1$ and $d(g) = k$. If g is a multiplication by a constant, then $|v(g)| \leq 2^{\text{size}(f)}|v(g_1)|$ (wlog g_2 is the constant), and the statement holds for g too. Finally, if g is a true multiplication gate, then $|v(g)| = |v(g_1)||v(g_2)| \leq (\max\{|v(g_1)|, |v(g_2)|\})^2$. Since $md(g) = 1 + \max\{md(g_1), md(g_2)\}$, it follows that the statement also holds for g.

Let $den(g)$ denote the absolute value of the denominator of $v(g)$ (written as an irreducible fraction). We show that for all gates g, it holds that $den(g) \leq 2^{\text{size}(f)^2(\text{size}(x)+\text{size}(f))}$. This is

enough to conclude our proof. Indeed, since we also have that $|v(g)| \leq 2^{\text{size}(f)^2(\text{size}(x)+\text{size}(f))}$, it follows that the absolute value of the numerator of $v(g)$ is

$$|v(g)| \cdot den(g) \leq 2^{2\cdot\text{size}(f)^2(\text{size}(x)+\text{size}(f))}.$$

As a result, it follows that

$$\text{size}(v(g)) \leq 2\cdot\text{size}(f)^2(\text{size}(x)+\text{size}(f))+\text{size}(f)^2(\text{size}(x)+\text{size}(f)) \leq 6\cdot\text{size}(f)^3\cdot\text{size}(x).$$

It remains to show that $den(g) \leq 2^{\text{size}(f)^2(\text{size}(x)+\text{size}(f))}$, which we prove by showing that $den(g) \leq 2^{d(g)\cdot 2^{md(g)}\cdot(\text{size}(x)+\text{size}(f))}$. Let M denote (the absolute value of) the product of all denominators appearing in the input x and the description of f, i.e., the denominators of the coordinates of the input x, and the denominators of the constants used by f. Note that $M \leq 2^{\text{size}(x)+\text{size}(f)}$. We prove by induction that for all gates g,

$$den(g) \text{ is a factor of } M^{d(g)\cdot 2^{md(g)}}$$

which in particular implies the bound on $den(g)$ above. First of all, note that any gate at depth 1 is an input or a constant, and thus satisfies the statement. Next, assume that the statement holds for all gates with depth $\leq k-1$ and consider some gate g at depth k. Let g_1 and g_2 denote its two inputs, which must satisfy that $d(g_1) \leq k-1$ and $d(g_2) \leq k-1$. If g is one of $\{\min, \max, <\}$, then it is easy to see that the statement immediately also holds for g. If g is an addition or subtraction gate, then, since $v(g_1)$ and $v(g_2)$ can both be expressed as fractions with denominator $M^{d(g)\cdot 2^{md(g)}}$, so can $v(g)$, and the statement also holds for g. If g is a multiplication by a constant, then $den(g)$ is a factor of $M^{d(g)\cdot 2^{md(g)}}$, since $den(g_1)$ is a factor of $M^{(d(g)-1)\cdot 2^{md(g)}}$ and the denominator of the constant is a factor of M (wlog assume that g_2 is the constant). Finally, if g is a true multiplication gate, then $den(g_1)$ and $den(g_2)$ are factors of $M^{d(g)\cdot 2^{md(g)-1}}$, and thus $den(g)$ is a factor of $(M^{d(g)\cdot 2^{md(g)-1}})^2 = M^{d(g)\cdot 2^{md(g)}}$ as desired. □

A.2 Linear arithmetic circuits are Lipschitz-continuous

Linear arithmetic circuits are only allowed to use the operations $\{+, -, \max, \min, \times\zeta\}$ and rational constants. The operation $\times\zeta$ denotes multiplication by a constant (which is part of the description of the circuit). Every linear arithmetic circuit is in particular a well-behaved arithmetic circuit, and so, by Lemma 3.3, can be evaluated in polynomial time. Furthermore, every linear arithmetic circuit represents a Lipschitz-continuous function such that the Lipschitz constant has polynomial bit-size with respect to the size of the circuit.

Lemma A.1. *Any linear arithmetic circuit $f : \mathbb{R}^n \to \mathbb{R}^m$ is $2^{\text{size}(f)^2}$-Lipschitz-continuous (w.r.t. the ℓ_∞-norm) over $\mathbb{R}^n$.*

Proof. For any gate g of the circuit f, let $L(g)$ denote the Lipschitz-constant of the function which outputs the value of g, given the input x to the circuit. As in the proof of Lemma 3.3, it is convenient to partition the gates of f according to their depth. Note that for all the gates g at depth 1, i.e., the input gates and the constant gates, it holds that $L(g) \leq 1$. We show that any gate g at depth k satisfies $L(g) \leq 2^{k\cdot\text{size}(f)}$. It immediately follows from this that f is $2^{\text{size}(f)^2}$-Lipschitz-continuous (w.r.t. the ℓ_∞-norm) over $\mathbb{R}^n$.

Consider a gate g at depth k with inputs g_1 and g_2 (which lie at a lower depth). If g is $+$ or $-$, then $L(g) \leq L(g_1)+L(g_2) \leq 2\max\{L(g_1), L(g_2)\} \leq 2\cdot 2^{(k-1)\cdot\text{size}(f)} \leq 2^{k\cdot\text{size}(f)}$. If g is max or min, then it is easy to see that $L(g) \leq \max\{L(g_1), L(g_2)\} \leq 2^{k\cdot\text{size}(f)}$. Finally, if g is $\times\zeta$, then $L(g) \leq |\zeta|\cdot L(g_1) \leq 2^{\text{size}(f)}2^{(k-1)\cdot\text{size}(f)} = 2^{k\cdot\text{size}(f)}$, where we used the fact that $|\zeta| \leq 2^{\text{size}(f)}$. □

B Mathematical Tools

B.1 Tools from Convex Analysis and a Generalization of Farkas' Lemma

Let $D \subseteq \mathbb{R}^n$ be a non-empty closed convex set. Recall that the projection $\Pi_D : \mathbb{R}^n \to D$ is defined by $\Pi_D(x) = \operatorname{argmin}_{y \in D} \|x - y\|$, where $\|\cdot\|$ denotes the Euclidean norm. It is known that $\Pi_D(x)$ always exists and is unique. The following two results are standard tools in convex analysis, see, e.g., [Bertsekas, 1999].

Lemma B.1. *Let D be a non-empty closed convex set in $\mathbb{R}^n$ and let $y \in \mathbb{R}^n$. Then for all $x \in D$ it holds that*

$$\langle y - \Pi_D(y), x - \Pi_D(y) \rangle \leq 0.$$

Proposition B.2. *Let D_1 and D_2 be two disjoint non-empty closed convex sets in $\mathbb{R}^n$ and such that D_2 is bounded. Then, there exist $c \in \mathbb{R}^n \setminus \{0\}$ and $d \in \mathbb{R}$ such that $\langle c, x \rangle < d$ for all $x \in D_1$, and $\langle c, x \rangle > d$ for all $x \in D_2$.*

We will need the following generalization of Farkas' Lemma, which we prove below. For $\varepsilon = 0$, we recover the usual statement of Farkas' Lemma.

Lemma B.3. *Let $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^n$ and $\varepsilon \geq 0$. Then exactly one of the following two statements holds:*

1. $\exists x \in \mathbb{R}^n : Ax \leq 0$, $\langle b, x \rangle > \varepsilon \|x\|$,
2. $\exists y \in \mathbb{R}^m : \|A^T y - b\| \leq \varepsilon$, $y \geq 0$.

Proof. Let us first check that both statements cannot hold at the same time. Indeed, if this were the case, then we would obtain the following contradiction

$$\varepsilon \|x\| < \langle b, x \rangle = \langle A^T y, x \rangle + \langle b - A^T y, x \rangle \leq \langle y, Ax \rangle + \|b - A^T y\| \|x\| \leq \varepsilon \|x\|$$

where we used the fact that $\langle y, Ax \rangle \leq 0$ and the Cauchy-Schwarz inequality.

Now, let us show that if statement 2 does not hold, then statement 1 must necessarily hold. Let $D_1 = \{A^T y : y \geq 0\}$ and $D_2 = \{x : \|x - b\| \leq \varepsilon\}$. Note that since statement 2 does not hold, it follows that D_1 and D_2 are disjoint. Furthermore, it is easy to check that D_1 and D_2 satisfy the conditions of Proposition B.2. Thus, there exist $c \in \mathbb{R}^n \setminus \{0\}$ and $d \in \mathbb{R}$ such that $\langle c, x \rangle < d$ for all $x \in D_1$, and $\langle c, x \rangle > d$ for all $x \in D_2$. In particular, we have that for all $y \geq 0$, $\langle Ac, y \rangle = \langle c, A^T y \rangle < d$. From this it follows that $Ac \leq 0$, since if $[Ac]_i > 0$ for some i, then $\langle Ac, y \rangle \geq d$ for $y = \frac{|d|}{[Ac]_i} e_i$.

In order to show that $x := c$ satisfies the first statement, it remains to prove that $\langle b, c \rangle > \varepsilon \|c\|$. Note that by setting $y = 0$, we get that $0 = \langle c, A^T 0 \rangle < d$. Let $z = b - \varepsilon \frac{c}{\|c\|}$. Since $z \in D_2$, it follows that $\langle c, z \rangle > d > 0$. Since $\langle c, z \rangle = \langle c, b \rangle - \varepsilon \|x\|$, statement 1 indeed holds. □

B.2 Proof of Lemma 3.4 (Taylor's Theorem)

We restate the Lemma here for convenience.

