An Engineers Primer on Field Computation

ADVANCED SEMINAR submitted by

Igor Krawczuk

NEUROSCIENTIFIC SYSTEM THEORY
Technische Universität München

Prof. Dr Jörg Conradt

Supervisor: Dipl.-Inf. Nicolai Waniek
Final Submission: January 14, 2015

In your final hardback copy, replace this page with the signed exercise sheet.

Abstract

The purpose if this thesis is to give the interested reader a starting point into the topic of Field Computation. Field Computation is a term coined by Bruce J. MacLennan in 1987 "Technology-Independent Design of Neurocomputers: The Universal Field Computer" [Mac87] presenting a framework for describing massively parallel computations. In this paper and those related[Mac03, Mac99b, Mac94, Mac97, Mac] he also often uses the universal Turing Machine as an analogy for his universal Field computer and when researching, one often finds related papers on hyper-computation.
This seminar thesis aims to provide the interested reader an overview of the principles of field computation, discuss whether it might be distinct from Turing Computation or indeed an example of hyper-computation, as well as provide context in which this framework might be useful. To ensure knowledge of the mathematical background, we also present a short primer on the required fundamentals.

Contents

Chapter 1

Introduction

1.1 Problem Statement

The current computational paradigms seem to differ wildly from the computation done in our brains. Because of the so called 100 step phenomenon[Mor, TH] a strong line of thinking exists within neuroscience that assumes the brain computes in a massively parallel, but very shallow manner, i.e. with few sequential steps. This is in stark contrast with our classical computation techniques, which use a large number of sequential steps with very little parallelizations. In recent years, there have been advances in the field of parallel processing technology, offering thousands and hundreds of thousand of cores. Since traditional algorithm design usually deals with at most dozens or hundred of parallel processors, not thousands or more, new paradigms of computation are becoming a necessity.Bruce J. MacLennan developed one candidate for this new paradigm in 1987, building a mathematical framework of dealing with massively parallel operations. However, to date his research in this field has not been picked up by many scholars. There is also no accessible summary of his research on field computation, which often touches topics unfamiliar to engineers. This thesis aims to fill this niche by summarizing 20 of his papers and giving physicists and electrical engineers who might not be familiar with computer science concepts like *Turing Computability* or *Concurrency* the context to understand them.

1.2 Literature Review And Related Work

As mentioned before, there has been little work done based on MacLennans field computability theory, however there he has been mentioned in papers on the topic of *hyper-computation* or *super-computation*, exploring the possibility of computation models more powerful than Turing Machines. For this we refer to [Sta04, Sta06]. There has also been research done on the development of parallel algorithms[HS86, BM96]. However, these approaches differ from MacLennans in that they aim to break the task into smaller sub-tasks, which can be done sequentially with only

local state knowledge whereas MacLennan develops a new mathematical formalism and new operations to deal with problems and algorithms in an inherently parallel fashion.

Chapter 2

Mathematical Review

In this chapter we will present an overview of functional analysis and computability theory, two mathematical domains required to properly discuss the utility and computability of field computation. We assume that the reader has learned concepts like calculus and analysis before, but that they are engineers and not mathematicians and thus some of the definitions and properties might not be immediately familiar .

2.1 General Definitions and Concepts

During definitions and analysis of Fields, we will have to deal with the concepts of Openness and Closedness in a topological sense. Closedness can be defined either as a fundamental property with regards to the limits of the set, or with respect to one or more operators.

Definition 1 (Open and Closed Sets)

1. *A set in a topological space is closed if and only if it contains all of its limit points. A set which is not closed is open.*

2. *A set is closed under one or more operations if the operation(s) on members of the set always produce another member of the same set.*

When dealing with transformations, we will also need a concept of how sensitive the transformation are to changes to their inputs. This is the concept of continuity, which we will mainly encounter in 3 degrees of strength.

Definition 2 (Continuities)

1. *A function is continuous if small changes in input lead to small changes in the functions output. We thus have a guarantee that there are no "jumps" in the function. Formally we write* $\lim_{x \to c} f(x) = f(c)$

2. *A function over a metric space (defined later, see* **??***) is uniformly continuous if for every real number* ϵ *there exists a* δ *such that a change of* δ *in the functions input only changes the output to* ϵ*.*

3. *A function over a metric space is Lipschitz continuous if there exists a real K for which the relation $\frac{ChangeinOutput}{ChangeinInput} \leq K$ holds*

Since we will also be defining and making use of binary operators, we desire some way to describe their behavior

Definition 3 (Linearity and Bi-linearity)

1. *An Operator L on a set of elements F is said to be linear if for every pair of $f, f_1, f_2 \in F$ and every scalar s the relations $L(f_1 + f_2) = L(f_1) + L(f_2)$ and $L(tf) = tL(f)$ hold.*

2. *A Binary Operator is said to be bi-linear, if it is linear with respect to both elements $f_1, f_2 \in F$ in every pair $L(f_1, f_2)$.*

Lastly, since we will be operating on continua, we need a formal way to evaluate the structures we analyze in fixed points. For this we make use of the Kronecker delta function and the Dirac delta function:

Definition 4 (Kronecker delta function)
The Kronecker delta function is defined as a function of to variables which evaluates to 0 if the differ and 1 if the are equal. We write:

$$\delta_{i,j} = f(i,j) = \begin{cases} 1 \text{ if } i = j \\ 0 \text{ else} \end{cases}$$

Definition 5 (Dirac delta function)
*While the Dirac delta function is not a formal function [**?**],we can heuristically define it as a function which is zero everywhere except at the origin, where its value is infinite. Simultaneously we constrain it such that its integral over the whole range equals one. We write:*

$$\delta_i = f(i) = \begin{cases} 1 \text{ if } i = 0 \\ 0 \text{ else} \end{cases}$$

$$\int_{\infty}^{\infty} \delta_i = 1$$

2.2 Functional Analysis

Field computability makes use of a selection of concepts from functional analysis. Important to note is that there is a distinction between MacLennans Fields and the mathematical concept of a field (see definition 22). In this section we will briefly present basic definitions and build up the mathematics of fields and operations on those fields. As we will see later in section 3.2, almost all of the properties are equivalent, with exceptions only in edge cases.

