Optimal single-path information propagation in gradient-based algorithms

Optimal Single-Path Information Propagation in Gradient-based Algorithms[☆]

Giorgio Audrito[a,b], Ferruccio Damiani[a,b], Mirko Viroli[c]

[a] *Dipartimento di Informatica, University of Torino, Torino, Italy*
[b] *Centro di Competenza per il Calcolo Scientifico, University of Torino, Torino, Italy*
[c] *DISI, University of Bologna, Cesena, Italy*

Abstract

Scenarios like wireless network networks, Internet of Things, and pervasive computing, promote full distribution of computation as well as opportunistic, peer-to-peer interactions between devices spread in the environment. In this context, computing estimated distances between devices in the network is a key component, commonly referred to as the *gradient* self-organisation pattern: it is frequently used to broadcast information, forecast pointwise events, as carrier for distributed sensing, and as combinator for higher-level spatial structures. However, computing gradients is very problematic in an environment affected by mutability in the position and working frequency of devices: existing algorithms fail in reaching adequate trade-offs between accuracy and reaction speed to environment changes.

We propose *BIS (Bounded Information Speed) gradient*, a fully-distributed algorithm that uses time information to achieve a smooth and predictable reaction speed, and prove it is optimal across algorithms following a single-path-communication strategy to spread information. We empirically evaluate BIS gradient and compare it with other approaches, showing that BIS achieves the best accuracy while keeping smoothness under control, and accordingly provides improved performance when used as building block in

[☆]This work has been partially supported by: EU Horizon 2020 project HyVar (`www.hyvar-project.eu`), GA No. 644298; ICT COST Action IC1402 ARVI (`www.cost-arvi.eu`); Ateneo/CSP D16D15000360005 project RunVar (`runvar-project.di.unito.it`).

Email addresses: `giorgio.audrito@unito.it` (Giorgio Audrito), `ferruccio.damiani@unito.it` (Ferruccio Damiani), `mirko.viroli@unibo.it` (Mirko Viroli)

more complex algorithms for creating spatial structures and performing distributed collection of data.

Keywords: Aggregate Programming, Gradient, Information Speed, Reliability, Spatial Computing

1. Introduction

The increasing availability of computational devices of every sort, spread throughout our living and working environments, is creating new challenges in the engineering of complex software systems, especially in contexts like the Internet-of-Things, Cyber-Physical Systems, Pervasive Computing, and so on. To take full opportunity of such large-scale computational infrastructures, models that focus on the aggregate behaviour of devices have been proposed for their ability to provide pervasive and intelligent sensing, coordination, and actuation over the physical world [1]. They raise the abstraction layer, through algorithms that provide distributed data structures that more easily capture the goal of a large-scale situated system.

In particular, in this context, "collective" programs take as input data located in physical positions of space, typically perceived by (virtual or physical) sensors [2], and produce analogous data as outputs, to be used to feed (virtual or physical) actuators, having an effect on other computational components, on the physical world, or on humans in it. Ultimately, a distributed computational process continuously executing over "space" and "time" can hence be viewed in terms of such an input/output transformation, which can involve complex coordination patterns, reuse existing library components, and be in need of satisfying multiple non-functional requirements: scalability, resilience to unpredictable changes, and heterogeneity and dynamism of the communication infrastructure. Accordingly, a key issue when trying to scale with the complexity of a collective application of this kind is the lack of libraries of reusable distributed algorithms with guaranteed resilience and performance to match the requirements of nowadays dynamic environments.

A prototypical example is given by the so-called *gradient algorithm* [3, 4, 5, 6], 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.[1] Gradients are very

[1] In a traditional settings, this essentially solves a *shortest path* (SP) problem in a

commonly used: to broadcast information, forecast events, dynamically partition networks, ground distributed sensing [7], anticipate future events [8], and to combine into higher-level spatial structures [4]. However, the known distributed algorithms for gradient computation are not fully satisfactory, as they involve relevant trade-offs between scalability, resiliency and precision.

This paper proposes the *BIS (Bounded Information Speed) gradient*, an improvement of the classic gradient algorithm in [6] (called "classic" henceforth), relying on time information to achieve smooth and predictably efficient reaction to changes. Given a rising speed v (i.e., increase in distance estimate over time) as a parameter, it enforces an information propagation speed (i.e., space travelled by information over time) equal to v. This allows to scale from the classic gradient (where essentially $v = 0$) to a reaction speed that we prove to be optimal (among those algorithms that spread information across single-path communications) when v equals the average information speed. If v is greater than such an average, however, a metric distortion is induced that may cause the algorithm to systematically overestimate gradient values. It is thus crucial to tune correctly the parameter in order to achieve the best accuracy.

To face this problem, we compute mathematical estimates of the average single-path communication speed, and use them for validating the performance of BIS gradient with respect to the three most performing distributed algorithms: classic [6], CRF [9] (a variation aimed at speeding up the raising of values) and FLEX [3] (addressing changes in network configuration). We thus show that the BIS gradient achieves the best accuracy while keeping smoothness under control. This comparison is carried out through a general approach to the empirical evaluation of performance for distributed algorithms (and gradient algorithms in particular). We consider gradients both in isolation, and as constituent building blocks of more complex algorithms for creating spatial structures and performing distributed collection of data.

The remainder of this paper is organised as follows. Section 2 provides the background for this paper and discusses related works, introducing the relevant gradient algorithms. Section 3 describes the proposed BIS gradient algorithm together with the mathematical estimates of average single-path information speed. Section 4 discusses possible implementations of BIS and of some relevant applications of it. Section 5 discusses the methodology

weighted network like addressed, e.g., by the Dijkstra's algorithm.

for empirical evaluation of spatial computing algorithms and compares the various gradient algorithms accordingly. Section 6 concludes and outlines possible directions of future research.

This document is an extended version of the prior work in [10] with the addition of two prototypical application scenarios for gradients (channel broadcasting and data collection, empirically evaluated in Sections 5.3 and 5.4), and documented source code (in *field calculus*, reviewed in Section 2.1) for all algorithms and application scenarios presented in this paper (Sections 2.4 and 4). Most notably, novel improvements to the broadcast and channel pattern are discussed in Sections 4.1 and 4.2: computationally efficient broadcast in sub-networks and improved channel shaping.

2. Background and Related Work

We now present a brief overview of gradient-related works in distributed computing, while focussing on the gradient algorithms that are closer to our approach and applicative scope: *classic*, *CRF* and *FLEX* gradients. The algorithms will be presented both through the equations that define their computation in each single node and with their code in field calculus [11, 12], a tiny functional language designed as *lingua franca* for collective adaptive systems. Section 2.1 presents the field calculus syntax and informal semantics. Section 2.2 introduces the built-in functions and code conventions used in this paper. Section 2.3 presents and overview of the different approaches to gradient calculations and their different scopes. Section 2.4 presents the state-of-the-art gradient algorithms relevant to the scope of this paper.

2.1. Field Calculus

We hereby recollect the most basic characteristics of field calculus [11, 12], an aggregate programming language providing a functional composition mechanism for distributed computations. The advantages of using it is twofold: *(i)* it concisely allows us to formally express the algorithms used in this paper with actual executable code; and *(ii)* its compositional model will emphasise the "building block" nature of the proposed gradient algorithms. In field calculus, each program `P` can be interpreted in two complementary ways: according to a *local* viewpoint, or to a *global* viewpoint.

Under the "global" viewpoint [13], computation is modelled as occurring on the overall network of interconnected devices, interpreted as a single

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

Figure 1: Syntax of field calculus.

"spatial computing" machine, evolving over time both topologically and regarding sensors and inputs. The data abstraction manipulated is hence a whole distributed space-time *field evolution* Φ, mapping computation events ϵ (space-time points where and when a device evaluates its program) to associated data values. Under the "local" viewpoint, instead, single devices logically undergo computation of a same program P in asynchronous rounds, during which each device *(i)* perceives contextual information through sensors; *(ii)* retrieves local information stored in the previous round, and collects messages received from neighbours while sleeping, in the form of *neighbouring fields* ϕ mapping neighbour device identifiers δ to values v; *(iii)* evaluates the program P, manipulating the data values retrieved from neighbours, context or local memory; *(iv)* stores local data, broadcasts messages to neighbours, and produces output values (possibly fed to actuators); *(v)* goes back to sleep for some time until the next round;

Figure 1 presents the syntax of the field calculus. A program P consists of a sequence of function declarations (of the kind "def d($\texttt{x}_1, \ldots, \texttt{x}_n$) {e}") and of a main expression e. The syntax of expressions comprises:

- variables x (used as function formal parameters) and values v, which in turn can be either *neighbouring field values* ϕ (associating neighbour devices to local values—not allowed to appear in source programs), or *local values* (built-in functions b, user-defined functions d, anonymous functions ($\overline{\texttt{x}}$)=>e, data values c($\overline{\ell}$)—numbers, literals, etc.);
- nbr-expressions nbr{e}, modelling neighbourhood interaction through sharing the value of expression e: each device δ broadcasts the value of e in δ, and evaluates the expression into a neighbouring field value ϕ associating to each neighbour δ' of δ the latest evaluation of e in δ';

- `rep`-expressions $\texttt{rep}(\texttt{e}_1)\{(\texttt{x})\texttt{=>}\texttt{e}_2\}$, repeatedly evolving a local state through time, by computing expression $\texttt{e}_2$ substituting the variable `x` with the value calculated for the whole `rep`-expression at the previous computational round, if present, or with the value of $\texttt{e}_1$ otherwise;
- branching statements $\texttt{if}(\texttt{e}_1)\{\texttt{e}_2\}\{\texttt{e}_3\}$, which ensure correct matching of outgoing and ingoing `nbr`-messages through a process called *alignment*, forcing $\texttt{e}_2$ and $\texttt{e}_3$ to be computed into two independent *clusters* (consisting of the devices that evaluated $\texttt{e}_1$ to `True` or `False` respectively): in fact, devices that computed different branches $\texttt{e}_2$ and $\texttt{e}_3$ also computed different "sharing" `nbr`-expressions, and such expressions in $\texttt{e}_3$ are ignored by $\texttt{e}_2$ and vice-versa, effectively splitting that part of the computation into non-communicating parts;
- function calls $\texttt{e}(\overline{\texttt{e}})$ where a functional expression `e` is applied by-value to its arguments: as with branching statements, correct matching of messages is guaranteed through alignment, splitting the computation into several clusters (one for each possible value of `e`).

A third "limit" viewpoint is also possible for *self-stabilising* programs [14, 15], i.e., programs such that given an input that is everywhere constant after a certain time t_0, it produces an output that is everywhere constant after a certain time $t_1 \geq t_0$, and this "stabilised" output after t_1 depends *only* on the values of the "stabilised" input after t_0. All the programs presented in this paper can be readily shown to be self-stabilising by means of the techniques in [14, 15]. Under the limit viewpoint, a program receives in input a list of *computational fields*, space-distributed data structures mapping device identifiers δ to values `v`, and produces as output another computational field. This input-output mathematical function represents the limit behaviour of the program, thus permitting to abstract the time information manipulated under the global viewpoint, and is usually the most convenient viewpoint for programmers (when applicable), which are in this way enabled to reason in terms of *mathematical* composition of spatial input-output mappings.

