Tela User's Guide

Pekka Janhunen, tela-suggestions@fmi.fi, tela-bugs@fmi.fi v1.21, 13 February 1995

Contents

1 Introduction

Tela is a numerical computing environment mainly targeted for numerical simulation pre- and postprocessing work. The language supports arbitrary rank arrays (hence the name, Tensor Language), and has powerful programming capabilities including F90 style vector notation.

The on-line help is intended to be the primary source of documentation. The command **help** (??) gives help of reserved words, intrinsic functions, standard functions, and user functions. The function docview() can be used to view the same documentation in plain ASCII form, in TeX DVI form, and in Hypertext Markup Language (HTML) form. Right from the beginning you should be familiar with the **help** command, and learn to browse the HTML documentation. If you want a printed "manual", you can convert the DVI files into PostScript using the standard TeX tool **dvips** and print them.

The file you are now reading is the User's Guide. Other relevant document files include *telahelp* (`html:telahelp.html`), *telafuncs* (`html:telafuncs.html`) and *telafuncsSectioned* (`html:telafuncsSectioned.html`). The **help** (??) can retrieve the equivalent information.

(For Xmosaic viewers only: If the previous links do not work, you can try the universal links to the FMI/GEO WWW site: *telahelp.html* (`http://www.geo.fmi.fi/prog/tela/telahelp.html`), *telafuncs.html* (`http://www.geo.fmi.fi/prog/tela/telafuncs.html`) and *telafuncsSectioned.html* (`http://www.geo.fmi.fi/prog/tela/telafuncsSectioned.html`)).

2 Simple interactive use

When you start Tela from the Unix prompt without arguments (usually by typing **tela**), the system enters interactive mode after reading some initialization files and is ready to parse and execute command lines. The Tela prompt is "> ". The GNU Readline library is the first layer interacting with the user. This library makes it easy to modify and reexecute previous commands. The most important key combinations are (Ctrl means keeping Control down when pressing the key):

```
Ctrl-P      Get Previous line
Ctrl-N      Get Next line
Ctrl-F      Forward cursor one character position
Ctrl-B      Backward cursor one character position

Ctrl-A      Set cursor position at beginning of line
Ctrl-E      Set cursor position at End of line
Ctrl-K      Delete (Kill) text beyond cursor position

TAB         Try to complete reserved word, identifier or filename
Enter       Execute command line (cursor need not be at end position)
```

```
Ctrl-C        Interrupt command during command execution
Ctrl-D        Quit Tela (must be alone in line)
```

You have probably used other software which also uses the GNU Readline library, e.g. the GNU debugger **gdb**. If your terminal is properly configured, you can use the arrow keys instead of Ctrl-P, Ctrl-N, Ctrl-F and Ctrl-B to move in the history buffer. You should get familiar with these commands as soon as possible, since it will greatly speed up your working with Tela. The command completion is able to complete Tela reserved words, function names (either intrinsic, standard, or user-defined), variables names and filenames. Filename completion is invoked if the TAB key is pressed inside a string ("..."). This can be useful with the **source** command, for example.

Tela uses the X window system to display graphics. The graphics is displayed by a separate program, named PlotMTV, but calling it is transparent to the user. The DISPLAY environment variable must be properly defined, otherwise the graphics will not work.

Try entering the following commands first ("> " means the Tela prompt which you obviously do not type):

```
> x=0:0.1:4*pi
> plot(x,sin(x))
```

The first command creates a vector of real numbers ranging from 0 to 4 times Pi with step 0.1 and assigns the vector to variables x. The second command then plots sin(x) versus x. A graph of sin(x) should appear in a new PlotMTV window, if not, the system is not properly installed.

In Matlab, an expression is printed if there is no semicolon at the end of command line. In Tela, however, an expression is printed if it is not assigned to any variable, regardless of any semicolon. In fact, Tela internally inserts a semicolon at the end of every command line before actually parsing it. The insertion is not done when reading commands from a file.

This leads us to the next topic: how to execute commands from a file. Type

```
> source("demo")
```

A menu of choices appears, but let us first explain what the **source** command does. It looks for file named "demo.t" in the current TELAPATH (if the file name contains a period, the ".t" is not appended). The TELAPATH is a list of directories where input files are searched, similar in function to the Unix PATH. The file "demo.t" is found in some standard directory and commands in it are executed. The current TELAPATH can be seen by giving the command **telapath()**.

When reading commands from a t-file, there should be a semicolon at the end of each statement (or actually separating the consecutive statements, but the difference is unessential here).

OK, now play with the demos. Press a number and ENTER to the menus, to get back to command level select Quit.

