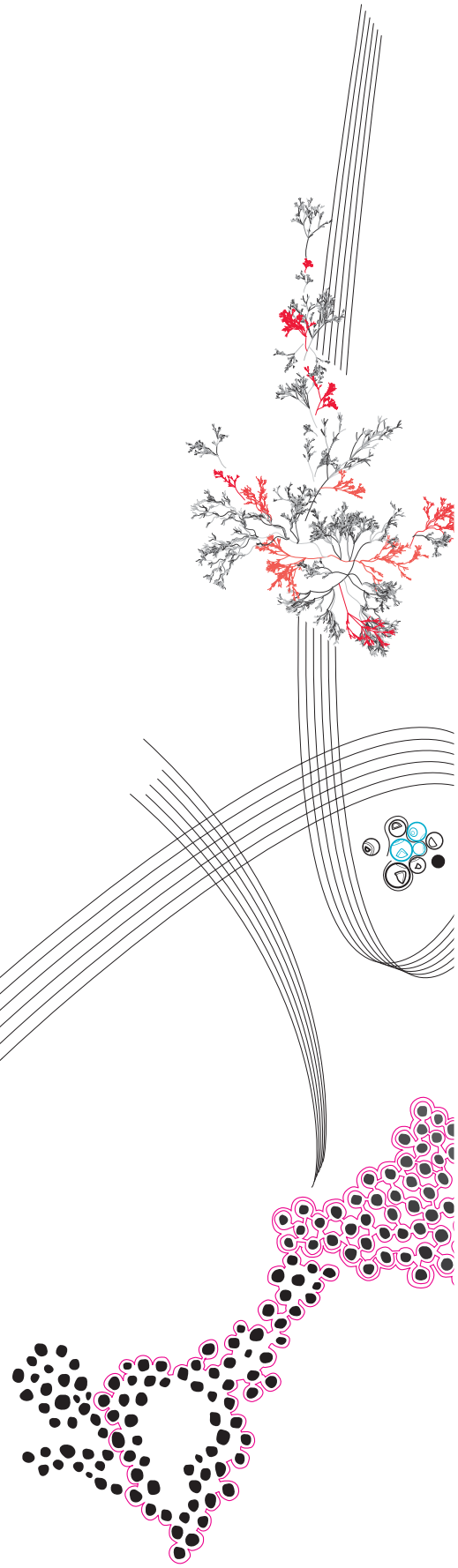


BSc Thesis Applied Mathematics

The Pixel Array Method

Leander van der Bijl

Supervisor: Dr. Matthias Schlottbom

June, 2019

Department of Applied Mathematics
Faculty of Electrical Engineering,
Mathematics and Computer Science

UNIVERSITY OF TWENTE.

Preface

This report was written for my bachelor assignment, which concludes the last module of the Applied Mathematics bachelor on the University of Twente. I learned a lot during the writing of this paper as I did not have a lot of experience with scientific writing.

Firstly, I would like to thank my supervisor, Dr. Matthias Schlottbom, for supervising me during the writing of this bachelor assignment and helping me when I was not sure how to proceed.

Furthermore, I would like to thank Jarco Slager for proofreading this report. I would also like to thank Lotte Gerards, Femke Boelens, Lucas Jansen Klomp, Wisse van der Meulen, Annemarie Jutte, Jesse van Werven, Tessa Rutjens, Nienke Gerards and Sven Dummer for helping me with Overleaf problems, working with me at the university and helping me relax during the lunch break.

The Pixel Array Method

Leander van der Bijl*

June, 2019

Abstract

The Pixel array method is a method that was proposed only a few years ago. It is used for finding all solutions of a set of equations within a certain bound. The pixel array method has a few strong properties. For example, it does not return any false negatives when used. However, because of this property, the error can be quite large. In this paper, a formal definition of the pixel array method will be given with some illustrating examples. The method will then be adjusted such that it is also capable of returning the steady state solutions of a differential equation, this new method will be referred to as the PASS method. MATLAB is used for calculating the steady state answers of discrete approximations of the heat equation and the Fisher-KPP equation with given boundary conditions such that these can be compared to their actual solutions. The heat equation is approximated quite well by the PASS method and we see that the PASS solutions converge to the actual solutions as we increase the resolution. However, the PASS approximation of the Fisher-KPP method does not converge as nicely. Even though this equations only has 2 solutions, the PASS method returns hundreds and the amount keeps growing when increasing the resolution. Most of these PASS solutions are for boundary conditions that do not actually have a steady state solution. This is due to the tolerance, which has to be higher than some threshold in order for it to be valid. Our conclusion is therefore that the pixel array method is a nice method with strong properties. However, further research is needed with respect to this tolerance and the possibilities of lowering the threshold in order for it to give meaningful results for relations that change rapidly.

Keywords: Pixel array method, PASS method, steady state solutions, heat equation, Fisher-KPP equation, hypergraph

1 Introduction

In 2016, a new method for finding all solutions of a set of relations within a certain bound was introduced [5]. Soon after, many applications of this method were investigated. The method can be useful for problems related to hypergraphs [1] or optimizing the lifetime of a battery [3]. However, this paper will mainly focus on finding its limitations when calculating the results of some sets of functions which have already been algebraically solved. Specifically, this paper will focus on the application of the pixel array method to differential equations. The differential equations that were chosen and the slight adjustments that had to be made to the pixel array method were inspired by [2]. Section 2 will briefly describe the pixel array method used in this paper, Section 3 will formally define this method while also giving examples and stating the mathematical foundation. Finally, Section 4 will define an altered version of the pixel array method, called the PASS method, that can

*Email: leandervdbijl@gmail.com

be used for solving differential equations and then use it to solve the heat equation and the Fisher-KPP equation. We will also discuss the results found by the PASS method and compare them to the actual results.

2 Method description

In this section, the pixel array method and its properties are explained. The method was proposed by David I. Spivak, Magdalen R.C. Dobson, Sapna Kumari and Lawrence Wu [5]. The outline of the pixel array method that is used in this paper essentially follows [5], but there are some alterations.

The pixel array method is a method that can be used for finding all solutions of a set of relations within a certain bound. The set of relations will be defined as a set of equations and inequalities. These relations do not have to be differentiable or continuous.

We define the input of the pixel array method as follows:

- A set of equations or inequalities, which we will denote by R.
- All upper and lower bounds of the variables that are used in the relations. Let Δ be the set of all variables used in the relations, then the lower bound is given by $a(p)$ and the upper bounds by $b(p)$ for all $p \in \Delta$.
- The resolution of each variable that is used in the relations. These will be denoted by $r(p)$ for all $p \in \Delta$.
- A certain threshold ϵ which will define the accuracy of the method.
- A set P' of variables which we want to find the solutions of, these will be called exposed variables. The variables that are not exposed will be called unexposed variables.

The threshold is not part of the input in [5], but it is necessary for the method to work. [5] does discuss what kind of properties such a threshold must satisfy but a formal algorithm of finding one was not given. Therefore, we will define it as an input instead and show later how one could go about finding it for specific cases.

Before the formal definition of the pixel array method is given, we describe the outline of the method to give the reader an idea about the structure of the method:

1. Every relation f in R will be transformed in a so called boolean array.
2. We obtain a wiring diagram that defines how the arrays should be multiplied.
3. We multiply the boolean arrays to get a final boolean array which we can translate to solutions of our set of equations R.

Each of these steps will be further elaborated on in Section 3.

3 The mathematical details

This section will go through all the steps of the pixel array method that have been mentioned in the last section, while also giving a formal mathematical definition of it.

3.1 Creating the boolean arrays

The first step of our method requires us to map a relation to a boolean array. Which, when visualized, can be seen as a plot of a function. This subsection will explain how such an array can be created.

We need to start with several definitions. After most definitions, a simple example will be given to give the reader some intuition.

