BACHELOR'S THESIS

Degree in Mathematics

Title: PATHS AND COMPATIBLE HAMILTONIAN CYCLES

Author: PORTA REGUÉ, ORIOL

Advisor: SILVEIRA, RODRIGO IGNACIO

Department: DEPARTAMENT DE MATEMÀTIQUES

Academic year: 2017

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat de Matemàtiques i Estadística

Universitat Politècnica de Catalunya

Facultat de Matemàtiques i Estadística

Degree in Mathematics

Bachelor's Degree Thesis

Paths and compatible Hamiltonian cycles

Oriol Porta Regué

Supervised by Rodrigo Ignacio Silveira

September 2017

Abstract

This project is focused on studying conditions under which a given set of points has a spanning path that is compatible with a *Hamiltonian cycle*. We prove that being *monotone* or *self-approaching* paths is enough condition to ensure there will be a compatible Hamiltonian cycle. Moreover, we study the condition of being a path that coincides with the MST of the set of points and prove some interesting results in order to help future research to prove that this is enough condition.

Resum

Aquest projecte està centrat en estudiar condicions que fan que un conjunt de punts tinguin un camí d'expansió que sigui compatible amb un *cicle Hamiltonià*. Hem demostrat que ser un camí *monòton* o *self-approaching* és condició suficient per asegurar que hi ha un cicle Hamiltonià compatible. A més, hem estudiat la condició de ser un camí que coincideix amb el MST del conjunt de punts i demostrat alguns resultats interessants per ajudar en futurs investigacions per demostrar que aquesta condició és suficient.

Resumen

Este proyecto está centrado en estudiar condiciones que hacen que un conjunto de puntos tengan un camino de expansión que sea compatible con un *ciclo Hamiltoniano*. Hemos probado que ser un camino *monótono* o *self-approaching* es condición suficiente para asegurar que hay un ciclo Hamiltoniano compatible. Además, hemos estudiado la condición de ser un camino que coincida con el MST del conjunto de puntos y probado algunos resultados para ayudar en futuras investigaciones para probar que ésta condición es suficiente.

Keywords: Monotone, self-approaching, MST, minimum spanning tree, Hamiltonian cycle, compatible, crossing.

Acknowledgements

To Rodrigo, for accepting to be my director and for all the help and dedicated hours even being on holidays.

To Farners, for all the support and help during all these months.

Contents

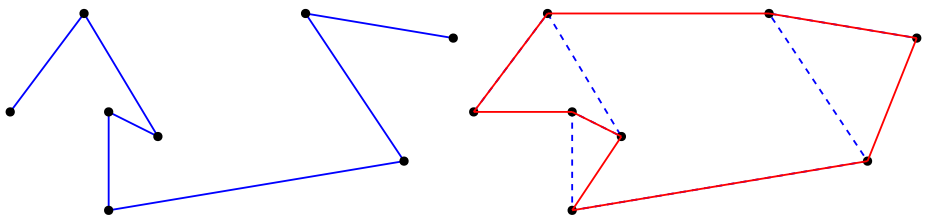
Chapter 1

Introduction

Imagine that you are the president of a country that wants to connect with trails all the cities using the cheapest way, now imagine that some business company wants to go through all the cities without visiting them twice and finally imagine that these trails can use the same paths but cannot intersect each other. In graph theory, your first problem is called finding the *minimum spanning tree* of a set of points (in your case cities), your second problem is one of the most important problems in this field, which is finding a *Hamiltonian path* of a set of points. The requirement that the paths cannot intersect is wanting them to be *compatible*.

This project is focused on studying conditions under which a given set of points has a spanning tree that is compatible with a Hamiltonian cycle. See Figure 1.1 for an example. To simplify this problem, instead of using a generic spanning tree, we have considered a spanning path, and have started studying the problem for some known types of paths such as the monotone paths, self-approaching paths and a path that coincides with an MST of the set of points; we call these paths *MST paths*.

FIGURE 1.1: The image on the left shows a given spanning path, and the image on the right shows the given path, using blue dashed edges, and a Hamiltonian cycle compatible with it, using red edges.



In this chapter we will introduce the basic concepts needed to understand the work done, a summary of the state of the art related with it and the way we will approach the above-mentioned problem.

A *graph* is a pair $G = (V, E)$ where V is a finite set of objects, which are called *vertices*, and E is a set of relations between pairs of vertices, which are called *edges*. If these relations have no orientation (i.e. the edge (x, y) is identical to the edge (y, x)) then the graph is called *undirected*. In this work, we will abuse of this term using it to refer to a *Euclidean undirected geometric graph*, which is a graph with the particularity that every vertex is associated to a point in the Euclidean plane and every edge is associated to a straight line segment. Placing it in an Euclidean space allows us to define the *weight* of an edge as its Euclidean length. We will also consider all vertices in general position, which means that there are no three vertices on the same straight line.

A graph is called *plane* if it has no intersection points (*crossings*) lying in the interior of its edges. Moreover, two plane graphs G_1 and G_2 are said to be *compatible* if their union, removing the duplicated edges, is also a plane graph. In this case, G_1 will be called G_2-*compatible* and G_2 will be called G_1-*compatible*.

Taking into consideration the above-mentioned concepts, a graph of the form $V = \{x_0, x_1, ..., x_k\}$ and $E = \{x_0x_1, x_1x_2, ..., x_{k-1}x_k\}$ where the x_i are all distinct is called *path*, and a path where the initial and final vertices are the same is called *cycle*. Moreover, a *Hamiltonian cycle* is a cycle that goes through all the vertices in a graph.

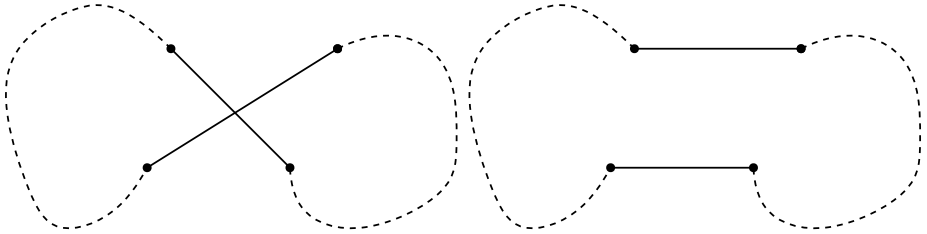
Another concept which is really close to that of a path is the concept of *tree*, which is a graph where any two vertices are connected by exactly one path (i.e., it has no cycles). Furthermore, a *spanning tree* is a tree which includes all of the vertices of G, and a spanning tree which has the minimum possible total edge weight will be called *minimum spanning tree* (*MST* for short).

Minimum spanning trees are a type of *proximity graph*. Moreover, an MST is included in a more generic type of graphs called *Relative Neighborhood Graphs (RNG)*, which are the graphs that connect two points p and q by an edge whenever there does not exist a third point r that is closer to both p and q than they are to each other. At the same time, these *RNG* are included inside the *Gabriel Graphs (GG)*, which are the graphs that connect two points p and q by an edge whenever the circle which has pq as the diameter is empty. Finally, the *GG*'s edges are a subset of the *Delaunay triangulation (DT)*'s edges, which is a triangulation in the plane such that no point is inside the circumcircle of any triangle in the triangulation. Note that MST are the most restrictive of the hierarchy, while *DT* are the least.

Our proofs are focused on finding an algorithm which, given a monotone, self-approaching or MST path P, creates a plane Hamiltonian cycle compatible with it. The algorithm studied is based on that of J. van Leeuwen and A.A. Schoone [32], which proves that a non-plane Hamiltonian cycle can be transformed to a plane one using $O(n^3)$ operations

called flips (Figure 1.2). This way to solve crossings maintains the connectivity of the graph and from now on we will use it as the way to solve intersections between two edges. As the reader will notice, the input to our problem is a path P that is Hamiltonian but not a cycle. To solve this, we will add an edge from the path's initial vertex to the final vertex, converting our Hamiltonian path to a Hamiltonian cycle which may not be plane.

FIGURE 1.2: Flip to remove an intersection from a cycle in the algorithm in [32]



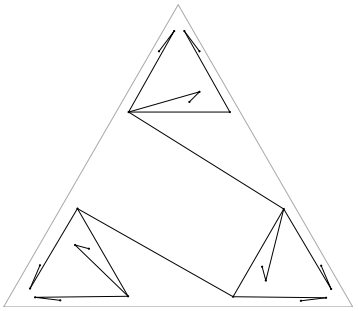
Our algorithm is divided in three main steps. Although we have proved that such algorithm is useful for the monotone paths and the self-approaching paths, we have found that the algorithm can generate non-compatible results for some RNG paths (and also for GG and DT paths), but we haven't found a counterexample for the MST paths. In the case of an MST, we have proved two of the three main steps of the algorithm not having been able to prove the third one, although we conjecture that it can also be proved. This step will be left as an open problem for future studies.

This problem belongs to a conjecture that has been present for a while, which says that given a set of points in the plane, there is always a spanning tree compatible with a Hamiltonian cycle. Very recently, the more generic conjecture which said that given a set of points in the plane, for all spanning trees, there is a compatible Hamiltonian cycle, was disproved by Mareke Van Garderen, Wouter Meulemans, Bettina Speckmann and Csaba Tóth [31] (Figure 1.3). Their counterexample uses extremely acute angles and shapes with an almost closed triangle with more points inside, which reduces considerably the amount of compatible Hamiltonian cycles that can be created. But, as the reader will note, these distributions are not present in a Gabriel graph and obviously neither in MST graphs nor the other types of paths that we will study in the next chapters. The fact that the example in Figure 1.3 uses edges that could never be MST is the motivation to consider MST edges in this work.

1.1 Previous related research

The topic of this work is closely related to three different topics which have been studied extensively: *augmentation problems*, *Hamiltonian problems* and *minimum spanning tree*

FIGURE 1.3: This image from [31] shows a schematic picture of a counterexample to the conjecture which says that given a set of points in the plane, for all spanning trees, there is a compatible Hamiltonian cycle. No Hamiltonian cycle compatible with the tree shown exists.



problems.

1.1.1 Augmentation problems

Given a graph $G = (V, E)$, we say that a second graph $G' = (V, E')$ obtained by adding (or replacing) a set of edges to G is an *(edge) augmentation* of G. The goal of this operation is to ensure that the augmented graph G' has some desired property. Note that in general we could augment a graph with both new vertices and edges, or even *subdivide* an edge (by replacing an edge with a path). To follow the state of the art of the augmentation problems we refer to the survey written by Ferran Hurtado and Csaba D. Tóth [23].

Another way to focus on compatibility could be by considering G as a set of disjoint segments. Then we may want to add new segments among the endpoints in order to obtain a crossing-free spanning tree with certain desirable properties. Alternatively, we may be given an arbitrary plane geometric graph G and we might want to add the minimum number of edges to increase its vertex or edge-connectivity, where the *vertex-connectivity* (resp. *edge-connectivity*) is the minimum number of vertices (resp. edges) whose deletion from a graph G disconnects G. In a third example, we may consider whether from any given plane spanning tree G we can construct an augmentation G' containing a Hamiltonian cycle or, when V is even, a perfect matching, which is a matching (i.e., an independent edge set) where every vertex of the graph is incident to exactly one edge of the matching.

The augmentation problem can be stated in general terms as: given a plane straight-line graph G and a property $\mathcal{P}$, the goal is to find an augmentation G' with property $\mathcal{P}$. In our case, the plane straight-line graph G is a plane MST, monotone or self-approaching path and our goal is to find an augmentation G' that is a G-compatible plane Hamiltonian cycle.

1.1.2 Hamiltonian problems

The Hamiltonian problem is generally considered to be determining conditions under which a graph contains a spanning cycle. Named after Sir William Rowan Hamilton (and his Icosian game), this problem traces its origins to the 1850s. To follow the current state of the art of this problem there are many surveys that provide ample background of previous work [16–18].

Another related problem that has been present for a while is the problem of finding an efficient algorithm to determine whether a given graph G is Hamiltonian. However, it seems to be quite difficult and so no one has succeeded yet and it still is now considered one of the most popular NP-complete problems. Nevertheless, for certain (non-trivial) classes of restricted graphs, there are polynomial-time algorithms that solve it [3, 6, 7].

The best known example of a problem of this type is the *travelling salesman problem*, in which one is interested in finding a Hamiltonian cycle (or path) G of minimum possible length on a given set of points in the plane. It is well-known that this problem is NP-hard [28], and there are numerous results on approximating the optimal solution and on the properties of such solution. Including some heuristics to approximate it using non-plane graphs. See, e.g., [25, 32].

Another branch of this type of problems is focused on counting the number of Hamiltonian cycles given a set of n vertices. Over the past few years there have been some research, some of it still ongoing, trying to improve the upper and lower bounds on this number [5, 9].

Related to Hamiltonian cycles but fitting on another type of problems called matching problems, there is the *Disjoint compatible matching conjecture* which was proved as true a few years ago [24] and says: "Let S be a set of points in the plane in general position such that $|S|$ is divisible by 4. Then for every perfect matching M of S there is another perfect matching, N, of S such that no segment of M crossings a segment of N".

1.1.3 Minimum Spanning tree problem

The minimum spanning tree problem is one of the most typical and well-known problems of combinatorial optimization, and methods for its solution have played a central role in the design of computer algorithms [1, 12, 19, 22, 30]. It is a standard practice to refer to Kruskal [26] and Prim [29] as the sources of the problem and its first efficient solutions, even though both of these papers refer to Borůvka [8]. The problem has a lot of importance and popularity due to its efficient algorithms, which makes it practical to solve it for large graphs [4].

The main applications of this problem are in the design of computer and communication networks, power and leased-line telephone networks, wiring connections, links in a transportation network, piping in a flow network, etc. [10, 14, 15, 27, 29]. It often appears as a subproblem in a solution of another problem, and MST algorithms are used in several exact and approximation algorithms for the traveling salesman problem, the multiterminal flow problem, the matching problem, etc. [11, 13, 20, 21]

1.2 Contributions

This project studies conditions under which a set of points has a spanning path that is compatible with a Hamiltonian cycle and proves that being *monotone* or *self-approaching* are enough conditions. These proofs are based on an algorithm, which given a spanning path that satisfies one of these conditions finds a plane Hamiltonian path that is compatible with it. This algorithm tries to provide a way to prove that the condition of being a path that coincides with an *MST* of the set of points is also enough, which is only partially proved.

The project also provides an implementation in *C++* of a program that determines if an *MST* path is compatible with a Hamiltonian cycle obtained with our algorithm and an implementation in *Unity* (*C#*) of a program that gives an easy way to study if a spanning path is compatible with a Hamiltonian cycle. This latest implementation done in Unity is portable for different types of devices like Android mobiles and tablets or PCs.

1.3 Structure of the project

This project starts with an experimental chapter that gives some intuition on how to first approach the problem. The third chapter explains the defined algorithm and how it proves that a spanning path with certain conditions is compatible with a Hamiltonian cycle, and in the Chapters 4 and 5 we prove that being a monotone path or a self-approaching path are enough conditions. In the sixth chapter we partially prove that our algorithm gives a Hamiltonian cycle compatible with a given MST path to give a way to prove that they are compatible for future research. The seventh chapter presents the conclusions of this project and some directions for future research.

Chapter 2

Preliminary computational study about MST paths

Before trying to prove different conditions that determine whether a path is compatible with a Hamiltonian cycle or not, we have decided to write two programs to help us see if the condition of being an MST is enough to make sure there will always exist a compatible Hamiltonian cycle. Our programs are based on J. van Leeuwen and A.A. Schoone's algorithm [32] that we have mentioned in the introduction and try to give us a first intuition on whether the condition of being an MST path is enough.

To reach this first intuition we have created an automatic case generation program and a graphical application used to generate particular cases. Based on that input the program runs our algorithm and determines whether the final Hamiltonian cycle is compatible with the initial path. We should note that these programs have been created to serve as support for the project and they are not intended to be a final result, that's why they can be efficiently and graphically improved.

In this chapter we are going to provide further details about these programs and comment the obtained results. The code can be found in the appendices of this work.

2.1 Automated case generator

This program is written in C++ and is in charge of generating a huge amount of cases. Every case creates an MST path, applies the algorithm of J. van Leeuwen and A.A. Schoone [32] and checks if the resulting Hamiltonian plane cycle is compatible with the initial MST path.

To execute this program we will need to specify certain parameters like the amount of cases that we want to generate and how many points do we want on it. The results of this program are saved in a folder with subfolders for every case that has already been generated, which are named after the case number and, in case the resulting Hamiltonian cycle is not compatible, they are named *#case_INTERSECTING*. Before running the program, it checks the last case generated and starts counting from it. In every iteration, it creates a new subfolder with three files containing the initial MST points *"input.dat"*, the edges of the initial MST path *"mst.dat"* and the edges of the resulting Hamiltonian cycle *"resulting_cycle.dat"*.

This has been done this way in order to run as many cases as we want even if the program stops in the middle of the execution and to easily see all the non-compatible cases filtering the subfolders by their name with the word *INTERSECTING*. To see the results graphically we recommend the program *gnuplot* (Figure 2.1) which you can use with the following commands:

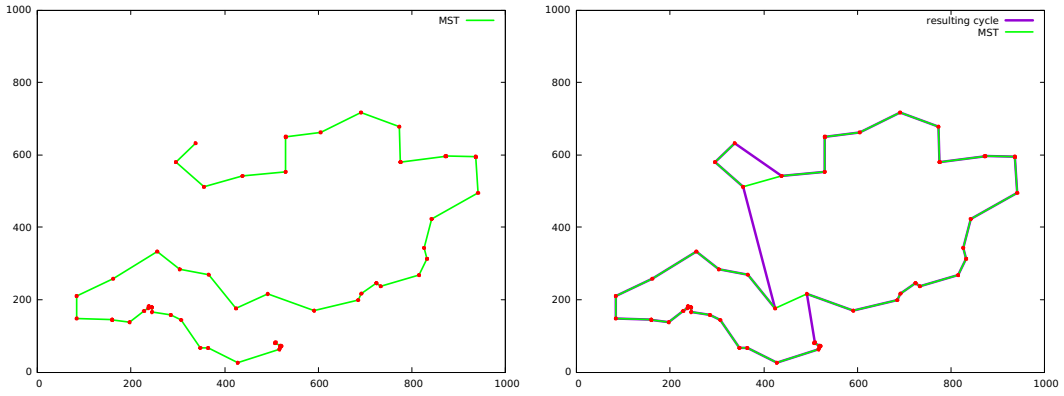
Set axis range: set xrange[0:1000] set yrange[0:1000]

Plot initial MST path: plot "mst.dat" u 1:2 with line lw 2 lt 3 lc rgb "green" title 'MST', "input.dat" u 1:2:(3) with circles fill solid lc rgb "red" notitle

Plot result: plot "resulting_cycle.dat" u 1:2 with line lw 3 title 'resulting cycle', "mst.dat" u 1:2 with line lw 2 lt 3 lc rgb "green" title 'MST', "input.dat" u 1:2:(3) with circles fill solid lc rgb "red" notitle

An example can be seen in Figure 2.1.

FIGURE 2.1: The image on the left shows the initial MST path and the image on the right shows the initial MST path in green and the resultant Hamiltonian cycle in purple.



After generating 400,000 cases with at most 200 points, every case resulted as compatible. This suggested that counterexamples, if any, would be hard to find in random inputs,

thus this has encouraged us to create another program to check specific cases to keep looking into this.

2.2 Graphical application

This application is focused on finding specific cases by testing handmade paths, mainly MST, that can be given the shape you want. It has been programmed with Unity, so it can be run either on a Windows computer or a mobile or tablet device.

FIGURE 2.2: The image on the left shows the blank canvas and the image on the right shows the canvas with the points added.

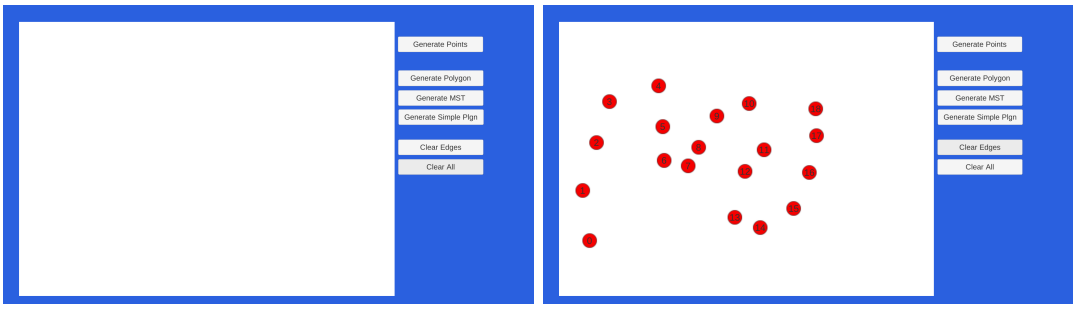
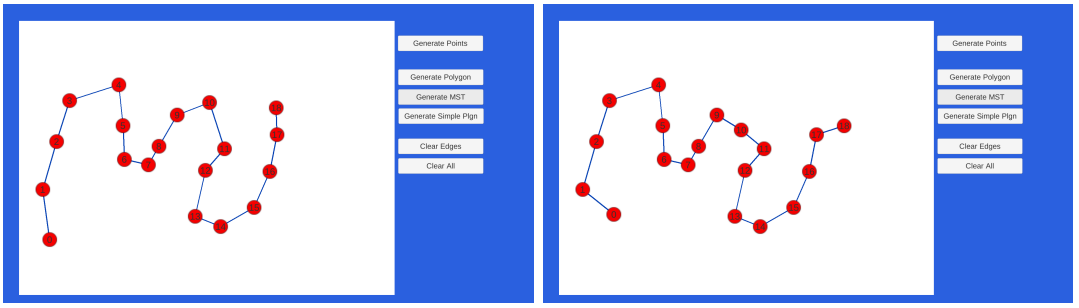
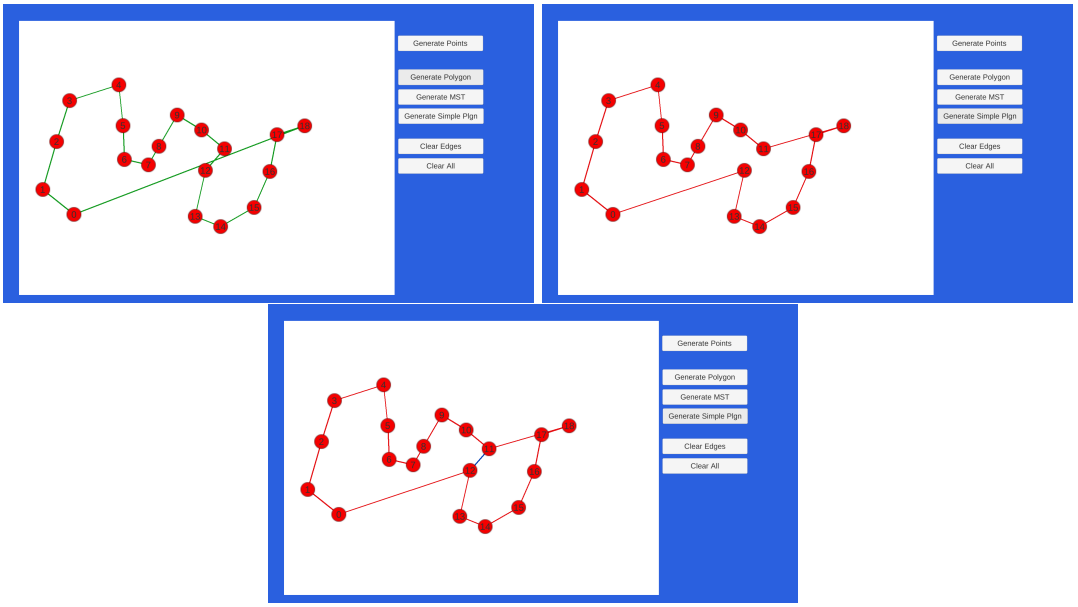


FIGURE 2.3: These images show the MST path and the possibility of moving edges.



This application works as follows: starting from a blank canvas, you can add the vertices that will form the initial path you want to investigate. This blank canvas is tactile and allows you to add the vertices by double clicking on it wherever you want them to be (Figure 2.2) or using the random vertices generator *"Generate Points"* button and move any already created vertex around the canvas (Figure 2.3). Once the vertices have already been added, we can see the initial cycle upon which the algorithm will be applied using the button *"Generate polygon"* (Figure 2.4). This cycle will be created following the order the vertices have been added in. You can also check if this cycle is formed by an MST path and an added edge between the initial and the final vertices by clicking on the button *"Generate MST"* and checking for any differences. Finally, you can apply the J. van Leeuwen and A.A. Schoone's algorithm [32] to the initial cycle using the button *"Generate Simple Plgn"* (Figure 2.4).

FIGURE 2.4: The image on the left shows the initial cycle and the image on the right shows the resultant Hamiltonian cycle. Finally, the image on the bottom shows in red the resultant Hamiltonian cycle and in blue the edges of the MST path that are not shared with the resultant Hamiltonian cycle.



In the tests done with this application using MST paths we haven't either been able to find any reasons to make us believe that being an MST is not enough to determine if a path has a compatible Hamiltonian cycle. That's why we decided to try to prove it.

Chapter 3

The Algorithm

J. van Leeuwen and A.A. Schoone [32] proved that a non-plane Hamiltonian cycle can be transformed to a plane one using $O(n^3)$ operations called *flips*, which consist on removing the crossing edges and replacing them with a new couple of edges that don't cross, maintaining the cycle unique. The interesting part of this algorithm is the fact that it works no matter the order used to solve the crossings. In our case, we will apply the algorithm by J. van Leeuwen and A.A. Schoone by using an initial Hamiltonian cycle composed of a plane path $P = (p_1, p_2, ..., p_n)$ and the edge p_1p_n. The resulting plane Hamiltonian cycle after applying the algorithm by Van Leeuwen and Schoone [32] may not be P-compatible. Our algorithm gives an order to solve the crossings, for which we can guarantee that the resulting Hamiltonian cycle is P-compatible for certain types of paths. In the following chapters we will prove it for monotone paths and self-approaching paths, and we will do a first attempt to prove it for MST-paths. In other words, we will prove the following theorems:

Theorem 4.1. *Given a monotone path P, there is always a Hamiltonian cycle that is P-compatible.*

Theorem 5.1. *Given a self-approaching path P, there is always a Hamiltonian cycle that is P-compatible.*

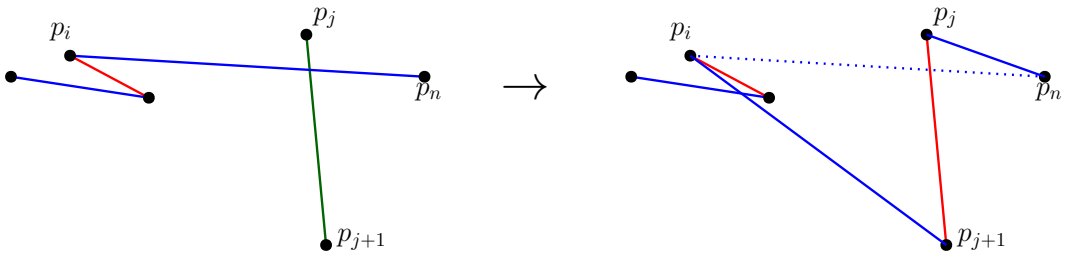
And we will do a first attempt to prove the following one:

Conjecture 6.1. *Given an MST path P, there is always a Hamiltonian cycle that is P-compatible.*

The algorithm is divided in three phases that will transform an initial non-plane Hamiltonian cycle, composed by a plane path $P = (p_1, p_2, ..., p_n)$ and an edge p_1p_n, to a plane Hamiltonian cycle. To do the explanation more readable, the edges from P will be referred to as *green* edges, the additional edges created by the algorithm will be referred

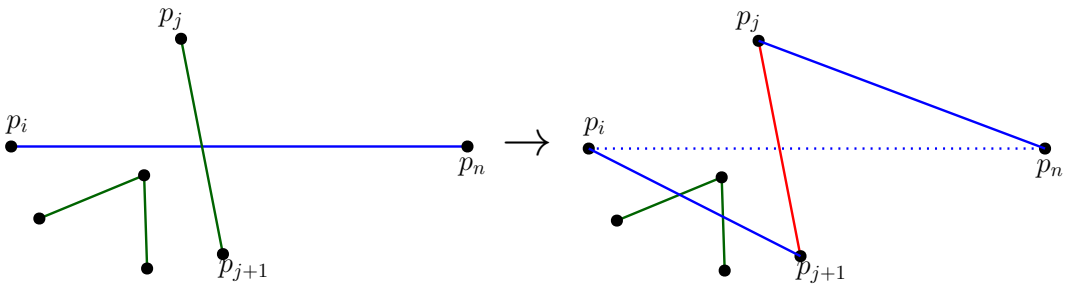
to as *blue* edges and the removed green ones will be referred to as *red* edges. Note that p_1p_n is the first blue edge.

