Documentation on TCE modules

Banens, J.P.A.

Published: 01/01/1994

Document Version
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the author's version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 02. Jan. 2019
Documentation on TCE modules
April 19, 1994

WFW Report 94.050

Included is user documentation on

- exp - standard experiment functions
- mat - matrix arithmetic
- ml - MATLAB exchange basic functions
- tce - general info about TCE
- tim - timing
- ts  - time series

April 19, 1994, Jos Banens
1 Introduction

We like to use a standard experiment interface. This not only gives experiment programs a uniform look, but also encourages the use of "plug-in" simulators, as the simulator just need to implement the standard functions.

The running of experiments, apart from main program and control and/or estimation strategies (which have to be coded by the user; some example files are available, see section 7), consists of three key features:
- fetching measurements;
- outputting commands to actuators;
- timing.

This is what the exp-module implements. It is compatible with TCE, see tce.doc for more information.

Some preliminary remarks have to be made. First, almost always you want your experiment sampled on a regular (but generally free to choose) time base. The timing in the exp-library automatically maintains this constant sample frequency. Secondly, although the specifications of the exp-library are common for different experimental environments, the implementation differs from one experimental environment to another. For instance, the number and the meaning of the measurements are specific for the experiment. This document describes only the general specifications. A specific implementation will have its own dedicated document which explains the specific signals involved. You need that document too.

When you plan to make an exp-implementation for a new experimental (or simulation) environment, please read the implementation documentation exp_a.doc in addition to this document.

2 Tutorial

Here is a simple program using the main exp_ functions.

#include <tce.h> /* Remarks below */
double u[2], /* (1) */
    z[4];
double wait[3];

void epilog(void)  { } /* (2) */

void main(void) {
    int i;

    exp_init(z,4,u,0.005,wait); /* (3) */
    for ( i=0; i<1000; i++ ) {
        exp_get(); /* (4) */
        /* Compute commands u using measurements z */
        exp_put(); /* (5) */
    }
    exp_fini(); /* (6) */
}

Remarks:
(1) Here we use global variables to hold (up to) 2 commands in u and (up to) 4 measurements in z. As C and C++ index from zero, the individual elements of u are u[0] and u[1]; u[2] does not exist! Analog indexing holds for y (and any other array).
The three element array wait is required by the exp_ functions.
(2) Using the exp_ functions, you must write a function epilog. This one is the minimal one. More on this is explained below.
(3) Function exp_init initializes the experimental environment. Exactly four measurements are to be collected in vector z and command values will reside in vector u. The (constant) sample time to be used is 0.005 seconds and wait count information is to be written in wait.
The specific implementation of exp_init dictates the length of u (i.e. how many commands must be supplied). It also decides if it can accommodate the four measurements you want to have. More on wait count information may be found below.
Function exp_init enters a time-critical environment. All its arguments are used in exp_get, exp_put and/or exp_fini, so all must have a sufficient large scope (see tce.doc for details). Here, we made them global variables.
When something fails or is incorrect, exp_init will abort your program while displaying a clear message about the trouble. It is possible that there is no support for four measurements or that the sample time is out of the range permitted. Alternatively, there may be hardware or interface malfunctioning.
One (potentially severe) problem cannot be detected. You may ask for four measurements in vector z, but supply a vector with three elements only. The fourth measurement will be written at some other location in memory, probably corrupting one or more other values. The same holds for too short vectors u or wait (the latter must have at least three elements).
(4) Function exp_get waits until the next sample moment (with sample time as specified by the fourth argument of exp_init); it then fetches 4 measurements into z as specified by the first argument of exp_init.
(5) Function exp_put outputs the value(s) in u (as specified by the third argument of exp_init) to the experiment.
(6) Function exp_fini brings the experiment in a safe state, zeroes the commands and finishes the real-time environment.
Remark: as your latest call to exp_put may have commanded large actuator signals, you must call exp_fini immediately at the end of the experiment, before you do other things like writing collected data to disk (which may take quite a time while you have your actuators still active) or exiting your program (which may leave your actuators active still longer).

3 Functions

All functions except exp_init are for real-time use. Interrupts are disabled by exp_init and re-enabled by exp_fini.
All arguments of exp_init must have a sufficiently large scope to cover all exp_ function calls. See tce.doc for details.

void exp_fini(void)
Finishes the experimental environment, which is opened by exp_init. Leaves the experiment in a safe state. Establishes the wait count values in vector wait (the last argument of exp_init). More on wait counts may be found in section 5.

void exp_get(void)
Waits until the next sample moment, then fetches all measurements into argument z of exp_init. There is no return code. When something fails, exp_get will abort through the emergency route, see section 6.

void exp_init(double* z,int nz,double* u,double T,double* wait)
Initializes the experimental environment:
- designates z to hold exactly nz measurements;
- designates u to hold the commands;
- creates a sampled data system environment with sample time T [s];
- designates wait to hold three wait count values.
Upon the call of exp_init, the values in z and u do not matter: they are not used.
Function exp_init returns with actual measurements in z, but aborts upon any problem.
When running C++ and using class mat, you may use variants of exp_init, described in the next section.

void exp_put(void)
Outputs the current value(s) in u (the third argument of exp_init) to the experiment. There is no return code. When something fails, exp_put will abort through the emergency route, see section 6.

long exp_wait_count(void)
Returns the most recent wait count value. See section 5.

void exp_wait_reset(void)
Resets the wait count information. See section 5.

void exp_wait_stamp(void)
Copies collected wait count information (since the latest call of exp_wait_reset) to vector wait as specified by the last argument of exp_init:
wait[0] - minimum
wait[1] - maximum
wait[2] - average
Remark: the wait count information is collected in internal format; you get it presented in wait only by calling exp_wait_stamp.
4 Additional functions in C++

void exp_init(mat& z, mat& u, double T, double* wait)
    Initializes the experimental environment:
    - designates z to hold the measurements;
    - designates u to hold the commands;
    - creates a sampled data system environment with sample time T [s];
    - designates wait to hold three wait count values.
    Both z and u must have their intended length before calling exp_init, but the values in them do not matter.
    Function exp_init returns with actual measurements in z, but aborts upon any problem.

void exp_init(mat& y, mat& z, mat& u, double T, double* wait)
    Initializes the experimental environment:
    - designates z to hold the measurements;
    - designates y to hold a subset of z;
    - designates u to hold the commands;
    - creates a sampled data system environment with sample time T [s];
    - designates wait to hold three wait count values.
    All of y, z and u must have their intended length before calling exp_init; y must contain indices in z (numbered from 1 in MATLAB style) to select a subset of z, for instance to be used for a Kalman filter; the values in z and u do not matter. Function exp_init returns with actual measurements in y and z, but aborts upon any problem.

5 Wait counting

Function exp_get waits until the next sample moment. While waiting (in a polled environment), it counts. The resulting final wait count value is proportional to the wait time. There is no fixed translation from count to time as this depends on the kind of the event which indicates the next sample moment and on the speed of the processor.
There normally are many calls to exp_get, so there are as many wait count values involved. However, most of them will be (almost) equal when you run a constant-duty loop. Function exp_get maintains statistical wait count information: minimum, maximum and total. To maintain this kind of values you need some starting point: function exp_wait_reset.
Function exp_init finishes with
    exp_wait_reset();
    exp_get();
Referring to the example in section 2, the highest wait count may be expected between the exp_get which finishes exp_init and the first exp_get in your loop as there is no "Compute u .." here. Function exp_fini calls exp_wait_stamp.
You may do (detailed) things yourself with the exp_wait functions described above.
Remark: As all interrupts are off between exp_init and exp_fini, the real time clock is not running either. So, you cannot use the tim functions (see tim.doc) for timing measurements here. Decipher the wait counts, they may tell you a lot about timing.
The exp functions may implement a simulator, not the real experimental environment. In this case, interrupts and the real time clock operate normally. However, using the exp functions, your program will run either the real experiment or the simulator without any recoding. So, avoid the use of timing with the tim functions.
6 Emergency actions

When a serious problem occurs in the time-critical situation, i.e. between `exp_init` and `exp_fini`, where the experiment is running and you may have momentarily large actuator signals, abortion of the program is done through special emergency code. This code calls `exp_fini` to (try to) bring the experiment in a safe state. However, you may be (and probably are) collecting data for later post-processing. You may be interested in the data collected up to the problem. This is why the emergency code calls a second function with the predefined name `epilog` (no arguments, returning `void`; the name is predefined by `exp_init`).

When you write a function `epilog`, to be called after your regular `exp_fini`, which processes the collected data, this function will be appropriate for the emergency action as well.

When you forget to supply the function, the linker will complain:

"Function epilog not found"

to remember you.

7 Example programs

There are a number of example programs available. The example in section 2 is one of them, but it lacks an environment: there are no provisions to let you specify how to compute $u$; it does not collect any data for postprocessing.

There are other example programs as well, which includes an environment. These are not appended to this document as written text, but available as files in the \TCE directory. In this way it is easier to grab one of these template files and change it to accommodate your goals. To accommodate the environment, they use functions from other TCE-modules as well. To understand them in detail, you should refer to the documentation of these modules as well.

The available template files are:

- `exp1.c` - Proportional feedback, no matrices
- `exp2.c` - State feedback & Kalman Filter

8 Availability

As the `exp` library has implementations for each specific experimental (or simulation) environment, it is not included in the TCE libraries.

When you are interested in an experimental environment `<env>` (with any appropriate short text `<env>`), look for files `exp_<env>.lib` and `sim_<env>.lib` for the specific experiment and simulation libraries respectively. Add the appropriate library to your project.

Look also for a document `exp_<env>.doc` which will explain details about the specific experimental environment.

There will be a file `exp_<env>.def` with definitions; it may serve as an include file in your program.

The best place to look for files is in directory \TCE.

Remark that files in directory \TCE may never be modified by a casual user. Other users depend on them too.
Contents

1 Introduction 2

2 Tutorial 2

3 Class mat 3
  3.1 Declarations 3
  3.2 Operators 3
  3.3 Functions 4
    3.3.1 MATLAB compatible functions returning mat 4
    3.3.2 Replacements for MATLAB operators 5
    3.3.3 Special signal handling functions 5
    3.3.4 MATLAB exchange functions 6
    3.3.5 Time series support functions 7
    3.3.6 Other functions 8
  3.4 Low level access 8
    3.4.1 Addressing individual elements 9
    3.4.2 Type of the elements 9
  3.5 Subarrays 9
  3.6 Array management 11
  3.7 Error handling 12

4 Application Notes 12
  4.1 Matrices and scalars 12
  4.2 Time series in MATLAB 13
  4.3 Writing your own mat functions 13
  4.4 Speeding-up 15
    4.4.1 Evaluation of expressions 16
    4.4.2 Administrative overhead 17
  4.5 Trouble shooting 18

5 Availability 19

6 Special classes 20
  6.1 Special matrices 20
    6.1.1 Class dmat - diagonal matrices 20
    6.1.2 Class pmat - packed matrices 21
    6.1.3 Class smat - symmetric matrices 21
  6.2 Class ivec - integer vectors 21
1 Introduction

This document describes class mat, which implements, in C++, basic matrix operations and functions. A subset of MATLAB's functionality is available, together with 'Load from MATLAB' and 'Save to MATLAB' and time-series support.

C++ and class mat are not a copy of, nor a substitute for, MATLAB: they lack the interactive way of work which is MATLAB's strong side. Moreover, the implementation is very restricted compared with MATLAB and it has no complex arithmetic. However, as things are compiled in C++, both evaluation of matrix algebra expressions and execution of surrounding language constructions like loops, is done much faster than in MATLAB. Exactly for this reason, C++ and class mat may be useful, for instance to implement real-time control schemes.