2.2.1 Deriving Vector spaces

General fields

In mathematics, a field describes a set of elements that fulfill the field axioms. It can also be characterized as a ring (a set of elements for which general definitions of addition and multiplication are given) that is additionally both commutative and for which there exists a multiplicative inverse for every element. The following axioms constitute the field axioms:

Definition 6 (Field Axioms)
Basic requirements: Existence of addition and multiplication, with subtraction and division defined in terms of inverse operations of these.
For a Field $\mathbb{F}$ *we also require:*

1. *Associativity of addition and multiplication,* $\forall a, b, c \in \mathbb{F}$ *the following relations hold:* $a + (b + c) = (a + b) + c$ *and* $a(b \cdot c) = (a \cdot b)c$
2. *Commutativity of addition and multiplication,* $\forall a, b \in \mathbb{F}$ *the following relations hold:* $a + b = b + ac$ *and* $a \cdot b = b \cdot ac$
3. *Distributivity of multiplication over addition,* $\forall a, b, c \in \mathbb{F}$ *the following relations hold:* $c(a + b) = ca + cb$
4. *Existence of additive and multiplicative identity elements. For addition , there exists an element* $0 \in F$ *for which* $a \in F, a + 0 = a$*. For multiplication, there exists an element* 1 *for which* $1 * a = a$ *holds. To exclude trivial examples, we require* $1 \neq 0$
5. *Existence of additive and multiplicative inverse elements.* $\forall a \in F$ *there is a* $-a$ *such that* $a + (-a) = 0$*. Similarity, for every* $a \neq 0$ *there is an* a^{-1} *such that* $a \cdot a^{-1} = 1$
6. *Closure of F under addition and multiplication. As defined under REF XXX, addition and multiplication of members of F results again in a member of F. Formally* $\forall a, b \in F, a + b \in F$ *and* $a \cdot b \in F$

Common examples for fields are the real numbers $\mathbb{R}$ and complex numbers $\mathbb{C}$.

Vector spaces

Over these fields we can define *Vector spaces*, sets of elements with two operations - vector addition and scalar multiplication - that fulfill certain criteria. Here vectors denote elements of the vector space V over a field F, whereas scalars refer to elements of the field F. A vector can differ from the familiar geometrical vectors and refer to

other structures, in our case functions. However as mentioned above, we must be able to define vector addition and scalar multiplication consistent with the following axioms:

Definition 7 (Vector space)
Vector addition and multiplication of vectors with scalar must be defined. For a vector space $V over F$ *with arbitrary vectors* $u, v, w \in V$ *and arbitrary scalars* $a, b \in F$

1. *Vector addition must be associative,* $u + (v + w) = (u + v) + w$
2. *Vector addition must be commutative* $u + v = v + u$
3. *There must exists a vector* $0 \in V$ *for which* $v + 0 = v$*. This is called the zero vector and serves as additive identity element*
4. *There must be an additive inverse* $-v \forall v \in V$ *such that* $v + (-v) = 0$
5. *Scalar multiplication must be compatible with field multiplication* $a \cdot (bv) = (ab)v$
6. *There must be an identity element* 1 *for scalar multiplication such that* $1v = v$
7. *Scalar multiplication with respect to vector addition must be distributive,* $a \cdot (u + v) = au + av$
8. *Scalar multiplication with respect to field addition must be distributive,* $(a + b)v = av + bv$

Function Spaces

We can also define vector spaces of with functions as vectors. They need to fulfill the same axioms. Generally, a function space describes any set of functions mapping a set X to a set Y. If Y is a field (be it scalar or vector), the functions implicitly have scalar multiplication and vector addition defined (see definition 7) and thus are inherently vector spaces [Yam12]

Inner product space

For a subset of vector spaces we can define an inner product $\langle \cdot | \cdot \rangle$, forming inner product spaces.
An inner product is an operator that maps a combination of two vectors from the vector space onto the field over which the space has been defined. It also has to satisfy the following 3 axioms:

Definition 8 (Inner product)

1. *Conjugate symmetry* $\langle \cdot | \cdot \rangle = f(x, y) = \overline{f(y, x)}$ *which reduces to symmetry for a vector space over R and to complex conjugate symmetry over a vector space C.*

2. *Linearity in the first argument (see definition 3)*

3. *Positive-definiteness, meaning* $\langle x \cdot x \rangle > 0, \langle x \cdot x \rangle = 0 => x = 0$

2.2.2 Operations on Vector spaces

With this product we can define a norm to measure distances in the vector space and define the concept of orthogonality between vectors.

Definition 9 (Norm and Distance)
A Norm in a vector space is a function which assigns a strictly positive scalar value to any given vector and has some special following properties. We write:

For a Vector field V *over a field* F*, the the function* $p : V \to \mathbb{R}$ *is a norm if and only if for* $a \in F$ *and* $u, v \in V$

1. $p(av) = |a|p(v)$
2. $p(u + v) \leq p(u) + p(v)$
3. *if* $p(v) = 0$ *then* v *is the zero vector*

The Distance *between* u *and* v *is then defined as the norm of their difference.*

Definition 10
Two vectors are orthogonal if and only if their inner product is 0.

Measure space

Next we define the concept of a measure or metric space. Simply said, a measure space is a set for which the distance between all members of the set is defined. The most familiar measure space is the Euclidean space, with the Euclidean distance as measure function. The formal definition is as follows:

Definition 11 (Measure space)
d: Metric on set M, $d : M \times M \to \mathbb{R}$ *such that* $\forall x, y, z \in M$ *the following holds:*

1. *The metric is non-negative,* $d(x, y) \geq 0$
2. *if the metric is 0, the items are identical,*$d(x, y) = 0 \iff x = y$
3. *the metric is symmetrical,* $,d(x, y) = d(y, x)$
4. *the metric fulfills a triangle inequality,* $,d(x, z) \leq d(x, y) + d(y, z)$

Measure spaces enable us to define Lebesgue integration on the vector space, a form of integration which is more generally applicable than the more familiar Riemann integration[GHD+10], as well as the concepts of *Completeness* and L^p spaces.