Lemma 3.4 (Taylor's theorem)**.** *Let $f : \mathbb{R}^n \to \mathbb{R}$ be continuously differentiable and let $D \subseteq \mathbb{R}^n$ be convex. If ∇f is L-Lipschitz-continuous (w.r.t. the ℓ_2-norm) on D, then for all $x, y \in D$ we have*

$$\left| f(y) - f(x) - \langle \nabla f(x), y - x \rangle \right| \leq \frac{L}{2} \|y - x\|^2.$$

Proof. Let $g : [0,1] \to \mathbb{R}$ be defined by $g(t) = f(x + t(y-x))$. Then, g is continuously differentiable on $[0,1]$ and $g'(t) = \langle \nabla f(x+t(y-x)), y-x\rangle$. Furthermore, g' is $(L\|x-y\|^2)$-Lipschitz-continuous on $[0,1]$, since

$$\begin{aligned}|g'(t_1) - g'(t_2)| &= |\langle \nabla f(x + t_1(y-x)) - \nabla f(x + t_2(y-x)), y-x\rangle| \\ &\le \|\nabla f(x + t_1(y-x)) - \nabla f(x + t_2(y-x))\| \cdot \|y-x\| \\ &\le L \cdot \|t_1(y-x) - t_2(y-x)\| \cdot \|y-x\| \\ &\le L \cdot |t_1 - t_2| \cdot \|y-x\|^2\end{aligned}$$

where we used the Cauchy-Schwarz inequality. We also used the fact that ∇f is L-Lipschitz on D, and $x + t(y-x) \in D$ for all $t \in [0,1]$. Now, we can write

$$f(y) - f(x) - \langle \nabla f(x), y-x\rangle = g(1) - g(0) - g'(0) = \int_0^1 (g'(t) - g'(0))\,\mathrm{d}t$$

and thus

$$|f(y) - f(x) - \langle \nabla f(x), y-x\rangle| \le \int_0^1 |g'(t) - g'(0)|\,\mathrm{d}t \le \int_0^1 L \cdot \|x-y\|^2 \cdot |t|\,\mathrm{d}t = \frac{L}{2}\|y-x\|^2.$$

□

C Minor Observations on Continuous-Localopt and KKT

Lemma C.1. *For all integers $k_2 > k_1 > 0$, k_1*D-Continuous-Localopt *reduces to k_2*D-Continuous-Localopt *using a promise-preserving reduction.*

Proof. For $x \in \mathbb{R}^{k_2}$, we write $x = (x_1, x_2)$, where $x_1 \in \mathbb{R}^{k_1}$ and $x_2 \in \mathbb{R}^{k_2 - k_1}$. Let (ε, p, g, L) be an instance of k_1D-Continuous-Localopt. The reduction constructs the instance (ε, p', g', L) of k_2D-Continuous-Localopt, where

$$p'(x) = p'(x_1, x_2) = p(x_1) \qquad \text{and} \qquad g'(x) = g'(x_1, x_2) = (g(x_1), 0).$$

Clearly, the arithmetic circuits for p' and g' can be constructed in polynomial time and are well-behaved.

Since $|p'(x) - p'(y)| = |p(x_1) - p(y_1)| \le L\|x_1 - y_1\| \le L\|x-y\|$, it is clear that any violation $x, y \in [0,1]^{k_2}$ of L-Lipschitzness for p' also yields a violation $x_1, y_1 \in [0,1]^{k_1}$ for p. Similarly, since

$$\|g'(x) - g'(y)\| = \|(g(x_1), 0) - (g(y_1), 0)\| = \|g(x_1) - g(y_1)\| \le L\|x_1 - y_1\| \le L\|x-y\|,$$

any violation x, y of L-Lipschitzness for g' also yields a violation x_1, y_1 for g. Thus, any violation of the constructed instance is always mapped back to a violation of the original instance, and the reduction is indeed promise-preserving.

Finally, note that any proper solution $x \in [0,1]^{k_2}$ of the constructed instance, i.e., such that $p'(g'(x)) \ge p'(x) - \varepsilon$, immediately yields a solution $x_1 \in [0,1]^{k_1}$ of the original instance. □

Lemma C.2. Continuous-Localopt *with codomain $[0,1]$ for function p is equivalent to* Continuous-Localopt *without this restriction.*

Proof. It is clear that the version with the restriction trivially reduces to the version without the restriction. Thus, it remains to show the other direction, namely that CONTINUOUS-LOCALOPT without the codomain restriction reduces to the restricted version.

Let (ε, p, g, L) be an instance of CONTINUOUS-LOCALOPT with domain $[0,1]^n$ and without a codomain restriction for p. The reduction constructs the instance $(\varepsilon', p', g, L')$ of CONTINUOUS-LOCALOPT with domain $[0,1]^n$, where $\varepsilon' = \frac{\varepsilon}{2nL}$, $L' = \max\{L, \frac{1}{2n}\}$ and

$$p'(x) = \min\left\{1, \max\left\{0, \frac{1}{2} + \frac{p(x) - p(z_c)}{2nL}\right\}\right\}$$

where $z_c = (1/2, 1/2, \ldots, 1/2)$ is the centre of $[0,1]^n$. Note that the arithmetic circuit computing p' can be computed in polynomial time given the circuit for p, and that the modification of p will require using gates $\times\zeta$, but no general multiplication gates. Thus, the circuit for p' is also well-behaved. Note, in particular, that the value $p(z_c)$ can be computed in polynomial time in the size of the circuit for p. It follows that the reduction can be computed in polynomial time.

First of all, let us show that any point $x \in [0,1]^n$ such that $p'(x) \neq \frac{1}{2} + \frac{p(x)-p(z_c)}{2nL}$ will immediately yield a violation of the L-Lipschitzness of p. Indeed, if x and z_c satisfy the L-Lipschitzness of p, then this means that

$$|p(x) - p(z_c)| \leq L\|x - z_c\| \leq nL$$

since $x, z_c \in [0,1]^n$. As a result, it follows that $\frac{1}{2} + \frac{p(x)-p(z_c)}{2nL} \in [0,1]$ and thus $p'(x) = \frac{1}{2} + \frac{p(x)-p(z_c)}{2nL}$.

In the rest of this proof we assume that we always have $p'(x) = \frac{1}{2} + \frac{p(x)-p(z_c)}{2nL}$, since we can immediately extract a violation if we ever come across a point x where this does not hold. Let us now show that any solution of the constructed instance immediately yields a solution of the original instance. Clearly, any violation of the L'-Lipschitzness of g is trivially also a violation of L-Lipschitzness.

Next assume that $x, y \in [0,1]^n$ are a violation of L'-Lipschitzness of p'. Let us show by contradiction that x, y must be a violation of L-Lipschitzness for p. Indeed, assume that x, y satisfy the L-Lipschitzness for p, then

$$|p'(x) - p'(y)| = \left|\frac{p(x) - p(y)}{2nL}\right| \leq \frac{1}{2n}\|x - y\|$$

which is a contradiction to x, y being a violation of L'-Lipschitzness of p'.

Finally, consider any proper solution of the constructed instance, i.e., $x \in [0,1]^n$ such that $p'(g(x)) \geq p'(x) - \varepsilon'$. Then it follows straightforwardly that $p(g(x)) \geq p(x) - 2nL\varepsilon'$, which implies that x is a solution to the original instance, since $2nL\varepsilon' = \varepsilon$. Note that the reduction is also promise-preserving, since we always map violations of the constructed instance back to violations of the original instance. □

Lemma C.3. GENERAL-CONTINUOUS-LOCALOPT *with fixed dimension* $n = 1$ *can be solved in polynomial time. As a result, this also holds for* KKT, GD-LOCAL-SEARCH *and* GD-FIXPOINT.

Proof. This is a straightforward consequence of the fact that finding Brouwer fixed points in one dimension is easy. Consider any instance $(\varepsilon, A, b, p, g, L)$ of GENERAL-CONTINUOUS-LOCALOPT with $n = 1$. It is easy to see that any ε/L-approximate fixed point of $x \mapsto \Pi_D(g(x))$ immediately yields a solution to the GENERAL-CONTINUOUS-LOCALOPT instance.

Thus, we proceed as follows. First of all, from A and b we can directly determine $t_1, t_2 \in \mathbb{R}$ such that $D = [t_1, t_2]$. Note that the bit-size of t_1 and t_2 is polynomial in the input size. Then define a grid of points on the interval $[t_1, t_2]$ such that the distance between consecutive points is ε/L^2. Finally, using binary search, find two consecutive points x_1 and x_2 such that $g(x_1) \geq x_1$ and $g(x_2) \leq x_2$. One of these two points has to be an ε/L-approximate fixed point of $x \mapsto \Pi_D(g(x))$ (or we obtain a violation of Lipschitz-continuity). Binary search takes polynomial time, because the number of points is at most exponential in the input size.

Since the other three problems reduce to General-Continuous-Localopt using domain-preserving reductions (see Section 5), it follows that they can also be solved in polynomial time when $n = 1$. □

Lemma C.4. KKT *on domain* $[0,1]^n$ *can be solved in polynomial time in* $1/\varepsilon$, L *and the sizes of the circuits for* f *and* ∇f.

Proof. This follows from the fact that the problem can be solved by Gradient Descent in polynomial time in those parameters. Let $(\varepsilon, f, \nabla f, L)$ be an instance of KKT with domain $[0,1]^n$. First, compute $f(0)$ in polynomial time in size(f). If f is indeed L-Lipschitz-continuous, then it follows that $f(x) \in I = [f(0) - \sqrt{n}L, f(0) + \sqrt{n}L]$ for all $x \in [0,1]^n$. If we ever come across a point where this does not hold, we immediately obtain a violation of L-Lipschitz-continuity of f. So, for the rest of this proof we simply assume that $f(x) \in I$ for all $x \in [0,1]^n$.