Definition 3.1 (Pack)**.** *A pack T is defined as a tuple (P, a, b, r), where P is a finite set and $a, b : P \to \mathbb{R}$ are functions for which $a(p) \leq b(p)$ for all $p \in P$. $r : P \to \mathbb{N}_{\geq 2}$ is also a function. Each element $p \in P$ is called a port. $a(p)$ and $b(p)$ are called its lower and upper bound respectively, and $r(p)$ its resolution.*

We will define a small pixel array problem such that we can give some examples of this definition in this context.

Example 3.2 (A simple pixel array problem)**.** *For this problem, we consider the set of relations $\{f_1, f_2\}$ where $f_1 : x^2 = z$ and $f_2 : 1 - y^2 = z$. We use the bounds $a(x) = a(y) = a(z) = -1.1$ and $b(x) = b(y) = b(z) = 1.1$, the resolutions $r(x) = r(y) = r(z) = 100$ and x, y are the exposed variables.*

Example 3.3. *Consider Example 3.2. Some examples of packs in this case could be $T_1 = (\{x, y\}, a, b, r)$ or $T_2 = (\{x, z\}, a, b, r)$.*

Definition 3.4. *For any pack $T = (P, a, b, r)$, the set of entries in T is defined to be the following product of finite sets:*

$$\textit{Entr(T)} := \prod_{p \in P} [r(p)] \text{ where } [r(p)] := \{1, 2, \ldots, r(p)\}$$

Definition 3.5. *For any pack $T = (P, a, b, r)$, the bounding box for T is defined to be the following product of semi-open intervals:*

$$\textit{BBox(T)} := \prod_{p \in P} [a_p, b_p)$$

This bounding of of T simply equals the set of all possible values the variables in the set P can have.

Example 3.6. *Consider Example 3.2 and the pack $T = (\{x, y, z\}, a.b.r)$. We can then define Entr(T) as the set of all $\{(e_1, e_2, e_3)\}$ such that $e_i \in \mathbb{N}$, $1 \leq e_i \leq 100$ for $i \in \{1, 2, 3\}$*

Similarly, one can define BBox(T) as the set of all $\{x_1, x_2, x_3\}$ such that $x_i \in [-1.1\ 1.1)$ for $i \in \{1, 2, 3\}$

Definition 3.7 (Boolean array)**.** *Let T be a pack with $P = \{p_1, p_2, \ldots, p_n\}$ and resolution $r : P \to \mathbb{N}_{\geq 2}$. A size-$T$ boolean array is a function A :Entr$(T) \to \{0, 1\}$. Given an array $A \in$Arr(T), where Arr(T) is the set of T-sized boolean arrays, and an entry $e \in$Entr(T), we define $A(e) \in \{0, 1\}$ as the value of A at e.*

Definition 3.7 explains how one could go about mapping a pack T to a boolean array $A \in \mathrm{Arr}(T)$ but still need to define a method that helps to decide whether $A(e) = 0$ or $A(e) = 1$ for any input $e \in \mathrm{Entr}(T)$. Before we can define such a method, we need a function that maps all values $b \in B$ to an entry $e \in \mathrm{Entr}(T)$. We will use the following function to define the preimage:

$$\mathrm{Pixel}(e) = \prod_{i=1}^{n} \big[a(p_i) + \delta(p_i)(e(i) - 1), a(p_i) + \delta(p_i)e(i)\big). \tag{1}$$

Where $P = \{p_1, p_2, \ldots, p_n\}$ and $\delta(p) = \frac{b(p)-a(p)}{r(p)}$. Each pixel is thus a half-open subcube of B and all pixels have the same size. Namely, if our pack has m variables, its ith coordinate is of size $\delta(i)$ where $1 \leq i \leq m$ and $i \in \mathbb{N}$.

Example 3.8. *Consider Example 3.2. We can calculate* $\delta = \frac{2.2}{100} = 0.022$*, and therefore* $Pixel(1, 1, 1) = \{x_1, x_2, x_3\}$ *such that* $x_i \in [-1.1 - 1.078)$ *for* $i \in \{1, 2, 3\}$

We will now start connecting functions to packs:

Definition 3.9 (Functionally defined relation)**.** *Assume* $T = (P, a, b, r)$ *is a pack with a bounding box* B *of* T*,* $f : B \to \mathbb{R}^k$ *is a function for some* $k \in \mathbb{N}$ *and* $S \subseteq \mathbb{R}^k$*. Then the functionally defined relation of* f, S *on* T *is a relation of the form*

$$R_{f,S} := \{x \in B \mid f(x) \in S\}$$

Where S *will be called the target subset and where the part to the right of the symbol* $\mid$ *indicates the condition that has to hold in order for the part to the left to be included in the set.*

Example 3.10. *Consider Example 3.2. In order to find the functionally defined relation* $R_{f,S}$ *on* $T_1 = (\{x, z\}, a, b, r)$*, we need to rewrite* f_1 *as* $g_1 = 0$ *where* $g_1 := x^2 - z$*. We can now define the target subset* S_1 *as* 0 *and we obtain the functionally defined relation* $R_{g_1,0} = \{x \in BBox(T_1) \mid g_1(x) = 0\}$*. Similarly, we let* $g_2 = 1 - y^2 - z$*,* $T_2 = (\{y, z\}, a, b, r)$ *and we define* $R_{g_2,0} = \{x \in BBox(T_2) \mid g_2(x) = 0\}$*.*

We would like to assign one point $x'(e)$ in $\mathrm{Pixel}(e)$ to e such that, when we calculate the result of $f(x'(e))$, we also gain some knowledge about the other values of f in $\mathrm{Pixel}(e)$. In order to find such a value, we will define an error measurement of a set N on a functionally defined relation.

Definition 3.11 (The N-error)**.** *Let* $R_{f,S} \subseteq B$ *be a functionally defined relation on a bounding box* B*. For any subset* $N \subseteq B$*, we define the* N*-error set of* $R_{f,S}$ *to be the set of distances*

$$D_f(N, S) := \{d(f(x), y) \in \mathbb{R}_{\geq 0} | x \in N, y \in S\}$$

for $l \geq 0$ *and* $u \geq 0$ *and where the part to the right of the symbol* $\mid$ *indicates the condition that has to hold in order for the part to the left to be included in the set. We say that the* N*-error of* $R_{f,S}$ *is always above* l *if* $D_f(N, S) \subseteq (l, \infty)$*. Similarly, the* N*-error of* $R_{f,S}$ *achieves* u *if* $D_f(N, S) \cap [0, u] \neq \emptyset$*. Finally, the* N*-error of* $R_{f,S}$ *is bounded by* u *if* $D_f(x, S) \cap [0, u] \neq \emptyset$ *for all* $x \in N$

Now that we have a measurement of the error, all we need is a threshold ϵ to compare it with. Such a threshold will need to satisfy the following property:

Definition 3.12 (Valid tolerance)**.** *Let $T = (P, a, b, r)$ be a pack, c_e the center of Pixel(e) and f : Pixel$(e) \to \mathbb{R}$ for all $e \in$Entries(T). A tolerance ϵ will be called valid if it satisfies the following condition:*

$$|f(c_e) - f(x)| < \epsilon \text{ for all } x \in \text{Pixel}(e) \tag{2}$$

Definition 3.13 (The ϵ-tolerance sample-in-center plot)**.** *Suppose given a pack $T = (P, a, b, r)$ with bounding box $B :=$BBox(T) and a functionally-defined relation $R_{f,S}$ on it. For each entry $e \in$Entr(T), let c_e denote the center of the corresponding Pixel(e). For any valid tolerance ϵ, we define the ϵ-tolerance sample-in-center plot to be the array $A :$Entr$(T) \to \{0, 1\}$ given by*

$$A(e) = \begin{cases} 1 \text{ if the} \{c_e\}\text{-error achieves } \epsilon \\ 0 \text{ otherwise} \end{cases}$$

This method will give us the following control over the error:

Definition 3.14 (ϵ-accurate plot)**.** *Let $T = (P, a, b, r)$ be a pack, R a functionally defined relation on it, and $\epsilon > 0$. We say that $A \in$Arr(T) is an ϵ-accurate plot of $R_{f,S}$ on T if the following hold for any entry $e \in$Entr(T)*

- *If the Pixel(e)-error of $R_{f,S}$ achieves 0 then $A(e) = 1$.*
- *If the Pixel(e)-error of $R_{f,S}$ is always above ϵ then $A(e) = 0$.*
- *If $A(e) = 1$, then the Pixel(e)-error of $R_{f,S}$ is bounded by 2ϵ.*

The proof of this error is straightforward and can be found in [5].

