Enif – Toward a New Interface for EMME/2

Heinz Spiess [*]

October 2000

Abstract

This paper is aimed at presenting a first introduction to Enif, a new software which provides a modern graphical user interface for accessing existing EMME/2 data banks. Enif is not intended to replace EMME/2, but to coexist with EMME/2 and complement it. The main goal of the paper is to discuss the design of Enif and explain the technical concepts on which it is based. These are not limited to the graphic user interface and the production of high quality graphic output, but go all the way down to defining new mechanisms how to store and handle network data and how to provide user configurable objects. In fact, the only thing Enif has in common with EMME/2 is that both can be used to access the same EMME/2 data banks.

Output examples produced with the current pre-production implementation of Enif are included in order to illustrate how Enif's features can be combined in a very flexible way to produce a great variety of network related graphic output.

Contents:

[*]EMME/2 Support Center, Haldenstrasse 16, CH-2558 Aegerten, Switzerland. Email: heinz@spiess.ch

Introduction

This year, it is 20 years that the initial design of EMME/2 was decided upon. Since that time, EMME/2 has continuously been enhanced and improved, adding many new features and modeling capabilities. However, the basic design described in the initial publications on EMME/2 (1, 3) is essentially still valid today. On the one hand, this proves the importance of building on a solid conceptual base – without it, EMME/2 would never have gone this far and kept growing all the time. On the other hand, times have changed a lot since the early eighties, and interactive computing is a totally different thing today than it was then. While the "age" of the basic concepts behind EMME/2 was never felt as a hindrance to enhancing the software with new modeling features, the "age" of the graphic subsystem of EMME/2 is evident, both to the users who have to operate the system, and to us developers who are seriously limited by the outdated technology assumptions on which EMME/2's graphic subset is based.

While it was possible to improve the graphic subsystem of EMME/2 in many ways over the past 20 years, these enhancements had always to be done in a way which did not affect compatibility with older versions. While this always was (and still is) an important criterion for EMME/2 developments, over the past years it became clear that the need for a new graphical interface cannot be satisfied by gradual and upward compatible enhancements of the current EMME/2 system, but that only a completely new and radically different design would give us the necessary leeway to satisfy the interactive graphic expectations of the EMME/2 users now and for the years to come.

This cannot and does not mean that the traditional EMME/2 software is to be discontinued and its functionality replaced by a new and incompatible piece of software. Rather, we foresee both the traditional EMME/2 and the new software based on the new concepts will have to coexist for quite some time. This is not only necessary for the continuity of the large existing EMME/2 application base, but the huge fundus of tools and modeling capabilities that EMME/2 offers is too valuable to just throw away and start from scratch.

Thus, the project *Enif*, which is presented for the first time in this paper, is *not* aimed at replacing the traditional EMME/2 modules in the foreseeable future. But it *is* aimed at providing a new way of interfacing graphically with EMME/2 data within a reasonable short time frame and it *is* designed to be solid enough to be able to grow into the future and eventually provide a functionality equivalent (but not the same!) as does EMME/2 currently.

In this paper the concepts behind Enif are explained and they are illustrated using examples from the current implementation, which can be used to produce graphic and list output based on EMME/2 data banks. The emphasis is put on making the reader understand the foundation on which Enif is built. The user interface itself, i.e. the details of the interaction between the user and Enif are not dealt with explicitly, since a) the current implementation is not final and the external aspects may still change and b) this paper is not intended to be a reference manual.

Enif – The Basic Concepts

When the Enif project started in the summer of 1999, its design was approached with the following goals in mind:

1. Provide all functional features needed within the framework of a consistent modern graphical user interface.
2. Remain compatible with all current EMME/2 applications at the data bank level. This compatibility implies being able to access the data available in these data banks. It does not imply that Enif should be able to produce identical output as does EMME/2.
3. Run with identical functionality on all platforms on which the current version of EMME/2 is running. This means that Enif must be able to run both under the Microsoft Windows family of operating systems as well as under the various flavors of Unix supported by EMME/2.

4. Build a solid base which is strong enough, both on the conceptual level and on the implementation side, to build upon the future developments of EMME/2 for many years to come.
5. Provide useful functionality to the EMME/2 users within a reasonably short time frame.

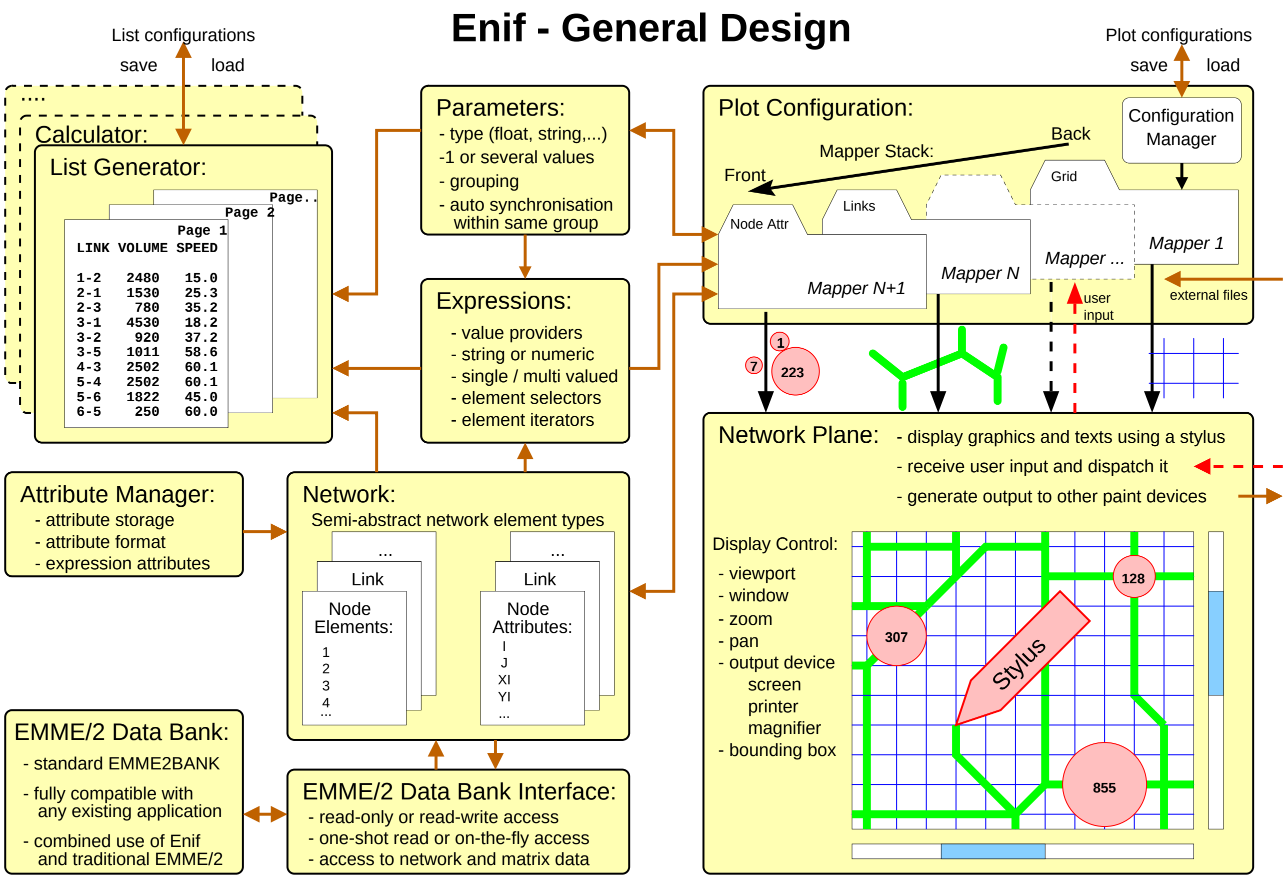
In the past 15 months most effort went into the development of the general concepts and the basic mechanisms needed for their implementation. These concepts are more important for understanding the "philosophy" behind Enif than the details of graphical interface presented to the user on screen.

The basic design of Enif is shown in Figure 1 in the form of a block diagram. The central part of the diagram describes the internal concepts which, as such, are not specific for the graphical user interface, but form the abstract pilers on which Enif is built. This includes the internal representation of the *network*, the attributes, the *expressions* and the *parameters*, which are the basic objects used to build user configurable objects in Enif.

The graphical interface part of Enif is depicted on the right hand side of Figure 1. It is based on an abstract *network plane* which receives the graphical information generated by a set of *mappers* using an abstract drawing and a writing tool we call the *stylus*.

Other functionalities can be based on the foundation provided by Enif, as indicated in the upper left corner of the block diagram. One example of this is the *list generator*.

The access to the EMME/2 data bank is completely modular and separated from the rest of the Enif code. This implies that accessing network data from other sources than an EMME/2 data bank could be achieved by simply providing the corresponding alternate network I/O modules, should this ever be required.



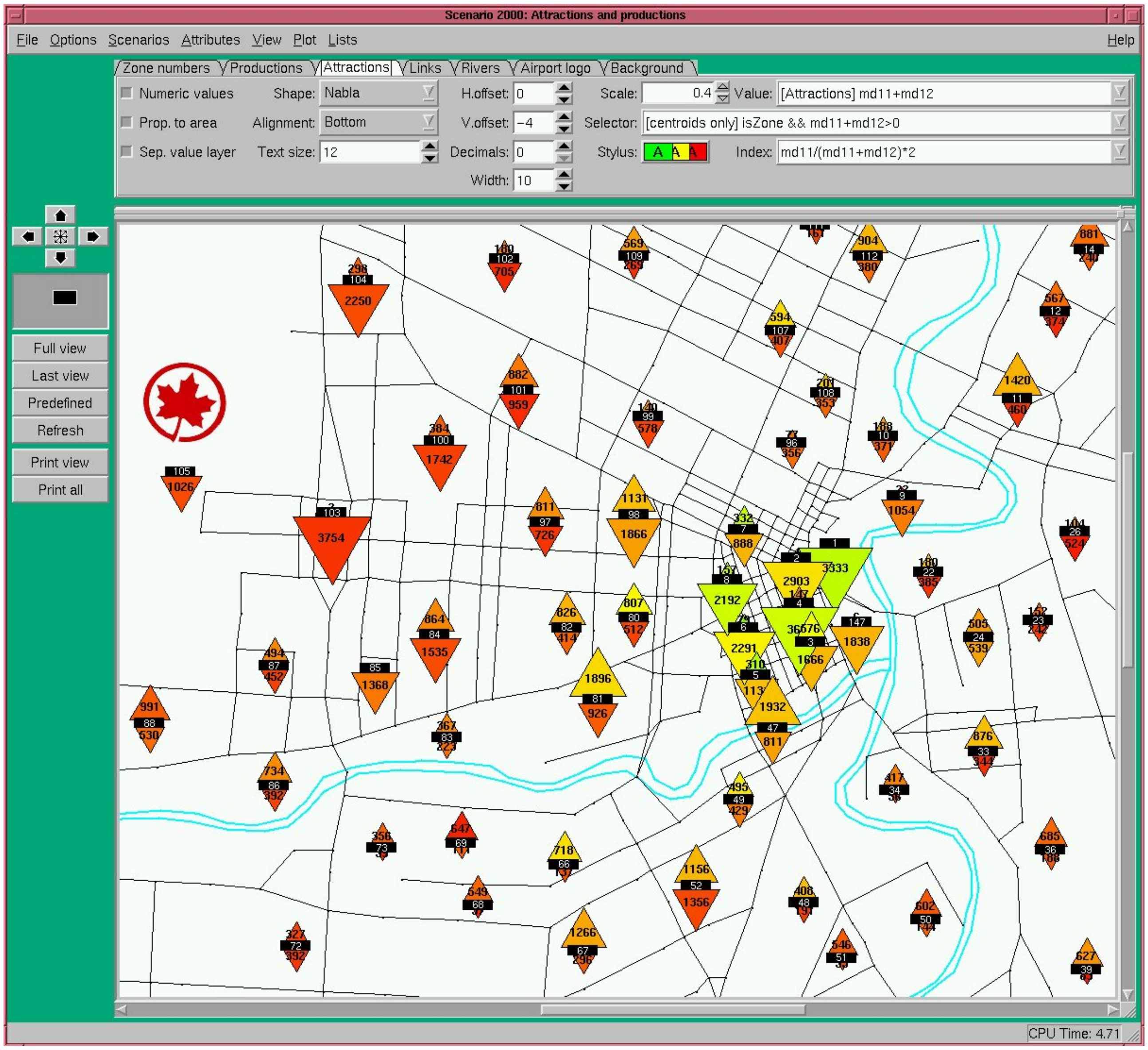
```
LINK VOLUME SPEED

1-2   2480   15.0
2-1   1530   25.3
2-3    780   35.2
3-1   4530   18.2
3-2    920   37.2
3-5   1011   58.6
4-3   2502   60.1
5-4   2502   60.1
5-6   1822   45.0
6-5    250   60.0
```

Figure 1: General design of Enif

In the following sections, all these fundamental building blocks of Enif will be discussed in more detail. While reading on, it is a good idea to return back to Figure 1 from time to time. What at the beginning might look like a bunch of empty phrases, will —hopefully!— after having reached the end of this paper be filled with concrete meaning and serve as a useful summary of the functionality of Enif.

In order to give an idea how Enif presents itself on the screen, a typical Enif window is shown in Figure 2. It shows the zonal productions and attractions colored according to the corresponding mode share of the auto mode.

The main part of the Enif window shows the current view of the network plane. Located above it, the tabbed mapper control dialogs are used to access the configurable parameters of the current plot configuration. At the left side of the window, the view controls can be used to change the current view or to send the current plot to the printer. A menu bar on the top gives access to the various available tools and options, and, finally, the status bar on the very bottom of the window is used to display transient messages.


Scenario 2000: Attractions and productions
File Options Scenarios Attributes View Plot Lists Help
Zone numbers Productions Attractions Links Rivers Airport logo Background
Full view
Last view
Predefined
Refresh
Print view
Print all
CPU Time: 4.71

Numeric values ☐ Prop. to area ☐ Sep. value layer ☐
Shape: Nabla
Alignment: Bottom
Text size: 12
H.offset: 0
V.offset: -4
Decimals: 0
Width: 10
Scale: 0.4
Selector: [centroids only] isZone && md11+md12>0
Stylus: A A A
Value: [Attractions] md11+md12
Index: md11/(md11+md12)*2

Figure 2: A typical Enif window

Network and Network Elements

Since one important goal of Enif is to remain 100% compatible with existing EMME/2 application data, the network structure is essentially given by EMME/2. However, the internal network representation is completely different from the one used in EMME/2 (2, 3). The access and interpretation of EMME/2 network data is completely encapsulated in a separate modular network I/O interface.