Note that the length of interval I is $2\sqrt{n}L$, which is polynomial in L and n. By using the reduction in the proof of Proposition 5.4, we can solve our instance by solving the instance $(\varepsilon', \eta, f, \nabla f, L)$ of GD-Local-Search, where $\varepsilon' = \frac{\varepsilon^2}{8L}$ and $\eta = \frac{1}{L}$. The important observation here is that this instance of GD-Local-Search can be solved by applying Gradient Descent with step size η and with any starting point, in at most $\frac{|I|}{\varepsilon'} = \frac{16\sqrt{n}L^2}{\varepsilon^2}$ steps. Indeed, every step must improve the value of f by ε', otherwise we have found a solution. It is easy to see that each step of Gradient Descent can be done in polynomial time in size(∇f), n and $\log L$. Since the number of steps is polynomial in $1/\varepsilon$, L and n, the problem can be solved in polynomial time in $1/\varepsilon$, L, n, size(f) and size(∇f). Finally note that $n \leq$ size(f) (because f has n input gates). □

D General-Brouwer and General-Real-Localopt

In this section we define the computational problems General-Brouwer and General-Real-Localopt and prove that they are PPAD- and PLS-complete respectively. These two completeness results follow straightforwardly from prior work. The membership of General-Brouwer in PPAD and of General-Real-Localopt in PLS are used in this paper to show that our problems of interest lie in PPAD ∩ PLS.

Definition 11. General-Brouwer:
Input:

- precision parameter $\varepsilon > 0$,
- $(A, b) \in \mathbb{R}^{m \times n} \times \mathbb{R}^m$ defining a bounded non-empty domain $D = \{x \in \mathbb{R}^n : Ax \leq b\}$,
- well-behaved arithmetic circuit $g : \mathbb{R}^n \to \mathbb{R}^n$,
- Lipschitz constant $L > 0$.

Goal: Compute an approximate fixed point of g on domain D. Formally, find $x \in D$ such that

$$\|\Pi_D(g(x)) - x\| \leq \varepsilon.$$

Alternatively, we also accept a violation of L-Lipschitzness of g as a solution. Namely, $x, y \in D$ such that $\|g(x) - g(y)\| > L\|x - y\|$.

Proposition D.1. General-Brouwer *is* PPAD*-complete.*

Proof. Various formulations and special cases of the problem of finding a Brouwer fixed point are known to be PPAD-complete [Papadimitriou, 1994; Chen and Deng, 2009; Etessami and Yannakakis, 2010]. The PPAD-hardness of our General-Brouwer problem immediately follows from the PPAD-hardness of the problem on the domain $[0, 1]^2$ and when g is a linear arithmetic circuit, which is known from [Mehta, 2014].

The containment in PPAD essentially follows from Proposition 2 in [Etessami and Yannakakis, 2010], where it is shown that finding an approximate fixed point of a Brouwer function that is efficiently computable and continuous, when the domain is a bounded polytope, is in PPAD. In General-Brouwer, the function is not guaranteed to be continuous, but instead we allow violations of Lipschitz-continuity as solutions. However, it can easily be seen that the proof in [Etessami and Yannakakis, 2010] also applies to this case. Alternatively, we can also use our Theorem E.1 to approximate the circuit g by a linear arithmetic circuit (which is necessarily Lipschitz-continuous with a polynomially representable Lipschitz-constant, see Lemma 3.3) and then use [Etessami and Yannakakis, 2010, Proposition 2] directly. Note that since D is bounded, we can easily compute $M > 0$ such that $D \subseteq [-M, M]^n$ (using linear programming). Then, using Theorem E.1 and Remark 3, we can approximate g by a linear arithmetic circuit on the domain D. □

Definition 12. General-Real-Localopt:
Input:

- precision/stopping parameter $\varepsilon > 0$,
- $(A, b) \in \mathbb{R}^{m \times n} \times \mathbb{R}^m$ defining a bounded non-empty domain $D = \{x \in \mathbb{R}^n : Ax \leq b\}$,
- well-behaved arithmetic circuits $p : \mathbb{R}^n \to \mathbb{R}$ and $g : \mathbb{R}^n \to \mathbb{R}^n$,
- Lipschitz constant $L > 0$.

Goal: Compute an approximate local optimum of p with respect to g on domain D. Formally, find $x \in D$ such that

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

Alternatively, we also accept a violation of L-Lipschitzness of p as a solution. Namely, $x, y \in D$ such that $|p(x) - p(y)| > L\|x - y\|$.

Proposition D.2. General-Real-Localopt *is* PLS*-complete.*

Proof. The PLS-hardness of General-Real-Localopt immediately follows from Theorem 2.1 in [Daskalakis and Papadimitriou, 2011], where it is shown that the problem is

PLS-complete in the special case where the domain is $[0,1]^3$. The proof of membership in PLS for the domain $[0,1]^3$ immediately generalizes to $[0,1]^n$, even for non-fixed n. Thus, it remains to show that we can reduce General-Real-Localopt to the special case where the domain is $[0,1]^n$.

Note that since D is bounded, we can easily compute $M > 0$ such that $D \subseteq [-M, M]^n$ (using linear programming). We extend p and g to the whole hypercube $[-M, M]^n$ by using the projection onto D, namely $\widehat{p}(x) = p(\Pi_D(x))$ and $\widehat{g}(x) = g(\Pi_D(x))$. Since $\|x-y\| \geq \|\Pi_D(x)-\Pi_D(y)\|$ for all $x, y \in \mathbb{R}^n$, it follows that any violation of L-Lipschitzness for $\widehat{p}$ immediately yields a violation for p. If $x \in [-M, M]^n$ is an ε-approximate local optimum of $\widehat{p}$ with respect to $\widehat{g}$, i.e., $\widehat{p}(\Pi_D(\widehat{g}(x))) \geq \widehat{p}(x) - \varepsilon$, then it immediately follows that $\Pi_D(x) \in D$ is an ε-approximate local optimum of p with respect to g. Thus, we have reduced the problem to the case where the domain is a hypercube $[-M, M]^n$.

The final step is to change the domain from $[-M, M]^n$ to $[0,1]^n$, which can easily be achieved by letting $\tilde{p}(x) = \widehat{p}(2M \cdot x - M \cdot \mathbf{e})$ and $\tilde{g}(x) = (\widehat{g}(2M \cdot x - M \cdot \mathbf{e}) + M \cdot \mathbf{e})/2M$. Here $\mathbf{e} \in \mathbb{R}^n$ denotes the all-ones vector. A violation of $2ML$-Lipschitzness for $\tilde{p}$ immediately yields a violation of L-Lipschitzness for $\widehat{p}$. Furthermore, if $x \in [0,1]^n$ is an ε-approximate local optimum of $\tilde{p}$ with respect to $\tilde{g}$, then it is easy to see that $(2M \cdot x - M \cdot \mathbf{e}) \in [-M, M]^n$ is an ε-approximate local optimum of $\widehat{p}$ with respect to $\widehat{g}$. We thus obtain an instance on the domain $[0,1]^n$ with the functions $\tilde{p}$ and $\tilde{g}$ and $L' = 2ML$ instead of L. By using the same arguments as in [Daskalakis and Papadimitriou, 2011, Theorem 2.1], it follows that the problem lies in PLS. Note that we do not actually need to construct arithmetic circuits that compute $\tilde{p}$ and $\tilde{g}$ (from the given circuits for p and g), because it suffices to be able to compute the functions in polynomial time for the arguments in [Daskalakis and Papadimitriou, 2011] to go through. □

E Approximation by Linear Circuits

In this section, we show that functions computed by arithmetic circuits can be approximated by *linear* arithmetic circuits with a very small error. In linear arithmetic circuits we are only allowed to use the gates $+$, $-$, max, min, $\times\zeta$ and rational constants. In particular, we cannot use general multiplication gates or comparison gates.

Theorem E.1. *Given a well-behaved arithmetic circuit $f : [0,1]^n \to \mathbb{R}^d$, a purported Lipschitz constant $L > 0$, and a precision parameter $\varepsilon > 0$, in polynomial time in $\mathrm{size}(f)$, $\log L$ and $\log(1/\varepsilon)$, we can construct a linear arithmetic circuit $F : [0,1]^n \to \mathbb{R}^d$ such that for any $x \in [0,1]^n$ it holds that:*

- $\|f(x) - F(x)\|_\infty \leq \varepsilon$, *or*
- *given x, we can efficiently compute $y \in [0,1]^n$ such that*

$$\|f(x) - f(y)\|_\infty > L\|x - y\|_\infty.$$

Here "efficiently" means in polynomial time in $\mathrm{size}(x)$, $\mathrm{size}(f)$, $\log L$ and $\log(1/\varepsilon)$.

Our proof of this result relies on existing techniques introduced by Daskalakis et al. [2009] and Chen et al. [2009b], but with a modification that ensures that we only get a very small error. Indeed, using the usual so-called sampling trick with averaging does not work here. We modify the sampling trick to output the *median* instead of the average.

Since we believe that this tool will be useful in future work, we prove a more general version of Theorem E.1. This more general version is Theorem E.2 and it is presented and

proved in the next subsection, where we also explain how Theorem E.1 is easily obtained from Theorem E.2.

Remark 3. Note that in Theorem E.1 the domain $[0,1]^n$ can be replaced by $[-M,M]^n$ for any $M > 0$ (in which case the running time is polynomial in the same quantities and in $\log M$). This is easy to show by using a simple bijection between $[0,1]^n$ and $[-M,M]^n$. This also holds for the more general statement in Theorem E.2. In fact, the result holds for any convex set $S \subseteq [-M,M]^n$, as long as we can efficiently compute the projection onto S. Furthermore, the choice of the ℓ_∞-norm in the statement is not important, and it can be replaced by any other ℓ_p-norm, if f is L-Lipschitz-continuous with respect to that norm.