2.2. Built-in Functions and Code Conventions

In the field calculus code to come, we shall use $[\overline{\texttt{e}}]$ as a shorthand for $\texttt{tuple}(\overline{\texttt{e}})$, and $\texttt{let x} = \texttt{e}_1 \texttt{ in } \texttt{e}_2$ as a shorthand for $((\texttt{x})\texttt{=>}\texttt{e}_2)(\texttt{e}_1)$. Constructs will be coloured in red, built-in functions in blue and user-defined functions in violet. We assume the existence of the following built-in functions:

- `mux`, the classic multiplexer operator selecting its second or third argument based on the boolean evaluation of the first;

- `maxHood` and `minHood`, which compute the maximum (resp. minimum) in the range of a neighbouring field value;
- `minHoodLoc`, which selects the minimum of a neighbouring field value and a provided local value;
- `1st`, `2nd` and `3rd`, which select components of a value of `tuple` type;
- `nbrRange`, which evaluates to the neighbouring field value of short-range distances with neighbours, as perceived by a local sensor;
- `getDeltaTime`, which evaluates to the time elapsed from the previous computation round, as perceived by a local sensor;
- `nbrLag`, which evaluates to a neighbouring field of time differences with the messages received from neighbours, as perceived by a local sensor.

We also overload local built-in operators (e.g., +) to behave pointwise on neighbouring field values when needed.

The following algorithms will take as input a computational field `source` designating the selected sources, such that `source` is 0 in sources and ∞ otherwise. Furthermore, space and time distances from neighbours will be stored in variables `dist` and `lags`, whereas the local algorithm estimate (taking into account the `source` value only) will be stored in variable `loc`. Notice that the time difference between the present round and the value obtained from a neighbour can be calculated as:

$$\texttt{lags} = \texttt{nbrLag}() + \texttt{nbr}\{\texttt{getDeltaTime}()\}.$$

This is due to the fact that neighbours' values are obtained through `nbr{old}` where `old` is the bound variable of the main `rep`-operator, which represents the overall result of the algorithm in the *previous* round. Thus, `nbr{old}` are the algorithm's values in the *second-last* rounds of neighbours, from which time `nbrLag`() + `nbr{getDeltaTime`()} has elapsed.

2.3. Gradient-based Approaches

In this paper we are concerned with coordination strategies for situated networks, where the objective can be represented in terms of a global, system-level "pattern" to be achieved by local interactions between neighbouring devices, showing inherent resilience with respect to unpredicted changes—in network topology, scale, inputs coming from sensors, and so on. This viewpoint is endorsed by a number of works in a recent thread of research, in the context of coordination models and languages [5, 16, 17], multi-agent systems [18, 19, 20] and spatial computing [1, 4, 7, 21, 22]. In spite of various

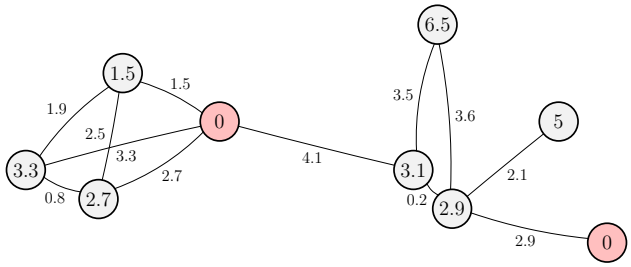

Figure 2: The computational field given by the gradient computed in a sample network with two source devices (in red), and distance labels on links between devices.

differences and peculiarities, they all promote the idea of creating complex distributed algorithms as *spatial computations*, where few basic communication and coordination mechanisms are provided to the programmer, who uses them by progressively stacking building blocks into layers of increasing complexity.

In this context, gradient data structures, or *gradients* for short, are pervasively used as key building blocks [4, 7]. They produce a map—also called a *computational field* [23, 11]—assigning to each device δ in a network N its estimated distance from the closest *source* device (an input for the problem), computed by the shortest-path through weighted links in the network (see Fig. 2 for an example of one such computational field).

Applications of gradients are countless. Other than to trivially estimate long-range distances (possibly according to metrics computed during execution of the algorithm), gradient computations enact an outward progressive propagation of information along optimal paths. Thus, they are used as forward "carrier" for broadcasting information, forecasting events, and dynamically partitioning networks [7]. Also, used backwards, one can make information flow back to the source, to move or steer mobile agents or data towards the source, or to summarise or average distributed information, i.e., to generally support distributed sensing [7]. Other applications include: considering future events so as to provide proactive "adaptation" [8], managing semantic knowledge in situated environments [24], create high-level spatial structures [4], elect leaders on a spatial basis [25], and so on.

Due to their usefulness, several works also study how to establish gradients in contexts where local estimation of distances is not available [26, 27], and others take them as basic example to study self-stabilisation techniques [5, 28].

2.4. Gradient-based Implementations

According to the framework presented in [14], it is suggested to associate to fundamental building blocks (including the gradient) a library of alternative implementations, among which one can pick the right implementation for each specific use in the application at hand. It is therefore of interest to analyse different trade-offs in the implementation of gradient algorithms, with the goal of identifying approaches guaranteeing reactivity and smoothness in the way gradients (and the many applications on top) can respond to dynamic environments. In its most basic form, the gradient can be calculated through iterative and asynchronous application of a triangle inequality constraint in each device δ, starting with ∞ everywhere:[2]

$$G(\delta) = \begin{cases} 0 & \text{if } \mathit{source}(\delta) \\ \min\{G(\delta') + w(\delta', \delta) : \ \delta' \in N \text{ linked}^3 \text{ with } \delta\} & \text{otherwise} \end{cases}$$

where $w(\delta', \delta)$ is any given positive metric, providing a notion of distance between devices. We call this procedure *classic* gradient, and report its field calculus code in Figure 4 according to the conventions in Section 2.2. Assume repeated fair application of this calculation in a fixed network, namely, devices update infinitely often as they never stop working. Then the gradient will converge to the correct value at every point [28], either at the limit for devices that get suddenly disconnected to the source (which raise indefinitely towards the correct value ∞) or in finite time otherwise. However, the performance of this algorithm in a mutable environment[4] is impaired by several limitations.

- *Speed Bias*: if devices are continuously moving, the values produced by the algorithm systematically underestimate[5] the correct value of the gradient; with an error that increases with the movement speed.

[2]Generalisations of the distance estimation problem are possible, such as allowing arbitrary values for source devices. However, since this generalisation is seldom used and somehow straightforward, in the remainder of this paper we focus on the distance estimation problem for which source devices have distance zero.

[3]The network topology is not assumed to be constant or globally known: a device δ' is linked with δ provided that a recent broadcast of messages from δ' reached δ.

[4]In a static environment, the algorithm converges in the optimal number of steps given by the Floyd-Warshall algorithm.

[5]Sporadic overestimation is also possible, however the "minimising" nature of the algorithm propagates lower estimates and disperses higher ones.

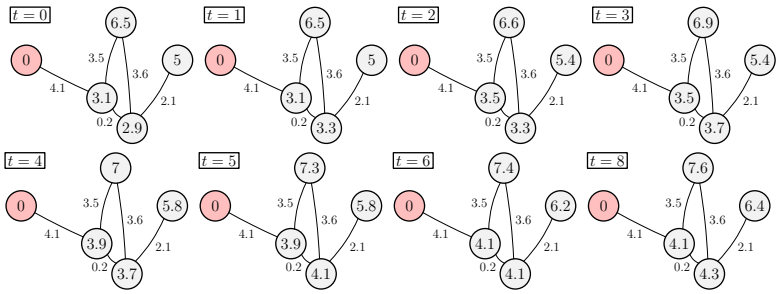

Figure 3: Evolution after loss of the right source. In each round all devices compute in order, from the one holding the highest value to the one with the lowest. Each device rises by 0.4 every two rounds, because of the short link at the middle of the graph.

- *Rising Value*: in response to quick changes in the network (e.g., a change in the set of source devices), the algorithm can rapidly correct values that need to drop, while it is very slow in correcting values that need to rise. In other words, the algorithm can badly underestimate values for long periods of time after such changes. Precisely, the rising speed of this algorithm is bounded by the distance between the pair of closest devices: Fig. 3 shows an example of this phenomenon on a part of the network in Fig. 2. This problem is also known as *count-to-infinity* in the context of routing algorithms [29].

- *Smoothness*: in presence of random noise in distance estimates, it might be preferable not to strictly follow the triangle inequality, so as to reduce the resultant flickering in the output values and improve accuracy. More importantly, if the distance estimates are used for a more complex coordination mechanism (e.g., for moving values towards the sources by "descending" the shortest-paths tree obtained from the gradient, see Section 4.3), then each variation in the estimates might change the resulting shortest-paths tree, effectively disrupting the outcome of the coordination for some time.

In order to overcome these limitations, several refined algorithms have been proposed. To the best of our knowledge, those that better address those problems are J. Beal's CRF gradient (Constraint and Restoring Force) [9] and FLEX gradient (Flexible) [3], considered in turn and reported in Figure 4 according to the conventions in Section 2.2.

```
def classic(source) {
  rep (Infinity) { (x) =>
    minHoodLoc(nbr{x} + nbrRange(), source)
  }
}
```

```
def CRF(source, speed) {
  let lags = nbrLag() + nbr(getDeltaTime()) in
  let dist = nbrRange() in
  let loc = [source, 0] in
  1st(rep (loc) { (old) =>
    let neigh = mux (nbr{1st(old)}+dist <= 1st(old) - lags*2nd(old),
      [nbr{1st(old)} + dist, 0], loc) in
    let new = minHoodLoc(neigh, loc) in
    if (new == old || 1st(new) < Infinity) {
      new
    } {
      [1st(old)+speed*getDeltaTime(), speed]
    }
  })
}
```

```
def FLEX(source, epsilon, frequency, distortion, radius) {
  let loc = [source, 0] in
  1st(rep (loc) { (old) =>
    let dist = max(nbrRange(), distortion*radius) in
    let new = minHoodLoc([nbr{1st(old)}+dist, 0], loc) in
    let slopeinfo = maxHood([(1st(old)-nbr{1st(old)}) / dist, nbr{1st(old)}, dist]) in
    if (old == new || 1st(new) == 0 || 2nd(old) == frequency ||
        1st(old) > max(2*1st(new), radius) || 1st(new) > max(2*1st(old), radius)) {
      new
    } {
      if (1st(slopeinfo) > 1+epsilon) {
        [2nd(slopeinfo) + (1+epsilon)*3rd(slopeinfo), 2nd(old)+1]
      } {
        if (1st(slopeinfo) < 1-epsilon) {
          [2nd(slopeinfo) + (1-epsilon)*3rd(slopeinfo), 2nd(old)+1]
        } {
          [1st(old), 2nd(old)+1]
        }
      }
    }
  })
}
```

Figure 4: Field calculus code for the state-of-the-art algorithms *classic*, *CRF* and *FLEX*.

