Optimisation of Car Park Designs

Problem presented by

Ian Wise & Roland Trim

ARUP (Bristol, UK)

Executive Summary

The problem presented by ARUP to the Study Group was to investigate methods for maximising the number of car parking spaces that can be placed within a car park. This is particularly important for basement car parks in residential apartment blocks or offices where parking spaces command a high value. Currently the job of allocating spaces is done manually and is very time intensive.

The Study Group working on this problem split into teams examining different aspects of the car park design process. We report three approaches taken. These include a so-called 'tile-and-trim' method in which an optimal layout of cars from an 'infinite car park' are overlaid onto the actual car park domain; adjustments are then made to accommodate access from one lane to the next. A second approach seeks to develop an algorithm for optimising the road within a car park on the assumption that car parking spaces should fill the space and that any space needs to be adjacent to the network. A third similar approach focused on schemes for assessing the potential capacity of a small selection of specified road networks within the car park to assist the architect in selecting the optimal road network(s).

This problem is a variant of the 'bin packing' problem, well known in computer science. It is further complicated by the fact that two different classes of item need to be packed (roads and cars), with both local (immediate access to a road) and global (connectivity of the road network) constraints. Bin-packing is known to be NP-hard, and hence the problem at hand has at least this level of computational complexity.

None of the approaches produced a complete solution to the problem posed. Indeed, it was quickly determined by the group that this was a very hard problem (a view reinforced by the many different possible approaches considered) requiring far longer than a week to really make significant progress. All approaches rely to differing degrees on optimisation algorithms which are inherently unreliable unless designed specifically for the intended purpose. It is also not clear whether a relatively simple automated computer algorithm will be able to 'beat the eye of the architect'; additional sophistication may be required due to subtle constraints.

Apart from determining that the problem is hard, positive outcomes have included:

- Determining that parking perpendicular to the road in long aisles provides the most efficient packing of cars.
- Provision of code which 'tiles and trims' from an infinite car park onto the given car park with interactive feedback on the number of cars in the packing.
- Provision of code for optimal packing in a parallel-walled car park.
- Methods for optimising a road within a given domain based on developing cost functions ensuring that cars fill the car park and have access to the road. Provision of code for optimising a single road in a given (square) space.
- Description of methods for assessing the capacity of a car park for a set of given road network in order to select optimal road networks.
- Some ideas for developing possible solutions further.

Version 1.0
May 2, 2013
iv+52 pages

Report author

R. Porter (University of Bristol)

Contributors

John Billingham (Nottingham University)
Joel Bradshaw (University of Nottingham)
Marcin Bulkowski (Polish Academy of Sciences)
Peter Dawson (University of Warwick)
Pawel Garbacz (Polish Academy of Sciences)
Mark Gilbert (University of Oxford)
Lara Gosce (University of Bristol)
Poul Hjort (Technical University of Denmark)
Martin Homer (University of Bristol)
Mike Jeffrey (University of Bristol)
Dario Papavassiliou (University of Warwick)
Richard Porter (University of Bristol)
David Kofoed Wind (Technical University of Denmark)

ESGI91 was jointly organised by
University of Bristol
Knowledge Transfer Network for Industrial Mathematics

and was supported by
Oxford Centre for Collaborative Applied Mathematics
Warwick Complexity Centre
Natural Environment Research Council

Contents

1 Introduction **1**

2 Problem statement **2**

3 Tile and trim **2**
3.1 Physical Justification 2
3.2 The Lane Width 4
3.3 Infinite car park density function 5
3.4 Bounded Areas 6

4 Optimising the road **8**
4.1 Discrete approach 8
4.2 Variational approach 23

5 Optimising over road networks **25**
5.1 Details 26
5.2 Examples 29
5.3 Conclusions and possible extensions 34

A Appendices **37**
A.1 MATLAB code for the tile and trim approach 37
A.2 Python code for the tile and trim approach 43

Bibliography **52**

1 Introduction

(1.1) Plans for new urban developments such as office blocks or residential apartments one very often include an integrated basement car park. Due to the high value of car parking spaces the design of the car park is very important. For example, in central London, a parking space may be worth up to £100,000 whilst elsewhere across the country the value might still be as high as £20,000. Thus, developers are very keen to ensure that they are maximising the potential capacity of the space. Typically basement car parks are formed by an irregular polygonal boundary (rarely a simple rectangle) and are subject to building design constraints. For example, the car park may include columns which support the building above as well as larger internal structures such as lift shafts and stairwells. In addition, there are constraints imposed by access to parking spaces such as the car park entrance/exit which is often fixed by the position of the building in relation to the road outside, as well as respecting limitations on vehicle manoeuvrability Thus there are many constraints that need to be accommodated into the design.

(1.2) Currently there are no design tools either to assist or automate parking space allocation. An architect uses CAD tools to populate a space manually and such is the value of spaces that positions of supporting columns and other internal obstacles can be adjusted (within limits) to maximise the spaces available. This is a time-consuming operation which can reportedly take up to 3 weeks. Even then, there is no guarantee that the architect has found the optimal solution. There is thus high value in producing algorithms which automate or assist in the design of the car park.

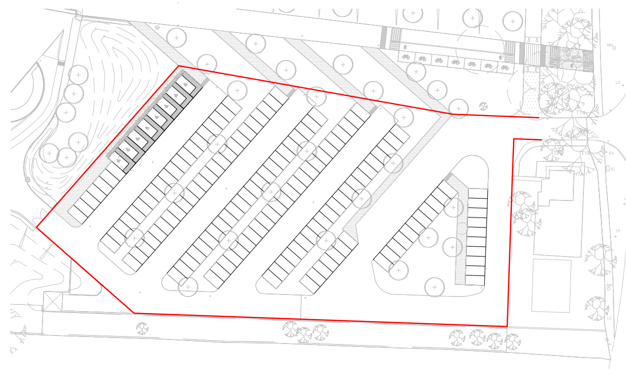


Figure 1: An example of a relatively simple surface car park with no internal obstacles. The boundary is in red. There are 147 normal spaces and 8 disabled spaces. This arrangement is clearly not designed to maximise spaces.

2 Problem statement

(2.1) The statement of the problem is relatively simple. Given a car park perimeter, an entrance/exit positioned on the perimeter, the layout of any internal obstacles and constraints such as the minimum size of parking spaces, widths of road and turning circles of cars, how does one design a car park which maximises the number of car parking spaces ?

(2.2) To make things simpler, we were allowed to assume that the car park boundary is polygonal. We also make the assumption that obstacles such as narrow supporting columns are absent although we should attempt to include larger internal obstacles.

(2.3) Three approaches are taken and described in sections 3, 4 and 5 below. Additional code is supplied in the Appendices.

3 Tile and trim

(3.1) *Consider a hypothetical hotel with a countably infinite number of rooms, all of which are occupied. One might be tempted to think that the hotel would not be able to accommodate any newly arriving guests, as would be the case with a finite number of rooms. Suppose a new guest arrives and wishes to be accommodated in the hotel. Because the hotel has infinitely many rooms, we can move the guest occupying room 1 to room 2, the guest occupying room 2 to room 3 and so on, and fit the newcomer into room 1. By repeating this procedure, it is possible to make room for any finite number of new guests.*[1]

3.1 Physical Justification

(3.2) We first study the tiling of cars in a parking lot of infinite length, and variable width a. By tiling we mean that the cars are arranged in a regular pattern. Each pattern is then based on repetitions of a fundamental 'cell'.

(3.3) For each value of a we to pick the tiling which is optimal, in the sense that the tiling maximizes the density ρ of cars (within the fundamental cell):

$$\rho = \frac{\text{area occupied by cars}}{\text{total area}} \, . \tag{1}$$

Heuristically, one can think of this parking lot as an infinite street of width a, along which we wish to park rectangular (door-less) cars. For every width a of the street we wish to have an optimal parking strategy, under

the constraint that cars must still be able to drive along the road, possibly only in one direction, park in every parking space, and exit any parking space.

(3.4) We will study the density function $\rho(a)$ for values $w < a < \infty$.

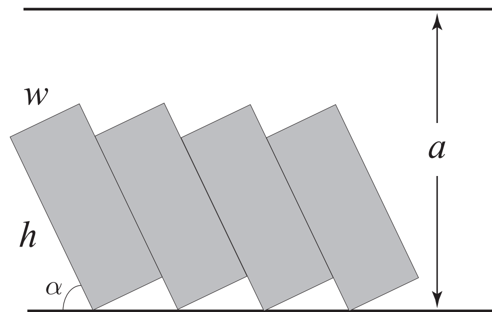


Figure 2: The basic set-up. In an infinite corridor, cars are packed in a pattern to maximize the density. The width a of the corridor is a variable. What is the optimal packing strategy for given aspect of the rectangular tiles, and a given value of the corridor width a ?

(3.5) We begin by studying, as in figure 4, a value of a so large that a single row of cars can be stacked at an angle α with the horizontal, but so small that there is no room for a second row of cars. Let the height of a car be h, and the width w. The density in this situation can be calculated by considering the parallelogram in figure 3.

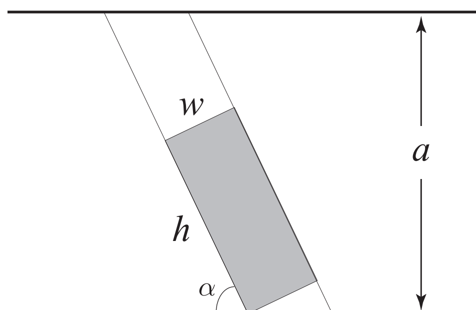


Figure 3: The overall density is determined by the density within a given cell. This is most easily calculated by taking as the cell the parallelogram illustrated in the figure.

(3.6) The base of the parallelogram is $w \sec(\alpha)$, and the height is a. This parallelogram is a cell which can tile the entire corridor. In the parallelogram,

the car takes up an area of $h \cdot w$, so

$$\rho_1 = \frac{hw}{aw \sec \alpha} = \frac{h}{a} \sin \alpha$$

a function which has its maximum at $\alpha = \pi/2$, the 'rectilinear' stacking.

(3.7) This preliminary calculation has been for a single row of cars, without a view to the repetition of rows and the 'lane' space, the width of the strip with no cars, where the cars drive to enter or leave the parking spaces. We now briefly consider this issue.

3.2 The Lane Width

(3.8) The lane width in a given packing width is not a geometrically well defined concept. Even assuming a standard width w for a car, and thus w as a lower bound for the lane width of all lanes in all packings, realistic packings must have a somewhat greater lane width.

(3.9) Various design manuals operate with the concept of the turning radius R of a standard car; this is the radius of the smallest circle which the centre point of the car is able to move in (figure (9). Using this concept provides loosely an upper bound on the lane width, because cars (and most drivers) are able to perform 3-point or 4-point turns, rather than merely a 2-point manoeuvre.

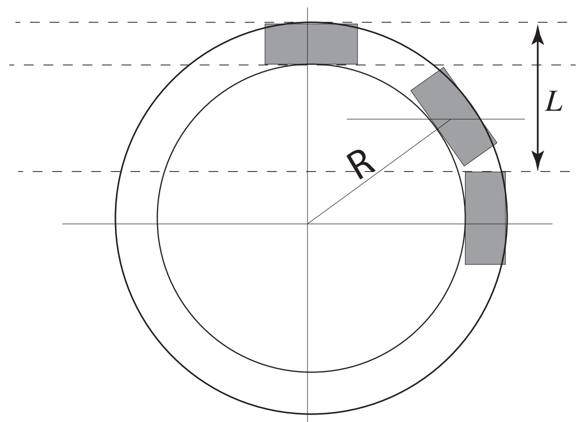


Figure 4: The turning radius

(3.10) The most lane width demanding pattern is the packing with $\alpha = \pi/2$. Often a lane width of the order of $w + h$ are seen in this design. Herringbone packings with values of α less than $\pi/2$ give easier access and exit manoeuvres, and have smaller lane width.

(3.11) For the sake of calculations, let us assume that the necessary lane with varies with angle α as

$$L(\alpha) = w + h\sin(\alpha) \tag{2}$$

(3.12) This is a monotone function which has the value w for small values of α, and the value $w + h$ for $\alpha = \pi/2$. Using this, we can now evaluate the density of the infinitely repeated double-row pattern with variable angle α.