E.1 General Statement and Proof

In order to make the statement of the result as general as possible, we consider a class of functions $\mathcal{F}$. Every function $f \in \mathcal{F}$ has an associated representation, and we let $\text{size}(f)$ denote the length of the representation of f. For example, if $\mathcal{F}$ is the class of functions represented using a certain type of circuit, then $\text{size}(f)$ is the size of the circuit corresponding to f. The following definition is inspired by a similar notion in [Etessami and Yannakakis, 2010].

Definition 13. A class $\mathcal{F}$ of functions is said to be *polynomially-approximately-computable* if there exists a polynomial q such that for any function $f \in \mathcal{F}$ where $f : [0,1]^n \to \mathbb{R}^d$, any point $x \in [0,1]^n$, and any precision parameter $\delta > 0$, a value $v \in \mathbb{R}^d$ such that $\|f(x) - v\|_\infty \leq \delta$ can be computed in time $q(\text{size}(f) + \text{size}(x) + \log(1/\delta))$.

The next theorem basically says that any polynomially-approximately-computable class can be approximated by linear arithmetic circuits, as long as the functions are Lipschitz-continuous.

Theorem E.2. *Let $\mathcal{F}$ be a polynomially-approximately-computable class of functions. Given $f \in \mathcal{F}$ where $f : [0,1]^n \to \mathbb{R}^d$, $L > 0$ and $\varepsilon > 0$, in polynomial time in $\text{size}(f)$, $\log L$ and $\log(1/\varepsilon)$, we can construct a linear arithmetic circuit $F : [0,1]^n \to \mathbb{R}^d$ such that for any $x \in [0,1]^n$ it holds that:*

- $\|f(x) - F(x)\|_\infty \leq \varepsilon$, *or*
- *given x, we can efficiently compute $y \in [0,1]^n$ such that*

$$\|f(x) - f(y)\|_\infty > L\|x - y\|_\infty + \varepsilon/2.$$

Here "efficiently" means in polynomial time in $\text{size}(x)$, $\text{size}(f)$, $\log L$ and $\log(1/\varepsilon)$.

Note that Theorem E.2 immediately implies Theorem E.1, since the class of all well-behaved arithmetic circuits mapping $[0,1]^n$ to $\mathbb{R}^d$ is polynomially-approximately-computable. (In fact, it is even exactly computable.) Note that since $L\|x - y\|_\infty + \varepsilon/2 \geq L\|x - y\|_\infty$ for any $\varepsilon > 0$, we indeed immediately obtain Theorem E.1.

Proof of Theorem E.2. First of all, note that we can assume that $d = 1$. Indeed, if $f : [0,1]^n \to \mathbb{R}^d$, then we can consider $f_1, \ldots, f_d : [0,1]^n \to \mathbb{R}$ where $f_i(x) = [f(x)]_i$, and construct linear arithmetic circuits $F_1, \ldots, F_d$ approximating $f_1, \ldots, f_d$ (as in the statement of the theorem). By constructing $F(x) = (F_1(x), \ldots, F_d(x))$, we have then obtained a linear arithmetic that satisfies the statement of the theorem. Indeed, if for some $x \in [0,1]^n$ we have $\|f(x) - F(x)\|_\infty > \varepsilon$, then it follows that there exists $i \in [n]$

such that $|f_i(x) - F_i(x)| > \varepsilon$. From this it follows that we can efficiently compute y with $|f_i(x) - f_i(y)| > L\|x-y\|_\infty + \varepsilon/2$, which implies that $\|f(x) - f(y)\|_\infty > L\|x-y\|_\infty + \varepsilon/2$. Note that $d \leq \text{size}(f)$, so this construction remains polynomial-time with respect to $\text{size}(f)$, $\log L$ and $\log(1/\varepsilon)$.

Consider any $L > 0$, $\varepsilon > 0$ and $f \in \mathcal{F}$ where $f : [0,1]^n \to \mathbb{R}$. Pick $k \in \mathbb{N}$ such that $N := 2^k \geq 4L/\varepsilon$. We consider the partition of $[0,1]^n$ into N^n subcubes of side-length $1/N$. Every $p \in [N]^n$ then represents one subcube of the partition, and we let $\widehat{p} \in [0,1]^n$ denote the centre of that subcube. Formally, for all $p \in [N]^n$, $\widehat{p} \in [0,1]^n$ is given by

$$[\widehat{p}]_i = \frac{2p_i - 1}{2N} \qquad \text{for all } i \in [n].$$

For any $p \in [N]^n$, let $\tilde{f}(\widehat{p})$ denote the approximation of $f(\widehat{p})$ with error at most $\varepsilon/16$. Note that $\tilde{f}(\widehat{p})$ can be computed in time $q(\text{size}(f) + \text{size}(\widehat{p}) + \log(16/\varepsilon))$ (where q is the polynomial associated to $\mathcal{F}$). Since $\text{size}(\widehat{p})$ is polynomial in $\log L$ and $\log(1/\varepsilon)$, we can compute a rational number $M > 0$ such that $\text{size}(M)$ is polynomial in $\text{size}(f)$, $\log L$ and $\log(1/\varepsilon)$, and it holds that $|\tilde{f}(\widehat{p})| \leq M$ for all $p \in [N]^n$. We then define

$$C(p) := \left\lfloor \left(\tilde{f}(\widehat{p}) + M\right) \frac{16}{\varepsilon} \right\rfloor + 1$$

for all $p \in [N]^n$. Note that $C(p) \in [1, 32M/\varepsilon + 1] \cap \mathbb{N}$. Pick $m \in \mathbb{N}$ such that $2^m \geq 32M/\varepsilon + 1$. Then, $C : [N]^n \to [2^m]$ and we construct a boolean circuit $\{0,1\}^{kn} \to \{0,1\}^m$ that computes C. Importantly, the Boolean circuit can be constructed in polynomial time in $\text{size}(f)$, $\log L$ and $\log(1/\varepsilon)$. Before we move on, note that for all $p \in [N]^n$, letting $V(p) := (C(p)-1)\frac{\varepsilon}{16} - M$, it holds that

$$|f(\widehat{p}) - V(p)| \leq \left|f(\widehat{p}) - \tilde{f}(\widehat{p})\right| + \left|\tilde{f}(\widehat{p}) - V(p)\right| \leq \frac{\varepsilon}{8} \tag{7}$$

since $|f(\widehat{p}) - \tilde{f}(\widehat{p})| \leq \varepsilon/16$ and

$$\begin{aligned}
&\left|\tilde{f}(\widehat{p}) - V(p)\right| \\
&= \left|\tilde{f}(\widehat{p}) + M - \left\lfloor \left(\tilde{f}(\widehat{p}) + M\right) \frac{16}{\varepsilon} \right\rfloor \frac{\varepsilon}{16}\right| \\
&\leq \left|\tilde{f}(\widehat{p}) + M - \left(\tilde{f}(\widehat{p}) + M\right) \frac{16}{\varepsilon} \frac{\varepsilon}{16}\right| + \left|\left(\tilde{f}(\widehat{p}) + M\right) \frac{16}{\varepsilon} - \left\lfloor \left(\tilde{f}(\widehat{p}) + M\right) \frac{16}{\varepsilon} \right\rfloor\right| \frac{\varepsilon}{16} \\
&\leq \frac{\varepsilon}{16}.
\end{aligned}$$

Using Lemma E.3, which is our key lemma here and is stated and proved in the next subsection, we can construct a linear arithmetic circuit $F : [0,1]^n \to \mathbb{R}$ in polynomial time in $\text{size}(C)$ (and thus in $\text{size}(f)$, $\log L$ and $\log(1/\varepsilon)$), such that for all $x \in [0,1]^n$

$$F(x) \in \left[\min_{p \in S(x)} C(p), \max_{p \in S(x)} C(p)\right]$$

where $S(x) \subseteq [N]^n$ is such that

1. $|S(x)| \leq n + 1$,
2. $\|x - \widehat{p}\|_\infty \leq 1/N$ for all $p \in S(x)$, and
3. $S(x)$ can be computed in polynomial time in $\text{size}(x)$ and $\log N$.

We modify the linear circuit so that instead of outputting $F(x)$, it outputs $(F(x)-1)\frac{\varepsilon}{16}-M$. Note that this is straightforward to do using the arithmetic gates at our disposal. Since $V(p) = (C(p)-1)\frac{\varepsilon}{16} - M$, we obtain that for all $x \in [0,1]^n$

$$F(x) \in \left[\min_{p \in S(x)} V(p), \max_{p \in S(x)} V(p)\right].$$

We are now ready to complete the proof. For this it suffices to show that if $|f(x) - F(x)| > \varepsilon$, then the second point in the statement of the theorem must hold. Assume that $x \in [0,1]^n$ is such that $|f(x) - F(x)| > \varepsilon$. It immediately follows that there exists $p^* \in [N]^n$ such that $|f(x) - V(p^*)| > \varepsilon$. By Equation (7), it follows that $|f(x) - f(\widehat{p^*})| > \varepsilon - \varepsilon/8 = 7\varepsilon/8$. Note that we might not be able to identify p^*, since we can only approximately compute f. Thus, we instead proceed as follows. We compute $p' = \operatorname{argmax}_{p \in S(x)} |f'(x) - f'(\widehat{p})|$, where f' denotes computation of f with error at most $\varepsilon/32$. Note that p' can be computed in polynomial time in $\text{size}(x)$, $\text{size}(f)$, $\log L$ and $\log(1/\varepsilon)$, since f' and $S(x)$ can be computed efficiently.