Completeness

In order to define integrals for the vector space, we need to ensure that the vector $v(x+n) - v(x)$ exists for $\lim_{n\to 0}$ with $v(x)$ denoting a vector from the vector space and $v(x+n)$ denoting a vector with a distance proportional to n away from this vector, such that $v(x) = v(x+n)|_{n=0}$. Formally we can define this using Cauchy sequences.

Definition 12 (Completeness)
Any Cauchy sequence of points in the vector space - meaning a sequence of points which become arbitrarily close to each other as the sequence progresses - has its limit also within that vector space.
so for $\phi_a, \phi_b \in VoverM, d(phi_a, \phi_b) \to 0$ *both* ϕ_a *and* ϕ_b *should lie in* V*.*

l^p spaces

If we define a function space of measurable functions (functions between measure spaces) so that the *pth* power of every function is Lebesgue integrable this function space forms an l^p space. With this, we ensure by definition that we can integrate all functions within this space using the Lebesgue integral and the p-norm, the latter usually defined with the inner product.

Definition 13 (l^p spaces)
The set of measurable functions f *from a measure space* M *to* $\mathbb{C}$ *or* $\mathbb{R}$ *forms a* l^p *space with p-norm* $\|f\|_p$ *if that p-Norm is finite. Formally we write:*
$\|f\|_p = (\int_M |f|^p)^{1/p} < \infty$ *The* L^p *space if composed of all functions for which the integral converges.*

All l^p spaces are complete, since we require that the Lebesgue integral exists and converges over their absolute values, which implies the existence of the necessary limit within the space.

Hilbert space

For the special case of $p = 2$ the l^p space forms a Hilbert space. This analogous of using the inner product $< f|f >$ to define the p-norm $\|f\|_2$. Hilbert spaces generalize most geometric concepts like projections, change of basis, length, angle etc. to any finite or infinite number of dimensions.
All Hilbert Spaces are also Banach spaces, meaning both the Fréchet derivative and Gâteaux derivative are defined on them.

Gâteaux derivative

The Gâteaux derivative generalizes the idea of differentiation along a vector, which is itself a generalization of partial derivatives. It is defined for locally convex topological vector spaces (for example Banach spaces and thus Hilbert spaces) . The definition of the Gâteaux derivative for Banach spaces is as follows:

Definition 14 (Gâteaux derivative)
If X and Y are Banach spaces, the set $U \subset X$ is open and F is a function from $X \to Y$, then the Gateaux differential $dF(u;\psi)$ of F at $u \in U$ in the direction $\psi \in X$ is defined as
$dF(u;\psi) = \lim_{\tau \to 0} \frac{F(u+\tau\psi)-F(u)}{\tau} = \frac{d}{d\tau} F(u+\tau\psi)|_{\tau=0}$

Fréchet derivative

While the Gâteaux derivative generalizes differentiation of multivariate functions in a given direction, the Fréchet derivative generalizes the notion of the derivative of a real-valued function of a single, real variable to vector valued functions of multiple real variables. Its definition for Banach spaces is as follows:

Definition 15 (Fréchet derivative)
For two Banach spaces V and W, an open subset $U \subset V$ and a function $f : U \to W$, f is called Fréchet differentiable at $x \in U$ if the bounded linear operator $A : V \to W$ that fulfills the following relation exists:
$\lim_{h \to 0} \frac{\|f(x+h)-f(x)-Ah-rVert_W}{\|h\|_V} = 0$
If there exists such an operator A, it is unique and we call it the Fréchet derivative of f at x. We write $Df(x) = A$.

Taylor series for function spaces

With the Fréchet derivative defined for transformations between two Banach spaces, we can define a Taylor expansion of a a transformation two function spaces which are also Banach spaces as follows[**?**]:

Definition 16 (Taylor Series)
Suppose U is any open subset of $\Phi(\Omega_1)$ and $F : \Phi(\Omega_1) \to \Phi(\Omega_2)$ is a map which is C^n in U (that is, the first n derivatives of T exist). Let $\phi \in U$ and $\alpha \in \Phi(\Omega_1)$ be

such that $\phi + \theta\alpha \in U \forall \theta \in [0, 1]$. *Then:*
$T(\phi + \alpha) = \Sigma_{k=0}^{n-1} \frac{T^(k)(\phi)(\alpha)^k}{k!} + R_n(\phi, \alpha)$
with
$R_n(\phi, \alpha) = \int_0^1 \frac{(1-\theta)^{n-1} T^{(n)}(\phi+\theta\alpha)(\alpha)^n}{(n-1)!}$

This will be important later when we define and analyze nonlinear operators on Fields.

Generalized Fourier Series

Like the Taylor series, we can generalize the Fourier series to be applicable to function spaces. Using the concept of orthogonality and Norm (see definitions 9 and 10) we can define orthogonal bases $e_k, k \in 0...\infty$ as the set of vectors e_k which are orthogonal to all other vectors in the set and also have a norm of 1. We can then define the Generalized Basis Expansion

Definition 17 (Generalized Basis Expansion)
Letf be a mapping between two Banach spaces with the inner product $\langle \cdot, \cdot \rangle$ *defined. f can then be represented by an expansion over its orthogonal constituents or bases* $\{\phi_n(x)\}$ *as follows:*
$f(x) = \Sigma_{n=1}^{\infty} c_n \phi_n(x)$
the real valued expansion coefficients c_n *are formally defined as*
$c_n = \frac{\langle \phi_n, f \rangle}{\langle \phi_n, \phi_n \rangle}$

which takes the role of the Fourier series. Equivalent to the familiar one dimensional scalar case,we can define Parsevals equality or identity, which states the the sum of the squares of the orthogonal components of an element of a Hilbert space is equal to the square of the element (all defined over the inner product). This is equivalent to a generalization of the Pythagorean theorem in Hilbert spaces.