Class mat conforms to the TCE standard, although it's a rather special module which may be useful in quite different applications as well. See tce.doc for general information.

To exchange information with MATLAB, TCE includes a module ml, documented in ml.doc. For real-time purposes, there is a time-series support module ts, documented in ts.doc. Class mat includes an extension of the basic MATLAB exchange and time series support to handle matrices as well. This is documented here.

Besides class mat, there are some related classes available:
- class dmat, which implements diagonal matrices;
- class pmat, which implements packed matrices (for sparsely filled ones);
- class smat, which implements symmetric matrices;
- class ivec, which implements integer vectors with (just) enough support to be useful as index vectors in sub-array operations.

Documentation on them is found in section 6.

2 Tutorial

Let’s compute
\[ c = a \ast b; \]
where a, b and c are matrices. Matrices a and b are made by MATLAB and saved in file in.mat using MATLAB’s save facility:
\[
\text{save in a b}
\]
Matrix c is to be returned to MATLAB in file out.mat. You get it with MATLAB's load command.

Here is our C++ program:

```c
#include <mat.h> // (1)

void main(void) {
    mat a,b,c; // (2)

    ml_open("in",0); // (3)
    ml_get(a,"a"); // (4)
    ml_get(b,"b");
    c = a*b; // (5)

    ml_open("out",1);
```

// Remarks below
3 Class mat

3.1 Declarations

You must declare all matrices before you can use them (in MATLAB, declaration is not necessary). Declarations are written as:

```cpp
mat A, b, clist[4];
```

Here, `clist` is an array of matrices (not found in MATLAB). Above matrices have initially size 0*0.

You cannot do any computations with zero-size matrices, but matrices are sized automatically upon their usage in assignments (as for `c` in `c = a * b` in the example above) or in other matrix construction contexts (as in `ml-get` in the example above).

You may specify a matrix dimension upon declaration:

```cpp
mat B(5, 3);
```

declares `B` to be a 5*3 matrix initialized with all elements equal to zero.

3.2 Operators

The implementation of the standard operators is done in the same way as in MATLAB. You may apply the operators `+`, `-`, `*` and `/` to matrix and scalar operands. When at least one of the operands is a matrix, the result is a matrix. When one operand is a matrix and the other a scalar, the scalar is applied to all elements of the matrix. This works exactly as in MATLAB.

Some remarks must be made (all operands are assumed to be matrices unless stated otherwise):

1. The binary operation `s/m` with scalar `s` is not implemented.
2. The operator `^` (to the power) does not exist in C++, sorry.
3. The transpose operator `t` is not there either; use function `tran` instead.
4. The operator `\` does not exist in C++; so, you cannot write `A \ b`. Use `solve(A, b)` instead. The functions `tran` and `solve` are documented in the next section.

The compound assignment is a well-known and widely used C++ feature which allows the compiler to generate efficient code. The compound assignment

```cpp
R += A;
```

is equal to

```cpp
R = R + A;
```

For all operations, where the resulting `R` may be made by just updating the old one, the compound assignments may be applied to class `mat` variables as well. Their implementation is much more efficient, so you are encouraged to use them. Not available are `R *= A` and `R /= A` with a matrix `A` as `R*A` or `R/A` cannot be computed by just updating `R`. 

```cpp
ml_put(c, "c"); // (6)
```
3.3 Functions

3.3.1 MATLAB compatible functions returning mat

All functions act as their MATLAB versions, however with single (mat) output argument only. They may be used in expressions as they return a mat.

    // Remarks:
    abs (mat&)  
    cos (mat&)  
    diag (mat&)  // only for square matrix or vector
    exp (mat&)  // not MATLAB's expm !!
    eye (int)   
    eye (mat&)  // only for square matrix
    feval(double(*f)(double),mat&)  // see (1) below
    inv (mat&)  // only for square matrix
    log (mat&)  
    max (mat&)  
    max (mat&,mat&)  
    max (mat&,double)  
    max (double,mat&)  
    min (mat&)  
    min (mat&,mat&)  
    min (mat&,double)  
    min (double,mat&)  
    norm (mat&,int=0)  // see (5) below
    ones (int,int=0)  // see (2) below
    ones (mat&)  
    rand (int,int=0)  // see (2) and (4) below
    rand (mat&)  
    sign (mat&)  // see (3) below
    sin (mat&)  
    sum (mat&)  
    tan (mat&)  
    zeros(int,int=0)  // see (2) below
    zeros(mat&)  

Remarks:
(1) Function feval evaluates any standard C++ function with a single argument. cos(a) is equal to feval(cos,a). Evaluation is done elementwise.
(2) The functions with two int arguments accept a single argument as well; in this case, as in MATLAB, the second argument is assumed to be equal to the first one.
(3) There is a special set of signal handling functions. See below.
(4) With a third rand function, compatible with MATLAB, you may influence the behavior of the random number generator:
    rand("uniform")
    rand("normal")
    rand("seed",seed)

Only the first letter of uniform, normal or seed is needed, but it must be enclosed within double quotes ("), where MATLAB uses single quotes (’) to represent a string. The last function sets the internal seed value of the generator to the (long integer) value seed. If seed==0, you get
an unpredictable random sequence (i.e. one which will be different for different runs of your program). This is also the default behavior. Except for the quotes, this is equivalent to the MATLAB version. However, do not expect to get MATLAB's sequence for the same seed value.

(5) The implementation is restricted compared to MATLAB. Available are only the 1, 2 and infinity norm for vectors and the 1 and infinity norm for matrices. The optional second argument must be 0, 1 or 2 and specifies which norm you want; use 0 if you want the infinity norm; this is also the default.

With standard C++ preprocessor features you may use inf for 0 as well when you insert at the top of your program a line:

```cpp
#define inf 0 \ which translates every occurrence of the word \verb!inf! to 0.
```

This line is not included in file mat.h to avoid misunderstandings which may occur when this kind of preprocessor translations are done automatically.

### 3.3.2 Replacements for MATLAB operators

- `mat solve(mat& A,mat& b)` // returns $A \backslash b$; square $A$ only
- `mat tran(mat& A)` // returns the transpose of $A$

Remark that `solve(A,b)` is equivalent to $\text{inv}(A) \ast b$; this is different from $b/A$, which is equivalent to $b \ast \text{inv}(A)$.

For symmetric, positive definite matrices, there are faster variants:

- `mat sinv(mat& A)` // returns the inverse of $A$
- `mat ssolve(mat& A,mat& b)` // returns $A \backslash b$; square $A$ only

However, there is no check on the symmetry of $A$. See also diagonal and symmetric matrices in section 6.

Function `solve` uses LU-decomposition and backsubstitution. You may do things yourself with the functions `ludec` and `lusolve`:

```cpp
x = solve(A,b)
```

is equivalent with:

```cpp
LU = ludec(A);
x = lusolve(LU,b);
```

where you may apply several matrices $b$ using the same LU.

Function `ssolve` may be split in analogous manner in Choleski decomposition and backsubstitution:

```cpp
x = ssolve(A,b) is equivalent with $R = \text{chdec}(A)$; $x = \text{chsolve}(R,b)$; with upper triangular matrix $R$.
```

### 3.3.3 Special signal handling functions

- `mat clip(mat& A,mat& min,mat& max)`
  
  Returns $A$ clipped between $\text{min}$ and $\text{max}$. The size of $\text{min}$ and $\text{max}$ must match the size of $A$. Clipping is done element-by-element.

- `mat clip(mat& A,mat& limit)`
  
  Returns $\text{clip}(A, -\text{limit}, \text{limit})$.

- `mat clip(mat& A,double min,double max)`
  
  Returns $A$ with all elements clipped between $\text{min}$ and $\text{max}$.

- `mat clip(mat& A,double limit=1.0)`
  
  Returns $\text{clip}(A, -\text{limit}, \text{limit})$. Argument limit may be omitted and defaults to 1.0.
mat sat(mat& A, mat& w, mat& h)
Returns saturation value of A: \( R = \text{sat}(A, w, h) \) with
\[
R(i,j) = -h(i,j) \quad \text{if } A(i,j) \leq -w(i,j)
\]
\[
R(i,j) = h(i,j) \quad \text{if } A(i,j) \geq w(i,j)
\]
\[
R(i,j) = h(i,j)/w(i,j)*A(i,j) \quad \text{otherwise}
\]
The size of \( w \) and \( h \) must match the size of \( A \).

mat sat(mat& A, mat& w)
Returns clip(A, w, ones(A)).

mat sat(mat& A, double w, double h=1.0)
Returns saturated A: \( \text{clip}(h/w*A, h) \). Argument h may be omitted and defaults to 1.0.

mat sign(mat& A, mat& max)
Returns the result of an element-wise sign: \( R = \text{sign}(A, \text{max}) \) will result in:
\[
R(i,j) = \text{max}(i,j) \times \text{sign}(A(i,j))
\]
The size of \( \text{max} \) must match the size of \( A \).

mat sign(mat& A, double max=1.0)
Returns sign(A, max*ones(A)). Argument max may be omitted and defaults to 1.0.

There are also so-called member function versions of all above signal handling functions. They are faster as they operate directly on a matrix; they are comparable with the compound assignments in section 3.2. In C++ terminology, the matrix is called an object, in this case an object of class mat.
The first clip function in member function version is defined as:

mat& mat::clip(mat& min, mat& max)
Returns object clipped between \( \text{min} \) and \( \text{max} \). The size of \( \text{min} \) and \( \text{max} \) must match the size of object. Clipping is done element-by-element.

and its use is:
\[
A\text{.clip(min, max)};
\]
where \( A \) is changed to its clipped value. It is equivalent with \( A = \text{clip}(A, \text{min, max}) \).
All other functions are similar, i.e. replacing their first argument with a mat object to operate on.

### 3.3.4 MATLAB exchange functions

Refer to ml/doc for MATLAB exchange support basic functions. The functions in this section are a supplement on that.

mat ml_get_mat(char* name)
Returns a matrix equal to MATLAB variable name, but aborts upon any error including name not found, MATLAB file not open for reading (see ml_open in ml.doc).
Remark: the specification char* name means that you may (or better must) a string here. In C++, a string must be enclosed in double quotes (").

void ml_get_mat(mat& A, char* name)
Implements \( A = \text{ml_get_mat}(\text{name}) \).

void ml_put_mat(mat& A, char* name)
Writes \( A \) as MATLAB variable name in currently open MATLAB file (see ml_open in ml.doc). Aborts upon any error.
You may use the generic function names ml_get and ml_put for ml_get_mat and ml_put_mat respectively.
When you have a number of matrices to write and you want a one-to-one correspondence with respect to their names:

```c
ml_put_mat(A,"A");
ml_put_mat(res,"res");
```

You may use a C++ preprocessor feature to use a shorthand: with

```c
#define pm(a) ml_put_mat(a,#a)
```

somewhere at the top of your program, you may write

```c
pm(A); pm(res);
```
to get the same two `ml_put_mat` statements.

### 3.3.5 Time series support functions

Refer to `ts.doc` for time-series support basic functions. The functions in this section are a supplement on that. The only supplementary functions are mat-oriented versions of `ts_init` and `ts_load`. The functionality of `ts_get`, `ts_put` and `ts_save` (and their _all variants) extends automatically to the mat-versions. See `ts.doc` for details. For time series purposes, a m*n mat is assumed to be a vector of length m*n. In the vector, the matrix is represented columnwise (see also section 3.4).

See section 4.2 for notes on interpretation in MATLAB.

```c
int ts_init(mat &A,unsigned int nt,char* name)
```

Creates a memory area where (up to) nt time stamps of A may be stored. To be used as preparation of `ts_put` or `ts_put_all` calls, where at most nt calls are supported. Returns handle, but aborts upon any error.

