Block-synchronous harmonic control for scalable trajectory planning

Bernard Girau, Amine Boumaza, Bruno Scherrer, Cesar Torres-Huitzil

4

Block-synchronous harmonic control for scalable trajectory planning

Bernard Girau•, Amine Boumaza•, Bruno Scherrer•, Cesar Torres-Huitzil*
• *LORIA – INRIA Nancy Grand Est, France*
* *Polytechnic University of Victoria, Mexico*

1. Introduction

Trajectory planning consists in finding a way to get from a starting position to a goal position while avoiding obstacles within a given environment or navigation space. Harmonic functions may be used as potential fields for trajectory planning (Connolly et al., 1990). Such functions do not have local extrema (unlike other potential based methods as in (Khatib, 1986)), so that control algorithms may reduce to *locally* ascend the potential until they reach a *global* maximum, when obstacles correspond to minima of the potential and goals correspond to maxima.

Harmonic control has had some impact on the robotics community (Masoud & Masoud, 2002; Zelek, 1998; Alvarez et al., 2003; Feder & Slotine, 1997; Huber et al., 1996; Kazemi et al., 2005; Sweeney et al., 2003; Wang & Chirikjian, 2000). Nevertheless, very few hardware implementations have been proposed. They are usually analog, therefore they suffer from a very long and complex design process, and a lack of flexibility (environment size, precision). This chapter presents a parallel hardware implementation of this navigation method on reconfigurable digital circuits. Trajectories are estimated after the iterated computation of the harmonic function, given the goal and obstacle positions of the navigation problem. The proposed implementation locally computes the direction to choose to get to the goal at any point of the environment. Dynamic changes in this environment may be taken into account, for example when obstacles are discovered during an on-line exploration.

To fit real-world applications, our implementation has been designed to deal with very large environments while optimizing computation time. To do so, iterated updates are performed in a block-synchronous mode that takes advantage of large embedded SRAM memory resources. The proposed implementation maps the massively distributed structure of the space-discretized estimation of the harmonic function onto the circuit. When the size of the environment of navigation exceeds the resources available on the circuit, this environment is split into several so-called blocks.

For each block, an area-saving serial arithmetic is used within a global scheme that simultaneously ensures pipelining and parallelism of the iterative computations. These

computations are scheduled for the different blocks so as to make a compromise between pipelining efficiency and block coherency. Moreover, an increasing precision is used throughout the convergence process, so as to further optimize computing times. Thanks to this increasing precision of the chosen serial arithmetic, first iterated updates are faster, the convergence of each block is reached sooner, and the serial detection of this convergence is also faster. When the whole process has converged for a given precision, iterated updates start again with an increased precision. This process is repeated until the necessary precision of the harmonic function estimation is reached.

Our implementation is able to handle up to 16 blocks of size 96x48 (73728 nodes) with a 64-bit precision on a single FPGA Xilinx Virtex XC4VLX160. The main module consists of an array of identical nodes that compute the iterated updates of the harmonic function estimation. Specialized modules have been designed to handle communications between contiguous blocks. Convergence detection is performed in a pyramidal way. All nodes communicate with a decision module that computes the navigation direction by means of a local interpolation of the discretized harmonic function. Obstacles and goals may be dynamically added and memorized for any block, resulting in a new iterated process to update the harmonic function estimation.

Besides all implementation works, we carefully justify our algorithmic and technological choices through both theoretical and empirical studies of the required precisions and convergence times. We study the asymptotic convergence of the proposed algorithmic scheme and we analytically derive some bounds on its rate of convergence. The implementation results together with this theoretical analysis show that the proposed architecture simultaneously improves speed, power consumption, precision, and allows to tackle large environment sizes.

Section 2 describes the principles of harmonic functions and of their use for trajectory planning. Section 3 summarizes the advantages of hardware parallel implementations on FPGAs for embedded navigation in autonomous systems, and it justifies the choice of serial arithmetics, especially with respect to the precision requirements of harmonic control. Section 4 defines the block-synchronous computation scheduling, and its optimization analysis through theoretical and experimental results. The proposed hardware architecture and its implementation results are described in section 5.

2. Harmonic control

In this part, we begin by a brief reminder of harmonic functions and some of their properties Then we discuss their application to the navigation problem.

2.1 Harmonic functions

Let Ω be an open subset of $\Re^n$, $\partial\Omega$ its boundary and $\overline{\Omega}$ its closure such that $\overline{\Omega} = \Omega \cup \partial\Omega$.

Definition Let $u : \overline{\Omega} \to \Re$ be a real function, twice continuously differentiable, and

$\Omega \subset \Re^n$ with $n > 1$. Function u is harmonic iff $\Delta u = \sum_{i=1}^{n} \frac{\partial^2 u}{\partial x_i^2} = 0$ (Laplace's equation).

Harmonic functions satisfy interesting properties:

- The maximum principle states that: if u is a non-constant continuous function on $\overline{\Omega}$ that is harmonic on Ω, then u attains its maximum and minimum values over $\overline{\Omega}$ on the boundary $\partial\Omega$.
- Applying the divergence theorem on harmonic functions, the following equation holds: $\int_s \overrightarrow{\nabla u}.\vec{n}ds = 0$

where s is the boundary of a closed region strictly in Ω and $\vec{n}$ is a normal vector of s. This equation expresses that the normal flow of the gradient vector field through the region bounded by s is zero. It follows that there can be no local minimum or maximum of the potential inside a bounded region of Ω.

2.2 Application to navigation

To solve the navigation problem using harmonic functions, we consider the problem as a Dirichlet problem: its solution is to find the function u that is harmonic on Ω (navigation space) and that satisfies boundary conditions on $\partial\Omega$ (obstacles and navigation goal):

$$\begin{cases} \forall x \in \Omega, & \Delta u(x) = 0 \\ \forall x \in \partial\Omega, & u(x) = g(x) \end{cases} \tag{1}$$

where the function $g : \partial\Omega \to \Re$ represents boundary conditions on $\partial\Omega$. These conditions define the values of the navigation function on obstacles and goals. Without loss of generality we choose $g(x) = 0$ for obstacles and $g(x) = 1$ for goals. Solving the Dirichlet problem consists in finding the function u that is harmonic on Ω and that has value 0 on obstacle positions and value 1 on goal positions.

The navigation problem is then solved as follows: a simple ascent along the gradient of u provides a trajectory towards a given goal from any starting position. The properties of harmonic functions ensure that such a path exists and it is free of local optima.

2.2.1 Numerical method to solve Laplace's equation

In this part we consider the case where $\Omega = \Re^2$. Traditionally, Laplace's equation is solved using methods from finite differences on a regular grid (discrete sampling of Ω). Using a Taylor approximation of the second derivatives we obtain the following discrete form of $u(x_1, x_2)$

$$\begin{aligned} 0 = \Delta_\delta u(x_1, x_2) = \frac{1}{\delta^2}\big[&u(x_1 + \delta, x_2) + u(x_1 - \delta, x_2) \\ &+ u(x_1, x_2 + \delta) + u(x_1, x_2 - \delta) - 4u(x_1, x_2)\big] \end{aligned} \tag{2}$$

where δ is the sampling of the grid nodes that represents Ω. In this form, the equation can be solved using relaxation methods such as Jacobi or Gauss-Seidel whose principle is to iteratively replace each node value with the simple average of its four neighbours. Figure 1 shows different trajectories generated by simulations using this numerical scheme.