Definition 18 (Parsevals equality)
$\|\phi\|^2 = \sum_k (\phi \cdot e_k)^2$

Similarly, we can define and Parsevals relation, which states that the sum of the orthogonal inner products of the elements are equal to the inner product of the elements.

Definition 19 (Parsevals relation)
$\phi \cdot \psi = \sum_k (\phi \cdot e_k)(\psi \cdot e_k)$

Generalizing Transformations of the Hilbert Space

Since we will be working with Vector spaces over fields, we give the definition of a linear transformation between two Vector spaces as follows:

Definition 20 (Linear Transformation)
Given two vector spaces V, W over the the same field F, we call a mapping $f : V \rightarrow W$ a linear map *or* linear transformation *if it satisfies:*

1. $(x + y) = f(x) + f(y)$
2. $f(\alpha x) = \alpha f(x)$ *for* $\alpha in F$

This means any linear transformation will not change the structure of the Vector spaces. For example, a multiplication with a scalar factor will not change the direction of a vector, just its amplitude or value.

Chapter 3

A new type of computing?

3.1 Motivation

In [Mac] as well as [Mac87] and [Mac97] that in order to fully develop fields like AI and sensory processing, we must leverage *massively parallel* computing. Massively parallel is defined as follows[Mac87]:

Definition 21 (Massively parallel)
A computational system can be considered massively parallel if the number of processing elements is so large, that it may conveniently be treated as a continuum.

Note the phrase "conveniently". Much of MacLennan arguments derive from a certain sense of pragmatism, which is not unique to him. Even within traditional computer science and physics, we accept approximations both in representations of numbers (think floating point) and in the derivation of theorems and concepts (think Kirchhoff's law). As long as we can clearly define the precision bounds in which we compute, we are willing to accept small errors. Even traditional Turing machines, the standard model of computation (for a brief primer, see appendix A) uses a very pragmatic definition of computable numbers, computing the number "to the desired precision"(appendix A.5).
MacLennan further argues that in order to fully exploit the emerging world of massively parallel computers (whether it be through breakthroughs in molecular computing [Zau05] or through the increasing parallel capabilities of traditional silicon computers[HVG+06] [Hil] [JRP11] we must also develop a framework to reason in parallel and design algorithms treating the computation as truly, massively parallel, not just as doing multiple sequential operations at the same time. It is the derivation of this framework, which we will present in this chapter. We will try to to immediately relate it to Turing computability by thinking of the field computer as a Turing Machine specialized on parallel execution. We will also argue why this is a valid and even natural model.

3.2 Our new Symbols : Fields

For the definition of this new, massively parallelized computation framework, MacLennan[Mac] suggests the definition of the *Field* as the basic computing unit (similar, but not identical to a mathematical lowercase *field* of functions). He specifically defines Fields to be physically realizable, which poses restrictions among other things on their dynamic range and gradient. He defines the general properties of Fields as follows[Mac]:

Definition 22 (Fields)

1. *Fields must be realizable. This translates into a number of mathematical properties. Their dynamic range is limited, they are all uniformly continuous (most are even continuous under a Lipschitz condition, see definition 2). We also require their gradient to be finite, for in the real world there are no infinite rate of changes within continua.*

2. *Fields are functions from a measure space* Ω *to a range* K*, with* K *being a subset of an algebraic field (see section 3.2). We write* $\phi : \Omega \to K$
   Fields also belong to a linear space $\Phi_K(\Omega)$ *, fulfilling most of the axioms of vector spaces for the vector addition* $\phi + \psi$ *and scalar multiplication* $a\phi$ *for* $a \in \Omega$ *and* $\phi, \psi \in K$ *:*

3. *Associativity,* $(\phi + \psi) + \chi = \phi + (\psi + \chi)$

4. *Commutativity,* $\phi + \psi = \psi + \phi$

5. *Additive identity element* $0 \in \Omega$ *for which* $\phi + 0 = 0 + \phi = \phi$

6. *Unitive element* 1 *of* K *for which* $1\phi = \phi$

7. *Additive inverse* $-\phi$ *for every* $\phi \in \Phi_k$ *such that* $\phi + (-\phi) = 0$

8. $a(\phi + \psi) = a\phi + a\psi$

9. $(ab)\phi = a(b\phi)$

10. $(a + b)\phi = a\phi + b\phi$

MacLennan specifically notes [Mac] that the fields cannot be closed under these operations, since this would require the range to be unbounded. Thus Fields do *not* fulfill the field axioms and are not fields in the mathematical sense. They are, however a *subset* of the field of functions $\Omega \to K$. Specifically, they are the physically realizable subset of this field. Taking this into account, all definitions we carry over from mathematical fields have exceptions and are undefined whenever the operation would cause us to leave Ω or K . This can be thought of analogous to the undefined behavior when dealing with overflows in classical programming, or the undefined

relations mentioned in example A.1.1.
We can further define an inner product for Φ_k as follows:

Definition 23 (Inner product)
For two fields $\phi, \psi \in \Phi(\Omega)$ *we define*
$\phi \cdot \psi = \int_\Omega \phi_t \psi_t dt$

It satisfies the properties of a real inner product as defined in definition 8. With this Inner product we define a Norm and Orthogonality as usual. Since we require Fields to have bounded domain and range, we know that every $\|\phi\| \leq \beta_\phi |\Omega|^1/2$ .This means all Fields are finite in space, which again means the space they belong to is a Hilbert space. We can therefore therefore use the Fréchet derivative to define Taylor expansions and use the Norm and the concept of Orthogonality to define a Fourier series for Fields. These are important when it comes to defining and proving operators.