3.3 Infinite car park density function

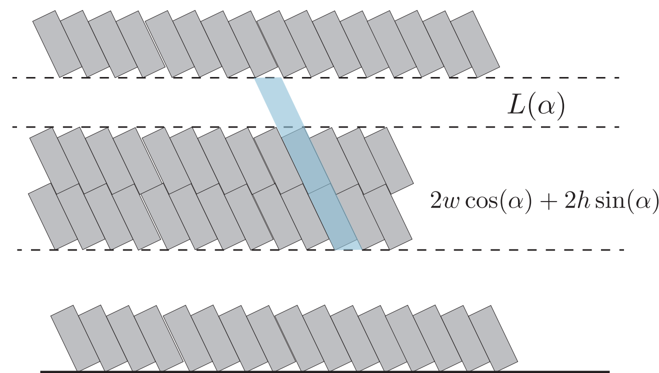


Figure 5: Several rows of cars, in a herringbone pattern. The herringbone pattern permits more rows, with a smaller lane width between the rows. Stacking with a value of α closer to $\pi/2$ gives rows having a larger lane width (width of access roads) which decreases the density, but also with a tighter packing which increases the density.

(3.13) A fundamental cell now has two car areas in it, and, for general values of α, two small triangles with waste space. Using the geometry of figure 3.3, we get that the density function is

$$\rho(\alpha) = \frac{2hw}{[w + h\sin(\alpha) + 2w\cos(\alpha) + 2h\sin(\alpha)]\, w\sec(\alpha)} \tag{3}$$

(3.14) A plot of this function: shows the value of ρ increasing towards the maximal value:

$$\rho(\pi/2) = \frac{2hw}{[3h + w]w} \tag{4}$$

Thus, even is the lane width generally is smaller for the $\alpha < \pi/2$ herringbone pattern, the denser packing of the $\alpha = \pi/2$ wins out.

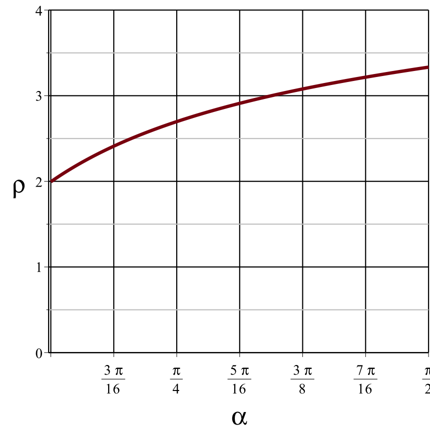


Figure 6: A plot of the density function $\rho(\alpha)$ giving the density of cars as a function of herringbone angle α. Maximum occurs for the rectilinear packing, $\alpha = \pi/2$.

(3.15) Thus, under quite general assumptions, the rectilinear double-row parking pattern is the optimal way to tile (pack) the infinite plane with parking spaces.

3.4 Bounded Areas

(3.16) For a plane figure of area A, a quick and rough estimate of the maximal number of possible car spaces is therefore $A/\rho(\pi/2)$.

(3.17) A slightly more refined estimate lies in the following strategy to estimate the maximal number of parking spaces in a given finite connected polygon in the plane:

1. Overlap the given finite polygon with the 'best' packing for the infinite plane.
2. Turn and shift the given polygon to maximise the number of cars from the background packing falling inside the polygon.
3. Trim away a small number of car spaces to provide connection between otherwise unconnected lanes, ensuring that all spaces are accessible.
4. if there are pillars or obstacles inside the polygon, remove the conflicting car spaces.

(3.18) We have written a code in the programming language PYTHON which performs the above strategy. The source code is included in the appendix. A screen-shot from the code is seen below.

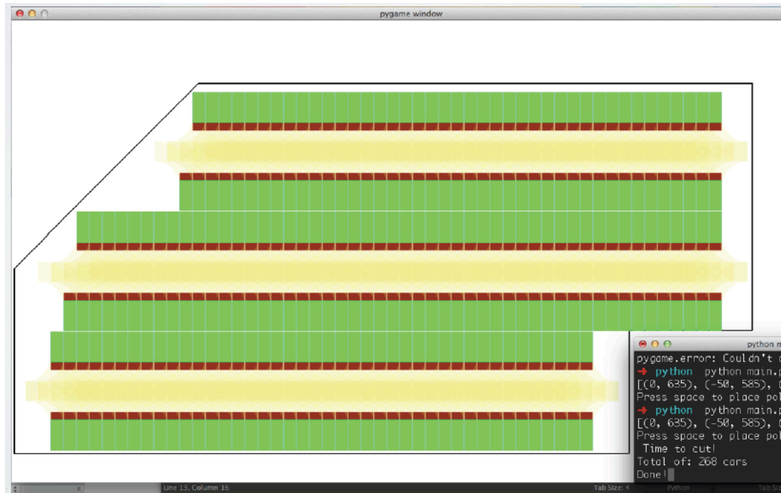


Figure 7: Output from the PYTHON code showing double rows (in green) of cars with access lanes (in yellow) placed inside a test polygon. The small window insert shows how the code keeps running track of the number of cars inside the polygon, allowing for manual fine tuning of the placement of the rectangle relative to the background packing.

(3.19) This code enables the user very quickly to estimate within a number on the order of the number of lanes visible, the optimal total number of car spaces.

(3.20) Note that we have only provided a proof that the rectilinear double-row pattern is optimal among other herringbone patterns in the infinite plane. It is possible that finite polygons permit a slightly better pattern that utilises the particular shape of the given polygon. In particular, one or more rows of $\alpha < \pi/2$ spaces may be squeezed in, rather than dropping a full row of $\alpha = \pi/2$ spaces.

(3.21) In order to investigate significance of this effect, we went back to the infinite strip of variable height, and constructed in MATLAB a code which optimizes the number of cars in rows, allowing for $\alpha < \pi/2$ rows, with corresponding smaller lane width. This code is also provided in the appendix (Note: MATLAB is a commercial package, requiring a license to run.

(3.22) The figure shows a value for a where there is no room for two double rows, but there is room for a single double row plus one row with $\alpha < \pi/2$. The code provides the correct value of α, which is important, since we have seen than one should use the maximum value of α possible, given the constraints.

(3.23) The code used to generate the results in this section of the report is contained in Appendices 1 (components of MATLAB code) and 2 (python code).

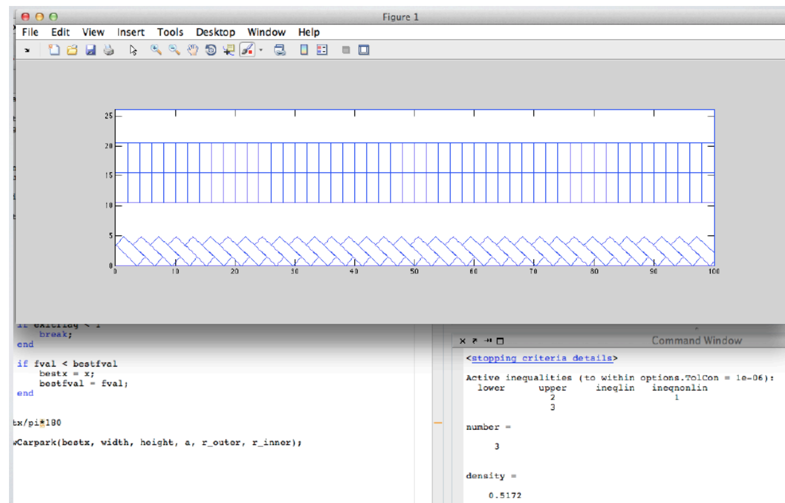


Figure 8: Output from the MATLAB code showing

(3.24) For the Python code ones need the package pygame, downloadable from: `http://www.pygame.org/` To run the script, just evoke the main.py, with the command:

```
> python main.py
```

4 Optimising the road

(4.1) Any allowable configuration of car parking spaces must be adjacent to a road network that allows access to any space. In this section, we focus on constructing this road, and treat the positioning of the cars as a problem that can be dealt with separately.

4.1 Discrete approach

(4.2) In the discrete approach that we developed during the Study Group, we also treat the road as a closed loop, but approximate it as a piecewise linear curve connecting a set of nodes. We then try to determine the position of the nodes that minimises a cost function, which we define in a manner designed so that at its minimum (i) as much of the domain as possible is within a distance l_r of one of the linear parts of the loop, (ii) as little of the domain as possible is is within a distance l_r of more than one non-contiguous segments of the loop, (iii) the angle at each node is between $\pi/2$ and $3\pi/2$, (iv) the loop is everywhere more than l_r from the boundary of the domain. Here, l_r is the half-width of the road plus the length of a parking space.

(4.3) In order to define the domain, we use a set of points that lie within the

domain. We experimented with various choices of points, and found that a random distribution within the domain with density $(2/l_r)^2$ works reasonably well. However, we only had time to experiment with a unit square. Note that this means the smaller l_r, the larger the domain relative to a parking space.

(4.4) The cost function that we used is most easily explained using the MATLAB code that we wrote during and immediately after the meeting.

```
function cost = cost(xys0,Nr,xyi,Ni,lroad)
%xys0 - a vector containing the coordinates of the nodes of
%the road

%Nr - the number of nodes in the road

%xyi - a vector containing the points that define the domain

%Ni - the number of points that define the domain

%lroad - the half-width of the road plus the length of a
%parking space


%Extract the nodes that define the road.

xys = xys0(1:2*Nr);
xys = reshape(xys,Nr,2);


%Make the road into a closed loop.

xys = [xys; xys(1,:)];


%Initialise the cost function.

cost = 0;


%Penalise heavily if a node is further than lroad from the
%interior of the unit square.

for i = 1:Nr
    if xys(i,1)<lroad, cost = cost+1e12*(lroad-xys(i,1)); end
    if xys(i,2)<lroad, cost = cost+1e12*(lroad-xys(i,2)); end
    if xys(i,1)>1-lroad, cost = cost+1e12*(xys(i,1)-1+lroad); end
    if xys(i,2)>1-lroad, cost = cost+1e12*(xys(i,2)-1+lroad); end
end
```

```
%Penalise based on the position of the road relative to the
%interior points.

for i = 1:Ni
% Loop over interior points

    v = zeros(1,Nr); dist0 = zeros(1,Nr);
    dmin = 1e12;

    for j = 1:Nr

    %Determine the minimum distance from each line segment of
    %the road.

        dist1 = lsdist(xys(j,:), xys(j+1,:),xyi(i,:));
        dist2 = lsdist(xys(j+1,:), xys(j,:),xyi(i,:));
        dist = max(dist1,dist2);
        dmin = min(dmin,dist);
        if dist<=lroad

        %Keep a record of road segments from which the interior node
        %is less than a distance lroad away.

            v(j) = 1;
            dist0(j) = dist;
        end
    end

    %If the node is less than lroad away from two
    %noncontiguous segments, penalise.

    if length(find(diff(v)))>2
        cost = cost+(lroad-sum(dist0(v==1))/sum(v))/lroad;
    end


    %If the interior point is not within lroad of any
    %segement of the road, penalise.

    if dmin>lroad
        cost = cost+(dmin-lroad)/lroad;
    end

end

%Penalize small angles.
```

```
%cb = (1+cos(angle between road segments))/2 and
%varies between zero and one.

%Calculate cb at the first node.

cb = 0.5*(1+cosbeta(xys(Nr,:),xys(1,:),xys(2,:)));


%If the angle is less than pi/2, penalise.

if cb>0.5
    cost = cost+5*(cb-0.5)/0.5;
end


%Repeat for the other nodes.

for i = 2:Nr
    cb = 0.5*(1+cosbeta(xys(i-1,:),xys(i,:),xys(i+1,:)));
    if cb>0.5
        cost = cost+5*(cb-0.5)/0.5;
    end
end

%%

function lsdist = lsdist(a,b,c)

%Find the distance of the point with position vector c from
%the line segment between the points with position vectors
%a and b. This is either the shortest distance from the straight
%line through a and b or, if this point does not lie between
%a and b, the distance from the nearer of a and b.

cb = c-b; ab = a-b; lcb = norm(cb); lab = norm(ab);
if (lcb == 0)||(lab == 0)
    lsdist = 0;
else
    cbeta = dot(cb,ab)/lcb/lab;
    if cbeta<=0
        lsdist = lcb;
    else
        lsdist = lcb*sqrt(1-cbeta^2);
    end
end

%%
```

```
function cosbeta = cosbeta(a,b,c)

%Calculate the cosine of the angle between the lines ab and cb.

ab = a-b; cb = c-b;
cosbeta = dot(ab,cb)/norm(ab)/norm(cb);
```

(4.5) In order to minimise the cost function, we use repeated passes of firstly a genetic algorithm and secondly the Nelder-Mead simplex algorithm, as implemented in MATLAB (*ga* and *fminsearch*). This was done using the default parameters, and better tuning may improve the rate of convergence.