The typical and recommended way to use Tela when developing something bigger than one line is the following: Open an editor window and a shell (Xterm) executing Tela on your workstation. Create a t-file, for example "mine.t" with the editor and write some Tela commands in it. Remember to use semicolons as statement separators. Try it out by typing **source("mine.t")** in the Tela window. You must have started Tela from the same directory where "mine.t" lies, or you can use the Tela **cd** command to change to that directory. (See **help cd**.) Doing this repeatedly you only need save the file from the editor and press Ctrl-P ENTER in Tela window to execute the file with changes you just made. This is only slightly clumsier than program development with Turbo Pascal in the old times: instead of a single keypress you now have to press three keys and switch the window with the mouse once to execute the program!

3 Basics of the language

The first thing to be mentioned about the Tela language is that it is in many respects similar to C, or C++. So much so that I usually use the GNU Emacs C++ mode to edit Tela code, since there is not a specific Tela mode (maybe you could contribute one?). There is an **if** statement (3.1.1), a **while** statement (3.1.2) and a **for** statement (3.1.4). There are of course many syntactic differences also, but two of them are the most fundamental:

1. In C, a semicolon ends a statement whilst in Tela a semicolon is a separator between successive statements. Tela's use of semicolons is equivalent to Pascal rather than C in this respect. This means for example that the right brace is usually followed by a semicolon in Tela, or at least that a semicolon does no harm there.

2. In C the assignment is an operator, i.e. a=b is syntatically an expression not a statement. In Tela the assignment is a statement. This is also similar to Pascal. This implies that the use of assignment is more restricted in Tela than it is in C or C++.

3.1 Flow of control statements

The following subsubsections describe the Tela control structures.

3.1.1 The if statement

The **if** statement has the following syntax (*stmt* = statement):

```
if (expr)
   stmt
```

or

```
if (expr)
    stmt
else
    stmt
```

The conditional expression *expr* must be enclosed in parentheses. There must be no semicolon between *stmt* and **else**.

The expression must evaluate to integer scalar or array. The condition is considered true if the scalar is nonzero or if all the array elements are nonzero. Examples:

```
if (x!=0)
    format("x is nonzero, x=``\n",x)    // notice no semicolon
else {
    x = 1;
    format("x was zero and is now 1\n");// this semicolon is optional
};  // this semicolon separates this if stmt from the next stmt, if any
```

Nested **if** statements are written in the following style:

```
if (a == 1) {
    /* action for a==1 */
} else if (a == 2) {
    /* action for a==2 */
} else if (a == 3) {
    /* action for a==3 */
} else {
   /* action for none of the above */
};
```

3.1.2 The while statement

The **while** loop statement has the following syntax:

```
while (expr)
    stmt ;
```

For example:

```
    while (!found) {
        LookForIt();
        counter++
    };
```

The statement is executed until *expr* evaluates to true. The "trueness" of *expr* is similar to the **if** statement (3.1.1): a scalar and all elements of an integer array must be nonzero for *expr* to be true.

3.1.3 The repeat statement

The **repeat** loop statement has the form

```
repeat
    stmt1;
    stmt2;
    ...
until expr;
```

The statements *stmt1*, *stmt2*, ... are iterated until *expr* evaluates to true (nonzero). The statements are always executed at least once. It is analogous to the **repeat** statement in Pascal.

3.1.4 The for statement

The **for** loop statment has the form

```
for (initstmt; expr; updatestmt) stmt;
```

It is equivalent to the **while** statement

```
initstmt;
while (expr) {
    stmt;
    updatestmt;
};
```

The syntax is rather similar to C, but there is a difference: the *initstmt*, *updatestmt* and *stmt* are statements, not expressions. On the other hand there is no comma operator in Tela. Thus you cannot write **for(i=0,j=0; i¡10; i++)** but instead you must write

```
for ({i=1; j=1}; i<=10; i++) { /* ... */ };
```

We intentionally changed the loop running from 1, not 0, to remind that in Tela the first array index is 1, not 0 as in C.

3.2 Expressions and assignments

The following subsubsections describe Tela expressions and assignment statements.