Matrix A must have its intended size before you call `ts_init`. However, its content is not important. By just declaring

```c
mat A;
```
you have a 0*0 matrix A. Use

```c
A = zeros(m,n);
```

before calling `ts_init` to give A its intended size. All subsequent calls to relevant `ts_put` or `ts_save` functions (or the _all versions) will operate on the A and the size of A as it is upon the call to `ts_init`. So A must have a sufficiently large scope to cover the calls of `ts_put(_all)` and `ts_save(_all)` and the size of A should not change.

When you try to put more than nt time stamps (for instance by calling `ts_put_all` too many times), registration stops after nt time stamps. See `tce.doc` for further details.

```c
int ts_load(mat &A,char* name)
```

Creates and loads a memory area from MATLAB variable name in the currently open MATLAB file (file must be open for reading). To be used as preparation of `ts_get` or `ts_get_all` calls. Returns handle, but aborts upon any error.

Matrix A must have its intended size before you call `ts_load`, a sufficiently large scope to cover all calls of `ts_get(_all)` and its size should not change; see also the comments with `ts_init` above.

The total size of A (i.e. A.m*A.n) must match the number of columns in `name`. Every row in `name` is assumed to contain an instance of A. Loaded data is supplied, one instance after the other, in successive `ts_get` or `ts_get_all` calls, and defines a new content for A. When running out of new instances, the complete sequence is repeated in a cyclic way.

Function `ts_load` itself does not supply any (initial) value for A. Returns handle, but aborts upon any error.
3.3.6 Other functions

Conversion functions

mat expr(mat&) \ Returns the argument; see section 4.3

do\ble scalar(mat&) \ Returns a 1*1 matrix as a scalar.

Remark: representing a double as a 1*1 matrix (i.e. the reverse of function scalar) is done by
assignment:
mat A; \verb A = 3.75;

Display functions

void mat_id(mat& A) \ Displays size and some identification of A.

void mat_show(mat& A) \ Displays the content of A.

This is not MATLAB and you will not be very happy when showing the content of large
matrices. Both functions are also available for all special versions of mat (dmat, pmat, smat
and ivec).

Floating point operation counting

long mat_flops(void) \ Returns number of flops (cf. flops in MATLAB).

long mat_flops_reset(void) \ Resets flops to zero.

List generating function

The function list is documented separately in section 3.5.

Subarray functions

These are documented separately in section 3.5

Array management functions

An array is the content of a mat. Arrays are managed dynamically. Normally, you need
to care about array management. For details about array management and the available
functions, see section 3.6

3.4 Low level access

Class mat is defined as

```cpp
class mat {
    int m,n;
    real *p;
    int c;
}
```
together with a number of member functions. It describes a m*n matrix, where the elements are
in array p. Field c is for internal usage only. The specifier real for the elements is explained
below.

Be careful to address fields of a class mat variable directly. You do not create a 3*2 matrix A by
just setting A.m to 3 and A.n to 2. More is involved here to create the associated array in p. The
only reasonable low level accesses are inspecting the size of the matrix:
```
if ( A.m==3 ) ...
```
or read/write individual matrix elements as described in the next paragraph.
3.4.1 Addressing individual elements

The individual elements are addressed through \*p and are stored column-wise (as does MATLAB). When you are not familiar with C++, please do not care about the details of the notation "\*p"; it just says that p is an array where the individual elements may be accessed by a single index. Indexing in p is from zero (MATLAB uses 1). Suppose A is a 3*2 matrix, thus A.m=3 and A.n=2. Here is the translation for the elements of A:

```c
class mat MATLAB
A.p[0] A(1,1)
A.p[1] A(2,1)
A.p[3] A(1,2)
A.p[4] A(2,2)
```

Remarks:
(1) A.p[6] does not exists. There is no check on the validity of indices.
(2) You may shift to indexing from 1 by using a local variable which is related to A.p:
```
real *q; // a local variable
q = A.p-1; // set q to address of (non-existing) element A.p[-1]
```
Now you may use q[1] for A.p[0], q[2] for A.p[1] etc. Take care of using valid indices only: q[0] is not a legal element of A. Be careful with the association of a local variable to A.p: a statement A = <some expression> will probably change A.p which invalidates q!

3.4.2 Type of the elements

Class mat specifies the matrix elements to be of type real. C++ defines two standard types for real-valued numbers: float and double. According to the C++ standard, type float uses standard precision, type double should use a better precision when possible (when not, it is equal to float). Better precision requires more memory space and computations generally take (somewhat) more time. On a PC, a float takes 4 bytes and its precision is about 7 decimal digits; a double takes 8 bytes and its precision is about 16 decimal digits. The difference in computational speed is roughly 20 percent. MATLAB uses double only.

Class mat is available in a double as well as in a float version. Type real refers to either double or float. Conceptually and functionally they are equivalent, but beware of eventual inadequate precision; however, partial sums of inproduct calculations are kept with high accuracy in both cases. Control and estimation schemes need some kind of robustness anyway and are regularly evaluated in low precision.

3.5 Subarrays

In MATLAB, you may access a part of a matrix by using scalar or vector indices. Class mat has limited support for this too. In MATLAB you may index with matrix-construction elements like

```
[ 1 3 5 7 ] or 1:10. You use them in statements like:
B = A(1:10,4)
B = A(1:10,:)
B = A([1 3 8],:)
k = 3:10
B = A(:,k)
```
eetc. where all indices and right hand sides may be expressions, i.e. also
B = A(:,k)+2
B = A(:,k+1)+2

are fine.
Unfortunately, this cannot be implemented in C++ in the same general way:
- the matrix constructor \([\ ]\) is not available;
- the matrix constructor \(1:10\) is available in a modified form (see function \(\text{list}\) below);
- the 'all elements' indication ':\)' in MATLAB is available in a modified form '0'.

\[
\text{mat list(double start,int n,double incr=1)}
\]

Generates the MATLAB list
\[
\text{(start:incr:end)}
\]
where \(\text{end} = \text{start}+(n-1)*\text{incr}\), i.e., with \(n\) elements. You may omit \(\text{incr}\), it defaults to 1.

With function \(\text{list}\), the C++ equivalents of above MATLAB constructions are
\[
\begin{align*}
B &= A(\text{list}(1,10),4) \\
B &= A(\text{list}(1,10),0) \\
// B &= A([1 3 8],:) \quad // \text{is not available} \\
k &= \text{list}(3,8) \quad // \text{remark the value of the second argument} \\
B &= A(0,k) \\
B &= A(0,k)+2 \\
B &= A(0,k+1)+2
\end{align*}
\]

In general:
\[
A(r,c)
\]
returns a matrix with contains rows \(r\) and columns \(c\) of matrix \(A\); \(r\) and \(c\) may be integer scalars or one-dimensional matrices; their values are truncated to integer values and used as indices with indexing from 1 as in MATLAB. A scalar value \(r=0\) or \(c=0\) indicates all rows resp. columns and is equivalent to MATLAB's ':'. Matrix \(A\) must contain all elements which are asked for.

The other way around:
\[
A(1:10,4) = \text{zeros}(10,1); \\
k = 1:10; \\
A(:,k) = \text{zeros}(6,10);
\]
is different. An expression is not allowed at the left of the equal sign. Writing
\[
A(1:10,4)+2 = \text{zeros}(10,1)
\]
in MATLAB is illegal, as expressions are not allowed here. C++ is stronger in this: it does not allow any of above partial assignments to \(A\). As a remedy, there is a 'ssa' function, where 'ssa' stands for 'set sub array'. Above MATLAB constructions may be written in C++ as:
\[
\begin{align*}
A.\text{ssa}(\text{list}(1,10),4,\text{zeros}(10,1)); \\
k &= \text{list}(1,10); \\
A.\text{ssa}(0,k,\text{zeros}(6,10));
\end{align*}
\]
and the last one is still illegal.

In general, MATLAB's
\[
A(r,c) = v;
\]
is written in C++ as
\[
A.\text{ssa}(r,c,v);
\]
It sets values \(v\) in rows \(r\) and columns \(c\) of matrix \(A\) with \(r\) and \(c\) as discussed above; the size of \(v\) must match \(A(r,c)\) as in MATLAB. MATLAB allows you to specify indices outside the current range of \(A\) (which is then extended to an appropriate size); C++ does not.

The 'ssa' function has an 'asa' companion which stands for 'add sub array':
\[
A.\text{asa}(r,c,v);
\]
is equivalent to MATLAB's
\[
A(r,c) = A(r,c) + v;
\]

As stated above, the indices \( r \) and \( c \) are either integer scalars or one-dimensional matrices which elements are assumed to be (in fact are truncated to) integer values. The processor in a PC takes a lot of time in converting a real to an integer value. A special class \texttt{ivec} is available, specially meant to be used for indexing purposes. See 6.2.

### 3.6 Array management

There is no need to study this section in detail, but you are encouraged to read it once. Creating and destroying an object of any class is in C++ done by so-called constructors and destructors respectively. The compiler takes care of calling them where appropriate. For convenience, we will call class \texttt{mat} objects matrices in this section.

A global matrix, i.e., one which is declared outside any function, is created before the program actually starts and destroyed only after the program actually finishes.

A local matrix, i.e., one which is declared within a function, is created when the function is entered and destroyed when the function returns. When such a function is called many times, you'll have local matrices created and destroyed in each call.

To evaluate expressions, C++ uses so-called auto objects (matrices in our case) to hold intermediate results. These are like anonymous local variables, created where they are needed and destroyed after their usage (either immediately at the end of the expression or when the function returns).

What is actually done while creating or destroying a matrix, is defined in the constructor resp. the destructor of class \texttt{mat}. A matrix needs an associated array of the right size. The constructor of class \texttt{mat} takes care of supplying one by setting back some memory area for it. The destructor must take back the supplied memory area. If it does not, subsequent calls to the constructor must supply new memory areas as the old ones are assumed to be still in use. So, constructor and destructor must work together to manage the arrays.

In C++ you normally ask the Operating System to supply some free memory area of the size you want to have; when you do not need it any more, you give it back to the Operating System. In this way, the constructor and destructor may pass the memory management problem to the Operating System.

The problem is, that all Operating Systems, and especially DOS, introduce a lot of overhead for memory management and actually are too slow for arithmetic on small matrices. To remedy this, class \texttt{mat} has some kind of array caching: it maintains a pool where arrays may be stocked. The destructor stocks the array of the matrix which is going to be destroyed in the pool and the constructor tries to supply an array from the pool. Only when there is no matching one, the Operating System is asked for it.

Keeping arrays in the pool will result in larger demands on memory, as there are normally spare arrays in the pool. When working with large matrices, this may introduce a problem. So, class \texttt{mat} may be instructed not to use the pool. In this way, you optimize for memory at the cost of a speed penalty.

So far the background of array management. You need this to understand the functions described below.

\begin{verbatim}
int mat_a_cnt(int what=0)

Returns array management information:

what function returns:
\end{verbatim}
When the pool is in use, you normally have arrays in stock and no arrays are destroyed by the Operating System. The "missing" arrays (created but not in the pool) are those which are actually in use.

```c
long mat_a_lcnt(int what)
```

Returns detailed array management information:

- what function returns:
  - 0 total search length to find arrays in the pool
  - 1 number of arrays requested (constructor actions)
  - 2 number of arrays released (destructor actions)

You need a special version of the library to get this information. The normal versions of the library do not spend time keeping track of this kind of detailed information: you'll get 0 returned.

```c
void mat_a_lrpt(void);
```

Reports the three mat_a_lcnt values on a single line at the screen. See remark at `mat_a_rpt`.

```c
void mat_a_rpt(void);
```

Reports the three mat_a_cnt values on a single line at the screen.