FIGURE 3.1: These images show how a crossing between two blue edges is created after solving a crossing in the first phase.



The *first phase* consists in iteratively solving the furthest crossing from p_n in the blue edge that has p_n as end point and finishes once there are no crossings in the blue edge connected to p_n. On every iteration, the algorithm will remove the blue edge between a vertex p_i and p_n, will turn a green edge p_jp_{j+1} to a red one, and will create two new blue edges, which will be p_ip_{j+1} and p_jp_n, or p_ip_j and $p_{j+1}p_n$ (Figure 3.1 and 3.2). Every time the algorithm solves a crossing in this phase two types of crossings can appear in the new blue edge that doesn't connect with p_n: a crossing between two blue edges (Figure 3.1) and a crossing between a green edge and a blue one (Figure 3.2). The crossings from the first type will be solved in the second phase, and those from the second one will be solved in the third phase.

FIGURE 3.2: These images show how a crossing between a blue edge that is not connected with p_n and a green one is created after solving a crossing in the first phase.



The *second phase* solves all the crossings between blue edges that have been generated as a consequence of solving a previous crossing in the first or second phases, and does it in the same order they have been generated. On each iteration, the algorithm will remove two blue edges p_ip_j and p_kp_l and will create two blue edges p_ip_k and p_jp_l, or p_ip_l and p_jp_k (Figure 3.3). In case they are consecutive (i.e. one red edge shares endpoints with both edges), instead of creating two blue edges, the algorithm can create only one blue edge and turn the red edge connected with both edges to a green one (Figure 3.4). Every time the algorithm solves a crossing in this phase two types of crossings can appear in the new blue edges: a crossing between two blue edges and a crossing between a green

edge and a blue one. The crossings from the first type will be solved in the current phase, and those from the second one will be solved in the third phase.

FIGURE 3.3: These images show the case in which the algorithm generates two new blue edges. The image on the left shows the crossing before being solved and the image on the right shows the crossing after being solved. The dotted blue edges represent the removed ones.

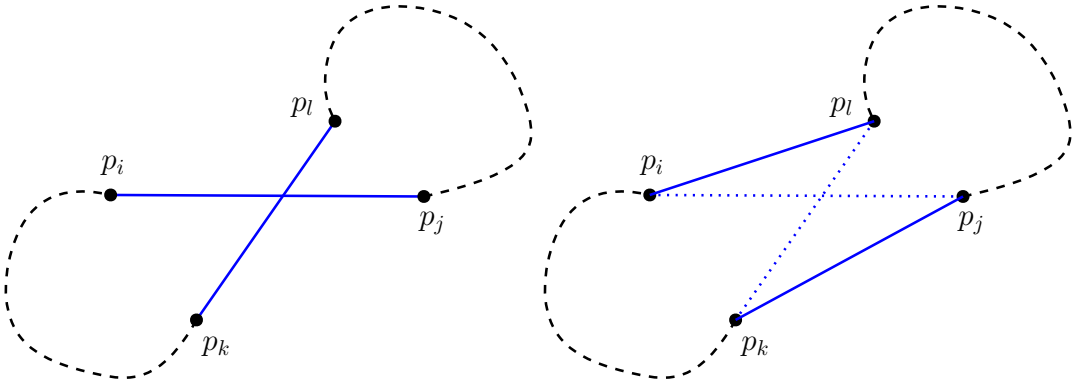
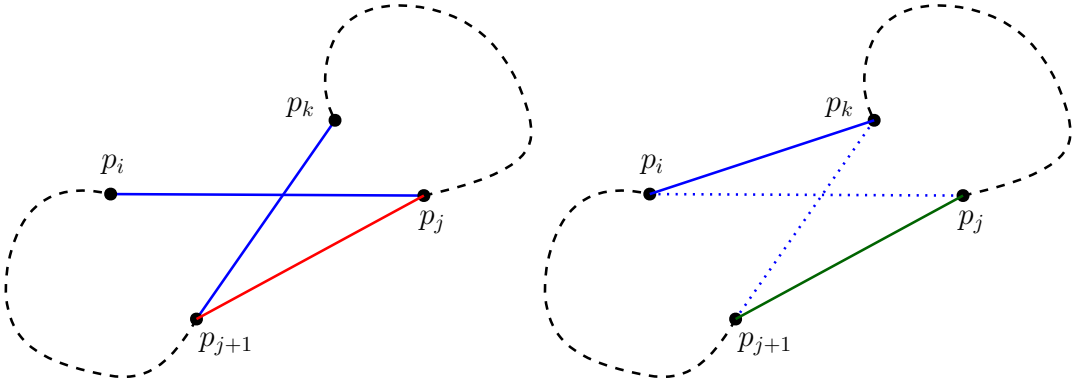


FIGURE 3.4: These images show the case in which the algorithm generates one new blue edge and turns a red edge to a green one. The image on the left shows the crossing before being solved and the image on the right shows the crossing after being solved. The dotted blue edges represent the removed ones.



Finally, the *third phase* removes all the crossings between green and blue edges generated by the first and the second phases applying the same procedure used in the first phase, and in the second one if needed, on every intersected blue edge. After this phase, there will be no crossings between two blue edges nor between a blue edge and a green one. In case the initial path is self-approaching, the order in which the crossings are solved doesn't matter. However, we haven't been able to prove whether the order matters or not in case the initial path is an MST path.

After applying the algorithm to a path P, we get a Hamiltonian cycle created by the blue edges and the green ones, and a set of red edges. These red edges represent the edges that have been removed from the initial path and the only ones that could not be compatible with the resulting Hamiltonian cycle. The green and red edges represent the initial path, so they have to be compatible. In conclusion, when the algorithm finishes,

if there are no intersections between red and blue edges, the resulting Hamiltonian cycle will be P-compatible.

Chapter 4

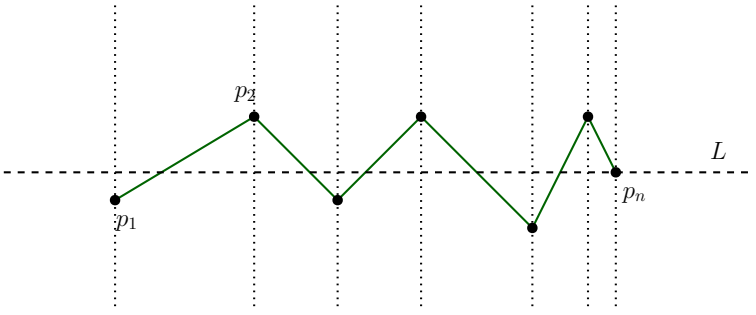
Monotone Path

In this chapter we are going to prove the following theorem by proving that the already mentioned algorithm transforms an initial monotone path P to a Hamiltonian cycle that is P-compatible:

Theorem 4.1. *Given a monotone path P, there is always a Hamiltonian cycle that is P-compatible.*

A path $P = p_1p_2...p_n$ is *monotone* if there is a straight line L such that every line perpendicular to L intersects the path at most once (Figure 4.1). Without loss of generality, we assume in this chapter that L is horizontal (i.e., P is a so-called x-monotone path).

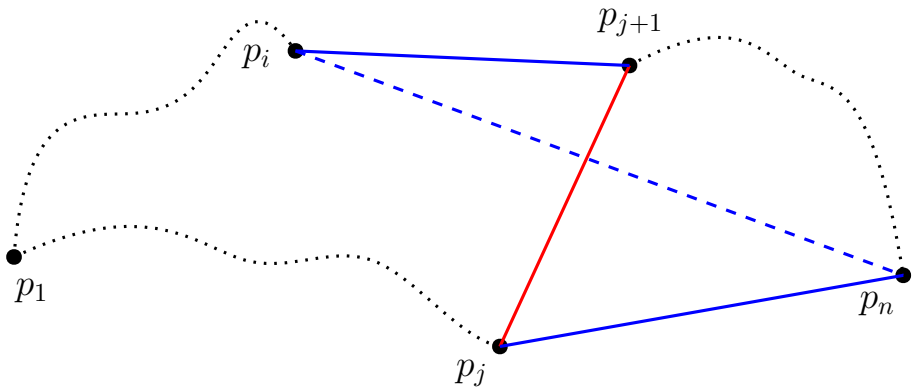
FIGURE 4.1: The image shows a monotone path and a straight line L, such that every line perpendicular to L intersects the path at most once.



First of all, note that given an initial monotone path $P = p_1p_2...p_n$, if a green edge p_jp_{j+1} intersects a blue edge p_ip_n then $i < j < j + 1 < n$. We can use it to prove the following lemma:

Lemma 4.2. *(Figure 4.2) Given a monotone path $P = p_1p_2...p_n$, the way the algorithm solves every crossing between a blue edge of the shape p_ip_n and a green one of the shape p_jp_{j+1} in the first phase consists in removing the blue edge, turning the green edge to a red one and creating two new blue edges p_ip_{j+1} and p_jp_n.*

FIGURE 4.2: The image shows the crossing after being solved. The dotted black curves represent the monotone subpaths that connect p_1 and p_n with the crossing edges. The dashed blue edge represents the removed one.



Note that in the generic case of the algorithm there is also the possibility to create the two new blue edges p_ip_j and $p_{j+1}p_n$ instead of p_ip_{j+1} and p_jp_n.

Proof. If we are solving this crossing, we know that p_i is connected to p_n through a subpath that contains p_j and p_{j+1}, in such order. So, removing the edges p_jp_{j+1} and p_ip_n, we will have two subpaths that are disconnected, one of them going from p_i to p_j and the other one going from p_{j+1} to p_n, and the only way to connect these paths subpaths again using two compatible edges is adding the blue edges p_ip_{j+1} and p_jp_n. □

Now, we are going to prove that this type of path is transformed to a P-compatible Hamiltonian path after applying the first phase of the algorithm. To do this we will prove the following lemma:

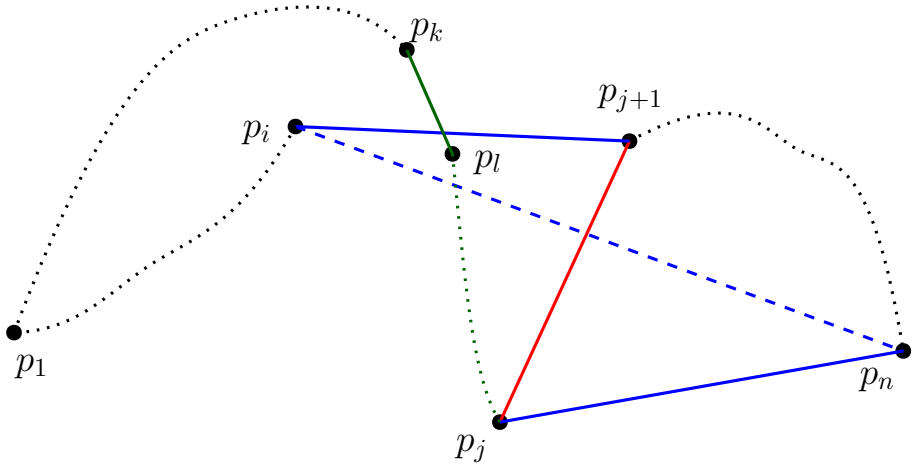
Lemma 4.3. *All intersections created during the first phase are placed on the blue edge p_ip_n of the current iteration.*

Proof. Let the green edge p_jp_{j+1} and the blue edge p_ip_n be the crossing that is going to be solved on the current iteration (i.e. the furthest crossing from p_n in the blue edge that has p_n as the endpoint), applying Lemma 3.2, the new blue edges will be p_ip_{j+1} and p_jp_n and we will only need to prove that p_ip_{j+1} doesn't intersect any green or blue edges, which we will do by contradiction.

Suppose there is an edge p_kp_l, where $k < l$, which intersects p_ip_{j+1} (Figure 4.3). Then, since $k < j$, the edge p_kp_l is in the monotone subpath that connects p_1 and p_j. Furthermore, to be an intersection, k must be less than j and l must be greater than i and less or equal than j. With these facts, we note that there must be a green monotone subpath from p_k to p_l that includes p_l and intersects with the blue edge p_ip_n, which contradicts

the fact that the solved green edge was the furthest crossing from p_n in the blue edge $p_i p_n$. In conclusion, $p_i p_{j+1}$ doesn't intersect any green or blue edge.

FIGURE 4.3: The image shows the crossing after being solved. The dotted black curves represents the monotone subpaths that connect p_1 and p_n with the crossing edges and the dotted green curve represent the monotone subpath that connects p_l and p_j. The dashed blue edge represents the removed one.



□

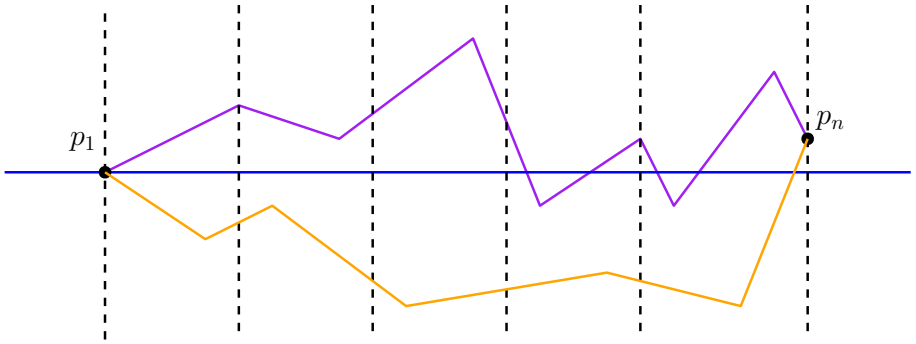
Now we know that the algorithm solves all crossings after the first phase, but we want to prove that the resulting plane Hamiltonian cycle is compatible with the initial monotone path $P = p_1 p_2 ... p_n$. To prove that, we can divide the edges from the resulting plane Hamiltonian cycle in order to create two monotone disjoint paths $Q = p_1 p_{i_1} p_{i_2} ... p_n$ and $R = p_1 p_{j_1} p_{j_2} ... p_n$ by the following way:

As all used points are from the initial monotone path, there is a straight line L such that every line perpendicular to L intersects the initial monotone path at most once. Using this and Lemma 4.2, it's immediate that every line perpendicular to L between p_1 and p_n crosses the resultant plane Hamiltonian cycle twice, so we can divide the edges between the ones that contain the uppermost crossings of these perpendicular lines and the other ones (Figure 4.4).

Now we can use the following lemma to prove that the resulting plane Hamiltonian cycle is compatible with the initial monotone path $P = p_1 p_2 ... p_n$:

Lemma 4.4. *Given two monotone disjoint and compatible paths $Q = q_1 q_2 ... q_m$ and $R = r_1 r_2 ... r_l$, where all points are different but the first and the last one (i.e. $q_1 = r_1$ and $q_m = r_l$), any monotone path P whose points are the union of the points of Q and R, will be compatible with Q and R.*

FIGURE 4.4: This image shows the two monotone disjoint paths created by the above-mentioned procedure.



Proof. $\forall p_i p_{i+1} \in P$ we have 4 possibilities depending on which monotone path they are in:

1. If $p_i, p_{i+1} \in Q$, then $p_i p_{i+1} \in Q$ and since Q is compatible with R, $p_i p_{i+1}$ is compatible with $Q \cup R$.

2. If $p_i, p_{i+1} \in R$, then $p_i p_{i+1} \in R$ and since R is compatible with Q, $p_i p_{i+1}$ is compatible with $Q \cup R$.

3. If $p_i \in Q$, $p_{i+1} \in R$, we can prove by contradiction that $p_i p_{i+1}$ is compatible with $Q \cup R$. Imagine there is an edge of the shape $p_j p_k \in Q$ that is not compatible with $p_i p_{i+1}$. Then, j must be less than i, and k must be greater than $i+1$, which is a contradiction with the fact that Q is monotone because the point $p_i \in Q$, the edge $p_j p_k \in Q$ and $j < i < k$. In case there is an edge of the shape $p_j p_k \in R$ the proof is analogous to the previous one. In conclusion, $p_i p_{i+1}$ is compatible with $Q \cup R$.

4. If $p_i \in R$, $p_{i+1} \in Q$, we can prove by contradiction that $p_i p_{i+1}$ is compatible with $Q \cup R$, and this prove is analogous to the previous one.

In conclusion, all the edges in P are compatible with Q and R. □

Chapter 5

Self-approaching Path

In this chapter, we are going to prove the following theorem proving that the already mentioned algorithm transforms any initial self-approaching path P to a Hamiltonian cycle that is P-compatible:

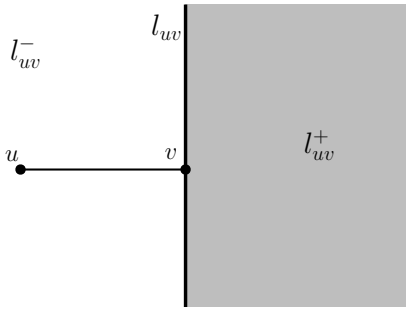
Theorem 5.1. *Given a self-approaching path P, there is always a Hamiltonian cycle that is P-compatible.*

A curve from s to t is *self-approaching* if for any three points a, b, c appearing in that order along the curve, the Euclidean distance between a and c is greater or equal than the distance between b and c.

Another way to define this type of path is as follows:

Lemma 5.2. *[2] A piecewise-smooth curve is self-approaching iff for each point a on the curve, the line perpendicular to the curve at a does not intersect the curve at a later point.*

FIGURE 5.1: Given two points u and v, let l_{uv} be the line that passes through v and is perpendicular to the line passing through u and v, let l^+_{uv} denote the closed half-plane not containing u with boundary l_{uv} and let l^-_{uv} be the complementary half-plane.

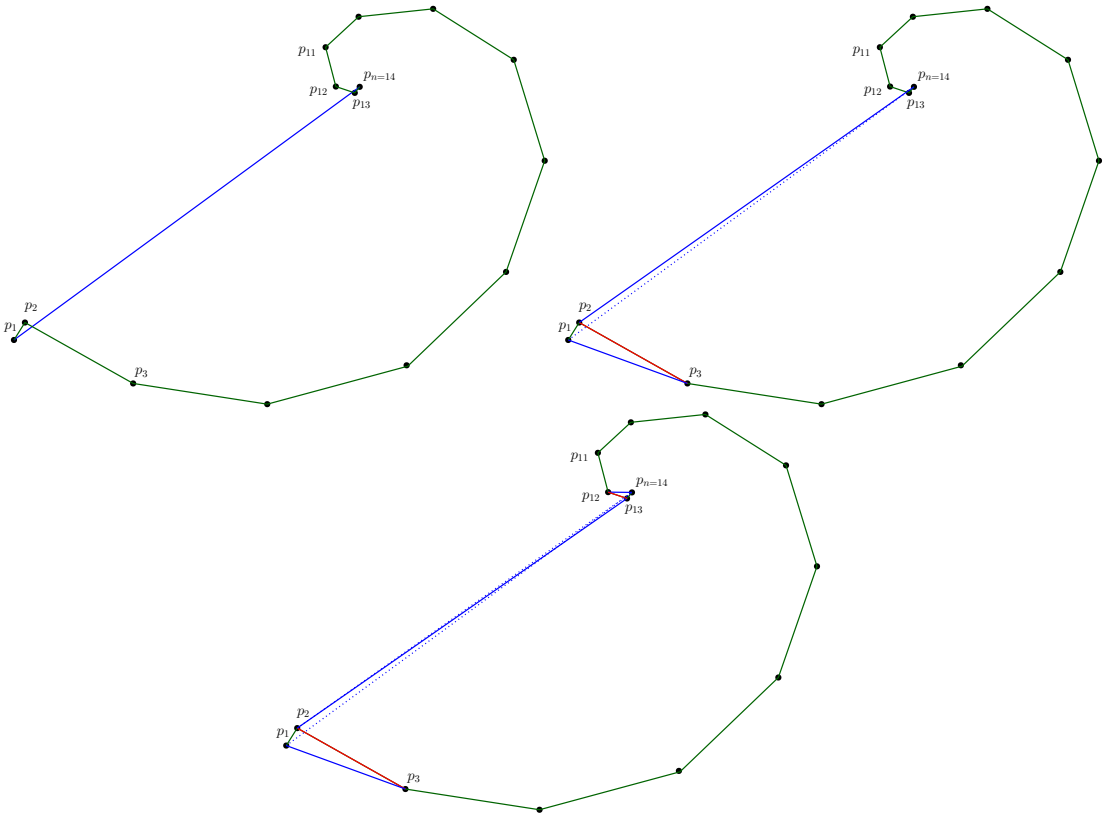


Given two points u and v, let l_{uv} be the line that passes through v and is perpendicular to the line passing through u and v, let l^+_{uv} denote the closed half-plane not containing u with boundary l_{uv} and let l^-_{uv} be the complementary half-plane (Figure 5.1). Using this notation, we can restate the lemma as follows:

Corollary 5.3. *[2] Let $P = (v_1, v_2, ..., v_n)$ be a directed path embedded in $\mathbb{R}^2$ via straight line segments. Then, P is self-approaching iff for all $1 < i < j \leq n$, the point v_j lies in $l^+_{v_{i-1}v_i}$.*

In the next sections we will prove the following lemmas, which show that the algorithm creates a plane Hamiltonian cycle (green and blue edges) that is compatible with all the red edges. Such resulting plane Hamiltonian cycle will be compatible with our initial self-approaching path (green and red edges) and the theorem will be proved.

FIGURE 5.2: The image on the top-left shows an initial self-approaching path in green and the p_1p_n edge in blue that creates a Hamiltonian cycle ($n = 14$) and the image on the top-right shows the Hamiltonian cycle after solving the first crossing. The image on the bottom shows the plane Hamiltonian cycle, built by blue and green edges, and red edges showcasing the edges that have been removed from the initial self-approaching path. We show the previously removed blue edges in a dotted edge.



Lemma 5.8. *The blue edges remaining at the end of phase 1 do not cross any red edges.*

Lemma 5.13. *The blue edges remaining at the end of phase 2 do not cross any red edges.*

Lemma 5.15. *The blue edges remaining at the end of phase 3 do not cross any red edges.*

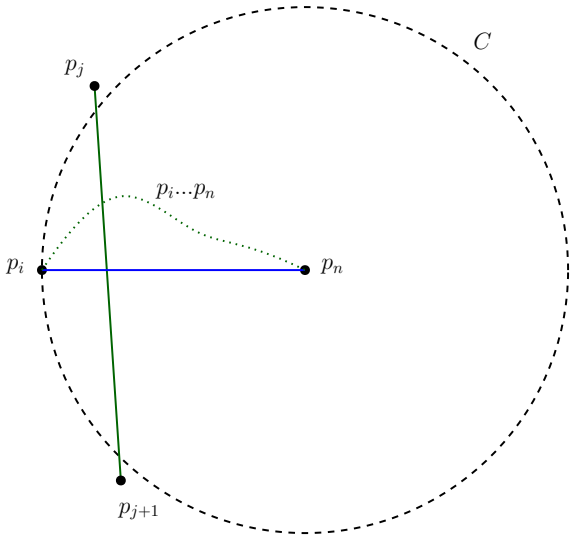
Note that having all blue edges compatible with the red edges is equivalent to being compatible with the initial self-approaching path.

5.1 Phase 1

First of all, we observe the following:

Lemma 5.4. *The only edges of the initial self-approaching path $P = (p_1, p_2, ..., p_n)$ that can cut the edge p_ip_n are edges of the shape p_jp_{j+1} where $i < j$.*

FIGURE 5.3: This image shows the construction used to prove Lemma 5.4.

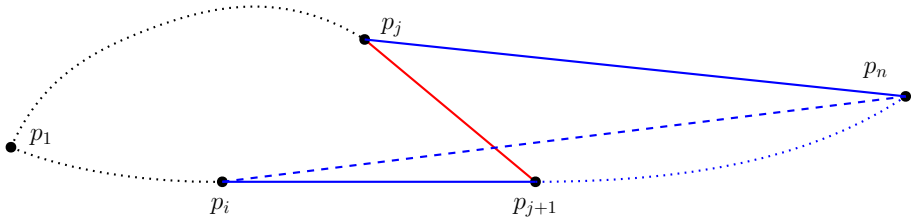


Proof. (Figure 5.3) Suppose that there is an edge p_jp_{j+1} of the initial self-approaching path P that cuts the edge p_ip_n and $j < i$. Given this, we can draw a circumference C with radius $dist(p_i, p_n)$ and center at p_n and, using the fact that $dist(p_j, p_n) > dist(p_{j+1}, p_n) > dist(p_i, p_n)$, given by the definition of self-approaching, we know that $p_j, p_{j+1} \notin C$ and the sub-path $p_i...p_n \in C$. If p_jp_{j+1} cuts p_ip_n then p_jp_{j+1} divides C in two parts C_1 and C_2 where $p_i \in C_1$ and $p_n \in C_2$. Note that in this situation, the sub-path $p_i...p_n$ cuts p_jp_{j+1}, which is a contradiction with the fact that P is plane. □

With this lemma we can prove the following corollaries:

Corollary 5.5. *Given a self-approaching path $P = p_1p_2...p_n$, the way the algorithm solves every crossing between a blue edge of the shape p_ip_n and a green one of the shape p_jp_{j+1} in the first phase consists in removing the blue edge, turning the green edge to a red one and creating two new blue edges p_ip_{j+1} and p_jp_n (Figure 5.4).*

FIGURE 5.4: The image shows the crossing after being solved. The dotted black curves represent the subpaths that connect p_1 and p_n with the crossing edges and the dashed blue edge represents the removed one.



Proof. If we are solving this crossing, we know that p_i is connected to p_n through a subpath that contains p_j and p_{j+1} in such order. So, removing the edges p_jp_{j+1} and p_ip_n, we will have two subpaths that are disconnected, one of them going from p_i to p_j and the other one going from p_{j+1} to p_n, and the only way to connect these subpaths again using two compatible edges is adding the blue edges p_ip_{j+1} and p_jp_n. □

Corollary 5.6. *Given $P = (p_1, p_2, ..., p_n)$ as the initial self-approaching path, after all the iterations in phase 1 the blue edges will be of the shape $p_1p_{i_1+1}$, $p_{i_1}p_{i_2+1}$, $p_{i_2}p_{i_3+1}$, ..., $p_{i_k}p_n$ and the red ones will be of the shape $p_{i_1}p_{i_1+1}$, $p_{i_2}p_{i_2+1}$, ..., $p_{i_k}p_{i_k+1}$ where $1 < i_1 < i_2 < i_3 < ... < i_k < n$ and k is the number of crossings solved.*

Proof. In the initial state of the algorithm, there is a blue edge p_1p_n and there aren't red edges. If there is a crossing between it and an edge of the shape $p_{i_1}p_{i_1+1}$, which is the furthest crossing from p_n, after the algorithm has solved it there will be two blue edges $p_1p_{i_1+1}$ and $p_{i_1}p_n$ and a red one $p_{i_1}p_{i_1+1}$.