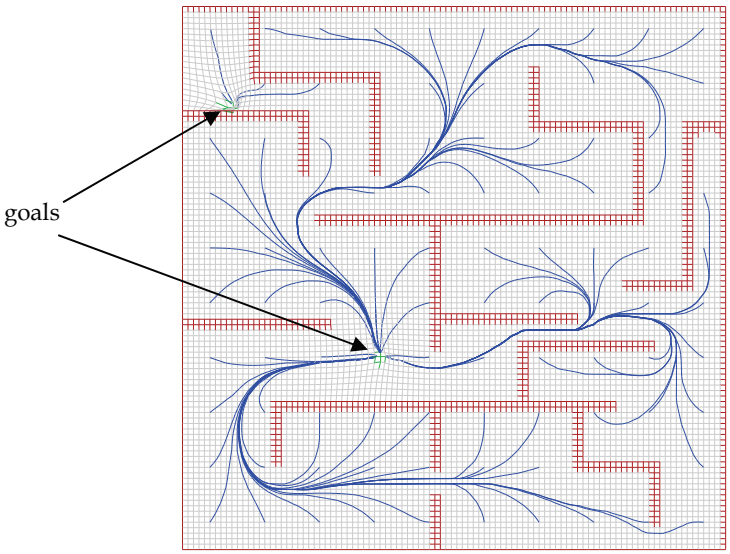

Fig. 1. Trajectories generated by harmonic control (100x100 grid, equally spread starting nodes, two goals)

2.2.2 Properties of harmonic navigation functions

Harmonic navigation functions have many interesting properties which motivated their use in numerous applications especially in robotics (Tarassenko & Blake, 1991; Connolly & Grupen, 1993; Masoud & Masoud, 2002; Zelek, 1998; Alvarez et al., 2003; Feder & Slotine, 1997; Huber et al., 1996; Kazemi et al., 2005; Sweeney et al., 2003; Wang & Chirikjian, 2000):

Global navigation: Complete trajectories may be generated towards a goal position from anywhere in the environment, since there are no local minima.

Dynamic trajectory planning: Unexpected updates of the environment may be taken into account, since harmonic functions are computed by iterative relaxation methods. Therefore newly detected obstacles may be integrated in the model as new boundary conditions during computation, so that harmonic control is available in dynamic environments or in environments explored on-line (Zelek, 1998; Boumaza & Louchet, 2003).
Parallel computation: An interesting property of the computations described above is their massively parallel distribution. Computing node values only requires local information of the neighbouring nodes. Fine-grain parallel implementations appear as an opportunity, as discussed in the next section.

3. Towards a rapid and scalable implementation

The results shown in figure 1 were obtained from software simulations carried out on a PC. The aim of our work is to design an embedded system for robot navigation. Computation speed is not the only criterion (trajectory decision must be performed in real time). Power consumption is also essential for autonomy. Besides, computation precision and scalability appear as critical issues for harmonic control, as discussed below. These combined aspects motivate the design of a parallel hardware implementation. In such a work, the number of inputs/outputs and above all the level of parallelism have a direct influence on the obtained implementation consumption and speed. A massively parallel implementation is a real challenge, taking into account constraints such as precision, grid size, dynamic updates, etc.

3.1 Implementation environment

Since the appearance of configurable circuits, such as *field programmable gate arrays* (FPGAs), algorithms may be implemented on fast integrated circuits with software-like design principles. VLSI designs lead to very high performances. But the time needed to realize an ASIC (application specific integrated circuit) is too long, especially when different configurations must be tested. The chip production time may take up to 6 months.

FPGAs, such as Xilinx FGPA (Xilinx, 2000), are based on a matrix of *configurable logic blocks* (CLBs) that contain a few logic cells. Each logic cell is able to implement small logic functions (4 or 5 inputs) with a few elementary memory devices (flip-flops or latches) and some multiplexors. Each logic cell is independently programmable. Similarly, the routing structure that connects the logic cells as well as the CLBs can be configured. A FPGA approach simply adapts to the handled application, whereas a usual VLSI implementation requires costly rebuildings of the whole circuit when changing some characteristics. A design on FPGAs requires the description of several operating blocks. Then the control and the communication schemes are added to the description, and an automatic ``compiling'' tool maps the described circuit onto the chip.

3.2 Technological choices

When a massively distributed model is mapped onto a FPGA, the main issues are the huge number of operators, and the routing problems due to the dense interconnections. A first standard technological choice to solve these problems is to use serial arithmetics: smaller operators may be implemented, and they require less connection wires, at the cost of several clock cycles to handle each arithmetic operation. Another essential technological choice is to estimate the minimum precision required to keep satisfactory results with as small as possible operators and memory resources.

3.2.1 Serial arithmetics

Serial arithmetics correspond to computation architectures where digits are provided digit after digit. Serial arithmetics lead to operators that need small implementation areas and less inputs/outputs, and that easily handle different precisions, without an excessive increase of the implementation area. Serial systems are characterised by their delay, i.e, the number d such that p digits of the result are deduced from $p+d$ digits of the input values.

Two main kinds of serial arithmetics are available: LSBF or MSBF (least/most significant bit first). Standard serial operators work in a LSBF mode. Though our model requires the

computation of maximum values (gradient ascent, detection of local maxima), that may only be computed in a MSBF mode, we mostly use standard LSBF serial operators to optimize the required area of the main computations[1]. Nevertheless, our implementation simultaneously handles a read access in MSBF mode to detect local maxima: since we use dual port SRAM blocks with fully independant ports, we take advantage of two simultaneous read addresses (controled by two reverse counters in the control modules).

3.2.2 Computation precision

Software simulation are usually performed to study the precision that is required by an application before its hardware implementation. Precision issues appear as a critical problem for harmonic control. It has already been mentioned as a major limitation for analog implementations (Tarassenko & Blake, 1991). Computing a harmonic potential over a large grid may result in gradients too small to use because the allowable precision is easily reached. Connolly (Connolly & Grupen, 1993) proposes a relationship between the precision required for floating point representation on a PC and the number of nodes on the grid. He argues that the precision should at least represent $1/N$ (requiring at least $\log_2 N$ bits), where N is the total number of grid nodes, to circumvent precision problems.

We may first discuss Connolly's estimation from a theoretical point of view. We argue that $1/N$ is not a sufficient precision for some kinds of environments. More precisely, we argue that the precision might have to represent at least $1/2^{O(L)}$ (therefore requiring some $O(L)$ bits), where L is the maximum trajectory length in the environment. To prove this assertion, let us consider a "corridor" of length L and width 1, with an obstacle on the left ($x_0 = 0$), and the goal at the other side ($x_L = 1$). At each intermediate node $0 < i < L$, according to equation (2), the following relation is fulfilled: $x_i = \frac{1}{4}(x_{i-1} + x_{i+1})$

This is a linear recurrent series of order 2. The roots of the associated polynomial are:

$$r_+ = 2 + \sqrt{3} \qquad r_- = 2 - \sqrt{3}$$

so that the generic term of the series is

$$x_i = \lambda r_+^i + \mu r_-^i$$

Since $x_0 = 0$ and $x_L = 1$, we finally obtain $x_i = \frac{(2+\sqrt{3})^i - (2-\sqrt{3})^i}{(2+\sqrt{3})^L - (2-\sqrt{3})^L}$

When $L \to \infty$: $x_1 \approx \frac{2\sqrt{3}}{(2+\sqrt{3})^L}$

Since the used precision must at least ensure that $x_0 \neq x_1$, the computation of the harmonic function in such an environment requires at least $O(L)$ bits, as argued above. A similar result

[1] The only existing radix-2 MSBF serial arithmetics is called *on-line* arithmetics. It uses a redundant number representations system, which induces less area-saving operators.

would be obtained in a square grid containing for example a spiral of length L and width 1, which can be obtained with an environment of approximately $2L$ nodes. This further motivates the use of an embeded implementation in which high precisions may be handled.

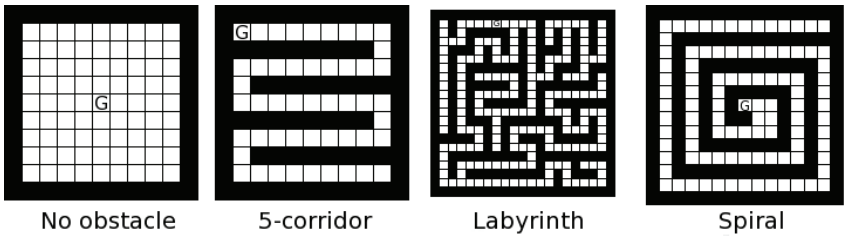

Fig. 2. Four different discrete environments: obstacles are in black, and 'G' denotes the goal.