Remark: writing to the screen takes a relative large amount of time. So do not call this function in a time-critical situation.

```c
int mat_a_save(int save);
```

When save=0, sets array management strategy to 0, i.e. "delete immediately"; when save>0, sets array management strategy to 1, i.e. "preserve in the pool". Returns previous value of strategy, either 0 or 1 for "delete immediately" and "preserve in the pool" respectively.

Remark: the default value is 1, which optimizes for speed.

```c
long mat_a_tot(void)
```

Returns total length (in number of reals) of arrays in the pool.

### 3.7 Error handling

Class mat functions and operators (which actually are functions as well) abort upon any error. They display a message which includes an identification of the function which encounters the problem.

When running under TCE, the error handler passes to the standard tce error handler which may call emergency functions as well. So, your experiment should be left in a safe state.

The main errors which may occur are incompatible matrix dimensions, inverting a singular matrix etc., but also insufficient memory may occur. Of course, special functions like ml_get_mat may encounter specific problems. All messages are self-explaining.

### 4 Application Notes

#### 4.1 Matrices and scalars

For MATLAB, a scalar is in fact a 1*1 matrix. In C++, a scalar is of type float or double, and a matrix, also a 1*1 matrix, is an object of type mat. As a result, C++ typing is more strict.
April 19, 1994

Maybe you have encountered a typical problem in MATLAB:

\[ I = \text{eye}(3); \quad \% \text{3*3 unity matrix} \]
\[ I = \text{eye}(A); \quad \% \text{unity matrix of the size of A} \]

Now this:

\[ A = 6; \quad \% \text{1*1 matrix A} \]
\[ I = \text{eye}(A); \quad \% \text{6*6 matrix because } A=6 \]

In C++ you don't have this confusion, as it's always clear whether the argument is a matrix or a scalar. There is another side too: with a 10*1 matrix \( v \), MATLAB allows you to write

\[ B = \frac{v}{\text{max}(v)}; \]

as \( \text{max}(v) \), a 1*1 matrix, is treated as a scalar and you may divide a matrix by a scalar. In C++, the equivalent function \( \text{max} \) will return a 1*1 matrix, and the division will fail due to incompatible matrix dimensions! There is a function \( \text{scalar} \), which transforms a 1*1 matrix into a scalar, and

\[ B = \frac{v}{\text{scalar}(\text{max}(v))}; \]

will be fine.

4.2 Time series in MATLAB

When you save time series and want to use them in MATLAB, MATLAB's load may have problems with the total size of the saved data. There is an additional MATLAB function \( \text{xget} \) to overcome this problem. See ts.doc for more information.

Once you have your data loaded in MATLAB, there may arise another problem. When you made a time series of a vector, you'll have it in MATLAB in the same format as returned by MATLAB control toolbox functions \( \text{step} \) and \( \text{lsim} \). However, when you made a time series of a matrix, it conceptually should be a three-dimensional tensor, but MATLAB and class \( \text{mat} \) do not support those structures. You'll find any instance of the m*n matrix A as a row vector in the time series matrix. You may reconstruct the individual matrices using standard (but not widely known) MATLAB features:

```matlab
load Ah \% Time series history of A. Gives you
A = \text{zeros}(m,n); \% variable Ah with p rows by m*n columns;
A(:, :) = Ah(k,:); \% picks out k'th row of Ah and fills A
\% with it in a m*n matrix format.
```

The other way around, creating a time series of matrices in MATLAB (eventually to be used in \( \text{ts_load} \)), is possible too: creating \( p \) instances of a m*n matrix \( A \) into a row oriented time series history \( Ah \) goes like:

```matlab
Ah = \text{zeros}(p,m*n); \% create beforehand to avoid memory
\% fragmentation in MATLAB
for i=1:p
    A = \text{<any m*n matrix>};
    Ah(i,:) = A(:,');
end
```

4.3 Writing your own mat functions

Probably you want to write functions which operate on matrices. Let us use as an example a function which evaluates an output equation in a state-space notation:


\[ y = C \times x \]

Of course, you do not need a function for a simple expression like this (you better write the expression itself), but it serves as an example only.

You may write a function

```cpp
void output1(void)
```

which relies on global variables to do the job.

You also may write a function:

```cpp
void output2(mat& y, mat& C, mat& x)
```

This is a more flexible one as you can call it with different arguments. There is a very small speed penalty as the compiler must supply the arguments upon every call. The compiler just supplies the address of the matrices, a very small amount of information, independent of the size of the matrices. When you call output2 always with the same arguments, you may revert to the output1 implementation to avoid this.

Alternatively, you may write a variant of output2:

```cpp
void output2a(mat& y, mat C, mat x)
```

This function may be implemented exactly as output2, but things are different here. When you call this function, the compiler supplies the address of y, but the complete C and x. It does this by making copies of C and x and supplying the addresses of the copies. The copies are especially made for the output2a call and not used elsewhere. When you do not have a special intention with the copies, you are wasting time here. So, this variant is not recommended in general.

Another variant of output2 may be:

```cpp
void output2b(mat y, mat& C, mat& x)
```

Now you get a copy of y, which is used nowhere else, to give it a new value \( C \times x \). As your function is supposed to supply y itself, this variant will not do the job. So forget it.

Above versions (not output2b!) supply their result in the first argument and the implementation may be just

\[ y = C \times x; \]

When more complex calculations has to be done, we may use local matrices to help us and play around with them and with the arguments as we like. However, above functions do not return anything, which is indicated by `void`. As a consequence, they cannot be used in matrix expressions. Referring to a standard function like `inv`, the difference may be clear: `inv` returns a matrix and thus may be used in expressions.

You may write functions which return a matrix as well. However, when you do not have an urgent need for it, skip reading the rest of this section.

Let us try

```cpp
mat output3(mat& C, mat& x)
```

which is assumed to return the matrix result, so you may call it as

\[ y = output3(C, x); \]

or use it in matrix expressions. The implementation of output3 is just

```cpp
return C \times x;
```

and there is no problem. When more complex calculations has to be done, we may need local matrices to help us. The implementation now may look like:

```cpp
mat y; // one or more local matrices
y = C \times x; // or more complex stuff
return y; // return a local matrix
```

A C++ compiler has the habit to destroy all local variables just before actually returning from a function; this includes destroying y! When encountering the `return y` statement, the compiler
knows that \( y \) is going to be destroyed before the function returns, so right here it makes a copy. The copy survives and is returned by the function.

In both `return C*x` and `return y` this scheme is used and involves more overhead than implementations like `output1` or `output2` which does not return the matrix as the result of the function. However, in `return C*x` things are not that bad, as the copy process just grabs the array which holds `C*x`, leaving `C*x` without it. In `return y`, the copy process (which does not know the context) cannot do this as \( y \) may be needed later on; so it actually makes a copy. In this case you may have a lot more overhead. For this purpose only, function `expr` may be of help: replace

```c++
return y;
```

by

```c++
return expr(y);
```

which tells the copy process that \( y \) does no longer needs its array, so it may be grabbed from it. Use function `expr` only in a `return` sequence where the argument is a local matrix, not a matrix expression.

A variant of `output3` may be considered:

```c++
mat& output3a(mat& C, mat& x)
```

which returns the address of a `mat` instead of the `mat` itself. The implementation may be equal to that of `output3`, but the copy process during the `return` sequence now involves copying the address of the resulting matrix only. Although this is a very fast action, you will probably run into problems: when the resulting matrix is an expression or a local matrix, it is lost before the function actually returns; only the address of this lost matrix is preserved! So, this is not the way to implement your function.

Remark:
Function `output3a` may set its result into a global matrix and return the address of this global matrix; this is OK, as global matrices are not destroyed.

Another possibility, widely used in early C++ days, is another variant of `output2`:

```c++
mat& output2c(mat& y, mat& C, mat& x)
{
    y = C*x;
    return y;
}
```

where the address of one of the arguments is returned in addition to the normal calculations. This makes `output2c` useful for use in expressions, but you have to supply explicitly a variable to hold the result as well.

To summarize: be very careful when writing functions which return a `mat`. Don't, unless you need them urgently. When you insist on returning a matrix, return a matrix expression

```c++
return C*x;
```

when possible, or return a local matrix through `expr`:

```c++
return expr(y); // local matrix y only!
```

### 4.4 Speeding-up

A class like `mat` allows you to handle matrices in a pretty natural and general way. Unfortunately, there is a penalty for generality: the code executed by class `mat` will not be as fast as optimized dedicated code. Class `mat` uses time for function calls and for administrative purposes, the latter one mainly with respect to dynamic management of the associated arrays (see section 3.6 for details). On the other side, actual execution of a function, say multiplying two matrices, is done by specially optimized code; you do not get own written code that fast easily. There are a couple of points which you may consider when you are up to greater execution speeds.
The first point must be your algorithm itself, regardless of the environment where it will execute. This is not the right place to discuss this issue in any detail, but you should give it enough attention before proceeding further to notation and implementation considerations.

A second point is related to the way how compilers evaluate expressions and is discussed in paragraph 4.4.1. This is a general topic, not specific for class mat, and is geared towards reducing the number of operations (or flops) involved. It has a better pay-off for larger matrices (e.g. bringing back the flop-count from order \( n^{-3} \) to order \( n^{-2} \)).

The third point is related to the administrative overhead which is specific for class mat and is discussed in paragraph 4.4.2. The administrative overhead concentrates on the management of the associated arrays, which it is essentially independent of the size of the matrices involved. So, efficiency topics of this kind pay-off especially when your matrices are small.

### 4.4.1 Evaluation of expressions

This paragraph discusses the way how expressions are evaluated and the consequences of it when the expressions involve matrices.

Most (probably all) C++ compilers evaluate expressions left-to-right at equal priority level. As an example, in

```plaintext
a+b+c
```
a+b is evaluated first. When the operands are matrices, this is no problem as the matrices must have the same dimensions. However, while evaluating

```plaintext
a*b*c
```
again a*b is evaluated first. Now assume a and b to be \( n\times n \) matrices and c to be a \( n\times1 \) vector, the final result being a \( n\times1 \) vector. Evaluation of a*b takes \( n^{-3} \) operations (with every operation counting as two flops), and multiplying the result with c takes another \( n^{-2} \) operations. The total score is of the order \( n^{-3} \). In this case things may be done with \( 2n^{-2} \) operations in total by first evaluating b*c, resulting in a \( n\times1 \) vector, and then premultiplying the result by a.

You get the faster order of evaluation simply by imposing the preferred priority:

```plaintext
a*(b*c)
```

which forces the evaluation of b*c to be done first. The difference in efficiency is an order of magnitude in terms of \( n \).

Why the compiler does not take care of this? Historically, standard C++ operands are of scalar types where the issue is not relevant. However, do not expect C++ compilers to be smart enough in the near future to optimize this kind of problems automatically. The best strategy depends on the actual sizes of the matrices and the compiler cannot know these beforehand (you may get from MATLAB a \( l\times n \) matrix a and \( n\times n \) matrices b and c as well!). So, when you want efficient code, you better pay some attention to it yourself. The flop-counting function mat_flops may be of help.

MATLAB does exactly the same left-to-right evaluation. So, adding brackets to force the evaluation priority may help also, but MATLAB's interpreter overhead dominates (lack of) speed in many cases, except for large matrices.