3.2.1 Operators

Expressions are composed of atomic expressions and operators. Atomic expressions can be variable names, literal constants (integers, reals, imaginary constants, characters, array constructors, and strings), function calls, or array references. Operators usually obey the precedence rules in other programming languages where possible. The operators are, from lowest to highest precedence:

```
Operators       Associativity       Meaning
---------       -------------       -------
:               non-associative     Vector creation
||              left                Logical OR
&&              left                Logical AND
== !=           left                Equality and nonequality
> >= < <=       left                Greater than etc.
+ -             left                Addition and subtraction
* ** / mod      left                Pointwise multiplication,
                                      matrix multiplication,
                                      pointwise division,
```

```
                                     modulus operation
- +          non-associative       Unary minus and plus
^            right                 Exponentiation
!            non-associative       Logical NOT
.' '         non-associative       Transpose, Hermitian transpose
```

In Fortran, the operator ** would be exponentiation, in Tela it is matrix multiplication. In C ^ would be logical XOR, in Tela it is exponentiation. In Matlab * denotes matrix multiplication and **.*** pointwise multiplication, in Tela * is pointwise multiplication and ** is the matrix product. There are no matrix power and division operators in Tela as there are in Matlab. The equivalent of matrix division in Tela is the function **linsolve**. The vector creation operator uses the notation **a:step:b**, which follows the Matlab convention. In Fortran-90 the step is the last member, **a:b:step**. The set of operators is the same as in C, with the addition of vector creation, matrix multiplication, modulus, exponentiation, transpose and Hermitian transpose operators. These additional operators follow Matlab conventions except for the difference in pointwise and matrix nature of multiplication.

3.2.2 Atomic expressions

Atomic expressions can be:

```
variable names         a, b_89, $x, $_x9$
integer constants      0, 23
real constants         1.23, 4.5E3, 0.03
imaginary constants    1.23i, 4.5E3i, 0.03i
characters             'a'
array constructors     #(1,2,3), #(a,b; c,d)
strings                "with possible escape sequences\n"
function calls         f(), f(1), f(a+b,g(c))
array references       a[i,j], a[1], a[1:imax,2:jmax-1]
mapped references      a<[i,j]>, a<[1]>
```

Variable names, or any identifier names for that matter, start with a letter. The rest of the characters can also be digits or underscores. The dollar sign is also accepted as a “letter” in identifiers.

Imaginary constants are obtained from real constants by appending **i** with no intervening white space. There is no explicit notation for complex constants, you must use the addition or subtraction operators as in **2+3i** or **0.5-0.75i**. The way to denote the imaginary unit “i” is to write **1i**. Notice that “i” here is part of the syntax. There is no predefined variable or constant named “i”, and 2+3*i will generally not work as expected (unless you have assigned **i = 1i**, but this is not recommended).

Character constants are equivalent to their ASCII codes (integers) if used in arithmetic expressions. This practice is similar to C.

Array constructors are a bit more complicated. Syntactically an array constructor has the form **#(***component-list***)**, where *component-list* consists of expressions separated by commas and/or semicolons. Commas denote appending the components to form a vector. For example, **#(1:5,10)** will produce the integer vector **#(1,2,3,4,5,10)**, and **#(1,2.3,4:-1:1,34)** produces **#(1, 2.3, 4, 3, 2, 1, 34)**. The last example gives a real vector because one of the components was fractional number.

A semicolon in the array constructor denotes composing a higher-rank array of the components, which must be lower-rank arrays (or scalars). For example, a matrix can be constructed from its row vectors **v1** and **v2** by **#(v1;v2)**. The precedence of a semicolon inside array constructor is lower than the precedence of a comma, thus **#(a,b; c,d)** will construct of 2x2 matrix.

The array constructors work for higher rank arrays as well. The result of array constructor using commas has rank equal to the highest rank of the components, and the ranks of the components may not differ by more than one. The important exception is the case where all the components are scalars; in this case the result is a vector. The semicolon array constructor always produces a result which has rank one greater than the rank of the components, which must be same for all components in the semicolon case. Using array constructors for higher-rank components has been rare in practice.

Notice that the commas and semicolons have completely different meaning inside array constructors than outside them.

Strings may contain escape sequences similar to C language strings.

Array references follow the Pascal syntax, separating the dimensions with a comma. The indices may be vectors, which follows the Fortran-90 and Matlab array syntax ideas ("gather" operations).

In addition to normal array references, Tela also supports *mapped* array references. In mapped referencing the index objects must all agree in type (they are usually arrays). The number of indices must be equal to the rank of the indexed array and the result will have size equal to an index object. For example,

```
A<[1:5,1:5]>
```

will produce the first five diagonal elements of matrix A as a vector. Mapped indexing can be used to extract N-dimensional component subsets from M-dimensional arrays, both N and M being arbitrary. The function **intpol** (linear interpolation) can be thought as a generalization of mapped indexing, where the "index" expressions need not be integers.

3.2.3 Assignments

Assignments can take the following three forms:

```
variable = expr ;

variable[index1,...] = expr ;

[var1,var2,...] = fname(expr1,expr2,...) ;
```

The first form is a simple assignment, where the value of the expressions is assigned to a variable. The second form is the "scatter" operation, or indexed assignment. The indices follow the same rules as if the array reference appears on the right-hand-side of an assignment (see previous subsubsection).

