# Net2Plan 0.1.8 Manual

## Contents

# 1 Description of the document

This guide identifies installation prerequisites, describes how to install Net2Plan, and explains how to run it and extend its functionalities.

Contents

# 2 What is Net2Plan?

Net2Plan is an open-source tool devoted to the optimization and planning of communication networks. The tool is implemented as a MATLAB toolbox together with a Graphical User Interface (GUI).

Net2Plan has been thought as a tool to assist the teaching of communication networks planning courses. It allows testing several built-in algorithms and users (i.e. students) can easily implement and test their own-made algorithms. This process is facilitated by a set of included libraries. The integration of the user-made algorithms in the tool is straightforward. The algorithms are implemented as MATLAB .m files with a mandatory format for input and output parameters. Algorithms in this form can be readily integrated into Net2Plan by simply storing them in the appropriate directory (see section 10).

Contents

# 3 Background

## 3.1 Topology

In Net2Plan a network topology is defined as a graph $G(N, E)$, where $N$ is the set of nodes and $E$ is the set of links. The number of nodes and links are the cardinality of sets $N$ and $E$, which are $|N|$ and $|E|$, respectively. Network topologies are considered as *multi-digraphs*. This means that every link is unidirectional (directed graph or digraph), and that network topologies can have multiple links between the same node pair. No self-links are allowed.

Given a link $e \in E$, $a(e)$ denotes the initial node of the link, and $b(e)$ denotes its destination node. We use $l(e)$ to denote the link length (measured in km). Given a node $n$, $\delta^+(n)$ is the set of outgoing links from $n$ ($\delta^+(n) = \{e \in E : a(e) = n\}$) and $\delta^-(n)$ is the set of incoming links to $n$ ($\delta^-(n) = \{e \in E : b(e) = n\}$). In Fig. 1 an example of network topology is shown.

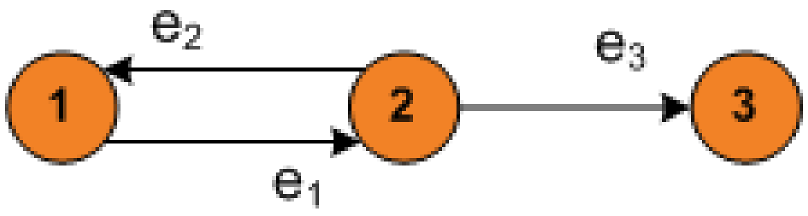


Figure 1: Topology with $|N| = 3$ nodes, $N = \{n_1, n_2, n_3\}$, $|E| = 3$, $E = \{e_1, e_2, e_3\}$. End nodes of $e_1$ are $a(e_1) = n_1$ and $b(e_1) = n_2$. Outgoing links from node $n_2$ are $\delta^+(n_2) = \{e_2, e_3\}$, and the incoming one is $\delta^-(n_2) = \{e_1\}$

Given a network topology $G(N, E)$, a path $p$ is an ordered sequence of links $p = (e_1, \ldots, e_k)$, such that the destination node of a link $e_i$ is the origin node of the following link $e_{i+1}$. The set of all possible paths in a network is denoted as $P$. The first node of the path $a(p)$ is the origin node of the first link $a(e_1)$, and the last node in the path $b(p)$ is the destination node of the last link $b(e_k)$. Finally, given a link $e \in E$, $P_e$ is the set of paths which traverse the link $e$. For example in Fig. 2, given $p_1 = \{e_{12}, e_{23}, e_{34}\}$ and $p_2 = \{e_{42}, e_{23}\}$, $P_{e_{23}} = \{p_1, p_2\}$. In addition, $l(p)$ is the number of hops in path $p$.

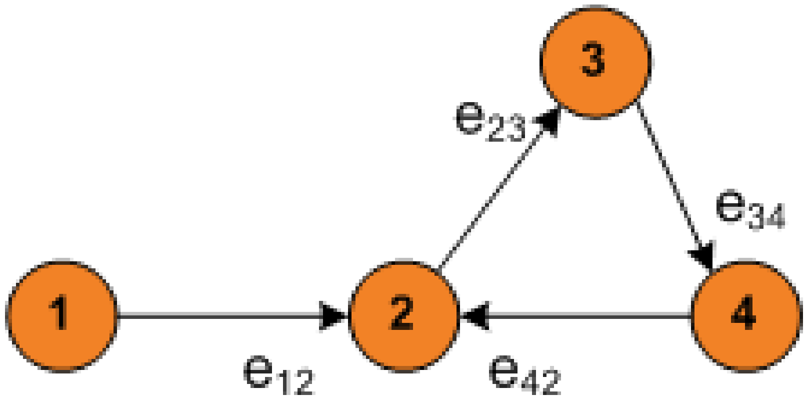


Figure 2: Topology with $|N| = 4$ nodes, $N = \{n_1, \ldots, n_4\}$, $|E| = 4$, $E = \{e_{12}, e_{23}, e_{34}, e_{42}\}$

An important concept in networks is the so-called *connectivity*. A network is connected if it is possible to find a path from every node to each other. For example, the topology in Fig. 1 is not connected since nodes 1 and 2 are not reachable from node 3. In its turn, the topology in Fig. 2 is connected.

## 3.2 Installed capacities

Each node $e$ in a network has associated a real number $u_e \geq 0$ which represents its *capacity*. The capacity is the amount of traffic the link is able to carry. Thus, capacities and traffics are measured in the same units. Unless stated otherwise, the traffic and capacities in Net2Plan are measured in Erlangs.

For the sake of simplicity, we denote the set of link capacities as a vector $\mathbf{u} = \{u_e, e \in E\}$. In addition, the carried traffic by link $e$ is denoted by $y_e$. The vector $\mathbf{y} = \{y_e, e \in E\}$ represents the traffic carried in all the links in the network.

Typically link capacities are limited to a discrete range of values due to technological reasons, such as STM-$N$ carriers in SDH. In these cases, capacities are referred as *modular capacities*.

## 3.3 Traffic model

Traffic is modelled through a set of *demands* (or *commodities*) $D$. Each demand $d \in D$ represents an offered traffic flow to the network. In general, a the traffic of a demand $d$ can have one or more ingress nodes in the set $a(d)$, and one or more egress nodes denoted by the set $b(d)$; however, in current version of Net2Plan demands are considered *unicast*. This means that each demand $d$ has one ingress node and oner egress node, thus $|a(d)| = |b(d)| = 1$. In addition, self-demands are not allowed.

The offered traffic by a demand $d$ is represented as $h_d$. Such value is measured in the same units as link capacities $\mathbf{u}$ (i.e. Erlangs). In some situations, it is not possible to carry all the offered traffic by the demands. Then, $r_d \leq h_d$ represents the amount of traffic belonging to demand $d$ that is carried. In their vector form, $\mathbf{h} = \{h_d, d \in D\}$ and $\mathbf{r} = \{r_d, d \in D\}$ represent offered and carried traffic, respectively.

A simplified approach to model the offered traffic between nodes is the so-called *traffic matrix*. A traffic matrix is a $|N|$x$|N|$ matrix (where $|N|$ is the number of nodes in the network) in which each pair ($i$,$j$) represents the traffic from node $i$ to node $j$.

The main drawback of representing the offered traffic by means of traffic matrices, is that the traffic matrix representation assumes that at most one demand exists between each pair of nodes. Then, if we compute the traffic matrix representation of a demand set $D$ where some node pairs have more than one demand between them, an ambiguity can occur. This situation is posed in Fig. 3, where demands 1 and 2 from the demand set on the left are grouped in a single entry in the traffic matrix. The same result is obtained with the demand set on the right.

$$\begin{array}{cccc} \hline d & a_d & b_d & h_d \\ \hline 1 & 1 & 2 & 5 \\ 2 & 1 & 2 & 3 \\ 3 & 2 & 1 & 4 \\ \hline \end{array} \rightarrow \begin{pmatrix} 0 & 8 \\ 4 & 0 \end{pmatrix} \leftarrow \begin{array}{cccc} \hline d & a_d & b_d & h_d \\ \hline 1 & 1 & 2 & 8 \\ 2 & 2 & 1 & 4 \\ \hline \end{array}$$

Figure 3: Example of ambiguity in traffic matrices

## 3.4 Routing

Routing is the process of selecting paths in a network along which to send the traffic. In Net2Plan, the routing is represented in the form of *demand-path variables*. As stated in section 3.1, where paths were defined as sequences of links from a origin node to a destination node, a path for a demand is defined as a path from the ingress node to the egress node of that demand.

Formally, for each demand $d$ in the network, a set of paths $P_d \subset P$ is defined between the end nodes of the demand ($a(d) = a(p), b(d) = b(p)$). Then, traffic routing is represented by a vector:

$$\mathbf{x} = \{x_{dp}, d \in D, p \in P_d\}$$

where $x_{dp}$ is the fraction of the offered traffic $h_d$ of demand $d$, that is carried through the path $p$. We denote as $P_{de}$ the set of paths in $P_d$ that traverses link $e$ ($P_{de} = P_d \bigcap P_e$)

Next, two typical routing constraints are described:

- *Unsplittable or non-bifurcated routing*: In the routing is non-bifurcated, each traffic demand is carried by at most one path. Conversely, if a demand $d$ is carried by more than one path (i.e. $x_{dp_1} > 0, x_{dp_2} > 0$, for two different paths $p_1, p_2$), we say that the routing of the demand is *balanced* among those paths.

- *Integral flows routing*: Force to carry integer amounts of traffic. In some cases, $h_d$ is measured on integer units and it has no sense to carry non-integer fractions of traffic (i.e. technologies based on virtual circuits). Thus an integral routing is constrained to carry an integral amount of traffic in each path (or an integer multiple of a base value).

## 3.5 Quick reference card

Table 1 summarizes information about background underlying Net2Plan.

| Element | Parameter | Description |
|---|---|---|
| Nodes | $N$ | Set of nodes $n \in N$ |
| | $\delta^{+}(n)$, $\delta^{-}(n)$ | Set of outgoing and incoming links from/to node $n$ |
| Links | $E$ | Set of links $e \in E$ |
| | $a(e)$, $b(e)$ | Origin and destination nodes of link $e$ |
| | $l_e$ | Length of link $e$ (Km) |
| | $u_e$ | Capacity of link $e$ (Erlangs) |
| | $\mathbf{u}$ | Vector form of $u_e$ |
| | $y_e$ | Traffic carried by link $e$ (Erlangs) |
| | $\mathbf{y}$ | Vector form of $y_e$ |
| | All links are considered unidirectional | |
| | Self-links are not allowed $(a(e) \neq b(e))$ | |
| Paths | $P$ | Set of all possible paths in the network $p \in P$ |
| | $P_e$ | Subset of the paths in $P$ that traverse link $e$ |
| | $p = \{\dots\}$ | Sequence of links traversed in path $p$ |
| | $a(p)$, $b(p)$ | Origin and destination nodes of path $p$ |
| | $l(p)$ | Number of hops in path $p$ |
| Demands | $D$ | Set of demands $d \in D$ |
| | $a(d)$, $b(d)$ | Ingress and egress nodes of demand $d$ |
| | $h_d$ | Offered traffic for demand $d$ (Erlangs) |
| | $\mathbf{h}$ | Vector form of $h_d$ |
| | $r_d$ | Carried traffic for demand $d$ (Erlangs) |
| | $\mathbf{r}$ | Vector form of $r_d$ |
| | $P_d \subset P$ | Set of valid paths defined to carry traffic from demand $d$ $(a(d) = a(d)$ and $b(p) = b(d))$ |
| | Demands are unicast | |
| | Self-demands are not allowed $(a(d) \neq b(d))$ | |
| | $P_{de}$ | Set of valid paths for $d$ that traverse $e$ $(a(d) = a(d)$ and $b(p) = b(d))$ |
| Routing | $x_{dp}$ | Fraction of $h_d$ carried by $p \in P_d$ |
| | $\mathbf{x}$ | Vector form of $x_{dp}$ |

Table 1: Summary of network elements involved in Net2Plan

Contents

# 4 Addressing network design problems

In practical network design different variables can be involved: link capacities, the traffic routing, the topological design of the network, the storage capacity at each node, and so on. Usually, network design problems receive some of this information as input parameters (e.g. traffic demand, and network topology) and try optimize the rest (e.g. capacities in the links and traffic routing) according to a performance merit of interest. Clearly, the number of possible variants and subtypes of network design problems is infinite. Moreover, different technologies add their own particular aspects to network design. For this reason, network design has become a mixture of art and engineering.

In an attempt to provide a (somehow) systematic criteria to cathegorize network design problems, in Net2Plan we adopted the following scheme, which is just an extension of the network design problems' taxonomy in Kleinrock's book [1]:

- *Topology Assignment* (TA): The nodes and/or links in the network are decision variables to optimize (graph $G(N, E)$).
- *Capacity Assignment* (CA): The capacities in the links are decision variables to optimize (vector $\mathbf{u} = \{u_e, e \in E\}$)
- *Flow Assignment* (FA): The routing of the traffic demands is to be optimized (vector $\mathbf{x} = \{x_{dp}, d \in D, p \in P_d\}$)
- *Bandwidth Assignment* (BA): The traffic carried by each demand is to be optimized (vector $\mathbf{r} = \{r_d, d \in D\}$)

According to this naming scheme, combinations of these problems are named combining the acronyms. For example, a *capacity and flow assignment* (CFA) problem involves the joint computation of routing and link capacities. We remark that this taxonomy should be considered as an attempt to give a didactic organization to the utmost diversity of planning problems that arise in communication networks.

Contents

## 4.1 Algorithms

Typically network design problems are presented as optimization problems, this means that there are a set of performance criteria to maximize or minimize, subject to some constraints (qualitative statements about network design and performance), given a set of input parameters (i.e. partial network designs). Optimization algorithms (or just "algorithms") are the methods that compute a numerical solution to a given problem instance. In Net2Plan, an algorithm is a .m file with a given signature (see section 8 for more information) that fix the format of the input and output parameters.

Net2Plan is divided into two main parts: graphical user interface (GUI) and kernel. The main idea is that all the design algorithms, independently of their specific target, receive as input parameter the current network structure (see section 4.2), and return an updated version of this network structure. Then, the Net2Plan kernel and GUI, are devoted to just process the designs returned by the algorithms: check its validity (e.g. the topology has a correct format, the links are not oversubscribed etc), graphically display the network design, and compute some reports and performance merits.