Because of Definition 3.13 and its implied error stated in 3.14, we are now able to create boolean array plots for any relation in our set R which have a bounded error and do not give any false negatives. Finding an ϵ that satisfies Equation (2) is not straightforward, and it gets harder if a function is not continuous or differentiable as we can not use the gradient in these cases.

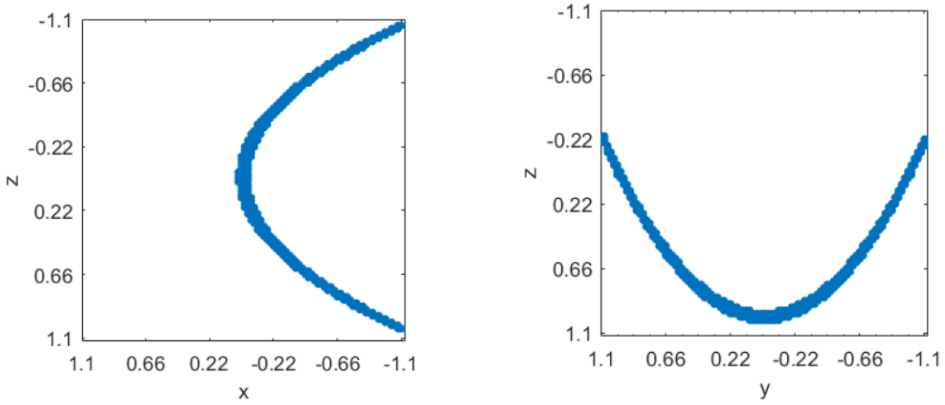
Example 3.15. *Consider Example 3.2 and the functionally defined relations in Example 3.10. Before we can start creating the boolean arrays, we need to find a valid tolerance ϵ, meaning that it satisfies 2. The gradient of g_1 is equal to $(2x, -1)$. By the definition of Pixel(e), we know that we are dealing with a box shaped interval of possibilities. We can also note that the derivative of g_1 with respect to z is constant. This means that we investigate the maximal change of the function that is caused by varying x and z separately and then add them.*

We will start with the change caused by varying x. The derivative of g_1 with respect to x gives us $2x$, which is a strictly increasing function. This, combined with the fact that both x^2 is a symmetric function, means that the function will change faster the higher the value of $|x|$. The interval is also symmetric, therefore it does not matter whether we consider the negative or positive values of x. We will consider the positive values of x. The size of

a pixel is equal to $\frac{1.1--1.1}{100} = \frac{2.2}{100} = \frac{1.1}{50}$. *Therefore, when calculating a value at the middle of a pixel, both* x *and* z *can vary by* $\frac{1.1}{50*2} = \frac{1.1}{100}$. *This means that the pixels in which the function varies most are the pixels with* x *interval* $[1.1 - \frac{1.1}{100}1.1)$. *We can now calculate the maximal change in* g_1 *that is caused by varying* x *by integrating the derivative with respect to* x *over this interval:* $\int_{1.1-\frac{1.1}{100}}^{1.1} 2x\ dx = [x^2]_{1.1-\frac{1.1}{100}}^{1.1} = (1.1)^2 - (1.1 - \frac{1.1}{100})^2 = 0.0241$ *rounded up to 4 decimals. Similarly, we can see that* z *is also strictly increasing. As it does not depend on* x, *we will just take the most positive increasing direction. This gives* $\int_{1.1-\frac{1.1}{100}}^{1.1} 1\ dx = 1.1 - (1.1 - \frac{1.1}{100}) = \frac{1.1}{100} = 0.011$. *As both functions are positively increasing, we can simply add both values and conclude that* g_1 *satisfies equation* (2) *when* ϵ *is chosen as* $0.0241 + 0.011 = 0.0351$.

We still need to find a threshold for $g_2 := 1 - y^2 - z$. *This gives the gradient* $(-2y, -1)$ *This is almost equal to the gradient of* g_1 *except the values are negative and* x *is replaced by* y. *However, all variables have the same intervals and because of symmetry, the same calculations can be made as for* g_1. *Hence, The value* 0.0351 *will also bound the change of* g_2 *in a pixel with respect to its center. We can now initiate the pixel array method with threshold* $\epsilon = 0.0351$.

Doing so gives two boolean arrays, one for each variable. A visualization of these arrays is given in Figure 1. Where we adjust the axis to the intervals that correspond to Pixel(e) *for each entry* e. *The white points correspond to zeros in the array and blue points to ones. The MATLAB code that was used can be found in Appendix A.*



(A) The boolean array of $f_1 : x^2 = z$ (B) The boolean array of $f_2 : 1 - y^2 = z$

FIGURE 1: The boolean arrays of g_1 and g_2

3.2 The wiring diagram and generalized array multiplication

The next step of the pixel array method is to combine all of the boolean arrays into a new boolean array, where the entries correspond to the pixels of the exposed variables. In order to accomplish this, we define a function that combines several packs into one.

Definition 3.16 (Wiring diagram)**.** *Let* $T_1, T_2, \ldots, T_n, T'$ *be packs and define a new pack* $\bar{T}$, *where* $\bar{P} = P_1 \cup P_2 \cup \cdots \cup P_n \cup P'$ *(*$\cup$ *is the disjoint union) with induced bounds* $a, b : \bar{T} \to \mathbb{R}$

and resolution function $r : \bar{T} \to \mathbb{N}_{\geq 2}$. *We define a wiring diagram* $\Phi : T_1, T_2, \ldots, T_n \to T'$ *as a function that consists of a tuple* $(\Delta, \varphi, a_\Delta, b_\Delta, r_\Delta)$ *where*

- Δ *is a finite set of which we will call the elements links.*
- $\varphi : \bar{T} \to \Delta$ *is a surjective function such that* $T_1 \cup T_2 \cup \cdots \cup T_n \to \Delta$ *is also surjective.*
- $a_\Delta, b_\Delta : \Delta \to \mathbb{R}$ *are functions such that* $a_\Delta \circ \varphi = a$ *and* $b_\Delta \circ \varphi = b$.
- $r_\Delta : \Delta \to \mathbb{N}_{\geq 2}$ *is a function such that* $r_\Delta \circ \varphi = r$.

Every wiring diagram $\Phi : T_1, T_2, \ldots, T_n \to T'$ is related to an array multiplication formula. That is, a function in the form of

$$\mathrm{Arr}(\Phi) : \mathrm{Arr}(T_1) \times \mathrm{Arr}(T_2) \times \cdots \times \mathrm{Arr}(T_n) \to \mathrm{Arr}(T')$$

This generalized array multiplication formula can be seen in Equation 3:

$$A'(e') = \sum_{e \in \mathrm{Entr}(\Delta) |\ \mathrm{Entr}'_\Phi(e) = e'} \prod_{i=1}^{n} A_i(\mathrm{Entr}^i_\Phi(e)). \tag{3}$$

Where $\mathrm{Entr}^i_\Phi(e) : \mathrm{Entr}(\Delta) \to \mathrm{Entr}(T_i)$ and $\mathrm{Entr}'_\Phi(e) : \mathrm{Entr}(\Delta) \to \mathrm{Entr}(T')$ and where all arrays are boolean arrays, hence we will also use the boolean multiplication and addition. As we only have two elements in our boolean system, we can define both operations by two 2×2 tables:

Table 1: Boolean addition.

