Robot path planning using interval analysis

Student: Hadi Jaber

IASE 2012

Supervisor: Dr. Luc Jaulin

ENSTA Bretagne/OSM

Contents

Acknowledgements

I want to thank my supervisor Dr. Luc Jaulin, for all his comments. He has offered much advice and encouragement that was a great source of comfort to a new graduate student.

Also I want to thank all of my friends who made studying at ENSTA Bretagne enjoyable and sociable, specially Benoit Delauney and Olivier Debant.

Introduction

In the past two decades, many advances have been made in the field of robotics. For a robot to be autonomous, it must make its own decisions to achieve its goal. One major component of an autonomous robot is its path planner. This is the component responsible for getting the robot from one location or state to another while avoiding obstacles.

Path planning is used in many fields other than robotics. Transport engineers use traffic models that plan various paths through a traffic network to model the flow of traffic. Games also require some form of path planning to move computer controlled players around the game environment. Both applications require a fast path planner, since multiple paths often need to be planned at given time instances.

The time and space complexity of robotic path planning algorithms must also be low, since in most cases embedded processors have limited computational power.
Path planners are usually required to give the shortest collision free path, however, other objectives such as danger or fuel consumption could also be used. Another common objective is to cover an entire area. This objective is used by domestic robots that clean floors or by security robots that have to periodically patrol a given area.

For any mobile device, the ability to navigate in its environment is one of the most important capabilities of all. Staying operational, i.e. avoiding dangerous situations such as collisions and staying within safe operating conditions (temperature), radiation, exposure to weather, etc.) However if any tasks are to be performed that relate to specific places in the robot environment, navigation is a must. In the following, we will present an overview of the skill of navigation and try to identify the basic blocks of a robot navigation system, types of navigation systems, and closer look at its related building components.

Robot Navigation

Robot navigation means its ability to determine its own position in its frame of reference and then to plan a path towards some goal location. In order to navigate in its environment, the robot or any another mobility device requires representation i.e. a map of the environment and the ability to interpret that representation

Robot localization denotes the robot's ability to establish its own position and orientation within the frame of reference. Path planning is effectively an extension of localization, in that it requires the determination of the robot's current position and a position of a goal location, both within the same frame of reference or coordinates. Map building can be in the shape of a metric map or any notation describing locations in the robot frame of reference.

In robotic navigation, path planning is aimed at getting the optimum collision-free path between a starting and target locations. The planned path is usually decomposed into line segments between ordered sub-goals or way points. In the navigation phase, the robot follows those line segments toward the target. The navigation environment is usually represented in a data structure called the "configuration space". Depending on the surrounding environment and the running conditions, the optimality criterion for the path is determined. For example, in most of indoor navigation environments, the optimum path is the safest one, i.e. being as far as possible from the surrounding obstacles, whereas for outdoor navigation, the shortest path is more recommended.

Piano Movers' Problem

"Given an open subset U in n-dimensional space and two compact subsets C_0 and C_1 of U, where C_1 is derived from C_0 by a continuous motion, is it possible to move C_0 to C_1 while remaining entirely inside U?"
This is what the "piano mover's problem" is all about according to Wolfram Mathworld website (See [2]).

The goal is to find a path that moves the robot from the initial to the final position, while avoiding obstacles at all times.
The given of this problem:

i. A set of obstacles
ii. The initial position of a robot
iii. The final position of a robot

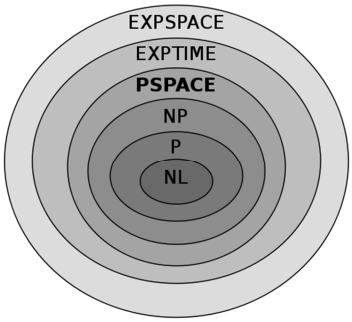
As it turns out, the problem is very complex and many solutions were and still are developed. The desired solution is a computer application that gets the target point as input and automatically plan and control the robot motion until it gets there.

State of the art

The robot path planning problem, which asks for the computation of collision free paths in environments containing obstacles, has received a great deal of attention in the last decades. In the basic problem, there is one robot present in a static and known environment, and the task is to compute a collision-free path describing a motion that brings the robot from its current position to some desired goal position. Variations and extensions of this basic problem statement are numerous.

In order to build robots that can autonomously act in real-life environments, path planning problems as sketched above need to be solved. However, it has been proven that, in general, solving even the basic path planning problem requires time exponential with the robots number of degrees of freedom. In spite of this discouraging problem complexity, various complete planners have been proposed. Their high complexity, however, makes them impractical for most applications. And every extension of the basic path planning problem adds in computational complexity.

For example, if we have n robots of d degrees of freedom each, the complexity becomes exponential in n*d. Or if we allow for moving obstacles, the problem becomes exponential in their number. Most motion planning problems are PSPACE-hard.



Assuming uncertainties in the robots sensing and control, leads to an exponential dependency on the complexity of the obstacles.
The above bounds deal with the exact problem, and therefore apply to complete planners. These are planners that solve any solvable problem, and return failure for each non-solvable one. So for most practical problems it seems impossible to use such complete planners. This has lead many researchers to consider simplifications of the problem statement.

Most of the work assumes the robot has a complete and accurate model of its environment before it begins to move.

Algebraic approach

This approach uses explicit representation of obstacles, using complicated algebra (Visibility computations/projections) to find the path.

This method is complete, but impractical.

Probabilistic approach

This approach generates randomly robot configurations (nodes), and discards nodes that are invalid. In addition, it connects pairs of nodes to form roadmap and discards paths that are invalid.

This approach doesn't work as well for some problems, it is hard to connect nodes on constraint surfaces, and may not terminate when no path exists.

Approaches based on potential functions

Many approaches to solve this problem are based on the use of potential functions, introduced by Khatib[5]. In the potential field approach, the obstacles to be avoided are represented by a repulsive potential, and the goal is represented by an attractive potential. According to the force generated by the sum of these potential fields, the object is expected to reach (if the method does not stop at any local minimum) its goal configuration without colliding with obstacles.

Interval Analysis approach

Other approaches based on the subdivision of the C-space have also been considered. They partition the C-space with a set of non-overlapping boxes, those that have been proved to be inside S, those that have been proved to be outside S and those for which nothing has been proved.

There are methods used to decide if a box is inside or outside the feasible configuration space S, however they are not based on interval analysis and are limited to a small class of problems and they meet some difficulties with orientation parameters.

Interval analysis is able to prove that a given box is inside or outside S for a huge class of problems. These methods for the path planning problem were introduced by Jaulin [1].

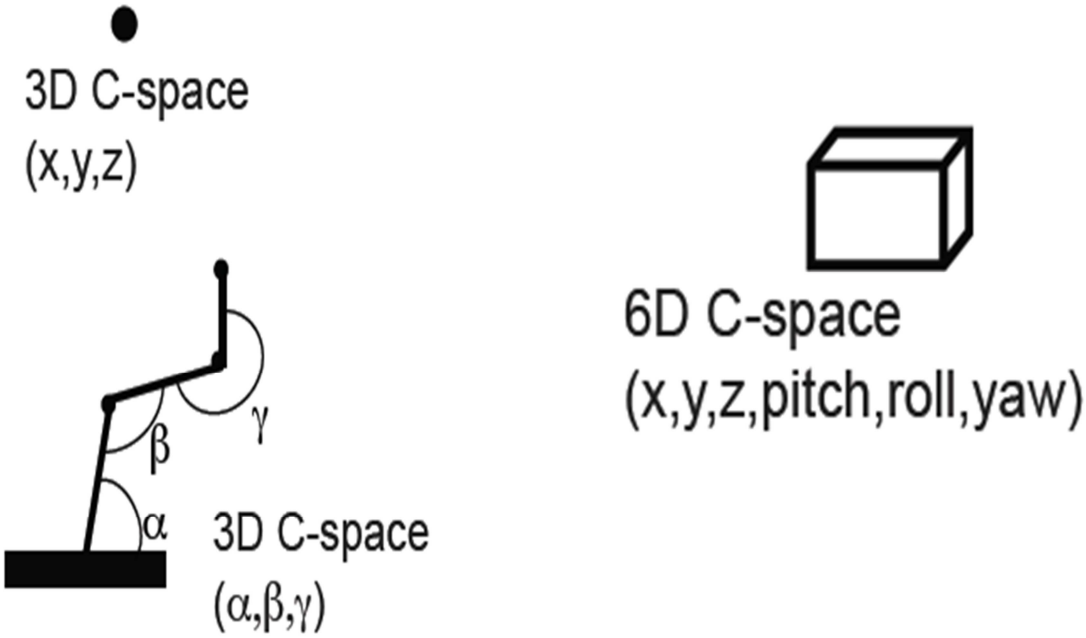
Our objective in this project is to implement this algorithm with Qt, which permits the portability and inter-operability of the implementation on various systems. This project would later be embedded in a Amadeus card to test on a robot with three degrees of freedom.