In the r-th step of the algorithm, there are r blue edges $p_1p_{i_1+1}$, $p_{i_1}p_{i_2+1}$, $p_{i_2}p_{i_3+1}$, ..., $p_{i_r}p_n$ and $r-1$ red ones $p_{i_1}p_{i_1+1}$, $p_{i_2}p_{i_2+1}$, ..., $p_{i_r}p_{i_r+1}$ where $1 < i_1 < i_2 < i_3 < ... < i_r < n$. If there is a crossing between $p_{i_r}p_n$ and an edge of the shape $p_{i_{r+1}}p_{i_{r+1}+1}$, which is the furthest crossing from p_n, after the algorithm has solved it there will be $r+1$ blue edges $p_1p_{i_1+1}$, $p_{i_1}p_{i_2+1}$, $p_{i_2}p_{i_3+1}$, ..., $p_{i_r}p_{i_{r+1}+1}$, $p_{i_{r+1}}p_n$ and r red ones $p_{i_1}p_{i_1+1}$, $p_{i_2}p_{i_2+1}$, ..., $p_{i_r}p_{i_r+1}$, $p_{i_{r+1}}p_{i_{r+1}+1}$ where $1 < i_1 < i_2 < i_3 < ... < i_r < i_{r+1} < n$.

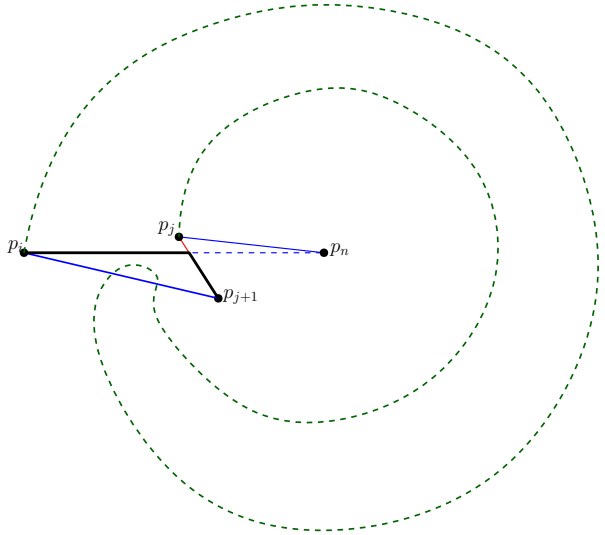
So after k crossings solved, the blue edges will be of the shape $p_1p_{i_1+1}$, $p_{i_1}p_{i_2+1}$, $p_{i_2}p_{i_3+1}$, ..., $p_{i_k}p_n$ and the red ones will be $p_{i_1}p_{i_1+1}$, $p_{i_2}p_{i_2+1}$, ..., $p_{i_k}p_{i_k+1}$ where $1 < i_1 < i_2 < i_3 < ... < i_k < n$. □

Moreover, we can say something about the new crossings created between green and blue edges that will be solved in the third phase:

Lemma 5.7. *Given $P = (p_1, p_2, ..., p_n)$ as the initial self-approaching path, after solving a crossing in phase 1 between two edges p_ip_n and p_jp_{j+1}, the new blue edge p_ip_{j+1} could*

only be cut by a self-approaching edge p_kp_{k+1} if $i < k < j$. If this happens, the subpath $p_ip_{i+1}...p_jp_{j+1}$ goes in and out of the area inside p_ip_n, p_jp_{j+1} and p_ip_{j+1} through the edge p_ip_{j+1}, one or more times (Figure 5.5).

FIGURE 5.5: The image shows the possible intersection between green and blue edges that will appear. In such graph, the black edges show edges that cannot be cut. The green dashed curves are not in a real scale.



Proof. Using the same reasoning behind Lemma 5.4 but changing p_n for p_{j+1}, it's immediate that k should be greater than i. Now, to prove that k should be smaller than j, we can draw a circumference C with radius $dist(p_{j+1}, p_n)$ and center at p_n and note that $p_j \notin C$, which implies that $p_i \notin C$ because $p_ip_n \cap p_jp_{j+1} \neq \emptyset$. In other words, $p_ip_{j+1} \notin C$ and the sub-path $p_{j+1}...p_n \in C$, which implies that these edges cannot intersect each other and k must be smaller than j.

To prove the second part just note that the phase 1 took the farthest cut in p_ip_n, so there will be no crossings with green edges between p_i and the intersection point between p_ip_n and p_jp_{j+1}. Furthermore, p_jp_{j+1} cannot intersect with any other green edge because they would both belong to the self-approaching path. □

Lemma 5.8. *The blue edges remaining at the end of phase 1 do not cross any red edges.*

Proof. Let $P = (p_1, p_2, ..., p_n)$ be the initial self-approaching path. Taking into account Corollary 5.6, the blue edges after all the iterations in phase 1 will be of the shape $p_1p_{i_1+1}$, $p_{i_1}p_{i_2+1}$, $p_{i_2}p_{i_3+1}$, ..., $p_{i_k}p_n$, and, on the other hand, all red edges after all the iterations in phase 1 will be of the shape $p_{i_1}p_{i_1+1}$, $p_{i_2}p_{i_2+1}$, $p_{i_3}p_{i_3+1}$, ...

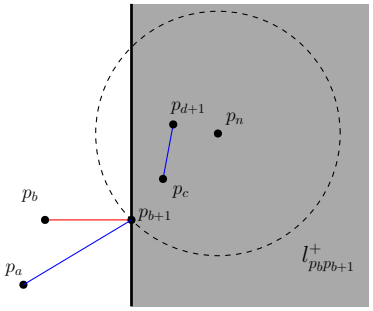
According to the previous lemma, the only red edge that can intersect $p_{i_j}p_{i_{j+1}+1}$ is $p_{i_{j+1}}p_{i_{j+1}+1}$, which is impossible. □

5.2 Phase 2

Taking into account Corollary 5.6, two blue edges can be defined as *consecutive* if they are of the shape p_ip_{k+1} and p_kp_j where $i < k < j$.

Lemma 5.9. *If all the blue edges are of the shape* $p_1p_{i_1+1}$, $p_{i_1}p_{i_2+1}$, $p_{i_2}p_{i_3+1}$, ..., $p_{i_k}p_n$ *and the red ones are of the shape* $p_{i_1}p_{i_1+1}$, $p_{i_2}p_{i_2+1}$, ..., $p_{i_k}p_{i_k+1}$ *where* $1 < i_1 < i_2 < i_3 < ... < i_k < n$, *then if two of these blue edges intersect each other, they must be consecutive (Figure 5.6).*

FIGURE 5.6: Construction explained at the proof of Lemma 5.9



Proof. Given $P = (p_1, p_2, ..., p_n)$ as the initial self-approaching path, if the blue edges are of the shape $p_1p_{i_1+1}$, $p_{i_1}p_{i_2+1}$, $p_{i_2}p_{i_3+1}$, ..., $p_{i_k}p_n$ and the red ones are of the shape $p_{i_1}p_{i_1+1}$, $p_{i_2}p_{i_2+1}$, ..., $p_{i_k}p_{i_k+1}$ where $1 < i_1 < i_2 < i_3 < ... < i_k < n$, we can take two non-consecutive blue edges p_ap_{b+1} and p_cp_{d+1} and a red edge p_bp_{b+1} where $a < b < c < d$. By Corollary 5.2, p_c, p_{d+1} and p_n lie in the half-plane $l^+_{p_bp_{b+1}}$ and p_b lies in the other half-plane $l^-_{p_bp_{b+1}}$, which implies that $p_a \in l^-_{p_jp_{j+1}}$ too, meaning $p_ap_{b+1} \cap p_cp_{d+1} = \emptyset$. In conclusion, if two blue edges intersect each other, they must be consecutive. □

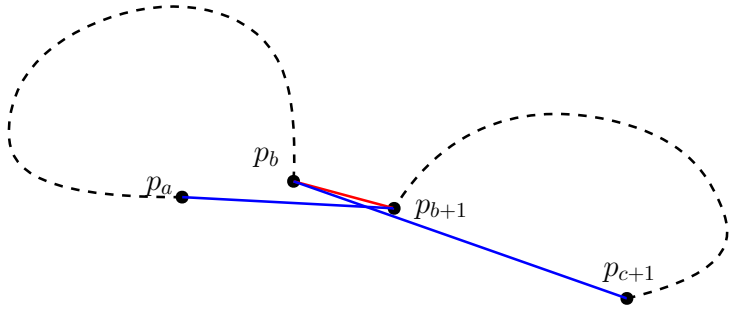
Now, like in the first phase we will prove the following lemma about how the crossing are solved in the second phase:

Lemma 5.10. *If the blue edges are of the shape* $p_1p_{i_1+1}$, $p_{i_1}p_{i_2+1}$, $p_{i_2}p_{i_3+1}$, ..., $p_{i_k}p_n$ *and the red ones are edges of the shape* $p_{i_1}p_{i_1+1}$, $p_{i_2}p_{i_2+1}$, ..., $p_{i_k}p_{i_k+1}$ *where* $1 < i_1 < i_2 < i_3 < ... < i_k < n$, *then if there is a crossing between two consecutive blue edges* p_ap_{b+1} *and* p_bp_{c+1} *where* $a < b < c$, *the way the algorithm solve this crossing is removing these blue edges, adding a new blue edge* p_ap_{c+1} *and turning the red edge* p_bp_{b+1} *to a green one.*

Proof. First of all, note that the initial green edges were the edges of the initial self-approaching path and at the r-th state the only way to not be in the set of green edges is if they have been turned to red ones. In other words, in the r-th state the initial green

path has been divided into a set of green subpaths that have the endpoints of the red edges as their endpoints. So after k crossings solved, these green subpaths will be $p_1p_{i_1}$, $p_{i_1+1}p_{i_2}$, $p_{i_2+1}p_{i_3}$, ..., $p_{i_k+1}p_{i_n}$ where $1 < i_1 < i_2 < i_3 < ... < i_k < n$, with some of them of length 0 if p_{i_j+1} is the same point as $p_{i_{j+1}}$.

FIGURE 5.7: This image shows a crossing between two consecutive blue edges p_ap_{b+1} and p_bp_{c+1} where $a < b < c$, and two subpaths (black dashed curves) defined by a union of blue and green edges that go from p_a to p_b and from p_{b+1} to p_{c+1} and have no points in common. Note that these subpaths are not scaled.



If there is a crossing between two consecutive blue edges p_ap_{b+1} and p_bp_{c+1} where $a < b < c$, there are two subpaths defined by a union of blue and green edges that go from p_a to p_b and from p_{b+1} to p_{c+1} and have no points in common (Figure 5.7). Furthermore, the union of these subpaths with the two consecutive blue edges p_ap_{b+1} and p_bp_{c+1} is the Hamiltonian cycle of the current step of the algorithm. So, to solve a crossing between p_ap_{b+1} and p_bp_{c+1} while keeping the unique cycle, the algorithm removes these edges, adds the new blue edge p_ap_{c+1} and turns the red edge p_bp_{b+1} to a green one. □

And like in the previous phase,

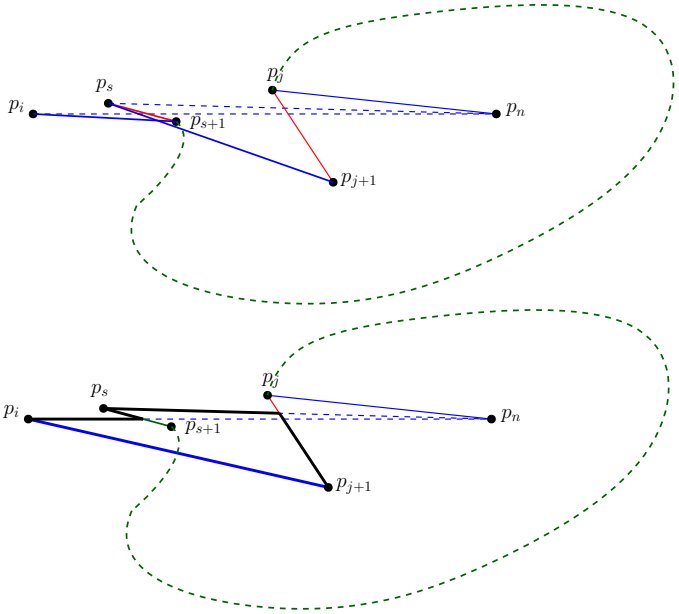
Corollary 5.11. *Given $P = (p_1, p_2, ..., p_n)$ as the initial self-approaching path, after all the iterations in phase 2, the blue edges will be edges of the shape $p_1p_{i_1+1}$, $p_{i_1}p_{i_2+1}$, $p_{i_2}p_{i_3+1}$, ..., $p_{i_k}p_n$ and the red ones will be edges of the shape $p_{i_1}p_{i_1+1}$, $p_{i_2}p_{i_2+1}$, ..., $p_{i_k}p_{i_k+1}$ where $1 < i_1 < i_2 < i_3 < ... < i_k < n$.*

Proof. First of all we need to prove that we can apply the lemmas 5.9 and 5.10 every time we solve a crossing in the second phase and thanks to Corollary 5.6 we know that we can apply it on the first time we solve a crossing in this phase. So, we need to prove that if we are solving a crossing between $p_{i_{\alpha-1}}p_{i_\alpha+1}$ and $p_{i_\alpha}p_{i_{\alpha+1}+1}$ while all the blue edges are of the shape $p_1p_{i_1+1}$, $p_{i_1}p_{i_2+1}$, ..., $p_{i_k}p_n$ and the red ones are of the shape $p_{i_1}p_{i_1+1}$, $p_{i_2}p_{i_2+1}$, ..., $p_{i_k}p_{i_k+1}$ where $1 < i_1 < i_2 < i_3 < ... < i_k < n$, then after solving the crossing, all the blue edges will be of the shape $p_1p_{j_1+1}$, $p_{j_1}p_{j_2+1}$, ..., $p_{j_m}p_n$ and the red ones will be of the shape $p_{j_1}p_{j_1+1}$, $p_{j_2}p_{j_2+1}$, ..., $p_{j_m}p_{j_m+1}$ where $1 < j_1 < j_2 < j_3 < ... < j_m < n$ and $m = k - 1$. This is immediate using Lemma 5.10 if we notice that solving a crossing

between $p_{i_{\alpha-1}}p_{i_\alpha+1}$ and $p_{i_\alpha}p_{i_{\alpha+1}+1}$ removes these two blue edges and adds another one $p_{i_{\alpha-1}}p_{i_{\alpha+1}+1}$ that maintains the blue edges' pattern and removes the red edge $p_{i_\alpha}p_{i_\alpha+1}$, which also maintains the red edges' pattern. □

Lemma 5.12. *(Figure 5.8). Given $P = (p_1, p_2, ..., p_n)$ as the initial self-approaching path, after solving a crossing in phase 2 between two blue edges p_ip_{s+1} and p_sp_{j+1} where $i < s < j$, the new blue edge p_ip_{j+1} can only be cut by a self-approaching edge p_kp_{k+1} if $i < k < j$. The transformed green edge can only go out of the closed area defined by p_i, the intersection point between p_ip_n and p_sp_{s+1}, p_s, the intersection point between p_sp_n and p_jp_{j+1}, p_{j+1} and p_i through the blue edge p_ip_{j+1}.*

FIGURE 5.8: The top image shows an intersection between two blue edges. The bottom image shows the solution of such intersection and the next intersection between green and blue edges that will appear. In such graph, the black edges show edges that cannot be cut. The green dashed curves are not in a real scale.



Proof. The proof of the first part is analogous to that of Lemma 5.7.

To prove the second part, just note that phase 1 took the farthest cut in p_ip_n and p_sp_n, so there will be no crossings with green edges between p_i and the intersection point between p_ip_n and p_sp_{s+1} nor between p_s and the intersection point between p_sp_n and p_jp_{j+1}. Furthermore, p_sp_{s+1} and p_jp_{j+1} cannot intersect with any other green edges because they both belong to the self-approaching path. □

This phase doesn't create any red edges, it only removes some of them to make them green again so it's immediate that:

Lemma 5.13. *The blue edges remaining at the end of phase 2 do not cross any red edges.*

Proof. Let $P = (p_1, p_2, ..., p_n)$ be the initial self-approaching path. Taking into account Corollary 5.11, the blue edges after all the iterations in phase 2 will be of the shape $p_1p_{i_1+1}$, $p_{i_1}p_{i_2+1}$, $p_{i_2}p_{i_3+1}$, ..., $p_{i_k}p_n$, and, on the other hand, all red edges after all the iterations in phase 1 will be of the shape $p_{i_1}p_{i_1+1}$, $p_{i_2}p_{i_2+1}$, $p_{i_3}p_{i_3+1}$, ...

According to Lemma 5.12, the only red edge that can intersect $p_{i_j}p_{i_{j+1}+1}$ is $p_{i_{j+1}}p_{i_{j+1}+1}$, which is impossible. □

5.3 Phase 3

Thanks to Corollary 5.11, it's known that all blue edges right before starting this phase will be of the shape $p_1p_{i_1+1}$, $p_{i_1}p_{i_2+1}$, $p_{i_2}p_{i_3+1}$, ..., $p_{i_k}p_n$ where $i_1 < i_2 < ... < i_n$. Furthermore, the lemmas 5.7 and 5.12 prove that every crossing placed in a blue edge p_ip_{j+1} belongs to the sub-path $p_ip_{i+1}...p_jp_{j+1}$ and this will be true as well through the third phase because such phase is an iteration of the phases 1 and 2 over all the blue edges with intersections. Note that blue edges intersections can come from solving the crossings from different blue edges, but the proof of Lemma 5.9 can be applied anyway if the blue edges keep following the pattern $p_1p_{i_1+1}$, $p_{i_1}p_{i_2+1}$, $p_{i_2}p_{i_3+1}$, ..., $p_{i_k}p_n$.

So, if the way to solve a crossing between two edges p_ip_{j+1} (blue) and p_kp_{k+1} (green) where $i < k < j$ is the same as phase 1, the new blue edges will be p_ip_{k+1} and p_kp_{j+1}. Moreover, if the way to solve a crossing between two consecutive blue edges is the same as in phase 2, the new blue edge will be p_ip_{j+1}. In other words:

Lemma 5.14. *Given $P = (p_1, p_2, ..., p_n)$ as the initial self-approaching path, the blue edges after all the iterations in phase 3 will be of the shape $p_1p_{i_1+1}$, $p_{i_1}p_{i_2+1}$, $p_{i_2}p_{i_3+1}$, ..., $p_{i_k}p_n$ and will not intersect any other blue or green edges.*

Finally, by the same way we proved Lemma 5.8, we can prove that:

Lemma 5.15. *The blue edges remaining at the end of phase 1 do not cross any red edges.*

Chapter 6

MST Path: towards a proof

In this chapter we are going to present a few conjectures that in case they are proved, they are enough to prove that the already mentioned algorithm transforms our initial MST path P to a Hamiltonian cycle that is P-compatible (Figure 6.1), which would prove the main conjecture in this work:

Conjecture 6.1. *Given an MST path P, there is always a Hamiltonian cycle that is P-compatible.*

The conjectures mentioned above are:

Conjecture 6.2. *The blue edges remaining at the end of phase 1 do not cross any red edges.*

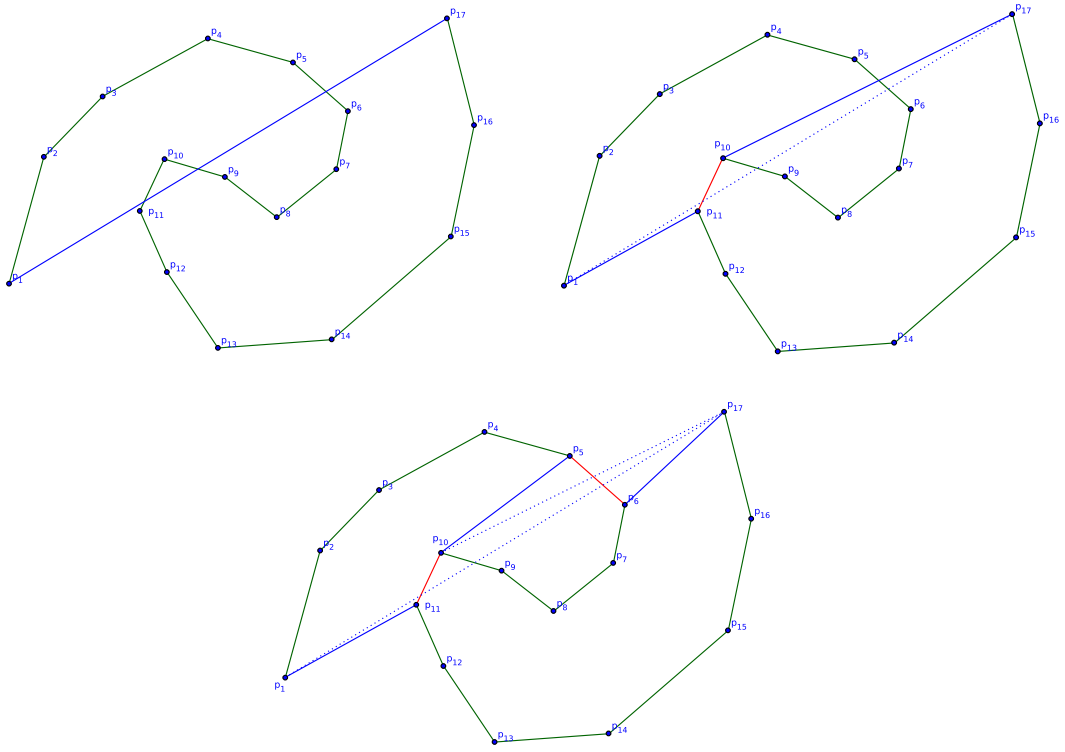
Conjecture 6.3. *The blue edges remaining at the end of phase 2 do not cross any red edges.*

Conjecture 6.4. *The blue edges remaining at the end of phase 3 do not cross any red edges.*

Note that having all blue edges not crossing with the red edges is equivalent to being compatible with the initial MST path.

First of all, notice that on every iteration during the first phase the algorithm removes a blue edge, turns a green edge to a red one and adds two blue edges, which are connected by the red one. The union of the blue edges and the red one is a subpath of length three that has the same endpoints as the removed blue edge. Furthermore, note that on every iteration during the second phase there are two possibilities: the algorithm can remove two blue edges and add two new ones, which connect the same endpoints, or can remove two blue edges, turn a red edge to a green one and add a blue edge, which connect the

FIGURE 6.1: The image on the top-left shows an initial MST path in green and the p_1p_n edge in blue, which creates a non-plane Hamiltonian cycle and the image on the top-right shows the state of the graph after solving the first iteration. Finally, the image on the bottom shows a plane Hamiltonian cycle built by blue and green edges and the red edges defining the removed edges from the initial MST path. We show the previously removed blue edges in a dotted edge.



endpoints of the subpath defined as the union of the red edge and the removed blue edges. With these facts, we can realize that right after iteration i there is a polyline $Q_i = b_0^i a_1^i b_1^i ... a_{m-1}^i b_{m-1}^i a_m^i$ where $m \leq n/2$, $p_1 = b_0^i$ and $p_n = a_m^i$, which is formed by the red and the blue edges of the i-th iteration (Figure 6.2). Note that the edges of the shape $b_j^i a_{j+1}^i$ are the blue ones and the edges of the shape $a_j^i b_j^i$ are the red ones.

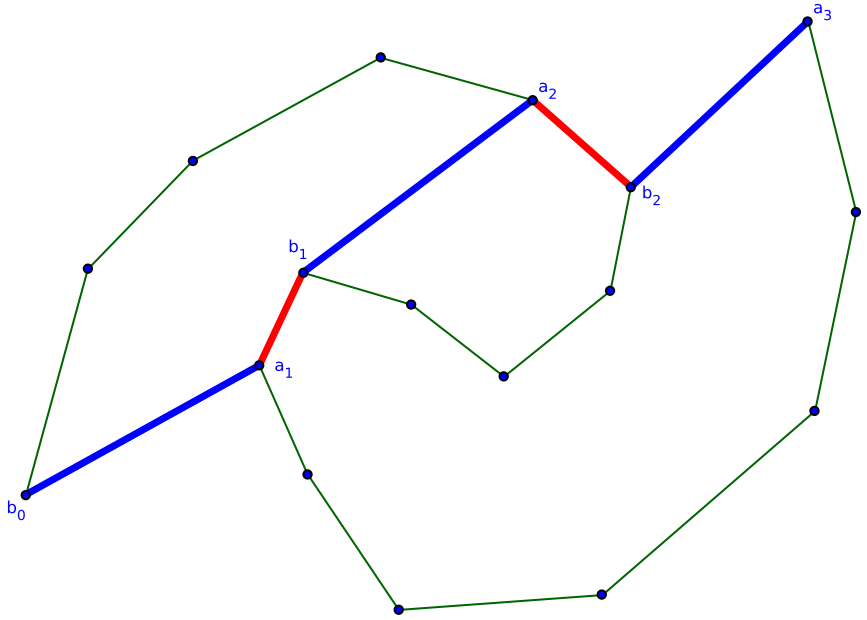
6.1 Phase 1

We have simplified the proof of Conjecture 6.2 in one lemma that we are going to prove and a new simpler conjecture:

Lemma 6.5. *During phase 1 a red edge cannot intersect a previous blue edge. In other words, a red edge $a_j^i b_j^i$ cannot intersect a blue edge $b_k^i a_{k+1}^i$ if $k < j$.*

Conjecture 6.6. *During phase 1 a blue edge cannot intersect a previous red edge. In other words, a blue edge $b_j^i a_{j+1}^i$ cannot intersect a red edge $a_k^i b_k^i$ if $k \leq j$.*

FIGURE 6.2: This image shows the polyline Q formed by the red and blue edges.



To prove Lemma 6.5 we will begin proving that:

Lemma 6.7. *During phase 1, if a red edge $a^i_j b^i_j$ intersects a blue edge $b^i_k a^i_{k+1}$, then*

$$min\{dist(a^i_j, p_n), dist(b^i_j, p_n)\} > min\{dist(a^i_{k+1}, p_n), dist(b^i_{k+1}, p_n)\}$$

(note that $a^i_{k+1} b^i_{k+1}$ is the next red edge of the mentioned blue edge).

Before starting to prove it, we need the following definitions:

Definition 6.8. Given two points a and b, we define $Circ(a, b)$ as the circle that has ab as the diameter.

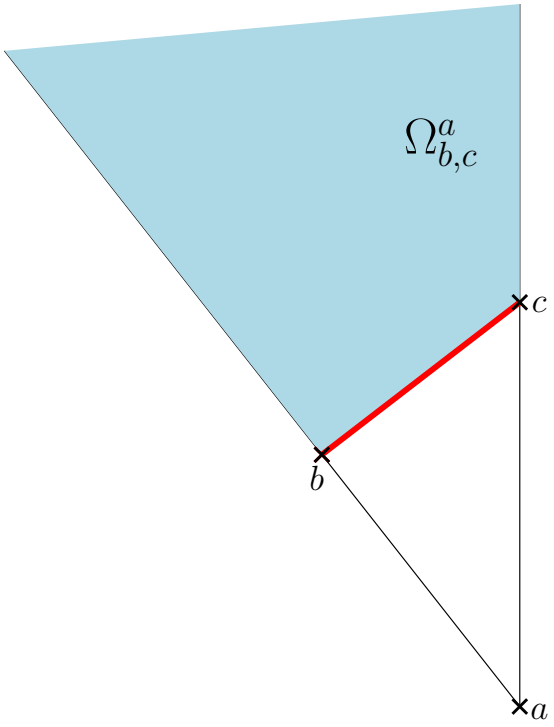
Definition 6.9. Given two points a and b, we define $Len(a, b)$ as the union of all points whose distance from a and b is smaller than the distance between a and b (i.e. $\{x \in \mathbb{R}^2 : max(dist(x, a), dist(x, b)) < dist(a, b)\}$).

With this definition we can announce the following property of MST edges that has been inherited from the RNG edges: *Given any edge ab of an MST, $Len(a, b) = \emptyset$.*

Definition 6.10. Given three points a, b and c, the region $\Omega^a_{b,c}$ is the unbounded region defined by the ray from a through b, the ray from a through c and the edge bc (Figure 6.3).