We now show that $y = \widehat{p'} \in [0,1]^n$ satisfies the second point in the statement of the theorem. First of all, note that $|f'(x) - f'(\widehat{p^*})| > 7\varepsilon/8 - 2\varepsilon/32 = 13\varepsilon/16$. By the choice of p', it must be that $|f'(x) - f'(\widehat{p'})| \geq |f'(x) - f'(\widehat{p^*})| > 13\varepsilon/16$, which implies that $|f(x) - f(\widehat{p'})| > 13\varepsilon/16 - 2\varepsilon/32 > 3\varepsilon/4$. On the other hand, since we have that $\|x - \widehat{p'}\|_\infty \leq 1/N \leq \varepsilon/4L$ (because $p' \in S(x)$), it follows that $L\|x - \widehat{p'}\|_\infty \leq \varepsilon/4$. Thus, we indeed have that $|f(x) - f(y)| > L\|x - y\|_\infty + \varepsilon/2$, as desired. Since p' can be computed efficiently, so can $y = \widehat{p'}$. □

E.2 Key Lemma

Let us recall some notation introduced in the proof of Theorem E.2. For $N \in \mathbb{N}$, consider the partition of $[0,1]^n$ into N^n subcubes of side-length $1/N$. Every $p \in [N]^n$ then represents one subcube of the partition, and we let $\widehat{p} \in [0,1]^n$ denote the centre of that subcube. Formally, for all $p \in [N]^n$, $\widehat{p} \in [0,1]^n$ is given by

$$[\widehat{p}]_i = \frac{2p_i - 1}{2N}$$

for all $i \in [n]$.

Lemma E.3. *Assume that we are given a Boolean circuit $C : \{0,1\}^{kn} \to \{0,1\}^m$, interpreted as a function $C : [N]^n \to [2^m]$, where $N = 2^k$. Then, in polynomial time in $\text{size}(C)$, we can construct a linear arithmetic circuit $F : [0,1]^n \to \mathbb{R}$ such that for all $x \in [0,1]^n$*

$$F(x) \in \left[\min_{p \in S(x)} C(p), \max_{p \in S(x)} C(p)\right]$$

where $S(x) \subseteq [N]^n$ is such that

1. *$|S(x)| \leq n + 1$,*
2. *$\|x - \widehat{p}\|_\infty \leq 1/N$ for all $p \in S(x)$, and*
3. *$S(x)$ can be computed in polynomial time in $\text{size}(x)$ and $\log N$.*

Proof. We begin by a formal definition of $S(x)$ and prove that it has the three properties mentioned in the statement of the Lemma. We then proceed with the construction of the linear arithmetic circuit.

For $N \in \mathbb{N}$, consider the partition of $[0,1]$ into N subintervals of length $1/N$. Let $I_N : [0,1] \to [N]$ denote the function that maps any point in $[0,1]$ to the index of the subinterval that contains it. In the case where a point lies on the boundary between two subintervals, i.e., $x \in B = \{1/N, 2/N, \ldots, (N-1)/N\}$, the tie is broken in favour of the smaller index. Formally,

$$I_N(x) = \min \left\{ \ell \in [N] \;\middle|\; x \in \left[\frac{\ell - 1}{N}, \frac{\ell}{N} \right] \right\}.$$

We abuse notation and let $I_N : [0,1]^n \to [N]^n$ denote the natural extension of the function to $[0,1]^n$, where it is simply applied on each coordinate separately. Thus, if we consider the partition of $[0,1]^n$ into N^n subcubes of side-length $1/N$, then, for any point $x \in [0,1]^n$, $p = I_N(x) \in [N]^n$ is the index of the subcube containing x. For $x \in \mathbb{R}^n \setminus [0,1]^n$, we let $I_N(x) := I_N(y)$ where y is obtained by projecting every coordinate of x onto $[0,1]$.

Letting $\mathbf{e} \in \mathbb{R}^n$ denote the all-ones vector, we define

$$S(x) = \left\{ I_N(x + \alpha \cdot \mathbf{e}) \;\middle|\; \alpha \in \left[0, \frac{1}{2N} \right] \right\}.$$

In other words, we consider a small segment starting at x and moving up simultaneously in all dimensions, and we let $S(x)$ be the set of subcubes-indices of all the points on the segment. We can now prove the three properties of $S(x)$:

1. Note that for any $i \in [n]$, there exists at most one value $\alpha \in [0, 1/2N]$ such that $[x + \alpha \cdot \mathbf{e}]_i \in B = \{1/N, 2/N, \ldots, (N-1)/N\}$. We let α_i denote that value of α if it exists, and otherwise we let $\alpha_i = 1/2N$. Thus, we obtain $\alpha_1, \alpha_2, \ldots, \alpha_n \in [0, 1/2N]$ and we rename them β_i so that they are ordered, i.e., $\beta_1 \leq \beta_2 \leq \cdots \leq \beta_n$ and $\{\beta_i \,|\, i \in [n]\} = \{\alpha_i \,|\, i \in [n]\}$. By the definition of I_N, it is then easy to see that $\alpha \mapsto I_N(x + \alpha \cdot \mathbf{e})$ is constant on each of the intervals $[0, \beta_1]$, $(\beta_1, \beta_2]$, $(\beta_2, \beta_3]$, $\ldots$, $(\beta_{n-1}, \beta_n]$ and $(\beta_n, 1/2N]$. Since these $n+1$ intervals (some of which are possibly empty) cover the entirety of $[0, 1/2N]$, it follows that $|S(x)| \leq n + 1$.

2. Consider any $p \in S(x)$. Let $\alpha \in [0, 1/2N]$ be such that $p = I_N(y)$ where $y = x + \alpha \cdot \mathbf{e}$. For any $i \in [n]$, it holds that $x_i \leq y_i \leq x_i + 1/2N$. There are two cases to consider. If $I_N(y_i) = I_N(x_i)$, then this means that x_i lies in the subinterval of length $1/N$ centred at $[\widehat{p}]_i$, and thus, in particular, $|x_i - [\widehat{p}]_i| \leq 1/2N \leq 1/N$. The only other possibility is that $I_N(y_i) = I_N(x_i) + 1$, since $\alpha \in [0, 1/2N]$. But for this to happen, it must be that $(2I_N(x_i) - 1)/2N \leq x_i \leq 2I_N(x_i)/2N$, since $\alpha \leq 1/2N$. By definition, $[\widehat{p}]_i = (2I_N(y_i) - 1)/2N = (2I_N(x_i) + 1)/2N$, and so we again obtain that $|x_i - [\widehat{p}]_i| \leq 1/N$. Since this holds for all $i \in [n]$, it follows that $\|x - \widehat{p}\|_\infty \leq 1/N$.

3. Given $x \in [0,1]^n$, the values $\alpha_1, \ldots, \alpha_n \in [0, 1/2N]$, defined in the first point above, can be computed in polynomial time in size(x), n and $\log N$. Then, $S(x)$ can be computed by simply evaluating $I_N(x + \alpha \cdot \mathbf{e})$ for all $\alpha \in \{\alpha_1, \ldots, \alpha_n, 1/2N\}$, which can also be done in polynomial time in size(x), n and $\log N$. Note that since $n \leq$ size(x), the computation of $S(x)$ runs in polynomial time in size(x) and $\log N$.

We can now describe how the linear arithmetic circuit $F : [0,1]^n \to \mathbb{R}$ is constructed. Let $C : \{0,1\}^{kn} \to \{0,1\}^m$ be the Boolean circuit that is provided. It is interpreted as a function $C : [N]^n \to [2^m]$, where $N = 2^k$. Let $x \in [0,1]^n$ be the input to the linear arithmetic circuit. F is constructed to perform the following steps.

Step 1: Sampling trick. In the first step, we create a sample T of points close to x. This is a standard trick that was introduced in the study of the complexity of computing Nash equilibria [Daskalakis et al., 2009; Chen et al., 2009b]. Here we use the so-called equi-angle sampling trick introduced by Chen et al. [2009b]. The sample T consists of $2n+1$ points:

$$T = \left\{ x + \frac{\ell}{4nN} \cdot \mathbf{e} \;\middle|\; \ell \in \{0, 1, 2, \ldots, 2n\} \right\}.$$

Note that these $2n+1$ points can easily be computed by F given the input x. The following two observations are important:

1. for all $y \in T$, $I_N(y) \in S(x)$ (by definition of $S(x)$),
2. Let $T_b = \{y \in T \mid \exists i \in [n] : \mathrm{dist}(y_i, B) < \frac{1}{8nN}\}$, where $B = \{1/N, 2/N, \ldots, (N-1)/N\}$ and $\mathrm{dist}(y_i, B) = \min_{t \in B} |y_i - t|$. We call these the *bad* samples, because they are too close to a boundary between two subcubes. The points in $T_g = T \setminus T_b$ are the *good* samples. It holds that $|T_b| \leq n$. This is easy to see by fixing some coordinate $i \in [n]$, and noting that there exists at most one point $y \in T$ such that $\mathrm{dist}(y_i, B) < \frac{1}{8nN}$. Indeed, since the samples are successively $1/4nN$ apart, at most one can be sufficiently close to any given boundary. Furthermore, since the samples are all $1/2N$ close, at most one boundary can be sufficiently close to any of them (for every coordinate). Thus, since there is at most one bad sample for each coordinate, there are at most n bad samples overall.