A similar flop-reducing issue is related to the use of the inverse of a matrix:

```plaintext
c = inv(A)*b
```
The implementation is done by both class mat and MATLAB as (with anonymous iA):

```plaintext
iA = solve(A,eye(A));   // or MATLAB: iA = A\eye(A)
c = iA*b;
```

When you do not need the inverse explicitly, the faster implementation is a direct

```plaintext
c = solve(A,b);     // or MATLAB: c = A\b
```

For symmetric, positive definite matrices, you better use the faster functions \texttt{sinv} and \texttt{ssolve}...
for \texttt{inv} and \texttt{solve} respectively. Where appropriate, you should consider the use of the special matrix representations \texttt{dmat} for a diagonal matrix, \texttt{pmat} for a packed (sparse) matrix or \texttt{smat} for a symmetric matrix. They are documented in the section 6.

4.4.2 Administrative overhead

One part of the overhead which you may influence is related to the management of arrays. Refer to section 3.6. Do not touch the strategy which class \texttt{mat} uses to manage arrays. Its default is for speed, using the pool to preserve arrays as much as possible.

In a statement like
\begin{verbatim}
    a = a+b;
\end{verbatim}
the expression \texttt{a+b} is evaluated first and the result is then assigned to \texttt{a}. The two actions are not combined by most compilers. As a consequence, the actual code looks like:
\begin{verbatim}
    tmp = operator+(a,b);
    a = tmp;
\end{verbatim}
where \texttt{tmp} is some anonymous matrix (an auto variable in C++ terminology) and there are two assignments.

Replace it by the compound assignment
\begin{verbatim}
    a += b;
\end{verbatim}
which does the same job without introducing the extra \texttt{tmp} matrix and the extra assignment.

When you split up your program in different functions to perform different tasks (you are encouraged to do so), you may need local matrices to help performing a function's task. When such a function is called over and over again, these local matrices are created and deleted every time, which involves some overhead. Although declaring these locally used matrices within the function is the preferred programming method, it is more efficient to declare them global (thus outside any function) as this avoids the repeated creation and deleting. Now you need unique names (which is not needed for local variables) and other functions may play around with them as well. Moreover, you will not gain much here.

If you need a local matrix \texttt{h} which is created over and over again, and its first usage is something like
\begin{verbatim}
    h = <some matrix expression>;
\end{verbatim}
declare it just as
\begin{verbatim}
    mat h;
\end{verbatim}
without specifying its size:
\begin{verbatim}
    mat h(m,n);
\end{verbatim}
as the latter is equivalent to (but somewhat faster then)
\begin{verbatim}
    mat h;
    h = zeros(m,n);
\end{verbatim}
As you can see, the intermediate \texttt{h = zeros(m,n)} is meaningless here.

Writing functions with matrix arguments does not involve more then the (standard small) argument overhead when you specify them as \texttt{mat&}. When you specify them as \texttt{mat}, you will get a copy, which is specially made for the occasion and not used elsewhere; this introduces overhead proportional to the size of the matrix. See section 4.3 for details.

When you want to write functions which return a matrix, and thus may be used in matrix expressions, you must be careful and read section 4.3 thoroughly.

The normal libraries (see section 5) do full checks on the size of matrices. Of course, this takes (some) time. There are special version which omit the size checking. Please swap to these versions only when you are completely sure that size checking is not needed.
When you have a program which runs a real-time loop, and you go for the speed, your loop will re-use arrays when they are available, thus avoiding the Operating System overhead for creating them. During the first pass through the loop, you still have the Operating System overhead when the arrays are not yet there. There is no standard trick to handle this situation. Some thoughts about it may be:

1. It does no harm, because I'm not that time-critical.
2. It does no harm, because I've to deal with a transient anyway.
3. It does no harm, because I activate my actuators after all calculations have been done only; how much (extra) time is involved in the first time calculations is not of interest.
4. I call the relevant functions once before I enter the loop to have them create their matrices.
5. I write a special function with appropriate local matrices (or arrays of matrices) and initializes them all with function \texttt{zeros}. I call this function before I run my loop. By then, matrices are created and transferred to the pool when the function returns. Thus I have already created all, or most of the, matrices which are needed in the loop.

### 4.5 Trouble Shooting

Unfortunately, you may encounter troubles. These come in two categories: your fault or not your fault. You must capture both of them.

The first category includes incompatible matrix dimensions, inverting a singular matrix, asking a MATLAB matrix which is not there and the like. You have to find a solution yourself.

The second category is related to lost arrays. Normally you will never encounter this problem, but read the rest of this section once and remember it when you may have run into it. Read section 3.6 for more information about array management.

When an array is lost, its memory is effective out-of-use during the rest of the run. When the array is lost in a function which is called over and over again, you effectively loose many arrays and ultimately run into insufficient memory.

Who lost the array? You didn't as you cannot (so it's not your fault). Class mat implementation didn't neither: it just implements a reliable constructor/destructor pair to handle the arrays. The compiler has to be blamed for it: it forgets to call the destructor for some (auto) matrices; this must be qualified as wrong code or, if you want, a bad compiler.

Here is a known compiler problem from Turbo C, version 2:

```c
double my_norm(mat& A) {
    return scalar(max(abs(A)));
}
```

where the destructor for the auto matrices involved by the computation of \texttt{abs} and \texttt{max} is not called; as the auto matrices go out-of-scope immediately, the arrays are lost (the error is corrected in version 3).

When you write:

```c
double my_norm(mat& A) {
    double r;
    r = scalar(max(abs(A)));
    return r;
}
```

everything is fine.

You experience the problem ultimately by calling \texttt{my_norm} enough times, but, when running short experiments, the problem may stay unnoticed. In the mean time, however, you will have quite a speed penalty. The constructor has to supply new matrices every time. As these are not available,
it must ask the Operating System for it. Right here, there is a simple way to verify if this problem occurs: run your program with two different loop counts and register the value of mat_a_cnt(1) (see section 3.6). When they are equal, there are no lost arrays.

Remark: You only need to do this verification when you suspect a "lost array" problem. When it is there, look for a remedy of the kind of the example above: split matrix calculations from the return statement.

5 Availability

Class mat is included in the C++-versions of tce. The normal versions are:

<table>
<thead>
<tr>
<th>Library</th>
<th>Processor</th>
<th>Real</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcel0.lib</td>
<td>all</td>
<td>double</td>
</tr>
<tr>
<td>tcel1.lib</td>
<td>&gt;=386</td>
<td>double</td>
</tr>
<tr>
<td>tcel0f.lib</td>
<td>all</td>
<td>float</td>
</tr>
<tr>
<td>tcel1f.lib</td>
<td>&gt;=386</td>
<td>float</td>
</tr>
</tbody>
</table>

All libraries include the special classes ivec, dmat, pmat and smat as well; see section 6.

To include the class definitions and function prototypes, write

```cpp
#include <tce.h>
#define real float
#include <tce.h>
```

at the top of your program when you want to use a real=double library, or when you want to use a real=double version.

The tce.h include file includes all available TCE functions. It includes a number of other .h files, which must be present as well.

Special versions are available for >=386 processors only and have extra letters appended to their base name. The double versions (use header file tce.h) are:

- tcelii.lib – generates a very large file MAT.I with array management information
- tceli.lib – includes mat_a_cnt administration
- tceliin.lib – no checking of matrix sizes, no administrative data

The float versions are tcelifi.lib, tcelifl.lib and tcelifi.lib respectively. Do not use the libraries tceli.lib and tcelifi.lib when time may be critical; they are for specialists only.

All above files may be found in a main directory \TCE (or on a TCE distribution floppy). They may be referred directly from there (you need no own copy); for the header files to be found, you need to add directory \TCE to the search path of include files. For the integrated development environment of Turbo (or Borland) C you do this by setting

```
Options | Directories | Include files to
\TC\INCLUDE;\TCE
```
6 Special classes

6.1 Special matrices

The classes dmat (diagonal matrices), pmat (packed matrices) and smat (symmetric matrices) are classes which are derived from class mat, the latter being called the parent class. Derived classes have all properties of their parent class, but may have additional members (i.e. declarations and/or functions). Moreover, they may implement own versions of functions which are already in the parent class.

So, you may handle matrices of these special versions just as normal class mat matrices, but you should obey the special characteristics: a diagonal matrix should have non-zero diagonal elements only! For all derived classes, there is a function which assigns a normal mat to the appropriate special version. Upon this assignment the special characteristics are forced to hold. All normal arithmetic at the matrix level maintains the special characteristics automatically, but you may corrupt them by assigning values to individual elements and/or by assigning to subarrays. In this case, results of calculations may become unpredictable.

6.1.1 Class dmat - diagonal matrices

When you have a matrix expression like

\[ A' \cdot W \cdot A \]

and \( W \) has only non-zero elements on its diagonal, there is going to be a lot of time spent multiplying by zero. This is how you may avoid this:

```plaintext
mat A,R,W;
dmat dW; // Declaration of a Diagonal MAT

A = ml_get_mat("A");
W = ml_get_mat("W");
R = tran(A)*W*A; // This is the time consuming one

dW = W; // dW represents diagonal matrix W
R = tran(A)*dW*A; // This is the fast one. Result is equal when W is in effect diagonal.
```

The recipe is simple:
- Start with the normal matrix (\( W \) in this example);
- Convert to the special format by just assigning a mat to a dmat;
- Use the dmat in expressions.

When \( W \) has non-zero off-diagonal elements, they are ignored; in fact, \( dW = W \) is equivalent with MATLAB's

\[ dW = \text{diag(diag}(W)\text{)}; \]

Remark: in above example you may have made the shortcut

\[ dW = ml_get_mat("W"); \]

as the conversion of a mat (the result of ml_get_mat) to a dmat is done upon the assignment.
6.1.2 Class pmat - packed matrices

Packed matrices are matrices where only the non-zero elements are represented. There are provisions to speed-up matrix multiplication calculations with respect to packed matrices in the form of class pmat, which is only useful when the original matrix is really sparse (i.e. contains mainly zero elements). The "break-even" point is roughly at about 50 percent non-zeroes.

Upon the assignment to a pmat, the location of the non-zero elements is established and stored in some (internal) packed information. Do not change a zero element to a non-zero value by direct assigning to the element as this invalidates the packed information.

Remark: The name "packed matrix" may be misleading: the matrix itself is not packed (you'll find the individual elements at their normal places); only the location of the non-zero elements is stored and used in matrix multiplications.

6.1.3 Class smat - symmetric matrices

A symmetric matrix, called a smat, maintains its symmetry automatically when you assign a (matrix) value to it. With mat A and smat sA, the assignment

\[ sA = A \]

effectively involves

\[ sA = (A + \text{tran}(A))/2 \]

but is implemented more efficiently.

There is no automatic check for A to be symmetric (or close to symmetry) as this takes additional time and class smat does not suspect you to do a dirty job.

You create a smat, as with the dmat described above, just by assigning a mat to it.

Apart from the different assignment to a class smat variable, functions inv and solve, when applied to a smat, implement the more efficient versions sinv and ssolve automatically. As a consequence, your smat should be positive definite in this case. When it is not, just forget the use of smat for this particular matrix.

Remark: In Lyaponov-like equations you may find expressions like

\[ A*P + P*\text{tran}(A) \]

with a symmetric matrix P, resulting in a symmetric matrix. There is a function

\[ \text{smat aat(mat& M)} \]

which returns \( M + \text{tran}(M) \) as a smat.

With this function, you may write the Lyaponov expression as

\[ \text{aat}(A*P) \]

and avoid the second matrix multiplication or the introduction of a help matrix to represent \( A*P \).

6.2 Class ivec - integer vectors

An ivec is a one-dimensional vector with integer values. It may be useful for array indexing, as it is faster than indexing with normal mat expressions.