Configuration space

Each coordinate of the C-space represents a degree of freedom of the object. The number of independent parameters needed to specify an object configuration corresponds to the dimension of the C-space.

The start configuration and the goal configuration become two points A and B of the C-space.

The difficulty to solve the motion planning problem depends on the number of degrees of freedom (dimension of configuration space), and the geometric complexity.



Set Inversion Via Interval Analysis Algorithm

This algorithm was invented by Dr. Luc Jaulin see[1]

$$\text{FEASIBLEPATH1}\left([t], \vec{a}, \vec{b}, [\vec{p}_0], \varepsilon\right)$$

If $[t](\vec{a}) \neq 1$ or $[t](\vec{b}) \neq 1$, return ("Error: $\vec{a}$ and $\vec{b}$ should be feasible");

If $\vec{a} \notin [\vec{p}_0]$ or $\vec{b} \notin [\vec{p}_0]$, return ("Error: $\vec{a}$ and $\vec{b}$ should belong to $[\vec{p}_0]$");

Stack $= \{[\vec{p}_0]\}$; $\Delta\mathcal{P} = \emptyset$; $\mathcal{P}^- = \emptyset$;

While Stack $\neq \emptyset$;

Pop into $[\vec{p}]$;

If $[t]([\vec{p}]) = 1$, $\mathcal{P}^- = \mathcal{P}^- \cup \{[\vec{p}]\}$;

If $[t]([\vec{p}]) = [0,1]$ and width$([\vec{p}]) \leq \varepsilon$, $\Delta\mathcal{P} = \Delta\mathcal{P} \cup \{[\vec{p}]\}$;

If $[t]([\vec{p}]) = [0,1]$ and width$([\vec{p}]) > \varepsilon$,

Bisect$([\vec{p}])$ and stack the two resulting boxes;

EndWhile;

$\mathcal{P}^+ = \mathcal{P}^- \cup \Delta\mathcal{P}$; $\mathcal{G}^+ = \text{Graph}(\mathcal{P}^+)$; $\mathcal{G}^- = \text{Graph}(\mathcal{P}^-)$;

$v_a = \text{vertex}([\vec{p}_a])$, where $[\vec{p}_a] \in \mathcal{P}^+$ and $\vec{a} \in [\vec{p}_a]$;

$v_b = \text{vertex}([\vec{p}_b])$, where $[\vec{p}_b] \in \mathcal{P}^+$ and $\vec{b} \in [\vec{p}_b]$;

$\mathcal{L}^+ = \text{SHORTESTPATH}\ (\mathcal{G}^+, v_a, v_b)$; If $\mathcal{L}^+ = \emptyset$, return ("No path");

If $v_a \notin \mathcal{G}^-$ or $v_b \notin \mathcal{G}^-$, return ("Failure");

$\mathcal{L}^- = \text{SHORTESTPATH}\ (\mathcal{G}^-, v_a, v_b)$; If $\mathcal{L}^- \neq \emptyset$, return $\mathcal{L}^-$ else return ("Failure");

Cameleon algorithm

The motivations behind this algorithm are: the computing time of the graph algorithms are low compare to that of evaluating the inclusion tests [t] and to reduce the bisection of boxes. So Cameleon puts efforts to bisect and analyze zones of the C-space only when needed.

$$\text{FEASIBLEPATH2}\left([t], \vec{a}, \vec{b}, [\vec{p}_0]\right)$$

If $[t](\vec{a}) \neq 1$ or $[t](\vec{b}) \neq 1$, return ("Error: $\vec{a}$ and $\vec{b}$ should be feasible");

If $\vec{a} \notin [\vec{p}_0]$ or $\vec{b} \notin [\vec{p}_0]$, return ("Error: $\vec{a}$ and $\vec{b}$ should belong to $[\vec{p}_0]$");

Denote by $\mathcal{P}$ the paving containing the single box $[\vec{p}_0]$;

Repeat

$\mathcal{P}^+ = \text{Subpaving}(\mathcal{P}, 1 \in [t]([\vec{p}]))$; $\mathcal{G}^+ = \text{Graph}(\mathcal{P}^+)$;

$v_a = \text{vertex}([\vec{p}_a])$, where $[\vec{p}_a] \in \mathcal{P}^+$ and $\vec{a} \in [\vec{p}_a]$;

$v_b = \text{vertex}([\vec{p}_b])$, where $[\vec{p}_b] \in \mathcal{P}^+$ and $\vec{b} \in [\vec{p}_b]$;

$\mathcal{L}^+ = \text{SHORTESTPATH}\ (\mathcal{G}^+, v_a, v_b)$;

If $\mathcal{L}^+ = \emptyset$, return ("No path");

$\mathcal{P}^- = \text{Subpaving}(\mathcal{P}, [t]([\vec{p}]) = 1)$; $\mathcal{G}^- = \text{Graph}(\mathcal{P}^-)$;

If $v_a \in \mathcal{G}^-$ and $v_b \in \mathcal{G}^-$, $\mathcal{L}^- = \text{SHORTESTPATH}\ (\mathcal{G}^-, v_a, v_b)$;

If $\mathcal{L}^- \neq \emptyset$, return $\mathcal{L}^-$;

$\mathcal{C} = \{[\vec{p}] \in \mathcal{P}^+ \mid \text{vertex}([\vec{p}]) \in \mathcal{L}^+ \text{ and } [t]([\vec{p}]) = [0,1]\}$;

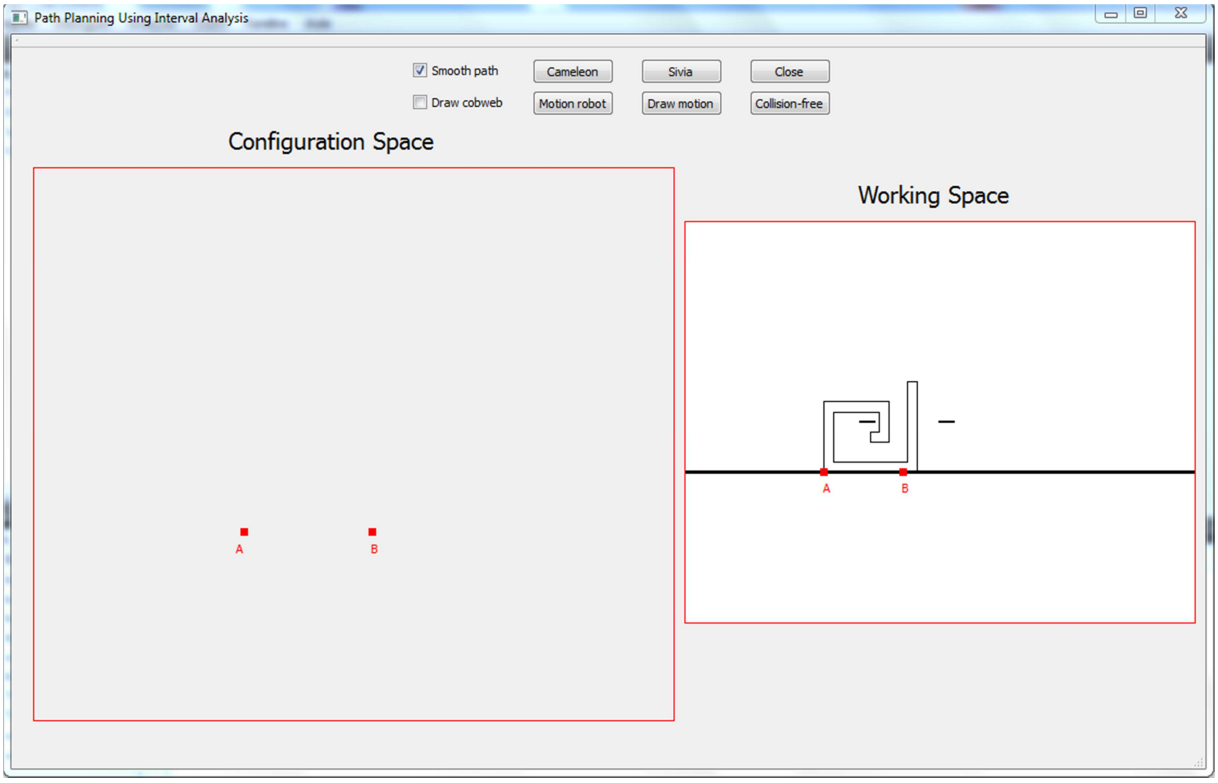
Bisect all subboxes of $\mathcal{C}$, thus obtaining a new paving $\mathcal{P}$;

Until False.