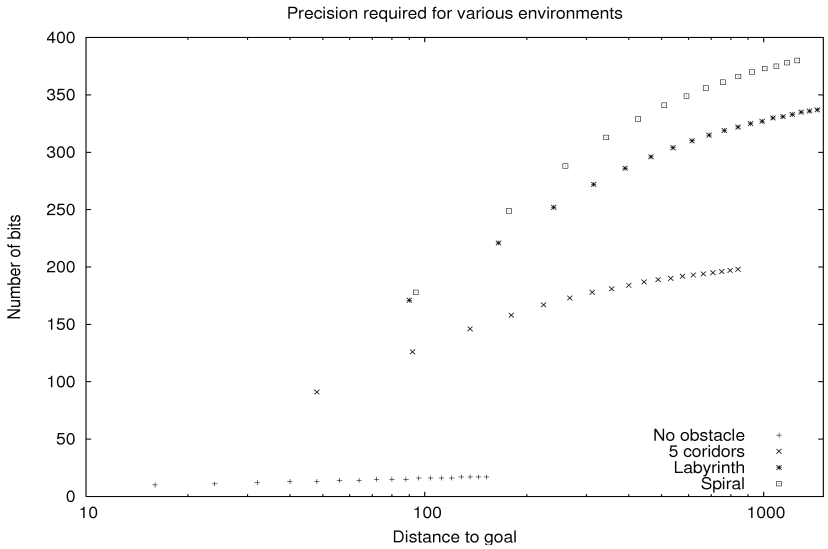

Fig. 3. Number of bits required with respect to the maximal distance L to the goal for different magnifications of the environments of Figure 2. The number of bits grows slightly slower than $O(\log(L))$.

Nevertheless, the study of the precision required by the computation of the harmonic function in a discretized environment should take "likely" environments into account. Therefore we have carried out numerous experiments with large randomly generated mazes. It follows that in most environments, the maximum distance to the goal is close to the square root of the environment size, and that a precision proportional to $1/L$ (i.e. a number of bits proportional to $\log(L)$) is generally sufficient to ensure that the computation of the harmonic function converges such that no local minimum or maximum exists (i.e. a trajectory is found from any node). Figure 2 shows some experimental environments, and figure 3 shows the minimum precision that is required with respect to distance L. It should

be pointed out that in case of insufficient precision, large areas of the grid may share the same value, hence a null gradient that results in incomplete trajectories.

Implementation note

Implementations based on serial arithmetics may be more easily extended to larger precisions than implementations based on parallel arithmetics. Since the size of serial adders and comparators does not depend on precision (unlike multipliers and elementary functions), our implementation may handle large precisions by means of rather simple changes in the control modules.

3.3 Dealing with very large environments

As mentioned above, scalability and precision are key issues for harmonic control in large environments. In this chapter, two major implementation choices deal with these issues.

3.3.1 Block-parallel computation

Despite the technological choices discussed above, the size of the discretized environment we are able to map in a fully parallel way onto FPGAs is limited (around 50x50 nodes in our preliminary work in (Girau & Boumaza, 2007)). To handle much larger environments (or finer discrete resolutions), we propose a block-synchronous (or block-parallel) implementation: the environment is partitioned into several blocks, each block of nodes being handled in a fully parallel way by the FPGA while the different blocks are sequentially handled. In this section, we formalize this computation scheduling, and we analyse both its constraints and its performance so as to optimize our implementation.

Let $N \times M$ be the total number of nodes (location units) in the discretized environment. Let $n \times m$ be the number of nodes that may be simultaneously handled on the FPGA. The environment is partitioned into $B = \frac{N}{n} \times \frac{M}{m}$ blocks of $n \times m$ nodes.

Implementation note

Our implementation performs computations in a block-synchronous way. Moreover up to I consecutive iterations are performed for each block before handling the next block, so as to improve the use of pipelining.

3.3.2 Increasing precision

Possible dynamic updates of the environment require fast computations so that the navigation trajectories adapt to these changes in real-time. Since the computation time in a serial implementation partially depends on the computation precision, and since several iterations are required to let our system converge to a good approximation of the expected harmonic function, we use an increasing precision so as to optimize the convergence time.

The first iterations are performed with a chosen reduced precision. When the whole system has converged for a given precision (for all blocks of nodes), iterated updates start again with an increased precision. This process is repeated until the necessary precision of the harmonic function estimation is reached.

Thanks to this increasing precision of the chosen serial arithmetic, the global computation time of our implementation is optimized:

- first iterations are faster: since they handle reduced precisions with serial operators, the first iterations are faster than with an immediate maximum precision,
- next updates take advantage of a "good" starting point: when the system has converged for a given precision, the next iterations use an increased precision, and the additional convergence time only corresponds to computing the additional bits of the harmonic function estimation,
- the convergence of each block is reached sooner: *I* consecutive iterations are performed for each block before handling the next block, except if the computations within this block converge before the *I* iterations, which may happen more rapidly with reduced precisions,
- the serial detection of the convergence of each block is also faster: convergence detection is based on a comparison between the old and new node states, and the comparison time is proportional to the handled precision; moreover, when many blocks have already converged, this earlier convergence detection allows to switch more rapidly to the blocks that still need updates.

4. Optimizing block-synchronous harmonic control

The implementation choices discussed in the previous section lead to a block-parallel algorithm that may be described as follows:

```
p=initial_precision
while (p<=required_precision)  /* i.e. local maxima exist */
    converge:=false
    while (not(converge)) do
        converge:=true
        for b:=1 to B do
            i:=0
            block_cvg=false
            while ((i<I) and (not(block_cvg))) do
                block_cvg:=true
                forall nodes (x,y) in block b do
                    update node(x,y)
                    if (node(x,y) changed)
                        block_cvg:=false
                    endif
                endforall
                i++
            endwhile
            if (not(block_cvg))
                converge:=false
            endif
        endfor
    endwhile
    increase(p)
endwhile
```

In this section, we study the computation time of this algorithm, so as to optimize *I* and the increasing scheme of the precision. First of all, we define more precisely the convergence criteria that are used for the different loops of our algorithm. Then we analyze the convergence rate of the estimation method of the harmonic function as discussed in **2.2.1**. We show this computation is contracting with a contraction coefficient that is strictly less than 1. We then take into account the error that corresponds to the computation approximations (considering reduced precisions) by analyzing it as a noisy iterative contraction. We finally introduce in this theoretical analysis the block-synchronous approach and the use of increasing precisions through the iterative process.

4.1 What are the convergence criteria ?

In the above algorithm, the main loop stops when the required precision is reached. Though this notion of required precision is studied in **3.2.2** with respect to a static analysis of the harmonic function along a trajectory, the iterated computation of equation (2) requires a dynamic evaluation of the convergence of the whole process. Since we use a block-synchronous version with an increasing precision, different kinds of convergence are used:

- The inner loop that handles the computations for one block uses the stabilization of the node states within the block as convergence criterion.
- Similarly, the stabilization of all blocks is used as convergence criterion in the intermediate loop that performs the computations of all blocks for a given precision.
- The main loop may not know in advance what is the required final precision. The goal of the all process is to provide valid trajectories. This is the case when *no local minimum or maximum exists* in Ω. Therefore we stop the algorithm when no such local extremum is detected.

4.2 How much contracting is the computation of a harmonic function ?

When computing a harmonic function on Ω (inner space) $\cup \partial\Omega$ (boundary), the system we need to solve may be written in matrix form:

$$H = \mathbf{H}H + C$$

where *C* is a vector that has non-zero values only for points in $\partial\Omega$, and where $\mathbf{H}$ is a matrix that has zero lines for points in $\partial\Omega$ and that only has non-zero values (1/4) on 4 columns on the lines corresponding to Ω points.

Matrix $\mathbf{H}$ is substochastic (coefficients are non-negative and on each row their sum is lower than 1). It corresponds to a (vanishing) random walk on Ω: when a point is inside the domain (in Ω), it moves uniformly towards its 4 neighbours; when a point is on the border (in $\partial\Omega$), it vanishes (its mass disappears at the next instant).

