AperTO - Archivio Istituzionale Open Access dell'Università di Torino

Compositional Blocks for Optimal Self-Healing Gradients

This is the author's manuscript

Original Citation:

Availability:

This version is available http://hdl.handle.net/2318/1655254 since 2018-12-16T18:30:55Z

Publisher:

Institute of Electrical and Electronics Engineers Inc.

Published version:

DOI:10.1109/SASO.2017.18

Terms of use:

Open Access

Anyone can freely access the full text of works made available as "Open Access". Works made available under a Creative Commons license can be used according to the terms and conditions of said license. Use of all other works requires consent of the right holder (author or publisher) if not exempted from copyright protection by the applicable law.

(Article begins on next page)

16 October 2024

This is the author's version of the contribution published as:

Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, Mirko Viroli. Compositional Blocks for Optimal Self-Healing Gradients. 2017. Proceedings - 11th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2017, 9 October 2017, Article number 8064033, Pages 91-100.

DOI: 10.1109/SASO.2017.18

The publisher's version is available at:

http://ieeexplore.ieee.org/document/8064033/

When citing, please refer to the published version.

The final publication is available at

http://ieeexplore.ieee.org

Compositional Blocks for Optimal Self-Healing Gradients

Giorgio Audrito*, Roberto Casadei†, Ferruccio Damiani* and Mirko Viroli†
*Computer Science Department and C3S, University of Torino
Email: {giorgio.audrito, ferruccio.damiani}@unito.it
†DISI Department, University of Bologna
Email: {roby.casadei, mirko.viroli}@unibo.it

***Abstract*—With the constant increase in the number of interconnected devices in today networks, and the high demand of adaptiveness, more and more computations can be designed according to self-organisation principles. In this context, a key building block for large-scale system coordination, called *gradient*, is used to estimate distances in a fully-distributed way: it is the basis for a vast variety of higher level patterns including information broadcast, events forecasting, distributed sensing, and so on. However, computing gradients is very problematic in mobile environments: the fastest self-healing gradient conceived so far (called BIS) achieves a reaction speed proportional to the single-path speed of information in the network. In this paper we introduce a new gradient algorithm, *SVD (Stale Values Detection) gradient*, which uses broadcasts to reach a reaction speed that is equal to the multi-path speed of information, namely, the fastest speed possibly achievable by network algorithms. We then combine SVD with other blocks (metric correction, smooth filtering, BIS gradient, information damping) proposing a composed block called *ULT(imate) gradient*. We evaluate the resulting algorithm and compare it with other approaches, showing it scores best both on accuracy and smoothness while keeping communication cost under control.**

***Index Terms*—adaptive algorithm; aggregate programming; computational field; gradient.**

I. Introduction

With the constant increase in the number of interconnected devices in today networks, more and more computations are carried on by an infrastructure involving a heterogeneous, potentially large, and very reactive set of computational nodes, each coupled with physical environment and providing sensing and actuation to interact with it. The technological landscape of such systems, as well as their high demand of adaptiveness and openness, call for the adoption of self-organisation principles in system design: there, one has to find proper abstractions and techniques to turn global-level requirements and properties into individual-level behaviour. In this setting, single computational devices have a partial knowledge of system and environment, and systematically share it with others in their (logical or physical) context or neighbourhood to globally co-create a faithful spatio-temporal representation of the "distributed state" of computation, on top of which collective adaptive behaviour can be achieved.

This work is partially supported by project HyVar which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644298, by ICT COST Action IC1402 ARVI, and by Ateneo/CSP project RunVar.

Several frameworks were proposed to ground this approach with models and tools facilitating system construction, one survey of which may be found in [1]. For goals and proposed paradigm they can be divided in different groups: works abstracting from device interaction and making it implicit in the distributed computational model (e.g., TOTA [2] and Hood [3]), information systems tackling the problem of reading/streaming/writing space-time regions of sensed data (e.g., KQML [4] and TinyDB [5]), approaches seeking at defining geometric and topological constructions (e.g., ASCAPE [6] and Origami Shape Language [7]), works considering computation as transformation of space-time data structures (e.g., Protelis [8], Proto [9], and MGS [10]), and finally attempts to devise catalogues of bio-inspired patterns [11]–[13].

On the one hand, these works suggest that self-organisation can be engineered by a compositional approach, where reusable distributed algorithms or components, providing basic functionalities, are safely composed to construct more and more complex and specialised behaviour, from low-level general libraries to application-specific services [14]. In particular, one of such blocks, called *gradient* (widely used, e.g., in [2], [8], [9], [11]), grounds a significant set of distributed collective systems. It is used to create a global distributed data structure to estimate distances in a fully-distributed way, and it is the basis for a vast variety of higher level patterns including information broadcast, events forecasting, distributed sensing, steering mobile actors in physical environments, and so on.

On the other hand, previous works also show that non-functional properties seriously affect the attempt at finding good implementations of such components [15]. In general, networks of devices deployed in space easily result in very dynamic scenarios, featuring continuous nodes mobility, changes in the shape of network topology, temporary and permanent faults, noisy perception of sensors, and so on. Altogether, these factors can make it very complicated to reach a stable and faithful representation of global knowledge. In general, one seeks for solutions where the computational systems can properly trade off reactiveness to changes with the resources devoted to intercept and execute adaptation.

As a paradigmatic example, computing gradients is very problematic in mobile environments: the classical approach of iterating the triangle inequality [16] suffers from speed problems when estimating increasing distances, and with reactiveness problems with mobile networks, which are only partially alleviated by the solutions proposed so far (e.g., by Beal [16], [17]); more recently, the BIS (Bounded Information Speed) gradient algorithm has been proposed by Audrito et al. [18] to optimise gradient self-healing, by achieving a reaction speed proportional to the single-path speed of information in the network.

In the path towards finding an "ultimate" implementation for gradients, providing tunable performance in diverse situations, we introduce a new gradient algorithm, *SVD (Stale Values Detection) gradient*. It uses broadcasts to reach a reaction speed that is equal to the multi-path speed of information, namely, the fastest speed possibly achievable by network algorithms. We then combine SVD with other blocks (metric correction, smooth filtering, BIS gradient, information damping) proposing a composed block called *ULT gradient*, acting as a candidate reference implementation for gradients in libraries of reusable components of distributed system behaviour.

We evaluate the resulting algorithm and compare it with other approaches, showing it scores best both on accuracy and smoothness while keeping communication cost under control.

The remainder of this paper is organised as follows: Section II reviews aggregate programming and the field calculus, which give a ground to formalise the proposed algorithms, Section III discusses existing gradient algorithms, Section IV develops SVD algorithm and states its optimality, Section V defines ULT gradient by composition of various blocks, Section VI empirically evaluates ULT and compares it with other approaches, and Section VII concludes with final remarks.

II. Aggregate Programming and Field Calculus

The gradient algorithms discussed in this paper can be implemented on top of a variety of computational models and frameworks: it is only assumed that devices work asynchronously and can interact with neighbours via broadcast-like communication. In presenting motivations, design and technical details, though, we shall use aggregate programming [14] as a sort of common model, a *lingua franca* with formal semantics, succinctly expressing the required mechanisms and algorithms.