Internally all network elements are represented using a common base class. This allows the implementation of many functionalities in a uniform way, independent of the actual type of network elements. This includes functionalities such as data access, handling of network attributes, expression evaluation and the definition of sub-networks by element selection.

Element type:	**Pointers:**	**Flags:**
mode	next mode in sequence	`isAuto, isTransit, isAuxAuto, isAuxTransit`
node	next node in sequence, first outgoing link, first incoming link	`isZone, isIntersection`
link	I-node, J-node, next link with same I-node, next link with same J-node, first outgoing turn, first incoming turn	`isAccess, isEgress, isConnector, isAuto, isTransit, isAuxAuto, isAuxTransit`
transit vehicle	next transit vehicle in sequence, transit mode	
transit line	next transit line in sequence, first transit segment, transit mode	
transit segment	next segment of same line, transit line, I-node, link	`isFirst, isMidLayover, isLast, isHidden`
turn	incoming link, outgoing link, next turn with same incoming link, next turn with same outgoing link	`isUTurn`
zone	node	
origin	zone	
destination	zone	
O-D pair	origin, destination	

Table 1: Element type specific pointers and flags

The network element types include all those already used in EMME/2, i.e. *mode*, *node*, *link*, *transit vehicle*, *transit line*, *transit segment* and *turn*. In contrast to EMME/2, matrix related data is also considered part of the generalized network. This leads to the additional network element types *zone*, *origin*, *destination* and *O-D pair*.

While most data handling can be implemented at the abstract network element level, all network specific aspects have to be implemented at the level of the specific element type, e.g. finding the outgoing or incoming links of a node or following the segments along the itinerary of a transit line. This is achieved by associating with each network element type a type specific set of pointers to other network elements, as well as a set of flags. These are shown in Table 1.

The data associated with an individual network element consists of three parts:

- a type independent part containing index and flag information;
- a mandatory type dependent part containing all structural data necessary for the corresponding element type (this part essentially consists of generic element identifiers and type specific pointers);

- an optional, user configurable and application specific part containing information which needs to be accessed efficiently.

In Enif, a *network* is defined as the collection of its network elements, an attribute list for each element type (see below) and some auxiliary data, such as titles and element counts.

Network Attributes and Attribute Lists

One of the most basic and most important functionalities when dealing with large transportation networks is to provide a flexible and efficient way of accessing attribute values associated with network elements. In Enif, this is implemented using a very generalized approach consisting of *network attributes* which are structured into *attribute lists*.

Enif's concept of network attribute is much broader than the one used in EMME/2. While in EMME/2 a network attribute always implies a value which is physically stored in the EMME/2 data bank, in Enif the term attribute covers any kind of predefined *value provider* for the network elements of a specific type.

Network attributes are characterized by the following properties:

- Element type: Defines the type of network elements for which this attribute can provide values.
- Value type: Defines the type of value which is provided by this attribute, such as *boolean*, *integer*, *floating point*, *string* or *pointer* to another network element.
- Data source: Attribute values may be obtained in many different ways:
  - directly from the information stored within the network element's data buffer;
  - using "on-the-fly" access by calling the EMME/2 data bank I/O module whenever a value is needed;
  - by indirect access to an attribute of another network element via a pointer attribute;
  - by calling a type specific network element function;
  - by providing the same constant value for all network elements;
  - by evaluating an expression based on the other attributes of the same element (see below).
- Format: Information on the default formatting of the attribute values, such as field width, decimal precision and alignment.
- Protection and validation: Attributes can be declared read-only (e.g. assignment results) or have write permission so that their values may be modified. For writable numerical attributes, the allowed value range may be defined by lower and upper bounds, as well as a default value which is used to initialize newly created elements.

A special data caching mechanism is provided which allows the user to define which attributes should be *cached*, i.e. stored directly in the data buffer of each network element, instead of reading it on-the-fly from the external data bank whenever needed.

Obtaining an attribute value for a specific network element is done simply by calling the corresponding method of the attribute with the pointer to the network element as argument.

Constant attributes are attributes which are not associated with a particular network element. They can be evaluated for all network element types (or even outside the context of network elements altogether) and will always return the same constant value.

Attributes of the same element type are combined into *attribute lists*. The network contains one attribute list for the attributes of each element type, and an additional one for constant attributes associated with the network. The latter includes constants such as the scenario number, the scenario title, mode masks and the scalars defined in the data bank.

Additional "private" attribute lists may be created when needed, e.g. in the context of a list or a specific mapper.

Expressions

In Enif, most functionalities which access network data are not implemented by accessing network attributes directly, as is the case in EMME/2. Rather, wherever possible, the available functionalities (such as selecting subnetworks, plotting or listing network related data) are based on *expressions*.

In general, an expression is made up of an arbitrary combination of *operands*, *operators* and calls to *intrinsic functions*, using syntax rules compatible with those used by the expressions in EMME/2. In practice, however, an expression is often as simple as a single constant value (e.g. "`1`") or an attribute name (e.g. "`volau`").

In contrast to EMME/2, where expressions are limited to handle exclusively numerical values, Enif expressions support both *numerical* and *string* values. A strict distinction is made between numerical and string values. Each operator has well defined operand types and each intrinsic function requires its arguments to be of the specified types. Special intrinsic functions can be used to convert strings to numbers and vice versa, should this become necessary.

Each operand of an expression must correspond to one of the following:

- a numeric value (e.g. `0`, `3.14159`),
- a constant string enclosed in double quotes (e.g. `"abc"`),
- a numeric or string attribute from one of the associated attribute lists (e.g. `volau`),
- a valid subexpression enclosed in parentheses (e.g. `(volau+volad)`),
- the result of a call to an intrinsic function (e.g. `max(speed,60)`.

A list of all available operators is shown in Table 2, grouped in the order of increasing operator precedence. It also shows the required operand types for each operator, as well as the type of the result of the operation. All operators are binary operators, with the exception of the "`+`" and "`-`" operators; these can be used as unary operators at the beginning of subexpressions.

Note that all logical and comparison operators return 0 for FALSE and 1 for TRUE. Operands of logical operators are assumed TRUE for all non-zero values, FALSE for zero values.

The available intrinsic functions are shown in Table 3. Note that some functions allow for a variable number of arguments. For technical reasons, the number of arguments for these functions is currently limited to a maximum of 30. Two noteworthy functions of this type are the `lookup()` and the `which()` functions. The function `lookup(`$i, v_1, v_2, v_3, \ldots$`)` takes an index i as first argument and returns v_i, the i-th of the following values. The function `which(`$v, v_1, v_2, v_3, \ldots$`)` takes a value v as first argument and compares it with the following values v_i. It returns the index i of the first match $v = v_i$ found, or 0 otherwise.

In addition to operands and operators, an expression may also contain *comments* enclosed in brackets, e.g. "`[auto network without connectors] isAuto && not(isConnector)`". This is useful to explain the meaning of an expression to those users who are not (yet) Enif experts.

Depending on the context, an expression may allow only a numerical result or also accepts a string result. But even expressions which are limited to numerical results may include string valued subexpressions.

In certain contexts it is possible to specify an expression returning more than a single value. This is done by simply specifying several subexpressions, separated by commas. The maximum allowed number of expressions is given by the particular context. E.g. the expression

```
volau-volad, volad
```

provides two values, the auto volumes minus the additional volumes as first value and the additional volumes as second value.

A special case is the *empty expression*, i.e. an expression with no operands at all. Empty expression will always return a zero value as result.

While the expressions in Enif are much more powerful than those in EMME/2, they remain essentially compatible with EMME/2 expressions. All attributes, operators and intrinsic functions which are

Operator:	Description:	Operand and result types:
`||` or `.or.`	logical OR	*num*`||`*num* → *num*
`&&` or `.and.`	logical AND	*num*`&&`*num* → *num*
`.xor.`	logical XOR	*num*`.xor.`*num* → *num*
`!=` or `.ne.`	not equal	*num*`!=`*num* → *num*
`<` or `.lt.`	less than	*num*`<`*num* → *num*
`<=` or `.le.`	less than or equal	*num*`<=`*num* → *num*
`==` or `.eq.`	equal	*num*`==`*num* → *num*
`>=` or `.ge.`	greater or equal	*num*`>=`*num* → *num*
`>` or `.gt.`	greater than	*num*`>`*num* → *num*
`!=`	lexically not equal	*str*`!=`*str* → *num*
`<`	lexically less than	*str*`<`*str* → *num*
`<=`	lexically less than or equal	*str*`<=`*str* → *num*
`==`	lexically equal	*str*`==`*str* → *num*
`>`	lexically greater than	*str*`>`*str* → *num*
`>=`	lexically greater or equal	*str*`>=`*str* → *num*
`~`	string matched by regular expression	*str*`~`*str* → *num*
`!~`	string not matched by regular expression	*str*`!~`*str* → *num*
`&`	bitwise AND	*num*`&`*num* → *num*
`+`	add	*num*`+`*num* → *num*
`+`	string concatenation	*str*`+`*str* → *str*
`-`	subtract	*num*`-`*num* → *num*
`.addle.`	add if less or equal to zero	*num*`.addle.`*num* → *num*
`.max.`	maximum	*num*`.max.`*num* → *num*
`.min.`	minimum	*num*`.min.`*num* → *num*
`.pdfum.`	add and truncate when negative	*num*`.pdfum.`*num* → *num*
`|`	bitwise OR	*num*`|`*num* → *num*
`*`	multiply	*num*`*`*num* → *num*
`.mod.`	modulo	*num*`.mod.`*num* → *num*
`/`	divide	*num*`/`*num* → *num*
`^` or `**`	power	*num*`^`*num* → *num*

Table 2: Operators in expressions

available in EMME/2 are also available in Enif.

Expressions are always stored as normal strings, which also implies that they can easily be saved to files and read back when needed. However, when expressions are actually used to provide values, their string representation is automatically compiled into an efficient internal RPN token list. Special cases, such as empty expressions, constants or expressions consisting of a single attribute are recognized as such, so that they can be handled with less overhead. These features allow for a very efficient evaluation of the same expression for large numbers of network elements.

Since the user can enter and modify expressions at any time, it may happen that he or she enters an invalid expression. For this reason, expressions entered or modified interactively by the user are compiled immediately after the return key is pressed. If the expression is found to be invalid, the background of the expression field becomes red (or to be more precise: changes to a special color which is configurable in the user preferences) and the cursor is moved to the place in the expression where the error was detected. If this is not enough to reveal the cause of the error to the user, he can also check on the diagnostic window, where he will find a detailed description of the error.

Function:	**Description:**	**Argument and result types:**
`abs( )`	absolute value	`abs(`*num*`)` → *num*
`Atan( )`	arc tangent (degrees)	`Atan(`*num*`)` → *num*
`atan( )`	arc tangent (radians)	`atan(`*num*`)` → *num*
`Atan2( )`	arc tangent (degrees)	`Atan2(`*num*`,`*num*`)` → *num*
`atan2( )`	arc tangent (radians)	`atan2(`*num*`,`*num*`)` → *num*
`ceil( )`	ceiling function	`ceil(`*num*`)` → *num*
`Cos( )`	cosine function (degrees)	`Cos(`*num*`)` → *num*
`cos( )`	cosine function (radians)	`cos(`*num*`)` → *num*
`erf( )`	Gaussian error function	`erf(`*num*`)` → *num*
`exp( )`	exponential function	`exp(`*num*`)` → *num*
`floor( )`	floor function	`floor(`*num*`)` → *num*
`get( )`	return corresponding stack value	`get(`*num*`)` → *num*
`if( )`	if-then-else on string values	`if(`*num*`,`*str*`,`*str*`)` → *str*
`if( )`	if-then-else on numerical values	`if(`*num*`,`*num*`,`*num*`)` → *num*
`index( )`	index of first substring	`index(`*str*`,`*str*`)` → *num*
`int( )`	truncate to integer	`int(`*num*`)` → *num*
`justify( )`	left/right justify string	`justify(`*str*`,`*num*`)` → *str*
`left( )`	left substring	`left(`*str*`,`*num*`)` → *str*
`length( )`	string length	`length(`*str*`)` → *num*
`lgam( )`	logarithm of absolute gamma function	`lgam(`*num*`)` → *num*
`ln( )`	natural logarithm	`ln(`*num*`)` → *num*
`log10( )`	base-10 logarithm	`log10(`*num*`)` → *num*
`lookup( )`	text lookup	`lookup(`*num*`,`*str*`,`*str*`,...)` → *str*
`lookup( )`	numeric lookup	`lookup(`*num*`,`*num*`,`*num*`,...)` → *num*
`lower( )`	convert string to lower case	`lower(`*str*`)` → *str*
`match( )`	string matching	`match(`*str*`,`*str*`)` → *num*
`max( )`	maximum	`max(`*num*`,`*num*`,`*num*`,...)` → *num*
`min( )`	minimum	`min(`*num*`,`*num*`,`*num*`,...)` → *num*
`nint( )`	round to nearest integer	`nint(`*num*`)` → *num*
`not( )`	logical NOT	`not(`*num*`)` → *num*
`number( )`	convert string to number	`number(`*str*`)` → *num*
`put( )`	put argument on stack	`put(`*num*`)` → *num*
`puti( )`	define stack index for next put()	`puti(`*num*`)` → *num*
`rand( )`	random value between 0 and argument	`rand(`*num*`)` → *num*
`right( )`	right substring	`right(`*str*`,`*num*`)` → *str*
`sign( )`	sign function	`sign(`*num*`)` → *num*
`Sin( )`	sine function (degrees)	`Sin(`*num*`)` → *num*
`sin( )`	sine function (radians)	`sin(`*num*`)` → *num*
`sqrt( )`	square root	`sqrt(`*num*`)` → *num*
`string( )`	convert number to string	`string(`*num*`)` → *str*
`string( )`	convert number to string with decimals	`string(`*num*`,`*num*`)` → *str*
`upper( )`	convert string to upper case	`upper(`*str*`)` → *str*
`which( )`	which among string arguments	`which(`*str*`,`*str*`,`*str*`,...)` → *num*
`which( )`	which among numerical arguments	`which(`*num*`,`*num*`,`*num*`,...)` → *num*

Table 3: Intrinsic functions in expressions

Element Selectors and Iterators

In EMME/2, sub-network selection is implemented by a simple mechanism, in which a set of selection clauses is specified by the user, each clause being composed of an attribute and a corresponding value range. Network elements are selected if they satisfy the combination of all selection clauses.