Since $\mathbf{H}$ is substochastic, its eigenvalues take values in [0,1]. We now show that 1 cannot be an eigenvalue. To do so, consider a similar substochastic matrix $\mathbf{H}'$, which corresponds to the Markov chain which only vanishes on the borders of the *N*x*M* rectangular domain. A standard result (see (Young, 1971)) is that the biggest eigenvalue of $\mathbf{H}'$ is

$$\gamma = \frac{1}{2}\left(\cos\left(\frac{\pi}{N}\right) + \cos\left(\frac{\pi}{M}\right)\right)$$

Intuitively, the biggest eigenvalue (and more precisely, its distance to 1) corresponds to the amount by which this Markov chain is vanishing at each time step. As the former Markov chain $\mathbf{H}$ vanishes more than $\mathbf{H}'$, its eigenvalues should not be greater than γ. More formally, $\mathbf{H}$ is equal to $\mathbf{H}'$ on some lines and null on the others, thus for any vector d,

$$\left|\mathbf{H}d\right| \le \mathbf{H}\left|d\right| \le \mathbf{H}'\left|d\right| \le \gamma\left|d\right|$$

where $|x|$ is the componentwise absolute value of x, and where the first inequality comes from the fact that $\mathbf{H}$ only has non-negative components (it is some sort of generalized triangle inequality). This means that the eigenvalues of $\mathbf{H}$ cannot be greater than γ. As a consequence, the iterative algorithm

$$H_{k+1} \leftarrow \mathbf{H}H_k + C$$

is converging at a linear rate $\le \gamma$. When N and M are big, we have:

$$\gamma \approx 1 - \frac{\pi^2}{4}\left(\frac{1}{N^2} + \frac{1}{M^2}\right) \tag{3}$$

4.3 General analysis of a noisy iterative contraction fixed point computation

Let us consider a contraction operator $\mathbf{O}$, with contraction factor at most γ. Let us assume that there is an error bounded by e at each iteration:

$$H_{k+1} \leftarrow \mathbf{AO}H_k$$

where $\mathbf{A}$ is some approximation operator satisfying:

$$\left\|\mathbf{A}f - f\right\| \le e$$

In our case, $\mathbf{A}$ is related to the round-off error.

4.3.1 Convergence: general case

Unfortunately, the process of mixing a contraction mapping and a round-off does not stabilize in general. For example, if the error is the round-off to the closest integer, let us assume that we have the following contraction (with fixed point 0):

$$x_{k+1} \leftarrow -0.6x_k$$

Then if $x_0 = 1$, we have the following sequence:

$$x_1 = -0.6 \quad \overline{x_1} = -1 \quad x_2 = 0.6 \quad \overline{x_2} = 1$$

in other words we have a loop.

One might think that the problem in the above example is due to the fact that the factor in front of x is negative. But one can also find multidimensional examples where convergence does not occur and which involves only positive terms.

4.3.2 Weak convergence, rate of convergence

Though not converging in general, it can be proved that the above system converges in some weak sense. Namely it leads to oscillations around the fixed point. At each iteration, let us assume:

$$\left\| H_{k+1} - \mathbf{O}H_k \right\| = \left\| \mathbf{AO}H_k - \mathbf{O}H_k \right\| \le e$$

Then, writing H_* the fixed point of $\mathbf{O}$, we have

$$\begin{aligned} \left\| H_{k+1} - H_* \right\| &\le \left\| H_{k+1} - \mathbf{O}H_k \right\| + \left\| \mathbf{O}H_k - H_* \right\| && \text{(triangular inequality)} \\ &= \left\| H_{k+1} - \mathbf{O}H_k \right\| + \left\| \mathbf{O}H_k - \mathbf{O}H_* \right\| && (H_* \text{ fixed point of } \mathbf{O}) \\ &\le e + \gamma \left\| H_k - H_* \right\| && \text{(definition of contraction)} \end{aligned}$$

Thus by induction,

$$\left\| H_k - H_* \right\| \le e + \gamma e + \ldots + \gamma^{k-1} e + \gamma^k \left\| H_0 - H_* \right\| \le \frac{e}{1-\gamma} + \gamma^k \left\| H_0 - H_* \right\|$$

It shows that even if the process does not converge, it is asymptotically $\frac{e}{1-\gamma}$-close to H_*.

We may exploit the above equation to derive some rate of convergence:

$$\left\| H_k - H_* \right\| \le \varepsilon \Leftarrow \frac{e}{1-\gamma} + \gamma^k \left\| H_0 - H_* \right\| \le \varepsilon \Leftrightarrow k \ge \frac{\log\left(\dfrac{\varepsilon - \dfrac{e}{1-\gamma}}{\left\| H_0 - H_* \right\|} \right)}{\log \gamma}$$

For instance, for any $\lambda > 0$, taking $\varepsilon = \frac{(1+\lambda)e}{1-\gamma}$, we see that:

$$k \geq \frac{\log\left(\frac{\lambda e}{(1-\gamma)\|H_0 - H_*\|}\right)}{\log \gamma} \Rightarrow \|H_k - H_*\| \leq \frac{(1+\lambda)e}{1-\gamma} \tag{4}$$

We may interprete λ as a margin that is added to the $\frac{e}{1-\gamma}$-wide asymptotical interval around H_*. This margin defines a wider interval where H_k finally lies.

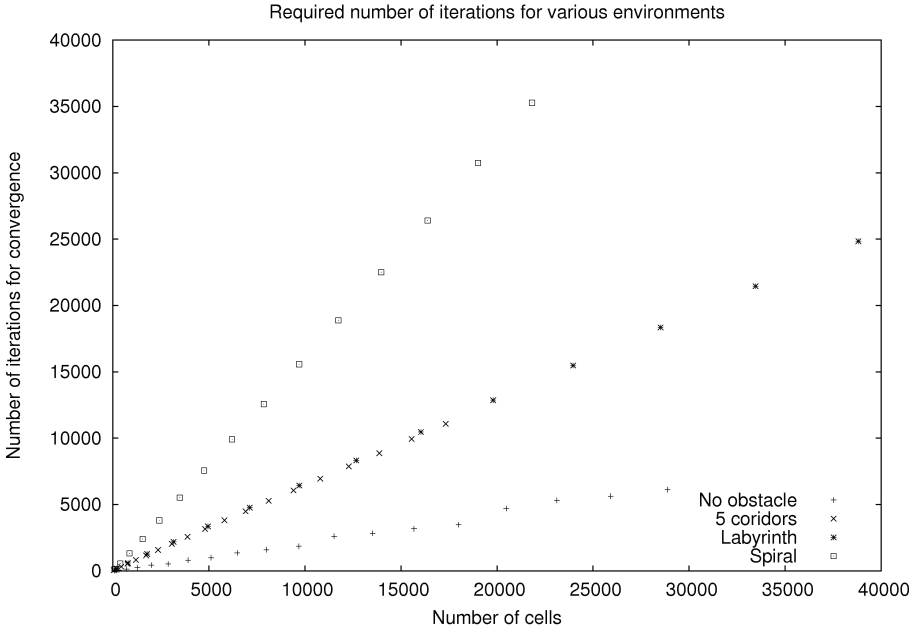

Fig. 4. Number of iterations required for full convergence (stabilization).

4.3.3 Experimental convergence, rate of convergence

Experimentally, the convergence of the iterated computation of the harmonic function (as in equation (2)) not only converges in a weak sense, but fully converges whatever the fixed precision. To validate this assertion, we have carried out numerous experiments with various environments. These experiments are illustrated by figure 4. They establish that the number of required iterations linearly depends on the number of nodes in the environment, which validates the above estimation of k (that depends on $1/\log(\gamma)$ which is roughly proportional to N^2 and M^2).

Implementation note

Following these experiments, we consider the complete stabilization of all bits of all nodes to define the convergence criterion within the inner loops of our algorithm (see **4.1**).

4.4 An increasing precision algorithm

We now consider the case where the harmonic function is iteratively computed using arithmetics with different precisions: $e_1 = 2^{-p_1}, e_2 = 2^{-p_2}, \ldots, e_T = 2^{-p_T}$. Write $H^{(0)}$ the initial estimate. The analysis of the previous subsection suggests that we can estimate the sequence of potentials $H^{(1)}, H^{(2)}, \ldots, H^{(i)}, \ldots$ with respective precisions

$$\frac{(1+\lambda)e_i}{1-\gamma} \approx (1+\lambda)K.2^{-p_i}$$

with the constant

$$K = \frac{1}{1-\gamma} \approx \frac{4}{\pi^2\left(\frac{1}{N^2}+\frac{1}{M^2}\right)} \tag{5}$$

for some positive value λ.