Note that when we are solving a crossing between a green edge $p_k p_{k+1}$ and a blue edge $b^i_{j-1} p_n$, we create the two new blue edges $b^i_{j-1} a^i_j$ and $b^i_j p_n$, where $a^i_j = p_k$ and $b^i_j = p_{k+1}$,

FIGURE 6.3: Image of $\Omega^a_{b,c}$.



and turn the green edge $p_k p_{k+1}$ to a red one that we are going to call $a^i_j b^i_j$. With this, it's immediate to see that the blue edge $b^i_{j-1} a^i_j$ is inside the region $\Omega^{p_n}_{a^i_j b^i_j}$ of his next red edge $a^i_j b^i_j$ in Q_i (Figure 6.4). So if any red edge $a^i_k b^i_k$ intersects a blue edge, this red edge has to be inside $\Omega^{p_n}_{a^i_j b^i_j}$, or at least part of it (Figure 6.5). Furthermore, the situation where the edge $a^i_k b^i_k$ intersects the ray from a^i_j in the same direction but with opposite sense of p_n and the ray from b^i_j in the same direction but with opposite sense of p_n cannot happen because then the crossing between $a^i_j b^i_j$ and $b^i_{j-1} p_n$, which turned $a^i_j b^i_j$ to a red edge, was not the furthest crossing from p_n on the edge $b^i_{j-1} p_n$. So, Lemma 6.7 is proved just proving the following lemma:

Lemma 6.11. *During the first phase, given two red edges bc and de and the point p_n where $e \in \Omega^{p_n}_{b,c}$, then b or c are closer to p_n than d or e (Figure 6.6):*

$$min\{dist(p_n, b), dist(p_n, c)\} < min\{dist(p_n, d), dist(p_n, e)\}$$

Proof. If $d \in \Omega^{p_n}_{b,c}$ then is immediate that the minimum value between $dist(p_n, b)$ and $dist(p_n, c)$ is smaller than the minimum value between $dist(p_n, d)$ and $dist(p_n, e)$.

So we just need to prove that the lemma is true if $d \notin \Omega^{p_n}_{b,c}$.

FIGURE 6.4: The image on the left shows the crossing we are going to solve and the image on the right shows that after solving the crossing, $b^i_{j-1}a^i_j$ is inside the region $\Omega^{p_n}_{a^i_j b^i_j}$

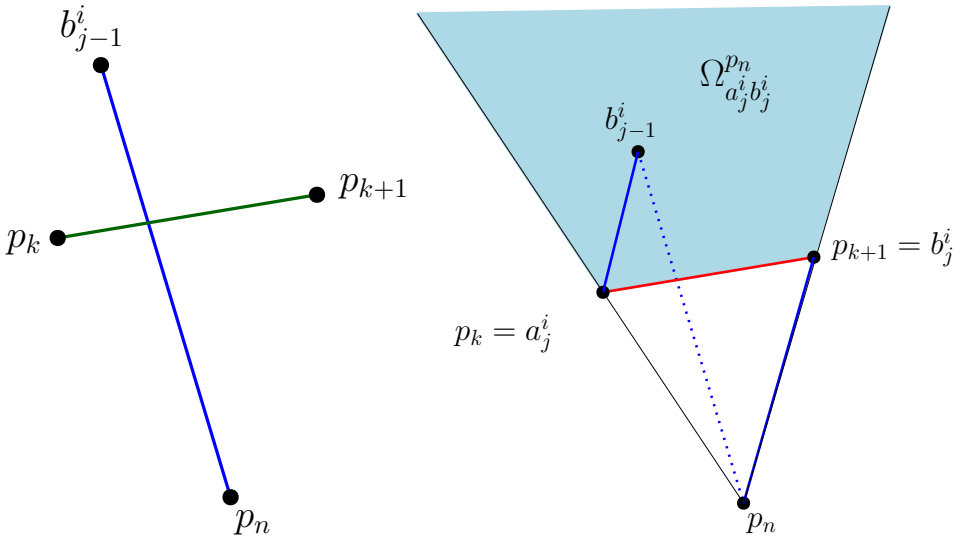
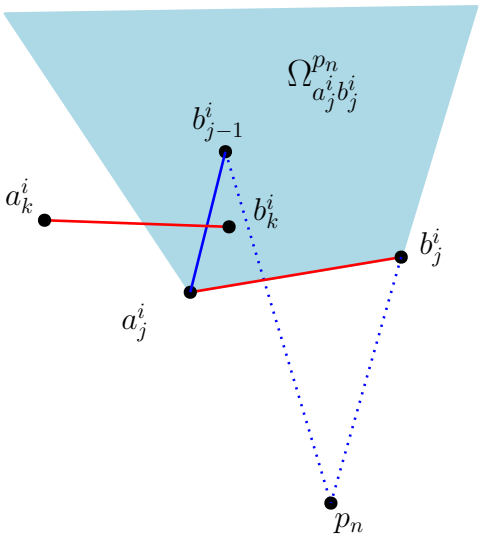


FIGURE 6.5: This image shows that if a red edge $a^i_k b^i_k$ intersects a blue edge, this red edge has to be inside $\Omega^{p_n}_{a^i_j b^i_j}$, or at least part of it.

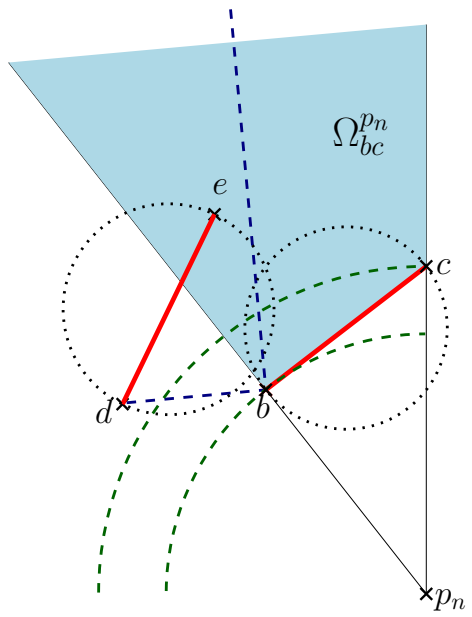


From now on, polar coordinates with origin in p_n will be used, so $\forall q$, $dist(p_n, q) = r_q$. Furthermore, without loss of generality, we can fix $r_B < r_C$ and $\alpha_B < \alpha_C$, so we only need to prove that $r_b < min\{r_d, r_e\}$.

Note that, since the red edges are edges from the MST path, $Len(b, c) = \emptyset$, $Len(d, e) = \emptyset$ and $bc \cap de = \emptyset$ and, if the angle between bd and be is greater than $\frac{\pi}{2}$, then $b \in Circ(d, e)$ by the *Thales' Theorem*[1], which implies $b \in Len(d, e)$ (Figure 6.7).

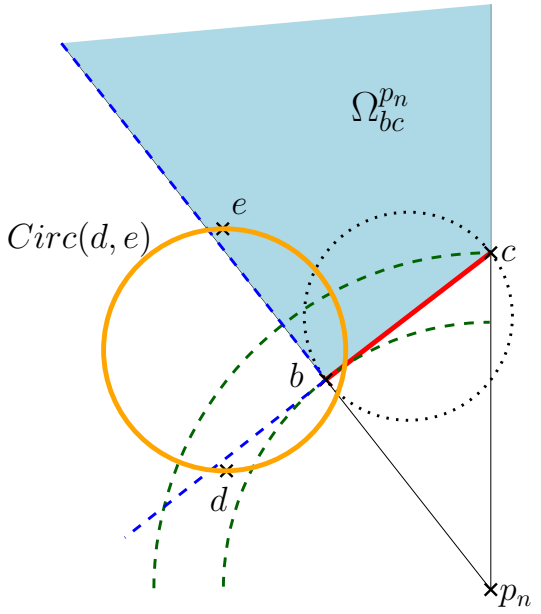
[1] **Thales' theorem:** If A, B and C are distinct points on a circle where the line AC is the diameter of the circle, then the angle between BA and BC is a right angle.

FIGURE 6.6: This image shows two red edges bc and de and the point p_n where $e \in \Omega_{b,c}^{p_n}$.



Before starting, observe that if we define two halfplanes using a line perpendicular to the segment $p_n e$ that contains p_n, d and e have to be in the same halfplane, because if not, $p_n \in Len(e, d)$.

FIGURE 6.7: This image shows that if the angle between bd and be is greater than $\frac{\pi}{2}$, then $b \in Circ(d, e)$ and the circumferences with center p_n and radius r_b and r_c.



Now, if $r_d < r_b$, we can divide the possibilities depending on the angle: $\alpha_d < \alpha_b$, $\alpha_b < \alpha_d < \alpha_c$ and $\alpha_c < \alpha_d$. If $\alpha_d < \alpha_b$ then, the angle between bd and be is greater than $\frac{\pi}{2}$, which implies that b has to be in $Len(d, e)$, which is not possible because $Len(d, e) = \emptyset$. If $\alpha_b < \alpha_d < \alpha_c$ then, taking into account the lemma's hyphotesis, d has

to be outside $\Omega^{p_n}_{b,c}$, which implies that d has to be inside the triangle p_nbc. Note that this cannot be possible because $de \cap bc \neq \emptyset$. And finally, if $\alpha_c < \alpha_d$ and $r_d < r_b < r_c$, either the angle between cd and ce is greater than $\frac{\pi}{2}$, which implies that c has to be in $Len(d, e)$, which is not possible because $Len(d, e) = \emptyset$, or $ed \cap bc \neq \emptyset$, which is also a contradiction of the lemma's hyphotesis. In conclusion, r_d must be greater than r_b.

Note that if $r_b < r_c$, then if $e \in \Omega^{p_n}_{b,c}/Len(b, c)$, it's immediate that $r_e > r_b$. And so on, $r_b < min\{r_d, r_e\}$. □

Once Lemma 6.7 is proved we need to prove this lemma:

Lemma 6.12. *Given two red edges $a^i_j b^i_j$ and $a^i_k b^i_k$ if $j < k$ then*

$$min\{dist(a^i_j, p_n), dist(b^i_j, p_n)\} > min\{dist(a^i_k, p_n), dist(b^i_k, p_n)\}$$

Proof. First of all, note that if we have two red edges $a^i_j b^i_j$ and $a^i_{j+1} b^i_{j+1}$ with only one blue edge between them, then the second red edge is created because previously we solved the crossing between $b^i_j p_n$ and $a^i_{j+1} b^i_{j+1}$, so $b^i_j \in \Omega^{p_n}_{a^i_{j+1} b^i_{j+1}}$. Now, taking into account Lemma 6.10 we know that

$$min\{dist(a^i_j, p_n), dist(b^i_j, p_n)\} > min\{dist(a^i_{j+1}, p_n), dist(b^i_{j+1}, p_n)\}$$

And with that it's immediate to conclude the following:

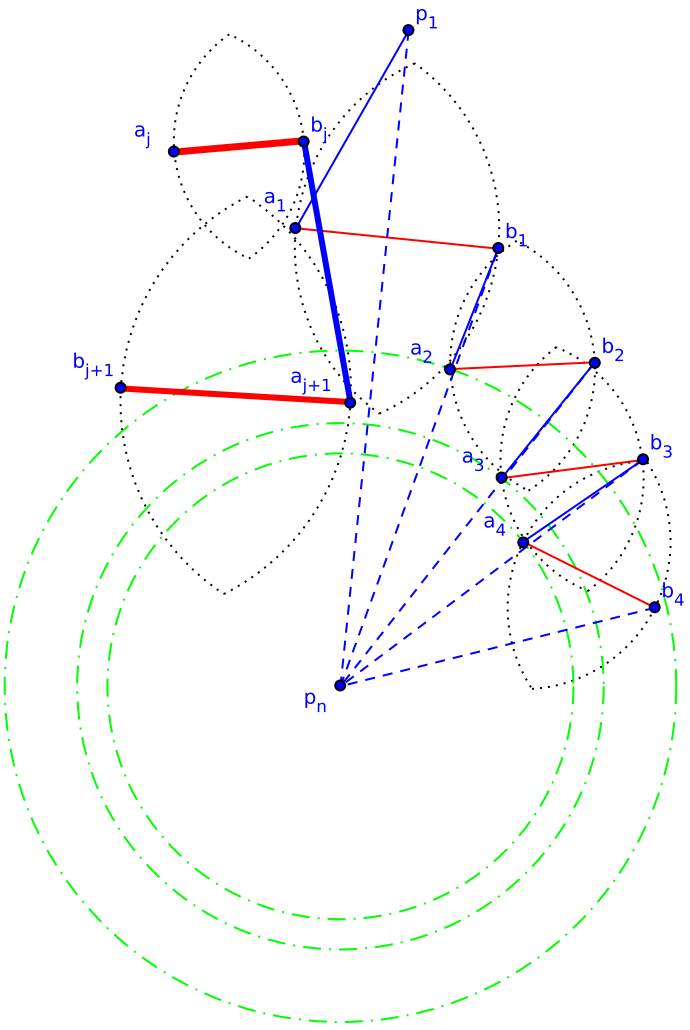
Given two red edges $a^i_j b^i_j$ and $a^i_k b^i_k$ if $j < k$, then $min\{dist(a^i_j, p_n), dist(b^i_j, p_n)\} > min\{dist(a^i_{j+1}, p_n), dist(b^i_{j+1}, p_n)\} > ... > min\{dist(a^i_k, p_n), dist(b^i_k, p_n)\}$.

□

Now, with the lemmas 6.7 and 6.12 in place, it's immediate that Lemma 6.5 holds.

Observing Conjecture 6.6 we have tried to create a blue edge $b^i_j a^i_{j+1}$ that intersects a red edge $a^i_k b^i_k$ where $k \leq j$ and have noticed that to get to this situation, the polyline of red and blue edges needs to do a full rotation around p_n. We have also noticed that the above-mentioned situation needs a notorious distance from p_n, which goes against Lemma 6.12. (Figure 6.8). Since this seems difficult to achieve with MST edges, we believe that the conjecture is true.

FIGURE 6.8: This image shows how the distance between the red edges and p_n is reduced on every iteration and how the existence of the blue edge $b_j a_{j+1}$ that intersects $a_1 b_1$ needs a previous red edge with notorious distance from p_n.



6.2 Phase 2

Assuming Conjecture 6.2 is true, we present a way to divide Conjecture 6.3 in one proved lemma and three new conjectures:

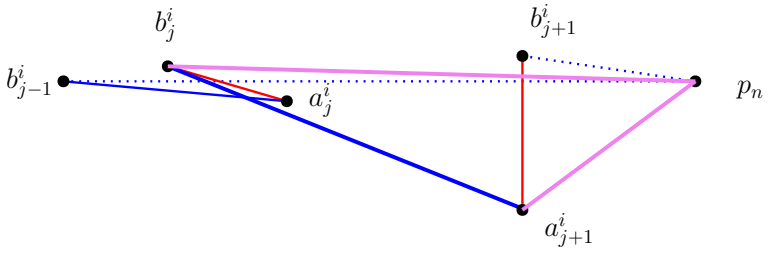
Lemma 6.13. *Given a crossing between two consecutive blue edges $b^i_{j-1}a^i_j$ and $b^i_j a^i_{j+1}$, created while applying phase 1, and a red edge $a^i_k b^i_k$, if $j < k$ then this red edge will not intersect any of the three new possible blue edges $b^i_{j-1}b^i_j$, $b^i_{j-1}a^i_{j+1}$ and $a^i_j a^i_{j+1}$ generated when solving the crossing.*

Proof. To prove that we are going to prove that $b^i_{j-1}b^i_j$, $b^i_{j-1}a^i_{j+1}$ and $a^i_ja^i_{j+1}$ are inside $\Omega^{p_n}_{a^i_{j+1}b^i_{j+1}}$. If a red edge $a^i_kb^i_k$ intersects one of these blue edges, this red one has to be inside $\Omega^{p_n}_{a^i_jb^i_j}$ or at least part of it has to, so we can apply Lemma 6.11 to know that $min\{dist(p_n, a_{j+1}), dist(p_n, b_{j+1})\} < min\{dist(p_n, b^i_k), dist(p_n, a^i_k)\}$. Comparing this with the result of Lemma 6.12 we can realize that if $j < k$ then this red edge will not intersect any of the three new possible blue edges $b^i_{j-1}b^i_j$, $b^i_{j-1}a^i_{j+1}$ and $a^i_ja^i_{j+1}$ generated when solving the crossing.

Now we are going to prove that the blue edges $b^i_{j-1}b^i_j$, $b^i_{j-1}a^i_{j+1}$ and $a^i_ja^i_{j+1}$ are inside $\Omega^{p_n}_{a^i_{j+1}b^i_{j+1}}$. First of all note that $\Omega^{p_n}_{a^i_{j+1}b^i_{j+1}}$ is a convex polygon so we only need to prove that $b^i_{j-1}, a^i_j, b^i_j, a^i_{j+1} \in \Omega^{p_n}_{a^i_{j+1}b^i_{j+1}}$, and we can easily see that the vertex a^i_{j+1} is inside $\Omega^{p_n}_{a_{j+1}b_{j+1}}$ by definition and the vertex b^i_j is inside $\Omega^{p_n}_{a^i_{j+1}b^i_{j+1}}$ by construction, so we only need to see that a^i_j and b^i_{j-1} are in $\Omega^{p_n}_{a^i_{j+1}b^i_{j+1}}$

If $b^i_{j-1}a^i_j$ and $b^i_ja^i_{j+1}$ intersect each other, then a^i_j is inside the triangle $p_nb^i_ja^i_{j+1}$ (Figure 6.9). This triangle is divided into regions by the red edge $a^i_{j+1}b^i_{j+1}$ with the fact that one region is inside $\Omega^{p_n}_{a^i_{j+1}b^i_{j+1}}$ and the other one is outside it. The vertex a^i_j cannot be in the region outside $\Omega^{p_n}_{a^i_{j+1}b^i_{j+1}}$ because b^i_j is in $\Omega^{p_n}_{a^i_{j+1}b^i_{j+1}}$ and $a^i_jb^i_j$ cannot intersect $a^i_{j+1}b^i_{j+1}$ because they are both edges of the initial MST path. In conclusion, a^i_j is inside $\Omega^{p_n}_{a^i_{j+1}b^i_{j+1}}$.

FIGURE 6.9: This image shows how if $b^i_{j-1}a^i_j$ and $b^i_ja^i_{j+1}$ intersect each other, then a^i_j is inside the triangle $p_nb^i_ja^i_{j+1}$.



b^i_{j-1} is in $\Omega^{p_n}_{a^i_jb^i_j}$ and since a^i_j and b^i_j are in $\Omega^{p_n}_{a^i_{j+1}b^i_{j+1}}$, b^i_{j-1} is also in $\Omega^{p_n}_{a^i_{j+1}b^i_{j+1}}$.

In conclusion, $b^i_{j-1}, a^i_j, b^i_j, a^i_{j+1} \in \Omega^{p_n}_{a^i_{j+1}b^i_{j+1}}$ so $b^i_{j-1}b^i_j$, $b^i_{j-1}a^i_{j+1}$ and $a^i_ja^i_{j+1}$ are inside $\Omega^{p_n}_{a^i_{j+1}b^i_{j+1}}$ and this is what we needed to prove. □

Conjecture 6.14. *Given a crossing between two consecutive blue edges $b^i_{j-1}a^i_j$ and $b^i_ja^i_{j+1}$, created while applying phase 1, and a red edge $a^i_kb^i_k$, if $j > k$ then this red edge will not intersect any of the three new possible blue edges $b^i_{j-1}b^i_j$, $b^i_{j-1}a^i_{j+1}$ and $a^i_ja^i_{j+1}$ generated when solving the crossing.*

Note that this conjecture is the same as Lemma 6.13 with the exception that the red edges are previous to the intersecting blue edges. And together we get:

Given a crossing between two consecutive blue edges $b^i_{j-1}a^i_j$ and $b^i_j a^i_{j+1}$, created while applying phase 1, any red edge will not intersect any of the three new possible blue edges $b^i_{j-1}b^i_j$, $b^i_{j-1}a^i_{j+1}$ and $a^i_j a^i_{j+1}$ generated when solving the crossing.

Moreover, during our studies, we have noticed that the following conjecture could be true:

Conjecture 6.15. *If two blue edges intersect each other, either they are consecutive or there exists another blue edge between them that intersects one of them.*

Which increases the value of the previous lemma and conjecture because gathering them together we get:

Given a crossing between two blue edges, created while applying phase 1, any red edge will not intersect any of the three new possible blue edges generated when solving the crossing.

Note that in the previous lemma and conjectures we didn't say anything about the new blue edges created through this second phase that intersect another blue edge. The following conjecture covers this point:

Conjecture 6.16. *After solving a crossing between two blue edges generated while applying the second phase of the algorithm, the new blue edges don't intersect any red edges and in case they do, they also intersect a blue edge generating a crossing that is going to be solved during the second phase of the algorithm.*

6.3 Phase 3

In this phase we are solving all the crossings between blue and green edges that the algorithm created during the previous two phases and the new ones that will be created during this phase. To solve all green crossings in our blue edges of the shape $b^i_j a^i_{j+1}$, the algorithm uses the phases 1 and 2 with the difference that instead of starting from the edge p_1p_n, it starts from $b^i_j a^i_{j+1}$. Assuming the conjectures 6.2 and 6.3 are true, our problematic situation is with the crossings between the blue edges created while applying the first and second phases to different blue edges.

This situation is hard to find so we have tried to find a similar one with more relaxed conditions using a path where all edges were from the *Relative Neighborhood Graph* (i.e. every edge ad, $Len(a,d) = \emptyset$), and in fact we have found one example (Figures 6.10, 6.11, 6.12). Observing it, we have noticed that the structure that caused the fact that the algorithm didn't work cannot be present in the MST path, due to some edges that

FIGURE 6.10: This image shows a path with edges from a *Relative Neighborhood Graphs* where the algorithm doesn't work.

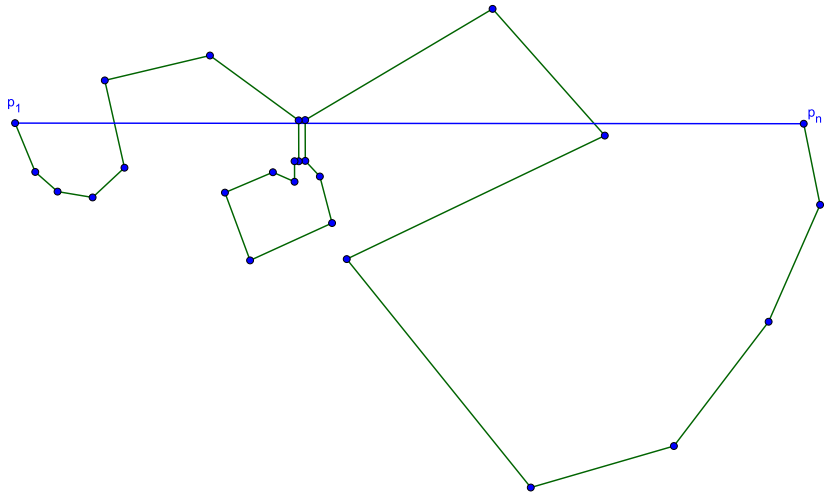
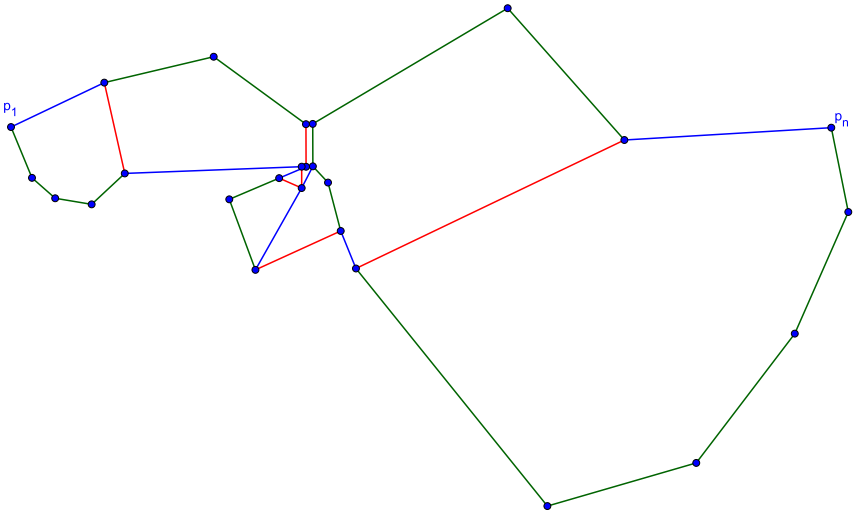


FIGURE 6.11: This image shows the final state after applying the algorithm on the graph of the Figure 6.10.



are not in the path formed by green and red edges that should be in the minimum spanning tree and some others that shouldn't (Figure 6.13), and for this reason we have introduced our last conjecture:

Conjecture 6.4 *The blue edges remaining at the end of phase 3 do not cross any red edges.*

Note that the path used is not a path coinciding with an RNG of a set of points, only a path using a subset of the edges of the RNG of a set of points. So the conjecture can still be true for this case.

FIGURE 6.12: This image shows a zoom in of the Figure 6.11.

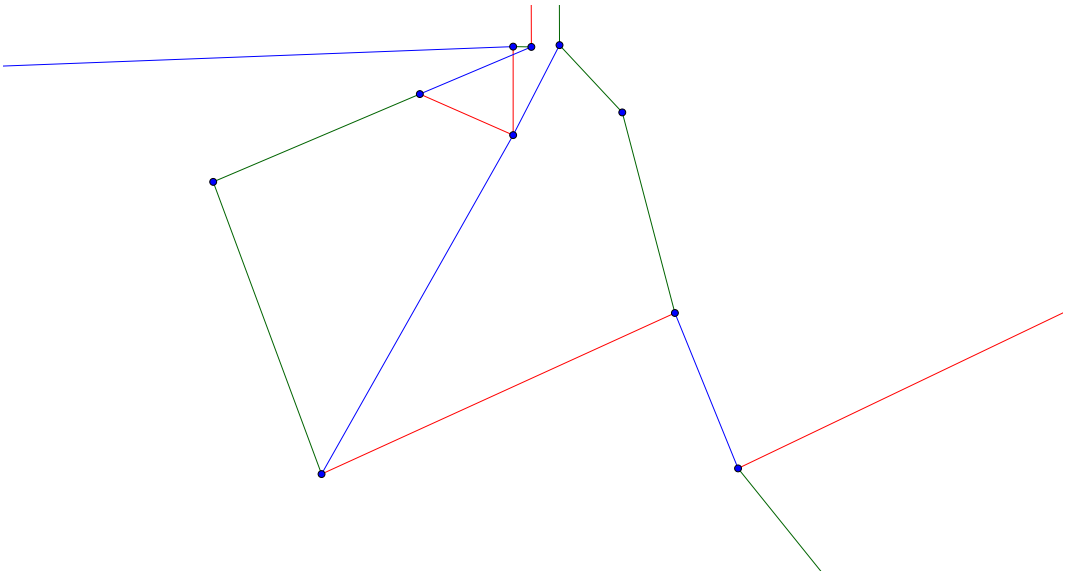
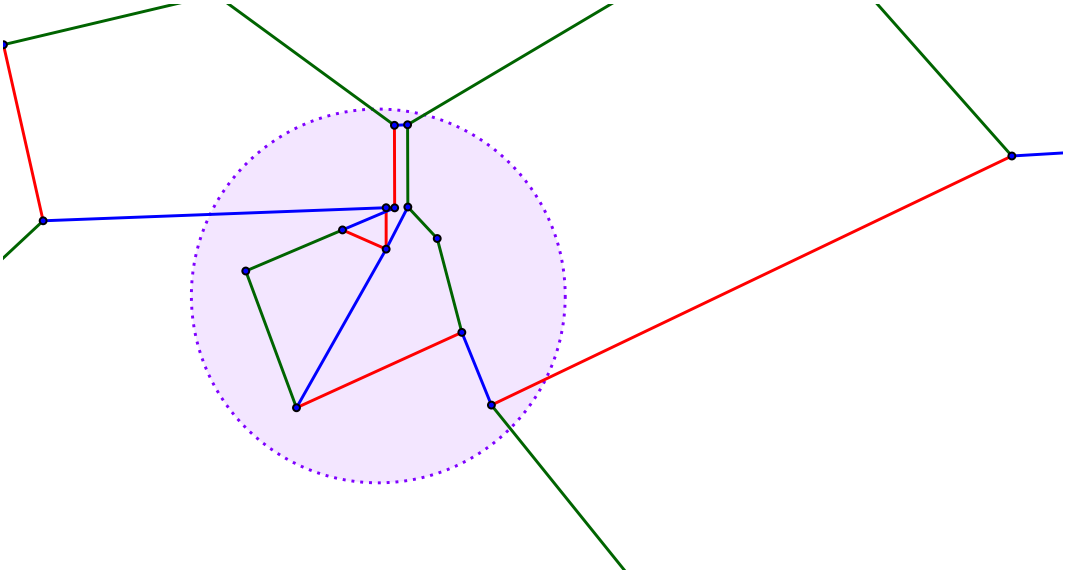


FIGURE 6.13: This image shows that in the example 6.11 there are some positions that cannot belong to an MST path.