The third form is actually not an assignment but a function call with several output arguments a la Matlab. The output variables must be separated by commas (In Matlab the commas may be left out.) The output variables must be simple identifiers, not expressions. For example, you cannot say

```
[b[1],b[2]] = f();          // WRONG!!!
```

You must use auxiliary variables, as in

```
[b1,b2] = f();
b[1] = b1; b[2] = b2;
```

There are some chances that this limitation might be removed in some future version.

3.3 Defining functions

Examples of function definition statements:

```
function f() { /* body */ };     // the simplest form

function f(x) { /* body */ };

function y=f(x) local(a) { /* ... */; y = sin(x) };

function [x,y] = f(a,b) { /* ... */ };

function [x,y;z,w] = f(a,b;n) global { /* ... */ };
```

The definition always begins with the reserved word **function**. After **function** comes the output argument specification (possibly empty), followed by the function name and input argument list, possible **local** or **global** declarations and finally the function body enclosed in curly braces.

Input arguments are passed by reference. They may not be modified in the function body. (If you try, you get a warning message.) Thus the calling program may *think* that the input arguments are passed by value even though they actually aren't. In C++ the type of the input arguments would be **const Ttype&**, and in Fortran-90 they would correspond to the "intent input" arguments. Input arguments are listed following the function name both when defining and calling a function.

Output arguments are listed in brackets before the "=" sign and the function name both in definition and calling phases. If there is only one output argument, the brackets may be dropped (see the third example above). Output arguments are also passed by reference, but obviously they may and should be modified by the function body.

By default, input arguments are obligatory and outputs arguments are optional. That is, the calling program must supply all input arguments, but it may leave out some or all of the output arguments. For example, if **f** is defined as

```
function [a,b,c] = f(x,y) { /* ... */ };
```

all the following are legal call forms for **f**:

```
[X,Y,Z] = f(2+3i,sin(X+y));     // use all output arguments

[X,Y] = f(1,2);     // ignore the third output argument

X = f(1,2);         // ignore second and third

[] = f(1,2);        // ignore all
```

The default behavior of obligatory and optional arguments can be changed by using the semicolon inside argument list. At most one semicolon may appear inside an argument list. The rule is simple: all arguments *before* the semicolon are obligatory, that is, required, and all arguments *after* the semicolon are optional. This rule applies to both input and output argument lists. Examples:

```
function f1(x,y;z,w) { /* ... */ };

function [a;b] = f2(x,y) { /* ... */ };

function [a;] = f3() { /* ... */ };

function [a;b] = f4(x;y) { /* ... */ };
```

Function **f1** has two obligatory input arguments and two optional ones. It has no output arguments. (It is, however, permissible to assign the "value" of **f1** to a variable as in **x=f1(0,0)**. The variable **x** has the void value after the assignment. The void value may be created explicitly by using an expression consisting of just the colon "**:**", and it doesn't output anything when printed.)

Function **f2** has to obligatory input arguments, since there is no semicolon in the input argument list. One of the two possible output arguments is obligatory and the other one is optional. Function **f3** has no input arguments but one obligatory output argument. Notice that the semicolon may be also the first or last thing in the argument list. Finally, function **f4** has both input and output optional and obligatory arguments.

Whether or not an optional input argument is present can be tested e.g. by the standard function **isdefined**, which returns 1 if the argument is defined and 0 if it is undefined. The test will fail also if the caller supplied the argument but it was bound to an undefined variable; this behavior is usually what is wanted since passing an undefined variable as argument is practically the same as not passing any argument at all.

The role of obligatory and optional input arguments, and also the meaning of optional output arguments, is clear. But what about obligatory output arguments, why should anyone want to use such things? The answer is simple: they correspond to arguments that are *both* input *and* output. By making an output argument obligatory you force the caller to bind it to some variable, which probably has some initial value that the function body may use.

Maybe we should pause for a while and recall the important concepts introduced in this subsection. By default, input arguments are obligatory and output arguments are optional. How it could be otherwise? Input arguments are read in the function body, so they may not be undefined. By making the output arguments optional by default we allow the caller the freedom to ignore some output arguments. The semicolon modifies the default behavior. Everything to the left of the semicolon becomes obligatory and everything to the right of the semicolon becomes optional. Input/output arguments should be declared as obligatory output arguments.