	0	1
0	0	1
1	1	1

Table 2: Boolean Multiplication.

	0	1
0	0	0
1	0	1

The array multiplication formula may look quite complicated, but it is easy to interpret. If we see Δ as the set of all variables used in the relations and all sets P_i of packs T_i as sets of variables used in a relation R_i, then the generalized matrix formula searches for all inputs in $\mathrm{Entr}(\Delta)$ that are mapped to e' by $\mathrm{Entr}'_\Phi(e)$. Here follows an example of how this mapping work:

Example 3.17. *say* $e = (a, b, c) \in \Delta$, *where* $a \in r(p_1), b \in r(p_2), c \in r(p_3)$ *and* $P' = (p_1, p_3)$. *Then* $Entr'_\Phi(e) = (a, c)$.

After we found such an entry, we check the values of all other boolean matrices in the points that correspond to their variables, and then we multiply these values. Therefore, because of the boolean multiplication rules, $A'(e')$ will only be 1 if all $A_i(\mathrm{Entr}^i_\Phi(e))$ are equal to 1. And 0 if they are not.

When a boolean array A' is constructed in this way, it will give the solution set of the exposed variables with an error that is bounded by 2ϵ, where $\epsilon = \max(\epsilon_1, \epsilon_2, \ldots, \epsilon_n)$ and ϵ_i is the threshold corresponding to A_i. This array does not have any false negatives when all ϵ_i are chosen such that equation 2 holds.

Example 3.18. *Consider Example 3.2 and the packs we defined in Example 3.15. The visual representation of its wiring diagram can be seen in Figure 2. Where P_1 and P_2 refer to the sets of packs T_1 and T_2 respectively and the edges that connect them indicate which variables they have in common. The exposed variables will be connected to the outer ring, which represents the final pack after the array multiplication is finished.*

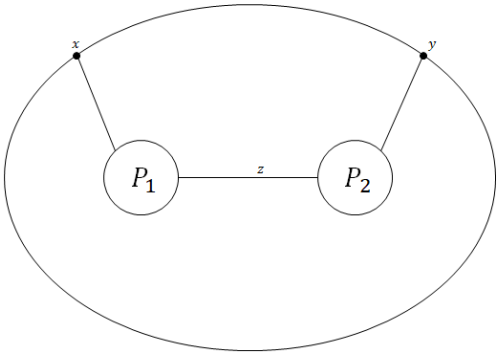
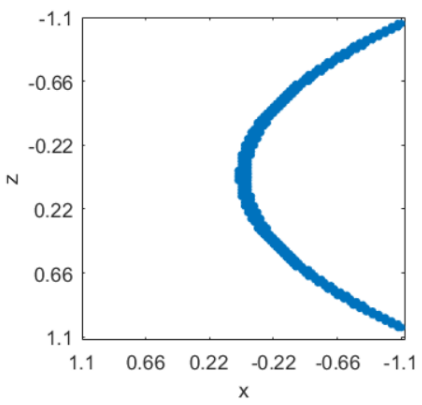
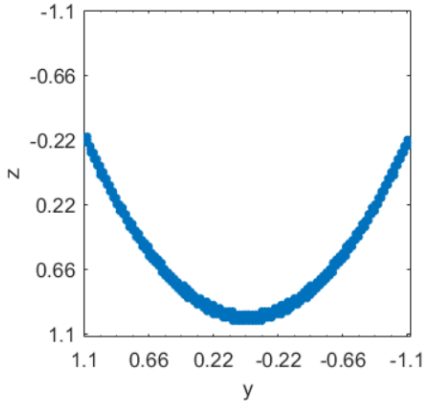


FIGURE 2: The wiring diagram of Example 3.18

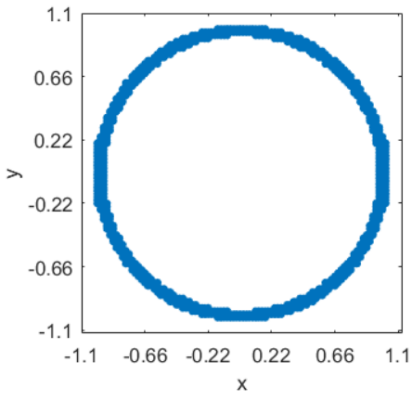
Example 3.19. *Consider Example 3.18. Using this wiring diagram, we can use the generalized array multiplication algorithm to combine both boolean arrays from Example 3.15 and create a boolean array that corresponds to the variables we want to expose. The result of this can be seen in Figure 3c. This result seems to resemble the unit circle. This is the correct answer as $x^2 = z$ and $1 - y^2 = z$ implies $x^2 + y^2 = 1$ which is the formula of the unit circle. The MATLAB code that was used can be found in Appendix A.*



(A) The boolean array of $f_1 : x^2 = z$.



(B) The boolean array of $f_2 : 1 - y^2 = z$.



(C) The result after array multiplying.

FIGURE 3: The array multiplication that returns an answer to the problem in Example 3.2.

3.3 Clustering

Now that we have defined a wiring diagram and an array multiplication method in the last section, we are able to combine several boolean arrays to one boolean array which makes us able to solve the pixel array method problems. However, the generalized array multiplication algorithm requires a lot of calculations which will often take a long time. This is due to the fact that it calculates the solutions of all variables simultaneously. A way of speeding of the algorithm is by clustering.

Assume we have $\Delta = \{x_1, x_2, \ldots, x_n\}$ and resolutions $r(x_1), r(x_2), \ldots, r(x_n) \geq 2$. The generalized array multiplication method will consider all entries in Δ. Therefore, our cost will be $r_1 \cdot r_2 \cdot \ldots \cdot r_n$ (in this case, $\cdot$ means standard multiplication). Hence, the cost of the generalized array algorithm for any set L and $r : L \to \mathbb{N}_{\geq}2$ will be denoted by

$$r^L := \prod_{l \in L} r(l) \tag{4}$$

Clustering means that we are not calculating the result of the wiring diagram at once, but instead we split it up into several smaller wiring diagrams and only then calculate the

result.

We will now formally define a cluster and investigate when it is useful to include one or more in the calculation of $\mathrm{Arr}(T')$.

Definition 3.20 (Cluster)**.** *Let $\Phi : T_1, T_2, \ldots, T_n \to T'$ be a wiring diagram. A cluster is choice of subset $C \subseteq \{1, 2, \ldots, n\}$. By symmetry, we assume $C = \{1, 2, \ldots, m\}$ for some $m \leq n$.*

Definition 3.21 (interior and exterior links/diagrams)**.** *Let $C = \{1, 2, \ldots, m\}$ be a cluster and let $\varphi : T_1 \cup T_2 \cup \cdots \cup T_n \cup T' \to \Delta$ be the partition as in Definition* 3.16*. Consider the images*

$$\Delta'_C := \varphi(T_1 \cup T_2 \cup \cdots \cup T_m) \text{ and } \Delta''_C := \varphi(T_{m+1} \cup T_{m+2} \cup \cdots \cup T_n \cup T'),$$

which we will call the sets of C-interior links and C-exterior links respectively. Let $Q_C = \Delta'_C \cap \Delta''_C$ be their intersection, we call Q_C the C-intermediate pack. Define

$$\Phi'_C : T_1, T_2 \ldots, T_m \to Q_C \text{ and } \Phi''_C : Q_C, T_{m+1}, T_{m+2} \ldots, T_n \to T'$$

to be the evident restrictions of φ with links Δ'_C and Δ''_C, respectively. We call Φ'_C the interior diagram and Φ''_C the exterior diagram, and refer to the pair (Φ'_C, Φ''_C) as the C-factorization of Φ.