Aggregate programming is an approach for designing resilient distributed systems by abstracting away from individual devices behaviour and focusing on the global, aggregate behaviour of the collection of all (or a subset of) the devices. This approach provides smooth composition of distributed behaviour, allowing the development of highly reusable "building block" operators capturing common coordination patterns [19], thus raising the abstraction level and allowing programmers to work with general-purpose functionalities or user-friendly APIs capturing common use patterns. These building blocks may have several possible implementations, equivalent for their "functional" behaviour but differing in their performance and resilience properties. The choice of a

$\mathtt{P}$	$::=$	$\overline{\mathtt{F}}\ \mathtt{e}$	program
$\mathtt{F}$	$::=$	$\mathtt{def}\ \mathtt{d}(\overline{\mathtt{x}})\ \{\mathtt{e}\}$	function declaration
$\mathtt{e}$	$::=$	$\mathtt{x} \mid \mathtt{v} \mid \mathtt{e}(\overline{\mathtt{e}}) \mid \mathtt{if}(\mathtt{e}_0)\{\mathtt{e}_1\}\{\mathtt{e}_2\}$ $\mid \mathtt{rep}(\mathtt{e})\{(\mathtt{x}) \texttt{=>} \mathtt{e}\} \mid \mathtt{nbr}\{\mathtt{e}\}$	expression
$\mathtt{v}$	$::=$	$\phi \mid \ell$	value
ϕ	$::=$	$\overline{\delta} \mapsto \overline{\ell}$	neighbouring field value
ℓ	$::=$	$\mathtt{c}(\overline{\ell}) \mid \mathtt{f}$	local value
$\mathtt{f}$	$::=$	$\mathtt{b} \mid \mathtt{d} \mid (\overline{\mathtt{x}}) \texttt{=>} \mathtt{e}$	function value

Fig. 1. Syntax of Field Calculus.

"best" implementation may depend on the specific application contest at hand, thus suggesting a recently proposed [19] two-phases engineering workflow where *(i)* a specification is first implemented by composing "building blocks", and then *(ii)* the implementation is tuned by substitution of functionally equivalent but higher-performance coordination mechanisms. Proposing and evaluating different alternatives for the *gradient* block would be the matter of this work.

A. Field Calculus

The Field Calculus is a tiny functional language for formally and practically expressing aggregate programs: a detailed account of it is given in [20]—we hereby recollect its most basic characteristics in order to be able to use it as common language to express the algorithms used in this paper with actual executable code.

In field calculus (see Fig. 1) a program $\mathtt{P}$ consists of a sequence of function declarations $\overline{\mathtt{F}}$ and of a main expression $\mathtt{e}$—following [21], the overbar notation denotes metavariables over sequences: e.g., $\overline{\mathtt{e}}$ ranges over sequences of expressions $\mathtt{e}_1, \ldots, \mathtt{e}_n$ with $n \geq 0$. On each device δ the expression $\mathtt{e}$ evaluates to a value $\mathtt{v}$ that may depend on the state of δ (values sensed by δ, the result of previous evaluation, and information coming from neighbours). Therefore the expression $\mathtt{e}$ induces a *computational field* Φ, which can be represented as a time-varying map $\delta_1 \mapsto \mathtt{v}_1, \ldots, \delta_n \mapsto \mathtt{v}_n$, assigning a value $\mathtt{v}_i$ to each device δ_i in a network. Each device δ updates its value (by evaluating $\mathtt{e}$) in asynchronous computational rounds. The syntax of an expression comprises the following constructs:

- a variable $\mathtt{x}$, used as function formal parameter;
- a value $\mathtt{v}$, which in turn could be either a *neighbouring field value* ϕ (associating to each device a map from neighbours to local values—allowed to appear in intermediate computations but not in source programs), or a *local value* (built-in functions $\mathtt{b}$, user-defined functions $\mathtt{d}$, anonymous functions $(\overline{\mathtt{x}})\texttt{=>}\mathtt{e}$, or plain values—numbers, literals, tuples and so on);
- a function call $\mathtt{e}(\overline{\mathtt{e}})$ where an expression of functional type $\mathtt{e}$ is applied by-value to arguments $\mathtt{e}_1, \ldots \mathtt{e}_n$;
- an $\mathtt{if}$-expression $\mathtt{if}(\mathtt{e}_0)\{\mathtt{e}_1\}\{\mathtt{e}_2\}$, modelling domain restriction, which computes $\mathtt{e}_1$ in the devices where $\mathtt{e}_0$ is true, and $\mathtt{e}_2$ on the others;
- a $\mathtt{rep}$-expression $\mathtt{rep}(\mathtt{e}_1)\{(\mathtt{x})\texttt{=>}\mathtt{e}_2\}$, modelling time evolution and state preservation, which evaluates an ex-

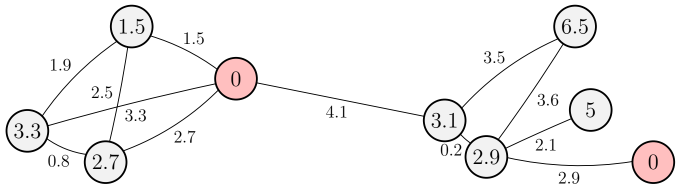

Fig. 2. Gradient computed in a sample network with two source devices.

pression $\mathtt{e}_2$ on a device δ by substituting the variable `x` with the value calculated for the whole `rep`-expression at the previous computational round on δ—in the first computation round on δ (as well as in any computational round on δ that comes immediately after a computational round on δ where the `rep`-expression has not been evaluated)[1] `x` is substituted with the value of $\mathtt{e}_1$;

- or an `nbr`-expression `nbr{e}`, modelling neighbourhood observation and producing a neighbouring field value that represents an "observation map" of neighbour's values for expression `e`, namely, associating to each device δ (that has evaluated `nbr{e}` during its last update)[2] a map from neighbours to their latest evaluation of `e`.

In order to conveniently present the code of the algorithms, we extend the calculus with few additional constructs, which can be defined as syntactic sugar in terms of the presented ones: let-binding expressions `let x =` $\mathtt{e}_1$ `in` $\mathtt{e}_2$ (operationally equivalent to `((x)=>`$\mathtt{e}_2$`)`$\mathtt{e}_1$), binary operators in infix notation, and the notation $[\mathtt{e}_1, \ldots, \mathtt{e}_n]$ for tuples.

III. Gradient Algorithms

A fundamental "building block" commonly considered in several frameworks for self-organisation is the so-called *gradient* computation [16], which amounts to computing shortest paths from all nodes to a given set of source nodes, through a fully distributed process to be iteratively executed to promptly react to any change in the environment. Gradients are known to be a basic building block for self-organising coordination [11], [17], [22]–[24], being frequently used for a variety of purposes: to broadcast information, forecast events, dynamically partition networks, ground distributed sensing, anticipate future events, and to combine into higher-level spatial structures [1].

Due to their usefulness, gradients have been studied in many different contexts: in particular, when local estimation of distances is available [16]–[18] and when it is not [22], [25], [26]. In the remainder of this paper we shall focus on the first case, thus assuming that local device-to-device estimation of distances is always available.

A. Classic Gradient

In its most basic form the gradient can be calculated through an algorithm, that we call *classic* gradient, starting

[1]For instance, if the main expression is `if(`$\mathtt{e}_0$`){rep(`$\mathtt{e}_1$`){(x)=>`$\mathtt{e}_2$`}}{`$\mathtt{e}_3$`}`, then a computational round on a given device δ evaluates `rep(`$\mathtt{e}_1$`){(x)=>`$\mathtt{e}_2$`}` if and only if $\mathtt{e}_0$ evaluates to true in that computational round.