3.3 Transitions - Field Transformations

We can now define operators on these fields that map from one field to another. This will be the analogue to state transitions in serial Turing Machines. Since in the physical world, we will have to deal with noise we require all Field transformations to be continuous by using the norm:
$\lim n \to 0 \|\phi_n - \phi\| = 0 \to \lim_{n \to 0} \|T(\phi_n) - T(\phi)\| = 0$

This gives our framework some robustness against noise.

3.3.1 Transformations required to define an Universal Field Computer

Analogous to the universal Turing machine, which can simulate any computation with the operations based around reading a symbol, performing altering the symbol and/or moving the head, MacLennan showed [**?**] that a reasonably universal field computer could approximate any mathematical operation using the following primitive operations:

1. local addition: $(\phi\psi)_t = \phi_t + \psi_t$
2. outer product:$(\phi \wedge \psi)_{st} = \phi_t \psi_t$
3. general product:$(\phi\psi)_{su} = \int_\Omega \phi_{st} + \psi_{tu} d\mu(t)$

Note that these are not symbolic operations, they are well defined linear operations on the mathematically defined Field of real or complex numbers. In order to express them as a machine, one has to define a mapping between a suitable physical metaphor for the abstract mathematical process. For example, in order to express local addition one can imagine a two Fields $f1, f2$ over $\Omega = [0, 0.5]$ represented by the voltage levels of 5000 parallel wires each. In order to perform local addition, one would connect pairs of the wires to an adding circuit (for example using an op-amp). This would be an example of a non-universal Fields computer, which can only compute addition of two Fields.

3.3.2 Precision of Transformations

It should also be noted that the key concept here is *approximation*. When designing frameworks and algorithms, there are always trade-offs to be made. Field computation sacrifices absolute precision for an easier way to model massively parallel operations. On this scale, any number of singular incorrect points is irrelevant [Mac89]. The overall precision depends on the implementation and length of computation. If for example we want to calculate a fraction to a certain digit, we need to take care that a field computer consists of components able to support that calculation (high end components, enough time for computation etc).

3.4 Tape - representing the Fields

Of course, using 10000 wires to represent two fields of such a small range is not practical. Representing the values however is the biggest challenge. As defined in definition 22, we have are constrained to physical media by definition. This aids in implementation, but makes it hard to find a suitable medium for storing the fields. Whereas an ideal Turin machine can simply assume a finite but arbitrarily long tape and has had the benefit of very close physical metaphors for its abstractions (voltage levels for discrete symbols, different storage media made up of discrete units or regions) a storage medium for Fields is hard to come by. Since he is presenting a mathematical framework, MacLennan himself skirts this issue. He only vaguely mentions ideas about using frequencies, intensity, concentration and other possible continuous properties of optical and chemical media as storage units without going into details. Jiang et. al. touched one possible line of research in [JRP11]. If we loosen the constraint of truly continuous fields and use MacLennans own definition of the *continuum limit of massively parallel operation*[Mac99a], we can simply model the Field storage unit as one tape of a multi-tape multi-head Turing machine. Each field would be represented by one tape, with one head on every available position on the tape. As can be shown [UJ79], this is no more expressive than a conventional Turing Machine, though it might be faster on certain problems.

3.5 Summary - Comparison with Universal Turing machine

Given our analogies so far, can we treat a field computer as a Turing machine? For a truly continuous (one could say, *ideal*) field computer, the answer would be no. This is due to the infinitely many values in the metric space of the domain over which our fields are defined. A Turing machine can only deal with arbitrary long but finite tapes and finite sets of symbols, which would be impossible if we were to deal with a true continuum. However, in order to stay realizable we have to step away from truly continuous representation, even MacLennan himself restricts the criterion of a continuum to be "sufficiently" continuous. So, given the expected problem resolutions and ranges, we need sufficiently large but finite elements in our measure space in order to operate on them with field computing. This means that real world field computers can be treated as Turing machines with either an encoding of the elements in the measure space, or simply with symbols corresponding to each possible value. The multi-tape example is more intuitive, as we can imagine the Turing machine storing each field on a separate tape.
In this way, we can model a field computer as a Turing machine. Indeed, MacLennan himself states in his more practically oriented papers [Mac99a, Mac97, Mac87] that the distinction becomes meaningless in real life and that it all comes down to trade-offs.
As to the question of hyper computability, since an implementable universal field computer needs compute approximations of continuous operation on a discrete space instead of a truly operating on a continuous space, it can be simulated by a Turing machine. If it were truly able to perform hyper-computation andperform computation not simulatable by a Turing Machine, it would not be implementable.

3.6 Concepts and Intuition

It is however just as impractical to think about Turing machine programming when dealing with field computers as it is with classical computers. A Turing machine decomposes every operation to a symbolic algorithm, which becomes cumbersome (as seen in example A.2.1 quickly. We need equivalents of abstractions like registers,variables, functions etc.

While in traditional computers we use variables and registers to store values, in a field computer there is no direct representation of the values. Instead, we encode them in a function over the measure space. How we encode them is left to the designer of the algorithm and should be chosen according to the problem. For example, a shape might be encoded as a curve, with the point sampled from the measure space as the curve parameter. A gray-scale image might be encoded in a two dimensional field, with the value at each point in the measure space correspond-

ing to the brightness. As with classical computers, some values are better suited for encoding in a field computer, some less. Likewise some operations are cumbersome on a field computer that are trivial to implement on a classical computer (for example the mean of a finite number of values) and vis á vis (for example, convolutions and filtering are trivial operations in field computers).

One intuitive way to reason about fields is to treat them as infinite dimensional vectors. E.g., a vector can be thought of as a function $f : \mathbb{N} \to \mathbb{R}$ returning the vector element $n_i \in \mathbb{R}$ for an index $i \in \mathbb{N}$, with strictly limited i. A real field does the same thing, except it performs the map $\mathbb{R} \to \mathbb{R}$ instead.