Development Environment: Qt

Qt is a cross-platform application and UI framework with APIs for C++ programming and Qt Quick for rapid UI creation.

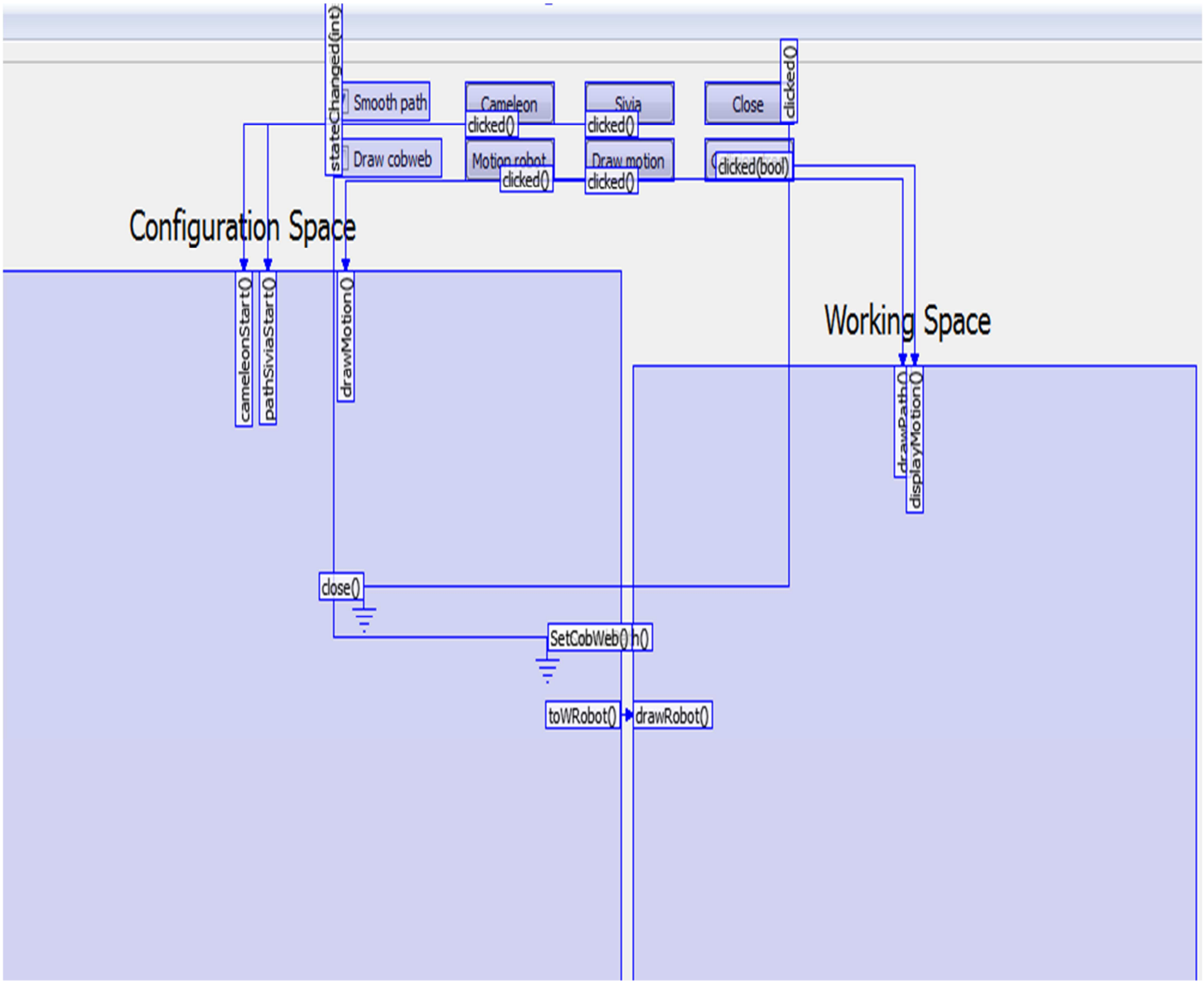
Software Architecture



The first widget which represents the configuration space is associated to the class widgetCameleon.

This class has 4 attributes: two bondbox*pA and pB, corresponding to the initial position and the desired position, a QImage image1 and priorBox that correspond to the initial domain.

Signals and slots



There are two check boxes: smooth Path and cobweb, if the first when is checked, the software try to find a feasible path with approximate angles. The cobweb checkbox when is checked the software draw the graph discretization of the initial domain after the Sivia algorithm and the Cameleon algorithm.

To illustrate the Principe of the configuration space we get the coordinates of the mouse when the mouse moves over the first widget. For this reason, there are two global variables correspond to X and Theta (Because in this case the robot has two degrees of freedom one translation and one rotation).

```
//pour recupere les coordonnées de la souris dans widgetCameleon
ui->widgetCameleon1->setMouseTracking(true);
```

In the initialization of the main window we should set the mouse tracking:
In the widgetCameleon.cpp we should put:

```
void widgetCameleon::mouseMoveEvent(QMouseEvent *event)
{
   double x = event->x();//Recuperer les coordonnes de la souris dans widgetCameleon
   double y = event->y();

   double echx = width()/Width(priorBox[1]);//Pixel to X
   double echy = height()/Width(priorBox[2]);//Pixel to Theta
   X= ((double)x/echx)+priorBox[1].inf;
   Theta= priorBox[2].inf+((double)(height()-y)/echy);
   toWRobot();//Signal vers widgetrobot pour dessiner le robot correspondant a cette position
}
```

And we should put the keyword virtual:

```
virtual void mouseMoveEvent(QMouseEvent *event);
```

Graphics

Qt's 2D graphics engine is based on the QPainter class.The QPainter class performs low-level painting on widgets and other paint devices.

QPainter provides highly optimized functions to do most of the drawing GUI programs require. It can draw a variety of shapes from simple lines to complex shapes like pies and chords. It can also draw aligned text and pixmaps. Normally, it draws in a "natural" coordinate system, but it can also do view and world transformation. QPainter can operate on any object that inherits the QPaintDevice class.

The common use of QPainter is inside a widget's paint event: Construct and customize (e.g. set the pen or the brush) the painter. Then draw. One should remember to destroy the QPainter object after drawing

QPainter can be used to draw on a "paint device", such as a QWidget, a QPixmap, a QImage, or a QSvgGenerator. QPainter can also be used in conjunction with QPrinter for printing and for generating PDF documents. This means that we can often use the same code to display data on-screen and to produce printed reports.

Warning: When the paintdevice is a widget, QPainter can only be used inside a paintEvent() function or in a function called by paintEvent(); that is unless the Qt::WA_PaintOutsidePaintEvent widget attribute is set. On Mac OS X and Windows, you can only paint in a paintEvent() function regardless of this attribute's setting.

Solution:

High-Quality Rendering with QImage

When drawing, we may be faced with a trade-off between speed and accuracy. For example, on X11 and Mac OS X, drawing on a QWidget or QPixmap relies on the platform's native paint engine. On X11, this ensures that communication with the X server is kept to a minimum; only paint commands are sent rather than actual image data. The main drawback of this approach is that Qt is limited by the platform's native support: On X11, features such as antialiasing and support for fractional coordinates are available only if the X Render extension is present on the X server.

On Mac OS X, the native aliased graphics engine uses different algorithms for drawing polygons than X11 and Windows, with slightly different results.

When accuracy is more important than efficiency, we can draw to a QImage and copy the result onto the screen. This always uses Qt's own internal paint engine, giving identical results on all platforms. The only restriction is that the QImage on which we paint must be created with an argument of either QImage::Format_RGB32 or QImage::Format_ARGB32_Premultiplied

We create a QImage of the same size as the widget in premultiplied ARGB32 format, and a QPainter to draw on the image. The initFrom() call initializes the painter's pen, background, and font based on the widget. We perform the drawing using the QPainter as usual, and at the end we reuse the QPainter object to copy the image onto the widget. This approach produces identical high-quality results on all platforms, with the exception of font rendering, which depends on the installed fonts.