Step 2: Bit extraction. In the second step, we want to compute $I_N(y)$ for all $y \in T$. This corresponds to extracting the first k bits of each coordinate of each point $y \in T$, because $N = 2^k$. Unfortunately, bit extraction is not a continuous function and thus it is impossible to always perform it correctly with a linear arithmetic circuit. Fortunately, we will show that we can perform it correctly for *most* points in T, namely all the good points in T_g.

Consider any $y \in T$ and any coordinate $i \in [n]$. In order to extract the first bit of y_i, the arithmetic circuit computes

$$b_1 = \min\{1, \max\{0, 8nN(y_i - 1/2)\}\} =: \phi(y_i - 1/2).$$

Note that if $y_i \geq 1/2 + 1/8nN$, then $8nN(y_i - 1/2) \geq 1$ and thus $b_1 = 1$. On the other hand, if $y_i \leq 1/2 - 1/8nN$, then $8nN(y_i - 1/2) \leq -1$ and thus $b_1 = 0$. This means that if $\mathrm{dist}(y_i, B) \geq 1/8nN$, the first bit of y_i is extracted correctly. Note that $B = \{1/N, 2/N, \ldots, (N-1)/N\} = \{1/2^k, 2/2^k, \ldots, (2^k - 1)/2^k\}$.

To extract the second bit, the arithmetic circuit computes $t := y_i - b_1/2$ and

$$b_2 = \phi(t - 1/4).$$

By the same argument as above, b_2 is the correct second bit of y_i, if $|t - 1/4| \geq 1/8nN$, i.e., if $|y_i - 1/4| \geq 8nN$ and $|y_i - 3/4| \geq 8nN$. Thus, if $\mathrm{dist}(y_i, B) \geq 1/8nN$, the second bit is also computed correctly, since $1/4, 3/4 \in B$.

To extract the third bit, the arithmetic circuit updates $t := t - b_2/4$ and computes $b_3 = \phi(t - 1/8)$. We proceed analogously up to the kth bit b_k. By induction and the same arguments as above, it follows that the first k bits of y_i are computed correctly by the arithmetic circuit as long as $\mathrm{dist}(y_i, B) \geq 1/8nN$. In particular, this condition always holds for $y \in T_g$.

By performing this bit extraction for each coordinate of each $y \in T$, we obtain the purported bit representation of $I_N(y)$ for each $y \in T$. The argumentation in the previous paragraphs shows that for all $y \in T_g$, we indeed obtain the correct bit representation of $I_N(y)$. For $y \in T_b$, we have no control over what happens, and it is entirely possible that the procedure outputs numbers that are not valid bits, i.e., not in $\{0, 1\}$.

Step 3: Simulation of the Boolean circuit. In the next step, for each $y \in T$, we evaluate the circuit C on the bits purportedly representing $I_N(y)$. The Boolean gates of C are simulated by the arithmetic circuit as follows:

- $\neg b := 1 - b$,
- $b \vee b' := \min\{1, b + b'\}$,
- $b \wedge b' := \max\{0, b + b' - 1\}$.

Note that if the input bits b, b' are valid bits, i.e., in $\{0, 1\}$, then the Boolean gates are simulated correctly, and the output is also a valid bit.

For $y \in T_g$, since the input bits indeed represent $I_N(y)$, the simulation of C will thus output the correct bit representation of $C(I_N(y)) \in [2^m]$. We can obtain the value $C(I_N(y)) \in [2^m]$ itself by decoding the bit representation, i.e., multiplying every bit by the corresponding power of 2 and adding all the terms together.

Let $V(y) \in \mathbb{R}$ denote the value that this step outputs for each $y \in T$. For $y \in T_g$, we have that $V(y) = C(I_N(y))$. For $y \in T_b$, there is no guarantee other than $V(y) \in \mathbb{R}$.

Step 4: Median using a sorting network. In the last step, we want to use the $|T| = 2n + 1$ values that we have computed (namely $\{V(y) \mid y \in T\}$) to compute the final output of our arithmetic circuit. In previous constructions of this type, in particular [Daskalakis et al., 2009; Chen et al., 2009b], the circuit would simply output the average of the $V(y)$. However, this is not good enough to prove our statement, because even a single bad point y can introduce an inversely polynomial error in the average.

In order to obtain a stronger guarantee, the arithmetic circuit instead outputs the *median* of the multiset $\{V(y) \mid y \in T\}$. The median of the given $2n + 1$ values can be computed by constructing a so-called sorting network (see, e.g., [Knuth, 1998]). The basic operation of a sorting network can easily be simulated by the max and min gates. It is easy to construct a sorting network for $2n + 1$ values that has size polynomial in n. The output of the sorting network will be the same values that it had as input, but sorted. In other words, the sorting network outputs $V_1 \leq V_2 \leq \cdots \leq V_{2n+1}$ such that $\{V_j \mid j \in [2n + 1]\} = \{V(y) \mid y \in T\}$ as multisets. The final output of our arithmetic circuit is V_{n+1}, which is exactly the median of the $2n + 1$ values.

Recall from step 1 that $|T_b| \leq n$ and thus $|T_g| \geq n + 1$. It immediately follows that either V_{n+1} corresponds to $V(y)$ of a good sample y, or there exist $i < n + 1$ and $j > n + 1$ such that both V_i and V_j correspond to good samples. In other words, the output of the circuit satisfies $F(x) \in [\min_{y \in T_g} C(I_N(y)), \max_{y \in T_g} C(I_N(y))]$. As noted in step 1, $I_N(y) \in S(x)$ for all $y \in T$. Thus, we obtain that $F(x) \in [\min_{p \in S(x)} C(p), \max_{p \in S(x)} C(p)]$.

It follows that the linear arithmetic circuit F that we have constructed indeed satisfies the statement of the Lemma. Furthermore, the construction we have described can be performed in polynomial time in size(C), n, m and k. Since size(C) $\geq \max\{n, k, m\}$, it is simply polynomial time in size(C). □

F Annotated Source Code

This appendix is a literate Haskell file, which is a Haskell source file that is also a valid latex document. This source file is `Proof.lhs`, which is responsible for carrying out the automated proof for each square. Other modules in the program are responsible for parsing the input templates and breaking them down into individual squares, and also creating the output report as a pdf, but it is this module that is crucial for the correctness of the proof. The full source code of the program is available at https://github.com/jfearnley/PPADPLS.

All of the Haskell code in the module will appear as inline listings in the document. For example, the following code is the preamble of the module that imports the libraries that we will be using.

```
module Proof where

import Data.SBV
import Data.Matrix
import Control.Monad
import GHC.Generics
```

A quick introduction to Haskell. In Haskell function application is written using spaces. So calling a function $f(x+1, y+2)$ would be written as `f (x + 1) (y + 2)`. The `$` operator implements function application, so `f $ a` means "apply function `f` to argument `a`. The `$` operator is often used to remove brackets from expressions, so instead of writing `f (x + 1)` we can instead write `f $ x + 1`.

The syntax `f :: Int -> Float -> String` is a type annotation that denotes that `f` is a two argument function that takes an `Int` as its first argument, a `Float` as its second argument, and it returns a `String`. The following example defines a two argument function that takes two integers and returns their sum.

```
example :: Int -> Int -> Int
example x y = x + y
```

There are many resources available online if more information on Haskell syntax is desired.

Introduction to the SBV library. The Haskell SBV library allows us to write programs in Haskell and then prove theorems about those programs using an SMT solver. This will allow us to succinctly specify the input to the solver as a readable and checkable Haskell program, rather than a very long and unreadable SMT formula.

We will construct an SMT formula over the algebraic real numbers, which is represented by the `SReal` type in SBV. It is important to understand that the SMT solver will genuinely verify the correctness of the formula over the set of algebraic reals. So there will be no approximations or rounding errors to worry about in the proof.

As a first example, the following code uses SBV to attempt to prove the formula

$$\forall x \cdot (x > 0 \Rightarrow x^2 \neq 2),$$

which states that there is no positive square root of two.

```
noRootTwo :: IO ThmResult
noRootTwo = prove $ do
```

```
    x <- symbolic "x" :: Symbolic SReal
    return $ x .> 0 .=> x * x ./= 2
```

The first line defines `x` to be a symbolic algebraic real variable, and the second line then directly constructs the formula in terms of `x`. Note that when we use the `SReal` type, we must prefix all of our comparison operators with a `.` symbol, so the usual Haskell not-equal operator `/=` becomes `./=`.

The formula is clearly false, and so when we run the proof we get the following output.

```
ghci> noRootTwo
Falsifiable. Counter-example:
  x = root(2, x^2 = 2) = 1.414213562373095... :: Real
```

Here the SMT solver identifies that the formula is false, and gives a counter example, which is of course $\sqrt{2}$. Note that the answer is given symbolically.

If we configure SBV to print out its interaction with the solver, then the following interaction is produced.

```
[GOOD] ; --- literal constants ---
[GOOD] (define-fun s1 () Real 0.0)
[GOOD] (define-fun s5 () Real (/ 2.0 1.0))
[GOOD] ; --- skolem constants ---
[GOOD] (declare-fun s0 () Real) ; tracks user variable "x"
[GOOD] ; --- formula ---
[GOOD] (define-fun s2 () Bool (> s0 s1))
[GOOD] (define-fun s3 () Bool (not s2))
[GOOD] (define-fun s4 () Real (* s0 s0))
[GOOD] (define-fun s6 () Bool (distinct s4 s5))
[GOOD] (define-fun s7 () Bool (or s3 s6))
[GOOD] (assert (not s7))
[SEND] (check-sat)
[RECV] sat
```

Here we can see that our formula has been translated into an SMT formula. Then the SMT solver is asked to find a satisfying assignment of the formula's negation. In this case it succeeds, meaning that our formula was false.