Declare an ivec with:

\[ \text{ivec iv; or} \]
\[ \text{ivec iv(n); the latter one creating an ivec of length n.} \]

A one-element ivec may be created just by assigning an int to it:

\[ iv = 5; \]

The (more important) list constructor is function ilist:

\[ iv = \text{ilist(int start,int n,int incr=1);} \]
and performs just as function list (but only for integer values). See 3.5.
To convert a one-dimensional mat A into an ivec, use function ivec_make:

```cpp
ivec iv = ivec_make(mat& A);
```
or its synonym imake.

An ivec may be converted into a normal (one-dimensional) mat just by assignment:

```cpp
A = iv;
```

The available operators are: adding and subtracting ivecs and integer scalars, and multiplying by an integer scalar; compound assignments (for instance ivec1 += ivec2) are available too.

Class ivec is declared as:

```cpp
class ivec {
    int m;
    int *p;
}
```

where n is the length of the vector and the values are in p[0], p[1] etc.
Contents

1 Introduction 1
2 Tutorial 1
   2.1 Example 1 ........................................... 1
   2.2 Example 2 ........................................... 2
   2.3 Writing data in MATLAB readable form ............... 3
3 Functions 3
4 Other MATLAB exchange facilities 4
5 Direct exchange 4
6 Using a dual processor system 5
7 Availability 5

1 Introduction

Module ml contains MATLAB oriented exchange functions to be used in your C or C++ program. This document describes the basic user-oriented functions only. Documentation on implementation is available in ml_a.doc and is meant for specialists only.

The library handles real matrices only. When MATLAB supplies a complex one, you'll get the real part only.

2 Tutorial

2.1 Example 1

Suppose, you want from MATLAB values for:

\[ \begin{align*}
    sf & \quad \text{sample frequency} \\
    ns & \quad \text{number of samples}
\end{align*} \]

You decide to represent them in a MATLAB variable sys:

\[
\text{sys} = [100; 1000]
\]

and you want to have them available in your C program.

Save, within MATLAB, the information in some .mat file, say, setup.mat:

```matlab
save setup sys
```

Your C program main function may look like:

```c
double sys[2] = {0, 0};
double sf;
int ns;

ml_open("setup", 0);
ml_get_vec(sys, 2, "sys"); /* (1) */
sf = vec[0]; /* (2) */
```

/* Remarks below */
ns = (int)vec[1];
/* Check if sf and/or ns hold reasonable values */

Remarks:
(1) You must open a MATLAB file environment with function ml_open. The first argument specifies the filename and the second argument must be 0 for reading from the file or non-zero for writing to the file. Upon a file open error, ml_open aborts your program, so there is no need to check this.

Function ml_open adds extension ".mat" to the filename when there is no extension specified (when your filename is really "setup", so without extension, please specify it as "setup."); this prevents ml_open from appending the default extension.

Note the use of the double quote (") to specify strings in C. MATLAB uses the single quote (') for the same purpose.

(2) Function ml_get_vec tries to fetch MATLAB vector "sys" with two elements into variable sys. The MATLAB vector may be either a row or a column vector. It aborts upon errors, but accepts missing the MATLAB vector and a MATLAB vector with different size. When MATLAB vector "sys" is not found, ml_get_vec just leaves the values in sys unchanged, allowing you to work with default values. When MATLAB vector "sys" is shorter than 2, ml_get_vec just reads the available values into sys. When it is longer than 2, ml_get_vec uses the first two values only. You are assumed to do some checks on the usefulness of the values. When you do not want to run with default values, just change the code to something like (see next note for indexing):

```c
sys[1] = -1;  /* Unreasonable value for ns */
ml_get_vec(sys ,2,"sys");
```

When less than two values are read into sys, sys[1] will still be -1 and may not pass your checks, so effectively this forces full specification of vector sys.

(3) It is standard C practice to number a vector from zero. Declaring
double v[2];
gives you a two element vector with elements v[0] and v[1]. This may be confusing as element v[2] does not exists (there is no check on this!) and MATLAB numbers the same elements v(1) and v(2) respectively. You may overcome this by:
double vc[2];
double *v = vc-1;

Now, v is a vector with exactly the elements v[1] and v[2]. Again, there is no check on indexing illegal elements! When you are not familiar with C, be cautious with tricks like these.

### 2.2 Example 2

As the first example, but you decided to fetch the variables sf and ns from two different scalar MATLAB variables with the same names. As above, they are supposed to reside in a file called setup.mat. This is how the MATLAB session may look like:
sf = 100; ns = 1000;
save setup sf ns

and this is a possible fragment from your C program:
double sf;
int ns;

```c
ml_open("setup",0);
sf = ml_get_scalar("sf",100);
ns = (int)ml_get_scalar("ns",-1);
```

The second argument of function ml_get_scalar specifies the default value which is used when the MATLAB variable is not found. Any other problem, including a non-scalar MATLAB variable,
will cause ml_get_scalar to abort your program. Apparently, this example does not accept a default for ns, as its default value is not reasonable.

Function ml_get_scalar returns a double. The prefix "(int)" in the second call to ml_get_vec is called typecasting and tells C that you want this value transformed into an int. It may be omitted as ns is declared to be an int, but you may get a compiler warning about losing precision. It is good practice to typecast explicitly, showing that you are aware of the transformation.

2.3 Writing data in MATLAB readable form

You may also write scalars and vectors in a .mat file. The available functions are, you guess it, ml_put_scalar and ml_put_vec. Before you can put anything, you must open a .mat file for writing:

```c
ml_open("results",1);
```

Here again the extension ".mat" is appended automatically and the second argument indicates that you are going to put things in it. Any possibly existing file results.mat is deleted without warning!! Now you may use any of the ml_put-functions anytime. The program fragment below creates MATLAB variables "scalar" and "vec" in file results.mat:

```c
double s; /* a scalar */
double v[3]; /* a vector */
```

```c
s = 23.4;
v[0] = 0; v[1] = 5; v[2] = 2.3;
ml_open("results",1);
ml_put_scalar(s,"scalar");
ml_put_vec(v,3,"vec");
```

As usual, your program will be aborted upon any error.

3 Functions

Remark: All ml_get functions, when confronted with a complex MATLAB matrix, fetch the real part only. All put functions create real matrices only.

```c
void ml_close(void);
```

Closes file environment opened by ml_open. Normally, there is no need to explicitly call ml_close as a next call to ml_open implies ml_close when needed, and files are closed automatically when the program ends.

```c
void ml_get_array(double *array,int m,int n,char *vname);
```

Tries to fetch the content of MATLAB variable vname of size m*n into array, where data is represented linear, columnwise. Aborts when vname is not found or of wrong size.

```c
double ml_get_scalar(char *vname,double def);
```

Tries to return the scalar content of MATLAB variable vname. If vname is not found, def is returned. Any other trouble, including a vname of non-scalar size, causes abortion of the program.

```c
int ml_get_vec(double *vec,int size,char *vname);
```

Tries to fetch the content of MATLAB variable vname into array vec[0]..vec[size-1]. Returns the number of elements written in vec. Several cases may occur:
- `vname` is not found: returns 0 with `vec` unchanged;
- `vname` has length `n<=size`: writes `n` values in `vec[0]..vec[n-1]` and returns `n`;
- `vname` has length `n>size`: writes `size` values in `vec[0]..vec[size-1]` and returns `size`.

Aborts, as usual, upon file access troubles.

```c
void ml_open(char *fname, int mode);
```
Opens file `fname` for reading (mode==0) or writing (mode!=0). When no extension is supplied, ".mat" is appended.
Prior to calling any `ml_get` function, you should call `ml_open` with mode==0; prior to calling any `ml_put` function you should call `ml_open` with mode!=0.
Aborts upon any error.
Remark: when an earlier `ml_open` has not been followed by a `ml_close`, `ml_open` starts with closing the old file.

```c
void ml_put_array(double *array, int m, int n, char *name);
```
Writes (linear, columnwise) content of `array` as an `m*n` MATLAB matrix `name`.
Aborts upon any error.

```c
void ml_put_scalar(double val, char *vname);
```
Plts `val` as a MATLAB `1*1` matrix `vname`.
Aborts upon any error.

```c
void ml_put_vec(double *vec, int n, char *vname);
```
Plts `vec[0] .. vec[n-1]` as MATLAB `n*1` matrix `vname`.
Aborts upon any error.

In the C++ versions you may use the generic names `ml_get` and `ml_put` for any of above `ml_get` and `ml_put`-functions respectively.

## 4 Other MATLAB exchange facilities

For collecting time-series data, you are invited to use the `ts`-functions, which has their own MATLAB exchange facilities. See `ts.doc` for documentation.
To work with matrices, please use class `mat`. It has its own `ml_get` and `ml_put` facilities. See `mat.doc` for documentation.

## 5 Direct exchange

Normally, your C program is compiled and linked into an executable (EXE) file, which may or may not be activated directly from within MATLAB. Exchange from and to MATLAB must go through files in this case.
Eventually, you may want to transform your C program into a MEX or MX3 version, which is directly incorporated in MATLAB (or MATLAB386 respectively). In this case there is no need to exchange information through files.
For this purposes the `ml` library is available in a MEX version as well. Due to the lack of an appropriate compiler, there is no MX3 available (yet?).
As the number of (compatibility and memory limitation) troubles tend to be larger with MEX than with EXE implementations, you are not encouraged to use the MEX version.
6 Using a dual processor system

When running with a secondary system, for instance the dSpace (see x.doc), all \texttt{ml\_get} and \texttt{ml\_put} functions pass their data from the MATLAB file to the secondary system or vice-versa. The primary system serves as a pass-through (where the data will become available as well). This is assumed to be the normal operation. When you want to pass the data to or from the primary system only, without the secondary system being involved in any way, append a "0" to any of the \texttt{ml\_get} or \texttt{ml\_put} functions: \texttt{ml\_get\_vec0} for \texttt{ml\_get\_vec}, etc.

7 Availability

The \texttt{ml} library is included in all \texttt{tce} libraries; see \texttt{tce.doc} for more information. The MEX version is \texttt{ml.mex}; it is a C implementation. For a dual processor version, please contact Jos Banens.
# Introduction

TCE (Tools for Control Experiments) is a collection of small modules, coded in C and C++, which implement basic tools for control experiments. The tools are intended to be used efficiently in a time-critical environment.

The currently available tools are geared towards cooperation with MATLAB.

Within the TCE family you will find different modules for different tasks. A TCE module is just a collection of functions which conform to the TCE conventions which are discussed below. Documentation of the individual modules may be found in the document's own .doc file or paper. For details about implementation and availability, these documents may refer to the TCE conventions and to this document.

TCE itself is also a (very small) module. It contains some error handling functions to be used by the user-oriented modules. Documentation on the tce-functions may be found in tce_a.doc and is needed to implement TCE modules only.

TCE is also a platform to make the modules easily available. Refer to the sections 5 and 6.

## TCE conventions

We'll use TCE modules ml and ts as examples: module ml implements basic MATLAB oriented services and module ts implements time series support. Their usage here is for illustration only and does not serve as (full) documentation.

### 2.1 Naming conventions

All functions (and global variables) of a TCE module have names which begin with the module name and an underscore. Examples:

- `ml_open` (from module ml)
- `ts_init` (from module ts)
2.2 Time-critical functions

TCE modules differentiate between time-critical and other functions. Time-critical functions are optimized for usage in a real-time loop, the others are not.

2.3 Arguments

Time-critical functions do not have a lot of arguments. In many cases they refer to arguments specified by a related initializing function. You must be alert to have the scope of the arguments involved large enough to cover the call of the time-critical functions. The section 3 discusses this in some detail.