By re implementing QWidget::paintEvent ()

```
//-------------------------------------------------------------------------
void widgetrobot::paintEvent(QPaintEvent *)
{
    QPainter widgetPainter(this);
    widgetPainter.drawImage(0, 0, image2);
    widgetPainter.setPen(QPen(QColor(Qt::red)));
    widgetPainter.drawRect(QRectF(0, 0, width() - 1, height() - 1));
}
//-------------------------------------------------------------------------
```

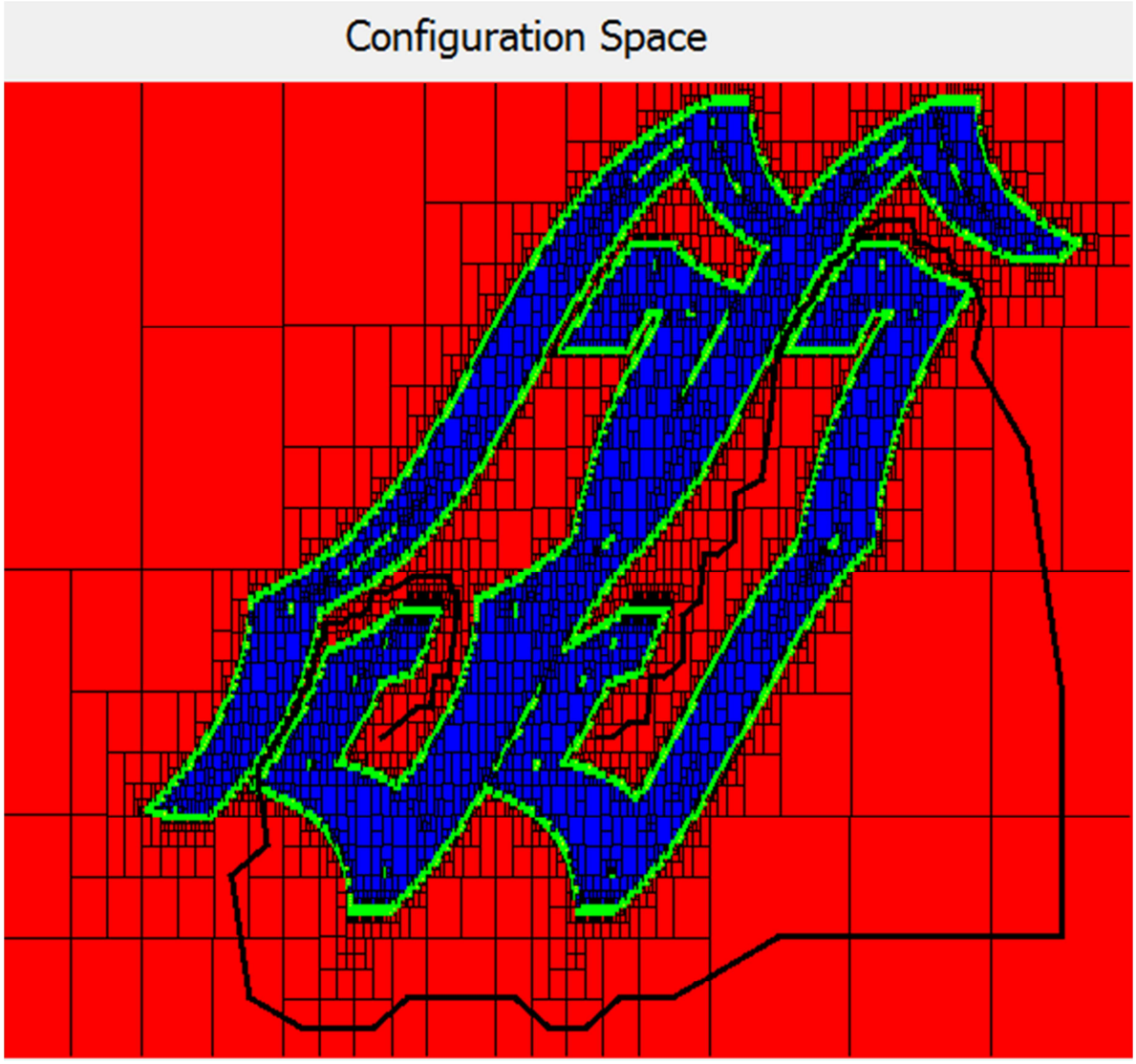
Drawing boxes in the QWidget

To draw boxes (here vector of two intervals), and to refresh the image in the QWidget during the execution of the program.

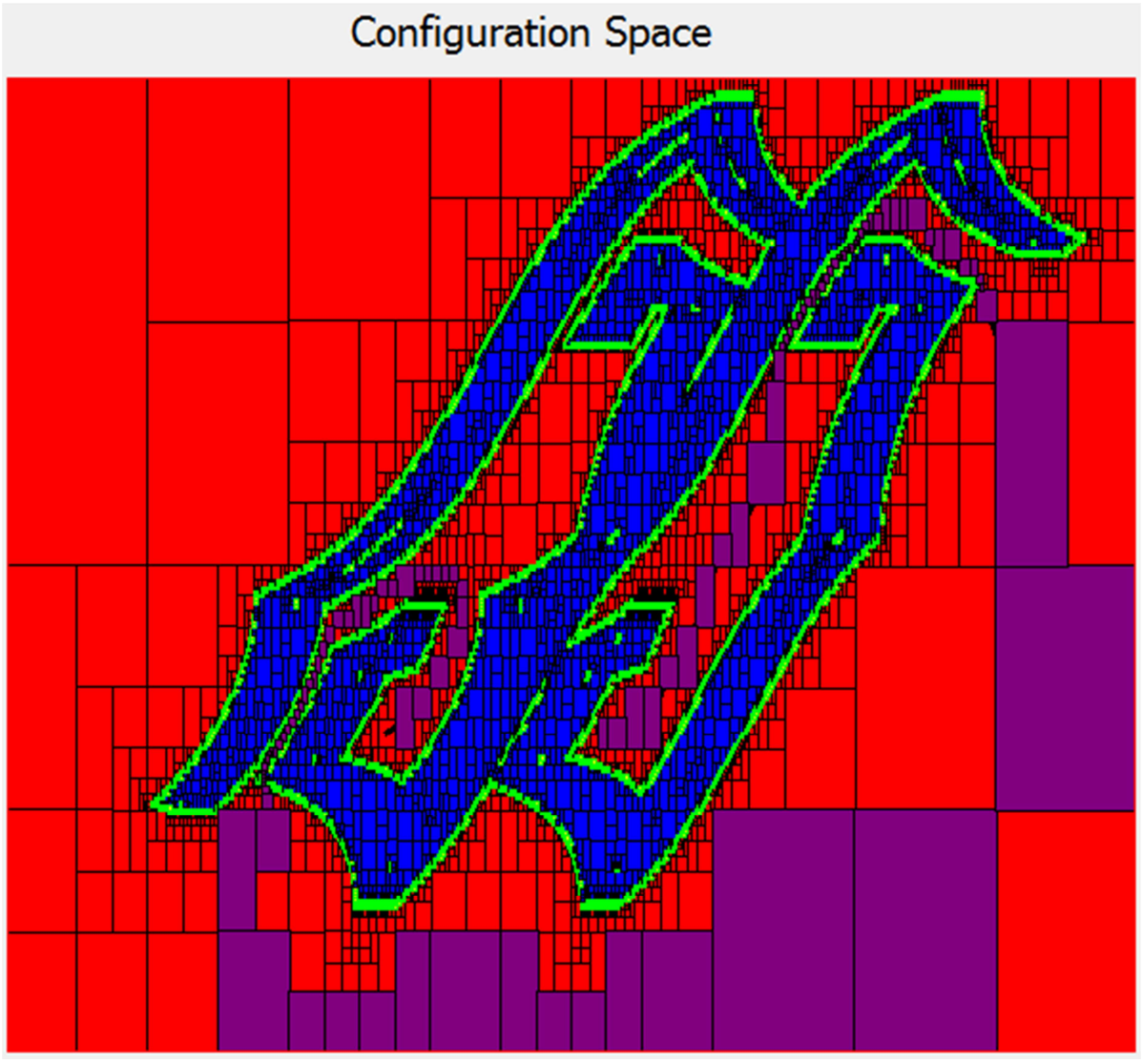
```
//*************************************************************************
//******************     Affichage de la box     **********************
//*************************************************************************
double xToPix(double x,box & priorBox,QImage *image)
{   double echx = image->width()/Width(priorBox[1]);
    return (x-priorBox[1].inf)*echx;
}

//-----------------------------------------------------------------------------------------------
double yToPix(double y,box & priorBox,QImage *image)
{
    double echy = image->height()/Width(priorBox[2]);
    return image->height()-(y-priorBox[2].inf)*echy;
}
//-----------------------------------------------------------------------------------------------
void DrawBox (box& b,box &priorBox,QColor c,QColor border,QImage* image,QPainter *painterImage)
{
    if (b.IsEmpty()) return;
    painterImage->setPen(QPen(c));
    QRect cr;
    cr.setTopLeft(QPoint(xToPix(b[1].inf,priorBox,image),yToPix(b[2].sup,priorBox,image)));
    cr.setBottomRight(QPoint(xToPix(b[1].sup,priorBox,image),yToPix(b[2].inf,priorBox,image)));

    painterImage->fillRect(cr,c );
    painterImage->setPen(QPen(border));
    painterImage->drawRect(cr);

}
```