(4.6) This approach was neither entirely unsuccessful nor an unqualified success. When $l_r = 1/4(N-1)$ for $N = 2, 3, \ldots$, a likely optimal solution is fairly obvious, and is shown in Figure 1 for $N = 3$, $l_r = 1/8$. Figure 2 shows the solution to which our algorithm converges in this case, which is very close. If we start from an initial guess that is less good, as shown in Figure 3, the algorithm sometimes converges to a reasonable solution, for example, Figure 4, but more frequently, with a different seed in the random number generator for the genetic algorithm, gets stuck in a plausible, but local minimum of the cost function, as shown in Figure 5.

(4.7) When $N = 4$ and $l_r = 1/12$, starting from the reasonable initial guess shown in Figure 6, the algorithm converges to the local minimum shown in Figure 7. Again, this is some distance from the likely optimal solution shown in Figure 8.

(4.8) Although these results are not too encouraging, we also tried the intermediate case, $l_r = 1/10$, for which the optimal solution is not obvious. Starting from the initial guess shown in Figure 9, the algorithm converges to the solution shown in Figure 10, which is unlikely to be the global minimum of our cost function.

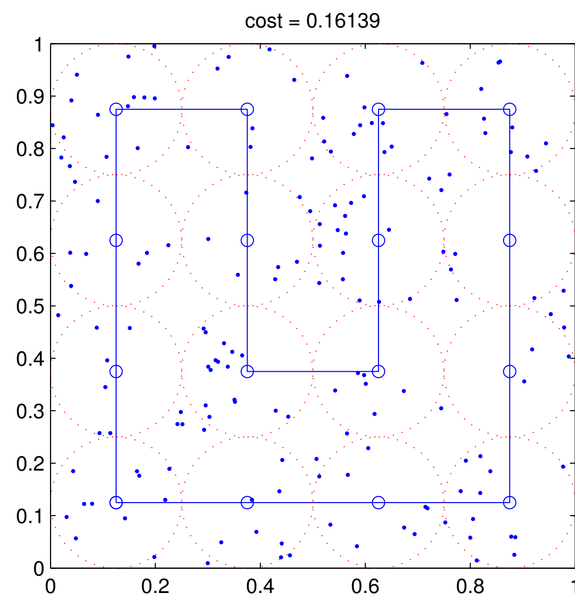


Figure 1: Our initial guess of the optimum solution when $l_r = 1/8$. The dashed circles have radius l_r, and indicate the area that each road segment is meant to cover, but remember that the interior points (shown as dots) should be within l_r of each straight segment, not just the nodes.

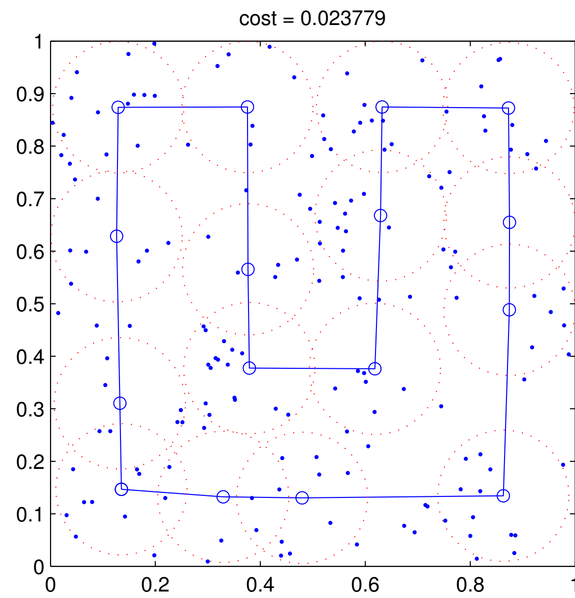


Figure 2: Final solution from initial guess shown in Figure 1.

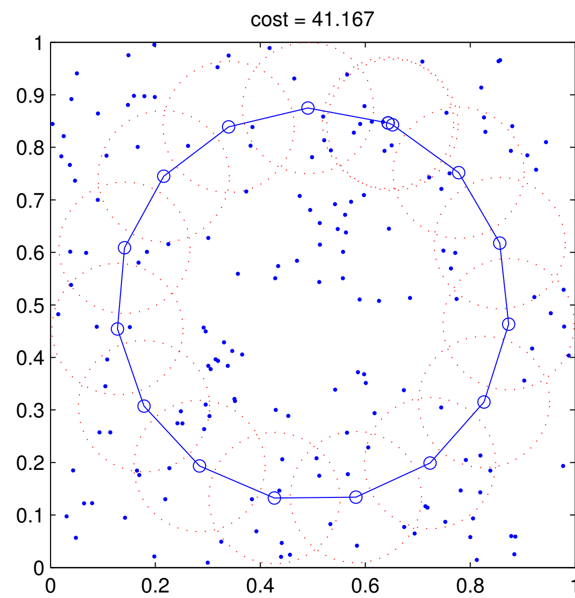


Figure 3: A poor initial guess when $l_r = 1/8$.

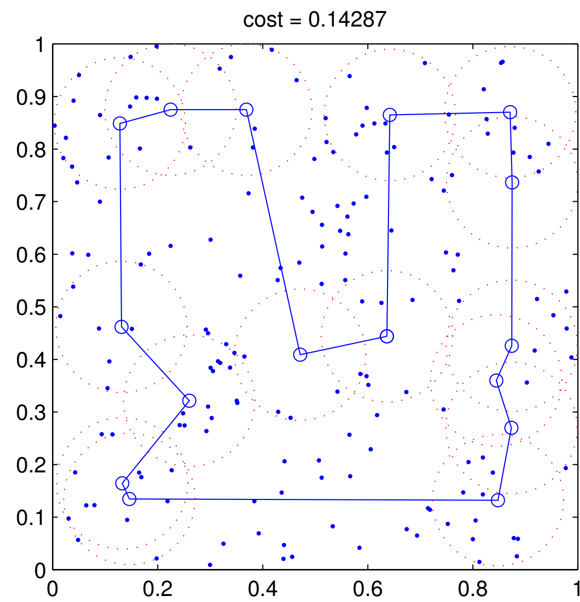


Figure 4: The final solution from the initial guess shown in Figure 3.

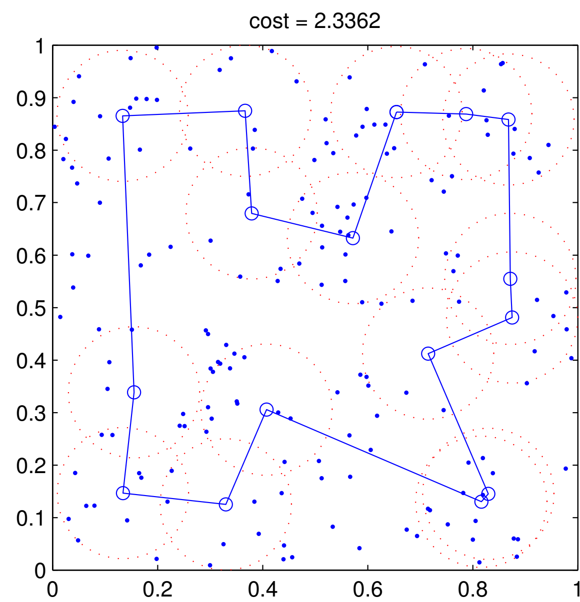


Figure 5: The final solution from the initial guess shown in Figure 3, with a different set of random numbers in the genetic algorithm to that for the solution shown in Figure 4.

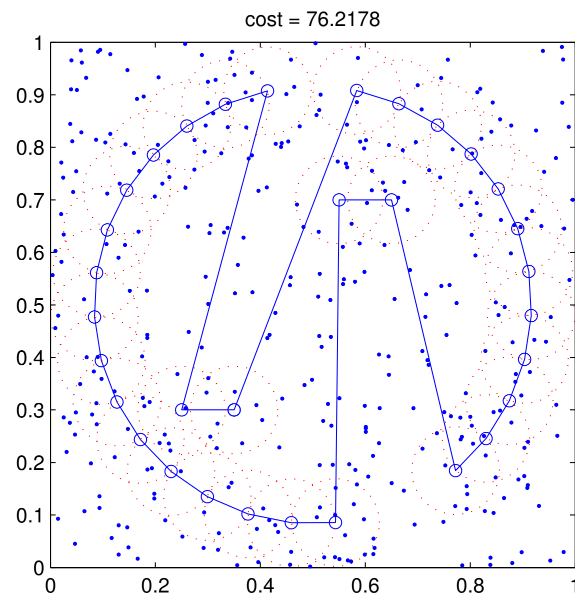


Figure 6: A reasonable initial guess when $l_r = 1/12$.

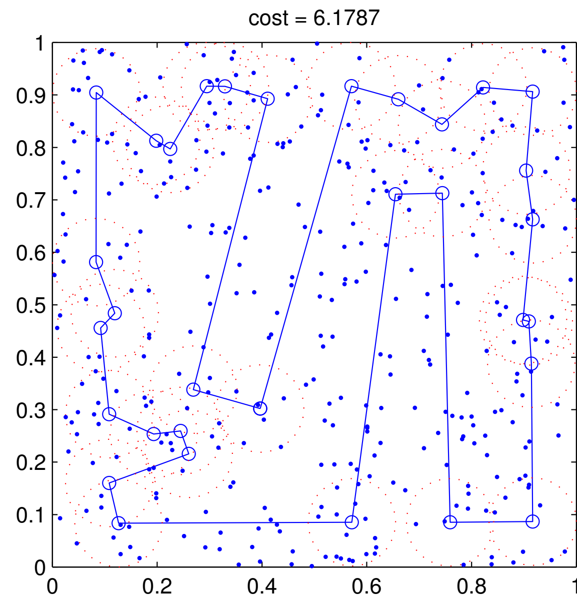


Figure 7: The final solution from the initial guess shown in Figure 6.

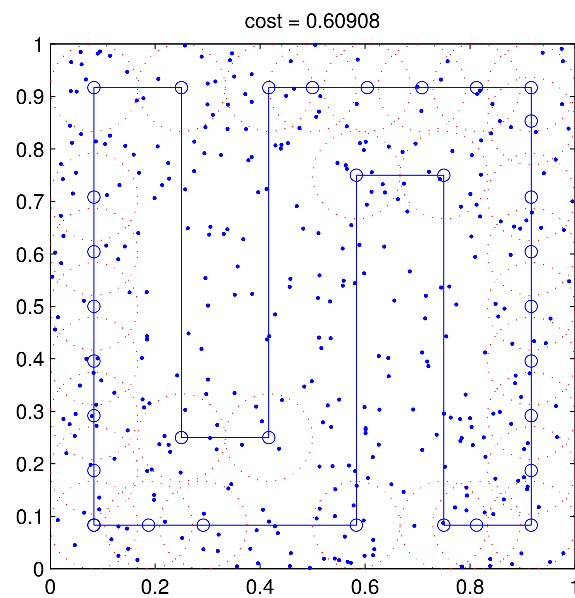


Figure 8: A probable best solution when $l_r = 1/12$.

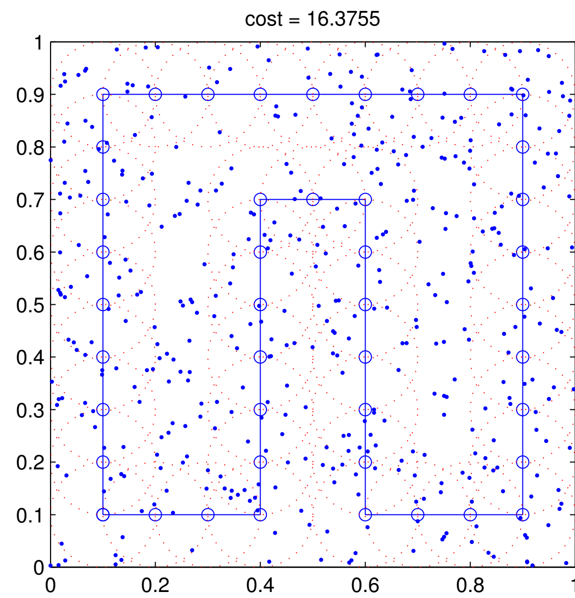


Figure 9: An initial guess when $l_r = 1/10$.