2.4 Error checking

Non time-critical functions in TCE modules perform full error checking. Upon errors, they display a message and abort your program. This relieves you from writing error handling code yourself. Time-critical functions generally do not perform checks on arguments (most of them do not have any arguments at all). They assume that you call them correctly and do not waste time by checking this. However, they do functional checks. When they encounter problems, they may do one of two things (documented in their own documentation):
- set some error indication (which will stay set, you may inspect it later) and continue;
- run some emergency code: call up to two emergency functions, display a message, and abort.
Which strategy is used depends on what is most reasonable or safe. No one will continue when unpredictable results may result. The section 4 discusses this in some detail.

2.5 Documentation

We refer to module ml here as an example. The ml module has a document ml.doc, which contains the general user-oriented documentation. It is available as a file ml.doc and as a printed document. A TCE module may also have more detailed documentation which discusses underlying features and is aimed for specialists (for instance for implementors of other modules or new features). For the ml module, this underlying document is ml-a.doc; generally, append _a to the base name of the module. Normally, there is no need to refer to this information. Be also prepared on the different style of it: it is aimed at specialists only.

3 Scope of variables

Time-critical functions do not have much arguments. In most cases they refer to arguments specified by a related initializing function. You must be alert to have the scope of the arguments involved to be large enough to cover the call of the time-critical functions. An example may clarify this. With

```c
double z[7];
ts_init(z,7,10000,"measurements");
```
you tell the time-series support functions that you want up to 10000 time-stamps of the 7-element vector z to be recorded under the name "measurements". Actual recording of the next time stamp is done by code like

```c
... new values in z ... /* symbolic notation */
ts_put_all();
```
in your real-time loop. Function ts_put_all will record the actual value of the vector z specified as first argument of ts_init. For proper functioning, the z in
... new values in z ...

must be the same as the z used in ts_init. When your program looks like:

```c
int main(void) {
  double z[7];
  ts_init(z,7,10000,"measurements");
  for (...) {
    ... new values in z ...
    ts_put_all();
  }
  ...
}
```

there is no problem: there is only a single variable z involved. When you use different functions prolog and run for initialization and real-time duty respectively, your program may look like:

```c
void prolog(void) {
  double z[7];
  ts_init(z,7,10000,"measurements");
}
void run(void) {
  double z[7];
  for (...) {
    ... new values in z ...
    ts_put_all();
  }
}
void main(void) {
  prolog();
  run();
  ...
}
```

but here you violate the rule: the z in function run is not the same as the z in function prolog. Function ts_put_all cannot detect this (even if it could, it should not do it, not spending time to check you violating the rules). In fact, the z in prolog exists inside prolog only.

When you use different functions (you are encouraged to do so), you must declare z to be global (i.e., outside any function, preferably at the top of your program) and avoid local declarations with the same name. Use only a single, global z to hold the measurements in your program:

```c
double z[7];
void prolog(void) {
  ts_init(z,7,10000,"measurements");
}
void run(void) {
  for (...) {
    ... new values in z ...
    ts_put_all();
  }
}
void main(void) {
  prolog();
  run();
  ...
```
and you will not have any problem. The documentation of individual TCE modules will make clear what variables need to have a global (or at least sufficiently large) scope.

4 Error handling

Assume you write some program:

\[\text{prolog(); run(); epilog();}\]

(here relying on global data structures), where function run activates, runs and stops (i.e. leaving in a safe state) some experiment.

Function prolog typically has an administrative task: setting-up data which is needed for run, it does not actually starts any experiment related actions. In the same way, function epilog is not involved with controlling the experiment either. It only does administrative tasks related to post-processing or, generally, handling the data collected by run.

When something is (likely to be) wrong during prolog, it may be better not to start the experiment, thus to abort before calling run. This is what TCE-functions actually do: they abort upon errors or (serious) problems with a clear message about the reason.

TCE-functions in epilog act the same way, but they realize that, although some function may fail, some other function may still supply relevant information. They try to keep things going.

When something is wrong in run, things are different. Now we may be in the middle of some control sequence and just aborting may be dangerous for the experimental environment. Moreover, any collected data is lost when the abortion is done without calling function epilog.

This is why TCE-functions in run, when they encounter serious problems, call some TCE error handling function which may call two functions before displaying a message and aborting. The first function stops the experiment, the second one does the epilog.

5 Available modules

- exp - Standard experiment interface
- mat - Matrix algebra implementation (C++ versions only)
- ml - MATLAB basic exchange functions
- ts - Time series basic functions
- tim - Timing functions

6 Availability

For easy access, the available modules are, as much as is possible, collected in a single tce-library. However, there are different versions of the library.

None of tce-libraries includes exp, as this one contains code which is specific for a particular experimental environment. When you want to drive a specific experiment, please look for a file exp_<xxx>.lib with some appropriate letters for <xxx>. See exp.doc for details.

The tce-library comes in C versions and a C++ versions. The C tce-libraries are
tce00.lib - for all processors
The C++ versions do include the mat module. In the mat module, matrix elements may be represented as float or double, the latter holding better precision. The C++ tce-libraries are

- `tce01.lib` - for >=386 processors
- `tce10.lib` - for all processors
- `tce11.lib` - for >=386 processors
- `tce10f.lib` - for all processors
- `tce11f.lib` - for >=386 processors

where the first two use double and the last two use float.

There may be special versions of the library as well. See the documentation of the individual modules.

To let C or C++ check your function calls, you must include so-called prototypes for all functions. This is generally done by including one or more header files. To use the functions you must start your code with

```c
#include <tce.h>
#define real float
#include <tce.h>
```

where the define must precede the include; alternatively, you may use

```c
#include <tcef.h>
```

with is effectively the same.

The libraries and the include files will reside in a main directory

```
\TCE
```

on your current disk. When they are not, ask for a TCE diskette.

There are more header files, one or two for every module. File `tce.h` refers to these, so do not remove them.

To collect things together, you must

1. make a project (or a makefile) which collects all code together to build an executable file;
2. set your include path to include \TCE in addition to the standard path.

In the Turbo (or Borland) C integrated development environment, you accomplish these steps by:

1. create a project and insert in it what you need. When all of your code is in file `my_code.c` and you want to run the XY-table, your project should look like:

   ```c
   my_code.c
   \tce\exp_xy.lib
   \tce\tce11.lib
   ```

   (do not forget the extension .lib) which collects your code, the experimental environment for the XY-table and the standard tce functions.
2. select

   ```c
   Options | Directories | Include files
   ```

   and set it to

   ```c
   \TCE|INCLUDE;\TCE
   ```

   when this is not yet there.

Remark: You must run the large memory model, found under

```c
Options | Compiler
```

and you must select C or C++, found under

```c
Options | C++ options
```

Setting options is for the current project only, so inspect (and eventually set) them for new projects.
Contents

1 Introduction 1
2 Functions 1
3 Availability 1

1 Introduction

The functions `tim_t0` and `tim_t1` implement a way to measure real clock times accurately. For relative short times (say up to 50 milliseconds), the accuracy is within some microseconds with a typical standard deviation of 1 microsecond. For larger times, the maximum timing error is less than 0.1 percent.

The implementation relies on the time-of-day interrupts, which normally occur 18.2 times per second. Interrupts must be on for the timing to work properly. Function `tim_t0` takes an initial time stamp and function `tim_t1` returns the elapsed time since the last call to `tim_t0`. The current implementation is not made to measure an elapsed time of more than one day: you'll get the time modulo one day only. Also, when someone changes the time-of-day clock between `tim_t0` and `tim_t1`, or switched off interrupts during some time, the resulting time information is unreliable. Function `tim_t1` returns 0 when no previous call to `tim_t0` has been made, and it returns a negative value (effectively -1.0) when timing failed to work.

Functions `tim_t0` and `tim_t1` may be used in real time situations. They do not consume time in wait loops or the like. However, they take some processing time, and, as stated above, the real-time interrupt must be on for proper timing. Use test program `ttim.exe` (found in the tce directory) to find out about actual times for your system.

2 Functions

```c
void tim_t0(void);

Stamps actual time.

double tim_t1(void);

Returns elapsed time (in seconds) since last call to `tim_t0`. Returns 0.0 when there was no prior call to `tim_t0` or -1.0 when timing fails to work.

Timing relies on the real-time clock of DOS and fails when real-time interrupts are disabled.
```

3 Availability

Included in all tce libraries. See tce.doc for more information.

There is a Matlab version of the timing functions too (the built-in Matlab timing is not very accurate). To save Matlab memory space, the functionality of `tim_t0` and `tim_t1` is compressed into a single Matlab function `tim(i)`, implemented as a mex function (thus not available in Matlab386). Try 'help tim' within Matlab.
1 Introduction

Module ts contains functions to handle Time Series data in a time-critical environment. It is able to handle large amounts of data, using extended memory, without any speed penalty when running under DOS. It reads and writes MATLAB-compatible files.

2 Tutorial

2.1 Problem statement

Suppose you are in a sampled data system. Periodically you get measurement data and compute other data (filter states, inputs etc.). You want to record these data. It may be a lot, eventually much more then can be supported easily by (standard) DOS. Anyway, you cannot afford to waste time in the time-critical loop, so direct saving to disk is impossible. You want to have the data available in MATLAB afterwards.

Reference trajectories go the other way: they are available prior to running the time-critical loop and must be accessible efficiently within the loop. They may come from MATLAB as well. The ts-functions deal with this kind of data handling.

The ts-functions work with vectors (say filter state xf, measurements y, inputs u) and will record the history of these vectors in the form of a matrix (say filter state history xfh, measurement history yh, input history uh).

It's normal MATLAB practice to have the discrete time running along the rows and the signals making up the columns. This is rather strange (selecting momentary signals will give you a row vector, where all state-space related notation expects column vectors) but standard (accepted) practice. The ts-functions conform to the MATLAB practice: yh(i,:) = y' (i.e. the transpose...
of \( y \) at some discrete time related with index \( i \). With 6 measurements in \( y \) and 5000 discrete time points, you’ll get a 5000*6 matrix.

### 2.2 Example 1

Suppose you want to record time histories only (reference data is not involved in this example). You want to save up to 5000 samples of

- `double y[3]` : measurements (3*1 vector)
- `int u[2]` : inputs (2*1 vector of int)

A fragment of your code may look like:

```c
/* Remarks below */

ts_init0(y,3,5000,"y");     /* (1) */
ts_init0(xf,6,5000,"xf");
ts_init3(u,2,5000,"u");

/ * ... Initialize experiment ... */

while ( /* ... more ... */ ) {
    /* ... Fetch measurements, ... */
    /* ... compute filter state ... */
    /* ... and inputs ... */
    ts_put_all();               /* (3) */
}

/ * ... Shut down experiment ... */

ml_open("exp2",1);            /* (4) */
ts_save_all(0);               /* (5) */
```

Remarks:

1. Any of the functions `ts_init0`, `ts_init1`, `ts_init2` and `ts_init3` initialize a time history matrix, which may be filled through calls of `ts_put_all`. The first argument establishes the vector involved. The last argument gives it a name (quote names with " in C, with ' in MATLAB). The final digit in the different `ts_init` names indicates the type of data in the time-series matrix:
   - 0 - `double`
   - 1 - `float` (i.e. single precision)
   - 2 - `long` (i.e. 32 bits signed integral)
   - 3 - `int` (i.e. 16 bits signed integral)

   MATLAB uses double only. The reason for the others is saving space (and some time): on a PC, a double takes 8 bytes, float and long take 4 and int only 2.

2. When you run a C++ version of the ts-module, you may use `ts_init` always (without the final digit in the name). The C++ compiler will find out the details upon the type of the first argument.