For a slightly more involved example, we will prove that Pythagoras's theorem is true. We will use three symbolic points $p, q, r \in \mathbb{R}^2$. The following function detects whether $\vec{pq}$ and $\vec{qr}$ form a right-angle, by checking whether their cross product is zero. Note that this function returns an `SBool`, which is a symbolic Boolean variable.

```
rightAngle :: (SReal, SReal) -> (SReal, SReal) -> (SReal, SReal) -> SBool
rightAngle (p1, p2) (q1, q2) (r1, r2) = v1 * u1 + v2 * u2 .== 0
    where (v1, v2) = (p1 - q1, p2 - q2)
          (u1, u2) = (r1 - q1, r2 - q2)
```

The next function checks whether $a^2 + b^2 = c^2$ for the triangle defined by p, q, and r.

```
pythagoras :: (SReal, SReal) -> (SReal, SReal) -> (SReal, SReal) -> SBool
pythagoras (p1, p2) (q1, q2) (r1, r2) = aSquared + bSquared .== cSquared
    where aSquared = (p1 - q1)^2 + (p2 - q2)^2
          bSquared = (q1 - r1)^2 + (q2 - r2)^2
          cSquared = (r1 - p1)^2 + (r2 - p2)^2
```

Finally, we can ask SBV to prove the theorem. Here we define six symbolic variables that encode the coordinates of p, q, and r, and then we encode a formula that states that if p, q, and r form a right-angled triangle, then the conclusion of Pythagoras's theorem must hold.

```
pythagorasProof :: IO ThmResult
pythagorasProof = prove $ do
    [p1, p2] <- symbolics ["p1", "p2"]
    [q1, q2] <- symbolics ["q1", "q2"]
    [r1, r2] <- symbolics ["r1", "r2"]
    return $     rightAngle (p1, p2) (q1, q2) (r1, r2)
             .=> pythagoras (p1, p2) (q1, q2) (r1, r2)
```

When we run the proof, the SMT solver verifies that Pythagoras's theorem is true.

```
ghci> pythagorasProof
Q.E.D.
```

F.1 Our Proof

We now turn to our own proof. The first few lines simply define the constants that we will use in the proof.

```
eps :: SReal
eps = 0.01

delta :: SReal
delta = 0.5

colorOffset :: SReal
colorOffset = 4
```

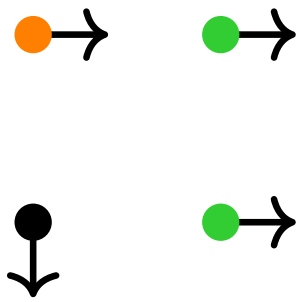
Next we define the types that will be used to input data into the proof. A `RawPoint` contains a colour string that is one of `"red"`, `"orange"`, `"black"`, `"green"`, or `"blue"`, and a direction string that is one of `"up"`, `"down"`, `"left"`, or `"right"`.

```
data RawPoint = RawPoint
    { col :: String
    , dir :: String
    } deriving (Show, Eq, Generic)
```

A `RawSquare` contains four `RawPoints`, corresponding to the four corners of the square. We will use `zz` as a suffix for the point at $(0, 0)$, `zo` as the suffix for the point at $(0, 1)$, `oz` as the suffix for the point at $(1, 0)$, and `oo` as the suffix for the point at $(1, 1)$.

```
data RawSquare = RawSquare
    { rzz :: RawPoint
    , rzo :: RawPoint
    , roz :: RawPoint
    , roo :: RawPoint
    } deriving (Show, Eq, Generic)
```

For example, consider the following square.



This would be represented as the following value.

```
RawSquare { rzz = RawPoint {col = "black",  dir = "down"}
          , rzo = RawPoint {col = "orange", dir = "right"}
          , roz = RawPoint {col = "green",  dir = "right"}
          , roo = RawPoint {col = "green",  dir = "right"}
          }
```

Symbolic squares. Next we define two types that represent the continuous function that we will build from the input data. The type `Point` contains four symbolic values: `f` gives the value of the function at the point, `fx` gives the gradient with respect to x at the point, `fy` gives the gradient with respect to y at the point, and `fxy` gives the gradient with respect to x and y at the point.

```
data Point = Point
    { f   :: SReal
    , fx  :: SReal
    , fy  :: SReal
    , fxy :: SReal
    } deriving Show
```

A `Square` contains four symbolic points, one for each corner of the square. We use the same naming convention as `RawSquare` for these points.

```
data Square = Square
    { zz :: Point
    , zo :: Point
    , oz :: Point
    , oo :: Point
    } deriving Show
```

Converting raw squares into squares. The next section of code turns the input data into a symbolic square, following the instructions given by the reduction from the paper.

We begin by defining the five colors that we used in the paper. We will represent the value of f at each point of the square symbolically. We will introduce five symbolic values `red`, `orange`, `black`, `green`, and `blue`. Then, if a point in the square is red, we will represent the value of f at that point as `red + offset`, where the offset is some number between 0 and 2. We will then quantify over all values that satisfy `red > orange + 4`, `orange > black + 4`, `black > green + 4`, and `green > blue + 4`. In this way, we are

able to show that the theorem holds no matter what particular values each color takes within the square, so long as those values are sufficiently far apart.

The following functions define the offsets for each color. Each of them takes `x` and `y` as inputs, which should both be assumed to belong to $\{0, 1\}$. The output is a number in $\{0, 1, 2\}$ that describes how that color behaves across the square.

The red function increases as we move right and down.

```
redGrad :: SReal -> SReal -> SReal
redGrad x y = x + (1-y)
```

The orange function increases as we go down or left.

```
orangeGrad :: SReal -> SReal -> SReal
orangeGrad x y = (1-x) + (1 - y)
```

The black function increases as we go up or right.

```
blackGrad :: SReal -> SReal -> SReal
blackGrad x y = x + y
```

The green function is the same as the orange function, while the blue function is the same as the red function, so we just reuse the implementations.

```
greenGrad :: SReal -> SReal -> SReal
greenGrad = orangeGrad

blueGrad :: SReal -> SReal -> SReal
blueGrad = redGrad
```

The `convertPotential` function takes a raw point, the list of symbolic values

`[red, orange, black, green, blue]`,

and an x and y coordinate for the point. It outputs the symbolic value of f at the point computed as we describe above.

```
convertPotential :: RawPoint -> [SReal] -> SReal -> SReal -> SReal
convertPotential point vars x y = convert (col point)
    where convert "red"    = vars !! 0 + redGrad    x y
          convert "orange" = vars !! 1 + orangeGrad x y
          convert "black"  = vars !! 2 + blackGrad  x y
          convert "green"  = vars !! 3 + greenGrad  x y
          convert "blue"   = vars !! 4 + blueGrad   x y
          convert x        = error ("unable to parse color " ++ show x)
```

The `convertDirection` function converts the direction given by the raw point into a gradient. It outputs a pair of symbolic values (x', y'), where x' is the gradient with respect to x, and y' is the gradient with respect to y. Recall from the paper that an arrow pointing upwards indicates that the function *decreases* as we move upwards, and does not move as we move left or right. Therefore this corresponds to setting $x' = 0$ and $y' = -\delta$, where δ is the parameter indicating how steep the function should be. The following function implements this, along with the other three cases.

```
convertDirection :: String -> (SReal, SReal)
convertDirection "up"    = ( 0, -delta)
convertDirection "down"  = ( 0,  delta)
convertDirection "right" = (-delta,  0)
convertDirection "left"  = ( delta,  0)
```

The next function converts a raw point to a symbolic point using the functions defined above. It sets `f` to be the value of the potential at the point, it sets `fx` and `fy` according to the gradients returned by `convertDirection`, and it sets `fxy` to be zero. All of these are in accordance with the reduction defined in the paper.

```
rawPointToPoint :: RawPoint -> [SReal] -> SReal -> SReal -> Point
rawPointToPoint point vars x y = Point
                               { f   = convertPotential point vars x y
                               , fx  = lr
                               , fy  = ud
                               , fxy = 0
                               }
    where (lr, ud) = convertDirection (dir point)
```

Finally, we can turn a raw square into a symbolic square. This function simply converts each point of the raw square using the function above, while passing in the appropriate coordinates.

```
rawSquareToSquare :: RawSquare -> [SReal] -> Square
rawSquareToSquare sq vars = Square
                               { zz = rawPointToPoint (rzz sq) vars 0 0
                               , zo = rawPointToPoint (rzo sq) vars 0 1
                               , oz = rawPointToPoint (roz sq) vars 1 0
                               , oo = rawPointToPoint (roo sq) vars 1 1
                               }
```

Bicubic interpolation. Recall that the bicubic interpolation inside this square will be a polynomial of the form:

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

where the coefficients a_{ij} are computed as follows

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

The following code implements this. We will use the `matrix` library to do the matrix multiplications. We first define `RealMatrix` to be a matrix type that holds symbolic values, and we then define the two constant matrices seen above.

```
type RealMatrix = Matrix SReal

leftMatrix :: RealMatrix
leftMatrix = fromList 4 4 [  1,  0,  0,  0
                          ,  0,  0,  1,  0
                          , -3,  3, -2, -1
                          ,  2, -2,  1,  1
                          ]

rightMatrix :: RealMatrix
rightMatrix = fromList 4 4 [ 1, 0, -3,  2
                           , 0, 0,  3, -2
                           , 0, 1, -2,  1
                           , 0, 0, -1,  1
                           ]
```

Next we define the `bicubic` function that takes a square and outputs the coefficient matrix, using the formula above.

```
bicubic :: Square -> RealMatrix
bicubic (Square zz zo oz oo) = leftMatrix * pointMatrix * rightMatrix
    where pointMatrix = fromList 4 4 [ f  zz, f  zo, fy  zz, fy  zo
                                     , f  oz, f  oo, fy  oz, fy  oo
                                     , fx zz, fx zo, fxy zz, fxy zo
                                     , fx oz, fx oo, fxy oz, fxy oo
                                     ]
```

Finally we provide two functions to extract the derivative of $f(x, y)$ with respect to x and with respect to y. These two functions take the coefficient matrix computed by `bicubic`, an x coordinate and a y coordinate, and output the gradient at that point.