In Enif, all network element selections are based on expressions. A *selector* is an expression which is used to determine which elements of a given type are used in a certain functionality. The expression is evaluated for each element of the given type and if the expression result is non-zero (or a non-empty string in the case of a string valued expression) the element is selected, whereas elements yielding a zero numeric value (or an empty string) are not selected. An exception to this rule is if the selection expression is empty, in which case all elements will be selected (this is different from EMME/2 where a selection without any clauses implied an empty set of network elements). In order to set a selector to select no elements, it suffices to set the selector expression explicitly to 0.

In order to allow an efficient handling of subsets, the selector expression is only evaluated once for each network element and the results are used to build a list of pointers to the selected elements which can then be traversed very efficiently for as many times as needed without any additional computational overhead.

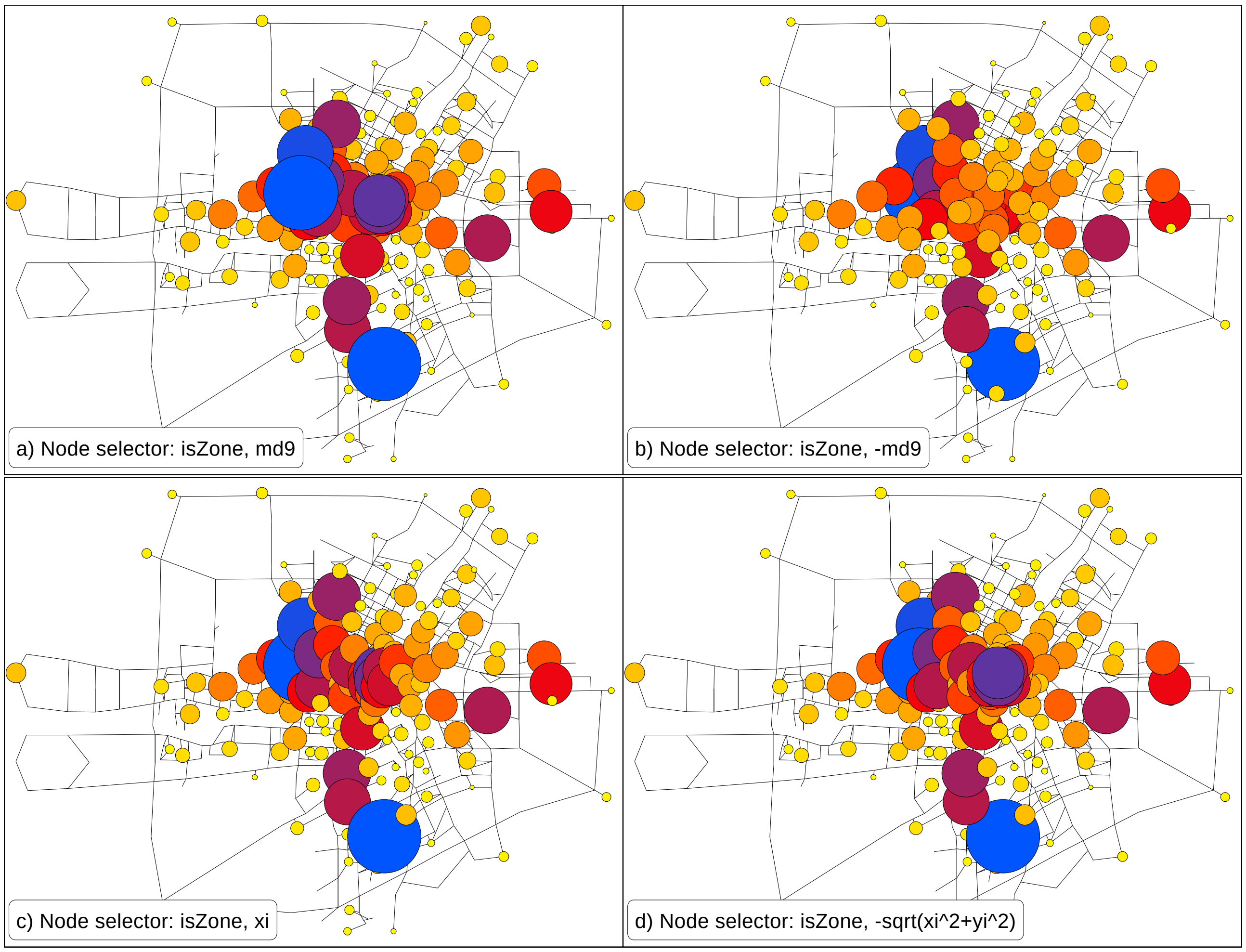

a) Node selector: isZone, md9
b) Node selector: isZone, -md9
c) Node selector: isZone, xi
d) Node selector: isZone, -sqrt(xi^2+yi^2)

Figure 3: Same node value plot using four different iterator expressions

While the element selection is used to decide which elements to use or not use in a certain functionality, it does not imply any particular order in which the elements are processed. In EMME/2, network elements are always processed in the sequence of ascending identifiers, i.e. in the same order they are stored in the data bank. In Enif, it is possible to define the order in which the selected elements of a given type are processed by means of a so called *iterator expression*. If an iterator expression is specified, the selected elements are processed in the order of increasing values of the iterator expression (or ascending collating sequence for string valued expressions). If no iterator expression is specified, the standard sequence following increasing element identifiers is used by default.

Thus, in Enif the concept of element selection always includes the option of also specifying the processing order of the selected elements. This is done by allowing the selector expression to provide the iterator value as an optional second subexpression. E.g. using the link selector "`volau>1000, -volau`" in a link list will select all links having an auto volume larger than 1000 and will sort the list to have the largest volumes first.

While the use of iterator expressions is evident for lists, it has also important implications when displaying overlapping graphic elements: elements drawn later will hide those elements drawn earlier at the same position. This is illustrated in Figure 3. It contains four times the same node value plot showing the attractions (stored in destination matrix `md9`) as proportional circles for the subset of nodes which correspond to zones (element selection: `isZone`). The only difference between the plots is the iterator expression. The following processing orders are shown: a) according to increasing attractions (`md9`), b) decreasing attractions (`-md9`), c) from left to right (`xi`) and d) from outside toward the network center (`-sqrt(xi^2+yi^2)`, the origin of the coordinate system happens to be in the city center).

The Network Plane

All concepts described so far dealt with the internal representation of network data and were not related at all to the graphical user interface provided by Enif. So let's now turn our attention to the graphic display of Enif and the basic concepts behind it.

In Enif the term *network plane* is used to denote an abstract functionality which provides the possibility to display arbitrary drawings (usually depicting transportation networks, hence the name network plane) and texts on the screen or on other output devices.

The network plane only offers the necessary infrastructure to display drawings and texts. It does, however, not generate any of those itself. The "clients" which are using the services of the network plane are the *mappers*. These are described in a separate section below.

Without going into too many technical and implementational details, here is a list of the services provided by the network plane:

- The *network plane* itself, an abstract surface on which drawing and texts can be generated using a well defined set of drawing and writing functions.
- The *network plane window* which provides an on-screen view of all or part of the network plane.
- A *legend window*, in which additional information describing the parameters used to create the plot are displayed for documentation and reference.
- A *magnifier* feature which can display a part of the current view at a magnified scale.
- A mechanism which obtains the *bounding box* of each currently defined mapper (i.e. the coordinate rectangle which bounds the region in which the mapper wants to display its information) and allows the mappers to update this information whenever it changes.
- *Coordinate transformations* from user defined network coordinates to screen coordinates and vice versa.
- A *view control system* which provides all necessary facilities to define and change the view, i.e. the part of the network plane which is currently visible in the network plane window. This implies all the necessary infrastructure and controls to allow zooming, panning, scrolling of the network window, as well as accessing previous or predefined windows or returning to the full view window.
- A *view updating mechanism* which will inform the clients when a new part of the network plane becomes visible and needs to be (re-)drawn.

- A system to *dispatch interactive user input* (mouse events and key strokes) to either the view control system or to the active input mapper.
- A *print facility which* sends the current plot to be output on a printer – either the content of the entire network plane or only the current view.

The Stylus

Generating even the most elaborate plot can finally be reduced to the following three simple basic tasks: drawing of lines, filling of regions and writing of texts. Let's now look at each of these tasks and see what display properties are associated with each of them:

- Outlines are drawn with a *pen* which is defined by its *color*, the *pen width* and a *line pattern*.
- Regions are *filled* in a given *color* using a certain *fill style*.
- *Texts* are displayed in a specified *color* using a font from a certain *family* in a specified *size* and optional *font attributes*, such as bold, italic or underlined.

General purpose graphic programs would normally require the specification of these display properties separately and independently for each single graphic element. However, when displaying transportation networks there are too many network elements, rendering it neither practical nor efficient to have to deal with each of them individually. Thus, it is crucial to conceptualize the graphic display properties in a way which allows a unified specification for large sets of network elements, but yet is flexible enough to customize plots to cover all (or at least most) needs.

In this spirit, all graphic drawing and writing in Enif is done by a versatile "all-in-one" type tool we shall call a *stylus*.

A *simple stylus* consists of the specification of one set of the following display properties:

<ins>Pen color</ins> as a full 24-bit RGB color specification;

<ins>Pen pattern</ins> as one of the following: solid line, dashed, dotted, dash-dot, dash-dot-dot, no pen;

<ins>Pen width</ins> as a value between 0 and 16 (0 denoting the smallest possible pen width of the output device);

<ins>Fill color</ins> as a full 24-bit RGB color specification;

<ins>Fill style</ins> as one of the following: solid fill, 94% fill, 88% fill, 63% fill, 50% fill, 37% fill, 12% fill, 6% fill, horizontal lines, vertical lines, cross hatch, diagonal up, diagonal down, diagonal cross, no filling;

<ins>Text color</ins> as a full 24-bit RGB color specification;

<ins>Text attribute</ins> as one of the following: normal, bold, italic, underline;

<ins>Text size factor</ins> as one of the following: 50%, 64%, 80%, 100% (normal size), 120%, 150%, 200%, no text;

<ins>Text font family</ins> as one of the following: default, sans serif, serif, typewriter, decorative, user 1, user 2, user 3 (the corresponding font family names which are actually used can be defined in the user preferences);

Figure 4 illustrates the above choices for the non-color properties of a simple stylus. If a stylus is used in a particular way which does not make use of some of the properties (such as e.g. for a background which has neither an outline nor any text associated), they are left unspecified. On the other hand it is always possible to configure a stylus to selectively suppress outline, filling or text contents, by selecting the properties *no pen*, *no filling* and/or *no text*.

Note that the above display properties do not determine the character size of the text, they merely provide a relative text size factor. The actual text size is obtained by multiplying a text size parameter (provided by the mapper as a separate parameter) by this factor. This allows changing the overall size of the texts very easily via one single parameter, while maintaining the relative sizes specified in the used styli.

An *indexed stylus* consists of the specification of several sets of these display properties. An sequential index is associated with each set, starting with index 0. When used with a particular *index value*, it

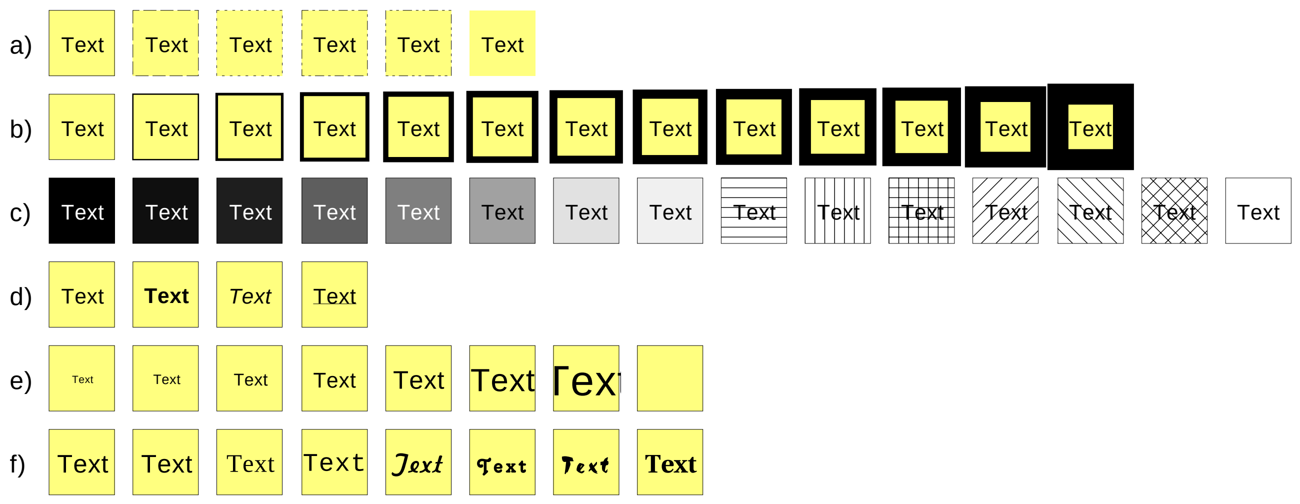


Figure 4: Non-color properties of a simple stylus: a) pen pattern, b) pen width, c) fill pattern, d) text attribute, e) text size factor, f) text font family

provides for a convenient way to specify a systematic variation of the display properties and, thus, can be looked at as a generalization of the *color index* used in EMME/2.

Index values outside the range of defined value sets are always referred to either the first or the last set.

As the index values are not limited to integer values, but may assume any floating point value, *interpolation* properties for pen, fill and text operations are associated with each specified stylus property set. Each interpolation property can be set to one of the following: *round to next lower*, *round to next higher*, *round to nearest* and *linear interpolation*. This information is used to determine which set of properties to use for fractional index values.

Linear interpolation implies that an interpolated color is used made up from the interpolated red, green and blue components of the colors specified in the next lower and next higher property sets. In case of a pen interpolation, the pen width is also interpolated in the same way. For all other (non-interpolatable) properties, the setting "linear interpolation" behaves the same as "round to nearest".

Indexed styli provide a very powerful tool to translate numerical values into discrete or continuous color sequences. As we shall see later, when looking at network plots, the index values are often provided by evaluating the so called *stylus index expression*.