[2]For instance, if the main expression is a an `if`-expression `if(`$\mathtt{e}_0$`){`$\mathtt{e}_1$`}{`$\mathtt{e}_2$`}` and the expression `nbr{e}` occurs the left branch $\mathtt{e}_1$, then the neighbours of δ such that $\mathtt{e}_0$ evaluated to false in their last computational round are not considered.

with $G(\delta) = \infty$ everywhere except for sources where $G(\delta)$ is constantly 0, hence applying the triangle inequality constraint:

$$G(\delta) = \min\{G(\delta') + w(\delta', \delta) : \delta' \text{ linked with } \delta\}.$$

Written in field calculus, it would be a function taking a 0-ary function `metric` which computes a field of distances with neighbours, and a value `source` appointing source devices, represented as the "gradient estimate in absence of communication links" (thus `source` should be 0 for source devices and ∞ for all the others):

```
def classic(source, metric) {// has type: (float, ()→field(float)) → float
  rep (infinity) { (dist) =>
    mux(source==0){minhood(nbr(dist) + metric())}}}
```

Repeated fair application (i.e., each device updates infinitely often) of this calculation in a fixed network will converge to the correct value at every point [27]. However, the performance of this algorithm in a mutable environment is impaired by three main limitations [18].

First, there can be a *speed bias*: if devices are continuously moving, the values produced by the algorithm are typically an underestimation, with an error that increases with the movement speed—a phenomenon observed also in those contexts where local distance estimates are not available [22]. Second, it suffers from the *rising value* problem (also known as *count-to-infinity* in the context of routing algorithms [28]): in response to changes in the network, the algorithm can rapidly correct values that need to drop, while it is very slow in correcting values that need to rise, badly underestimating distances for long periods of time after such changes—the rising speed of this algorithm is bounded by the distance between the pair of closest devices [16]. Third, there is an issue of *smoothness*: in the presence of error in distance estimates, flickering in the output values might be reduced by avoiding strict use of the triangle inequality. Moreover, if distance estimates are used for an higher-order coordination mechanism (e.g., for moving values towards sources by "descending" the shortest-paths tree obtained from the gradient), then each variation in the estimates can change the resulting connection tree, effectively disrupting the outcome of the coordination for some time.

In order to overcome these limitations, several refined algorithms have been proposed, two of which are briefly discussed in the following for they contribute in the definition of ULT gradient: Beal's FLEX gradient (Flexible) [17] and Audrito et al.'s BIS gradient (Bounded Information Speed) [18].

B. FLEX Gradient and Information Damping

The FLEX gradient [17] is designed to improve smoothness through application of a "filtering function" to the outcome of the minimisation, which reduces changes while granting an overall error of at most a given parameter ϵ. Assume that we are repeatedly computing a value by selecting a corresponding neighbour's value and adding to it a certain differential Δ (e.g. a metric in a gradient computation). According to this viewpoint, we can define (a simplified version of) FLEX as a damping function F, taking as arguments the old value $\mathtt{v}_{\text{old}}$, the new value v_{new}, and the differential Δ; and as parameters an allowed error ϵ and a time period T.

$$F(\mathrm{v}_{\text{old}}, \mathrm{v}_{\text{new}}, \Delta) = \begin{cases} \mathrm{v}_{\text{new}} & \text{if } \mathrm{v}_{\text{old}} > 2\mathrm{v}_{\text{new}}, \mathrm{v}_{\text{new}} > 2\mathrm{v}_{\text{old}}, \\ & \text{or } T \text{ elapsed since a change} \\ \mathrm{v}_{\text{new}} + \frac{\epsilon\Delta}{2} & \text{if } \mathrm{v}_{\text{old}} > \mathrm{v}_{\text{new}} + \epsilon\Delta \\ \mathrm{v}_{\text{new}} - \frac{\epsilon\Delta}{2} & \text{if } \mathrm{v}_{\text{old}} < \mathrm{v}_{\text{new}} - \epsilon\Delta \\ \mathrm{v}_{\text{old}} & \text{otherwise} \end{cases}$$

Since a maximum distortion of ϵ is allowed for the differential Δ, the total error introduced by the damping is also bounded by ϵ. Since the result of damping changes only when violating the maximum error (or every T time, for finer adjustments), volatility of values is reduced and communication costs can also be possibly reduced, assuming that only changing values are broadcast to neighbours (thus absence of a broadcast is interpreted as a steady value).

Although outperformed by the following BIS gradient, the technique used by FLEX can also be understood as a *plug-in* applicable to many gradient-like computations, granting reduction of communication cost and volatility at the expense of a controlled increase in error.

C. BIS Gradient

The *Bounded Information Speed* (BIS) gradient improves over the classical gradient and FLEX by enforcing a minimum information speed v requested by the user [18]. As long as v does not surpass the average single-path communication speed, the algorithm is able to compute correct estimates of the gradient with increased responsiveness. Values of v that are too much higher induce a metric distortion, causing the algorithm to overestimate values.

For each device in the network, we compute both the usual gradient estimate $G(\delta)$ and a lag estimate $L(\delta)$, representing the time elapsed since the message initially spread from a source. Lags are estimated through local time differences, so that no overall clock synchronisation is required. When considering a candidate neighbour δ' of a device δ, the time lag relative to this neighbour is:

$$L(\delta, \delta') = L(\delta') + \lambda(\delta', \delta)$$

where $\lambda(\delta', \delta)$ is the lag of the message from δ' to δ. We then take into account this value when calculating the gradient estimate relative to this neighbour:

$$G(\delta, \delta') = \max\{G(\delta') + w(\delta', \delta),\ vL(\delta, \delta') - r\}$$

where w is the distance between devices and r is the communication radius. This formula accounts to assuming that messages propagate at least at speed v, so that the gradient estimate is lower bounded by $vL(\delta, \delta')$ (with the additive constant $-r$ to ensure that some error is taken into account).

The overall estimates of $G(\delta)$ and $L(\delta)$ for non-source devices are then obtained by minimising $G(\delta, \delta')$ over neighbours (we assume that pairs are ordered lexicographically):

$$[G(\delta), L(\delta)] = \min\{[G(\delta, \delta'), L(\delta, \delta')] :\ \delta' \text{ linked with } \delta\}$$

The performance of BIS gradient can be theoretically determined, and proved to be optimal among algorithms with a single-path information flow whenever the parameter v is close to the average single-path information speed.

Theorem 1 (BIS Performance Bound [18])**.** *Information speed in BIS gradient, calculated w.r.t. the gradient estimates, is at least v. Furthermore, values constrained by obsolete information increase at least at speed v.*

Theorem 2 (BIS Optimality [18])**.** *The BIS gradient with v equal to the average single-path information speed v_{avg} attains optimal reactivity among algorithms with a single-path information flow.*

It is thus crucial to correctly determine the average single-path information speed v_{avg} of a network, in order to achieve the best performance. In [18] an estimate of the average information speed over a single hop is given. Over multiple hops, speeds do not simply add up: in fact, the expected single-path information speed changes and can be better approximated by the easier (experimentally verified) formula $v_{\text{avg}} = R/3T$ where R is the average communication radius and T is the average interval between computation rounds. It follows that if R and T are known to the self-organising system designer, v_{avg} can be readily calculated. If they are not available, or cannot assumed to be constant throughout the system's life, they can still be computed by a simple aggregate program: e.g., by Laplacian averaging over the maximum distance and average time interval measured.