Chapter 7

Conclusions

This project was focused on studying conditions under which a given set of points has a spanning path that is compatible with a *Hamiltonian cycle*. We have proved that being *monotone* or *self-approaching* paths is enough condition to ensure there will be a compatible Hamiltonian cycle. Moreover, we have studied the condition of being a path that coincides with the MST of the set of points and we have proved some interesting results in order to help future research to prove that this is enough condition.

Our proofs have been focused around an algorithm that, given a path P of the above-mentioned types of paths, creates a plane Hamiltonian cycle compatible with it. The algorithm studied is based on that of J. van Leeuwen and A.A. Schoone [32], which proves that a non-plane Hamiltonian cycle can be transformed to a plane one using $O(n^3)$ operations called flips.

Throughout the project we have proved the following theorems:

Theorem 4.1. *Given a monotone path $P = p_1p_2...p_n$, there is always a Hamiltonian path that is P-compatible.*

Theorem 5.1. *Given a self-approaching path $P = p_1p_2...p_n$, there is always a Hamiltonian path that is P-compatible.*

To do so, we have used the fact that, in these particular cases, if a green edge of the shape p_kp_{k+1} intersects a blue edge of the shape p_ip_j where $i < j$, then $i < k$. This fact is also present in some others paths, like spiral paths, so we propose the following conjecture:

Conjecture 7.1. *Given a spiral path P, there is always a Hamiltonian path that is P-compatible.*

In addition, in light of the results achieved in the computational studies presented in the second chapter and based on the insights gained proving the different results in this work, we have announced the following conjecture for which we have proposed in the sixth chapter a potential way to prove it:

Conjecture 6.1. *Given an MST path P, there is always a Hamiltonian path that is P-compatible.*

If this conjecture holds, it can be extended to a path that coincides with the *Relative Neighborhood Graph* of the set of points and even to a path that coincides with the *Gabriel Graph* of the set of points.

Finally, note that the algorithm defined in the third chapter and used to find a Hamiltonian cycle compatible with a given monotone or self-approaching path is the algorithm of J. van Leeuwen and A.A. Schoone [32], but solving crossings with a particularly defined order. Due to this and the fact that the algorithm of J. van Leeuwen and A.A. Schoone transforms a given cycle to a plane one no matter the order used to solve all the crossings, we propose the following conjectures for future research:

Conjecture 7.2. *Applying the J. van Leeuwen and A.A. Schoone's algorithm to any monotone path P, without any defined order to solve crossings, we always find a Hamiltonian cycle compatible with it.*

Conjecture 7.3. *Applying the J. van Leeuwen and A.A. Schoone's algorithm to any self-approaching path P, without any defined order to solve crossings, we always find a Hamiltonian cycle compatible with it.*

Bibliography

[1] A. V. Aho and J. E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1974.

[2] S. Alamdari, T. M. Chan, E. Grant, A. Lubiw, and V. Pathak. Self-approaching graphs. *CoRR*, abs/1306.5460, 2013.

[3] T. Asano, S. Kikuchi, and N. Saito. A linear algorithm for finding hamiltonian cycles in 4-connected maximal planar graphs. *Discrete Applied Mathematics*, 7(1):1 – 15, 1984.

[4] J. L. Bentley and J. H. Friedman. Fast algorithms for constructing minimal spanning trees in coordinate spaces. *IEEE Transactions on Computers*, C-27(2):97–105, Feb 1978.

[5] S. Biswas, S. Durocher, D. Mondal, and R. I. Nishat. *Hamiltonian Paths and Cycles in Planar Graphs*, pages 83–94. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[6] R. E. Bixby and D.-L. Wang. *An algorithm for finding hamiltonian circuits in certain graphs*, pages 35–49. Springer Berlin Heidelberg, Berlin, Heidelberg, 1978.

[7] J. Bondy and V. Chvatal. A method in graph theory. *Discrete Mathematics*, 15(2):111 – 135, 1976.

[8] O. Boruvka. O Jistém Problému Minimálním (About a Certain Minimal Problem) (in Czech, German summary). *Práce Mor. Prírodoved. Spol. v Brne III*, 3, 1926.

[9] K. Buchin, C. Knauer, K. Kriegel, A. Schulz, and R. Seidel. *On the Number of Cycles in Planar Graphs*, pages 97–107. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[10] W. Chou and A. Kershenbaum. A unified algorithm for designing multidrop teleprocessing networks. In *Proceedings of the Third ACM Symposium on Data Communications and Data Networks: Analysis and Design*, DATACOMM '73, pages 148–156, New York, NY, USA, 1973. ACM.

[11] N. Christofides. The shortest hamiltonian chain of a graph. *SIAM Journal on Applied Mathematics*, 19(4):689–696, 1970.

[12] N. Christofides. *Graph Theory: An Algorithmic Approach (Computer Science and Applied Mathematics)*. Academic Press, Inc., Orlando, FL, USA, 1975.

[13] N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, 1976.

[14] J. A. Dei Rossi and C. Rand Corp., Santa Monica. *A Cost Analysis of Minimum Distance TV Networking for Broadcasting Medical Information [microform] / J. A. Dei Rossi and Others*. Distributed by ERIC Clearinghouse [Washington, D.C.], 1970.

[15] L. R. Esau and K. C. Williams. On teleprocessing system design: Part ii a method for approximating the optimal network. *IBM Syst. J.*, 5(3):142–147, Sept. 1966.

[16] R. J. Gould. Updating the hamiltonian problema survey. *Journal of Graph Theory*, 15(2):121–157, 1991.

[17] R. J. Gould. Advances on the hamiltonian problem – a survey. *Graphs and Combinatorics*, 19(1):7–52, Mar 2003.

[18] R. J. Gould. Recent advances on the hamiltonian problem: Survey iii. *Graphs and Combinatorics*, 30(1):1–46, Jan 2014.

[19] R. L. Graham and P. Hell. On the history of the minimum spanning tree problem. *IEEE Ann. Hist. Comput.*, 7(1):43–57, Jan. 1985.

[20] M. Held and R. M. Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18(6):1138–1162, 1970.

[21] M. Held and R. M. Karp. The traveling-salesman problem and minimum spanning trees: Part ii. *Mathematical Programming*, 1(1):6–25, Dec 1971.

[22] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms*. Computer Software Engineering Series. Computer Science Press, 1978.

[23] F. Hurtado and C. D. Tóth. *Plane Geometric Graph Augmentation: A Generic Perspective*, pages 327–354. Springer New York, New York, NY, 2013.

[24] M. Ishaque, D. L. Souvaine, and C. D. Tóth. Disjoint compatible geometric matchings. *Discrete & Computational Geometry*, 49(1):89–131, Jan 2013.

[25] R. Johnson and M. G. Pilcher. The traveling salesman problem, edited by e.l. lawler, j.k. lenstra, a.h.g. rinnooy kan, and d.b shmoys, john wiley & sons, chichester, 1985, 463 pp. *Networks*, 18(3):253–254, 1988.

[26] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[27] H. Loberman and A. Weinberger. Formal procedures for connecting terminals with a minimum total wire length. *J. ACM*, 4(4):428–437, Oct. 1957.

[28] C. H. Papadimitriou. The euclidean traveling salesman problem is np-complete. *Theor. Comput. Sci.*, 4(3):237–244, 1977.

[29] R. C. Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, Nov 1957.

[30] E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice Hall College Div, 1977.

[31] M. Van Garderen, W. Meulemans, B. Speckmann, and C. Tóth. Unpublished manuscript, 2016.

[32] J. van Leeuwen and A. Schoone. *Untangling a Traveling Salesman Tour in the Plane*. Rijksuniversiteit. Vakgroep Informatica, 1980.

Appendix A: Code of the Automated case generator

Listing 1: main.cpp

```
#include <iostream>
#include <fstream>
#include <boost/lexical_cast.hpp>
#include <sys/stat.h>
#include <sys/types.h>
#include "Graph.h"
#include "Point.h"
#include <algorithm>
#include <cstdlib>

using namespace std;

// number of attemps of adding a new edge when the randomly generated poins are
    not MST
#define INPUT_ATTEMPTS 500000
// defines the screen as [0:1000][0:1000]
#define MAX_SCREEN 1000
// defines the jump of the next added point [x-100:x+100][y-100:y+100]
#define MAX_JUMP 100

bool areInline(Point a, Point b, Point c) {
    return a.x * (b.y - c.y) + b.x * (c.y - a.y) + c.x * (a.y - b.y) == 0;
}

bool isMSTPath(const vector<Point>& inputPoints, Point p, int n) {
    if (n < 2) return true;
    // checks if the point is already used
    if (p.isEquals(inputPoints[n-1])) return false;
    double newEdgeLength = p.distTo(inputPoints[n-1]);
    double maxEdgeLength = -1;
    for (int i = n-2; i>=0; --i) {
        double edgeLength = inputPoints[i].distTo(inputPoints[i+1]);
        maxEdgeLength = max(maxEdgeLength, edgeLength);
        // checks if the point is already used
        if (p.isEquals(inputPoints[i])) return false;
        // checks if the path after adding a new point is also the MST of the
            set of points
```

```
        if (edgeLength >= p.distTo(inputPoints[i]) && edgeLength >=
            p.distTo(inputPoints[i+1])) return false;
        if (p.distTo(inputPoints[i]) <= newEdgeLength) return false;
        if (newEdgeLength >= inputPoints[n-1].distTo(inputPoints[i]) &&
            newEdgeLength >= p.distTo(inputPoints[i])) return false;
        if (p.distTo(inputPoints[i]) <= maxEdgeLength) return false;
        // checks if points are in general positions
        if (areInline(p, inputPoints[i], inputPoints[i+1])) return false;
    }
    return true;
}

Point getHalfDistancePoint(Point source, Point target) {
    return Point(source.x + (target.x - source.x) / 2, source.y + (target.y -
        source.y) / 2);
}

vector<Point> generateInitialMST(int NUM_POINTS) {
    vector<Point> inputPoints(NUM_POINTS);
    int i = 0;
    while(i < NUM_POINTS) {
        Point p;
        if(i==0) p = Point::generateRandomPoint(0, MAX_SCREEN, 0, MAX_SCREEN);
        else p = Point::generateRandomPoint( max(0,inputPoints[i-1].x-MAX_JUMP),
            min(MAX_SCREEN, inputPoints[i-1].x+MAX_JUMP),
            max(0,inputPoints[i-1].y-MAX_JUMP), min(MAX_SCREEN,
            inputPoints[i-1].y+MAX_JUMP));
        if (i > 0) {
            while (p.isEquals(inputPoints[i-1])) {
                p = Point::generateRandomPoint(
                    max(0,inputPoints[i-1].x-MAX_JUMP), min(MAX_SCREEN,
                    inputPoints[i-1].x+MAX_JUMP),
                    max(0,inputPoints[i-1].y-MAX_JUMP), min(MAX_SCREEN,
                    inputPoints[i-1].y+MAX_JUMP));
            }
            int num_tries = 0;
            while (!isMSTPath(inputPoints, p, i)) {
                Point nextPoint = getHalfDistancePoint(inputPoints[i - 1], p);
                if (nextPoint.isEquals(p)) {
                    p = Point::generateRandomPoint(
                        max(0,inputPoints[i-1].x-MAX_JUMP), min(MAX_SCREEN,
                        inputPoints[i-1].x+MAX_JUMP),
                        max(0,inputPoints[i-1].y-MAX_JUMP), min(MAX_SCREEN,
                        inputPoints[i-1].y+MAX_JUMP));
                } else {
                    p = nextPoint;
                }
                if (num_tries < INPUT_ATTEMPTS) ++num_tries;
                else {
                    inputPoints.resize(i);
                    return inputPoints;
                }
            }
        }
        inputPoints[i++] = p;
```

```
    }
    return inputPoints;
}

bool fileExists(const std::string& fileName) {
     std::fstream file;
     file.open(fileName.c_str(), std::ios::in);
     if (file.is_open() == true)
     {
         file.close();
         return true;
     }
     file.close();
     return false;
}

int getNextTestIndex(int i) {
    while (fileExists("cases/" + boost::lexical_cast<std::string>(i) +
        "/input.dat")
        || fileExists("cases/" + boost::lexical_cast<std::string>(i) +
            "_INTERSECTING/input.dat")) ++i;
    return i;
}

int main(int argc, char* argv[]) {

    if(argc == 2 || argc == 3){
        int numIntersecting = 0;
        int numNOTIntersecting = 0;
        int NUM_POINTS = atoi(argv[1]);
        int NUM_CASES = 1;
        if(argc == 3)
            NUM_CASES = atoi(argv[2]);
        int case_index = getNextTestIndex(0);
        for (int i = 0; i < NUM_CASES; ++i) {
            string dir = "cases/" + boost::lexical_cast<std::string>(case_index);

            mkdir("cases", 0777);
            mkdir(dir.c_str(), 0777);

            string vertices_filename = dir + "/input.dat";
            string simple_polygon_filename = dir + "/resulting_cycle.dat";
            string mst_filename = dir + "/mst.dat";

            vector<Point> inputPoints = generateInitialMST(NUM_POINTS);
            Graph graph(inputPoints);
            graph.generateVerticesFile(vertices_filename);
            Graph mst = graph.createMSTGraph();
            Graph simplePolygon = graph.createSimplePolygon();
            simplePolygon.generateFile(simple_polygon_filename);
            mst.generateFile(mst_filename);
            if (simplePolygon.isIntersecting(mst)) {
                ++numIntersecting;
                graph.paintPolygon(vertices_filename, simple_polygon_filename,
                    mst_filename, MAX_SCREEN);
```

```
                mkdir((dir + "_INTERSECTING").c_str(), 0777);
                rename(vertices_filename.c_str(), (dir +
                    "_INTERSECTING/input.dat").c_str());
                rename(simple_polygon_filename.c_str(), (dir +
                    "_INTERSECTING/simple_polygon.dat").c_str());
                rename(mst_filename.c_str(), (dir +
                    "_INTERSECTING/mst.dat").c_str());
                rmdir(dir.c_str());
                cout << "Case " << case_index << "("<< inputPoints.size()
                    <<")"<< ": INTERSECTING" << endl;
            } else {
                ++numNOTIntersecting;
                cout << "Case " << case_index << "("<< inputPoints.size()
                    <<")"<< ": NOT INTERSECTING" << endl;
            }
            case_index = getNextTestIndex(case_index + 1);
        }

        cout<<endl;
        cout<<"SUMMARY"<<endl;
        cout<<"----------------------"<<endl;
        cout<<"Intersecting: "<<numIntersecting<<endl;
        cout<<"NOT Intersecting: "<<numNOTIntersecting<<endl;
    } else {
        cerr << "Invalid number of parameters: TFG (NUM_POINTS) (NUM_CASES)"
            <<endl;
    }
}
```

Listing 2: Graph.h

```
#ifndef GRAPH
#define GRAPH

#include <vector>
#include <queue>
#include "Point.h"

using namespace std;

class Graph {
    public:
        Graph(vector<Point> inputPoints);
        Graph(vector<Point> inputPoints, vector<pair<int, int> > inputEdges);
        ~Graph();
        vector<Point> getVertices();
        vector<pair<int, int> > getEdges();
        void printVertices();
        void generateVerticesFile(string fileName);
        void generateFile(string fileName);
        Graph createMSTGraph();
        Graph createSimplePolygon();
        bool isIntersecting(Graph& graph);
        static void paintPolygon(string points_file_name, string
            polygon_file_name, string mst_file_name, int screenSize);
```

```
    private:
        vector<Point> vertices;
        vector<pair<int, int> > edges;
        const static int INF = 100000000;
        queue<pair<int, int> > getAllCross(vector<Point>& vertices,
            vector<pair<int, int> >& edges);
        bool doIntersect(vector<Point>& vertices, pair<int, int> edge1,
            pair<int, int> edge2);
        bool onSegment(Point p, Point q, Point r);
        int orientation(Point p, Point q, Point r);
        void getCicleSize(int currentEdge, vector<bool>& visitedEdges,
            vector<pair<int, int> >& edges, int& cicle_size);
        void resolveCross(pair<int, int> cross, queue<pair<int, int> >&
            list_of_cross, vector<Point>& vertices, vector<pair<int, int> >&
            edges);
        static bool isSameEdge(pair<int, int> edge1, pair<int, int> edge2);
};

#endif // !
```

Listing 3: Graph.cpp

```
#include "Graph.h"
#include "gnuplot-iostream.h" //Gnuplot class handles POSIX-Pipe-communikation
    with Gnuplot
#include <iostream>
#include <queue>
#include <vector>
#include <algorithm>

Graph::Graph(vector<Point> inputPoints) {
    int num_points = inputPoints.size();
    vertices = inputPoints;
    edges = vector<pair<int,int> > (num_points);
    for (int i = 0; i < num_points; ++i) {
        edges[i] = pair<int,int> (i, (i + 1) % num_points);
    }
}

Graph::Graph(vector<Point> inputPoints, vector<pair<int, int> > inputEdges) {
    vertices = inputPoints;
    edges = inputEdges;
}

Graph::~Graph()
{
}

vector<Point> Graph::getVertices() {
    return vertices;
}

vector<pair<int, int> > Graph::getEdges() {
    return edges;
}
```

```
void Graph::printVertices() {
    if (vertices.size()) {
        printf("Points (%d):\n", (int)vertices.size());
        for (int i = 0; i < vertices.size(); ++i) {
            cout<<vertices[i].toString()<<endl;
        }
    }
    else {
        printf("There are no points.");
    }

}

void Graph::generateVerticesFile(string fileName) {
    ofstream input_data;
    input_data.open (fileName.c_str());

    int num_points = vertices.size();
    for(int i = 0; i < num_points; ++i) {
        input_data << vertices[i].x << " " << vertices[i].y << endl;
    }

    input_data.close();
}

void Graph::generateFile(string fileName) {
    ofstream input_data;
    input_data.open (fileName.c_str());

    int num_edges = edges.size();
    for(int i = 0; i < num_edges; ++i) {
        input_data << vertices[edges[i].first].x << " "
            << vertices[edges[i].first].y << endl
            << vertices[edges[i].second].x << " "
            << vertices[edges[i].second].y << endl << endl;
    }

    input_data.close();
}

Graph Graph::createMSTGraph() {
    int num_points = vertices.size();

    priority_queue< pair<double, pair<int, int> > > q;
    vector<double> dist(num_points, INF);
    vector<int> father(num_points, -1);
    for(int i = 0; i < num_points; ++i) {
        q.push(pair<double, pair<int, int> >(-dist[i], pair<int,int>(i, i)));
    }
    int first_node = q.top().second.first;
    dist[first_node] = 0;

    while( !q.empty() )  {
        int node = q.top().second.first;
```

```
        int nodes_father = q.top().second.second;
        q.pop();

        if (father[node] == -1) {
            father[node] = nodes_father;
            for(int i = 0; i < num_points; ++i) {
                if (i != node && father[i] == -1 && dist[i] >
                    vertices[i].sqDistTo(vertices[node])) {
                    dist[i] = vertices[i].sqDistTo(vertices[node]);
                    q.push(pair<double, pair<int, int> >(-dist[i],
                        pair<int,int>(i, node)));
                }
            }
        }
    }
    vector<pair<int, int> > edges(num_points - 1);
    for(int i = 0; i < num_points; ++i) {
        if ( i != father[i] ) {
            edges[i] = pair<int,int> (i, father[i]);
        }
    }

    return Graph(vertices, edges);
}

bool Graph::onSegment(Point p, Point q, Point r) {
    if (q.x < max(p.x, r.x) && q.x > min(p.x, r.x) &&
        q.y < max(p.y, r.y) && q.y > min(p.y, r.y))
        return true;

    return false;

}

int Graph::orientation(Point p, Point q, Point r) {
    // See http://www.geeksforgeeks.org/orientation-3-ordered-points/
    // for details of below formula.
    float val = (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y);

    if (val == 0) return 0;  // colinear

    return (val > 0)? 1: 2; // clock or counterclock wise
}

bool Graph::doIntersect(vector<Point>& vertices, pair<int, int> edge1, pair<int,
    int> edge2) {

    Point p1 = vertices[edge1.first];
    Point q1 = vertices[edge1.second];
    Point p2 = vertices[edge2.first];
    Point q2 = vertices[edge2.second];

    // Find the four orientations needed for general and
    // special cases
    float o1 = orientation(p1, q1, p2);
```

```
    float o2 = orientation(p1, q1, q2);
    float o3 = orientation(p2, q2, p1);
    float o4 = orientation(p2, q2, q1);

    // General case
    if (o1 != o2 && o3 != o4 && o1 != 0 && o2 != 0 && o3 != 0 && o4 != 0)
        return true;

    // Special Cases
    // p1, q1 and p2 are colinear and p2 lies on segment p1q1
    if (o1 == 0 && onSegment(p1, p2, q1)) return true;

    // p1, q1 and p2 are colinear and q2 lies on segment p1q1
    if (o2 == 0 && onSegment(p1, q2, q1)) return true;

    // p2, q2 and p1 are colinear and p1 lies on segment p2q2
    if (o3 == 0 && onSegment(p2, p1, q2)) return true;

    // p2, q2 and q1 are colinear and q1 lies on segment p2q2
    if (o4 == 0 && onSegment(p2, q1, q2)) return true;

    return false; // Doesn't fall in any of the above cases
}

void Graph::getCicleSize(int currentEdge, vector<bool>& visitedEdges,
    vector<pair<int, int> >& edges, int& cicle_size) {
    if (visitedEdges[currentEdge])
        return;
    visitedEdges[currentEdge] = true;
    cicle_size++;
    for (int i = 0; i < edges.size(); ++i) {
        if ((edges[i].first == edges[currentEdge].first || edges[i].first ==
            edges[currentEdge].second ||
            edges[i].second == edges[currentEdge].first || edges[i].second ==
                edges[currentEdge].second)
            && !visitedEdges[i]) {
            getCicleSize (i, visitedEdges, edges, cicle_size);
        }
    }
}

void Graph::resolveCross(pair<int, int> cross, queue<pair<int, int> >&
    list_of_cross, vector<Point>& vertices, vector<pair<int, int> >& edges) {
    int edge1_index = cross.first;
    int edge2_index = cross.second;
    pair<int, int> edge1 = edges[edge1_index];
    pair<int, int> edge2 = edges[edge2_index];

    int p1 = edge1.first;
    int q1 = edge1.second;
    int p2 = edge2.first;
    int q2 = edge2.second;

    if (doIntersect (vertices, edge1, edge2)) {
        edges[edge1_index] = pair<int, int> (p1, p2);
```

```
        edges[edge2_index] = pair<int, int> (q1, q2);

        vector<bool> visitedEdges(edges.size());
        int cicle_size = 0;
        getCicleSize (edge1_index, visitedEdges, edges, cicle_size);

        if (cicle_size < edges.size()) {
            edges[edge1_index] = pair<int, int> (p1, q2);
            edges[edge2_index] = pair<int, int> (q1, p2);
        }

        for (int i = 0; i <edges.size(); ++i) {
            if (edge1_index != i && doIntersect (vertices, edges[i],
                edges[edge1_index])) {
                list_of_cross.push (pair<int, int> (i, edge1_index));
            }
        }
        for (int i = 0; i <edges.size(); ++i) {
            if (edge2_index != i && doIntersect (vertices, edges[i],
                edges[edge2_index])) {
                list_of_cross.push (pair<int, int> (i, edge2_index));
            }
        }
    }
}

queue<pair<int, int> > Graph::getAllCross(vector<Point>& vertices,
    vector<pair<int, int> >& edges) {
    queue<pair<int, int> > list_of_cross;
    for(int i = 0; i < edges.size(); ++i) {
        for(int j = i + 1; j < edges.size(); ++j) {
            if (doIntersect (vertices, edges[i], edges[j])) {
                list_of_cross.push(pair<int, int> (i, j));
            }
        }
    }
    return list_of_cross;
}

Graph Graph::createSimplePolygon() {
    vector<pair<int,int> > new_edges(edges);
    queue<pair<int, int> > list_of_cross = getAllCross (vertices, new_edges);
    while (!list_of_cross.empty()) {
        pair<int, int> cross = list_of_cross.front();
        list_of_cross.pop();
        resolveCross (cross, list_of_cross, vertices, new_edges);
    }
    return Graph(vertices, new_edges);
}

bool Graph::isSameEdge(pair<int, int> edge1, pair<int, int> edge2) {
    return (edge1.first == edge2.first && edge1.second == edge2.second)
        || (edge1.second == edge2.first && edge1.first == edge2.second);
}
```

```
bool Graph::isIntersecting(Graph& graph) {
    vector <pair<int, int> > extra_edges = graph.getEdges();
    for (int i = 0; i < edges.size(); ++i) {
        for (int j = 0; j < extra_edges.size(); ++j) {
            pair<int, int> edge1 = edges[i];
            pair<int, int> edge2 = extra_edges[j];
            if (!isSameEdge(edge1, edge2) && doIntersect(vertices, edge1,
                edge2)) {
                return true;
            }
        }
    }
    return false;
}

void Graph::paintPolygon(string points_file_name, string polygon_file_name,
    string mst_file_name, int screenSize) {
    Gnuplot gp;
    gp << "set xrange [0:"<<screenSize<<"]\nset yrange [0:"<<screenSize<<"]\n";
    gp << "plot '"<< polygon_file_name <<"' using 1:2 with line lw 3 title
        'resulting cycle', "
        << "'"<< mst_file_name <<"' using 1:2 with line lw 2 lt 3 lc rgb
            \"green\" title 'MST', "
        << "'"<< points_file_name <<"' using 1:2:(3) with circles fill solid lc
            rgb \"red\" notitle, "
        << std::endl;
}
```

LISTING 4: Point.h

```
#ifndef POINT
#define POINT

#include <string>

using namespace std;

class Point {
    public:
        int x;
        int y;

    public:
        Point();
        Point(int x, int y);
        ~Point();
        static Point generateRandomPoint(int minX, int maxX, int minY, int maxY);
        string toString();
        bool isEquals(const Point & p) const;
        double sqDistTo(const Point & p) const;
        double distTo(const Point & p) const;
};

#endif // !
```

LISTING 5: Point.cpp

```
#include "Point.h"
#include <cstdlib>
#include <sys/time.h>
#include <string>
#include <sstream>
#include <cmath>

Point::Point() {
    x = 0;
    y = 0;
};

Point::Point(int x, int y) {
    this -> x = x;
    this -> y = y;
};

Point::~Point() {}

Point Point::generateRandomPoint(int minX, int maxX, int minY, int maxY){
    // getting current timestamp in microseconds
    struct timeval tp;
    gettimeofday(&tp, NULL);
    long int seed = tp.tv_sec * 1000000 + tp.tv_usec;

    srand(seed);
    int x = rand() % (maxX + 1 - minX) + minX;
    int y = rand() % (maxY + 1 - minY) + minY;
    return Point(x, y);
}

string Point::toString() {
    stringstream sstm;
    sstm << "( " << x << ", " << y << " )";
    return sstm.str();
}

bool Point::isEquals(const Point & p) const {
    return x == p.x && y == p.y;
}

double Point::sqDistTo(const Point & p) const {
    return (x - p.x) * (x - p.x) + (y - p.y) * (y - p.y);
}

double Point::distTo(const Point & p) const {
    return sqrt(this->sqDistTo(p));
}
```