The example in Figure 5 illustrates how to use an indexed stylus for generating gray-scale displays. In this example, an index value of zero corresponds to a black filling, a value of 1 to a white filling, values in between to the corresponding level of gray. In order to visually differentiate text and outline from the filling, they are colored white for fillings in black or dark gray, and black for fillings in light gray or white. The stylus used to produce Figure 5 uses only two indices, 0 and 1, which have the following properties:

Property:	Set 0:	Set 1:
Pen color:	white	black
Pen interpolation	nearest	*
Fill color:	black	white
Fill interpolation:	linear	*
Text color:	white	black
Text interpolation:	nearest	*

(*: not important, as the interpolation property of the last set is never used)

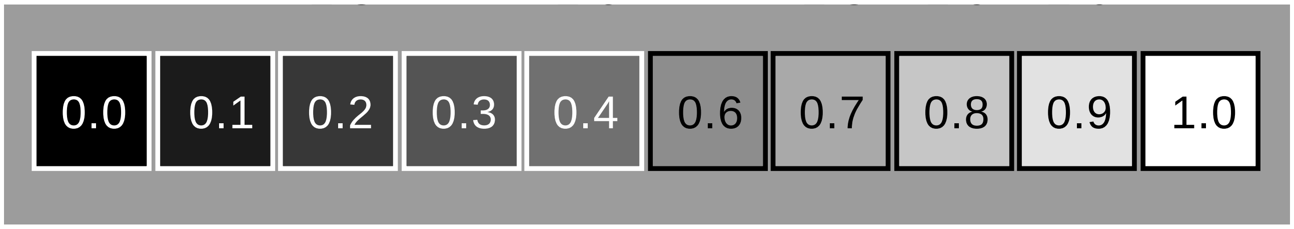


Figure 5: Gray-scale interpolation with an indexed stylus

A stylus can always be modified interactively by pushing the corresponding *stylus button*. This opens a popup menu which, depending on the context, provides options for all allowed modifications, such as changing of the properties, adding or removing indices, reversing the indices or copying/exchanging colors. Once a complex stylus is configured, it is also possible to associate a name with it and store it as a *predefined stylus* so that it can be recalled later on, whenever the same stylus configuration is needed again. The set of predefined styli becomes part of the user preferences.

Parameters and Configurable Objects

Enif is based to a large extent on user *configurable objects*, such as mappers, plots, lists or preferences (which are all to be explained later). The mechanism which provides configurability is implemented via a special class of objects called *parameters*.

A *parameter* is an object which is defined by the following properties:

- It is created and owned by a configurable object (owner)
- It has a name which is unique among the parameters belonging to the same owner.
- It is used to store one or several values of a given type, such as *flags*, *integer value*, *floating point value*, *string*, *text*, *expression*, *selector*, *stylus* or *bounding box*.
- If more than one value is stored in the same parameter, the different values are identified by a sequential index starting at 0.
- It emits a public signal whenever the parameter's value changes. This signal can be caught by other objects which need to be notified in order to adjust themselves to the new parameter value.
- If an optional group specification is defined for the parameter, it will automatically synchronize its values with all other parameters within the same scope which share the same group information.
- It can read from or write to an external file its value(s) and optional group information using a standard format.

At first sight these parameter properties may seem rather abstract. But as we shall see further on, it is this parameter concept which is responsible in large part for the flexibility of Enif.

In addition to storing simple numerical or string values, parameters are also used to store much more complex structures, such as the expressions, the selectors and the styli described in the previous sections. In the case of a stylus, each value corresponds to one complete set of display properties, so that an indexed stylus is implemented simply as a multi-valued parameter of the type stylus. The actual number of values of a parameter is determined at run time and can be changed dynamically.

By assigning the same group name to several parameters, the user can create logical groups of parameters, which automatically synchronize their values. So subsequently the user only needs to change the value of any one of the parameter in the group; all others will follow the change automatically. The usefulness of this grouping feature will become easier to understand later on when discussing the automatic synchronization of parameters between collaborating mappers.

Configurable objects organize their parameters in one or more *parameter lists*. Within a parameter list, the names of the parameters must always be unique.

The Mapper – Doing One Thing at a Time

A *mapper* is a user configurable object which knows how to graphically display one type of information in certain way. Taken alone, one single mapper will only be able to provide a very limited graphic display. As we shall see later on, the strength of the mappers does not lie in their stand-alone use, but in combining them to generate complex and flexible displays.

As each type of mapper is implemented individually, there is conceptually no limit to the number of different mappers nor the type of task they can perform. However, all mappers must adhere to the same rules when it comes to interfacing with the network plane.

A mapper consists of the following components, of which some are mandatory and some optional:

- A *set of parameters* which defines all user configurable aspects of the mapper. One of these parameters always contains a set of *flags*, some of them being generic (defined in the same way for all mappers) and some may be specific for the given mapper type. The parameters normally include at least one stylus which is used to define the display properties of the output generated by this mapper. In addition to simple numerical values, such as text size, link offset or attribute scale, the parameters often also include network element selectors and expressions.
- A *network plane painting routine* which can be called by the network plane whenever a part of the network plane must be displayed.
- A *mapper control dialog* which allows the user to interactively configure the mapper's parameters, or at least the more important ones. The visual appearance of the mapper control dialog is standardized and parameters of the same type always use the same standardized input fields.
- A *bounding box* specifying the rectangle on the network plane which this mapper will provide display output for.
- An optional *input event handler* which, if available and enabled, causes the mapper to take over mouse and keyboard, allowing the user to interact with the mapper directly on the network plane. Mappers providing this feature are called *input mappers*.
- An optional *legend provider* which sends legend information to the legend window.
- An optional *tip provider* which will display context sensitive tips (also known as "balloon messages") when the mouse rests on an object displayed by this mapper on the network plane view.
- An optional *set of active network elements*. This set of elements is usually defined by the mapper's selector(s). If available and enabled, this feature allows other mappers to identify the active network elements via special flag attributes (`is*Active`), and, e.g., to use this information in their own selector.

Besides the above components, a mapper is free to access other, usually external resources. In particular, mappers may access external files, e.g. to read in objects to be displayed on the network plane, such as annotations, bit-mapped images or polygons.

Each mapper provides the following *generic flags* to enable or disable the corresponding services provided by the network plane:

Flag:	Enables/disables the following feature:
Control	Visibility of mapper control dialog
Screen	Mapper displays information on network plane view on screen
Printer	Mapper displays information on printed output
Magnifier	Mapper display is visible on magnifier
Legend	Legend information is written into the legend window
Input	Mapper takes over mouse and keyboard events when it becomes active
Bounding Box	Mapper's bounding box is used to compute the full view bounding box of the network plane
Active Selection	Mapper's active elements will be made accessible to other mappers

Since the output of each mapper can be controlled individually to enable or disable graphic output to screen, printer and magnifier, it is possible to design plot configurations which appear differently on the different output devices. This has some interesting applications:

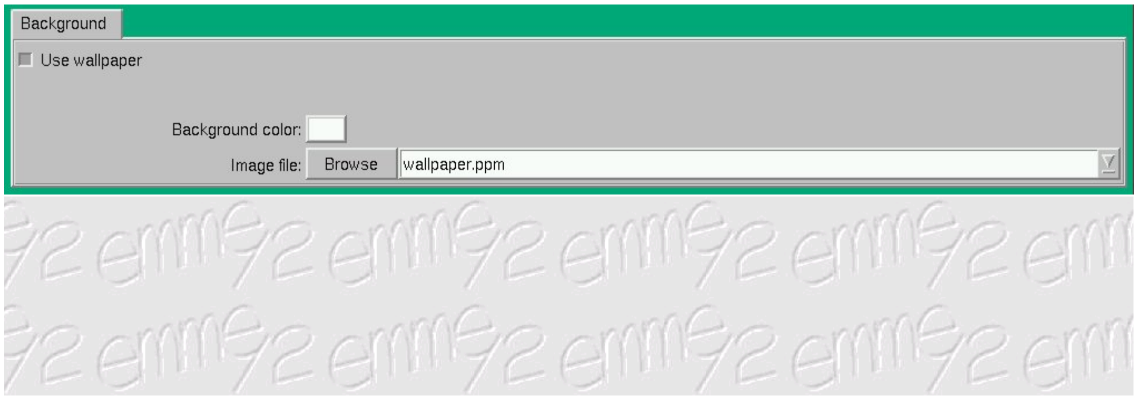
- Certain graphic items, such as e.g. a colored background, which are nice to see on the screen, but have negative effects when being printed (bad readability, increased consumption of ink or toner) may be disabled specifically for the printouts.
- Certain small displayed items (e.g. tiny texts), which are readable only on the printer and the magnifier (since these provide a higher resolution) may be disabled on the normal screen display.
- The magnifier feature is not limited to display the same information simply bigger, but may also be used to display additional information or even to instantaneously display a completely different view of the same objects.

An *active input mapper*, which receives the mouse and keyboard events occurring on the network plane, is visually distinguished by a different background color of the mapper control dialog. This color is user configurable. In the examples shown further down, a light green background is used for input mappers, instead of the default gray background for non-input mappers.

As this paper is not meant as a reference manual, there is no point going into the details of each of the mapper types currently available in Enif. Also, the currently implemented mappers are essentially limited to output-only mappers. There are a few network editing mappers which are still in their very early experimental phase and not of any practical use yet.

Thus, we shall limit the presentation of specific mapper types to simply illustrate some of them by means of depicting an example of their configuration in the control dialog and a small stripe of the network plane output generated using this configuration:

Figure:	Mapper:	Description:
Fig. 6	Background	Paints the background either in a constant color or using a tiled bit-mapped image.
Fig. 7	Grid	Generates a coordinate grid using a specified grid distance.
Fig. 8	Image	Displays a bit-mapped image file at a certain position scaled to a certain size. Many different graphic file formats (such as JPEG, PNG, TIFF, ...) are supported. The example shows a scanned image of a city map.
Fig. 9	Annotation	One or several EMME/2 annotation files are displayed.
Fig. 10	Node Box	Draws node boxes of a specified size and optionally writes the node numbers or node labels into them.
Fig. 11	Node Value	Evaluates one or several node values and represents them as a proportionally sized symbol and/or a text string. The example shows initial boardings and final alightings at transit stops.
Fig. 12	Node Polygon	Reads a file containing polygons and displays them. An indexed stylus can be used to color them according to arbitrary node characteristics. Optional node values can be evaluated and displayed either at the corresponding node positions or at the center of gravity of the polygons.
Fig. 13	Link Base	Draw link bases consisting of an arbitrary combination of a link bar, an I-node circle and a J-node circle.
Fig. 14	Link Value	Evaluates a link value and displays it numerically and/or as a proportional bar.
Fig. 15	Transit Line	Displays transit line itineraries and optionally annotates them with segment and/or stop values.
Fig. 16	Segment Value	Evaluates transit segment values, aggregates them to link values and displays these on the links as numeric values and/or as a proportional link bar.
Fig. 17	Intersection	Draws intersections and displays turn values as proportional width turns.
Fig. 18	Shortest Path	Computes shortest paths from or to one or several root nodes and displays the corresponding trees.
Fig. 19	Diagram	Several mappers are available to compute and display scattergrams and histograms for the various network element types. The example shown contains a link histogram with the distribution of vehicle miles by link speed.

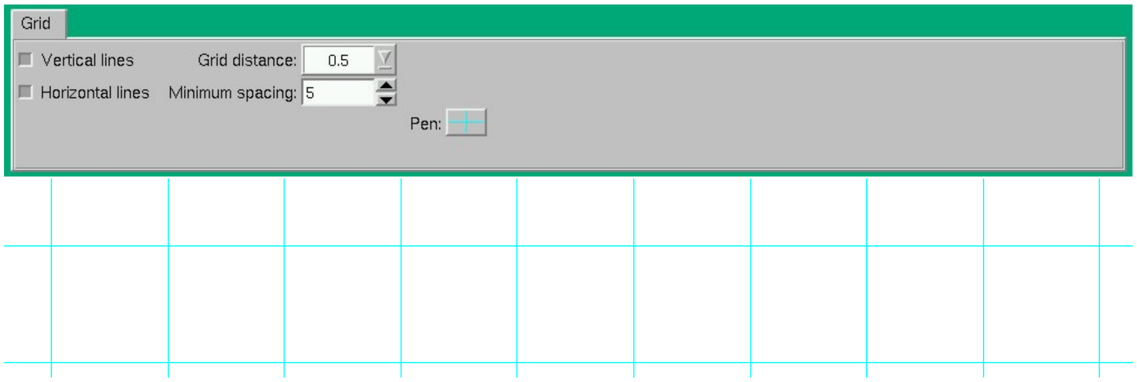

Background

Use wallpaper

Background color:

Image file: Browse wallpaper.ppm

Figure 6: Background Mapper

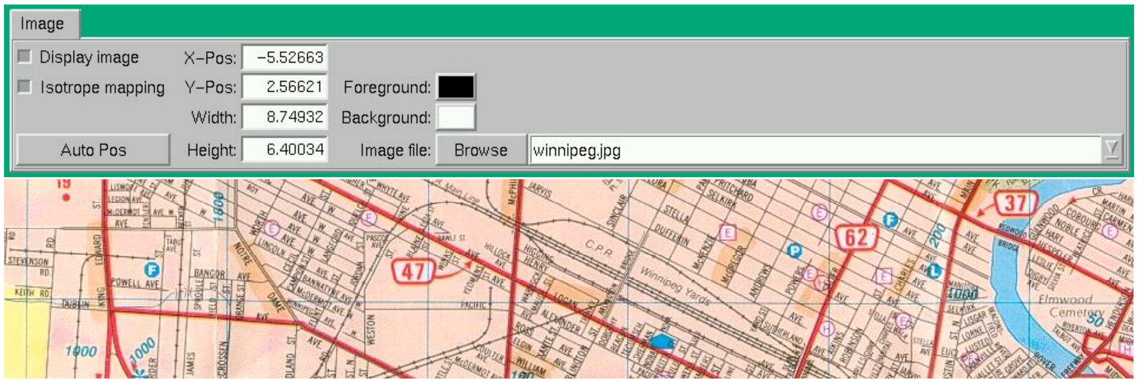

Grid

Vertical lines Grid distance: 0.5

Horizontal lines Minimum spacing: 5

Pen:

Figure 7: Grid Mapper



Image

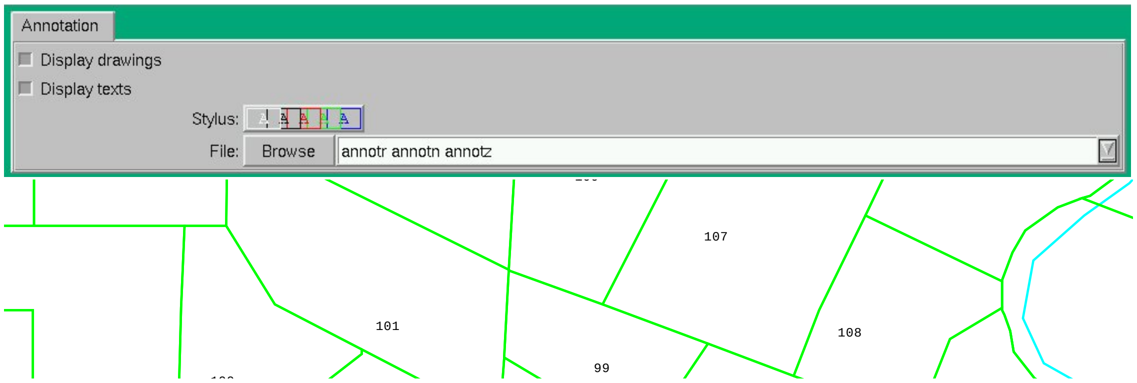
Display image X-Pos: −5.52663

Isotrope mapping Y-Pos: 2.56621 Foreground:

Width: 8.74932 Background:

Auto Pos Height: 6.40034 Image file: Browse winnipeg.jpg

Figure 8: Image Mapper


Annotation

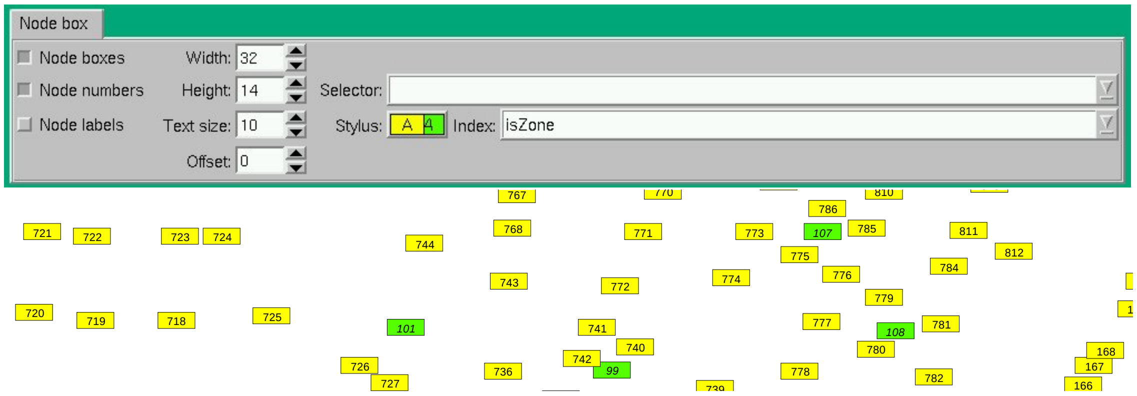
Display drawings

Display texts

Stylus:

File: Browse annotr annotn annotz

Figure 9: Annotation Mapper


Node box

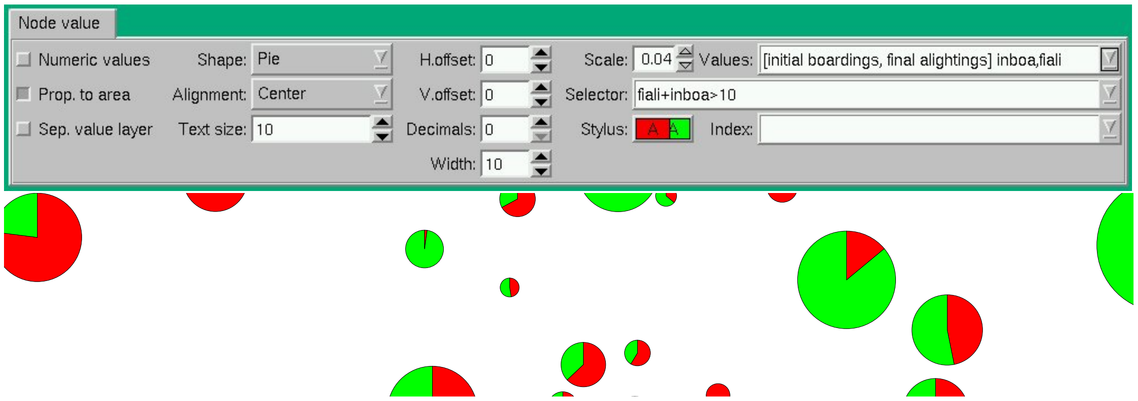
Node boxes Width: 32

Node numbers Height: 14 Selector:

Node labels Text size: 10 Stylus: Index: isZone

Offset: 0

Figure 10: Node Box Mapper


Node value

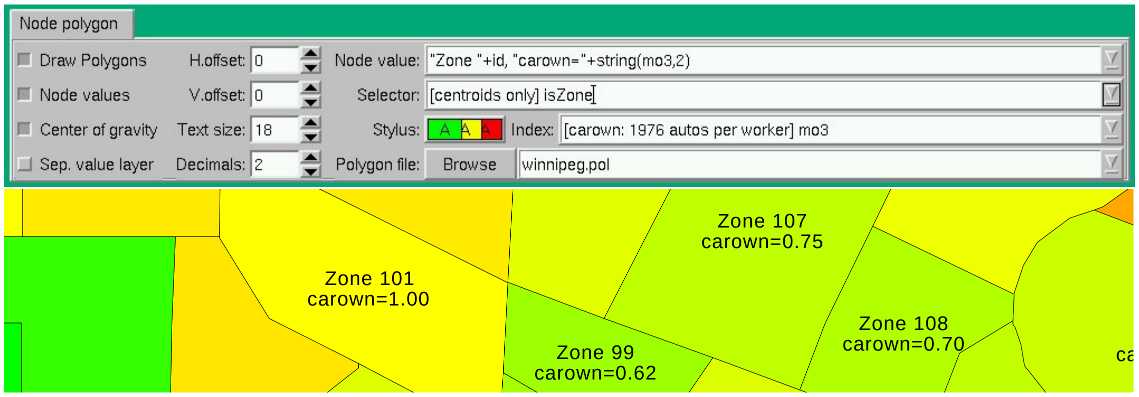
Numeric values Shape: Pie H.offset: 0 Scale: 0.04 Values: [initial boardings, final alightings] inboa,fiali

Prop. to area Alignment: Center V.offset: 0 Selector: fiali+inboa>10

Sep. value layer Text size: 10 Decimals: 0 Stylus: Index:

Width: 10

Figure 11: Node Value Mapper


Node polygon

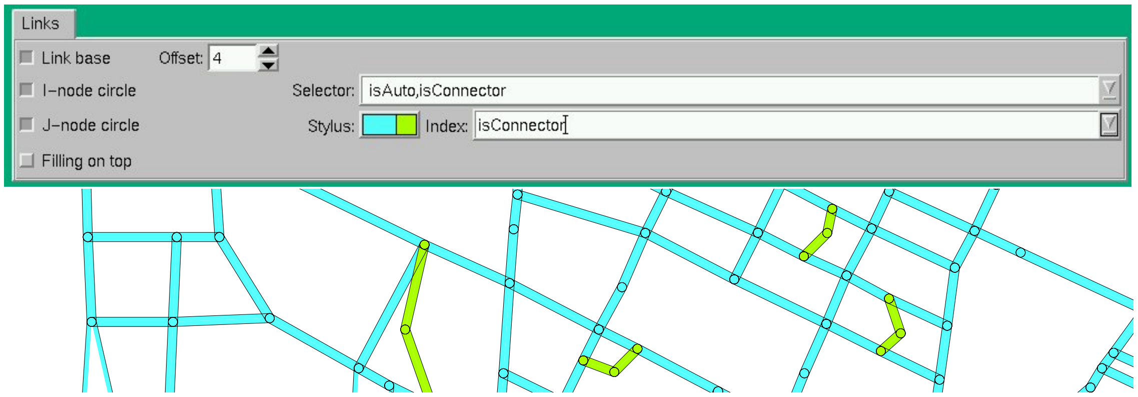
Draw Polygons H.offset: 0 Node value: "Zone "+id, "carown="+string(mo3,2)

Node values V.offset: 0 Selector: [centroids only] isZone

Center of gravity Text size: 18 Stylus: Index: [carown: 1976 autos per worker] mo3

Sep. value layer Decimals: 2 Polygon file: Browse winnipeg.pol

Figure 12: Node Polygon Mapper


Links

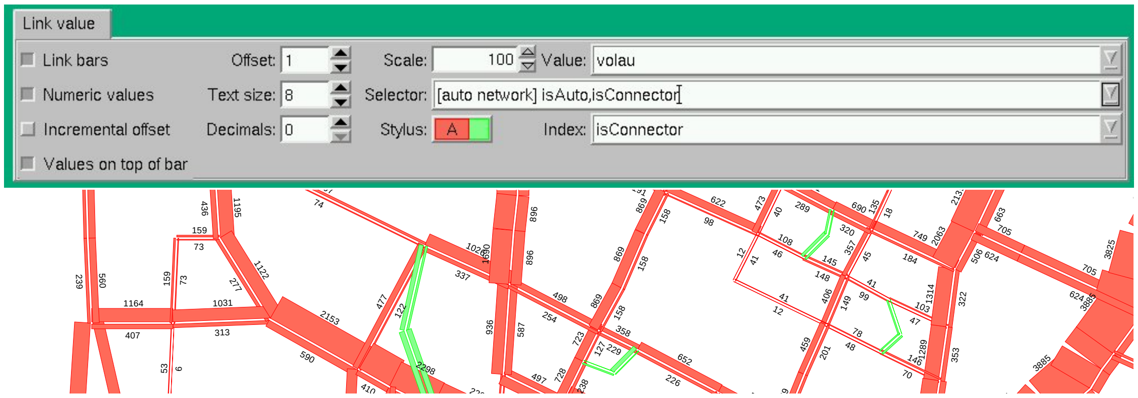
Link base Offset: 4

I–node circle Selector: isAuto,isConnector

J–node circle Stylus: Index: isConnector

Filling on top

Figure 13: Link Base Mapper



Link value

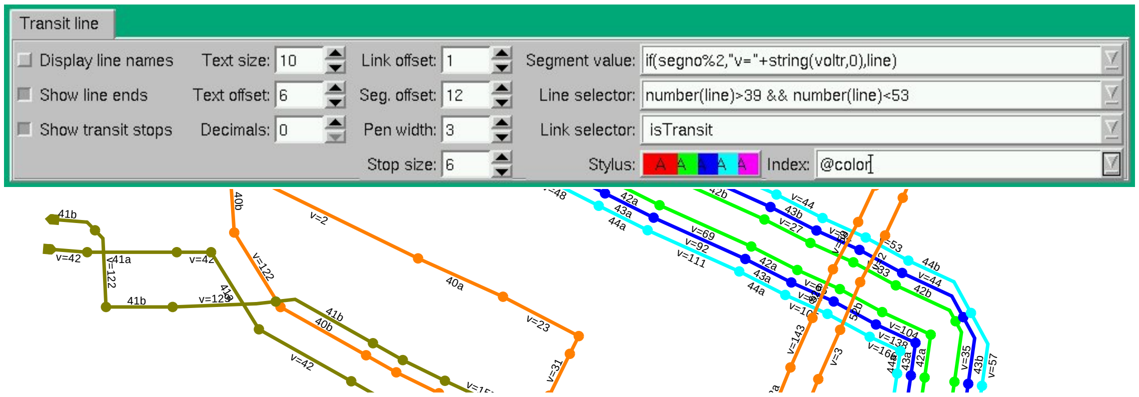
☐ Link bars Offset: 1 Scale: 100 Value: volau

☐ Numeric values Text size: 8 Selector: [auto network] isAuto,isConnector

☐ Incremental offset Decimals: 0 Stylus: 🟥🟩 Index: isConnector

☐ Values on top of bar

Figure 14: Link Value Mapper



Transit line

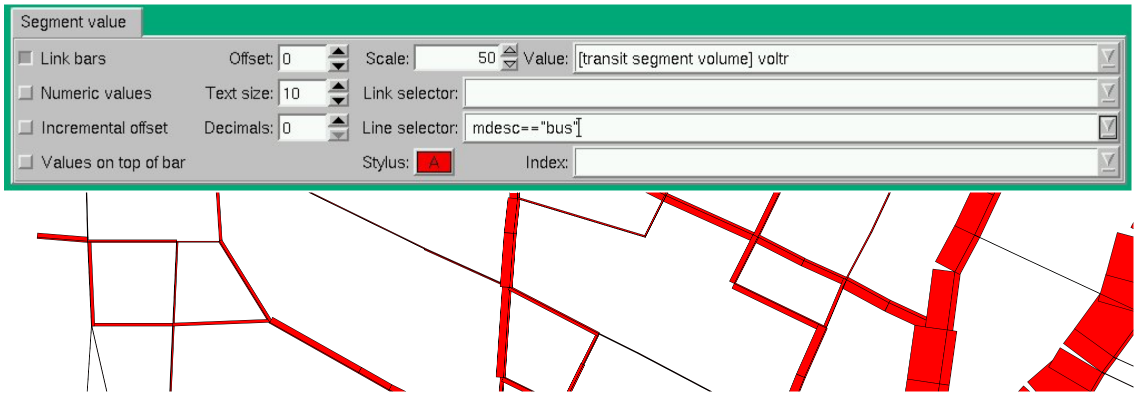
☐ Display line names Text size: 10 Link offset: 1 Segment value: if(segno%2,"v="+string(voltr,0),line)

☐ Show line ends Text offset: 6 Seg. offset: 12 Line selector: number(line)>39 && number(line)<53

☐ Show transit stops Decimals: 0 Pen width: 3 Link selector: isTransit

Stop size: 6 Stylus: 🟥🟩🟦🟦🟪 Index: @color

Figure 15: Transit Line Mapper



Segment value

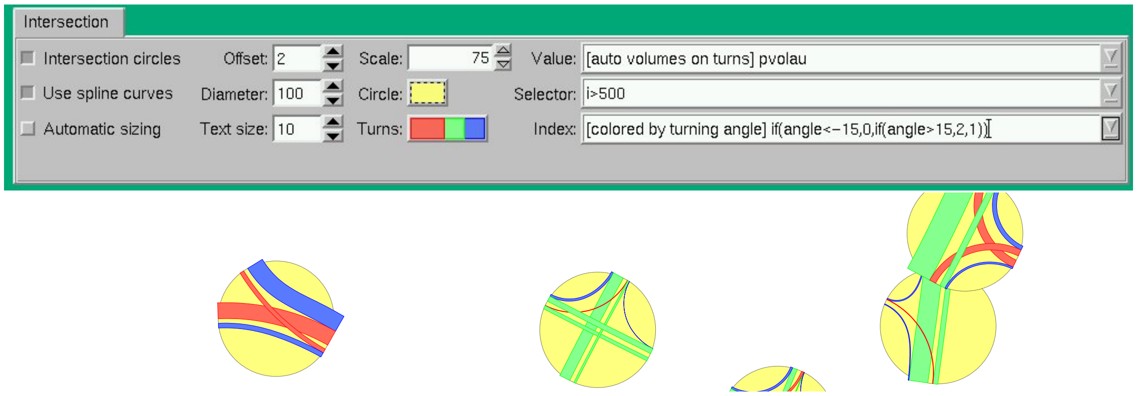
☐ Link bars Offset: 0 Scale: 50 Value: [transit segment volume] voltr