IV. SVD Gradient

Since BIS gradient is optimal among algorithms with a single-path information flow, to obtain a faster self-healing algorithm it is necessary to use some form of multi-path communication, namely, where the value at a device is constructed by the result of flows coming from different paths connecting to the source. This intuition leads us to the *SVD (Stale Values Detection) gradient*, which detects obsolete information through broadcast of time data, propagating a "reconfiguration" message whenever such situation occurs (through broadcast as well). In fact, SVD gradient behaves as the classic gradient until an "error" is detected and fast reconfiguration occurs; thus differing from BIS gradient which gradually adapts to network changes.

SVD has the same inputs `source` and `metric` of classic gradient seen in previous section, but it rather calculates four values in each device:

- `x`, the distance estimate;
- `sid`, the identifier of the source device from which distance `x` is attained;
- `t`, the multi-path time interval from source `sid`; i.e., the interval between the device's current time and the most recent information from `sid` that reached the current device through broadcast;
- `obs`, whether the current values have been detected as obsolete and thus should not be used to calculate other device's values.

These values are updated in a `rep` statement, combined into a single tuple `old = [x,t,sid,obs]`, where only `x` is finally returned by the algorithm (by operator `1st` extracting the first component):

```
def SVD(source, metric) { // has type: (float, ()→field(float)) → float
  let loc = [source, source, uid, false] in
  1st(rep (loc) { (old) =>
    let xs = minhood(mux(
      nbr{4th(old)} and anyhood(nbr{not 4th(old)}),
      [source,uid], [nbr{1st(old)}+metric(),nbr{3rd(old)}]
    )) in
    let t = minhood(mux(nbr{3rd(old)} != 2nd(xs),
      source, nbr{2nd(old)} + lagmetric()
    )) in
    let loop = 2nd(xs) == uid and 1st(xs) < source in
    let obsolete = detect(time()-t) or loop or anyhood(
      nbr{4th(old)} and nbr{3rd(old)}==2nd(xs) and
      nbr{2nd(old)}+lagmetric() < t+0.0001
    ) in
    min(loc, [1st(xs), t, 2nd(xs), obsolete])
  })
}
```

The code above computes the tuple `loc` and then uses it as initial value for the `old` variable in the `rep`-expression. Both `x` and `t` are distances (spatial and temporal, respectively) from the source, thus their initial values are the same: ∞ if there are no sources, and 0 if the device is a source. The body of the `rep`-expression performs the following steps:

- First, the new values for `x` and `sid` are calculated together into variable `xs`. This step follows the classic gradient calculation, thus minimising neighbours' values for `x` plus their relative distance, but discarding neighbours where `obs` is true (unless `obs` is true for all neighbours).
- Then, the most recent information `t` for the new `sid` is retrieved through minimising neighbours' values for `t` plus their relative time distance, discarding neighbours with a different `sid`.
- Finally, it is computed in `obsolete` whether the newly produced information is to be considered obsolete. This happens in three cases:
  - if any neighbour with the same `sid` and a `t` not older than the device's one[3] has already been claimed obsolete (`anyhood` expression);
  - if the device's value happens to be calculated from itself (i.e. `sid` is the device itself), with a value smaller than the one currently determined by `source`;
  - if function `detect` realises that the time `time()` − `t` when the currently used information started from `sid` is too old to be reliable.

Function `detect` is the heart of the algorithm, being responsible to kick-start the reconfiguration process.

```
def detect(time) { // has type: float →  boolean
  let repcnt = rep (0) { (old) =>
    if (abs(time-delay(time)) < 0.0001) {
      old + deltatime()
    } {0}
  } in
  repcnt > 3rd(rep ([0, 0, 0]) { (old) =>
    if (repcnt > 0) { old } {
      let avg = 0.9*1st(old) + 0.1*delay(repcnt) in
      let sqa = 0.9*2nd(old) + 0.1*sqr(delay(repcnt)) in
      let dev = sqrt(sqa - sqr(avg)) in
      [avg, sqa, avg+7*dev]
    }
  })
}
```

This function keeps track into `repcnt` of how much time is elapsed since the first time the current information (originated from the source in time `time`) reached the current device: if `time` is the same as its previous value `delay(time)`,[4] the time interval `deltatime()` between the current time and the previous round is added to the counter, which is reset otherwise. If a source device `sid` gets disconnected from the network, all the `repcnt` values of devices dependent on `sid` constantly increase; while in a regular situation they raise and reset randomly, approximately according to an exponential distribution.

Function `detect` then raises an alert whenever `repcnt` reaches a value that is unlikely to happen according to its previously measured behaviour. This is checked by tracking:

- `avg`, an estimate of the average *peak* value for `repcnt`; obtained by exponentially filtering with a factor 0.1 the peak values of `repcnt` (obtained as `delay(repcnt)` when `repcnt` is reset to 0);
- `sqa`, an estimate of the average *square* of `repcnt` peak values, obtained by an analogous exponential filter.

From them the standard deviation σ can be calculated as $\sqrt{\texttt{sqa} - \texttt{avg}^2}$, and a bound is set to $\texttt{avg} + 7\sigma$. Under an exponential distribution assumption, this bounds corresponds to the $1 - e^{-8} \simeq 99.97\%$ quantile (under a normality assumption would correspond to approximately the $1 - 10^{-12}$ quantile). The mean and square-mean estimators are initially set to 0, in order to avoid need of an additional parameter; which implies that alerts are continually raised for some rounds on network start-up. This could be avoided by setting the initial value to any reasonable value (such as $[2T, 8T^2, 16T]$ where T is the average time interval between rounds).

A. Expected Performance and Parameters Estimation

Thanks to its multi-path communication structure, SVD gradient can attain a globally optimal performance.

Theorem 3 (SVD Optimality)**.** *SVD gradient attains optimal reactivity on input discontinuities, reaching multi-path reaction speed, modulo a fixed delay ℓ which depends on some network configuration parameters.*

Proof. When a new source is connected, reaction speed is multi-path for any gradient algorithm: values are recomputed towards the new source using every possible available path. Assume that a source s is disconnected in time t_0, and let δ be the device holding the smallest bound ℓ computed in function `detect`, among devices depending on s. The time information originating from s in t_0 reaches δ at multipath

[3]That is, adding the neighbour's `t` with the relative time distance, we obtain an interval not longer than the current `t`.

[4]Function `delay(v)` is a simple building block which returns the same values it receives in input delayed by one computation round.

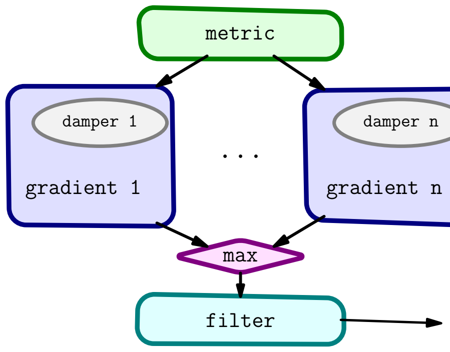

Fig. 3. How the different blocks are composed (arrows represent data flow).

speed, since time distance is calculated through minimising among every neighbour (i.e., broadcasting). After that moment t_1, the temporal distance estimate in δ raises constantly as $t = t_{\text{cur}} - t_0$.