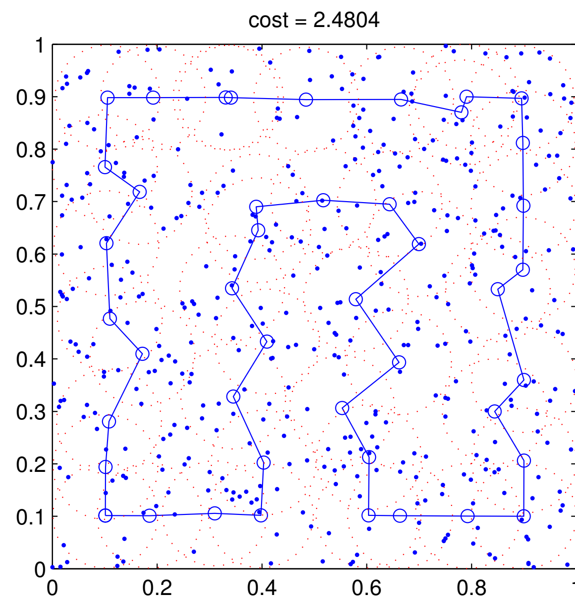


Figure 10: The final solution from the initial guess shown in Figure 9.

(4.9) In conclusion, although there is some potential to this approach, it was not as successful as we had hoped. Further work in this direction could focus on:

- global minimization methods that better suit the structure of the problem, since we do have evidence that the global minimum of the cost function that we have defined does indeed represent a good solution of the problem, but that it is extremely hard to find without an excellent initial guess, which is unlikely to be available in more complex geometries.
- refining the choice of cost function with the aim of eliminating some of the troublesome local minima and/or improving its structure so that it is easier to search using a bespoke global optimization algorithm.

4.2 Variational approach

(4.10) Another approach to finding a road network through a car park Ω that optimises the number of parking spaces that can be fitted around it is to require that each point $\mathbf{x} \in \Omega$ should be 'near' a desired amount of road.

(4.11) In this approach, the quality of the road network is defined as the spatial integral (over Ω) of some function of the absolute value of the difference between the actual and desired road length (at each point).

(4.12) For simplicity we consider only non-branching continuous paths through Ω, i.e. paths given by the images of continuous functions of the form

$$f : [0, 1] \to \Omega$$

(4.13) Physically, it may be appropriate to require that f be piecewise smooth or at least piecewise continuously differentiable.

(4.14) Branching paths and more complex networks could be considered by relaxing the continuity condition on f to piecewise continuity, and forcing boundary conditions on the discontinuity points $z_i \in [0, 1]$.

Physical Justification

(4.15) Finding the densest arrangement of parking spaces and road is equivalent to finding a road network that covers the car park Ω (no point is too far from a road) with sufficient road separation to allow car parking spaces between roads, i.e. no point $\mathbf{x}$ should have too much road in the circle $B_w(\mathbf{x})$ centred around it with radius w, where w is equal to the length of a parking space and half the width of a road.

(4.16) So, finding a good road network is equivalent to minimising the average of some function of the absolute difference between the actual and desired road length in each $B_w(\mathbf{x})$

(4.17) It is not immediately obvious what the desired length of road in each $B_w(\mathbf{x})$ is, so we will define k to be the amount of road in $B_w(\mathbf{x})$ divided by w.

Functional to be minimised

(4.18) The length of $f(y)$ in a region is the integral of its first derivative. So, the amount of road in $B_w(\mathbf{x})$ is given by

$$\int_0^1 \mathcal{I}_{\|f(y)-\mathbf{x}\|_2<w}\; f'(y)\mathrm{d}y$$

(4.19) For simplicity, we choose to make our cost function equal to the square of the difference between the actual and desired amounts of road, so the cost function at each point is

$$\left(\int_0^1 \mathcal{I}_{\|f(y)-\mathbf{x}\|_2<w}\; f'(y)\mathrm{d}y\right)^2$$

and the total cost function (the functional to be minimised) is given by the spatial integral

$$J[f] = \int_\Omega \left(\int_0^1 \mathcal{I}_{\|f(y)-\mathbf{x}\|_2<w}\; f'(y)\mathrm{d}y - kw\right)^2 \mathrm{d}\mathbf{x}\ . \tag{5}$$

Euler-Lagrange

(4.20) Rearranging (5), we can write $J[f]$ as an integral over the parameter of a function $L[z, f, f']$,

$$\begin{aligned} J[f] &= \int_\Omega \left(\int_0^1 \mathcal{I}_{\|f(z)-\mathbf{x}\|_2<w}\; f'(z)\mathrm{d}z - kw\right)\left(\int_0^1 \mathcal{I}_{\|f(y)-\mathbf{x}\|_2<w}\; f'(y)\mathrm{d}y - kw\right)\mathrm{d}\mathbf{x} \\ &= \int_0^1\int_0^1\int_\Omega \left(\mathcal{I}_{\|f(z)-\mathbf{x}\|_2<w}\; f'(z) - kw\right)\left(\mathcal{I}_{\|f(y)-\mathbf{x}\|_2<w}\; f'(y) - kw\right)\mathrm{d}\mathbf{x}\,\mathrm{d}y\,\mathrm{d}z \\ &= \int_0^1 L[z, f(z), f'(z)]\mathrm{d}z \end{aligned}$$

where

$$L[z, f(z), f'(z)] = \int_0^1\int_\Omega \left(\mathcal{I}_{\|f(z)-\mathbf{x}\|_2<w}\; f'(z) - kw\right)\left(\mathcal{I}_{\|f(y)-\mathbf{x}\|_2<w}\; f'(y) - kw\right)\mathrm{d}\mathbf{x}\,\mathrm{d}y\ .$$

Then the Euler-Lagrange equation

$$\frac{\partial L}{\partial f} - \frac{\mathrm{d}}{\mathrm{d}z}\frac{\partial L}{\partial f'} = 0$$

of this functional L is given by

$$\begin{aligned}
&\int_\Omega \Bigg(\Bigg(\Bigg(\Big[f'(z)\cdot(\mathbf{x}-f(z))\delta(\|f(z)-\mathbf{x}\|_2 - w)\Big]\frac{f'(z)}{\|f'(z)\|_2} \\
&\qquad + \mathcal{I}_{\|f(z)-\mathbf{x}\|_2<w}\left[\frac{f'(z)}{\|f'(z)\|_2} - \frac{f''(z)\cdot f'(z)}{\|f'(z)\|_2^3} f'(z)\right]\Bigg) \\
&\qquad \times \int_0^1 \left(\mathcal{I}_{\|f(y)-\mathbf{x}\|_2<w}\|f'(y)\|_2 - kw\right)\mathrm{d}y\Bigg) \\
&+ \Bigg(\Bigg(\Big[f'(z)\cdot(\mathbf{x}-f(z))\delta(\|f(z)-\mathbf{x}\|_2 - w)\Big]\|f'(z)\|_2 + \mathcal{I}_{\|f(z)-\mathbf{x}\|_2<w}\frac{f''(z)\cdot f'(z)}{\|f'(z)\|_2}\Bigg) \\
&\qquad \times \int_0^1 \mathcal{I}_{\|f(y)-\mathbf{x}\|_2<w}\frac{f'(y)}{\|f'(y)\|_2}\mathrm{d}y\Bigg)\Bigg)\,\mathrm{d}\mathbf{x} \\
&= \int_\Omega \Big[(\mathbf{x}-f(z))\delta(\|f(z)-\mathbf{x}\|_2 - w)\|f'(z)\|_2\Big]\int_0^1 \left(\mathcal{I}_{\|f(y)-\mathbf{x}\|_2<w}\|f'(y)\|_2 - kw\right)\mathrm{d}y \\
&\qquad + \left(\mathcal{I}_{\|f(z)-\mathbf{x}\|_2<w}\|f'(z)\|_2 - kw\right)\int_0^1 (\mathbf{x}-f(y))\delta(\|f(y)-\mathbf{x}\|_2 - w)\|f'(y)\|_2 \ .
\end{aligned}$$

5 Optimising over road networks

(5.1) The work in Section 4 describes an attempt to select an optimal road network with the emphasis on making each part of the car park accessible from the road. Presently, the work in Section 4 only considers a single continuous road.

(5.2) The approach in this section bypasses this difficulty and starts with an assumption that a car park designer can use intuition to readily identify a small number of possible candidate road networks within the car park which takes into account all of the complicated geometric features such as the irregular shaped boundary, the position of the entrance and internal obstacles such as lift shafts and stairwells.

(5.3) Presently, a designer would then have to make a detailed assessment of each possible layout which might typically involve the time-consuming job of fully populating parking spaces with cars and making small adjustments to the position of the roads within each given layout to arrive at the total optimal car park capacity for each layout. Only then would the designer be confident of which layout is optimal.

(5.4) It is the purpose of the present approach to develop a coarse algorithm which takes each candidate road network and quickly decides which one (or perhaps more than one) layout offers the best potential for maximising capacity.

(5.5) The approach does not address the actual packing of cars distributed around the network and within the parking lot. This would require a higher level optimisation approach in which individual car parking spaces are distributed in some optimal packing. We give some ideas as to how this might be done in the summary section at the end of this part of the report. It may also prove to be the case that a particular road network selected as 'optimal' by our approach does not give the maximal number of parking spaces within the lot. The algorithm that has been developed does give an output which is indicative of the total capacity and so one ought to be able to identify such cases if predicted capacities for multiple networks are close.

(5.6) The principal reason for this uncertainty is that we make some crude assumptions in our model about how parking spaces are defined. Thus, we adopt the general rule (established in Section 3) that 'nose in' parking (i.e. parking 90 degrees to the road) provides the most efficient packing of cars. Using this we assume that, along each linear segment of the road network, we always attempt to park nose in and make no account for other types of parking tiling patterns such as parallel parking or herring-bone. In spite of this, the output of the algorithm does allow for (in fact, normally, require) angled parking configurations as the nose-in parking stalls assigned in the algorithm can overlap with the boundary of the car park, interior obstacles or with other parking stalls, implying a necessity on angled parking to populate stalls of reduced width. It is the purpose of the algorithm to minimise this overlap and thus produce the most efficient parking solution available.

5.1 Details

(5.7) The user defines a (assumed small) set of plausible networks within the parking lot. These are defined mathematically by a network with N vertices and M line segments each attached to two of the vertices within the network. One of the nodes of the network should be attached to the car park entrance. There no internal obstacles in the examples seen in later figures, but these are not precluded from the problem.

(5.8) It is envisaged that the position of the nodes within the network do not need to be exactly specified by the designer and that it is the particular topology of each network that is important. Thus, we envisage using an optimisation procedure which iterates over the positions of the nodes within the network to yield the maximum capacity for that particular network topology. This is

very similar to the approach outlined in Section 4. However, if the designer is able to supply an estimate of the position of the nodes within the network close to what is perceived as optimal, the optimisation procedure can be expected to converge to the optimal solution fairly rapidly. This is somewhat different to the approach taken in Section 4 where initial configurations are typically taken far away from the final optimised solution and struggle to converge.

(5.9) It is now assumed that along each of the M line segments defining the road network one wishes to park cars at 90 degrees to that segment on both sides of the road ('nose in'), this being optimal for very long line segments. Thus, notionally attached to each line segment are two rectangles either side of the network which include both the width of the road (3m either side of the line defining the road) and the space occupied by cars parked nose in (adding another 5m to the virtual width of that road segment).

(5.10) Thus, in total there are $2M$ rectangular domains assigned to the network which now act as candidates for parking cars (and M rectangular sections of road).

(5.11) A further set of rectangles are also created each of width 5m whose long sides are aligned with the each of the line segments forming the exterior boundary of the car park and all lie outside the car park itself. If the boundary has B sides, then there are B such rectangles. They can be defined by the B vertices that define the boundary.