☐ Numeric values Text size: 10 Link selector:

☐ Incremental offset Decimals: 0 Line selector: mdesc=="bus"

☐ Values on top of bar Stylus: 🟥 Index:

Figure 16: Transit Segment Value Mapper



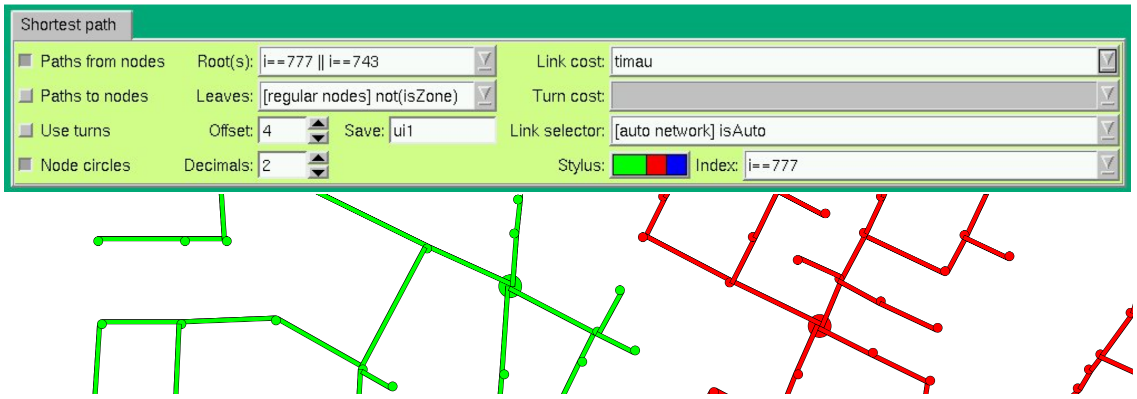
Intersection

☐ Intersection circles Offset: 2 Scale: 75 Value: [auto volumes on turns] pvolau

☐ Use spline curves Diameter: 100 Circle: 🟨 Selector: i>500

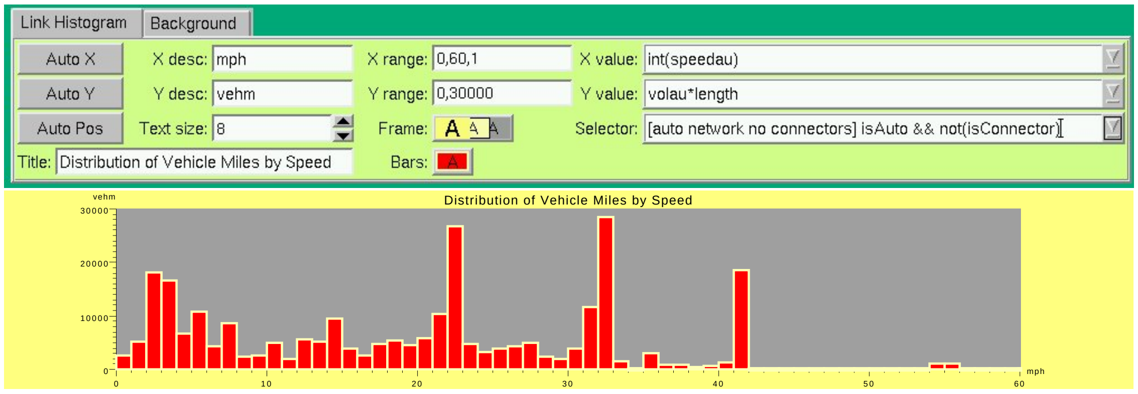
☐ Automatic sizing Text size: 10 Turns: 🟥🟩🟦 Index: [colored by turning angle] if(angle<−15,0,if(angle>15,2,1))

Figure 17: Intersection Mapper


Shortest path

Paths from nodes Root(s): i==777 || i==743 Link cost: timau
Paths to nodes Leaves: [regular nodes] not(isZone) Turn cost:
Use turns Offset: 4 Save: ui1 Link selector: [auto network] isAuto
Node circles Decimals: 2 Stylus: Index: i==777

Figure 18: Shortest Path Mapper


Link Histogram | Background

Auto X X desc: mph X range: 0,60,1 X value: int(speedau)
Auto Y Y desc: vehm Y range: 0,30000 Y value: volau*length
Auto Pos Text size: 8 Frame: Selector: [auto network no connectors] isAuto && not(isConnector)
Title: Distribution of Vehicle Miles by Speed Bars:

Figure 19: Diagram Mapper

Note that the above list of mapper types is by no means complete. It just shows those mappers which have already reached a stage of development which is close to production level. Some additional mappers are already under development now and we have already many more ideas for new mappers to be developed in the future.

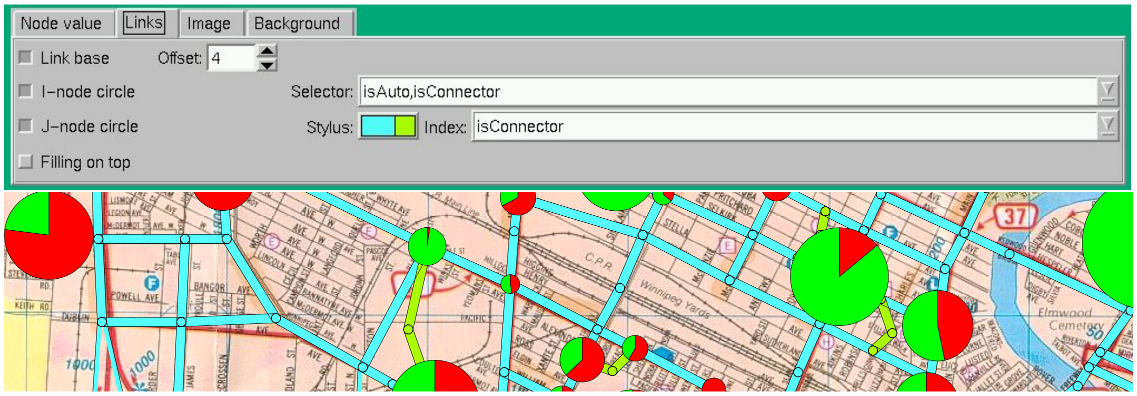
Combining Mappers to Obtain Plots

Now that we have looked at the mappers individually, it is time to combine them to the final product — the *plot*.

This is done by the creation of a *mapper stack*, which is an ordered list of configured mappers that are called one by one to generate the contents of the plot on the network plane. The mapper control dialogs of these mappers (except for those whose control windows are disabled by the corresponding mapper flag) are arranged in the form of a tabbed dialog, where the tabs are arranged from right to left, having the first drawn mapper (background) on the right, the last (foreground) on the left. This is illustrated in Figure 20, which combines some of the mappers taken from the examples of the previous section, in the following order:

- Background mapper from Figure 6
- Image mapper showing a scanned street map from Figure 8
- Link base mapper showing the base network from Figure 13
- Node value mapper showing initial boardings and final alightings from Figure 11

The parameters of a mapper can be accessed by clicking on the mapper's tab, which causes the corresponding mapper control dialog to be displayed. The mapper whose control dialog is currently


Node value | Links | Image | Background

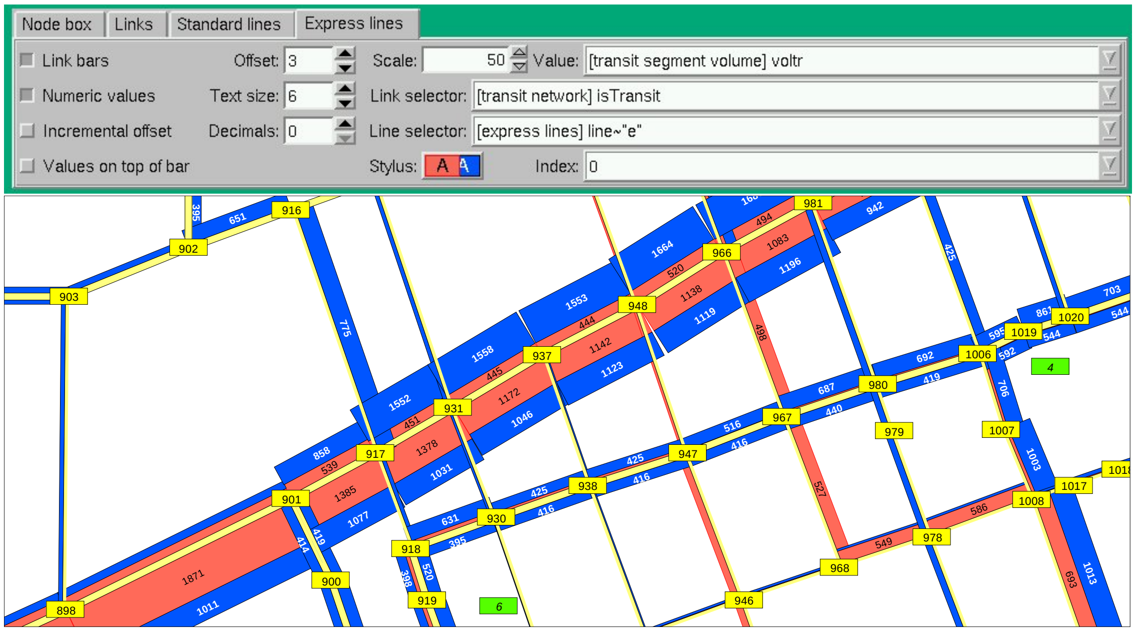
☐ Link base   Offset: 4
☐ I–node circle   Selector: isAuto,isConnector
☐ J–node circle   Stylus: 🟦🟩 Index: isConnector
☐ Filling on top

Figure 20: Combined output of several mappers

displayed, is called the *active mapper*. If this mapper happens to be an input mapper with its input flag enabled, it will take over the control of the mouse and keyboard events which occur on the network plane.

The parameter grouping mechanism, which was presented earlier, is an important feature when combining mappers. As each mapper has its own set of parameters, when combining several mappers to a plot, often some of these parameters correspond to the same logical value. E.g. the offset of a link value bar normally corresponds to the offset used for the link base, or, when displaying several volumes in a multi-layer band width plot, the same scale is usually used for all of them. Without an automatic synchronization of these values between the different mappers, the user would be obliged to change all these value separately for each mapper — which might become quite cumbersome, even for simple plots.

The parameter grouping allows associating an optional group name with each parameter. Subsequently, when a parameter value in one mapper is changed, the change is signaled to all other mappers, which in turn propagate the signal to their own parameter list, causing parameters with the same group name (and compatible value type) to be updated.


Node box | Links | Standard lines | Express lines

☐ Link bars   Offset: 3   Scale: 50   Value: [transit segment volume] voltr
☐ Numeric values   Text size: 6   Link selector: [transit network] isTransit
☐ Incremental offset   Decimals: 0   Line selector: [express lines] line~"e"
☐ Values on top of bar   Stylus: A A   Index: 0

Figure 21: Transit volumes for express and standard busses

An example using parameter grouping is illustrated in Figure 21. This plot is defined as the combination of five mappers (from back to front): simple white background mapper with its control dialog disabled, transit segment value mapper for the volumes of the express bus lines, transit segment value mapper for the volumes of the standard bus lines, link base mapper and node box mapper. The following parameter groupings are used in this plot configuration:

- text size for express and standard bus volumes
- number of decimals for express and standard bus volumes
- scale for express and standard bus volumes
- link offset of link base and express bus volumes
- link selector of link base, standard and express bus volumes
- stylus for express and standard bus volumes (express bus volumes use index 0, standard bus volumes use index 1)

With this grouping, the user can change any of the above parameters for any one of the applicable mappers and the other parameters in the same group will follow the change automatically.

A *mapper control manager* implements all the functionality needed to create and edit the mapper stack. It provides the following operations:

- *add* a new (unconfigured) mapper to the mapper stack
- *delete* a mapper from the stack
- *duplicate* an existing mapper including its complete configuration
- change a mapper's position within the stack by *moving* it up or down
- set or unset the *generic flags* of a mapper
- *edit* a mapper's parameter values and/or *group specifications*
- read one or several configured mappers from a configuration file and *include* them into the stack

A *plot configuration* (or often just short "*plot*") is defined by a stack of fully configured mappers and some additional parameters related directly to the plot, such as plot flags, name, description, caption and icon text and the name of the active mapper.

Note that a plot configuration is not directly dependent on a particular EMME/2 application or scenario, but can be used with any network. When a new network is loaded, e.g. after switching to a different scenario, the mappers automatically resynchronize their network dependent parameters. Of course, if mappers access user defined network attributes, these have to be defined in the new network, otherwise the expressions will be signaled as invalid.

The current plot configuration can be *saved* to a file and *reloaded* again later on, whenever the user wishes to produce the same type of plot again. In addition, plot configuration files can be *registered* in the users preferences, so that they will be accessible directly on the Enif "Plot" menu by their name.

In the remaining part of this section, the possibilities of Enif are illustrated by some example plots which were produced using the standard Winnipeg EMME/2 demonstration data bank.

Fig. 22	A simple plot composed only of a background mapper, a node polygon mapper and some legend texts. It uses an indexed stylus with linearly interpolated fill colors to produce a zone map showing the different levels of car ownership in different colors.
Fig. 23	This plot is composed of a satellite picture in the background on which are overlaid an annotation of the rivers (they don't really fit the satellite image to well...) and a "street" network layer composed of several link base mappers.
Fig. 24	A detailed auto volume plot which shows link and turn volumes.
Fig. 25	A typical bandwidth plot showing the auto, transit and auxiliary transit assignment results as a multi-layer volume plot.
Fig. 26	This plot shows the distribution of vehicle miles traveled by auto speed. An indexed stylus using a linear color interpolation is used to color the bars by speed from red (very slow) to dark green (very fast). In order to visualize where the different speeds occur on the network, a small network plot is superimposed on the top right corner by means of a link base mapper using the same indexed stylus.