There is still one thing about function declaration, namely the **global/local** declaration that may be placed in between the input argument list and the function body. The declaration may take one of four possible forms:

```
local           // Everything is automatically local. The default.
global          // Everything is automatically global.
local(a,b,...)  // The listed symbols are local, others are global.
global(a,b,...) // The listed symbols are global, others are local.
```

All "free" symbols that appear in the function body are either local to the function or global. "Free" symbol means a symbol which is not one of the input or output arguments, has not been declared autoglobal using the standard function **autoglobal**, and is not used as a function name in the function body. By default, all free symbols are local. This corresponds to the Matlab convention. By inserting the word **global** with no variable list, however, you can make all free symbols refer to the globally visible symbols. By inserting **local(a,b,...)** you declare the listed symbols as local; other symbols remain global. This corresponds to the practice normally used in compiled languages such as C, C++ and Pascal. And finally, by using the **global(a,b,...)** declaration you can use the complementary approach where every symbol *except* those listed are local. The last case again mimics Matlab with global declaration.

Thus you can choose among different strategies here. In some functions it is more natural to list the local variables rather than the global ones, if not for other reason but because sometimes the local variables are less in number than global ones, and vice versa. The most improtant thing to remember is to remember: Think about global/local every time you define a new function. The experience is that many, maybe even most, error situations in Tela arise from forgetting to properly declare a variable global or local.

The system has some autoglobal variables, which are always global even if they are not explicitly declared. Among these are constants such as **pi**, **Inf**, **eps**, **on**, **off**, and some color and line style names. Thus it is unnecessary to say **global(pi)** if **pi** is used in a function. It is possible to override the autoglobal character by explicitly listing the symbol in **local** declaration; for me it is bad programming style however.

On trick that can be useful when developing code is to at first make most variables global. In this way they are available in the workspace for inspection after the function has completed execution or stopped in runtime error. When the function is working you can then make as many symbols local as possible.

3.4 The package mechanism

Typically in Tela you write a t-file with many, maybe a few tens of functions. Sometimes the functions need to communicate among themselves not only via arguments but also via global variables, as in Pascal or C. For example if you are developing a fluid simulation code the values of physical parameters such as grid spacing, viscosity etc. are most naturally declared global and not explicitly passed to every function that uses them. We have been taught that using globals is bad programming style, however, in this case it actually increases the modularity of the program because if you introduce more physical parameters you need not change the call form and definition of every function.

Problems may arise if you use such a t-file in conjunction with other t-files. The global variables then share the same name space. Also the internal auxiliary function names may be the same in two t-files. One solution would be to use untypical symbol names, but this is not elegant.

The keyword **package** is there to hide internal symbol names from external access. The use is very simple. Just enclose the whole t-file in curly braces and put **package global(var1,var2,...)** in front. In the list you should put all symbol names that you want to be externally visible. Usually these are function names, but they may also be (global) variable names. All other symbols are then put in a private name space and they correspond to the **static** variables in C and C++.

It is also possible to use **package local** declaration as a complementary approach, in analogy with global/local in function definition, but this form is rarely used in practice. You may also put a character string between **package** and **global** or **local** to name your package. This would be required if you want to put stuff from more than one t-file in one and a same package. Actually, if the string is left out, the fully qualified t-file name is used as a unique package name.

Despite the syntactic similarity of global/local in package declaration and function definition, the meaning is quite different. Symbols which are local to a package are usually global to functions inside the package. (If they are also local in the functions, declaring them local in the package context does not make any difference.) Local variables in a function are currently implemented as slots in a runtime stack, whereas global symbols (in the function sense) are bound to workspace variables, which have name, value and attributes. The only thing the **package** mechanism does is that it prepends to all local (in the package sense) symbol names an invisible string which is unique to every package. This way the variables are physically in the same global hash table but with unique names.

4 Writing t-files

Tela source files (t-files, filenames ending with ".t") consist of Tela statements separated by semicolons. A function definition is just one type of statement in Tela, thus you can define as many functions in one t-file as you wish. This feature sets Tela apart from Matlab, where each M-file always defines only one function. Functions become defined in the code generation phase. For this reason the order in which the functions appear in the t-file is unessential, and a function call may textually precede its definition. Whether this is good programming style is up to you. Usually there is no point in putting function definitions inside other functions; if you do it, the result is the same as if it were a toplevel definition.

Examples of t-files can be found in /usr/local/lib/tela/t, or whereever you have installed Tela. It is a bit difficult to give a simple meaningful example which defines multiple functions. A comprehensive example is provided by the file "madala.t", for example. It also uses the **package** mechanism to make it a clean entity that does not unnecessarily conflict with other packages' name spaces.