CRF Gradient. The CRF gradient [9] is designed to address the rising value problem by ignoring some *Constraints* (i.e., neighbours[6]), while assuming a *Restoring Force* inducing a uniform rise in absence of constraints. The algorithm takes as parameter an empirically tuned speed v_0, and associates a "rising speed" $v(\delta)$ to each device so that: if the value of the device is currently constrained (either by being a source or by the value of some neighbour) then $v(\delta) = 0$; otherwise if the value is *not* constrained (i.e., all neighbours have

[6]Recall that in the classic gradient, the value $G(\delta)$ is obtained by combining the "triangle-inequality" constraints $G(\delta) \leq G(\delta') + w(\delta', \delta)$ for each neighbour δ'.

been discarded) then $v(\delta) = v_0$ (i.e., the gradient estimate is increasing at speed v_0). Before applying the minimisation as in the classic gradient, the CRF gradient considers a neighbour δ' as "able to exert constraint" if and only if

$$G(\delta') + w(\delta', \delta) \leq G(\delta) - \lambda(\delta', \delta) \cdot v(\delta)$$

where $\lambda(\delta', \delta)$ measures time lag, i.e., how old is the information in δ about δ'. The above condition checks whether the constraint given by δ' is able to bound the currently (i.e., not yet updated) value of the gradient *as shifted back to the time when the constraint was calculated*. If the current device is not yet rising, the condition amounts to the constraint being able to reduce the current value; otherwise it becomes more restrictive.

If some neighbour that is able to exert constraint exists, the value is calculated similarly to the classic gradient. Otherwise, a fixed rising speed is enforced (thus rising by $v_0 \cdot \Delta t$ where Δt is the time interval between the last two rounds):

$$G(\delta) = \begin{cases} 0 & \text{if } \mathit{source}(\delta) \\ \min\{G(\delta') + w(\delta', \delta) : \ \delta' \text{ exerts constraint}\} & \text{if some } \delta' \text{ exists} \\ G(\delta) + v_0 \cdot \Delta t & \text{otherwise} \end{cases}$$

Through this algorithm the rising speed is then equal to v_0, provided that v_0 is small enough,[7] thus addressing the rising value problem.

FLEX Gradient. The FLEX gradient [3] 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. Precisely, it first calculates the "maximum local slope":

$$s(\delta) = \max \left\{ \frac{G(\delta) - G(\delta')}{w(\delta', \delta)} : \ \delta' \in N \text{ linked with } \delta \right\}$$

This slope is then used to calculate the gradient estimation as:

$$G(\delta) = \begin{cases} 0 & \text{if } \mathit{source}(\delta) \\ G(\delta') + (1 + \epsilon) \cdot w(\delta', \delta) & \text{if } s(\delta) > 1 + \epsilon \\ G(\delta') + (1 - \epsilon) \cdot w(\delta', \delta) & \text{if } s(\delta) < 1 - \epsilon \\ G(\delta) & \text{otherwise} \end{cases}$$

[7] If v_0 is too large, the algorithm flickers and does not stabilise to correct values.

where δ' is the device achieving maximum slope (according to the values available to the current device). The above formula, in other words, selects the closest value to $G(\delta)$ in the interval from $G(\delta')+(1-\epsilon)\cdot w(\delta',\delta)$ to $G(\delta')+(1+\epsilon)\cdot w(\delta',\delta)$, thus attempting to reduce local changes as much as possible while introducing a metric distortion below ϵ. Two further optimisations are also introduced in FLEX gradient: first, the classical gradient formula is used instead of the above one whenever the current value is over a factor 2 from the old value, or anyway every once in a while (details can be found in [3])—this prevents a systematic error of ϵ to persist indefinitely in a static environment after a network change; second, a distorted metric $w'(\delta',\delta) = \max(w(\delta',\delta), k)$ is used, for a certain constant k—this adds some further error in the output of the algorithm, but it also ensures that the rising speed is at least k (since k becomes the shortest possible "distorted" distance between devices).

3. BIS Gradient

We are now ready to present the *Bounded Information Speed (BIS) gradient*, which uses temporal interval estimates to enhance spatial distance estimates. By determining the average *information speed* for *single-path* algorithms, a conversion between the two measures is possible, and allows us to obtain an "optimal reactivity" as we will show in Theorem 3. Section 3.1 defines single- and multi-path information speeds, while providing a statistical estimate for the former. Section 3.2 presents BIS gradient in equation form (actual code will be given in Section 4.1), investigates its performance and proves optimality among algorithms with a single-path information flow. Section 3.3 shows how BIS gradient can be modified to incorporate the main elements of FLEX gradient, in case stability of values is preferred over accuracy of results.

3.1. Information Speed

In large-scale opportunistic networks, devices typically perform an interaction through short-range message passing between neighbour devices. The speed achieved by information in this process constitutes an upper bound for responsiveness to environment and input changes, in a similar way to the speed of light, which is an upper bound for causal relationship between events. Depending on the pattern followed by information exchanges, we can distinguish between two main achievable speeds: *single-path* and *multi-path*.

Definition 1 (Information Speed). The *single-path information speed* is the space travelled over time by messages through a (possibly mutable) spanning tree in the network. The *multi-path information speed* is the same quantity assuming messages are exchanged through all possible links in the network.

Clearly, the upper bound for causal relationship in a network is given by the multi-path information speed. This communication pattern requires multiple informations to be aggregated in each node, in order to avoid a program state explosion, and is thus typical of "aggregation" algorithms (such as broadcasting and collecting). Conversely, communication in existing gradient algorithms (and in particular in the BIS gradient we shall introduce in the next subsection) is usually structured on an implicit (shortest-paths) spanning tree: messages from all neighbours are received, but only one of them is selected and passed over for subsequent computations. Since "discarded" messages do not contribute to the resulting value, the information available in every node is the result of a *single* path of computation; in other words, information follows the single-path communication pattern for existing gradient algorithms. For this reason, in the remainder of this section we shall focus on single-path information speed and estimate its average v_{avg} in a random network. This estimate would be crucial to determine the value to be passed to BIS gradient for its parameter v.

Consider a network of computing devices, each of them running an algorithm with a certain time period P on data available from neighbour devices within a certain radius R. Let D be a random variable for the distance and T for the time interval between the event of a device sending a message and the event of another device using that message for computation. Then the average speed S achieved by information can be expressed as:[8]

$$E(S) = E\left(\frac{D}{T}\right) = \frac{E(D)}{E(T)} \cdot \left(1 + \frac{V(T)}{E(T)^2}\right) \tag{1}$$

truncating the bivariate Taylor expansion of the ratio function to the second order (see [30] for a complete proof of this fact). Given the details of the specific application setting where an algorithm is executed, the expected

[8]Following standard statistic notation, we use $E(X)$ for the mean and $V(X)$ for the variance of a random variable X.

single-path information speed in Equation 1 can be calculated through standard means. As a prototypical example, we apply this equation to the field calculus model of computation (see Section 2.1), obtaining a rough estimate that will be used in the experiments of Section 5. Further work will be needed to extend this estimate to cover more disparate models of computations and increase its accuracy; however, this falls beyond the scope of this paper.

The average distance crossed by a message can be calculated as the average radius of communication R times the average distance of a uniformly chosen random point in an n-dimensional unit ball, giving a total $E(D) = \frac{n}{n+1}R$. If devices are moving at a certain average speed v, this estimate should be adapted to take into account that messages with a certain lag T could come from a further distance up to $v \cdot T$. For algorithms following a "random" spanning tree, this extra term cancels out as it may equally likely increase or decrease the distance travelled by a message, depending on the relative orientation of v with that of the message. However, algorithms based on shortest-paths spanning trees show a preference for additive terms, and an exact calculation of the expected distance in this setting is complex and depends on many factors. In this context, we propose to add half of the maximum increase to the expected distance travelled to obtain a roughly acceptable estimate $E(S) = E\left(\frac{D}{T}\right) + \frac{v}{2}$, which however imprecise will still show its effectiveness in Section 5.

The average time interval in field calculus can be modelled as $T = P \cdot (I + I \cdot F)$ where P represents the period of a random device, I represents the imprecision of a single device, F represents a random phase between devices, as a uniform distribution of values in $[0, 1]$. In this model, $E(T) = \frac{3}{2}E(P) \cdot E(I) = \frac{3}{2}Q$ (where Q is the average computation period) and:

$$1 + \frac{V(T)}{E(T)^2} = \frac{E(T^2)}{E(T)^2} = \frac{E(P^2)}{E(P)^2} \cdot \frac{E\left((I + I\,F)^2\right)}{E(I + I\,F)^2} = \frac{E(P^2)}{E(P)^2} \cdot \left(1 + \frac{V(I + I\,F)}{E(I + I\,F)^2}\right)$$
$$= \frac{E(P^2)}{E(P)^2} \cdot \left(1 + \frac{V(I) + \frac{V(I)}{3} + \frac{E(I)^2}{12}}{\frac{9}{4}E(I)^2}\right) = \left(1 + \frac{V(P)}{E(P)^2}\right) \cdot \left(\frac{28}{27} + \frac{16}{27}\frac{V(I)}{E(I)^2}\right)$$

Notice that $\frac{V(X)}{E(X)^2}$ is the square of the relative standard error $\hat{\sigma}^2(X)$. Thus the average single-path information speed v_{avg} can be estimated as:

$$v_{\text{avg}} = E(S) = \frac{2}{3}\frac{n}{n+1}\frac{R}{Q} \cdot \left(1 + \hat{\sigma}^2(P)\right) \cdot \left(\frac{28}{27} + \frac{16}{27}\hat{\sigma}^2(I)\right) + \frac{v}{2} \qquad (2)$$

where v is the movement speed of devices. This equation tells us that: the speed is mainly proportional to the ratio of communication radius over computation period; the speed increases with the dimensionality of space, i.e., is lower for devices aligned in a row and higher for devices in 3-dimensional space;[9] the speed increases with the relative error of computation periods, both among different devices and inside a single device. Equation 2 will be used later to estimate the v parameter of the BIS gradient algorithm. We remark that the average above is computed for a single hop of communication. Over multiple hops, the relative standard error decreases while the average does not change significantly. In case the network parameters (average radius of communication, computation period, etc.) cannot be assumed to be constant, a simple algorithm can still estimate v_{avg} continuously according to Equations 1 or 2 above (by averaging the relevant quantities through low-pass filters).