LISTING 6: gnuplot-iostream.h

```
// vim:foldmethod=marker

/*
```

```
Copyright (c) 2013 Daniel Stahlke (dan@stahlke.org)

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
*/

/* A C++ interface to gnuplot.
* Web page: http://www.stahlke.org/dan/gnuplot-iostream
* Documentation: https://github.com/dstahlke/gnuplot-iostream/wiki
*
* The whole library consists of this monolithic header file, for ease of
    installation (the
* Makefile and *.cc files are only for examples and tests).
*
* TODO:
*   What version of boost is currently required?
*   Callbacks via gnuplot's 'bind' function.  This would allow triggering user
    functions when
*   keys are pressed in the gnuplot window.  However, it would require a PTY
    reader thread.
*   Maybe temporary files read in a thread can replace PTY stuff.
*/

#ifndef GNUPLOT_IOSTREAM_H
#define GNUPLOT_IOSTREAM_H

// {{{1 Includes and defines

#define GNUPLOT_IOSTREAM_VERSION 2

#ifndef GNUPLOT_ENABLE_CXX11
#   define GNUPLOT_ENABLE_CXX11 (__cplusplus >= 201103)
#endif

// C system includes
#include <cstdio>
#ifdef GNUPLOT_ENABLE_PTY
#   include <termios.h>
#   include <unistd.h>
#ifdef __APPLE__
```

```
#   include <util.h>
#else
#   include <pty.h>
#endif
#endif // GNUPLOT_ENABLE_PTY

// C++ system includes
#include <fstream>
#include <iostream>
#include <sstream>
#include <stdexcept>
#include <string>
#include <utility>
#include <iomanip>
#include <vector>
#include <complex>
#include <cstdlib>
#include <cmath>

#if GNUPLOT_ENABLE_CXX11
#   include <tuple>
#endif

#include <boost/iostreams/device/file_descriptor.hpp>
#include <boost/iostreams/stream.hpp>
#include <boost/version.hpp>
#include <boost/utility.hpp>
#include <boost/tuple/tuple.hpp>
#include <boost/mpl/bool.hpp>
// This is the version of boost which has v3 of the filesystem libraries by
    default.
#if BOOST_VERSION >= 104600
#   define GNUPLOT_USE_TMPFILE
#   include <boost/filesystem.hpp>
#endif // BOOST_VERSION

// This is used because VS2008 doesn't have stdint.h.
#include <boost/cstdint.hpp>

// Note: this is here for reverse compatibility.  The new way to enable blitz
    support is to
// just include the gnuplot-iostream.h header after you include the blitz header
    (likewise for
// armadillo).
#ifdef GNUPLOT_ENABLE_BLITZ
#   include <blitz/array.h>
#endif

#ifdef BOOST_STATIC_ASSERT_MSG
#   define GNUPLOT_STATIC_ASSERT_MSG(cond, msg) BOOST_STATIC_ASSERT_MSG((cond),
    msg)
#else
#   define GNUPLOT_STATIC_ASSERT_MSG(cond, msg) BOOST_STATIC_ASSERT((cond))
#endif
```

```
// If this is defined, warn about use of deprecated functions.
#ifdef GNUPLOT_DEPRECATE_WARN
#   ifdef __GNUC__
#       define GNUPLOT_DEPRECATE(msg) __attribute__ ((deprecated(msg)))
#   elif defined(_MSC_VER)
#       define GNUPLOT_DEPRECATE(msg) __declspec(deprecated(msg))
#   else
#       define GNUPLOT_DEPRECATE(msg)
#   endif
#else
#   define GNUPLOT_DEPRECATE(msg)
#endif

// Patch for Windows by Damien Loison
#ifdef _WIN32
#   include <windows.h>
#   define GNUPLOT_PCLOSE _pclose
#   define GNUPLOT_POPEN  _popen
#   define GNUPLOT_FILENO _fileno
#else
#   define GNUPLOT_PCLOSE pclose
#   define GNUPLOT_POPEN  popen
#   define GNUPLOT_FILENO fileno
#endif

#ifdef _WIN32
#   define GNUPLOT_ISNAN _isnan
#else
// cppreference.com says std::isnan is only for C++11.  However, this seems to
    work on Linux
// and I am assuming that if isnan exists in math.h then std::isnan exists in
    cmath.
#   define GNUPLOT_ISNAN std::isnan
#endif

// MSVC gives a warning saying that fopen and getenv are not secure.  But they
    are secure.
// Unfortunately their replacement functions are not simple drop-in
    replacements.  The best
// solution is to just temporarily disable this warning whenever fopen or getenv
    is used.
// http://stackoverflow.com/a/4805353/1048959
#if defined(_MSC_VER) && _MSC_VER >= 1400
#   define GNUPLOT_MSVC_WARNING_4996_PUSH \
        __pragma(warning(push)) \
        __pragma(warning(disable:4996))
#   define GNUPLOT_MSVC_WARNING_4996_POP \
        __pragma(warning(pop))
#else
#   define GNUPLOT_MSVC_WARNING_4996_PUSH
#   define GNUPLOT_MSVC_WARNING_4996_POP
#endif

#ifndef GNUPLOT_DEFAULT_COMMAND
#ifdef _WIN32
```

```
// "pgnuplot" is considered deprecated according to the Internet.  It may be
    faster.  It
// doesn't seem to handle binary data though.
//# define GNUPLOT_DEFAULT_COMMAND "pgnuplot -persist"
// On Windows, gnuplot echos commands to stderr.  So we forward its stderr to
    the bit bucket.
// Unfortunately, this means you will miss out on legitimate error messages.
#   define GNUPLOT_DEFAULT_COMMAND "gnuplot -persist 2> NUL"
#else
#   define GNUPLOT_DEFAULT_COMMAND "gnuplot -persist"
#endif
#endif

// }}}1

namespace gnuplotio {

    // {{{1 Basic traits helpers
    //
    // The mechanisms constructed in this section enable us to detect what sort
        of datatype has
    // been passed to a function.

    // This can be specialized as needed, in order to not use the STL interfaces
        for specific
    // classes.
    template <typename T>
    struct dont_treat_as_stl_container {
        typedef boost::mpl::bool_<false> type;
    };

    BOOST_MPL_HAS_XXX_TRAIT_DEF(value_type)
        BOOST_MPL_HAS_XXX_TRAIT_DEF(const_iterator)

        template <typename T>
    struct is_like_stl_container {
        typedef boost::mpl::and_<
            typename has_value_type<T>::type,
            typename has_const_iterator<T>::type,
            boost::mpl::not_<dont_treat_as_stl_container<T> >
        > type;
        static const bool value = type::value;
    };

    template <typename T>
    struct is_boost_tuple_nulltype {
        static const bool value = false;
        typedef boost::mpl::bool_<value> type;
    };

    template <>
    struct is_boost_tuple_nulltype<boost::tuples::null_type> {
        static const bool value = true;
        typedef boost::mpl::bool_<value> type;
    };
```

```
    BOOST_MPL_HAS_XXX_TRAIT_DEF(head_type)
        BOOST_MPL_HAS_XXX_TRAIT_DEF(tail_type)

        template <typename T>
    struct is_boost_tuple {
        typedef boost::mpl::and_<
            typename has_head_type<T>::type,
            typename has_tail_type<T>::type
        > type;
        static const bool value = type::value;
    };

    // More fine-grained, but doesn't compile!
    //template <typename T>
    //struct is_boost_tuple {
    //  typedef boost::mpl::and_<
    //      typename boost::is_class<T>::type,
    //      typename boost::mpl::and_<
    //          typename has_head_type<T>::type,
    //          typename boost::mpl::and_<
    //              typename has_tail_type<T>::type,
    //              typename boost::mpl::or_<
    //                  typename is_boost_tuple_nulltype<typename
        T::tail_type>::type,
    //                  typename is_boost_tuple<typename T::tail_type>::type
    //              >::type
    //          >::type
    //      >::type
    //  > type;
    //};
    //
    //template <>
    //struct is_boost_tuple<boost::tuples::null_type> {
    //  typedef boost::mpl::bool_<false> type;
    //};

    // }}}1

    // {{{1 Tmpfile helper class
#ifdef GNUPLOT_USE_TMPFILE
    // RAII temporary file.  File is removed when this object goes out of scope.
    class GnuplotTmpfile {
    public:
        GnuplotTmpfile() :
            file(boost::filesystem::unique_path(
                boost::filesystem::temp_directory_path() /
                "tmp-gnuplot-%%%%-%%%%-%%%%-%%%%"))
        { }

    private:
        // noncopyable
        GnuplotTmpfile(const GnuplotTmpfile &);
        const GnuplotTmpfile& operator=(const GnuplotTmpfile &);
```

```
    public:
        ~GnuplotTmpfile() {
            // it is never good to throw exceptions from a destructor
            try {
                remove(file);
            }
            catch (const std::exception &) {
                std::cerr << "Failed to remove temporary file " << file <<
                    std::endl;
            }
        }

    public:
        boost::filesystem::path file;
    };
#endif // GNUPLOT_USE_TMPFILE
    // }}}1

    // {{{1 Feedback helper classes
    //
    // Used for reading stuff sent from gnuplot via gnuplot's "print" function.
    //
    // For example, this is used for capturing mouse clicks in the gnuplot
        window.  There are two
    // implementations, only the first of which is complete.  The first
        implementation allocates a
    // PTY (pseudo terminal) which is written to by gnuplot and read by us.
        This only works in
    // Linux.  The second implementation creates a temporary file which is
        written to by gnuplot
    // and read by us.  However, this doesn't currently work since fscanf
        doesn't block.  It would
    // be possible to get this working using a more complicated mechanism
        (select or threads) but I
    // haven't had the need for it.

    class GnuplotFeedback {
    public:
        GnuplotFeedback() { }
        virtual ~GnuplotFeedback() { }
        virtual std::string filename() const = 0;
        virtual FILE *handle() const = 0;

    private:
        // noncopyable
        GnuplotFeedback(const GnuplotFeedback &);
        const GnuplotFeedback& operator=(const GnuplotFeedback &);
    };

#ifdef GNUPLOT_ENABLE_PTY
#define GNUPLOT_ENABLE_FEEDBACK
    class GnuplotFeedbackPty : public GnuplotFeedback {
    public:
        explicit GnuplotFeedbackPty(bool debug_messages) :
            pty_fn(),
```

```
            pty_fh(NULL),
            master_fd(-1),
            slave_fd(-1)
        {
            // adapted from http://www.gnuplot.info/files/gpReadMouseTest.c
            if (0 > openpty(&master_fd, &slave_fd, NULL, NULL, NULL)) {
                perror("openpty");
                throw std::runtime_error("openpty failed");
            }
            char pty_fn_buf[1024];
            if (ttyname_r(slave_fd, pty_fn_buf, 1024)) {
                perror("ttyname_r");
                throw std::runtime_error("ttyname failed");
            }
            pty_fn = std::string(pty_fn_buf);
            if (debug_messages) {
                std::cerr << "feedback_fn=" << pty_fn << std::endl;
            }

            // disable echo
            struct termios tios;
            if (tcgetattr(slave_fd, &tios) < 0) {
                perror("tcgetattr");
                throw std::runtime_error("tcgetattr failed");
            }
            tios.c_lflag &= ~(ECHO | ECHONL);
            if (tcsetattr(slave_fd, TCSAFLUSH, &tios) < 0) {
                perror("tcsetattr");
                throw std::runtime_error("tcsetattr failed");
            }

            pty_fh = fdopen(master_fd, "r");
            if (!pty_fh) {
                throw std::runtime_error("fdopen failed");
            }
        }

    private:
        // noncopyable
        GnuplotFeedbackPty(const GnuplotFeedbackPty &);
        const GnuplotFeedbackPty& operator=(const GnuplotFeedbackPty &);

    public:
        ~GnuplotFeedbackPty() {
            if (pty_fh) fclose(pty_fh);
            if (master_fd > 0) ::close(master_fd);
            if (slave_fd  > 0) ::close(slave_fd);
        }

        std::string filename() const {
            return pty_fn;
        }

        FILE *handle() const {
            return pty_fh;
```

```
        }

    private:
        std::string pty_fn;
        FILE *pty_fh;
        int master_fd, slave_fd;
    };
    //#elif defined GNUPLOT_USE_TMPFILE
    //// Currently this doesn't work since fscanf doesn't block (need something
        like "tail -f")
    //#define GNUPLOT_ENABLE_FEEDBACK
    //class GnuplotFeedbackTmpfile : public GnuplotFeedback {
    //public:
    //  explicit GnuplotFeedbackTmpfile(bool debug_messages) :
    //      tmp_file(),
    //      fh(NULL)
    //  {
    //      if(debug_messages) {
    //          std::cerr << "feedback_fn=" << filename() << std::endl;
    //      }
    //      GNUPLOT_MSVC_WARNING_4996_PUSH
    //      fh = std::fopen(filename().c_str(), "a");
    //      GNUPLOT_MSVC_WARNING_4996_POP
    //  }
    //
    //  ~GnuplotFeedbackTmpfile() {
    //      fclose(fh);
    //  }
    //
    //private:
    //  // noncopyable
    //  GnuplotFeedbackTmpfile(const GnuplotFeedbackTmpfile &);
    //  const GnuplotFeedbackTmpfile& operator=(const GnuplotFeedbackTmpfile &);
    //
    //public:
    //  std::string filename() const {
    //      return tmp_file.file.string();
    //  }
    //
    //  FILE *handle() const {
    //      return fh;
    //  }
    //
    //private:
    //  GnuplotTmpfile tmp_file;
    //  FILE *fh;
    //};
#endif // GNUPLOT_ENABLE_PTY, GNUPLOT_USE_TMPFILE
    // }}}1

    // {{{1 Traits and printers for entry datatypes
    //
    // This section contains the mechanisms for sending scalar and tuple data to
        gnuplot.  Pairs
```

```
// and tuples are sent by appealing to the senders defined for their
    component scalar types.
// Senders for arrays are defined in a later section.
//
// There are three classes which need to be specialized for each supported
    datatype:
// 1. TextSender to send data as text.  The default is to just send using
    the ostream's `<<`
// operator.
// 2. BinarySender to send data as binary, in a format which gnuplot can
    understand.  There is
// no default implementation (unimplemented types raise a compile time
    error), however
// inheriting from FlatBinarySender will send the data literally as it is
    stored in memory.
// This suffices for most of the standard built-in types (e.g. uint32_t or
    double).
// 3. BinfmtSender sends a description of the data format to gnuplot (e.g.
    `%uint32`).  Type
// `show datafile binary datasizes` in gnuplot to see a list of supported
    formats.

// {{{2 Basic entry datatypes

// Default TextSender, sends data using `<<` operator.
template <typename T, typename Enable = void>
struct TextSender {
    static void send(std::ostream &stream, const T &v) {
        stream << v;
    }
};

// Default BinarySender, raises a compile time error.
template <typename T, typename Enable = void>
struct BinarySender {
    GNUPLOT_STATIC_ASSERT_MSG((sizeof(T) == 0), "BinarySender class not
        specialized for this type");

    // This is here to avoid further compilation errors, beyond what the
        assert prints.
    static void send(std::ostream &stream, const T &v);
};

// This is a BinarySender implementation that just sends directly from
    memory.  Data types
// which can be sent this way can have their BinarySender specialization
    inherit from this.
template <typename T>
struct FlatBinarySender {
    static void send(std::ostream &stream, const T &v) {
        stream.write(reinterpret_cast<const char *>(&v), sizeof(T));
    }
};

// Default BinfmtSender, raises a compile time error.
```

```
template <typename T, typename Enable = void>
struct BinfmtSender {
    GNUPLOT_STATIC_ASSERT_MSG((sizeof(T) == 0), "BinfmtSender class not
        specialized for this type");

    // This is here to avoid further compilation errors, beyond what the
        assert prints.
    static void send(std::ostream &);
};

// BinfmtSender implementations for basic data types supported by gnuplot.
// Types from boost/cstdint.hpp are used because VS2008 doesn't have
    stdint.h.
template<> struct BinfmtSender< float> { static void send(std::ostream
    &stream) { stream << "%float"; } };
template<> struct BinfmtSender<double> { static void send(std::ostream
    &stream) { stream << "%double"; } };
template<> struct BinfmtSender<boost::int8_t> { static void
    send(std::ostream &stream) { stream << "%int8"; } };
template<> struct BinfmtSender<boost::uint8_t> { static void
    send(std::ostream &stream) { stream << "%uint8"; } };
template<> struct BinfmtSender<boost::int16_t> { static void
    send(std::ostream &stream) { stream << "%int16"; } };
template<> struct BinfmtSender<boost::uint16_t> { static void
    send(std::ostream &stream) { stream << "%uint16"; } };
template<> struct BinfmtSender<boost::int32_t> { static void
    send(std::ostream &stream) { stream << "%int32"; } };
template<> struct BinfmtSender<boost::uint32_t> { static void
    send(std::ostream &stream) { stream << "%uint32"; } };
template<> struct BinfmtSender<boost::int64_t> { static void
    send(std::ostream &stream) { stream << "%int64"; } };
template<> struct BinfmtSender<boost::uint64_t> { static void
    send(std::ostream &stream) { stream << "%uint64"; } };

// BinarySender implementations for basic data types supported by gnuplot.
    These types can
// just be sent as stored in memory, so all these senders inherit from
    FlatBinarySender.
template<> struct BinarySender< float> : public FlatBinarySender< float> { };
template<> struct BinarySender<double> : public FlatBinarySender<double> { };
template<> struct BinarySender<boost::int8_t> : public
    FlatBinarySender<boost::int8_t> { };
template<> struct BinarySender<boost::uint8_t> : public
    FlatBinarySender<boost::uint8_t> { };
template<> struct BinarySender<boost::int16_t> : public
    FlatBinarySender<boost::int16_t> { };
template<> struct BinarySender<boost::uint16_t> : public
    FlatBinarySender<boost::uint16_t> { };
template<> struct BinarySender<boost::int32_t> : public
    FlatBinarySender<boost::int32_t> { };
template<> struct BinarySender<boost::uint32_t> : public
    FlatBinarySender<boost::uint32_t> { };
template<> struct BinarySender<boost::int64_t> : public
    FlatBinarySender<boost::int64_t> { };
```

```
template<> struct BinarySender<boost::uint64_t> : public
    FlatBinarySender<boost::uint64_t> { };

// Make char types print as integers, not as characters.
template <typename T>
struct CastIntTextSender {
    static void send(std::ostream &stream, const T &v) {
        stream << int(v);
    }
};
template<> struct TextSender<          char> : public CastIntTextSender<
             char> { };
template<> struct TextSender<   signed char> : public CastIntTextSender<
    signed char> { };
template<> struct TextSender< unsigned char> : public CastIntTextSender<
    unsigned char> { };

// Make sure that the same not-a-number string is printed on all platforms.
template <typename T>
struct FloatTextSender {
    static void send(std::ostream &stream, const T &v) {
        if (GNUPLOT_ISNAN(v)) { stream << "nan"; }
        else { stream << v; }
    }
};
template<> struct TextSender<      float> : FloatTextSender<      float> { };
template<> struct TextSender<     double> : FloatTextSender<     double> { };
template<> struct TextSender<long double> : FloatTextSender<long double> { };

// }}}2

// {{{2 std::pair support

template <typename T, typename U>
struct TextSender<std::pair<T, U> > {
    static void send(std::ostream &stream, const std::pair<T, U> &v) {
        TextSender<T>::send(stream, v.first);
        stream << " ";
        TextSender<U>::send(stream, v.second);
    }
};

template <typename T, typename U>
struct BinfmtSender<std::pair<T, U> > {
    static void send(std::ostream &stream) {
        BinfmtSender<T>::send(stream);
        BinfmtSender<U>::send(stream);
    }
};

template <typename T, typename U>
struct BinarySender<std::pair<T, U> > {
    static void send(std::ostream &stream, const std::pair<T, U> &v) {
        BinarySender<T>::send(stream, v.first);
        BinarySender<U>::send(stream, v.second);
```

```
    }
};

// }}}2

// {{{2 std::complex support

template <typename T>
struct TextSender<std::complex<T> > {
    static void send(std::ostream &stream, const std::complex<T> &v) {
        TextSender<T>::send(stream, v.real());
        stream << " ";
        TextSender<T>::send(stream, v.imag());
    }
};

template <typename T>
struct BinfmtSender<std::complex<T> > {
    static void send(std::ostream &stream) {
        BinfmtSender<T>::send(stream);
        BinfmtSender<T>::send(stream);
    }
};

template <typename T>
struct BinarySender<std::complex<T> > {
    static void send(std::ostream &stream, const std::complex<T> &v) {
        BinarySender<T>::send(stream, v.real());
        BinarySender<T>::send(stream, v.imag());
    }
};

// }}}2

// {{{2 boost::tuple support

template <typename T>
struct TextSender<T,
    typename boost::enable_if<
    boost::mpl::and_<
    is_boost_tuple<T>,
    boost::mpl::not_<is_boost_tuple_nulltype<typename T::tail_type> >
    >
    >::type
> {
    static void send(std::ostream &stream, const T &v) {
        TextSender<typename T::head_type>::send(stream, v.get_head());
        stream << " ";
        TextSender<typename T::tail_type>::send(stream, v.get_tail());
    }
};

template <typename T>
struct TextSender<T,
    typename boost::enable_if<
```

```
    boost::mpl::and_<
    is_boost_tuple<T>,
    is_boost_tuple_nulltype<typename T::tail_type>
    >
    >::type
> {
    static void send(std::ostream &stream, const T &v) {
        TextSender<typename T::head_type>::send(stream, v.get_head());
    }
};

template <typename T>
struct BinfmtSender<T,
    typename boost::enable_if<
    boost::mpl::and_<
    is_boost_tuple<T>,
    boost::mpl::not_<is_boost_tuple_nulltype<typename T::tail_type> >
    >
    >::type
> {
    static void send(std::ostream &stream) {
        BinfmtSender<typename T::head_type>::send(stream);
        stream << " ";
        BinfmtSender<typename T::tail_type>::send(stream);
    }
};

template <typename T>
struct BinfmtSender<T,
    typename boost::enable_if<
    boost::mpl::and_<
    is_boost_tuple<T>,
    is_boost_tuple_nulltype<typename T::tail_type>
    >
    >::type
> {
    static void send(std::ostream &stream) {
        BinfmtSender<typename T::head_type>::send(stream);
    }
};

template <typename T>
struct BinarySender<T,
    typename boost::enable_if<
    boost::mpl::and_<
    is_boost_tuple<T>,
    boost::mpl::not_<is_boost_tuple_nulltype<typename T::tail_type> >
    >
    >::type
> {
    static void send(std::ostream &stream, const T &v) {
        BinarySender<typename T::head_type>::send(stream, v.get_head());
        BinarySender<typename T::tail_type>::send(stream, v.get_tail());
    }
};
```

```
    template <typename T>
    struct BinarySender<T,
        typename boost::enable_if<
        boost::mpl::and_<
        is_boost_tuple<T>,
        is_boost_tuple_nulltype<typename T::tail_type>
        >
        >::type
    > {
        static void send(std::ostream &stream, const T &v) {
            BinarySender<typename T::head_type>::send(stream, v.get_head());
        }
    };

    // }}}2

    // {{{2 std::tuple support

#if GNUPLOT_ENABLE_CXX11

    // http://stackoverflow.com/questions/6245735/pretty-print-stdtuple

    template<std::size_t> struct int_ {}; // compile-time counter

    template <typename Tuple, std::size_t I>
    void std_tuple_formatcode_helper(std::ostream &stream, const Tuple *,
        int_<I>) {
        std_tuple_formatcode_helper(stream, (const Tuple *)(0), int_<I - 1>());
        stream << " ";
        BinfmtSender<typename std::tuple_element<I, Tuple>::type>::send(stream);
    }

    template <typename Tuple>
    void std_tuple_formatcode_helper(std::ostream &stream, const Tuple *,
        int_<0>) {
        BinfmtSender<typename std::tuple_element<0, Tuple>::type>::send(stream);
    }

    template <typename... Args>
    struct BinfmtSender<std::tuple<Args...> > {
        typedef typename std::tuple<Args...> Tuple;

        static void send(std::ostream &stream) {
            std_tuple_formatcode_helper(stream, (const Tuple *)(0),
                int_<sizeof...(Args)-1>());
        }
    };

    template <typename Tuple, std::size_t I>
    void std_tuple_textsend_helper(std::ostream &stream, const Tuple &v,
        int_<I>) {
        std_tuple_textsend_helper(stream, v, int_<I - 1>());
        stream << " ";
```

```
        TextSender<typename std::tuple_element<I, Tuple>::type>::send(stream,
            std::get<I>(v));
    }

    template <typename Tuple>
    void std_tuple_textsend_helper(std::ostream &stream, const Tuple &v,
        int_<0>) {
        TextSender<typename std::tuple_element<0, Tuple>::type>::send(stream,
            std::get<0>(v));
    }

    template <typename... Args>
    struct TextSender<std::tuple<Args...> > {
        typedef typename std::tuple<Args...> Tuple;

        static void send(std::ostream &stream, const Tuple &v) {
            std_tuple_textsend_helper(stream, v, int_<sizeof...(Args)-1>());
        }
    };

    template <typename Tuple, std::size_t I>
    void std_tuple_binsend_helper(std::ostream &stream, const Tuple &v, int_<I>)
        {
        std_tuple_binsend_helper(stream, v, int_<I - 1>());
        BinarySender<typename std::tuple_element<I, Tuple>::type>::send(stream,
            std::get<I>(v));
    }

    template <typename Tuple>
    void std_tuple_binsend_helper(std::ostream &stream, const Tuple &v, int_<0>)
        {
        BinarySender<typename std::tuple_element<0, Tuple>::type>::send(stream,
            std::get<0>(v));
    }

    template <typename... Args>
    struct BinarySender<std::tuple<Args...> > {
        typedef typename std::tuple<Args...> Tuple;

        static void send(std::ostream &stream, const Tuple &v) {
            std_tuple_binsend_helper(stream, v, int_<sizeof...(Args)-1>());
        }
    };

#endif // GNUPLOT_ENABLE_CXX11

    // }}}2

    // }}}1

    // {{{1 ArrayTraits and Range classes
    //
    // This section handles sending of array data to gnuplot.  It is rather
        complicated because of
    // the diversity of storage schemes supported.  For example, it treats a
```

```
// `std::pair<std::vector<T>, std::vector<U>>` in the same way as a
// `std::vector<std::pair<T, U>>`, iterating through the two arrays in
    lockstep, and sending
// pairs <T,U> to gnuplot as columns.  In fact, any nested combination of
    pairs, tuples, STL
// containers, Blitz arrays, and Armadillo arrays is supported (with the
    caveat that, for
// instance, Blitz arrays should never be put into an STL container or you
    will suffer
// unpredictable results due the way Blitz handles assignment).  Nested
    containers are
// considered to be multidimensional arrays.  Although gnuplot only supports
    1D and 2D arrays,
// our module is in principle not limited.
//
// The ArrayTraits class is specialized for every supported array or
    container type (the
// default, unspecialized, version of ArrayTraits exists only to tell you
    that something is
// *not* a container, via the is_container flag).  ArrayTraits tells you the
    depth of a nested
// (or multidimensional) container, as well as the value type, and provides
    a specialized
// sort of iterator (a.k.a. "range").  Ranges are sort of like STL
    iterators, except that they
// have built-in knowledge of the end condition so you don't have to carry
    around both a
// begin() and an end() iterator like in STL.
//
// As an example of how this works, consider a std::pair of std::vectors.
    Ultimately this gets
// sent to gnuplot as two columns, so the two vectors need to be iterated in
    lockstep.
// The `value_type` of `std::pair<std::vector<T>, std::vector<U>>` is then
    `std::pair<T, U>`
// and this is what deferencing the range (iterator) gives.  Internally,
    this range is built
// out of a pair of ranges (PairOfRange class), the `inc()` (advance to next
    element)
// method calls `inc()` on each of the children, and `deref()` calls
    `deref()` on each child
// and combines the results to return a `std::pair`.  Tuples are handled as
    nested pairs.
//
// In addition to PairOfRange, there is also a VecOfRange class that can be
    used to treat the
// outermost part of a nested container as if it were a tuple.  Since tuples
    are printed as
// columns, this is like treating a multidimensional array as if it were
    column-major.  A
// VecOfRange is obtained by calling `get_columns_range`.  This is used by,
    for instance,
// `send1d_colmajor`.  The implementation is similar to that of PairOfRange.
//
```

```
// The range, accessed via `ArrayTraits<T>::get_range`, will be of a
    different class depending
// on T, and this is defined by the ArrayTraits specialization for T.  It
    will always have
// methods `inc()` to advance to the next element and `is_end()` for
    checking whether one has