Definition 3.22 (Trivial cluster)**.** *Using the same notation as the last two definitions. We say C is a trivial cluster if either $\Delta'_C = Q_C$ or $\Delta''_C = Q_C$. We refer to $L' := \Delta'_C - Q_C$ as the set of properly internal links and $L'' := \Delta''_C - Q_C$ as the set of properly external links in the C-factorization. Hence C is trivial if and only if $L' = \emptyset$ or $L'' = \emptyset$. If C is not trivial, it is a nontrivial cluster.*

Using the same notation as in the definitions, we conclude that the cost of a wiring diagram clustered by a nontrivial cluster C is always smaller or equal than the cost of the wiring diagram that is not clustered. This is because we get the following equation when we subtract the costs:

$$r^Q(r^{L'+L''} - (r^{L'} + r^{L''})) \geq 0 \tag{5}$$

We derive this formula by noting that $\Delta = Q + L + L''$,$\Delta' = L' + Q$ and $\Delta'' = L'' + Q$ and using this in Equation (4).

Example 3.23. *We will give an example of the clustering of a wiring diagram. Consider the unclustered wiring diagram in Figure 4 (we use the same notation as in Example 3.18).*

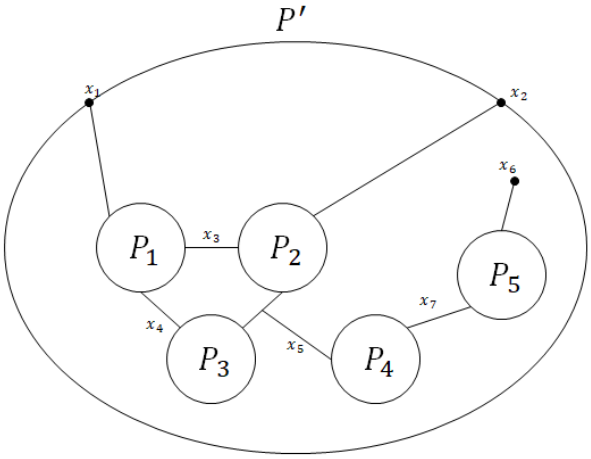


FIGURE 4: An unclustered wiring diagram.

This is a wiring diagram that would be much faster to calculate the boolean array of when clustered. Because of Equation (5)*, we know that any non-trivial cluster will speed of the process. Therefore, we will create a random grouping of packs which are represented by the colored dotted ellipses around them. The clustering is defined by treating the dotted ellipse and its interior as a wiring diagram by itself and calculating its result. This will give the same result as the unclustered diagram, but it requires much less calculations.*

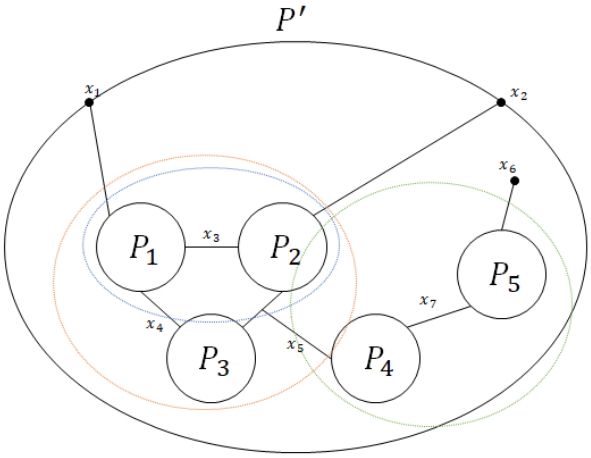


FIGURE 5: A clustered wiring diagram.

4 Application to differential equations

In this section, we are going to apply the pixel array method to differential equations to investigate the possibility of finding steady state solutions with this method. We will start by slightly adjusting the method such that it will be suitable for this purpose, this new method will be called the PASS method[2]. We will first analyze some easy differential equations and proceed to more difficult ones later on.

4.1 Modifying the pixel array method

One can directly apply the pixel array method to any discretized partial differential equation to find its steady state solutions. All the theory explained in Section 3 will still hold for this case. However, when solving the steady state of a differential equation, the values of the solution are often important as well, not just the existence of one. Therefore we will modify our pixel array method such that it will return all values that solve the problem.

Our solution set will no longer be a boolean array. Instead, we will change it to an array of tuples of local solutions. Hence, our array entries will look like

$$\{(x_{11}, x_{12}, \ldots, x_{1n}), (x_{21}, x_{22}, \ldots, x_{2n}), \ldots, (x_{m1}, x_{m2}, \ldots, x_{mn})\}$$

when we have m solutions for a relation that requires n variables. In order to use this solution set. We also need to alter our addition and multiplication rules. We define multiplication of our solution set in the following way:

$$\{x_1, x_2, \ldots, x_n\} \cdot \{y_1, y_2, \ldots, y_m\} = \{(x_1, y_1), (x_1, y_2), \ldots, (x_1, y_m), (x_2, y_1), (x_2, y_2), \ldots, (x_n, y_m)\}.$$

And the addition operation of our solution set will be defined as:

$$\{x_1, x_2, \ldots, x_n\} + \{y_1, y_2, \ldots, y_m\} = \{x_1, x_2, \ldots, x_n, y'_1, y'_2, \ldots, y'_k\}.$$

Where $y'_1, y'_2, \ldots, y'_k$ are elements y_i from the set $\{y_1, y_2, \ldots, y_m\}$ for which holds that $y_i \neq x_j$ for all $j \in \mathbb{N}$ and $1 \leq j \leq n$. This modified pixel array method will be called the PASS method.

4.2 The heat equation

The first differential equation we will test the PASS method on will be the heat equation. This equation is one in the form of:

$$u_t = D(x)u_{xx} \tag{6}$$

Where u_t denotes the derivative of u with respect to time and u_x the derivative of u with respect to its location x. Hence, u_{xx} is the second order derivative with respect to its location.

We start by solving the steady states algebraically. A steady state implies that the derivative with respect to time is equal to zero. Therefore $u_t = 0$, which means $D(x)u_{xx} = 0 \implies u_{xx} = 0$. We will now integrate two times on both sides:

$$u = \int \int 0 \; dx \; dx = \int a \; dx = ax + b.$$

Where a, b are constant real numbers. The solutions for the steady states of the heat equation can therefore be represented by the set of all straight lines.

The heat equation can be interpreted as the temperature of a line on every point. Assume the line has length 1 and we have a temperature x_0 at the point 0 and a temperature x_1 at the point 1, hence the beginning and end of the line respectively. Then the equation has reached its steady state if all points x in between have a temperature y that satisfies the equation $y = x(x_1 - x_0) + x_0$.

Before we can use the PASS method, we need to discretize the problem. We will approximate u_{xx} by the following discretization:

$$\frac{u_{i-1} - 2u_i + u_{i+1}}{h^2}. \tag{7}$$

Where we have split our line into $\frac{1}{h}$ pieces and get the set of points $H := \{0, h, 2h, \ldots, 1\}$ and we define the value u_i to be the temperature on the point $i \in H$. In our steady state solution, u_{xx} is equal to zero. Therefore, we only have to consider

$$u_{i-1} - 2u_i + u_{i+1} = 0. \tag{8}$$

This is a function we can use the PASS method for.

In this section, we will be considering 7 partitions. This means that we will be considering 7 variables $u_0, u_1, \ldots, u_6$ and 5 relations in the form of Equation (8) that have to hold, our target set S will be 0 in this case. Every variable u_i for $i \in 0, 1, \ldots, 6$ will have the lower bound a, upper bound b and resolution r. The variables we expose are u_0, u_6 as they correspond to the boundary conditions of the differential equation.