The idea behind Net2Plan is that *you can progressively design your network.* This is, you can chain successive algorithms, each one completing a part of the network design. For instance, you can start with a network where only the nodes are defined. Then, an algorithm is used to define the links in the network according to some figure of performance. Afterwards, an algorithm can be run to jointly decide on the capacity in the links and routing of the flows, for a given traffic matrix. As a result, Net2Plan can be a powerful tool for *communications network planning* courses, since students can see step by step how their designs grow.

Net2Plan assists the task of creating and evaulating network design algorithms by providing built-in example algorithms. In section 10, a list of built-in algorithms can be found. In section 8 we explain in more detail how to integrate new algorithms in Net2Plan.

Contents

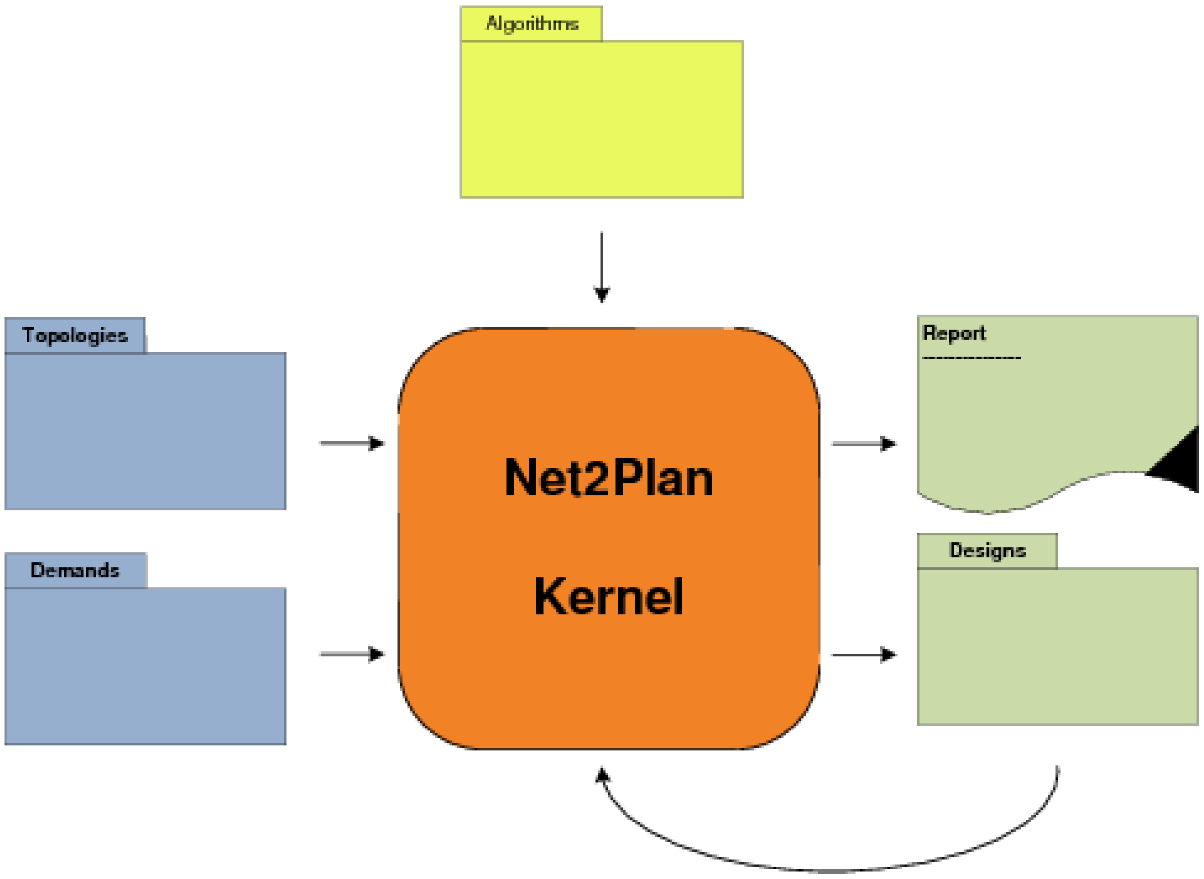


Figure 4: Design flow in Net2Plan

## 4.2 Network Structure

In Net2Plan, data files are stored as .xml files containing all kind of information: network topology, traffic demands, routing, and so on. This information is organized in a hierarchy of layers:

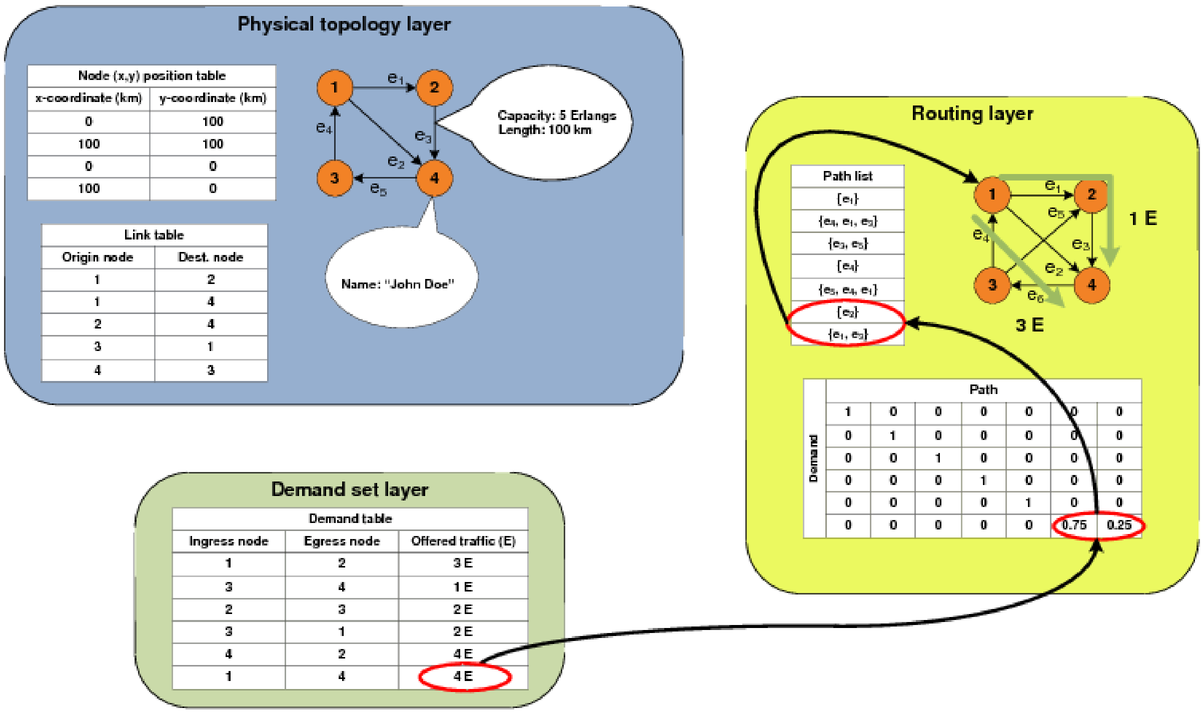
- Physical topology layer: Information about nodes and links
- Demand set layer: Information about traffic demands
- Routing layer: Information about traffic routing over the physical topology

*Physical topology layer* contains $(X, Y)-$coordinates of nodes, their names and a set of optional and user-defined attributes per each node. In addition, it contains origin and destination node of each link, its length and capacity, and also a set of optional per-link attributes.

*Demand set layer* contains ingress and egress nodes, average offered and carried traffic volume and a set of optional and user-defined attributes per demand. It contains information about carried traffic, if routing layer information is available.

*Routing layer* contains a set of paths to carry traffic from demands across the network. Each path is represented by the set of links traversed in that path. In addition, it contains a routing matrix which shows how traffic demands are carried over the paths.

Fig. 5 shows an example of complete network design including all items stated previously.



| Node (x,y) position table | |
|---|---|
| x-coordinate (km) | y-coordinate (km) |
| 0 | 100 |
| 100 | 100 |
| 0 | 0 |
| 100 | 0 |

| Link table | |
|---|---|
| Origin node | Dest. node |
| 1 | 2 |
| 1 | 4 |
| 2 | 4 |
| 3 | 1 |
| 4 | 3 |

| Path list |
|---|
| $\{e_1\}$ |
| $\{e_4, e_1, e_3\}$ |
| $\{e_3, e_5\}$ |
| $\{e_4\}$ |
| $\{e_5, e_4, e_1\}$ |
| $\{e_2\}$ |
| $\{e_1, e_3\}$ |

| | Path | | | | | | |
|---|---|---|---|---|---|---|---|
| Demand | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| | 0 | 0 | 0 | 0 | 0 | 0.75 | 0.25 |

| Demand table | | |
|---|---|---|
| Ingress node | Egress node | Offered traffic (E) |
| 1 | 2 | 3 E |
| 3 | 4 | 1 E |
| 2 | 3 | 2 E |
| 3 | 1 | 2 E |
| 4 | 2 | 4 E |
| 1 | 4 | 4 E |

Figure 5: Graphical representation of a network structure

In order to use this information within Net2Plan, a structure called *netStruct* is created with the following fields:

- **netStruct.nodeXYPositionTable(**$N$**x2)**: $XY$ coordinates of nodes, where $N$ is the number of nodes, first column is $X$ coordinate, second one is $Y$

- **netStruct.nodeName{1x$N$}**: Cell containing $N$ strings with the name of each node
- **netStruct.nodeAttributes{1x$N$}**: Cell containing attributes of nodes. Each cell contains one struct and each value is stored as a string
- **netStruct.linkTable($E$x2)**: Each row represents a physical link, where first column is origin node $a(e)$, and second column is destination node $b(e)$
- **netStruct.linkCapacityInErlangs(1x$E$)**: Vector of link capacities
- **netStruct.linkLengthInKm(1x$E$)**: Vector of link lengths. In general it would be the euclidean distance in km between the nodes $a(e)$ y $b(e)$
- **netStruct.linkAttributes{1x$E$}**: Cell containing attributes of links. Each cell contains one struct and each value is stored as a string
- **netStruct.demandTable($D$x2)**: Each row represents a traffic demand, where first column is ingress node $a(d)$, and second column is egress node $b(d)$
- **netStruct.offeredTrafficInErlangs(1x$D$)**: Vector of offered traffic per demand $h_d$
- **netStruct.demandAttributes{1x$D$}**: Cell containing attributes of demands. Each cell contains one struct and each value is stored as a string
- **netStruct.pathList{1x$P$}**: Cell containing sequence of links
- **netStruct.routingMatrix($D$x$P$)**: Each entry $(d, p)$ represents the fraction of $h_d$ carried by the path $p$

As a convention, every user-defined attribute is assumed to be a string. Algorithms must check these attributes and convert them to the appropriate type, in case they use them.

According to the network depicted in Fig. 5, netStruct would be as follows:

```
netStruct.nodeXYPositionTable = [0 100; 100 100; 0 0; 100 0];
netStruct.nodeName = {'Node 1', 'Node 2', 'Loren Ipsum', 'John Doe'};
netStruct.nodeAttributes = cell(1,4);
netStruct.nodeAttributes{1} = struct();
netStruct.nodeAttributes{2} = struct();
netStruct.nodeAttributes{3} = struct();
netStruct.nodeAttributes{4} = struct();

netStruct.linkTable = [1 2; 1 4; 2 4; 3 1; 4 3];
netStruct.linkCapacityInErlangs = [10 3 5 10 10];
netStruct.linkLengthInKm = [100 141.42 100 100 100];
netStruct.linkAttributes = cell(1,5);
netStruct.linkAttributes{1} = struct();
netStruct.linkAttributes{2} = struct();
netStruct.linkAttributes{3} = struct();
netStruct.linkAttributes{4} = struct();
netStruct.linkAttributes{5} = struct();

netStruct.demandTable = [1 2; 3 4; 2 3; 3 1; 4 2; 1 4];
netStruct.offeredTrafficInErlangs = [3 1 2 2 4 4];
netStruct.demandAttributes = cell(1, 6);
netStruct.demandAttributes{1} = struct();
netStruct.demandAttributes{2} = struct();
netStruct.demandAttributes{3} = struct();
netStruct.demandAttributes{4} = struct();
netStruct.demandAttributes{5} = struct();
netStruct.demandAttributes{6} = struct();
netStruct.pathList = {[1] [4 1 3] [3 5] [4] [5 4 1] [2] [1 3]};
netStruct.routingMatrix = [ 1 0 0 0 0 0 0;
0 1 0 0 0 0 0;
0 0 1 0 0 0 0;
0 0 0 1 0 0 0;
0 0 0 0 1 0 0;
0 0 0 0 0 0.75 0.25];
```

Finally the mapping of the previous netStruct on .xml file is shown:

```
<?xml version="1.0" encoding="utf-8"?>
<network>
   <physicalTopology>
      <node id="1" name="Node 1" xCoord="0" yCoord="100"/>
      <node id="2" name="Node 2" xCoord="100" yCoord="100"/>
      <node id="3" name="Loren Ipsum" xCoord="0" yCoord="0"/>
      <node id="4" name="John Doe" xCoord="100" yCoord="0"/>
      <link destinationNodeId="2" id="1" linkCapacityInErlangs="10"
            linkLengthInKm="100" originNodeId="1"/>
      <link destinationNodeId="4" id="2" linkCapacityInErlangs="3"
            linkLengthInKm="141.42" originNodeId="1"/>
      <link destinationNodeId="4" id="3" linkCapacityInErlangs="5"
            linkLengthInKm="100" originNodeId="2"/>
      <link destinationNodeId="1" id="4" linkCapacityInErlangs="10"
            linkLengthInKm="100" originNodeId="3"/>
      <link destinationNodeId="3" id="5" linkCapacityInErlangs="10"
            linkLengthInKm="100" originNodeId="4"/>
   </physicalTopology>
   <demandSet>
      <demandEntry egressNodeId="2" id="1" ingressNodeId="1"
            offeredTrafficInErlangs="3"/>
      <demandEntry egressNodeId="4" id="2" ingressNodeId="3"
            offeredTrafficInErlangs="1"/>
      <demandEntry egressNodeId="3" id="3" ingressNodeId="2"
            offeredTrafficInErlangs="2"/>
      <demandEntry egressNodeId="1" id="4" ingressNodeId="3"
            offeredTrafficInErlangs="2"/>
      <demandEntry egressNodeId="2" id="5" ingressNodeId="4"
            offeredTrafficInErlangs="4"/>
      <demandEntry egressNodeId="4" id="6" ingressNodeId="1"
            offeredTrafficInErlangs="4"/>
   </demandSet>
   <routingInfo>
      <pathList>
         <path id="1">
            <linkEntry>1</linkEntry>
         </path>
         <path id="2">
            <linkEntry>4</linkEntry>
            <linkEntry>1</linkEntry>
            <linkEntry>3</linkEntry>
         </path>
         <path id="3">
            <linkEntry>3</linkEntry>
            <linkEntry>5</linkEntry>
         </path>
         <path id="4">
            <linkEntry>4</linkEntry>
         </path>
         <path id="5">
            <linkEntry>5</linkEntry>
            <linkEntry>4</linkEntry>
            <linkEntry>1</linkEntry>
         </path>
         <path id="6">
            <linkEntry>2</linkEntry>
         </path>
         <path id="7">
            <linkEntry>1</linkEntry>
            <linkEntry>3</linkEntry>
         </path>
      </pathList>
      <routing>
         <routingEntry demandId="1" id="1" pathId="1" value="1"/>
         <routingEntry demandId="2" id="2" pathId="2" value="1"/>
         <routingEntry demandId="3" id="3" pathId="3" value="1"/>
         <routingEntry demandId="4" id="4" pathId="4" value="1"/>
         <routingEntry demandId="5" id="5" pathId="5" value="1"/>
         <routingEntry demandId="6" id="6" pathId="6" value="0.75"/>
         <routingEntry demandId="6" id="7" pathId="7" value="0.25"/>
      </routing>
   </routingInfo>
</network>
```

Contents

## 5 Installation guide and system requirements

To install Net2Plan, save the compressed file in any directory. Then, extract all the files and folders into a new directory, for example, `C:\Work\Net2Plan`.