When t overcomes $t_1 + \ell$, function `detect` has been fed with the same reference time $t_{\text{cur}} - t = t_0$ for ℓ time, thus `repcnt` $> \ell$ and an alert `obs` is raised. The alert information propagates on every possible path as well, until reaching the border of the region depending on s; then information bounces back again at multi-path speed recomputing distances using other sources.

Thus information travels from the source to the border of the region (and back) at multi-path speed, covering a space that is equal to the amount which the gradient estimates needs to rise, *except* for ℓ time during which the algorithm is waiting for an alert: in other words, reconfiguration is happening at multi-path speed with a delay ℓ. □

Thus, the average delay ℓ corresponds to the average "lowest bound computed by `detect`" on devices connected to a source. Extensive simulations revealed that those bounds were always below $2T + 8\sigma + 8vT^2/R + dT/\sqrt{N}$, where T is the average time interval between rounds, σ is the standard deviation of time intervals, v is the average movement speed of devices, R is the average communication radius, N is the average number of neighbours, and d is the current gradient estimate. Bounds tend to increase with the distance from the source, as reflected by term $\frac{d}{\sqrt{N}}T$. However, since the delay corresponds to the *lowest* bound in the region, this term can be ignored obtaining a bound for ℓ of $2T + 8\sigma + 8vT^2/R$. The average value for ℓ is practically much lower due to the minimisation, even if quite difficult to estimate precisely (usually between T and $5T$, that is, between 1 and 5 rounds).

V. Composite Gradient

In Section III we presented several issues about gradient computations. In order to fully solve these issues, it is necessary to combine different techniques altogether:

- *metric correction*, in order to prevent the *speed bias*;
- *output filtering*, in order to improve *smoothness*;
- *combining fast self-healing algorithms*, in order to react to input discontinuities and network changes;
- *FLEX-style damping*, in order to decrease communication cost and improve *smoothness* (at the expense of some controlled error).

We now present original techniques addressing all these points, in some cases built upon existing ones. These techniques can be combined together (as depicted in Fig. 3) to form an "ultimate" gradient algorithm, which we call *ULT gradient*, where each of this components can be tuned (or even deactivated) depending on the application context.

A. Metric Correction

When devices are moving, classical gradient algorithms systematically underestimate values according to the so-called *speed bias*. This is due to the fact that metric estimates are calculated in a specific time, and then applied to compare events happening in different times. More precisely, taking the field calculus setting as reference:

- the metric estimate is computed at time t_1 when device δ_1 sends his message to device δ_2;
- it is used in the following update event in δ_2, happening at some time $t_2 > t_1$;
- it relates to data produced in the *preceding* update event in δ_1, happening at some time $t_0 < t_1$.

Thus the distance between the relevant update events in δ_1 and δ_2 is not correctly measured, introducing some random error which results in systematic underestimation due to the "minimisation" nature of gradient algorithms.

This issue can be completely solved in presence of GPS data, where distance between events can be computed exactly by comparing absolute positions, as in function `gpsrange`.

```
def gpsmetric() { // has type: () → field(float)
  let pos = coordinates() in
  norm(pos - nbr{delay(pos)}) }
```

Even if GPS data is not available, but some form of speed estimation is available (either in terms of sensors or pre-computed constants), we can modify the metric in order to guarantee that distances are never underestimated.

```
def speedmetric(speed) { // has type: float → field(float)
  nbr{speed*deltatime()} + nbrrange() + speed*nbrlag() }
```

In `speedmetric`, the built-in metric `nbrrange` (giving a map from neighbours to estimated distances) is adjusted both by the space travelled by neighbour devices in their last round (left) and by the space travelled by the current device since the metric data was received (right). Thus, it is guaranteed to produce overestimates of real values, which turns out in a drastically improved precision (also due to the "minimisation" carried by gradient algorithms).

B. Smooth Filtering

In order to improve *smoothness*, it is possible to directly smooth the output of a gradient algorithm with *filters*. The most common type of filters, *exponential filters* (e.g., the one used in function `detect`, Section IV) indeed succeed to drastically improve smoothness, and its use can be suggested in nearly-static environments. However, if devices are moving significantly, exponential filters tend to delay the gradient adjustments thus producing a significant increase in error. An alternative approach is given by the following *inertial filter*, which is able to improve smoothness with lower error increase.

```
def inertialfilter(val, factor) { // has type (float,float) → float
  let dt = deltatime() in
  let at = expfilter(dt, factor) in
  let ad = expfilter(abs(val - delay(val)), factor) in
  rep (val) { (old) =>
    if (isfinite(old)) {
      let v = sign(val-old)*min(abs(val-old)/dt,ad/at) in
      old + v * dt
    } {val}
  }
}
```

This function proceeds in the following steps:

- firstly, an estimate `at` of the average time interval between rounds `deltatime()` is kept through exponential filtering (represented by function `expfilter`);
- an estimate `ad` of the average value change between rounds is obtained through exponential filtering as well;
- these two estimates are combined into a "change speed" estimate $v_{\text{avg}} = \texttt{ad}/\texttt{at}$, and the current value change is slowed down to v_{avg} if needed.

Notice that a uniform linear value change is followed closely by the inertial filter without a delay; and as the given factor approaches 0, the filter smooths out values to a uniform linear change. In fact, in a variable environment with significant movement speeds, uniform linear change can be regarded as the smoothest possible behaviour.

C. Self-Healing Algorithms

In order to obtain correct values at all times, it is crucial for the underlying gradient algorithms to attain fast reaction speeds. Audrito at al. [18] showed that BIS gradient is able to outperform both classic, CRF and FLEX gradients under any environment and input conditions (see Section III-C). In Section IV we presented SVD gradient, and argued that it is able to attain an optimal reaction speed on input discontinuities, even faster than that of BIS. However, it is not true that SVD outperforms BIS under any environment and input conditions.

- BIS instantly reacts to input changes with continuous small adjustments; whereas SVD behaves like the classic gradient until a sudden reconfiguration happens, thus having worse values in the start-up phase and worse smoothness overall. On the other hand, SVD values become more precise for some time after a reconfiguration.
- In networks with low device density and space/time variability, multi-path information speed can be very close to single-path information speed; possibly reducing SVD advantage to being negligible. On the other hand, dense networks with variability can have very different multi-path and single-path speeds.
- Finally, SVD is more resource-demanding than BIS due to a more complex code.

It follows that whenever resource consumption is strictly bounded, or the network has low density and variability, BIS is overall preferable. In all the other cases, it is advisable to take the "best" from both algorithms, as in the following *ULT gradient*.

```
def ULT(source, metric, radius, speed, factor) {
  // has type (float, ()→field(float), float, float, float) → float
  let svd = SVD(source, metric) in
  let bis = BIS(source, metric, radius, speed) in
  inertialfilter(max(svd, bis), factor)
}
```

Through maximising between BIS and SVD gradients, ULT is able to instantly react to continuous small adjustments, snapping up to correct values when input discontinuities are detected, thus always outperforming both BIS and SVD (at the expense of an increased code complexity/resource demand).

In the above code and in the remainder of the paper, we assume that ULT gradient also incorporates *metric correction* and *smooth filtering*, in order to achieve the best results under every environment and input conditions.