(5.12) A final set of I rectangles are used to define internal obstacles within the car park (they are all assumed to have rectangular components)

(5.13) We then make a number of calculations based on these $2M$ rectangular parking stalls and the area of overlap with each of the following:

1. other parking stalls or the road – these areas are defined by A_{ij} for $i, j = 1, \ldots, 2M$ and clearly $A_{ij} = A_{ji}$. The total area of overlap between all parking stalls in the car park is

$$A_1 = \sum_{i=1}^{2M-1} \sum_{j=i+1}^{2M} A_{ij};$$

which excludes double counting and stall overlap with itself.

2. the B rectangles defining the exterior boundary – these are B_{ij} where $i = 1, \ldots, M$ and $j = 1, \ldots, B$. The total area of overlap of parking

stalls with the boundary is

$$A_2 = \sum_{i=1}^{2M} \sum_{j=1}^{B} B_{ij};$$

3. the I rectangles defining the internal obstacles – these are I_{ij} where $i = 1, \ldots, M$ and $j = 1, \ldots, I$. The total area of overlap of parking stalls with the interior obstacles is

$$A_3 = \sum_{i=1}^{2M} \sum_{j=1}^{I} I_{ij}.$$

The value of A_1, being representative of overlap between adjacent parking stalls, acts as a summative 'penalty' or 'cost' associated with: (i) bends in the road (the larger the angle the road turns through the higher the cost); (ii) junctions in the network (with right-angled junctions costing less than angled junctions); (iii) roads that are bunched too closely together for optimal nose in parking.

(5.14) The value of A_2, being a summative cost of parking stall overlap with the boundary of the car park give the cost of: (i) roads too close to the wall to allow optimal nose in parking against the wall (with increased penalty with increased overlap); (ii) ends of the road branches which do not meet the wall at right angles (with more cost assigned to more oblique parking against a wall).

(5.15) The value of A_3 acts as a cost for overlapping parking stalls with internal areas, again in proportion to the area of overlap.

(5.16) A further cost can be calculated from the combined values of A_1, A_2 and A_3 above, in addition to knowledge of the areas of each rectangular parking stall the road and the total area of the car park itself, all defined within the algorithm. This is the 'void' area, including the area of the road itself in which parking is not allowed. We call this A_4.

(5.17) In an ideal network, A_1, A_2 and A_3 are all zero whilst A_4 is minimal. This in fact can only happen for a linear car park comprised of a single straight road with the exact amount of space (no more, no less) for exact nose-in parking either side of the road. Any other network which includes non-trivial geometry of the network increases A_1, A_2, A_3 and A_4 (as a percentage of total car park area).

(5.18) Thus we may define a total cost function to be a combination of all A_1, A_2, A_3 and A_4. The simplest such function would simply be a linear weighted

sum of each component, namely

$$A_{total} = \sum_{i=1}^{4} \alpha_i A_i$$

where α_i are user prescribed weights associated with each component. For example, you may decide to penalise overlap with boundary more than overlap between stalls.

(5.19) Another possibility is simply to use A_1 to A_4 along with the areas known to the algorithm to define the total available non-overlapping area in which cars can park, A_{park}, say. Then a crude estimate of the number of cars that can be parked within the car park is given by

$$n_{cars} = [A_{park}/A_{car}]$$

where $[\cdot]$ denotes integer part of and A_{car} is the area taken up by a parking space. Then n_{cars} can be become an objective function which one attempts to maximise. This is probably the most sensible proxy for the cost and this is the measure we use in the examples to define how well each network operates.

(5.20) Another possibility is simply to take A_4, the voided area of the car park and minimise this.

5.2 Examples

(5.21) In these examples, the networks are prescribed by the user and have only been 'hand-optimised'. I.e., the positions of the nodes have been selected manually to get the best parking solutions. In the examples, we are interested in illustrating that one particular candidate network outperforms other networks considered.

(5.22) **Example 1** We start with an example which can be calculated by hand. The car park is rectangular measuring 96m by 48m and has an entrance halfway along the short edge with an opening of 6m. Throughout, roads are 6m wide and parking bays are assumed to 5m by 2.5m. Then with network 1 (shown in figure 9) with horizontal aisles the analytic solution gives a total packing of 211 cars. The algorithm predicts $n_{cars} = 205$, amongst other measures of fitness of the design, as shown below:

```
Total car park area: 4608.0 m^2
Stall area (total): 3120.0 m^2
Stall area (less all overlaps): 2570.0 m^2
Number parked: 205
```

```
Total area per car: 22.4 m^2/car
Overlap area (stall/stall): 250.0 m^2
Overlap area (stall/road): 300.0 m^2
Overlap area (stall/extern): 0.0 m^2
Void area: 2038.0 m^2
```

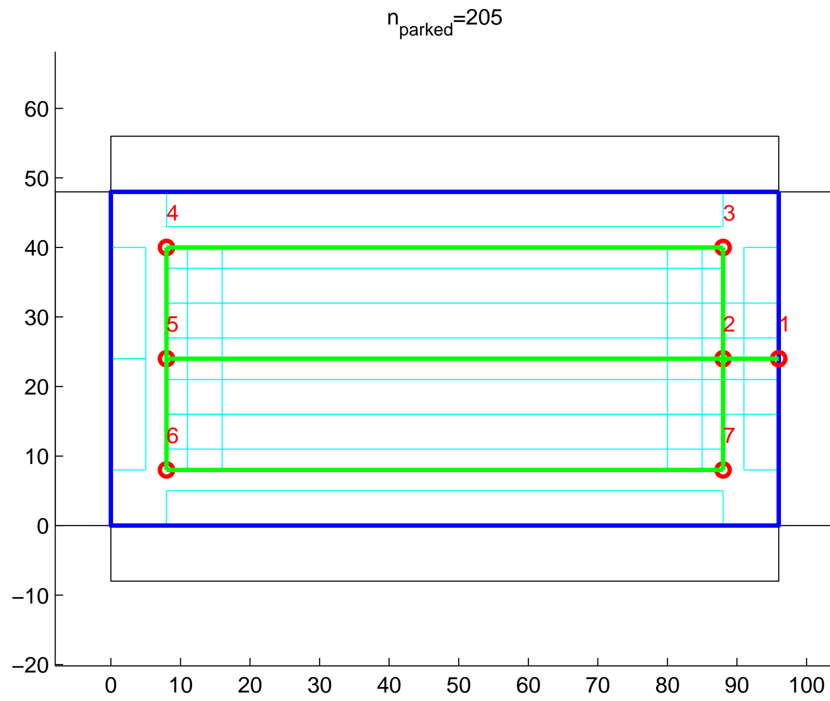
(5.23) With network 2, which contains vertical aisles, the analytic solution gives a total of 195 cars whilst the algorithm predicts $n_{cars} = 191$.

(5.24) In both cases, the network used in the algorithm is prescribed to be exactly that given by the analytic solution. The under prediction in capacity is associated with the way in which parking is only aligned along the segments between the nodes and not along the extensions of those stalls into corners where there is still extra capacity.

(5.25) **Example 2** We use the 'model car park' invented by the study group to test different parking solutions. Its basic shape is that of a 60m by 30m rectangle with a triangle cut off one corner and a square section removed from the opposite corner. The entrance/exit is midway along the upper right-hand edge. The parking spaces are now 4.8m by 2.4m.

(5.26) Here, we do not know which solution is optimal and hence a number of possible networks have been selected by the team working on this problem. The networks are shown in figures 10 and 11, and n_{cars} given in the captions for each. The candidate networks are of two broad classes, characterised by largely 'horizontal' and 'vertical' road layouts (parallel to the long and short edges of the domain respectively). The best of these four networks has a capacity $n_{cars} = 285$, an improvement of some 15% over the worst. In each case, capacity is increased (by 5–10%) by removing circulation from the road network, reducing it to a tree-structure. This makes intuitive sense (minimises road usage), though will of course impact on other measures of design optimality not considered here. On the other hand, the benefits of the 'horizontal' over the 'vertical' network in the tree-structure case are relatively small ($n_{cars} = 285$ and 273 respectively), even though the number of junctions is significantly smaller (4 and 9 respectively).

(5.27) **Example 3** A real car park has been supplied by ARUP. There are 147 parking spaces in this arrangement which is clearly not designed to be optimal. Using the network provided by ARUP and defining the spaces to be slightly longer than before (6.15m by 2.4m) to take some account of the footpaths running alongside parking spaces. Here the algorithm predicts $n_{cars} = 202$: clearly parking spaces have been fitted into spaces left in the original design; see figure 12(a). We improve the design still further, so that $n_{cars} = 220$ by simply moving the bottom perimeter road to a stall-