The only input that is still needed for using the PASS method, is the threshold ϵ. We obtain this by studying the gradient of the relations, which are all in the form of Equation (8). We therefore obtain the gradient $(1, -2, 1)$, this gradient does not depend on any variable and therefore we can take an arbitrary pixel and calculate $\epsilon := \max|f(c) - f(x)|$ where c is the center of the pixel and x is any other coordinate inside the pixel. Starting at the center of a pixel, each variable can change at most $\delta = \frac{b-a}{2r}$ in either the positive or negative direction before leaving the pixel. We may therefore assume all variables move in a direction that increases the function value, this will happen with gradient $(1, 2, 1)$. Therefore, $\epsilon = (1 + 2 + 1)\delta = 4\delta$.

We now have all the information needed to run the PASS method. However, doing so with the generalized array algorithm will take a lot of time. Computing solutions with a relatively low resolution of 20 already takes several hours. We will solve this by clustering our wiring diagram. The visual representation of the wiring diagram we are dealing with can be seen in Figure 6.

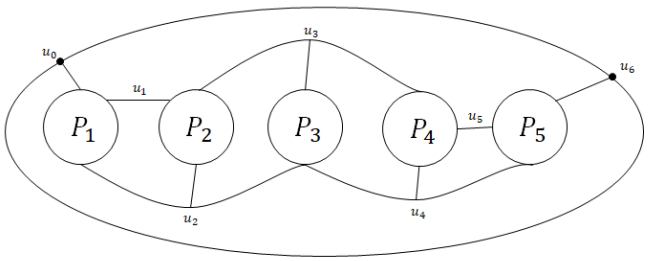


FIGURE 6: The wiring diagram without clustering.

We choose to cluster the diagram in groups of 2 as this will minimize the computational costs the most. The visual representation of this clustered wiring diagram can be seen in Figure 7. In this figure, the same symbols are used as in Example 3.23.

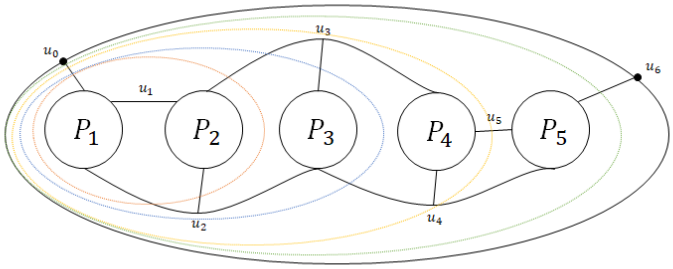


FIGURE 7: The clustered wiring diagram.

We will now calculate the results of this clustered wiring diagram for several resolution values by using the MATLAB code in Appendix B. Table 3 and Table 4 indicate whether a solution is found between certain boundary conditions for resolutions 5 and 20 respectively. A green filled rectangles implies such a solution does exist and a white filled rectangle implies it does not. As can be seen in the tables, a solution between any two boundary conditions always exists. This result confirms our algebraically found solution, which stated that every straight line between two boundary conditions is a steady state solution.

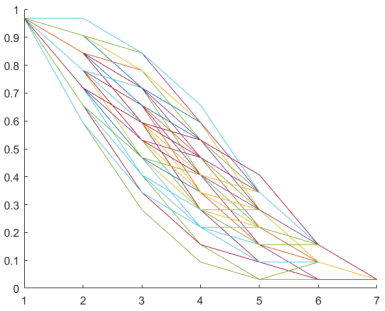
	0.1	0.3	0.5	0.7	0.9
0.1					
0.3					
0.5					
0.7					
0.9					

TABLE 3: The results for resolution 5.

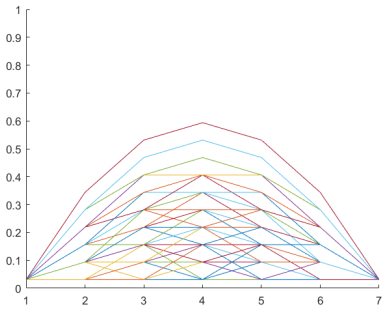
	0.0313	0.0938	0.1563	0.2188	0.2813	0.3438	0.4063	0.4688	0.5313	0.5938	0.6563	0.7188	0.7813	0.8438	0.9063	0.9688
0.0313																
0.0938																
0.1563																
0.2188																
0.2813																
0.3438																
0.4063																
0.4688																
0.5313																
0.5938																
0.6563																
0.7188																
0.7813																
0.8438																
0.9063																
0.9688																

TABLE 4: The results of resolution 20.

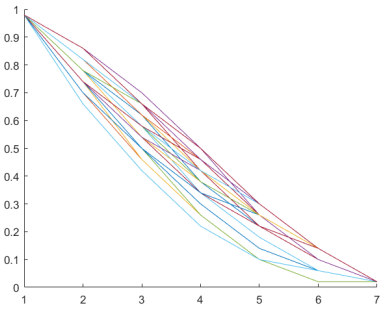
However, these tables do not give us any information whether the solutions are actually straight lines or not. That is where we need the modified solution set of the PASS method for. We use this solution set to plot all solution between 2 boundary points for several resolution values. These plots can be seen in Figure 8. The vertical axes represent the value of u_i for all i given by the horizontal axes. All the plots on the left will represent the solutions where u_0 has its value in the pixel that is closest to 1 and u_6 has its value in the pixel that is closest to 0.The plots on the right represent the solutions where both u_0 and u_6 have its value in the pixel that is closest to 0.



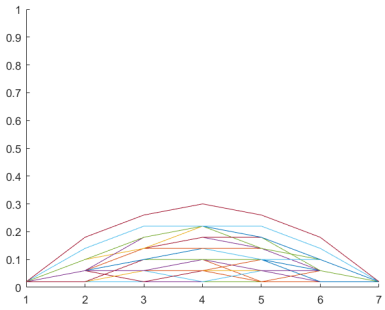
(A) resolution:16



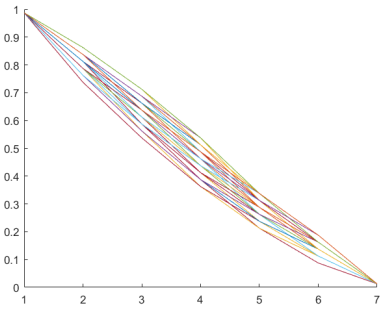
(B) resolution:16



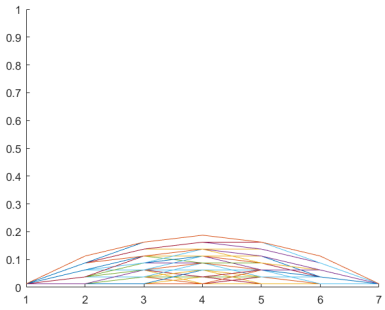
(C) resolution:25



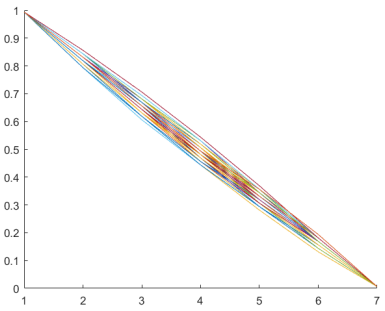
(D) resolution:25



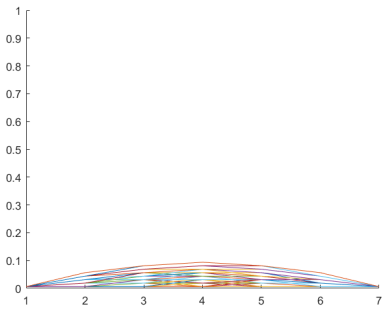
(E) resolution:40



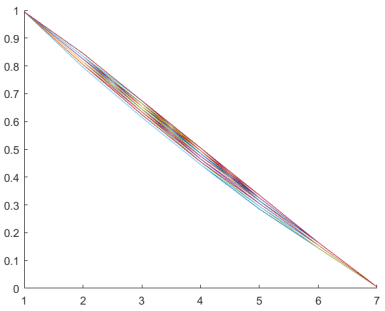
(F) resolution:40



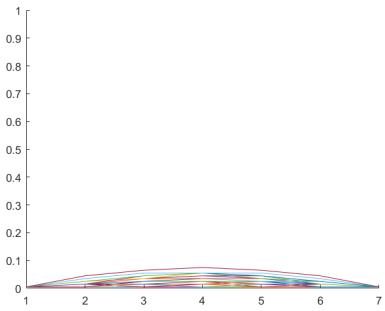
(G) resolution:80



(H) resolution:80



(I) resolution:100



(J) resolution:100

FIGURE 8: All solutions between two boundary conditions for several resolutions.