5 Program code representations

Thus far we have used and studied only one form of existence of our programs, namely the native Tela source code (t-files and command line). Both internally and externally, programs exist in other forms too. Here is a complete list of all code representations in Tela:

```
- t-code (the native Tela code)
- "TreeCode" (internal representation for t-code)
- "FlatCode" (another internal representation for t-code)
- C-tela code (code similar to C++)
- Compiled object code (machine language)
```

When a program is **source**'d (??) in or when a command line is typed, a lexical analyzer first breaks it in tokens, that is, keywords, separators, numbers, identifiers etc. The output of the lexical analyzer is a stream of tokens without hierarchical structure. A parser continues to produce a treelike representation, the TreeCode. The TreeCode is no longer a stream but has structure in it. TreeCode is immediately translated into FlatCode, which is a sequence of virtual machine instructions that the Tela kernel can understand, and it is reminiscent to assembly language. You can see both TreeCode and FlatCode representations of your input by setting the VerboseMode on:

```
>VerboseMode(on)
0
>1 + 2.3*pi
BLOCK[SET[PLUS[1,TIMES[2.3,pi]]],NOP]
Source file: "stdin"
no input args, no output args, no locals, stack frame size 1.
Maximum number of operands is 3.
0       MUL     $0,2.3,pi
4       INC     $0
6       PRI     $0
8.22566
```

Alternatively, you can invoke Tela with the –verbose or -v flag. The FlatCode of any *source*'d (??) function can be seen by using *disasm* (??):

```
>disasm(mean)
Disassembly of 'mean', Source file: "std.t"
1 input arg, 1 output arg, no locals, stack frame size 4.
Maximum number of operands is 4.
0       CALL    sum,1,$3,$1
5       CALL    length,1,$0,$1
10      DIV     $2,$3,$0
```

The definition of *mean* is

```
function y = mean(x)
// mean(x) computes the arithmetic mean of a numeric array x.
{
    y=sum(x)/length(x)
};
```

and you can easily guess what each of the FlatCode instructions do.

C-tela code is another way to write new functions for Tela. Many Tela builtin functions are written in C-tela; those which are not are either in native t-code or are so-called intrinsic functions (meaning that they generate virtual machine instructions directly like macros). C-tela code can be compiled into object files using the program **telakka** (??). The object files can be either statically or dynamically linked with the rest of the Tela kernel.

There exists a standard function *t2ct* (??) to translate t-code into C-tela code. The conversion is made possible by the fact each FlatCode instruction can be made to correspond simple C-tela codings. As of this writing, however, the Tela compiling scheme is still somewhat incomplete and under development. When it is fully functional, you can go all the way from t-code down to object code automatically.

6 Writing C-tela files

C-tela files are C++ code equipped with Tela's function header syntax. There is a preprocessor, ctpp, that translates C-tela in ordinary C++. Calling ctpp is the responsibility of the tool program **telakka** (??).

Let us begin with an example of a C-tela source file (let it be mine.ct):

```
// Our first C-tela example

[] = myfn(x)
/* myfn(x) checks that x is a positive scalar and outputs it.
   Error codes:
   1: x is not a positive scalar */
{
    if (!x.IsScalar()) return 1;
    switch (x.kind()) {
        case Kint:
            if (x.IntValue() <= 0) return 1;
            break;
        case Kreal:
            if (x.RealValue() <= 0) return 1;
            break;
        case Kcomplex:
            return 1;
        default:
            return 1;
    }
    cout << "myfn: x = " << x << '\n';
    return 0;
}
```

You can compile it using **telakka** (??):

```
unix> telakka -c mine.ct
```

Then from Tela you can link the object file and test myfn as follows.

```
>link("mine.o")
>help myfn
myfn(x) checks that x is a positive scalar and outputs it.
>myfn(2)
myfn: x = 2
>myfn(-2)
Warning from C-function 'myfn':
  x is not a positive scalar.
>myfn(2+3.4i)
Warning from C-function 'myfn':
  x is not a positive scalar.
>myfn(#(1.2,3,4))
Warning from C-function 'myfn':
  x is not a positive scalar.
```

There are several points to be noticed with this simple example:

```
- The function is declare using the [...] = f(...) type syntax.
- The /* ... */ comment immediately following the header is "hot".
  The Tela help command finds and displays the comment, up to line
  "Error codes:".
- The function should return zero on success, positive integer on
  nonfatal error and negative integer on fatal error. The error codes
  should be listed in the comment, following, one per line, the
  "Error codes:" line.
- All C++ constructs are available, in addition several classes and
  their associated members from Tela headers. The function arguments
  ("x" in our case) are of type Tobject and they can be e.g. outputted
  using cout << x.
```