To run Net2Plan, start MATLAB, change current directory to the installation directory, and run `startup`.

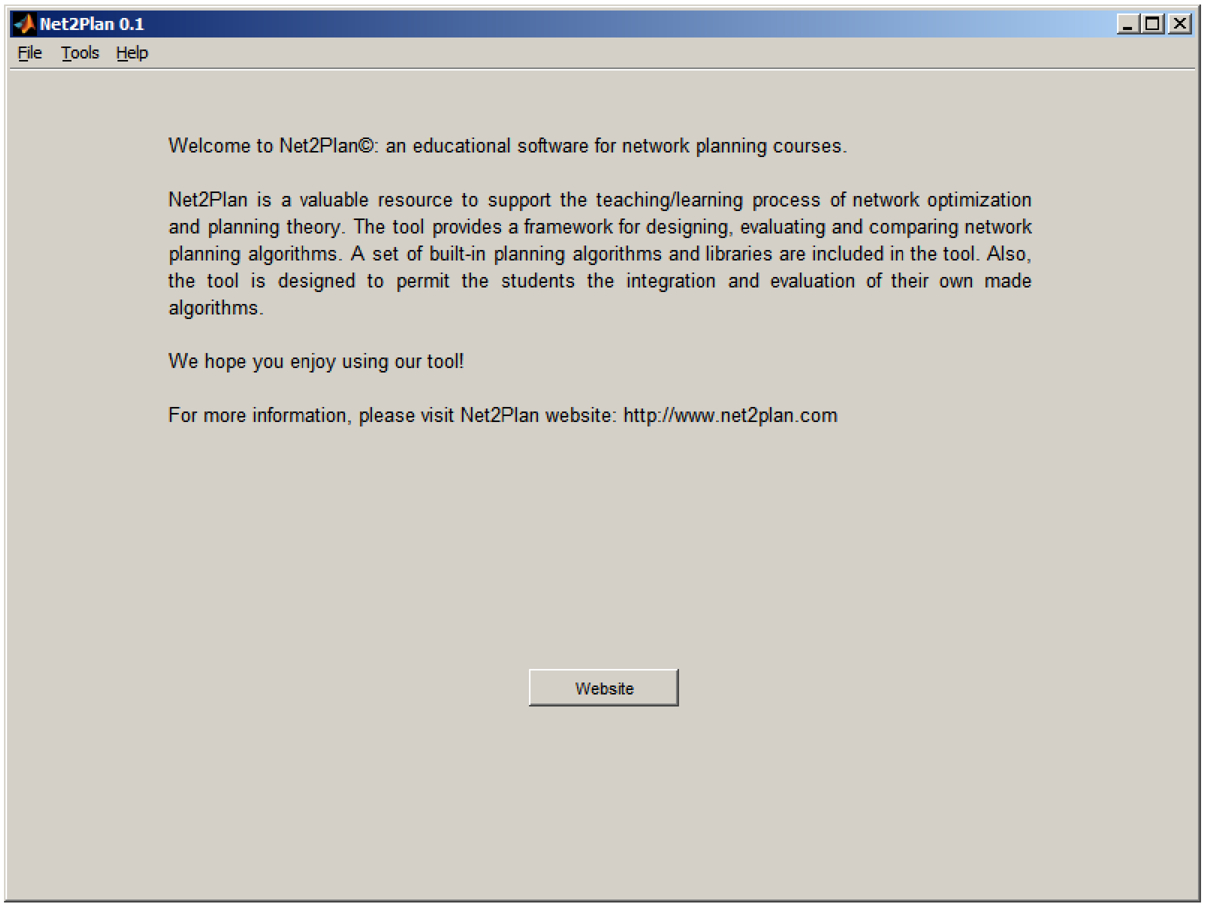
Net2Plan requires MATLAB 7.9.0 (R2009b) or higher versions and a screen resolution of, at least, 800x600 pixels.

Contents

## 6 Starting Net2Plan

In order to start the Net2Plan tool, execute MATLAB, change the current directory to \Net2Plan and execute `startup.m`. As a result, the welcome screen will be shown. If you want to use CVX solver you must initialize it using the appropriate CVX command, before executing Net2Plan.

In the top menu you can choose between the different options which Net2Plan provides. Below are listed and explained:



Welcome to Net2Plan©: an educational software for network planning courses.

Net2Plan is a valuable resource to support the teaching/learning process of network optimization and planning theory. The tool provides a framework for designing, evaluating and comparing network planning algorithms. A set of built-in planning algorithms and libraries are included in the tool. Also, the tool is designed to permit the students the integration and evaluation of their own made algorithms.

We hope you enjoy using our tool!

For more information, please visit Net2Plan website: http://www.net2plan.com

Figure 6: Welcome screen

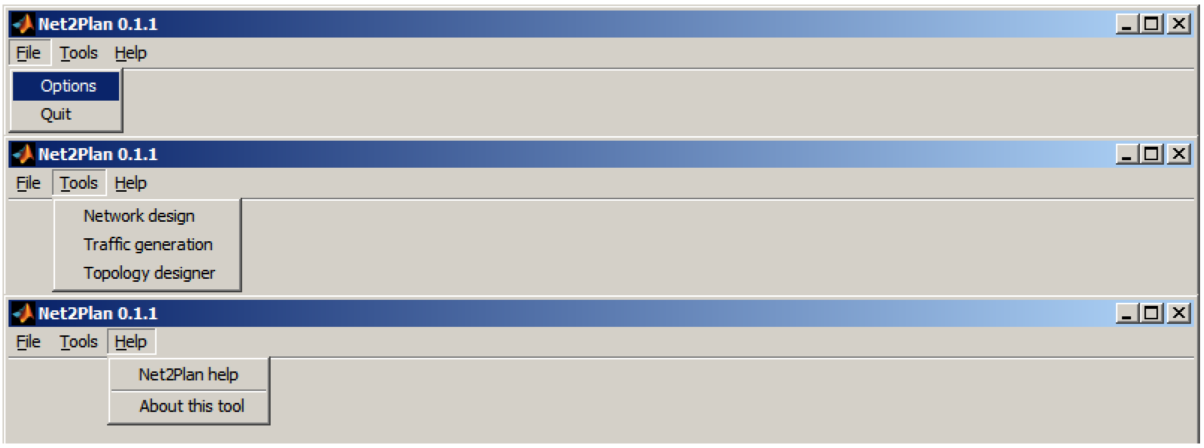


Figure 7: Main menu

- *File* menu: Allows to configure several options for Net2Plan
- *Tools* menu: Allows to access to the set of applications developed for Net2Plan: Network design, Traffic generation and Topology designer.
- *Help* menu: Allows to access to the help of Net2Plan and return to the welcome (about) screen

Contents

## 6.1 File menu

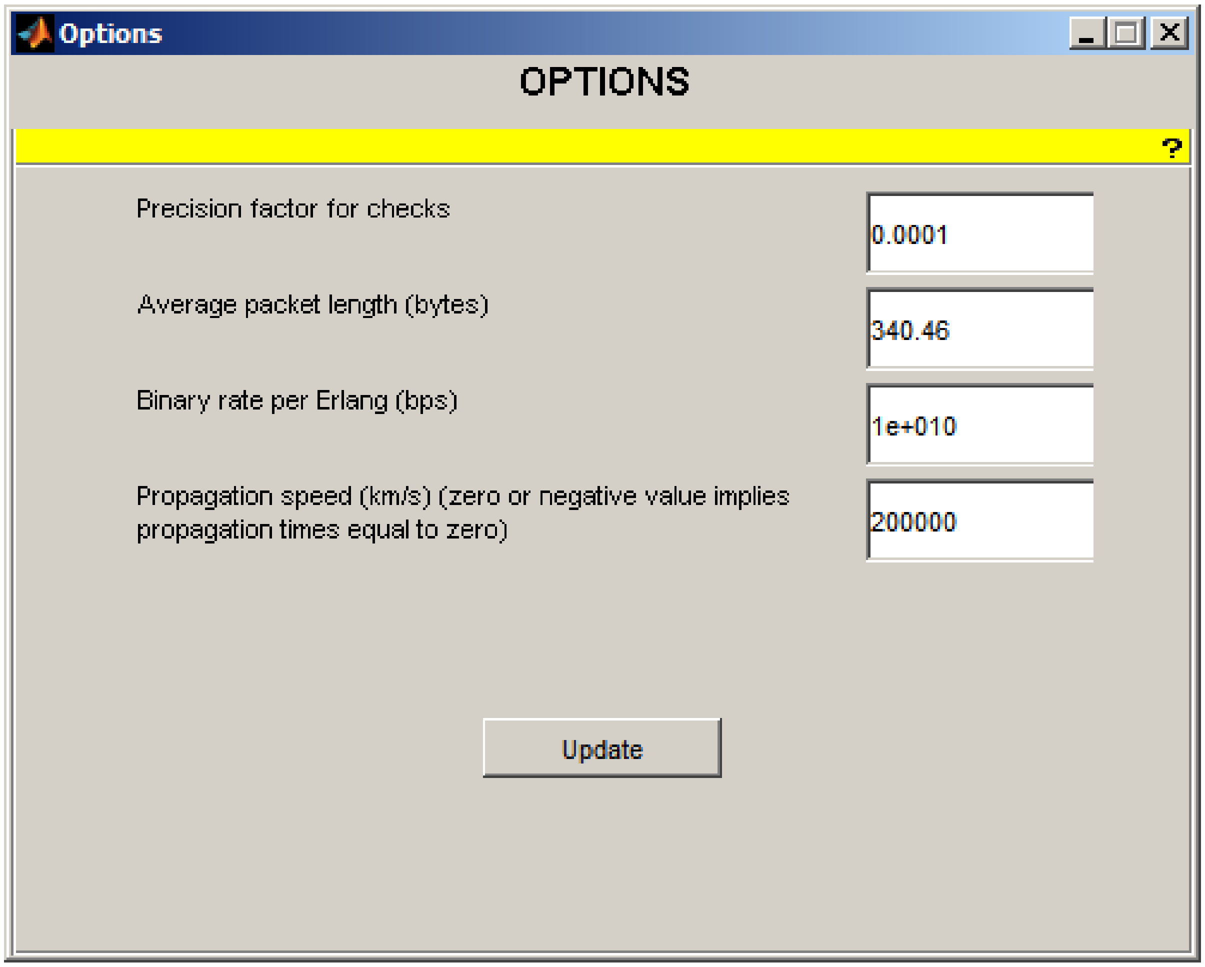
This menu has two items: Options and Quit.

Contents

### 6.1.1 Options

Use Options to set Net2Plan-wide parameters. These options have a global scope to all Net2Plan modules: are used within the kernel, and to compute delay metrics, for example. In this version there are only four options:

- *Precision factor for checks* (field `precisionFactor` in `options` structure): This parameter allows considering in the kernel small tolerances in the sanity-checks of the network designs. It avoids situations in which numerical inaccuracies would be interpreted as errors, i.e. if carried traffic by a link is greater than link capacity, kernel throws an error, but in some cases due to solvers precision might be a numerical error, for example $u_e = 10$ and $y_e = 10.000001$ in general would trigger an error, but with this precision factor if $u_e \leq y_e \leq u_e + PRECISIONFACTOR$ then $y_e = u_e$. Its value is constrained to be in range (0,1).
- *Average packet length (bytes)* (field `averagePacketLengthInBytes` in `options` structure): To get average packet delay metrics in the reports, it is needed to convert the traffic in the links from adimensional Erlang units to seconds. This parameter divided by the next one allows such conversion. Its value is constrained to be greater than zero.
- *Binary rate per Erlang (bps)* (field `binaryRateInBitsPerSecondPerErlang` in `options` structure): Binary rate equivalent to 1 Erlang. Its value is constrained to be greater than zero.



Precision factor for checks: 0.0001

Average packet length (bytes): 340.46

Binary rate per Erlang (bps): 1e+010

Propagation speed (km/s) (zero or negative value implies propagation times equal to zero): 200000

Figure 8: Options

- *Propagation speed (km/s)* (field `propagationSpeedInKmPerSecond` in `options` structure): Velocity of propagation in the media considered for the links. It allows to compute propagation times. A zero or negative value implies that propagation times are equal to zero.

Options are saved when you press "Update" button. Then new values are checked and saved in the options file.

Contents

### 6.1.2 Quit

Use Quit to close Net2Plan.

Contents

## 6.2 Tools menu

This menu has three items: Network design, Traffic generation and Topology designer.