3. When something goes wrong, any `ts_init` function will supply an appropriate message and abort your program. This frees you from inspecting (and writing code) for error conditions.

4. This is assumed to be a time critical loop.

5. Function `ts_put_all` is the only time-critical function in our example. It has no arguments, does not return a value and does not report any message. It does check for proper operation and will fail only when you call it too many times (more than 5000 times in this example). It will not
abort upon failure (as this may be quite dangerous in a time-critical environment); it just stops recording. 
You don't tell here what data to record to the time histories. You did it with ts_init. As a consequence, ts_put_all will always put the same vectors y, xf and u into the time histories. You cannot swap to other vectors. So, when you intended (or forced to intend) to specify all 5000 times the same vectors, why should you repeat them in the time-critical loop? This is why specification is done at initialization. 
However, there is a programming consequence. When you split-up your code into different functions for - amongst others - initialization and real-time action, you cannot use locally defined vectors y, xf and u in these functions. The vectors local to your initialization function are gone when leaving the function and their addresses, registrated by ts_init, are no longer valid. This cannot be detected. So, don't. Use global vectors instead: declare them outside any function, preferable at the top of your program.

(4) To save the time series for MATLAB usage, you must open a MATLAB file. Here you open file "exp2.mat" for writing (as indicated by the second argument). An existing file exp2.mat is overwritten without notice. See documentation in ml.doc for details. 
(5) All recorded time histories are saved in file exp2.mat using the names as specified during ts_init. A file like exp2.mat is called a xmat-file (which is not a standard MATLAB convention). Its structure is identical to a normal .mat file. However, due to compact representation of float (type 1), long (type 2) and int (type 3), it need not be identical to a normal .mat file. When it stores only double precision matrices, it is.
The single argument of ts_save_all specifies if you want the histories saved as they are (value 0) or translated to all double precision matrices (any non-zero value); The C++ version of ts allows you to omit the argument, which defaults to 1. 
If the argument is non-zero, you may use MATLAB's load facility to read the entire file. However, this may give problems when the total size (adding everything together) is large. To overcome this, there is a selective load facility added to MATLAB, called xget, which may fetch selective data from any xmat (or normal .mat) file. It fetches also type 1, 2 and 3 data. See MATLAB's help xget or type xinfo within MATLAB.
As described above, you get the time-points running along the rows of the matrices. Function ts_save_all will only save that part of the time-histories which is actually filled by ts_put_all. When you called ts_put_all 1260 times in the loop, you'll get a 1200*3 matrix y, a 1200*6 matrix xf and a 1200*2 matrix u. The '5000' in the calls to ts_init serves as an upperbound only.

2.3 Example 2
Same as Example 1, except a time-series of a vector with reference data is added.
Your code may look like:

```c
/* Remarks below */
ml_open("r2",0);
/* (1) */
ts_load0(r,2,"r");
/* (2) */
ts_init0(y ,3,5000,"y ");
ts_init0(xf,6,5000,"xf");
ts_init3(u ,2,5000,"u ");

/* ... Initialize experiment ... */
```
while ( /* ... more ... */ ) {
    ts_get_all(); /* (3) */
    /* ... Fetch measurements, ... */
    /* ... compute filter state ... */
    /* ... and inputs ... */
    ts_put_all();
}

/* ... Shut down experiment ... */

ml_open("exp2",1);
ts_save_all(0); /* (4) */

Remarks:
(1) MATLAB file "r2.mat" is opened for reading.
(2) The ts_load0 function loads a time-history. You'll get subsequent vectors from it by every
call to ts_get_all. The final 0 in ts_load0 indicates type double. See Example 1, remark (1).
Again, in a C++ implementation it may be omitted.
Data for vector r will be fetched from matrix "r" in file "r2.mat". The file must be a MATLAB-
file (made by MATLAB's save facility) or an xmat file (see above). The data in the file must be
an*n*2 matrix: it must have exactly two columns as specified by the second argument.

(3) For ts_get_all holds the same as for ts_put_all: it fetched values for the next time stamp
and puts them into r (as specified by ts_load0). However, you may call ts_get_all as many
times as you want. When it runs out of data (i.e. time points), it just starts over again, which
effectively results in cyclic reference signals (which may introduce discontinuities).
(4) Function ts_save_all saves the matrices y, xf and u as in the first example. It does not save
matrix r as this matrix is assumed already present somewhere. However, you may have added
another ts_init for r as well.

2.4 Example 3

This example is only meant to introduce some of the lower level functions of the ts-library. It
should not be regarded as a guide to think about filters. Refer to the first example. Suppose you
want to save the filter state xf only once for every five samples.
Your code may look like:

```c
/* Remarks below */

int i,hy,hxf,hu;

hy = ts_init0(y,3,5000,"y"); /* (1) */
hxf = ts_init0(xf,6,1000,"xf");
hu = ts_init3(u,2,5000,"u");

/* ... Initialize experiment ... */

i = 0;
while ( /* ... more ... */ ) {
    /* ... Fetch measurements, ... */
    /* ... compute filter state ... */
```
November 20, 1994  ts.doc  5

/* ... and inputs ... */
ts_put(hy);
nts_put(hu);
if ( i==0 ) { ts_put(hxf); i = 5; }
i--;

/* ... Shut down experiment ... */

ml_open("exp2",1);
ts_save_all(0);

Remarks:
(1) Functions ts_init return a so-called handle (a small non-negative integer). In C, you need not use such a return value, so you may write
   ts_init0(y,3,5000,"y");
as we did in the first example, as well as
   hy = ts_init0(y,3,5000,"y");
as we do here.
When saving the filter state at a lower frequency, we may lower the maximal number of samples for xf to 1000 to stay compatible with the maximal 5000 samples in the other ts_init calls.
(2) Function ts_put records a single vector in its time-history. It needs a handle as an argument. 
It does not check if the handle is a valid one as the ts-functions do not want to spend the check time in the real-time loop. However, all ts_init functions return valid handles (they abort upon errors), so the only responsibility of your code is not to change the values of the handles.
As you may have guessed already, there is also a function ts_get. Function ts_save_all has a ts_save variant too. See the function descriptions below.
(3) As a result of lower frequency calls to ts_put(hxf), we may end up with a 1200*3 matrix y, 1200*2 matrix u and 240*6 matrix xf. There is no explicit sample-time information in the saved data, as the MATLAB compatibility prohibits this. Of course you can construct your own time vectors as time series as well.

3 Functions

3.1 Time-critical functions

void ts_get(int handle);

Gets data for handle as specified by the ts_load function which returned the handle. Does not check the validity of the handle. Uses the available time-stamps in a cyclic manner: the one after the last will be the first again.

void ts_get_all(void);

Gets data for all handles returned by ts_load functions. Uses all data cyclic. Different time series need not have the same length, so the cycle frequency may be different for different handles.

void ts_put(int handle);

Puts data for handle as specified by the ts_init function which returned the handle. Does not check the validity of the handle. Upon "too many calls, no more room to store", an error flag is set and data is not stored.

void ts_put_all(void);
Puts data for all handles returned by ts_init functions. Upon "too many calls, no more room to store" (which may occur with regard to one or more of the handles), an error flag is set and the particular data is not stored.

3.2 Other functions

int ts_error(int handle);
Returns 1 when there is an error reported for handle, otherwise 0. The only error is "too many calls to ts_put(_all) as all other errors caused abortion of the program.

int ts_error_all(void);
Returns the sum of ts_error(handle) for all active handles.

void ts_fini(void);
Finishes the ts-environment, invalidating all handles. In normal applications there is no need to explicitly call ts_fini.

int ts_init0(double *buf, int ns, unsigned int nt, char *name);
Initializes memory area where (up to) nt time stamps of (exactly) ns signal values may be stored. To be used as preparation of ts_put or ts_put_all calls (at most nt calls are supported). Function ts_put(_all) will store the ns values which are momentarily in buf. While calling ts_init0, the content of buf is not important. Data type is double (8 bytes/value).

Returns handle, but aborts upon any error.

int ts_init1(float *buf, int ns, unsigned int nt, char *name);
As ts_init0, but data type is float (4 bytes/value).

int ts_init2(long *buf, int ns, unsigned int nt, char *name);
As ts_init0, but data type is long (integral, 4 bytes/value).

int ts_init3(int *buf, int ns, unsigned int nt, char *name);
As ts_init0, but data type is int (integral, 2 bytes/value).

int ts_load0(double *buf, int ns, char *name);
Creates and loads memory area from variable name in currently open ml file (file must be open for reading). The memory area will contain a number of time stamps of ns signals each. Loaded data is supplied, one time stamp after the other, in successive ts_get or ts_get_all calls, eventually repeating the sequence in a cyclic manner. The ns signal values are supplied in buf. Function ts_load0 itself does not supply any (initial) values in buf. MATLAB variable name must be a n*ns matrix of type double (which is the normal type for MATLAB). The number of rows n may be anything >0, but the number of columns must match ns.

Returns handle, but aborts upon any error.

int ts_load1(float *buf, int ns, char *name);
As ts_load0, but data type is float. Type in name must be either 0 (double) or 1 (float).

int ts_load2(long *buf, int ns, char *name);
As ts_load0, but data type is long. Type in name must be either 0 (double) or 2 (long).

int ts_load3(int *buf, int ns, char *name);
As ts_load0, but data type is int. Type in name must be either 0 (double) or 3 (int).
void ts_memory(int ext);
  Use Extended memory (ext!=0) or DOS memory (ext==0). Takes effect to subsequent calls
to ts_init and/or ts_load only.
Default is using Extended memory on a 386 processor (or up) whenever this is available,
otherwise DOS memory is tried. Old 86 and 286 processors can use DOS memory only.

void ts_replay(int handle);
  Changes the state of the time series with identification handle from "initialized by ts_init,
operated by ts_put(_all)" to "made by ts_load, operated by ts_get(_all)"; resets to
start of time series.
  Aborts on invalid handle or empty time series.

void ts_replay_all(void);
  Changes the state of all time series "initialized by ts_init, operated by ts_put(_all)" to
"made by ts_load, operated by ts_get(_all)"; resets to start of time series.
  Aborts on any empty time series.

void ts_save(int handle, int ml);
  Appends memory content of handle (returned by some ts_init function call) to currently
open ml file (file must be open for writing). Matrix name is as specified in ts_init. Format
is equal to internal format (type 0, 1, 2 or 3 for double, float, long or int resp.) if ml==0.
Any non-zero ml translates internal format to type 0 (double, compatible with standard
MATLAB).
  Aborts upon any error.

void ts_save_all(int ml);
  Appends all data which was collected using all ts_put and ts_put_all calls into currently
opened ml file (open for writing). Matrix names are as specified in the corresponding ts_init
calls. Format is equal to internal format (type 0, 1, 2 or 3 for double, float, long or int resp.)
if ml==0. Any non-zero ml translates internal format to type 0 (double, compatible with
standard MATLAB).
  Aborts upon any error.

3.3 Synonyms in C++
Function ts_save_all() is equal to ts_save_all(1); it thus defaults to saving in MATLAB
format.

  ts_load may be used for either ts_load0, ts_load1, ts_load2 or ts_load3. ts_init may be
used for either ts_init0, ts_init1, ts_init2 or ts_init3.

4 Error handling
Except for the time-critical functions, your program is aborted upon any error (after supplying a
reasonable message). The time-critical functions ts_get and ts_get_all cannot fail; ts_put and
ts_put_all stop recording time-series upon overflow. Functions ts_error and ts_error_all
will tell if this has happened.

5 Ts-created files in MATLAB
When you saved type 0 matrices only (eventually by the built-in conversion facility in ts_save or
  ts_save_all), you may be able to use MATLAB's load facility. This tries to load the complete