Results

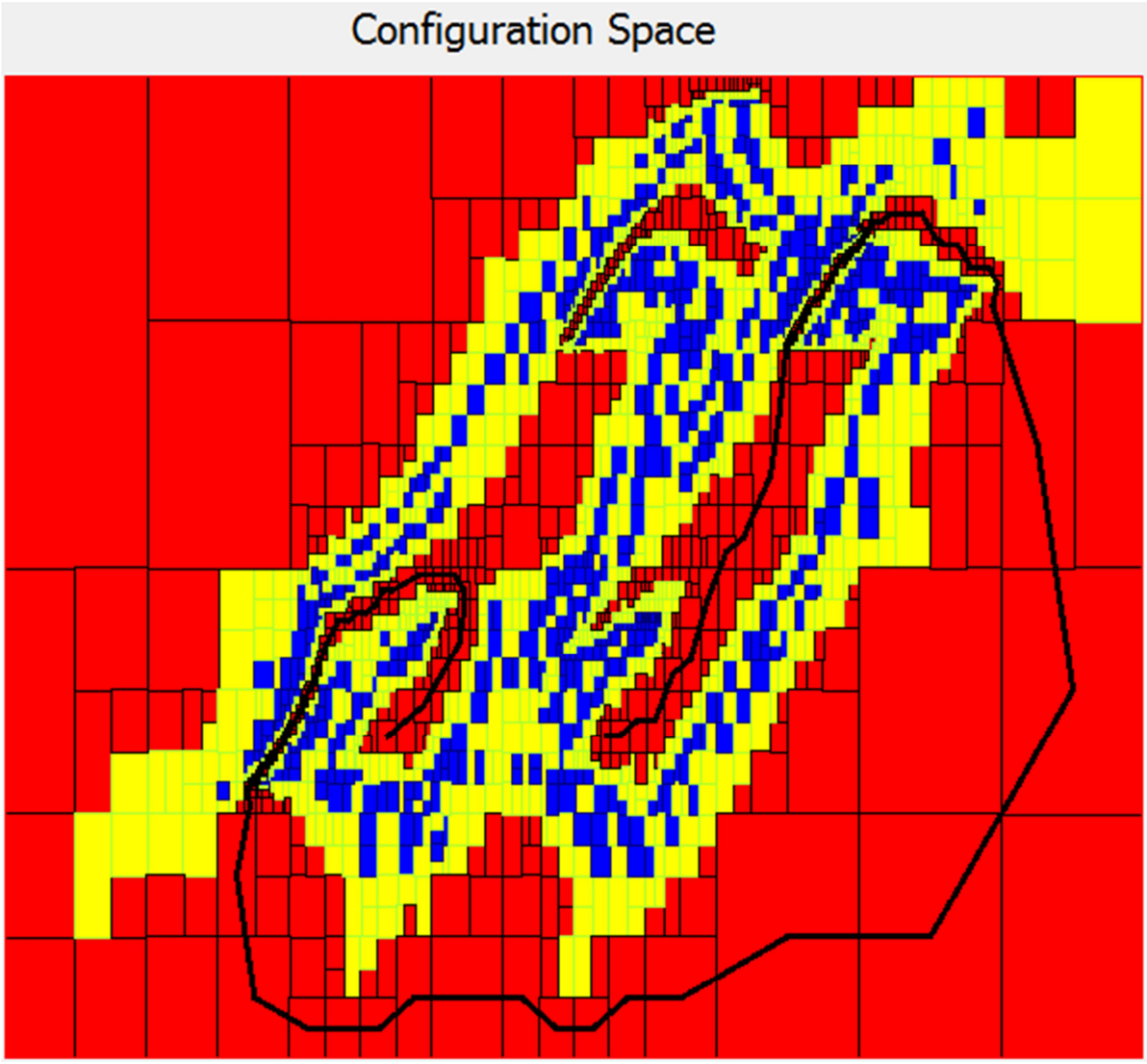
SIVIA without smooth path



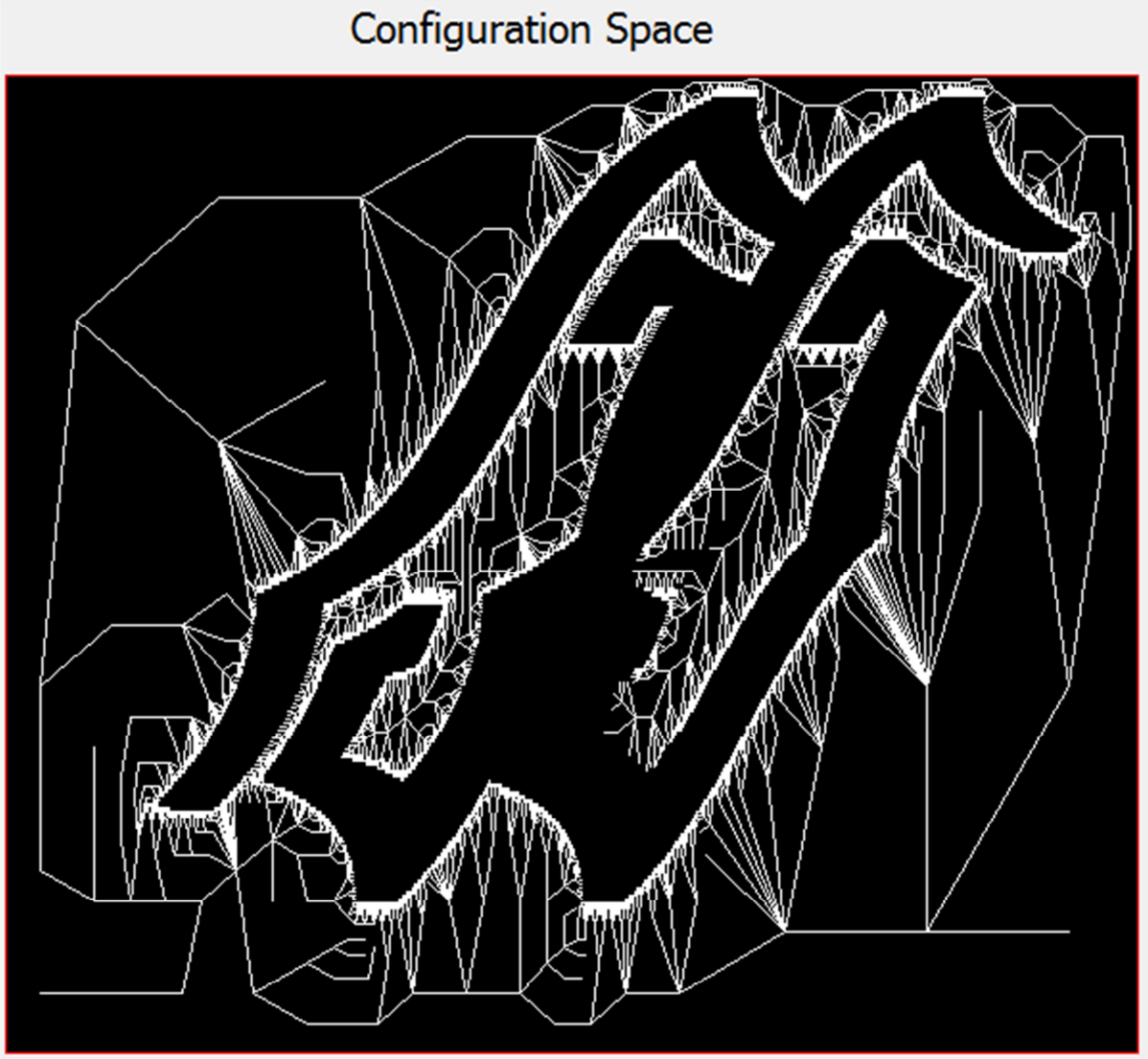
SIVIA with smooth path