In field computation, functions have to treated as chained operators on a single input ,which is in itself a (possibly time varying) independent function. This is very similar to the substitution of the lambda calculus [Chu85] and modern functional programming approaches[Wad92][Hud89].
Overall, field computation is not a new computational model, but a mathematical framework which can aid in designing massively parallel algorithms. Once defined in this framework, they are easier implemented in parallel computing media which can keep its metaphors close enough to the abstraction.

3.7 Possible Implementations

Given the difficulties of representing fields, feasible implementation for practical field computers are hard to imagine. While there have been some approaches using optical and electric continuous computation media [Mac99a] it seems that with current technology, a finite or continuum-limit Field computer is all that is achievable.
The most promising technology *currently* available are FPGA chips. Large scale FPGA chips deliver around 1300 logic elements per dollar [Mod13] - measured in LUTs, Look Up tables, which of course also contain some additional logic - with the limiting factor being the IO pins. If chip designer increase the parallel IO capabilities and/or we can can develop computer architectures built around massive parallelism, even today we can theoretically build a weak general purpose field computer. If we assume we can formulate most algorithms with around 100 steps, taking the 100 step phenomenon as our baseline, and we need to use at most around 300 [MMB12, MA09] LUTs per step and element of the field, we can implement a field computer for a very rough estimate of 23 dollar per field element. This is of course only a ballpark estimation of the cost of implementing a discrete Field computer using FPGA today. If we want to implement a continuous Field computer, we have to add a factor between 10^3 and 10^6, depending on the range and the precision needed. In any case, while costly, if one were forced to construct a field computer it would be doable. Of course, with $23\$ * 5000 * 3 \approx 350000\$$ for the example in section 3.3.1, building real world applications with this would be prohibitively expensive.

Other possibilities include implementation of the Field computer primitives on platforms like SpiNNaker and its related platforms, either directly or modeled as a Neural Network[KLP+08], as a VLSI or as a library running on hugely distributed systems. In any case, if implemented on classical architectures, the frameworks abstractions are best captured by functional programming approaches which allow the inherent parallel nature to persist when implementing the primitives. The formalism here serves both to *define the required hardware* and to *simplify the creation of algorithms* once a suitable hardware can be found.

Chapter 4

Conclusion

In conclusion, while field computation is cumbersome to work with when dealing with scalar values and problems unsuitable to parallel computation, it can provide a useful abstraction when designing parallel algorithms. It is best used for mathematically modeling massively parallel computation, implemented with functional programming in mind.

Claims of super- or hyper-Turing computation capabilities seemingly rely on physical implementation details which are currently impossible to achieve. One example for this is a storage of truly continuous, in essence infinitely partition-able measure spaces on which to base our Fields. Implementable Field computers are an abstraction of specialized Turing machines designed for parallel use.

As for the architecture of a Field computer, much of the design details depend on the computing medium chosen. The most practical implementation which is currently possible is based on FPGAs or neuromorphic distributed systems architecture like SpiNNaker. Advances in molecular and natural computation might also make implementation of a field computer more sensible.
Regarding future work, development of an actual field computation stack has to happen on multiple levels,both scientific and engineering topics. To build a specialized field computer, one has to research and implement storage media to hold either data - be it continuous or discrete - which can be accessed in parallel. Furthermore, one has to develop a logic and then an instruction set that both build upon the mathematical framework presented in this paper. The development of the instruction set would of course be coupled with the development of an architecture on which it could be run. Following this, one can define algorithmic primitives and data structures to be used as primitives when programming field computers.
Alternatively to developing a new computation media, one could try and develop synthesizable VHDL or Verilog libraries implementing the set of primitives on conventional silicon.

While the universal field computer is not here and might not be here, the formalism serves to clearly define the requirement for a suitable computing and storage medium for continua. It is also useful as an abstraction to design parallel algorithms which can be suitable written as a field. This serves as a generalization of the vector formalism to infinitely fine grained - continuous - sampling.

Appendix A

Brief History of Turing Computability

As Field Computation is sometimes mentioned in the context on non-Turing computability[Sta06, Sta04, Mac03, Mac09] we will briefly present the history, mathematical foundation and mainstream modern interpretation of the Turing Machine, including an example Turing machine.

A.1 History and Motivation- The *Entscheidungsproblem*

Already during the emergence of computing machines[1] there were notions of a machine that could determine the correctness of mathematical statements. Leibniz realized that the first step towards this would have to be an unambiguous formal language. David Hilbert and Wilhelm Ackermann posed the Problem in their work "Principles of mathematical Logic" as part of their presentation of first-order logic.

A.1.1 First-order Logic

First-order logic is the clean formal language which Leibniz knew would be necessary before a machine that could "solve" mathematical problems could even be though about. It is a formal language intended for reasoning over mathematical structures (*Symbols*) in which each statement can be broken down into pairs of subjects and predicates.
Subjects,or Entities, are the structures we want to act upon. They might be numbers, variables, spaces of variables or functions.
Predicates serve to define or modify the properties of the subjects.
Each predicate can only refer to a single subject, but a subject might itself be made up of a Predicate/subject pair. They may also be compared, in the manner of "if

[1] starting with Blaise Pascals mechanical calculator and Gottfried von Leibniz' *Stepped Reckoner*

predicate 1 is true, so is predicate 2" and similar statements
Variables are placeholders for Subjects which can be quantified over. Possible qualifications are *all*,*no*,*some*. These quantified variables may be used with predicates to reason over groups of subjects.