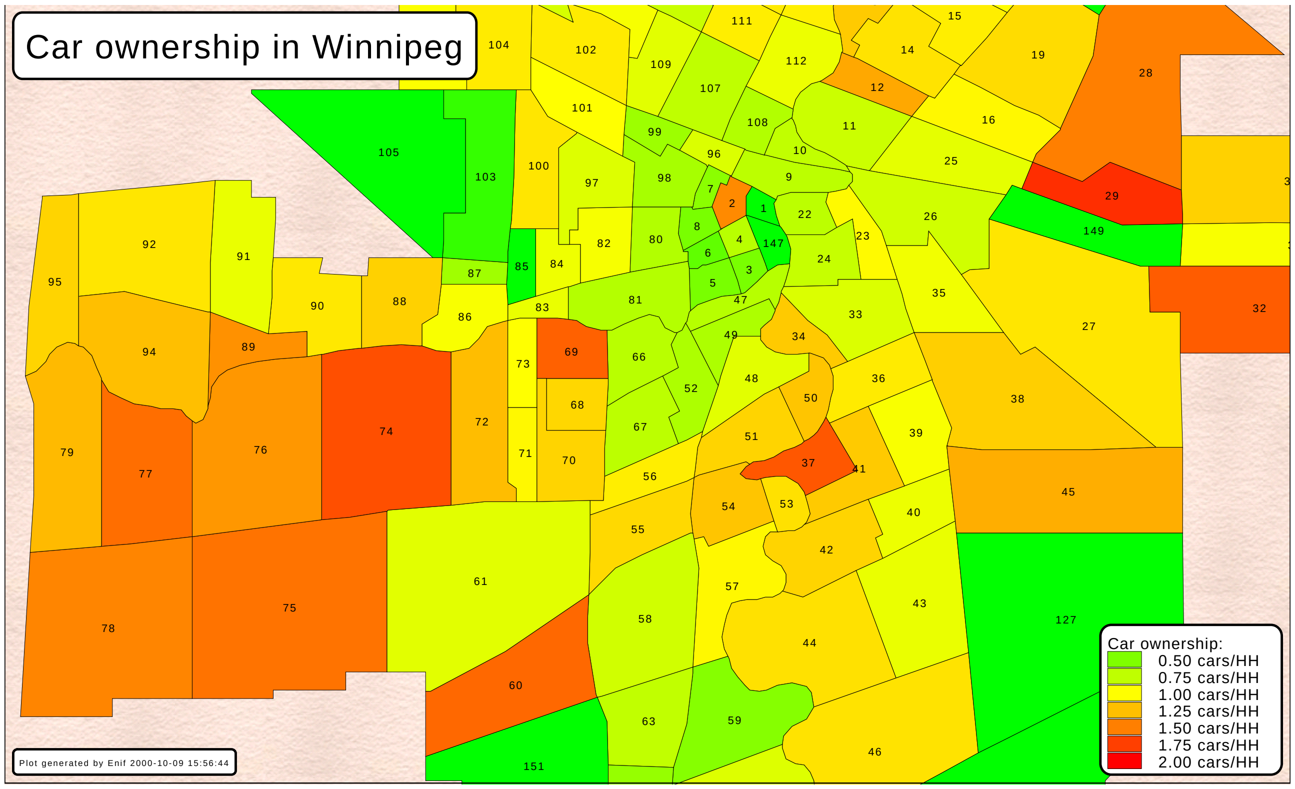


Figure 22: Car ownership on zone polygons using an indexed stylus

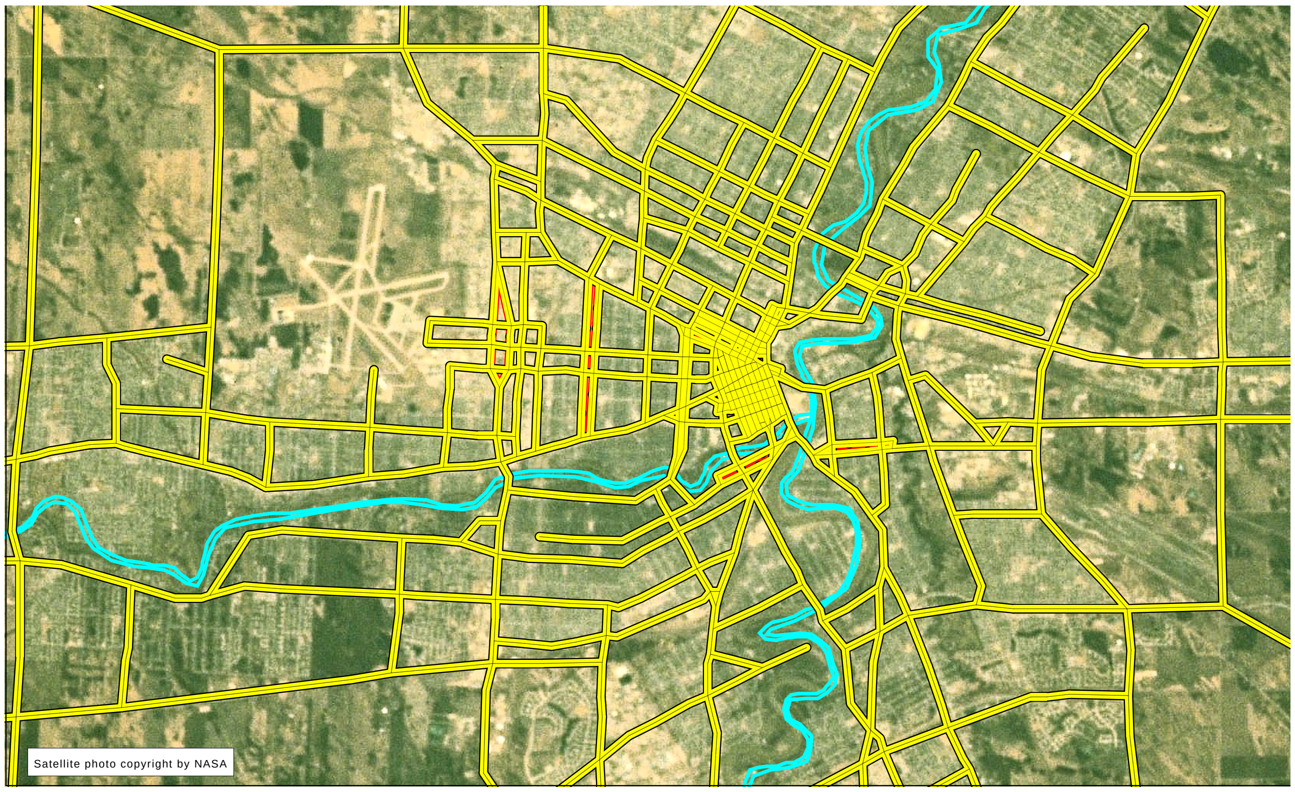


Figure 23: Street network on top of satellite image

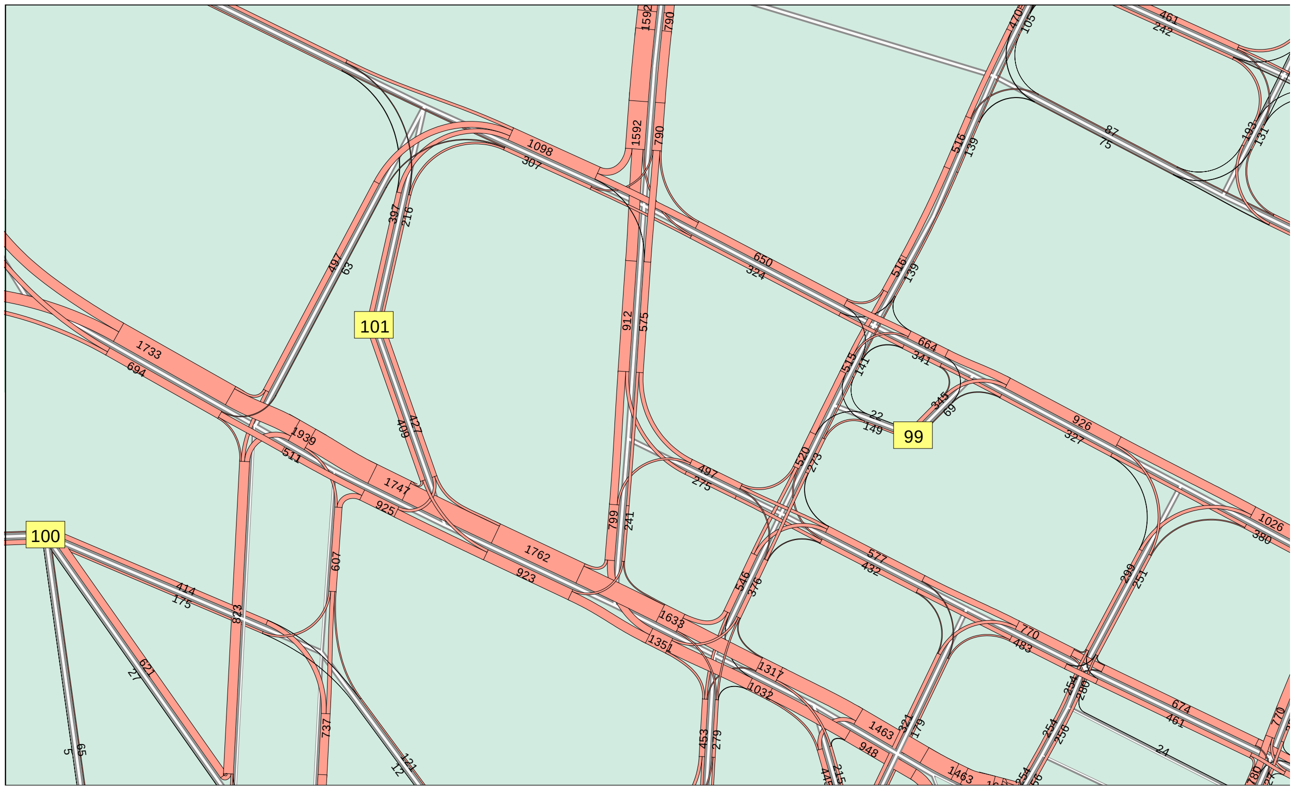


Figure 24: Link and turning volumes on auto network

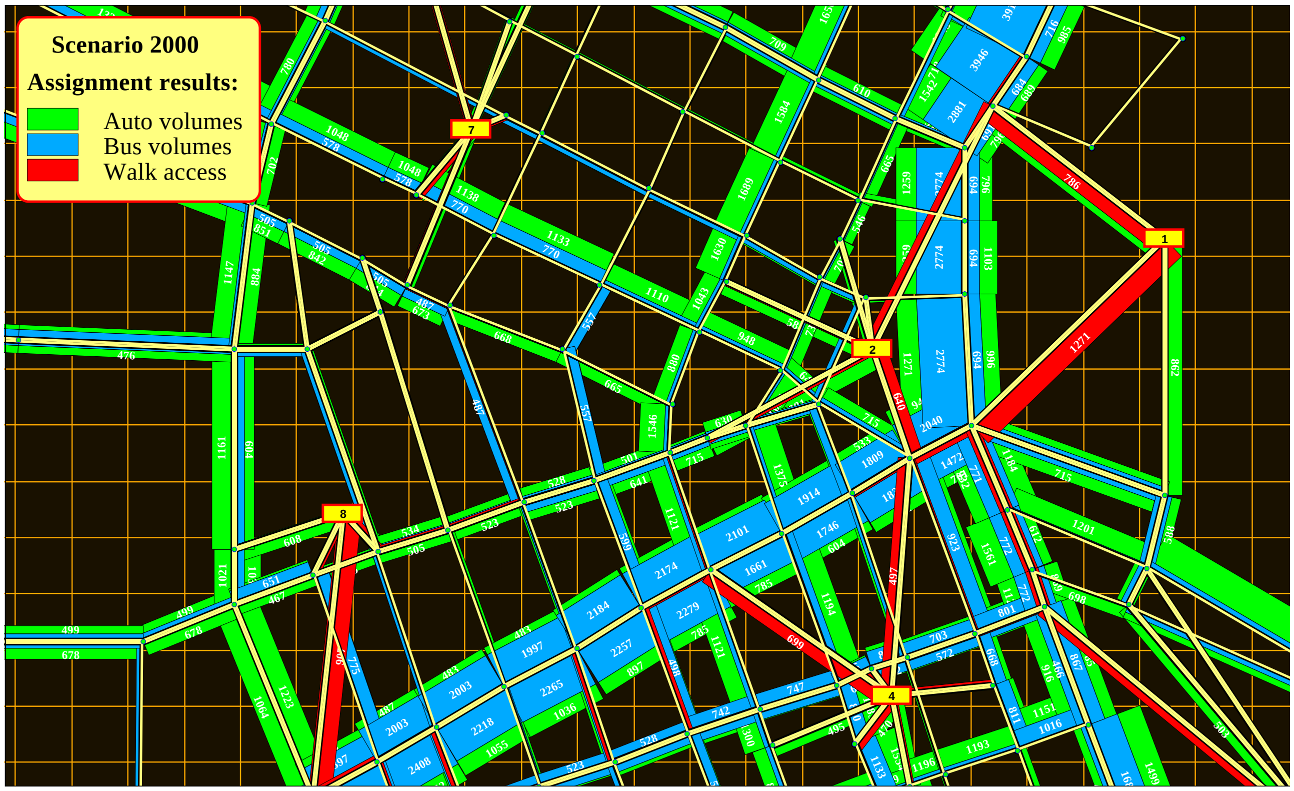


Figure 25: Auto, bus and walk volumes in the CBD

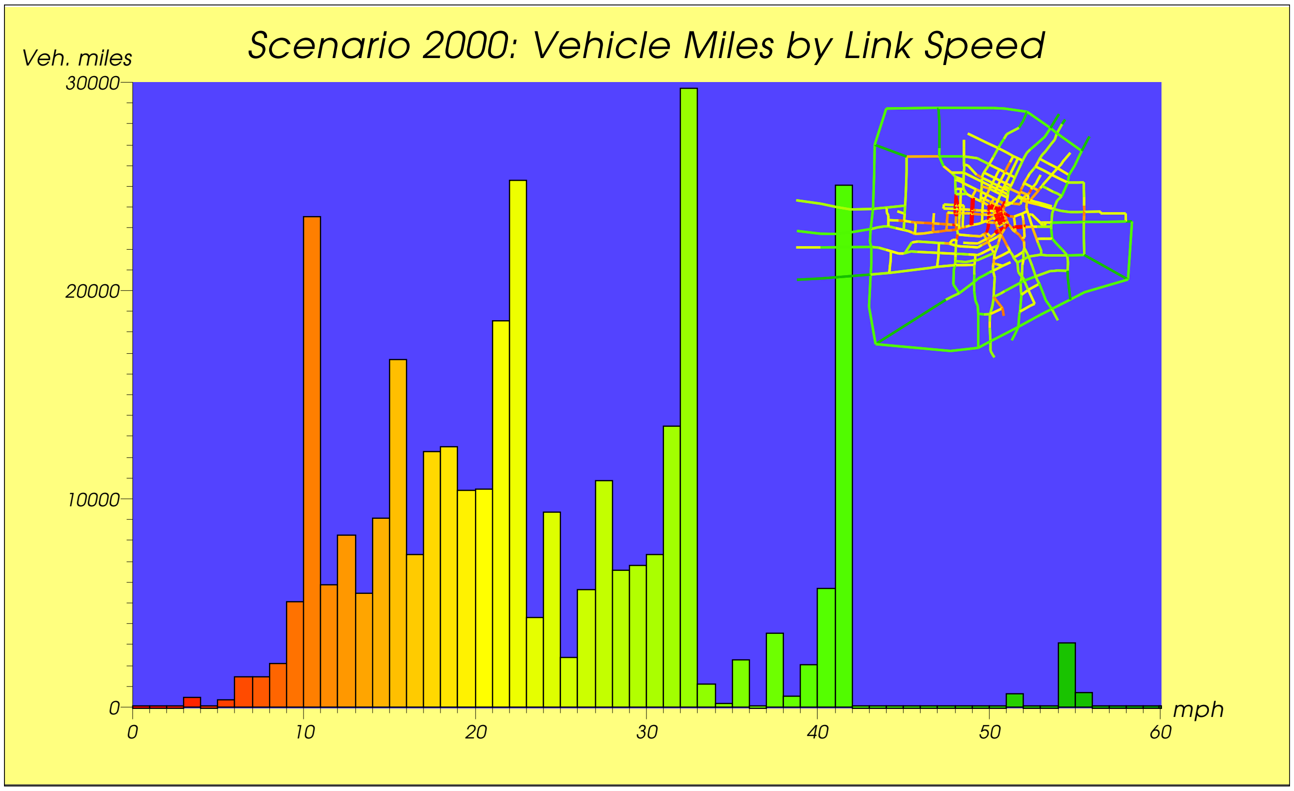


Figure 26: Integrating a network plot into a diagram

Lists

After having looked at Enif's graphic capabilities to produce network plots and diagrams, it should not be forgotten that often text-only type output is just as important for doing the day-to-day work in transportation planning.