Contents

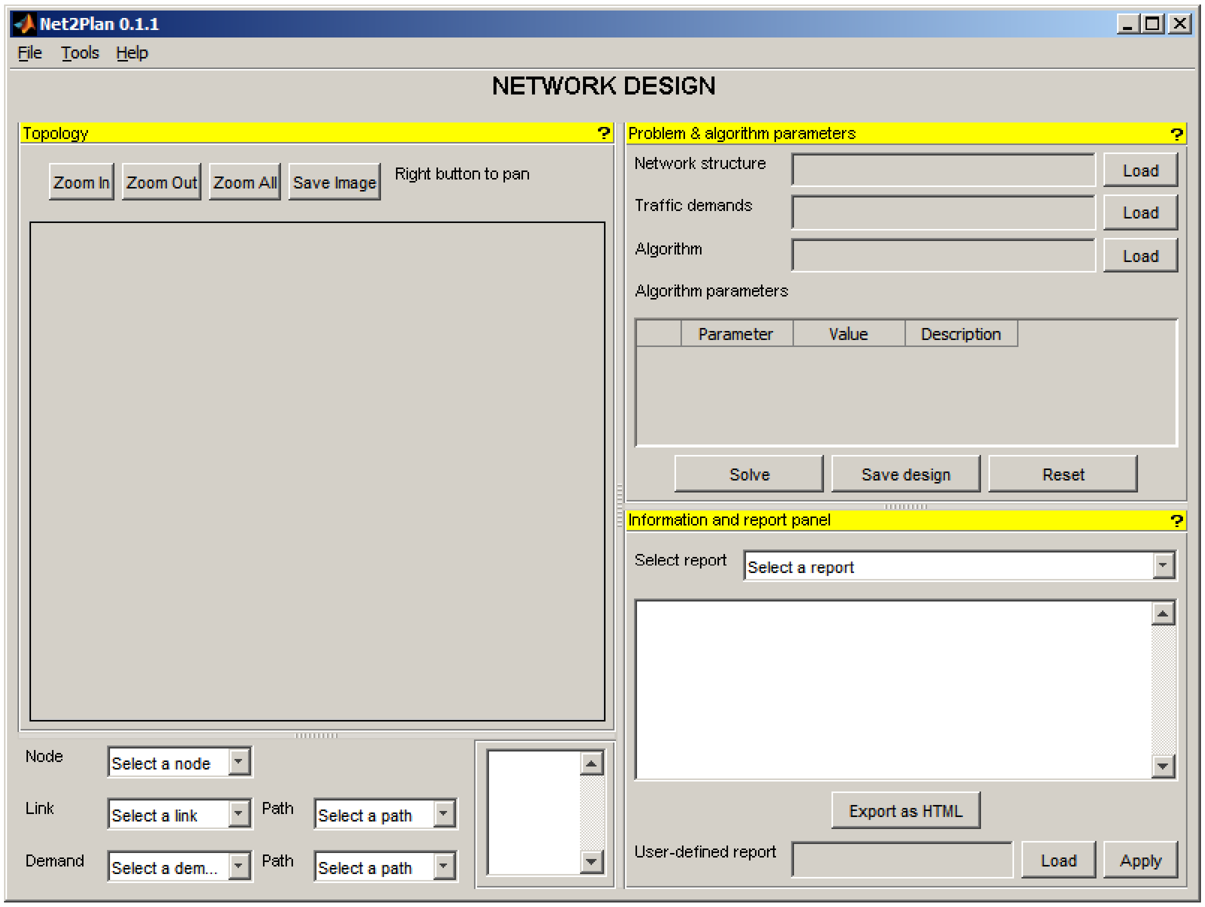


Figure 9: Network design window

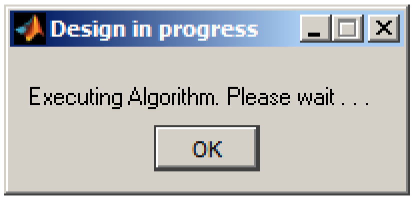
### 6.2.1 Network design

Selecting **Network design** opens the Network design window, which you can use to execute your algorithms.

The workspace of the window is divided into three areas: input data (top-right area), plot area (left), and results report area (bottom-right area).

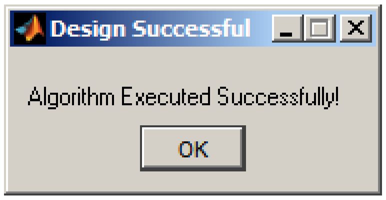
**Problem & algorithm parameters** The user should define the input parameters for the execution in the input area. In general, to calculate a new design the user has to specify the following things:

- *Network structure*: a .xml file with a previously computed and saved network design. The design can be partial: e.g. only the nodes of the network are present.
- *Traffic demands*: a .xml file describing a demand set. If the network structure already contains a demand set, it will show “Loaded from network structure”. If then you load a new .xml file, you will be asked to overwrite it and to remove routing information. Asking “No” you cancel that process.
- *Algorithm*: a MATLAB code file (*.m* extension), implementing the network design algorithm to be applied.
- *[Optional] Algorithm parameters*: if the algorithm selected requires specific parameters, they are shown in a table. Each parameter has a default value that the user can change.


Design in progress

Executing Algorithm. Please wait . . .

Figure 10: “Please wait” popup


Design Successful

Algorithm Executed Successfully!

Figure 11: “Execution successful” popup

Once those inputs are filled, the algorithm can be executed by clicking on “Solve”. A popup will be shown during the execution of the algorithms (see Fig. 10). When the execution finishes, a message will be shown, informing if the algorithm run well (see Fig. 11) or some error was thrown (see Fig. 12).

**Topology** In this panel you can see the current state of the network design. If the design includes routing information, it is possible to visualize the paths which carry traffic of a demand, the paths traversing a link,...

In addition, a small box shows brief information about the item selected in the topology (node, link, demand, path).

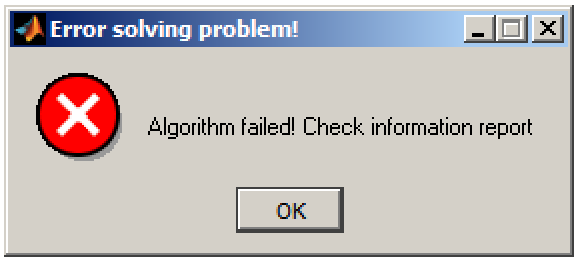
Here the user can “play” with the topology. The user can zoom in/out/all, save a .fig file of the view, or pan (move) axes holding right button on the mouse.

**Information and report panel** The algorithm solution found and its performance and cost merits are written into a set of reports in the text field at the Information and report panel. The user can choose between several types of reports showing topology metrics, demand set metrics... The solution can be saved in a XML file clicking the “Save design” button. A window is shown to choose where to save the file.

Moreover, users can define their own reports (see section 9) and attach it using the corresponding option.

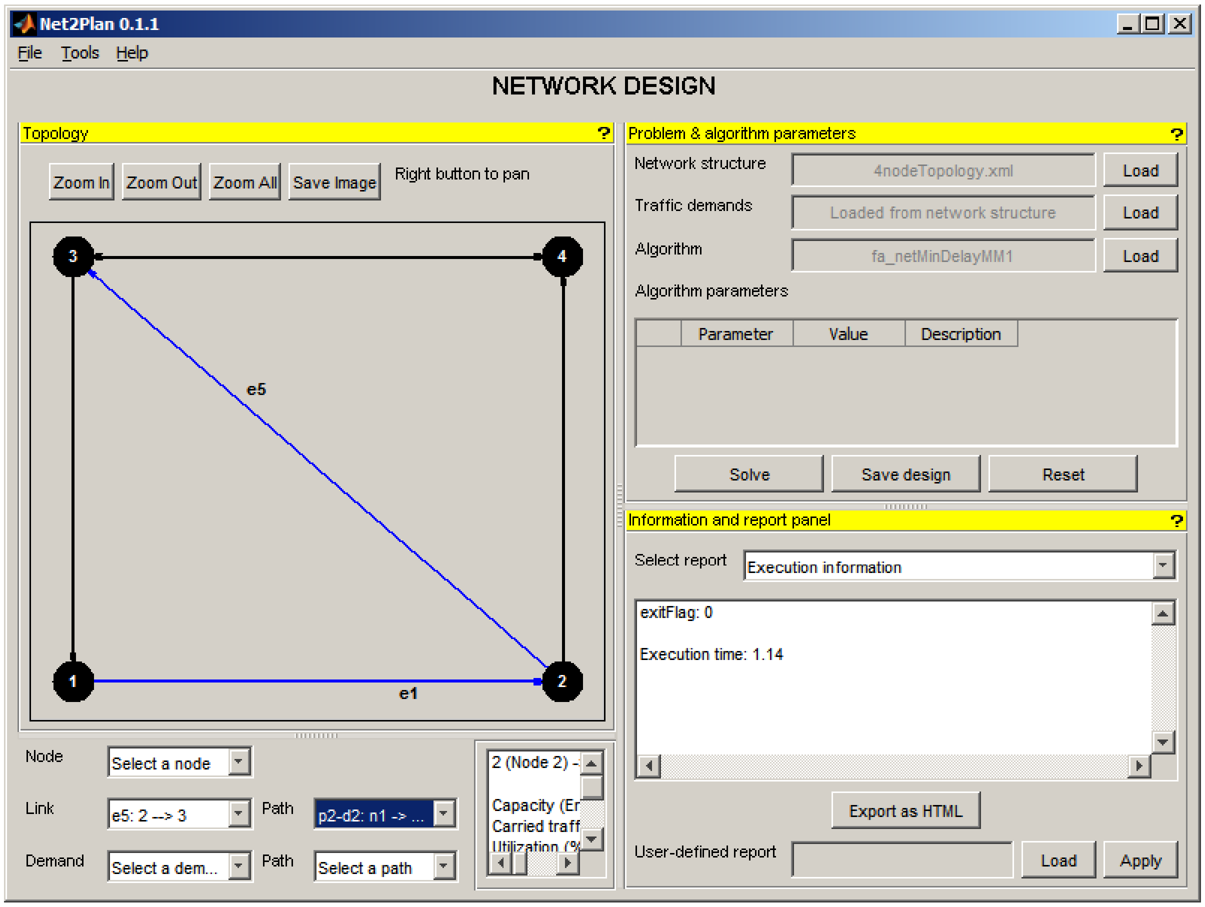
Finally, reports can be exported to a HTML file.

Contents


Error solving problem!

Algorithm failed! Check information report

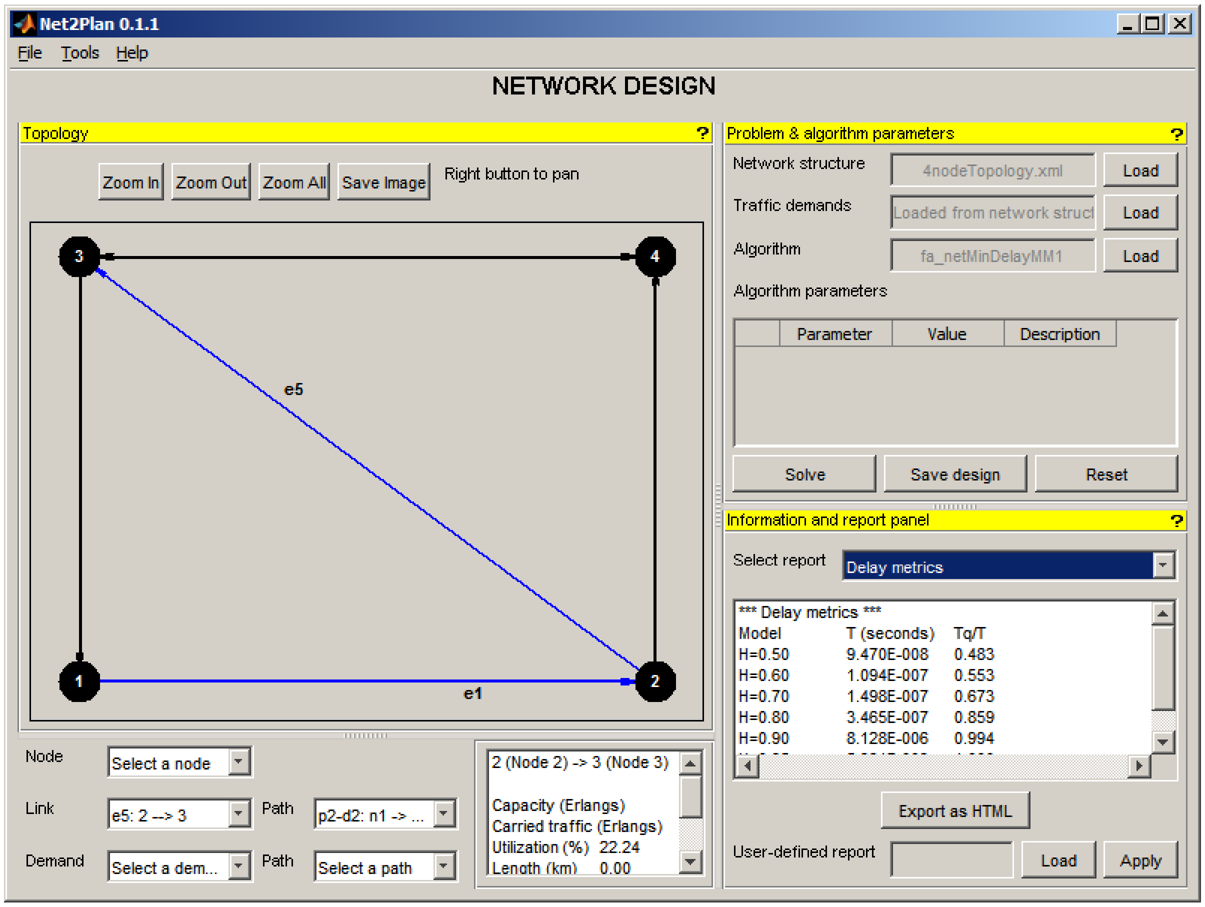
Figure 12: “Execution failed” popup



```
exitFlag: 0

Execution time: 1.14
```

Figure 13: Showing a path traversing a given link



Network structure: 4nodeTopology.xml
Traffic demands: Loaded from network struct
Algorithm: fa_netMinDelayMM1