4.4.1 Initial and final precisions

At the end, one will have: $\left\|H^{(T)} - H_*\right\| \leq (1+\lambda)K.2^{-p_T}$

The final precision p_T can be set such that the eventual approximation is smaller than some given $\varepsilon > 0$:

$$(1+\lambda)K.2^{-p_T} \leq \varepsilon \Leftarrow p_T \geq \frac{\log(1/\varepsilon) + \log(K) + \log(1+\lambda)}{\log(2)} = p_{\max}(\varepsilon)$$

The initial precision p_1 should be at least such that $\varepsilon < 1$ (the harmonic function is between 0 and 1), so this means that we should take:

$$p_1 \geq \frac{\log(K) + \log(1+\lambda)}{\log(2)} = p_{\min}$$

4.4.2 Worst-case analysis

Given the analysis of the previous subsection, estimating $H^{(i)}$ from $H^{(i-1)}$ with precision $\frac{(1+\lambda)e_i}{1-\gamma}$ will be done in less than k_i iterations with:

$$k_i = \frac{\log\left(\frac{\lambda e_i}{(1-\gamma)\left\|H^{(i-1)} - H_*\right\|}\right)}{\log\gamma} = \frac{\log\left(\frac{\lambda K.2^{-p_i}}{\left\|H^{(i-1)} - H_*\right\|}\right)}{\log\gamma}$$

Since $\log(\gamma) \approx \gamma - 1 = -1/K$, we finally get:

$$k_i \approx K\left(p_i \log(2) + \log(1/\lambda) + \log(1/K) + \log\left(\left\|H^{(i-1)} - H_*\right\|\right)\right)$$

Remark: log(1/K) tends to $-\infty$ when the environment gets bigger, but it is compensated by the fact that $p_i > \log(K)/\log(2)$ (see previous subsection).

The first number of iterations k_1 depends on the initial value $H^{(0)}$:

$$k_1 \approx K\left(p_1 \log(2) + \log(1/\lambda) + \log(1/K) + \log\left(\left\|H^{(0)} - H_*\right\|\right)\right)$$

The subsequent numbers of iterations depend on the previous precision: for $i \geq 2$

$$\begin{aligned} k_i &\approx K\left(p_i \log(2) + \log(1/\lambda) + \log(1/K) + \log\left(\left\|H^{(i-1)} - H_*\right\|\right)\right) \\ &\leq K\left(p_i \log(2) + \log(1/\lambda) + \log(1/K) + \log(e_{i-1})\right) \\ &= K\left((p_i - p_{i-1} + 1)\log(2) + \log(1/\lambda)\right) \end{aligned}$$

To summarize, k_1 may be estimated as an affine function of the initial number of bits p_1, and each k_i may be estimated as an affine function of the precision increase (in terms of number of added bits). Both affine functions share the same linear coefficient *K*.log(2).

$$k_1 = a + bp_1 \qquad\qquad k_i = a' + b(p_i - p_{i-1})$$

Let us now assume that the computation time of equation (2) is also an affine function of the number of bits $\alpha + \beta p$, when very large precisions are used. It is for example the case on a standard processor for which vectors of integers code for very large precision numbers, and it is of course the case with hardware serial operators. Then the global computation time when the final precision is used from the beginning is:

$$(a + bp_{\max}).(\alpha + \beta p_{\max}) = b\beta p_{\max}^2 + o(p_{\max}^2)$$

Whereas the global computation time with increasing precision is

$$(a+bp_1).(\alpha+\beta p_1)+\sum_{i=2}^{T}(a'+b(p_i-p_{i-1})).(\alpha+\beta p_i) \quad (6)$$

If the increase of the precision is such that the number of bits is an arithmetic series $p_i = i\frac{p_{\max}}{T}$, then equation (6) becomes

$$(a+b\frac{p_{\max}}{T}).(\alpha+\beta\frac{p_{\max}}{T})+\sum_{i=2}^{T}(a'+b\frac{p_{\max}}{T}).(\alpha+\beta i\frac{p_{\max}}{T})$$

$$=p_{\max}^2\left(\frac{b\beta}{T^2}\left(1+\sum_{i=2}^{T}i\right)+o(1)\right)=\frac{1}{2}b\beta p_{\max}^2\left(1+\frac{1}{T}\right)+o(p_{\max}^2)$$

This shows that the increasing precision approach roughly divides the computation time by 2 when very large precisions are required (therefore when large environments are handled, as discussed in **3.2.2**).

As such, this result holds without block-synchronous. The next subsection introduces the block-synchronous aspect to prove the relevance of the increasing precision approach within our main algorithm.

4.5 A block-synchronous algorithm with increasing precision

Let us now take into account the fact that an environment may be too large to fit within one single FPGA. Then one has to partition this environment into *B* blocks, as explained before. Since our FPGA implementation uses serial arithmetics with interleaved loops, its computation time for *k* iterations at precision $e = 2^{-p}$ is

$$\tau\ k\ (p+d)$$

where *d* is the serial delay for the computation of equation (2), and τ is its latency.

We can conclude that the overall time without increasing precision is:

$$B\tau((a+bp_{\max})(p_{\max}+d))$$

4.5.1 Introducing an increasing precision

Following **4.4.2**, we now assume that our implementation fully applies the algorithm depicted at the beginning of section **4**, with *I* consecutive iterations for each block to ensure

pipelining. We first consider that these consecutive iterations do not modify the total number of iterations to perform for each precision (see below for an empirical study of this assertion). Assuming again that the number of bits is an arithmetic series, it follows that the overall computation time now is:

$$B\,\tau\,\frac{(a+b\frac{p_{\max}}{T})}{I}.\left(I(\frac{p_{\max}+d}{T})\right)$$

$$+B\,\tau\sum_{i=2}^{T}\frac{(a'+b\frac{p_{\max}}{T})}{I}.\left(I(i\frac{p_{\max}+d}{T})\right) \quad (7)$$

As in **4.4.2**, this equation leads to roughly divide the computation time by 2 thanks to the increasing precision. Yet it does not appear as obvious that the *I* consecutive iterations provide a significant improvement (in the above equation, it just appears as a way to minimize the effect of the delay). But the above estimation considers that all *I* iterations are performed for all blocks in the algorithm. This is highly pessimistic, as shown below.

4.5.2 Taking advantage of "still" blocks

The above notion of pipelined interleaved loops within each block corresponds to the most inner loop of our algorithm:

```
while ((i<I) and (not(block_cvg))) do
```

Therefore, *I* iterations are performed only for blocks that have not converged so far. In most experiments, it clearly appears that large parts of the environment stay unchanged (still) for several iterations while distant blocks slowly propagate the changes. This is even more true when small precisions are handled.

We have performed experiments on a PC in order to compare the overall number of computations when one uses several blocks with detection of early stabilization within each block, to the case where there is only one block. These experiments are reported in figure 5 for the "Labyrinth" environment. This figure illustrates the speed-up obtained with different block sizes over the purely synchronous case. The curves show the ratio between the number of iterations needed to converge in the asynchronous case and the number of iterations needed to converge in the block-synchronous case with respect to the maximum number of iterations by block *I*. The experiments show that partitioning by blocks speeds up the computation time, although the successive iterations are performed by a block without updating the neighbouring blocks. This speedup is observed provided that *I* is not too large and an early detection of block stabilization is performed. Similar results have been obtained for all kinds of environments.

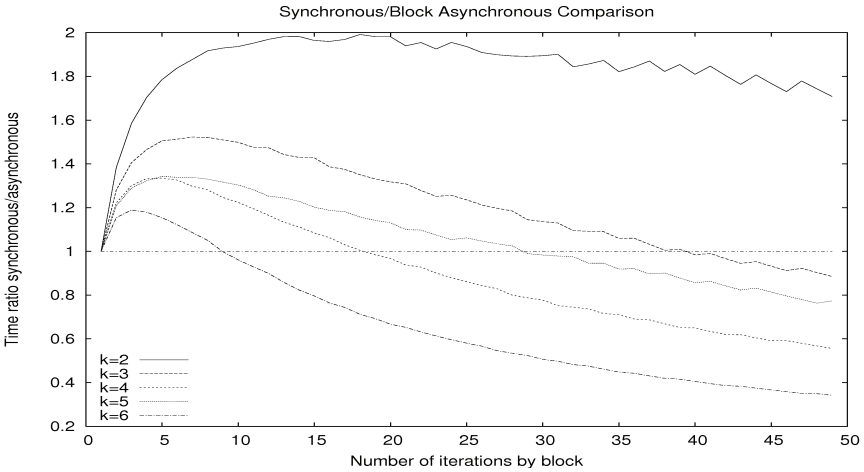

Fig. 5. Comparison between asynchronous and block-synchronous overall computation time. Environment: labyrinth. The total number of blocks is indicated by 'k': for example when 'k=4', there are B=4x4 blocks.