// advanced past the final element.  For nested containers,
    `deref_subiter()` returns a range
// iterator for the next nesting level.  When at the innermost level of
    nesting, `deref()`
// returns the value of the entry the iterator points to (a scalar, pair, or
    tuple).
// Only one of `deref()` or `deref_subiter()` will be available, depending
    on whether there are
// deeper levels of nesting.  The typedefs `value_type` and `subiter_type`
    tell the return
// types of these two methods.
//
// Support for standard C++ and boost containers and tuples of containers is
    provided in this
// section.  Support for third party packages like Blitz and Armadillo is in
    a later section.

// {{{2 ArrayTraits generic class and defaults

// Error messages involving this stem from treating something that was not a
    container as if it
// was.  This is only here to allow compiliation without errors in normal
    cases.
struct Error_WasNotContainer {
    // This is just here to make VC++ happy.
    //
        https://connect.microsoft.com/VisualStudio/feedback/details/777612/class-template-sp
    typedef void subiter_type;
};

// Error messages involving this stem from calling deref instead of
    deref_subiter for a nested
// container.
struct Error_InappropriateDeref { };

// The unspecialized version of this class gives traits for things that are
    *not* arrays.
template <typename T, typename Enable = void>
class ArrayTraits {
public:
    // The value type of elements after all levels of nested containers have
        been dereferenced.
    typedef Error_WasNotContainer value_type;
    // The type of the range (a.k.a. iterator) that `get_range()` returns.
    typedef Error_WasNotContainer range_type;
    // Tells whether T is in fact a container type.
    static const bool is_container = false;
    // This flag supports the legacy behavior of automatically guessing
        whether the data should
```

```
        // be treated as column major.  This guessing happens when `send()` is
            called rather than
        // `send1d()` or `send2d()`.  This is deprecated, but is still supported
            for reverse
        // compatibility.
        static const bool allow_auto_unwrap = false;
        // The number of levels of nesting, or the dimension of multidimensional
            arrays.
        static const size_t depth = 0;

        // Returns the range (iterator) for an array.
        static range_type get_range(const T &) {
            GNUPLOT_STATIC_ASSERT_MSG((sizeof(T) == 0), "argument was not a
                container");
            throw std::logic_error("static assert should have been triggered by
                this point");
        }
    };

    // Most specializations of ArrayTraits should inherit from this (with V set
        to the value type).
    // It sets some default values.
    template <typename V>
    class ArrayTraitsDefaults {
    public:
        typedef V value_type;

        static const bool is_container = true;
        static const bool allow_auto_unwrap = true;
        static const size_t depth = ArrayTraits<V>::depth + 1;
    };

    // This handles reference types, such as are given with boost::tie.
    // It also allows for instance "ArrayTraits<T[N]>" to match "ArrayTraits<T
        (&) [N]>".
    // I think this is okay to do... The alternative is to use remove_reference
        all over the place.
    template <typename T>
    class ArrayTraits<T&> : public ArrayTraits<T> { };

    // FIXME - is this okay?
    // It supports gp.send1d(std::forward_as_tuple(x, std::move(y)));
#if GNUPLOT_ENABLE_CXX11
    template <typename T>
    class ArrayTraits<T&&> : public ArrayTraits<T> { };
#endif

    // }}}2

    // {{{2 STL container support

    template <typename TI, typename TV>
    class IteratorRange {
    public:
        IteratorRange() { }
```

```
    IteratorRange(const TI &_it, const TI &_end) : it(_it), end(_end) { }

    static const bool is_container = ArrayTraits<TV>::is_container;
    typedef typename boost::mpl::if_c<is_container,
        Error_InappropriateDeref, TV>::type value_type;
    typedef typename ArrayTraits<TV>::range_type subiter_type;

    bool is_end() const { return it == end; }

    void inc() { ++it; }

    value_type deref() const {
        GNUPLOT_STATIC_ASSERT_MSG(sizeof(TV) && !is_container,
            "deref called on nested container");
        if (is_end()) {
            throw std::runtime_error("attepted to dereference past end of
                iterator");
        }
        return *it;
    }

    subiter_type deref_subiter() const {
        GNUPLOT_STATIC_ASSERT_MSG(sizeof(TV) && is_container,
            "deref_subiter called on non-nested container");
        if (is_end()) {
            throw std::runtime_error("attepted to dereference past end of
                iterator");
        }
        return ArrayTraits<TV>::get_range(*it);
    }

private:
    TI it, end;
};

template <typename T>
class ArrayTraits<T,
    typename boost::enable_if<is_like_stl_container<T> >::type
> : public ArrayTraitsDefaults<typename T::value_type> {
public:
    typedef IteratorRange<typename T::const_iterator, typename
        T::value_type> range_type;

    static range_type get_range(const T &arg) {
        return range_type(arg.begin(), arg.end());
    }
};

// }}}2

// {{{2 C style array support

template <typename T, size_t N>
class ArrayTraits<T[N]> : public ArrayTraitsDefaults<T> {
public:
```

```
    typedef IteratorRange<const T*, T> range_type;

    static range_type get_range(const T(&arg)[N]) {
        return range_type(arg, arg + N);
    }
};

// }}}2

// {{{2 std::pair support

template <typename RT, typename RU>
class PairOfRange {
    template <typename T, typename U, typename PrintMode>
    friend void deref_and_print(std::ostream &, const PairOfRange<T, U> &,
        PrintMode);

public:
    PairOfRange() { }
    PairOfRange(const RT &_l, const RU &_r) : l(_l), r(_r) { }

    static const bool is_container = RT::is_container && RU::is_container;

    typedef std::pair<typename RT::value_type, typename RU::value_type>
        value_type;
    typedef PairOfRange<typename RT::subiter_type, typename
        RU::subiter_type> subiter_type;

    bool is_end() const {
        bool el = l.is_end();
        bool er = r.is_end();
        if (el != er) {
            throw std::length_error("columns were different lengths");
        }
        return el;
    }

    void inc() {
        l.inc();
        r.inc();
    }

    value_type deref() const {
        return std::make_pair(l.deref(), r.deref());
    }

    subiter_type deref_subiter() const {
        return subiter_type(l.deref_subiter(), r.deref_subiter());
    }

private:
    RT l;
    RU r;
};
```

```
template <typename T, typename U>
class ArrayTraits<std::pair<T, U> > {
public:
    typedef PairOfRange<typename ArrayTraits<T>::range_type, typename
        ArrayTraits<U>::range_type> range_type;
    typedef std::pair<typename ArrayTraits<T>::value_type, typename
        ArrayTraits<U>::value_type> value_type;
    static const bool is_container = ArrayTraits<T>::is_container &&
        ArrayTraits<U>::is_container;
    // Don't allow colwrap since it's already wrapped.
    static const bool allow_auto_unwrap = false;
    // It is allowed for l_depth != r_depth, for example one column could be
        'double' and the
    // other column could be 'vector<double>'.
    static const size_t l_depth = ArrayTraits<T>::depth;
    static const size_t r_depth = ArrayTraits<U>::depth;
    static const size_t depth = (l_depth < r_depth) ? l_depth : r_depth;

    static range_type get_range(const std::pair<T, U> &arg) {
        return range_type(
            ArrayTraits<T>::get_range(arg.first),
            ArrayTraits<U>::get_range(arg.second)
        );
    }
};

// }}}2

// {{{2 boost::tuple support

template <typename T>
class ArrayTraits<T,
    typename boost::enable_if<
    boost::mpl::and_<
    is_boost_tuple<T>,
    boost::mpl::not_<is_boost_tuple_nulltype<typename T::tail_type> >
    >
    >::type
> : public ArrayTraits<
    typename std::pair<
    typename T::head_type,
    typename T::tail_type
    >
> {
public:
    typedef typename T::head_type HT;
    typedef typename T::tail_type TT;

    typedef ArrayTraits<typename std::pair<HT, TT> > parent;

    static typename parent::range_type get_range(const T &arg) {
        return typename parent::range_type(
            ArrayTraits<HT>::get_range(arg.get_head()),
            ArrayTraits<TT>::get_range(arg.get_tail())
        );
```

```
        }
    };

    template <typename T>
    class ArrayTraits<T,
        typename boost::enable_if<
        boost::mpl::and_<
        is_boost_tuple<T>,
        is_boost_tuple_nulltype<typename T::tail_type>
        >
        >::type
    > : public ArrayTraits<
        typename T::head_type
    > {
        typedef typename T::head_type HT;

        typedef ArrayTraits<HT> parent;

    public:
        static typename parent::range_type get_range(const T &arg) {
            return parent::get_range(arg.get_head());
        }
    };

    // }}}2

    // {{{2 std::tuple support

#if GNUPLOT_ENABLE_CXX11

    template <typename Tuple, size_t idx>
    struct StdTupUnwinder {
        typedef std::pair<
            typename StdTupUnwinder<Tuple, idx - 1>::type,
            typename std::tuple_element<idx, Tuple>::type
        > type;

        static typename ArrayTraits<type>::range_type get_range(const Tuple
            &arg) {
            return typename ArrayTraits<type>::range_type(
                StdTupUnwinder<Tuple, idx - 1>::get_range(arg),
                ArrayTraits<typename std::tuple_element<idx,
                    Tuple>::type>::get_range(std::get<idx>(arg))
            );
        }
    };

    template <typename Tuple>
    struct StdTupUnwinder<Tuple, 0> {
        typedef typename std::tuple_element<0, Tuple>::type type;

        static typename ArrayTraits<type>::range_type get_range(const Tuple
            &arg) {
            return ArrayTraits<type>::get_range(std::get<0>(arg));
        }
```

```
    };

    template <typename... Args>
    class ArrayTraits<std::tuple<Args...> > :
        public ArrayTraits<typename StdTupUnwinder<std::tuple<Args...>,
            sizeof...(Args)-1>::type>
    {
        typedef std::tuple<Args...> Tuple;
        typedef ArrayTraits<typename StdTupUnwinder<Tuple,
            sizeof...(Args)-1>::type> parent;

    public:
        static typename parent::range_type get_range(const Tuple &arg) {
            return StdTupUnwinder<std::tuple<Args...>,
                sizeof...(Args)-1>::get_range(arg);
        }
    };

#endif // GNUPLOT_ENABLE_CXX11

    // }}}2

    // {{{2 Support column unwrap of container (VecOfRange)
    //
    // VecOfRange (created via `get_columns_range()`) treats the outermost level
        of a nested
    // container as if it were a tuple.  Since tuples are sent to gnuplot as
        columns, this has the
    // effect of addressing a multidimensional array in column major order.

    template <typename RT>
    class VecOfRange {
        template <typename T, typename PrintMode>
        friend void deref_and_print(std::ostream &, const VecOfRange<T> &,
            PrintMode);

    public:
        VecOfRange() { }
        explicit VecOfRange(const std::vector<RT> &_rvec) : rvec(_rvec) { }

        static const bool is_container = RT::is_container;
        // Don't allow colwrap since it's already wrapped.
        static const bool allow_auto_unwrap = false;

        typedef std::vector<typename RT::value_type> value_type;
        typedef VecOfRange<typename RT::subiter_type> subiter_type;

        bool is_end() const {
            if (rvec.empty()) return true;
            bool ret = rvec[0].is_end();
            for (size_t i = 1; i<rvec.size(); i++) {
                if (ret != rvec[i].is_end()) {
                    throw std::length_error("columns were different lengths");
                }
            }
```

```
            return ret;
        }

        void inc() {
            for (size_t i = 0; i<rvec.size(); i++) {
                rvec[i].inc();
            }
        }

        value_type deref() const {
            value_type ret(rvec.size());
            for (size_t i = 0; i<rvec.size(); i++) {
                ret[i] = rvec[i].deref();
            }
            return ret;
        }

        subiter_type deref_subiter() const {
            std::vector<typename RT::subiter_type> subvec(rvec.size());
            for (size_t i = 0; i<rvec.size(); i++) {
                subvec[i] = rvec[i].deref_subiter();
            }
            return subiter_type(subvec);
        }

    private:
        std::vector<RT> rvec;
    };

    template <typename T>
    VecOfRange<typename ArrayTraits<T>::range_type::subiter_type>
        get_columns_range(const T &arg) {
        typedef typename ArrayTraits<T>::range_type::subiter_type U;
        std::vector<U> rvec;
        typename ArrayTraits<T>::range_type outer =
            ArrayTraits<T>::get_range(arg);
        while (!outer.is_end()) {
            rvec.push_back(outer.deref_subiter());
            outer.inc();
        }
        VecOfRange<U> ret(rvec);
        return ret;
    }

    // }}}2

    // }}}1

    // {{{1 Array printing functions
    //
    // This section coordinates the sending of data to gnuplot.  The ArrayTraits
        mechanism tells us
    // about nested containers and provides iterators over them.  Here we make
        use of this,
```

```
// deciding what dimensions should be treated as rows, columns, or blocks,
    telling gnuplot the
// size of the array if needed, and so on.

// If this is set, then text-mode data will be sent in a format that is not
    compatible with
// gnuplot, but which helps the programmer tell what the library is
    thinking.  Basically it
// puts brackets around groups of items and puts a message delineating
    blocks of data.
static bool debug_array_print = 0;

// This is thrown when an empty container is being plotted.  This exception
    should always
// be caught and should not propagate to the user.
class plotting_empty_container : public std::length_error {
public:
    plotting_empty_container() : std::length_error("plotting empty
        container") { }
};

// {{{2 Tags (like enums for metaprogramming)

// These tags define what our goal is, what sort of thing should ultimately
    be sent to the
// ostream.  These tags are passed to the PrintMode template argument of the
    functions in this
// section.
//
// ModeText   - Sends the data in an array in text format
// ModeBinary - Sends the data in an array in binary format
// ModeBinfmt - Sends the gnuplot format code for binary data (e.g.
    "%double%double")
// ModeSize   - Sends the size of an array.  Needed when sending binary data.
struct ModeText { static const bool is_text = 1; static const bool is_binfmt
    = 0; static const bool is_size = 0; };
struct ModeBinary { static const bool is_text = 0; static const bool
    is_binfmt = 0; static const bool is_size = 0; };
struct ModeBinfmt { static const bool is_text = 0; static const bool
    is_binfmt = 1; static const bool is_size = 0; };
struct ModeSize { static const bool is_text = 0; static const bool is_binfmt
    = 0; static const bool is_size = 1; };

// Whether to treat the outermost level of a nested container as columns
    (column major mode).
struct ColUnwrapNo { };
struct ColUnwrapYes { };

// The user must give a hint to describe how nested containers are to be
    interpreted.  This is
// done by calling e.g. `send1d_colmajor()` or `send2d()`.  This hint is
    then described by the
// following tags.  This is passed to the OrganizationMode template argument.
struct Mode1D { static std::string class_name() { return "Mode1D"; } };
struct Mode2D { static std::string class_name() { return "Mode2D"; } };
```

```
struct Mode1DUnwrap { static std::string class_name() { return
    "Mode1DUnwrap"; } };
struct Mode2DUnwrap { static std::string class_name() { return
    "Mode2DUnwrap"; } };
// Support for the legacy behavior that guesses which of the above four
    modes should be used.
struct ModeAuto { static std::string class_name() { return "ModeAuto"; } };

// }}}2

// {{{2 ModeAutoDecoder
//
// ModeAuto guesses which of Mode1D, Mode2D, Mode1DUnwrap, or Mode2DUnwrap
    should be used.
// This is provided for reverse compatibility; it is better to specify
    explicitly which mode to
// use.  Since this is only for reverse compatibility, and shouldn't be
    used, I'm not going to
// spell out what the rules are.  See below for details.

template <typename T, typename Enable = void>
struct ModeAutoDecoder { };

template <typename T>
struct ModeAutoDecoder<T,
    typename boost::enable_if_c<
    (ArrayTraits<T>::depth == 1)
>::type>
{
    typedef Mode1D mode;
};

template <typename T>
struct ModeAutoDecoder<T,
    typename boost::enable_if_c<
    (ArrayTraits<T>::depth == 2) &&
    !ArrayTraits<T>::allow_auto_unwrap
>::type>
{
    typedef Mode2D mode;
};

template <typename T>
struct ModeAutoDecoder<T,
    typename boost::enable_if_c<
    (ArrayTraits<T>::depth == 2) &&
    ArrayTraits<T>::allow_auto_unwrap
>::type>
{
    typedef Mode1DUnwrap mode;
};

template <typename T>
struct ModeAutoDecoder<T,
    typename boost::enable_if_c<
```

```
    (ArrayTraits<T>::depth > 2) &&
    ArrayTraits<T>::allow_auto_unwrap
        >::type>
{
    typedef Mode2DUnwrap mode;
};

template <typename T>
struct ModeAutoDecoder<T,
    typename boost::enable_if_c<
    (ArrayTraits<T>::depth > 2) &&
    !ArrayTraits<T>::allow_auto_unwrap
        >::type>
{
    typedef Mode2D mode;
};

// }}}2

// The data is processed using several levels of functions that call each
    other in sequence,
// each defined in a subsection of code below.  Because C++ wants you to
    declare a function
// before using it, we begin with the innermost function.  So in order to
    see the sequence in
// which these are called, you should read the following subsections in
    reverse order.  Nested
// arrays are formated into blocks (for 2D data) and lines (for 1D or 2D
    data), then further
// nesting levels are formatted into columns.  Also tag dispatching is used
    in order to define
// various sorts of behavior.  Each of these tasks is handled by one of the
    following
// subsections.

// {{{2 send_scalar()
//
// Send a scalar in one of three possible ways: via TextSender,
    BinarySender, or BinfmtSender,
// depending on which PrintMode tag is passed to the function.

template <typename T>
void send_scalar(std::ostream &stream, const T &arg, ModeText) {
    TextSender<T>::send(stream, arg);
}

template <typename T>
void send_scalar(std::ostream &stream, const T &arg, ModeBinary) {
    BinarySender<T>::send(stream, arg);
}

template <typename T>
void send_scalar(std::ostream &stream, const T &, ModeBinfmt) {
    BinfmtSender<T>::send(stream);
}
```

```
// }}}2

// {{{2 deref_and_print()
//
// Dereferences and prints the given range (iterator).  At this point we are
    done with treating
// containers as blocks (for 2D data) and lines (for 1D or 2D data).  Any
    further levels of
// nested containers will at this point be treated as columns.

// If arg is not a container, then print it via send_scalar().
template <typename T, typename PrintMode>
typename boost::disable_if_c<T::is_container>::type
    deref_and_print(std::ostream &stream, const T &arg, PrintMode) {
    const typename T::value_type &v = arg.deref();
    send_scalar(stream, v, PrintMode());
}

// If arg is a container (but not a PairOfRange or VecOfRange, which are
    handled below) then
// treat the contents as columns, iterating over the contents recursively.
    If outputting in
// text mode, put a space between columns.
template <typename T, typename PrintMode>
typename boost::enable_if_c<T::is_container>::type
    deref_and_print(std::ostream &stream, const T &arg, PrintMode) {
    if (arg.is_end()) throw plotting_empty_container();
    typename T::subiter_type subrange = arg.deref_subiter();
    if (PrintMode::is_binfmt && subrange.is_end()) throw
        plotting_empty_container();
    if (debug_array_print && PrintMode::is_text) stream << "{";
    bool first = true;
    while (!subrange.is_end()) {
        if (!first && PrintMode::is_text) stream << " ";
        first = false;
        deref_and_print(stream, subrange, PrintMode());
        subrange.inc();
    }
    if (debug_array_print && PrintMode::is_text) stream << "}";
}

// PairOfRange is treated as columns.  In text mode, put a space between
    columns.
template <typename T, typename U, typename PrintMode>
void deref_and_print(std::ostream &stream, const PairOfRange<T, U> &arg,
    PrintMode) {
    deref_and_print(stream, arg.l, PrintMode());
    if (PrintMode::is_text) stream << " ";
    deref_and_print(stream, arg.r, PrintMode());
}

// VecOfRange is treated as columns.  In text mode, put a space between
    columns.
template <typename T, typename PrintMode>
```

```
void deref_and_print(std::ostream &stream, const VecOfRange<T> &arg,
    PrintMode) {
    if (PrintMode::is_binfmt && arg.rvec.empty()) throw
        plotting_empty_container();
    for (size_t i = 0; i<arg.rvec.size(); i++) {
        if (i && PrintMode::is_text) stream << " ";
        deref_and_print(stream, arg.rvec[i], PrintMode());
    }
}

// }}}2

// {{{2 print_block()
//
// Here we format nested containers into blocks (for 2D data) and lines.
    Actually, block and
// line formatting is only truely needed for text mode output, but for
    uniformity this function
// is also invoked in binary mode (the PrintMode tag determines the output
    mode).  If the goal
// is to just print the array size or the binary format string, then the
    loops exit after the
// first iteration.
//
// The Depth argument tells how deep to recurse.  It will be either `2` for
    2D data, formatted
// into blocks and lines, with empty lines between blocks, or `1` for 1D
    data formatted into
// lines but not blocks.  Gnuplot only supports 1D and 2D data, but if it
    were to support 3D in
// the future (e.g. volume rendering), all that would be needed would be
    some trivial changes
// in this section.  After Depth number of nested containers have been
    recursed into, control
// is passed to deref_and_print(), which treats any further nested
    containers as columns.

// Depth==1 and we are not asked to print the size of the array.  Send each
    element of the
// range to deref_and_print() for further processing into columns.
template <size_t Depth, typename T, typename PrintMode>
typename boost::enable_if_c<(Depth == 1) && !PrintMode::is_size>::type
    print_block(std::ostream &stream, T &arg, PrintMode) {
    if (PrintMode::is_binfmt && arg.is_end()) throw
        plotting_empty_container();
    for (; !arg.is_end(); arg.inc()) {
        //print_entry(arg.deref());
        deref_and_print(stream, arg, PrintMode());
        // If asked to print the binary format string, only the first
            element needs to be
        // looked at.
        if (PrintMode::is_binfmt) break;
        if (PrintMode::is_text) stream << std::endl;
    }
}
```

```
// Depth>1 and we are not asked to print the size of the array.  Loop over
    the range and
// recurse into print_block() with Depth -> Depth-1.
template <size_t Depth, typename T, typename PrintMode>
typename boost::enable_if_c<(Depth>1) && !PrintMode::is_size>::type
    print_block(std::ostream &stream, T &arg, PrintMode) {
    if (PrintMode::is_binfmt && arg.is_end()) throw
        plotting_empty_container();
    bool first = true;
    for (; !arg.is_end(); arg.inc()) {
        if (first) {
            first = false;
        }
        else {
            if (PrintMode::is_text) stream << std::endl;
        }
        if (debug_array_print && PrintMode::is_text) stream << "<block>" <<
            std::endl;
        if (arg.is_end()) throw plotting_empty_container();
        typename T::subiter_type sub = arg.deref_subiter();
        print_block<Depth - 1>(stream, sub, PrintMode());
        // If asked to print the binary format string, only the first
            element needs to be
        // looked at.
        if (PrintMode::is_binfmt) break;
    }
}

// Determine how many elements are in the given range.  Used in the
    functions below.
template <typename T>
size_t get_range_size(const T &arg) {
    // FIXME - not the fastest way.  Implement a size() method for range.
    size_t ret = 0;
    for (T i = arg; !i.is_end(); i.inc()) ++ret;
    return ret;
}

// Depth==1 and we are asked to print the size of the array.
template <size_t Depth, typename T, typename PrintMode>
typename boost::enable_if_c<(Depth == 1) && PrintMode::is_size>::type
    print_block(std::ostream &stream, T &arg, PrintMode) {
    stream << get_range_size(arg);
}

// Depth>1 and we are asked to print the size of the array.
template <size_t Depth, typename T, typename PrintMode>
typename boost::enable_if_c<(Depth>1) && PrintMode::is_size>::type
    print_block(std::ostream &stream, T &arg, PrintMode) {
    if (arg.is_end()) throw plotting_empty_container();
    // It seems that size for two dimensional arrays needs the fastest
        varying index first,
    // contrary to intuition.  The gnuplot documentation is not too clear on
        this point.
```

```
    typename T::subiter_type sub = arg.deref_subiter();
    print_block<Depth - 1>(stream, sub, PrintMode());
    stream << "," << get_range_size(arg);
}

// }}}2

// {{{2 handle_colunwrap_tag()
//
// If passed the ColUnwrapYes then treat the outermost nested container as
    columns by calling
// get_columns_range().  Otherwise just call get_range().  The range
    iterator is then passed to
// print_block() for further processing.

template <size_t Depth, typename T, typename PrintMode>
void handle_colunwrap_tag(std::ostream &stream, const T &arg, ColUnwrapNo,
    PrintMode) {
    GNUPLOT_STATIC_ASSERT_MSG(ArrayTraits<T>::depth >= Depth, "container not
        deep enough");
    typename ArrayTraits<T>::range_type range =
        ArrayTraits<T>::get_range(arg);
    print_block<Depth>(stream, range, PrintMode());
}

template <size_t Depth, typename T, typename PrintMode>
void handle_colunwrap_tag(std::ostream &stream, const T &arg, ColUnwrapYes,
    PrintMode) {
    GNUPLOT_STATIC_ASSERT_MSG(ArrayTraits<T>::depth >= Depth + 1, "container
        not deep enough");
    VecOfRange<typename ArrayTraits<T>::range_type::subiter_type> cols =
        get_columns_range(arg);
    print_block<Depth>(stream, cols, PrintMode());
}

// }}}2

// {{{2 handle_organization_tag()
//
// Parse the OrganizationMode tag then forward to handle_colunwrap_tag() for
    further
// processing.  If passed the Mode1D or Mode2D tags, then set Depth=1 or
    Depth=2.  If passed
// Mode{1,2}DUnwrap then use the ColUnwrapYes tag.  If passed ModeAuto
    (which is for legacy
// support) then use ModeAutoDecoder to guess which of Mode1D, Mode2D, etc.
    should be used.

template <typename T, typename PrintMode>
void handle_organization_tag(std::ostream &stream, const T &arg, Mode1D,
    PrintMode) {
    handle_colunwrap_tag<1>(stream, arg, ColUnwrapNo(), PrintMode());
}

template <typename T, typename PrintMode>
```

```
void handle_organization_tag(std::ostream &stream, const T &arg, Mode2D,
    PrintMode) {
    handle_colunwrap_tag<2>(stream, arg, ColUnwrapNo(), PrintMode());
}

template <typename T, typename PrintMode>
void handle_organization_tag(std::ostream &stream, const T &arg,
    Mode1DUnwrap, PrintMode) {
    handle_colunwrap_tag<1>(stream, arg, ColUnwrapYes(), PrintMode());
}

template <typename T, typename PrintMode>
void handle_organization_tag(std::ostream &stream, const T &arg,
    Mode2DUnwrap, PrintMode) {
    handle_colunwrap_tag<2>(stream, arg, ColUnwrapYes(), PrintMode());
}

template <typename T, typename PrintMode>
void handle_organization_tag(std::ostream &stream, const T &arg, ModeAuto,
    PrintMode) {
    handle_organization_tag(stream, arg, typename
        ModeAutoDecoder<T>::mode(), PrintMode());
}

// }}}2

// The entry point for the processing defined in this section.  It just
    forwards immediately to