Select report: Delay metrics

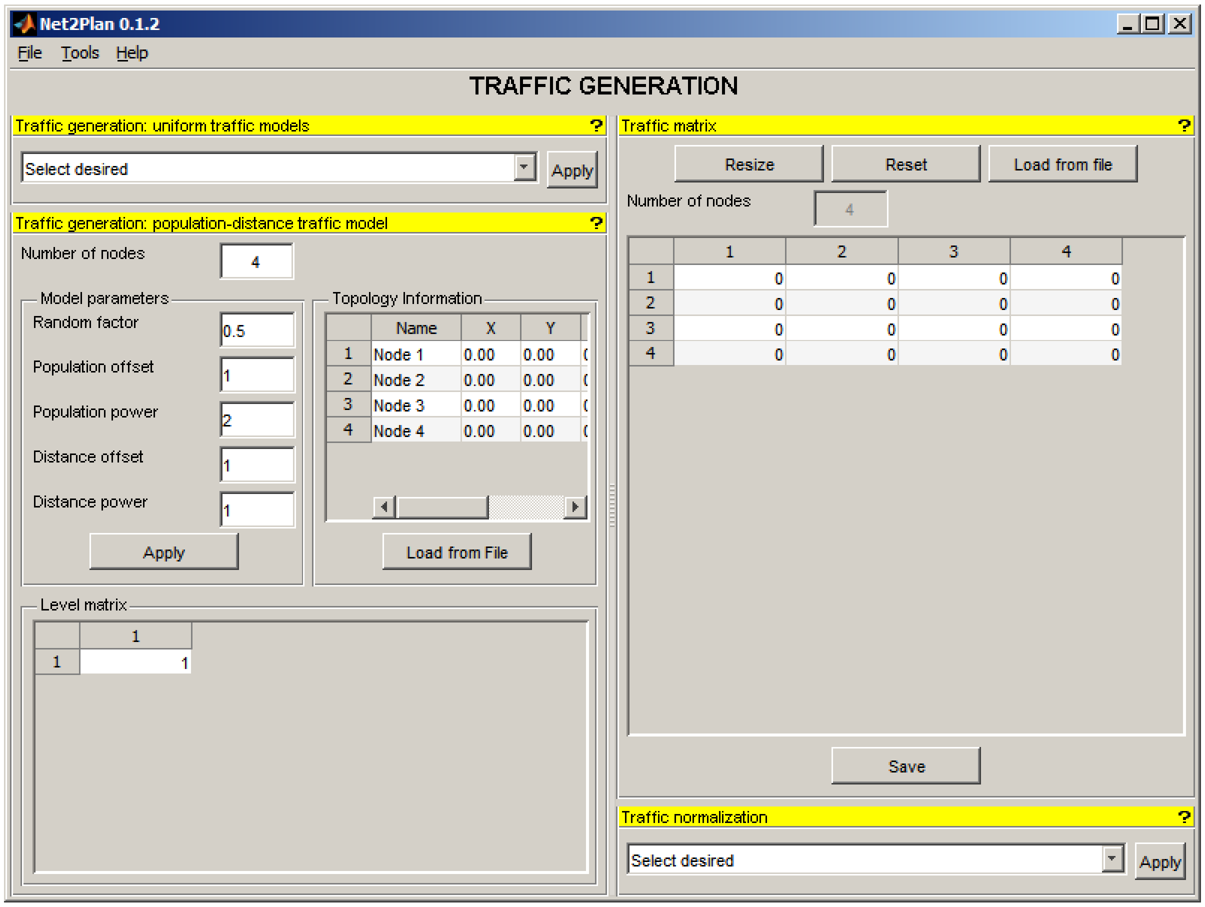
```
*** Delay metrics ***
Model       T (seconds)   Tq/T
H=0.50      9.470E-008    0.483
H=0.60      1.094E-007    0.553
H=0.70      1.498E-007    0.673
H=0.80      3.465E-007    0.859
H=0.90      8.128E-006    0.994
```

Node: Select a node
Link: e5: 2 --> 3    Path: p2-d2: n1 -> ...
Demand: Select a dem...    Path: Select a path

```
2 (Node 2) -> 3 (Node 3)

Capacity (Erlangs)
Carried traffic (Erlangs)
Utilization (%)  22.24
Length (km)      0.00
```

Figure 14: Showing delay metrics



TRAFFIC GENERATION

Traffic generation: uniform traffic models

Select desired

Traffic generation: population-distance traffic model

Number of nodes: 4

Model parameters

| | |
|---|---|
| Random factor | 0.5 |
| Population offset | 1 |
| Population power | 2 |
| Distance offset | 1 |
| Distance power | 1 |

Topology Information

| | Name | X | Y |
|---|---|---|---|
| 1 | Node 1 | 0.00 | 0.00 |
| 2 | Node 2 | 0.00 | 0.00 |
| 3 | Node 3 | 0.00 | 0.00 |
| 4 | Node 4 | 0.00 | 0.00 |

Level matrix

| | 1 |
|---|---|
| 1 | 1 |

Traffic matrix

Number of nodes: 4

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 |

Traffic normalization

Select desired

Figure 15: Traffic generation window

### 6.2.2 Traffic generation

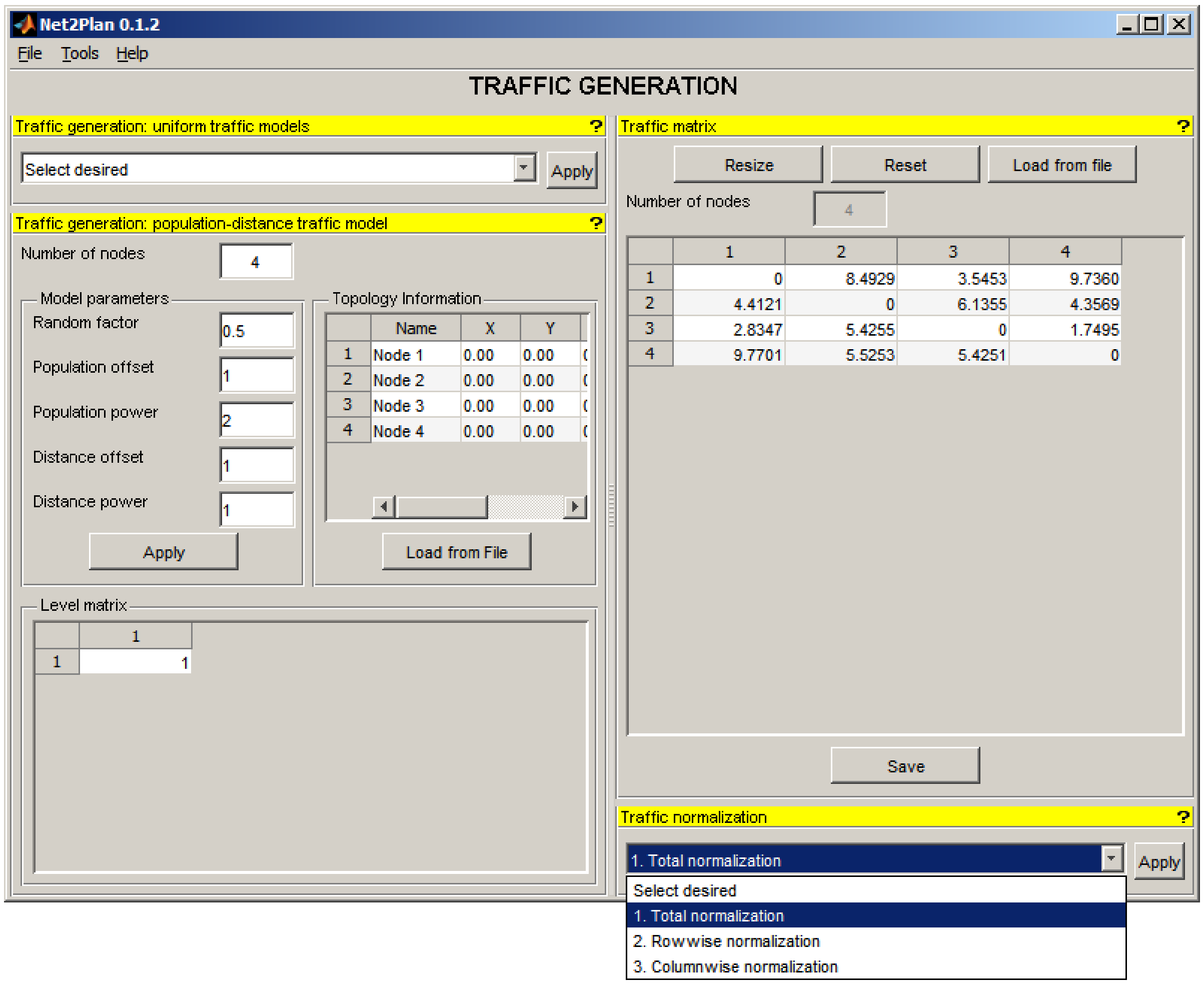
Selecting **Traffic generation** under **Tools** menu activates the Traffic generation window. This interface allows the user to generate a *.xml* file representing a traffic matrix. This figure displays the workspace window for this option.

The Traffic generation window is divided into four different parts:

- Traffic matrix
- Traffic normalization
- Traffic generation: uniform traffic models
- Traffic generation: population-distance traffic model

**Traffic matrix** The user initiates the process by selecting the number of nodes $N$ in the network using **Resize** option. The traffic matrix will be like this:

$$\begin{pmatrix} \gamma_{11} & \gamma_{12} & \cdots & \gamma_{1i} \\ \gamma_{21} & \gamma_{22} & & \vdots \\ \vdots & & \ddots & \\ \gamma_{i1} & \cdots & & \gamma_{ij} \end{pmatrix} \tag{1}$$



Number of nodes: 4
Random factor: 0.5
Population offset: 1
Population power: 2
Distance offset: 1
Distance power: 1

| | Name | X | Y |
|---|---|---|---|
| 1 | Node 1 | 0.00 | 0.00 |
| 2 | Node 2 | 0.00 | 0.00 |
| 3 | Node 3 | 0.00 | 0.00 |
| 4 | Node 4 | 0.00 | 0.00 |

| | 1 |
|---|---|
| 1 | 1 |

Number of nodes: 4

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 8.4929 | 3.5453 | 9.7360 |
| 2 | 4.4121 | 0 | 6.1355 | 4.3569 |
| 3 | 2.8347 | 5.4255 | 0 | 1.7495 |
| 4 | 9.7701 | 5.5253 | 5.4251 | 0 |

1. Total normalization

Figure 16: Traffic normalization

where $i$ and $j$ are the node identifiers. Self-demands are not allowed ($\gamma_{ii} = 0$).

Next, the user have to options: to introduce values manually, or to use traffic generation patterns.

Also you can open an existing *.xml* file to edit it, or reset to all-zero matrix.

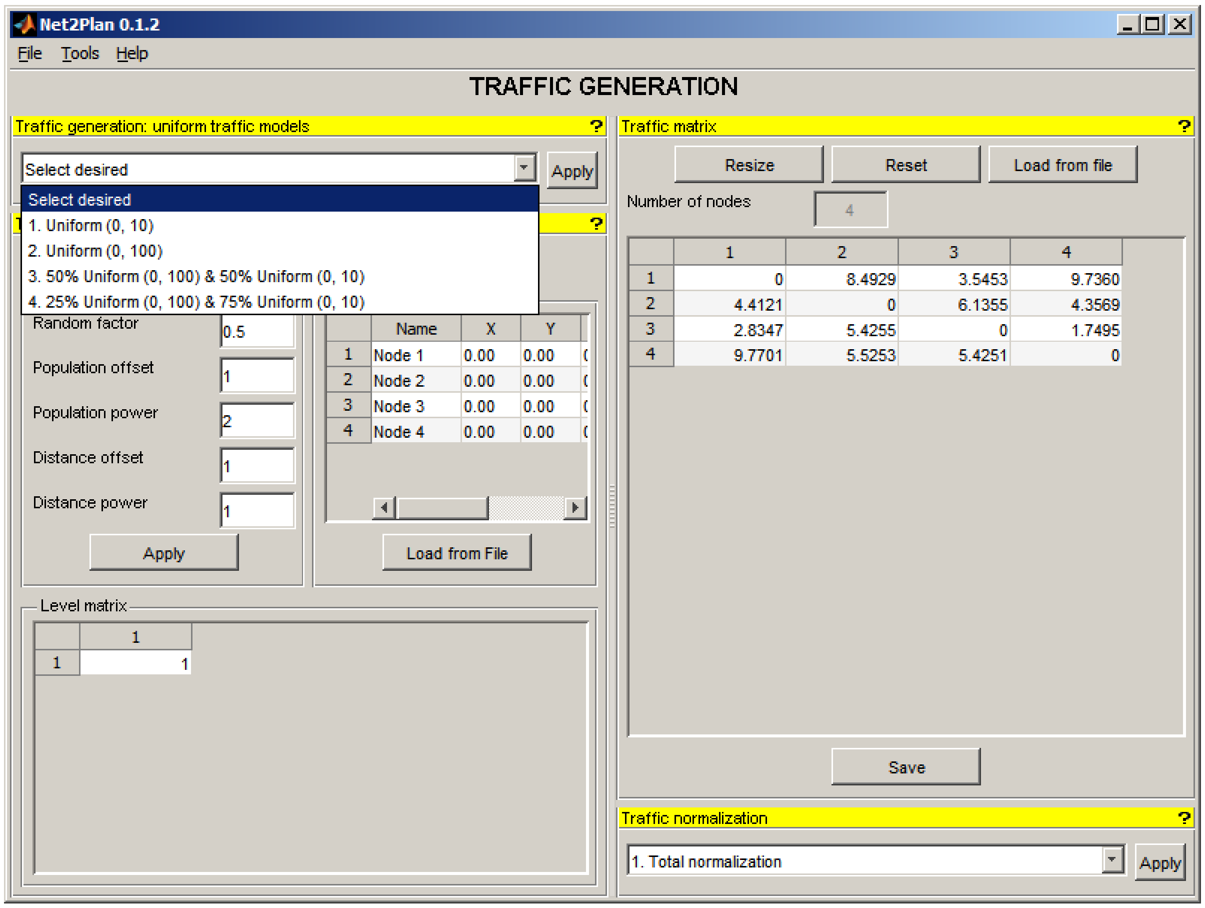
Finally, the user can save it into the file system with the Save button.

**Traffic normalization** Now, the user can apply an automatic normalization of the matrix. Three types of normalization methods are implemented: total, row and column normalization [2]. The first adjusts the traffic matrix so that the total traffic offered to the network matches a user defined value. The second (third) modifies the matrix so that the $i$-th row (column) of the matrix sums $x_i$, being $x_i$, $i = 1 \dots N$ a vector defined by the user. Note that this means that the total traffic transmitted (received) by node $i$ is fixed to exactly $x_i$.

- Total normalization: the sum of matrix elements have to be the value of total offered traffic, by following the next expression:

$$M'_T(i,j) = M_T(i,j) \cdot \frac{\text{Total Offered Traffic}}{\sum_{ij} M_T(i,j)} \tag{2}$$

  where $M'_T$ is the normalized traffic matrix and $M_T$ is the original traffic matrix.