Figure 5 also shows that the speed-up appears greater when the block size is larger. This is confirmed by another series of experiments that are illustrated by figure 6. This figure shows the evolution of the ratio $\hat{I}/I$ until convergence for different values of B on the same environment (labyrinth), where $\hat{I}$ is the average number of iterations performed by the blocks before early detection of stabilization. The experiments show that the smaller the blocks, for example 6x6 (k=6), the larger the number of iterations required. They also show that $\hat{I}$ rapidly reaches a constant value until convergence.

Following our experiments, it finally appears that the block-synchronous approach that uses an increasing precision divides the computation time by some coefficient between 2 and 4.

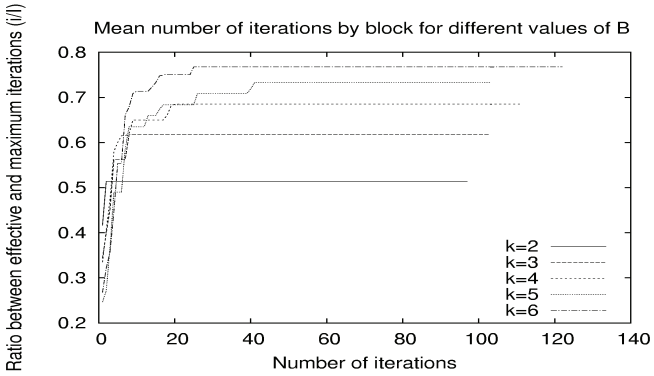

Fig. 6. Evolution of the number of successive iterations performed by the blocks before early detection of stabilization.

5. Hardware implementation of harmonic control

Though harmonic control has been widely used in robotics, few hardware implementations have been proposed. Their technological choices are mostly motivated by the fact that analog resistive grids may easily compute the harmonic function as in equation (2). For example in (Stan et al., 1994) an analog implementation of a 16×16 grid is proposed. The main limitation of this work is the precision (as for most analog implementations). To our knowledge, digital FPGA-based implementations have not yet been proposed.

In this work, the discretized computation of the harmonic function is performed to make navigation decisions for a robot in an environment that may be explored on-line. Navigation commands are obtained by means of linear interpolations of the values of the neighbouring nodes (north, east, west and south), so as to obtain the best navigation orientation in $[0,2\pi[$. It results in global trajectories towards the goal position from any starting point.

5.1 General architecture

The fine grain pipelined internal structure of the proposed digital architecture was implemented using fixed point arithmetic. Since node values range from 0 to 1, an unsigned representation with p+1 bits is chosen, with a p-bit fractional part. As mentioned above, large wordlengths may easily be handled.

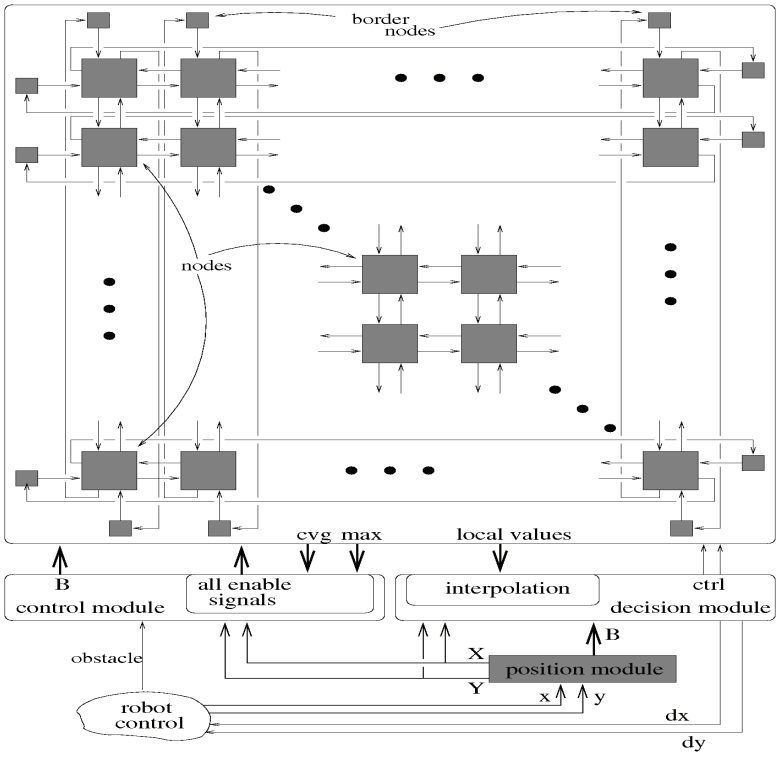

Fig. 7. General architecture

Figure 7 illustrates the general architecture of the implementation of harmonic control in a given environment. Since the environment is split into several blocks, this architecture mostly consists of a grid of $n \times m$ identical node modules (gathered 18 by 18 to handle on-chip data storage and access) surrounded by border node modules, a control module, a decision module, and a module to interact with the robot.

The role of each component is the following:

- Each node module computes its corresponding node value within the currently handled block. All node modules use interleaved loops that together perform all required computations within simultaneous local recurrent pipeline schemes. This kind of parallelism is particularly efficient for serial implementations of recurrent iterated computations within massively distributed models (Girau & Torres-Huitzil, 2007). The control of these computations are synchronized in the whole block so that node modules may serially communicate their values to their neighbours. In order to simplify the block-diagram of figure 7, only few buses and wires are shown: the signals that carry the node values from and to any neighbouring node (and possibly to opposite border nodes).
- The node modules are split in groups of 3x6 nodes that share common storage resources: a single dual port SRAM block stores the values of the 18 nodes, and its R/W accesses are controlled by a single set of counters (see **5.2**).
- The border nodes are simpler than the node modules. They only store the values of the immediate neighbours of the most outer nodes within each block, and they serially generate these values when required. The only difficulty is to handle the addressing scheme so that the values stored within each of the 4 possible borders are updated when the block that contains them is being computed. This update requires the long-range connections from the node modules on each side of the block to the opposite border nodes. Moreover, when the borders lie outside the whole set of blocks, the border nodes simply generate the constant value 0 that denotes obstacles.
- The interaction with the robot has not yet been implemented. It strongly depends on the exact configuration of the robot and of the FPGA board. It includes a position modules, which role is mainly to compute the coordinates `(B,X,Y)` of the closest grid point (block, node) around the real coordinates `(x,y)` of the robot in its environment.
- The control module generates the enable signals that are sent to all node modules to control their individual behaviour when an asynchronous event occurs:
    - convergence of the computation of the harmonic function (when all blocks have confirmed that no local extremum has been detected),
    - early detection of the convergence of the computation within the currently handled block (it depends on the local convergence signals computed by all nodes within the block, see **5.2**),
    - detection of an unknown obstacle (at node `(B,X,Y)`).
- The decision modules forwards the current position coordinates `(B,X,Y)` of the robot to the nodes. It collects the navigation information that are provided by the node modules in return (local values of all nodes in block `B`). Then it performs the

linear interpolation (see **5.3**) to compute the navigation direction that must be given to the robot.

In the following subsections, the hardware architecture for the node module and its main components will be described in some detail.