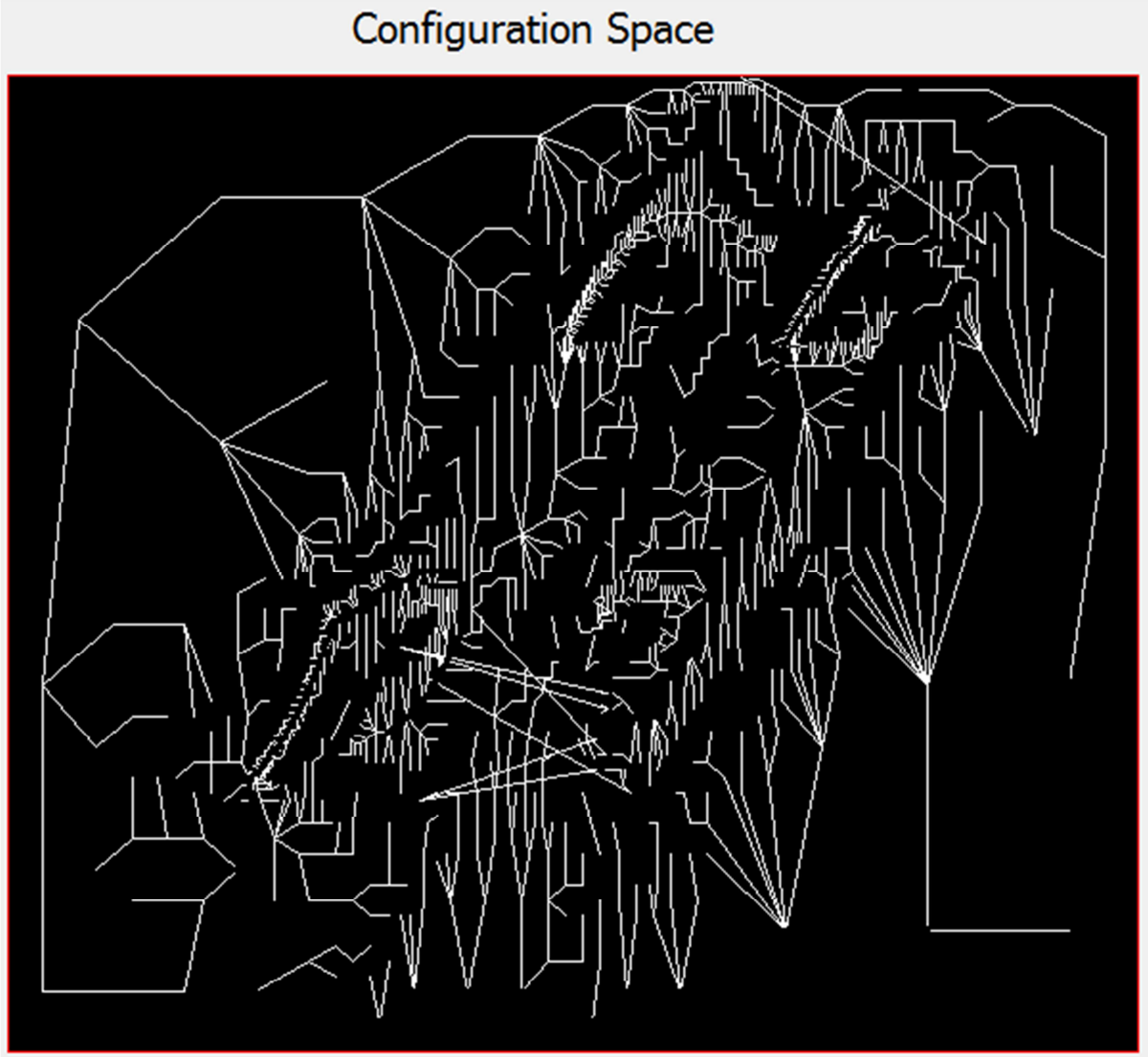
Cameleon with smooth path



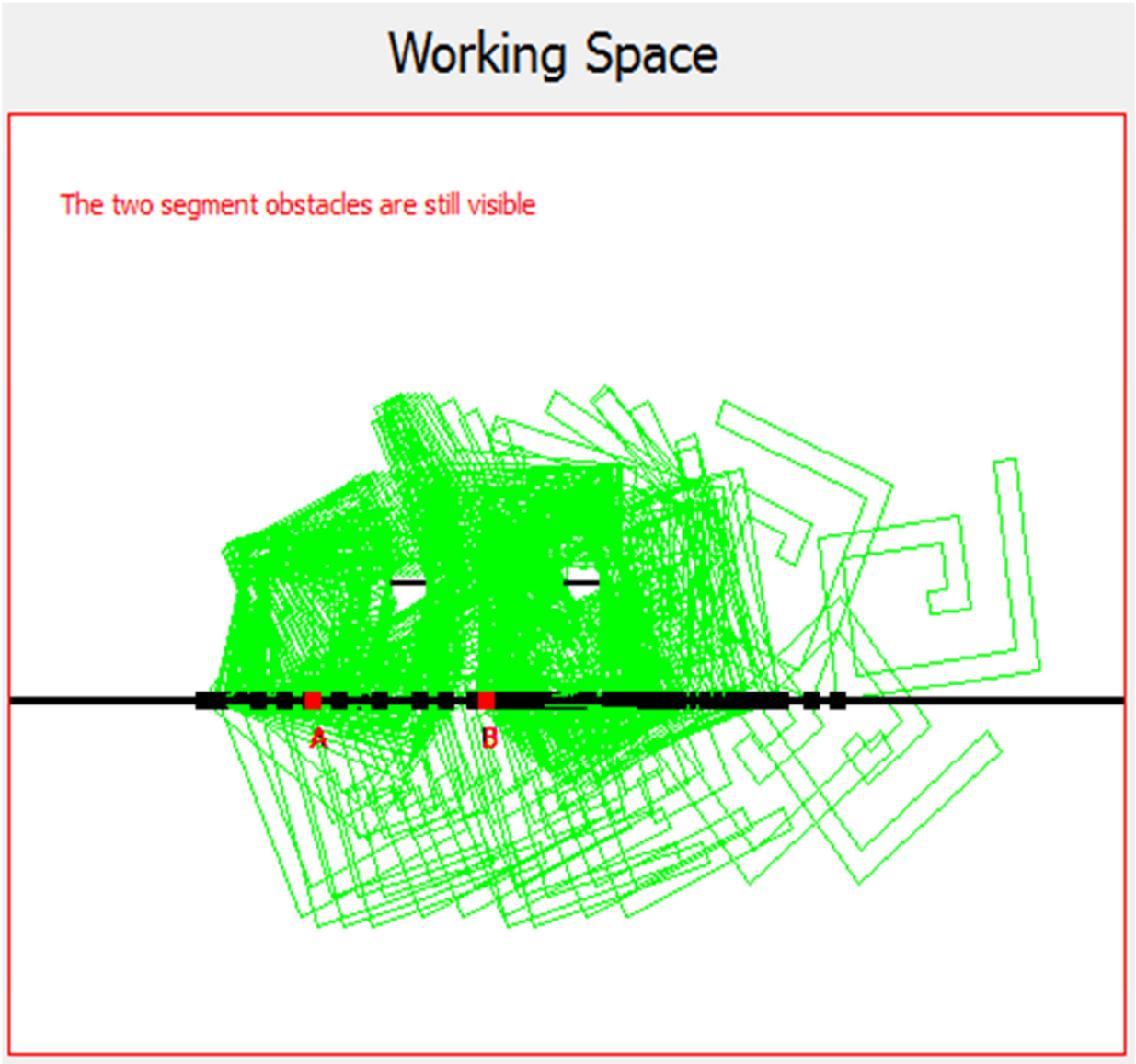
SIVIA Graph