TRAFFIC GENERATION

Traffic generation: uniform traffic models

Select desired [Apply]

- Select desired
- 1. Uniform (0, 10)
- 2. Uniform (0, 100)
- 3. 50% Uniform (0, 100) & 50% Uniform (0, 10)
- 4. 25% Uniform (0, 100) & 75% Uniform (0, 10)

| | |
|---|---|
| Random factor | 0.5 |
| Population offset | 1 |
| Population power | 2 |
| Distance offset | 1 |
| Distance power | 1 |

[Apply]

| | Name | X | Y |
|---|---|---|---|
| 1 | Node 1 | 0.00 | 0.00 |
| 2 | Node 2 | 0.00 | 0.00 |
| 3 | Node 3 | 0.00 | 0.00 |
| 4 | Node 4 | 0.00 | 0.00 |

[Load from File]

Level matrix

| | 1 |
|---|---|
| 1 | 1 |

Traffic matrix

[Resize] [Reset] [Load from file]

Number of nodes 4

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 8.4929 | 3.5453 | 9.7360 |
| 2 | 4.4121 | 0 | 6.1355 | 4.3569 |
| 3 | 2.8347 | 5.4255 | 0 | 1.7495 |
| 4 | 9.7701 | 5.5253 | 5.4251 | 0 |

[Save]

Traffic normalization

1. Total normalization [Apply]

Figure 17: Uniform traffic models

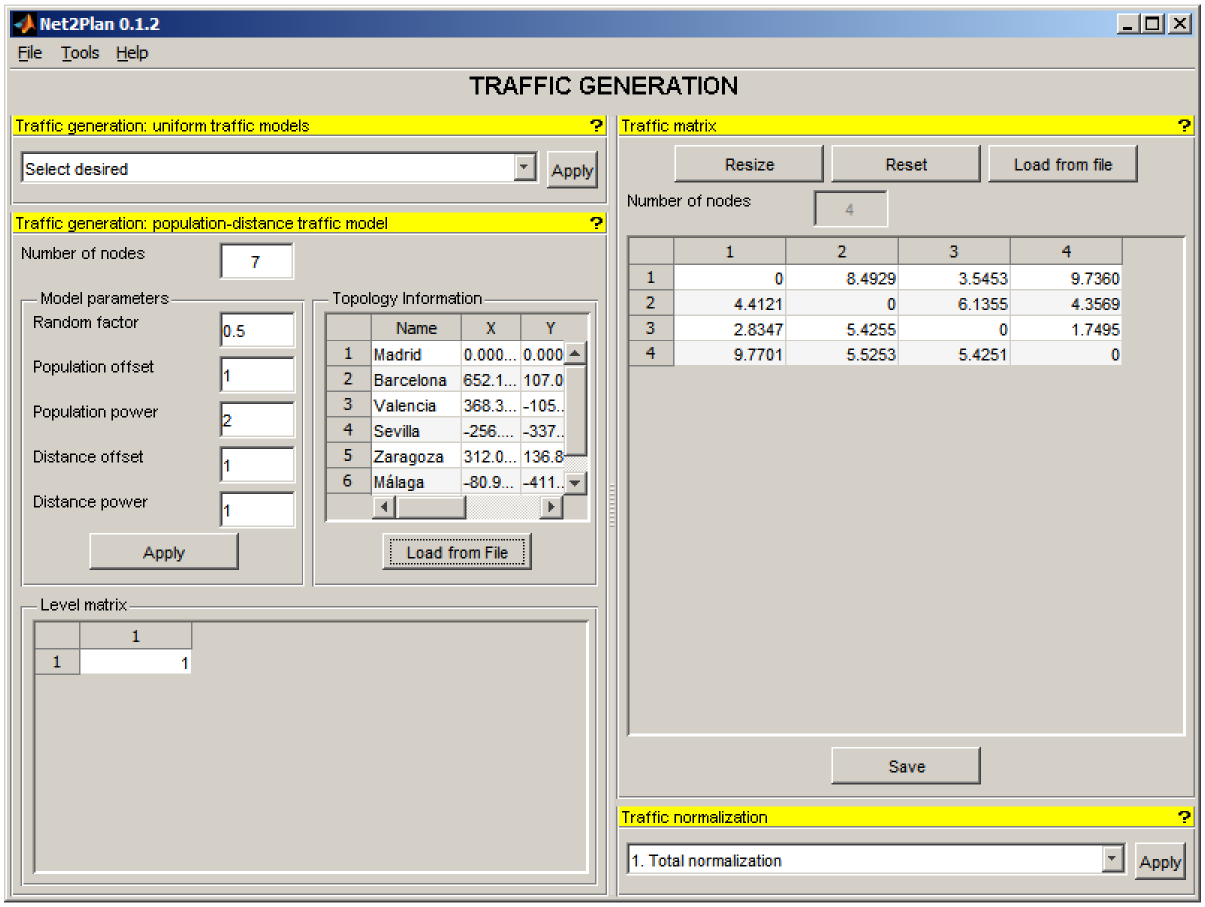
The user introduces the total offered traffic volume in a popup.

- Row normalization: the sum of each traffic matrix row have to be the value that the user introduces in a popup.
- Column normalization: the sum of each traffic matrix column have to be the value that the user introduces in a popup.

**Traffic generation: uniform traffic models** After this, the user can select one of the traffic generation patterns among the five modes available: four of them are simpler, and based on the uniform distribution.

- Uniform (0, 10): Generates an $NxN$ traffic matrix with random values in the range (0,10)
- Uniform (0, 100): Generates an $NxN$ traffic matrix with random values in the range (0,100)
- 50% Uniform (0, 100) & 50% Uniform (0, 10): Generates an $NxN$ traffic matrix with 50% of its entries with random values in the range (0,100), and the rest of the entries with random values in the range (0,10)
- 25% Uniform (0, 100) & 75% Uniform (0, 10): Generates an $NxN$ traffic matrix with 25% of the entries with random values in the range (0,100) and the rest of the entries with random values in the range (0,10)

Please note at the end of this process diagonal values of traffic matrix are zero, since self-demands are not allowed.



Traffic generation: uniform traffic models

Select desired

Traffic generation: population-distance traffic model

Number of nodes: 7

Model parameters

Random factor: 0.5

Population offset: 1

Population power: 2

Distance offset: 1

Distance power: 1

Topology Information

| | Name | X | Y |
|---|---|---|---|
| 1 | Madrid | 0.000... | 0.000 |
| 2 | Barcelona | 652.1... | 107.0 |
| 3 | Valencia | 368.3... | -105.. |
| 4 | Sevilla | -256.... | -337.. |
| 5 | Zaragoza | 312.0... | 136.8 |
| 6 | Málaga | -80.9... | -411.. |

Level matrix

| | 1 |
|---|---|
| 1 | 1 |

Traffic matrix

Number of nodes: 4

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 8.4929 | 3.5453 | 9.7360 |
| 2 | 4.4121 | 0 | 6.1355 | 4.3569 |
| 3 | 2.8347 | 5.4255 | 0 | 1.7495 |
| 4 | 9.7701 | 5.5253 | 5.4251 | 0 |

Traffic normalization

1. Total normalization

Figure 18: Population-distance traffic model

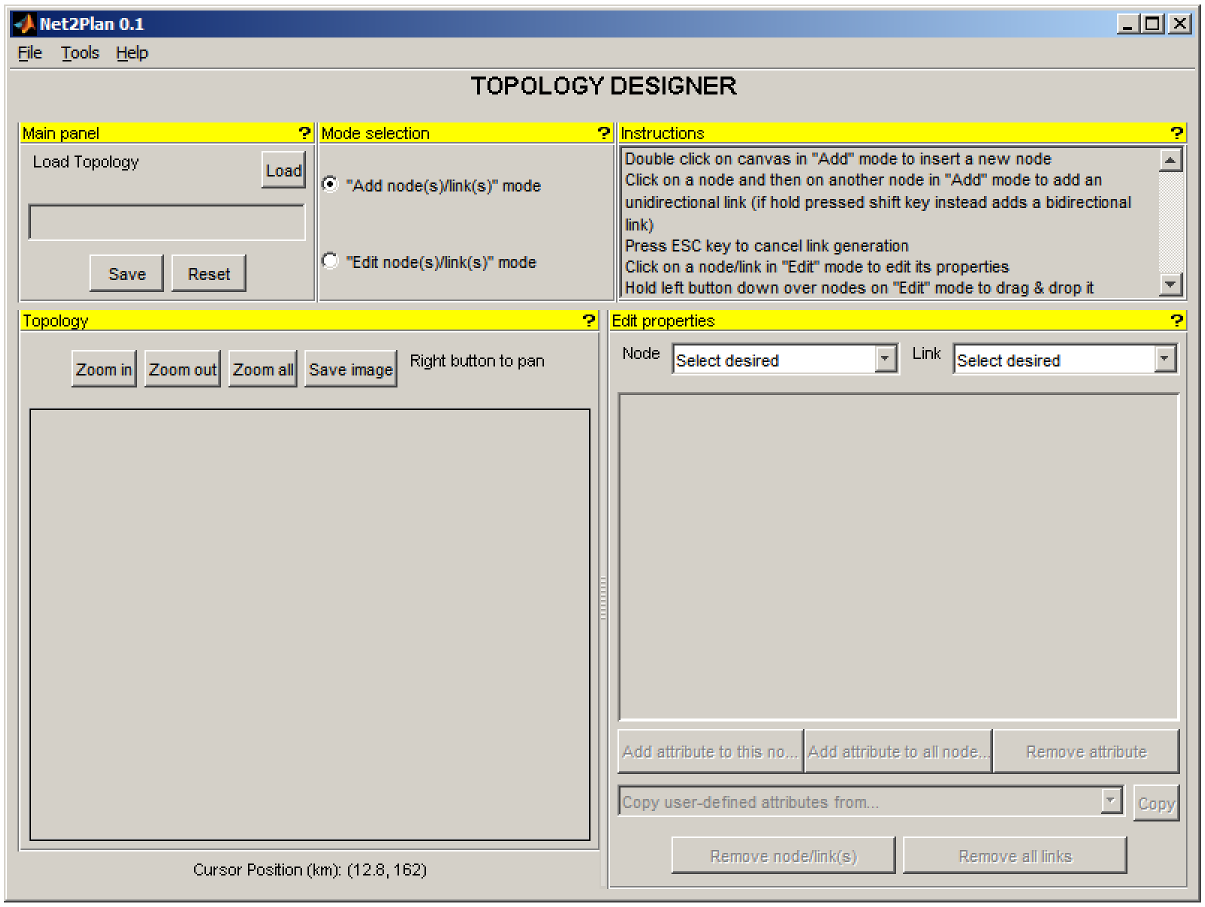
**Traffic generation: population-distance traffic model** The fifth mode allows creating a traffic matrix according to the model described in [2]. This latter model applies the information of node position, population and node level, present in a topology information table (user can load a topology from a *.xml* file). The distribution based on populations and distances follows the next expression:

$$\gamma_{ij} = \frac{Level(L_i, L_j) \cdot (1 - rf + 2 \cdot rf \cdot rand()) \cdot \left( \dfrac{Pop_i \cdot Pop_j}{Pop_{max}^2} + Pop_{off} \right)^{Pop_{Power}}}{\left( \dfrac{dist(i,j)}{dist_{max}} + dist_{off} \right)^{Dist_{Power}}} \tag{3a}$$

$$0 \le rf \le 1 \begin{cases} rf = 0 \Rightarrow (1 - rf + 2 \cdot rf \cdot rand()) = 1 \Rightarrow \text{without random component} \\ rf = 1 \Rightarrow (1 - rf + 2 \cdot rf \cdot rand()) = 2 \cdot rand() = 0 \dots 2 \end{cases} \tag{3b}$$

where *Level* is a $L \times L$ two-dimensional matrix ($L$: number of levels or node types defined by the user). This matrix allows us to introduce asymmetric traffic in the traffic generator; $Pop_i$ is the population of the node $i$; $Pop_{max}$ is the maximum population; dist is a matrix $N \times N$ ($N$: number of nodes) with the distances between nodes; $dist_{max}$ is the maximum distance.

Contents



Double click on canvas in "Add" mode to insert a new node
Click on a node and then on another node in "Add" mode to add an unidirectional link (if hold pressed shift key instead adds a bidirectional link)
Press ESC key to cancel link generation
Click on a node/link in "Edit" mode to edit its properties
Hold left button down over nodes on "Edit" mode to drag & drop it

Figure 19: Topology designer window

### 6.2.3 Topology designer

Selecting Topology designer under Tools menu activates the Topology designer window. This interface allows the user to generate a .xml file representing a network topology. This figure displays the workspace window for this option.

**Main panel** Here you can load and save network designs.

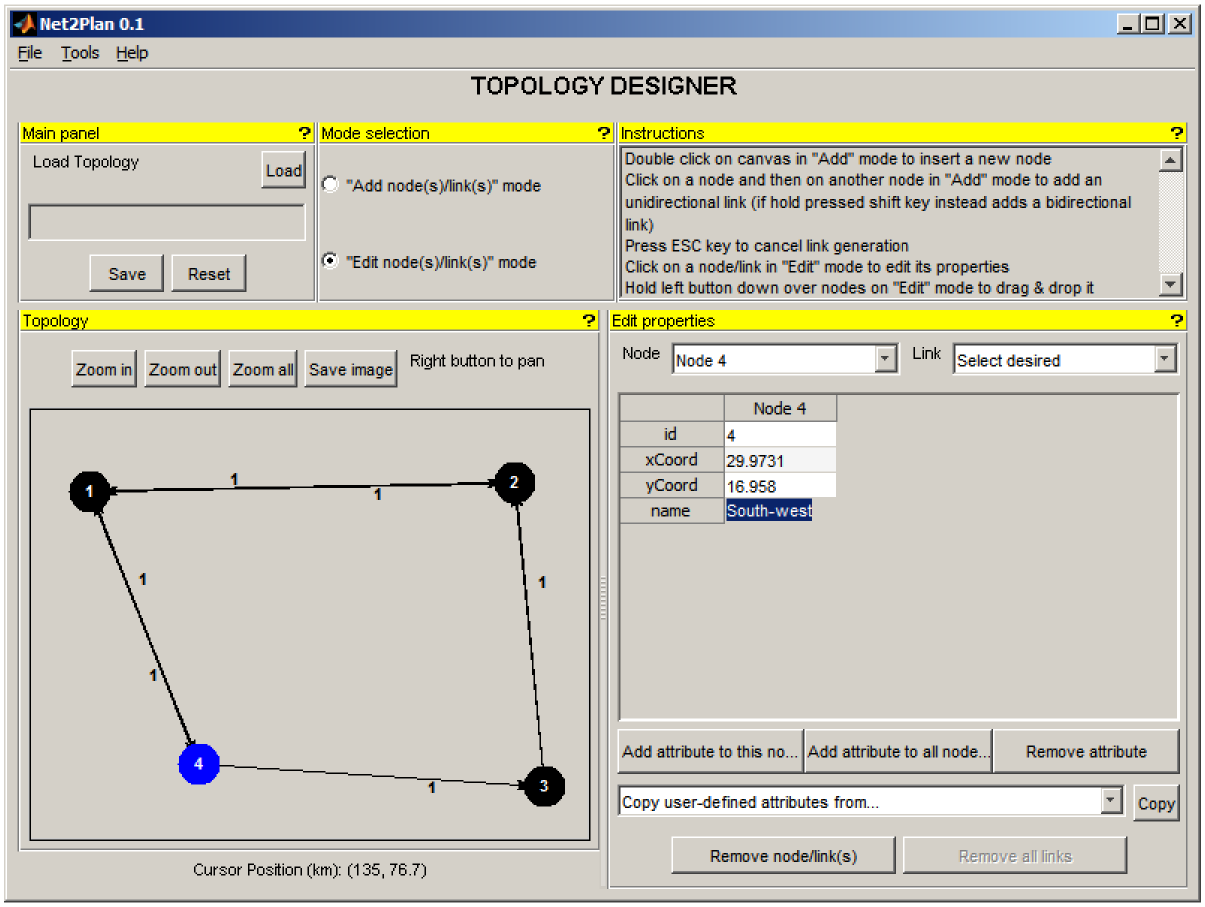
**Mode selection** Here you can change the mode of the tool. In “Add mode” you can add nodes and links, and in “Edit” mode you can edit them.