Rules for argument syntax are very similar to those found in Tela. Output arguments are enclosed in square brackets and input arguments in parentheses. Optional arguments are separated from obligatory arguments with a semicolon, parameters to the right of the semicolon are optional and parameters on the left hand side of the semicolon are obligatory. If there is no semicolon in the output argument list, then all output arguments are optional. If there is no semicolon in the input argument list, then all input arguments are obligatory. In addition, it is possible to use the ellipsis (...) notation to denote an arbitrary number of optional arguments. The ellipsis is only allowed as the last thing in a parameter (either input or output) list. The following function has two obligatory output arguments, one optional output argument, two obligatory input arguments, and any number of optional input arguments, the first of which is named c:

```
[x,y; z] = f(a,b; c...)
```

The function body can refer to the arguments simply by name, and they are of class Tobject. Actually the argument lists are implemented using four C++ function parameters: the input argument list (argin), the length of the input argument list (Nargin), the output argument list (argout), and the length of the output argument list (Nargout). Nargin and Nargout are of type int. argin and argout are arrays of pointers to Tobject. The named arguments are just C preprocessor macros. For example, if x is the first input argument, its definition is

```
#define x (*(argin[0]))
```

The pointers contained in the argument arrays are protected against modification by the C++ keyword const. In addition the input array individual objects are also const. The C++ compiler will therefore give an error message if you try to assign or otherwise modify an input argument. Also Nargin and Nargout carry the const attribute, because changing them is unnecessary.

Ellipsis arguments have no names and therefore must be referenced using the argin and argout arrays explicitly. This is easy, for example to process all input arguments starting from the second one (which is *argin1, remember that in C arrays start with subscript 0), you could use a code like this:

```
static void Process(const Tobject& obj) { /* ... */ }

[] = myfunc(x...)
/* Help message ... */
{
    /* ... */
    for (int i=1; i<Nargin; i++)
        Process(*argin[i]);
    /* ... */
}
```

As mentioned earlier, C-tela code is C++ code equipped with the Tela-like function header syntax. In addition to this, there are also some small restrictions on C-tela source files:

```
1. The function header must be on one line, and the first character
(the left square bracket) must be on first column.

2. The right brace which ends the C-tela function body must be in the
first column. No other right brace inside the function body may be in
first column.

3. You cannot temporarily remove C-tela functions from your source
file using preprocessor directies, e.g. by enclosing the lines in
#if 0 ... #endif. If you want to do this, you must also enclose the
function headers in a comment.
```

These rules should not limit your programming capabilities in any serious way, and most of the time you do not have to even think about them.

Then what can you do with the Tela objects, that are of type Tobject in C++? Firstly, every Tobject has a tag containing that object's kind. The possible kinds are (from header file object.H):

```
enum Tkind {               // Object kinds (types)
    Kint,                  // Integer scalar
    Kreal,                 // Real scalar
    Kcomplex,              // Complex scalar
    KIntArray,             // Integer array (n-dimensional)
    KRealArray,            // Real array
    KComplexArray,         // Complex array
    KObjectArray,          // Object (pointer) array, currently not used
    Kfunction,             // User-defined function, written in Tela
    KCfunction,            // Compiled and linked C-Tela function
    KIntrinsicFunction,    // Special "functions" generating inline code: abs, min, max ..
    Kvoid,                 // Empty value, when printed prints nothing
    Kundef                 // Undefined value, the default for new symbols
};
```