3.2. Computing Gradient through Information Speed

As exemplified in Fig. 3, in presence of a rising value problem, the increase of distance estimate per round is bounded by the currently shortest link in the network ℓ. We accordingly obtain an average information speed proportional to $\frac{2}{3}\frac{\ell}{Q}$ instead of $\frac{2}{3}\frac{n}{n+1}\frac{R}{Q}$, which can be arbitrarily slower as ℓ approaches zero. This fact suggests us to prevent the rising value problem by *lower bounding* the information speed to make this "slow" rise impossible.

The *Bounded Information Speed* (BIS) gradient improves over the classical gradient by enforcing a minimum information speed v requested by the user. 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. Greater values of v induce instead a metric distortion, causing the algorithm to systematically overestimate values. In the remainder of this paper, we shall thus express v as a fraction of v_{avg} (the average single-path communication speed, which we estimate through Equation 2).

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

[9] Gradient algorithms have preference for shortest-path links, so that information tends to propagate linearly regardless of the dimensionality of the space. This fact does not contradict the above estimate, which assumes that transmission links are chosen randomly (assumption viable also for gradient algorithms in sparse networks).

the message started from a source. Lags are estimated through local time differences,[10] so that no overall clock synchronisation is required. When considering a candidate neighbour[11] 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:

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

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 $v \cdot L(\delta, \delta')$ (with the additive constant $-r$ to ensure that some error is taken into account).

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

$$[G(\delta), L(\delta)] = \begin{cases} [0, 0] & \text{if } \textit{source}(\delta) \\ \min\{[G(\delta, \delta'), L(\delta, \delta')] :\ \delta' \in N \text{ linked with } \delta\} & \text{otherwise} \end{cases}$$

This algorithm generalises the classic gradient algorithm, as shown in the following.

Theorem 1 (Degenerate BIS). *The BIS gradient with $v = 0$ is equivalent to the classic gradient.*

Proof. If $v = 0$, $G(\delta, \delta') = \max\{G(\delta') + w(\delta', \delta),\ 0 \cdot L(\delta, \delta') - r\}$ is equal to $G(\delta') + w(\delta', \delta)$ so that $L(\delta)$ is implicitly discarded.

In particular, the same result would hold for devices with no lag estimator so that $\lambda(\delta, \delta')$ is always 0. For devices with an internal timer (so that a lag estimator can be defined), tweaking the parameter v close to the average single-path information speed provides a guaranteed reactivity, which is optimal among algorithms with a single-path information flow.

[10] More specifically, each device computes $\lambda(\delta', \delta)$ as $t_{\text{now}} - t_{\text{mess}}$, where t_{now} is its current time and t_{mess} is its local time when the message from δ' was received. Since $\lambda(\delta', \delta)$ is obtained by subtracting local times on a same clock, it is an absolute interval thus not requiring inter-device synchronization (assuming homogeneous clock speed).

[11] We recall that neighbouring relations may evolve over time, and the given equations are repeatedly applied in asynchronous rounds (see Section 2.4).

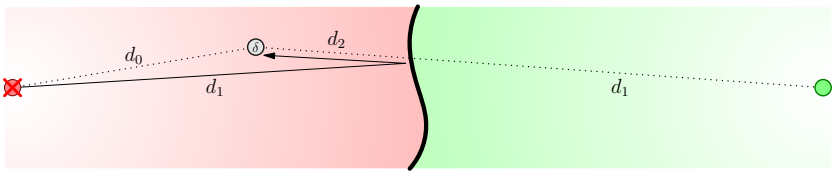

Figure 5: Information flow upon disconnection of a source device.

Theorem 2 (Performance Bound). *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.*

Proof. Since $G(\delta, \delta') \geq v \cdot L(\delta, \delta') - r$ for all δ', also $G(\delta) \geq v \cdot L(\delta) - r$ concluding the first part. For the second part, consider an information that started propagating from a certain source at time t_0 and is now obsolete (e.g., the source has been disconnected), and fix a device δ computing in times $t_1, \ldots, t_n$ constrained by such obsolete information. Since $L(\delta) = t_i - t_0$ in each computing round $i \leq n$, $G(\delta) \geq v \cdot L(\delta) - r = v \cdot (t_i - t_0) - r$ concluding the second part.

Theorem 3 (Optimality). *The BIS gradient with v equals to the average single-path information speed v_{avg} attains optimal reactivity among algorithms with a single-path information flow.*

Proof. As a prototypical example, consider an already stabilised network with a selected source device and its corresponding influence region, i.e., the set of devices whose distances are calculated w.r.t. the selected source (red). Suppose that the selected source device is suddenly disconnected at time $t = 0$. In any algorithm with a single-path information flow, the information about this disconnection flows through the influence region at average speed v. For example, device δ in Fig. 5 is reached by this information at time $\frac{d_0}{v}$, and it cannot change its value from d_0 before that time.

In the best case scenario, after the information about the disconnection reaches the border a new wave of information can bounce back towards the inside of the region, bringing values calculated from other sources (green). Since the shortest path from the disconnected source to the border and then back to device δ has length $d_1 + d_2$ (black arrow) and information flows at speed v, the earliest time when δ can reach the correct value is $\frac{d_1+d_2}{v}$. Notice that this value is d_1+d_2 since the distance from the border to the two sources is the same.

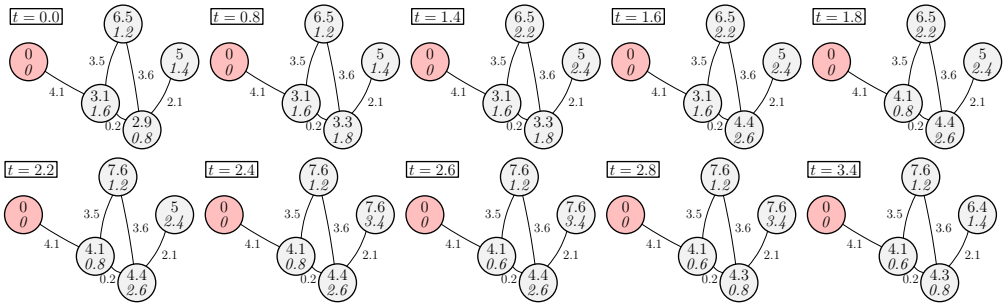

Figure 6: BIS gradient sample execution; lag estimates are in italic, and a single device fires every 0.2 time units.

Then δ holds value d_0 at time $\frac{d_0}{v}$ and value d_1+d_2 at time $\frac{d_1+d_2}{v}$, effectively rising at speed $(d_1 + d_2 - d_0)/(\frac{d_1+d_2}{v} - \frac{d_0}{v}) = v$. Since this is the best-case scenario, a faster rising speed is not possible thus proving optimality of BIS gradient.

Figure 6 shows the execution of BIS gradient on the same network of Figure 3, with the same firing order between devices. The assumed communication radius is $r = 6$, and the average speed is $v_{\text{avg}} = \frac{2}{3}6/1 = 4$ since devices are steady and the average fire rate is 1. The last fire of the disconnected left source happens at $t = -1$ before of the shown graphs, and lags of nodes depending on that source keep rising accordingly. At $t = 1.8$ reconfiguration starts, reaching correct values by $t = 3.4$ instead of $t = 8$ necessary for the classic gradient.

3.3. Reducing Volatility and Communication Cost

An improved reactivity to changes naturally translates into an increase in volatility of values, thus reducing the degree of *smoothness*. This holds true also for the BIS gradient: in a mutable environment, even calculating the exact gradient all the times would perform poorly on smoothness, since it would rapidly adapt all the values as noise and small movements take place.

In order to improve smoothness of rapidly self-healing algorithms, it is then necessary to insert a damping component. Also the FLEX gradient, designed for improved smoothness, can be seen as the embedding of the following damping function into the classical gradient computation:

$$\texttt{damp}(\texttt{old}, \texttt{new}) = \begin{cases} \texttt{new} + \epsilon \cdot w(\delta', \delta) & \text{if } \texttt{old} > \texttt{new} + \epsilon \cdot w(\delta', \delta) \\ \texttt{new} - \epsilon \cdot w(\delta', \delta) & \text{if } \texttt{old} < \texttt{new} - \epsilon \cdot w(\delta', \delta) \\ \texttt{old} & \text{otherwise} \end{cases}$$

In future works, it is therefore natural to investigate whether the insertion of this damping function (or others) into algorithms other than the classic gradient would achieve the same effect. In Section 5 we shall show that this is true to some extent, allowing the BIS gradient for an improved smoothness.

4. Implementation and Usages of Gradient Algorithms

We now present implementations of the gradient algorithm introduced in the previous section, and of its usage as foundational *building block* for some more complex applications (in particular, channel broadcasting and data collection). Section 4.1 presents the field calculus code (see Section 2.1) of the BIS gradient algorithm, together with extensions to be used as building blocks for more complex systems. Section 4.2 uses these blocks to define a *channel broadcast* service, as a sample application, which will later be experimentally evaluated in Section 5.3. Section 4.3 discusses the application of gradients to data collection, one of the main building blocks in the context of distributed sensing, which will later be experimentally evaluated in Section 5.4.

4.1. BIS Gradient as a Building Block

Figure 7 presents the code of BIS gradients as presented in Section 3, according to the conventions in Section 2.2. The field calculus code of the algorithms is obtained through an almost straightforward translation of the formulas in the corresponding sections. In Viroli et al. [15] it is argued that the following three main *building blocks* form a combinator set able to express many relevant self-stabilising distributed systems:

- **G**, which propagates `initial` values outwards from a set of sources, while updating them through a function `accumulate`. This building block is mostly used to either calculate distances or broadcast values, but it can in principle be applied to many different tasks.
- **C**, which accumulates values inwards into a set of sources, computing in them an aggregate summary of the original values: e.g., the minimum or the sum of these values.
- **T**, which evolves a local value through time via a decay function.

```
def BIS(source, speed, radius) {
  let lags = nbrLag() + nbr(getDeltaTime()) in
  let dist = nbrRange() in
  let loc = [source, source] in
  1st(rep (loc) { (old) =>
    let dx = nbr{1st(old)}+dist in
    let dt = nbr{2nd(old)}+lags in
    minHoodLoc([max(dx, dt*speed-radius), dt], loc)
  })
}
```

```
def BISflex(source, speed, radius, epsilon, frequency) {
  let lags = nbrLag() + nbr(getDeltaTime()) in
  let dist = nbrRange() in
  let loc = [source, source, 0] in
  1st(rep (loc) { (old) =>
    let dx = nbr{1st(old)}+dist in
    let dt = nbr{2nd(old)}+lags in
    let new = minHoodLoc([max(dx, dt*speed-radius), dt, dist], loc) in
    let round = rep (0) {(x) => x+1} % frequency in
    let delta = epsilon*3rd(new) in
    if (old == new || 1st(new) == 0 || round == 0 ||
        1st(old) > max(2*1st(new), radius) || 1st(new) > max(2*1st(old), radius)) {
      new
    } {
      let sign = if (1st(old) < 1st(new)) {1} {-1} in
      let diff = abs(1st(new) - 1st(old)) in
      if (diff > delta) {
        [1st(new) - delta*sign, 2nd(new), 3rd(new)]
      } {
        old
      }
    }
  })
}
```

Figure 7: Field calculus code for BIS gradient with or without a *FLEX* damping.