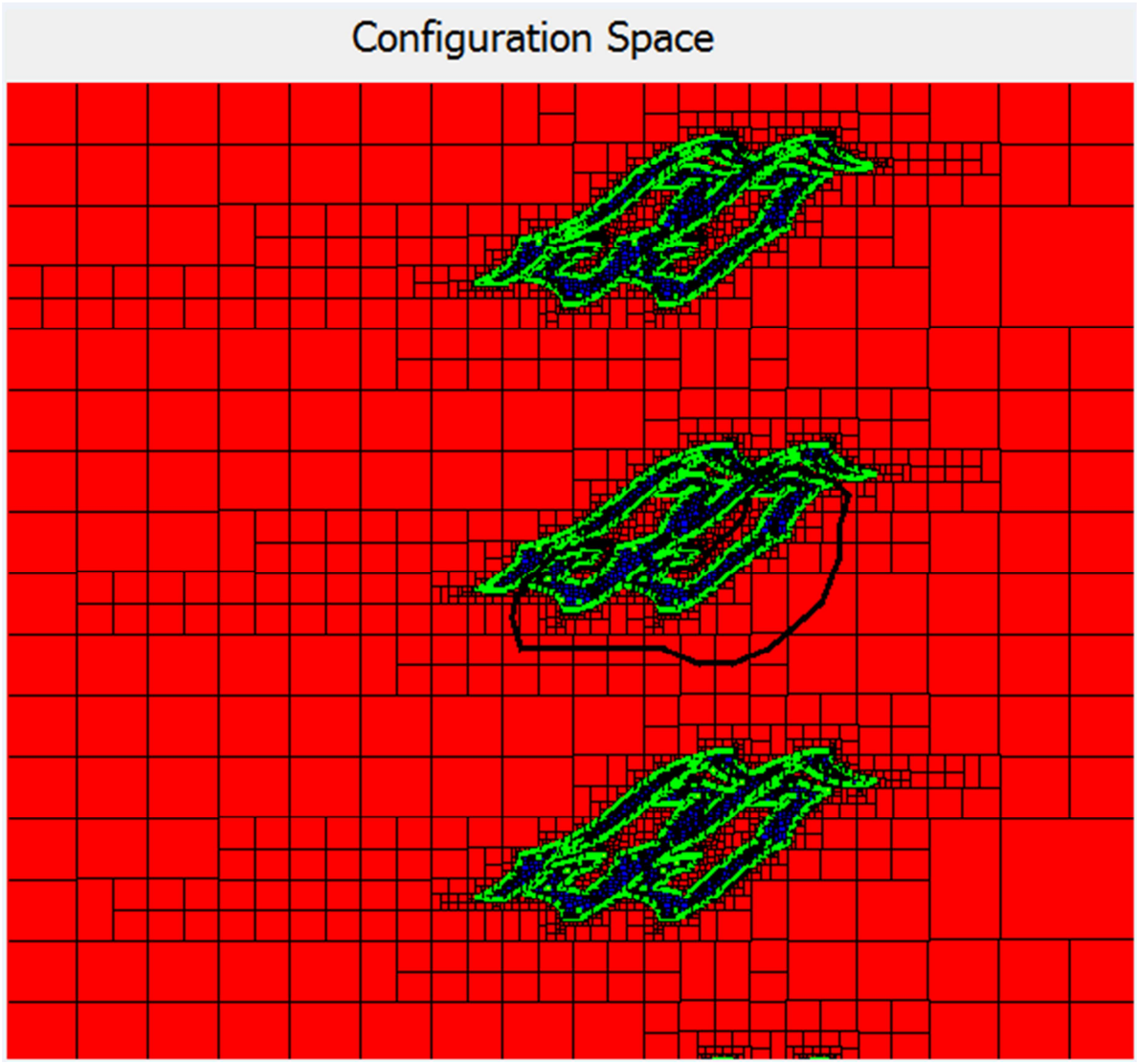
Cameleon Graph



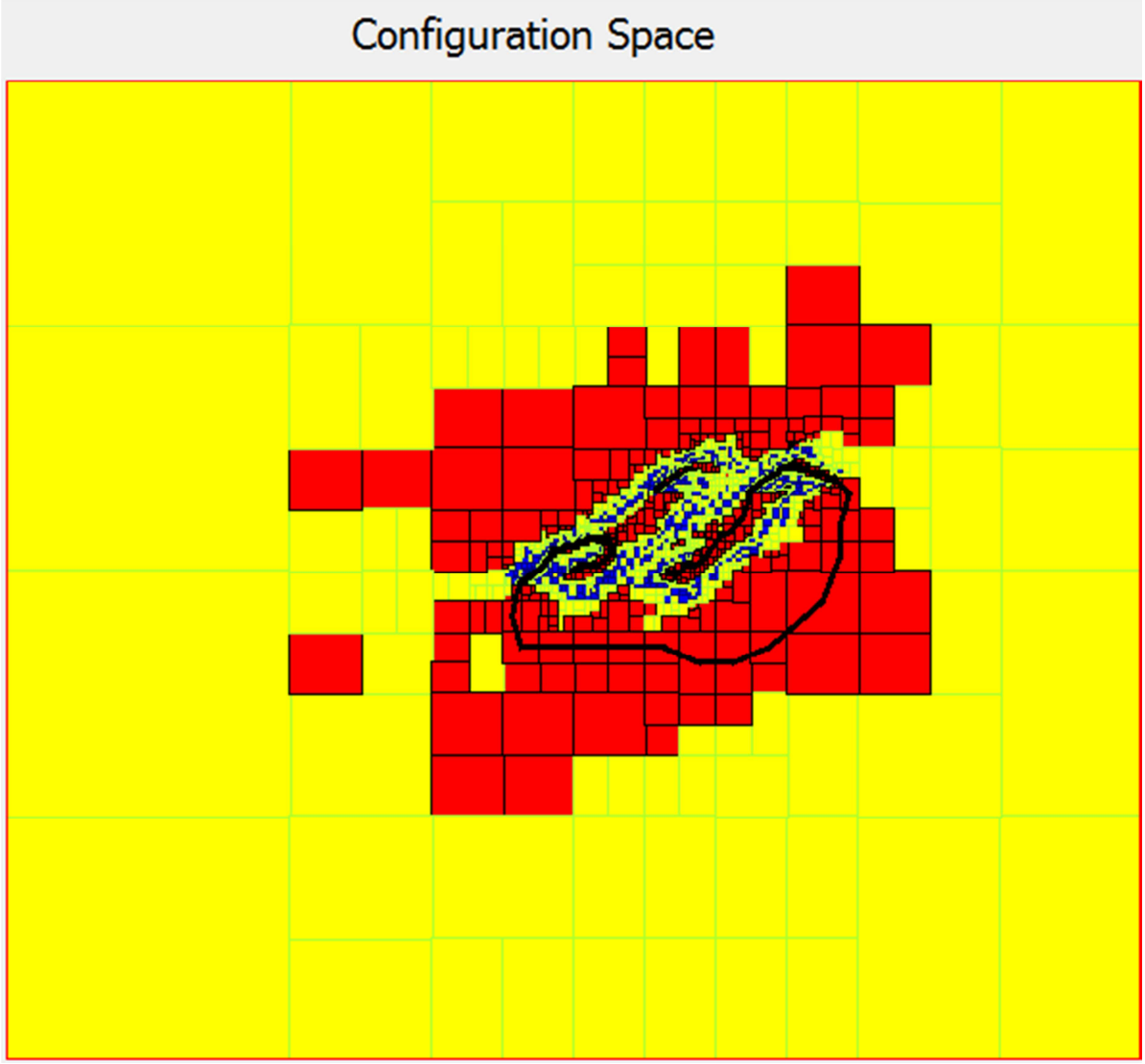
The two segment obstacles are still visible