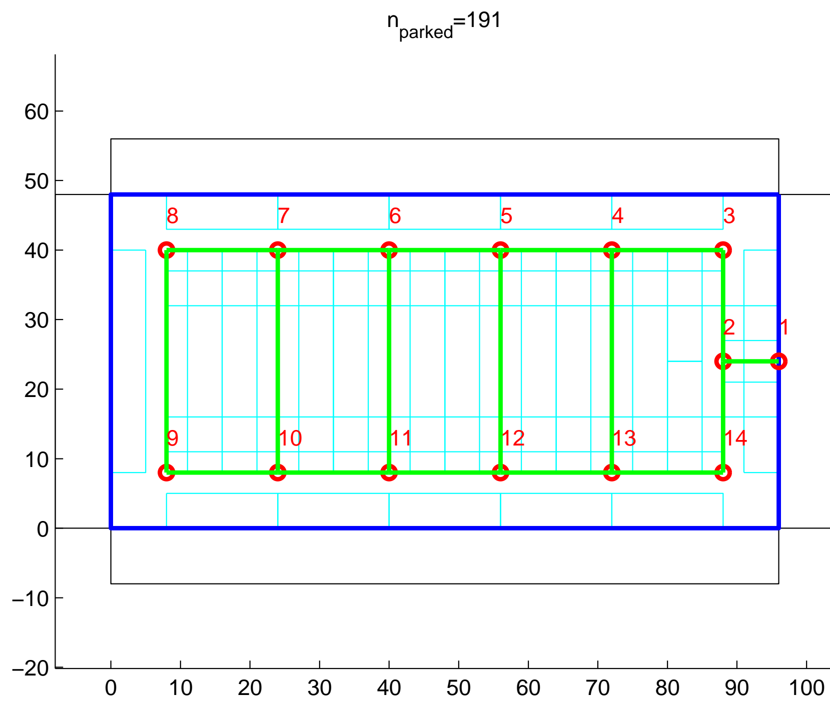


Figure 9: Automatic network capacity calculation: example 1

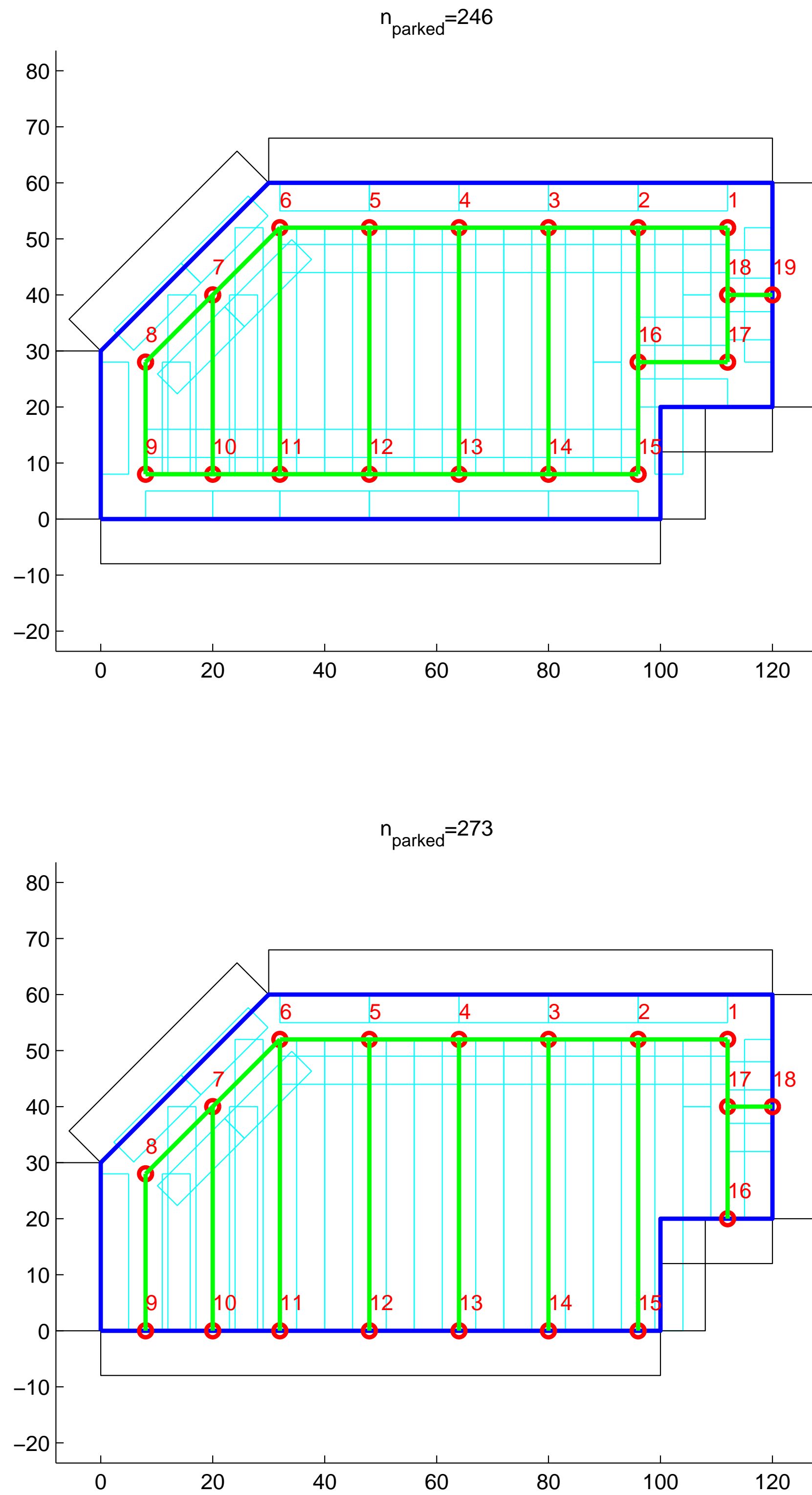


Figure 10: Automatic network capacity calculation: example 2, ‘vertical’ networks.

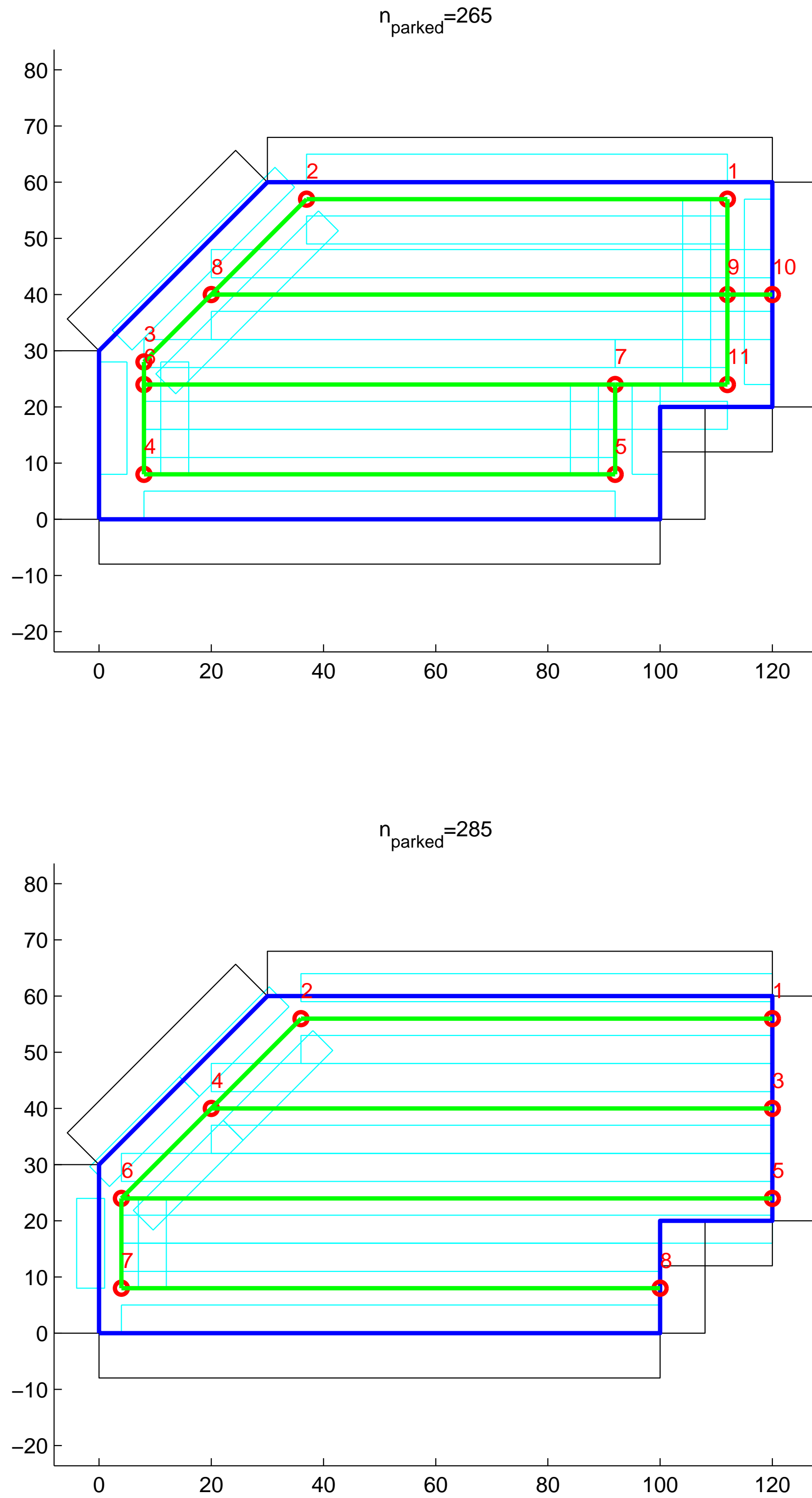


Figure 11: Automatic network capacity calculation: example 2, 'horizontal' networks.

length plus half-road-width from the edge of the domain; see figure 12(b). A similar tactic for the right-hand road would presumably also work well.

(5.28) In figure 13 we show a different road network, motivated by the previously observed design principles to (a) create a tree-structure road network, (b) keep the outermost roads a constant distance from the edge of the domain, and inner roads a constant distance from each other (so as to allow perpendicular parking), and (c) attempt to maximise distances between junctions. The algorithm predicts an increased capacity of $n_{cars} = 233$.

5.3 Conclusions and possible extensions

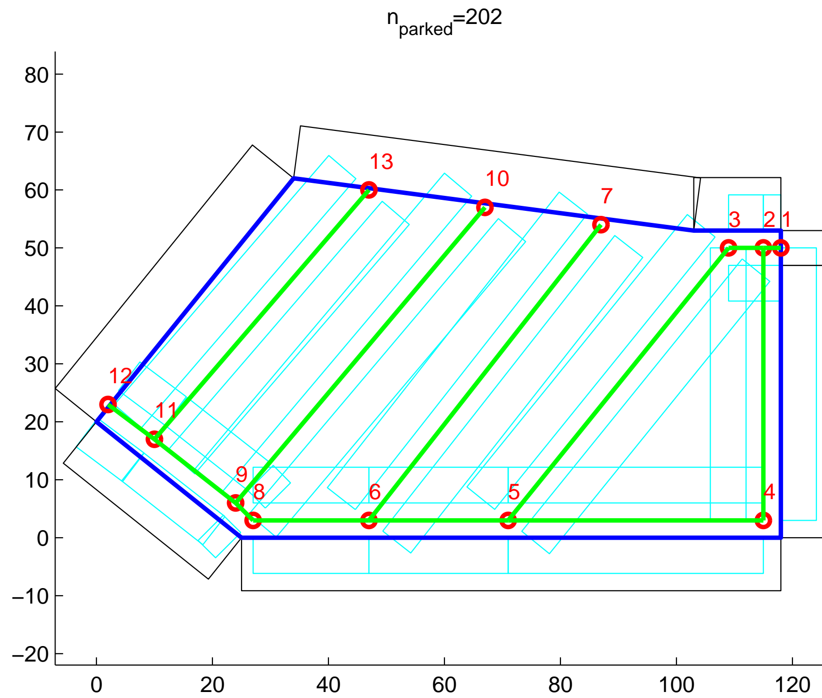
(5.29) We have demonstrated a simple approach which quickly identifies how different road networks within a give car park compare with one another. In a model problem for which analytic solutions are known, the algorithm accurately identifies the better of two possible networks. In more complicated examples, the algorithm also seems to correctly select the best networks from a set of possible networks. The algorithm can be applied to complicated geometries including those with internal obstacles and gives a good approximation to the capacity of a particular network.

Automating the design of the network

(5.30) It may be possible to generate a set of candidate networks for a given car park by using a branching algorithm. In light of the optimal results illustrated in Section 5.2 we see that optimal network have minimal bends in the roads and minimal branches in the networks. So it may be possible to design networks by 'growing' them organically from the entrance with branches being grown dynamically in order to fill the space. See for example [3], [4].

Populating cars around the network

(5.31) Once a network has been optimised and parking stalls have been assigned by the method described in this section, one needs to populate the stalls with cars. One possible approach to do this automatically in a way which is designed to optimise the use of the space within stalls of non-optimal size and shape is the following. One assigns a negative 'charge' along the road network and puts positive charges on the short edge of each individual parking space. Then one populates the available parking spaces with (say) 10% more spaces than predicted by the algorithm above, but makes the space 20% smaller so there is 'plenty of room'. A piece of software is designed which jiggles the spaces around and the spaces become electrically attracted to the roads. Then, dynamically, the parking spaces are increased in size and as this happens the spaces have to continuously realign themselves



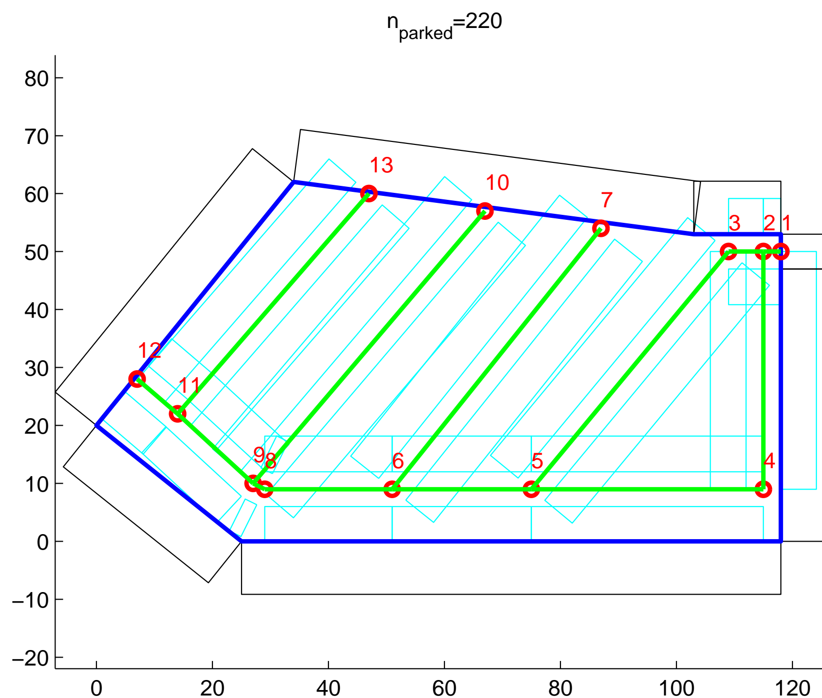
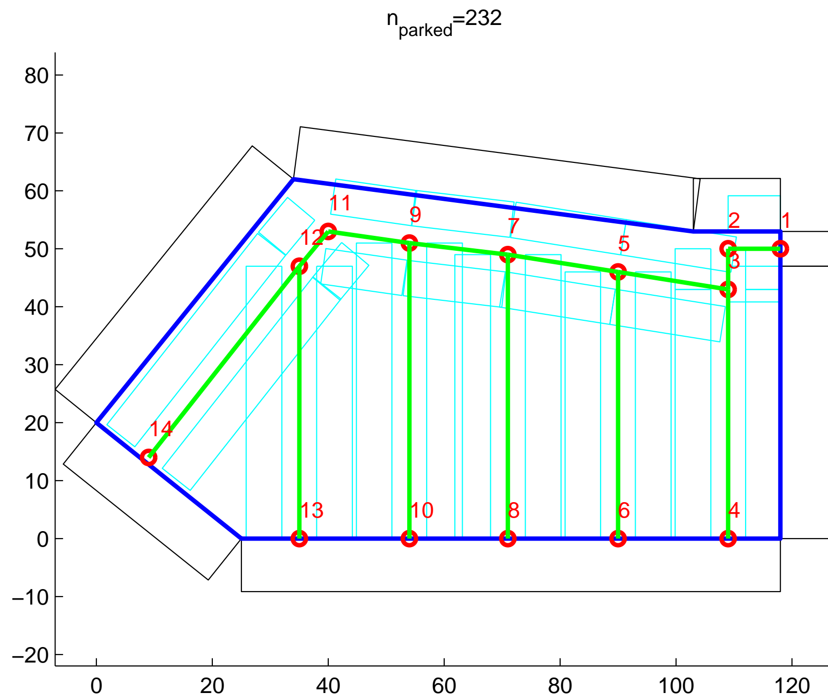


Figure 12: Automatic network capacity calculation: example 3, showing implemented (left) and improved (right) road layouts.