For the x derivative, we observe that for a fixed value of y, we have that

$$f(x, y) = c_3x^3 + c_2x^2 + c_1x + c_0$$

where $(c_0, c_1, c_2, c_3) = A \cdot (1, y, y^2, y^3)^T$ and A is the matrix of coefficients. Therefore the x derivative at this point is $3c_3x^2 + 2c_2x + c_1$.

```
xDerivative :: RealMatrix -> SReal -> SReal -> SReal
xDerivative interpolation x y = 3*c3*x^2 + 2*c2*x + c1
    where yVec = fromList 4 1 [1, y, y^2, y^3]
          [c0, c1, c2, c3] = toList (interpolation * yVec)
```

Similarly, for a fixed value of x we have that

$$f(x, y) = c_3y^3 + c_2y^2 + c_1y + c_0$$

where $(c_0, c_1, c_2, c_3)^T = (1, x, x^2, x^3) \cdot A$ and A is the matrix of coefficients. Therefore the y derivative at this point is $3c_3y^2 + 2c_2y + c_1$.

```
yDerivative :: RealMatrix -> SReal -> SReal -> SReal
yDerivative interpolation x y = 3*c3*y^2 + 2*c2*y + c1
    where xVec = fromList 1 4 [1, x, x^2, x^3]
          [c0, c1, c2, c3] = toList (xVec * interpolation)
```

The proof. We are now ready to specify the formula that we will present to the SMT solver. We begin by specifying the constraints that will be used in the formula.

The following function checks that a symbolic value lies in $[0, 1]$. We will use this to ensure that the symbolic point (x, y) lies in the $[0, 1]^2$ square.

```
inSquare :: SReal -> SBool
inSquare x = (x .>= 0) .&& (x .<= 1)
```

The next constraint enforces the constraints on the symbolic values for the colors, which we described earlier. It takes the list of symbolic values `[red, orange, black, green, blue]`, and checks that all of the values are suitably far apart (`colorOffset` was defined to be 4 earlier on in the file).

```
colorConstraints :: [SReal] -> SBool
colorConstraints vars =  vars !! 0 .> vars !! 1 + colorOffset
                      .&& vars !! 1 .> vars !! 2 + colorOffset
                      .&& vars !! 2 .> vars !! 3 + colorOffset
                      .&& vars !! 3 .> vars !! 4 + colorOffset
```

Next we specify a list of tautologies. Note that each statement in the list is unconditionally true. We introduce these because, without them, the SMT solver does not verify the formula within a reasonable amount of time. So, we modify our formula from "if A then B", to "if A and T then B", where T is a tautology. Since the solver, at its core, is based on the DPLL algorithm, this in effect suggests a possible case analysis to the solver, which in practice dramatically speeds up the proof. Indeed, these tautologies were derived from our own experience with useful cases when we proved the theorem for some of the squares by hand.

Unfortunately we did not find one single tautology that worked for all of the squares. So here we give three possibilities. For each square we will try the proof with each of the tautologies, and we will consider the square to be verified if the solver terminates with at least one tautology.

```
tautologies :: SReal -> SReal -> [SBool]
tautologies x y = [ x .>= y   .|| y   .>= x
                  , y .>= 0.5 .|| 0.5 .>= y
                  , x .>= 0.5 .|| 0.5 .>= x
                  ]
```

Now we specify the conclusion of the proof. We want to prove that no ε-stationary points exist within the square, which means that at each point (x, y) in the interpolated square, it is not the case that both the x derivative and the y derivative are within ϵ of zero. The `nonZeroGradient` function takes the matrix of coefficients from the interpolation, and a symbolic point (x, y) as arguments. It computes the x derivative and the y derivative, and returns true if the gradients satisfy the condition.

```
notEpsClose :: SReal -> SBool
notEpsClose x = (x .< -eps) .|| (x .> eps)


nonZeroGradient :: RealMatrix -> SReal -> SReal -> SBool
nonZeroGradient interpolation x y = notEpsClose (xDerivative interpolation x y)
                                 .|| notEpsClose (yDerivative interpolation x y)
```

Finally we have the main proof function. This function takes as input a raw square, and a number t between 0 and 2 indicating which tautology we would like to use. This function defines the symbolic values for the potentials, as well as a symbolic point (x, y). In then asks the SMT solver to prove the formula "if the colour constraints hold, and (x, y) is in the $[0, 1]^2$ square, and tautology t holds, then the gradient is not less than or equal to ϵ at (x, y)".

```
proof :: RawSquare -> Int -> IO ThmResult
proof raw tautologyIdx = prove $ do
        setTimeOut 1000
        vars   <- symbolics ["red", "orange", "black", "green", "blue"]
        [x, y] <- symbolics ["x", "y"]
        let inter     = bicubic $ rawSquareToSquare raw vars
            tautology = tautologies x y !! tautologyIdx
        return $     colorConstraints vars
                 .&& inSquare x
                 .&& inSquare y
                 .&& tautology
                 .=> nonZeroGradient inter x y
```

Dealing with proof output. The proof returns a `ThmResult` that tells us the output of the solver. The following three functions tell us whether the SMT solver was successful or timed out, and in the case where it was successful, whether it proved or disproved the theorem.

```
successful :: ThmResult -> Bool
successful (ThmResult (Unknown _ _)) = False
successful (ThmResult (ProofError _ _ _)) = False
successful _ = True

proved :: ThmResult -> Bool
proved result = not $ modelExists result

disproved :: ThmResult -> Bool
disproved result = modelExists result
```

The following function verifies whether the theorem holds for a particular square. It runs `proof` three times, once for each tautology. It then checks whether any of those runs succeed, and if any run did succeed it returns the `ThmResult` from that run.

```
verifySquare :: RawSquare -> IO ThmResult
verifySquare square = do
    results <- mapM (proof square) [0..2]
    let finished = filter successful results
    when (any proved finished && any disproved finished) $
        error "solver returned inconsistent results!"
    return $ if not . null $ finished then head finished
                                      else head results
```

The boundary constraints. When a square lies on the boundary of the instance, there are extra constraints to check on the boundary of the square. Specifically, in addition to satisfying the theorem checked by `verifySquare`, we must also check that the direction that we follow as we decrease the gradient does not move us outside the instance. The following functions verify this condition, and they will be applied to any square that lies on the boundary of the instance.

We first define the `BoundarySide` type that encodes which side of the instance the square touches. For the four corners squares, which lie on two sides simultaneously, we will run the proof twice, once for each side.

```
data BoundarySide = BoundaryLeft
                  | BoundaryRight
                  | BoundaryBottom
                  | BoundaryTop
                  deriving (Show, Eq)
```

In this setting, we only care about the points that lie directly on the boundary. So, for a square that lies on the left side of the instance, we only need to consider the points (x, y) satisfying $x = 0$, and for those on the right side of the instance we only consider the points satisfying $x = 1$. The following function returns true for any point that lies directly on a given boundary of the instance.

```
sideConstraint :: BoundarySide -> SReal -> SReal -> SBool
sideConstraint BoundaryLeft   x _ = x .== 0
sideConstraint BoundaryRight  x _ = x .== 1
sideConstraint BoundaryBottom _ y = y .== 0
sideConstraint BoundaryTop    _ y = y .== 1
```

If a point lies on the boundary, we want the gradient to be larger than ϵ, while also not causing gradient descent to walk directly outside the instance. For a point on the left boundary of the instance, it is sufficient to check that either the y gradient has magnitude larger than ϵ, or that the x gradient is *negative* and less than $-\epsilon$. In the latter case, this ensures that gradient descent will walk into the instance, rather than outside the boundary.

The following function takes as arguments the boundary side, the x gradient, and the y gradient, and implements the appropriate check for the side that we are on. Note that these conditions are correct for all boundary points except the four corners of the domain, for which we have directly checked that they are not a solution in the paper.

```
sideTheorem :: BoundarySide -> SReal -> SReal -> SBool
sideTheorem BoundaryLeft   x' y' = x' .< -eps .|| y' .< -eps .|| y' .> eps
sideTheorem BoundaryRight  x' y' = x' .>  eps .|| y' .< -eps .|| y' .> eps
sideTheorem BoundaryBottom x' y' = y' .< -eps .|| x' .< -eps .|| x' .> eps
sideTheorem BoundaryTop    x' y' = y' .>  eps .|| x' .< -eps .|| x' .> eps
```

Finally, we can implement the proof for the boundary. The following function is similar to the `proof` function, but now we check the formula "if (x, y) is in the square and lies on the given boundary of that square, and the colour constraints hold, then the gradients must satisfy `sideTheorem`". Fortunately, we do not need any tautologies for the SMT solver to prove this theorem.

```
boundaryProof :: RawSquare -> BoundarySide -> IO ThmResult
boundaryProof raw side = prove $ do
        setTimeOut 1000
        vars   <- symbolics ["red", "orange", "black", "green", "blue"]
        [x, y] <- symbolics ["x", "y"]
        let inter = bicubic $ rawSquareToSquare raw vars
            xGrad = xDerivative inter x y
            yGrad = yDerivative inter x y
        return $     sideConstraint side x y
                  .&& colorConstraints vars
                  .&& inSquare x
                  .&& inSquare y
                  .=> sideTheorem side xGrad yGrad
```