D. FLEX-style Damping

In Section III-B we pointed out that FLEX gradient [17] can be seen as a "damping" plug-in F for gradient-like computations [18]. It can thus be applied to BIS, SVD and ULT, both for their spatial components (e.g., `x` in SVD) and for their temporal components (e.g., `t` in SVD), obtaining improved smoothness and reduced communication costs.

VI. Empirical Evaluation

In order to compare the performance of the different gradient algorithms presented in this paper, we chose an environment able to trigger the issues presented in Section III (speed bias, rising value, smoothness). Following the guidelines presented by Audrito et al. [18], we thus tested the following scenarios.

- *Environment:* we initially put 1000 devices with communication radius $10m$ and average update rate $1s$ randomly into a $500m \times 20m$ corridor, producing a network 50 hops wide. We tested this environment with increasing *variability* and *noise* both in space and in time. Several conducted tests revealed that the effect of increasing variability or noise (both in space or in time) is similar, so we decided to report graphs with simultaneously increasing variability of both types. We ranged the parameter from 0 (no movement, uniform update rate) to 1 (long range movements together with high-frequency Brownian motion, 50% relative standard error in update rate between different devices plus another 50% in each device). We modelled randomly distributed events according to a Weibull distribution [29].
- *Input:* we provided the algorithms with two sources, steadily located on both ends of the corridor until time 200, when the left source was abruptly disconnected. In this way, reaction to discontinuous input is measured (in the middle of the graphs) as well as behaviour under constant input (at the sides of the graphs).
- *Output:* for each scenario we measured *precision* as absolute error w.r.t. Euclidean distance, *stability* (or smoothness) as the absolute acceleration of values (i.e., the

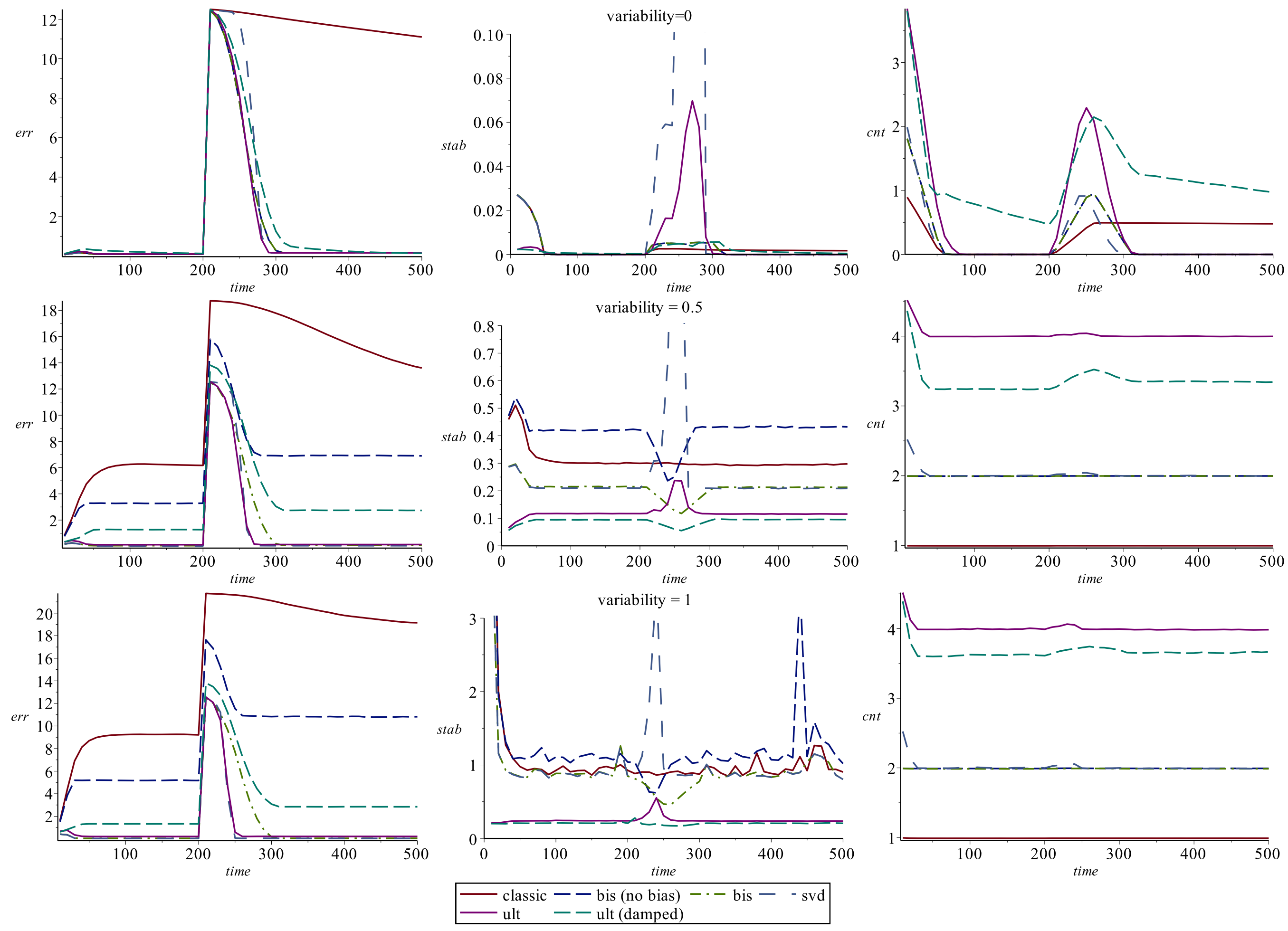

Fig. 4. Performance comparison between gradients algorithms. Variability levels (both in space and time) range from 0 (top) to 1 (bottom). In each environment, we measured absolute error (left), stability of values (centre) and communication cost as number of exports (right). In each graph, we can see the impact of input discontinuity (at time 200) as well as behaviour under steady conditions (sides of each graph).

absolute value of the second derivative of the computed gradient values), and *communication cost* as the number of values exchanged with neighbours (assuming that steady values do not need to be broadcast).

A. *Impact on gradient formation*

Figure 4 summarises the evaluation results, which were obtained (like all the experiments reported in this paper) with Protelis [8] (an incarnation of the Field Calculus [20]) as programming language to code the model, Alchemist as simulator [30] and the Supercomputer OCCAM [31] to run the experiments. We tested classic, BIS with and without speed bias correction ($1.5v_{\text{avg}}$ speed parameter), SVD with parameters auto-tuning, ULT with and without FLEX damping (20% damping tolerance, 0.2 filter factor). We run 20 instances of each scenario with different random seeds and averaged the results, which had a 16% relative standard error among them—thus not affecting the result of our evaluation.

The addition of speed bias correction allowed to virtually eliminate the systematic error both on algorithms BIS and SVD (and their combination ULT). SVD becomes increasingly faster than BIS in recovering from discontinuities as variability increases, and the response delay of SVD proves to be unnoticeable from the graphs. ULT is able to fully match the performance of SVD: the filtering factor 0.2 does not introduce significant additional error. SVD has the worst peak in stability, which is effectively mitigated by the filter included in ULT; so that ULT achieves the better stability scores in high variability scenarios. FLEX damping allows to further slightly improve stability and reduce communication cost, at the expense of a predictable systematic error: a trade-off which might still be convenient when the gradient values are used to directing information flows (see Section VI-B).