As we can see in Figure 8, the solutions converge more and more to a straight line as the resolution increases. This makes sense as we know that the error is bounded by 2ϵ and ϵ decreases as the resolution increases It should be noted though that the method only approximates the discrete approximation of the heat equation and not the actual heat equation, therefore the error can sometimes be larger than 2ϵ. However, the plots indicate that the PASS method correctly approximates the solutions of the actual heat equation as well.

4.3 The Fisher-KPP equation

In this section, we will introduce a differential equation that is a lot more difficult to find the solutions of, the Fisher-KPP Equation (9).

$$u_t = u_{xx} + \mu u(1-u) \tag{9}$$

According to [2], this function should only have steady state solutions for $u = 0$ and $u = 1$. We will use the PASS method to find all steady state solutions of the equation and then compare it to the actual solutions. The discrete version is given by Equation (10).

$$\frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} + \mu u_i(1-u_i) \tag{10}$$

For this problem, we use the same wiring diagram structure as we did for the heat equation. We find a valid tolerance ϵ by using similar calculations as for the heat equation and Example 3.15. The PASS method is used with uniform bounds and resolutions, the MATLAB code can be found in Appendix C. We start by using the following values: lower bound $a = 0$, upper bound $b = 3$, $\mu = 1$, $h = 1/6$ and resolution $r = 100$. This will give a tolerance of 216.0748. The results of the PASS method for these values are quite similar to the ones for the heat equation. We once again have a solution for all boundary conditions and there are hundreds of solutions in total. The correct solutions are also included, however, there are so many solutions that we do not consider this as an accurate result. Varying the variables did not change these results.

Changing the tolerance to the very low value of 0.6 did return only the right two solutions. However, this tolerance is way below the validity threshold and therefore the method loses all its mathematical foundation when using such a value.

An alternative way of finding only the right results is splitting up Equation (10) in its two terms and inserting these equations separately in the PASS method. However, when doing so, one neglects a lot of solutions. This simplification can only be made with strong mathematical foundation, which we do not have.

5 Conclusion

The pixel array method was often able to very precisely find all solutions within the given bounds. The method does not return any false negatives, which is a strong property, and the results are given in a format that can easily be visualized. However, the method has

a big flaw. Namely, the finding of a valid tolerance ϵ. It takes quite a bit of time to find a tolerance that is valid, and we saw during the approximation of the Fisher-KPP equation that it may very well be possible that the tolerance is simply too high to get any meaningful results. This creates the doubt that the method might not be that useful for complex functions, especially when they are not differentiable or continuous as sudden function changes will increase the valid tolerance threshold. In conclusion, the method is interesting and seems promising. However, further research should be done with respect to lowering the valid tolerances in order for the method to be applicable to complex problems.

References

[1] Brendan Fong and David I Spivak. Seven Sketches in Compositionality: An Invitation to Applied Category Theory. *arXiv e-prints*, page arXiv:1803.05316, Mar 2018.

[2] Cynthia T. Liu and David I. Spivak. Evaluating the Pixel Array Method as Applied to Partial Differential Equations. *arXiv e-prints*, page arXiv:1808.01724, Aug 2018.

[3] J. Lu and J. Andrian. Using pixel array method to optimize battery remaining useful life model. In *2018 IEEE Vehicle Power and Propulsion Conference (VPPC)*, pages 1–4, Aug 2018.

[4] David I. Spivak. The operad of wiring diagrams: formalizing a graphical language for databases, recursion, and plug-and-play circuits. *arXiv e-prints*, page arXiv:1305.0297, May 2013.

[5] David I. Spivak, Magdalen R. C. Dobson, Sapna Kumari, and Lawrence Wu. Pixel Arrays: A fast and elementary method for solving nonlinear systems. *arXiv e-prints*, page arXiv:1609.00061, Aug 2016.

A MATLAB code for example 3.18

A.1 The main file

```
f1 = @(x) x(2) - x(1)^2;
f2 = @(x) x(1) + x(2)^2 -1;
R = {f1,f2};
resolutions.lowrange = [-1.1 -1.1 -1.1];
resolutions.uprange = [1.1 1.1 1.1];
resolutions.res = [100 100 100];
exposevars = [1 3];
usedvars = [1 2; 2 3];
threshold = .0351;
answerarray = Pixelarray(R,resolutions,exposevars,usedvars,
    threshold);
spy(answerarray)
```

A.2 Initializing the pixel array method

```
function outputMat = Pixelarray(R,resolutions,exposevars,usedvars
    ,threshold)
M = cell(1,length(R));
```

```
for i = 1:length(R)
    resi.lowrange = resolutions.lowrange(usedvars(i,:));
    resi.uprange = resolutions.uprange(usedvars(i,:));
    resi.res = resolutions.res(usedvars(i,:));
    M{i} = Creatematrix(R{i},resi,threshold);
    spy(M{i})

end

outputMat = GAM(M, usedvars, resolutions, exposevars);


end
```

A.3 Creating the boolean arrays

```
function matrix = Creatematrix(f,resolutions,threshold)

lowrange = resolutions.lowrange;
uprange = resolutions.uprange;
res = resolutions.res;


steps = zeros(1,length(res));
for i = 1:length(res)
    steps(i) = (uprange(i) - lowrange(i))/res(i);

end


if length(res) == 1
matrix = zeros(1,res);
else
matrix = zeros(res);
end
values = ones(1,length(res));
inputs = lowrange;

for i = 1:numel(matrix)
    for j = 1:length(res)
        if values(j) == res(j)
            values(j) = 1;
            inputs(j) = lowrange(j) + 0.5*steps(j);
        elseif values(j) ~= res(j)
            values(j) = values(j) + 1;
            inputs(j) = lowrange(j) + (values(j)-0.5)*steps(j);
            break;
        end

    end
```

```
    matrix(i) = abs(f(inputs)) < threshold;

end
end
```

A.4 The generalized array multiplication algorithm

```
function multiplied = GAM(M, usedvars, ress, exposevars)
res = ress.res;
multiplied = zeros(res(exposevars));

testvar = 1;
delta = zeros(res);

values = ones(1,length(res)+1);
for e = 1:numel(delta)
    a = 1;
    for i = 1:length(M)
        Mi = M{i};
        indexx = values(usedvars(i,:));
        indexx3 = Arrayelements(Mi,indexx);
        a = a*Mi(indexx3);


    end
    for j = 1:length(res)
        if values(j) == res(j)
            values(j) = 1;
        elseif values(j) ~= res(j)
            values(j) = values(j) + 1;
            if j == 4
                testvar = testvar + 1
            end
            break;
        end
    end
    indexx2 = Arrayelements(multiplied, values(exposevars));
    multiplied(indexx2) = multiplied(indexx2) + a;
end

multiplied = logical(multiplied);




end
```

A.5 A function that is needed for filling the array

```
function realindex = Arrayelements(array, indexarray)
realindex = 1;
```

```
sizee = size(array);
for i = length(indexarray):-1:1
    realindex = realindex + (indexarray(i)-1)*prod(sizee(1:i-1));
end

end
```