Let us then list the most useful Tobject public members:

```
class Tobject {
  /* ... */
    // --- constructors
    Tobject();                         // default constructor: it will have Undefined value
    Tobject(Tint a);                   // construct integer scalar object
    Tobject(Tchar ch);                 // construct character object
    Tobject(Treal a);                  // construct real scalar object
    Tobject(Tcomplex a);               // construct complex scalar object
    Tobject(Treal x, Treal y);         // construct complex scalar object
    Tobject(const Tint itab[], int N, int stringflag=0);    // construct integer vector or string
    Tobject(const Treal xtab[], int N); // construct real vector
    Tobject(const Tcomplex ztab[], int N);  // construct complex vector
    Tobject(const Tint itab[], const TDimPack& dp); // construct integer array
    Tobject(const Treal xtab[], const TDimPack& dp);    // construct real array
    Tobject(const Tcomplex ztab[], const TDimPack& dp); // construct complex array
    Tobject(const Tchar *str);         // construct string
    Tobject(const Tobject& obj);       // copy constructor: use other object's value

    // --- assignments
    Tobject& operator=(Tint a);        // assign integer scalar
    Tobject& operator=(Treal a);       // assign real scalar
    Tobject& operator=(const Tcomplex& a);  // assign complex scalar
    Tobject& operator=(Tchar ch);      // assign character
    Tobject& operator=(const Tchar *str);   // assign string
    Tobject& operator=(const Tobject& obj); // copy assignment: assign other object's value
    void izeros(const TDimPack& dp);    // set to zeroed int array of given dims
    void rzeros(const TDimPack& dp);    // set to zeroed real array of given dims
    void czeros(const TDimPack& dp);    // set to zeroed complex array of given dims
    void ireserv(const TDimPack& dp);   // set to uninitialized int array of given dims
```

```
    void rreserv(const TDimPack& dp);   // set to uninitialized real array of given dims
    void creserv(const TDimPack& dp);   // set to uninitialized complex array of given dims
    void SetToVoid();                   // set to void (empty) value
    void SetToUndefined();              // set to undefined value
    void SetStringFlag();               // set string flag (assuming it is IntArray already)
    void SetCharFlag();                 // (assume it is Kint already)
    void ClearStringFlag();             // unset string flag (assuming it is IntArray already)
    void ClearCharFlag();               // (assume it is Kint already)

    // --- comparison
    int operator==(const Tobject& obj) const;
    int operator!=(const Tobject& obj) const;

    // --- inquiry functions
    Tkind kind() const;                 // inquire object kind
    int length() const;                 // number of elements of array object
    Tint rank() const;                  // rank of array object
    Tint IntValue() const;              // value of integer scalar object
    Treal RealValue() const;            // value of real scalar object
    const Tcomplex& ComplexValue() const;   // value of complex scalar object
    Tint& IntRef();                     // modifiable lvalue of integer scalar object
    Treal& RealRef();                   // modifiable lvalue of real scalar object
    Tcomplex& ComplexRef();             // modifiable lvalue of complex scalar object
    Tint *IntPtr() const;               // start address of elements of integer array
    Treal *RealPtr() const;             // start address of elements of real array
    Tcomplex *ComplexPtr() const;       // start address of elements of complex array
    int IsNumerical() const;            // nonzero if object is numerical (scalar or array)
    int IsArray() const;                // nonzero if object is array
    int IsNumericalArray() const;       // nonzero if it is numerical array
    int IsScalar() const;               // nonzero if object is (numerical) scalar
    int IsFunction() const;             // nonzero if object is Tela-function
    int IsCfunction() const;            // nonzero if object is C-tela function
    int IsString() const;               // nonzero if object is string
    int IsChar() const;                 // nonzero if object is character
    int IsNonzero() const;              // tests whether object is not zero
    const TDimPack& dims() const;       // return array object's dimensions

    // --- other operations
    friend ostream& operator<<(ostream& o, const Tobject& obj);     // outputter

    // --- destructor
    ~Tobject();
};
```

Using all these members seems to be overwhelming task at first. But there is certain logic behind this design. Basically, you can

7 Interfacing issues Here we explain the main mechanisms to communicate with other programs and data files from Tela. How to extend Tela by your own C/C++ functions has been explained in the previous section.

7.1 Matlab interfacing

There are three principal mechanisms to use Tela together with Matlab.

You can import and export Matlab binary files to and from Tela using the **import**, **import1**, **export_matlab** commands. Also Matlab-readable plain ASCII files can be read by **import1**. Even all your workspace variables can be exported or imported easily in binary form. That is, Tela has the same functionality as two separate Matlab processes would have.

It is possible to translate your Matlab scripts and M-files to Tela language. A translator program **m2t** exists for this purpose. See **man m2t** for details.

If you have the **matlabeng.ct** module linked within Tela, you can call Matlab functions and evaluate strings as Matlab commands at run time. To enable this, there is function **matlab_start**, which starts Matlab in the background and sets up necessary communication links. This is a good aid in development phase if e.g. some important function is missing in Tela. You can borrow it from Matlab.

7.2 HDF interfacing

Tela's native binary format is the HDF (Hierarchical Data Format from Univ. Illinois/NCSA) format. HDF files created with the **save** command can be read by various programs supporting HDF, for example Spyglass tool programs and IDL. The **import** command can also read many other types of HDF files meaningfully. The function **export_RIS8** exports 8-bit raster image sets in HDF format.

7.3 ASCII files

Generic ASCII files can be read by the **import1** command. Multidimensional datasets can also be read by inserting the size information on the first line in a simple format (see **help import1**). ASCII files can be written using the **fformat** and **fprintf** functions.

7.4 Graphics formats

The **PlotMTV** program is responsible for the graphics. It will save the image in PostScript file if you click the appropriate button. In **PlotMTV1.4.1t** and later you can also save the image as a GIF (Graphics Interchange Format) file by pressing another button. When the main PlotMTV window has been divided in several data windows, pressing the GIF button will include all data windows in the file. Pressing Shift-G under these circumstances will include only the current data window. The GIF saving will work properly only if the whole window is visible on your workstation screen (or in the virtual screen, i.e. it must be in video-RAM).