In order to better analyse the conditions under which SVD becomes preferable than BIS, we also run tests on these two algorithm while varying parameters of the environment:

- **Length**. We varied the length of the corridor from $250m$ to $1500m$, while keeping density fixed (thus varying the number of devices from 500 to 3000).
- **Radius**. We varied the communication radius of devices from $5m$ to $15m$.

Simultaneously, we also considered variabilities from 0 to 1 as in the previous graphs. We run 10 instances of each scenario with different random seeds and averaged the results, which had a 3.7% relative standard error among them. The results are

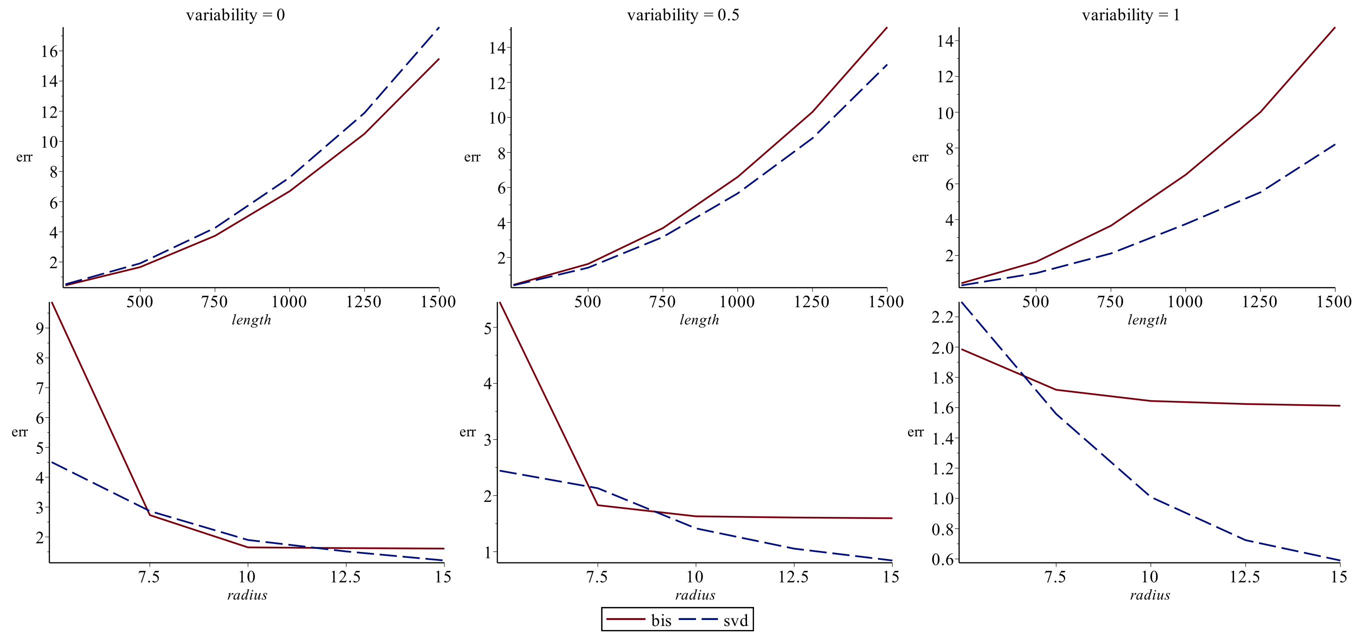

Fig. 5. Performance comparison between BIS and SVD gradients, varying length of the corridor (top) and communication radius (bottom), under variabilities from 0 (left) to 1 (right). Each point measures the average total error from time 0 to 500.

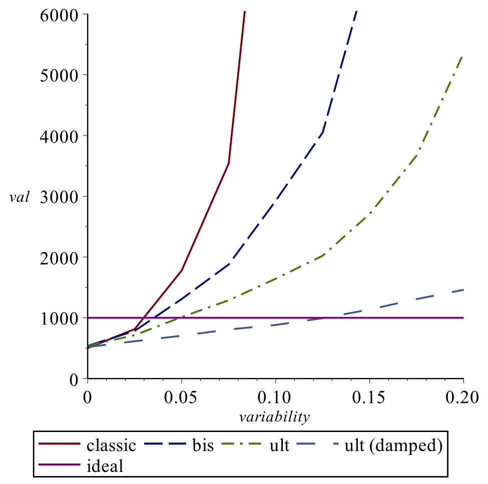

Fig. 6. Values for node counting with different gradient algorithms, taken after disconnection of a source (sink), for different variability levels.

reported in Fig. 5. As already shown in the previous graphs, an increase in variability gives advantage to SVD. The length of the corridor does not influence the relative performance of the two algorithms. The communication radius, instead, has a larger impact: when it is too low, multi-path communication reduces to single-path giving advantage to BIS; when it is higher SVD error converges towards 0 while BIS is not noticeably influenced.

B. Evaluation of Gradients as Components

In order to better grasp the impact of the various algorithms we now consider a main application of gradients, namely, as carriers for distributed sensing. From a sink/source node one can spread a gradient, used as "potential field", and let all nodes send the values they sensed downward such a field, combining with others *en route*, until reaching the target and hence resulting there in the outcome of distributed sensing. As described in detail in [19] (so we omit the code here), this can be obtained by orthogonally stacking `C` component (convergent-cast) on top of a gradient. Intuitively, the ability of a gradient algorithm to promptly and resiliently construct such a potential field is essential for the efficient operation of `C` and the stabilisation of its results—i.e., for limiting errors in the values collected at the sinks.

Starting from the scenario described in Section VI-A, we make the sink count the number of overall nodes in the system—a paradigmatic case of distributed sensing where each device senses value 1, and en route combination is by mathematical sum. We analyse the impact of four versions of the gradient – namely classic, BIS, ULT, and ULT-damped – by comparing their resulting total count with the ideal value. Counts are averaged in the time-frame $[200, 400]$ (i.e., shortly after the input discontinuity), variability is kept in the range $[0, 0.20]$ (as it significantly impacts the quality of `C`), the filter factor for ULT is set to 0.1, and the damping factor to 0.3. We run 20 instances of each scenario with different random seeds and averaged the results, which had a 19% relative standard error among them. The results are reported in Fig. 6. As expected, ULT significantly outperforms BIS, while Classic shortly fails in keeping up with continuous changes in network topology. Additionally, as discussed in previous section, the damped version of ULT, though impairing precision in the distance estimation, is almost necessary to provide reasonable results for counting, for it causes the gradient to have a smoother "shape", favouring descent of values towards the sink.

VII. Conclusion

Based on existing and new algorithms and techniques related to the problem of estimating distances in fully-distributed systems, in this paper we propose *ULT gradient*, a gradient algorithm that provides the best performance so far in the diverse situations in which a gradient may need to operate. This is rooted in two key ingredients: SVD algorithm that achieves optimality among multi-path gradient algorithms, and mixing of composition/filtering/damping techniques to ensure that all the performance problems of gradient algorithms in dynamic environments are properly addressed. Most specifically, ULT improves over the gradient algorithm with best performance trade-offs to date, BIS [18], in that it more quickly recovers from changes in networks with high variability (asynchronicity and node mobility) and high density (large neighbourhoods/high communication radius).