**Instructions** Here instructions to design topologies are shown. In sections below are reproduced.

**Topology** This panel is used for graphically design networks. It is very similar to the topology panel in the network design tool, but it has certain new functionalities.

Nodes are inserted by double clicking into the canvas with the “Add” mode activated.

Links are inserted by clicking first in the origin node and then in the destination node. It is possible to insert unidirectional links or bidirectional ones (in this case, you must press SHIFT key during that process).



"Add node(s)/link(s)" mode [✓ "Edit node(s)/link(s)" mode]

Double click on canvas in "Add" mode to insert a new node
Click on a node and then on another node in "Add" mode to add an unidirectional link (if hold pressed shift key instead adds a bidirectional link)
Press ESC key to cancel link generation
Click on a node/link in "Edit" mode to edit its properties
Hold left button down over nodes on "Edit" mode to drag & drop it

Node: Node 4

| | Node 4 |
|---|---|
| id | 4 |
| xCoord | 29.9731 |
| yCoord | 16.958 |
| name | South-west |

Cursor Position (km): (135, 76.7)

Figure 20: Change a node name

**Edit properties** When you have nodes and links placed, you can edit their properties by clicking on them with the "Edit" mode activated. Not all properties can be changed, i.e. "Node Identifier" is generated internally. The following figures show two examples.

You can define new attributes for nodes and links using the corresponding buttons, and also it is possible to copy properties from nodes or links to others. But as for editing, you cannot copy certain properties, such as "Node / Link id", "Origin node", "Destination node", ... When copy links, if there are bidirectional links, you are asked for select the links in which you can paste values.

Finally, you can delete nodes or links by means of "Remove ..." buttons. Moreover, when you delete a node, associated links are deleted as well.

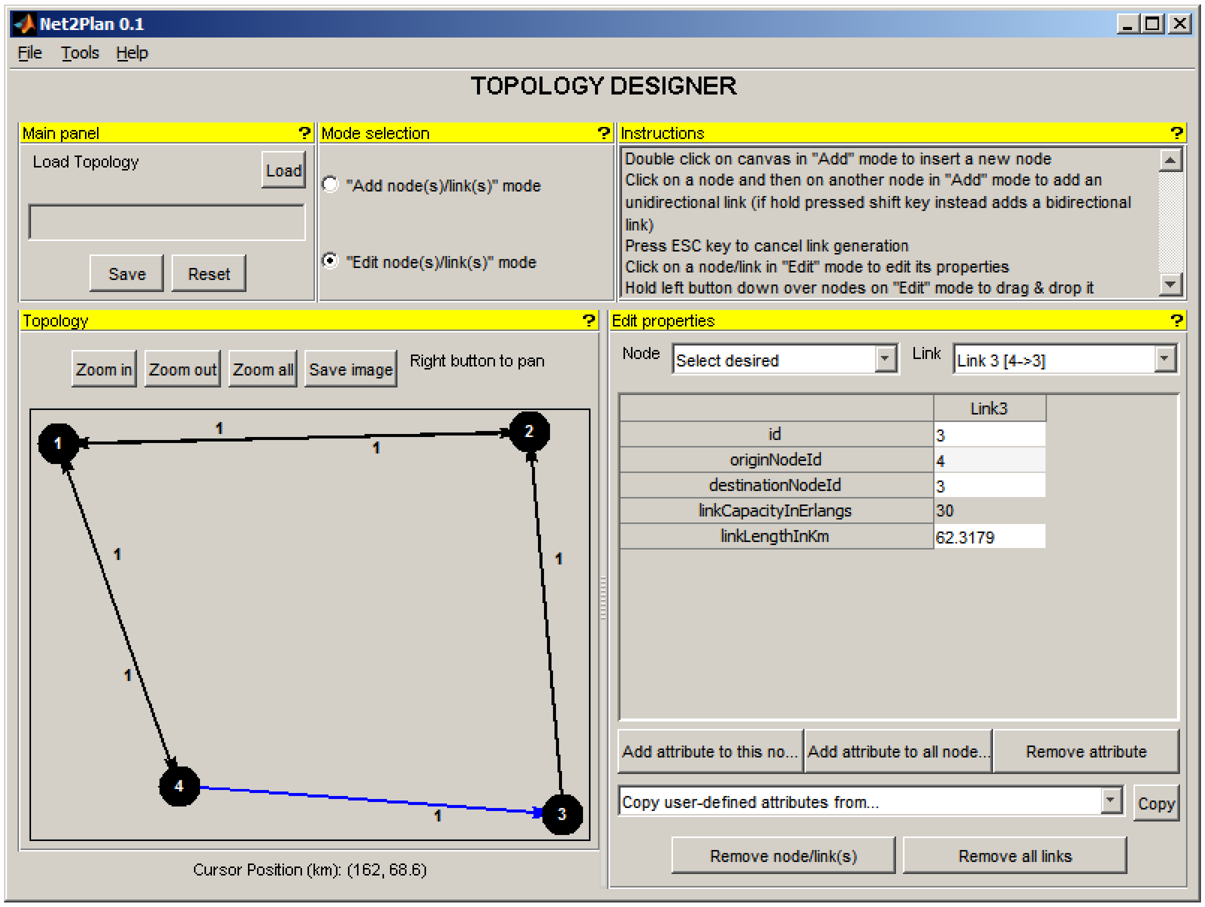
## 6.3 Help menu

This menu has two items: Net2Plan help and About this tool.

### 6.3.1 Net2Plan help

You can read this file.

Contents



"Add node(s)/link(s)" mode
[✓ "Edit node(s)/link(s)" mode]

Double click on canvas in "Add" mode to insert a new node
Click on a node and then on another node in "Add" mode to add an unidirectional link (if hold pressed shift key instead adds a bidirectional link)
Press ESC key to cancel link generation
Click on a node/link in "Edit" mode to edit its properties
Hold left button down over nodes on "Edit" mode to drag & drop it

Node: Select desired
Link: Link 3 [4->3]

| | Link3 |
|---|---|
| id | 3 |
| originNodeId | 4 |
| destinationNodeId | 3 |
| linkCapacityInErlangs | 30 |
| linkLengthInKm | 62.3179 |

Copy user-defined attributes from...

Cursor Position (km): (162, 68.6)

Figure 21: View properties of a link

### 6.3.2 About this tool

It shows the welcome screen.

Contents

# 7 Toolbox Directory Structure

This section describes the directory structure in the toolbox. It has the following folders:

| Directory | Description |
|---|---|
| \algorithms | Includes built-in algorithms. User-made algorithms must be placed here |
| \data | Includes topologies, traffic matrices and result files. User-made designs can be placed here |
| \help | Includes help file |
| \kernel | Includes kernel of the tool. It is recommended not to modify any file here |
| \libraries | Includes auxiliary libraries to develop algorithms |
| \userDefinedReports | Includes user-defined reports. User-made reports must be placed here |

Table 2: Toolbox Directory Structure

Contents

# 8 Integrating new algorithms

New algorithms can be implemented as MATLAB functions (.m files) with a given signature. Integration of algorithms simply requires saving them in any directory of the computer, although it is a good practice to store them in \`algorithms` directory.

The signature must be like this:

**[exitFlag, exitMsg, netStruct] = algorithmName (netStruct, algorithmParameters, net2planParameters)**

The input parameters are defined as follows:

- *netStruct*: Network structure according to section 4.2 containing nothing/part/all of this information: topology, demand set... Each algorithm should check that the network structure received contains the network information it needs as input parameter.
- *algorithmParameters*: Structure containing algorithm-specific parameters. All parameters are passed to the algorithm as strings: the algorithm is in charge of performing the adequate conversions itself.
- *net2planParameters*: Structure containing Net2Plan-wide configuration options according to section 6.1.1. As stated in that section, these options have a global scope to all Net2Plan modules, for example they are used to compute delay metrics.

The output parameters are:

- *exitFlag*: If equal to zero, the kernel assumes that the algorithm was successful. Otherwise, the kernel considers that an error happened.

- *exitMsg*: Exit message. Only shown if exitFlag≠0.
- *netStruct*: The updated network design. Same format as the input parameter *netStruct*.

The task of generating new network design algorithms in Net2Plan, is facilitated by a set of built-in libraries included (see section 11).

## 8.1 Defining algorithm parameters

When creating a new algorithm, the developer can specify a set of input parameters to tune the algorithm functionality. The kernel recognizes that the algorithm has input parameters

To declare the input paramters of an algorithm to the kernel, the .m implementnig the algorithm must include in the help text of the file, one line per parameter in the format:

```
PARAM: parameterName | parameterDefaultValue | parameterDescription
```

When the Net2Plan user selects an algorithm, the kernel processes the help string of the .m file, and obtains the input parameters to the algorithm, their default values, and a description string that is displayed in the GUI to inform the user. When the user clicks the solve button, the kernel passes to the algorithm the algorithmParameters structure. In this structure, each parameter is passed to the algorithm as a char. The algorithm implementation should perform the appropriate conversions to the specific type, making any sanity checks if needed. For example, the next snippet shows how to define a new algorithm with two parameters and how to convert the input parameters to a numerical format.

```
function [exitFlag, exitMsg, netStruct] = CA_myAlgorithm
                        (netStruct, algorithmParameters, net2planParameters)

    % This algorithm ...
    %
    % PARAM: rho | 0.9 | Network congestion
    % PARAM: u_max | 5 | Maximum capacity allowed

    % Get parameters
    rho = str2double(algorithmParameters.rho);
    u_max = str2double(algorithmParameters.u_max);

    % Sanity checks
    assert(rho <= 1 && rho >= 0, '"rho" must be between [0,1]');

    ...

end
```

Contents

# 9 User-defined report generation

In addition to reports generated by Net2Plan user can define reports which returns every kind of information that user requires.

As stated for algorithms, user-defined reports can be implemented as MATLAB functions with a given signature. Integration of user-defined reports simply requires saving them in any directory of the computer, although it is a good practice to store them in `\userDefinedReports` directory.

The signature must be like this:

**msg = functionName(netStruct, net2planParameters)**

Input parameters are structures defined according to sections 4.2 and 6.1.1, and the output parameter must be a string. User is responsible to validity of these reports and to check netStruct prior to their computation.

Next, an example of user-defined report is shown:

```
function msg = userText(netStruct, net2planParameters)

    msg = 'It works!';

end
```

Contents

## 10 Built-in algorithms

In table 3 we enumerate all the algorithms included in Net2Plan distribution. These algorithms are organized in folders according to the problem type they address.

Contents

## 11 Built-in libraries

In table 4 are shown all libraries included within Net2Plan. These libraries are organized in folders according to the function performed.

Contents

## 12 Authors

Net2Plan tool has its origins in 2011, during the preparation of the teaching materials for two new Communication Networks Planning courses at Universidad Politécnica de Cartagena (Spain) taught by Prof. Pablo Pavón Mariño:

- “Telecommunication networks theory” (2nd year, 2nd quarter, “Degree in Telematics Engineering” and “Degree in Telecommunication Systems Engineering”)

| Family | Algorithm | Description |
|---|---|---|
| Bandwidth assignment | `ba_networkUtilityMaximization` | Solve the Network Utility Maximization (NUM) problem giving an alpha-fair bandwidth for traffic demands. Require CVX solver installed and running |
| Capacity assignment | `ca_fixValue` | Set a constant capacity value for all links |
| | `ca_minAvDelayConcaveCost` | Minimizes the average network delay, with concave cost constraints. Requires CVX solver installed and running |
| | `ca_minAvNetDelayLinearCost` | Minimizes the average network delay, with linear cost constraints. Requires CVX solver installed and running |
| | `ca_netMinimaxUtilization` | Minimizes the maximum link utilization, with linear cost constraints. Requires CVX solver installed and running |
| Capacity and flow assignment | `cfa_shortestPathInKmFixedUtilization` | Shortest path routing and then set the capacities to match a given link utilization |
| Flow assignment | `fa_minimaxUtilization_xde` | Minimizes the maximum link utilization, using flow-link formulation. Requires CVX solver installed and running |
| | `fa_minimaxUtilization_xdp` | Minimizes the maximum link utilization, using flow-path formulation. Requires CVX solver installed and running |
| | `fa_minNetDelay_xde` | Minimizes the average network delay, using flow-link formulation. Requires CVX solver installed and running |
| | `fa_shortestPath_km` | Route all the traffic in the shortest path using link length (km) as the cost figure |
| | `fa_shortestPath_numHops` | Routes all the traffic in the shortest path using 1 as cost per link |
| Topology assignment | `la_bidirectionalMinimumSpanningTree_Prim_Km` | Generates a bidirectional minimum spanning tree using Prim's algorithm and link length as cost per link |
| | `la_randomBidirectionalUntilConnected` | Randomly generates bidirectional links until the network becomes connected |
| | `la_randomUnidirectionalUntilConnected` | Randomly generates unidirectional links until the network becomes connected |
| | `la_tspNearestNeighbourKmBidirectional` | Generates a ring topology using nearest neighbour algorithm with cost of a link given by the link length in km |
| | `la_unidirectionalMinimumSpanningTree_Prim_Km` | Generates a unidirectional minimum spanning tree using Prim's algorithm and link length as cost per link |
| | `na_randomUniform` | Returns a set of nodes randomly placed within a given grid |
| | `na_topFiveSpain` | Returns a set of nodes (along with a "population" attribute) representing the five largest cities in Spain (in terms of population) |
| | `na_topSevenSpain` | Returns a set of nodes (along with a "population" attribute) representing the seven largest cities in Spain (in terms of population) |