// handle_organization_tag().  This function is only here to give a sane
    name to the entry
// point.
//
// The allowed values for the OrganizationMode and PrintMode tags are
    defined in the beginning
// of this section.
template <typename T, typename OrganizationMode, typename PrintMode>
void top_level_array_sender(std::ostream &stream, const T &arg,
    OrganizationMode, PrintMode) {
    handle_organization_tag(stream, arg, OrganizationMode(), PrintMode());
}

// }}}1

// {{{1 FileHandleWrapper

// This holds the file handle that gnuplot commands will be sent to.  The
    purpose of this
// wrapper is twofold:
// 1. It allows storing the FILE* before it gets passed to the
    boost::iostreams::stream
//    constructor (which is a base class of the main Gnuplot class).  This
    is accomplished
//    via multiple inheritance as described at
    http://stackoverflow.com/a/3821756/1048959
```

```
// 2. It remembers whether the handle needs to be closed via fclose or
    pclose.
struct FileHandleWrapper {
    FileHandleWrapper(std::FILE *_fh, bool _should_use_pclose) :
        wrapped_fh(_fh), should_use_pclose(_should_use_pclose) { }

    void fh_close() {
        if (should_use_pclose) {
            if (GNUPLOT_PCLOSE(wrapped_fh)) {
                std::cerr << "pclose returned error" << std::endl;
            }
        }
        else {
            if (fclose(wrapped_fh)) {
                std::cerr << "fclose returned error" << std::endl;
            }
        }
    }

    int fh_fileno() {
        return GNUPLOT_FILENO(wrapped_fh);
    }

    std::FILE *wrapped_fh;
    bool should_use_pclose;
};

// }}}1

// {{{1 Main class

class Gnuplot :
    // Some setup needs to be done before obtaining the file descriptor that
        gets passed to
    // boost::iostreams::stream.  This is accomplished by using a multiple
        inheritance trick,
    // as described at http://stackoverflow.com/a/3821756/1048959
    private FileHandleWrapper,
    public boost::iostreams::stream<boost::iostreams::file_descriptor_sink>
{
private:
    static std::string get_default_cmd() {
        GNUPLOT_MSVC_WARNING_4996_PUSH
            char *from_env = std::getenv("GNUPLOT_IOSTREAM_CMD");
        GNUPLOT_MSVC_WARNING_4996_POP
            if (from_env && from_env[0]) {
                return from_env;
            }
            else {
                return GNUPLOT_DEFAULT_COMMAND;
            }
    }

    static FileHandleWrapper open_cmdline(const std::string &in) {
        std::string cmd = in.empty() ? get_default_cmd() : in;
```

```
            assert(!cmd.empty());
            if (cmd[0] == '>') {
                std::string fn = cmd.substr(1);
                GNUPLOT_MSVC_WARNING_4996_PUSH
                    FILE *fh = std::fopen(fn.c_str(), "w");
                GNUPLOT_MSVC_WARNING_4996_POP
                    if (!fh) throw(std::ios_base::failure("cannot open file " +
                        fn));
                return FileHandleWrapper(fh, false);
            }
            else {
                FILE *fh = GNUPLOT_POPEN(cmd.c_str(), "w");
                if (!fh) throw(std::ios_base::failure("cannot open pipe " +
                    cmd));
                return FileHandleWrapper(fh, true);
            }
        }

    public:
        explicit Gnuplot(const std::string &_cmd = "") :
            FileHandleWrapper(open_cmdline(_cmd)),
            boost::iostreams::stream<boost::iostreams::file_descriptor_sink>(
                fh_fileno(),
#if BOOST_VERSION >= 104400
                boost::iostreams::never_close_handle
#else
                false
#endif
                ),
            feedback(NULL),
            tmp_files(),
            debug_messages(false)
        {
            *this << std::scientific << std::setprecision(18);  // refer
                <iomanip>
        }

        explicit Gnuplot(FILE *_fh) :
            FileHandleWrapper(_fh, 0),
            boost::iostreams::stream<boost::iostreams::file_descriptor_sink>(
                fh_fileno(),
#if BOOST_VERSION >= 104400
                boost::iostreams::never_close_handle
#else
                false
#endif
                ),
            feedback(NULL),
            tmp_files(),
            debug_messages(false)
        {
            *this << std::scientific << std::setprecision(18);  // refer
                <iomanip>
        }
```

```
    private:
        // noncopyable
        Gnuplot(const Gnuplot &);
        const Gnuplot& operator=(const Gnuplot &);

    public:
        ~Gnuplot() {
            if (debug_messages) {
                std::cerr << "ending gnuplot session" << std::endl;
            }

            // FIXME - boost's close method calls close() on the file
                descriptor, but we need to
            // use sometimes use pclose instead.  For now, just skip calling
                boost's close and use
            // flush just in case.
            do_flush();
            // Wish boost had a pclose method...
            //close();

            fh_close();

            delete feedback;
        }

        void clearTmpfiles() {
            // destructors will cause deletion
            tmp_files.clear();
        }

    private:
        void do_flush() {
            *this << std::flush;
            fflush(wrapped_fh);
        }

        std::string make_tmpfile() {
#ifdef GNUPLOT_USE_TMPFILE
            boost::shared_ptr<GnuplotTmpfile> tmp_file(new GnuplotTmpfile());
            // The file will be removed once the pointer is removed from the
            // tmp_files container.
            tmp_files.push_back(tmp_file);
            return tmp_file->file.string();
#else
            throw(std::logic_error("no filename given and temporary files not
                enabled"));
#endif // GNUPLOT_USE_TMPFILE
        }

    public:
        // {{{2 Generic sender routines.
        //
        // These are declared public, but are undocumented.  It is recommended
            to use the functions in
```

```
// the next section, which serve as adapters that pass specific values
    for the OrganizationMode
// tag.

template <typename T, typename OrganizationMode>
Gnuplot &send(const T &arg, OrganizationMode) {
    top_level_array_sender(*this, arg, OrganizationMode(), ModeText());
    *this << "e" << std::endl; // gnuplot's "end of array" token
    do_flush(); // probably not really needed, but doesn't hurt
    return *this;
}

template <typename T, typename OrganizationMode>
Gnuplot &sendBinary(const T &arg, OrganizationMode) {
    top_level_array_sender(*this, arg, OrganizationMode(), ModeBinary());
    do_flush(); // probably not really needed, but doesn't hurt
    return *this;
}

template <typename T, typename OrganizationMode>
std::string binfmt(const T &arg, const std::string &arr_or_rec,
    OrganizationMode) {
    assert((arr_or_rec == "array") || (arr_or_rec == "record"));
    std::string ret;
    try {
        std::ostringstream tmp;
        tmp << " format='";
        top_level_array_sender(tmp, arg, OrganizationMode(),
            ModeBinfmt());
        tmp << "' " << arr_or_rec << "=(";
        top_level_array_sender(tmp, arg, OrganizationMode(), ModeSize());
        tmp << ")";
        tmp << " ";
        ret = tmp.str();
    }
    catch (const plotting_empty_container &) {
        ret = std::string(" format='' ") + arr_or_rec + "=(0) ";
    }
    return ret;
}

// NOTE: empty filename makes temporary file
template <typename T, typename OrganizationMode>
std::string file(const T &arg, std::string filename, OrganizationMode) {
    if (filename.empty()) filename = make_tmpfile();
    std::fstream tmp_stream(filename.c_str(), std::fstream::out);
    top_level_array_sender(tmp_stream, arg, OrganizationMode(),
        ModeText());
    tmp_stream.close();

    std::ostringstream cmdline;
    // FIXME - hopefully filename doesn't contain quotes or such...
    cmdline << " '" << filename << "' ";
    return cmdline.str();
}
```

```
// NOTE: empty filename makes temporary file
template <typename T, typename OrganizationMode>
std::string binaryFile(const T &arg, std::string filename, const
    std::string &arr_or_rec, OrganizationMode) {
    if (filename.empty()) filename = make_tmpfile();
    std::fstream tmp_stream(filename.c_str(), std::fstream::out |
        std::fstream::binary);
    top_level_array_sender(tmp_stream, arg, OrganizationMode(),
        ModeBinary());
    tmp_stream.close();

    std::ostringstream cmdline;
    // FIXME - hopefully filename doesn't contain quotes or such...
    cmdline << " '" << filename << "' binary" << binfmt(arg, arr_or_rec,
        OrganizationMode());
    return cmdline.str();
}

// }}}2

// {{{2 Deprecated data sending interface that guesses an appropriate
    OrganizationMode.  This is here
// for reverse compatibility.  Don't use it.  A warning will be printed
    if
// GNUPLOT_DEPRECATE_WARN is defined.

template <typename T> Gnuplot GNUPLOT_DEPRECATE("use send1d or send2d")
    &send(const T &arg) { return send(arg, ModeAuto()); }

template <typename T> std::string GNUPLOT_DEPRECATE("use binfmt1d or
    binfmt2d")
    binfmt(const T &arg, const std::string &arr_or_rec = "array")
{
    return binfmt(arg, arr_or_rec, ModeAuto());
}

template <typename T> Gnuplot GNUPLOT_DEPRECATE("use sendBinary1d or
    sendBinary2d")
    &sendBinary(const T &arg) { return sendBinary(arg, ModeAuto()); }

template <typename T> std::string GNUPLOT_DEPRECATE("use file1d or
    file2d")
    file(const T &arg, const std::string &filename = "")
{
    return file(arg, filename, ModeAuto());
}

template <typename T> std::string GNUPLOT_DEPRECATE("use binArr1d or
    binArr2d")
    binaryFile(const T &arg, const std::string &filename = "", const
        std::string &arr_or_rec = "array")
{
    return binaryFile(arg, filename, arr_or_rec, ModeAuto());
}
```

```
// }}}2

// {{{2 Public (documented) data sending interface.
//
// It seems odd to define 16 different functions, but I think this ends
    up being the most
// convenient in terms of usage by the end user.

template <typename T> Gnuplot &send1d(const T &arg) { return send(arg,
    Mode1D()); }
template <typename T> Gnuplot &send2d(const T &arg) { return send(arg,
    Mode2D()); }
template <typename T> Gnuplot &send1d_colmajor(const T &arg) { return
    send(arg, Mode1DUnwrap()); }
template <typename T> Gnuplot &send2d_colmajor(const T &arg) { return
    send(arg, Mode2DUnwrap()); }

template <typename T> Gnuplot &sendBinary1d(const T &arg) { return
    sendBinary(arg, Mode1D()); }
template <typename T> Gnuplot &sendBinary2d(const T &arg) { return
    sendBinary(arg, Mode2D()); }
template <typename T> Gnuplot &sendBinary1d_colmajor(const T &arg) {
    return sendBinary(arg, Mode1DUnwrap()); }
template <typename T> Gnuplot &sendBinary2d_colmajor(const T &arg) {
    return sendBinary(arg, Mode2DUnwrap()); }

template <typename T> std::string file1d(const T &arg, const std::string
    &filename = "") { return file(arg, filename, Mode1D()); }
template <typename T> std::string file2d(const T &arg, const std::string
    &filename = "") { return file(arg, filename, Mode2D()); }
template <typename T> std::string file1d_colmajor(const T &arg, const
    std::string &filename = "") { return file(arg, filename,
    Mode1DUnwrap()); }
template <typename T> std::string file2d_colmajor(const T &arg, const
    std::string &filename = "") { return file(arg, filename,
    Mode2DUnwrap()); }

template <typename T> std::string binFmt1d(const T &arg, const
    std::string &arr_or_rec) { return binfmt(arg, arr_or_rec, Mode1D());
    }
template <typename T> std::string binFmt2d(const T &arg, const
    std::string &arr_or_rec) { return binfmt(arg, arr_or_rec, Mode2D());
    }
template <typename T> std::string binFmt1d_colmajor(const T &arg, const
    std::string &arr_or_rec) { return binfmt(arg, arr_or_rec,
    Mode1DUnwrap()); }
template <typename T> std::string binFmt2d_colmajor(const T &arg, const
    std::string &arr_or_rec) { return binfmt(arg, arr_or_rec,
    Mode2DUnwrap()); }

template <typename T> std::string binFile1d(const T &arg, const
    std::string &arr_or_rec, const std::string &filename = "") { return
    binaryFile(arg, filename, arr_or_rec, Mode1D()); }
```

```
        template <typename T> std::string binFile2d(const T &arg, const
            std::string &arr_or_rec, const std::string &filename = "") { return
            binaryFile(arg, filename, arr_or_rec, Mode2D()); }
        template <typename T> std::string binFile1d_colmajor(const T &arg, const
            std::string &arr_or_rec, const std::string &filename = "") { return
            binaryFile(arg, filename, arr_or_rec, Mode1DUnwrap()); }
        template <typename T> std::string binFile2d_colmajor(const T &arg, const
            std::string &arr_or_rec, const std::string &filename = "") { return
            binaryFile(arg, filename, arr_or_rec, Mode2DUnwrap()); }

        // }}}2

#ifdef GNUPLOT_ENABLE_FEEDBACK
    public:
        // Input variables are set to the mouse position and button.  If the
            gnuplot
        // window is closed, button -1 is returned.  The msg parameter is the
            prompt
        // that is printed to the console.
        void getMouse(
            double &mx, double &my, int &mb,
            std::string msg = "Click Mouse!"
        ) {
            allocFeedback();

            *this << "set mouse" << std::endl;
            *this << "pause mouse \"" << msg << "\\n\"" << std::endl;
            *this << "if (exists(\"MOUSE_X\")) print MOUSE_X, MOUSE_Y,
                MOUSE_BUTTON; else print 0, 0, -1;" << std::endl;
            if (debug_messages) {
                std::cerr << "begin scanf" << std::endl;
            }
            if (3 != fscanf(feedback->handle(), "%50lf %50lf %50d", &mx, &my,
                &mb)) {
                throw std::runtime_error("could not parse reply");
            }
            if (debug_messages) {
                std::cerr << "end scanf" << std::endl;
            }
        }

    private:
        void allocFeedback() {
            if (!feedback) {
#ifdef GNUPLOT_ENABLE_PTY
                feedback = new GnuplotFeedbackPty(debug_messages);
                //#elif defined GNUPLOT_USE_TMPFILE
                //// Currently this doesn't work since fscanf doesn't block
                    (need something like "tail -f")
                //          feedback = new
                    GnuplotFeedbackTmpfile(debug_messages);
#else
                // This shouldn't happen because we are in an `#ifdef
                    GNUPLOT_ENABLE_FEEDBACK`
```

```
                // block which should only be activated if GNUPLOT_ENABLE_PTY is
                    defined.
                GNUPLOT_STATIC_ASSERT_MSG((sizeof(T) == 0), "No feedback
                    mechanism defined.");
#endif
                *this << "set print \"" << feedback->filename() << "\"" <<
                    std::endl;
            }
        }
#endif // GNUPLOT_ENABLE_FEEDBACK

    private:
        GnuplotFeedback *feedback;
#ifdef GNUPLOT_USE_TMPFILE
        std::vector<boost::shared_ptr<GnuplotTmpfile> > tmp_files;
#else
        // just a placeholder
        std::vector<int> tmp_files;
#endif // GNUPLOT_USE_TMPFILE

    public:
        bool debug_messages;
    };

    // }}}1

} // namespace gnuplotio

  // The first version of this library didn't use namespaces, and now this must
      be here forever
  // for reverse compatibility.
using gnuplotio::Gnuplot;

#endif // GNUPLOT_IOSTREAM_H

// {{{1 Support for 3rd party array libraries

// {{{2 Blitz support

// This is outside of the main header guard so that it will be compiled when
    people do
// something like this:
//    #include "gnuplot-iostream.h"
//    #include <blitz/array.h>
//    #include "gnuplot-iostream.h"
// Note that it has its own header guard to avoid double inclusion.

#ifdef BZ_BLITZ_H
#ifndef GNUPLOT_BLITZ_SUPPORT_LOADED
#define GNUPLOT_BLITZ_SUPPORT_LOADED
namespace gnuplotio {

    template <typename T, int N>
    struct BinfmtSender<blitz::TinyVector<T, N> > {
        static void send(std::ostream &stream) {
```

```
        for (int i = 0; i<N; i++) {
            BinfmtSender<T>::send(stream);
        }
    }
};

template <typename T, int N>
struct TextSender<blitz::TinyVector<T, N> > {
    static void send(std::ostream &stream, const blitz::TinyVector<T, N> &v)
        {
        for (int i = 0; i<N; i++) {
            if (i) stream << " ";
            TextSender<T>::send(stream, v[i]);
        }
    }
};

template <typename T, int N>
struct BinarySender<blitz::TinyVector<T, N> > {
    static void send(std::ostream &stream, const blitz::TinyVector<T, N> &v)
        {
        for (int i = 0; i<N; i++) {
            BinarySender<T>::send(stream, v[i]);
        }
    }
};

class Error_WasBlitzPartialSlice { };

template <typename T, int ArrayDim, int SliceDim>
class BlitzIterator {
public:
    BlitzIterator() : p(NULL) { }
    BlitzIterator(
        const blitz::Array<T, ArrayDim> *_p,
        const blitz::TinyVector<int, ArrayDim> _idx
    ) : p(_p), idx(_idx) { }

    typedef Error_WasBlitzPartialSlice value_type;
    typedef BlitzIterator<T, ArrayDim, SliceDim - 1> subiter_type;
    static const bool is_container = true;

    // FIXME - it would be nice to also handle one-based arrays
    bool is_end() const {
        return idx[ArrayDim - SliceDim] == p->shape()[ArrayDim - SliceDim];
    }

    void inc() {
        ++idx[ArrayDim - SliceDim];
    }

    value_type deref() const {
        GNUPLOT_STATIC_ASSERT_MSG((sizeof(T) == 0), "cannot deref a blitz
            slice");
```

```
            throw std::logic_error("static assert should have been triggered by
                this point");
        }

        subiter_type deref_subiter() const {
            return BlitzIterator<T, ArrayDim, SliceDim - 1>(p, idx);
        }

    private:
        const blitz::Array<T, ArrayDim> *p;
        blitz::TinyVector<int, ArrayDim> idx;
    };

    template <typename T, int ArrayDim>
    class BlitzIterator<T, ArrayDim, 1> {
    public:
        BlitzIterator() : p(NULL) { }
        BlitzIterator(
            const blitz::Array<T, ArrayDim> *_p,
            const blitz::TinyVector<int, ArrayDim> _idx
        ) : p(_p), idx(_idx) { }

        typedef T value_type;
        typedef Error_WasNotContainer subiter_type;
        static const bool is_container = false;

        // FIXME - it would be nice to also handle one-based arrays
        bool is_end() const {
            return idx[ArrayDim - 1] == p->shape()[ArrayDim - 1];
        }

        void inc() {
            ++idx[ArrayDim - 1];
        }

        value_type deref() const {
            return (*p)(idx);
        }

        subiter_type deref_subiter() const {
            GNUPLOT_STATIC_ASSERT_MSG((sizeof(T) == 0), "argument was not a
                container");
            throw std::logic_error("static assert should have been triggered by
                this point");
        }

    private:
        const blitz::Array<T, ArrayDim> *p;
        blitz::TinyVector<int, ArrayDim> idx;
    };

    template <typename T, int ArrayDim>
    class ArrayTraits<blitz::Array<T, ArrayDim> > : public
        ArrayTraitsDefaults<T> {
    public:
```

```
        static const bool allow_auto_unwrap = false;
        static const size_t depth = ArrayTraits<T>::depth + ArrayDim;

        typedef BlitzIterator<T, ArrayDim, ArrayDim> range_type;

        static range_type get_range(const blitz::Array<T, ArrayDim> &arg) {
            blitz::TinyVector<int, ArrayDim> start_idx;
            start_idx = 0;
            return range_type(&arg, start_idx);
        }
    };

} // namespace gnuplotio
#endif // GNUPLOT_BLITZ_SUPPORT_LOADED
#endif // BZ_BLITZ_H

  // }}}2

  // {{{2 Armadillo support

  // This is outside of the main header guard so that it will be compiled when
      people do
  // something like this:
  //    #include "gnuplot-iostream.h"
  //    #include <armadillo>
  //    #include "gnuplot-iostream.h"
  // Note that it has its own header guard to avoid double inclusion.

#ifdef ARMA_INCLUDES
#ifndef GNUPLOT_ARMADILLO_SUPPORT_LOADED
#define GNUPLOT_ARMADILLO_SUPPORT_LOADED
namespace gnuplotio {

    template <typename T> struct dont_treat_as_stl_container<arma::Row  <T> > {
        typedef boost::mpl::bool_<true> type; };
    template <typename T> struct dont_treat_as_stl_container<arma::Col  <T> > {
        typedef boost::mpl::bool_<true> type; };
    template <typename T> struct dont_treat_as_stl_container<arma::Mat  <T> > {
        typedef boost::mpl::bool_<true> type; };
    template <typename T> struct dont_treat_as_stl_container<arma::Cube <T> > {
        typedef boost::mpl::bool_<true> type; };
    template <typename T> struct dont_treat_as_stl_container<arma::field<T> > {
        typedef boost::mpl::bool_<true> type; };

    // {{{3 Cube

    template <typename T>
    class ArrayTraits<arma::Cube<T> > : public ArrayTraitsDefaults<T> {
        class SliceRange {
        public:
            SliceRange() : p(NULL), col(0), slice(0) { }
            explicit SliceRange(const arma::Cube<T> *_p, size_t _row, size_t
                _col) :
                p(_p), row(_row), col(_col), slice(0) { }
```

```
    typedef T value_type;
    typedef Error_WasNotContainer subiter_type;
    static const bool is_container = false;

    bool is_end() const { return slice == p->n_slices; }

    void inc() { ++slice; }

    value_type deref() const {
        return (*p)(row, col, slice);
    }

    subiter_type deref_subiter() const {
        GNUPLOT_STATIC_ASSERT_MSG((sizeof(T) == 0), "argument was not a
            container");
        throw std::logic_error("static assert should have been triggered
            by this point");
    }

private:
    const arma::Cube<T> *p;
    size_t row, col, slice;
};

class ColRange {
public:
    ColRange() : p(NULL), row(0), col(0) { }
    explicit ColRange(const arma::Cube<T> *_p, size_t _row) :
        p(_p), row(_row), col(0) { }

    typedef T value_type;
    typedef SliceRange subiter_type;
    static const bool is_container = true;

    bool is_end() const { return col == p->n_cols; }

    void inc() { ++col; }

    value_type deref() const {
        GNUPLOT_STATIC_ASSERT_MSG((sizeof(T) == 0), "can't call deref on
            an armadillo cube col");
        throw std::logic_error("static assert should have been triggered
            by this point");
    }

    subiter_type deref_subiter() const {
        return subiter_type(p, row, col);
    }

private:
    const arma::Cube<T> *p;
    size_t row, col;
};

class RowRange {
```

```
        public:
            RowRange() : p(NULL), row(0) { }
            explicit RowRange(const arma::Cube<T> *_p) : p(_p), row(0) { }

            typedef T value_type;
            typedef ColRange subiter_type;
            static const bool is_container = true;

            bool is_end() const { return row == p->n_rows; }

            void inc() { ++row; }

            value_type deref() const {
                GNUPLOT_STATIC_ASSERT_MSG((sizeof(T) == 0), "can't call deref on
                    an armadillo cube row");
                throw std::logic_error("static assert should have been triggered
                    by this point");
            }

            subiter_type deref_subiter() const {
                return subiter_type(p, row);
            }

        private:
            const arma::Cube<T> *p;
            size_t row;
        };

    public:
        static const bool allow_auto_unwrap = false;
        static const size_t depth = ArrayTraits<T>::depth + 3;

        typedef RowRange range_type;

        static range_type get_range(const arma::Cube<T> &arg) {
            //std::cout << arg.n_elem << "," << arg.n_rows << "," << arg.n_cols
                << std::endl;
            return range_type(&arg);
        }
    };

    // }}}3

    // {{{3 Mat and Field

    template <typename RF, typename T>
    class ArrayTraits_ArmaMatOrField : public ArrayTraitsDefaults<T> {
        class ColRange {
        public:
            ColRange() : p(NULL), row(0), col(0) { }
            explicit ColRange(const RF *_p, size_t _row) :
                p(_p), row(_row), col(0) { }

            typedef T value_type;
            typedef Error_WasNotContainer subiter_type;
```

```
            static const bool is_container = false;

            bool is_end() const { return col == p->n_cols; }

            void inc() { ++col; }

            value_type deref() const {
                return (*p)(row, col);
            }

            subiter_type deref_subiter() const {
                GNUPLOT_STATIC_ASSERT_MSG((sizeof(T) == 0), "argument was not a
                    container");
                throw std::logic_error("static assert should have been triggered
                    by this point");
            }

        private:
            const RF *p;
            size_t row, col;
        };

        class RowRange {
        public:
            RowRange() : p(NULL), row(0) { }
            explicit RowRange(const RF *_p) : p(_p), row(0) { }

            typedef T value_type;
            typedef ColRange subiter_type;
            static const bool is_container = true;

            bool is_end() const { return row == p->n_rows; }

            void inc() { ++row; }

            value_type deref() const {
                GNUPLOT_STATIC_ASSERT_MSG((sizeof(T) == 0), "can't call deref on
                    an armadillo matrix row");
                throw std::logic_error("static assert should have been triggered
                    by this point");
            }

            subiter_type deref_subiter() const {
                return subiter_type(p, row);
            }

        private:
            const RF *p;
            size_t row;
        };

    public:
        static const bool allow_auto_unwrap = false;
        static const size_t depth = ArrayTraits<T>::depth + 2;
```

```
    typedef RowRange range_type;

    static range_type get_range(const RF &arg) {
        //std::cout << arg.n_elem << "," << arg.n_rows << "," << arg.n_cols
            << std::endl;
        return range_type(&arg);
    }
};

template <typename T>
class ArrayTraits<arma::field<T> > : public
    ArrayTraits_ArmaMatOrField<arma::field<T>, T> { };

template <typename T>
class ArrayTraits<arma::Mat<T> > : public
    ArrayTraits_ArmaMatOrField<arma::Mat<T>, T> { };

// }}}3

// {{{3 Row

template <typename T>
class ArrayTraits<arma::Row<T> > : public ArrayTraitsDefaults<T> {
public:
    static const bool allow_auto_unwrap = false;

    typedef IteratorRange<typename arma::Row<T>::const_iterator, T>
        range_type;

    static range_type get_range(const arma::Row<T> &arg) {
        //std::cout << arg.n_elem << "," << arg.n_rows << "," << arg.n_cols
            << std::endl;
        return range_type(arg.begin(), arg.end());
    }
};

// }}}3

// {{{3 Col

template <typename T>
class ArrayTraits<arma::Col<T> > : public ArrayTraitsDefaults<T> {
public:
    static const bool allow_auto_unwrap = false;

    typedef IteratorRange<typename arma::Col<T>::const_iterator, T>
        range_type;

    static range_type get_range(const arma::Col<T> &arg) {
        //std::cout << arg.n_elem << "," << arg.n_rows << "," << arg.n_cols
            << std::endl;
        return range_type(arg.begin(), arg.end());
    }
};
```

```
    // }}}3

} // namespace gnuplotio
#endif // GNUPLOT_ARMADILLO_SUPPORT_LOADED
#endif // ARMA_INCLUDES

  // }}}2

  // }}}1
```

LISTING 7: Makefile

```
LIBS = -lboost_filesystem -lboost_system -lboost_thread -lboost_iostreams
CC=g++
PROJECT_NAME=TFG

all::   Main

#GraphModule.o: GraphModule.cpp
#   $(CC) -c GraphModule.cpp $(CFLAGS) $(DEFINES) $<

#GPSexercice: GraphModule.o main.cpp
#   $(CC) main.cpp GraphModule.o -o GPSexercice $(CFLAGS) $(LIBS)

Point.o: Point.cpp Point.h
    $(CC) -c Point.cpp

Graph.o: Graph.cpp Graph.h Point.h
    $(CC) -c Graph.cpp

Main: Graph.o Point.o main.cpp
    $(CC) main.cpp Graph.o Point.o -o $(PROJECT_NAME) $(LIBS)

clean:
    rm -f *.o Main
```

Appendix B: Code of the Graphical application

The code and the applications for Android (.apk) and Windows (.exe) can be found in the following url:

https://drive.google.com/open?id=0B38snCnf398OS0NuN3pEQnl3dzA