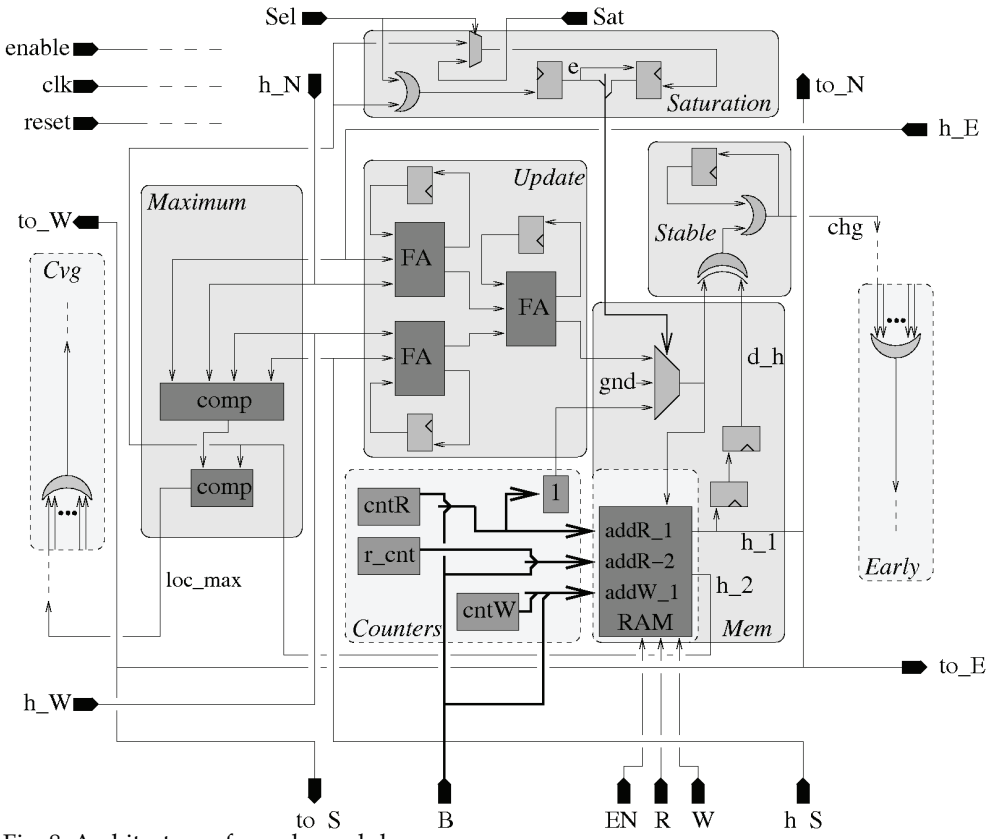

Fig. 8. Architecture of a node module

5.2 Node module implementation

The architecture for a node module is shown in the simplified block diagram on figure 8. It uses 1-bit inputs and outputs to exchange data among nodes and with the global modules. Inputs are mainly used to receive the neighbouring node values (signals `h_N`, `h_E`, `h_W`, `h_S`) and global control signals (standard signals `clk`, `reset`, `enable`, signals `Sel` and `Sat` to indicate obstacle/goal changes, and SRAM controls `EN`, `R`, `W`). The local value `h` of the harmonic function is sent to all 4 neighbours (signals `to_N`, `to_E`, `to_W`, `to_S`) as well as to the global interpolation module so as to compute the navigation orientation if the robot is found to be located in the area that corresponds to the local node (see **5.3** for the description of the interpolation process).

The proposed hardware node module is constituted by five main sub-modules: the iterative computation of the harmonic function is performed by the `Update` module, the local convergence of this computation (stabilization) is detected by the `Stable` module, the node receives orders to behave as an obstacle or a goal through the `Saturation` module, the `Maximum` module checks for the presence of a local maximum, and communication with the dual port SRAM block that stores the node value is controled by the `Mem` module. Figure 8 shows this architecture, as well as its interaction with shared resources that are surrounded by dotted lines (the local `Counters` and `RAM` modules are shared by a group/cluster of 3x6 node modules, the `Early` module that detects the early stabilization of the block is part of the global control module, and the navigation decision is performed when all blocks have indicated no more local maximum through the `Cvg` module). The functionality of the main modules and their implementation details are described below.

Update: This module performs the iterated computation of the harmonic function value $h_{(i,j)}$ where (*i*,*j*) are the coordinates of the node in the environment. As described in (2), each iteration computes:

$$h_{(i,j)}(t+1) = \frac{h_{(i-1,j)}(t) + h_{(i+1,j)}(t) + h_{(i,j-1)}(t) + h_{(i,j+1)}(t)}{4}$$

Three standard Full-Adder cells compute this average, without any shift or division operator, since the output value is sent to the RAM with a write address that is delayed by 2 clock cycles (division by 4). Additional flip-flops are required to store the carry values. Since there is no more carry equal to 1 when writing the last bit of the result in the RAM, no additional reset clock cycle is required.

Stable: This module serially compares the output of the iterated computation to the stored value (delayed by two flip-flops in the `Mem` module). This local convergence test is then sent to a global OR gate (in the `Early` module) to disable the inner computation loop of the block when early stabilization has been detected before *I* iterations.

Maximum: This module uses a comparison between all neighbouring values and a comparison with the local value so as to determine whether the local node corresponds to a local maximum. This local information is sent to the `Cvg` module that uses a global `OR` gate so as to check for the presence of any local maximum in the current block. This global information is handled by the general `Control` module to finish the whole computation of harmonic values, so that the determination of the trajectory may start.

Counters: This module is shared by 18 node modules. It generates the read (resp. write) addresses for the dual port RAM by means of counters `cntR`, `r_cnt` (resp. `cntW`). Both read addresses are sent to the ports of the RAM to handle both LSBF (counter `cntR`) and MSBF (reverse counter `r_cnt`) modes. When handling *d*-bit precision data, these counters are reset each *d*+2 cycles (the RAM is written with a 2-clock cycles delay). Value 1 (for obstacles) is computed as a logical function of `cntR`.

Mem and Saturation: The local node value is not directly the output of the `Update` module. It may also be a constant 0 or 1 (goal or obstacle). A multiplexer selects the correct value with respect to a control given by the `Saturation` module that memorizes the `Sat` value to be the constant value of the grid point when the node is selected by the global control module (signal `Sel`). Since multiple blocks are handled, these constant values must also be stored in and retrieved from the RAM. To do that, we add a special bit (the MSB) to the values stored in memory (this bit is set to 1 when the local value is constant). The `Saturation` module also receives this information.

5.3 Convergence detection and determination of the navigation direction

Control: This global module performs the usual scheduling of the loops of the algorithm through various counters. It mostly computes the number `B` of the current block, and it takes into account the stabilization and the convergence (no maximum) of the different blocks to adapt the global scheduling. Moreover, it handles the different counters such that the computations are performed with an increasing precision until global convergence.

Decision: This module operates after convergence of the iterations. Knowing the coordinates of the node `(B,X,Y)` that corresponds to the current position of the robot, this module acts as a sequential program that computes the maximum slope among the four triangles that are defined by the node and two of its immediate neighbours. It simply reduces to the determination of the two consecutive neighbours (B,X′,Y′) and (B,X″,Y″) which values maximize the sum of their difference with respect to the center node value $\left(h_{(B,X,Y)} - h_{(B,X',Y')}\right)^2 + \left(h_{(B,X,Y)} - h_{(B,X'',Y'')}\right)^2$. Then the best trajectory direction (gradient ascent) is given by a vector which coordinates are equal to $\left(h_{(B,X,Y)} - h_{(B,X',Y')}\right)$ and $\left(h_{(B,X,Y)} - h_{(B,X'',Y'')}\right)$ or to their opposite values (it depends on the position of `(X',Y')` and `(X″,Y″)` with respect to `(X,Y)`).

5.4 Implementation results

In order to deal with larger environments, elemental node modules are gathered together to form a 2D grid of 3x6 clusters. Most of the connections among nodes are local with the four neighbours. The main reason to group in such a configuration is due to the 18-bit width of the shared block ram (technological constraints of the targeted FPGA) that is used to store the values of each node. The depth of the BlockRAMs is 1K. It allows handling a wide range of arithmetic precisions such as 64-128 bits per word without modifying the memory organization.