Example A.1.1
For an example of first order logic, we define the variable a and the predicates isacat and isananimal. If we then state
For every(quantifier) a(variable, if a is a cat, a is an animal (two compared predicates)
we have also implicitly stated:

1. *There are animals which are not cats*
2. *There are no cats which are not animals*
3. *If a is not a cat, it may or may not be an animal (undefined) We can now build up an unambiguously defined framework of predicates and subjects exactly describing our system.*

A.1.2 Entscheidungsproblem as given by Hilbert

Given this definition of first-order logic, the Entscheidungsproblem in the form given by Hilbert and Ackermann asks for an algorithm that takes in a statement formulated with a first order logic and optionally additional constraining axioms and returns an answer "Yes" and "No", indicating whether the statement is valid or not. In order to answer the question, the idea of an "algorithm" had to be also formally defined. This was done independently both by Alonzo Church with his lambda calculus[Chu85, **?**] and his doctoral student Alan Turing with his universal Turing machine[Tur36], which became the basis for the modern computer. [2]

A.2 Turing Computability and the Turing Machine

Alan Turing imagined a machine which had a finite but always sufficiently long tape on which the symbols it would act upon are stored. The machine could only ever access one symbol at a time, the scanned symbol and would be influenced by the symbol as well as able to alter the scanned symbol. It would also have the capability to move the tape left and right, gaining access to the other symbols. The manner

[2] another formal model of computation was developed by Gödel and Kleene with the μ-recursive functions[Kle79, GKR34]

in which it did this would be defined by the machines state (which had to be finite in number) , which could be influenced by the scanned symbols. Each state has a number of transitions (also finite).
The parts in unison encode the computational steps of an algorithm. As an example , we describe a Turing machine which increments a given binary number by one.[]

Example A.2.1
We start with a tape which has the number we want to increment printed on it in binary,with leading zeros (in this case, 0011 for the number 3). The machine knows the symbols 0,1 and blank*. To make things simpler, we assume the head starts on the first digit and the number is printed such that the first digit is the first position on the tape. We then define the following algorithm.*

1. *Start in State 0.*
2. *if in State 0: If the symbol read is not* blank*, move the head to the right (along the tape).Else switch to state 1 and move the head to the left.*
3. *if in State 1: if the symbol read is a 1 change it to zero. If the symbol read is a 0 change it to 1 and end the algorithm.*
4. *go back to step 1*

Applied to our sample case, 0010, the machine performs the following steps:

1. *Start in State 0.*
2. *Read symbol is a 0, move to the right.*
3. *Read symbol is a 0, move to the right.*
4. *Read symbol is a 1, move to the right.*
5. *Read symbol is a 1, move to the right.*
6. *Read symbol is* blank*, move to the left and go into State 1*
7. *Read Symbol is a 1, change to 0, move to the left.*
8. *Read Symbol is a 1, change to 0, move to the left.*
9. *Read Symbol is a 0, change to 1, halt.*

After 9 steps, we have increased the number on the paper from 0011 *to* 0100

While the example above is that of a specialized Turing machine encoding a single algorithm that requires external preparation (setting of initial state, encoding etc.), Alan Turing proved that a universal Turing machine could be constructed which could simulate all other Turing machines. Later it was proven that this universal

Turing machine would only be slower by a logarithmic factor[AB09] when compared with the specialized machine. Turing thus created a framework to describe computations, with the following formal definitions:

Definition 24 (Turing Machine)

1. *Q is a finite, non-empty set of states*
2. *$F \subseteq Q$ is the set of final or accepting states*
3. *Γ is a finite, non-empty set of tape alphabet symbols*
4. *$b \in \Gamma$ is the blank symbol*
5. *$\Sigma \subseteq \Gamma \setminus \{b\}$ is the set of input symbols*
6. *$\delta : (Q \setminus F) \times \Gamma \to Q \times \Gamma \times \{L, R\}$ here δ is the transition relation between a combination of state and read symbol and the alteration of these two, combined with moving the tape either left or right*

The systems full state, also called complete configuration, consists of the current state, any non-blank symbols on the tape and the position of the tape

Any algorithm computable in finite steps using this framework is defined as "Turing Computable".

A.3 The Church Turing Thesis and the answer to the Entscheidungsproblem

There had been other formalizations of computation models, namely the lambda calculus of Turings doctoral father, Alonzo Church, as well as the general recursive functions defined by Gödel and Herbrand. Turing and Church proved the equivalency of these defined classes of computable functions.
It's relevance to the Entscheidungsproblem was that Church and Turing proved independently that there are statements for which a lambda calculus or a Turing machine could not determine the validity. Turing did this by first equating the Entscheidungsproblem with the Halting Problem (given a program and an input, determine without running the program whether the program will halt), which he had previously proven was not solvable by a Turing machine.[Tur36] He and his doctoral father assumed that all of the most powerful formal computation models are functionally equal to another. This provided a negative answer to the Entscheidungsproblem in all implementable systems. There remains some debate whether the assumption is true, but given that "effectively computable" remains an informal definition, this is an irresolvable dispute. The majority seems to agree that all computable functions are Turing Computable[3] and vice versa. This assumption is

[3]Turing himself proved that his machines could perform all computations a human could do with pen and paper

known as the Church-Turing thesis.

A.4 Limits of Turing machines

While Turing machines seem to capture all human computable problems, there are instances where they are a cumbersome model. Traditional computers using von Neumann [GH93]or Harvard architectures provide multiple abstractions to make dealing with computation more convenient than programming a pure Turing machine. They are also an ill fitting model when it comes to concurrent computation. While it is possible to simulate Concurrent computation, for example with an interleaving model of computation, there are more elegant and expressive models for this. Indeed, there exists is a whole field of mathematics dedicated to the study of concurrent system, called the process calculus. For more on this topic see [Mil80] and [AG07].

A.5 Summary

Turing computability is defined as being solvable by a Turing machine. A Turing machine is defined for finite, discrete alphabets, as well as a finite number of discrete states and atomic transitions. Given the right instructions on tape, a universal Turing machine can compute any Turing computable algorithm. The Church-Turing thesis states that this probably captures all algorithms computable by humans, which cover all algorithms to date. In order for these algorithms to be generally applicable, one has to include encoding and decoding routines that map between the input values and the symbols represented on the tape.

For our discussion of field computation, we should keep in mind the inherent discreteness of the Turing machine. We should also keep in mind all real numbers are accurately computable by Turing machines given infinite time and tape, but that in practice we only concern ourselves with the *required precision* and terminate the computation once it is reached[Fre83]. Another constraint is the Beckenstein bound[Bek72, CMO01] which limits the precision achievable with both a real-world Turing machine, as well as all other physical computers.