Distance estimation algorithms (such as BIS gradient) are closely related to the G building block: however, the former is more general as it involves arbitrary accumulation along shortest paths. Nonetheless, an enhancement of BIS gradient taking into account `initial` values and `accumulate` functions can be directly obtained.

Figure 8 presents possible implementations of the G building block. Function `G-BIS` extends BIS by adding a third tuple element in the main `rep`-loop, which tracks the values as they are accumulated. This element is set as `initial` in the local `loc` value and updated (for each neighbour) as `accumulate(nbr{3rd(old)})` in the main loop. Then, when the neighbour inducing the best estimate for the distance is selected, the corresponding updated value is carried over with it.

Function `G`, instead, implements the G-mechanism as a separate function taking distances as input, showing that in fact any distance estimation algorithm can be converted into a G block. Furthermore, such an "exter-

```
def G-BIS(source, initial, accumulate, speed, radius) {
  let lags = nbrLag() + nbr(getDeltaTime()) in
  let dist = nbrRange() in
  let loc = [source, source, initial] in
  3rd(rep (loc) { (old) =>
    let dx = nbr{1st(old)}+dist in
    let dt = nbr{2nd(old)}+lags in
    let v = nbr{3nd(old)} in
    minHoodLoc([max(dx, dt*speed-radius), dt, accumulate(v)], loc)
  })
}
```

```
def G(initial, accumulate, gradient) {
  rep (initial) { (old) =>
    2nd(minHoodLoc([nbr{gradient}, accumulate(nbr{old})], [gradient, initial]))
  }
}
```

```
def G'(initial, null, accumulate, gradient) {
  rep (null) { (old) =>
    mux(gradient == 0, initial,
      2nd(minHood(
        mux(nbr{old} == null, [Infinity, null], nbr{[gradient, old]})
      ))
    )
  }
}
```

Figure 8: Implementations of the **G** block.

nal" approach allows for better factoring of source code, possibly reusing computed distances multiple times and reducing the overall computational stress. Notice that the result of composing `G` with BIS is *not* fully equivalent to `G-BIS`. In the latter, values are accumulated through the paths that the algorithm deems to be "shortest". In the former, information about which paths are preferred is not available, and values are thus accumulated greedily from the neighbours with lowest *gradient*. This choice tends to minimise the number of hops the information has to travel, hence improving over `G-BIS` when the G block is used for, e.g., broadcast. However, in other contexts where the `accumulate` function is space-dependent the integrated algorithm `G-BIS` might be preferable.

Furthermore, function `G` does not perform properly also when executed into a sub-cluster of the devices used to compute the gradient.[12] In this case, a device might not have any neighbour with lower gradient inside the cluster, hence getting assigned an invalid value (as if it were a source). In addition, devices might greedily select a neighbour with an invalid value or which just entered the cluster, in both cases propagating the errors.

[12] We shall see in the next section the *channel*, in which this situation happens.

```
def broadcast(value, null, grad) {
  G'(value, null, (v) => v, grad)
}
```

```
def distance(grad_source, grad_dest) {
  let loc = min([grad_source, grad_dest], [grad_dest, grad_source]) in
  broadcast(2nd(loc), -Infinity, 1st(loc))
}
```

```
def elliptic-channel(grad_source, grad_dest, source_dest, width) {
  sqr(grad_source + grad_dest) <= sqr(source_dest) + sqr(width)
}
```

```
def rectangular-channel(grad_source, grad_dest, source_dest, width) {
  let s = sqr(grad_source) - sqr(width/2) in
  let d = sqr( grad_dest ) - sqr(width/2) in
  4*s*d <= sqr(sqr(source_dest) - s - d)
}
```

```
def channel-communication(value, null, source, destination, width, gradient) {
  let grad_source = gradient(source) in
  let grad_dest   = gradient(destination) in
  let source_dest = distance(grad_source, grad_dest) in
  let channel = rectangular-channel(grad_source, grad_dest, source_dest, width) in
  if (channel) { broadcast(value, null, grad_source) } { null }
}
```

Figure 9: Generic code for broadcasting and channel establishment.

These issues can be overcame by tweaking the function `G` through the addition of a `null` value, as shown in function `G'` (Figure 8). In this function, the `initial` value is assigned only in sources, and `null` is used in its place everywhere else. When a neighbour has to be selected for propagation, it is thus possible to discard the neighbours with a `null` value, implementing a preference for values originating from real sources.

4.2. Channel Communication and Broadcast

Gradient algorithms and other instances of the G block can be composed together to form increasingly complex patterns. Among them, the *channel* pattern aims at selecting a geometrically-shaped region of devices, suitable to be used as a "communication channel" for broadcasts from a source (or other distributed computations), and has the remarkable feature of *not* involving any other building block (C, T, etc.). Thus, we chose it as an example for experimentally evaluating the performance of BIS and other gradient algorithms in broadcast-like behaviour.

Figure 9 presents the functions involved in the channel pattern, parametric in a given `gradient` algorithm. Firstly, a `broadcast` function is easily implemented by calling block `G'` with an identity `accumulate` function. Then, a single broadcast is used to disseminate in the whole network information

about the distance between a given source and destination. This could be accomplished through a simple `broadcast`(`grad_source`, `grad_dest`), which broadcasts from the destination its distance to the source: however, function `distance` slightly improves this method by simultaneously broadcasting both from the source and from the destination.

Given the total distance t between source and destination, a maximum width w of the channel, and the distances s, d of a device from source and destination, we are able to compute whether the device belongs to the corresponding channel of a given shape. Firstly, we can assume the channel to be elliptical, so that the maximum width corresponds to the minor axis and t to the focal distance: in this case, a device belongs to the ellipse if and only if $s + d \leq \sqrt{t^2 + w^2}$ (as implemented in function `elliptic-channel`). We can also assume instead the channel to be rectangular, gaining the added benefit of a constant width throughout all of its length. In this case, assuming that $s, d \geq r = w/2$, a device belongs to the rectangle if and only if

$$\sqrt{s^2 - r^2} + \sqrt{d^2 - r^2} \leq t$$

since $\sqrt{s^2 - r^2}$, $\sqrt{d^2 - r^2}$ are the projected left and right distances of rectangle points to the segment joining distance and destination. If we rewrite the above equation removing roots, we obtain

$$4 \cdot (s^2 - r^2) \cdot (d^2 - r^2) \leq (t^2 - (s^2 - r^2) - (d^2 - r^2))^2$$

which also works[13] for $s, d \leq r$ and is implemented in `rectangular-channel`.

Finally, all of these functions can be combined together to form the final channel-broadcast pattern. After distances for the given device from source and destination are computed using the given `gradient` algorithm, the distance between source and destination is collected by function `distance`. Whether the device belongs to the channel or not is then stored in `channel`, and a broadcast is performed in the cluster corresponding to the channel.[14]

[13] More precisely, the above equation defines a locus of points consisting of a rectangle from the source to the destination of given width, augmented on the short sides by small circumferences arcs (centered in the destination or source and pointing at an opposite corner of the rectangle).

[14] This pattern is useful whenever the values to be broadcast are significantly heavier than simple distance estimates, so that the complexity of the function is dominated by the final broadcast.

4.3. Data Collection

In Section 4.1 we argued that gradient algorithms are closely related to the G building block, and can in fact be extended to cover its behaviour. However, gradient algorithms are also crucial for the computation of the C building block, as they provide a necessary input to it: an estimate of the distance from the sources, to guide data flow towards them.

The two main state-of-the-art implementations of the C building block are called *single-path* and *multi-path*, and are thoroughly discussed in Viroli et al. [14]. As inputs, they receive a `value` to be aggregated (for every device), an `accumulate` function with a corresponding `null` value (e.g., + with 0), a `gradient` directing the aggregation flow, and a `root` function that "inverts" the effects of `accumulate` (only for multi-path C). The purpose of both algorithms is to compute in the gradient sources the overall network aggregate of the given values with the given accumulating function: e.g., the sum of all values if `accumulate` is +, which in turn may provide an estimate of the overall average when divided by the number of devices (obtained by summing 1 for each device through C).

The single-path algorithm computes the aggregate by flowing data through the spanning tree of shortest paths from the source, obtained by selecting a single parent for each device. While this algorithm might be preferable in static networks, the multi-path algorithm is usually more efficient whenever variable inputs are taken into account (moving devices, etc.).

The multi-path algorithm computes the aggregate by flowing data through all possible paths in the network, as in the following code that relies on the built-in functions `countHood`, which counts the number of `true` values in a neighbouring field value, and `foldHood`$(\texttt{f}, \phi, \ell)$, which aggregates with function `f` the initial value ℓ with every value in ϕ.

```
def multi-path(value, null, accumulate, root, gradient) {
  1st(rep ([value, value]) { (old) =>
    let nchild = countHood(nbr{gradient} < gradient) in
    let toacc = mux(nbr{gradient} > gradient, nbr{2st(old)}, null) in
    let total = foldHood(accumulate, toacc, value) in
    [total, root(total, nchild)]
  })
}
```

In each device, the number of *children* (i.e., neighbours closer to the source) is computed in `nchild`. Then, a `total` value is obtained by aggregating the values "shared" by each *parent* (i.e., neighbour further away from the source) together with the local `value`. Finally, the `total` value is split into

`nchild` equal parts through `root(total, nchild)`,[15] and the reduced result is "shared" to children.

5. Analysis and Verification

We now evaluate empirically the performance of BIS gradient, by comparing it in several scenarios with respect to the state-of-the-art algorithms reviewed in Section 2. The parameters and characteristics of simulations are chosen according to a general methodology for evaluating approximated localised algorithms.[16] All the experiments made in this paper are available online.[17]

Section 5.1 presents the general methodology and discusses the requirements of an effective test scenario for gradient algorithms. Section 5.2 evaluates the performance of gradient algorithms in isolation, as both *error* and *smoothness*. Section 5.3 evaluates their performance when used inside the *channel* pattern (see Section 4.2). Section 5.4 evaluates their performance when used inside the *multi-path C* building block (see Section 4.3).

5.1. Performance Indicators

In order to empirically evaluate the performance of an approximated localised algorithm, several aspects need to be taken into account. We divide them into *environment* characteristics, *input* properties, and *output* requirements.

Environment. A spatio-temporal computing environment is characterised by its degree of steadiness, which both in time and in space can be further specified through measures of *noise* and *variability*. We classify as *noise* the small high-frequency variations that are not intended to alter the expected output of the algorithm: in space, it corresponds to short-range Brownian movements; in time, it corresponds to random fluctuation in the frequency of events (in each device). We classify as

[15] For example, if accumulate is + then `root(total, nchild)` = `total/nchild`.