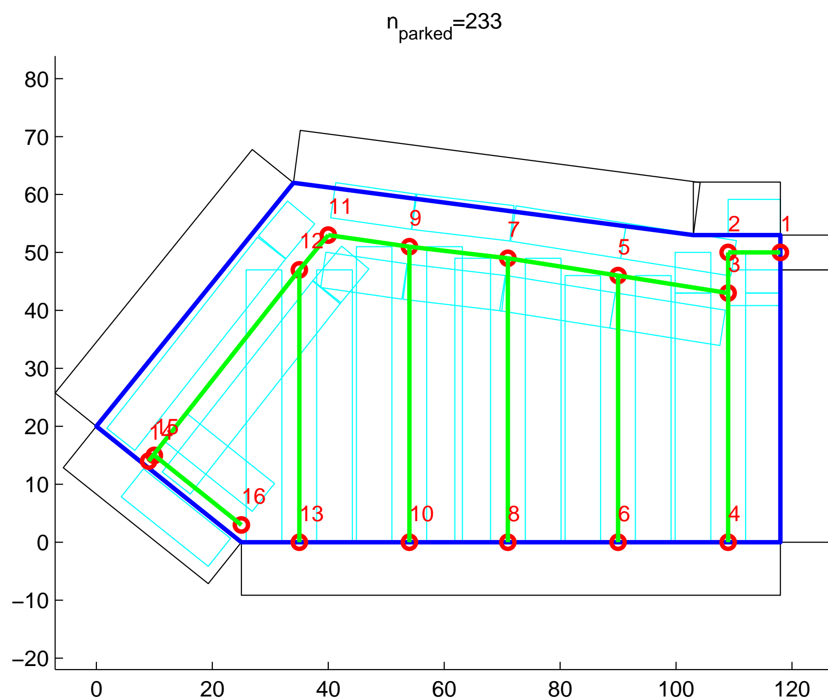


Figure 13: Automatic network capacity calculation: example 3, alternative road layouts.

within the confines of the stalls that have been assigned. At each part of this dynamic growth of the spaces, if a space overshoots the given stall then it is removed and the spaces thus continue to grow until they reach full size. The idea is that at this point the stalls will be populated with the maximum number of spaces, all capable of accessing the road sensibly.

The code to produce the results for this section has been written in MATLAB and is available from M. Homer upon request.

A Appendices

A.1 MATLAB code for the tile and trim approach

(A.1) cf.m

```
function [y,ceq] = cf(alpha, b, h, n, a, r_inner, r_outer)
% This function is the constraint function for the optimization problem.
% What this function represents is the constraint that the total height
% of the cars and the lanes should be less than a.

% Compute the total height of the cars
y = (sum((ones(1,n)*b).*cos(alpha) + (ones(1,n)*h).*sin(alpha)) - a);

% Define function for computing height of lane
swing = @(alpha) 2*cos(alpha)*b + 2*sin(alpha)*h + r_outer - ...
1/2 *(sin(alpha)*r_outer + sin(alpha)*r_inner + sin(alpha)*b + ...
cos(alpha)*h);
H = @(alpha) 2*(b*cos(alpha) + h*sin(alpha));

% Compute height of lanes
for i = 1:2:(n-1)
    y = y + max(swing(alpha(i)) - H(alpha(i))/2, swing(alpha(i+1)) - ...
    H(alpha(i+1))/2);
end

% and possible one more lane at the top
if mod(n,2) == 1
    y = y + swing(alpha(n)) - H(alpha(n))/2;
end

ceq = [];
```

(A.2) IP.m

```
% By Sherif A. Tawfik, Faculty of Engineering, Cairo University
% [x,val,status]=IP1(f,A,b,Aeq,beq,lb,ub,M,e)
% this function solves the following mixed-integer linear programming problem
```

```
%   min f*x
%  subject to
%        A*x <=b
%        Aeq * x = beq
%        lb <= x <= ub
%        M is a vector of indeces for the variables that are constrained to be in
%        e is the integarilty tolerance
% the return variables are :
% x : the solution
% val: value of the objective function at the optimal solution
% status =1 if successful
%        =0 if maximum number of iterations reached in he linprog function
%        =-1 if there is no solution
% Example:
%        maximize 17 x1 + 12 x2
%        subject to
%               10 x1 + 7 x2 <=40
%                   x1 +   x2 <= 5
%                   x1, x2 >=0 and are integers
% f=[-17, -12]; %take the negative for maximization problems
% A=[ 10  7; 1 1];
% B=[40; 5];
% lb=[0 0];
% ub=[inf inf];
% M=[1,2];
% e=2^-24;
% [x v s]= IP(f,A,B,[],[],lb,ub,M,e)

function [x,val,status]=IP1(f,A,b,Aeq,beq,lb,ub,M,e)
options = optimset('display','off');
bound=inf; % the initial bound is set to +ve infinity
[x0,val0]=linprog(f,A,b,Aeq,beq,lb,ub,[],options);
[x,val,status,b]=rec(f,A,b,Aeq,beq,lb,ub,x0,val0,M,e,bound);
% a recursive function that processes the BB tree

function [xx,val,status,bb]=rec(f,A,b,Aeq,beq,lb,ub,x,v,M,e,bound)
options = optimset('display','off');
% x is an initial solution and v is the corressponding objective function value

% solve the corresponding LP model with the integarily constraints removed
[x0,val0,status0]=linprog(f,A,b,Aeq,beq,lb,ub,[],options);

% if the solution is not feasible or the value of the objective function is
% higher than the current bound return with the input intial solution
if status0<=0 | val0 > bound
    xx=x; val=v; status=status0; bb=bound;
    return;
end
```

```
% if the integer-constraint variables turned to be integers within the
% input tolerance return
ind=find( abs(x0(M)-round(x0(M)))>e );
if isempty(ind)
    status=1;
    if val0 < bound
% this solution is better than the current solution hence replace
        x0(M)=round(x0(M));
        xx=x0;
        val=val0;
        bb=val0;
    else
        xx=x;  % return the input solution
        val=v;
        bb=bound;
    end
    return
end

% if we come here this means that the solution of the LP relaxation is
% feasible and gives a less value than the current bound but some of the
% integer-constraint variables are not integers.
% Therefore we pick the first one that is not integer and form two LP problems
% and solve them recursively by calling the same function (branching)

% first LP problem with the added constraint that Xi <= floor(Xi) , i=ind(1)
br_var=M(ind(1));
br_value=x(br_var);
if isempty(A)
    [r c]=size(Aeq);
else
    [r c]=size(A);
end
A1=[A ; zeros(1,c)];
A1(end,br_var)=1;
b1=[b;floor(br_value)];

% second LP problem with the added constraint that Xi >= ceil(Xi) , i=ind(1)
A2=[A ;zeros(1,c)];
A2(end,br_var)=-1;
b2=[b; -ceil(br_value)];


% solve the first LP problem
[x1,val1,status1,bound1]=rec(f,A1,b1,Aeq,beq,lb,ub,x0,val0,M,e,bound);
status=status1;
if status1 >0 & bound1<bound
```

```
% if the solution was successfull and gives a better bound
   xx=x1;
   val=val1;
   bound=bound1;
   bb=bound1;
else
    xx=x0;
    val=val0;
    bb=bound;
end

% solve the second LP problem
[x2,val2,status2,bound2]=rec(f,A2,b2,Aeq,beq,lb,ub,x0,val0,M,e,bound);

if status2 >0 & bound2<bound
% if the solution was successfull and gives a better bound
    status=status2;
    xx=x2;
    val=val2;
    bb=bound2;
end
```

(A.3) drawCarpark.m

```
function drawCarpark(alpha, b, h, a, r_outer, r_inner)

% Define some of the required functions
swing = @(alpha) 2*cos(alpha)*b + 2*sin(alpha)*h + r_outer - ...
1/2 *(sin(alpha)*r_outer + sin(alpha)*r_inner + sin(alpha)*b + cos(alpha)*h);
H = @(alpha) 2*(b*cos(alpha) + h*sin(alpha));

% Plot the overall parking space
spaceWidth=100;
xv = [0 0+spaceWidth 0+spaceWidth 0 0];
yv = [0 0 0+a 0+a 0];
figure(1), plot(xv,yv); axis([0 100 0 a]); axis image;
hold on;

% Loop over each of the rows
y_offset = 0;
% The distance from the bottom of the entire park to the bottom of this row

for i = 1:length(alpha)
    % Plot each of the cars in the row
    car_width = 2/sin(alpha(i));
    num_cars = 100/car_width;
    for j = 1:num_cars % We randomly decide on 10 such cars
        angle = pi/2 - alpha(i);
```

```
            x = (j-1) * 2/sin(alpha(i)) + h*cos(alpha(i));

            xv = [0 b b 0 0];
            yv = [0 0 h h 0];

            R = [cos(angle) -sin(angle);sin(angle) cos(angle)];
            y = y_offset;

            XY = R * [xv;yv];
            figure(1), plot(XY(1,:) + [x x x x x],XY(2,:) + [y y y y y]);
        end

        % Update the vertical offset for this row
        y_offset = y_offset + 1/2*H(alpha(i));

        if mod(i,2) == 1 && i < length(alpha)
            y_offset = y_offset + max(swing(alpha(i)) - H(alpha(i))/2, ...
                           swing(alpha(i+1)) - H(alpha(i))/2);
        end
    end
    hold off;
```

f.m

```
function [y] = f(alpha, b, h, n, a)
% This is the cost function for the optimisation problem.
% What we want to "minimize" is the negative density.
% The density is computed numerically by looking at a strip of width 100.

totalArea = 100*a;

carsArea = 0;
for i = 1:n
    carWidth = 2/sin(alpha(i));
    numCars = 100/carWidth;
    carsArea = carsArea + b*h*numCars;
end

y = -carsArea/totalArea;
```

(A.4) mildlyadvanced.m

```
% This script solves a mixed integer program.
% The problem solved is a simplification where we have fixed the
% rotation to be orthogonal.

clear all;
close all;
```

```
clc;

width = 2;
height = 5;
lambda = 1.5;
a = 15;
n = 10;

% Now we define each of our vectors/matrices
% A, b, f, lb, ub, M
b = [a;0];
f = - ones(1,n) * width * height;
lb = zeros(1,n);
ub = ones(1,n);
M = 1:n;

% Start with an empty A
A = zeros(2, n);
for i = 1:n
    A(1,i) = (1 - (-1)^(i-1))/2 * height + (1 + (-1)^(i-1))/2 * ...
    (height+lambda*height);
    A(2,i) = -((1 + (-1)^(i-1))/2 - (1 - (-1)^(i-1))/2);
end

% Integrality tolerance
e = 2^-24;

[x v s] = IP(f, A, b, [], [], lb, ub, M, e)

rho = -v / (a*width)
```