Future works will be developed along several dimensions. First, mathematically or empirically estimating actual single-path and multi-path speed in a given network can allow us to better characterise advantages and disadvantages of the various approaches. Second, specific implementation techniques can be developed to increase the speed of multi-path solutions (and hence, SVD/ULT), by more promptly using neighbour values to compute new distance estimations—e.g., in the field calculus this can be done with a new implementation approach for the `rep` and `nbr` constructs. Third, empirical evaluation in large and realistic simulation scenarios (e.g. smart mobility and mobile wireless sensor networks) can allow us to more precisely investigate the performance issue of gradient algorithms.

Acknowledgment

We would like to thank the anonymous reviewers for comments and suggestions for improving the presentation.

References

[1] J. Beal, S. Dulman, K. Usbeck, M. Viroli, and N. Correll, "Organizing the aggregate: Languages for spatial computing," in *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, M. Mernik, Ed. IGI Global, 2013, ch. 16, pp. 436–501, longer version at: `http://arxiv.org/abs/1202.5509`.

[2] M. Mamei and F. Zambonelli, "Programming pervasive and mobile computing applications: The tota approach," *ACM Trans. on Software Engineering Methodologies*, vol. 18, no. 4, pp. 1–56, 2009.

[3] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler, "Hood: a neighborhood abstraction for sensor networks," in *Proceedings of MobiSys 2004*. ACM Press, 2004.

[4] T. Finin, R. Fritzson, D. McKay, and R. McEntire, "Kqml as an agent communication language," in *Proceedings of ACM CIKM '94*. New York, NY, USA: ACM, 1994, pp. 456–463.

[5] S. R. Madden, R. Szewczyk, M. J. Franklin, and D. Culler, "Supporting aggregate queries over ad-hoc wireless sensor networks," in *Workshop on Mobile Computing and Systems Applications*, 2002.

[6] M. Inchiosa and M. Parker, "Overcoming design and development challenges in agent-based modeling using ascape," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 99, no. Suppl 3, p. 7304, 2002.

[7] R. Nagpal, "Programmable self-assembly: Constructing global shape using biologically-inspired local interactions and origami mathematics," Ph.D. dissertation, MIT, Cambridge, MA, USA, 2001.

[8] D. Pianini, M. Viroli, and J. Beal, "Protelis: Practical aggregate programming," in *ACM SAC 2015*, April 2015, pp. 1846–1853.

[9] J. Beal and J. Bachrach, "Infrastructure for engineered emergence in sensor/actuator networks," *IEEE Intelligent Systems*, vol. 21, pp. 10–19, March/April 2006.

[10] J.-L. Giavitto, C. Godin, O. Michel, and P. Prusinkiewicz, "Computational models for integrative and developmental biology," Univerite d'Evry, LaMI, Tech. Rep. 72-2002, 2002.

[11] J. L. Fernandez-Marquez, G. D. M. Serugendo, S. Montagna, M. Viroli, and J. L. Arcos, "Description and composition of bio-inspired design patterns: a complete overview," *Natural Computing*, vol. 12, no. 1, pp. 43–67, 2013.

[12] M. Puviani, G. Cabri, and F. Zambonelli, "A taxonomy of architectural patterns for self-adaptive systems," in *In ACM C3S2E13 2013*. ACM, 2013, pp. 77–85.

[13] L. Gardelli, M. Viroli, and A. Omicini, "Design patterns for self-organising systems," in *Multi-Agent Systems and Applications V*, ser. LNAI. Springer, Sep. 2007, vol. 4696, pp. 123–132, in CEEMAS'07.

[14] J. Beal, D. Pianini, and M. Viroli, "Aggregate programming for the Internet of Things," *IEEE Computer*, vol. 48, no. 9, 2015.

[15] J. Beal, M. Viroli, D. Pianini, and F. Damiani, "Self-adaptation to device distribution changes," in *IEEE SASO 2016*, 2016, pp. 60–69, Best paper of IEEE SASO 2016.

[16] J. Beal, J. Bachrach, D. Vickery, and M. Tobenkin, "Fast self-healing gradients," in *Proceedings of ACM SAC 2008*. ACM, 2008, pp. 1969–1975.

[17] J. Beal, "Flexible self-healing gradients," in *Proceedings of the 2009 ACM Symposium on Applied Computing*, ser. SAC '09. ACM, 2009, pp. 1197–1201.

[18] G. Audrito, F. Damiani, and M. Viroli, "Optimally-self-healing distributed gradient structures through bounded information speed," in *Coordination Languages and Models*, ser. LNCS, J.-M. Jacquet and M. Massink, Eds., vol. 10319. Springer, 2017, pp. 59–77.

[19] M. Viroli, J. Beal, F. Damiani, and D. Pianini, "Efficient engineering of complex self-organising systems by self-stabilising fields," in *IEEE SASO 2015*. IEEE, 2015, pp. 81–90.

[20] F. Damiani, M. Viroli, D. Pianini, and J. Beal, "Code mobility meets self-organisation: A higher-order calculus of computational fields," in *Formal Techniques for Distributed Objects, Components, and Systems*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2015, vol. 9039, pp. 113–128.

[21] A. Igarashi, B. C. Pierce, and P. Wadler, "Featherweight Java: A minimal core calculus for Java and GJ," *ACM Transactions on Programming Languages and Systems*, vol. 23, no. 3, 2001.

[22] Q. Liu, A. Pruteanu, and S. Dulman, "Gradient-based distance estimation for spatial computers," *Comput. J.*, vol. 56, no. 12, pp. 1469–1499, 2013, DOI: 10.1093/comjnl/bxt124.

[23] A. Lluch-Lafuente, M. Loreti, and U. Montanari, "Asynchronous distributed execution of fixpoint-based computational fields," *CoRR*, vol. abs/1610.00253, 2016.

[24] M. Viroli and F. Damiani, "A calculus of self-stabilising computational fields," in *Coordination Languages and Models*, ser. LNCS. Springer-Verlag, 2014, vol. 8459, pp. 163–178.

[25] J. Katzenelson, "Notes on amorphous computing," in *MIT Artificial Intelligence Laboratory*. Citeseer, 2000.

[26] R. Nagpal, H. Shrobe, and J. Bachrach, "Organizing a global coordinate system from local information on an ad hoc sensor network," in *Information Processing in Sensor Networks*. Springer, 2003, pp. 333–348.

[27] F. Damiani and M. Viroli, "Type-based self-stabilisation for computational fields," *Logical Methods in Computer Science*, vol. 11, no. 4, 2015, DOI: 10.2168/LMCS-11(4:21)2015.

[28] E. M. Royer and C. Toh, "A review of current routing protocols for ad hoc mobile wireless networks," *IEEE Personal Commun.*, vol. 6, no. 2, pp. 46–55, 1999.

[29] W. Weibull *et al.*, "A statistical distribution function of wide applicability," *Journal of applied mechanics*, vol. 18, no. 3, pp. 293–297, 1951.

[30] D. Pianini, S. Montagna, and M. Viroli, "Chemical-oriented simulation of computational systems with ALCHEMIST," *J. Simulation*, vol. 7, no. 3, pp. 202–215, 2013.

[31] M. Aldinucci, S. Bagnasco, S. Lusso, P. Pasteris, S. Vallero, and S. Rabellino, "The Open Computing Cluster for Advanced data Manipulation (occam)," in *The 22nd International Conference on Computing in High Energy and Nuclear Physics (CHEP)*, San Francisco, USA, 2016.