[16] With *approximated localised algorithm* we denote any spatially-distributed iterative process that aims to approximate a target global *input/output* transformation (taking into account environmental data, as described in Section 1). For example, this is the case for *gradient* algorithms that approximate *shortest-path distances* (output) given a source set and an environmental configuration (input).

[17] `https://bitbucket.org/gaudrito/scp-optimal-gradient`

variability the larger low-frequency variations that are intended to alter the expected output: in space, it corresponds to long-range directional movements; in time, it corresponds to systematic error in the frequency of events (changing between devices or through time).

Input. To assess the performance of an algorithm, we need to split tests into two further possible situations: *constant* input, to isolate and measure the responses to environment variations; *discontinuous* input, where a sudden change happens at a certain point in time, to measure the healing speed of the algorithm.

Output. Given a test environment and input, we need to measure two different qualities of the output generated by the algorithm: *precision* and *smoothness*. *Precision* is the deviation from the ideal outcome: with a constant input, it measures systematic error (e.g., *speed bias* for gradient algorithms); with a discontinuous input, it measures healing speed (e.g., *rising value* for gradient algorithms). *Smoothness* is the volatility of the output values, usually measured as the integral of absolute differences between consecutive values (first derivative of the output), and aims for gradual and unidirectional changes in the output values, absorbing noise. It needs to be measured both on constant and discontinuous input.

Performance assessment of approximated localised algorithms thus requires extensive testing over several different environments, combining diverse degrees of noise and variability (both in space and in time). Among the different possibilities, we find it generally appropriate to include: zero-noise zero-variability (in both space and time), in which the basic *self-stabilisation* property[18] is measured [28]; high-noise high-variability (in both space and time), in which a bottom line of guaranteed performance is measured in an extreme case; further intermediate cases, which can help differentiate how performance is affected by the different types of mutability (in space or time, as noise or variability), depending on the specific application. In each of those scenarios, performance is measured through *precision* and *smoothness*; on an input that is first constant for a long enough period of time to reach stable

[18] An algorithm is *self-stabilising* if given a constant environment, it eventually reaches a correct output for any possible initial state.

results, and then change discontinuously and keeps the new value constant until stable results are reached again.

5.2. Isolation Test: Swapping Source Corridor

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 2:

- *speed bias*, by considering environments with increasing variability in space;
- *rising value*, through arranging devices densely into a long corridor with a source at one end, so that the ratio between the longest and shortest distance between devices is high;
- *smoothness*, by measuring it in each test scenario.

Following the guidelines introduced in Section 5.1, we thus tested the following scenarios.

- *Environment:* we put 1000 devices with communication radius $10m$ and average fire rate $1s$ uniformly at random into a $500m \times 20m$ corridor, producing a network 50 hops wide. We tested this environment with increasing *variability* in space (long range movements) from 0 (none), to 0.5 (moderated) and 1 (high), and either zero or high[19] *noise* (both in space and in time) and *variability* in time. We modelled random intervals between rounds of computation according to a Weibull distribution [31], which is the most commonly used family of positive random variables (having exponential and Rayleigh distributions as special cases).
- *Input:* we provided the algorithms with a single source, steadily located on the left end of the corridor until time 300, and then abruptly moved to the opposite right end. 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).

[19] Brownian motion and 50% relative standard error in fire rate between different devices plus another 50% in each device.

- *Output:* for each scenario we measured *precision* as absolute error w.r.t. Euclidean distance and *smoothness* as absolute difference between values in consecutive rounds (both averaged).

Figure 10 summarises the evaluation results with high noise and time variability, which were obtained (similarly also to the experiments in next subsections) with Protelis [32] (an incarnation of the field calculus [11]) as programming language to code the model, Alchemist as simulator [33] and the Supercomputer OCCAM [34] to run the experiments. Results with zero noise and time variability are not shown as they were almost identical to the corresponding ones in Figure 10, showing that these variables do not affect significantly the behaviour of any of the considered algorithms. We tested classic, CRF, FLEX, BIS with $v = 0.5v_{\text{avg}}$, BIS with $v = 0.5v_{\text{avg}}$ and FLEX damping, BIS with $v = 0.9v_{\text{avg}}$, BIS with $v = 0.9v_{\text{avg}}$ and FLEX damping. The tolerance of the FLEX damping was set to 10%. We run 10 instances of each scenario with different random seeds and averaged the results.

The rising value problem corresponds to the spikes in the middle of the graphs, which are considerably shorter (faster healing) for BIS gradient, even when $v = 0.5v_{\text{avg}}$. Speed bias is visible from the increase in error baseline under increasing space variability, and is more contained for BIS gradient (in particular when v is high). The only setting where BIS does not achieve the best precision is under constant input and zero space variability, where the error value is still small and in fact determined by the small variations reported in smoothness.

As expected, the increase in precision corresponds to a decreased smoothness, so that BIS gradient has the highest value volatility (increasing with v). Embedding the FLEX damping into BIS proves to be effective in reducing fluctuations for all values of v, so that BIS with $v = 0.5v_{\text{avg}}$ and FLEX damping score better than CRF gradient and comparably similar to FLEX and classic gradients, while still achieving a much higher precision.

Overall, these results prove that BIS gradient achieves a much higher healing speed and accuracy (especially when v is high), while still keeping smoothness under control (especially when FLEX dumping is also used). This properties are readily appreciable in practical applications where inputs cannot be assumed to be constant: as we shall show in the next subsection, in these settings BIS gradient remains effective whereas other gradient algorithms fail to produce sensible results, disrupting the higher-order coordination mechanisms relying on them.

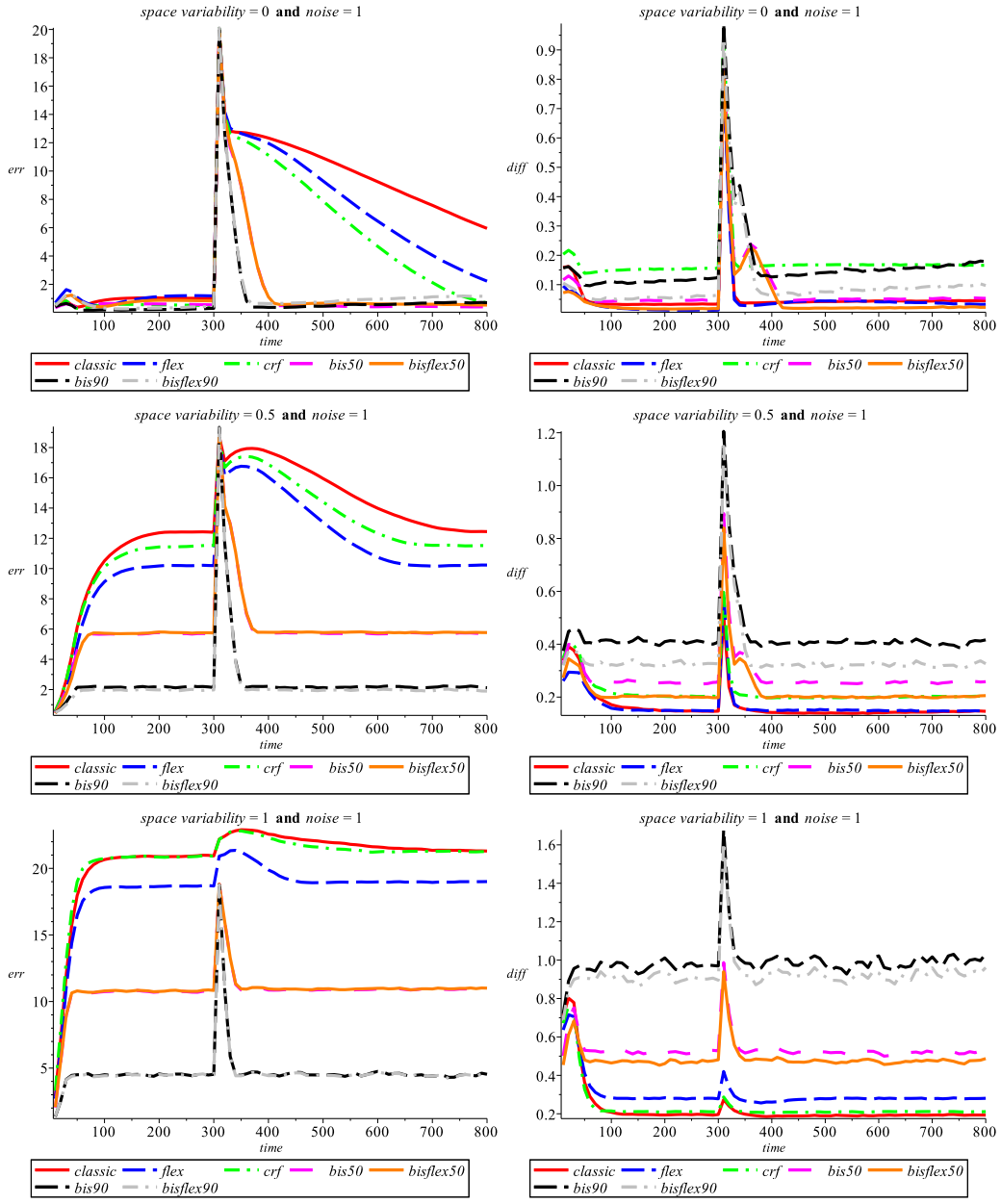

Figure 10: Precision (left) and smoothness (right) of gradient algorithms under increasing space variability (from top to bottom) and high noise and time variability.

5.3. Integration Test: Channel Maintenance

In order to compare the effectiveness of gradient algorithms as building blocks for complex patterns, we evaluated their performance when used in the channel pattern (see Section 4.2), which does not involve other building blocks, thus avoiding unwanted possible interplays between them. Following the guidelines (see Section 5.1), we thus tested the following scenarios.

- *Environment:* we put 1000 devices with communication radius $10m$ and average fire rate $1s$ randomly into a $200m \times 50m$ corridor, producing a network about 20 hops wide. We tested this environment by increasing simultaneously space noise, time noise and time variability (from 0 to 1 in the scale of Section 5.2). We did not consider space variability (long range movements), since it often disrupts the output of the channel pattern: in future works, we plan to further improve the algorithms in order to be able to apply the channel pattern also in these situations.

- *Input:* we provided the algorithms with a source and destination steadily located on opposite corners (south-west and north-east) of the corridor until time 300, and then abruptly moved to the opposite corners (south-east and north-west); so that 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 the average error between the ideal rectangular channel region and the actual region computed by the algorithms, and the delay with which messages from the source reach the destination. As a comparison, we also measured the analogous delay with a full broadcast (without the channel): however, we decided not to include them in the graphs since these delays are almost identical.

Figure 11 summarises the evaluation results. We tested classic, CRF, FLEX, BIS and BIS with FLEX damping (10% tolerance). We run 50 instances of each scenario with different random seeds and averaged the results, which had an average 7.2% relative standard error.