Node hardware resource utilization XC2V6000-5ff1517	
Number of Slice Flip Flops	11/67,584
Number of 4 input LUTs	22/67584
Number of occupied Slices	13/33792
Frequency	361 MHz

Table 1. Synthesis results for a node module

This work uses two PCI bus board. The first one is equiped with a Virtex XC2V6000-4FF1517 FPGA from Xilinx, with up to 6,000,000 system gates. Such a FPGA contains 67,584 logic cells. The second one is equiped with three FPGAs, the largest one being a Virtex-4 XC4VLX160ff1513-12 FPGA from Xilinx, that contains 135,168 logic cells, to be compared with the 200,448 ones of the current largest Virtex-4. The design was synthesized, placed and routed automatically in Xilinx Foundation ISE 7.1i. Results are shown in tables 1, 2 and 3. Each node module requires 26 logic cells, and each cluster of 18 node modules requires 470 logic cells (counters included). On a XC2V6000, 144 dual port SRAM blocks are available, that may be configured as 1Kx18 RAMs to be shared by clusters of 18 node modules. The whole architecture may implement a 36x66 grid on less than 92 % of the XC2V6000 logic cells.

3x6 cluster hardware resource utilization XC2V6000-5ff1517	
Number of Slice Flip Flops	198/67,584
Number of 4 input LUTs	398/67584
Number of occupied Slices	235/33792
Frequency	300 MHz

Table 2. Synthesis results for a 18-node cluster module

We use the remaining 12 SRAM blocks to implement the storage facilities of the 204 border blocks (for a 36x66 grid). Handling addresses for these special blocks mostly reduces to generate in a parallel way `B+1`, `B-1`, `B+k`, `B-k` (all values modulo k×k, where k×k is the total number of blocks). A few slices are sufficient (around 10 for up to 5x5 blocks). Then the `Control` module (844 slices) and the `Decision` module are added ($3p$ + 682 slices for a p-bit precision). So that for example 99.4 % of this FPGA is finally used for the implementation of the whole algorithm with 4 blocks and p=255 (see below for the speed).

Larger blocks (up to 54x108 nodes) may be implemented on the current largest FPGAs with this approach (the available SRAM blocks being the critical resource). As an example, table 3 shows the hardware resource utilization for a 48x96 block on the Virtex-4.

48x96 block hardware resource utilization XC4VLX160ffff1513-12	
Number of Slice Flip Flops	50699/135168
Number of 4 input LUTs	101396/135168
Number of occupied Slices	59914/67584
Frequency	150 MHz

Table 3. Synthesis results for a 48x96 block

Software implementations of the harmonic function computation on a microprocessor based computer, Pentium 4,2 GHz, require around 100 μs per iteration with a 36×66 block. In the proposed hardware implementation, p+2 clock cycles are required per iteration for precision p, with an estimated clock frequency of 150 MHz. Thus, the implementation on the Virtex-2 provides a speed factor up to 100x (for a 128-bit precision that corresponds to some average-sized environments in our reported experiments), that would even increase with the number of nodes in the grid (sequential vs parallel implementation). But the

implementation speed is not the only advantage of our implementation. Power consumption is a key factor for embedded implementations, and above all large precisions may be handled by the proposed serial implementation when few blocks are used (up to 1K bits when only one block is used).

When using multiple blocks (for large environments) together with an increasing precision, the computation time linearly increases with the number of blocks as in the sequential implementation on PC. Therefore, the speedup is not intrinsically changed. But following our experiments in section **4**, the number of iterations decreases before convergence, so that a final speed factor is up to 400x. As an example, with a not too complex maze with 9500 nodes divided into 4 blocks, a p=255 precision, and I=6 consecutive iterations for each block at most, a speedup of 270x is obtained.

It might be noticed that the depth of the SRAM memory blocks appears as the main limitation to handle larger environments (more blocks). Considering that the most recent Virtex-5 FPGAs contain different SRAM blocks, some of them twice larger than the ones we use, this limitation should also evolve with the technological improvements of FPGAs.

6. Conclusion

This chapter presents an embedded architecture to solve the navigation problem in robotics, that computes trajectories along a harmonic potential, using a FPGA implementation. This architecture includes the iterated estimation of the harmonic functions. The goals and obstacles of the navigation problem may be changed during computation. The trajectory decision is also performed on-chip, by means of local computations of the preferred direction at each point of the discretized environment. The proposed architecture uses a massively distributed grid of identical nodes that interact with each other within mutually dependant serial streams of data to perform pipelined iterative updates of the local harmonic function values until global convergence.

When the environment size is too large for a fully parallel implementation on the used FPGA, our implementation takes advantage of the available SRAM to handle larger environments that are partitioned into blocks. It results in an iterated computation mode that is both globally asynchronous and block-synchronous.

The proposed architecture also introduces the use of an increasing precision. First of all, this approach enables our implementation to reach the required precision for convergence without having to over-estimate it initially (resulting in excessively long computations). Then it also enables an optimization of the overall computation time. This optimization is carefully studied from a theoretical and experimental point of view, with respect to both the block-synchronous approach and the increasing precision technique.

Despite all these results, our implementation may still appear as not able to handle particularly large and complex environments. This is intrinsically linked to the nature of the harmonic control that rapidly requires huge precisions for such environments. The main perspective of this work is to extend it to optimal control, a more generic (and tunable)

trajectory planning method, that uses similar computations without requiring such huge precisions.

References

Alvarez, D.; Alvarez, J.C. and Gonzalez, R.C. (2003). Online motion planning using Laplace potential fields, *Proceedings of the Int. Conf. Robotics and Automation*, IEEE CNF.

Boumaza, A. and Louchet, J. (2003). Mobile robot sensor fusion using flies, In: *Applications of Evolutionary Computing*, volume 2611 of LNCS, pages 357–367.

Connolly, C.I.; Burns, J.B. and Weiss, R. (1990). Path planning using laplace's equation. *Proceedings of the Int. Conf. on Robotics and Automation*, pages 2102–2106, IEEE CNF.

Connolly, J.C. and Grupen, R. (1993). On the applications of harmonic functions to robotics. *Journal of Robotic and Systems*, 10(7):931–946.

Feder, H.J.S. and Slotine, J.J.E. (1997). Real-time path planning using harmonic potentials in dynamic environments, *Proc. of the Int. Conf. Robotics and Automation*, IEEE CNF.

Girau, B. and Boumaza, A. (2007). Embedded harmonic control for dynamic trajectory planning on FPGA, *Proceedings of Int. Conf. on Artificial Intelligence and Applications.*

Girau, B. and Torres-Huitzil, C. (2007). Massively distributed digital implementation of an integrate-and-fire LEGION network for visual scene segmentation. *Neurocomputing*, 70.

Huber, M.; MacDonald, W.S. and Grupen, R.A. (1996). A control basis for multilegged walking. *Proceedings of the IEEE Int. Conf. Robotics and Automation*, IEEE CNF.

Kazemi, M.; Mehrandezh, M. and Gupta, K. (2005). An incremental harmonic function-based probabilistic roadmap approach to robot path planning, *Proceedings of the IEEE Int. Conf. Robotics and Automation*, IEEE CNF.

Khatib, O. (1986). Real-time obstacle avoidance for manipulators and mobile robots. *International Journal of Robotic Research*, 5(1):90–98.

Masoud, A.A. and Masoud, S.A. (2002). Motion planning in the presence of directional and obstacle avoidance constraints using nonlinear anisotropic, harmonic potentialfields: A physical metaphor. *IEEE Transactions on Systems, Man, & Cybernetics, Part A: systems and humans*, 32(6):705–723.

Stan, M.; Burleson, W.; Connolly, C. and Grupen, R. (1994). Analog vlsi for robot path planning. *Journal of VLSI Signal Processing*, 8(1):61–73.

Sweeney, J.D.; Li, H.; Grupen, R.A. and Ramamritham, K. (2003). Scalability and schedulability in large, coordinated, distributed robot systems, *Proceedings ofthe IEEE Int. Conf. Robotics and Automation*, IEEE CNF.

Tarassenko, L. and Blake, A. (1991). Analogue computation of collision-free paths,*Proceedings of the Int. Conf. on Robotics and Automation*, pages 540–545, IEEE CNF.

Wang, Y. and Chirikjian, G.S. (2000). A new potential field method for robot path planning, *Proceedings of the Int. Conf. Robotics and Automation,* vol. 2, pages 977–982, IEEE CNF.

Xilinx, editor (2000). *The Programmable Logic Data Book*. Xilinx, Inc.

Young, D. (1971). *Iterative solutions of large linear system,* Academic Press, New York.

Zelek, J.S. (1998). Complete real-time path planning during sensor-based discovery, *Proceedings of the IEEE/RSJ Int. Conf. on Intelligent Robots and systems.*