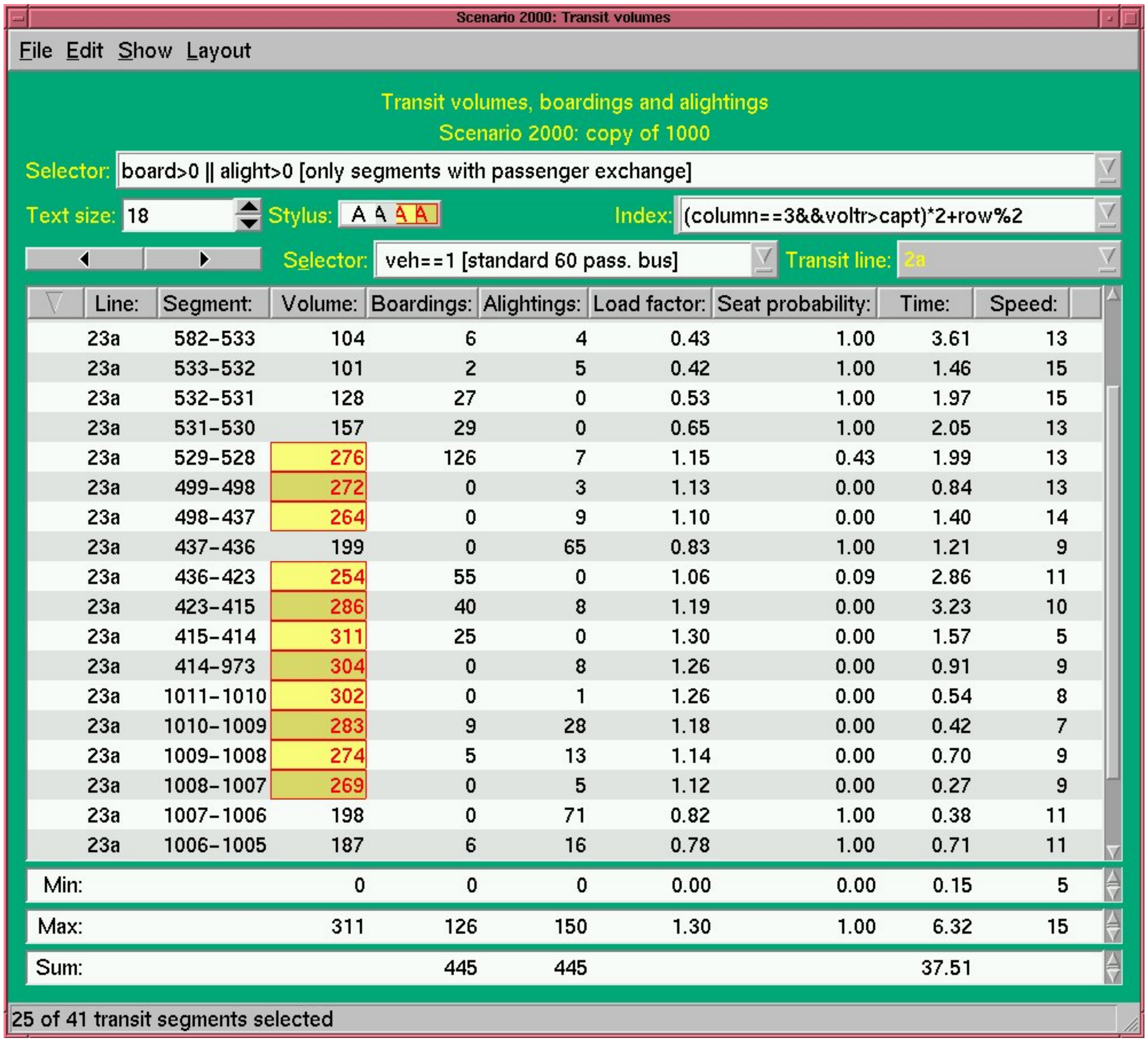
This aspect is covered in Enif's concept of *network lists*. A network list is a user configurable object which displays textual and numerical information in tabular form for a set of network elements. Its implementation relies on the same basic concepts that were already discussed earlier: attributes, expressions, selectors, parameters and stylus. Using the "Enif terminology" which we have introduced in the previous sections, a network list can be described as an object consisting of the following components:

- Network element type: Defines the type of network elements that can be displayed by this list.
- Element selector: Defines the subset of network elements to be included in the list and, using the optional iterator expression, the order in which the selected elements appear in the list.
- Column attribute list: An attribute list containing one attribute for each column of the list. Each of these attributes is either pointing directly to one of the existing network attributes or is defined by a column expression to be evaluated "on-the-fly" when a list value is to be displayed.
- Column formatting information: Besides the usual formatting (field width, decimal precision, alignment), the value field can optionally be delimited by top, bottom, left and/or right rules, and for each column the pertinent summary information (min, max, sum, average) can be specified.
- Indexed stylus: Defines the visual appearance of each value field in the list. The stylus index is an expression of the corresponding network element type which is evaluated for each list value. The special constant attributes `row` and `column` can be used to make the stylus index depend on the row and/or column index of the displayed value field. Each list value is displayed using the resulting stylus properties by filling background, writing the value string and, finally, drawing the activated rules around the value field.

- List summaries: At the end of the list, optional minimum, maximum, sum and/or average summary rows display the requested column summary information.
- Other parameters: Similar as for plots, for each list configuration a name, a description, a window caption and an icon text can be specified.
- Optional "by-element" selector: A special variant of lists are the so called *"by element"-lists*. These are lists which display a set of network elements which is determined *by* a related network element, such as e.g. "O-D pairs by origin", "transit segments by line" or "turns by intersection". In this case, an additional by-element selector specifies the subset and the order of the available by-elements. The current by-element can be chosen directly from the subset or stepped through consecutively in forward or backward direction.

The contents of a list can be sent to a printer or punched to a file. Network elements can be marked on the list and the marked elements can be temporarily hidden or copied to the clipboard for pasting into other applications.

Figure 27 shows an example of a screen shot of a list containing assignment results for the segments of a particular transit line. Note the use of the stylus to produce a "zebra" list background and to highlight the volumes of overcrowded segments.



Transit volumes, boardings and alightings
Scenario 2000: copy of 1000

Selector: board>0 || alight>0 [only segments with passenger exchange]

Text size: 18 Stylus: A A A A Index: (column==3&&voltr>capt)*2+row%2

Selector: veh==1 [standard 60 pass. bus] Transit line: 2a

Line:	Segment:	Volume:	Boardings:	Alightings:	Load factor:	Seat probability:	Time:	Speed:
23a	582–533	104	6	4	0.43	1.00	3.61	13
23a	533–532	101	2	5	0.42	1.00	1.46	15
23a	532–531	128	27	0	0.53	1.00	1.97	15
23a	531–530	157	29	0	0.65	1.00	2.05	13
23a	529–528	276	126	7	1.15	0.43	1.99	13
23a	499–498	272	0	3	1.13	0.00	0.84	13
23a	498–437	264	0	9	1.10	0.00	1.40	14
23a	437–436	199	0	65	0.83	1.00	1.21	9
23a	436–423	254	55	0	1.06	0.09	2.86	11
23a	423–415	286	40	8	1.19	0.00	3.23	10
23a	415–414	311	25	0	1.30	0.00	1.57	5
23a	414–973	304	0	8	1.26	0.00	0.91	9
23a	1011–1010	302	0	1	1.26	0.00	0.54	8
23a	1010–1009	283	9	28	1.18	0.00	0.42	7
23a	1009–1008	274	5	13	1.14	0.00	0.70	9
23a	1008–1007	269	0	5	1.12	0.00	0.27	9
23a	1007–1006	198	0	71	0.82	1.00	0.38	11
23a	1006–1005	187	6	16	0.78	1.00	0.71	11
Min:		0	0	0	0.00	0.00	0.15	5
Max:		311	126	150	1.30	1.00	6.32	15
Sum:			445	445			37.51	

25 of 41 transit segments selected

Figure 27: Example of a list as it appears on screen

Preferences and Initialization File

Configuration settings which pertain to running the Enif program as such (i.e. not related to the configuration of plots or lists) are handled by a configurable object we refer to as the *preferences*. The preferences consist of a set of parameters which are divided into the following groups:

- General parameters containing information about the default data bank, the default startup plot configuration or the diagnostic log file.
- Layout parameters which define the screen appearance of the Enif program.
- Font parameters which allow the customization of the fonts used by Enif.
- Stylus parameters containing the system defined styli and an open set of user defined styli.
- Network caching parameters which are used to determine which network attributes are put into the network element's data buffers and which are read "on-the-fly" from the EMME/2 data bank when needed.
- Plot related parameters controlling the features of the network plane (e.g. magnifier size and factor) and defining whether the network uses right hand or left hand traffic.
- Print parameters are used to setup the printing characteristics used for plots and lists.
- Predefined views allow the user to store and recall network windows to easily access a certain part of the network plane by a logical name.
- Registered plot and list configurations are used to define which plot and list configurations can be accessed directly from Enif's "Plot" and "List" menus.

All system defined parameters are organized in a single parameter list. Separate parameter lists are used to contain each of the following types of user defined parameters: predefined views, registered plot configurations, registered list configurations and user defined styli. This allows each parameter type to have its own name space, so that no conflict occurs if e.g. a predefined view uses the same name as a registered list.

The values of all preference parameters can be saved to an *initialization file*, so that they can be used as starting values for subsequent Enif sessions. By default the initialization file `enif.ini` is used, but it is also possible to specify an alternate initialization file on the Enif command line using the `-i` option.

Implementation

Since EMME/2 is running both on Microsoft Windows based PC and under various flavors of Unix, an important requirement for Enif is that it, too, will be running under both types of operating environments.

This goal is accomplished by programming Enif in the C++ programming language (4) and basing it on the commercial version of *Qt*(5), a C++ cross-platform GUI application framework developed by TrollTech AB in Olso, Norway. Qt not only provides a platform independent method to access the different windowing systems, but, in addition, it features a very powerful concept of signals and slots. The latter are also extensively used in Enif to implement the communication and synchronization between non-graphic objects, such as e.g. in the grouping mechanism for parameters. Another important advantage of Qt is the fact that it is distributed as source code. This eliminates the risks of having Enif depend on "black box" type precompiled components beyond our control.

The generic language used in Enif is English — just as is the case for EMME/2. However, Qt provides an extensive support for internationalization which makes it possible to use Enif in any other language by providing the proper translation files. At the moment, a French translation file is available for Enif to demonstrate this mutli-lingual capability.

Conclusions

This article presented a very first look at Enif, a new software aimed at accessing existing EMME/2 data banks by means of a consistent modern graphical user interface.

The development of Enif is still in an early stage. Most of the work done so far was concentrated on developing a broad and consistent conceptual base and implementing the corresponding basic tools and mechanisms. Based on this, initial functionalities to provide read-only access to EMME/2 data banks for producing plots and lists have been implemented. This current implementation allows us to demonstrate that the goals set forth at the beginning of the project can indeed be reached.

A lot of work remains to be done in order to build all the desired functionality into Enif. However, even the limited functionality which is already implemented today —generating plots and lists— is on its own a worth while addition to EMME/2. It will help to overcome the drawbacks of EMME/2's "old" graphic interface and ease the task of transforming the computational results of EMME/2 model runs into high-quality graphical output, both for interactive work on screen and for presenting results graphically in reports. For this reason, we intend to release a first version of Enif to all EMME/2 users as soon as our internal tests has proven the code to be robust enough for general distribution. This version will essentially contain the functionality presented in this article, i.e. it will allow read-only access to EMME/2 data banks for generating plots and lists, but it will not include any possibilities to modify the contents of the data bank. In a second phase, we then plan to gradually add network editing features to Enif. Building transportation modeling capabilities directly into Enif may be considered a desirable long term goal, but it is much too early now for making any commitments in this direction.

Even with the advent of Enif, the traditional EMME/2 modules will continue to keep their importance, as they will remain responsible for all tasks associated with the actual modeling of transportation networks. Thus, Enif should by no means be looked at as a *replacement* for EMME/2, but as a —hopefully very useful— *complement* to EMME/2!

References

(1) Babin A., Florian M., James-Lefèbvre L., Spiess H. (1982). EMME/2: Interactive graphic method for road and transportation planning. *Transportation Research Records* **866**, 1-9.

(2) INRO Consultants, Inc. (1999). EMME/2 User's Manual, Release 9.0.

(3) Spiess H. (1984). *Contributions à la théorie et aux outils de planification de réseaux de transport urbain.* Ph.D. thesis, Département d'informatique et de recherche opérationnelle, Centre de recherche sur les transports, Université de Montréal, Publication 382.

(4) Stroustrup B. (1999). *The C++ Programming Language.* Third Edition. Addison-Wesley, ISBN 0201889544.

(5) TrollTech AB (2000). *QT: The Official Documentation.* New Riders Publishing, ISBN 1578702097.