Regarding error in channel selection, all algorithms performed similarly under constant input except for CRF, which had significantly lower performance in scenarios with variability. Reactions to discontinuities, instead, presented a clear hierarchy of algorithms: BIS, followed by FLEX, CRF and classic (which was never able to rebuild a functioning channel before the end

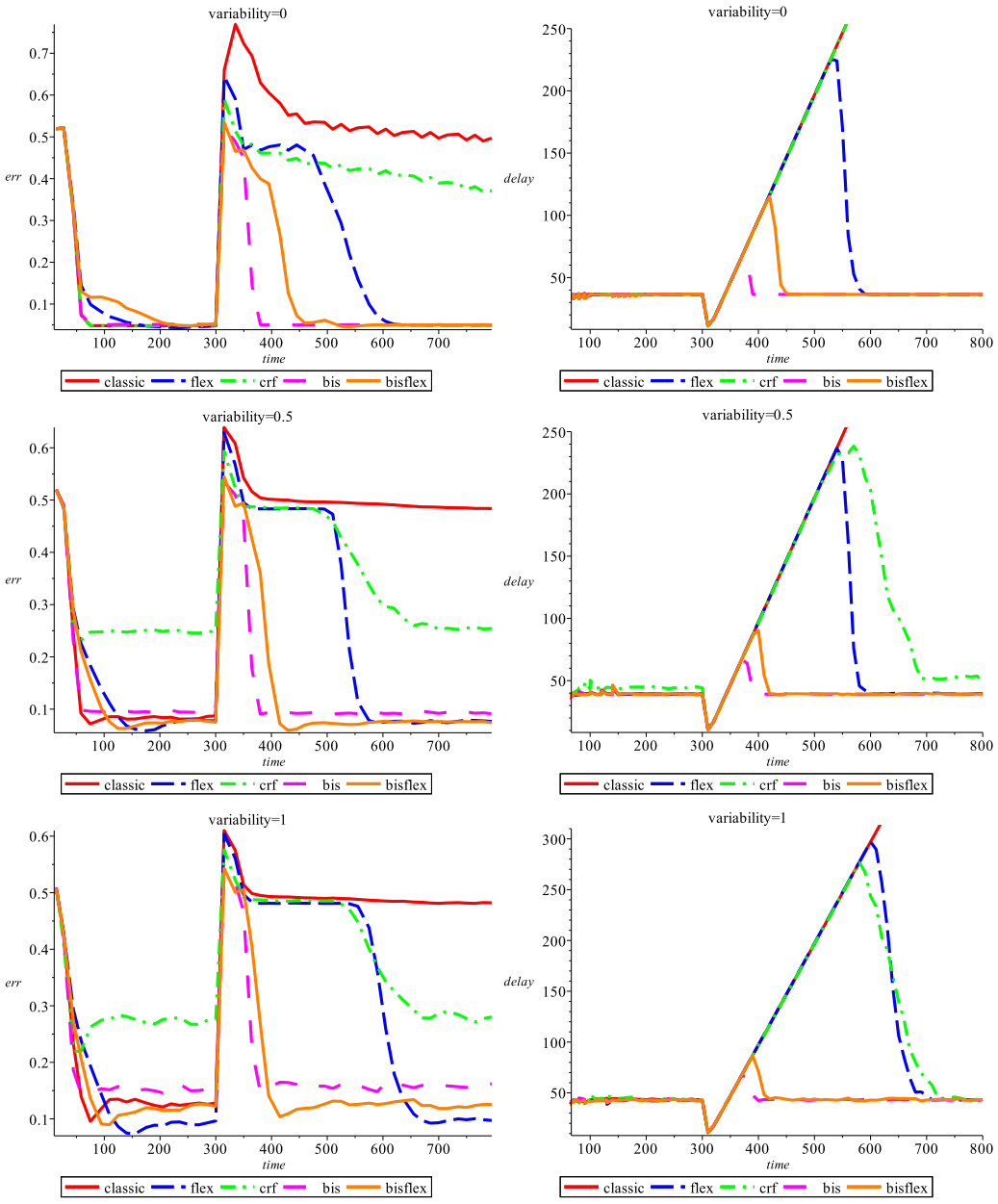

Figure 11: Error in channel selection (left) and efficiency of broadcasting through it (right) of gradient algorithms under increasing noise and variability (from top to bottom).

of the experiment). As predicted, the addition of FLEX damping slightly reduced BIS performance under no variability, while becoming increasingly better with high variabilities.

Regarding delay of broadcast messages, there was no significant difference between algorithms under constant input. The only relevant factor thus was the instant in which a functioning channel was rebuilt, so that BIS always scored better than BIS with FLEX damping, followed by FLEX and CRF (which was increasingly better under increasing variability, in spite of the high error in channel selection), with classic scoring worst as it were never able to recover by the end of the experiment.

5.4. Integration Test: Data Collection

As a further inspection on gradient algorithms as building blocks, we evaluated their effectiveness when used to perform data collection through block *multi-path C* (see Section 4.2). Following the guidelines (see Section 5.1), we thus tested the following scenarios.

- *Environment:* as in Section 5.2, we put 1000 devices with communication radius $10m$ and average update rate $1s$ randomly into a $500m \times 20m$ corridor. We tested this environment with increasing *space variability* from 0 to 0.2 and *noise* from 0 to 0.5 (in the scale of Section 5.2). Higher values significantly impaired the performance of all algorithms considered.

- *Input:* we provided the algorithms with a single source, steadily located on the left end of the corridor until time 300, and then abruptly moved to the opposite right end. We made the source 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.

- *Output:* for each scenario we measured the absolute error between the source count and the actual number of devices.

Figure 12 summarises the evaluation results. We tested classic, CRF, FLEX, BIS and BIS with FLEX damping (10% tolerance). We run 50 instances of each scenario with different random seeds, obtaining results with a significant relative standard error (varying from 41% to 203%). We thus

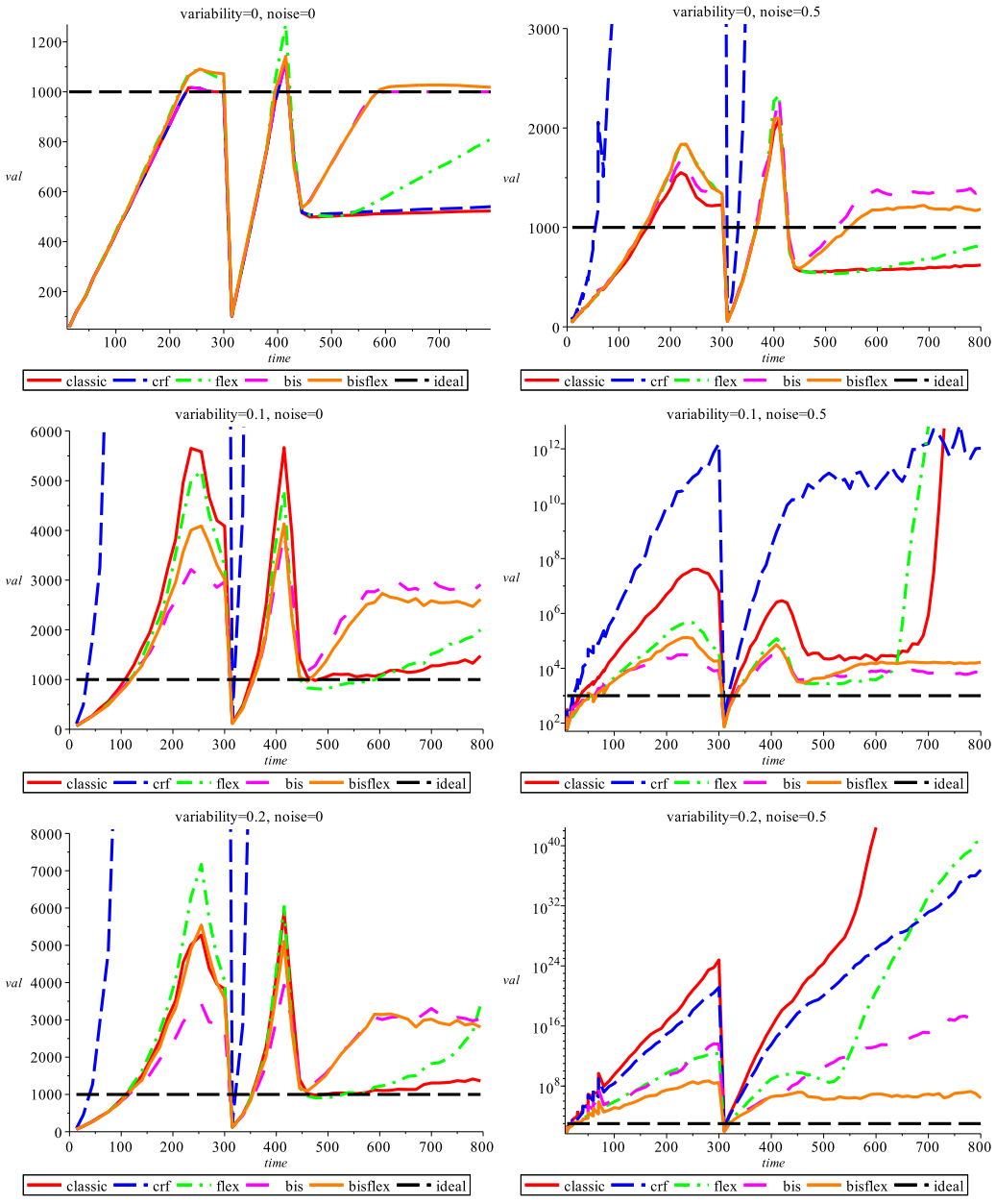

Figure 12: Error in device count of gradient algorithms under increasing noise (from left to right) and space variability (from top to bottom).

aggregated the results by selecting the *median* (instead of the *mean*), which is more reliable in highly volatile settings.

An high systematic error was reported, leading all algorithms to overestimate values as noise or variability increased. In scenarios with space variability the error was significant (above 200% for all algorithms), whereas in scenarios without variability the error stayed below 10%. In scenarios with sufficiently high noise and variability, all algorithms exhibited an exponential behaviour. However, the threshold above which this occurred was different for different algorithms: in increasing order, CRF, FLEX, classic, BIS, BIS with FLEX damping. Overall, this observations confirm the weaknesses of *multi-path C* already reported in previous works [35].

CRF gradient proved to be highly unstable in all scenarios with either noise or variability, obtaining virtually meaningless results (errors above 10^{10}%). Even in the scenario with zero noise and variability, CRF gradient obtained the worst performance, not being able to noticeably recover before the end of the simulation.

Before the discontinuity, the other algorithms behaved similarly, with BIS gradient (with or without damping) performing better in presence of space variability. After the discontinuity, BIS (with or without damping) was able to fully recover much faster (in about $300s$), followed by FLEX which had approximately 50% recovered by the end of the simulation. In scenarios with space variability and zero noise, this slow recovery helped classic and FLEX gradients to obtain a lower error, since the recovery process counteracted the overestimation for a longer transient. However, the fast recovery still accounts in favour of BIS gradient, as improved versions of the C block might reduce the systematic error and thus benefit from an increased reactivity.