Table 3: Summary of built-in algorithms included in Net2Plan

| Family | Library | Description |
|---|---|---|
| checks | `isBidirectional` | Check if the topology is bidirectional: same number of links between each node pair in both directions (assuming multi-digraphs) |
| | `isBidirectionalAndSymmetric` | Check if the topology is bidirectional and symmetric: same number of links between each node pair in both directions (assuming multi-digraphs) and same weights per direction. Can be used to check if links are bidirectional and distance-symmetric (if weight is link length), bidirectional and capacity-symmetric (if weight is link capacity), if demands are symmetric (if linkTable is demandTable and weight is offeredTrafficInErlangs) and so on |
| | `isConnected` | Check if the topology is connected |
| | `isDemandTable` | Check whether a demand table is valid |
| | `isLinkTable` | Check whether a link table is valid |
| | `isPathList` | Check whether a path list is valid |
| | `isSimpleGraph` | Check if a graph is simple: no more than one link per node pair |
| | `isTrafficMatrix` | Check whether a traffic matrix is valid |
| computeGraph | `floydAlgorithm` | Floyd's algorithm for finding all-pairs shortest path in a weighted graph with positive weights |
| | `primAlgorithm` | Function to generate a minimum spanning tree using Prim's algorithm |
| | `shortestPath` | Dijkstra's algorithm for finding the shortest path between two given nodes |
| conversions | `adjacencyMatrix2linkTable` | Function to convert a $\|N\| \times \|N\|$ adjacency matrix into a $\|E\| \times 2$ link table where each row represents the origin and destination nodes of the link |
| | `demandTable2trafficMatrix` | Function to convert a $\|D\| \times 2$ demand table in a $\|N\| \times \|N\|$ traffic matrix where each entry $(i, j)$ represents the units of traffic from $i$ to $j$ |
| | `linkTable2adjacencyMatrix` | Compute the adjacency matrix from a link table of a multi-digraph |
| | `linkTable2adjacencyMatrix` | Compute the incidence matrix from a link table of a multi-digraph |
| | `nodeXYPositionTable2distanceMatrix` | Compute distance matrix between each node pair from a XY-position table of the nodes |
| | `seqLinksPerPath2linkOccupancyPerPath` | Function to convert a pathList into a $\|P\| \times \|E\|$ matrix in which each entry $(p, e)$ represents how many times $p$ traverses link $e$ |
| | `sequenceOfLinks2sequenceOfNodes` | Given a linkTable and a sequenceOfLinks, return the sequenceOfNodes traversed |
| | `sequenceOfNodes2sequenceOfLinks` | Given a linkTable and a sequenceOfNodes, return the sequenceOfLinks traversed |
| | `trafficMatrix2demandTable` | Function to convert a $\|N\| \times \|N\|$ traffic matrix in a $\|D\| \times \|2\|$ demand table and $\|1\| \times \|D\|$ demand vector where the $i$-th row in demandTable contains the input and output nodes of the demand, and the $i$-th value in demandVector is the corresponding amount of traffic in Erlangs |
| | `xde2xdp` | Function to convert from node-link routing to link-path routing |
| | `xdp2xde` | Function to convert from link-path routing to node-link routing |
| metrics | `erlangBLossProbability` | Erlang-B probability of call blocking in a M/M/n/n queue system |
| | `inverseErlangB` | Return "numberOfServers" required to guarantee a certain "gradeOfService" under an utilization "load" in a M/M/n/n queue system |
| misc | `isCvxInstalledAndRunning` | Check whether CVX solver is installed and running |
| | `isSedumiInstalledAndRunning` | Check whether SeDuMi solver is installed and running |
| | `isYalmipInstalledAndRunning` | Check whether YALMIP solver is installed and running |
| | `removeRoutingInformation` | Function to remove routing information from netStruct when due to any operation (as a link capacity change), it becomes obsolete |
| | `trafficPerLink` | Function to compute carried traffic per link using a link-path formulation |

Table 4: Summary of built-in libraries included in Net2Plan

- “Network planning and management” (3rd year, 2nd quarter, “Degree in Telematics Engineering”)

Net2Plan is also a part of the Ph.D. work of José Luis Izquierdo Zaragoza, supervised by Prof. Pablo Pavón Mariño.

Aside from Net2Plan, authors also collaborate in the development of MatPlanWDM, an open-source tool for multilayer optical networks.

Pablo Pavón Mariño
José Luis Izquierdo Zaragoza

Contents

## 13 Bug Reports

The development of Net2Plan is to a large degree driven by users' bug reports. In most cases, when providing feedback, it is essential to include the following information:

- Description of the problem and tool you were using when happened (problem solver, traffic generator...)
- Network structure and traffic files
- A screenshot of your computer screen showing the error

By the way, please don't contact authors for doing your assignments or requesting new features, probably you won't get an answer.

Contents

## 14 References

[1] L. Kleinrock, *Queueing Systems, Volume 2: Computer Applications.* John Wiley & Sons, 1976.

[2] R. S. Cahn, *Wide Area Network Design: Concepts and Tools for Optimization.* The Morgan Kaufmann Series in Networking, Morgan Kaufmann, 1 ed., July 1998.

[3] P. Pavon-Marino, “Lectures of the Telecommunication Networks Theory course,” 2012.

[4] M. C. Grant and S. Boyd, “CVX: Matlab Software for Disciplined Convex Programming.” Website: `http://cvxr.com/cvx/` [Last accessed: January 10, 2012].

Contents

## A Release notes

Net2Plan 0.1.8 (June 18, 2012)

- Fixed: Minor bugs

Net2Plan 0.1.7 (June 4, 2012)

- Fixed: Minor bugs

Net2Plan 0.1.6 (May 15, 2012)

- Fixed: Minor bugs
- Improvement: Load/save dialog remember the last used folder
- Added: More algorithms are included (new CA, CFA and FA algorithms)

Net2Plan 0.1.5 (May 11, 2012)

- Fixed: Minor bugs
- Refactoring: Node and link assignment problems are moved to a new "topology assignment" directory
- Refactoring: Algorithms `ca_/fa_netMinCongestion` are renamed as `ca_/fa_netMinimaxUtilization`
- Improvement: Algorithms and user reports can be executed from any directory

Net2Plan 0.1.4 (April 18, 2012)

- Fixed: Minor bugs

Net2Plan 0.1.3 (April 12, 2012)

- Fixed: Minor bugs

Net2Plan 0.1.2 (March 28, 2012)

- Fixed: Minor bugs
- Change: A third input parameter (options) has been added to the signature of the algorithms containing Net2Plan-wide parameters. These options have a global scope to all Net2Plan modules
- Refactoring: Minor appearance changes in "Traffic generation"
- From this version, help file is also shipped in PDF

Net2Plan 0.1.1 (March 16, 2012)

- Fixed: Minor bugs
- Refactoring: “Problem solver” is now “Network design”

Net2Plan 0.1 (February 2012)

- Initial version

Contents

# B Metrics

In this section several performance and cost metrics are described. These metrics are useful to define objective functions or problem constraints. For a complete reference on network metrics, see [3].

Contents

## B.1 Topology

- Number of nodes
$$|N| \tag{4}$$
- Number of links
$$|E| \tag{5}$$
- Average node degree
$$\frac{|E|}{|N|} \tag{6}$$
- Network diameter: Maximum length among all pairs shortest path
$$\max\{l(p)\} \tag{7}$$
- Average shortest path length: Average length among all pairs shortest path
$$\overline{n} = \frac{\sum\limits_{p\in P} l(p)}{|N|\cdot(|N|-1)} \tag{8}$$
- Algebraic connectivity: Second smallest eigenvalue of the Laplacian matrix of adjacency matrix. This eigenvalue is greater than 0 if and only if topology is connected. The magnitude of this value reflects how well connected the overall topology is.
- Connected topology: Indicates whether topology is connected, this is, it is possible to find a path from one node to each other
- Average link distance (km)
$$\frac{\sum\limits_{e\in E} l(e)}{|E|} \tag{9}$$

Contents

## B.2 Link capacities

- Total capacity installed (Erlangs)

$$\sum_{e \in E} u_e \tag{10}$$

- Average capacity installed (Erlangs)

$$\frac{\sum_{e \in E} u_e}{|E|} \tag{11}$$

- Capacity module size (Erlangs): Shows the greatest common divider among all link capacities

Contents

## B.3 Offered traffic

- Number of demands

$$|D| \tag{12}$$

- Average number of demands per node pair

$$\frac{|D|}{|N| \cdot (|N| - 1)} \tag{13}$$

- Average offered traffic per demand (Erlangs)

$$\frac{\sum_{d \in D} h_d}{|D|} \tag{14}$$

Contents

## B.4 Routing (Carried traffic)

- Average number of hops

$$\overline{n} = \frac{\sum_{e \in E} y_e}{\sum_{d \in D} r_d} \tag{15}$$

- Average ingress traffic per node (Erlangs)

$$\frac{\sum_{d \in D} r_d}{|N|} \tag{16}$$

- Average egress traffic per node (Erlangs)

$$\frac{\sum_{d \in D} r_d}{|N|} \tag{17}$$

- Average traversing traffic per node (Erlangs)

$$\frac{\sum_{e \in E} y_e - \sum_{d \in D} r_d}{|N|} \tag{18}$$

- Bifurcation degree: Average number of paths which traverse each demand

$$\frac{|\{(d,p) | x_{dp} > 0, d \in D, p \in P_d\}|}{|D|} \geq 1 \tag{19}$$

- Average number of km per demand

$$\frac{\sum_{d \in D} r_d \left( \sum_{p \in P_d} x_{dp} \left[ \sum_{e \in p} l_e \right] \right)}{\sum_{d \in D} r_d} \tag{20}$$

- Average number of km per demand

$$\frac{\sum_{e \in E} y_e \cdot \rho_e}{\sum_{e \in E} y_e} \tag{21}$$

- Network congestion: Maximum load among all links

$$cg = \max_{e \in E} \{\rho_e\} \tag{22}$$

- Integral flows: Indicates whether all flows are routed in integer units of traffic
- Unsplittable flows: Indicates whether all flows are routed at most over one path

Contents

## B.5 Rate

- Throughput (Erlangs): Total traffic injected to the network

$$\sum_{d \in D} r_d \tag{23}$$

- % Lost traffic

$$100 \cdot \frac{\sum_{d \in D} (h_d - r_d)}{\sum_{d \in D} h_d} \tag{24}$$

- Average throughput per demand (Erlangs)

$$\frac{\sum_{d \in D} r_d}{|D|} \tag{25}$$

- Jain's fairness index (absolute)

$$\frac{\left(\sum_{d\in D}(h_d - r_d)\right)^2}{|D|\sum_{d\in D}(h_d - r_d)^2} \tag{26}$$

- Jain's fairness index (proportional)

$$\frac{\left(\sum_{d\in D}\frac{h_d - r_d}{h_d}\right)^2}{|D|\sum_{d\in D}\left(\frac{h_d - r_d}{h_d}\right)^2} \tag{27}$$

Contents

## B.6 Delay

In packet-switched networks, traffic sources split data into smaller pieces called *packets*, along with a header with control information. Per each received packet, switching nodes read its header and take appropriate forwarding decisions.

In real networks, traffic is highly unpredictable and often modelled as random processes. When it is said that a traffic source $d$ generates $h_d$ traffic units, it is referred as average traffic. As a result, link capacities would be not enough to forward traffic and nodes have to store packets in queues, so they are delayed until they can be transmitted (this delay is known as *queueing delay*). If this situation remains for a long time, queues are filled and links become *saturated*, provoking packet drops.

Network design tries to model statistically delays and drops in order to minimize their effects. In Net2Plan each link is modelled as a queue fed by a self-similar source with a given Hurst parameter, getting the whole network average delay using *Kleinrock's independence* assumption.

- Propagation delay per link (seconds)

$$T_e^{prop} = \frac{l_e}{v_e^{prop}} \quad \forall e \in E \tag{28}$$

- Transmission delay per link (seconds): Average duration of the transmission of a packet. Since we assume a link capacity measured in Erlangs, $R_b$ is the capacity per Erlang.

$$T_e^{tx} = \frac{S}{R_b u_e} \quad \forall e \in E \tag{29}$$

- Buffering delay per link (seconds)

$$T_e^{b} = T_e^{tx}\frac{\rho_e^{1/2(1-H)}}{(1-\rho_e)^{H/(1-H)}} \quad \forall e \in E \tag{30}$$

  where $H \triangleq$ Hurst Parameter (here $H$ doesn't mean total offered traffic). Choosing $H = 0.5$ yields to same result predicted by M/M/1 queue model.

- Delay per link (seconds)

$$T_e = T_e^{prop} + T_e^{tx} + T_e^{b} \quad \forall e \in E \tag{31}$$

- Average network delay (seconds)

$$T = \frac{1}{\sum\limits_{d \in D} r_d} \sum_{e \in E} y_e T_e \tag{32}$$

Contents

## B.7 Blocking

In circuit-switched networks, traffic sources reserve a given capacity during certain time, along paths followed by traffic demands. It is possible that if a new traffic source wants to reserve resources its petition would be blocked, since it would not be enough available resources to satisfy its demand.

Blocking performance metrics are computed using Erlang-B formula. Obviously only has sense when **u** is integer.

- Blocking probability of a link

$$B_e = \text{Erlang-B}(u_e, y_e) \tag{33}$$

- Average network blocking probability

$$B = \frac{\sum\limits_{e \in E} y_e \cdot B_e}{\sum\limits_{d \in D} r_d} \tag{34}$$

- Worst network blocking probability

$$\max_{e \in E} \{B_e\} \tag{35}$$

Contents

# C CVX solver

CVX is a MATLAB-based modelling system for convex optimization. CVX turns Matlab into a modelling language, allowing constraints and objectives to be specified using standard Matlab expression syntax.

CVX has been used in Net2Plan to develop some algorithms, not in kernel, thus you are free to install or not this solver. Please note that if CVX is not properly installed, algorithms which use it won't work (take a look into section 10 to see algorithms that use it).

CVX is not shipped with Net2Plan, but it is publicly (and no-charge) available on its website [4]. Once you download CVX (if you decided to use it), you must run `cvx_setup` in MATLAB to install it.

Net2Plan team don't provide any support to CVX.

Contents