Bibliography

[AB09] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Number January. 2009.

[AG07] Luca Aceto and AD Gordon. *Algebraic process calculi: The first twenty five years and beyond*. Number June. 2007.

[Bek72] JD Bekenstein. Black holes and the second law. *Lettere Al Nuovo Cimento (1971–1985)*, 1972.

[BM96] Guy E. Blelloch and Bruce M. Maggs. Parallel algorithms. In *ACM Computing Surveys*, volume 28, pages 51–54. 3 1996. doi:10.1145/234313.234339.

[Chu85] Alonzo Church. *The Calculi of Lambda-conversion*. 1985.

[CMO01] Rong-Gen Cai, Yun Soo Myung, and Nobuyoshi Ohta. Bekenstein bound, holography and brane cosmology in charged black hole backgrounds. page 17, 5 2001. doi:10.1088/0264-9381/18/24/308.

[Fre83] Rudolf Freund. Real functions and numbers defined by turing machines. *Theoretical Computer Science*, 23(3):287–304, 1983.

[GH93] M. D. Godfrey and D. F. Hendry. Computer as von neumann planned it. *IEEE Annals of the History of Computing*, 15:11–21, 1993. doi:10.1109/85.194088.

[GHD+10] F W Gehring, P R Halmos, C Deprima, I Herstein, J Kiefer, and W Leveque. *Undergraduate Texts in Mathematics*, volume 51 of *Undergraduate Texts in Mathematics*. 2010.

[GKR34] Kurt Gödel, Stephen Cole Kleene, and John Barkley Rosser. *On undecidable propositions of formal mathematical systems*. 1934.

[Hil] W. Daniel Hillis. The connection machine. *Citeseer*.

[HS86] W. Daniel Hillis and Guy L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 12 1986. doi:10.1145/7902.7903.

[Hud89] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys (CSUR)*, 21(3), 1989.

[HVG+06] Martin Herbordt, Tom Vancourt, Yongfeng Gu, Bharat Sukhwani, Josh Model, Al Conti, and Doug Disabello. Case studies in fpga acceleration of computational biology and their implications to development tools*. 2006.

[JRP11] Hua Jiang, Marc D. Riedel, and Keshab K. Parhi. Asynchronous computation with molecular reactions. In *Conference Record - Asilomar Conference on Signals, Systems and Computers*, pages 493–497, 2011. doi:10.1109/ACSSC.2011.6190049.

[Kle79] Stephen C. Kleene. Origins of recursive function theory. *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*, 1979. doi:10.1109/SFCS.1979.33.

[KLP+08] M.M. Khan, D.R. Lester, L.a. Plana, a. Rast, X. Jin, E. Painkras, and S.B. Furber. Spinnaker: Mapping neural networks onto a massively-parallel chip multiprocessor. *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, pages 2849–2856, 6 2008. doi:10.1109/IJCNN.2008.4634199.

[MA09] Khader Mohammad and Sos Agaian. Efficient fpga implementation of convolution. In *Conference Proceedings - IEEE International Conference on Systems, Man and Cybernetics*, pages 3478–3483, 2009. doi:10.1109/ICSMC.2009.5346737.

[Mac] Bruce J MacLennan. *Field computation: A theoretical framework for massively parallel analog computation. Parts I-IV.*

[Mac87] BJ Bruce J MacLennan. Technology-independent design of neurocomputers: The universal field computer. In *Proceedings, IEEE First International Conference on Neural Networks*, volume 3, pages 39–49, 1987.

[Mac89] BJ MacLennan. Continuous computation: Taking massive parallelism seriously. 1989.

[Mac94] Bruce J MacLennan. Continuous computation and the emergence of the discrete. 1994.

[Mac97] Bruce MacLennan. Field computation in motor control. *Advances in Psychology*, 119:37–73, 1997.

[Mac99a] BJ MacLennan. Field computation in natural and artificial intelligence extended version. *Information Sciences*, pages 1–28, 1999.

[Mac99b] Bruce J MacLennan. Field computation in natural and artificial intelligence. *Information Sciences*, 119(1):73–89, 1999.

[Mac03] BJ Bruce J MacLennan. Transcending turing computability. *Minds and Machines*, 13(1):3–22, 2003.

[Mac09] BJ MacLennan. Super-turing or non-turing? extending the concept of computation. *International Journal of Unconventional Computing*, pages 1–21, 2009.

[Mil80] Robin Milner. A calculus of communicating systems. *Lecture Notes in Computer Science*, 92, 1980.

[MMB12] CD Moreno, Pilar Martínez, and FJ Bellido. Convolution computation in fpga based on carry-save adders and circular buffers. *IT Revolutions*, pages 237–248, 2012.

[Mod13] Kevin Modzelewski. Choosing fpga parts 2013, 2013.

[Mor] Autorzy Hans P. Moravec. *Mind Children: The Future of Robot and Human Intelligence*.

[Sta04] Mike Stannett. *Hypercomputational models*. 2004.

[Sta06] Mike Stannett. The case for hypercomputation. *Applied Mathematics and Computation*, 178(1):8–24, 7 2006. doi:10.1016/j.amc.2005.09.067.

[TH] Gyuri Pápay Tony Hey, Juri Papay. The computing universe: A journey through a revolution.

[Tur36] Alan Turing. On computable numbers. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.

[UJ79] John Hopcroft Ullman and Jeffrey. *Introduction to Automata Theory, Languages and Computation*. 1 edition, 1979.

[Wad92] Phillip Wadler. The essence of functional programming. *... -SIGACT symposium on Principles of programming ...*, 1992.

[Yam12] Yutaka Yamamoto. *From Vector Spaces to Function Spaces: Introduction to Functional Analysis with Applications*. 2012.

[Zau05] KP Zauner. Molecular information technology. *Critical reviews in solid state and materials sciences*, 30(1):33–69, 2005.

License