B MATLAB code for the heat equation

B.1 The main file

```
R = {};
partitions = 7;
reso = 100;

for i = 1:partitions-2
    R{i} = @(x) x(1) - 2*x(2) + x(3);

end

resolutions.lowrange = zeros(1,partitions);
resolutions.uprange = 1*ones(1,partitions);
resolutions.res = reso*ones(1,partitions);
exposevars = [2 partitions-1];
usedvars = zeros(partitions,3);
for i = 1:partitions
    for j = 1:3
        usedvars(i,j) = i+j-1;
    end
end

delta = ((resolutions.uprange(1) - resolutions.lowrange(1))/
   resolutions.res(1));
threshold = 2*delta
answerarray = Pixelarray(R,resolutions,exposevars,usedvars,
   threshold);
```

B.2 Initializing the PASS method

```
function outputMat = Pixelarray(R,resolutions,exposevars,usedvars
   ,threshold)
M = cell(1,length(R));
i = 1;
resi.lowrange = resolutions.lowrange(usedvars(i,:));
resi.uprange = resolutions.uprange(usedvars(i,:));
resi.res = resolutions.res(usedvars(i,:));
M{1} = Creatematrix(R{i},resi,threshold,1);
```

```
M{2} = Creatematrix(R{i},resi,threshold,0);


for i = 3:length(M)-1
   M{i} = M{2};

end

M{end} = Creatematrix(R{i},resi,threshold,2);

 ress.lowrange = resolutions.lowrange([1 2 3 4]);
 ress.uprange = resolutions.uprange([1 2 3 4]);
 ress.res = resolutions.res([1 2 3 4]);
 M{2} = GAM(M([1,2]), [usedvars(1,:);usedvars(2,:)], ress, [1 3
     4]);



for i = 2:length(R)-2
 ress.lowrange = resolutions.lowrange([1 2 3 4]);
 ress.uprange = resolutions.uprange([1 2 3 4]);
 ress.res = resolutions.res([1 2 3 4]);
 M{i+1} = GAM(M([i,i+1]), [usedvars(1,:);usedvars(2,:)], ress, [1
      3 4]);
end


 ress.lowrange = resolutions.lowrange([1 2 3 4]);
 ress.uprange = resolutions.uprange([1 2 3 4]);
 ress.res = resolutions.res([1 2 3 4]);
 M{end} = GAM(M([end-1,end]), [usedvars(1,:);usedvars(2,:)], ress
     , [1 4]);



outputMat = M{end};


end
```

B.3 Creating the boolean arrays

```
function matrix = Creatematrix(f,resolutions,threshold,part)

lowrange = resolutions.lowrange;
uprange = resolutions.uprange;
res = resolutions.res;
```

```
6
7
8  steps = zeros(1,length(res));
9  for i = 1:length(res)
10     steps(i) = (uprange(i) - lowrange(i))/res(i);
11
12 end
13
14
15 if length(res) == 1
16 matrix = cell(1,res);
17 else
18 matrix = cell(res);
19 end
20 values = ones(1,length(res));
21 inputs = lowrange+0.5*steps;
22 values(1) = 0;
23
24 for i = 1:numel(matrix)
25         for j = 1:length(res)
26         if values(j) == res(j)
27             values(j) = 1;
28         elseif values(j) ~= res(j)
29             values(j) = values(j) + 1;
30             break;
31         end
32
33     end
34
35     for j = 1:length(inputs)
36           inputs(j) = lowrange(j) + (values(j)-0.5)*steps(j);
37     end
38
39
40
41     resultt = f(inputs);
42     if abs(resultt) <= threshold
43
44
45     if part == 1
46     matrix{i} = {[ inputs(1) inputs(2)]};
47     elseif part == 2
48
49     matrix{i} = {[ inputs(2) inputs(3)]};
50     else
51     matrix{i} = {inputs(2)};
52     end
53     end
54
```

```
end
end
```

B.4 The generalized array multiplication algorithm

```
function multiplied = GAM(M, usedvars, ress, exposevars)

res = ress.res;
if length(exposevars) > 1
multiplied = cell(res(exposevars));
else
multiplied = cell(1,res(exposevars));
end




testvar = 1;
delta = zeros(res);


values = ones(1,length(res));
values(1) = 0;


for e = 1:numel(delta)


    for j = 1:length(res)
        if values(j) == res(j)
            values(j) = 1;
        elseif values(j) ~= res(j)
            values(j) = values(j) + 1;
            if j == 4
                testvar = testvar + 1;
            end
            break;
        end
    end



    for i = 1:length(M)
        Mi = M{i};



```

```
44
45
46
47          indexx = values(usedvars(i,:));
48          indexx3 = Arrayelements(Mi,indexx);
49          if i == 1
50          a = Mi{indexx3};
51          else
52          a = Cellmultiplication(a,Mi{indexx3});
53          end
54
55
56      end
57
58      indexx2 = Arrayelements(multiplied,values(exposevars));
59      multiplied{indexx2} = Celladdition(multiplied{indexx2}, a);
60  end
61
62
63
64
65
66  end
```

B.5 A function that is needed for filling the array

```
1  function realindex = Arrayelements(array, indexarray)
2  realindex = 1;
3  sizee = size(array);
4  for i = length(indexarray):-1:1
5      realindex = realindex + (indexarray(i)-1)*prod(sizee(1:i-1));
6  end
7
8  end
```

B.6 Defining the tuple addition

```
1  function added = Celladdition(a,m)
2  if isempty(m)
3      added = a;
4  elseif isempty(a)
5      added = m;
6  else
7
8
9
10
11
12 addindex = [];
13
14
```

```
15
16 for j = 1:length(m)
17     for i = 1:length(a)
18         if length(a{i}) == length(m{j})
19             if sum(a{i} == m{j}) == length(a{i})
20                 break
21
22             end
23         end
24         if  i ==length(a)
25             addindex = [addindex j];
26         end
27
28     end
29 end
30
31
32 added = a;
33
34 for in = 1:length(addindex)
35     added{end+1} = m{addindex(in)};
36
37 end
38 end
```

B.7 Defining the tuple multiplication

```
1  function answercell = Cellmultiplication(a,m)
2  if isempty(a) || isempty(m)
3      answercell = {};
4  else
5
6  answercell = cell(1,length(a)*length(m));
7
8
9  ai = 0;
10 mi = 1;
11 for i = 1:length(answercell)
12     ai = ai + 1;
13
14     answercell{i} = [a{ai} m{mi}];
15
16     if mod(i,length(a)) == 0
17         ai = 0;
18         mi = mi + 1;
19     end
20
21
22 end
23 end
```

C MATLAB code for the Fisher-KPP equation

This uses the same functions as the heat equation, only the main function is different:

```
R = {};
partitions = 6;
mu = 1;
reso = 100;
a = 0;
b = 3;



h = (b-a)/partitions;

resolutions.lowrange = a*ones(1,partitions);
resolutions.uprange = b*ones(1,partitions);
resolutions.res = reso*ones(1,partitions);
exposevars = [2 partitions-1];
usedvars = zeros(partitions,3);


delta = ((resolutions.uprange(1) - resolutions.lowrange(1))/
    resolutions.res(1));


for i = 1:partitions-2
    R{i} = @(x)  (x(1) - 2*x(2) + x(3))/(h^2) + mu*x(2)*(1-x(2));

end



for i = 1:partitions
    for j = 1:3
        usedvars(i,j) = i+j-1;
    end
end
threshold = (delta/2)*(2/(h^2)) + abs((delta/2)*(-2/(h^2) + mu) -
    mu*(b^2-(b-delta/2)^2))
answerarray = Pixelarray(R, resolutions, exposevars, usedvars,
    threshold);
```