6. Conclusions and Future Works

This paper focusses on the problem of designing resilient and reactive algorithms to construct *gradient* data structures in dynamic environments, facing a wide range of possible perturbations to network topology (due to faults or devices being mobile) and inputs (source of the gradient). To this end, we have introduced BIS gradient, a new gradient algorithm of optimal self-healing speed among algorithms with a single-path communication scheme. Mathematical estimations to guide the selection of the proper parameter v for the target speed of change are provided. Then, validation of the proposed approach, with comparison with existing algorithms, has been

carried out in a variety of settings: *(i)* the gradient block in isolation under a wide variety of different conditions, *(ii)* usages of the gradient to broadcast information and create spatial structures, and *(iii)* to collect and summarise data items. In the future, experiments based on real traces could be carried out to further improve the confidence on the performance of this algorithm, and to more clearly identify practical situation in which it can be favoured. Overall, BIS is shown to better adapt to dynamic environments and react to changes with respect to previous algorithms.

We believe, however, that there is still some margin for further improvements. For instance, some form of broadcast could be used to surpass the theoretical limit given by single-path communication speed of BIS. *Smoothness* could be further improved by fine-tuning other damping functions other than the one given by the FLEX gradient. The *speed bias* could be addressed directly by introducing a metric distortion dependent on the movement speed of devices (in a similar way as it is done in [26]), and several mobility models could be considered to fine-tune both the information speed estimate and the metric distortion. Additionally, it is possible that variants of the proposed algorithm can provide additional benefits in specific applications of the gradient pattern; in particular, we are interested in the cases where gradients are used to support distributed sensing of information in highly heterogeneous and dense environments, specifically for crowd engineering applications. The ultimate goal of this research thread is to put forward a theoretical and practical framework for building self-stabilising distributed systems for IoT-like applications, where load-balancing, tuning of performance, and resiliency to changes in working conditions can be automatically managed by the underlying platform, without behaviour specification being affected at all.

Acknowledgements

We thank the anonymous COORDINATION and SCP referees for their comments and suggestions for improving the presentation.

[1] J. Beal, S. Dulman, K. Usbeck, M. Viroli, N. Correll, Organizing the aggregate: Languages for spatial computing, in: M. Mernik (Ed.), Formal and Practical Aspects of Domain-Specific Languages: Recent Developments, IGI Global, 2013, Ch. 16, pp. 436–501. doi:10.4018/978-1-4666-2092-6.ch016.

[2] N. Bicocchi, M. Mamei, F. Zambonelli, Self-organizing virtual macro sensors, TAAS 7 (1) (2012) 2:1–2:28. doi:10.1145/2168260.2168262.

[3] J. Beal, Flexible self-healing gradients, in: Proceedings of the 2009 ACM Symposium on Applied Computing (SAC), 2009, pp. 1197–1201. doi:10.1145/1529282.1529550.

[4] J. L. Fernandez-Marquez, G. D. M. Serugendo, S. Montagna, M. Viroli, J. L. Arcos, Description and composition of bio-inspired design patterns: a complete overview, Natural Computing 12 (1) (2013) 43–67. doi:10.1007/s11047-012-9324-y.

[5] A. Lluch-Lafuente, M. Loreti, U. Montanari, Asynchronous distributed execution of fixpoint-based computational fields, Logical Methods in Computer Science 13 (1). doi:10.23638/LMCS-13(1:13)2017.

[6] M. Viroli, F. Damiani, A calculus of self-stabilising computational fields, in: Coordination Languages and Models, Vol. 8459 of LNCS, Springer-Verlag, 2014, pp. 163–178. doi:10.1007/978-3-662-43376-8_11.

[7] J. Beal, D. Pianini, M. Viroli, Aggregate programming for the Internet of Things, IEEE Computer 48 (9) (2015) 22–30. doi:10.1109/MC.2015.261.

[8] S. Montagna, M. Viroli, J. L. Fernandez-Marquez, G. Di Marzo Serugendo, F. Zambonelli, Injecting self-organisation into pervasive service ecosystems, Mobile Netw Appl 18 (3) (2013) 398–412. doi:10.1007/s11036-012-0411-1.

[9] J. Beal, J. Bachrach, D. Vickery, M. M. Tobenkin, Fast self-healing gradients, in: Proceedings of the 2008 ACM Symposium on Applied Computing (SAC), 2008, pp. 1969–1975. doi:10.1145/1363686.1364163.

[10] G. Audrito, F. Damiani, M. Viroli, Optimally-self-healing distributed gradient structures through bounded information speed, in: Coordination Models and Languages, Vol. 10319 of LNCS, Springer, 2017, pp. 59–77. doi:10.1007/978-3-319-59746-1_4.

[11] F. Damiani, M. Viroli, D. Pianini, J. Beal, Code mobility meets self-organisation: a higher-order calculus of computational fields, in: S. Graf, M. Viswanathan (Eds.), Formal Techniques for Distributed Objects, Components, and Systems, Vol. 9039 of Lecture Notes in Computer Science, Springer International Publishing, 2015, pp. 113–128. doi:10.1007/978-3-319-19195-9_8.

[12] M. Viroli, G. Audrito, F. Damiani, D. Pianini, J. Beal, A higher-order calculus of computational fields, arXiv preprint, arXiv:1610.08116.

[13] M. Viroli, F. Damiani, J. Beal, A calculus of computational fields, in: Advances in Service-Oriented and Cloud Computing, Vol. 393 of Communications in Computer and Information Science, Springer, 2013, pp. 114–128. doi:10.1007/978-3-642-45364-9_11.

[14] M. Viroli, J. Beal, F. Damiani, D. Pianini, Efficient engineering of complex self-organising systems by self-stabilising fields, in: Self-Adaptive and Self-Organizing Systems (SASO), IEEE 9th International Conference on, IEEE, 2015, pp. 81–90. doi:10.1109/SASO.2015.16.

[15] M. Viroli, G. Audrito, J. Beal, F. Damiani, D. Pianini, Engineering resilient collective adaptive systems by self-stabilisation, ACM Trans. Model. Comput. Simul. 28 (2) (2018) 16:1–16:28. doi:10.1145/3177774.

[16] M. Viroli, M. Casadei, Biochemical tuple spaces for self-organising coordination, in: J. Field, V. T. Vasconcelos (Eds.), Coordination Models and Languages: 11th International Conference. Proceedings, Springer, 2009, pp. 143–162. doi:10.1007/978-3-642-02053-7_8.

[17] M. Viroli, D. Pianini, J. Beal, Linda in space-time: An adaptive coordination model for mobile ad-hoc environments, in: Coordination Models and Languages: 14th International Conference. Proceedings, 2012, pp. 212–229. doi:10.1007/978-3-642-30829-1_15.

[18] G. Castelli, M. Mamei, A. Rosi, F. Zambonelli, Engineering pervasive service ecosystems: The SAPERE approach, TAAS 10 (1) (2015) 1:1–1:27. doi:10.1145/2700321.

[19] N. Elhage, J. Beal, Laplacian-based consensus on spatial computers, in: W. van der Hoek, G. A. Kaminka, Y. Lespérance, M. Luck, S. Sen (Eds.), 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010), Volume 1-3, 2010, pp. 907–914. doi:10.1145/1838206.1838328.

[20] M. Viroli, M. Casadei, S. Montagna, F. Zambonelli, Spatial coordination of pervasive services through chemical-inspired tuple spaces, ACM Transactions on Autonomous and Adaptive Systems 6 (2) (2011) 14:1 – 14:24. doi:10.1145/1968513.1968517.

[21] J. Bachrach, J. Beal, J. McLurkin, Composable continuous-space programs for robotic swarms, Neural Computing and Applications 19 (6) (2010) 825–847. doi:10.1007/s00521-010-0382-8.

[22] J. Giavitto, O. Michel, J. Cohen, A. Spicher, Computations in space and space in computations, Tech. rep. (2004). doi:10.1007/11527800_11.

[23] F. Damiani, M. Viroli, J. Beal, A type-sound calculus of computational fields, Science of Computer Programming 117 (2016) 17 – 44. doi:10.1016/j.scico.2015.11.005.

[24] J. L. Fernandez-Marquez, A. Tchao, G. D. M. Serugendo, G. Stevenson, J. Ye, S. Dobson, Analysis of new gradient based aggregation algorithms for data-propagation in mobile networks, in: Sixth IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW 2012), IEEE Computer Society, 2012, pp. 217–222. doi:10.1109/SASOW.2012.45.

[25] J. Beal, M. Viroli, Building blocks for aggregate programming of self-organising applications, in: 2nd FoCAS Workshop on Fundamentals of Collective Systems, IEEE CS, 2014, pp. 8–13. doi:10.1109/SASOW.2014.6.

[26] Q. Liu, A. Pruteanu, S. Dulman, Gradient-based distance estimation for spatial computers, Comput. J. 56 (12) (2013) 1469–1499. doi:10.1093/comjnl/bxt124.

[27] R. Nagpal, H. E. Shrobe, J. Bachrach, Organizing a global coordinate system from local information on an ad hoc sensor network, in: Information Processing in Sensor Networks, Second International Workshop (IPSN 2003). Proceedings, 2003, pp. 333–348. doi:10.1007/3-540-36978-3_22.

[28] F. Damiani, M. Viroli, Type-based self-stabilisation for computational fields, Logical Methods in Computer Science 11 (4). doi:10.2168/LMCS-11(4:21)2015.

[29] E. M. Royer, C. Toh, A review of current routing protocols for ad hoc mobile wireless networks, IEEE Personal Commun. 6 (2) (1999) 46–55. doi:10.1109/98.760423.

[30] A. Stuart, J. K. Ord, Kendall's advanced theory of statistics. Vol. 1, sixth Edition, Edward Arnold, London; copublished in the Americas by Halsted Press - John Wiley & Sons, Inc. , New York, 1994.

[31] W. Weibull, et al., A statistical distribution function of wide applicability, Journal of applied mechanics 18 (3) (1951) 293–297.

[32] D. Pianini, M. Viroli, J. Beal, Protelis: practical aggregate programming, in: Proceedings of the 30th Annual ACM Symposium on Applied Computing, 2015, pp. 1846–1853. doi:10.1145/2695664.2695913.

[33] D. Pianini, S. Montagna, M. Viroli, Chemical-oriented simulation of computational systems with ALCHEMIST, J. Simulation 7 (3) (2013) 202–215. doi:10.1057/jos.2012.27.

[34] M. Aldinucci, S. Bagnasco, S. Lusso, P. Pasteris, S. Vallero, 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.

[35] G. Audrito, R. Casadei, F. Damiani, M. Viroli, Compositional blocks for optimal self-healing gradients, in: 11th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2017), 2017, pp. 91–100. doi:10.1109/SASO.2017.18.