(A.5) notlp.m

```
% This script solves the non-linear optimisation problem for Hilberts
% car strip.

clear all; clc;

% Specify parameters
width = 2;
height = 5;
a = 26;
r_inner = 4;
r_outer = 7;

% and parameters for the optimisation function
options = optimset('MaxFunEvals',1000,'MaxIter',1000);
```

```
bestfval = 0;

% Try 1 to 50 rows of cars
for n = 1:50
    lb = ones(n, 1) * atan(width/height);
    ub = ones(n, 1) * pi/2;

    % Find optimal solution x. The parameter ones(1,n)*0.3806 is a
    % starting guess which is not in any of the ends.
    [x,fval,exitflag] = fmincon(@(x) f(x, width, height, n, a), ...
    ones(1,n)*0.3806, [], [], [], [], lb, ub, @(x) ...
    cf(x, width,height,n,a,r_inner,r_outer), options);

    number = n
    density = -fval

    % If there is no solution, just stop
    if exitflag < 1
        break;
    end

    % If we get a better solution, save it
    if fval < bestfval
        bestx = x;
        bestfval = fval;
    end
end

% Write out the angles
bestx/pi*180

% and draw the carpark
drawCarpark(bestx, width, height, a, r_outer, r_inner);
```

A.2 Python code for the tile and trim approach

(A.6) main.py

```
import sys
import pygame
from pygame.locals import *
from colors import *
from carspace import *
from room import *

# Init pygame
```

```
pygame.init()

# Define screen size and the color white
size = 1300, 750
white = 255, 255, 255, 255

# Create screen layer
screen = pygame.display.set_mode(size, pygame.DOUBLEBUF)

# Create a convex polygon to be the room
roompol = [(300, 0), (1200, 0), (1200, 400), (1000, 400), (1000, 600),\
          (0, 600), (0, 300)]
carpark = room(roompol)

# Make a list of parking spaces
parkingSpacesUp = [carspace(21*i-40, 585-j*194, 0, True) \
 for i in range(0, 65) for j in range(0, 4)]
parkingSpacesDown = [carspace(21*i-40, 595-j*194-90, 0, False) \
 for i in range(0, 65) for j in range(0, 3)]
parkingSpacesTotal = parkingSpacesUp + parkingSpacesDown

# First we fill the background with cars
screen.fill(white)
for c in parkingSpacesTotal:
    c.draw(screen, False)
pygame.display.flip()

print "Press space to place polygon, arrows to move and x/z to rotate"

while 1:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    screen.fill(white)

    for c in parkingSpacesTotal:
        if c.collidesWithWalls(carpark):
            c.draw(screen, True)

    key_pressed = pygame.key.get_pressed()

    if key_pressed[K_LEFT]:
        carpark.posx -= 2
    if key_pressed[K_RIGHT]:
        carpark.posx += 2
    if key_pressed[K_UP]:
        carpark.posy -= 2
```

```
        if key_pressed[K_DOWN]:
            carpark.posy += 2
        if key_pressed[K_z]:
            carpark.rotation += 0.5
        if key_pressed[K_x]:
            carpark.rotation -= 0.5
        if key_pressed[K_SPACE]:
            print "Time to cut!"
            break
        if key_pressed[K_ESCAPE]:
            pygame.quit()
            sys.exit()

        carpark.draw(screen)

        pygame.display.flip()


    not_clipped = []

    for c in parkingSpacesTotal:
        if c.collidesWithWalls(carpark):
            not_clipped.append(c)

    screen.fill(white)
    carpark.draw(screen)

    for c in not_clipped:
        c.draw(screen, False)

    pygame.display.flip()

    print "Total of: " + str(len(not_clipped)) + " cars"
    raw_input("Done!")
```

(A.7) carspace.py

```
import pygame
from colors import *
from math import *
from room import *


class carspace:
    height = 50
    width = 20

    def __init__(self, posx, posy, rotation, up):
```

```
        self.posx = posx
        self.posy = posy
        self.rotation = rotation

        bbox_height = 60
        bbox_width = 20
        turn_width = 50
        turn_height = 50
        bboxup =    [(self.posx+40, self.posy+50),  # upper left car
                    (self.posx+40-turn_width, self.posy+50-turn_height),
# upper left turn
                    (self.posx+bbox_width+40+turn_width, \
                    self.posy+50-turn_height),  # upper right turn
                    (self.posx+bbox_width+40, self.posy+50),
# upper right car
                    (self.posx+bbox_width+40, self.posy+bbox_height+50),
                    (self.posx+40, self.posy+bbox_height+50)]
        bboxdown = [(self.posx+40, self.posy),
# upper left car
                    (self.posx+bbox_width+40, self.posy),
# upper right car
                    (self.posx+bbox_width+40, self.posy+bbox_height),
# lower right car
                    (self.posx+bbox_width+40+turn_width, \
                    self.posy+bbox_height+turn_height),  # lower right turn
                    (self.posx+40-turn_width, \
                    self.posy+bbox_height+turn_height),  # lower left turn
                    (self.posx+40, self.posy+bbox_height)]  # lower left car

        self.up = up

        if self.up:
            self.polyg = bboxup
        else:
            self.polyg = bboxdown

        self.carsprite = pygame.sprite.Sprite()  # create sprite

        if up:
            self.carsprite.image = \
            pygame.image.load("images/car_full2.png").convert_alpha()
# load ball image
        else:
            self.carsprite.image = \
            pygame.image.load("images/car_full3.png").convert_alpha()
# load ball image

        self.carsprite.rect = self.carsprite.image.get_rect()
```

```
# use image extent values
        self.carsprite.rect.topleft = [0, 0]
# put the ball in the top left corner

    def collidesWithWalls(self, room):

        roompolyg = room.getActualPolygon()

        for c in self.polyg:
            translated_point = (c[0], c[1])

            if not inside_complex_polygon(translated_point, roompolyg):
                return 0
        return 1

    def rotate(self, degrees):
        self.rotation += degrees

    def draw(self, screen, collides):
        if collides:
            #self.carsprite.set_alpha(50)
            screen.blit(self.carsprite.image, (self.posx, self.posy))
        else:
            #self.carsprite.set_alpha(255)
            screen.blit(self.carsprite.image, (self.posx, self.posy))
```

(A.8) colors.py

```
white = 255, 255, 255, 255
black = 0, 0, 0, 255
blue = 0, 0, 255, 255
red = 255, 0, 0, 255
green = 0, 255, 0, 255
gray = 100, 100, 100, 255
brown = 200, 150, 150, 255
pink = 255, 100, 180, 255
```

(A.9) room.py

```
import pygame
import random
from pygame.locals import *
from math import *
from colors import *


class room:
#A class for holding an entire parking lot to be filled with parking spaces
```

```
        posx = 0
        posy = 0
        rotation = 0

        def __init__(self, polyg):
            self.polyg = polyg
            self.color = white

            self.width = self.computeWidth()
            self.height = self.computeHeight()

        def computeWidth(self):
            minx = 100000
            maxx = -100000

            for p in self.polyg:
                if p[0] < minx:
                    minx = p[0]
                if p[0] > maxx:
                    maxx = p[0]

            return maxx-minx

        def computeHeight(self):
            miny = 100000
            maxy = -100000

            for p in self.polyg:
                if p[1] < miny:
                    miny = p[1]
                if p[1] > maxy:
                    maxy = p[1]

            return maxy-miny

        def getActualPolygon(self):
            centerx = self.posx + self.width/2
            centery = self.posy + self.height/2
            polygonToDraw = [rotate2d(self.rotation, (e[0] + self.posx, \
              e[1] + self.posy), (centerx, centery)) for e in self.polyg]
            return polygonToDraw

        def draw(self, screen):
            pygame.draw.polygon(screen, black, self.getActualPolygon(), 2)


    def createRandomRoom(cx, cy, diameterx, diametery, numpoints):
```

```
    p = makeRandomData(numpoints, diameterx, diametery)
    conH = list(convexHull(p))

    coords = []

    for i in conH:
        l = list(i)
        l[0] += cx
        l[1] += cy
        coords.append(tuple(l))

    coords = tuple(coords)

    return room(coords)


def inside_complex_polygon(r, P):
    # Determine if the point r is inside the polygon P
    polySides = len(P)  # Number of corners of polygon
    polyX = [e[0] for e in P]
    polyY = [e[1] for e in P]
    x = r[0]
    y = r[1]

    j = polySides - 1

    oddNodes = False

    for i in range(polySides):
        if (polyY[i] < y and polyY[j] >= y) or \
           (polyY[j] < y and polyY[i] >= y):
            if (polyX[i] + (y-polyY[i]) / \
               (polyY[j] - polyY[i])*(polyX[j]-polyX[i]) < x):
                oddNodes = not oddNodes
        j = i

    return oddNodes


def inside_convex_polygon(r, P):
    # Determine if the point r is inside the polygon P
    sign = 0
    n_vertices = len(P)

    for n in xrange(n_vertices):
        segment = P[n], P[(n+1) % n_vertices]
        affine_segment = (segment[1][0] - segment[0][0], \
                          segment[1][1] - segment[0][1])
```

```
        affine_point = (r[0] - segment[0][0], r[1] - segment[0][1])
        k = x_product(affine_segment, affine_point)

        if k != 0:
            k = int(k / abs(k))  # normalized to 1 or -1
            if sign == 0:  # the first case
                sign = k
            elif k != sign:
                return False
        else:
            return False

    return True


def x_product(a, b):
    return a[0]*b[1]-a[1]*b[0]


def myDet(p, q, r):
    #Calc. determinant of a special matrix with three 2D points.

    #The sign, "-" or "+", determines the side, right or left,
    #respectivly, on which the point r lies, when measured against
    #a directed vector from p to q.

    # We use Sarrus Rule to calculate the determinant.
    # (could also use the Numeric package...)
    sum1 = q[0]*r[1] + p[0]*q[1] + r[0]*p[1]
    sum2 = q[0]*p[1] + r[0]*q[1] + p[0]*r[1]

    return sum1 - sum2


def isRightTurn((p, q, r)):
    #Do the vectors pq:qr form a right turn, or not?

    assert p != q and q != r and p != r

    if myDet(p, q, r) < 0:
        return 1
    else:
        return 0


def makeRandomData(numPoints=10, sqrLength=100, sqrHeight=100):
    #Generate a list of random points within a square.
```

```
    # Fill a square with random points.
    xmin, xmax = 0, sqrLength
    ymin, ymax = 0, sqrHeight
    P = []

    for i in xrange(numPoints):
        rand = random.randint
        x = rand(xmin+1, xmax-1)
        y = rand(ymin+1, ymax-1)
        P.append((x, y))

    return P


def convexHull(P):
    # Calculate the convex hull of a set of points.

    # Get a local list copy of the points and sort them lexically.
    unique = {}
    for p in P:
        unique[p] = 1

    points = unique.keys()
    points.sort()

    # Build upper half of the hull.
    upper = [points[0], points[1]]
    for p in points[2:]:
        upper.append(p)
        while len(upper) > 2 and not isRightTurn(upper[-3:]):
            del upper[-2]

    # Build lower half of the hull.
    points.reverse()
    lower = [points[0], points[1]]

    for p in points[2:]:
        lower.append(p)
        while len(lower) > 2 and not isRightTurn(lower[-3:]):
            del lower[-2]

    # Remove duplicates.
    del lower[0]
    del lower[-1]

    # Concatenate both halfs and return.
    return tuple(upper + lower)
```

```
def rotate2d(degrees, point, origin):
    # A rotation function that rotates a point around a point to
    # rotate around the origin use [0,0]

    # The idea is to first translate the point such that the
    # rotation is about the origin, rotate and then move it back again.
    x = point[0] - origin[0]
    y = point[1] - origin[1]
    newx = (x*cos(radians(degrees))) - (y*sin(radians(degrees)))
    newy = (x*sin(radians(degrees))) + (y*cos(radians(degrees)))
    newx += origin[0]
    newy += origin[1]

    return newx, newy
```

Bibliography

[1] http://en.wikipedia.org/wiki/Hilbert's_paradox_of_the_Grand_Hotel

[2] J. D. Hill, G. Rhodes, S. Voller and C. Whapples (2005) *Car Park Designers' Handbook*, Thomas Telford Ltd.

[3] Q. Xia (2010) Numerical simulation of optimal transport paths, *Second International Conference of Computer Modeling and Simulation.* DOI 10.1109/ICCMS.2010.30

[4] Q. Xia (2007) The Formation of a tree leaf. *ESAIM: Control, Optimisation and Calculus of Variations.*