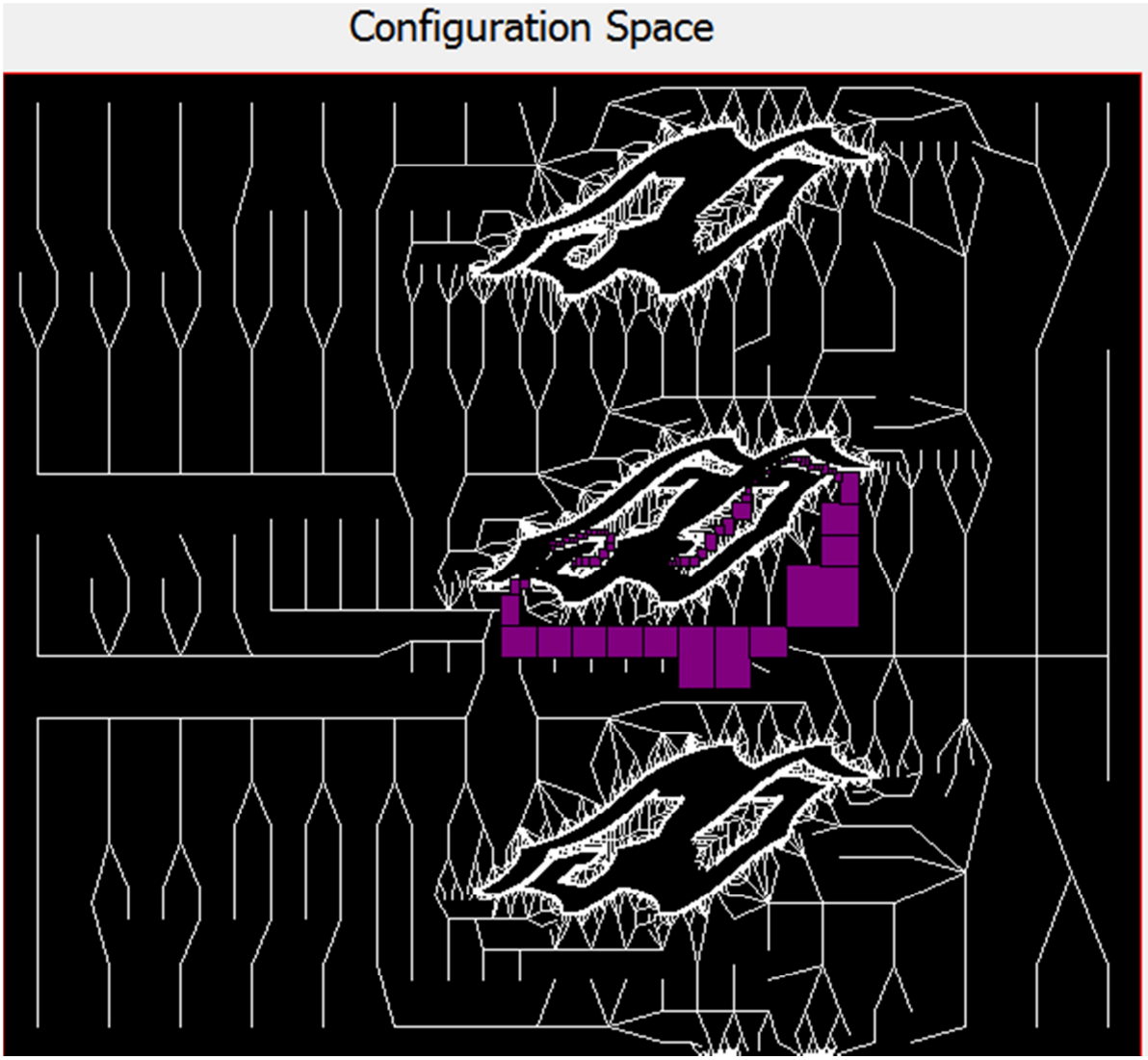
SIVIA with P0=[-100,100]*[-10,10]



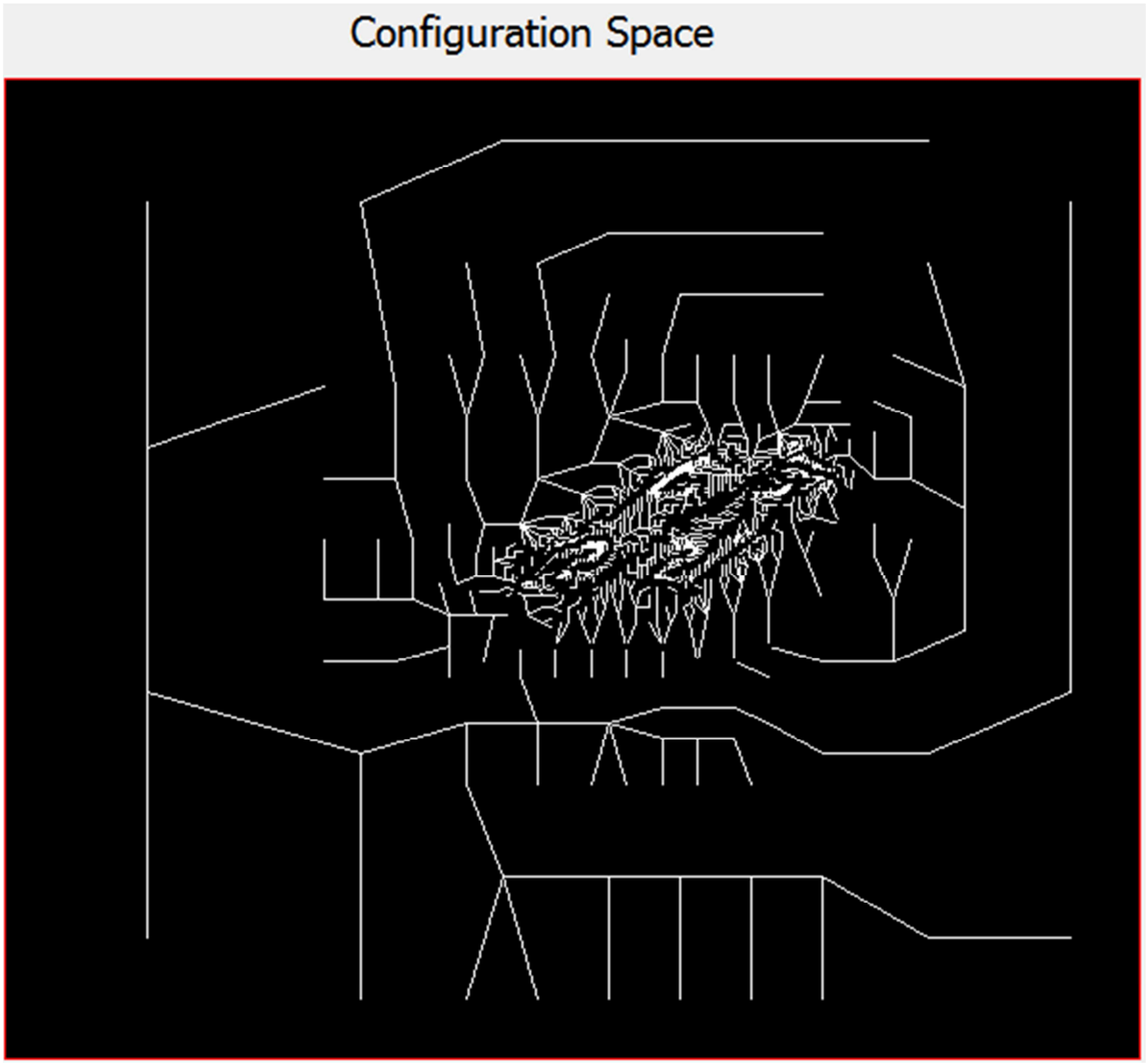
Cameleon with P0= [-100,100]*[-10,10]



Graph SIVIA with P0= [-100,100]*[-10, 10]



Graph Cameleon with P0= [-100,100]*[-10, 10]



Conclusion

We show that state-of-the-art algorithms converge to a NON-optimal solution almost-surely. The piano mover's problem has received a great deal of attention recently, in addition an interval analysis approach has been used to find a good feasible path from a start configuration to a goal configuration.
Interval analysis gives approximate but guaranteed results; we enjoyed the implementation of the chameleon algorithms on Qt that is a very efficient platform. The developed graphical interface permits the testing of the algorithm under various conditions using defined programming parameters.
Personally, this project improved my knowledge on C++ and motion planning algorithms.

Bibliography

[1] L. Jaulin (2001). Path planning using intervals and graphs. Reliable computing, issue 1, volume 7, 1-15,

- *[2] O Dunlaing, C. and Yap, C.K., A retraction method for planning the motion of a disc, Journal of Algorithms, 6, 1982, 104-111*
- *[3] Lozano-Pérez, T. and Wesley, M., An algorithm for planning collisionfree paths among polyhedral obstacles. Communications of the ACM, 22(10), 1979, 560-570*
- *[4] Nilsson, N.J., A mobile automaton: an application of artificial intelligence techniques, Proceedings of the 1st International Joint Conference on Artificial Intelligence, Washington D.C., 1969, 509-520*
- *[5] Khatib, O., Real-time obstacle avoidance for manipulators and mobile robots, International Journal of Robotics Research, 5(1), 1986, 90-9*

Tutorials

http://www.voidrealms.com